Adaptive, Nonlinear, Resource-Distribution Control

by

James Grosvenor Garnett

B.S. University of Colorado, 1988

M.S. University of Colorado, 1999

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

College of Engineering

2004

This thesis entitled:
Adaptive, Nonlinear, Resource-Distribution Control
written by James Grosvenor Garnett
has been approved for the College of Engineering

____

Elizabeth Bradley

____

Richard Osborne

Date ____________________

Garnett, James Grosvenor (PhD, Computer Science)

Adaptive, Nonlinear, Resource-Distribution Control

Thesis directed by Professor Elizabeth Bradley

Control systems for software resources, such as memory buffers or CPU cycles, are difficult to design due to the bursty nature of the demands for those resources, nonlinear effects that result from adjusting the control variables, and the unpredictable saturation dynamics that result when the resource under control is depleted. Variables derived from gaussian statistics, such as the average of the number of resources in use, are easy to compute, can be used to smooth burstiness, and facilitate controller stability, but may not be representative of the true system state. Abandoning statistics in favor of unaggregated information about the system dynamics becomes critical in resource starvation conditions, wherein minute changes in the operating environment can result in abrupt system failures.

This research describes an adaptive, nonlinear, model-reference software control algorithm in which the variable to be controlled is the full distribution of resource states. In this algorithm the plant is the resource, modeled by a Markov Chain, and the reference is an arbitrary (user-chosen) specification distribution. The state-transition probabilities of this model are estimated on-line from resource demand rates using linear filters, and the estimates are used to adapt the plant behavior to changing operating conditions. A nonlinear Proportional/Integral/Derivative (PID) control scheme is then used to regulate and reduce demands for resources, thereby shaping the stationary distribution of the resource usage to match the specification.

Demonstrations of this *resource-distribution control* paradigm have proven that it is capable of improving the stability and security of existing software systems, and that the method has low computational and memory overhead. In one example, a TCP/IP network router that previously failed under the load of a simulated Internet Denial of Service (DoS) attack was retrofitted with the controller and subsequently was able to block the attack while simultaneously passing valid Intranet traffic.

Dedication

In memory of my father, Grosvenor Howard Garnett

Acknowledgements

My sincere thanks go to my research advisor, Professor Elizabeth Bradley, who never hesitated to offer her time, funding, advice, and editorial skills. Under her guidance, I learned to appreciate the value and pleasure of careful science and good writing. Of Professor Bradley's diverse research group I particularly thank Natalie Ross, with whom I shared research space, many amusing conversations, and several years of the bureaucracy of graduate school.

My extended circle of research acquaintances helped me on countless occasions. Sean McCreary, Brad Huntting, and David Wilson were particularly helpful as I struggled with issues in network communications, applied mathematics, and stochastic processes, respectively. With these people I must include Lynda McGinley, who for years provided me with a job, an office, computing cycles, and above all friendship.

Research can be time-consuming and difficult. My mountaineering partners have repeatedly helped me to rediscover my center during these years, especially Marga Powell, Todd Harris, Christa Cline, Tom Wilson, Tonya Riggs, Jason Hill, David Wilson, and Bruce Immele. Thanks to you all, and to the rock, I have learned the fundamentals of concentration.

My most heartfelt thanks of all go to my family, including Elizabeth, Anne, Milvi, and G.H., for their uncompromising support.

Contents

Tables

Figures

Chapter 1

Introduction

Resource management is a fundamental and critical task in every nontrivial software system. Service managers must present consistent interfaces to underlying resources and control access to them according to desired policies. In current practice, however, this involves offering resources to whatever authorized source makes a request—up to an arbitrary, predetermined limit, after which resources are saturated and the system becomes unusable until some resource user willingly returns what it has reserved. This thesis addresses methods to model and control these kinds of systems under *starvation* conditions, when resources are scarce; in such conditions, the resource manager itself may be unable to reserve required resources and continue normal management duties, a situation that can lead to system instability and failure. I address resource starvation by *modeling* the dynamic state of resources, *identifying* dangerous regimes, and selectively *controlling* access to resources through service degradation. My approach extends current research in four ways: first, by using stochastic modeling as the basis for on-line control mechanisms that dynamically adapt to the state of the system; second, by advancing control-theoretic concepts typically used in physical systems as a mechanism for resource management; third, by using linear filters to estimate *probabilistic* resource system parameters to achieve model adaptation; and fourth, by combining the notions of Quality of Service and fault tolerance to provide guaranteeable degrees of system-level functionality.

The fundamental problem I address is how to reduce starvation side effects when resources are shared between different subsystems. An instance of such sharing can be found in the evolution of a typical large software project: initially distinct subsystems, each with its own resource pool and management interface, are combined over time in order to maintain overall simplicity as the software becomes more sophisticated. For example, the original Berkeley Unix Fast Filesystem (ffs) used very different resource pools and data structures than did its cousin the Network Filesystem (nfs) because although both are filesystem interfaces, ffs manages local magnetic disks whereas nfs handles network communications to a remote server. As Unix matured and more filesystem variations were implemented, a unified "virtual" filesystem layer was created. This Virtual Filesystem presented a single interface, a common resource management system, and a single resource pool, reducing the potential for programming errors and system overhead while increasing filesystem efficiency. This kind of abstraction and simplification (a common software engineering technique for generalizing software systems in order to make them more maintainable) is the system-level equivalent of moving from a distributed to a centralized architecture, and brings with it many of the problems associated with centralization—among other things, single points of failure when resources are shared. Subsystems can function independently when their buffer resources are separated, but one may cause the other to fail when both require the same resources. In the worst case, a potential exists for deadlock in which a buffer pool is empty but competing subsystems are blocked awaiting more resources. This situation, which can result in outright system failure, can be solved by control systems that manage access to the shared resource pool in a principled manner.

The goal of this research is to prioritize system survival and use. By *survival* I mean that rather than entering deadlock or failing altogether, resource-management systems should have the ability to back off from offering service when a load would otherwise cripple them, and thereby maintain overall control. Similiarly, by *use* I mean that this graceful degradation should be isolated to the smallest necessary subset of services (in the sense of system services that depend upon the shared resource) and should be precisely quantifiable as well, e.g. under resource demand level X, users should expect $Y\%$ of their requests to fail. This is therefore a Quality of Service (QoS) approach—but rather than reserving resources and thereby guaranteeing access

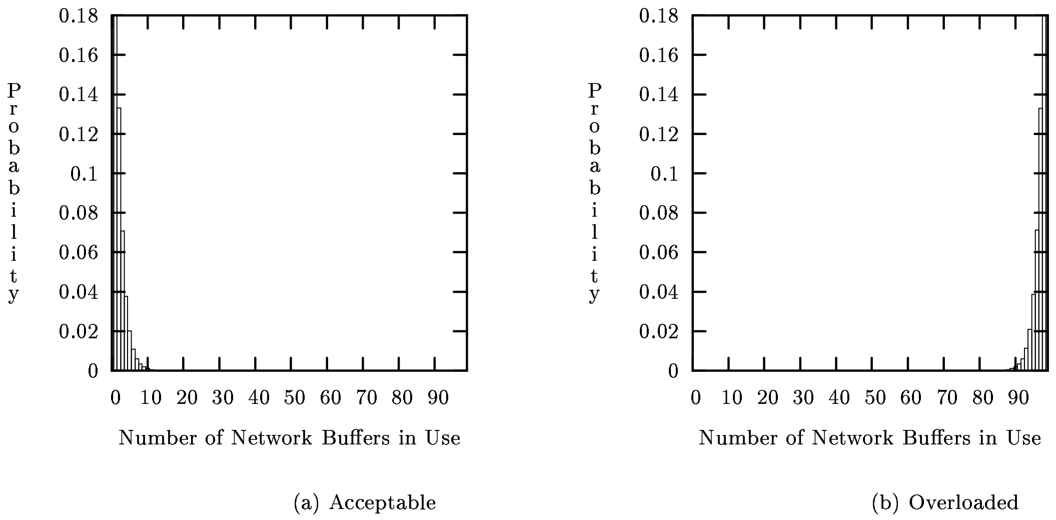


Figure 1.1: Network buffer usage-distributions simulations for a system with 100 buffers, illustrating the probability of different numbers of networks buffers being in use. Under a light network load, shown in (a), there are never more than 10 buffers in use and rarely more than 5 or 6; this is acceptable because many free buffers remain. Under a heavy load, shown in (b), there is a high probability that 90 or more buffers are in use; this constitutes an overload, and will result in broken communications.

to them, I guarantee instead that excessive resource demands will be degraded in a controlled fashion.

The QoS guarantees in this thesis work are specified at the level of a resource *usage distribution* (a measure of the percentage of time a given amount of resources is in use), and I specify them by defining explicit limits on how much time may be spent in "bad" regimes. This can be understood by way of the example of distributed denial of service (DDoS) attacks, in which attackers overwhelm a networked computer system with a coordinated stream of traffic. Each query packet in the stream temporarily consumes a memory buffer—the resource of interest in this case—until the data in the query can be processed by the victim. If the victim processes packets more slowly than they arrive, the memory buffer pool will become depleted and the result will be an "Overloaded" buffer usage-distribution of the kind shown in Figure 1.1(b). In this figure, a system with 100 network buffers is laboring, with most of those buffers in use most of the time. Once the buffer pool is completely drained of free buffers, network communications will cease—not only on the network interface bearing the brunt of the attack, but on *every* network interface. If each network interface were restricted in its buffer consumption—as specified by QoS limits that were guaranteed by a principled control system—then network communications could continue on unattacked network interfaces even when the attacked interface became congested.

Various solutions to the DDoS problem have been proposed, including:

- ingress and egress filters to block spoofed packets[1] [29, 72];
- traceback methods used to locate spoofing attackers and block their network path[88, 101];
- profiling approaches for identifying and blocking abnormal traffic[71];
- rate-limiting queueing disciplines that suppress excessive traffic[19, 27, 28, 31, 32, 55, 70, 73];
- *Explicit Congestion Notification* (ECN) schemes for communicating the existence of local congestion to upstream neighbors[30, 58, 93]; and
- software toolkits for annotating (and hence controlling) accesses to resources[92].

This list, although by no means complete, is representative of the state of the art. All of these approaches are weakened by their assumptions of specific patterns of network usage, by assumptions of cooperation between network neighbors, or by requirements for extensive software modifications. For example, filters to block packets with spoofed addresses will not block valid (but malicious) traffic; both profiling and rate limiting algorithms miss low-rate attacks; profiles must be remeasured if traffic patterns change; ECN methods assume that upstream routers are not themselves malicious; and software annotation toolkits not only add significant software complexity at the application level, but also require uncommon expertise on the part of the programmer and also destroy the portability of the resulting software by tying it to specific

[1] A packet with forged addresses is said to be *spoofed*. DDoS attacks are sometimes carried out with spoofed traffic in order to hide the source(s) and thereby extend the length of time over which the attack is successful.

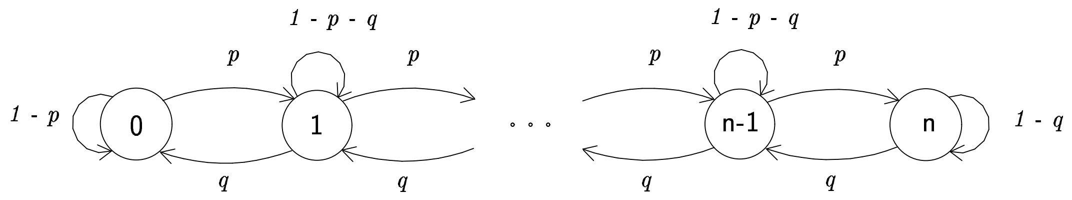


Figure 1.2: Markov Birth/Death Chain. States are numbered nodes, and the existence of of an edge to another node represents a positive probability of making the transition to that node. The value of state x represents x resource units in use.

resource management systems. Finally, most of these methods require adjustment of poorly understood tuning parameters.

In this thesis, I address all of these shortcomings by proposing a systematic QoS approach in which an engineer specifies the *desired limit distribution* through a set of target limits, and then designs a resource controller to force the actual distribution to meet those limits. The advantage of this approach is fourfold:

- It facilitates the design of robust, quantitative, adaptive controllers for resource management systems that currently possess hardcoded, sensitive, and poorly understood tuning parameters
- It handles variations in resource demands, which are normal in real systems and do not affect the overall distribution, but which can result in erratic and nonlinear behavior in current practice
- It assumes nothing about the cause of the resource starvation (e.g. the type, rate, or pattern of traffic in a DDoS attack)
- It allows the controlled system to remain autonomous by removing any dependence upon cooperation with external agents.

To achieve this type of control, I model the target system using a stochastic process, a *Markov Birth/Death Chain*, and use that model to design a nonlinear, adaptive, PID controller for the target. Such a *process model* is a natural choice for this domain, since probability distributions are fundamental to stochastic models, in contrast to functional models. The Markov Birth/Death Chain model, which I describe in detail in Chapter 3 of this dissertation, is used both to determine the theoretical distribution of the model and system, and as a reference by which to control the system. In a Markov Chain, state n represents n resources in use, e.g. the number of buffers, virtual pages, process table slots, etc., as in Figure 1.2. The state transitions for each vertex in this chain are annotated with the probabilities of the associated event, which reflect both the incoming resource demand distribution (resource consumption) and the service rate distribution (resource releases). The Birth/Death topology, so-called because transitions occur in unit increments (a "birth" increases the state number, and a "death" decreases it), is a specific form of Markov Chain in which transitions can occur only between neighboring states. This model is applicable to an enormous variety of resource management systems, and is a traditional tool of performance analysis and Queueing Theory, where it is used to determine operational characteristics of some system under a set of fixed assumptions. Used in this manner, traditional Markov models assume fixed transition probabilities between states, from which it is possible to calculate the chain's **stationary probability distribution**. This distribution represents the fraction of time that the system spends in any state. This is a simple one-variable functional relationship: for each state, there is a fixed fraction of time that the Chain will spend in that state over the long run. In computer performance analysis, such probability distributions are used to calculate the average, or *expected*, state of the modeled system. This single value is then used as a guideline by which resource limits or pool sizes are set.

In my approach, I use the Markov Chain not only as an analysis tool by which to understand the resource system, but also as one component in the synthesis of a controller for it. The Markov Chain serves as the *reference model* of the resource system to be controlled: a system with a fixed structure but variable state-transition probabilities. The controller manipulates these edge transition probabilities to change the shape of the Chain's theoretical distribution and thereby to force the resource system to meet the QoS requirements. I achieve distribution control by dropping resource requests, with probability computed by a two-stage, closed-loop

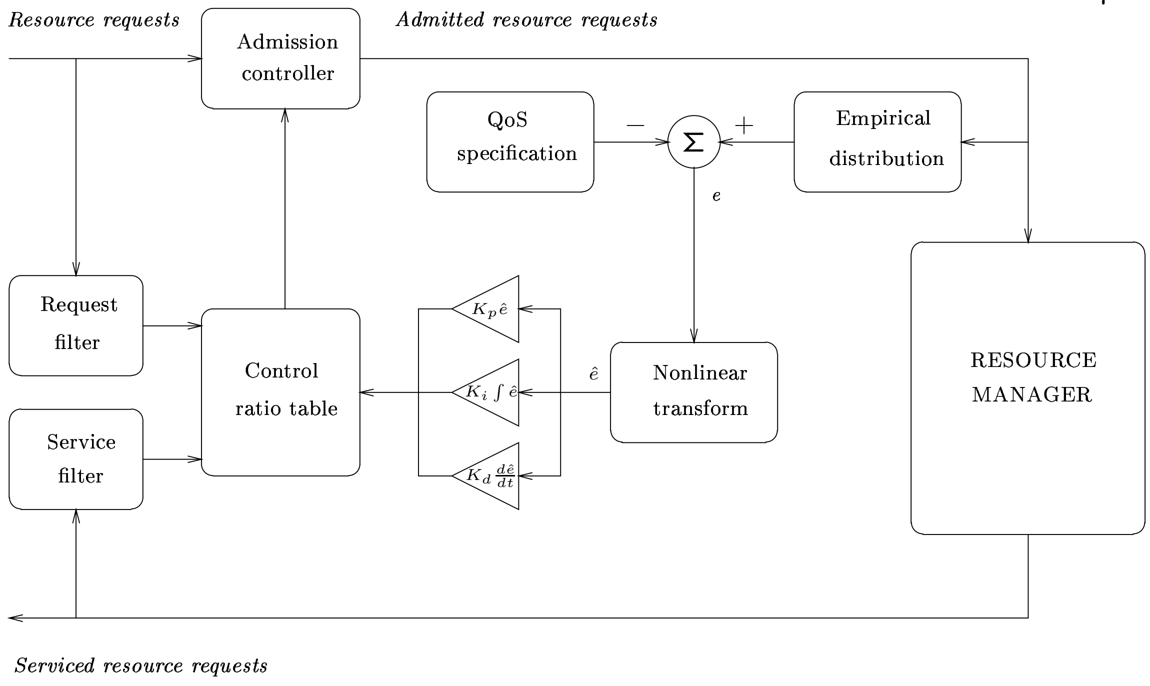


Figure 1.3: Nonlinear adaptive controller. For a fixed rate of resource requests, the admission controller can exactly guarantee the QoS specification using the ratio table. A PID feedback loop draws the empirical distribution towards the QoS specification under varying rates of request arrivals.

feedback control system (Figure 1.3). This two-part system uses an admission stage as an actuator, and a PID feedback stage to manipulate that actuator by computing when resource requests should be dropped in order to meet the QoS specification.

The decision to drop a request is made by way of a *control ratio table*. This table, indexed by current state, lists ratios of admitted requests to serviced requests that must be maintained in order to satisfy the QoS specification. For example, if the control ratio for the current state were 0.5, then half as many requests should be admitted than serviced. With the correct control ratio table, the admission state can shape the distribution of resource states to conform to the QoS specification. The fundamental insight that makes this possible is that the ratio of input resource requests to serviced resource requests is equal to the ratio of the stationary probabilities of adjacent states in the Birth/Death Chain. Hence, by precalculating these ratios from the QoS specification (using algorithms described in Chapter 4), measuring the input and service rates (via two on-line parameter estimators called the *request filter* and *service filter*, respectively, which are covered in Section 3.3), and then admitting only enough requests to maintain the appropriate ratio, the controller can guarantee that the stationary probability of each state in the QoS specification is correct *relative to its neighbors*. That is, the PID feedback loop guarantees the QoS-specified level of the first state in the specification, and therefore of all the remaining states, since they are neighbors. This second stage of the controller measures the difference between the specification (the control reference) and the empirically measured distribution (the system output) and generates an error output, e. It is possible to use this error to directly manipulate the ratio table in order to ensure conformance with the QoS specification.

However, as will be shown in Chapter 4, a resource system's empirical distribution can enter *quasi-stable* operating regimes, in which that distribution converges rapidly to a nonstationary steady state and only slowly evolves towards its stationary limit. This effect is nondeterministic: the distribution may be quasi-stable in a given situation but avoid quasi-stability in another, under identical operating conditions. To compensate for this, the controller must be able to react nondeterministically itself. An attractive approach to this, outlined in Section 4.4.2, is to precondition the proportional error e by a nonlinear transform that allows the controller to react nonlinearly, rather than in strict proportion to e. This transform is intended to capture the fact that a controller that is "stuck" in a quasi-stable operating regime may require a nudge in the form of a temporary boost in e in order to increase its rate of response. The result, $\hat{e}$, is then passed to the PID controller, in which access to the resource is managed according to the value of $\hat{e}$, its rate of change $d\hat{e}/dt$, and accumulated error over time, $\int \hat{e}dt$. As shown in the middle of the block diagram in Figure 1.3, each of these three terms is weighted by a constant (K_p, K_i, or K_d) that is chosen—by methods described in Chapter 3—to achieve the desired control response. These control law weights determine the speed and stability of the response, and can be selected using well-known heuristic or analytical methods to match the desired characteristics for a particular controller.

In Chapter 5, the combination of modeling, estimation, and control methods that forms the technical contribution of this thesis is demonstrated on an important example: *Denial of Service attacks* (DoS), in which external agents drain a system of resources and bring about its failure. In the same chapter, I show how the methods developed here may be applied to *virtual memory page allocation* problems in order to mitigate memory competition, which can cripple a system. These applications, although apparently quite specific, reappear in an extraordinarily wide array of forms and are among the most pressing security issues in systems engineering. Solutions enabled by this research are therefore widely applicable to computer science problems, including components of operating systems, (e.g. protocol stacks, kernel tables), at the application layer (e.g. servers for email, DNS, NFS, WWW, Directory (LDAP), databases, etc.), and in programming languages (e.g., garbage collection). Moreover, these results apply equally well in areas outside of software engineering, such as telephony, railways, smart traffic systems, and flight and spacecraft systems. In general, the approach in this thesis can be used in any environment where it is important to preserve critical functions, and where such preservation can be achieved through a controlled sacrifice of ancillary subsystems. As a result, data overloads that would otherwise result in a catastrophic failure become no more than a nuisance. The method described in the following chapters, then, makes information systems more robust, powerful, and fault-tolerant. As I will show, this approach is lightweight, intuitive, and widely applicable.

Chapter 2

Related Work

Work related to this thesis spans a variety of disparate fields. In this section, I first describe work related to the underlying modeling paradigm: Quality of Service, Markov Decision Processes, and Markov Chain Monte Carlo methods. An important focus of this thesis is the problem of finding an effective deterrent to DoS attacks, and so I provide more detail on this problem and its current solutions. This leads to an introduction to the current state of the control of software systems, and a dicussion of adaptive filtering.

2.1 Quality of Service

Quality of Service (QoS) is a general term covering research into methods for providing quantitative guarantees of some kind of service. These guarantees are typically specified in terms of performance levels—e.g., a minimum amount of bandwidth to be provided, or a maximum end-to-end delay—and are typically satisfied by requiring resource reservation in advance of needs. QoS approaches are common in *connection-oriented* applications such as Asynchronous Transfer Mode (ATM)[91], in which connections are requested, negotiated, established, and then eventually broken. During the negotiation phase, a client requests that servers in the connection path reserve resources to support a particular quality of service; if the resources cannot be reserved, the negotiation is terminated and the entire connection is broken. QoS-capable ATM switches have been widely deployed for telephony applications, where the bandwidth requirements of a connection can be prespecified and resources can be reserved at each switching node in the path of a call. With the emergence of Video on Demand (VoD)[119] and Voice-over-IP (VoIP)[113] and their concomitant network performance requirements, QoS is beginning to find its way into *connectionless* networking, as exemplified by TCP/IP. Since TCP/IP does not explicitly guarantee the availability of networking resources, the free bandwidth over the network links in a TCP/IP flow[1] may change erratically, leading to buffered (and hence delayed) packets at intermediate nodes and network link congestion that violates QoS guarantees. Hence research in this area centers upon *congestion avoidance* to provide the performance of QoS-capable connections over end-to-end TCP/IP flows, of which Explicit Congestion Notification (ECN)[30, 58, 93] is widely accepted as the "best" approach. ECN research overlaps significantly with denial of service and software control systems, so a description of ECN is deferred to section 2.3.1.

QoS is used in this thesis in its usual sense (i.e., guaranteeing service levels) but in a somewhat nontraditional fashion. The difference is twofold. First, the guarantees are probabilistic rather than absolute: the service levels are specified upon the distribution of resource states, and hence can be guaranteed absolutely in the asymptotic limit, but only probabilistically over a finite period of time. Second, these guarantees are specified in terms of how service is to be *degraded* rather than how it should be *provided*; the intent of this is to develop a principled, generalized approach to handling overload conditions. Previously, probabilistic QoS and service degradation have been used in frameworks designed for a single type of resource, and thus cannot be easily translated to a different system. Examples include video streams transmitted over IP that use probabilistic bandwidth guarantees to handling jitter in end-to-end transmission delays[96]; soft deadlines in realtime schedulers[2]; and lossy connection setup between ATM switches[95]. These solutions are all *ad hoc*, system-specific, and do not generalize, whereas the use of QoS and controlled service degradation, as described in this thesis, applies equally well to all of these examples.

[1] A *flow* is a high-level connection between two network nodes that may involve any number of intermediate nodes. For example, the transmission of a video clip over the Internet consisting of many thousands or millions of individual TCP packets constitutes a flow.

2.2 Markov Chains

Markov Chains are widely used and covered in numerous texts. Ross[100] and Norris[84] cover the general introductory mathematics, and Nelson[83] describes Markov Chains within the context of Queueing Theory and other computer science applications. The specific uses of Markov Chains that are most relevant to this thesis are *Markov Decision Processes* and the *Metropolis-Hastings algorithm*, along with associated convergence results.

