Network Analysis Without Exponentiality Assumptions

by

Mor Harchol-Balter

B.A. (Brandeis University) 1988

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Manuel Blum, Chair
Professor Sheldon Ross
Professor Venkat Anantharam

1996

The dissertation of Mor Harchol-Balter is approved:

Chair Date

Date

Date

University of California at Berkeley

1996

Network Analysis Without Exponentiality Assumptions

by
Mor Harchol-Balter

Abstract

Network Analysis Without Exponentiality Assumptions

by

Mor Harchol-Balter
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Manuel Blum, Chair

Theoretical analysis of networks often relies on exponentiality assumptions which are unrealistic but necessary for mathematical tractability. In some cases these assumptions are inconsequential, but in others their adoption may lead to invalid results.

We consider two broad areas of network analysis in which exponentiality assumptions are often used: The first of these is delay analysis of packet-routing networks. Delays occur in such networks when multiple packets seek to pass simultaneously through a bottleneck – i.e., a part of the network which can accommodate only a single packet at a given time. The goal of the analysis is to determine the average packet delay associated with a particular routing scheme and arrival rate. Frequently, for analytic tractability, bottleneck times are modeled as independent exponentially-distributed random variables, which is often inaccurate.

A second area of analysis employing such assumptions is CPU load balancing in networks of time-sharing workstations. Processes are continually generated at workstations and may at any point be migrated (for a price) to another workstation in an effort to minimize the average process slowdown. The aim of the analysis is to develop and evaluate effective migration strategies. For analytical tractability, process CPU lifetimes (total CPU usage) are commonly modeled as exponentially-distributed random variables. However our measurements show process lifetime distributions are actually Pareto (inverse-polynomial).

For the case of packet-routing, we prove that for a large class of networks, the exponentiality assumptions always yield an upper bound on the actual average packet delay. This result validates the use of exponential service-time assumptions in queueing-theoretic

analysis.

In contrast, we show that the exponentiality assumptions made in the analysis of process migration can yield incorrect results. This is potentially true even for models which accurately reflect CPU lifetime distributions in the first two moments. We introduce a new general technique whereby our measured process lifetime distribution is applied to derive a highly effective load balancing policy, without resorting to exponentiality assumptions. Also via the process lifetime distribution, we answer the long unresolved question of whether migrating active processes (preemptive migration) is necessary for load balancing, or whether migrating newborn jobs (remote execution) suffices.

Professor Manuel Blum
Dissertation Committee Chair

To my husband,

Robert H. Balter,

without whose love and understanding

I would not have survived my first year

of graduate school.

Contents

List of Figures

List of Tables

Acknowledgements

I would never have made it to Berkeley in the first place if it hadn't been for my undergraduate thesis advisor, Marty Cohn, who swore he would not write my recommendations unless I applied to the topmost graduate schools.

When I first started at Berkeley, I had no idea what I was getting into. I feel very lucky that my instincts led me to seek Manuel Blum as my advisor. Manuel has been a constant source of confidence throughout my Ph.D. He has encouraged me to go after controversial topics, to define new areas of research, and to create new problems within these areas. To foster a creative environment, Manuel has provided unconditional support, both defending and protecting me, and encouraging me to consult with as many other researchers as would listen to me. In short, Manuel has taught me what an advisor should be.

A few other professors have also influenced my graduate research. Among these are Sheldon Ross, whose excellent class and highly readable textbooks on stochastic processes and queueing theory first inspired me to start thinking about delays in packet-routing networks. Dick Karp has also been an important role model to me, both as a teacher and as a researcher, and I feel fortunate to have had the opportunity to work with him. Lastly, Venkat Anantharam, who went through my thesis with a fine-toothed comb, greatly improved the quality of the explanations throughout this thesis.

The best part of being at Berkeley is the graduate students you get to work with. Much of Part II of this thesis is taken from a paper, [31], co-authored with Allen Downey. When I first started working with Allen, the plan was that he would spend a month writing a simulator and then we'd go our separate ways. The month turned into a two-year project with many interesting surprises along the way. Large parts of Chapters 3 and 5 are from a paper, [32], co-authored with David Wolfe. I'll always remember fondly our long discussions at the blackboard over what's really the right problem to be solved.

I also formed many close friendships during graduate school with my fellow sufferers: Ari Juels, who fed me chocolate continuously to keep me going, and improved my writing immensely; Caroline Tice, who cheered me up on many occasions; Steve Smoot and Ron and Jody Parr, whose company is always a treat; Kim Keeton, who grieved with me on all the downs and celebrated with me on the ups; and Hal Wasserman, whom I could depend on to always show up in the office at midnight and keep me company. I'd also

like to acknowledge the many friends I made at WICSE, as well as the support of Sheila Humphreys and Kathryn Crabtree.

Lastly, I'd like to thank my family for their never-ending love and support: my father, who taught me that hard work will get you anywhere; my mother, who taught me that being kind to others is the most important thing in life; and my husband, who's the most important contributor to my success.

My graduate school experience was funded primarily by a fellowship from the National Physical Science Consortium. I am infinitely grateful for their support.

Chapter 1

Introduction: Thesis Themes

1.1 Network Analysis

Our overall goal is the development of new methods for network analysis, both for the purpose of evaluating the performance of existing network algorithms and for deriving new network algorithms.

Network analysis encompasses a wide variety of problems. One example entails a packet-routing network where packets arrive at each host at some rate, and each packet is routed along some path from its source host to its destination host. Here an important question is to derive a formula for the average time packets are delayed (by other packets in the network) as they move along their route. Another example is that of a network of workstations where processes are continually generated at each workstation. Some processes might run faster if migrated to other (less crowded) workstations, however there is a cost associated with migrating each process. Here the question is to develop an analytical criterion for which processes should be migrated and when, so as to minimize the average process slowdown.

1.2 Unrealistic Exponentiality Assumptions

Because most networked systems are complex, their analysis usually requires some modeling. In constructing the model, various simplifying assumptions are made, for analytical tractability. Some assumptions may be grossly inaccurate. It is the role of the modeler to ensure that these unrealistic assumptions do not affect the *outcome* of the analysis, or

that the effect is predictable and can be adjusted for. In this thesis we concentrate on one class of such assumptions which we refer to as *exponentiality assumptions*. These typically involve assuming that one or more variables associated with the problem are exponentially distributed random variables. For example in a packet-routing network the transmission times (time required to load a packet onto a wire) are often assumed to be exponentially-distributed random variables when in reality they are a constant function of the packet size and wire bandwidth. Another example occurs when analyzing a network of workstations where processes are continually generated at each workstation. In this context, the process CPU requirements are often assumed to be exponentially distributed for analytical tractability. We will show later that this assumption too is far from realistic. Other examples are the interarrival times of packets or processes to the network (from outside), which are commonly modeled as being exponentially distributed (i.e., a Poisson arrival process).

1.3 Why are Exponentiality Assumptions Needed?

So why are the exponentiality assumptions needed in the analysis of networks? Exponentiality assumptions are usually needed in order to allow one to model the system by a simple Markov chain, so that one can solve for the steady-state distribution of the system (see next few paragraphs for more detail). What's peculiar, however, is that these exponentiality assumptions have become so commonplace that now even many purely simulation studies (no analysis) use them (see Section 9.1.2), rather than using measured distributions or traces.

Below, we first review Markov models and their solution, and then we explain why the exponentiality assumptions are needed.

Recall a discrete-time Markov chain (DTMC) consists of a countable number of states $S_1, S_2, \ldots, S_n, \ldots$, and a stochastic process which moves from state to state *at discrete time steps*, such that if the process is currently in state S_i, there is some fixed probability P_{ij} that on the next time step the process will move to state S_j. A DTMC exhibits the *Markovian property*, because, given that the process is currently in state S_i, the conditional distribution of the next state and all future states is independent of past history (those states the process visited before state S_i.). A continuous-time Markov chain (CTMC) is the continuous time analogue of a DTMC. A CTMC is exactly like a DTMC, except that a random variable amount of time is spent in state S_i before the "coin is flipped" to

decide which state to transition into next. *This random variable amount of time must be exponentially distributed (memoryless) in order that the process be Markovian*; if the random variable were not memoryless, then the time until the next state transition would depend on the time already spent so far in the current state, i.e., on past history. By modeling a system as a Markov chain, one can determine the steady-state limiting distribution on the states through solving "balance equations," which equate, for each state, the rate of transitions into the state with the rate of transitions out of the state [68]. If the Markov chain is simple enough, the balance equations are solvable, however, the Markov chain may be sufficiently complex that no solution is found for the balance equations.

To see where exponentiality assumptions aid in the modeling process, consider a very simple example of a single host with a CPU. Jobs are continually generated at the host. Assume for simplicity that each process accesses the CPU in first-come-first-served (FCFS) order. Each job uses its required processing time and then terminates. In modeling this system, a state might represent the number of jobs currently queued at the CPU, including the job receiving service. Since new jobs may be created at any moment in time, and may complete at any moment in time, rather than only on discrete time steps, a CTMC is needed to model the system, rather than a DTMC. From the above discussion, in order to model this system by a CTMC, the time between state transitions must be exponentially distributed. This corresponds in our example to requiring job CPU lifetimes to be exponentially distributed random variables and the times between new job creations to be exponentially distributed (i.e., a Poisson arrival process).

In the above example, the CTMC is very simple since transitions are only possible between numerically-consecutive states, resulting in easily solvable balance equations. The steady-state distribution would be a probability distribution on the number of jobs at the host. In particular, from this probability distribution we could obtain the average number of jobs in the system, which, using Little's Law, yields the average delay experienced by jobs.

It turns out that the above example is simple enough that a Markov chain could still be formed even without the exponentiality assumptions by using a technique known as embedded DTMCs. However, without either exponentiality assumption, the Markov chain we would obtain would have very complicated transition probabilities and horrendous balance equations, which would not be solvable given a general arrival process and generally

distributed job CPU requirements[1].

The above example was very simple to model. Once jobs (or packets) are allowed to migrate between hosts in a network, it is usually impossible to even construct a Markov model without exponentiality assumptions (and sometimes further simplifying assumptions are necessary as well).

1.4 Impact of Exponentiality Assumptions: Harmful or Innocuous?

Given that much of the existing literature relies on exponentiality assumptions which are often unrealistic, a natural question is:

> *What effect do the unrealistic exponentiality assumptions have on the outcome of the analysis?*

For example, can we prove that the outcome of the analysis is invariant to the particular exponentiality assumption, so one should feel free to make the assumption? Perhaps the outcome of the analysis is affected by the exponentiality assumption, but always in a provably predictable manner (direction). Or, it could be the case that making the particular exponentiality assumption completely invalidates the outcome of the analysis.

1.5 Framework for Results

We consider two general areas of network analysis where exponentiality assumptions are often made. For each of these areas we show three things:

1. We demonstrate that the particular exponentiality assumption typically made is inaccurate.

2. We show which of three categories the exponentiality assumption fits into: (a) innocuous, (b) affects the result in a predictable manner, or (c) is harmful in that it leads to the wrong conclusion.

3. In the case that the assumption is harmful, we show how to do a partial analysis without resorting to the exponentiality assumption.

[1] Approximations, though, do exist for the G/G/1 queue.

Issue (1) above has been explored in the literature very recently, however the exploration has primarily been limited to the area of packet arrivals into a network, see for example [48], [19]. However, the large body of work on the hyperexponential, phase-type, and Erlang distributions indicates that issue (1) has been acknowledged for some time. Issue (2) has for the most part been ignored in the literature, particularly with respect to (2b) and (2c), although some insensitivity results, (2a), do exist [84]. As mentioned above, issue (3) has been addressed to some extent, although more work in this area is clearly necessary. One of the contributions of this thesis is simply raising these important issues.

1.5.1 Part I

In Part I of the thesis we examine the problem of predicting the average packet delay in a packet-routing network and show it fits into category 2(b) above. This problem comes up in both parallel and distributed applications which require packets to be routed in a network. As packets move along their routes, they are delayed by other packets. Predicting packet delays in networks is an essential requirement for obtaining quality of service guarantees and evaluating routing schemes and network designs. Throughout, we will be assuming that all packets have the same size (as in an ATM network).

Delays are incurred in networks when two or more packets simultaneously want to pass through a bottleneck — a part of the network which can only "serve" one packet at a time. For example, the time required to load a packet onto a wire represents a bottleneck.

Ideally, we would like an analytical "formula" for the average packet delay which applies to an *arbitrary* network topology, routing scheme, and arrival rate at the nodes.

Such formulas already exist in the queueing theory literature, but depend on a couple of exponentiality assumptions. One such assumption is that the arrival process of packets to the network from outside is a Poisson process. Several recent studies have shown this exponentiality assumption to be inaccurate [48], [19], claiming that the measured Ethernet and Web arrival process exhibits self-similar behavior, however no simple alternative functional distribution has yet been agreed upon in its place, although more complex processes have been proposed[2]. Another required exponentiality assumption is that the service time required at a bottleneck (or "server" in queueing theory terminology) is an independent exponentially-distributed random variable. To see how unrealistic this

[2] A distribution consisting of an aggregation of an infinite number of on-off sources with Pareto-distributed on-times was proposed recently in [49] and then generalized in [5].

exponential assumption is, imagine that every time we load packet p onto wire w it requires a different amount of time (not including the time p spends in queue), ranging from 0 to ∞, even though the size of p and the bandwidth of w haven't changed. In reality, the service time required at a bottleneck is some *constant* corresponding to each bottleneck (server), with possibly a different constant for each bottleneck (in the above example, this constant would be the inverse bandwidth of the wire, multiplied by the fixed packet size). Unfortunately, as we'll see in Section 2.3, the most powerful results in queueing theory only apply to exponential-time servers and not to constant-time servers.

In Part I we investigate the effect of the exponential service-time (bottleneck-time) assumption on the average packet delay. We work within a simple model of packet-routing networks. Our model assumes that packets are born with routes and packets travel along their routes independently of each other. Our model allows for arbitrary network topologies, bottleneck locations, bottleneck durations, and outside arrival rates. For the most part, we require that packets are served in a FCFS order and that all packets have the same fixed size.[3] We continue to assume a Poisson outside arrival process.

We prove that in this model, for a large class of networks, $\mathcal{S}$, assuming exponentially-distributed bottleneck times when deriving the delay always results in an upper bound on the actual delay. Thus the exponentiality assumption on bottleneck service times is actually a useful analytical tool in obtaining bounds on delays. Our work in this area involves characterizing this set $\mathcal{S}$. We hope our work will inspire others to consider the effect of other exponentiality assumptions and either similarly validate them, or find ways to do without them.

Chapter 2 introduces Part I. It also outlines of all the results proved in Part I, defines the necessary terminology, and describes previous work in the area of packet-routing.

1.5.2 Part II

In Part II, we consider the problem of CPU load balancing in a network of workstations and show it fits into category 2(c) above. CPU load balancing involves migrating processes across the network from hosts with high loads to hosts with low loads, so as to reduce the average completion time of processes. The exponentiality assumptions made

[3] Observe that our packet-routing network model is also identical to a traditional dynamic job shop model, as described in [13]. Here jobs are routed between machine centers, with each job requiring service only at certain machine centers in a particular order, and the service time depends only on the machine center, not on the job.

in this area are (1) assuming that the CPU requirements of processes (process lifetimes) are exponentially distributed, (2) assuming that packet interarrival times are exponentially distributed, and sometimes even (3) assuming that process migration times are exponentially distributed. We show that all these exponentiality assumptions are unrealistic, but the first one is the most harmful. We show that modeling the CPU lifetime distribution by an exponential distribution, or even a hyperexponential distribution with the correct mean and variance leads to spurious conclusions with respect to developing load balancing algorithms and evaluating their effectiveness.

We measure the UNIX process lifetime distribution, and determine a functional form for this distribution. Our measured lifetime distribution is very different from an exponential distribution and is well-modeled by a inverse-polynomial, Pareto-type, distribution. We then use this functional form as an analytical tool to derive a load balancing policy which does not rely on assuming any of the above mentioned exponentiality assumptions. We use a trace-driven simulation based on real process arrival times and real CPU usage to test our load balancing policy, and find it to be highly effective and robust. As a bonus, since our policy is analytically derived, it requires no parameters which must be hand-tuned.

Our measured distribution also allows us to answer via analysis and simulations the unresolved question in the load balancing literature regarding whether migrating active processes is significantly beneficial, or whether it suffices to only migrate newborn processes (remote execution).

Chapter 8 introduces Part II. It also outlines the results shown in Part II, defines the standard load balancing terminology, and describes previous work in the area of CPU load balancing.

Part I

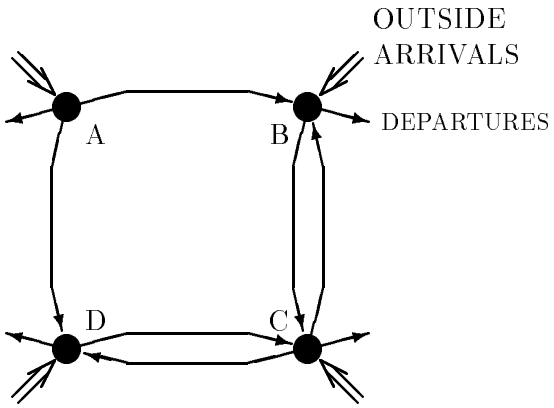
Validating Exponential Service Assumptions in Delay Analysis of Packet-Routing Networks

Chapter 2

Model, Definitions, and Approach

In Part I of this thesis we'll examine one area of network analysis, delay analysis in packet-routing networks, where exponentiality assumptions are usually essential for analysis. Two exponentiality assumptions are necessary: The first concerns the service time distribution of packets at bottlenecks in the network (this terminology will be defined in this chapter). The second involves the outside arrival process of packets at each host. Both of these exponentiality assumptions are unrealistic. We limit our work to only the first of these assumptions: exponential service times. We ask the question: Does the exponential service time assumption affect the outcome of the analysis? We will prove that although not entirely innocuous, the exponential service time assumption is not as harmful as one might think. We'll show that for a large class of packet-routing networks using the exponential service time assumption affects the result of the analysis in a predictable manner, in that it provides an upper bound on correct result.

We consider the problem of computing the steady-state average packet delay in an arbitrary packet-routing network where the outside arrival process is assumed to be Poisson. We begin by describing our abstract model for a packet-routing network (Section 2.1). We then reformulate the definition of a packet-routing network as a queueing network having constant-time servers (Section 2.2). Unfortunately, queueing networks with constant-time servers are almost never analytically tractable. However queueing networks with exponentially-distributed service times are (Section 2.3). In Section 2.4, we explain our approach which we use throughout Part I: Rather than try to compute delays for the constant-time server case directly, which is known to be elusive, we instead *prove* that the delay in the constant-time server case can be upper bounded by the delay in the appro-



Possible packet path	Average rate of birth
A →B →C →	1 every 10 sec
A →B →C →B →	1 every 15 sec
B →C →B →C →	1 every 20 sec
⋮	⋮

Figure 2.1: *A packet-routing network*

priate exponentially-distributed model, which we know how to solve. In Section 2.5, we outline the results of Part I and their significance. Section 2.6 covers previous results on bounding delays in packet-routing networks. In this context, we explain the appeal of using our approach (Section 2.7). In Section 2.8, we cover some known theorems which will be used throughout Part I. We conclude this chapter with intuition about our method and a few easy results (Section 2.9).

2.1 Packet-routing network model

Consider any physical network (e.g. ATM network) of arbitrary topology made up of nodes (e.g. hosts, routers) and wires, with fixed-size packets. The packets arrive continually at the nodes of the network from outside. Each packet is born with a route (path) which it must follow and each route occurs with some average frequency (possibly zero). For example, in the routing scheme of Figure 2.1, packets with path $A \to B \to C \to$ are born at an average rate of one every 10 seconds, and packets with path $B \to C \to B \to C \to$ are born at an average rate of one every 20 seconds. We will refer to such a network as a dynamic packet-routing network, $\mathcal{P}$.

With respect to delays, what differentiates packet-routing networks from each other is the location and duration of *bottlenecks*. A bottleneck is a part of the network through which only one packet can pass at a time. For example, the theoretical computer science community [44], [36], [81], generally considers each wire to be a bottleneck. Specifically, in their definition it takes some unit constant amount of time for a packet to traverse each wire and only one packet may traverse any given wire at a time. If a packet arrives at a wire which is currently being used, the packet must join the end of the queue of packets waiting to traverse that wire. A more realistic example is a packet-switched network, where each router node also constitutes a bottleneck, or a chain of bottlenecks, and although more than one packet may traverse a wire at a time, the packets must be separated by the transmission time (the time to get the packet onto the wire). That is, the transmission times represent the bottlenecks.

Since only one packet at a time may pass through each bottleneck, a queue of packets forms at each bottleneck. In most physical networks with fixed-size packets the time required for the packet at the *head* of the queue to pass through a given bottleneck is a *constant* (with possibly a different constant corresponding to each bottleneck). For example, the transmission time required to load a fixed-size packet onto a particular wire is a constant proportional to the inverse bandwidth of the wire. Any packet which must be loaded onto that wire requires the same constant transmission time.

Unless otherwise stated, we assume packets are served at a bottleneck in first-come-first-served (FCFS) order. In some of the future sections we will examine more general non-preemptive service orders (e.g., priority-based).

The delay of a packet is the total time it spends waiting in queues at bottlenecks from the time it is born until it reaches its destination. The average packet delay is the expected delay of a packet entering the network in steady-state.

Since it's the bottlenecks that differentiate packet-routing networks with respect to delays, a packet-routing network is best described in the form of a queueing network which is designed to explicitly show the bottlenecks.

2.2 Recasting a packet-routing network as a queueing network with constant-time servers

A *queueing network* $\mathcal{N}$ is an *abstract model* consisting of *servers* connected by *edges* together with a routing scheme, as shown in Figure 2.2. It's behavior is very similar to our definition of a packet-routing network, except that time is only spent at the servers, and no time is spent on the edges. Thus packets queue up at the servers of $\mathcal{N}$. Packets are born continually at the servers of $\mathcal{N}$, and each packet has a path associated with it that it follows. A queueing network is defined by four parameters:

service-time distribution The service time associated with a server is a random variable from a distribution. (Note the distribution — or just its mean — may be different for each server).

contention resolution protocol The order in which packets are served in case of conflict at a server.

outside arrival process In this thesis, whenever we speak of a queueing network, we will assume that outside arrivals occur at each server according to a Poisson process.

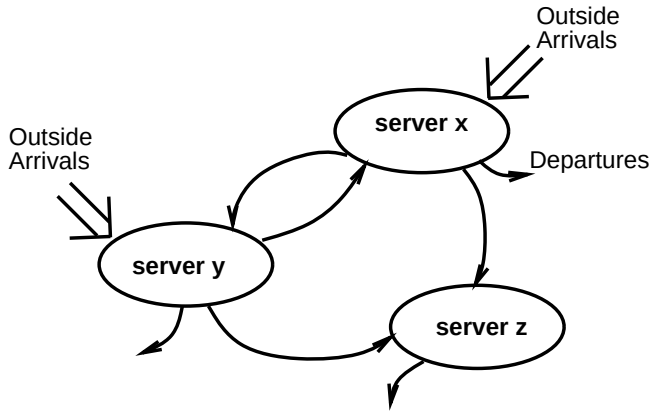
routing scheme Each packet is born with a route which it must follow and each route occurs with some average frequency (possibly zero).

Given a queueing network $\mathcal{N}$ where c_s is the mean service time at server s, define $\mathcal{N}_{C,FCFS}$ (respectively, $\mathcal{N}_{E,FCFS}$) to be the queueing network $\mathcal{N}$ where the service time at server s is a constant c_s (respectively, an independent exponentially distributed random variable with mean c_s) and the packets are served in a First-Come-First-Served order at each server.

Recasting a packet-routing network $\mathcal{P}$ as a queueing network $\mathcal{N}_{C,FCFS}$

Observe that a packet-routing network $\mathcal{P}$ in our model may be described as a queueing network of type $\mathcal{N}_{C,FCFS}$ as follows: *Corresponding to each (constant-time) bottleneck in $\mathcal{P}$, create a server in $\mathcal{N}$.*

For example, if one considers each entire wire of $\mathcal{P}$ to be a bottleneck, then there is a constant time server in $\mathcal{N}_{C,FCFS}$ corresponding to each wire in $\mathcal{P}$, where the service



ROUTING SCHEME:

Possible Packet Path:	Average rate of birth:
x –> y –>	1 every 10 seconds on avg.
x –> y –> z –>	1 every 15 seconds on avg.
y –> z –>	1 every 20 seconds on avg.

Figure 2.2: *For the purposes of this thesis, a queueing network denotes a networked configuration of servers together with a routing scheme.*

time at the server is equal to the time required to traverse the wire in $\mathcal{P}$ (possibly different for each wire). Finer-tuned modeling of transmission times and propagation times is only slightly more complicated: Let t denote the transmission time for the wire, and p denote the propagation time for the wire. Now simply use a chain of FCFS servers in $\mathcal{N}$ to represent each wire in $\mathcal{P}$, where the service time at each server in the chain is t and the number of servers in the chain is $\frac{p}{t}$. In this way we're able to model the fact that several packets may be on a wire in $\mathcal{P}$ at a time, but they must be separated by t, the time to load a packet onto a wire. Furthermore, once loaded onto a wire, the time for a packet to traverse the wire is $t \cdot \frac{p}{t}$, namely p, the propagation time. It is equally easy to use servers in $\mathcal{N}$ to model bottlenecks at the nodes in $\mathcal{P}$.

From now on, we will never refer to packet-routing networks again, but rather we will only address how to compute delays in queueing networks of type $\mathcal{N}_{C,FCFS}$.

2.3 Want to analyze network of constant time servers, but only know how to analyze exponential time servers

In the previous section we saw that most packet-routing networks can be formulated as queueing networks of type $\mathcal{N}_{C,FCFS}$. Our goal is to derive a general formula which applies to an arbitrary queueing network of type $\mathcal{N}_{C,FCFS}$ and returns the average packet delay as a function of the average route frequencies.

Unfortunately, no such formula is known (see Section 2.6). However, a queueing network consisting of FCFS servers with exponentially-distributed service times (i.e., a network of type $\mathcal{N}_{E,FCFS}$) is surprisingly easy to analyze. In fact, queueing theory provides results for not just the average packet delay, but also the distribution of queue sizes at each server and a host of other useful results which apply to arbitrary queueing networks of type $\mathcal{N}_{E,FCFS}$. Queueing network of type $\mathcal{N}_{E,FCFS}$ are a type of "multiclass Jackson queueing networks." The known results for multiclass Jackson queueing networks are explained in queueing textbooks such as [13]. For reference, in Appendix A, we include a subset of these results, tailored to our definition of a queueing network from Figure 2.2.

We briefly explain the reason why networks of exponential time servers are analytically tractable: First observe that networks of exponential-time servers can be modeled by continuous-time Markov chains (CTMC) (see Section 1.3). This is done as follows: Each state of the CTMC consists of the number of packets at each server including the one serving there, plus the class of each packet, where class refers to the packet's route. (Observe that because the service times at each server are exponentially distributed, there is no need to keep track of how long a packet has already been served at a server.) The time spent at each state before transitioning to another state is the time until either a new packet arrives from outside the network or a packet completes service at a server and moves to another server. Thus the time between state transitions is the minimum of several exponential random variables, which is in turn exponentially-distributed as required for a CTMC. The probability of transitioning from state S_i to state S_j next as opposed to S_k next is the probability that the event corresponding to a transition to S_j occurs before the event corresponding to a transition to S_k. This is well-defined as the probability that one exponential random variable is smaller (earlier) than another.

Given a Markov process, we can now solve for the stationary distribution by equating, for each state, the rate at which the process leaves that state with the rate at which it

enters that state ("balance equations"). Although in general balance equations do not necessarily have a closed-form solution, or an easily found one, it turns out (see Appendix A) that the balance equations which are derived from a queueing network of type $\mathcal{N}_{E,FCFS}$ are not only solvable, but have a particularly simple closed-form limiting distribution. In fact, it turns out that the limiting distribution on the number of packets at server i is independent of that at server j, for all i and j. This is known as a "product-form" solution. The product-form observation was first discovered by Jackson [35], [34], and has led to the coining of the term "queueing theory's independence assumptions" whenever exponential service times and outside interarrival times are assumed. Furthermore, the distribution on the number of packets queued at server i is a simple function of the total rate of packets arriving at server i. Given the average queue size at each server, it is then simple to apply Little's formula to obtain the expected packet delay.

By contrast, queueing networks with constant-time servers are almost never tractable, as we'll see in Section 2.6.[1] The above argument doesn't apply to networks of type $\mathcal{N}_{C,FCFS}$ because in general they can't be represented by a countable state Markov model. In particular, the state of the network depends on how much service time the packet has already received (past history).

Unfortunately, although queueing networks of type $\mathcal{N}_{E,FCFS}$ are analytically tractable, they are completely unrealistic as a model of packet-routing networks. For example, consider a packet-routing network with fixed-size packets. The transmission time (time to load a packet onto a wire) should depend only on the bandwidth of the wire. That is, the transmission time for a particular wire can be represented by a server with some *constant* service time. It does not make sense that the the transmission time should be exponentially distributed. It also does not make sense that the transmission time should vary with each packet crossing the wire, or with each iteration of the same packet crossing the same wire. The independence implications of exponentially-distributed service times further lead one to distrust networks of exponential servers as a model of packet-routing networks. For example, it seems false that the queue size at one bottleneck of a packet-routing network

[1] The following is an idea for analyzing networks of constant-time servers which seems plausible, but doesn't work: Replace each constant-time server with service time, say 1, by a chain of n exponential-time servers, each having mean $1/n$ (for large n, the total service time required by the chain of n exponential-time servers is approximately 1). Now analyze the delay in the exponential-server network. The reason this doesn't work is that only one packet at a time serves at the constant-time server, however several packets may simultaneously occupy the corresponding chain of exponential-time servers (at different servers in the chain).

should be independent of the queue size at all other bottlenecks.

Because most queueing-theory formulas such as those described above rely on the clearly unrealistic assumption of exponential service times, many computer scientists are reluctant (or outright unwilling) to use this huge body of queueing formulas for delays and queue sizes when analyzing packet-routing networks, and are suspicious of any analyses obtained using "traditional queueing-theoretic assumptions." As we'll see (Section 2.6) this fear exists even in the area of computer science theory research, where Poisson arrivals are accepted, but exponential service times are not. Allan Borodin commented on exactly this problem in [10].