2.2.1 Markov Decision Processes

Markov decision processes (MDPs)—also called *Controlled Markov Chains*, *stochastic dynamic programs*, or *Markov control processes*—are Markov Chain models that are used to develop general strategies that maximize the average performance of some system[20, 24, 41, 51, 98]. In a MDP, a *control agent* observes the state of the process at discrete time intervals and then performs a control action based upon that observation. The result is a reward (or, equivalently, a cost) that affects the next state transition in the model. The set of control actions taken by the agent over a set of n steps is called the *control strategy*, and the goal is to compute a strategy that will maximize the average rewards (or minimize the average costs). More formally, if action a taken in state i results in a reward $R(i, a)$ and a probability $P_{ij}(a)$ of moving into state j at the next transition, then the average reward over n steps, starting from state i, is denoted by $V_n(i)$ and is given by the recurrence equation

$$V_n(i) = \max_a \left[R(i, a) + \sum_j P_{ij}(a) V_{n-1}(j) \right] \tag{2.1}$$

which is the sum of the single-step reward in the current state plus the expectation of the rewards in the following $n - 1$ states. This equation represents an optimization task that can be solved by dynamic programming (hence the name *stochastic dynamic programming*). Its solution is a general control strategy that will maximize the average rewards over many different sample paths of the process that underlies the Markov Chain, as opposed to maximizing the reward in a single, specific path. The MDP method therefore adjusts the stationary distribution[2] of a Markov Chain such that the mean of that distribution is optimal in some desired sense. For this reason, the MDP approach cannot used to control the distribution of a *running* Markov Chain model (or a system that it models) in an instantaneous manner. The methods of this thesis, in contrast, are designed to facilitate exactly that kind of distribution control. Applications of MDPs range from wildlife population management (setting appropriate yearly harvest restrictions) to simple aspects of communications-network Quality of Service (setting conditions that must be met to guarantee QoS)[24, 117].

2.2.2 The Metropolis-Hastings Algorithm

Markov Chain Monte Carlo (MCMC) techniques are an application of Markov Chains to making efficient statistical estimates. In MCMC, one is faced with the problem of determining some statistic (e.g., an average) from the distribution of a known but highly complex system. If the statistic is too difficult to compute, one can generate a Markov Chain whose stationary distribution is the same as the system distribution, and then to sample the Markov Chain state in order to form an estimate of the desired statistic. If the Markov Chain has converged to its stationary behavior, and if enough samples are taken, then the Law of Large Numbers[3] guarantees that the estimate will asymptotically approach the correct value.

The *Metropolis-Hastings* algorithm [84, 99] is used to generate the Markov Chain for MCMC simulations. Given the probability distribution function (PDF) of the target system, Metropolis-Hastings generates a time-reversible Markov Chain whose stationary distribution is that PDF. In this thesis I take a similar approach, but rather than generating a Markov Chain with a desired stationary distribution, I develop an algorithm to modify the transition probabilities of an existing Markov Chain so as to achieve the desired stationary distribution without changing the Chain topology. Since the convergence of a Markov Chain to its stationary distribution has important bearing on the validity of the results obtained through MCMC simulations,

[2] The concept of a stationary distribution will be covered in detail in chapter 3.
[3] Also covered in chapter 3.

much effort has gone into determining bounds on Markov Chain convergence[46, 47, 49, 81]. The convergence, or *mixing* rate, is a measure of how quickly a Chain's empirical distribution approaches its stationary limit. Using the methods in this thesis, a Markov Chain can only be controlled as fast as it mixes. Hence a slowly mixing Chain requires the use of a slow, low-performance controller. Therefore it is important to show that Markov Birth/Death Chains are *rapidly mixing*, i.e. that they converge in a number of steps that is polynomially related to the number of states in the Chain. At present, only a few formal results are available about a Markov Chain's mixing rate. It is well known that the second largest eigenvalue of a Markov Chain transition matrix is related the mixing rate of the underlying Chain (see, for example, [59] for exercises illustrating this connection), implying a relationship between the topology of the Chain and its mixing properties. Mixing rates are typically assessed using the *conductance-based* approach described in [47], which uses set *capacity* (the amount of stationary probability mass in a set of states) relative to *ergodic flow* (a measure of the rate of probability mass transfer out of the set) to identify *bottlenecks*[4] and thereby convergence bounds. This technique, which relates the topology of a Chain, its transition probabilities, the second eigenvalue of its transition matrix, and its convergence rate, is now the textbook method of proving asymptotic convergence characteristics (e.g. see [82], pp. 322–323).

The conductance technique has been successfully applied to studies of the mixing rate of *expanders*, graphs that possess high vertex or edge expansion. *Vertex expansion* refers to the size of the neighborhood of any set of vertices relative to the size of the set itself, whereas *edge expansion* refers to the size of the set of edges between a set of vertices and its complement relative to the size of that set. In a Markov Chain with good vertex expansion, each state possesses many neighbors; in one with good edge expansion, each state has a high probability of transitioning to a neighbor state. Various authors have obtained mixing bounds on expanders. [46] and [105] showed that mixing rates are a function of the transition probabilities through the bottleneck sections of the Markov Chain—i.e. that graphs with good edge expansion are

[4] A bottleneck is a section of the Chain in which there are few or only low-probability transitions.

rapidly mixing. [60] showed that mixing improves in graphs in which small subsets possess high expansion, and [81] proved even faster mixing if the graph possesses both good vertex and edge expansion. In the general case—i.e. when a graph is not an expander—the exact relationship between Chain conductance and convergence rate is unknown.

In this thesis, I am not interested in the general case, but rather in the convergence of the *Birth/Death* form of Markov Chain. The Birth/Death Chain topology is significant in that it appears repeatedly in real-world applications, and some mixing rate results have been obtained for various restricted Birth/Death Chains[49]. In all cases, these results apply to Chains with fixed topologies and transition probabilities—as opposed to the models of this thesis, which possess dynamically modified transitions. Further work in identifying convergence characteristics of Markov Birth/Death Chains is therefore needed before any formal results can be justified. Although I present no proofs, I show through numerical simulations in Chapter 4 that Birth/Death Chains with dynamic transitions appear to be rapidly mixing under most circumstances, but with quasi-stable characteristics that necessitate the use of special control techniques (viz., nonlinearity).

2.3 Denial of Service

Denial of Service (DoS) is defined as an attack in which the goal is to deny the victim(s) access to a resource[43]. This is a broad definition that could apply as well to post office boxes as it could to computing systems; in this thesis, the focus is upon distributed, network-based DoS against victims over TCP/IP networks, called Network DoS in simple cases and Distributed DoS (DDoS) when there are multiple attackers.

Since 1989, incidences of Network DoS and DDos have been increasing rapidly in both extent and seriousness. Early research indicated that reported DoS attacks increased 50% each year between 1989 and 1995[44], and the *rate* of attacks itself now appears to be increasing rapidly[80]. Motivations for the attacks appear to be changing as well. Early attacks were probably the result of personal animosities or relatively innocent hacking attempts, but DoS attacks have since become a mechanism for disrupting commercial online transactions[3, 69, 75], retaliation against large corporations[14, 54, 110], silencing media sources[103], and even extortion by organized criminals[85, 102]. These facts indicate a growing need for effective deterrents, but solutions are complicated by the manifold variety of DoS methods.

DoS attacks can be successfully carried out using everything from weaknesses in the network protocols to brute-force traffic floods to cleverly-designed slow-rate approaches that are extremely difficult to detect. Some of these attacks are possible due to a failure on the part of the original designers to perform bounds checking, leading to overflow crashes; the *Ping of Death*[5] is one example. These types of attacks can be completely blocked by patching the coding problem. Most fixes are not this simple, since attacks can exploit the fact that *some* level of service must be provided to anonymous clients. For example, in order to establish a TCP connection between two network hosts, a so-called *three-way handshake is used.* In this handshake, the client starts by sending a connection-setup packet—called a *synchronization* or SYN packet—to the server. When the server receives the SYN packet, it reserves the resources necessary to establish and maintain the connection, and then waits for the client to acknowledge and thereby complete the connection setup. In the *TCP SYN* attack, the attacker(s) stop communicating after the initial SYN, leading to half-open TCP connections on the victim, protocol data structure starvation, loss of network communications, and possible system crashes[13]. Two ways of dealing with this problem are a *SYN cache* and *SYN cookies.* In a SYN cache, only minimal state information is set up at the beginning of the three-way TCP handshake, with full resource allocation occurring only at the completion of the handshake[56]. When using SYN cookies, half-open connections that are consuming kernel resources are dropped when the connection table fills and a new connection request appears; the *cookie* is a specially chosen identifier that allows the machine to reconstruct the vital components of a connection if it was not part of the SYN flood and hence dropped improperly[10]. The SYN cache alone is not an effective solution because it merely

[5] The Ping of Death involved sending a huge ICMP Echo Request (or *ping*)[12] that was fragmented at the sender and reassembled at the victim. The fragmented pieces were all smaller than the maximum IP packet size, but the reassembled fragments overflowed the victim's network buffer, causing a system crash.

pushes the allocation to a another level: the SYN cache table that maintains the "minimal" connection information is just a static table that can be attacked by methods similar to the basic SYN flood itself, thereby precluding the establishment of valid connections. SYN cookies, on the other hand, lose the TCP options information that is established during a normal three-way handshake. Hence neither solution is wholly adequate to prevent flooding attacks against the TCP connection table, though some further benefit has been reported from using *both* approaches: a first-level SYN cache followed by SYN cookies when the cache fills[56]. The methods introduced in this thesis are more effective at managing the TCP connection table than SYN caches, and they complement the use of SYN cookies.

Although the effectiveness of SYN-flood-type attacks stems from the anonymity of the attackers, requiring authorization is not an effective solution. Another DoS threat is the so-called authorization/authentication attack, which derives its name from the computational overhead associated with using strong authentication. In this attack, an anonymous attacker repeatedly attempts to authenticate itself to a victim in order to use some service, resulting in many wasted computational cycles. The attack can take many forms, including straightforward login authentication to repeated remote procedure call attempts[112]. If enough requests are initiated, the victim will use most or all of its CPU cycles to verify and refuse each request, thereby denying those resources to valid users. DoS attacks that exploit these weaknesses are usually fairly sophisticated, requiring expert knowledge to initiate.

Sophistication is not always necessary, however, in order to carry out a successful attack. A *DoS Flood* is a brute-force process that requires no expertise and succeeds by simply overwhelming its victim with traffic. The MyDoom virus was a DDoS attacker that infected thousands of Internet hosts, and then attacked its victims at a predetermined time[54, 110]. Flooding attacks possess a particular traffic characteristic—high rates of traffic—that facilitates their detection and subsequent prevention. For example, so-called *flash crowds*—a flood of network traffic that appears in response to some event, such as the floods to online news websites on September 11, 2001—exhibit *ramp-up*, *sustained-traffic*, and *ramp-down* phases[4, 5]. Unfortunately there exist DoS attacks that possess either a difficult-to-detect fingerprint or none at all.

Slow-rate or *stealthy* DoS attacks are a novel and emerging threat that are not easy to detect because they use very low traffic levels that can consist of completely valid traffic. Two variants are well known, both of which rely upon weaknesses in TCP protocol. The first, the so-called *shrew attack*,[6] degrades bandwidth in a TCP flow by using bursts of transmissions whose average rate is low, taking advantage of weaknesses in the TCP retransmission algorithms to incur repeated timeouts and retransmissions[53]. Shrew DoS attacks require that the attacker be a link in the TCP flow that it is attacking, and are able to reduce effective bandwidth in that flow by up to an order of magnitude with a very small amount of carefully-chosen disruptive traffic.

Another type of slow-rate DoS attack holds TCP flows open indefinitely, thereby draining the victim of the data-structure space necessary to support those connections[92]. Once a TCP connection is set up, the kernel resources required to support it must be kept in use. A victim can be crippled and left unable to communicate with other networked machines if enough connections are opened and held open, consuming all the available memory used for holding the data structures for TCP connections. This is similar to a SYN flood attack, but in this case the connection is established and validated by the victim, so SYN cookies will not solve the problem. Three basic approaches are used by DoS attackers to keep a connection open indefinitely: sending requests to the victim very slowly (e.g., sending a request that contains 2000 bytes but sending it with only one byte per packet); reading the results very slowly (e.g., requesting a web page of 2000 bytes and then reading it one byte at a time); and sending acknowledgements for each received packet very slowly, thereby incurring retransmissions from the victim. This list is by no means a complete taxonomy of DoS attacks, but it is representative of their variety and complexity. A more-complete exposition on the types of presently understood attacks can be found in [77].

[6] The shrew is a tiny animal that is able to challenge and defeat much larger opponents.

DoS attack prevention, to date, has consisted primarily of identifying the type of attack and then designing an *ad hoc* solution to block it. For example, most DoS attacks must be maintained for an extended period of time in order to be successful, so attackers might use forged, or *spoofed*, network addresses to hide the source of the attack. Spoofed packets can be blocked with ingress and egress filters[29, 72] which are used to prevent receiving (or transmitting) packets that have invalid addresses. Such filters block the attack at the victim; *traceback* methods can then be used to identify the attackers in a spoofed-address DoS attack, allowing administrators to stop the attack at the source[88, 101]. When addresses are not spoofed, (e.g., when the attacking machine has itself been hijacked in order to take part in the attack), filtering is not desirable because valid traffic from the attacker is then blocked. In these cases, brute-force attacks are often used in order to overwhelm victims quickly. Traffic-rate limiting is a standard solution in these cases, and it involves using either software annotations or active control mechanisms. Software annotations can be used to control access to a particular system resource that is vulnerable to DoS attack[92], but they are extremely invasive to the annotated software and require extensive knowledge of the software operation. Software annotations place the resource management burden on every subsystem that uses a resource, instead of centralizing management in one place. For example, under the annotation approach, the code to protect against an attack goes not in the memory allocator (a single location), but rather in the software that invokes that allocator (multiple locations). Active control mechanisms do not require this kind of extensive retooling of existing software. The D-WARD system, for example, is deployed at the network edges between security zones (e.g., between the untrusted Internet and a trusted internal network), and uses complex behavioral models of network traffic for each protocol and specific application to decide when data flows are abnormal and require rate-limiting[76, 78]. Other active control mechanisms are standard components in network congestion control, viz. Queueing Disciplines and Excplicit Congestion Notification techniques, which are described in the following section.

2.3.1 Queueing Disciplines and Explicit Congestion Notification

The problem of brute-force DoS attacks and the emergence of applications with strict bandwidth requirements, such as VoD[119] and VoIP[113], has led to intense research interest in ways to provide rate limits and QoS guarantees over TCP/IP networks. Unlike connection-oriented networks that provide delivery guarantees once a connection has been established, TCP/IP is a *best-effort* protocol suite that is vulnerable to data loss at any time. In a connection-oriented network—of which a telephone network is a good example—a fixed path is established between the two end nodes, and the quality of the connection (bandwidth, transmission delays, etc.) is established and unchanged for the life of the connection. In a best-effort network, however, each transmission is an individual piece of communication, akin to a single package sent through the postal system. Major advantages of connectionless (best-effort) over connection-based networking are the relative simplicity of its implementation and the independence it confers upon network nodes: a failed node results in broken connections in the telephone network, but merely a rerouting of traffic in the postal network. This independence of network topology means that link or node failures may change the data paths taken by individual packets in a TCP/IP flow, leading to highly variable delays and bandwidth from one transmission to the next. In addition, congestion at a node may result in erratic delays even if data travels the same network path over each transmission, which necessitates some manner of flow control management.

TCP/IP possesses a built-in congestion avoidance and control algorithm called *Slow Start*. This is the standard principle by which data in flows is injected into the network in order to avoid congestion. In this approach, a TCP sender begins by transmitting data slowly, and then gradually increases its rate until the receiver fails to acknowledge a transmission. This acknowledgement failure indicates a packet loss, which in turn signals the onset of congestion. This is essentially a *reactive* approach to handling network congestion. However, one can also *proactively* drop packets in a network flow, before congestion appears, and thereby initiate the sender's TCP flow control mechanisms in advance of congestion problems. The standard means by which to achieve this are Active Queue Management (AQM) techniques that use packet admission controllers and Queueing Disciplines.

A packet admission controller decides whether or not to accept (admit) an incoming packet for processing, and a Queueing Discipline is the strategy used to make the decision. The purpose of using an admission controller is congestion avoidance, since proper queue management can stabilize the variability in per-node delays by trading off queue utilization versus packet delay. A controller that admits most incoming packets will maximize utilization of the queue and minimize packet loss, but because those packets must traverse the length of the queue before being processed, it will also maximize the delay and minimize throughput of the flow. With careful tuning, admission controllers can be used to provide some measure of network QoS guarantees.

Queueing Disciplines have undergone a long development and are now relatively mature. The default behavior of an admission controller without AQM is *Drop-tail* (or Tail Drop) queueing: when the input queue for customers[7] becomes full, the newest (tail) arrival is dropped. Drop-tail queueing has a significant and deleterious effect on TCP throughput because a dropped packet causes senders to immediately re-enter Slow Start. Because network traffic is bursty[55], a full queue using a drop-tail policy can create a burst of dropped packets, and when the dropped packets contain segments from different TCP flows, this results in global synchronization as *every* sender falls back to Slow Start[23]. AQM mechanisms address these issues by dropping customers using more-sophisticated methods. Random Drop[45] operates by assigning proportions of queue slot fairly based upon customer classifications. Fair Queueing[19] addresses unfairnesses in Random Drop's classification scheme, and Stochastic Fairness Queueing[73] extended this development by reducing the space complexity[8] of the classifications. Random Early Detection (RED)[32] (also referred to as Random Early Discard) abandons attempts at classification and fairness and drops arrivals based upon the average length of the queue. RED quickly matured

[7] In queueing theory, a *customer* is a queue member—in the case of TCP packets arriving and queueing at a network host, a packet.

[8] *Space complexity* is an asymptotic measure of the amount of space (memory) used by an algorithm.

into Adaptive RED[27, 31] (which adapts the RED algorithm to different customer arrival rates) and Flow RED (FRED)[57], which applies RED to individual flows. BLUE[26] is a followup to RED that uses a much simpler approach to achieving adaptive behavior, and Stochastic Fair BLUE[28] is BLUE with customer classification and fairness reincorporated. Dynamic Buffer Limiting (DBL) is an approach to queue management that assumes the presence of so-called *belligerent* network peers. DBL is similar to FRED, and works by limiting the amount of the queue that can be occupied by flows that prove unresponsive to congestion notifications. In DBL, a flow that continues to deluge a victim, despite being asked not to, will be dynamically limited to a small number of queue slots; any packets that arrive when the flow has used all its available slots are dropped. DBL assumes, like the approach taken in this thesis, that remote transmitters may be malicious, but its control mechanism is essentially drop-tail queueing once a flow is marked as malicious, and performance suffers from the associated drawbacks. [36] combined some of the ideas of DBL with customer classifications and the Linux QoS manager to create a rudimentary QoS-based buffer management scheme that, although not a classical control approach, supports a tenet of this thesis, viz. that system-level protection against malicious overload is best performed at the resource management level. *Traffic Shaping* is a variant AQM scheme that uses delays instead of drops to shape customer arrival distributions. This method can be used to increase network throughput, and is possible due to the high variance in network packet arrival times. Such packet traffic tends to be self-similiar, resulting in fundamentally bursty traffic patterns[33, 55, 87, 118]. Shaping methods delay some classes of traffic in an attempt to effectively use the periods during which there are no bursts, or else attempt to prefetch the data during those same times. This increases the throughput over the communications links by reducing the overall delay[22, 37]. The approaches used in RED, BLUE, and DBL are representative of the state of the art in AQM controllers. Many variants of all of these algorithms have been developed, and RED and BLUE form the basis of a widely used queueing management software package, ALTQ[15]. Some important details of these algorithms are described below.

RED is designed to drop **any** customer arrival when the overall average queue length exceeds a preset limit. Starting from this limit (the *minimum threshold*), RED increases the probability of dropping a new arrival linearly with the queue's average length—from zero up to a maximum drop rate that is reached at a specified maximum average queue length threshold. The result is an algorithm that is effective in managing queue lengths, but that is unable to provide fair-share access to the queue, since all packets are dropped with the same probability: the few packets in a small class are just as likely to be dropped as the many packets in a large one. This is bad if the small-class connections are important (e.g. email) but the large-class ones consume the network bandwidth (e.g. a video stream), as it creates disproportionately large latencies in the small-class traffic. In addition, the parameters of RED must be carefully tuned for specific traffic scenarios. *Adaptive RED* overcomes the weakness of preset parameters by adjusting them dynamically via an *ad hoc* proportional control system, based on the current average queue length and a single preset parameter: the desired average queue length.

BLUE takes an approach similiar to RED by dropping customers without regard to class, but it maintains a single queue drop probability instead of the array of parameters needed by RED. This probability is incremented when packet loss is noted and decremented when there is none. In this sense it is somewhat akin to Adaptive RED with its single preset parameter (the desired average queue length). BLUE is effective because although persistently long average queue lengths are *possible* indicators of congestion, packet loss in a network is a *definite* sign of it. In addition, no human tuning of parameters is necessary as with RED, and automatic adjustment of the single, intuitive drop probability of BLUE is easier than the multiple parameters of RED. The data loss created by AQM methods like RED or BLUE, which use packet drops to signal imminent network congestion, results in retransmissions that can exacerbate congestion. The solution to this problem is to use explicit notifications instead of packet drops to signal congestion to upstream members of a network flow.

Explicit Congestion Notification (ECN) attempts to achieve this by assigning new meaning to existing bit fields in IP packets and using these bits as congestion-indication flags[93].

When a network node detects imminent local congestion (as measured by the length of its own packet queues), it can signal its neighbors to preemptively initiate their flow control mechanisms. This approach obviously relies upon cooperation between network neighbors as well as modifications to existing TCP/IP implementations. ECN performs well in cooperative network environments, and is in the process of largely replacing packet drops as the congestion-signalling scheme of choice[94]. Because of its requirement for node-to-node cooperation, however, ECN cannot be used in the types of malicious environments addressed in this thesis since malicious intent, by definition, precludes effective cooperation.

All of these AQM methods are *ad hoc*, linear, proportional closed-loop feedback schemes. They have two weaknesses with respect to DoS prevention. First, the variables to be controlled are usually gaussian statistics, such as the average queue length. This is necessary to obtain a measure of stability in the face of the enormous variance present in network traffic queue lengths, but creates the danger that the average queue length might not be related to the actual congestion state. For example, if the traffic characteristics exhibit a bimodal distribution of delays (i.e., some bursts of packets arrive with low delay, and some with high delay), then the average queue length may be low during periods of high congestion. This fact can be exploited by a slow-rate DoS attack. Second, these AQM algorithms were not designed using a formal control theoretic approach, and the result is instability. RED and Adaptive RED, for example, are known to create queue-length oscillations under certain loads[31], a problem that translates directly into inconsistent service delays and DoS vulnerabilities. These oscillations are a side effect of using slow proportional control, as explained in Chapter 3. In RED, this problem has been addressed by the application of classical control theory to improve the speed of the RED control response[42] using a fluid-flow model of TCP[79]. Simulation results in [42] show that significant improvements in RED's control response can be achieved with even a simple controller of the type described in Section 3.2. In contrast to these AQM approaches, the methods in this thesis are designed to use more sophisticated control approaches, do not depend upon aggregate statistics as control variables, are applicable to a wide variety of resource management problems, and incorporate nonlinearities in order to achieve considerable performance improvements over linear AQM methods.

2.4 Controlled Software Systems

The body of literature on principled control-theoretic methods—those that use transfer-function-based or PID-based design and analysis approaches—is vast. Linear control is covered in many texts, of which [38] and [89] (general theory) and [7] (PID theory) are personal favorites. PID control, in particular, has been studied extensively due to its prevalence in manufacturing processes; it is estimated that over 90% of control loops are of the PID type[7]. *Ad hoc* controllers for software systems are widespread, and some applications of control-theoretic techniques to software dynamics are emerging. In the field of real-time systems, for example, the Feedback-Control Earliest Deadline First (FC-EDF) algorithm is the control-theoretic successor to the Earliest Deadline First (EDF) scheduler. EDF is a well-established control approach that prioritizes jobs according to their worst-case deadlines. Classical hard-real-time scheduling approaches such as EDF are *open-loop* approaches that, when provided with complete knowledge of the tasks to be scheduled and the time in which to schedule them, can compute an optimal schedule. Without such *a priori* knowledge, hard scheduling guarantees cannot be made and performance may suffer as a consequence. In this case, one may "soften" the guarantees by allowing some jobs to miss their deadlines, such as in the probabilistic mean-time scheduling approach of [104]. In such *soft real-time* schedulers, feedback can be incorporated to optimize the schedules based on discrete-time instantaneous peformance. The feedback approach in [8], for example, provides for jobs that miss their deadlines to send a notification back to the scheduler. This notification can then be used to increase the jobs' priorities in the next scheduling epoch. This simple discrete-time approach, which is similar to that taken by both Explicit Congestion Notification (to induce fast network peers to slow down their communications) as well as the BLUE algorithm (to steer the average length of network queues towards a desired value), works well in practice. However, it is a heuristic method that is neither amenable to formal analyses through time- or frequency-domain transforms—in contrast to linear transform-function-based control theoretic approaches—nor supported by a broad base of operational experience and research (in contrast to PID approaches). In addition, the time quanta used can have a significant impact on the control system response. As a result, the performance of algorithms based on this approach can only be characterized through individual, empirical studies, and no provable statements can be made about the behavior the control systems. The formal methods of control theory, in contrast, facilitate both performance analysis as well as principled control system design. The FC-EDF algorithm, for example, was designed as a PID control system whose model parameters were chosen after so-called *BIBO* (Bounded Input/Bounded Output) stability analyses, a standard control-theoretic design procedure. The result is a control algorithm that optimizes performance metrics such as the scheduler miss-ratio[9] [64, 65], while simultaneously being provably stable. Studies show that FC-EDF dramatically outperforms EDF in soft real-time scheduling, and that it does so in a manner consistent with performance predictions made possible by the tools of control theory. The methods in this thesis are designed to support a closed-loop, control-theoretic approach, and I do not consider open-loop or non-control-theoretic methods further.

Closed-loop, control-theoretic methods have also been applied in a variety of software application areas in addition to those just mentioned in job scheduling. Examples of control-theoretic applications in software include admission control for the Lotus Notes electronic mail server[35, 86], operating system process scheduling[61, 64, 65, 109], distributed scheduling[108], service delays in web servers[1, 62, 66, 67, 97], processing delays in web servers[40], memory and CPU usage by web servers[21], cache decay[116], data migration in storage systems[63], integrated circuit thermal management[106], and dynamic voltage scaling[68, 122]. In all of these cases, a model of the system to be controlled—the *plant*, in the terminology of control theory—is required in order to apply control-theoretic techniques. Although this model can be developed from first principles, a more-common approach is to use system-identification techniques with off-

[9] The miss-ratio is the proportion of jobs that miss their deadlines out of all those jobs that were scheduled.

line statistical measurements to fit parameters to a pre-chosen, canonical differential-equation-based plant model. The steps required to use such a system-identification approach have been formalized in the *ControlWare* middleware architecture[121]. Here, in contrast, I use a process model (a Markov Birth/Death Chain), which possesses a natural correspondence to the state and behavior of the plant, and use on-line estimation to determine the model parameters. This approach has the advantages of being accessible (the state of the Markov Chain relates directly to the state of the resource) and adaptable, rather than abstract and fixed. Finally, the RED network congestion algorithm—an established, heuristic control approach—has been analyzed from a control-theoretic perspective using a fluid-transfer model of packets in the network[42, 79], which allowed its previously poorly-understood tuning parameters to be chosen in a more-principled manner. With the exception of [97], these control systems (and the analyses in [42, 79]) are linear. This is not a complete survey, but the applications and approaches are representative of current research. Although each of the applications in this paragraph partially overlap with the approach taken in this thesis—e.g., [66] is adaptive, [97] is nonlinear, and all use principled control-theoretic methods—the approach in this thesis is the first to combine adaptivity, nonlinearity, and distribution-based control in a *generalized* instead of *ad hoc* control architecture.

A common factor in all control applications is the need for *measurability*. In some cases—particularly in real-time systems in which the variance in job schedules may be low due to the presence of cyclic tasks—this is accomplished through the use of an instantaneous measurement of the plant. For example, the *miss ratio* used in [64, 65] represents the instantaneous, discrete state. Often, however, an instantaneous metric exhibits excessive variance and so it is therefore averaged over time, e.g. the averaged length of the server-connection queue in [35, 86]. However, such a solution is not always appropriate: in many applications, the instantaneous state is too bursty for stable control and the average may not be representative of the true system state—particularly when one assumes the presence of malicious attackers.

The approach in this thesis is a novel combination of adaptive, nonlinear, resource-distribution control that borrows from and extends upon the positive characteristics of previous methods. In contrast to all other related software-control work, the methods in this thesis use the full *distribution* of resource states when making control decisions and hence avoid the problems of state averaging. Moreover, the control algorithm used here does not require off-line parameter estimation, in contrast to system-identification approaches, and it is *adaptive*, in the sense that the controller measures its environment and automatically tunes a controller parameter in order to optimize the plant response. Finally, unlike previous work (with the exception of [97]), the control method described in this thesis is *nonlinear*, which facilitates the design of higher-performance controllers, as will be described in detail in Chapter 4.

2.5 Adaptive Filtering

A filter is a mechanism that estimates a quantity at time t from its past history. Two related concepts are *smoothing* and *prediction*. Smoothing uses additional data acquired after t to achieve the same goal as a filter, and is not causal. Prediction is a means by which to make a forecast about the quantity's characteristics after t. In this thesis, I am concerned only with filtering.

Filters are ubiquitous in computer science and are covered in numerous texts. Typical types of filters are moving averages (finite impulse response, or FIR filters), exponentially weighted moving averages (infinite impulse response, or IIR filters), and recursive least-squares regression filters. These general types of filters are *linear* and *time invariant*, allowing their filtering characteristics to be precisely quantified through time- or frequency-domain analysis. Adaptive filters contain adjustable parameters that are set automatically, usually as a function of the statistical characteristics of the filter input. The Kalman Filter, for instance, is an adaptive filter that adjusts its own parameters based upon the evolving error in the filter estimate. Good texts on adaptive filters and least-squares approaches like the Kalman Filter are [39] and [115].

Chapter 3

Theoretical Background

This chapter reviews concepts in Probability Theory, Stochastic Processes, classical Linear Control Theory, and Linear Filtering Theory that serve as background for this thesis. Probability is discussed in section 3.1: fundamental concepts in section 3.1.1 and the more-advanced topics of Stochastic Processes and Markov Chains in sections 3.1.2 and 3.1.3, respectively. Basic linear control and its PID variant are discussed in section 3.2, followed by adaptive, linear filtering in section 3.3. With the exception of the material in section 3.1.3 on the convergence of Markov Chains, the information in this chapter can be found in textbooks in the relevant area; its presentation here is intended more as a unified, consistent reference of nomenclature and notation for the following chapters than a detailed exposition of the topics.

3.1 Probability Theory, Stochastic Processes, and Markov Chains

This section introduces the basic principles of Probability Theory, Stochasic Processes, and Markov Chains, upon which the modeling methods and control algorithms in this thesis are built. Emphasis is placed upon Markov Chains and particularly upon the concept of the *stationary distribution* of a Markov Chain. Stationary distributions play a fundamental part in the control algorithms described in Chapter 4 and the proof-of-concept examples in Chapter 5. In particular, it is critical that the reader understand the relationship between Equation (3.6), the *detailed balance equation* (section 3.1.3), and the stationary distribution of a Markov Chain.

Although the information in this section is presented formally, it is neither rigorous nor complete. The intent is to provide the required mathematical background without overwhelming the reader with excessive detail. More complete references include introductory textbooks[83, 84, 100], as well as detailed references[50, 111].

3.1.1 Probability Theory

Modern Probability Theory is based upon Axiomatic Probability Theory[52] and comprises concepts borrowed from set theory and measure theory[111]. However, much of its language has been carried forward from its beginnings as a study of games of chance, initiated by Pascal, Fermat, and Laplace[114]. Axiomatic Probability Theory consists of three axioms in the context of a probability space and the theorems that follow. The *probability space* is a triple, $(\Omega, \mathbf{F}, P)$, consisting of

- a *sample space*, Ω, representing the set of all possible events or outcomes in a random experiment;
- a *family of subsets* of the sample space, $\mathbf{F} \subset \Omega$, such that $\mathbf{F}$ is a σ-algebra[1] ; and
- a probability measure, P, that assigns a nonnegative value $P(\mathcal{A})$ to all $\mathcal{A} \in \mathbf{F}$.

Any subset of Ω, including Ω itself, can be thought of as a possible event. The three *Axioms of Probability* are:

(1) For all $\mathcal{A} \in \mathbf{F}$, $0 \leq P(\mathcal{A}) \leq 1$.

(2) $P(\Omega) = 1$.

(3) For any set of mutually disjoint subsets $\{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_i\} \subset \Omega$,

$$P(\mathcal{E}_1 + \mathcal{E}_2 + \ldots + \mathcal{E}_i) = P(\mathcal{E}_1) + P(\mathcal{E}_2) + \ldots + P(\mathcal{E}_i)$$

The second axiom implies that there are no events outside the sample space, and leads naturally to the designation of Ω as the *certain* event. The third axiom (also called the *additivity property*)

[1] Informally, a σ-algebra is a group of sets that is closed under both finite and infinite unions, intersections, and complements.

states that the probability of one event from a group of independent events occurring is equal to the sum of the probabilities of the individual events; the additivity property does not hold if the events are not independent (i.e., the event sets are not disjoint).

The **Law of Total Probability** follows immediately from the Axioms but is of such importance that it is usually stated separately. Let $\mathcal{B}_i, 1 \leq i \leq n$, be a partition of the set Ω. Then for $n \geq 1$ and any event $\mathcal{A}$,

$$P(\mathcal{A}) = \sum_{i=1}^{n} P(\mathcal{A}\mathcal{B}_i) \tag{3.1}$$

where $\mathcal{A}\mathcal{B}_i$ represents the intersection of set $\mathcal{A}$ with set $\mathcal{B}_i$. Informally, the law of total probability states that the probability of an event is equal to the sum of the probabilities of those (disjoint) events of which it is composed. For example, if we roll a pair of dice, the probability of obtaining two rolls that sum to 11 is the probability of rolling a six on the first die and a five on the second, plus the probability of rolling a five on the first die and a six on the second. This law is particularly useful when computing *conditional probability*, which is fundamental to stochastic processes, and is defined as the probability that event $\mathcal{A}$ occurs, given that $\mathcal{B}$ has already occurred, and is denoted by $P(\mathcal{A}|\mathcal{B})$.

The concept of the random variable is the conceptual bridge between basic Probability Theory and Stochastic Process Theory. A *random variable* X on a probability space is a mapping $X : \Omega \rightarrow \mathcal{S}$, for some set $\mathcal{S}$. Such variables are *discrete* if Ω is countable,[2] and are otherwise *continuous*. Random variables are in fact functions, but the functional notation is usually suppressed. For example, if the sample space is given by the outcomes of five flips of a coin, the discrete random variable X might represent the number of heads in those five flips. The outcome $\mathcal{A} = \{H, H, T, H, T\}$ is an event in Ω, but rather than using the notation $X(\{H, H, T, H, T\}) = 3$ or $X(\mathcal{A}) = 3$, I usually just say $X = 3$ when the event is clear. The *expected value of a random variable* is its average value. Given a discrete random variable X, its expected value (denoted

[2] Shorthand for *countably infinite*.

by $E[X]$) is given by

$$E[X] \equiv \sum_{i=-\infty}^{\infty} iP(X = i)$$

providing that the summation converges absolutely[3] .

An important result regarding random variables is the *Strong Law of Large Numbers*. Consider a set of independent and identically distributed random variables:

$$X_1, X_2, \ldots, X_n$$

each having mean μ and variance σ. The law of large numbers states that the average and variance of these variables *converges almost surely*[4] to μ and σ, respectively, as $n \to \infty$. That is to say, the sample average and variance will converge to the true average and variance if a large enough number of samples are taken. A related result, the *Central Limit Theorem*, states that the distribution of sample averages will approach a gaussian distribution as the number of sample averages increases. There is now enough background to introduce stochastic processes.

3.1.2 Stochastic Processes

A *stochastic process* is a set of random variables $\{X(t) : t \in \mathcal{T}\}$ defined on a common probability space, where t is usually considered to be a time and $\mathcal{T}$ a set of points in time. $X(t)$ then represents the value of the stochastic process at time t. Note that this standard notation is somewhat unclear: each $X(t)$ is actually an $X(t, \omega)$ where $\omega \in \Omega$ is the event that occurred at time t. A *Markov Process* is a stochastic process that possesses the Markov Property, expressed formally as:

$$P\{X(s+t) = j | X(s) = i, X(u) = x_u, 0 \leq u < s\} = P\{X(s+t) = j | X(s) = i\}$$

where x_u represents the value of the process at time u. Intuitively, the Markov Property states that the process's next value depends only upon its current value, and not upon its history.

[3] A series $\sum x_n$ is absolutely convergent if $\sum |x_n|$ converges

[4] A sequence of random variables X_n converges almost surely to a limit L if $P(\lim_{n\to\infty} X_n = L) = 1$.

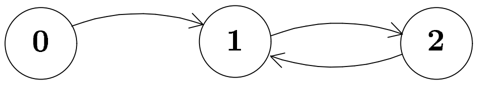


Figure 3.1: Markov Chain Reachability. Only states 1 and 2 communicate.

Because of this, certain kinds of Markov Processes are said to *lose memory* of their past (an idea to be made more precise shortly).

The terminology regarding the discreteness or continuity of a Markov Processes is inconsistent in the literature. I will call a Markov Process a *Markov Chain* if $\mathcal{S}$ is discrete or countable; if the potential for confusion exists, I call it a *continuous-time* Markov Chain when $\mathcal{T}$ is continuous, and a *discrete-time* Markov Chain when $\mathcal{T}$ is countable. The elements of $\mathcal{S}$ in Markov Chains are called its *states*, and their number and relationship to each other define the Chain's behavior over time.

3.1.3 Markov Chains

Markov Chains are usually represented using graphs, with one vertex for each state and directed edges for possible state transitions. The states themselves represent the specific, discrete values that may be assumed by the Markov Process that the graph is meant to portray. The presence of a transition between two states indicates a positive probability that the process will make the associated state change. If a transition exists between states i and j, I write $i \rightarrow j$. If some series of states and transitions exist such that

$$i \rightarrow k_0 \rightarrow k_1 \rightarrow \ldots k_n \rightarrow j \tag{3.2}$$

then (3.2) is called a *sample path* from i to j. In this case, I say that j is *accessible* from i and denote this fact by the notation $i \rightsquigarrow j$. If $i \rightsquigarrow j$ and $j \rightsquigarrow i$ then i and j are said to *communicate*, denoted by $i \leftrightsquigarrow j$. In Figure 3.1, states 1 and 2 are accessible from state 0 ($0 \rightarrow 1$, $0 \rightarrow 1 \rightarrow 2$, and $0 \rightsquigarrow 2$), but only states 1 and 2 communicate ($1 \leftrightsquigarrow 2$).

State classification is of fundamental importance in the theory of Markov Chains, since most existing mathematical results have been proved with respect to classes of states. These

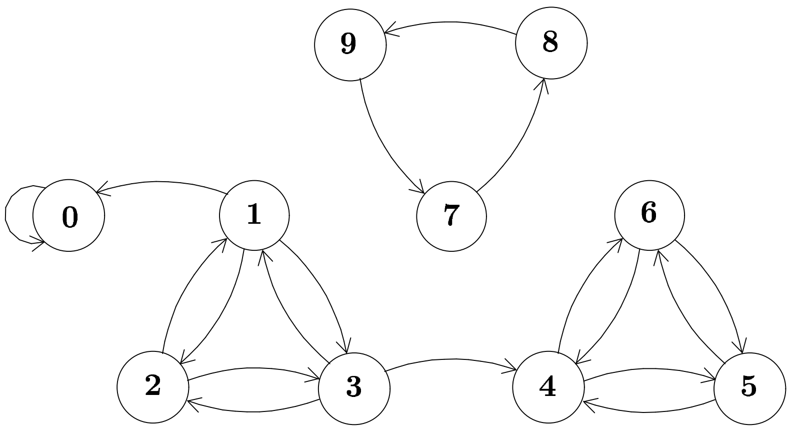


Figure 3.2: Markov Chain classes: absorbing (state 0); transient (states 1, 2 and 3); positive recurrent (states 4, 5 and 6); and periodic, positive recurrent (states 7, 8 and 9).

results are known as **class properties**. This is similiar to the concept of complexity classes in theoretical computer science: if a new theoretical problem can be shown to be isomorphic to a known NP-complete problem, for example, then it is just as 'hard' to solve as any other problem in the NP-complete class, at least in an asymptotic sense. The same is true of Markov Chain classes and class properties: if a Markov Chain state can be shown to be a member of a particular class of states in the chain, then it possesses all that class's properties.

Each state in a Markov Chain can be classified based upon its connectivity (transitions to other states). Four primary classes are represented in Figure 3.2:

- An *absorbing* state is one from which the exit transition probability is 0. When the chain enters an absorbing state, it is trapped there forever, as in state 0 of the Figure. The number of states in an absorbing class is always one.
- A *transient* state is one to which there exists a positive probability of never returning. States 1, 2 and 3 are transient.
- A *recurrent* state is one to which, once entered, the probability of returning is 1. Note that this implies that over an infinite period of time, all recurrent states will be visited infinitely often. Recurrence can be distinguished into two types:

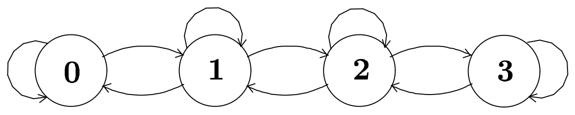


Figure 3.3: Markov Chain for Stationary Distribution Example

(1) The expected time between visits to a *positive recurrent* state is finite.

(2) The expected time between visits to a *null recurrent* state is infinite.

In Figure 3.2, states 4-9 are positive recurrent. Null recurrent states do not exist in finite Markov chains and will not appear again in this thesis.

- A *periodic* state is a state that can only be visited every d steps, where d is fixed and greater than 2. This is not a rigorous definition, but periodicity has an intuitive interpretation as illustrated by states 7, 8 and 9. Similiarly, one group of *aperiodic* counterparts is the group of states 4, 5 and 6.

A state that is positive recurrent and aperiodic is called an *ergodic* state. If a Markov Chain possesses only one class of states then it is said to be *irreducible.* An irreducible Markov Chain consisting of ergodic states is often called an *ergodic Markov Chain.* Many useful results have been proven regarding ergodic Markov Chains. The two most important, for the purposes of this thesis, are as follows.

(1) Ergodic Markov Chains possess a *stationary distribution.* That is, there is a probability distribution to which the state behavior of such a Markov Chain will converge.

(2) The stationary distribution of an ergodic Markov Chain is unique.

Proofs for these assertions can be found in numerous texts, e.g. [83].

It is informative to illustrate the concept of the stationary distribution through a simple example that also introduces many ideas that are important to understanding the active control strategy discussed in Chapters 4 and 5. Informally, I define the stationary distribution for a Chain to be the set of limiting probabilities that the Chain is in a given state at a given time.

That is, take an ergodic Markov Chain and let it 'run' for some period of time. In the limit, the percentage of time that it spends in each state is the stationary distribution.

To make this more formal, I work through the example as follows:

(1) Given the Birth/Death Markov Chain of Figure 3.3, I construct its *transition matrix*, **P** (defined below).

(2) Derive the two-step transition probability $P(X_2 = j|X_0 = i)$ for any two states i, j of the Chain. Note that a **step** can be either a fixed or variable interval, depending on whether the Markov Chain is discrete- or continuous-time.

(3) Show how this result generalizes to n steps.

Below is the transition matrix, **P**, for the Markov Chain of Figure 3.3. The transition matrix for an N-state Markov Chain is an $N \times N$ matrix, in which element P_{ij} from row i, column j of this matrix represents the probability that the process makes a transition from state i to state j. As is usual in computer science applications, the indices of the matrix begin at 0.

$$\mathbf{P} = \left| \begin{array}{llll} 1 - P_{01} & P_{01} & 0 & 0 \\ P_{10} & 1 - P_{10} - P_{12} & P_{12} & 0 \\ 0 & P_{21} & 1 - P_{21} - P_{23} & P_{23} \\ 0 & 0 & P_{32} & 1 - P_{32} \end{array} \right|$$

Note that this matrix represents the *one-step* transition matrix, i.e., P_{ij} is the probability of being in state j in one time step after starting from state i.

The next step is to derive the two-step transition matrix, $\mathbf{P}^{(2)}$, whose superscript indicates its distinction from the one-step matrix and whose entry $\mathbf{P}_{ij}^{(2)}$ is the probability of reaching state j after two steps from state i. I start by examining the scenario where the chain starts and ends in state 0. This is possible with one of two sample paths, either $0 \rightarrow 0 \rightarrow 0$, or $0 \rightarrow 1 \rightarrow 0$; the respective probabilities are $P_{00} \times P_{00}$ and $P_{01} \times P_{10}$. According to the law of total probability, the probability of being in state 0 after two steps is the sum of the two-step sample paths that start and end at 0, namely $P_{00}^2 + P_{01}P_{10}$. Similarly, if we seek the probability of being in state 1 two steps after starting from state 0, the possible sample paths are: $0 \rightarrow 0 \rightarrow 1$ and $0 \rightarrow 1 \rightarrow 1$, with respective probabilities $P_{00} \times P_{01}$ and $P_{01} \times P_{11}$. Continuing in this fashion, we can build the two-step transition matrix:

$$\mathbf{P}^{(2)} = \begin{vmatrix} P_{00}^2 + P_{01}P_{10} & P_{00}P_{01} + P_{01}P_{11} & P_{01}P_{12} & 0 \\ P_{10}P_{00} + P_{11}P_{10} & P_{10}P_{01} + P_{11}^2 + P_{12}P_{21} & P_{11}P_{12} + P_{12}P_{22} & P_{12}P_{23} \\ P_{21}P_{10} & P_{21}P_{11} + P_{22}P_{21} & P_{21}P_{22} + P_{22}^2 + P_{23}P_{32} & P_{22}P_{23} + P_{23}P_{33} \\ 0 & P_{32}P_{21} & P_{32}P_{22} + P_{33}P_{32} & P_{32}P_{23} + P_{33}^2 \end{vmatrix}$$

A careful inspection of the elements of $\mathbf{P}^{(2)}$ will reveal that they are the same as those of $\mathbf{P} \times \mathbf{P}$; in fact, it can be shown that the n-step probability transition matrix can be obtained through n matrix multiplications of $\mathbf{P}$ with itself. Hence we can drop the parentheses on the superscript and refer to the two-step transition matrix simply as $\mathbf{P}^2$. In the theory of Markov Chains, these ideas are formalized by the *Chapman-Kolmogorov Equations*[83] for n-step transition probabilities of discrete time Markov chains:

$$P_{ij}^{(n)} = \sum_{k \in \mathcal{S}} P_{ik}^{(m)} P_{kj}^{(n-m)} \qquad 0 \leq m \leq n \tag{3.3}$$

where $P_{ij}^{(n)}$ represents the probability that the Chain state will transition into state j in n steps, conditioned upon starting in state i. The n-step transition matrices are referred to as the *transient distributions*, and entry $P_{ij}^{(n)}$ is directly proportional to the average amount of time spent in state j after n steps from state i. As a result, the rows of the transient distribution matrices will generally be different for low values of n.

As n approaches infinity, all rows of the matrix of an ergodic Markov Chain will converge to the same stationary row vector, a result of the Markov Property and the communicability of each state with every other[83]. This vector, represented by $\boldsymbol{\pi}$, is called the *stationary distribution*, and it satisfies the following *global balance equation*:

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P} \tag{3.4}$$

The vector $\boldsymbol{\pi}$ is *stationary in time*: it will not change with any number of steps. This unique vector is different from the n-step transient vectors in that it represents the stationary, unchanging long-term trend of the Markov Chain. As such, the elements of the stationary vector are **unconditional**, meaning that the initial state of the chain is unimportant, in contrast to the n-step probabilities, which are conditioned upon the starting state of the Markov Chain.

Two important properties of ergodic Markov Chains can be inferred from the fact that the transient distributions always converge to a stationary distribution:

(1) **Memorylessness**: Markov Chains *lose memory* of their initial state.

(2) **Coverage**: Over an infinite period of time, each state will be visited an infinite number of times if its stationary probability is nonzero.

Both of these facts are useful from a control standpoint: *Memorylessness* implies that control systems need not be aware of system history, only current behavior, while *Coverage* means that if a 'bad' state is reachable, it will be entered at some point. For control purposes, this means that the system needs to be designed to make undesirable states unreachable by reducing their stationary probability to zero.

The stationary distribution can be determined through easier means than an infinite number of matrix multiplications. The stationary equation $\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P}$ is a system of n equations in n unknowns, which can be solved for $\boldsymbol{\pi}$. However, in every Markov Chain transition matrix, there exists one column that is a multiple of some other (see, e.g., [83] for details on why this is true). The result is that there are only $n-1$ equations for the n unknowns. Nevertheless, the system can still be solved for $\boldsymbol{\pi}$ using the first axiom of probability, which implies that

$$\sum_{i=0}^{n} \pi_i = 1 \tag{3.5}$$

This is known as the *normalization condition.* To solve mechanically for the stationary vector, one replaces the duplicated or scaled column with one that realizes the normalization condition, after which any standard linear matrix solution method (e.g., Gaussian Elimination, Cramer's Rule, etc.) can be used.

The solution of Equation (3.4) by way of Equation (3.5) is the general method to determine the stationary distribution of any Markov Chain, but the process can be simplified even further for certain Chain topologies. In particular, for any two states i and j of a finite Birth/Death Markov Chain—the Chain topology of interest in this thesis—the stationary distribution of the Chain is related to its transition probabilities by:

$$\boxed{\pi_i P_{ij} = \pi_j P_{ji}} \tag{3.6}$$

This relationship is known as the *detailed balance equation.* (It is also known as the *time reversiblty condition* in some texts, and Markov Chains that satisfy it are said to be *time reversible.*) Following [83], I refer to the quantity $\pi_i P_{ij}$ as the *probability flux* out of state i and into state j. Since P_{ij} is, in a sense, a measure of the rate of state transitions from state i to state j, it is convenient to think of probability flux as the rate of change in the probability mass that flows from i to j in a single step of the Markov chain. Probability flux is the basic quantity by which the convergence rate of Markov Chains is quantified.