2.4 Our approach: Bounding delay in $\mathcal{N}_C$ by that in $\mathcal{N}_E$

Our approach for analyzing networks with constant-time FCFS servers is very natural and surprisingly unexplored:

> *Our idea is to bound the average delay of $\mathcal{N}_{C,FCFS}$ (which we care about) by the average delay of the corresponding network $\mathcal{N}_{E,FCFS}$ (which we know how to compute).*

Note above that $\mathcal{N}_{E,FCFS}$ and $\mathcal{N}_{C,FCFS}$ refer to the same queueing network $\mathcal{N}$ (same routing scheme), except that in $\mathcal{N}_{C,FCFS}$ server s has some constant service time c_s, and in $\mathcal{N}_{E,FCFS}$ the service time at server s is exponentially-distributed with mean c_s.

Our overall goal is to identify the class $\mathcal{S}$ of queueing networks $\mathcal{N}$ for which

$$\text{AverageDelay}(\mathcal{N}_{C,FCFS}) \leq \text{AverageDelay}(\mathcal{N}_{E,FCFS}) \tag{2.1}$$

We will sometimes refer to the class $\mathcal{S}$ as the class of boundable networks. In Section 2.9 we give intuition for why we suspect most networks are in $\mathcal{S}$.

2.5 Outline and implications of our results

In this section we outline our results towards determining the class $\mathcal{S}$ of boundable networks.

We say a queueing network has *Markovian routing* if when a packet finishes serving at a server i, the probability that it next moves to some server j (or leaves the network) depends only on where the packet last served and is independent of its previous history (or

route). In the case of Markovian routing, the packets appear indistinguishable since they have no associated route. Thus a queueing network with Markovian routing can simply be described by a directed graph with probabilities on the edges.[2] We will sometimes use the term "non-Markovian routing" to indicate the general model where packets are born with routes. (The term queueing network – when not followed by a descriptor – will still refer to the general model as defined in Section 2.2.)

In Chapter 3 , we give an easy proof showing that all queueing networks $\mathcal{N}$ with Markovian routing are in $\mathcal{S}$, i.e.,

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) \leq \text{AvgDelay}(\mathcal{N}_{E,FCFS}).$$

We also show that $\mathcal{N}_{C,FCFS}$ is stable under the same conditions as $\mathcal{N}_{E,FCFS}$. Our proof is a coupling argument shown to hold on every sample point in a probability space. We give an example showing why this sample point based proof technique can not be extended to general queueing networks (where packets are born with paths).

Significance of this result: The problem of computing the average delay in a packet-routing network where packets are born with random destinations is an important problem in theoretical computer science (see Section 2.6) because most routing schemes are randomized, requiring packets to be sent to random intermediate destinations. Queueing networks with Markovian routing (and constant time servers) are in turn important because they model many[3] packet-routing networks in which the packets have random destinations. A couple common examples are the mesh network with greedy routing (packets are first routed to the correct column and then to the correct row) and the hypercube network with canonical routing (packets cross each dimension if needed in order). For these examples, when the destinations are random, rather than choosing the random destination when the packet is born, we can view the random destination as being decided a little at a time, by flipping a coin after each server. (Here a server corresponds to an edge in the original network). That is, in the above examples, it can be shown that knowledge of the current server alone is enough information to determine the probability of next moving to each of the other servers, so a

[2]Note an equivalent way to define a queueing network $\mathcal{N}$ is to say that each outside arrival to $\mathcal{N}$ is associated with some class. A packet of class ℓ at server i moves next to server j with probability p_{ij}^{ℓ}. In the special case of *Markovian routing*, the queueing network $\mathcal{N}$ has only one class of packets.

[3]But not all.

queueing network with Markovian routing fully describes the packet-routing network. For a more detailed explanation see [72] and [56].

The above result tells us that we can easily compute an upper bound on the average delay for any packet-routing network which can be modeled by a queueing network with Markovian routing.

In Chapter 4 , we find that we are able to prove much more general results for the case of light traffic. Here we no longer need to restrict ourselves to queueing networks with Markovian routing, but rather we can assume the general model where packets are born with a route.

The exact statement of the light-traffic result requires a few definitions, but speaking loosely, we prove that, for light traffic, a queueing network $\mathcal{N}$ has the property that $\text{AvgDelay}(\mathcal{N}_{C,FCFS}) < \text{AvgDelay}(\mathcal{N}_{E,FCFS})$ whenever the statement holds for the case of just two packets (distributed randomly) in the system. We show that for most queueing networks, including most of the standard interconnection networks, it is easy to see immediately that this simple condition is satisfied.

Significance of this result: If a queueing network $\mathcal{N}$ has the property that

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) \leq \text{AvgDelay}(\mathcal{N}_{E,FCFS}) \tag{2.2}$$

when the traffic load is light, then, simulations suggest (see Section 6.4), $\mathcal{N}$ is even more likely to have property 2.2 when the traffic load is heavier. The reason is that for all networks we've simulated the difference:

$$\text{AvgDelay}(\mathcal{N}_{E,FCFS}) - \text{AvgDelay}(\mathcal{N}_{C,FCFS})$$

is an increasing function of the traffic load.

Thus, all networks which have property 2.2 for light traffic are also very likely to be in $\mathcal{S}$. This suggests that many more (non-Markovian) networks are contained in $\mathcal{S}$.

The second significance of this result is as a new analysis technique for upper-bounding delay in light-traffic networks with constant-time servers, which heretofore have eluded analysis (see introduction to Chapter 4).

In Chapter 5 , we demonstrate a queueing network $\mathcal{N}$, s.t.

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) > \text{AvgDelay}(\mathcal{N}_{E,FCFS})$$

Significance of this result: This counterexample disproves the widely believed conjecture that for all networks $\mathcal{N}$, $\mathcal{N}_{C,FCFS}$ has lower average delay than $\mathcal{N}_{E,FCFS}$ (see Section 2.6). Constructing a queueing network which is not in $\mathcal{S}$ is also useful as a tool for defining $\mathcal{S}$, because (as we see in Chapter 6) it provides suggestions for other classes of networks which likely belong in $\mathcal{S}$.

In Chapter 6 , we discuss ongoing work. This includes conjectures for which further classes of networks are likely to be in $\mathcal{S}$, based on careful examination of the counterexample network from Chapter 5. We also present a simulation study, in progress, of how tight a bound AverageDelay($\mathcal{N}_{E,FCFS}$) provides on AverageDelay($\mathcal{N}_{C,FCFS}$), as a function of several network parameters such as load and route lengths.

Significance of this result: Recall that the reason we want to show many networks are in $\mathcal{S}$, is because that gives us a means for upper bounding the average packet delay in these networks. Ideally we would like the upper bound to be as tight as possible. The simulation study above allows us to use characteristics of the network to derive an adjustment factor which can be applied to AverageDelay($\mathcal{N}_{E,FCFS}$) to obtain a better estimate for AverageDelay($\mathcal{N}_{C,FCFS}$).

In Chapter 7 , we elaborate on future work in removing other unrealistic queueing-theoretic assumptions, aside from exponential service times.

2.6 Previous work

As mentioned in Section 2.3, analysis of queueing networks (as defined in Section 2.2) typically assumes that the servers have exponentially-distributed service times, or at least a phase-type service distribution (i.e., a small number of exponentials in sequence). Even light-traffic analysis of queueing networks, which should be easier, relies on the assumption of exponential or phase-type service times (see Reiman and Simon's papers [62], [63]).

Almost all analysis for constant-time servers (or generally distributed service times) is limited to the single-server network, or a chain of servers, (see for example [69, pp. 353–356], [89], [88]).

The theoretical computer science community has predominantly chosen to ignore queueing theory results because they rely on unrealistic service time assumptions. Instead

the theoretical computer science community has decided to concentrate on directly deriving bounds on delays for a few particular networks of type $\mathcal{N}_{C,FCFS}$.

To simplify the analysis, the theoretical computer scientists regard every edge of the packet-routing network as a bottleneck requiring unit time to traverse (i.e., they model a packet-routing network by a queueing network where for every edge in the packet-routing network, there is a *constant-time* server in the queueing network with service time 1). The most commonly used technique for bounding the delay in packet-routing networks is applying Chernoff bounds to bound the maximum number of packets which could possibly need to traverse a given edge during a window of time, with high probability. This yields an upper bound on queue sizes, which in turn yields upper bounds on delay.

Analyzing queueing networks of type $\mathcal{N}_{C,FCFS}$ is very difficult even in the restricted case of a particular topology, particular routing scheme, and equal service times at the servers. Most of the work done by the theoretical computer science community is in analyzing *static* packet-routing networks. Static packet-routing refers to the case where the packets to be routed are all present in the network when the routing commences, and there is exactly one packet per node. Static packet-routing is commonly called permutation routing because the packets are being permuted among the hosts. (The situation we usually refer to where packets arrive continually from outside the network is known as dynamic packet-routing).

Examples of research on *static* packet-routing networks are [44], [45], [82], [80], [4], [77], [28], [3], [16]. All of these are specific to a particular network and a particular routing scheme. They mostly concentrate on the problem of permutation routing, and use the Chernoff bound approach. Some research on static packet-routing networks applies to general networks (see [46] [58]). This research concentrates on worst-case bounds.

In most of the above routing schemes packets are first sent to *random* intermediate destinations so as to get around particularly bad permutations where many packets need to traverse the same edge at the same time.[4] Therefore, there's also been a lot of research which concentrates on computing delays for the case where the final destinations are random (see for example [44], [80], [45], [36]).

In 1990 Tom Leighton pioneered the study of *dynamic* packet-routing networks

[4]Valiant and Brebner first proposed the now common idea of 2-stage randomized routing in 1981, [82]. In the first stage packets are routed to random intermediate destinations, and in the second stage they are routed from the intermediate destinations to their final destinations.

(continual arrivals from outside) using the Chernoff bound approach, see [44], [36]. The outside arrival process is assumed to be discrete Poisson (a new packet is born at each node of the network at every second with probability p). The service time at each server is exactly one second. Leighton restricted his study to analyzing the mesh network (array), with random destinations, and with the greedy routing scheme, where packets first move to their destination's column and from there to their destination's row. Leighton achieves very tight bounds on delay, however *these results are specific to a particular network topology (mesh) and routing scheme (greedy)*. Also, the analysis is extremely involved.

The Chernoff bound analysis applied above to the mesh network could not be applied to the case of the ring network topology where the routing scheme is simply clockwise routing. In fact, there are no known bounds on delay for the ring network with constant-time servers. Coffman, Kahale, and Leighton, [18], were able to closely bound delay for the ring network with Markovian routing under the assumption that the ring size was infinite. We mention this because their approach is interesting. Their idea was to construct a discrete-time Markov chain model for the special case of a ring network with Markovian routing and constant-time servers. As we discussed earlier, the constant servers usually make Markov chain analysis impossible, however Coffman, Kahale, and Leighton added another assumptions that made the Markov model possible: They discretized the outside arrivals, so that packet arrivals were synchronized to the beginning of each second (a new packet is born at each node of the network at the start of every second with probability p), where the service time at every server was also exactly one second. In this way, they did not have to keep track of the amount of time a packet had served at a server. Obtaining a Markov model is not enough, though, because the Markov model may not be simple enough to solve (recall, in the case of exponential time servers, the Markov chain representation has a very simple product-form solution). However, Coffman, Kahale, and Leighton discovered that when they made the additional assumption that the size of the ring was infinite, their Markov chain became tractable.

Observe that the ring network with non-Markovian routing where the destinations are random cannot be modeled by a Markovian queueing network. This is part of what makes the ring queueing network so intractable.

Stamoulis and Tsitsiklis, [72], were the first to apply traditional queueing theory in delay analysis of problems from theoretical computer science, and their work has inspired our own. Their goal was to bound the average packet delay in the hypercube and butterfly

networks, where the packets are routed to random destinations. They ended up proving a more general result, which has inspired our entire approach to delay analysis. Stamoulis and Tsitsiklis proved that for all *layered* (i.e. acyclic) networks $\mathcal{N}$ with Markovian routing, $\mathcal{N}_{E,FCFS}$ has greater average packet delay than $\mathcal{N}_{C,FCFS}$. They then went on to show that the hypercube and butterfly network with canonical routing and random destinations can be modeled by layered queueing networks with Markovian routing of type $\mathcal{N}_{C,FCFS}$. In theorem 1 we generalize the Stamoulis and Tsitsiklis proof by removing their layered assumption and in doing so simplify their ten page proof to a page. Unbeknownst to Stamoulis, Tsitsiklis and to us, Righter and Shanthikumar, [64], earlier proved a slightly more general result for all networks with Markovian routing, namely that $\mathcal{N}_{E,FCFS}$ has greater average packet delay than $\mathcal{N}_{ILR,FCFS}$, where ILR denotes any service time distribution which has an increasing likelihood ratio. However, their proof requires specialized cross-coupling and conditioning arguments, and therefore we choose to present our own elementary one page proof which suffices for our purposes.

Surprisingly our literature search hasn't turned up any other previous work along the lines of bounding delays in networks of constant time servers by networks of exponential time servers. We believe this is due to a widespread belief that greater variance in service times *always* leads to greater average packet delay in queueing networks where the outside arrival process is Poisson, [90], [85], [26], [38], [70]. [5] Thus although no one has explicitly proven that networks of constant time servers always have less delay that the corresponding exponential server networks, everyone already believes it anyway. We even came across a paper that claimed to have proven that $\mathcal{S}$ contains all queueing networks [98], however we found a bug in their proof. In Section 2.9 we explain the intuition behind why it seems so believable that every queueing network should be in $\mathcal{S}$. In Chapter 5 we construct a queueing network which is *not* in $\mathcal{S}$. It is important to note that the counterexample queueing network we construct has a Poisson outside arrival process. It is far simpler to construct a counterexample for the case of batch arrivals. Figure 2.3 indicates why it's easy to create counterexamples for *batch* arrivals. Examples of the batch nature were demonstrated by [91] and [67] in the 70's.

[5] The average packet delay is an increasing function of the variance in the service time distribution for each of the following *single queue* networks: the M/G/1 queue, the M/G/1 queue with batch arrivals, the M/G/1 queue with priorities, and the M/G/k queue, [69, pp.353–356],[89], [88].

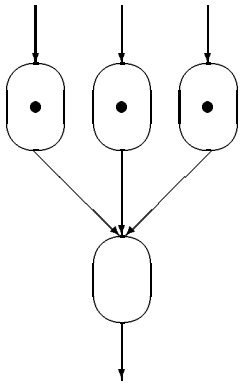


Figure 2.3: *Non-Poisson (in this case batch-Poisson) arrivals can favor more variance in service distributions. For example, if three packets arrive in a batch (serving in the top three servers above), they'll collide at the next server unless their service-completions are staggered.*

2.7 Appeal of our approach

Our approach to upper bounding the delay in a network $\mathcal{N}_{C,FCFS}$ of constant-time servers is to prove that $\mathcal{N}$ belongs to $\mathcal{S}$, and then use the delay in $\mathcal{N}_{E,FCFS}$ as an upper bound. This approach has several appeals. First, unlike the results obtained by the theoretical computer science community, our approach applies to a broad class of networks (all networks in $\mathcal{S}$), rather than requiring a separate proof for each individual topology and routing scheme. Second, it is often easier to prove that the delay in a class of networks of constant-time servers are bounded by the corresponding exponential server networks, than to analyze the constant-time networks directly. Thirdly, as mentioned earlier, most current practical network design and analysis doesn't take advantage of existing queueing theory because of the unrealistic assumptions queueing theory depends on. Our research has taken the first step in replacing one of these unrealistic assumptions (exponential service times) by the more realistic assumption (constant service times). We hope that further research in this direction will motivate more network and routing designers to make use of queueing theory.

2.8 Some necessary preliminary definitions

In the proofs that follow, we will need to make use of a network of constant-time servers where the service order (contention resolution protocol) at each server is processor sharing. In processor sharing, the server is shared equally by all the packets currently waiting at the server. For example, if the service time at the server is 2 seconds, and there are 3 packets waiting at the server, then each packet is being served simultaneously at a rate of $\frac{1}{6}$, and if the packets all started serving at the same time, they will complete service simultaneously 6 seconds later.

Given a queueing network $\mathcal{N}$ where c_s is the mean service time at server s, define $\mathcal{N}_{C,PS}$ to be the queueing network $\mathcal{N}$ where the service time at server s is the constant c_s, and the service order at each server is processor sharing. Our reason for introducing $\mathcal{N}_{C,PS}$ is because of a theorem by [8] and [37] (and described more recently in [84] and [39]) which states that for all queueing networks $\mathcal{N}$,

$$\text{AvgDelay}(\mathcal{N}_{C,PS}) = \text{AvgDelay}(\mathcal{N}_{E,FCFS}).$$

(note this theorem requires that the outside arrival process is a Poisson process).[6]

To prove that a queueing network $\mathcal{N}$ belongs to $\mathcal{S}$, it therefore suffices to prove that

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) \leq \text{AvgDelay}(\mathcal{N}_{C,PS}).$$

Thus, rather than having to compare the delay in a network of constant-time servers with that in the corresponding network of exponential-time servers, we will instead compare two networks, both with constant-time servers, but having different local scheduling at the servers.

2.9 Intuition for why networks of constant servers generally have less delay than networks of exponential servers

We end this chapter with some intuition for why we believe that most queueing networks are contained in $\mathcal{S}$.

[6] The proof of this powerful theorem requires modeling the $\mathcal{N}_{C,PS}$ network by a continuous-time Markov chain. To do this, the constant-time server in $\mathcal{N}_{C,PS}$ is broken into a chain of n exponential-time stages. Any number of packets may be in any stage at once. The rate at which a packet transitions from one stage to another depends on the number of packets occupying the entire chain. It can be shown that this continuous-time Markov chain has a product-form solution equal to that for $\mathcal{N}_{E,FCFS}$.

To begin, consider a queueing network $\mathcal{N}$ consisting of a *single server*. It is easy to prove that $\mathcal{N}$ is in $\mathcal{S}$ ($\mathcal{N}$ is solvable for generally-distributed service times — see [69] for a work conservation argument, or [92] for a tagging argument). Our argument consists of showing that the average delay in $\mathcal{N}_{C,FCFS}$ is less than that in $\mathcal{N}_{C,PS}$. We can actually show something stronger, namely that the statement is true for any arrival sequence, not just a Poisson arrival stream:

Claim 1 *If the same sequence of arrivals is fed into a FCFS queue and into a PS queue, then the i^{th} departure from the FCFS queue occurs no later than the i^{th} departure from the PS queue.*

Proof: Let R be the remaining work at the time of the i^{th} arrival, i.e., the total service time remaining for all packets which haven't yet departed at the time of the i^{th} arrival. Observe that R is the same for both queues. In both queues, the i^{th} arrival must wait for all earlier arrivals to depart before it can depart (i.e., in both queues the i^{th} arrival must wait time R), but only in the PS queue must a packet also wait while later arrivals get service. ■

Now consider a chain $\mathcal{N}$ of n servers, where all servers have mean service time 1 (see Figure 2.4). Packets arrive according to a Poisson process at the head of the chain and the routing scheme simply requires the packets to move through the chain. To see that $\mathcal{N}$ is in $\mathcal{S}$, simply apply Claim 1 followed by $n-1$ invocations of Claim 2 below. (Observe that this argument also holds for the chain network where the servers have unequal service times. Furthermore, the argument also applies to the rooted tree network in Figure 2.5 where every packet route is a tree path starting at the root.)

Claim 2 *If the sequence of arrivals to a (single server) FCFS queue is no later than the arrivals to a PS queue, then the i^{th} departure from the FCFS queue occurs no later than the i^{th} departure from the PS queue.*

Proof: Observe that if the arrivals at the FCFS queue were moved back in time to the times of the arrivals at the PS queue, then the departure times from the FCFS queue would occur later, if anything. The proof now follows from Claim 1. ■

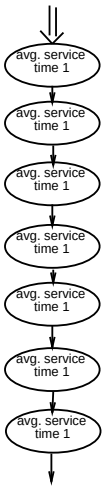


Figure 2.4: *Chain of n servers. $\mathcal{N}_{C,FCFS}$ clearly has less delay than $\mathcal{N}_{E,FCFS}$.*

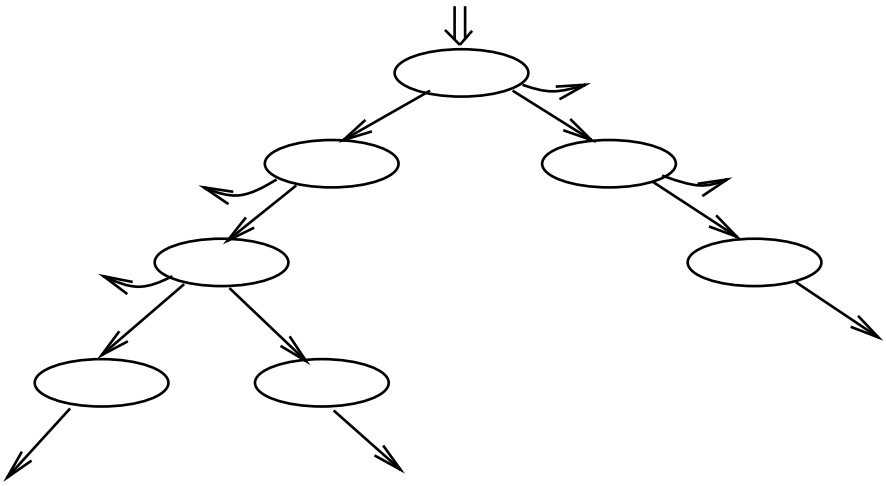


Figure 2.5: *Rooted tree where packet routes are paths starting at the root. $\mathcal{N}_{C,FCFS}$ clearly has less delay than $\mathcal{N}_{E,FCFS}$.*

Examining the chain example in Figure 2.4 in more detail provides intuition for why the average delay in $\mathcal{N}_{C,PS}$ should be greater than that in $\mathcal{N}_{C,FCFS}$ for more general networks, $\mathcal{N}$. Consider the delay experienced by a newly-arriving packet p in chain $\mathcal{N}$. In $\mathcal{N}_{C,FCFS}$, p is only delayed by the packets it finds queued up at the first server. After that initial delay, the packets are spread out and move in lockstep down the chain without interfering with each other. In $\mathcal{N}_{C,PS}$, p is also delayed by the packets it finds queued at the first server (same remaining work). However there are two additional sources of delay for p. First, p may be delayed by later arrivals. Second, p may be become part of a clump, incurring a $\Theta(n)$ delay.

A "clump" is a term we coin to denote a set of packets which meet and are extremely difficult to separate. Figure 2.6 illustrates some examples of clumping. One example of clumping occurs when two packets arrive at almost the same exact time (see Figure 2.6(a)). These packets are almost impossible to separate in $\mathcal{N}_{C,PS}$. Assuming the packets don't separate, they will continue to delay each other all the way down the chain, each incurring a delay of $\Theta(n)$ from the other. Figure 2.6(b) shows an example of how a clump might be broken. However, separating a clump often requires several packets to arrive in succession (to slow down one member of the clump), but that succession of packets itself is then likely to become a clump. One last, somewhat subtle, problem with clumps is that a clump travels much more slowly than all other packets, e.g., a clump of size k travels at $1/k^{\text{th}}$ the speed of other packets. Consequently, it is easier for other packets to catch up to the clump, at which time they may become part of the clump, and the problem worsens. This is illustrated in Figure 2.6(c). As we discuss later, the greater the arrival rate, the greater the clumping possibilities, and the more likely that the average delay in $\mathcal{N}_{C,PS}$ exceeds that in $\mathcal{N}_{C,FCFS}$.

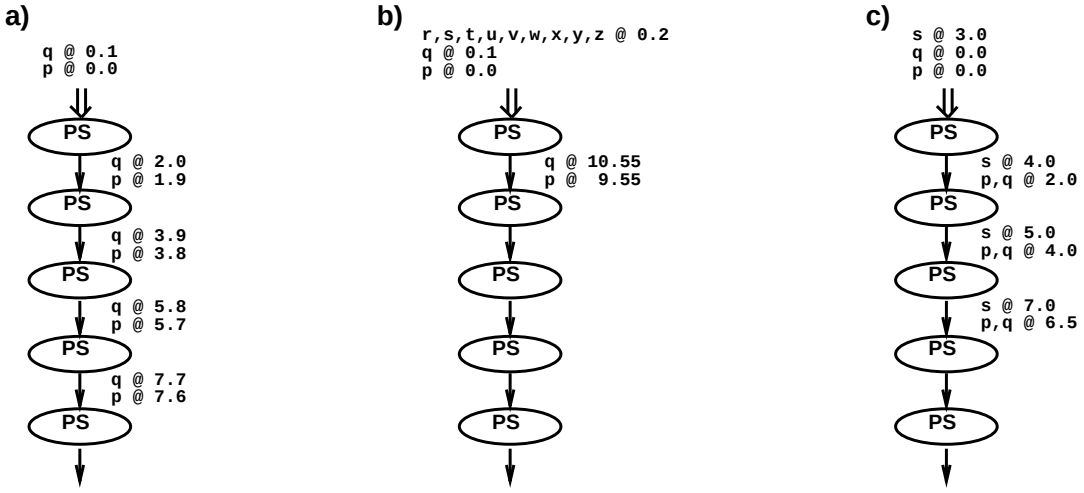


Figure 2.6: *Examples of Clumping. Chain network where all servers have service time* 1. *a) Packets p and q become part of the same clump, and delay each other all the way down the chain. b) Although p and q were initially part of the same clump, they were able to break apart from each other. c) Because clumps move more slowly than regular packets, other packets are likely to join the clump and become part of it.*

Chapter 3

Bounding Delays in Networks with Markovian Routing

In Section 3.1, we give a simple proof that all queueing networks with Markovian routing[1] are in $\mathcal{S}$.[2] Our proof is modeled after [72] who proved this result for *layered* Markovian networks. Whereas their proof uses induction on the levels of the network, we induct on time, thereby obviating the need for a layered network, and simplifying the proof from ten pages to a page. One consequence of our proof is a necessary and sufficient condition for the stability of networks of type $\mathcal{N}_{C,FCFS}$ with Markovian routing (Section 3.2). Our proofs of Section 3.1 and 3.2 also generalize to non-preemptive service orders other than FCFS (Section 3.3).

In Section 3.4 we show the difficulty behind extending the techniques used in our proof to queueing networks with non-Markovian routing (i.e., the general case where packets are born with paths).

3.1 Bounding Average Delay in Networks with Markovian Routing

Theorem 1 *For all queueing networks $\mathcal{N}$ with Markovian routing,*

$$AvgDelay(\mathcal{N}_{C,FCFS}) \leq AvgDelay(\mathcal{N}_{E,FCFS})$$

[1] This terminology was defined at the beginning of Section 2.5.

[2] This proof appeared in our paper [32].

Our proof will actually bound the average delay in $\mathcal{N}_{C,FCFS}$ by that in $\mathcal{N}_{C,PS}$, as defined in Section 2.8.

Our proof that the average delay in $\mathcal{N}_{C,FCFS}$ is bounded by that in $\mathcal{N}_{C,PS}$ will be valid for *any* sequence of outside arrivals, not just a Poisson arrival stream. However we will need the outside arrival process to be Poisson in order to claim that the average delay in $\mathcal{N}_{C,PS}$ is equal to that in $\mathcal{N}_{E,FCFS}$.

We start by recalling Claim 2 from Section 2.9 which states that if the sequence of arrivals to a (single server) FCFS queue is no later than the arrivals to a PS queue, then the i^{th} departure from the FCFS queue occurs no later than the i^{th} departure from the PS queue.

To generalize the statement from the single server to the network, we'll use a coupling argument. Consider the behavior of the two networks when coupled to run on the same *sample point* consisting of:

1. the sequence of arrival times at each server from *outside* the network, and
2. the choices for where the j^{th} packet served at each server proceeds next.

Note the above quantities are all independent for a network with Markovian routing. Also, the j^{th} packet to complete at a particular server in the two networks may not be the same packet. Our proof consists of proving the following claim:

Claim 3 *For a given sample point, the j^{th} service completion at any server of the FCFS network occurs no later than the j^{th} service completion at the corresponding server of the PS network.*

Proof: Assume the claim is true at time t. We show it's true at time $t' > t$, where t' is the time of the next service completion in either network. We distinguish between *outside arrivals* to a server (packets arriving from outside the network) and *inside arrivals* to a server (service completions), and make the following sequence of observations:

- During $[0, t')$, Claim 3 is true.
- Let's suppose the very next service completion is to occur at server q. During $[0, t')$, every arrival at PS server, q_{PS}, must have already occurred at the corresponding FCFS server, q_{FCFS} (see Figure 3.1). (This is true for inside arrivals because any

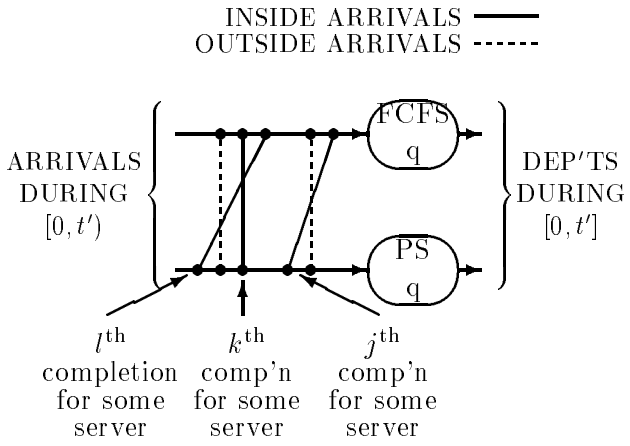


Figure 3.1: *Illustration of proof of Claim 3. Time moves from right to left, i.e., arrivals closer to server q occurred earlier. We consider the same server q in the FCFS network and the PS network. The arrival stream into PS server q is delayed relative to the arrival stream into FCFS server q. Therefore the departure stream out of PS server q is delayed relative to the departure stream out of FCFS server q.*

inside arrival at q_{PS} is, say, the j^{th} service completion at some server q'_{PS}, and by the previous observation, the j^{th} service completion at q'_{FCFS} is at least as early. By definition of the sample point, the observation is also true for outside arrivals.)

- Therefore, during $[0,t')$, the i^{th} packet to arrive at q_{FCFS} arrives no later than the i^{th} arrival at the corresponding server q_{PS} of the PS network.

- Hence, by Claim 2, we see Claim 3 holds during $[0,t']$. This includes the current service completion.

■

By Claim 3, it follows that for any sample point, the i^{th} departure from $\mathcal{N}_{C,FCFS}$ occurs no later than the i^{th} departure from $\mathcal{N}_{C,PS}$. This implies that

Number of packets in $\mathcal{N}_{C,FCFS}$ at time t $\leq_{st}$ Number of packets in $\mathcal{N}_{C,PS}$ at time t

which implies that

$\mathbf{E}\{$Number of packets in $\mathcal{N}_{C,FCFS}$ at time t$\} \leq \mathbf{E}\{$Number of packets in $\mathcal{N}_{C,PS}$ at time t$\}$

So by Little's Law [92, p. 236] we have shown that

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) \leq \text{AvgDelay}(\mathcal{N}_{C,PS})$$

which completes the proof of Theorem 1.