Markov Chain Convergence

Determining the convergence rate of Markov Chains is an open research area which as yet has yielded few general results, as described in Chapter 2. Informally, determining the convergence rate is a matter of characterizing how easily the probability mass moves through transient distributions towards the Chain's stationary distribution—i.e., how the probability flux between states is related to the stationary distribution. Markov Chains bear a close relationship to electrical circuits (see, for example, [84]); the stationary distribution of a Chain is analogous to the steady state of a circuit. Similarly, in a manner remniscent (although not strictly analogous) to conductance in an electrical circuit, the *conductance* of a Markov Chain is related to its asymptotic convergence to stationarity.

If we consider the Markov Chain as a graph, G, then we can define conductance by referencing the probability flux over the cut between a non-empty subset $S \subset G$ and its complement $\bar{S}$. The *capacity* of S is defined as

$$C_S = \sum_{i \in S} \pi_i \tag{3.7}$$

and represents the total probability mass that can exist in S. If probability mass is a moving quantity and probability flux tells how much of it is moving (and in which direction), then capacity measures how much accumulates in each state. The sum of all the fluxes leaving S is called the *ergodic flow* out of that state, and is given by

$$F_S = \sum_{i \in S, j \in \bar{S}} P_{ij} \pi_i \tag{3.8}$$

The quantity F_S / C_S is called the *conductance* of S and is denoted by Φ_S. Conductance is the probability that a Markov Chain state crosses the cut from S to $\bar{S}$, conditioned on the probability that it is already in S. Note that unlike its electrical counterpart, Markov Chain conductance is not symmetric: Φ_S need not be equal to $\Phi_{\bar{S}}$. Bearing this in mind, the *global conductance* of a graph G is defined as

$$\Phi(G) = \min_{0 < |S \subseteq G| < N} \max \Phi_S, \Phi_{\bar{S}} \tag{3.9}$$

which states that the lowest-conductance subset of a graph determines the conductance of the graph as a whole. Global conductance is a measure of the relative connectivity of the subsets of states in a Markov Chain, and hence has bearing on the chain's convergence towards stationarity. Markov Chains with low global conductance possess subsets of states that are in some sense isolated, leading to a *quasi-stable* condition in which the Chain converges quickly to an intermediate, nonstationary distribution, but only slowly evolves towards its true, stationary distribution.

3.2 Control Theory

In this section, I introduce some relevant classical feedback control concepts. Control theory is a combination of *analysis* methods that are used to characterize the input-output behavior of a system and *synthesis* techniques for designing control systems to control such a system. In the language of this discipline, the *plant* is the system to be controlled and the *reference* is the level to which the controller tries to hold the plant output. The fundamental method by which this is achieved is through inversion, as illustrated by the simple control model

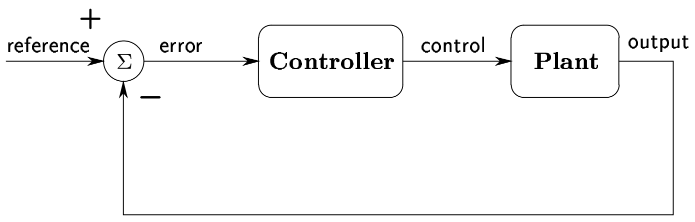
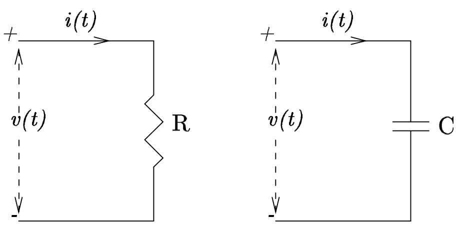


Figure 3.4: A Simple Feedback Control System. The difference between the plant output and the reference signal generates an error signal that the Controller uses to steer the plant output.

of Figure 3.4: invert the output of the plant, add the result to the desired or *reference* level, and use the resulting *error* as input to the controller, which then guides the plant output towards the reference. This reference signal is chosen to satisfy engineering needs. For example, it might be the temperature setpoint in a home's thermostat, a set-speed in a car's cruise control, or a desired altitude in an airplane's autopilot.

The design of a controller involves balancing a number of tradeoffs, among the most important of which are response speed and stability. (These are by no means the only characteristics of a controller that can be optimized.) To clarify this, I perform an analysis example from a classical control domain: electrical engineering. To analyze or design a controller in the manner of the example that follows, it is necessary to have a closed-form expression for both the Controller and the Plant components of Figure 3.4. Often, however, the plant model is unknown—yet control is still required. In these cases, heuristic control methods are required. An important heuristic control paradigm is the so-called Proportional/Integral/Derivative controller, or PID Control. This control type is particularly relevant to distribution control, since (as will become apparent in Chapter 4), the plant model of a Markov Chain is difficult to derive.

As in previous sections, the information in this section is simplified and condensed. Excellent control theory references that cover both linear feedback control systems as well as PID control are [38, 89].



(a) Resistor (b) Capacitor

Figure 3.5: Resistor and Capacitor Circuit Symbols.

3.2.1 Feedback Control: A Worked Example

The example in this section, which illustrates the basic tradeoffs in control design, comes from the domain of electrical engineering. It consists of a *second-order voltage follower* circuit, whose output is designed to match (follow) its output. (Although apparently useless, this circuit finds application wherever buffering is required.) The following definitions for voltage, current, resistance, capacitance, and operational amplifiers are standard background for this domain and circuit.

A *voltage* is a potential that varies in time with respect to a zero reference. This is similiar to gravity: one speaks of the gravitational potential between two bodies, not just of the 'gravity' of a mass. An instantaneous voltage can be positive or negative relative to the reference and is denoted by $v(t)$, or $V(s)$ after a Laplace transform (where s represents a frequency-domain variable, the counterpart to t, a time-domain variable).

A *current* is a time-varying signal that is induced by a voltage and that flows from higher to lower potential if a conductive path for it exists; currents are denoted by $i(t)$, or $I(s)$ after a Laplace transform.

Passive circuit elements are physical devices that do not amplify circuit signals and that possess specific constitutive current/voltage relationships. That is, upon application of some $v(t)$ to a passive element, a specific $i(t)$ flows through it. A *resistor*, as in Figure 3.5(a), is a conductive circuit element with value R (measured in *ohms*), whose current/voltage relationship

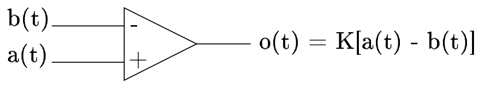


Figure 3.6: Operational Amplifier Circuit Symbol. The output voltage is the difference between the positive input, $a(t)$, minus the negative input, $b(t)$, multiplied by a gain constant, K.

is $V(s) = I(s)R$, known as *Ohm's Law.* A capacitor, as in Figure 3.5(b), is an element that stores electrical charge. It has a value C (measured in *farads*) and its current/voltage relationship is $I(s) = 1/(sC)V(s)$. This Laplace relationship is valid as long as no initial charge is held by the capacitor, which is what I assume for this example.

Active circuit elements, which are more complicated than passive elements, can be used to amplify voltages or currents. I will use only one active circuit element: the *operational amplifier*, shown in Figure 3.6. An operational amplifier (or *op-amp*) possesses a gain, K, which is taken to be infinite for an 'ideal' op-amp. Its output voltage, $o(t)$, is this gain times the difference between the voltage on the noninverting input, $b(t)$, and the voltage on the inverting input, $a(t)$. Ideal op-amps possess two important properties: *infinite input resistance* (so no current can flow into their inputs), and a *virtual short circuit* between the two inputs, such that $a(t) = b(t)$. The open-loop characteristic of an ideal op-amp is to produce $o(t) = +\infty$, $o(t) = 0$, or $o(t) = -\infty$, depending on its inputs.

An example circuit that uses these elements is shown in Figure 3.7. This is a feedback controller with reference signal $V_i(s)$ and output signal $V_o(s)$. Its steady-state analysis is straightforward in the Laplace domain, using Kirchhoff's Current Law, which states that the sum of currents flowing into any point in a circuit must be zero. Using the passive circuit element current/voltage relations, the characteristics of op-amps, and this law, we can compute the voltages in this circuit by summing the currents flowing into points 'a' and 'c' on this circuit:

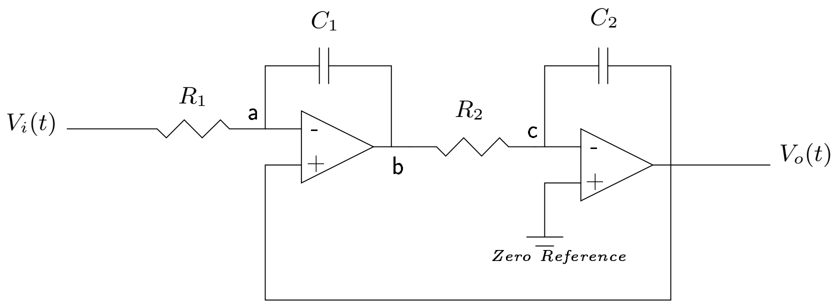


Figure 3.7: Second-order Op-Amp Feedback System.

$$\frac{V_o(s) - V_i(s)}{R_1} + (V_o(s) - V_b(s))sC_1 = 0$$
$$-\frac{V_b(s)}{R_2} - V_o(s)sC_2 = 0$$

Solving for the transfer function, $G(s)$, which relates the circuit output to its input:

$$G(s) = \frac{V_o(s)}{V_i(s)} = \frac{1/(R_1R_2C_1C_2)}{s^2 + (1/R_2C_2)s + 1/(R_1R_2C_1C_2)} \tag{3.10}$$

Letting $\omega_0^2 = 1/(R_1R_2C_1C_2)$ and $\zeta = \sqrt{R_1C_1/(4R_2C_2)}$ makes possible the following suggestive transformation of (3.10):

$$G(s) = \frac{\omega_0^2}{s^2 + 2\zeta\omega_0 s + \omega_0^2} \tag{3.11}$$

This equation, which represents the combined model of the controller and the plant (which are difficult to separate in this example), is one of several standard forms for a canonical second-order transfer function, and in which ζ is called the ***damping ratio*** and ω_0 the ***natural frequency.***

The tradeoff of response speed versus stability is demonstrated by applying a ***unit step*** function to the $v_i(t)$ input of this circuit. The unit step, $u(t)$, is defined as follows:

$$u(t) = \begin{cases} 0 & t < 0 \\ 1 & t > 0 \end{cases}$$

Its Laplace transform, $U(s)$, is $U(s) = 1/s$. Using the system transfer function and this trans-

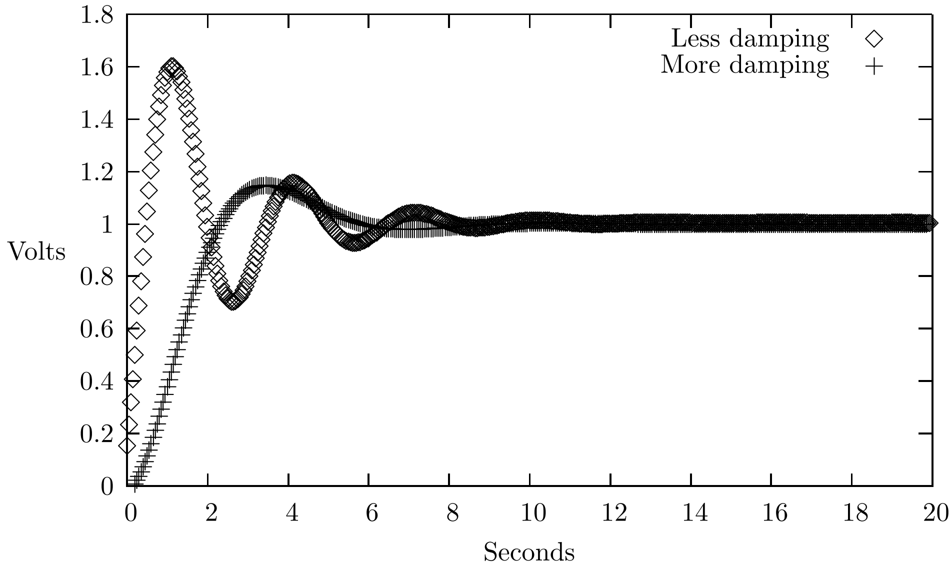


Figure 3.8: The Speed/Stability Tradeoff. A faster control response approaches the objective (1.0 volt) more quickly, but it overshoots and oscillates more than a slower one.

form, we have:

$$\begin{aligned} V_o(s) &= G(s)U(s) \\ &= \frac{\omega^2}{s\left(s^2 + 2\zeta\omega s + \omega^2\right)} \end{aligned} \tag{3.12}$$

The reverse Laplace transform of this expression is

$$v_o(t) = 1 - \frac{1}{\beta} e^{-\zeta\omega t} \sin(\beta\omega t + \theta) \tag{3.13}$$

where $\beta = \sqrt{1 - \zeta^2}$ and $\theta = \tan^{-1}(\beta/\zeta)$. Figure 3.8 shows the output response for two different sets of component values for the resistors and capacitors—one of them selected for fast response and the other for greater stability. The results clearly illustrate the tradeoff. The faster response rises more quickly, but it overshoots its settling point (1 volt) and then oscillates. The slower response, on the other hand, takes longer to rise, but overshoots less and reaches its settling point with fewer oscillations. The goal of this control system was to cause the plant output (the output of the circuit) to follow the unit step input exactly, i.e., to produce 0 volts for time $t < 0$, and exactly 1 volt afterwards. It was able to perform this task in both cases—

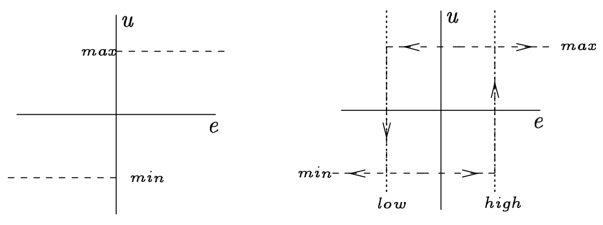
albeit with some delay, overshoot, and ringing—and it performed differently for different passive component choices. If one were to design a new circuit, the choice of the capacitor values would be dictated by the desired control response speed; one might choose smaller values for faster response but sacrifice stability by doing so. Note that the analysis reveals exactly *how* the performance differs as parameters vary, and is therefore a powerful tool for control-system designers to balance tradeoffs.

To synthesize a controller in this fashion, an engineer must know the model of the plant, choose the model of the controller, and select between design tradeoffs. One tradeoff—speed versus stability—has already been demonstrated, but actual controller design can involve the selection of many design parameters, e.g. controller overshoot, settling time, phase response, and more. When designing a linear controller for a linear plant model, an engineer chooses between these tradeoffs by tuning the parameters of the combined *transfer function* of plant and controller. For example, consider the simple controller of Figure 3.4. Its input (to the controller) is $e(t)$, or $E(s)$ in the Laplace domain; and we can denote its output (from the plant) by $o(t)$, or $O(s)$ in the Laplace domain. As in the previous electrical circuit example, the combined transfer function, $G(s)$, is given by

$$G(s) = \frac{O(s)}{E(s)}$$

Because the combined system is linear, this transfer function can be decomposed into contributions from the controller model and the plant model and will always reduce to a canonical first-order, second-order, third-order, etc. form. The order of the transfer function translates into a specific control architecture, and the desired characteristics are directly related to the details of that architecture. For example, if the combined transfer function of plant and controller corresponded to Equation (3.11), then the design engineer must choose the two parameters ω and ζ, from which the resistor and capacitor values can be directly computed.

This approach fails when the plant model is unknown, since the design characteristics cannot be related back to the variables of the transfer function, and hence neither the architecture



(a) No Hysteresis (b) With Hysteresis

Figure 3.9: On/Off Controller Characteristics with and without hysteresis.

of an appropriate controller nor its design parameters can be determined. It also fails when either the plant or the controller are nonlinear, since the principle of superposition—and hence these linear control methods—do not apply in nonlinear systems. Software systems, for example, are generally nonlinear and possess unknown plant models. In these cases, we may still use feedback control, but the only control option is a heuristic one such as PID control.

3.2.2 PID Control

Proportional/Integral/Derivative (PID) control does not require a closed-form expression for the system model. In the PID control paradigm, the Controller block of Figure 3.4 is a combination of P, I, and D stages in parallel that contribute to the control response in a manner that is proportional to the error, to the integral of the error over time, or to the derivative of the error over time, respectively. One or more of the stages may be missing (e.g., the controller may be Proportional only), but I will refer to any combination of P, PI, PD, etc. controller as a "PID Controller." PID controllers are prevalent in applications where the plant model is difficult to derive, in particular in process industries (e.g. manufacturing, chemical processing, pulp and paper production), where over 90% of the control loops are of the PID type[7].

The simplest type of PID controller is the P-only controller with infinite gain. This type of controller is sometimes called an On/Off controller. An On/Off controller provides a constant

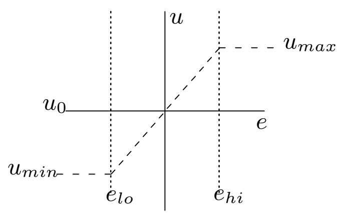


Figure 3.10: Proportional (P) Feedback Control System

control output based on the error signal, according to the following control law:

$$u = \begin{cases} u_{max} & e > 0 \\ u_{min} & e < 0 \end{cases}$$

The advantage of this controller is its simplicity and robustness. The disadvantage is that it can force the plant to oscillate around the reference level, an effect that can be mitigated by adding some hysteresis to the controller:

$$u = \begin{cases} u_{max} & e > e_{hi} \\ u_{min} & e < e_{low} \end{cases}$$

This modification allows the plant to vary slightly around $e = 0$ without triggering oscillations from the controller. Domestic thermostats (e.g. ovens) are typically On/Off controllers with hysteresis. Both cases are depicted in Figure 3.9.

Even with hysteresis, the On/Off controller oscillates too much to provide effective control in many cases, and its reliance upon a single high or low control level is generally too restrictive. The solution to this is to turn the abrupt transitions between u_{min} and u_{max} into a smooth continuum by introducing a finite proportional gain.

Proportional control applies a control signal that changes linearly between e_{low} and e_{hi}, as shown in Figure 3.10. Its control law is summarized as follows:

$$u = \begin{cases} u_{max} & e > e_{hi} \\ u_0 + K_p e & e_{low} < e < e_{hi} \\ u_{min} & e < e_{low} \end{cases}$$

Here, K_p is called the *proportional gain.* In some cases this gain can be calculated, but it is usually chosen by the control system designer based upon an *observed* tradeoff between overshoot, oscillation, and response speed. For example, in an electric oven, the desired baking temperature represents the reference signal and the gain represents the amount of power to apply to the heating elements. It would be possible to set the heating elements to maximum (red-hot), which would raise the temperature quickly, but it would also overshoot and become too hot, since the elements possess significant thermal inertia. The designer would note this phenomenon and reduce K_p. Note that this approach does not involve a mathematically precise formulation, but rather a general empirical heuristic and a "tuning" phase.

Proportional controllers avoid the oscillation problems of On/Off controllers, but suffer from steady-state errors[5] at low gain and overshoot or ringing at high gain. More formally, if we define e_{ss} as the steady-state error and u_{ss} as the steady-state control output, then by rearranging the P-controller law we obtain:

$$e_{ss} = \frac{u_{ss} - u_0}{K_p} \tag{3.14}$$

This implies that the steady-state error will only be zero if the gain K_p is infinite, or if the steady-state control output is equal to u_0 (sometimes called the *controller bias*)—which, in general, will not be true.

Increasing the proportional gain increases the speed of the control response—reducing the steady state error—but effectively turns the P controller into an oscillatory On/Off controller as the gain is increased, thereby reducing stability. An effective solution to the problem of steady-state error that does not involve infinite proportional gain is the addition of another

[5] The difference between the output of the plant and the reference in the limit as time goes to infinity

control action that corrects for steady-state error by integrating it over time and factoring that quantity into the control.

The addition of an Integral control stage can effectively address the problems of steady-state errors without requiring large proportional gains. The control law, with the addition of the Integral stage, is as follows:

$$u = \begin{cases} u_{max} & e > e_{hi} \\ K_p\left(\frac{1}{T_i}\int edt + e\right) & e_{low} < e < e_{hi} \\ u_{min} & e < e_{low} \end{cases}$$

where T_i is the time of integration. The quotient K_p/T_i is sometimes referred to as the *integral gain*, denoted K_i. Selection of K_i is dictated by how much steady state error needs to be offset. In the example of the oven, an integral stage is effectively introduced by adding insulation to the oven walls so that conduction losses are reduced or, ideally, offset exactly. The PI controller is in effect just a P controller with an integral term replacing u_0, so it adaptively selects the correct u_0 in order to eliminate steady-state errors. PI control can reduce steady-state errors to zero, but it treats rapidly and slowly changing plant outputs in the same manner, which can cause problems if the oven is just below the setpoint but heating up rapidly, for instance. In that case, the control system should react differently than if it is at the same temperature but cooling down. Adding a final control term that is related to the derivative of the error term addresses this.

A Derivative stage allows the controller to avoid excessive overshoot. The control law, with the addition of the Derivative stage, can be summarized as follows:

$$u = \begin{cases} u_{max} & e > e_{hi} \\ K_p\left(e + \frac{1}{T_i}\int edt + T_d\frac{de}{dt}\right) & e_{low} < e < e_{hi} \\ u_{min} & e < e_{low} \end{cases}$$

where T_d is the derivative time (that is, the rate of change of the error over the time period T_d). As before, the quotient K_p/T_i is the integral gain, but there is now an additional *derivative* *gain* term, $K_p T_d$. Choice of K_d is dictated by how much damping will be necessary in order to avoid oscillations when the plant output changes. For an oven, the derivative term would not be important during normal operation, but as soon as someone opens the door, heat will escape. The rate at which the heat escapes from this event depends upon whether the oven is in a hot or a frigid house, necessitating the choice of a small K_d in the former case and a larger K_d in the latter case, if one wants the same response rate.

A PID controller with all three P, I, and D stages—when properly tuned—can provide the "best" control in the sense that its control response is faster and more stable than P or PI controllers. The primary drawback in traditional design of PID controllers is the difficulty of tuning them, i.e. of selecting their three gain constants. Synthesis of PID controllers has historically proceeded in five steps, the last of which is this tuning step. These parts are:

(1) Characterize the *open-loop response*[6] of the plant in order to determine what needs to be improved.

(2) Add Proportional control to improve rise time.

(3) Add Integral control to eliminate steady-state error.

(4) Add Derivative control to decrease overshoot.

(5) Adjust each of K_p, K_i, and K_d to achieve the desired overall response.

This "recipe" for building and adjusting a PID controller is simplified: it is important to note that adding I and D stages (i.e., increasing the order of the controller) should be done only if absolutely required, since their inclusion increases the complexity (and can possibly decrease the stability) of the resulting controller[7]. The final adjustment step is difficult because the effects that are generally attributable to the three components (rise time, steady-state error, overshoot) are not linearally separable; adjusting K_p to improve the rise time, for example, can also increase the overshoot. Poorly understood tuning-parameter effects, as mentioned in Chapter 1, are a

[6] The behavior of the system without feedback control.

significant drawback. Fortunately, PID tuning is a well-studied topic: tuning methods for PID controllers range from simple ones (e.g. none at all, or simply trying to minimize one characteristic such as overshoot) to others that are more complicated (e.g. the *Ziegler-Nichols* method[123] or its modern version, *relay feedback*[6]). A straightforward approach that forms the basis for many tuning methods is to set K_p to a low value, slowly increase that gain until the control loop begins to oscillate, and then to reduce the gain to restore stability. The value to which it is reduced is what distinguishes the variations of the Ziegler-Nichols method, with that value set based upon the gain at which the system oscillated and its period of oscillation. A good general reference on PID control and tuning is [7], and a text dedicated completely to tuning is [120].

3.3 Linear Filtering

A filter, as defined in Section 2.5, is a mechanism that estimates a quantity at time t from its past history. Filters may be continuous or discrete, but I consider only the latter. The two fundamental types of filters are *Finite Impulse Response* (FIR) and *Infinite Impulse Response* (IIR) filters, which I describe in Sections 3.3.1 and 3.3.2, respectively. FIR and IIR filters can be combined to form Autoregressive, Moving-Average Filters (ARMA), which I discuss briefly in section 3.3.3. These types of filters are known as linear, time-invariant filters (LTI) because their response is linearly decomposable, and because their parameters are unchanging over time. *Adaptive* filters, which I describe in section 3.3.4, are extensions of the basic FIR and IIR paradigms that facilitate automatic adjustment of the filter parameters in order to maintain an optimal output. These types of filters are also linear, but not time-invariant. In the following sections, I will illustrate how various filters can be used to distill useful information from a time-series dataset.