3.2 Stability issues

A necessary condition for the stability of a queueing network is that the total arrival rate into any server of the network (including arrivals from inside and outside the network) is no more than the service rate at the server. The question is whether this is also a sufficient condition for stability.

A sufficient condition for the stability of a queueing network is that the expected time between which all the queues empty out is finite.

Stability is a well understood issue for any network of type $\mathcal{N}_{E,FCFS}$. In the case of $\mathcal{N}_{E,FCFS}$ a sufficient condition for stability is that the average arrival rate into each server is less than the service rate at that server. This is easy to see because, by the product-form distribution of $\mathcal{N}_{E,FCFS}$, under this condition the probability that all the queues are empty is non-zero, thus the expected time between which they all empty is finite.

Since networks of type $\mathcal{N}_{C,FCFS}$ don't satisfy product-form, it is harder to prove sufficient conditions for their stability. However observe that the stochastic ordering in the proof of Theorem 1 immediately implies that for any queueing network $\mathcal{N}$ with Markovian routing, $\mathcal{N}_{C,FCFS}$ is stable whenever $\mathcal{N}_{E,FCFS}$ is. For let $\mathcal{N}$ be any network with Markovian routing and assume that the average arrival rate into each server is less than the service rate at that server. Then,

$$\begin{aligned}
& \mathbf{Pr}\{\text{all queues of } \mathcal{N}_{C,FCFS} \text{ are empty}\} \\
= \;& 1 - \mathbf{Pr}\{\text{total number of packets in } \mathcal{N}_{C,FCFS} > 0\} \\
\geq \;& 1 - \mathbf{Pr}\{\text{total number of packets in } \mathcal{N}_{E,FCFS} > 0\} \\
= \;& \mathbf{Pr}\{\text{all queues of } \mathcal{N}_{E,FCFS} \text{ are empty}\} \\
> \;& 0.
\end{aligned}$$

3.3 Some Easy Generalizations

Observe that the proofs of Claims 2 and 3 do not depend on serving the packets in a FCFS order. For example, packets could be born with priorities, with servers serving higher-priority packets first, in a non-preemptive fashion. In fact, for a given sample point, the time of the j^{th} service completion at a server is the same for every non-preemptive contention resolution protocol which serves one packet at a time.

Naturally, we still require that the packet priorities are independent of the packet route, for otherwise we cannot perform the coupling argument required in the proof of Claim 3.

Interestingly, the stability claim thus also generalizes, even when the contention resolution protocol intentionally tries to starve a particular packet.

3.4 Extending the Sample Point Analysis

Can the proof of Theorem 1 be extended to networks which don't have Markovian routing, i.e. to the general case where each packet is born with a path? If each packet has a preassigned path, we obviously can't couple the two networks on the choices of where the j^{th} packet served at each server proceeds next. However, perhaps it is still possible to use the sample point idea. For example, could we prove that for all networks $\mathcal{N}$, if $\mathcal{N}_{C,FCFS}$ is run on the same outside arrival sequence as the corresponding network $\mathcal{N}_{C,PS}$, then $\mathcal{N}_{C,FCFS}$ has lower average delay than $\mathcal{N}_{C,PS}$? Or could we show that, given both networks are run on the same outside arrival sequence, at any time t the number of packets in $\mathcal{N}_{C,FCFS}$ is no more than that in $\mathcal{N}_{C,PS}$?

The answer to both these questions is no, as evidenced by the very simple network $\mathcal{N}$ in Figure 3.2. The figure shows $\mathcal{N}_{C,FCFS}$ and the corresponding network $\mathcal{N}_{C,PS}$, where both networks are run on the same outside arrival sequence. Although for most outside arrival sequences $\mathcal{N}_{C,FCFS}$ will have lower average delay than $\mathcal{N}_{C,PS}$, on this particular outside arrival sequence, $\mathcal{N}_{C,FCFS}$ behaves worse than $\mathcal{N}_{C,PS}$, both with respect to average delay and with respect to the number of packets present at time t.

However, this is only one bad sample point, and it seems clear that on most sample points $\mathcal{N}_{C,FCFS}$ will have better performance than $\mathcal{N}_{C,PS}$, where $\mathcal{N}$ is the network in Figure 3.2. Thus, $\mathcal{N}_{C,FCFS}$ may still have lower average delay than $\mathcal{N}_{C,PS}$, when the

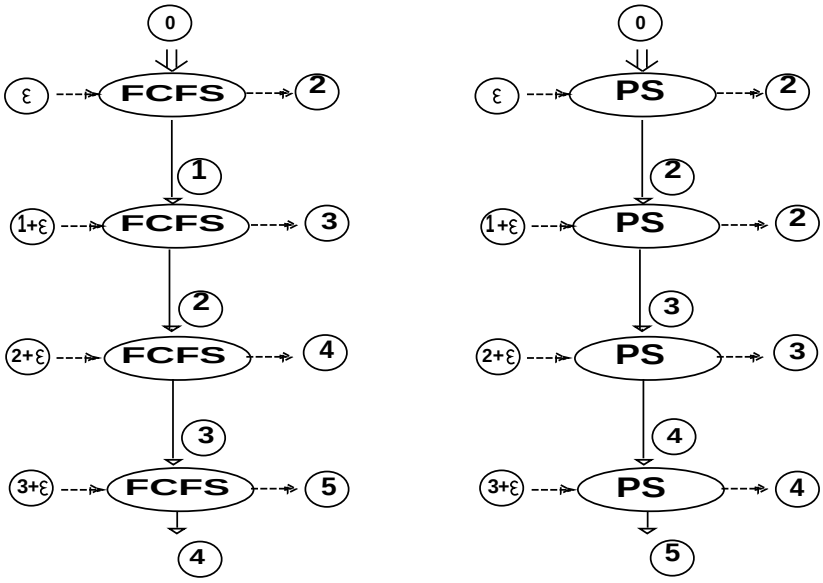


Figure 3.2: *One particular bad sample point for a queueing network with non-Markovian routing. In the above queueing network, $\mathcal{N}$, the packets which arrive on top, move down. The packets which arrive on the left, move right. The packets are labeled with the time at which they arrive at each location. All the servers have service time* 1*. On the left we see $\mathcal{N}_{C,FCFS}$ and on the right $\mathcal{N}_{C,PS}$, where both networks are run on the same exact arrival sequence. Although for most arrival sequences, $\mathcal{N}_{C,FCFS}$ has lower average delay than $\mathcal{N}_{C,PS}$, on this particular bad sample point, $\mathcal{N}_{C,FCFS}$ has greater average delay than $\mathcal{N}_{C,PS}$. Observe that the arrival sequence is constructed specifically so that in $\mathcal{N}_{C,FCFS}$ the packet moving down bumps into every cross packet. This sample point also shows that sometimes the number of packets present in $\mathcal{N}_{C,FCFS}$ is greater than the number in $\mathcal{N}_{C,PS}$. For example at time* 3.5*, there are* 3 *packets present in $\mathcal{N}_{C,FCFS}$ and only two in $\mathcal{N}_{C,PS}$.*

outside arrival processes are Poisson. In fact, our simulations of this particular network indicate this to be the case, although we can't prove it.

Chapter 4

Bounding Delays in the Case of Light Traffic

In Chapter 3 we saw that $\mathcal{S}$ contains all queueing networks with Markovian routing. The question remains whether $\mathcal{S}$ also contains networks which have general (non-Markovian) routing where packets are born with paths. It is easy to prove that $\mathcal{S}$ contains a few, particularly easy to analyze, non-Markovian networks (for example, a single server network where each packet passes exactly twice through the server), but it's difficult to prove that $\mathcal{S}$ contains a general class of non-Markovian networks. As we saw in Section 3.4, we can not apply the technique we used in Chapter 3 to proving that even simple non-Markovian networks are in $\mathcal{S}$, so we need a new technique. In this chapter[1] we give evidence that $\mathcal{S}$ contains most non-Markovian queueing networks. To do this, we start by restricting ourselves to the case of light traffic only.

Let $\lambda_{\mathcal{N}}$ denote the outside arrival rate into queueing network $\mathcal{N}$. Recall $\mathcal{S}$ is the set of queueing networks $\mathcal{N}$ for which

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) \quad \leq \quad \text{AvgDelay}(\mathcal{N}_{C,PS}) \tag{4.1}$$

for all (stable) values of $\lambda_{\mathcal{N}}$. Define $\mathcal{S}_{Light}$ to be the set of queueing networks $\mathcal{N}$ that satisfy equation 4.1 in the case of light traffic, i.e., for sufficiently small $\lambda_{\mathcal{N}}$.

We give a simple sufficient criterion for whether a queueing network is in $\mathcal{S}_{Light}$ (Section 4.1). This simple criterion enables us to prove many networks belong to $\mathcal{S}_{Light}$ which aren't known to belong to $\mathcal{S}$ (Section 4.2). In fact, we are able to show that most

[1] Much of this chapter appeared in our paper [30].

of the standard packet-routing networks are in $\mathcal{S}_{Light}$. This includes the highly intractable ring network, discussed in Section 2.6.

This result has three consequences:

1. *Evidence that many more networks are in* $\mathcal{S}$. By definition $\mathcal{S}$ is contained in $\mathcal{S}_{Light}$. However it also seems likely that $\mathcal{S}_{Light}$ is contained in $\mathcal{S}$. This latter statement is based on many network simulations we have performed (see Section 6.4) which indicate that the difference (AvgDelay($\mathcal{N}_{E,FCFS}$) − AvgDelay($\mathcal{N}_{C,FCFS}$)) increases quickly as the traffic load in the network is increased. Therefore, the above result suggests that many more (non-Markovian) networks are contained in $\mathcal{S}$.

2. *New method for light-traffic analysis of networks of type* $\mathcal{N}_{C,FCFS}$. The above result is also significant as a new simple method for light traffic analysis of networks of type $\mathcal{N}_{C,FCFS}$. All the previous work which we have encountered on light traffic analysis for general queueing networks requires the assumption that the service time distribution at the servers is exponential or phase-type, which doesn't include the case of constant service times which we're interested in analyzing. Reiman and Simon, [62], [63] bound the delay in queueing networks with light traffic where the servers have a phase type service time distribution. Their technique involves computing derivatives of the delay at the point of zero arrival rate. The light traffic analysis technique we use is simpler, and will hopefully lead to other simple light traffic analysis.

3. *Counterexample network.* The theorems from this chapter will form the basis of our counterexample network of Chapter 5.

4.1 Characterizing $\mathcal{S}_{Light}$

In this section we see that, speaking loosely, to test whether a queueing network $\mathcal{N}$ belongs to $\mathcal{S}_{Light}$ it suffices to check whether the expected delay created by exactly 2 packets in $\mathcal{N}_{C,FCFS}$ is smaller than the expected delay created by exactly 2 packets in $\mathcal{N}_{C,PS}$.

Theorem 2 *Given a queueing network,* $\mathcal{N}$, *whenever* $\lambda_{\mathcal{N}} < \frac{1}{8e^2km^2}$,

$$\begin{aligned} P_1 D_1^{FCFS} < \quad & AvgDelay(\mathcal{N}_{C,FCFS}) \quad < P_1(D_1^{FCFS} + \frac{2}{k}) \\ P_1 D_1^{PS} < \quad & AvgDelay(\mathcal{N}_{C,PS}) \quad < P_1(D_1^{PS} + \frac{2}{k}) \end{aligned}$$

where

m = *the length of the longest route in* $\mathcal{N}$*'s routing scheme, where length is measured by total mean time in service. We assume* $m \geq 1$.[2]

D_1^{FCFS} = *Expected delay on arrival,* p*, in* $\mathcal{N}_{C,FCFS}$ *due to* p*'s expected interaction with one arbitrary packet,* q*, which arrives uniformly at random within* m *time units of* p*, assuming no packets other than* p *and* q *are in the network at any time.*

D_1^{PS} = *Expected delay on arrival,* p*, in* $\mathcal{N}_{C,PS}$ *caused by exactly one other arrival,* q*, occurring uniformly at random within* m *time units of* p*, and assuming no packets other than* p *and* q *are in the network.*

$\lambda_{\mathcal{N}}$ = *the total arrival rate into* $\mathcal{N}$ *from outside.*

P_1 = $\mathbf{Pr}\{$ 1 *outside arrival during* $(-m, m)$, *and* $< i$ *outside arrivals total during* $(-im, im)$, $\forall i > 1$. $\}$

k = *a free parameter* ≥ 1.

Observe that k is a free parameter of $\lambda_{\mathcal{N}}$, so the term $\frac{2}{k}$ can be made as small as desired by decreasing $\lambda_{\mathcal{N}}$. We will prove Theorem 2 soon, but we will never apply it directly. The purpose of Theorem 2 is only to derive Corollary 3 and Corollary 6. The goal of the next three corollaries is to express a sufficient condition for a network to belong to $\mathcal{S}_{Light}$. Corollary 3 below states that to show that a queueing network $\mathcal{N}$ is in $\mathcal{S}_{Light}$, it is sufficient to show that $D_1^{FCFS} < D_1^{PS}$ (In Chapter 5, Corollary 6, we will prove that this is also a necessary condition).

Rewriting Theorem 2 we have:

Corollary 1 *Given a queueing network,* $\mathcal{N}$*, whenever* $\lambda_{\mathcal{N}} < \frac{1}{8e^2km^2}$ *then*

$$P_1(D_1^{PS} - D_1^{FCFS}) - P_1\frac{2}{k} < \text{AvgDelay}(\mathcal{N}_{C,PS}) - \text{AvgDelay}(\mathcal{N}_{C,FCFS}) < P_1(D_1^{PS} - D_1^{FCFS}) + P_1\frac{2}{k},$$

where D_1^{FCFS}*,* D_1^{PS}*,* $\lambda_{\mathcal{N}}$*,* P_1*,* k*, and* m *are as defined in the above theorem.*

Applying the left inequality of Corollary 1 we see:

Corollary 2 *Given a queueing network,* $\mathcal{N}$*, if* $D_1^{FCFS} < D_1^{PS} - \frac{2}{k}$ *then*

$$AvgDelay(\mathcal{N}_{C,FCFS}) < AvgDelay(\mathcal{N}_{C,PS}), \qquad \forall \lambda_{\mathcal{N}} < \frac{1}{8e^2km^2},$$

where D_1^{FCFS}, D_1^{PS}, $\lambda_{\mathcal{N}}$, k, and m are as defined in the above theorem.

In the above theorem and corollaries observe that k is a free parameter of $\lambda_{\mathcal{N}}$. Therefore $\frac{2}{k}$ above can be made as small as we wish by decreasing $\lambda_{\mathcal{N}}$. We can use this to simplify corollary 2 as follows: Suppose

$$D_1^{FCFS} < D_1^{PS}$$

Then, there exists some $\delta > 0$ such that

$$D_1^{FCFS} < D_1^{PS} - \delta$$

We can always choose k in Corollary 2 such that $\frac{2}{k} = \delta$. Thus:

Corollary 3 *Given a queueing network, $\mathcal{N}$, if $D_1^{FCFS} < D_1^{PS}$, then*

$$AvgDelay(\mathcal{N}_{C,FCFS}) < AvgDelay(\mathcal{N}_{C,PS}) \ , \quad \forall \lambda_{\mathcal{N}} < \frac{\delta}{8e^2 \cdot 2 \cdot m^2}$$

where D_1^{FCFS}, D_1^{PS}, $\lambda_{\mathcal{N}}$, and m are as defined in the above theorem, and where $\delta < D_1^{PS} - D_1^{FCFS}$.

The remainder of this section is devoted to proving Theorem 2.

Intuition behind proof of Theorem 2:

Theorem 2 roughly states that when the arrival rate is sufficiently low, the average packet delay in both the FCFS and the PS networks depends only on the average delay in the case where there are just two packets in the network. The intuition is as follows: Consider a packet p arriving into a very lightly loaded network. Chances are, there will be no packets in the network during p's time in the network, and so p won't be delayed at all. With some small probability, there will be one other packet, q, in the network during p's time, which may delay p. If we set the arrival rate low enough, we can ensure that the probability that there are more than one packet in the network during p's time in the network is *so* low that the delay contribution from that scenario can be hidden in the $\frac{2}{k}$ term.

Proof of Theorem 2:

By PASTA (Poisson Arrivals See Time Averages), the expected delay that a newly arriving packet experiences is equal to the average packet delay for the network. Let $\mathcal{N}$ be any queueing network. For the case of light traffic (i.e., $\lambda_{\mathcal{N}} < \frac{1}{8e^2 k m^2}$), we will compute

upper and lower bounds on the delay an arrival experiences in $\mathcal{N}_{C,FCFS}$. The proof for $\mathcal{N}_{C,PS}$ is identical and therefore omitted.

To compute an upper bound on the average delay in $\mathcal{N}_{C,FCFS}$, let p represent an arriving packet in $\mathcal{N}_{C,FCFS}$. We will determine an upper bound on the expected delay p experiences. It will be convenient to denote p's arrival time by 0.

The purpose of the next few definitions and lemmas is to break down the expected delay on p into a sum of the delay caused by interactions with 1 packet, the delay caused by interactions with 2 packets, the delay caused by interactions with 3 packets, etc., as shown in Equation 4.2 and Equation 4.3. To make this rigorous, we start with a few definitions. First, let $x(i)$ be a non-negative, integer-valued random variable, where

$$x(i) = \text{ number of arrivals during } (-im, im), \text{ excluding } p.$$

Let I be a non-negative, integer-valued random variable, where

$$I = max\{i : x(i) = i\}$$

(We will see in a moment that the arrival rate is sufficiently low so that I is well-defined). Define

$$E_i : \text{ the event that } I = i.$$

A few quick lemmas which we'll need soon:

Lemma 2.1 *If only i packets are present in $\mathcal{N}_{C,FCFS}$(or $\mathcal{N}_{C,PS}$), they may take at most time im to clear the network.*

Proof: At worst all i packets are on the same path and arrive at the same time. ■

Lemma 2.2 *If for some i, $x(i) > i$, then $\exists j > i$ such that $x(j) = j$, ... so $I \geq j$ with probability* 1. *Furthermore, with probability* 1, *$\exists j$ s.t. $x(j') < j'$, $\forall j' > j$.*

Proof: Observe that $x(i)$ is a non-decreasing integer-valued function of i. Let $\mathcal{L}$ be the line $x(i) = i$, as shown in Figure 4.1. Since $\mathbf{E}\{x(i)\}$ is less than i, if $x(i)$ is ever above $\mathcal{L}$, with probability 1 it must eventually cross $\mathcal{L}$ and come below $\mathcal{L}$ (by the Law of Large Numbers). Furthermore, with probability 1, there will be some point after which $x(i)$ will never again

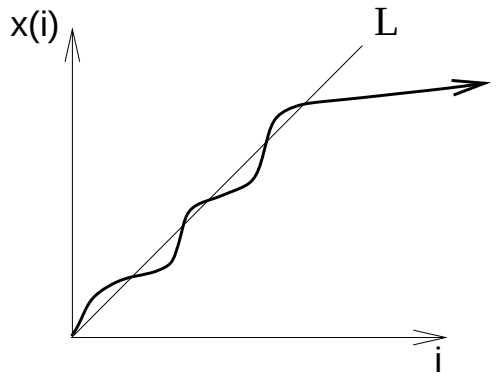


Figure 4.1: *Illustration of proof of Lemma 2.2. The last point at which $x(i)$ crosses $\mathcal{L}$ is referred to as I.*

cross above $\mathcal{L}$. ■

Lemma 2.3 $I = i \iff x(i) = i$ and $\forall j > i,\ x(j) < j$.

In particular, $I = 0 \iff x(i) < i$ *for all* i.

Proof: This follows from Lemma 2.2 above. ■

Rephrasing Lemma 2.2,

Lemma 2.4 *The E_i's are disjoint and $\bigcup E_i = \Omega$ with probability* 1.

We can now use Lemma 2.4 to express the expected delay on p (i.e., the expected time p spends waiting in queues during its time in $\mathcal{N}$).

$$\mathbf{E}\{\text{Delay on } p\} \quad = \quad \sum_i \mathbf{Pr}\{E_i\} \cdot \mathbf{E}\{\text{Delay on } p \mid E_i\} \tag{4.2}$$

To interpret the above expression, we apply Lemma 2.5 below, which allows us to rewrite equation 4.2 as equation 4.3. At this point we will have expressed the expected delay on p as a sum of the expected delay on p due to interactions with just one packet, just two packets, etc.

Lemma 2.5 *Let*

$$D_i = \mathbf{E}\{\textit{Delay on } p \mid E_i\}$$

Then

$$D_i = \mathbf{E}\{\textit{Delay on } p \mid x(i) = i \textit{ and no other packets are present in } \mathcal{N} \textit{ ever.}\}$$

Proof: Recall (see Lemma 2.3) that E_i is the event that there are exactly i arrivals in $(-im, im)$ and, for all $j > i$, there are strictly fewer than j arrivals in $(-jm, jm)$. Figure 4.2a below is an illustration of E_i.

We will show that the remaining work (i.e., remaining service time required by all packets in the network) at time $-im$ is zero given E_i; and therefore, the only packets which can possibly interfere with p are those packets which arrive during $(-im, im)$.

Observe that moving arrivals forward in time in Figure 4.2 (as much as permitted by E_i) can only increase the amount of remaining work left at time $-im$. Therefore, to maximize the remaining work at time $-im$, we assume each arrival occurs as early as it is possibly allowed to, as shown in Figure 4.2b. But in this worst case, the remaining work at time $-im$ is 0.

Note: the proof is symmetric if we consider arrivals occurring after p. In that case the worst case is that all i arrivals arrive at time 0. Observe that all the work brought in by p and the i arrivals will be completed by time $(i+1)m$. ■

Setting

$$P_i = \mathbf{Pr}\{E_i\}$$

in Equation 4.2 above, we now have:

$$\begin{aligned}
&\mathbf{E}\{\text{Delay on } p\} \\
&\quad = \sum_i P_i \cdot D_i \\
&\quad = \sum_i P_i \cdot \mathbf{E}\{\text{Delay on } p \mid \text{exactly } i \text{ arrvls. in } (-im, im) \ \&\ \text{no other packets in } \mathcal{N}\} \qquad (4.3)
\end{aligned}$$

From the proof of Lemma 2.1 we can upper bound the D_i terms,

$$D_i \leq im$$

So Equation 4.3 becomes:

$$\begin{aligned}
\mathbf{E}\{\text{Delay on } p\} &= \sum_i P_i \cdot D_i && (4.4) \\
&\leq P_0 \cdot 0 + P_1 \cdot D_1 + P_2 \cdot 2m + P_3 \cdot 3m + \ldots && (4.5)
\end{aligned}$$

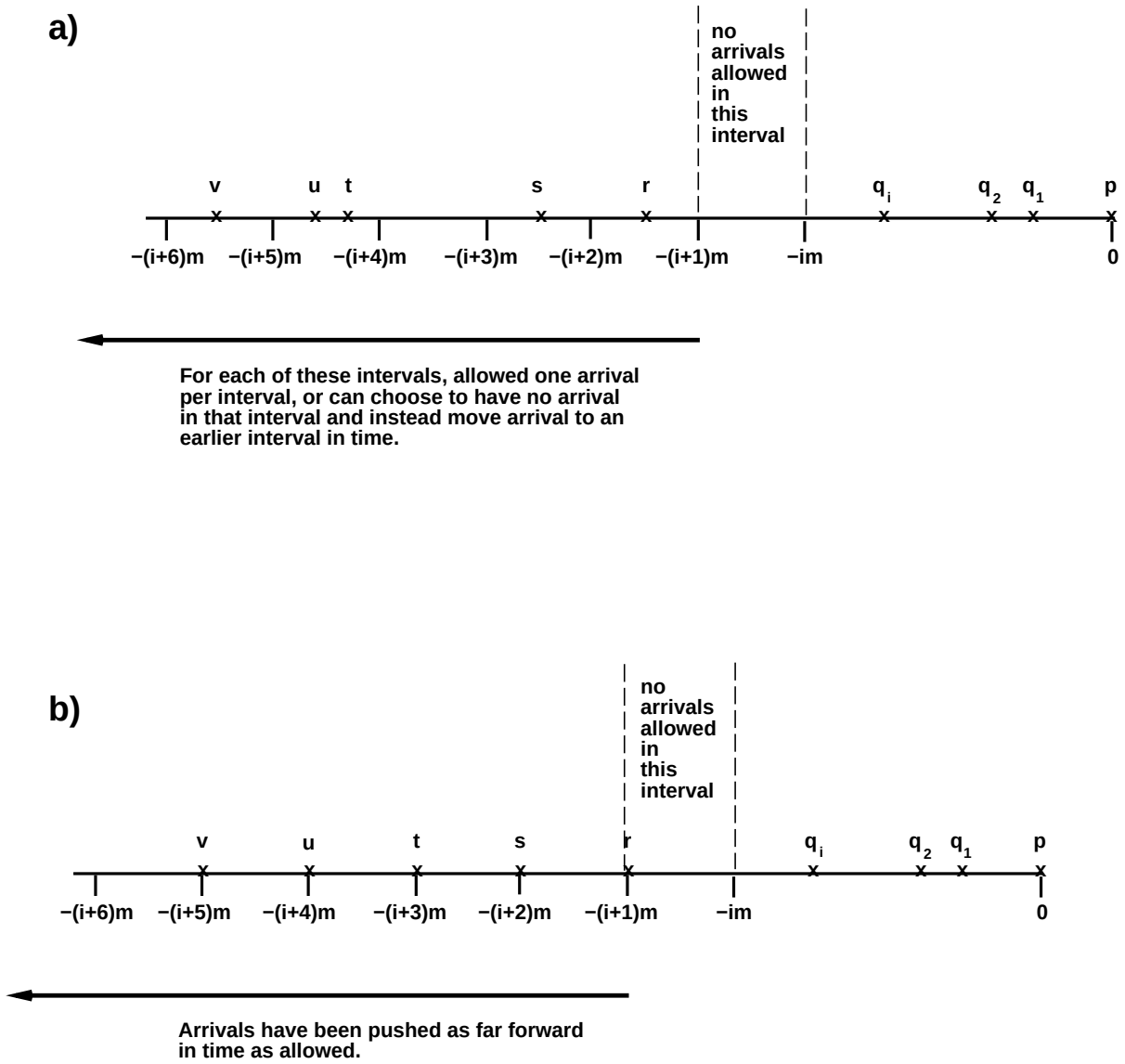


Figure 4.2: (a) *Example of event E_i. Time is broken into intervals of size m. The* $\mathbf{x}$*'s represent packet arrivals, and the packets are labeled with letters. Observe that there are i arrivals in $(-im, im)$. Also, there are no arrivals allowed in $(-(i+1)m, -im)$, because the number of arrivals in $(-(i+1)m, (i+1)m)$ must be strictly less than $(i+1)m$.* (b) *Event E_i which maximizes the remaining work at time $-im$.*

We now want to show that when $\lambda_{\mathcal{N}}$ is small enough (as stated in the theorem), all the terms in the summation starting with P_2 and on are negligible. The problem is that we don't know how to compute the P_i's. However we can easily compute an upper bound on P_i. Let

$$\hat{P}_i = \mathbf{Pr}\{x(i) = i\}$$

then

$$P_i = \mathbf{Pr}\{x(i) = i \text{ and } x(j) < j \text{ for all } j > i\} \leq \hat{P}_i,$$

So we can rewrite equation 4.5 above as

$$\mathbf{E}\{\text{Delay on } p\} \leq P_1 \cdot D_1 + \hat{P}_2 \cdot 2m + \hat{P}_3 \cdot 3m + \ldots \tag{4.6}$$

We now show that when $\lambda_{\mathcal{N}}$ is small enough (as stated in the theorem), all the $\hat{P}_i$ terms above are negligible. By definition of the Poisson Process,

$$\hat{P}_i = \frac{e^{-\lambda_{\mathcal{N}} \cdot 2im} (\lambda_{\mathcal{N}} \cdot 2im)^i}{i!}$$

For $i \geq 2$, we can express $\hat{P}_i$ in terms of $\hat{P}_1$ as follows:

$$\begin{aligned}
\hat{P}_i(i \geq 2) &= \frac{e^{-\lambda_{\mathcal{N}} \cdot 2im} (\lambda_{\mathcal{N}} \cdot 2im)^i}{i!} \\
&= \frac{i^i}{i!} \cdot e^{-\lambda_{\mathcal{N}} \cdot 2im} (\lambda_{\mathcal{N}} \cdot 2m)^i \\
&< e^i \cdot e^{-\lambda_{\mathcal{N}} \cdot 2m} (\lambda_{\mathcal{N}} \cdot 2m)^i \\
&= \hat{P}_1 \cdot (\lambda_{\mathcal{N}} \cdot 2m)^{i-1} \cdot e^i
\end{aligned}$$

Substituting $\lambda_{\mathcal{N}} = \frac{1}{8e^2 km^2}$, we have:

$$\begin{aligned}
\hat{P}_i(i \geq 2) &< \hat{P}_1 \cdot (\lambda_{\mathcal{N}} \cdot 2m)^{i-1} \cdot e^i \\
&= \hat{P}_1 \cdot \left(\frac{1}{4e^2 km}\right)^{i-1} \cdot e^i \\
&< \hat{P}_1 \cdot \frac{1}{k 4^{i-1} m}
\end{aligned} \tag{4.7}$$

Now, substituting $\hat{P}_i$, $i \geq 2$ into the formula for the expected delay on p, we have:

$$\begin{aligned}
\mathbf{E}\{\text{delay on } p\} &\leq P_1 D_1 + \hat{P}_2(2m) + \hat{P}_3(3m) + \ldots \\
&< P_1 D_1 + \frac{\hat{P}_1}{2k} + \frac{\hat{P}_1}{2^2 k} + \frac{\hat{P}_1}{2^3 k} + \ldots \\
&= P_1 D_1 + \hat{P}_1 \cdot \frac{1}{k}
\end{aligned} \tag{4.8}$$

We now apply one last lemma to convert from $\hat{P}_1$ back to P_1.