As in linear control theory, linear filtering theory is well-developed and extensive. The overview in the following sections conveys the power of the techniques, but it is by no means complete. The analysis and synthesis of filters that possess both poles and zeros, in particular,

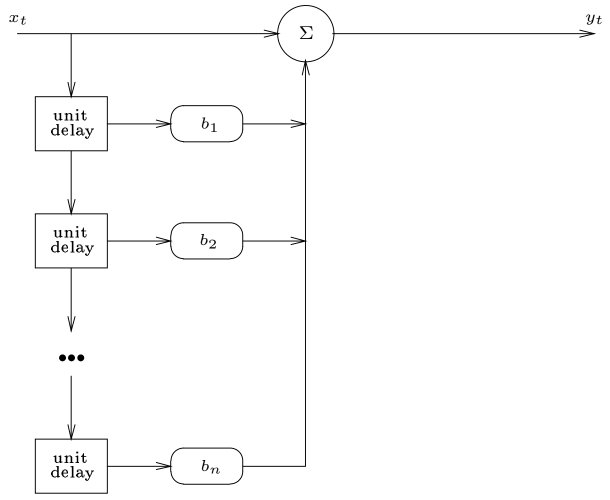


Figure 3.11: FIR Filter Block Diagram. The input time-series, x_t, passes through a set of unit delays, with terms at each tap point weighted by a constant and summed to produce the output series, y_t.

can be a complex process for which the interested reader should refer to the more detailed references cited in the following sections.

3.3.1 Finite Impulse Response Filters

A *Finite Impulse Response* filter (FIR), also known as a *Moving Average* (MA) filter, or a *Tapped Delay Line*, or a *Transversal* filter, consists of tapped, feed-forward, series-connected unit delay elements (Figure 3.11). Because there is no feedback from the filter output, a unit impulse input to such a filter causes an output that decays to zero in a finite period of time (hence the name Finite Impulse Response). Assume a series $\{x_t\}$ driving the input and a filtered series, $\{y_t\}$, appearing at the output. The relationship between the two in a FIR filter is given by

$$y_t = \sum_{k=0}^{N-1} b_i x_{t-k} \tag{3.15}$$

$$= b_0 x_t + b_1 x_{t-1} + \ldots + b_{N-1} x_{t-(N-1)} \tag{3.16}$$

where b_k are arbitrary weights chosen by the filter designer. If $b_0 = b_1 = \ldots = b_{N-1} = 1/N$, then this is a simple "moving average" of N terms. If we consider a unit impulse at time $t = 0$,

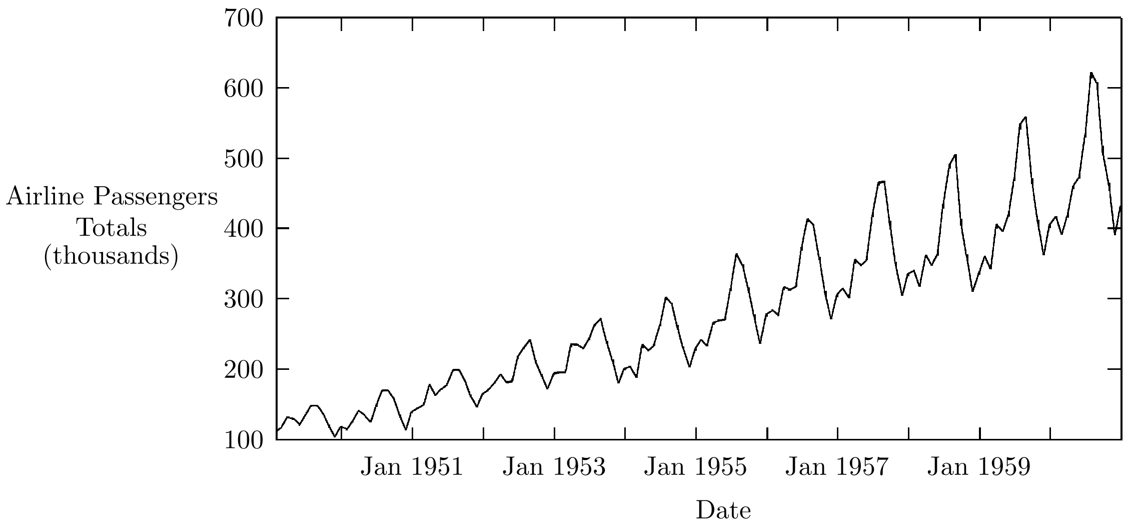


Figure 3.12: International airline passenger totals over a twelve year period starting January, 1949. The data exhibits a clear average upward trend as well as seasonal variations.

then the resulting output (filtered) series, $b_0, b_1, \ldots, b_N$, represents the *impulse response* of the system. Because N is finite, this response is guaranteed to drop to zero in a finite period of time.

Consider the time series of Figure 3.12, which depicts international airline passenger totals over a period of several years, with one data point taken each month[11]. There is a clear upward *trend* of growth over the years as the airlines carried more passengers, and each year exhibits a *seasonal variation* when more people fly during the winter and summer holidays. Moreover, the seasonal variation grows over time or—in more formal terms—the variance is correlated to the mean. Filtering, in the context of these passenger totals, is a question of predicting how much flight capacity will be needed in the next year. For this purpose, I wish to derive an estimate of the trend by removing the seasonal variation.

The design parameters for a FIR filter consist of the number of delay stages (the order of the filter) and the weight associated with each stage. To filter the data of Figure 3.12, I use a 12th-order filter with each weight equal to 1/12; this has the effect of averaging the previous

12 samples at every step, which is the desired effect for removing a yearly cycle. The equation for this filter is:

$$y(t) = \frac{1}{12}\sum_{k=0}^{11} x(t-k) \tag{3.17}$$

with a corresponding z-transform[7] of

$$Y(z) = \sum_{k=0}^{N-1} b_k z^{-k} \tag{3.18}$$

and a frequency response (when $z = e^{i\omega}$) of

$$Y(i\omega) = \sum_{k=0}^{N-1} b_k e^{-ik\omega} \tag{3.19}$$

$$= \sum_{k=0}^{N-1} b_k \left[\cos(k\omega) - i\sin(k\omega)\right] \tag{3.20}$$

Note that Equation (3.18) possesses only *zeros*, i.e. values of z^{-k} for which the output $Y(z)$—and hence $Y(i\omega)$—is equal to zero. In terms of frequency response, these points correspond to frequencies that the filter suppresses; for the example of Figure 3.12, we expect to see cyclic data at the filter input that occurs every 12 months to be suppressed at the filter output (i.e., a zero at one-cycle-per-twelve-months). To determine if this is the case, we compute the magnitude, or *gain* of the filter as a function of frequency (denoted here by $H(i\omega)$), that is given by

$$|H(i\omega)|^2 = [\mathrm{Re}\ Y(i\omega)]^2 + [\mathrm{Im}\ Y(i\omega)]^2$$

or

$$|H(i\omega)| = \sqrt{\left[\sum_{k=0}^{N-1} b_k \cos(k\omega)\right]^2 + \left[\sum_{k=0}^{N-1} b_k \sin(k\omega)\right]^2}$$

Here $x(t)$ is the input series (the raw passenger data) and $y(t)$ will be the resulting deseasonalized series. The magnitude response of the filter is shown in Figure 3.13. We see in this figure that there are twelve evenly-spaced zeros from $-\pi$ to π, where π represents half the sampling frequency, or six samples per year. Hence Figure 3.13 represents a filter with nulls at at $\pi/6$

[7] The z-transform is the discrete counterpart to the Fourier transform.

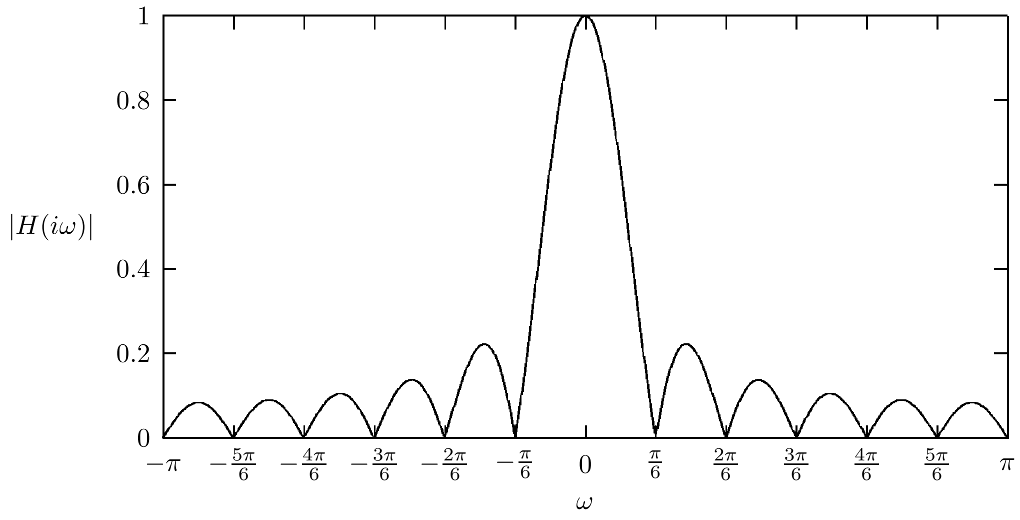


Figure 3.13: Magnitude response of the 12th-order FIR filter. The zero at $\pi/6$ corresponds to the removal of a 12-period (*i.e.* annual) cycle, and the unity gain at $\omega = 0$ to the preservation of the trend.

(one time per year, or 12-month cycles), $2\pi/6$ (two times per year, or 6-month cycles), $3\pi/6$ (4-month cycles), $4\pi/6$ (3-month cycles), $5\pi/6$ (2.4-month cycles), and π (2-month cycles). The unity gain at $\omega = 0$ serves to preserve the trend. Note that the highest-frequency cycle that can be nulled is half the sampling frequency, a well-known property called the Nyquist Effect.

If we apply this filter to the data of Figure 3.12, we obtain the results shown in Figure 3.14, in which the cycles have been removed. Note that there exists a *phase-delay* of twelve samples (months), demonstrated by the absence of any filtered output in the first year of data. The cause of this delay should be apparent: every input data point must pass through twelve unit delay elements before its full effect is felt at the filter output.

Discrete FIR filters possess a number of attractive characteristics for the purposes of this thesis. First, they can be easily realized in either hardware or software. Second, they are stable because they do not contain feedback loops. Finally, their analysis, as illustrated in this Section, is straightforward. In the next Section, I describe characteristics of IIR filters, which are sometimes harder to realize reliably in software, can become unstable, and are somewhat

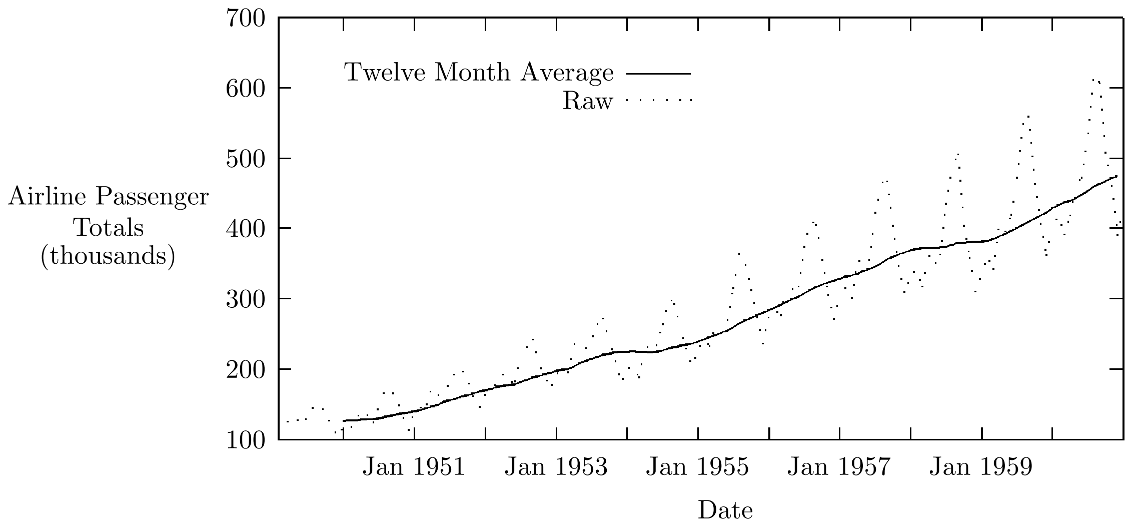


Figure 3.14: Results of applying the 12th-order FIR filter to the airline passenger data. The trend has been preserved and the annual cycles have been removed.

more difficult to analyze.

3.3.2 Infinite Impulse Response Filters

An *Infinite Impulse Response* (IIR) filter, also known as a *Autoregressive* (AR) filter, consists of a tapped feedback line of series-connected unit delay elements (Figure 3.15). The presence of feedback means that the response to a unit impulse will decay, but may never reach zero. Given an input series $\{x_t\}$ and an output series, $\{y_t\}$, the relationship between the two in an IIR filter is given by:

$$y_t = \sum_{k=1}^{N} a_i y_{t-k} + x_t \tag{3.21}$$

where a_k are weights chosen by the filter designer. The z-transform of the IIR filter is given by

$$X(z) = \frac{1}{1 - \sum_{k=1}^{N} a_k z^{-k}}$$

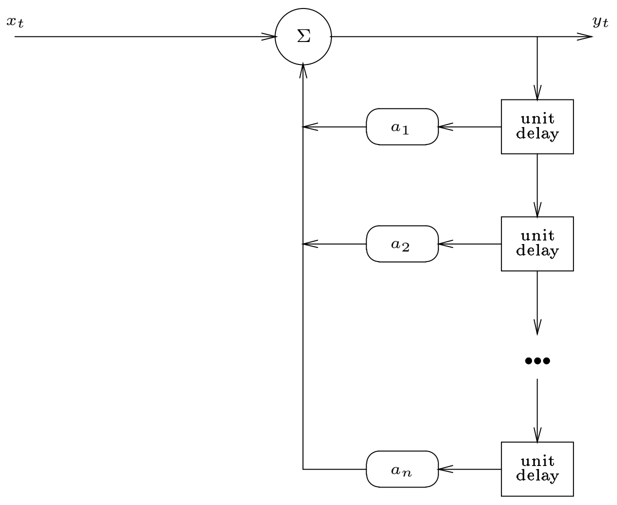


Figure 3.15: IIR Filter Block Diagram. The input time-series, x_t, passes directly to the output; the output time-series, y_t, is delayed, scaled, and summed with itself.

and its resulting frequency response (when $z = e^{i\omega}$) by

$$\begin{aligned} X(i\omega) &= \frac{1}{1 - \sum_{k=1}^{N} a_k e^{-ik\omega}} \\ &= \frac{1}{1 - \sum_{k=1}^{N} a_k \left[\cos(k\omega) - i\sin(k\omega)\right]} \end{aligned}$$

Note that this filter contains only *poles*, i.e. points where some values of z^{-k} will result in infinite $X(z)$. These are points of instability that must be considered carefully in the filter design.

A common IIR filter is the *Exponentially-Weighted Moving-Average* (EWMA) filter. Despite the name, it is indeed an IIR rather than an FIR filter, as I now demonstrate. An EWMA filter is a first-order IIR filter with the following input-output relationship:

$$y_t = y_{t-1} + \eta(x_t - y_{t-1}) \tag{3.22}$$

Here, η is called the *exponential weighting factor*, and is a number between 0 and 1. When $\eta = 0$, new data has no effect on the filter output, whereas when $\eta = 1$, the filter output tracks the data input exactly. Between these two extremes, the filter output is a smoothed version of the input, with more-recent inputs affecting the filter response to a greater degree as η increases in value. At $\eta = 1$, the EWMA filter is identical to a simple moving average (a FIR filter); for

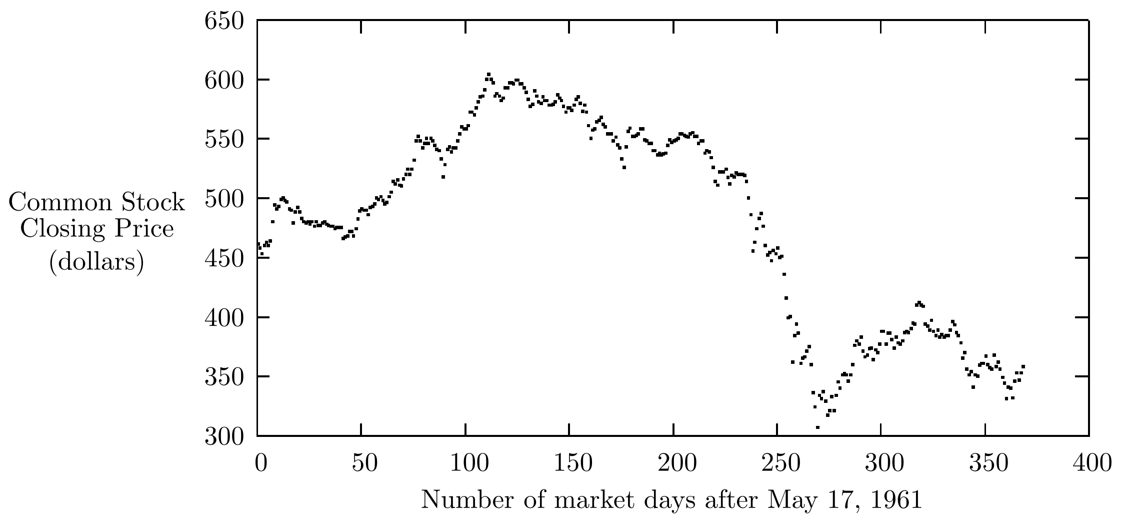


Figure 3.16: Daily IBM Common Stock closing prices from May 17, 1961 to November 2, 1962. No cyclic behavior is apparent in this noisy data, but the data follows a clear trend.

values of $\eta > 1$, older data points receive greater weight than those that are newer. (These two possibilities—$\eta \geq 1$—are not considered further in this thesis.) The EWMA equation can be brought into standard form with $a_0 = (1 - \eta)$ and input gain of η as follows:

$$y_t = (1 - \eta)y_{t-1} + \eta x_t \tag{3.23}$$

Weighting data points differently can be useful if newer data points are more indicative of the true value of the desired quantity than those that are older. For example, the trend of the passenger data in Figure 3.12 remains roughly constant, as shown in Figure 3.14 (i.e., its slope does not significantly change), so every data point of Figure 3.12 possesses the same "value" with respect to determining the underlying trend. However, if the trend is expected to be changing, then more-recent data points should reflect the instantaneous trend more accurately than older data points will. In this case, we should weight the data such that older data affects the filter output to a lesser degree.

One application for such a filter is noisy data that does not exhibit definite cyclic behavior, such as the data in Figure 3.16. This figure depicts the closing price of IBM common stock from

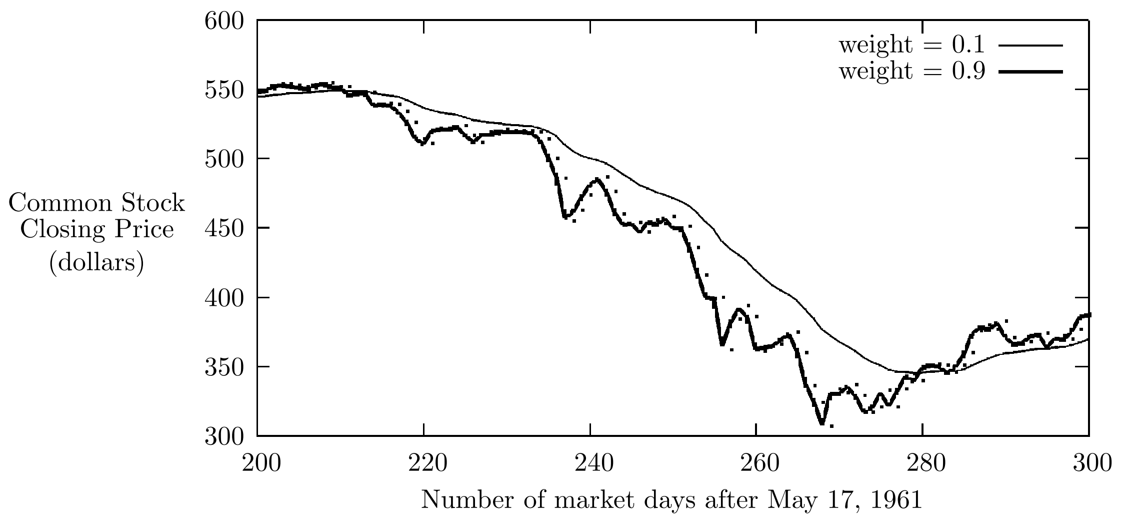


Figure 3.17: Daily IBM Common Stock closing prices after May 17, 1961, filtered by two exponentially-weighted moving averages. The lower the exponential weight, the less new data points affect the filter output, and the smoother that output becomes.

May 17, 1961, to November 2, 1962[11]. There is a clear trend over an extended period of time, but daily variations result in high variance around the trend. Because of the lack of obviously cyclic components in the data, we would expect an EWMA filter to be a good choice in order to distill the trend. Figure 3.17 shows a small portion of this data, overlaid with the output from EWMA filters with weights of 0.1 and 0.9, respectively. This Figure illustrates the effect of the exponential weight: a smaller weight makes the filter less responsive to noise, but introduces a bigger phase delay.

Note that a weighted FIR filter could have been used instead of the EWMA filter. The advantage of the EWMA approach, and indeed any IIR filter, is its easy adaptation to *recursive* computation: only the previous filter output and the current data point in the EWMA filter are needed in order to compute the next output, in contrast to the computation of the moving-average FIR filter, which requires n data points for an n^{th}-order filter.

The primary drawback of IIR filters is instability. This includes instablities due to the existence of poles in the filter equations, as well as numerical instability in software realizations.

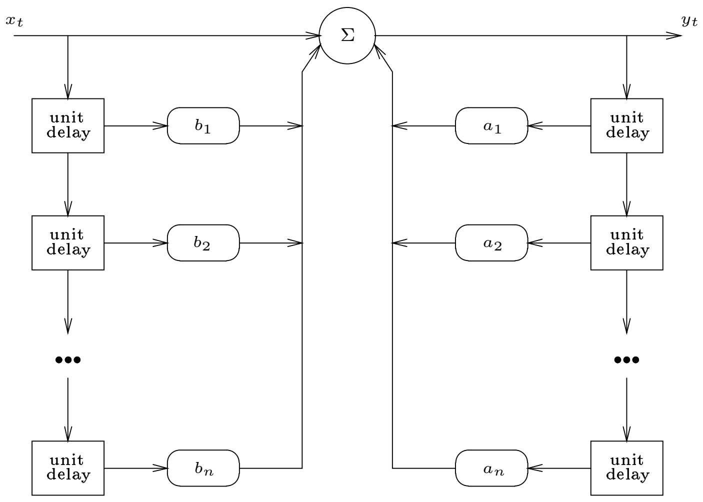


Figure 3.18: ARMA Filter Block Diagram. The input series, x_t, passes through combined IIR (AR) and FIR (MA) stages to form the output series, y_t.

Nonetheless, early filtering mechanisms were mostly IIR designs due to technological pressures, so an extensive body of literature exists on the design of IIR filters, as well as on methods of transforming standard IIR designs into FIR counterparts.

3.3.3 ARMA Filters

ARMA filters are the most general filtering paradigm, consisting of both IIR (AR) and FIR (MA) stages (Figure 3.18). ARMA filters possess both poles and zeroes. Generally speaking, ARMA filters are most useful when the exact characteristics of the signal to be blocked are known in advance. From a frequency-domain perspective, this means that ARMA filters could be used, for example, if a 60 Hz sinusoid signal were to be removed from an wide-spectrum input signal.

The design and analysis of ARMA filters is complex and extensively researched. For this reason, and because precise frequency-filtering characteristics are not needed in this thesis, I do not cover ARMA filters in any more detail. Interested readers should consult any reference on linear systems, such as [34].

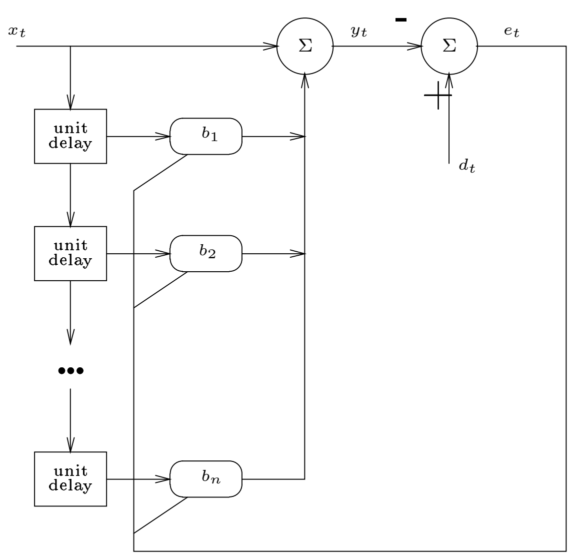


Figure 3.19: An *adaptive* FIR filter. The filter coefficients are adjusted to minimize the difference between an objective function, d_t, and the filter output, y_t.

3.3.4 Adaptive Filters

The filters in the previous sections are distinguished from one another by their structure (FIR or IIR) and by the scaling constants at the filter taps. We have seen that by changing the constants, we can transform one type of filter (e.g., the exponentially-weighted, moving average IIR filter) into a different type (a simple moving average FIR filter). In the general case, it is obvious that by manipulating the a_k and b_k constants of the ARMA filter paradigm, it is possible to create any possible variant or combination of IIR and FIR filters. The next obvious step is to create a feedback system whereby the *filter itself* can adapt its structure to produce an optimal result. Filters with this capacity, called *adaptive* filters, are feedback control systems with a reference signal (the optimal filter output given the statistical characteristics of the filter input), a controller, and a plant (the filter itself). Most adaptive filters use the FIR paradigm because of the difficulty of adjusting the coefficients in IIR filters.

A block diagram depicting the basic adaptive FIR scheme is shown in Figure 3.19. This is a FIR filter with the addition of an *desired output function*, d_t, and a second summation stage to compute the error between the desired output and the filter output. The error, e_t, is used to adjust the values of $b_1, b_2, \ldots, b_n$. To adjust the filter coefficients, one uses an *objective function* that quantifies the difference between the desired and actual outputs of the filter.

A widely used objective function is the *mean-square error* (MSE), which is defined as

$$E[e_t^2] = E[d_t^2 - 2d_t y_t + y_t^2] \tag{3.24}$$

Recall that the FIR filter equation is as follows:

$$y_t = \sum_{k=0}^{N-1} b_i x_{t-k} \tag{3.25}$$

$$= \mathbf{b}_t^T \mathbf{x}_t \tag{3.26}$$

where Equation (3.26) is the FIR equation in matrix form, in which

$$\mathbf{b}_t = [b_0 b_1 b_2 \ldots b_{N-1}]^T$$

and

$$\mathbf{x}_t = [x_t x_{t-1} x_{t-2} \ldots x_{t-N+1}]^T$$

representing the tap weight and input vectors, respectively. Using these relationships, we can rewrite the MSE objective function as:

$$E[e_t^2] = E[d_t^2] - 2\mathbf{b}_t^T \mathbf{p} + \mathbf{b}_t^T \mathbf{R} \mathbf{b}_t \tag{3.27}$$

where $\mathbf{p} = E[d_t \mathbf{x}_t]$ is the cross-correlation vector between the desired and input signals, and $\mathbf{R} = E[\mathbf{x}_t \mathbf{x}_t^T]$ is the autocorrelation matrix of the input signal. Note that $E[e_t^2]$ is a quadratic function of the filter coefficients $\mathbf{p}$ and $\mathbf{R}$. Hence, an obvious way to build an adaptive FIR filter is to use a gradient descent approach to find the value of $\mathbf{b}_t$ that minimizes $E[e_t^2]$. Both the Wiener and the Least-Mean-Square (LMS) filters use this approach; the Wiener filter is the classic realization of a gradient-descent adaptive filter, and the LMS filter is its most widely used variant[39].

The LMS filter algorithm is a method to update $\mathbf{b}_t$. In this algorithm, one generally starts by setting all the filter coefficients to zero and then updating them based upon the computed error signal, e_t, according to the following algorithm:

Algorithm 1 The Least Mean Square objective function

```
LEAST-MEAN-SQUARE()
1  $\mathbf{x}_0 \leftarrow [\,0\;0\;\ldots 0\,]^T$
2  $\mathbf{b}_0 \leftarrow [\,0\;0\;\ldots 0\,]^T$
3  for $t > 0$
4  do $e_t \leftarrow d_t - \mathbf{x}_t^T\mathbf{b}_t$
5      $\mathbf{b}_{t+1} \leftarrow \mathbf{b}_t + 2\mu e_t \mathbf{x}_t$
```

Here, μ is called the *convergence factor*. This parameter is analogous to the proportional gain of a P-type controller; it determines the speed of convergence, and the stability of the filter as it converges.

It should be clear that the proper choice of d_t, the desired output signal, is required in order to obtain useful results from an adaptive filter such as the LMS filter. To illustrate how this choice is made, consider the problem of measuring a fetus's heartbeat. In this application, the mother's heartbeat is much stronger than the fetal heartbeat. Hence the problem is one of removing the maternal heartbeat signal from an aggregate signal, and represents a noise cancellation problem. The desired output in this case would be a measurement of the mother's heartbeat, taken from chest sensors, while the filter input (x_t) would be the aggregate signal of mother and fetus as measured from an abdominal sensor. The filter output (y_t) would be an estimate of the maternal heartbeat in the aggregate signal, which can then be subtracted from the aggregate in order to obtain the fetal heartbeat; the "error" output (e_t) is the desired fetal heartbeat. Note that no assumptions are made about the filter inputs or the distribution of the noise signal. The result is that the LMS algorithm may converge slowly to the optimum filter coefficients, i.e. for a fixed convergence factor, and LMS filter may converge quickly for one input signal but slowly for another.

The *Kalman Filter* is an LMS filter that makes *a priori* assumptions about the structure of the perturbing noise and about the model of the system generating the filter input in order to speed convergence. In particular, one Kalman Filter assumption is that the noise in the system is gaussian. This assumption, combined with an assumed state model, negates the necessity for an explicit desired-output signal. The Kalman Filter is computationally more expensive than the basic LMS method, but its fast convergence has made it the most widely used LMS approach. It operates by taking a set of noisy measurements, the assumption that the noise is gaussian (although the mean and variance from each sensor may be different), and the knowledge of the system dynamics (from the system state equations) to form an estimate of the true system state. A simple example of the application of the Kalman filter would be the measurement of a fixed DC voltage using several different voltmeters. We could assume a linear state model (since the voltage is DC and unchanging), and could calculate gaussian statistics (mean and variance) from ongoing measurements from each of the voltmeters. The Kalman filter would take these statistics to find the best fit of the data to the linear model, i.e. a recursive least-squares regression. The power of the Kalman approach becomes apparent when we note that underlying the state model can be *any* linear function of time, and any number of measurements can be included. The Extended Kalman Filter relaxes even this requirement by allowing the use of nonlinear state models.

In summary, the filters covered in this section are useful in the following ways. FIR and IIR filters are applicable when the filtering problem is straightforward and intuitive, the ARMA paradigm is best when the frequency-domain characteristics of the problem are known, and adaptive filters are best when the statistical characteristics of the noise to be filtered are well understood. These are generalizations at best, but are nevertheless useful when seeking a filter for a novel application.

Chapter 4

Distribution Control

In this chapter, I outline the design of an adaptive controller whose goal is to shape the stationary distribution of a Birth/Death Markov Chain to match a QoS specification. I begin with formal definitions, followed by a description of the necessary and sufficient conditions for successful distribution control. Using this background, I describe the mathematics of distribution shaping, and then show how this translates into a control architecture. In the final section, I discuss the convergence of Markov Chains using novel simulation results. This Chapter depends on the background in Chapter 3; some information is repeated for clarification but without elaboration.

4.1 Definitions

The details in this section formalize the notions of *Quality of Service specifications*, *empirical distributions*, and *transition ratios* that are used in this thesis. In addition, the notation for several quantities of interest for control is introduced, namely the *free* and *constrained probablity mass*, the *instantaneous Chain state*, the *estimated transition probabilities*, the *desired request probability*, and the *necessary drop probability*.

Recall from the Chapter 3 that a Markov Birth/Death Chain is a Markov Chain with transitions only between states that differ in index by one (Figure 4.1). Formally, given a finite state set $\mathcal{S} = \{0, 1, 2, \ldots, n\}$, a *finite discrete-time Markov Birth/Death Chain* is a stochastic process $\{\mathcal{M}_t, t \in \mathbb{Z}^*\}$, that takes on values from $\mathcal{S}$. In order to guarantee the existence of and

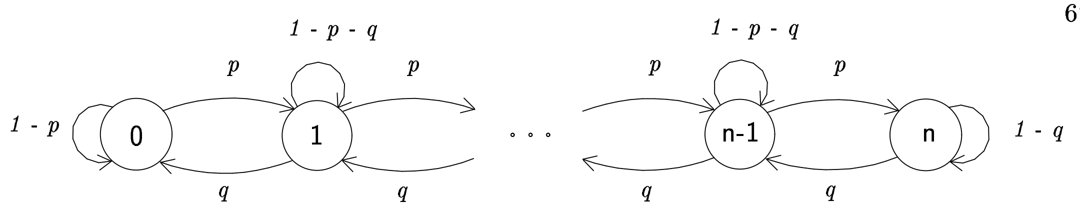


Figure 4.1: A general Markov Birth/Death Chain

convergence to a stationary probability distribution, I consider only finite Chains. This is key to establishing controllability.

The state transition matrix of $\mathcal{M}$, introduced in Section 3.1.3, is a stochastic $n+1$ by $n+1$ matrix $\mathcal{P}$ where element $\mathcal{P}_{ij}$ denotes the probability of making the transition $i \rightarrow j$ for $i, j \in \mathcal{S}$:

$$\mathcal{P} = \left| \begin{array}{llllllll} 1-p & p & 0 & 0 & . & 0 & 0 & 0 \\ q & 1-p-q & p & 0 & . & 0 & 0 & 0 \\ 0 & q & 1-p-q & p & . & 0 & 0 & 0 \\ . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & 0 & . & q & 1-p-q & p \\ 0 & 0 & 0 & 0 & . & 0 & q & 1-q \end{array} \right|$$

A *Quality of Service specification* on $\mathcal{M}$ is a discrete function $Q : \overline{\mathcal{S}} \rightarrow [0,1]$. It satisfies the condition $\sum_{i=0}^{n} Q(i) = 1.00$, and is defined for every element of the domain $\overline{\mathcal{S}} = \{\mathcal{S} - \{0, \ldots, \beta - 1\}\}$, and $\beta \in \{2, \ldots, n\}$. Hence Q is a partial function on $\mathcal{S}$ that is defined over the interval $[\beta, n]$; I refer to state β as the *bottleneck state* for reasons that will become apparent in Section 4.3.2. Note that no assumption of monotonicity is made here regarding Q, but in practical applications we normally assume that Q is monotonically decreasing in s (because we wish to degrade performance in a controlled fashion).

The *constrained probability mass*, π_c, is the sum of the elements in the QoS specification, while the *free probability mass*, π_f, is one minus this value:

$$\pi_c = \sum_{i=\beta}^{n} Q(i) \tag{4.1}$$

$$\pi_f = 1.00 - \pi_c \tag{4.2}$$

Because the QoS specification is a made upon a distribution, π_c must never sum to a value greater than 1.00. This specification is used as the "reference signal" for the control systems in this thesis: successful control means that the stationary distribution of a controlled Birth/Death Markov Chain is equal to (or at least no greater than) the Quality of Service specification in the set $\overline{\mathcal{S}}$. Usually, we do not want the stationary distribution to be lower than $\overline{\mathcal{S}}$, because that would mean that too many resource requests were being denied.

The *empirical distribution* of $\mathcal{M}$ is related to its stationary distribution through $\mathcal{P}$. As described in Section 3.1.3, the stationary distribution of $\mathcal{M}$, denoted by $\boldsymbol{\pi}$, can be determined by computing $\mathcal{P}^t$ and allowing $t \to \infty$, where t denotes the number of discrete steps (state changes) made by the Chain.[1] If t is finite, then $\mathcal{P}^t$ is called a *transient distribution matrix* and element $\mathcal{P}^t_{ij}$ represents the probability that the Chain is in state j after t steps, conditioned upon starting in state i. As t approaches infinity, all the rows of this matrix converge to the same value, $\boldsymbol{\pi}$. (This is a result of the memorylessness of the Markov Chain.) The *empirical* distribution of a Markov Chain is a transient distribution that is calculated over a time period Δt by recording the percentage of time that the Chain spends in each state, and then normalizing these values by Δt. Calculated in this manner, the empirical distribution is an $n+1$-element row vector denoted by $\mathcal{E}^{\Delta t}$ in which element $\mathcal{E}^{\Delta t}_i$ indicates the empirically measured probability that the Chain is in state i after Δt steps, from some *unspecified* starting state. ($\mathcal{E}^{\Delta t}$ will be referred to as the *Δt-empirical distribution* in this thesis.) Hence the empirical distribution can be thought of as a sample of one row of the transient distribution matrix. The *convergence rate* of the Markov Chain, as used in this thesis, is a measure of how quickly $\mathcal{E}^{\Delta t}$ approaches $\boldsymbol{\pi}$ on the interval $\overline{\mathcal{S}}$. Section 4.4 covers Chain convergence in greater detail.

[1] Recall that $\boldsymbol{\pi}$ is a vector: in this thesis, $\boldsymbol{\pi}$ always refers to the stationary distribution of a Markov Chain, and never the value 3.14159265 . . .

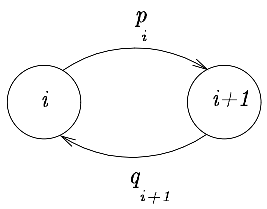


Figure 4.2: Annotations on the states and transitions of a Markov Birth/Death Chain for computing the transition ratio function: $\omega(i) = p_i/q_{i+1}$. Transition probabilities are subscripted with the index of the *source* state.

Mathematical manipulation of the empirical and stationary distributions of Birth/Death Markov Chains is made possible through the use of the *transition ratio function* that relates the stationary distribution to the transition probabilities. This function, $\omega : \{\mathcal{S} - n\} \rightarrow [0, 1]$, a consequence of the time-reversibility of Birth/Death Chains, is defined as follows:

$$\omega(i) = \frac{\pi_{i+1}}{\pi_i} \tag{4.3}$$

$$= \frac{p_i}{q_{i+1}} \tag{4.4}$$

This relationship is illustrated in Figure 4.2. Note that Equation (4.4) is equal to p/q since all upward and downward transitions in the Chains used in this thesis are assumed to occur with the same probabilities p and q, respectively. Intuitively, ω can be thought of as the tendency of the model (and the system it describes) to enter an overloaded condition: if $\omega > 1.00$, then $p > q$, leading to more probability mass accumulating in higher-numbered states (where higher-numbers mean more resources in use, and hence a "more loaded" operating condition). The *transition ratio matrix*, denoted by $\mathcal{R}$, is the matrix of transition ratios for each state in the QoS specification, i.e. $\mathcal{R}_i = \omega(i)$ for every $i \in \overline{\mathcal{S}}$.

The remaining notation necessary for this Chapter is related to the control algorithm. The *instantaneous Chain state*, s, is the current value of the Markov Chain state. Hence $s \in \mathcal{S}$ at all times. The *estimated input request probability*, $\hat{p}_{in}$, is an estimate of the value of p **before** any control is applied, while the *estimated service probability*, $\hat{q}$, is an estimate of the value of q. Both of these estimates are made by linear filtering mechanisms such as those described in

Section 3.3.4. Finally, the control system computes the *desired request probability*, p_d, in order to determine the value to which p must be changed in order to guarantee that $\mathcal{E}^{\Delta t} \leq Q(s)$. This value p_{drop} is the *necessary drop probability* that is used by the admission stage actuator in order to determine whether or not to drop an incoming request.

4.2 Necessary and Sufficient Conditions for Control

Certain conditions must be met in order to guarantee the success of the control paradigm described in the following sections. These consist of structural constraints on the resource system, which define how resources may be managed, as well as control requirements that must be satisfied in order to achieve effective control.

Structural constraints are conditions that the resource management software system must meet in order for the Markov Birth/Death Chain to be a good model of it. These conditions are:

(1) **FINITENESS**: the resource system must possess a finite number of states;

(2) **SERIALIZATION**: state-changing events must occur singly; and

(3) **UNIT CHANGES**: state changes must occur in unit increments.

Collectively, these conditions guarantee the applicability of the Birth/Death Chain topology: they are individually necessary and jointly sufficient. The Finiteness condition ensures that the states of the Chain will be discrete, the Serialization condition ensures that state-changing events do not occur simultaneously, and the Unit Changes condition guarantees that the Chain can transition only into state $i \pm 1$ from state i when an allocation or deallocation takes place.

The two control conditions are *observability* and *controllability*. In the design of the control paradigm that follows, I have assumed that state i of the Markov Chain model of the resource management system represents i resources in use, that the transition probability p represents the probability of a single resource allocation occuring, and that the probability q represents a single resource deallocation. Each of these quantities must be *observable*, i.e. they must be either directly measurable or at least amenable to estimation. In order to control the instantaneous state to achieve distribution shaping, I further assume that we may modify p. Hence p must be *controllable*: the control system must be able to affect its value directly. With respect to q, I assume that q *will never be modified by the control system.* This is an important—and reasonable—assumption. q represents the probability with which admitted resource requests are being serviced, which cannot be increased (since the resource management system is presumably already servicing requests as fast as it can), nor should it be decreased (since reducing the service probability implies slowing down the resource manager). Note that I assume only that the controller does not modify q. However, the operating environment—in particular, the speed of the resource manager—*will* result in changes to q.

In the following sections, I assume that these conditions are all met. Explicit examples will be provided in Chapter 5 to show that not only is this possible, but it is also common in practice.

4.3 Birth/Death Chain Distribution Shaping

I now describe the steps needed to control the distribution of a plant that can be modeled by a Birth/Death Markov Chain to meet a prespecified QoS specification. These steps are composed of an initialization component followed by active control at every resource request arrival. The idea underlying the control algorithm is to manipulate the ratio of arriving to serviced resource requests in order to maintain $\boldsymbol{\pi}$ (the stationary distribution) equal to Q (the QoS specification). Note that this is very different from directly controlling the empirical distribution itself: ratio control, as described in this section, is a *principled approach* with *provable* effects on the stationary distribution. By Equations (4.3) and (4.4), we know that $\mathcal{R}_s = p_s/q_{s+1} = \pi_{s+1}/\pi_s$. Hence by maintaining the ratio p_s/q_{s+1} equal to $Q(s+1)/Q(s)$ wherever Q is defined, the stationary distribution of the Markov Chain will be successfully shaped to fit $\mathcal{Q}$, and the QoS specification will be satisfied. This is done by *observing* p, q, and s (the resource arrival and service probabilities, and the instantaneous resource state), while *controlling*

p by probabilistically dropping resource requests.

The desired request arrival probability, p_d, that will achieve this is calculated as follows. Assuming that q is independent of state, p_d is found from Equation (4.3) to be:

$$p_d = \mathcal{R}_s \hat{q} \tag{4.5}$$

where I have substituted the filter-estimated service rate, $\hat{q}$, for the unknown true service rate, q. For the purposes of control, we must relate this to the request *admission probability*, i.e. the probability with which an arriving request should be admitted to the resource management system. Requests arrive with probability p (estimated as $\hat{p}_{in}$); by independence of events, $p_d = \hat{p}_{in} p_{admit}$, or:

$$\begin{align} p_{admit} &= \frac{p_d}{\hat{p}_{in}} \tag{4.6} \\ &= \frac{\mathcal{R}_s \hat{q}}{\hat{p}_{in}} \tag{4.7} \end{align}$$

The probability with which an individual requst must be dropped—i.e., refused admission to the resource management system—is therefore equal to:

$$\begin{align} p_{drop} &= 1.00 - p_{admit} \tag{4.8} \\ &= 1.00 - \frac{\mathcal{R}_s \hat{q}}{\hat{p}_{in}} \tag{4.9} \end{align}$$

Note that Equation (4.9) uses four quantities: the instantaneous state (s), the estimate of the resource service probability ($\hat{q}$), the estimate of the uncontrolled resource request arrival probability ($\hat{p}_{in}$), and the transition ratio matrix ($\mathcal{R}$). The first is measured, the second and third are estimated, and the last is computed.

$\mathcal{R}$ values for most states are calculated from the QoS specification, Q, using Equation (4.3), but one value—$\mathcal{R}_{\beta-1}$—cannot be determined in this manner, since $\beta - 1 \notin \overline{\mathcal{S}}$, the domain of Q. As will be shown in Section 4.3.2, it is possible to determine a closed-form expression for $\mathcal{R}_{\beta-1}$, but the associated computational complexity is high. Here, another method is used: a model-reference closed-loop feedback control system. The remainder of this section discusses the initialization and support algorithms needed for this control.

4.3.1 Initialization

Several quantitites are needed for the control algorithms that follow. These need only be computed once, in an *initialization* step. The requisite quantities are the free and constrained probability mass in the system as well as the transition ratio matrix. We compute the transition ratio matrix, $\mathcal{R}$, using the assumption that the stationary probability distribution is equal to the QoS specification, $Q(x)$; this will ensure that the ratios will be correct for maintaining the stationary distribution at the QoS level, according to Equation (4.9).

The Initialization algorithm, INITIALIZE(), runs in time $O(n)$. Line 1 sums the constrained probability mass, line 2 computes the free probability mass, and lines 3 and 4 calculate $\mathcal{R}$ for every required state except $\beta - 1$ itself.

Algorithm 2 The control initialization algorithm

```
INITIALIZE()
1  $\pi_c \leftarrow \sum_{i=\beta}^{n} Q(i)$
2  $\pi_f \leftarrow 1.00 - \pi_c$
3  **for** $i \leftarrow \beta$ **to** $n - 1$
4  **do** $\mathcal{R}_i \leftarrow \frac{Q(i+1)}{Q(i)}$
```

4.3.2 Control

The control step is executed every time a transition into a constrained state is possible, i.e., when a resource request, if admitted, would result in a state change $s_t \rightarrow s_{t+1}$ and $s_{t+1} \in \overline{\mathcal{S}}$. This control step consists of up to three parts:

(1) if the current instantaneous state (before making the transition) is $s = \beta - 1$, then estimate the correct value of $\mathcal{R}_{\beta-1}$;

(2) compute the correct resource-request drop-probability, p_{drop}; and

(3) drop the request with probability p_{drop}.

$\mathcal{R}_{\beta-1}$ can be estimated either by a direct computation or by feedback control. The former applies whenever p and q are unchanging (or changing very slowly), but it poses an onerous computational burden in the usual case when these quantities are dynamic (due to an exponentiation requirement), and it is therefore unattractive as a repeated control step. Feedback control is applicable in all conditions and has modest computational requirements, but it is more complex to describe and to use because it involves multiple stages. The direct computation algorithm SIMPLE-BOTTLENECK-ESTIMATE() calculates an initial value of $\mathcal{R}_{\beta-1}$ for the feedback control algorithm. The feedback controller FEEDBACK-BOTTLENECK-ESTIMATE() uses measurements of the system behavior (viz. the empirical distribution of resource usage) and a PID loop to iteratively *steer the value of $\mathcal{R}_{\beta-1}$ towards a correct, instantaneous value.* The idea of this two-stage approach is to avoid startup transients, and the overall computation proceeds as follows. We first compute the free probability mass, π_f, and then relate π_f to $\pi_{\beta-1}$ to determine a closed-form expression for $\pi_{\beta-1}$. Given that expression, I then use Equation (4.3) to determine $\mathcal{R}_{\beta-1}$. Given estimates of p and q ($\hat{p}_{in}$ and $\hat{q}$, respectively), I compute the free probability mass as follows:

$$\begin{aligned}
\pi_f &= \pi_0 + \pi_1 + \ldots + \pi_{\beta-3} + \pi_{\beta-2} + \pi_{\beta-1} && (4.10)\\
&= \left(\frac{q}{p}\right)^{\beta-1} \pi_{\beta-1} + \left(\frac{q}{p}\right)^{\beta-2} \pi_{\beta-1} + \ldots + \left(\frac{q}{p}\right)^{2} \pi_{\beta-1} + \frac{q}{p}\pi_{\beta-1} + \pi_{\beta-1} && (4.11)\\
&= \sum_{i=0}^{\beta-1} \left(\frac{q}{p}\right)^{i} \pi_{\beta-1} && (4.12)
\end{aligned}$$

where Equation (4.11) follows from the detailed balance equation for Birth/Death Markov Chains from Equation (3.6). Summing the series, I conclude that:

$$\pi_{\beta-1} = \frac{\pi_f(1-\Omega)}{1-\Omega^{\beta-1}} \tag{4.13}$$

where $\Omega = q/p$. Since the actual values of p and q are not available, however, I use $\hat{\Omega} = \hat{q}/\hat{p}_{in}$. Using this value, Equation (4.3), and the fact that I wish π_β to be equal to $Q(\beta)$, we have

$$\begin{aligned}
\mathcal{R}_{\beta-1} &= \omega(\beta-1) && (4.14)\\
&= \frac{\pi_\beta}{\pi_{\beta-1}} && (4.15)\\
&= \frac{Q(\beta)(1-\hat{\Omega}^{\beta-1})}{\pi_f(1-\hat{\Omega})} && (4.16)
\end{aligned}$$

Equation (4.16) is the basis of the SIMPLE-BOTTLENECK-ESTIMATE() algorithm:

Algorithm 3 The simple estimation algorithm for $\mathcal{R}_{\beta-1}$

```
SIMPLE-BOTTLENECK-ESTIMATE()
1  $\hat{\Omega} \leftarrow \hat{q}/\hat{p}_{in}$
2  $\mathcal{R}_{\beta-1} \leftarrow \frac{Q(\beta)(1-\hat{\Omega}^{\beta-1})}{\pi_f(1-\hat{\Omega})}$
3  return $\mathcal{R}_{\beta-1}$
```

The running time of a single invocation of SIMPLE-BOTTLENECK-ESTIMATE() is dominated by the time required to compute the exponent. Because β is a state (and hence a non-negative integer in the finite state space of the Markov Chain) this can be reduced to $O(\log \beta)$ using repeated squaring (assuming $O(1)$ to perform the multiplications). Hence the running time of SIMPLE-BOTTLENECK-ESTIMATE() is a function of the size of the Quality of Service specification: the *more* states that are specified, the *faster* SIMPLE-BOTTLENECK-ESTIMATE() runs. (Recall that β, the bottleneck state number, becomes *smaller* as the size of the QoS specification becomes *larger*.) However, since the size (as measured by the number of states) of the QoS-specification is usually much smaller than n (i.e. $|\bar{\mathcal{S}}| \ll |\mathcal{S}|$), $\beta \approx n$ and thus I expect the running time of SIMPLE-BOTTLENECK-ESTIMATE() to be close to $O(\log n)$. This makes the algorithm more expensive when p and q change often, which I expect to be true in practice. Moreover, multiplication incurs a significant time penalty in practice, further reducing the usefulness of SIMPLE-BOTTLENECK-ESTIMATE() as the primary estimator function for $\mathcal{R}_{\beta-1}$ in dynamic environments. In summary, SIMPLE-BOTTLENECK-ESTIMATE() is primarily useful as an initialization algorithm for $\mathcal{R}_{\beta-1}$, rather than as a repeated control step.