Lemma 2.6

$$\frac{1}{2} \cdot \hat{P}_1 \leq P_1 \leq \hat{P}_1$$

Proof: The right inequality is true by equation 4.1 above. To see the left inequality, let

$$A_i : \text{ the event that } x(i) = i.$$

$$A'_i : \text{ the event that } x(i) \geq i.$$

and observe from the proof of Lemma 2.2 that

$$\mathbf{Pr}\left\{\bigcup_{i\geq k} A'_i\right\} = \mathbf{Pr}\left\{\bigcup_{i\geq k} A_i\right\}, \forall k$$

$$\begin{aligned}
P_1 &= \mathbf{Pr}\{\, x(1) = 1 \; and \; x(i) < i \text{ for all } i \geq 2\} \\
&= \hat{P}_1 - \mathbf{Pr}\left\{\bigcup_{i\geq 2} A'_i\right\} \\
&= \hat{P}_1 - \mathbf{Pr}\left\{\bigcup_{i\geq 2} A_i\right\} \\
&\geq \hat{P}_1 - \sum_{i\geq 2} \mathbf{Pr}\{A_i\} \\
&= \hat{P}_1 - \sum_{i\geq 2} \hat{P}_i \\
&> \hat{P}_1 - \frac{1}{2} \cdot \hat{P}_1 \quad \text{(by equation 4.7)} \\
&= \frac{1}{2} \cdot \hat{P}_1
\end{aligned}$$

■

Finally from Equation 4.8 and Lemma 2.6 we obtain the statement of the theorem:

$$\mathbf{E}\{\text{delay on } p\} \leq P_1(D_1 + \frac{2}{k})$$

To derive a simple lower bound for the expected delay in $\mathcal{N}_{C,FCFS}$, we go back to equation 4.4.

$$\begin{aligned} \mathbf{E}\{\text{Delay on } p\} &= \sum_i P_i \cdot D_i \\ &\geq P_1 \cdot D_1 \end{aligned}$$

□

Remarks on the proof:

It should now be clear why the arrival rate, $\lambda_{\mathcal{N}}$, had to be set so low. If we had obtained a better estimate on D_2, D_3, etc., we would not have required exponentially decreasing $\hat{P}_i$'s, and we could have tolerated a higher $\lambda_{\mathcal{N}}$.

4.2 Networks in $\mathcal{S}_{Light}$

As we saw in Corollary 3, to show that $\mathcal{N}$ is in $\mathcal{S}_{Light}$, it suffices to check whether that the expected delay created by exactly two packets (distributed randomly) in $\mathcal{N}_{C,FCFS}$ is smaller than the expected delay created by exactly two packets (distributed randomly) in $\mathcal{N}_{C,PS}$ (i.e., we can assume that there are no packets other than those two are in the network). In this section we show that many queueing networks satisfy this easy test, and in particular that most of the standard packet-routing networks satisfy this test. Throughout this section we will consider queueing networks with non-Markovian routing (i.e., the general case where packets are born with routes).

Consider the class of queueing networks with the following property **P1**: any two packet routes which intersect and then split up can never subsequently rejoin. Figure 4.3 illustrates property **P1**. To prove that all network with property **P1** are in $\mathcal{S}_{Light}$, consider two packets in $\mathcal{N}_{C,FCFS}$, and view the *same* two packets in $\mathcal{N}_{C,PS}$, and assume there are no other packets in the network. The two packets don't affect each other at all until they collide. the first time the two packets collide will occur at exactly the same time and will be at exactly the same location in $\mathcal{N}_{C,FCFS}$ and $\mathcal{N}_{C,PS}$. The delay incurred on the packets during the period of intersection will be smaller in $\mathcal{N}_{C,FCFS}$ than in $\mathcal{N}_{C,PS}$ (see Section 2.9). After the routes split up, the two packets will never again see each other in either network. Since the delay in every particular case of two packets is smaller in $\mathcal{N}_{C,FCFS}$ than $\mathcal{N}_{C,PS}$, certainly the average delay for the case of two packets is smaller in $\mathcal{N}_{C,FCFS}$ than $\mathcal{N}_{C,PS}$.

Most common packet-routing networks have property **P1**. For example, the ring network where packets are routed clockwise around the ring from their source directly to

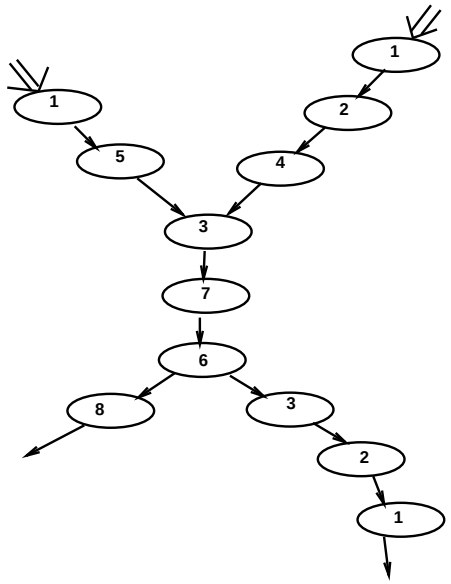


Figure 4.3: *Illustration of queueing network property* **P1**. *Any two packet routes in the queueing network which intersect and then split up, can never subsequently rejoin. The number indicates the service time associated with the server, which may be arbitrary.*

their destination satisfies property **P1**. The hypercube network with the standard canonical routing (packets are routed from their source to their destination by considering each dimension in order and either crossing it, or not) satisfies property **P1**. The mesh network with the standard greedy routing (packets are first routed to the correct column and then to the correct row) satisfies **P1**. So does the butterfly network with any routing scheme, and variants thereof like the omega network, the flip network, etc. The banyan network by its definition has property **P1** as well. Observe the above results do not follow from Chapter 3, because the above packet-routing networks are not necessarily Markovian.[3]

Next consider the class of queueing networks with the property **P2**: packet routes may intersect repeatedly, but the service time between the intersection points of any two routes must be equal. Figure 4.4 illustrates property **P2**. All networks with property **P2** are in $\mathcal{S}_{Light}$. The argument is similar to that above, but awkward to describe. Again consider two packets in $\mathcal{N}_{C,FCFS}$, and view the *same* two packets in $\mathcal{N}_{C,PS}$, and assume there are no other packets in the network. If the two packets collide, the collision will occur at the same server s in $\mathcal{N}_{C,FCFS}$ and $\mathcal{N}_{C,PS}$. Furthermore, the separation between the two packets

[3]We have not made any requirements about random destinations, and even if we had, that isn't necessarily enough to ensure that the network is Markovian, for example in the case of the ring network.

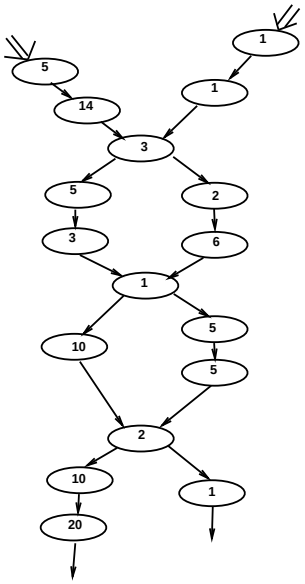


Figure 4.4: *Illustration of queueing network property* **P2***: packet routes may intersect repeatedly, but the service time between the intersection points of any two routes must be equal. In the picture, one route follows the left branches, the other route follows the right branches. The number indicates the service time associated with the server.*

arriving at s, *diff*, will be the same in both networks. The delay incurred on the packets during the meeting at s will be smaller in $\mathcal{N}_{C,FCFS}$ than in $\mathcal{N}_{C,PS}$. When the packets leave s, they will be separated by *diff* in $\mathcal{N}_{C,PS}$, but by c_s in $\mathcal{N}_{C,FCFS}$, where c_s is the service time at server s. At all subsequent path intersection points, the separation between the two packets in $\mathcal{N}_{C,FCFS}$ will be at least as great as that in $\mathcal{N}_{C,PS}$, thus the delay will be smaller in $\mathcal{N}_{C,FCFS}$ than in $\mathcal{N}_{C,PS}$. Since the delay in every particular case of two packets is smaller in $\mathcal{N}_{C,FCFS}$ than $\mathcal{N}_{C,PS}$, certainly the average delay for the case of two packets is smaller in $\mathcal{N}_{C,FCFS}$ than $\mathcal{N}_{C,PS}$.

The only case left is that where two packet routes may intersect repeatedly but the service time between the intersection points is not equal. This case is difficult to analyze because it's difficult to couple the $\mathcal{N}_{C,FCFS}$ and $\mathcal{N}_{C,PS}$ networks with respect to the two packets, as we have above. The problem is that two packets may arrive at a server at exactly the same time in $\mathcal{N}_{C,FCFS}$. This creates non-determinism, since either packet is equally likely to go first, which makes a difference when the two path lengths are not equal. We illustrate an example of this behavior by considering a special case of networks where *all servers have the same service time,* 1. Consider two packets in $\mathcal{N}_{C,FCFS}$, and view the *same*

two packets in $\mathcal{N}_{C,PS}$, and assume there are no other packets in the network. If the two packets collide, it will occur at the same location in both networks. That is, both packets arrive at some server, s, within *diff* seconds of each other, where *diff* < 1, and the packets delay each other at s in both networks. Now suppose the paths split, and rejoin, several times. If the two packets collide for the second time in $\mathcal{N}_{C,FCFS}$, they will necessarily do so in $\mathcal{N}_{C,PS}$ as well, however it is no longer true that the delay at this collision will be greater in $\mathcal{N}_{C,PS}$ than in $\mathcal{N}_{C,FCFS}$. The problem is that when the two packets collide for the second time in $\mathcal{N}_{C,FCFS}$, at server s', they necessarily arrive at s' at the exact same time (in $\mathcal{N}_{C,PS}$, they arrive separated by $(1 - \mathit{diff})$). Now at this point, either packet may serve first in $\mathcal{N}_{C,FCFS}$, but it is predetermined which packet serves first in $\mathcal{N}_{C,PS}$. Thus, if the packets meet for a third time in $\mathcal{N}_{C,FCFS}$, they will not necessarily meet at that point in $\mathcal{N}_{C,PS}$. Thus we can't easily couple the networks to say that for whichever points delay is experienced in $\mathcal{N}_{C,FCFS}$, more delay is experienced in $\mathcal{N}_{C,PS}$ at those points.

However, that does not mean that some other argument can't be used to show that the expected delay for the case of exactly two packets in $\mathcal{N}_{C,PS}$ is greater than that in $\mathcal{N}_{C,FCFS}$. In fact, we believe it to be the true, in the case where all servers have the same service time, 1. We provide some intuition only, since we have no proof:

The point is that in $\mathcal{N}_{C,FCFS}$, in the case where all servers have the same service time 1, it is difficult to force two packets to meet repeatedly, therefore it is unlikely that the 2-packet delay in $\mathcal{N}_{C,FCFS}$ significantly exceeds that in $\mathcal{N}_{C,PS}$. To see this, observe that once two packets collide in $\mathcal{N}_{C,FCFS}$, from then on they are synchronized to be exactly 1 second apart. Now, since all servers have service time 1, the next time the two packets collide will be such that they both arrive at some server at exactly the same time. But, then either packet may go first, so in order to assure that the two packets necessarily collide again, the network must be designed to assure that regardless of whether the packet on the left route was served first, or the one on the right route was served first, the packets will necessarily reunite at some point again. It turns out (we don't include the details) that it is in fact possible to create a network $\mathcal{N}$ that forces the two packets to meet $O(\sqrt{n})$ times in $\mathcal{N}_{C,FCFS}$, where n is the number of servers in $\mathcal{N}$. However, in the corresponding network, $\mathcal{N}_{C,PS}$, it turns out the number of collisions is similar. Furthermore, the probability that the two packets meet in the first place in $\mathcal{N}_{C,FCFS}$ or $\mathcal{N}_{C,PS}$ is extremely small, so the expected delay in both $\mathcal{N}_{C,FCFS}$ and $\mathcal{N}_{C,PS}$ turns out to be $o(1)$ in this case.

We will return to this problem again in the conjectures of Section 6.1.

Chapter 5

Counterexample: an unboundable network

Our goal in this chapter is to construct a queueing network[1] which does not belong to $\mathcal{S}$.[2] That is we demonstrate an $\mathcal{N}$ for which

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) > \text{AvgDelay}(\mathcal{N}_{C,PS}) = \text{AvgDelay}(\mathcal{N}_{E,FCFS}).$$

As explained in Section 2.6, this result disproves the widely held belief that queueing networks with exponential-time servers always have greater average delay than the corresponding networks with constant-time servers. Therefore we refer to $\mathcal{N}$ above as the "counterexample network." The construction of this counterexample network is important in defining $\mathcal{S}$, particularly because it gives us intuition for suggesting which further classes of networks *do* belong to $\mathcal{S}$ (see Chapter 6).

Our approach to finding a network not in $\mathcal{S}$ is to search for a network not in $\mathcal{S}_{Light}$. Once we find a network not in $\mathcal{S}_{Light}$, that network also won't be in $\mathcal{S}$ because, by definition (see Chapter 4), $\mathcal{S}$ is contained in $\mathcal{S}_{Light}$.

Recall in Chapter 4 we proved a simple *sufficient* condition for a network $\mathcal{N}$ to be in $\mathcal{S}_{Light}$. In this chapter, we will prove a *necessary* property of all networks in $\mathcal{S}_{Light}$ (see Section 5.1). We will then construct a network $\mathcal{N}$ which doesn't have this necessary property, and therefore is not in $\mathcal{S}_{Light}$ (see Sections 5.2 and 5.3). Since this property is easy to verify, the proof that our $\mathcal{N}$ is not in $\mathcal{S}_{Light}$ will be short (see Section 5.4).

[1] Just a reminder, throughout this thesis when we refer to a queueing network we assume the arrival process from outside the network is Poisson, unless otherwise specified.

[2] Much of the work in this chapter appeared in our paper, [32].

5.1 A necessary condition for networks in $\mathcal{S}_{Light}$

We present a sequence of corollaries to Theorem 2 from Chapter 4 which result in a necessary property of all networks in $\mathcal{S}_{Light}$.

Corollary 4 *Given a queueing network, $\mathcal{N}$,*

$$\textit{if} \quad AvgDelay(\mathcal{N}_{C,FCFS}) < AvgDelay(\mathcal{N}_{C,PS}) \quad \textit{for some } \lambda_{\mathcal{N}} \ < \frac{1}{8e^2km^2},$$

$$\textit{then} \quad D_1^{FCFS} < D_1^{PS} + \frac{2}{k}$$

where D_1^{FCFS}, D_1^{PS}, $\lambda_{\mathcal{N}}$, k, and m are as defined in Theorem 2.

Proof: Set the lower bound on AvgDelay($\mathcal{N}_{C,FCFS}$) to be less than the upper bound on AvgDelay($\mathcal{N}_{C,PS}$), in Theorem 2. ■

We now present corollaries paralleling Corollaries 2 and 3.

Corollary 5 *Given a queueing network, $\mathcal{N}$,*

$$\textit{if} \quad D_1^{FCFS} > D_1^{PS} + \frac{2}{k},$$

$$\textit{then} \quad AvgDelay(\mathcal{N}_{C,FCFS}) > AvgDelay(\mathcal{N}_{C,PS}) \quad \forall \lambda_{\mathcal{N}} \ < \frac{1}{8e^2km^2},$$

where D_1^{FCFS}, D_1^{PS}, $\lambda_{\mathcal{N}}$, k, and m are as defined in Theorem 2.

Corollary 6 *Given a queueing network, $\mathcal{N}$,*

$$\textit{if} \quad D_1^{FCFS} > D_1^{PS},$$

$$\textit{then} \quad AvgDelay(\mathcal{N}_{C,FCFS}) > AvgDelay(\mathcal{N}_{C,PS}) \quad \forall \lambda_{\mathcal{N}} \ < \frac{\delta}{8e^2 \cdot 2 \cdot m^2},$$

where $\delta < D_1^{FCFS} - D_1^{PS}$, and D_1^{FCFS}, D_1^{PS}, $\lambda_{\mathcal{N}}$, k, and m are as defined in Theorem 2.

From Corollary 6 above, it's clear our goal is to construct a network $\mathcal{N}$ for which:

$$D_1^{FCFS} > D_1^{PS}.$$

We'll construct a network for which:

$$D_1^{FCFS} >> D_1^{PS}.$$

5.2 Intuition for constructing a network not in $\mathcal{S}_{Light}$

We want to construct a network $\mathcal{N}$ for which the expected delay created by two packets in $\mathcal{N}_{C,FCFS}$ is high, given that there are no other packets in the network. To do this, we need a network with two properties:

1. We need to guarantee that with significant probability the two packets, p and q, will meet at least once, because if they never meet, there will be no delay. The "probability" here is over the location of packet q in the network when packet p enters the network.

2. Once the two packets do meet for the first time, we need some way of forcing them to continue to meet over and over again in $\mathcal{N}_{C,FCFS}$, so that they delay each other repeatedly.

The second property above will depend critically on having servers with different service times.

5.3 Network Description and Properties

Let $\mathcal{N}$ be the queueing network shown in Figure 5.1. The servers in $\mathcal{N}$ either have service time 1 or ϵ, as shown. There are n servers with service time 1 and $\frac{n}{2}$ with service time ϵ. The only outside arrivals are into the top server. Half the arriving packets are of type *solid* and half are of type *dashed* (by "type" we mean class). Packets of type *solid* are routed straight down, only passing through the time 1 servers. Packets of type *dashed* are routed through the dashed edges, i.e. through all the ϵ-servers and through every other 1-server. Packets arrive from outside $\mathcal{N}$ according to a Poisson Process with rate $\lambda_{\mathcal{N}}$, where in accordance with Corollary 6, we set $\lambda_{\mathcal{N}} = \frac{1}{8e^2 \cdot 2 \cdot n^2}$.

Observe that $\mathcal{N}$ has both of the properties described in Section 5.2: Because the *dashed*-type packets travel twice as fast as the *solid*-type packets, two packets of opposite type have a significant probability of catching up to each other. Also, (see Figure 5.2) once the packets do meet for the first time in $\mathcal{N}_{C,FCFS}$, the ϵ-servers force the packets to repeatedly crash thereafter. The key is that although packets p and q in Figure 5.2 both arrive at the same server almost simultaneously (time 4, then time 6, then time 8, etc.), in

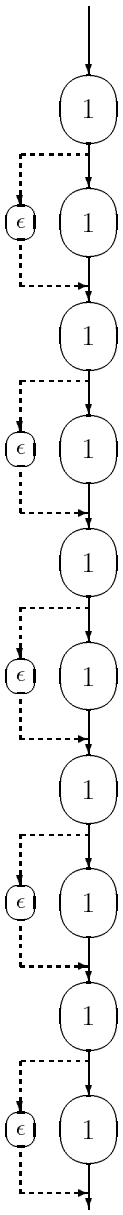


Figure 5.1: *Counterexample network $\mathcal{N}$ with n servers of mean service time 1 and $n/2$ with mean service time ϵ. Packets arrive at the top; half follow the dashed route, while half follow the solid route.*

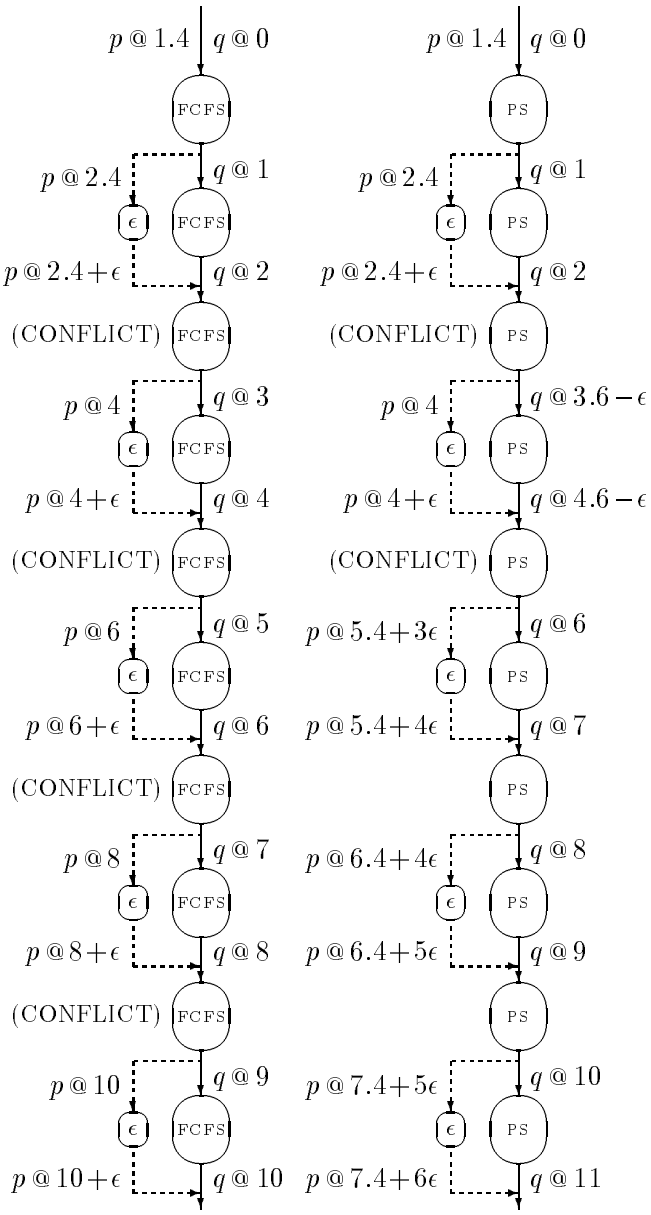


Figure 5.2: *Example illustrating how a packet, p, of type* dashed *and a packet, q, of type* solid *clash repeatedly in* $\mathcal{N}_{C,FCFS}$*, but only twice in* $\mathcal{N}_{C,PS}$*. The notation "p @ 1.4" indicates that packet p reaches that location at time* 1.4.

every case the ϵ-servers force q to arrive *just before* p, thus continuing the crash pattern.[3]

5.4 Delay analysis of counterexample network

The analysis is very simple. Let p denote a packet arriving into $\mathcal{N}$ at time 0. All we have to do is compute D_1^{PS} and D_1^{FCFS} and show that

$$D_1^{FCFS} > D_1^{PS}.$$

We explain in words first: Consider first $\mathcal{N}_{C,FCFS}$. D_1^{FCFS} is the expected delay on p caused by one other packet, q, arriving during $(-n, n)$, where there are no packets other than p and q in the network at any time. Now suppose q is of type *solid* and p is of type *dashed* and p arrives within $n/2$ seconds after q (the probability of this event is a constant: $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{4}$). Then, as shown in Figure 5.2, p will eventually catch up to q, and from this point on, q will delay p by one second at every other server throughout the rest of $\mathcal{N}_{C,FCFS}$. That is, p will be delayed by $\Theta(n)$ seconds. Now observe that the same scenario would only cause a delay of at most 2 seconds in $\mathcal{N}_{C,PS}$, because when p catches up to q, it will only interfere with q for two servers and then p will pass q forever.

A worse situation for $\mathcal{N}_{C,PS}$ is the case where q is the same type as p. If q and p meet, p will be delayed by $\Theta(n)$. Observe, however, that this scenario can only happen if the two packets both arrived at $\mathcal{N}_{C,PS}$ within a second of each other, which occurs with probability $\Theta\left(\frac{1}{n}\right)$ because q's arrival is uniformly distributed within the n seconds of p's arrival. Thus the contribution of this case to the average delay ends up being negligible.

More formally,

$$\begin{aligned}
D_1^{FCFS} &= \mathbf{E}\{\text{ Delay on } p \text{ caused by 1 other packet, } q, \text{ arriving in } (-n, n)\ \} \\
&= \mathbf{Pr}\left\{\ q \text{ is same type as } p\ \right\} \cdot \mathbf{E}\left\{\text{Delay} \;\middle|\; q \text{ is same type as } p\ \right\} \\
&\quad + \mathbf{Pr}\left\{\ q \text{ is opposite type from } p\ \right\} \cdot \mathbf{E}\left\{\text{Delay} \;\middle|\; q \text{ is opp. type from } p\ \right\} \\
&= \frac{1}{2} \cdot \Theta\left(\frac{1}{n} \cdot 1\right) \\
&\quad + \frac{1}{2} \cdot \Theta\left(1 \cdot n\right) \\
&= \Theta\left(n\right)
\end{aligned}$$

[3] Observe that because of the ϵ-servers, the nondeterminism that we saw in Section 4.2, caused by two packets arriving at a server at exactly the same time, is never invoked.

The second to last line is due to the following argument: If p and q are the same type, then they will meet with probability $\Theta(\frac{1}{n})$ and then the delay will only be $\Theta(1)$, since they will pass each other immediately. If p and q are of the opposite type, they will meet with constant probability, and then (assuming q is solid) q delays p all the way down the network.

$$\begin{aligned}
D_1^{PS} &= \mathbf{E}\{\text{ Delay on } p \text{ caused by 1 other packet, } q, \text{ arriving in } (-n, n)\ \} \\
&= \mathbf{Pr}\left\{\ q \text{ is same type as } p\ \right\} \cdot \mathbf{E}\left\{\text{Delay} \;\middle|\; q \text{ is same type as } p\ \right\} \\
&\quad + \mathbf{Pr}\left\{\ q \text{ is opposite type from } p\ \right\} \cdot \mathbf{E}\left\{\text{Delay} \;\middle|\; q \text{ is opp. type from } p\ \right\} \\
&= \frac{1}{2} \cdot \Theta\left(\frac{1}{n} \cdot n\right) \\
&\quad + \frac{1}{2} \cdot \Theta\left(1 \cdot 1\right) \\
&= \Theta\left(1\right)
\end{aligned}$$

The second to last line is due to the following argument: If p and q are the same type, then they will meet with probability $\Theta(\frac{1}{n})$ and then they will delay each other all the way down the network, $\Theta(n)$. If p and q are of the opposite type, they will meet with constant probability, and then they will pass each other immediately, causing only a constant delay.

Chapter 6

Conjectures, Simulations, and Analysis Techniques

This chapter is a collection of ongoing work motivated by the previous chapters.

Our original goal was to show that many queueing networks are in $\mathcal{S}$. Having observed that at least one queueing network is not in $\mathcal{S}$, it's clear we must narrow our goal to proving that some specific class of networks is in $\mathcal{S}$. Sections 6.1 and 6.2 offer conjectures for which classes of networks are likely to be in $\mathcal{S}$, based on examination of the counterexample network from Chapter 5.

Throughout Part I we have used several techniques for proving networks belong to $\mathcal{S}$ (or $\mathcal{S}_{Light}$). In Section 6.3 we suggest some additional approaches for proving that networks belong to $\mathcal{S}$.

Our purpose in proving that a network $\mathcal{N}$ belongs to $\mathcal{S}$ is to obtain an upper bound on the average delay in $\mathcal{N}_{C,FCFS}$. In Section 6.4 we address via simulations the question of how good a bound AverageDelay($\mathcal{N}_{E,FCFS}$) is on AverageDelay($\mathcal{N}_{C,FCFS}$). We show that although AverageDelay($\mathcal{N}_{E,FCFS}$) can be much larger than AverageDelay($\mathcal{N}_{C,FCFS}$), there are some predictable characteristics of a network which allow us to gauge, for that particular network $\mathcal{N}$, how far off AverageDelay($\mathcal{N}_{E,FCFS}$) will be from AverageDelay($\mathcal{N}_{C,FCFS}$). Thus by knowing AverageDelay($\mathcal{N}_{E,FCFS}$) and taking the distance factor into account, we can obtain a better estimate for AverageDelay($\mathcal{N}_{C,FCFS}$). The work in this section is still in progress.

6.1 Networks where all service times are equal

Conjecture 1 *Given any queueing network*[1] $\mathcal{N}$ *(with any routing scheme) for which all servers have the same mean service time,* 1*, then* $\mathcal{N}$ *is in* $\mathcal{S}$*.*

Conjecture 1 is motivated by the counterexample network $\mathcal{N}$ (Figure 5.1), which is highly dependent on the fact the $\mathcal{N}$'s servers have different service times. The delay is high in $\mathcal{N}_{C,FCFS}$(Figure 5.1) because the ϵ-servers ensure that the packets reunite over and over again. Suppose all the servers in $\mathcal{N}_C$ had the same service time, 1, (e.g., imagine the network in Figure 5.1 with all the ϵ-servers removed). Once two packets meet, their timings become synchronized. From then on, if they ever meet again at some other server s, they will have arrived at s at the exact same time. Thus with probability $\frac{1}{2}$, either packet will go first, so it's hard to guarantee that the two packets will repeatedly meet. The ϵ-servers are necessary to force the packet of type *solid* to always serve before the packet of type *dashed*. Thus it seems likely that every network where all servers have the same mean speed is in $\mathcal{S}$. Observe that it is not possible to simulate the effect of two different server speeds by simply creating different length chains of servers with speed 1, because of the pipelining effect.

Conjecture 1 is interesting because it includes all of the packet-routing networks analyzed by the theoretical computer science community (see Section 2.6).

It might be easier to first prove Conjecture 1 in the case of light traffic only. To do this we would only have to prove the conjecture in the case where there are exactly two packets (distributed randomly) in the network (see Section 4.2).

6.2 Heavy traffic networks

The counterexample network of Chapter 5 required the outside arrival rate to be very low. This was necessary to ensure that with high probability there would never be more than two packets in the network at a time. Had the outside arrival rate been higher, there would have been a non-negligible probability of having several packets in the network at once. In particular, this would imply a non-negligible probability of their being several packets on the same route (of the same type) in $\mathcal{N}_{C,PS}$, and with some probability two of these several packets would have started within one second of each other. Given two same-type packets in $\mathcal{N}_{C,PS}$, they would create a delay of $\Theta(n)$, where n is the number of

[1] Recall part of the definition of a queueing network is that the outside arrival process is a Poisson Process.

servers. Furthermore, once two packets formed a clump (see Section 2.9), that clump would move at half the speed of normal packets, so with high probability the other packets of the same type would soon join the clump, creating a bigger, slower clump.

The above scenario leads us to conjecture that any network $\mathcal{N}$ with a reasonably high arrival rate (say more than half the maximum load) has the property that AverageDelay($\mathcal{N}_{C,FCFS}$) < AverageDelay($\mathcal{N}_{C,PS}$). The exact conjecture must be formulated more carefully, though, because it could be the case that the maximum load in a network is artificially low because of one particular tiny bottleneck.

6.3 Alternative Approaches

In this thesis, we have used several different approaches in proving that networks belong to $\mathcal{S}$. There are a few additional approaches which we came up with, but which we weren't able to effectively use. We list them here, in case others are able to make use of them.

6.3.1 Proportions Network

In this section we describe an idea for getting more leverage out of the knowledge that all networks with Markovian routing are in $\mathcal{S}$, towards proving that many networks with non-Markovian routing are also in $\mathcal{S}$.

Let $\mathcal{N}$ be a queueing network (as defined in Section 2.2, i.e. the packets are born with fixed paths which they follow). Let $p_{i,j}$ be the proportion of packets which when finished serving at server i next move to server j. This quantity is easy to compute given the rate at which each route occurs in the routing scheme. Now consider a queueing network $\mathcal{N}'$ with Markovian routing, with probability $p_{i,j}$ on the edge between servers i and j. We refer to $\mathcal{N}'$ as the *proportions network* corresponding to $\mathcal{N}$.

Claim 4

$$AvgDelay(\mathcal{N}'_{C,FCFS}) \leq AvgDelay(\mathcal{N}'_{E,FCFS}) = AvgDelay(\mathcal{N}_{E,FCFS}).$$

Proof: The inequality in Claim 4 follows from Theorem 1. The equality in Claim 4 may be seen as follows (the following proof assumes familiarity with the notation in Appendix A):

Recall from Appendix A that for a queueing network $\mathcal{N}$,

$$\begin{aligned}\hat{\lambda}_i^{(l)}(\mathcal{N}) &= r_i^{(l)} + \sum_j p_{ji}^{(l)} \hat{\lambda}_j^{(l)} \\ \hat{\lambda}_i(\mathcal{N}) &= \sum_l \hat{\lambda}_i^{(l)} = \sum_l r_i^{(l)} + \sum_l \sum_j p_{ji}^{(l)} \hat{\lambda}_j^{(l)}\end{aligned}$$

The corresponding proportions network $\mathcal{N}'$ is a single class network, so by Appendix A we have for $\mathcal{N}'$,

$$\hat{\lambda}_i(\mathcal{N}') = r_i + \sum_j p_{ji} \hat{\lambda}_j$$

and substituting p_{ji} as defined for the proportions network $\mathcal{N}'$,

$$p_{ji} = \frac{\text{rate of packets from } j \text{ to } i \text{ in } \mathcal{N}}{\text{rate of packets leaving } j \text{ in } \mathcal{N}} = \frac{\sum_l \hat{\lambda}_j^{(l)} \cdot p_{ji}^{(l)}}{\hat{\lambda}_j}$$

we have

$$\hat{\lambda}_i(\mathcal{N}') = r_i + \sum_j \sum_l \hat{\lambda}_j^{(l)} p_{ji}^{(l)} = \hat{\lambda}_i(\mathcal{N})$$

■

From Claim 4 above, we see that to prove that the average delay in $\mathcal{N}_{C,FCFS}$ is less than that in $\mathcal{N}_{E,FCFS}$, it is sufficient to prove that

$$\text{AvgDelay}(\mathcal{N}_{C,FCFS}) \leq \text{AvgDelay}(\mathcal{N}'_{C,FCFS}) \ . \tag{6.1}$$

Determining for which networks inequality 6.1 above is true is an open question. Observe that inequality 6.1 is an equality for any rooted tree network, as shown in Figure 2.5. We have simulated a dozen or so different networks and for every network $\text{AvgDelay}(\mathcal{N}_{C,FCFS})$ and $\text{AvgDelay}(\mathcal{N}'_{C,FCFS})$ were close (but not always equal) in value, however a proof for why this is true eludes us. An example of a network and its corresponding proportions network is shown in Figure 6.1.

6.3.2 Convexity Analysis

Convexity analysis seems like a natural approach when trying to prove that networks with constant-time servers incur less delay than the corresponding networks with

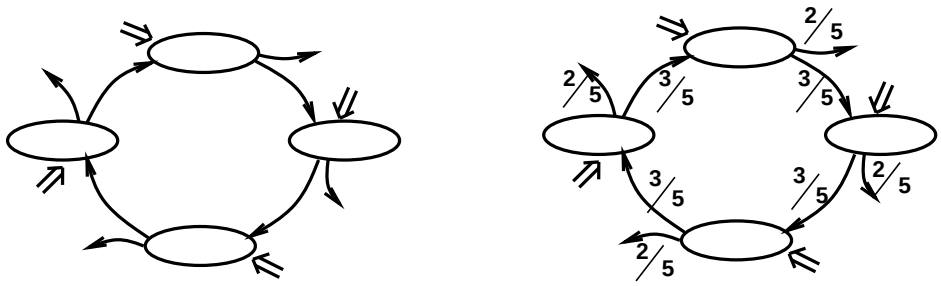


Figure 6.1: *On the left is the ring network $\mathcal{N}$ with clockwise routing. The routing scheme specifies that the outside arrival rate at each server is the same and the destinations are random (meaning the packet is equally likely to depart after serving at* 1, 2, 3, *or* 4 *servers). On the right is the corresponding proportions network $\mathcal{N}'$. Observe that in $\mathcal{N}'$ a packet may traverse the ring more than once, although this never happens in $\mathcal{N}$.*

exponential-time servers. The idea is to express the delay incurred up to time t as a convex function of the service times which occurred up to time t. By the properties of a convex function, it would then follow that the more variable the service times (exponential rather than constant), the greater the expected delay. Thus far we have only been able to make this approach work for networks with the property that, for every possible arrival sequence, AverageDelay($\mathcal{N}_{C,FCFS}$) is smaller than AverageDelay($\mathcal{N}_{E,FCFS}$). These are all trivial networks, such as those shown in Figure 2.4 and Figure 2.5.

6.4 How tight an upper bound do we obtain for networks in $\mathcal{S}$?

Our goal in this section is to study, for those networks $\mathcal{N}$ which we know to be in $\mathcal{S}$, how tight an upper bound AverageDelay($\mathcal{N}_{E,FCFS}$) provides on AverageDelay($\mathcal{N}_{C,FCFS}$). Obviously the tightness is affected by the characteristics of the particular network $\mathcal{N}$. Our goal is to figure out which characteristics matter, so that we can examine a particular network $\mathcal{N}$ and determine how far off AverageDelay($\mathcal{N}_{C,FCFS}$) will be from AverageDelay($\mathcal{N}_{E,FCFS}$). Ideally, in future research, we would like to be able to approximately deduce AverageDelay($\mathcal{N}_{C,FCFS}$) by assessing the characteristics of $\mathcal{N}$, evaluating some function $\mathcal{F}$ of these characteristics, and then multiplying AverageDelay($\mathcal{N}_{E,FCFS}$) by $\mathcal{F}$(characteristics of $\mathcal{N}$). That is,

$$\text{AverageDelay}(\mathcal{N}_{C,FCFS}) \approx \mathcal{F}(\text{characteristics of } \mathcal{N}) \cdot \text{AverageDelay}(\mathcal{N}_{E,FCFS})$$

The results in this section are highly preliminary because very few characteristics

are studied and very few networks are examined. We will use the notation $D_{\mathrm{E}}^{\mathcal{N}}$ to denote the average delay in network $\mathcal{N}$, given exponential servers. Likewise $D_{\mathrm{C}}^{\mathcal{N}}$ refers to the constant-server case. We will simply use D_{E} and D_{C} when referring to general networks. For simplicity, we assume all servers have mean service-time 1, for all networks we consider. We will use p to denote the load of the network, where load will be expressed as a fraction of the maximum possible (stationary) load for the network.

To get a sense of which network characteristics affect the difference between $D_{\mathrm{C}}^{\mathcal{N}}$ and $D_{\mathrm{E}}^{\mathcal{N}}$, we consider a few networks which we can easily analyze for the constant-server case. First consider the single-server network:

$$\begin{aligned} D_{\mathrm{E}}^{\text{single-server}} &= \frac{p}{1-p} \\ D_{\mathrm{C}}^{\text{single-server}} &= \frac{1}{2} \cdot \frac{p}{1-p} \end{aligned}$$

The difference $D_{\mathrm{E}} - D_{\mathrm{C}}$ clearly grows with p, but the ratio: $\frac{D_{\mathrm{E}}}{D_{\mathrm{C}}} = 2$ is constant with respect to p.

Now let's look at a chain network, as shown in Figure 2.4, consisting of n servers in succession. Packets arrive from outside at the first server, an traverse all n servers in order.

$$\begin{aligned} D_{\mathrm{E}}^{\text{chain}} &= n \cdot \frac{p}{1-p} \\ D_{\mathrm{C}}^{\text{chain}} &= \frac{1}{2} \cdot \frac{p}{1-p} \end{aligned}$$

Observe that $D_{\mathrm{C}}^{\text{chain}}$ is the same as $D_{\mathrm{C}}^{\text{single-server}}$, since an incoming packet is only delayed at the first server of the chain (thereafter, packets move in lockstep). $D_{\mathrm{E}}^{\text{chain}}$ is n times greater than $D_{\mathrm{E}}^{\text{single-server}}$. The difference, $D_{\mathrm{E}}^{\text{chain}} - D_{\mathrm{C}}^{\text{chain}}$, grows with both p and n. The ratio $\frac{D_{\mathrm{E}}^{\text{chain}}}{D_{\mathrm{C}}^{\text{chain}}} = 2n$ is constant with respect to p, however it increases as a function of n. For purposes of comparison with later networks, we computed the average delay for the chain network for various values of p and n. These are shown in Tables 6.1, 6.2, and 6.3.

The chain network brings up several questions for general networks:

1. Is it the number of servers that affects $D_{\mathrm{E}} - D_{\mathrm{C}}$ in the exponential-server network, or is it really the route lengths?

2. For a fixed load p, we saw $D_{\mathrm{C}}^{\text{chain}} = D_{\mathrm{C}}^{\text{single-server}}$. How does $D_{\mathrm{C}}^{\mathcal{N}}$ relate to $D_{\mathrm{C}}^{\text{single-server}}$ (both at the same fractional load p) for other networks $\mathcal{N}$?

3. We saw that the ratio $\frac{D_{\mathrm{E}}^{\text{chain}}}{D_{\mathrm{C}}^{\text{chain}}}$ was constant with respect to p and always equal to twice the route length. Is there a similar formula for $\frac{D_{\mathrm{E}}^{\mathcal{N}}}{D_{\mathrm{C}}^{\mathcal{N}}}$ for more general networks $\mathcal{N}$?

To answer these questions we measured delays in two more networks. We purposely chose networks which were "symmetric" or almost symmetric (symmetric except for endpoint servers) in the sense that all servers of the network appeared identical. Symmetric networks have the property that when the network is loaded to p fraction of maximum load, each server is also loaded to p fraction of maximum load, which facilitates comparing these networks with a single server loaded to p fraction of maximum capacity. First we studied a network we call the "Short-Paths network." It consists of a chain of n servers, numbered 1 through n, however, in this case we made all route lengths be short. The possible paths were $1 \longrightarrow 2 \longrightarrow$ (meaning the packet would serve at server 1, then serve at server 2, then leave), $2 \longrightarrow 3 \longrightarrow$, $3 \longrightarrow 4 \longrightarrow$, $4 \longrightarrow 5 \longrightarrow$, etc.. We simulated this network for $n = 16$, as shown in Tables 6.4, 6.5, 6.6.

In answer to our first question above, the difference between D_{C} and D_{E} does not seem to be affected by the number of servers in the network, but rather by the route lengths. Specifically, the fact that the route lengths were relatively short implied that $D_{\mathrm{E}}^{\text{short-paths}}$ was a good bound on $D_{\mathrm{C}}^{\text{short-paths}}$. In answer to our second question, $D_{\mathrm{C}}^{\text{short-paths}}$ for any fixed p was always greater than $D_{\mathrm{C}}^{\text{single-server}}$ for the corresponding p, in fact as much as 1.5 times as large. To see why this might be the case, observe that a newly arriving packet q into server, say 2, of the Short-Paths network is delayed not only by the load it finds at server 2, but also by half[2] of the arriving load into server 3. With respect to the third question, the ratio, $\frac{D_{\mathrm{E}}^{\text{short-paths}}}{D_{\mathrm{C}}^{\text{short-paths}}}$, seemed somewhat constant with respect to p, although it dropped off for higher loads (see Table 6.6). The ratio $\frac{D_{\mathrm{E}}^{\text{short-paths}}}{D_{\mathrm{C}}^{\text{short-paths}}}$ is less than twice the average route length (recall it was exactly twice for the chain network), in fact it is closer to 1.25 times the average path length.

The next network we measured was the clockwise ring network, with n servers. Packets are born (according to a Poisson Process) at every server of the ring. At birth, they are assigned a destination server at random, and they move clockwise along the ring

[2]Packet q experiences a total load of p at server 2, where half the arrivals at server 2 come from outside server 2 and the other half come from server 1. Once q leaves server 2, the packets that delayed it at server 2 will not delay it again because those packets now move in lockstep, however, q does have to contend with the outside arrivals into server 3 – half of server 3's load.

to that server. We simulated the ring network for various values of n and for various values of p, as shown in Tables 6.7, 6.8, and 6.9. Observe that for the ring network the mean route length is $\frac{n+1}{2}$. In answer to the second question, $D_{\mathrm{C}}^{\mathrm{ring}}$ for any fixed p was always greater than $D_{\mathrm{C}}^{\text{single-server}}$ for the corresponding p. Furthermore, the difference between $D_{\mathrm{C}}^{\mathrm{ring}}$ and $D_{\mathrm{C}}^{\text{single-server}}$ increases as n increases. Partly, $D_{\mathrm{C}}^{\mathrm{ring}}$ can be explained using the same type of explanation as for $D_{\mathrm{C}}^{\text{short-paths}}$ above, namely by summing up the extra load that an arriving packet into the ring sees once it's past the first server. However, even once this is added in, we see this still see that our estimate falls short of the measured $D_{\mathrm{C}}^{\mathrm{ring}}$, especially as n becomes large. Thus there is an additional level of complexity we are still missing. With respect to the third question, the ratio, $\frac{D_{\mathrm{E}}^{\mathrm{ring}}}{D_{\mathrm{C}}^{\mathrm{ring}}}$, seemed somewhat constant with respect to p, although it dropped off a little for higher loads in the case where n was large (see Table 6.9). The ratio $\frac{D_{\mathrm{E}}^{\mathrm{ring}}}{D_{\mathrm{C}}^{\mathrm{ring}}}$ is less than twice the average route length, in fact, it appears that as n gets larger, this ratio drops below 1 times the average path length.

In conclusion, a few very preliminary observations for general queueing networks are as follows:

1. The absolute difference, $D_{\mathrm{E}} - D_{\mathrm{C}}$, increases as the load increases and as the route lengths increase. This observation was confirmed for dozens of other networks which we simulated, but do not include here.

2. D_{C} appears greater than the delay experienced at a single-server network with constant service-time, loaded to the same fraction of maximum load.

3. The ratio, $\frac{D_{\mathrm{E}}}{D_{\mathrm{C}}}$, stays constant with respect to the load, or drops off slightly as load increases.

4. The ratio, $\frac{D_{\mathrm{E}}}{D_{\mathrm{C}}}$, increases as a function of the average route length (for fixed load).

We end by posing the following intriguing question, suggested by our simulations:

Is it the case that the ratio $\frac{D_{\mathrm{E}}}{D_{\mathrm{C}}}$ is greater for the chain network than for any other network with the same average path length?

$n\backslash p$	.25	0.5	0.75	0.8	0.85	0.9	0.95
2	.167	0.50	1.50	2.00	2.83	4.50	9.50
4	.167	0.50	1.50	2.00	2.83	4.50	9.50
8	.167	0.50	1.50	2.00	2.83	4.50	9.50
16	.167	0.50	1.50	2.00	2.83	4.50	9.50

Table 6.1: *Chain Network, Average delay for constant servers. (Computed). n is the number of servers. p is the fraction of maximum load.*

$n\backslash p$	.25	0.5	0.75	0.8	0.85	0.9	0.95
2	.67	2.0	6.0	8.0	11.33	18.0	38.0
4	1.33	4.0	12.0	16.0	22.66	36.0	76.0
8	2.66	8.0	24.0	32.0	45.33	72.0	152.0
16	5.32	16.0	48.0	64.0	90.66	144.0	304.0

Table 6.2: *Chain Network, Average delay for exponential servers. (Computed). n is the number of servers. p is the fraction of maximum load.*

$n\backslash p$	.25	0.5	0.75	0.8	0.85	0.9	0.95
2	4	4	4	4	4	4	4
4	8	8	8	8	8	8	8
8	16	16	16	16	16	16	16
16	32	32	32	32	32	32	32

Table 6.3: *Chain Network, Ratio: Averagedelay($\mathcal{N}_{E,FCFS}$)/Averagedelay($\mathcal{N}_{C,FCFS}$). (Computed). n is the number of servers. p is the fraction of maximum load.*

$n\backslash p$	.25	0.5	0.75	0.95
16	.24	0.76	2.36	15.0

Table 6.4: *Short Paths Network, Average delay for constant servers. (Measured). n is the number of servers. p is the fraction of maximum load.*

$n\backslash p$	.25	0.5	0.75	0.95
16	.64	1.91	5.68	35.5

Table 6.5: *Short Paths Network, Average delay for exponential servers. (Computed). n is the number of servers. p is the fraction of maximum load.*

$n \backslash p$	.25	0.5	0.75	0.95
16	2.67	2.51	2.41	2.37

Table 6.6: *Short Paths Network, Ratio: Averagedelay($\mathcal{N}_{E,FCFS}$)/Averagedelay($\mathcal{N}_{C,FCFS}$). n is the number of servers. p is the fraction of maximum load.*

$n \backslash p$	.25	0.5	0.75	0.8	0.85	0.9	0.95
2	0.23	0.68	2.01	2.67	3.77	5.96	12.45
4	0.28	0.88	2.73	3.65	5.19	8.26	17.30
8	0.32	1.06	3.45	4.70	6.75	10.92	23.50
16	0.35	1.18	4.07	5.57	8.21	13.60	30.20

Table 6.7: *Ring Network, Average delay for constant servers. (Measured). n is the number of servers. p is the fraction of maximum load.*

$n \backslash p$	.25	0.5	0.75	0.8	0.85	0.9	0.95
2	0.50	1.50	4.50	6.00	8.50	13.50	28.50
4	0.83	2.50	7.50	10.00	14.17	22.50	48.10
8	1.50	4.50	13.50	18.00	25.50	40.50	85.50
16	2.83	8.50	25.50	34.00	48.17	76.50	161.50

Table 6.8: *Ring Network, Average delay for exponential servers. (Computed). n is the number of servers. p is the fraction of maximum load.*

$n \backslash p$	.25	0.5	0.75	0.8	0.85	0.9	0.95
2	2.1739	2.2059	2.2388	2.2472	2.2546	2.2651	2.2892
4	2.9643	2.8409	2.7473	2.7397	2.7303	2.7240	2.7803
8	4.6875	4.2453	3.9130	3.8298	3.7778	3.7088	3.6383
16	8.0857	7.2034	6.2654	6.1041	5.8672	5.6250	5.3477

Table 6.9: *Ring Network, Ratio: Averagedelay($\mathcal{N}_{E,FCFS}$)/Averagedelay($\mathcal{N}_{C,FCFS}$). n is the number of servers. p is the fraction of maximum load.*

Chapter 7

Future Work

Our goal in Part I was to bound the average delay in general packet-routing networks. Traditional queueing-theoretic analysis requires several exponentiality assumptions. We showed how to "remove" (validate) one of these exponentiality assumptions: exponential service times, for a class of networks $\mathcal{S}$. This work is a major validation of queueing-theoretic analysis; it shows that we are justified in making the unrealistic queueing-theoretic assumption that the time required at a bottleneck is exponentially distributed because we can prove that under this assumption the computed delay will always be somewhat greater than the actual delay. In response to articles like, "Is Queueing Theory Dead?", [40], our answer is a resounding "No!"

Our work suggests a multitude of open problems. First, there's the question of extending $\mathcal{S}$. Figure 7.1 shows that we know $\mathcal{S}$ contains all queueing networks with Markovian routing. We also know there's at least one network not in $\mathcal{S}$. And, we know $\mathcal{S}_{Light}$ contains almost all queueing networks. One obvious open question is whether $\mathcal{S}$ equals $\mathcal{S}_{Light}$, as we have conjectured. Another open question is the conjecture raised in Section 6.2 which states that under sufficiently heavy traffic all networks are in $\mathcal{S}$. Another open question is the conjecture from Section 6.1 that all queueing networks where all servers have the same service time are in $\mathcal{S}$.

Other future work involves extending our model of a packet-routing network. For example, for most of our work we assumed that the service order of packets at bottlenecks was FCFS. It would be interesting to allow for any non-preemptive service order as we have in some of our proofs. Another extension of our model would be to allow for variable-size

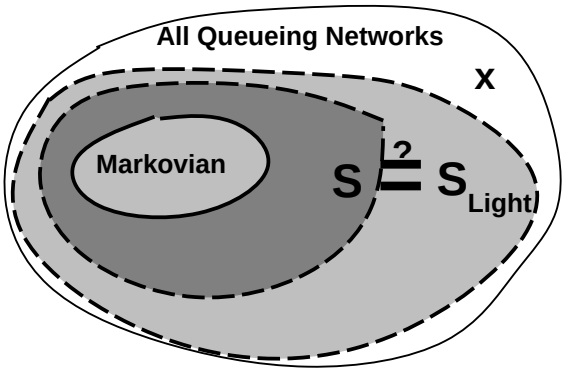


Figure 7.1: *Queueing networks in $\mathcal{S}$.*

packets. This brings up the interesting question of what are typical packet sizes. One recent paper, [19], provides evidence that packet sizes have a heavy-tailed Pareto distribution.

Next, there's the question of removing the other unrealistic exponentiality assumption traditionally made in queueing-theoretic analysis, namely the Poisson arrival process. An excellent study of Ethernet LAN traffic from Bellcore, [48], found the outside arrival traffic was far from Poisson, and exhibited self-similar fractal behavior. This was recently confirmed on Web traffic as well [19]. To obtain analytical bounds on network delays it is important to be able to model these bursty traffic arrivals via a process which is both accurate and analytically tractable.

Another unrealistic assumption which queueing theory often relies on is unbounded buffer sizes. Assuming bounded buffer sizes greatly complicates the analysis. However, it might be easier to instead try to relate delays in the case of bounded buffers (which we'd like to analyze) to the delays in the case of unbounded buffers (which we know how to analyze).

One last important open question is bounding other moments of the delay, such as the variance in the delay, rather than just mean delay.

Part II

Load Balancing of Processes in NOWs without Exponentiality Assumptions

Chapter 8

Introduction

In Part I of this thesis we investigated an area of network analysis, packet-routing network delay analysis, in which a particular unrealistic exponentiality assumption is often made, so as to make the analysis tractable. This exponentiality assumption involves assuming that the time required by a packet to pass through any network bottleneck (or "server") is exponentially distributed. We saw that although this assumption is unrealistic, it is "benign", in the sense that for a large class of networks we could prove that this assumption always affected the outcome of the analysis in a predictable way (specifically, it provided an upper bound on the actual average packet delay). Thus we saw that in the case of packet-routing analysis the unrealistic exponentiality assumption was actually an effective analytical tool.

In Part II of the thesis[1], we investigate a second area of network analysis, CPU load balancing on a network of workstations, for which we reach the opposite conclusion. CPU load balancing involves migrating processes across the network from hosts with high loads to hosts with low loads, so as to reduce the average completion time of processes. Here too unrealistic exponentiality assumptions are frequently made for analytical tractability, or simply for lack of better information. The most common exponentiality assumptions made are assuming that the CPU requirements of processes (process lifetimes) are exponentially distributed and assuming that packet interarrival times are exponentially distributed. Unfortunately, in the context of CPU load balancing, these exponentiality assumptions, particularly the lifetime one, are very dangerous. We show that assuming exponentially distributed process lifetimes, is not only inaccurate, but can lead to wrong conclusion with

[1] A large portion of Part II appeared in an earlier form in our paper, [31].

respect to developing load balancing algorithms and evaluating their effectiveness. We furthermore demonstrate that assuming any lifetime distribution other than the correct one can affect the analysis in unpredictable ways, making the result of the analysis or simulation untrustworthy.

Our investigation of the effect of exponentiality assumptions in load balancing is motivated by a fundamental question which has remained unresolved in the literature:

> Most systems and studies only migrate newborn processes (a.k.a. remote execution or non-preemptive migration). *Does it pay to also migrate active processes (preemptive migration), given the additional CPU cost of migrating their accumulated memory?*

A related question is

> *If we are going to migrate active processes, which active processes should we migrate?*

We will see that previous work has not satisfactorily answered either of these questions. We will see that the answer to these questions depends strongly on understanding the process lifetime distribution, not just its general shape, but its exact functional form. We will see that a great stumbling block in addressing these questions has been 1) a lack of knowledge of the functional form of the process lifetime distribution, 2) a lack of understanding of the importance of the process lifetime distribution, and 3) a lack of knowledge about how to use the functional form of the process lifetime distribution in analysis.

We will address all three of these issues. In particular, we will measure the UNIX process lifetime distribution and show how to explicitly apply the functional form of this distribution in analysis. That is, we will avoid resorting to exponentiality assumptions, which, in the area of load balancing analysis, are harmful. The process lifetime distribution will be used to derive a migration criterion (migration policy) which answers the question of which active processes are worth migrating. Since our migration policy is analytically derived, it will have no parameters which must be hand-tuned. We will then use a trace-driven simulation based on real process arrival times and CPU lifetimes to show that preemptive migration is far more powerful than non-preemptive migration, and that it definitely pays to migrate active processes.

This chapter covers preliminaries. Section 8.1 covers all the necessary load balancing terminology. Section 8.2 describes previous work on load balancing. Although we

show many systems are capable of migrating active processes, almost all the previous work is in the area of non-preemptive migration. Hardly any deals with preemptive migration policies, and the little that has been done is contradictory. Section 8.3 describes the specific questions we want to answer. Section 8.4 describes our model. In Section 8.5 we outline all of Part II. Lastly, in Section 8.6, we expand in more detail on previous work in the area of preemptive policies (Section 8.6 might be easier to read after completing Part II).

8.1 Load balancing taxonomy

On a network of shared processors, *CPU load balancing* is the idea of migrating processes across the network from hosts with high loads to hosts with lower loads. The motivation for load balancing is to reduce the average completion time of processes and improve the utilization of the processors. Analytic models and simulation studies have demonstrated the performance benefits of load balancing, and these results have been confirmed in existing distributed systems (see Section 8.2).

Process migration for purposes of load balancing comes in two forms: *remote execution* (also called *non-preemptive* migration), in which some newborn processes are (possibly automatically) executed on remote hosts, and *preemptive migration*, in which running processes may be suspended, moved to a remote host, and restarted. In non-preemptive migration only newborn processes are migrated, whereas in preemptive migration a process may be migrated at any point during its life.

Load balancing may be done explicitly (by the user) or implicitly (by the system). Implicit migration policies may or may not use *a priori* information about the function of processes, how long they will run, etc.

A load balancing strategy consists of two parts: the migration policy and the location policy. The *migration policy* determines both when migrations occur and the bigger issue of *which processes are migrated*. The location policy determines how a target host is selected for the migrated process. Previous work ([96] and [42]), has suggested that choosing the target host with the shortest CPU run queue is both simple and effective. Our work confirms the relative unimportance of location policy. In this thesis, we will focus almost exclusively on the more challenging problem of the migration policy, particularly the question of choosing which processes are worth migrating.

The issue in choosing a good migrant, in both the case of non-preemptive policies

and preemptive policies, is choosing a process which will live long enough (continue to use enough CPU) so as to compensate for its migration cost.

Even in the case of newborn processes (non-preemptive migration), the migration cost of most newborn processes is significant relative to their expected lifetime. Therefore, implicit non-preemptive policies require some *a priori* information about job lifetimes. This information is often implemented as an eligibility list (e.g. [73]) that specifies (by process name) which processes are worth migrating.

In contrast, preemptive migration policies do *not* use *a priori* information, since this is often difficult to maintain and preemptive strategies can perform well without it. These systems use only system-visible data like the current age of each process or its memory size in choosing worthy migrants.

8.2 Classification of Previous Work on Load Balancing Policies

Very little work has been done on studying preemptive migration policies, and the little work that has been done has been equivocal.

8.2.1 Systems

Although many systems are capable of migrating active jobs, most don't have implicit load balancing *policies* implemented. Most systems only allow for *explicit* load balancing. That is, there is no load balancing policy; the user decides which processes to migrate, and when. For example, Accent [93], Locus [76], Utopia [95], DEMOS/MP [59], V [75], NEST [1], and MIST [14] fall into this category.

A few systems do have *implicit* load balancing policies, however they are strictly non-preemptive policies (active processes are only migrated for purposes other than load balancing, such as preserving workstation autonomy). For example, Amoeba [74], Charlotte [6], Sprite [20], Condor [52], and Mach [54] fall into this category. In general, non-preemptive load balancing strategies depend on *a priori* information about processes; e.g., explicit knowledge about the runtimes of processes or user-provided lists of migratable processes ([1], [51], [20], [95]).

Only a few systems have implemented automated preemptive load balancing poli-

cies: MOSIX [7] and RHODOS [27]. Preemptive migration has been found to be very effective for load balancing in MOSIX, [7].

8.2.2 Studies

Migration policies have been studied much more in simulations and analyses. Most of these studies consider only non-preemptive policies, for example, [53], [87], [15], [96], [61], [42], [9], [25], [50], [55], [94], [97], [29], [22].

Only a few studies have considered preemptive migration policies, in addition to non-preemptive ones: specifically, [23], [41], [47], and [12]. However these studies disagree on the effectiveness of preemptive migration. Eager, Lazowska, and Zahorjan, [23] claim to measure an upper bound on the effectiveness of preemptive migration and conclude that preemptive migration is not worth implementing because the maximum additional performance benefit of preemptive migration is small compared with the benefit of non-preemptive migration schemes. This result has been widely cited, and in several cases used to justify the decision not to implement migration or not to use migration for load balancing. For example, the designers of the Utopia system, [95], explain, "Our second design decision is to support remote execution only at task initiation time; no checkpointing or task migration is supported. ... For improving performance, initial task transfer may be sufficient; a modeling study by Eager, Lazowska and Zahorjan suggests that dynamic task migration does not yield much further performance benefit except in some extreme cases." Krueger and Livny, [41], however reach the opposite conclusion.

8.3 The questions we want to answer

There are two fundamental questions which are presently unresolved and which we will answer in this thesis:

- Is preemptive migration necessary for load balancing, or is it sufficient to only implement a non-preemptive policy?

  This question is presently unresolved due to conflicting results from previous studies, as explained in Section 8.2.2.

- What is a good preemptive migration policy?

As explained in Section 8.1, preemptive policies do not rely on a priori information about processes, but rather use only system-visible data (such as the current age of each process, the process' memory size, etc). The issue in specifying a migration policy is answering the question: What is the "right" age and the "right" memory size, etc. for a process to be eligible for migration? As we will see in Section 10.2, no previous study of preemptive migration policies has addressed this question satisfactorily.

We will see that both these questions depend heavily on understanding and *applying* the process CPU lifetime distribution.