The FEEDBACK-BOTTLENECK-ESTIMATE() algorithm calculates $\mathcal{R}_{\beta-1}$ with a lower computational overhead than that required by SIMPLE-BOTTLENECK-ESTIMATE(). As the name implies, a feedback mechanism is used instead of a direct calculation of $\mathcal{R}_{\beta-1}$. This mechanism minimizes the difference between the *desired distribution*, $Q(x)$, and the empirical distribution, $\mathcal{E}^{\Delta t}$, by adjusting $\mathcal{R}_{\beta-1}$ upwards or downwards when there is too much or too little probability mass in the constrained region, respectively.

The metaphor of *flowing probability mass* is useful for understanding this approach. By definition, the total probability mass in a distribution must sum to 1.00. Hence probability mass is *preserved*: if a Markov Chain state's stationary probability goes down from one time step to the next, then that probability mass must be accumulating somewhere else in the stationary vector. In some sense, it *flows* between states when the transition probabilities change. For example, if the ratio of p to q increases, then we expect probability mass to flow from lower-numbered states towards higher-numbered states. In a controlled Chain with a QoS specification, this flow is regulated in a manner remniscent of a valve in a water pipe: floods are reduced to trickles.

Feedback control of Markov Birth/Death Chains involves leveraging the time-reversibility of these Chains to manage the flow of probability mass between Chain states. Recall that the relationship between the stationary probabilities of neighbor states in the Birth/Death Markov Chain is given by the detailed-balance equation, Equation (3.6), and that I use this fact to calculate the ratio matrix, $\mathcal{R}$. By the lines 3 and 4 of INITIALIZE() on page 73, the value of the ratio matrix for any state i in the QoS specification ($\mathcal{R}_i$) is related to those of its neighbors ($\mathcal{R}_{i-1}$ and $\mathcal{R}_{i+1}$) by a constant factor. For example, consider the following QoS specification and control ratios:

$$\begin{aligned} Q(5) &= 0.40 & \mathcal{R}_5 &= 0.30/0.40 \\ Q(6) &= 0.30 & \mathcal{R}_6 &= 0.20/0.30 = 8/9\mathcal{R}_5 \\ Q(7) &= 0.20 & \mathcal{R}_7 &= 0.10/0.20 = 2/3\mathcal{R}_5 \\ Q(8) &= 0.10 \end{aligned}$$

Thus every state's value in $\mathcal{R}$ is related to $\mathcal{R}_{\beta-1}$ by some constant factor. By implication, if too much probability mass is allowed to flow into state β (by setting $\mathcal{R}_{\beta-1}$ too high), then the difference between the amount of probability masses in the resulting empirical distribution and the the QoS specification ($Q(x)$) will be directly related to $\mathcal{R}_{\beta-1}$. (Recall from Section 4.1 that the empirical distribution is denoted by $\mathcal{E}^{\Delta t}$ and hence referred to as the Δt-empirical distribution.) Controlling the value of $\mathcal{R}_{\beta-1}$ is thus equivalent to controlling the flow of probablity mass into β, and consequently the amount of mass in all of the constrained states. This is why

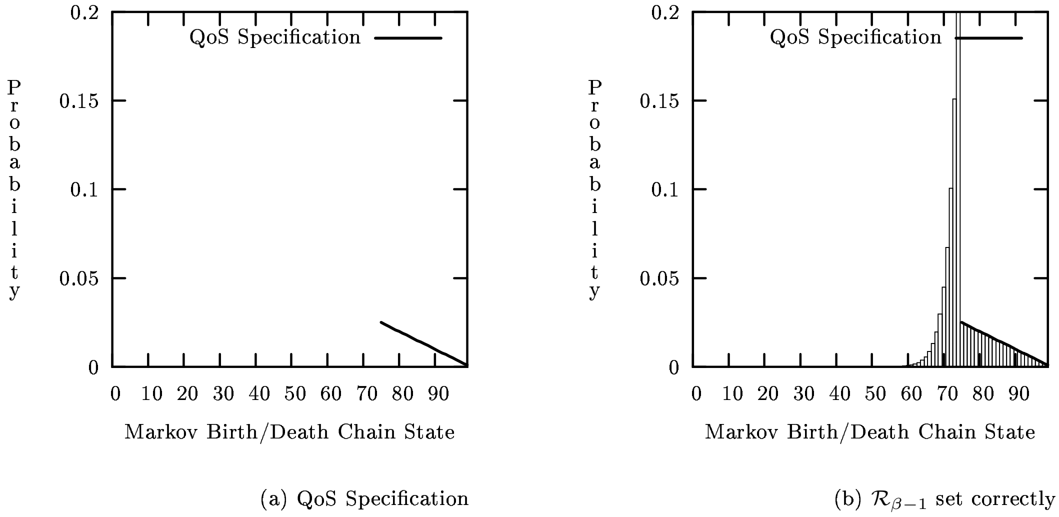
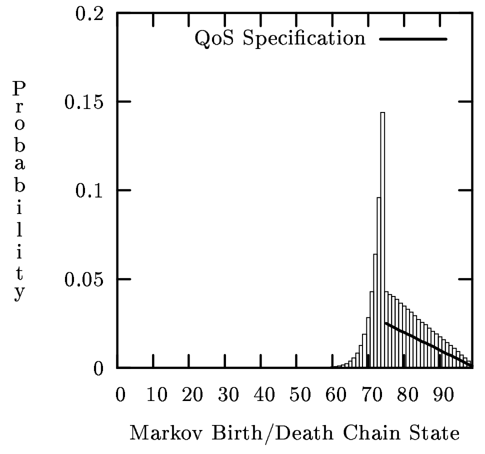


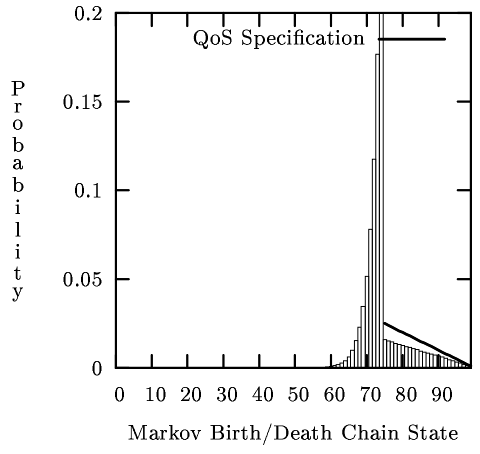
Figure 4.3: Linear QoS specification and a probability distribution from a Markov Birth/Death Chain that satisfies the specification. The key to achieving this is setting $\mathcal{R}_{\beta-1}$ to the correct value (depending on the values of p and q). Refer to Figure 1.1(b) for an illustration of the distribution of an uncontrolled Markov Birth/Death Chain.

I call β the *bottleneck state*, and why it plays such a large role in my control strategy.

This situation is illustrated in Figures 4.3 and 4.4, in which a control system is shaping the stationary distribution of a 101-state Markov Birth/Death Chain. Figure 4.3(a) depicts a linear QoS specification on the set of states $[75, 100]$, and is is being satisfied in Figure 4.3(b). The shaped stationary distribution in Figure 4.3(b) is the result of a controlling a Markov Chain with $q = 0.40$ and $p = 0.60$ in the unconstrained states. Without any control at all, the distribution would be the same as that shown in Figure 1.1(b). The value of $\mathcal{R}_{\beta-1}$ used in this controller is "correct" in the sense that exactly the right amount of probability mass has flowed into the constrained region. By contrast, Figures 4.4(a) and 4.4(b) illustrate the situation when the value of $\mathcal{R}_{\beta-1}$ is too low or too high, respectively. Note that the actual probability distributions in the constrained regions of these two examples are linearly scaled versions of the constrained distribution in Figure 4.3(b), and of the QoS specification, $Q(x)$, a result of the aforementioned linear relationship between the values of the transition ratio matrix. Note, too, that all three probability distributions—Figures 4.3(b), 4.4(a), and 4.4(b)—depict the free probability mass



(a) $\mathcal{R}_{\beta-1}$ set too Low



(b) $\mathcal{R}_{\beta-1}$ set too High

Figure 4.4: Probability distributions from Markov Birth/Death Chains in which $\mathcal{R}_{\beta-1}$ is set too low and too high, respectively, resulting in the accumulation of too much or too little probability mass in the QoS-constrained states. The vector difference between the QoS specification and the stationary distribution in these cases can be used with feedback control to determine the correct value of $\mathcal{R}_{\beta-1}$.

"piling up" in state $\beta - 1$ (state 74). This pileup of probability mass is due to the fact that the natural mass flow is strongly towards higher states ($0.60 > 0.40$), but the controller is managing the natural flow. This is, again, the motivation for the term *bottleneck state* in reference to state β. A smaller 'spike' would exist with a more liberal QoS specification, e.g. one that allowed more probability mass into the QoS-constrained states. These figures imply that the difference between the QoS specification and $\mathcal{E}^{\Delta t}$, the Δt-empirical distribution, of a running Markov Birth/Death Chain can be used as a measure of the error in $\mathcal{R}_{\beta-1}$ if the runtime—Δt—is long enough to allow the empirical distribution to approach its stationary limit.

The difference between the Δt-empirical distribution and the QoS specification—a metric for measuring success—can be computed in a variety of ways. This metric (which I denote by ϵ), can be as simple as the vector difference

$$\epsilon = \sum_{i=\beta}^{n} \mathcal{E}_i^{\Delta t} - Q(i) \tag{4.17}$$

This quantity will be positive when $\mathcal{R}_{\beta-1}$ is too high, close to zero when $\mathcal{R}_{\beta-1}$ is correct, and negative when $\mathcal{R}_{\beta-1}$ is too low. A drawback to using the vector difference is that it may be misleading if the Chain has not converged. To address this, the difference metric can be computed as the *relative pointwise distance* (**rpd**) defined in [105]:

$$\epsilon = \max_i \frac{\left|\mathcal{E}_i^{\Delta t} - Q(i)\right|}{Q(i)} \tag{4.18}$$

or the *total variation distance* (**tvd**) (a standard distribution measure):

$$\epsilon = \max_{\mathcal{T}\subseteq\mathcal{S}} \left| \sum_{i\in\mathcal{T}} \mathcal{E}_i^{\Delta t} - \sum_{i\in\mathcal{T}} Q(i) \right| \tag{4.19}$$

or any *norm*, such as L_1, L_2, or L_∞. With the exception of the vector difference, these are measures of the absolute value of the distance between the vectors, so other means must be used to determine the *sign* of that difference.[2] The function COMPUTE-ERROR() takes the simple approach, calculating erro signal, ϵ, as the difference between the empirical distribution and the QoS specification.

ϵ serves as input to a PID control system that modifies $\mathcal{R}_{\beta-1}$. This controller computes an offset $\hat{\epsilon}$ to $\mathcal{R}_{\beta-1}$ composed of weighted sum of the error, its integral, and its derivative. This calculation captures how far the empirical distribution is from the QoS specification, how long it has been that way, and how rapidly it is changing. Each of these values is computed over a time interval Δt, and each is scaled by a control gain constant (K_p, K_i, and K_d for the proportional, integral, and derivative stages, respectively). The values of Δt, K_p, K_i, and K_d depend on plant and operating conditions; methods for selecting these parameters were outlined in Chapter 3 on linear control, and the time interval Δt is chosen so as to allow the Markov Birth/Death Chain enough time to converge (covered in Section 4.4). Finally, the computed result is transformed via the nonlinear TRANSFORM() algorithm, whose form is discussed in Section 4.4.2. This transform improves control system response by mitigating an effect known as *quasi-stability*, in which convergence to stationarity can require unacceptably large Δt values unless $\mathcal{R}_{\beta-1}$ is temporarily increased in order to allow probability mass into the constrained

[2] The **rpd** and **tvd** are primarily useful for determining Δt, as described in Section 4.4.

region. Greater detail on quasi-stability is provided in Section 4.4. The complete feedback control method to estimate $\mathcal{R}_{\beta-1}$ is the FEEDBACK-BOTTLENECK-ESTIMATE() algorithm:

Algorithm 4 The feedback estimation algorithm for $\mathcal{R}_{\beta-1}$

```
FEEDBACK-BOTTLENECK-ESTIMATE()
1  $\epsilon \leftarrow$ COMPUTE-ERROR()
2  $\hat{\epsilon} \leftarrow K_p\epsilon + K_i \int_0^{\Delta t} \epsilon + K_d \frac{d\epsilon}{d\Delta t}$
3  $\hat{\epsilon} \leftarrow$ TRANSFORM($\hat{\epsilon}$)
4  **return** $\mathcal{R}_{\beta-1} + \hat{\epsilon}$
```

Lines 1 through 3 are each $O(n-\beta)$, so the overall running time of FEEDBACK-BOTTLENECK-ESTIMATE() is $O(1)$, assuming that $n-\beta$ is constant (which I do).

A missing element in both of the bottleneck estimator algorithms is the method by which I calculate $\hat{p}_{in}$ and $\hat{q}$. This is achieved through the use of a linear filtering mechanism, as described in Chapter 3. Two filters are required, each estimating the number of events (allocations or deallocations) that have occurred in the Δt time interval. If number of requests equals R, and the number of service events equal S (where R and S are the outputs from the request and service filters, respectively), then the probability of a request event is

$$\hat{p}_{in} = \frac{R}{R+S} \tag{4.20}$$

and the probability of a service event is

$$\hat{q} = \frac{S}{R+S} \tag{4.21}$$

The shorthand FILTER(p) will be used in the remainder of this thesis to denote the filter algorithm that combines the filter-estimates for R and S, Equation (4.20), to generate $\hat{p}_{in}$, and the shorthand FILTER(q) will denote a similar combination to estimate $\hat{q}$. The running times of these two algorithms depend on the linear filtering approach, but will be assumed to be $O(1)$ (which is possible using recursive linear filters such as the EWMA).

Having defined the algorithms that initialize the controller, filter the inputs, and estimate the bottleneck transition ratio, I can now specify the algorithm needed to perform the remaining two control steps, viz. computing the correct resource-request drop-probability, p_{drop}, and executing the actual resource-request drop. This algorithm, called CONTROL(), is summarized as follows:

Algorithm 5 The control algorithm

```
CONTROL()
 1  INITIALIZE()
 2  $\mathcal{R}_{\beta-1} \leftarrow$ SIMPLE-BOTTLENECK-ESTIMATE()
 3  while true
 4  do wait for a resource request arrival
 5      $\hat{p}_{in} \leftarrow$ FILTER($p$)
 6      $\hat{q} \leftarrow$ FILTER($q$)
 7      if $s = \beta - 1$
 8         then $\mathcal{R}_{\beta-1} \leftarrow$ FEEDBACK-BOTTLENECK-ESTIMATE()
 9      if $s \geq \beta - 1$
10         then $p_{drop} \leftarrow 1.00 - \frac{\mathcal{R}_s \hat{q}}{\hat{p}_{in}}$
11              if RANDOM() $< p_{drop}$
12                 then drop the arrival
13                 else admit the arrival
14         else admit the arrival
```

The RANDOM() function is an algorithm that generates a random number in the range $[0, 1]$; it is assumed to have a running time of $O(1)$. SIMPLE-BOTTLENECK-ESTIMATE() is known to have a running time of $O(\log n)$, but it and the INITIALIZE() function (which runs in $O(n)$ time) are initialization steps that run only once in the outer loop. The internal loop is dominated by FEEDBACK-BOTTLENECK-ESTIMATE(), which runs in $O(\beta)$ time; hence, the dominant running time of the control algorithm is primarily a function of the size of the Quality of Service specification.

The CONTROL() algorithm is an adaptive, nonlinear, model-reference feedback controller in the sense of [107]. Lines 12 and 13 are the actuator action, taken by the admission stage at the top left of Figures 1.3 and 4.5. The fundamental control calculation is made in lines 9–10, but the combined actions in lines 9–14 cannot, by themselves, accomodate anything other than a fixed ratio of p to q. Adaptivity for handling varying p's and q's is provided by the filtering and closed-loop feedback loop of lines 5–8. Note that this adaptivity does not apply to the PID gain constants, K_p, K_i, and K_d; these constants are set heuristically (e.g., using the Ziegler-Nichols

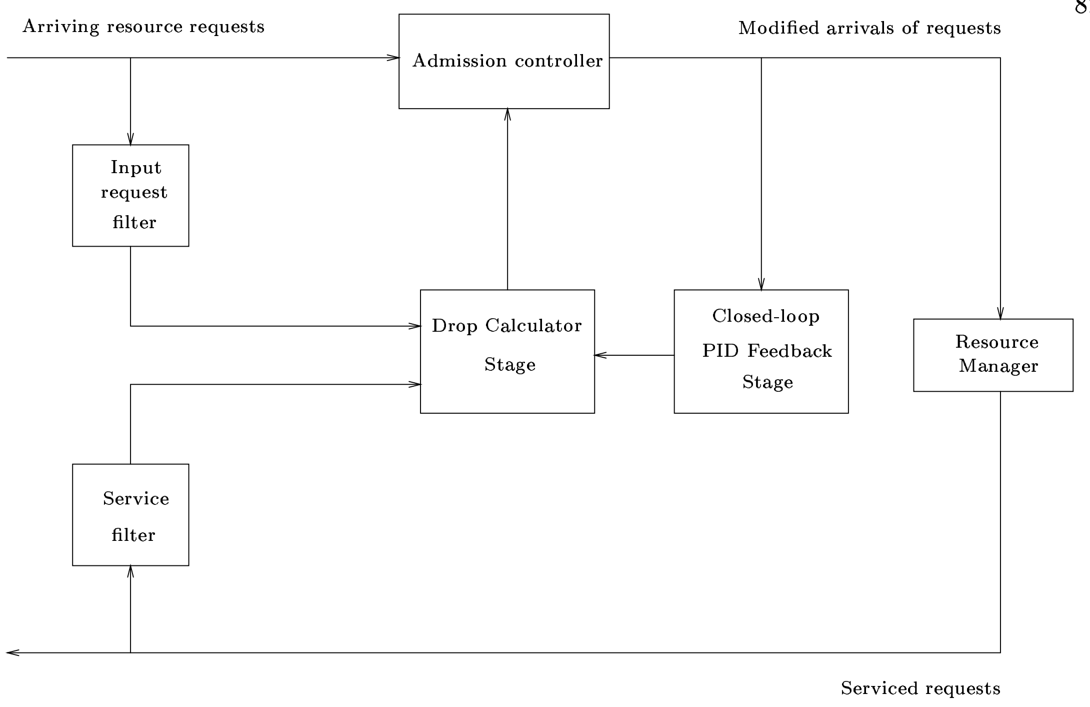


Figure 4.5: Control system block diagram. An drop-calculation stage computes the required admission probability based upon estimates from the input and service filters, and is steered by a closed-loop stage that accomodates any ratio of the two estimates.

or relay feedback methods), and hence are considered to be fixed and unchanging in this thesis for a given plant and problem. This assumption does not preclude the addition of mechanisms to adapt the values of the gain constants, but the adaptivity of the controller in this thesis is not a function of any such mechanisms, and so this is a nontraditional approach to adaptive control. The *nonlinearity* of CONTROL() arises from the TRANSFORM() algorithm in FEEDBACK-BOTTLENECK-ESTIMATE(), and the reference model is embodied in the transition ratio table that is computed from the Quality of Service specification, using the Markov Birth/Death Chain model equations.

A schematic diagram of the complete control system is shown in Figure 4.5. This diagram illustrates how the primary components of CONTROL() interact with the system in the control architecture. This Figure makes clear that the algorithm can be viewed as the combination of *drop-calculator* and *closed-loop* stages The drop-calculator stage performs the calculations in steps in lines 5 through 11 of the CONTROL() algorithm (with the exception of the feedback control portion in line 8), while the admission controller—the actuator—performs the checks of lines 11–13 to determine if it should admit arriving requests. The drop calculator uses only its estimates of the unmodified input request probability and the service probability. The closed-loop stage steers the drop calculator based on the error signal generated from the desired distribution ($Q(x)$) and the empirically-measured distribution of modified resource requests ($\mathcal{E}^{\Delta t}$). The control blocks of Figure 4.5 are expanded to give greater detail and insight into the composite system (controller and plant) in Figure 4.6.

This architecture and these algorithms comprise a complete, systematic approach to shape the stationary distribution of the Markov Birth/Death Chain using a principled control approach. To prove controllability, it remains to show that a Birth/Death Chain will converge to its stationary distribution in a reasonable amount of time.

4.4 Convergence

The control paradigm presented in this chapter is effective only if Markov Birth/Death Chains converge rapidly to their stationary distributions. This characteristic is known as *rapid mixing*; in this section, I formalize its definition. The method will still work if convergence is not rapid, but its effectiveness will be reduced. I also present novel results from numerical simulations to show that Markov Birth/Death Chains are rapidly mixing but that convergence can slow under a condition called *quasi-stability*. In these situations, it is possible to use a nonlinear transform to work around the convergence problems and to realize significant performance improvements.

4.4.1 Markov Birth/Death Chains are Rapidly Mixing

A Markov Birth/Death Chain will converge to its stationary distribution at a rate that is a function of the number of states in the Chain and of the conductance of the Chain[105]. For the purposes of this thesis, I am not concerned about the mixing rate of the entire Chain, but rather only in its mixing rate *in the states of the QoS specification*. Specifically, I am interested

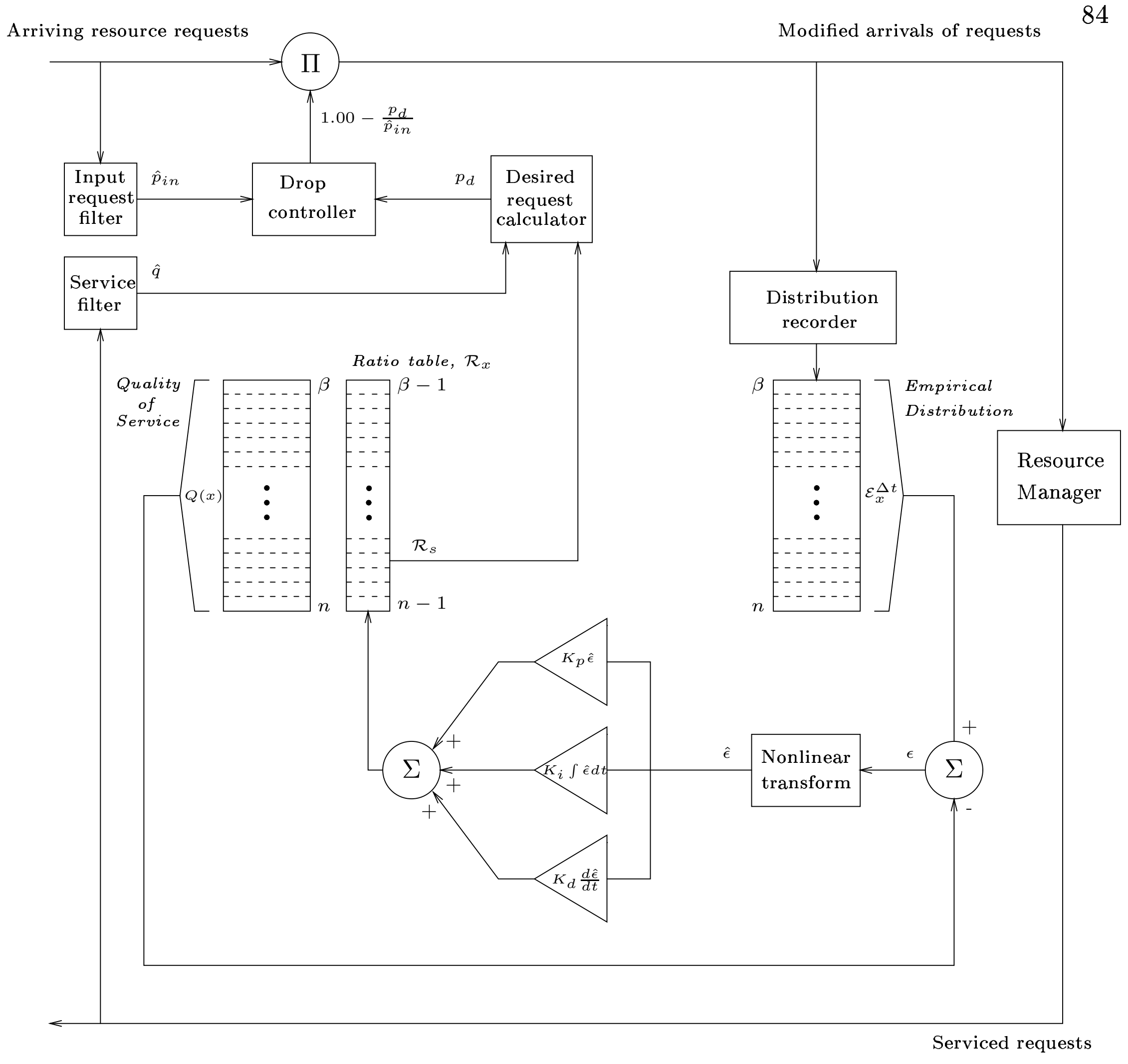


Figure 4.6: Control system detailed diagram.

knowing how the mixing rate in the QoS-constrained states changes when we

- keep the size of the set of QoS-constrained states fixed but vary the number of states in the Chain;
- vary the number of states in the QoS-constrained set while keeping the number of states in the Chain fixed; and
- keep both the size of the Chain and the size of the QoS-constrained set fixed, but vary the conductance between state $\beta - 1$ and state β.