8.4 Process Model

In our model, processes use two resources: CPU and memory (we do not consider I/O). Thus, we use "age" to mean CPU age (the CPU time a process has used thus far) and "lifetime" to mean CPU lifetime (the total CPU time from start to completion). The processors in our network are assumed to be time-sharing with round-robin scheduling (in Section 12.3 we consider the effect of alternative local scheduling policies). Since processes may be delayed while on the run queue or while migrating, the slowdown imposed on a process is

$$\text{Slowdown of process p} = \frac{\text{wall time (p)}}{\text{CPU time (p)}}$$

where *wall-time*(p) is the total time p spends running, waiting in queue, or migrating.

8.5 Outline of results

The effectiveness of load balancing — either by non-preemptive or preemptive migration — depends strongly on the nature of the workload, including the distribution of process lifetimes and the arrival process.

In Chapter 9 we see that the process lifetime distribution is commonly assumed to be exponentially distributed, and the arrival process is commonly assumed to be Poisson (Section 9.1). This assumption is made not only in the case of analytical studies, but also in simulation papers (Section 9.1.2).

We then measure the distribution of process lifetimes for a variety of workloads in an academic environment, including instructional machines, research machines, and ma-

chines used for system administration. We find that the distribution is predictable (with goodness of fit $> 99\%$) and consistent across a variety of machines and workloads. As a rule of thumb, the probability that a process with CPU age of one second uses more than T seconds of total CPU time is $1/T$ (Figure 9.1). We show that our measured lifetime distribution is very far from exponential (Section 9.4), and that our measured arrival process is far from Poisson (Section 11.1).

Our lifetime measurements are consistent with the results of [47], but this prior work has been incorporated into almost no subsequent analytic and simulator load balancing studies. This omission is unfortunate, since, as we show in Section 9.5, the results of these load balancing studies are highly sensitive to the lifetime distribution model.

In Chapter 10 we show that the lifetime distribution function ($1/T$) which we measured is surprisingly analytically tractable for the purposes of deriving a preemptive migration policy. At a high level, the distribution simply suggests that it is preferable to migrate older processes because these processes have a higher probability of living long enough (eventually using enough CPU) to amortize their migration cost. However, we can also use the functional model of the lifetime distribution explicitly as an analytical tool for deriving the eligibility of a process for migration as a function of its current age, migration cost, and the loads at its source and target host. The eligibility criterion we derive doesn't rely on free parameters that must be hand-optimized, as do previous preemptive migration policies (Section 10.2). This tool is generally useful for analysis of system behavior.

Our migration eligibility criterion *guarantees* that the slowdown imposed on a migrant process is lower (in expectation) than it would be without migration. According to this criterion, a process is eligible for migration only if its

$$\text{CPU age} > \frac{1}{n-m} \cdot \text{migration cost}$$

where n (respectively m) is the number of processes at the source (target) host (Section 10.3).

In Chapter 11 we use a trace-driven simulation to compare our preemptive migration policy (from Section 10.3) with a highly-optimized non-preemptive policy based on name-lists. The simulator (Section 11.1) uses start times and durations from traces of a real system, and migration costs chosen from a measured distribution.

We use the simulator to run three experiments: First (Section 11.2) we evaluate the effect of migration cost on the relative performance of the two strategies. Not surpris-

ingly, we find that as the cost of preemptive migration increases, it becomes less effective. Nevertheless, preemptive migration performs better than non-preemptive migration even with surprisingly large migration costs (despite several conservative assumptions that give non-preemptive migration an unfair advantage).

Next (Section 11.3) we choose a specific model of preemptive and non-preemptive migration costs (described in Section 11.2.1), and use this model to compare the two migration strategies in more detail. We find that preemptive migration reduces the mean delay (queueing and migration) by $35 - 50\%$, compared to non-preemptive migration. We also propose several alternative metrics intended to measure users' perception of system performance. By these metrics, the additional benefits of preemptive migration (compared to non-preemptive migration) appear even more significant.

In Section 11.4 we discuss in detail why simple preemptive migration is so much more effective than even a well-tuned non-preemptive migration policy. The reasons are closely tied to the lifetime distribution. Essentially, a process' age is the best predictor of how much CPU it will use in the future (and therefore whether it is worth migrating). No amount of prior information (in the form of namelists) given to non-preemptive migration policies can compensate for the fact that non-preemptive policies don't get to see the processes age.

Lastly, in Section 11.5 we use the simulator to compare our preemptive migration strategy with other preemptive schemes in the literature.

We finish with a summary in Section 12.1, a self-criticism of our model in Section 12.2, and a discussion of future work in Section 12.3.

8.6 Previous studies on preemptive policies

Recall from Section 8.2, there are only a few studies and one system which deal with preemptive migration policies. In Section 10.2 (the migration policy chapter) we will see that none of these migration policies are derived from the lifetime distribution, and that all of them require hand-tuned parameters. In Section 11.5, we evaluate these policies against our own. In this section we point out a few additional differences between these studies and our own.

Our work differs from Eager, Lazowska, and Zahorjan, [23], in both system model and workload description. [23] model a server farm in which incoming jobs have no affinity

for a particular processor, and thus the cost of initial placement (remote execution) is free. This is different from our model, a network of workstations, in which incoming jobs arrive at a particular host and the cost of moving them away, even by remote execution, is significant. In the distribution we observed, 70% of the processes had lifetimes shorter than the lowest cost of remote execution reported in a real system (see Section 9.3.2 and Table 11.1). In this environment, non-preemptive migration is much less effective for load balancing than in a server farm.

Also, [23] use a degenerate hyperexponential distribution of lifetimes that includes few jobs with non-zero lifetimes. When the coefficient of variation of this distribution matches the distributions we observed, fewer than 4% of the simulated processes have non-zero lifetimes. With so few jobs (and balanced initial placement) there is seldom any load imbalance in the system, and thus little benefit for preemptive migration. For a more detailed explanation of this distribution and its effect on the study, see [21].

Krueger and Livny, [41], investigate the benefits of adding preemptive migration to nonpreemptive migration. They use a hyperexponential lifetime distribution that approximates closely the distribution we observed; as a result, their findings are largely in accord with ours. One difference between their work and ours is that they used a synthetic workload with Poisson arrivals. The workload we observed, and used in our trace-driven simulations, exhibits serial correlation; i.e. it is more bursty than a Poisson process. Another difference is that their migration policy requires several hand-tuned parameters. In Section 10.3 we show how to use the distribution of lifetimes to eliminate these parameters.

Like us, Bryant and Finkel, [12], discuss the distribution of process lifetimes and its effect on preemptive migration policy, but their hypothetical distributions are not based on system measurements. Also like us, they choose migrant processes on the basis of expected slowdown on the source and target hosts, but their estimation of those slowdowns is very different from ours. In particular, they use the distribution of process lifetimes to predict a host's future load as a function of its current load and the ages of the processes running there. We have examined this issue and found (1) that this model fails to predict future loads because it ignores future arrivals, and (2) that current load is the best predictor of future load. Thus, in our estimates of slowdown, we will assume that the future load on a host is equal to the current load.

Chapter 9

The UNIX Process Lifetime Distribution is not Exponential: it's 1/T

This chapter will show that understanding the correct process CPU lifetime distribution is crucially important in designing and evaluating load balancing policies. We will measure that distribution in this chapter, and then we will use it explicitly in the next chapter in deriving a preemptive migration policy.

In Section 9.1 we point out how common it is to model process lifetimes as being exponentially distributed and process interarrival times as exponentially distributed. In Section 9.2 we see that studies of the process lifetime distribution are rare, and are often ignored. In Section 9.3, we measure the UNIX process CPU lifetime distribution and find that it consistently has the same functional form across a variety of instructional, research, and administrative machines. As a rule of thumb,

$$\mathbf{Pr}\{\text{Process lifetime} > T \text{ sec} \mid \text{age} > 1 \text{ sec}\} = 1/T$$

$$\mathbf{Pr}\{\text{Process lifetime} > b \text{ sec} \mid \text{age} = a > 1 \text{ sec}\} = \frac{a}{b}$$

In Section 9.4 we show that the $1/T$ distribution is very far from exponential. Lastly, in Section 9.5 we explain why assuming an exponential lifetime distribution, or any lifetime distribution other than the correct one, in the context of load balancing analysis, can completely invalidate the result of the analysis.

9.1 Exponential Lifetime Assumptions are commonly made

Exponential assumptions are prevalent in the load balancing literature, both in analytic papers (Section 9.1.1) and in simulation-based papers which do no analysis (Section 9.1.2). Typical exponential assumptions made are: 1) assuming that process CPU lifetimes are chosen from an exponential distribution and 2) assuming interarrival times of processes are exponentially distributed (i.e., the arrival process is Poisson).

9.1.1 Exponential assumptions in analysis papers

Examples of some analytical papers on load balancing which make exponential assumptions include [50], [55], and [22]. All of these require exponential assumptions for setting up a continuous-time Markov chain (CTMC) model of the system (see Section 1.3 which explains the general idea). In all cases, only non-preemptive migration is considered. Even so, the state-space of the CTMC model is still very large, so further simplifying assumptions are made to reduce the state-space.

9.1.2 Exponential assumptions in simulation papers

Exponential assumptions are also made in strictly simulation papers, e.g., [42], [61], [87], [25], [53], [94], [17], and many others. Most of these papers do not justify their use of exponentiality assumptions. One of these, [42], acknowledge that an exponential workload many not be typical, but state that they believe and exponential workload is accurate enough for the purpose of evaluating different load descriptors[1]. Another of these, [61] state that the memoryless assumptions were necessary in simulation to reduce storage requirements.

9.2 Previous Measurements of Lifetime Distribution

The general shape of the distribution of process lifetimes in an academic environment has been known for a long time [66]: there are many short jobs and a few long jobs, and the variance of the distribution is greater than that of an exponential distribution.

[1] A load descriptor is a metric for measuring load at a host. By far the most typical load descriptor is the length of the CPU run queue. Other load descriptors include the sum of ages of processes at the host, etc.

However, to the best of our knowledge no explicit measurements were made of the process lifetime distribution prior to 1986.

In 1986 Leland and Ott, [47], proposed a functional form for the process lifetime distribution, based on measurements of the lifetimes of 9.5 million UNIX processes between 1984 and 1985. Leland and Ott concluded that process lifetimes have a UBNE (used-better-than-new-in-expectation) type of distribution. That is, the greater the current CPU age of a process, the greater its expected remaining CPU lifetime.[2] Specifically, they found that for $T > 3$ seconds, the probability of a process' lifetime exceeding T seconds is rT^k, where $-1.25 < k < -1.05$ (r normalizes the distribution).

In contrast to [47], Rommel,[65], claimed that his measurements show that "long processes have exponential service times."

Despite the [47] study, many studies have continued to assume an exponential process lifetime distribution in their evaluation of migration strategies (e.g., [61], [42], [55], [17], [2], [25], [65], [50]). Many of these studies cite the [47] paper.

9.3 Our Measurements of Lifetime Distribution

Because of the importance of the process lifetime distribution to load balancing policies, and the confusion over its form, we performed an independent study of this distribution. We measured the lifetimes of over one million UNIX processes, generated from a variety of academic workloads, including instructional machines, research machines, and machines used for system administration. We obtained our data using the UNIX command *lastcomm*, which outputs the CPU time used by each completed process.

We observed that long processes (with lifetimes greater than 1 second) have a predictable and consistent distribution. Section 9.3.1 describes this distribution. Section 9.3.2 makes some additional observations about shorter processes.

9.3.1 Lifetime distribution when lifetime $> 1s$.

Figure 9.1 shows our process lifetime measurements on a heavily-used instructional machine in mid-semester. The plot shows only processes whose lifetimes exceed one second. The y-axis indicates the fraction of processes with lifetimes greater than T seconds. The

[2] In contrast, the exponential distribution is memoryless; the expected remaining lifetime of a process is independent of age.

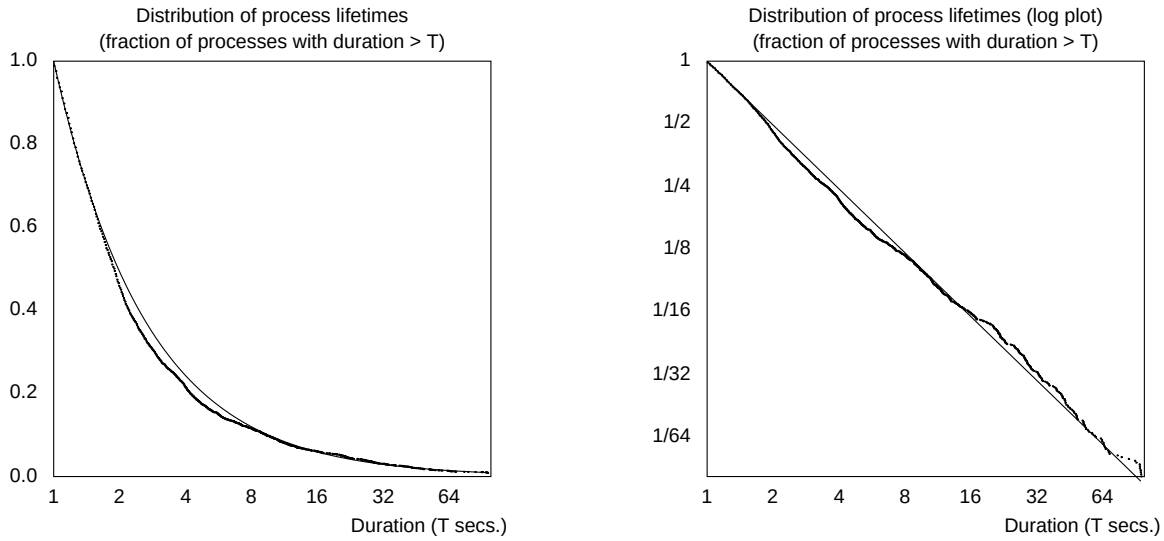


Figure 9.1: *Distribution of process lifetimes for processes with lifetimes greater than* 1 *second, observed on machine "po" mid-semester. The dotted (thicker) line shows the measured distribution; the solid (thinner) line shows the least squares curve fit. To the right: the same distribution shown on a log-log scale. The straight line in log-log space indicates that the process lifetime distribution can be modeled by* T^k*, where* k *is the slope of the line.*

dotted (heavy) line indicates the measured distribution; the solid (thinner) line indicates the least squares curve fit to the data using the proposed functional for $Pr\{Lifetime > T\} = T^k$.

For all the machines we studied, the process lifetime data (for processes with age greater than one second) fit a curve of the form T^k, where k ranged from about -1.3 to $-.8$ for different machines. Table 9.1 shows the estimated lifetime distribution curve for each machine we studied. The parameters were estimated by an iteratively weighted least squares fit (with no intercept, in accordance with the functional model)[3]. The standard error associated with each estimated parameter gives a confidence interval for that parameter (all of these parameters are statistically significant at a high degree of certainty). Finally, the R^2 value indicates the goodness of fit of the model — the values shown here indicate that the fitted curve accounts for greater than 99% of the variation of the observed values.[4] Thus, the goodness of fit of these models is very high. Table 9.1 demonstrates that the lifetime distribution measured by Leland and Ott still holds 10 years later, and on a variety

[3]See BLSS routine "robust", as described in [57].

[4]See BLSS routine "robust", as described in [57]. See also [43] for a more detailed description of the R^2 value.

Name of Host	Total Number Processes Studied	Number Processes with Age > 1	Estimated Lifetime Distribution Curve	Standard Error	R^2 value
po1	77440	4107	$T^{-0.97}$	.016	0.997
po2	154368	11468	$T^{-1.22}$	.012	0.999
po3	111997	7524	$T^{-1.27}$	.021	0.997
cory	182523	14253	$T^{-0.88}$	.030	0.982
pors	141950	10402	$T^{-0.94}$	.015	0.997
bugs	83600	4940	$T^{-0.82}$	.007	0.999
faith	76507	3328	$T^{-0.78}$	.045	0.964

Table 9.1: *The estimated lifetime distribution curve for each machine measured, and the associated goodness of fit statistics. Description of machines: Po is a heavily-used DEC-server5000/240, used primarily for undergraduate coursework. Po1, po2, and po3 refer to measurements made on po mid-semester, late-semester, and end-semester. Cory is a heavily-used machine, used for coursework and research. Porsche is a less frequently-used machine, used primarily for research on scientific computing. Bugs is a heavily-used machine, used primarily for multimedia research. Faith is an infrequently-used machine, used both for video applications and system administration.*

of machines.

Although the range of parameters (Table 9.1) we observed is fairly broad, in the absence of measurements from a specific system, assuming a distribution of $1/T$ is substantially more accurate than assuming that process lifetimes are exponentially distributed, as we show in Section 9.4.

Table 9.2 shows the lifetime distribution function, the corresponding density function, and the conditional distribution function. We will refer to the conditional lifetime distribution often during our analysis of migration strategies. The second column of Table 9.2 shows these functions when $k = -1$, which we will assume for our analysis in Chapter 10. This second column represents some useful *rules of thumb* in estimating UNIX process behavior in general. For example, the last row of Table 9.2 indicates that the *median* remaining lifetime of a job is its current age[5].

The functional form we are proposing (the fitted distribution) has the property that its moments (mean, variance, etc.) are infinite. Of course, since our observed distributions have finite sample size, they have finite mean (~ 0.4 seconds) and coefficient of variation ($5 - 7$). But moments are not robust summary statistics for distributions with a long tail — the behavior of the moments tends to be dominated by a few outlier points in the tail of the distribution. But we have little information about the behavior of this tail in real distributions, since there are few observations of processes longer than a few thousand seconds. Since we cannot characterize the distribution of the longest jobs, it is awkward to rely on summary statistics like mean and variance. Thus we use more robust summary statistics (median values or the estimated parameter k) to summarize distributions, rather than moments.

9.3.2 Process lifetime distribution in general

For completeness we discuss the lifetime distribution for processes with lifetimes less than one second. Since our measurements were made using the *lastcomm* command the shortest process we were able to measure was .01 seconds. For processes between .01 and 1 second, we did not find a consistent functional form, however for all machines we studied these processes had an even lower hazard rate than those of age > 1 second. That is, while the probability that a process of age $T > 1$ second lives another T seconds is approximately

[5]Observe this is somewhat different from the "past-repeats heuristic," which says the mean remaining lifetime of a job is it current age.

Process Lifetime Distribution for Processes of Age ≥ 1 second	When $k = -1$
$\mathbf{Pr}\{\text{Process lifetime} > T \text{ sec} \mid \text{age} > 1 \text{ sec}\} = T^k$	$= 1/T$
$\mathbf{Pr}\{\text{Process lifetime} = T \text{ sec} \mid \text{age} = 1 \text{ sec}\} = -kT^{k-1}$	$= 1/T^2$
$\mathbf{Pr}\{\text{Process lifetime} > b \text{ sec} \mid \text{age} = a > 1 \text{ sec}\} = \left(\frac{b}{a}\right)^k$	$= \frac{a}{b}$
$\mathbf{Pr}\{\text{Process lifetime} > 2T \text{ sec} \mid \text{age} = T > 1 \text{ sec}\} = (2)^k$	$= \frac{1}{2}$

Table 9.2: *The cumulative distribution function, probability density function, and conditional distribution function of processes lifetimes. The second column shows the functional form of each for the typical value* $k = -1.0$.

1/2, the probability that a process of age $T < 1$ second lives another T seconds is something greater than 1/2.

One other property of the lifetime distribution which we will need to refer to in later sections is that over 70% of all (newborn) jobs we measured have lifetimes smaller than .1 seconds.

9.4 The Exponential Distribution Does Not Fit the Measured Lifetime Data

In the previous section we showed that the T^k function is an excellent fit to describe the fraction of processes which use at least T seconds of CPU (starting with only those of CPU age at least 1 second). Figure 9.2 shows that the exponential distribution (even with two free parameters) is an extremely poor fit to the measured data.

The properties of an exponential distribution are very different from those of the distributions we observed. For example, the distributions we observed all have a tail of long-lived jobs (i.e., the distributions have high variance). An exponential distribution with the same mean would have lower variance; it lacks the tail of long-lived jobs.

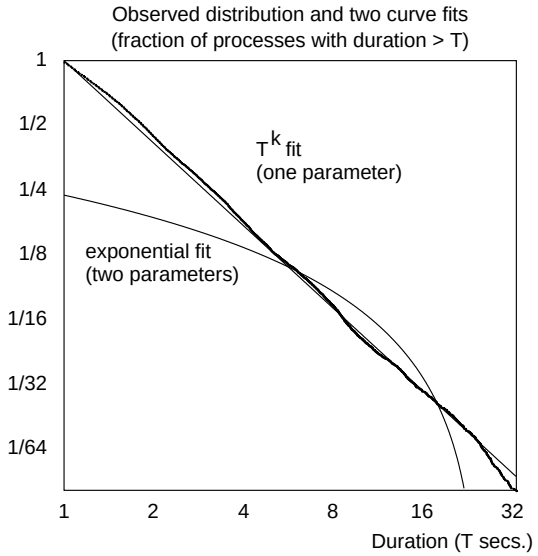


Figure 9.2: *In log-log space, this plot shows the distribution of lifetimes for the* $\sim$ 13000 *processes with lifetimes* > 1 *second (which we fed into our trace-driven simulation – see Chapter11), and two attempts to fit a curve to this data. One of the fits is based on the model proposed in this paper,* T^k*. The other fit is an exponential curve,* $c \cdot e^{-\lambda T}$*. Although the exponential curve is given the benefit of an extra free parameter, it fails to model the observed data. The proposed model fits well. Both fits were performed by iteratively-weighted least squares.*

9.5 Why the distribution is critical

In this section we argue that the particular shape of the lifetime distribution affects the performance of migration policies, and therefore that it is important to model this distribution accurately.

The choice of a migration policy depends on how the expected remaining lifetime of a job varies with age. In our observations we found a distribution with the UBNE property — the expected remaining lifetime of a job increases linearly with age. As a result, we chose a migration policy that migrates only old jobs.

But different distributions result in different relationships between the age of a process and its remaining lifetime. For example, a uniform distribution has the NBUE property — the expected remaining lifetime *decreases* linearly with age. Thus if the distribution of lifetimes were uniform, the migration policy should choose to migrate only young processes. In this case, we expect non-preemptive migration to perform better than preemptive migration.

As another example, the exponential distribution is memoryless — the remaining lifetime of a job is independent of its age. In this case, since all processes have the same expected lifetimes, the migration policy might choose to migrate the process with the lowest migration cost, regardless of age.

As a final example, processes whose lifetimes are chosen from a uniform log distribution[6] have an expected remaining lifetime that increases up to a point and then begins to decrease. In this case, the best migration policy might be to migrate jobs that are old enough, but not too old.

Thus different distributions, even with the same mean and variance, can lead to very different migration policies. In order to evaluate a proposed policy, it is critical to choose a distribution model with the appropriate relationship between expected remaining lifetime and age.

Some studies have used hyperexponential distributions to model the distribution of lifetimes. These distributions may or may not have the right behavior, depending on how accurately they fit observed distributions. [41] use a three-stage hyperexponential with parameters estimated to fit observed values. This distribution has the appropriate UBNE

[6]The logarithm of values from a uniform log distribution are uniformly distributed. We have observed lifetime distributions of this sort for parallel batch jobs.

property. But the two-stage hyperexponential distribution in [23] is memoryless for jobs with non-zero lifetimes; the remaining lifetime of a job is independent of its age (for jobs with nonzero lifetimes). According to this distribution, migration policy is irrelevant; all processes are equally good candidates for migration. This result is clearly in conflict with our observations.

Assuming the wrong lifetime distribution also impacts the evaluation of purely non-preemptive migration schemes. Recall the purpose of a non-preemptive policy is to choose which newborn processes to migrate. Many studies of non-preemptive migration simply assume that newborn jobs will have lifetimes longer than their migration costs. For example, [22], in their 1986 study, assume that the average cost of migrating a task is between 1% and 10% of the average lifetime of a newborn. Similarly, [50], in their 1993 study, assume that the average job migration time is between 1% and 20% of the average lifetime of a newborn. These assumptions may have been true in 1986, but they are far from accurate now. We will see that more than 70% of all newborn processes have shorter lifetimes than their migration cost (see Section 9.3.2 and Table 11.1). (This is why systems use name-lists to indicate which names tend to be long-lived.) Studies which assume (in the context of determining or evaluating non-preemptive policies) that most newborn processes live long enough to make migrating them worthwhile may end up with results which are too optimistic.

Lastly, we have thus far only considered the effect on load balancing studies of assuming an exponential lifetime distribution (or other incorrect lifetime distributions). However exponentiality assumptions in the arrival process (i.e., assuming a Poisson arrival process), which almost always goes hand-in-hand with assuming an exponential lifetime distribution, can also lead to inaccurate results. If arrivals are evenly spaced out, load balancing is less necessary. If arrivals are bursty, migration becomes more necessary (to alleviate the temporarily high load at one host). Poisson arrivals are evenly spaced out in sense that there is a low probability of having many arrivals at once. In Section 11.1, we see that our measured arrival process is burstier than Poisson. Therefore, assuming a Poisson arrival process may underestimate the usefulness of load balancing.

Chapter 10

Analysis without Exponentiality Assumptions: Applying the 1/T Distribution to Developing a Migration Policy

In the previous chapter we saw that the process lifetime distribution for UNIX processes is not exponential, as is commonly assumed, but rather has a $1/T$ distribution. In this chapter we'll see that although correct lifetime distribution was known, it was never explicitly applied towards load balancing policies. We conjecture this is due to a belief that the $1/T$ distribution is not analytically tractable (as opposed to exponential-based distributions). In this chapter we show this belief is false ; the $1/T$ distribution is in fact directly useful for analysis.

Section 10.1 discusses high-level recommendations with respect to developing a preemptive migration policy that can be gleaned from studying the lifetime distribution. Section 10.2 surveys preemptive migration policies in the literature, none of which explicitly use the lifetime distribution. The main weakness of these policies is that they rely on parameters which must be set optimally by hand for the particular system. In Section 10.3 we use the functional form of our measured lifetime distribution ($1/T$) explicitly as an analytical tool to derive our preemptive migration policy. Our policy specifies which processes are eligible for migration as a function of their age, migration cost, and the loads at the

source and target host. Our criterion also provably guarantees that the slowdown imposed on a migrant process is lower (in expectation) than it would be without migration.

10.1 Properties of the Lifetime Distribution and their Implications on Migration Policies

A migration policy is based on two decisions: when to migrate processes and which processes to migrate. The focus of Part II is the second question (The first question concerns the frequency with which preemptive migrations are allowed. We address the first question briefly in Section 11.1.1):

> Given that the load at a host is too high, how do we choose *which* process to migrate?

Our heuristic is to *choose the process that has highest probability of running longer than its migration time.*

The motivation for this heuristic is twofold. From the host's perspective, a large fraction of the migration time is spent at the host (packaging the process). The host would only choose to migrate processes that are likely to be more expensive to run than to migrate. From the process' perspective, migration time has a large impact on response time. A process would choose to migrate only if the migration overhead could be amortized over a longer lifetime.

Most existing migration policies are non-preemptive. That is, they only migrate newborn processes, because these processes have no allocated memory and thus their migration cost is less (see Section 11.2.1).[1] The problem with this policy is that, according to the process lifetime distribution (Section 9.3), these newborn processes are unlikely to live long enough to justify the cost of remote execution.

Thus a "newborn" migration policy is only justified if the system has prior knowledge about the processes and can selectively migrate only those processes likely to be CPU hogs. However, the ability of the system to predict process lifetimes by name is limited, as shown in Section 11.4.

[1] The idea of migrating newborn processes might also stem from the fallacy that process lifetimes have an exponential distribution, implying that all processes have equal expected remaining lifetimes regardless of their age.

Can we do better? Preemptive migration policies have knowledge of a process' age, and the lifetime distribution indicates that migrating *older* processes is wise, since they have the highest probability of living long enough to justify the cost of migration.

However there are a few unanswered concerns: First of all, since the vast majority of processes are short, there might not be enough long-lived processes to have a significant load balancing effect.

In fact, although there are few old processes, they account for a large part of the total CPU load. According to our process lifetime measurements (Section 9.3), typically fewer than 3.5% of processes live longer than 2 seconds, yet these processes make up more than 60% of the total CPU load. This is due to the long tail of the process lifetime distribution (see Figure 9.2). Furthermore, we will see that the ability to migrate even a few large jobs can have a large effect on system performance, since a single large job on a busy host imposes slowdowns on many small processes.

The remaining concerns are: How old is old enough? Also, isn't it possible that the additional cost of migrating old processes (the memory transfer cost) might overwhelm the benefit of migrating longer-lived processes?

Section 10.2 shows the solutions other preemptive policies have suggested to these questions. Section 10.3 shows the derivation of our preemptive policy.

10.2 Preemptive Policies in the Literature

As we mentioned in Chapter 8, there are almost no preemptive policies in the literature. In this section we describe the few that do exist. All these are based on the principle that a process should be migrated if it's "old enough", e.g. if its age exceeds .5 seconds. We refer to the .5 here as a "voodoo constant[2]," because it is a parameter which magically appears in the paper, was clearly optimized to that particular system, and must be re-tuned for any other system. There is no logic for why .5 should work better than 1 or .25.

Leland and Ott, [47], say that a process p is eligible for migration if:

$$\text{age}(p) \; > \; \text{ages of } k \text{ younger jobs at host}$$

[2]This terminology was first coined by Professor John Ousterhout at U.C. Berkeley.

The parameter k is a voodoo constant (which the paper refers to as the "MINCRIT" parameter).

Krueger and Livny, [41] go a step further by taking the job's migration cost into account. A process p is eligible for migration if:

$$\text{age}(p) \ > .1 * \text{migration cost}(p)$$

The .1 here is a voodoo constant.

The MOSIX system, [7] come up with a similar criterion to Krueger and Livny. A process p is eligible for migration if:

$$\text{age}(p) \ > 1 * \text{migration cost}(p)$$

The MOSIX policy has some interesting properties which we will discuss at the end of the next section.

These policies will be compared with our own in Section 11.5.

10.3 Our Migration Policy

The obvious disadvantage of preemptive migration is the need to transfer the memory associated with the migrant process; thus, the migration cost for an active process is much greater than the cost of remote execution. If preemptive migration is done carelessly, this additional cost might overwhelm the benefit of migrating processes with longer expected lives.

For this reason, we propose a strategy that guarantees that every migration improves the expected performance of the migrant process and the other processes at the source host.[3]

Whenever more than one process is running on a host and one process migrates away, the expected slowdown of the others decreases, regardless of the duration of the processes or the cost of migration. But the slowdown of the migrant process might increase, if the time spent migrating is greater than the time saved by running on a less-loaded host. Thus we will perform migration only if it improves the expected slowdown of the migrant process (the expectation is taken over the lifetime of the migrant).

[3]Of course, processes on the target host are slowed by an arriving migrant, but on a moderately-loaded system there are almost always idle hosts; thus the number of processes at the target host is usually zero. In any case, the number of processes at the target is always less than the number at the source.

If there is no process on the host that satisfies this criterion, no migration is done. If migration costs are high, few processes will be eligible for migration; in the extreme there will be no migration at all. But in no case is the performance of the system worse (in expectation) than the performance without migration.

Using the distribution of process lifetimes, we now show how to calculate the expected slowdown imposed on a migrant process, and use this result to derive a minimum age for migration based on the cost of migration. Denoting the age of the migrant process by a; the cost of migration by c; the (eventual total) lifetime of the migrant by L, the number of processes at the source host by n; and the number of processes at the target host (including the migrant) by m, we have:

$$\begin{aligned}
& \mathbf{E}\{\text{slowdown of migrant}\} \\
= & \int_{t=a}^{\infty} \mathbf{Pr}\left\{\begin{array}{c}\text{Lifetime of}\\ \text{migrant is t}\end{array}\right\} \cdot \left\{\begin{array}{c}\text{Slowdown given}\\ \text{lifetime is t}\end{array}\right\} dt \\
= & \int_{t=a}^{\infty} \mathbf{Pr}\{t \le L < t+dt | L \ge a\} \cdot \frac{na + c + m(t-a)}{t} \\
= & \int_{t=a}^{\infty} \frac{a}{t^2} \cdot \frac{na + c + m(t-a)}{t} dt \\
= & \frac{1}{2}\left(\frac{c}{a} + m + n\right)
\end{aligned}$$

If there are n processes at a heavily loaded host, then a process should be eligible for migration only if its expected slowdown after migration is less than n (which is the slowdown it expects in the absence of migration).

Thus, we require $\frac{1}{2}(\frac{c}{a} + m + n) < n$, which implies

$$\boxed{\text{Minimum migration age} = \frac{\text{Migration cost}}{n-m}}$$

This analysis extends easily to the case of heterogeneous processor speeds by applying a scale factor to n or m.

This analysis assumes that current load predicts future load; that is, that the load at the source and target hosts will be constant during the migration. In an attempt to

evaluate this assumption, and possibly improve it, we considered a number of alternative load predictors, including (1) taking a load average (over an interval of time), (2) summing the ages of the running processes at the target host, and a (3) calculating a prediction of survivors and future arrivals based on the distribution model proposed here. We found that current (instantaneous) load is the best single predictor, and that using several predictive variables in combination did not greatly improve the accuracy of prediction. These results are in accord with Zhou's thesis, [96] and with [42].

The MOSIX migration policy [7] is based on a restriction that is similar to the criterion we are proposing: the age of the process must exceed the migration cost. Thus, the slowdown imposed on the migrant process (due to migration) must be less than 2.0. This bound is based on the worst case, in which the migrant process completes immediately upon arrival at the target.

The MOSIX requirement is likely to be too restrictive, for two reasons. First, it ignores the slowdown that would be imposed at the source host in the absence of migration (presumably there is more than one process there, or the system would not be attempting to migrate processes away). Secondly, it is based on the worst-case slowdown rather than (as shown above) the expected slowdown. We will explicitly compare the MOSIX policy with ours in Section 11.5.

In our migration criterion, the term "migration cost" refers to any costs associated with migrating the process in question. In our model in this thesis we assume that the migration cost of a process is the time required to migrate its virtual memory (see Section 11.2.1), however our criterion is meant to hold for a general definition of migration cost including any cost associated with migrating a process, such as the cost of closing its files, the cost of future interactions, etc.. Such costs are beyond the scope of our model, but we discuss them in our future work section, Section 12.3.

Chapter 11

Trace-Driven Simulation and Results

At this point we're finally ready to answer the question we sent out to answer in Section 8.3, namely, is preemptive migration necessary, or is non-preemptive migration sufficient for load balancing? We use a trace-driven simulation of process migration to compare two migration strategies: our proposed age-based preemptive migration strategy (Section 10.3) and a non-preemptive strategy that migrates newborn processes according to the process name (similar to strategies proposed by [86] and [73]). Although we use a simple name-based strategy, we give it the benefit of several unfair advantages; for example, the name-lists are derived from the same trace data used by the simulator.

Section 11.1 describes the simulator and the two strategies in more detail. We use the simulator to run three experiments. First, in Section 11.2, we evaluate the sensitivity of each strategy to a wide range of values for preemptive and non-preemptive migration costs. We see that even when the preemptive migration cost is unnaturally high, preemptive migration still outperforms non-preemptive migration. Next, in Section 11.3, we choose values for these parameters that are representative of current systems (as described in Section 11.2.1) and compare the performance of the preemptive and non-preemptive strategies on a variety of metrics including: mean process slowdown, variance of slowdown, mean slowdown of only short jobs, mean slowdown of only long jobs, and number of severely slowed processes. In Section 11.4, we explain why our preemptive policy significantly outperformed the non-preemptive policy under all these metrics. We arrive at some general principles un-

derlying the effectiveness of preemptive migration, all of which are direct consequences of the process lifetime distribution. Lastly, in Section 11.5, we evaluate the analytic criterion for migration age (proposed in Section 10.3) used in our preemptive migration strategy, compared to criteria used in the literature.

11.1 The simulator

We have implemented a trace-driven simulation of a network of six identical workstations.[1] We selected six daytime intervals from the traces on machine po (see Section 9.3.1), each from 9:00 a.m. to 5:00 p.m. From the six traces we extracted the start times and CPU durations of the processes. We then simulate a network where each of six hosts executes (concurrently with the others) the process arrivals from one of the daytime traces.

Although the workloads on the six hosts are homogeneous in terms of the job mix and distribution of lifetimes, there is considerable variation in the level of activity during the eight-hour trace. For most of the traces, every process arrival finds at least one idle host in the system, but in the two busiest traces, a small fraction of processes (0.1%) arrive to find all hosts busy. In order to evaluate the effect of changes in system load, we divided the eight-hour trace into eight one-hour intervals. We refer to these as *runs* 0 through 7, where the runs are sorted from lowest to highest load. Run 0 has a total of $\sim$ 15000 processes submitted to the six simulated hosts; Run 7 has $\sim$ 30000 processes. The average duration of processes (for all runs) is $\sim$ .4 seconds. Thus the total utilization of the system, ρ, is between .27 and .54.

The birth process of jobs at our hosts is burstier than a Poisson process. For a given run and a given host, the serial correlation[2] in the process interarrival times is typically between .08 and .24, which is significantly higher than one would expect from a Poisson process (uncorrelated interarrival times yield a serial correlation of 0.0; perfect correlation is 1.0).

Although the start times and durations of the processes come from trace data, the

[1] The trace-driven simulator and the trace data are available at http://http.cs.berkeley.edu/~harchol/loadbalancing.html.

[2] Let $(x_1, x_2, \ldots, x_n)$ be the inter-arrival times. We measured the correlation between the vectors $(x_1, x_2, \ldots, x_{n-1})$ and $(x_2, x_3, \ldots, x_n)$ to determine how much the next inter-arrival time depends on the previous inter-arrival time.

memory size of each process, which determines its migration cost, is chosen randomly from a measured distribution (see Section 11.2). This simplification obliterates any correlations between memory size and other process characteristics, but it allows us to control the mean memory size as a parameter and examine its effect on system performance. In our informal study of processes in our department, we did not detect any correlations between memory size and process CPU usage. Krueger and Livny, [41], make the same observation in their department.

In our system model, we assume that processes are always ready to run (i.e. are never blocked on I/O). During a given time interval, we divide CPU time equally among the processes on the host.

In real systems, part of the migration time is spent on the source host packaging the transferred pages, part in transit in the network, and part on the target host unpacking the data. The size of these parts and whether they can be overlapped depend on details of the system. In our simulation we charge the entire cost of migration to the source host. This simplification is a pessimistic assumption for advocates of preemptive migration.

11.1.1 Strategies

We compare a non-preemptive migration strategy with our proposed preemptive migration strategy (from Section 10.3). For purposes of comparison, we have tried to make the policies as simple and as similar as possible. For both types of migration, we consider performing a migration only when a new process is born, even though a preemptive strategy might benefit by initiating migrations at other times. Also, for both strategies, a host is considered heavily-loaded any time it contains more than one process; in other words, any time it would be sensible to consider migration. Finally, we use the same location policy in both cases: the host with the lowest instantaneous load is chosen as the target host (ties are broken by random selection).

Thus the only difference between the two migration policies is which processes are considered eligible for migration:

name-based non-preemptive migration A process is eligible for migration only if its name is on a list of processes that tend to be long-lived. If an eligible process is born at a heavily-loaded host, the process is executed remotely on the target host. Processes cannot be migrated once they have begun execution.

The performance of this strategy depends on the list of eligible process names. We derived this list by sorting the processes from the traces according to name and duration and selecting the 15 common names with the longest *mean durations*. We chose a threshold on mean duration that is empirically optimal (for this set of runs). Adding more names to the list detracts from the performance of the system, as it allows more short-lived processes to be migrated. Removing names from the list detracts from performance as it becomes impossible to migrate enough processes to balance the load effectively. Since we used the trace data itself to construct the list, our results may overestimate the performance benefits of this strategy.

age-based preemptive migration A process is eligible for migration only if it has aged for some fraction of its migration cost. Based on the derivation in Section 10.3, this fraction is $\frac{1}{n-m}$, where n (respectively m) is the number of processes at the source (target) host. When a new process is born at a heavily-loaded host, all processes that satisfy the migration criterion are migrated away.

This strategy understates the performance benefits of preemptive migration, because it does not allow the system to initiate migrations except when a new process arrives. We have modeled strategies that allow migration at other times, and they do improve the performance of the preemptive strategy, but we have omitted them here to facilitate comparison between the two migration strategies.

As described in Section 10.3, we also modeled other location policies based on more complicated predictors of future loads, but none of these predictors yielded significantly better performance than the instantaneous load we use here.

We also considered the effect of allowing preemptive migration at times other than when a new process arrives. Ideally, one would like to initiate a migration whenever a process becomes eligible (since the eligibility criterion guarantees that the performance of the migrant will improve in expectation). But it is not feasible to check the system constantly to find all eligible jobs. One of the strategies we considered was to perform periodic checks of each process on a heavily-loaded host to see if any satisfied the criterion. The performance of this strategy was significantly better than that of the simpler policy (migrating only when a process arrives) and improved as the frequency of checking increased. However, because our system model does not quantify the overhead associated with these checks, we do not know what frequency is feasible in a real system.

11.1.2 Metrics

We evaluate the effectiveness of each strategy according to the following performance metrics:

mean slowdown Slowdown is the ratio of wall-clock execution time to CPU time (thus, it is always greater than one). The average slowdown of all jobs is a common metric of system performance. When we compute the ratio of mean slowdowns (as from different strategies) we will use normalized slowdown, which is the ratio of inactive time (the *excess* slowdown caused by queueing and migration delays) to CPU time. For example, if the (unnormalized) mean slowdown drops from 2.0 to 1.5, the ratio of normalized mean slowdowns is $.5/1.0 = .5$: a 50% reduction in delay.

Mean slowdown alone, however, is not a sufficient measure of the difference in performance of the two strategies; it understates the advantages of the preemptive strategy for these two reasons:

1. Skewed distribution of slowdowns: Even in the absence of migration, the majority of processes suffer small slowdowns (typically 80% are less than 3.0. See Figure 11.1). The value of the mean slowdown will be dominated by this majority.

2. User perception: From the user's point of view, the important processes are the ones in the tail of the distribution, because although they are the minority, they cause the most noticeable and annoying delays. Eliminating these delays might have a small effect on the mean slowdown, but a large effect on a user's perception of performance.

Therefore, we will also consider the following three metrics:

variance of slowdown : This metric is often cited as a measure of the unpredictability of response time [71], which is a nuisance for users trying to schedule tasks. In light of the distribution of slowdowns, however, it may be more meaningful to interpret this metric as a measure of the length of the tail of the distribution; i.e. the number of jobs that experience long delays. (See Section 11.3, Figure 11.3(b)).

number of severely slowed processes : In order to quantify the number of noticeable delays explicitly, we consider the number (or percentage) of processes that are severely impacted by queueing and migration penalties. (See Section 11.3, Figures 11.3(c) and 11.3(d)).

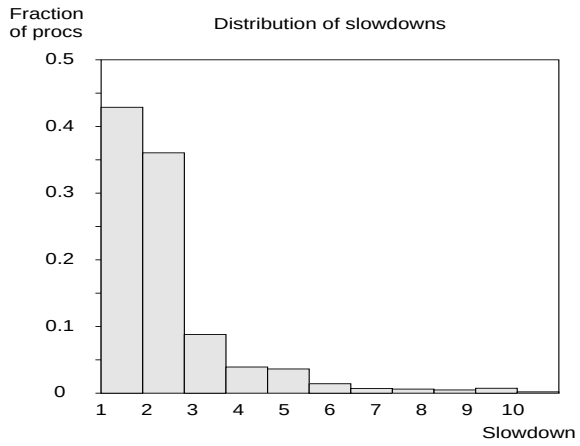


Figure 11.1: *Distribution of process slowdowns for run* 0 *(with no migration). Most processes suffer small slowdowns, but the processes in the tail of the distribution are more noticeable and annoying to users.*

mean slowdown of long jobs : Delays in longer jobs (those with lifetimes greater than .5 seconds) are more perceivable to users than delays in short jobs. (See Section 11.4.1, Figure 11.4).

11.2 Sensitivity to migration costs

The purpose of this section is to compare the performance of the non-preemptive and preemptive strategies over a range of values of migration costs. First in Section 11.2.1 we define our model for preemptive and non-preemptive migration costs, based on that used in real systems. In Section 11.2.2 we measure the performance of the preemptive policy compared with our non-preemptive policy as a function of the cost of preemptive migration.

11.2.1 Model of migration costs

This section presents the model of migration costs we will use in our simulator. We model the cost of migrating an active process as the sum of a *fixed migration cost* for migrating the process' system state plus a *memory transfer cost* that is proportional to the amount of the process' memory that must be transferred. We model *remote execution cost* as a fixed cost; it is the same for all processes.

Throughout this paper, we refer to the following parameters:

r the cost of remote execution, in seconds

f the fixed cost of preemptive migration, in seconds

b the memory transfer bandwidth, in MB's per second

m the memory size of migrant processes, in MB

$$\begin{aligned} \text{cost of remote execution} &= r \\ \text{cost of preemptive migration} &= f + m/b \end{aligned}$$

We refer to the quotient m/b as the memory transfer cost.

Observe that the amount of a process' memory (m) that must be transferred during preemptive migration depends on properties of the distributed system. At most, it might be necessary to transfer a process' entire memory. On a system like Sprite [20], which integrates virtual memory with a distributed file system, it is only necessary to write dirty pages to the file system before migration. When the process is restarted at the target host, it will retrieve these pages. In this case the cost of migration is proportional to the size of the resident set rather than the size of memory. In systems that use precopying (such as the V [75] system), pages are transferred while the program continues to run at the source host. When the job stops execution at the source, it will have to transfer again any pages that have become dirty during the precopy. Although the number of pages transferred might be increased, the delay imposed on the migrant process is greatly decreased. Additional techniques can reduce the cost of transferring memory even more ([93]).

The specific parameters of migration cost depend not only on the nature of the system (as discussed above) but also on the speed of the network. Tables 11.1 and 11.2 below show reported values for parameters on a variety of real systems. In the next section we will use a trace-driven simulator to evaluate the effect of these parameters on system performance.

11.2.2 Sensitivity of policies to migration costs

In this section we compare the performance of the non-preemptive and preemptive strategies over a range of values of migration costs r, f, b and m.

For the following experiments, we chose the remote execution cost $r = .3$ seconds. We considered a range for the fixed migration cost of $.1 < f < 10$ seconds.

System	Hardware	Cost of remote execution (r)
Sprite [20]	Network of SPARCstation1's connected by 10Mb/sec Ethernet	$r = .33$ seconds
GLUNIX [79],[78]	Network of HP workstations connected by ATM network	$r = .25 - .5$ seconds
MIST [60]	Network of HP9000/720 workstations connected by 10Mb/sec Ethernet	$r = .33$ seconds
Utopia [95]	Network of DEC 3100 and SUN SPARC IPC's connected by Ethernet	$r = .1$ seconds, after connection established

Table 11.1: *Values for non-preemptive migration cost in various systems. Many of these number obtained from personal communication with the authors.*

System	Hardware	Fixed Preemptive Cost (f)	Inverse Bandwidth ($1/b$)
Sprite [20]	Network of SPARCstation1's connected by 10Mb/sec Ethernet	$f = .33$ secs	2.00 sec/MB
MOSIX [7],[11].	Network of Pentium-90's connected by 100Mb/sec Ethernet	$f = .006$ secs	.44 sec/MB
MIST [60]	Network of HP9000/720's connected by 10Mb/sec Ethernet	$f = .24$ secs	.99 sec/MB

Table 11.2: *Values for preemptive migration costs from various systems. Many of these number obtained from personal communication with the authors. Observe that the memory transfer cost is the product of the inverse bandwidth ($1/b$) and the amount of memory that must be transferred* (m).

The memory transfer cost is the quotient of m (the memory size of the migrant process) and b (the bandwidth of the network). We chose the memory transfer cost from a distribution with the same shape as the distribution of process lifetimes, setting the mean memory transfer cost (MMTC) to a range of values from 1 to 64.

The shape of the memory transfer cost distribution is based on an informal study of memory-use patterns on the same machines from which we collected trace data. The important feature of this distribution is that there are many jobs with small memory demands and a few jobs with very large memory demands. Empirically, the exact form of this distribution does not affect the performance of either migration strategy strongly, but of course the mean (MMTC) does have a strong effect.

Figures 11.2a and 11.2b are contour plots of the ratio of the performance of the two migration strategies using normalized slowdown. Specifically, for each of the eight one-hour runs we calculate the mean (respectively standard deviation) of the slowdown imposed on all processes that complete during the hour. For each run, we then take the ratio of the means (standard deviations) of the two strategies. Lastly we take the geometric mean [33] of the eight ratios.

The two axes in Figure 11.2 represent the two components of the cost of preemptive migration, namely the fixed cost (f) and the MMTC (m/b). As mentioned above, the cost of non-preemptive migration (r) is fixed at .3 seconds. As expected, increasing either the fixed cost of migration or the MMTC hurts the performance of preemptive migration. The contour line marked 1.0 indicates the crossover where the performance of preemptive and non-preemptive migration is equal (the ratio is 1.0). For smaller values of the cost parameters, preemptive migration performs better; for example, if the fixed migration cost is .33 seconds and the MMTC is 2 seconds, the normalized mean slowdown with preemptive migration is $\sim$ 40% lower than with non-preemptive migration. When the fixed cost of migration or the MMTC are very high, almost all processes are ineligible for preemptive migration; thus, the preemptive strategy does almost no migrations. The non-preemptive strategy is unaffected by these costs so the non-preemptive strategy can be more effective.

Figure 11.2b shows the effect of migration costs on the standard deviation of slowdowns. The crossover point — where non-preemptive migration surpasses preemptive migration — is considerably higher in Figure 11.2b than in Figure 11.2a. Thus there is a region where preemptive migration yields a higher mean slowdown than non-preemptive migration, but a lower standard deviation. The reason for this is that non-preemptive

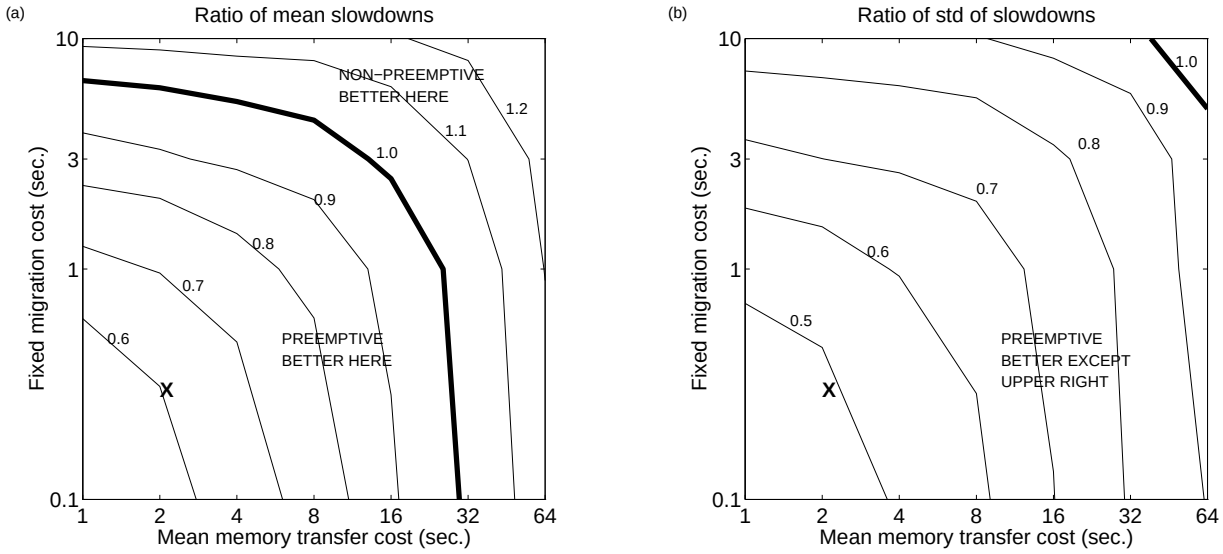


Figure 11.2: *(a) The performance of preemptive migration relative to non-preemptive migration deteriorates as the cost of preemptive migration increases. The two axes are the two components of the preemptive migration cost. The cost of non-preemptive migration is held fixed. The "X" marks the particular set of parameters we will consider in the next section. (b) The standard deviation of slowdown may give a better indication of a user's perception of system performance than mean slowdown. By this metric, the benefit of preemptive migration is even more significant.*

migration occasionally chooses a process for remote execution that turns out to be short-lived. These processes suffer large delays (relative to their run times) and add to the tail of the distribution of slowdowns. In the next section, we show cases in which the standard deviation of slowdowns is actually worse with non-preemptive migration than with no migration at all (three of the eight runs).

11.3 Comparison of preemptive and non-preemptive strategies

In this section we choose migration cost parameters representative of current systems (see Section 11.2.1) and use them to examine more closely the performance of the two migration strategies. The values we chose are:

r the cost of remote execution, .3 seconds

f the fixed cost of preemptive migration, .3 seconds

b the memory transfer bandwidth, .5 MB per second

m the mean memory size of migrant processes, 1 MB

In Figures 11.2a and 11.2b, the point corresponding to these parameter values is marked with an "X". Figure 11.3 shows the performance of the two migration strategies at this point (compared to the base case of no migration).

Non-preemptive migration reduces the normalized mean slowdown (Figure 11.3a) by less than 20% for most runs (and $\sim$ 40% for the two runs with the highest loads). Preemptive migration reduces the normalized mean slowdown by 50% for most runs (and more than 60% for two of the runs). The performance improvement of preemptive migration over non-preemptive migration is typically between 35% and 50%.

As discussed above, we feel that the mean slowdown (normalized or not) understates the performance benefits of preemptive migration. We have proposed other metrics to try to quantify these benefits. Figure 11.3b shows the standard deviation of slowdowns, which reflects the number of severely impacted processes. Figures 11.3c and 11.3d explicitly measure the number of severely impacted processes, according to two different thresholds of acceptable slowdown. By these metrics, the benefits of migration in general appear greater, and the discrepancy between preemptive and non-preemptive migration appears much greater. For example in Figure 11.3d, in the absence of migration, $7 - 18\%$ of processes are slowed by a factor of 5 or more. Non-preemptive migration is able to eliminate $42 - 62\%$ of these, which is a significant benefit, but preemptive migration consistently eliminates nearly all $(86 - 97\%)$ severe delays.

An important observation from Figure 11.3b is that for several of the runs, non-preemptive migration actually makes the performance of the system worse than if there were no migration at all. For the preemptive migration strategy, this outcome is nearly impossible, since migrations are only performed if they improve the slowdowns of all processes involved (in expectation). In the worst case, then, the preemptive strategy will do no worse than the case of no migration (in expectation).

Another benefit of preemptive migration is graceful degradation of system performance as load increases (as shown in Figure 11.3). In the presence of preemptive migration, both the mean and standard deviation of slowdown are nearly constant, regardless of the overall load on the system.

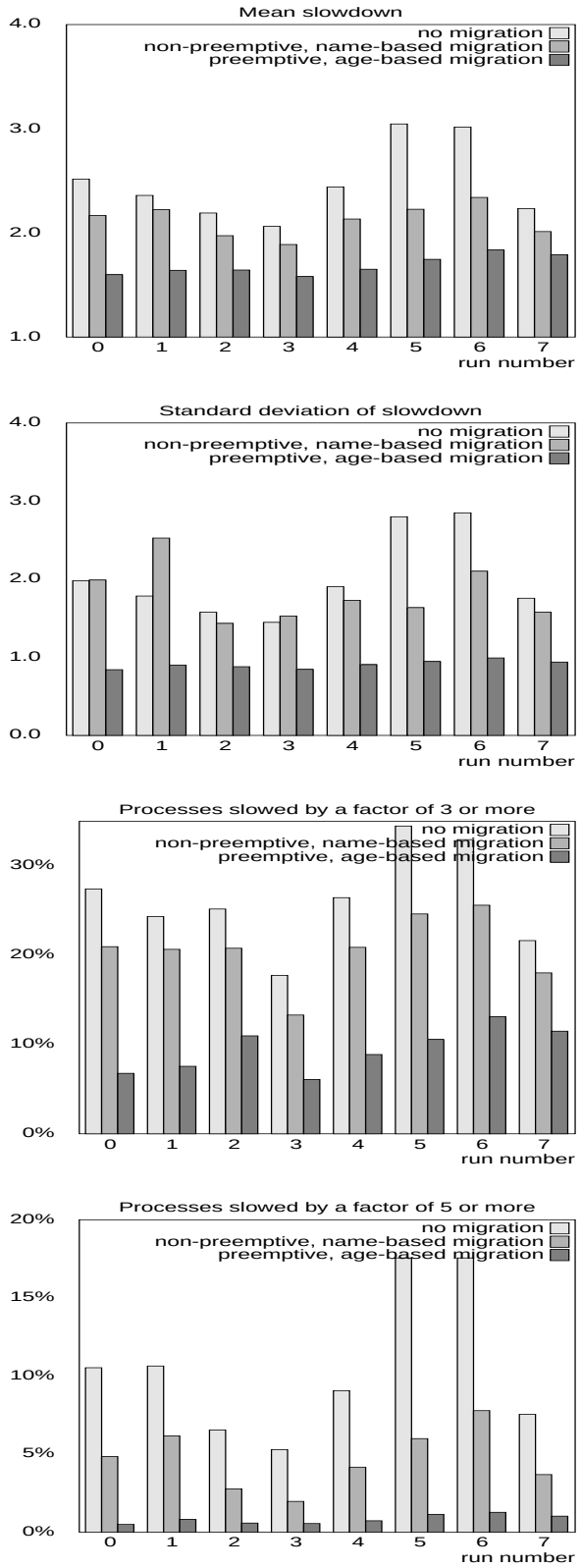


Figure 11.3: *(a) Mean slowdown. (b) Standard deviation of slowdown. (c) Percentage of processes slowed by a factor of 3 or more. (d) Percentage of processes slowed by a factor of 5 or more. All shown at cost point X from Figure 11.2.*

11.4 Why preemptive migration outperformed non-preemptive migration

The alternate metrics discussed above shed some light on the reasons for the performance difference between preemptive and non-preemptive migration. We consider two kinds of mistakes a migration system might make:

Failing to migrate long-lived jobs : This type of error imposes moderate slowdowns on a potential migrant, and, more importantly, inflicts delays on short jobs that are forced to share a processor with a CPU hog. Under non-preemptive migration, this error occurs whenever a long-lived process is not on the name-list, possibly because it is an unknown program or an unusually long execution of a typically short-lived program. Preemptive migration can correct these errors by migrating long jobs later in their lives.

Migrating short-lived jobs : This type of error imposes large slowdowns on the migrated process, wastes network resources, and fails to effect significant load balancing. Under non-preemptive migration, this error occurs when a process whose name is on the eligible list turns out to be short-lived. Our preemptive migration strategy eliminates this type of error by guaranteeing that the performance of a migrant improves in expectation.

Even occasional mistakes of the first kind can have a large impact on performance, because one long job on a busy machine will impede many small jobs. This effect is aggravated by the serial correlation between arrival times (see Section 11.1), which suggests that a busy host is likely to receive many future arrivals.

Thus, an important feature of a migration policy is its ability to identify long-lived jobs for migration. To evaluate this ability, we consider the average lifetime of the processes chosen for migration under each policy in our simulation. Under non-preemptive migration, the average lifetime of migrant processes under was 1.97 seconds (the mean lifetime for all processes is 0.4 seconds) and the median lifetime of migrants was .86 seconds. The non-preemptive policy migrated about 1% of all jobs, which accounted for 5.7% of the total CPU.

Our preemptive migration policy was better able to identify long jobs; the average lifetime of migrant processes under preemptive migration was 4.9 seconds; the median

lifetime of migrants was 1.98 seconds. The preemptive policy migrated 4% of all jobs, but since these migrants were long-lived, they accounted for 55% of the total CPU.

Thus the primary reason for the success of preemptive migration is its ability to identify long jobs accurately and to migrate those jobs away from busy hosts.

The second type of error did not have as great an impact on the mean slowdown for all processes, but it did impose large slowdowns on some small processes. These outliers are reflected in the standard deviation of slowdowns — because the non-preemptive policy sometimes migrates very short jobs, it can make the standard deviation of slowdowns worse than with no migration (see Figure 11.3b). Our age-based preemptive migration criterion eliminates errors of this type by guaranteeing that the performance of the migrant will improve in expectation.

There is, however, one type of migration error that is more problematic for preemptive migration than for non-preemptive migration: stale load information. A target host may have a low load when a migration is initiated, but its load may have increased by the time the migrant arrives. This is more likely for a preemptive migration because the migration time is longer. In our simulations, we found that these errors do occur, although infrequently enough that they do not have a severe impact on performance.

Specifically, we counted the number migrant processes that arrived at a target host and found that the load was higher than it had been at the source host when migration began. For most runs, this occurred less than 0.5% of the time (for two runs with high loads it was 0.7%). Somewhat more often, $\sim 3\%$ of the time, a migrant process arrived at a target host and found that the load at the target was greater than the *current* load at the source. These results suggest that the performance of a preemptive migration strategy might be improved by rechecking loads at the end of a memory transfer and, if the load at the target is too high, aborting the migration and restarting the process on the source host.

One other potential problem with preemptive migration is the volume of network traffic that results from large memory transfers. In our simulations, we did not model network congestion, on the assumption that the traffic generated by migration would not be excessive. This assumption seems to be reasonable: under our preemptive migration strategy fewer than 4% of processes are migrated once and fewer than .25% of processes are migrated more than once. Furthermore, there is seldom more than one migration in progress at a time.

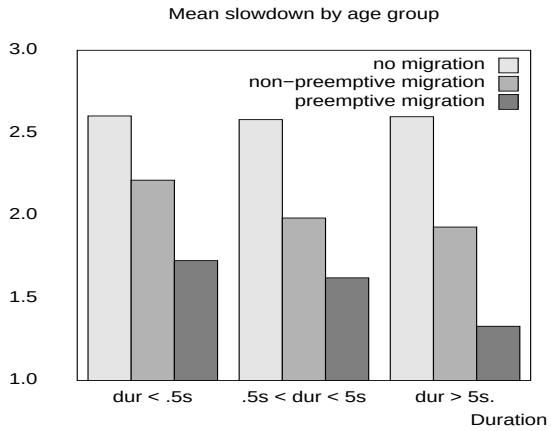


Figure 11.4: *Mean slowdown broken down by lifetime group.*

In summary, the advantage of preemptive migration — its ability to identify long jobs and move them away from busy hosts — overcomes its disadvantages (longer migration times and stale load information).

11.4.1 Effect of migration on short and long jobs

We have claimed that identifying long jobs and migrating them away from busy hosts helps not only the long jobs (which run on less-loaded hosts) but also the short jobs that run on the source host. To test this claim, we divided the processes into three lifetime groups and measured the performance benefit for each group due to migration. Figure 11.4 shows that migration reduces the mean slowdown of all three of these groups: for non-preemptive migration the improvement is the same for all groups; under preemptive migration the long jobs enjoy a slightly greater benefit.

This breakdown by lifetime group is useful for evaluating various metrics of system performance. The metric we are using here, slowdown, gives equal weight to all jobs; as a result, the mean slowdown metric is dominated by the most populous[3] group, short jobs. Another common metric, residence time, gives weight to jobs that is proportional to their lifetime. Thus the mean residence time metric reflects, primarily, the performance benefit for long jobs (under this metric preemptive migration appears even more effective).

[3]The number of jobs in each group increases roughly by a factor of 10 moving from longest to the shortest group.

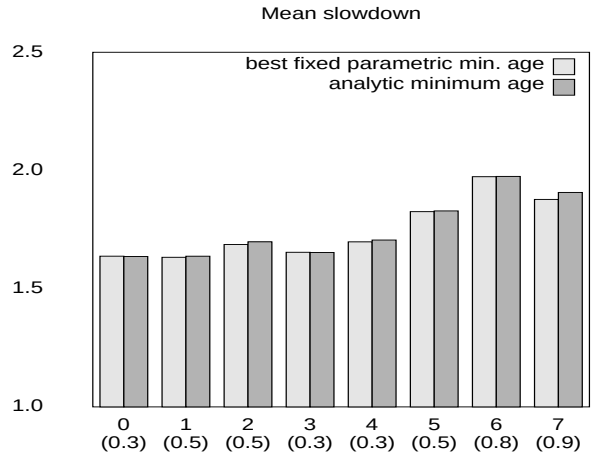


Figure 11.5: *The mean slowdown for eight runs, using the two criteria for minimum migration age. The value of the best fixed parameter α is shown in parentheses for each run.*

11.5 Evaluation of analytic migration criterion

As derived in Section 10.3, the minimum age for a migrant process according to our *analytic criterion* is

$$\text{Minimum migration age} = \frac{\text{Migration cost}}{(n-m)}$$

where n is the load at the source host and m is the load at the target host (including the potential migrant).

We will compare the analytic criterion with the *fixed parameter criterion*:

$$\text{Minimum migration age} = \alpha * \text{Migration cost}$$

where α is a free parameter. This parameter α is meant to model preemptive migration strategies in the literature, as discussed in Section 10.2. For comparison, we will use the *best fixed parameter*, which is, for each run, the best parameter for that run, chosen empirically by executing the run with a range of parameter values (of course, this gives the fixed parameter criterion a considerable advantage).

Figure 11.5 compares the performance of the analytic minimum age criterion with the best fixed parameter (α). The best fixed parameter varies considerably from run to run, and appears to be roughly correlated with the average load during the run (the runs are sorted in increasing order of total load).

The performance of the analytic criterion is always within a few percent of the performance of the best fixed value criterion. The advantage of the analytic criterion is that it is parameterless, and therefore more robust across a variety of workloads. We feel that the elimination of one free parameter is a useful result in an area with so many (usually hand-tuned) parameters.

Chapter 12

Conclusion and Future Work

This chapter concludes Part II. In Section 12.1 we summarize the main themes of our research on load balancing analysis. In Sections 12.2 and 12.3, we examine limitations of our load balancing model, and discuss how our results might be affected when our work is applied to a fully general system.

12.1 Summary

We began Part II with two questions:

1. Is preemptive migration necessary for load balancing?
2. What is a good preemptive policy?

Throughout Part II, we've argued that the answers to both these questions lie in understanding and applying the process lifetime distribution.

With respect to the first question above:

Is preemptive migration necessary for load balancing?

We began by pointing out that preemptive migration policies for load balancing are rarely considered both in real systems and in studies, despite the fact that several systems are capable of migrating active processes. We also noted that some systems' choice to not include a preemptive load balancing policy was influenced by the fact that the few studies of preemptive migration couldn't agree on whether or not it was necessary. Throughout Part II, we argue that our measured process lifetime distribution provides compelling evidence of

the power of preemptive migration. The reason researchers have overlooked the potential for preemptive migration appears to be a lack of knowledge about the lifetime distribution, and a lack of awareness of the sensitivity of load balancing studies to the assumed distribution. Our arguments are summarized in the following points:

- As the process lifetime distribution shows, current CPU age provides a consistently accurate prediction of future CPU usage (Section 9.3). Even the highly-tuned[1] name-list was not able to identify long-lived processes as well as our very simple *age*-based preemptive policy. The mean and median lifetimes of jobs selected by our age-based preemptive policy were about 2.5 times longer than those selected by the highly tuned non-preemptive policy (Section 11.4).

- As the process lifetime distribution shows, migrating only the very longest of jobs manipulates the majority of the CPU in the system. Our preemptive policy affected 55% of the total CPU by only migrating 4% of the jobs, while our non-preemptive policy affected only 5% of the total CPU (Section 11.4 and Section 9.3).

- Non-preemptive name-based strategies may fail to migrate a long-lived process which isn't on the name-list. Preemptive age-based policies are able to catch *all* worthy long-lived processes, because if they accidentally fail to migrate a long-lived job, they will recover their error later when the job ages a little more. The lifetime distribution tells us it's important to identify *all* worthy migrants, because, migrating one long job out of the way helps the many, many more small jobs that stay behind, and therefore has a large effect on the mean slowdown (Section 11.4).

- Assuming an exponential process lifetime distribution, as many studies do, defeats the purpose of preemptive migration, since it implies that a process' expected remaining lifetime is independent of its age (Section 9.5). This may help explain why most studies have ignored preemptive migration entirely.

- Even assuming a lifetime distribution which matches the correct one in both mean and variance may still be misleading, and may overestimate the effectiveness of non-preemptive migration and underestimate the effectiveness of preemptive migration (see Section 9.5 and Section 8.6).

[1] Recall, the names in the name-list were chosen from the same traces as the policy was later tested on. Furthermore, the number of names in the name-list was chosen so as to optimize the non-preemptive policy's performance on that test-case.

- Exclusive use of mean slowdown as a metric of system performance understates the benefits of load balancing as perceived by users, and especially understates the benefits of preemptive load balancing. One performance metric which is more related to user perception is the number of severely slowed processes. While non-preemptive migration cuts these in half, preemptive migration reduces these by a factor of ten or more (Section 11.3). Another metric which is closer to user perception is the slowdown of exclusively longer jobs (those with lifetimes more than a second), since slowdowns on shorter jobs are less perceivable to users. Preemptive migration shows even greater improvement with respect to reducing mean slowdown on this longer set of jobs, than in reducing mean slowdown in general (see Section 11.4.1).

With respect to the second question above:

What is a good preemptive policy?

We began by pointing out that previous preemptive policies in the literature are somewhat unsatisfactory in that they all rely on system-tuned parameters which must be optimized by hand to the particular system (Section 10.2). We were able to eliminate the free parameter by applying the functional form or our measured lifetime distribution. We made two main points:

- We showed how to use the $1/T$ observed lifetime distribution as an analytical tool for deriving a preemptive policy which specifies the minimum time a process must age before being migrated as a function of the process' migration cost, and the loads at its source and target host. This criterion is parameterless and robust across a range of loads (Section 10.3).
- We showed that our preemptive policy outperformed the non-preemptive policy even when memory transfer costs were atypically high (Section 11.2).

12.2 Weaknesses of the model

Our simulation ignores a number of factors that would affect the performance of migration in real systems:

CPU-bound jobs only : Our model considers all jobs CPU-bound; thus, their response time necessarily improves if they run on a less-loaded host. For I/O bound jobs,

however, CPU contention has little effect on response time. These jobs would benefit less from migration. To see how large a role this plays in our results, we noted the names of the processes that appear most frequently in our traces (with CPU time greater than 1 second, since these are the processes most likely to be migrated). The most common names were "cc1plus" and "cc1," both of which are CPU bound. Next most frequent were: trn, cpp, ld, jove (a version of emacs), and ps. So, although some jobs in our traces are in reality interactive, our simple model is reasonable for many of the most common jobs. In Section 12.3 we discuss further implications of a workload including interactive, I/O-bound, and non-migratable jobs.

environment : Our migration strategy takes advantage of the used-better-than-new property of process lifetimes. In an environment with a different distribution, this strategy will not be effective.

network contention : Our model does not consider the effect of increased network traffic as a result of process migration. We observe, however, that for the load levels we simulated, migrations are occasional (one every few seconds), and that there is seldom more than one migration in progress at a time.

local scheduling : We assume that local scheduling on the hosts is similar to round-robin. Other policies, like feedback scheduling, can reduce the impact of long jobs on the performance of short jobs, and thereby reduce the need for load balancing. We explore this issue in more detail in Section 12.3.

memory size : One weakness of our model is that we chose memory sizes from a measured distribution and therefore our model ignores any correlation between memory size and other process characteristics. Also, we've made the pessimistic simplification that a migrant's entire memory must be transferred, although this is not always the case.

12.3 Areas of Future Work

Of primary interest in future work is including interactive, I/O-bound, and non-migratable jobs into our workload. We conjecture on the effects thereof.

In a workload that includes interactive jobs, I/O-bound jobs, and daemons, there will be some jobs that should not or cannot be migrated. An I/O-bound job, for example,

will not necessarily run faster on a less-loaded host, and might run slower if it is migrated away from the disk or other I/O device it uses. A migrated interactive job might benefit by running on a less-loaded host if it performs sufficient CPU, but, will suffer performance costs for all future interactions. Finally, some jobs (e.g. many daemons) cannot be migrated away from their hosts.

Our proposed policy for preemptive migration can be extended to deal appropriately with interactive and I/O bound jobs by including in the definition of migration cost the additional costs that will be imposed on these jobs after migration, including network delays, access to non-local data, etc. The estimates of these costs might be based on the recent behavior of the job; e.g. the number and frequency of I/O requests and interactions. Even within this expanded cost model, there might be a class of jobs that are explicitly forbidden to migrate. These jobs could be assigned an infinite migration cost.

The presence of a set of jobs that are either expensive or impossible to migrate might reduce the ability of the migration policy to move work around the network and balance loads effectively. However, we observe that the majority of long-lived jobs are, in fact, CPU-bound, and it is these long-lived jobs that consume the majority of CPU time. Thus, even if the migration policy were only able to migrate a subset of the jobs in the system, it could still have a significant CPU load-balancing effect.

Another way in which the proposed migration policy should be altered in a more general environment is that n (the number of jobs at the source) and m (the number of jobs at the target host) should distinguish between CPU-bound jobs, and other types of jobs, since only CPU-bound jobs affect CPU contention, and therefore are significant in CPU load balancing.

Another important consideration in load balancing is the effect of local scheduling at the hosts. Most prior studies of load balancing have assumed, as we do, that the local scheduling is round-robin (or processor-sharing). A few assume first-come-first-serve (FCFS) scheduling, and even fewer assume feedback scheduling, where processes that have used the least CPU time are given priority over older processes. In our simulations we found that these policies could prevent long jobs from interfering with short jobs, and thereby reduce the need for load balancing and the effectiveness of migration (both non-preemptive and preemptive).

In reality, the local scheduling used in practice lies somewhere between processor-sharing and feedback. Decay-usage scheduling as used in UNIX has some characteristics

of both policies [24]. Young jobs do have some precedence, but old jobs that perform interaction or other I/O are given higher priority, which allows them to interfere with short jobs. The more recent "lottery scheduling" [83] behaves more like processor-sharing. To understand the effect of local scheduling on load balancing requires modeling the interaction and I/O patterns of processes, in future research.

Bibliography

[1] Rakesh Agrawal and Ahmed Ezzet. Location independent remote execution in NEST. *IEEE Transactions on Software Engineering*, 13(8):905–912, August 1987.

[2] Ishfaq Ahmad, Arif Ghafoor, and Kishan Mehrotra. Performance prediction of distributed load balancing on multicomputer systems. In *Supercomputing*, pages 830–839, 1991.

[3] Bill Aiello, Tom Leighton, Bruce Maggs, and Mark Newman. Fast algorithms for bit-serial routing on a hypercube. In *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 55–64, July 1990.

[4] R. Aleliunas. Randomized parallel communication. In *ACM-SIGOPS Symposium on Principles of Distributed Systems*, pages 60–72, 1982.

[5] Venkat Anantharam. On the sojourn time of sessions at an ATM buffer with long-range dependent input traffic. In *Proceedings of the 34th IEEE Conference on Decision and Control*, December 1995.

[6] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *IEEE Computer*, pages 47–56, September 1989.

[7] Amnon Barak, Guday Shai, and Richard G. Wheeler. *The MOSIX Distributed Operating System:Load Balancing for UNIX*. Springer Verlag, Berlin, 1993.

[8] F. Baskett, K.M. Chandy, R.R. Muntz, and F. Palacios-Gomez. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the Association for Computing Machinery*, 22:248–260, 1975.

[9] Flavio Bonomi and Anurag Kumar. Adaptive optimal load balancing in a nonhomogeneous multiserver system with a central job scheduler. *IEEE Transactions on Computers*, 39(10):1232–1250, October 1990.

[10] Allan Borodin. Towards a better understanding of pure packet routing. In *Algorithms and Data Structures. Third Workshop. WADS'93*, pages 14–25, Canada, August 1993.

[11] Avner Braverman, 1995. Personal Communication.

[12] Raymond M. Bryant and Raphael A. Finkel. A stable distributed scheduling algorithm. In *2nd International Conference on Distributed Computing Systems*, pages 314–323, 1981.

[13] John A. Buzacott and J. George Shanthikumar. *Stochastic Models of Manufacturing Systems*. Prentice Hall, 1993.

[14] Jeremy Casas, Dan L. Clark, Ravi Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. Mpvm: A migration transparent version of pvm. *Computing Systems*, 8(2):171–216, Spring 1995.

[15] Thomas L. Casavant and Jon G. Kuhl. Analysis of three dynamic distributed load-balancing strategies with varying global information requirements. In *7th International Conference on Distributed Computing Systems*, pages 185–192, September 1987.

[16] Yukon Chang and Janos Simon. Continuous routing and batch routing on the hypercube. In *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 272–278, 1986.

[17] Shyamal Chowdhury. The greedy load sharing algorithm. *Journal of Parallel and Distributed Computing*, 9:93–99, 1990.

[18] E.G. Coffman, Nabil Kahale, and F.T. Leighton. Processor-ring communication: A tight asymptotic bound on packet waiting times, July 1995. Manuscript. Not yet published.

[19] Mark Crovella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and causes. In *ACM Sigmetrics '96 Conference on Measurement and Modeling of Computer Systems*, pages 160–169, 1996.

[20] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software – Practice and Experience*, 21(8):757–785, August 1991.

[21] Allen B. Downey and Mor Harchol-Balter. A note on "The limited performance benefits of migrating active processes for load sharing". Technical Report UCB//CSD-95-888, University of California, Berkeley, November 1995.

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

[23] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. The limited performance benefits of migrating active processes for load sharing. In *SIGMETRICS*, pages 662–675. ACM, May 1988.

[24] D.H.J. Epema. An analysis of decay-usage scheduling in multiprocessors. In *ACM Sigmetrics '95 Conference on Measurement and Modeling of Computer Systems*, pages 74–85, 1995.

[25] D. J. Evans and W. U. N. Butt. Dynamic load balancing using task-transfer probabilites. *Parallel Computing*, 19:897–916, August 1993.

[26] Dominico Ferrari, 1994. Personal Communication.

[27] G.W. Gerrity, A. Goscinski, J. Indulska, W. Toomey, and W. Zhu. RHODOS–a testbed for studying design issues in distributed operating systems. In *Towards Network Globalization (SICON 91): 2nd International Conference on Networks*, pages 268–274, September 1991.

[28] Ronald I. Greenberg and Charles E. Leiserson. Randomized routing on fat-trees. In *26th Annual Symposium on Foundations of Computer Science*, pages 241–9, October 1985.

[29] Anna Hać and Xiaowei Jin. Dynamic load balancing in a distributed system using a sender-initiated algorithm. *Journal of Systems Software*, 11:79–94, 1990.

[30] Mor Harchol-Balter. Bounding delays in packet-routing networks with light traffic. Technical Report UCB//CSD-95-885, University of California, Berkeley, October 1995.

[31] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996. Also appeared in Proceedings Fifteenth ACM Symposium on Operating System Principles Poster Session, December, 1995.

[32] Mor Harchol-Balter and David Wolfe. Bounding delays in packet-routing networks. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 248–257, May 1995.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[34] J. R. Jackson. Networks of waiting lines. *Operations Research*, 5:518–521, 1957.

[35] J. R. Jackson. Job-shop-like queueing systems. *Management Science*, 10:131–142, 1963.

[36] Nabil Kahale and Tom Leighton. Greedy dynamic routing on arrays. In *Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 558–566, January 1995.

[37] F. P. Kelly. Networks of queues with customers of different types. *Journal of Applied Probability*, 12:542–554, 1975.

[38] Len Kleinrock, 1994. Personal Communication.

[39] Leonard Kleinrock. *Queueing Systems Volume II: Computer Applications*. John Wiley and Sons, New York, 1976.

[40] E. Koenigsberg. Is queueing theory dead? *OMEGA International Journal of Management Science*, 19(2/3):69–78, 1991.

[41] Phillip Krueger and Miron Livny. A comparison of preemptive and non-preemptive load distributing. In *8th International Conference on Distributed Computing Systems*, pages 123–130, June 1988.

[42] Thomas Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering*, 17(7):725–730, July 1991.

[43] Richard J. Larsen and Morris L. Marx. *An introduction to mathematical statistics and its applications*. Prentice Hall, Englewood Cliffs, N.J., 2nd edition, 1986.

[44] Tom Leighton. Average case analysis of greedy routing algorithms on arrays. In *2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 2–10, July 1990.

[45] Tom Leighton. Methods for message routing in parallel machines. In *24th Annual ACM Symposium on Theory of Computing*, pages 77–96, May 1992.

[46] Tom Leighton, Bruce Maggs, and Satish Rao. Universal packet routing algorithms. In *29th Annual Symposium on Foundations of Computer Science*, pages 256–69, October 1988.

[47] W. E. Leland and T. J. Ott. Load-balancing heuristics and process behavior. In *Proceedings of Performance and ACM Sigmetrics*, volume 14, pages 54–69, 1986.

[48] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[49] Nikolai Likhanov, Boris Tsybakov, and Nicolas D. Georganas. Analysis of an ATM buffer with self-similar (fractal) input traffic. In *Proceedings of the 14th Annual IEEE Infocom*, pages 985–992, 1995.

[50] Hwa-Chun Lin and C.S. Raghavendra. A state-aggregation method for analyzing dynamic load-balancing policies. In *IEEE 13th International Conference on Distributed Computing Systems*, pages 482–489, May 1993.

[51] M. Litzkow and M. Livny. Experience with the Condor distributed batch system. In *IEEE Workshop on Experimental Distributed Systems*, pages 97–101, 1990.

[52] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor - a hunter of idle workstations. In *8th International Conference on Distributed Computing Systems*, June 1988.

[53] Miron Livny and Myron Melman. Load balancing in homogeneous broadcast distributed systems. In *ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[54] Dejan S. Milojicic. *Load Distribution: Implementation for the Mach Microkernel.* PhD Dissertation, University of Kaiserslautern, 1993.

[55] Ravi Mirchandaney, Don Towsley, and John A. Stankovic. Adaptive load sharing in heterogeneous distributed systems. *Journal of Parallel and Distributed Computing*, 9:331–346, 1990.

[56] M. Mitzenmacher. Bounds on greedy routing algorithm for array networks. In *6th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1994.

[57] W. W. Norton. *The Berkeley Interactive Statistics System, Part II BLSS Reference Manual.* 1988.

[58] David Peleg and Eli Upfal. The generalized packet routing problem. *Theoretical Computer Science*, 53(2–3):281–93, 1987.

[59] M.L. Powell and B.P. Miller. Process migrations in DEMOS/MP. In *ACM-SIGOPS 6th ACM Symposium on Operating Systems Principles*, pages 110–119, November 1983.

[60] Robert Prouty, 1996. Personal Communication.

[61] Spiridon Pulidas, Don Towsley, and John A. Stankovic. Imbedding gradient estimators in load balancing algorithms. In *8th International Conference on Distributed Computing Systems*, pages 482–490, June 1988.

[62] Martin I. Reiman and Burton Simon. Light traffic limits of sojourn time distributions in markovian queueing networks. *Communications in Statistics - Stochastic Models*, 4(2):191–233, 1988.

[63] Martin I. Reiman and Burton Simon. Open queueing systems in light traffic. *Mathematics of Operations Research*, 14(1):26–59, 1989.

[64] Rhonda Righter and J. George Shanthikumar. Extremal properties of the fifo discipline in queueing networks. *Journal of Applied Probability*, 29:967–978, 1992.

[65] C. Gary Rommel. The probability of load balancing success in a homogeneous network. *IEEE Transactions on Software Engineering*, 17:922–933, 1991.

[66] Robert F. Rosin. Determining a computing center environment. *Communications of the ACM*, 8(7), 1965.

[67] Sheldon M. Ross. Average delay in queues with non-stationary Poisson arrivals. *Journal of Applied Probability*, 15:602–609, 1978.

[68] Sheldon M. Ross. *Stochastic Processes*. John Wiley and Sons, New York, 1983.

[69] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, Inc., Boston, 1989.

[70] J. George Shanthikumar, 1996. Personal Communication.

[71] A. Silberschatz, J.L. Peterson, and P.B. Galvin. *Operating System Concepts, 4th Edition*. Addison-Wesley, Reading, MA, 1994.

[72] George D. Stamoulis and John N. Tsitsiklis. The efficiency of greedy routing in hypercubes and butterflies. *IEEE Transactions on Communications*, 42(11):3051–3061, November 1994.

[73] Anders Svensson. History, an intelligent load sharing filter. In *IEEE 10th International Conference on Distributed Computing Systems*, pages 546–553, 1990.

[74] A.S. Tanenbaum, R. van Renesse adn H. van Staveren, and G.J. Sharp. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, pages 336–346, December 1990.

[75] Marvin M. Theimer, Keith A. Lantz, and David R Cheriton. Preemptable remote execution facilities for the V-System. In *ACM-SIGOPS 10th ACM Symposium on Operating Systems Principles*, pages 2–12, December 1985.

[76] G. Thiel. Locus operating system, a transparent system. *Computer Communications*, 14(6):336–346, 1991.

[77] Eli Upfal. Efficient schemes for parallel communication. *Journal of the ACM*, 31:507–517, 1984.

[78] Amin Vahdat, 1995. Personal Communication.

[79] Amin M. Vahdat, Douglas P. Ghormley, and Thomas E. Anderson. Efficient, portable, and robust extension of operating system functionality. Technical Report UCB//CSD-94-842, University of California, Berkeley, 1994.

[80] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–61, May 1982.

[81] L. G. Valiant. Optimality of a two-phase strategy for routing in interconnection networks. *IEEE Transactions on Computers*, C-32(9):861–3, September 1983.

[82] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication. In *13th Annual ACM Symposium on Theory of Computing*, pages 263–277, May 1981.

[83] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating System Design and Implementation*, pages 1–11, November 1994.

[84] Jean Walrand. *Introduction to Queueing Networks*. Prentice Hall, New Jersey, 1989.

[85] Jean Walrand, 1994. Personal Communication.

[86] J. Wang, S. Zhou, K.Ahmed, and W. Long. LSBATCH: A distributed load sharing batch system. Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto, April 1993.

[87] Yung-Terng Wang and Robert J.T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, c-94(3):204–217, March 1985.

[88] Ward Whitt. The effect of variability in the $GI/G/s$ queue. *Journal of Applied Probability*, 17(4):1062–1071, 1980.

[89] Ward Whitt. Comparison conjectures about the $M/G/s$ queue. *Operations Research Letters*, 2(5):203–209, December 1983.

[90] Ward Whitt. Minimizing delays in the $GI/G/1$ queue. *Operations Research*, 32(1):41–51, 1984.

[91] R. W. Wolff. The effect of service time regularity on system performance. In K. M. Chandy and M. Reiser, editors, *International Symposium on Computer Performance Modeling, Measurement, and Evaluation*, pages 297–304, Amsterdam, 1977.

[92] R. W. Wolff. *Stochastic Modeling and the Theory of Queues*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[93] E. R. Zayas. Attacking the process migration bottleneck. In *ACM-SIGOPS 11th ACM Symposium on Operating Systems Principles*, pages 13–24, 1987.

[94] Yongbing Zhang, Katsuya Hakozaki, Hisao Kameda, and Kentaro Shimizu. A performance comparison of adaptive and static load balancing in heterogeneous distributed systems. In *Proceedings of the 28th Annual Simulation Symposium*, pages 332–340, April 1995.

[95] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: a load-sharing facility for large heterogeneous distributed computing systems. *Software – Practice and Expeience*, 23(2):1305–1336, December 1993.

[96] Songnian Zhou. *Performance studies for dynamic load balancing in distributed systems.* PhD Dissertation, University of California, Berkeley, 1987.

[97] Songnian Zhou and Domenico Ferrari. A measurement study of load balancing performance. In *IEEE 7th International Conference on Distributed Computing Systems*, pages 490–497, October 1987.

[98] Avi Ziv and Jehoshua Bruck. Upper bound on the average delay in arbitrary interconnection networks. Technical Report RJ 9069 (80759), IBM Research Report, October 1992.

Appendix A

Analysis of queueing networks of type $\mathcal{N}_{E,FCFS}$

In this appendix we describe a subset of the known queueing theory formulas for queueing networks of type $\mathcal{N}_{E,FCFS}$. More general versions of these formulas appear in [13, pp. 322-327]; we include here only what we need for our definition of a queueing network (see Figure 2.2).

Given a queueing network $\mathcal{N}_{E,FCFS}$ with a routing scheme, we associate with each packet a class l corresponding to its associated route. Let $r_i^{(l)}$ denote the arrival rate of packets of class l from outside the network into server i. Let r_i denote the arrival rate of packets of any class from outside the network into server i, i.e.,

$$r_i = \sum_l r_i^{(l)}.$$

Let $\hat{\lambda}_i^{(l)}$ denote the total arrival rate of packets of class l into server i. This includes arrivals from outside and inside the network. Likewise, $\hat{\lambda}_i$ denotes the total arrival rate of packets into server i, i.e.,

$$\hat{\lambda}_i = \sum_l \hat{\lambda}_i^{(l)}.$$

Lastly, let $p_{ij}^{(l)}$ be the probability (or rather, proportion of times) that a packet of class l after serving at server i next moves to server j.

Traditionally, one starts by determining $\hat{\lambda}_i^{(l)}$ by solving the following system of

balance equations:

$$\hat{\lambda}_i^{(l)} = r_i^{(l)} + \sum_j p_{ji}^{(l)} \hat{\lambda}_j^{(l)}$$

It turns out that in our formulation of a queueing network, there is no need to bother setting up and solving these balancing equations. The simple solution below works:

$$\hat{\lambda}_i^{(l)} = (\text{Num. times server } i \text{ appears in route } l) \cdot (\text{Outside arrival rate of route } l \text{ packets})$$

Next, we obtain $\hat{\lambda}_i$ by summing over all classes:

$$\hat{\lambda}_i = \sum_l \hat{\lambda}_i^{(l)}.$$

Now define the utilization at server i, ρ_i to be

$$\rho_i = \frac{\hat{\lambda}_i}{\mu_i},$$

where μ_i is the service rate at server i. Then

$$\mathbf{E}\{N_i\} = \mathbf{E}\{\text{Number of packets at server } i\} = \frac{\rho_i}{1-\rho_i},$$

where the number of packets at server i includes the possible packet serving there currently.

$$\mathbf{E}\{\mathrm{N}\} = \mathbf{E}\{\text{Total number of packets in network}\} = \sum_i \mathbf{E}\{N_i\}$$

More specifically,

$$P_i(n_i) = \mathbf{Pr}\{n_i \text{ packets at server } i\} = \rho_i^{n_i}(1-\rho_i)$$

And

$$P(n_1, n_2, \ldots, n_k) = \mathbf{Pr}\left\{\begin{array}{c} n_1 \text{ packets at server 1} \\ n_2 \text{ packets at server 2} \\ \vdots \\ n_k \text{ packets at server } k \end{array}\right\} = \prod_i P_i(n_i)$$

The above statement is often referred to as queueing theory's independence assumptions, because it states that the number of packets at the different servers are independent (note that this statement is not actually an assumption, but rather a consequence of the definition of $\mathcal{N}_{E,FCFS}$). Given the expected number of packets in the network, we can now apply Little's formula to obtain the expected packet flow time:

$$\mathbf{E}\{\mathrm{F}\} = \mathbf{E}\{\text{Flow Time}\} = \frac{\mathbf{E}\{\mathrm{N}\}}{\sum_i r_i}$$

And finally,

$$\mathbf{E}\{\text{Delay}\} = \mathbf{E}\{\text{Flow Time}\} - \mathbf{E}\{\text{Service Time}\},$$

where $\mathbf{E}\{\text{Service Time}\}$ is the weighted average over the route frequencies of the service requirement for each route.