Answers to these questions tell us how the mixing rate is related to the size of the QoS specification, the size of the Chain, and the conductance between the QoS specification and the Chain, respectively. This section summarizes the results of experiments designed to provide those answers. It is important to note that these results are neither formal nor general proofs of convergence: conducting these experiments requires imposing artificial constraints on the Markov Chains, to be described in more detail as each experiment is outlined. The results of the numerical experiments in this section indicate that Markov Birth/Death chains with a QoS specification and a ratio-based control system are rapidly mixing over that QoS specification in most cases.

I start by first defining convergence and rapid mixing. I say that a Markov Birth/Death Chain has *converged* if the relative pointwise distance (**rpd**) between its empirical distribution and its stationary limit in the QoS subset of states is 0.25 or smaller. This means that the empirical probability of every QoS-specified state in a Chain that has converged will differ from its stationary limit by no more than 25%. This metric is arbitrary, but reasonable: I found, through repeated simulations of Markov Birth/Death Chains of up to 100,000 states, that convergence to an **rpd** of 1.00 was rapid, after which convergence slowed markedly. Since, in this thesis, values below 1.00 indicate that every QoS-constrained state has been visited in the course of the simulation (a fact that I make clear in Section 4.4.2), values below 1.00 indicate progressively "more converged" Chains. However, convergence to a value of 0.00 could require an infinitely long simulation; hence convergence in a finite period of time is indicated by an **rpd** value between 0.00 and 1.00. For a given measure (e.g., size of Chain, size of QoS subset), I say that a Markov Birth/Death Chain is *rapidly mixing* with respect to that measure if the number of steps needed for the Chain to converge is no more than a linear function of the measure; similar definitions were made in [46, 48], which defined rapid mixing on general Chains to be *mixing in polynomial time*. Hence, in this thesis, a Chain of $O(n)$ states that requires $O(\log n)$ or $O(n)$ steps to converge in **rpd** could be said to be rapidly mixing, but a Chain that required $O(a^n)$ steps for some constant a would not.

I performed three numerical studies to show that Markov Birth/Death Chains are rapidly mixing in this sense. The three studies shared five common characteristics. First, each Markov Birth/Death Chain possessed N states, of which Q states (i.e., the interval $[N - Q + 1, N]$) are QoS-specified states. That is, a QoS specification of Q, with associated ratio-based distribution control, was used on each Chain. Second, a linear QoS specification was used in the QoS-specified states, and a nonadaptive ratio-based distribution controller was used to guarantee it. (The controller could be nonadaptive, and hence not use the PID stage, because the transition probabilities were unchanging.) Third, the upward transition probability for each state not in the QoS-specified states was $p = 0.9$, while the downward transition probability for all states was $q = 0.1$; these numbers were chosen both to drive the flow of probability mass towards the QoS-specified states, as well as to simulate an aggressive resource-drain condition. Fourth, unless otherwise indicated, each Chain simulation began in a starting state that was uniformly randomly chosen from the interval $[0, N]$, and each simulation continued until the Chain had converged. Finally, each simulation produced 100 estimates of the average number of steps needed to achieve convergence, where each average was itself taken from 100 distinct Chain simulations (a total of 10,000 simulations). By the Central Limit Theorem,[3] these data follow a gaussian distribution whose mean will be the true average number of steps needed for convergence. The low variance in the results (as indicated by the error bars in the graphs that follow) suggests that the number of simulation runs was adequate.

The first study identified the relationship between the size of a Chain with a fixed-size QoS subset and the mixing rate of that subset. I used a chain with $Q = 25$ and various sizes of N ranging from $N = 1,000$ to $N = 100,000$. Figure 4.7 illustrates the results. In this figure, the number of steps is normalized relative to the first data point; normalizing the steps allows us to concentrate solely upon how the size of the Chain affects the mixing rate. Figure 4.7 shows a linear dependence of the mixing rate upon the size of the Chain, up to Chain sizes of 100,000 states. Hence I conclude that Markov Birth/Death Chains are probably rapidly mixing in Chain

[3] An informal definition of the Central Limit Theorem was given in Chapter 3.

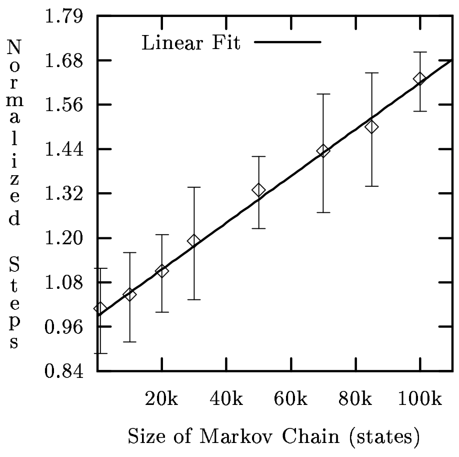
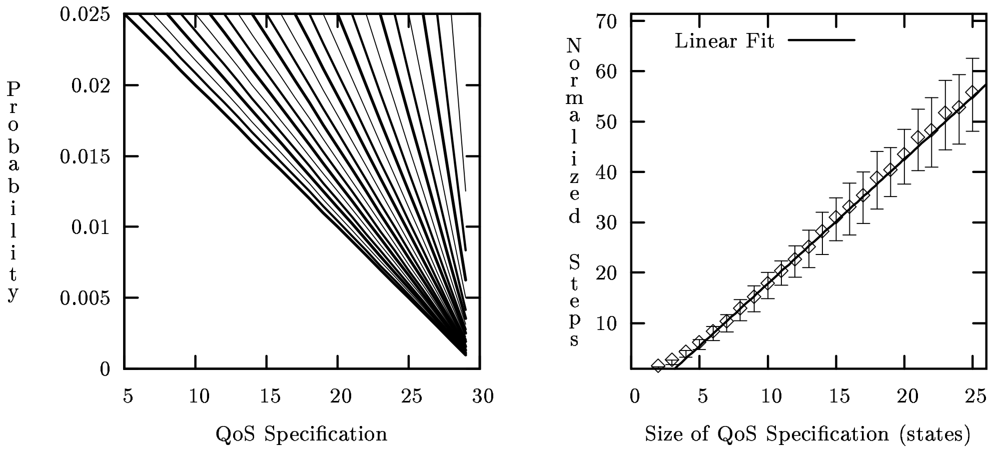


Figure 4.7: Relationship between the mixing rate in the QoS specified states and the overall size of the Chain. The number of steps required to converge is a linear function of the size of the Chain for Chains up to 100,000 states. The number of steps required is normalized relative to the first data point.



(a) QoS Specifications Used (b) Steps to Converge

Figure 4.8: Mixing rate in the QoS specified states as a function of the size of that QoS subset. The number of steps required to converge is a linear function of the size of the QoS subset. The number of steps required is normalized relative to the first data point.

size.

The next study identified the relationship between the size of the QoS-specified subset and the Chain mixing rate. I simulated a Chain of $N = 30$ states and $Q = \{2, \ldots, 25\}$ QoS-specified states. The various QoS subsets, shown in Figure 4.8(a), were chosen to be monotonically decreasing linear functions with the same probability in the first specified state, 0.025, so that

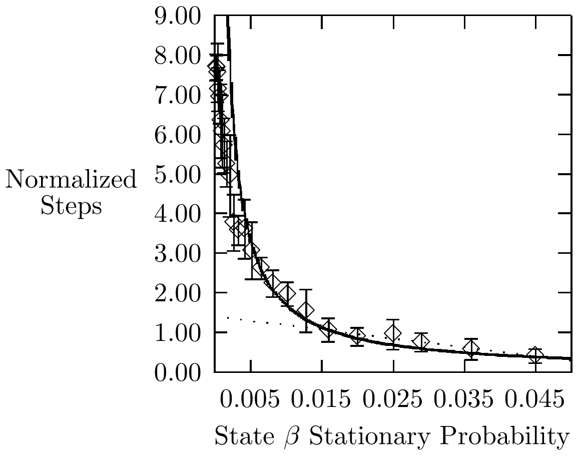


Figure 4.9: Mixing rate as a function of Chain conductance. The number of steps required to achieve convergence is inversely related to the stationary probability in state β (π_β); the solid line indicates an approximate fit to $y = \frac{A}{x} + B$, where y is the normalized steps, x is π_β, and A and B are constants. For values of $\pi_\beta \geq 0.015$, this relationship is approximately linear; the dotted line indicates an exact fit to $y = Ax + B$.

the conductance into the QoS-specified subset remained constant. The results, shown in Figure 4.8(b), indicate a linear relationship between the number of steps required for convergence and the size of the QoS subset; from this, I conclude that Markov Birth/Death Chains are most likely rapidly mixing in QoS subset size.

The final study addressed the relationship between conductance and mixing rate. In this study, I simulated a Chain of $N = 30$ states and $Q = 25$ QoS-specified states, but varied the QoS specification by changing the stationary probability of state β (the first state in the QoS specification). Lowering this value reduced the conductance on the transition from state $\beta - 1$ into state β, and hence should result in a decreased mixing rate. I expected this stationary probability to be inversely related to the the number of steps needed to converge (i.e., steps $\propto$ 1/conductance), since at a value of zero the Chain is broken and no stationary distribution exists, while at high values the Chain's convergence is not limited by the $\beta - 1 \rightarrow \beta$ transition probability. Figure 4.9 shows that there is indeed an inverse relationship between the stationary probability in state β and the steps required for the Chain to converge. For values of $\pi_\beta >$ 0.015, the relationship is approximately linear; this indicates that except in cases of extremely low conductivity between the QoS subset and the remainining states, conductance will not significantly affect the mixing rate of the Chain. This relationship can also help to guide the choice of a QoS specification: because the mixing rate drops dramatically when the stationary probability in state β is chosen to be very low, QoS specifications should be made with as high a value in state β as possible in order to maximize the mixing rate.

4.4.2 Quasi-Stability and Need for Nonlinear Control

In the previous section, I showed that Markov Birth/Death Chains appear to mix rapidly. Under certain circumstances, however, the mixing rate will be slowed due to an effect called *quasi-stability*. A quasi-stable Chain is one that converges rapidly to an intermediate, non-stationary distribution, and then only slowly evolves towards its true stationary limit. Although these Chains still mix rapidly in the sense that the although average number of steps required to converge is asymptotically linear, they will do so at a slower rate. This quasi-stable effect is probabilistic, and hence non-deterministic: it may or may not occur in the course of a Chain's convergence to stationarity. Because quasi-stability can slow mixing by several orders of magnitude (as will be demonstrated in this section), thereby significantly degrading the performance of controllers designed by the methods of this thesis, it is important to reduce the occurance of quasi-stable events.

Quasi-stability can be identified through a convergence plot, such as Figure 4.10. Two convergence traces are shown. The figure plots the number of simulation steps versus relative pointwise distance for the two simulations. The simulations were identical, in the sense that the same Chain, transition probabilities, and QoS specification were used for both simulations; the only difference was the random seed used to start the simulation. (The random seed determined the exact sequence of upward and downward state transitions, but not their overall distributions.) The dotted trace shows a quickly-converging simulation run that rapidly achieves an **rpd** of 0.25. The solid-line trace is a simulation run that became quasi-stable, displaying the characteristic of quasi-stability: a plateau at an **rpd** of 1.00. The cause of this the Chain state failing to transition

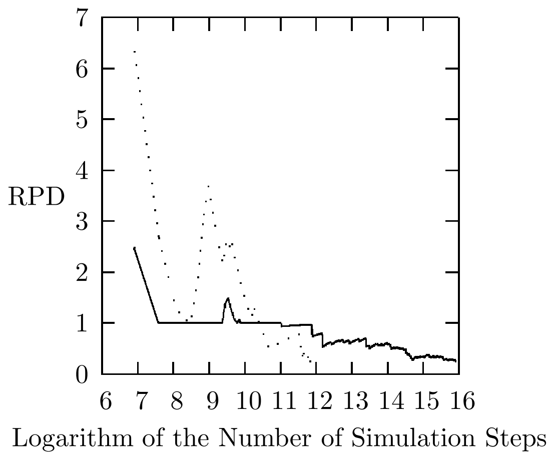


Figure 4.10: Quasi-stable convergence exhibited in simulation. The dotted line is a fast-converging simulation; its relative pointwise distance (**rpd**) drops quickly below 0.25. The solid line is a quasi-stable simulation that converges far more slowly.

into one or more states in the QoS specification region. To understand why the value of 1.00 is special, recall that the **rpd** is calculated as the difference between the empirical probability mass in the QoS-specified states and the QoS specification, normalized by the sum of the mass in that specification:

$$\epsilon = \max_i \frac{|\mathcal{E}_i^{\Delta t} - Q(i)|}{Q(i)}$$

If the Chain state does not transition into one or more QoS-specified states, then the empirical mass ($\mathcal{E}_i^{\Delta t}$) will be zero, and hence ϵ will equal 1.00.

The source of quasi-stability is a phenomenon related to the law of large numbers. Recall from Chapter 3 that this law states, informally, that a sample distribution of a random variable will approach the variable's true distribution as the sample size grows. Consider the case of flipping a fair coin. The ratio of heads to tails in an infinite number of flips is 1.00, a fact that will be reflected in the distribution of heads and tails of large samples. However, small series of coin flips can exhibit short-term deviations from the underlying distribution without affecting the long-term limit of that distribution. A set of ten flips in which all ten are heads is just as probable, for example, as a set of flips in which five are heads and five are tails.

It is this small-scale dynamical structure that leads to quasi-stability: sometimes a Markov Chain may not transition into a given state over an extended period of time, even though that transition may possess a relatively large probability.[4] However, just as in the coin-flips example, an infinite number of Chain steps will result in convergence to the stationary limit, regardless of whatever temporary variations may occur. These temporary variations, which are what I call *quasi-stability*, can degrade the performance of a distribution controller by forcing it to use a long Δt time constant when computing the empirical distribution. Figure 4.10 clearly shows that quasi-stability can have a significant and deleterious effect upon mixing rate *in the short term*. In that figure, the quasi-stable Chain required four orders of magnitude more steps to converge. Hence an important control goal is to identify a mechanism by which quasi-stability can be avoided and the mixing rate improved.

One solution would be to use a nonzero integral gain constant, K_i. The purpose of Integral stages, in general, is to compensate for steady-state errors, and since a Chain that is stuck in a quasi-stable regime will exhibit a large controller error over an extended period of time, incorporating such a stage would appear to be an obvious solution. This has some serious drawbacks, however. An integral stages increases the order of the controller—which could lead to instability, as discussed in Chapter 3—and this added control stage has associated computational overhead. The biggest problem, however, is that Integral stages are intended to compensate for the inherent steady-state error that results from using finite proportional gains, and not to correct for nondeterministic effects. When used improperly, an Integral control stage may induce the well-known effect of *integrator windup* and saturate the control system[7]. Integrator windup results when the controller reaches its actuator limits without achieving zero error between the control reference and the plant output. When this happens, the feedback is essentially broken and the controller runs as an open loop. If integrating action is used in the controller, it will continue to integrate and the integral term will become very large or "wind up;" only very large negative errors will allow the control system to recover, resulting in longterm

[4] For a fascinating and detailed treatment of this phenomenon, see "Fluctuations in Coin Tossing and Random Walks" in [25].

instability. In terms of this thesis, a quasi-stable plant could lead to a saturation of $\mathcal{R}_{\beta-1}$, which would allow too much probability mass to accumulate in the QoS-specified region.

The alternative is to use a nonlinear transform to "nudge" the Chain into convergence. The idea is to temporarily increase the conductance between state $\beta - 1$ and state β, allowing the Chain to make this difficult state transition without causing the Integral stage to produce a large error. A small (and rare) enough nudge will not affect the stationary probability to which the Chain converges, but it can drastically improve mixing rates.

Many different kinds of transforms can be useful for this. One attractive alternative is to probabilistically scale the error signal in the controller of Figure 4.6 such that it is temporarily amplified if the Chain is in state $\beta - 1$ and the **rpd** value is close to 1.00. This would have the effect of increasing the probability that the Chain will transition into the QoS-specified subset.

I performed a final study to assess this nonlinear transform approach. The results were dramatic. I considered a plateau of 10,000 or more steps at an **rpd** of 1.00 to be an indicator of quasi-stability. In the initial simulation—without the transform—I found that the Chain were quasi-stable in 67 out of 100 runs. With the transform, there were no quasi-stable runs in 100 further simulations, and the average number of steps to achieve convergence was reduced by 12.1%. This shows that the nonlinear transform stage in Figure 4.6 can significantly improve the performance of the controller.

Chapter 5

Examples

In this section I outline the design of adaptive controllers for two software systems that can be modeled by Markov Birth/Death Chains: the networking and the virtual memory subsystems of a Unix[1] operating system, OpenBSD[90]. BSD Unix is interesting because it is stable and widely deployed in server environments. The Department of Computer Science at the University of Colorado, for example, uses four FreeBSD machines as its primary login servers and a set of OpenBSD machines to handle electronic mail and firewall duties. The two controller examples outlined in this section for BSD subsystems demonstrate how the methods of this thesis can make substantive improvements to the stability and performance of already mature systems.

I first describe a controller for the network subsystem. Networking has come under increasing scrutiny because of the problems involved in providing end-to-end performance guarantees over a connection that may span many disparate network nodes, as well as due to the threat of *Denial of Service* (DoS) attacks. As described in Chapter 2, current research on end-to-end performance focusses on improving end-to-end data transfers in cooperative, high-bandwidth environments, and present solutions to DoS attacks are oriented towards rate limiting or methods for identifying and blocking the attacker at an upstream site (i.e., closer to the source rather than at the local system). Neither is wholly adequate from the standpoint of the victim machine. Attacks occur because the network enviroment is malicious, not cooperative, and can succeed even with very low amounts of traffic that consume negligible bandwidth. Systems in these en-

[1] Unix is a registered trademark of The Open Group; see [74] for details on the origins of the BSD variant.

vironments require methods to protect themselves—locally and quickly—without reliance upon cooperation from upstream hosts. In this chapter, I show in detail how to implement a software controller that degrades network performance in a controlled fashion, and is an effective solution for these types of attacks.

Because malicious attacks are not the only source of resource management problems, I also explored how the BSD virtual memory subsystem could be controlled in a manner similar to the DoS example. Virtual memory subsystems currently rely upon physical hardware that is particularly vulnerable to failure: spinning disk drives. Since a drive failure is equivalent to losing system memory, it results in the system losing functionality and becoming crippled. As in the network buffer example, a virtual memory subsystem controller can be designed to degrade performance in a controlled manner. It does this by selectively disallowing low-priority threads from running. This is in contrast to current practice, in which no control is applied when a failure occurs and all threads must compete equally for the remaining memory, leading to global memory starvation and possibly precluding management software from gracefully recovering from the overload.

Since the methods in this thesis apply to Markov Birth/Death Chain models, the first task is to show that the plant can be described using this structure. In the following sections, I present a detailed analysis of the networking and virtual memory subsystems of OpenBSD in order to verify that the conditions of Section 4.2 are satisfied.[2] I do this by identifying the resource of interest and then determining how it is managed in the operating system. For the Network subsystem, for example, I trace how the resource (the *mbuf cluster* network buffer) is allocated and deallocated by the buffer pool manager as packets arrive from the network and are processed by the operating system. In this subsystem, the number of allocated mbuf clusters is the resource state. Hence, for the Birth/Death paradigm to be applicable to the state of the mbuf cluster resource, I must show that (cf., page 70):

- the number of mbuf clusters is finite;

[2] Recall that these conditions are necessary and sufficient for the use of the Birth/Death model.

- mbuf cluster allocations and deallocations occur in increments of one; and
- each allocation or deallocation affects only *one* mbuf cluster.

As I will show in this section, the mbuf cluster management system satisfies these requirements. I also perform a similar analysis of the virtual memory subsystem to demonstrate that the resources of that subsystem, *anonymous memory pages*, are also managed in a fashion that satisfies the necessary and sufficient conditions for Birth/Death Chain modeling.[3] Both of these examples are in the realm of Operating Systems, so I conclude by discussing other applications of the control approach developed in this thesis.

5.1 Example #1: The Networking Subsystem

In this section I design, implement, and demonstrate the effectiveness of a software controller for the OpenBSD networking subsystem. This controller is intended to protect that system against local effects of network Denial of Service attacks. Local effects are those that manifest at the victim, such as victim crashes or loss of network communications on the victim. It is important to note that I am *not* performing global control. I specifically do not address, for example, the problem of upstream network bandwidth consumption in the path between the attacker(s) and the victim (which cannot be done, in general, in a malicious network).

The controller implemented here prevents network buffer-pool drainage resulting from the excessive network traffic of a DoS attack. It does this by selectively dropping certain network packets (i.e., refusing to admit them for processing), using a Markov Birth/Death Chain model of the buffer system that is used to store those packets. Network buffers in BSD-Unix networking subsystems are kernel buffers that are used to store data temporarily while it is transferred between the network and a user-level thread. For example, a worldwide web (WWW) request from a network client is initially received into a kernel network buffer and then copied the web server's own buffers. (The web server is the "user-level thread.) These kernel buffers—called

[3] An example of a resource in an operating system that does not satisfy the modeling conditions is the disk filesystem block pool, since the available space in a filesystem may be allocated in several blocks at once—violating the requirement of unit increments.

mbuf clusters in BSD—are statically allocated in a *buffer pool* at system startup, where the pool's maximum size is set to a fixed, compile-time constant. The buffers in the pool are used by all threads on the machine—including the kernel—for both transmission and reception of network data. A DoS attack can drain this pool of free buffers, thereby preventing the victim from receiving or transmitting network data; this leads to instability, deterioration of system performance, and even to complete system failure in the worst case. The goal of the controller in this example is to ensure that some free buffers always remain in the buffer pool, and to do so with a principled control algorithm that maintains a performance level specified by the system manager.

I start with an overview of communication networks, protocols, and the structure of mbuf clusters in OpenBSD. I will then illustrate the manner in which mbuf clusters are consumed and released back to the free pool, as well as how this process can be subverted by a Denial of Service attack to drain the mbuf cluster pool. I will then use this background to determine the states of the Markov Chain model, to choose a QoS specification, and finally to design a control system that guarantees that specification.

5.1.1 Overview of Network Communication

This section is an overview of communications as commonly used in networks such as the Internet. This description is far from complete; see [16] and [74] for more detailed expositions. The details in this section will clarify the communications framework of a typical Internet-connected system: what the network is, how the flow of data on it is structured, and the general manner in which the system processes the data.

A communications network can be represented by a connected graph in which nodes are computing systems and edges are data channels between nodes. The nodes may be any electronic systems capable of network commmunication, such as a general-purpose computer, a specialized router, or even an individual processor on a multiprocessor circuit board. The edges may be conductive circuit traces, serial lines, shared-medium broadcast cables, or empty space (i.e., using radio transmissions). While communications networks may take on a wide variety of physical forms, I used nodes that are a pair of general-purpose computers running OpenBSD, and edges that are 10Mb/s (megabit per second) Ethernets.

In the Ethernet software standard, data is transmitted and received serially between nodes on the network in groups of bits called *frames*. Each node on an Ethernet network may have one or more Network Interface Connections (NICs), and each NIC possesses a unique, hardcoded, 64-bit station address called its *hardware address*. Frames are subdivided into a fixed-size header containing the sender and receiver hardware addresses, some information about the type of data that follows, a *payload* portion, and a checksum that is used for error detection. The maximum size of the frames is limited to 1500 bytes, but no limit is placed on the number of frames that can be sent. Ethernet is a simple standard that assumes that all nodes are directly connected to each other on a single, shared medium—so frames cannot be routed **through** a node, only placed onto the shared medium—and it provides no provision for data integrity or guaranteed delivery.

In order to provide support for multi-tiered networks, integrity, and delivery guarantees, higher-level protocol packets are *encapsulated* as the payload of the Ethernet frame. The TCP/IP network protocol suite, the most commonly-used set of protocols on the Internet, is an example of this. It consists of three primary protocols: the Internet Protocol (IP); the User Datagram Protocol (UDP); and the Transmission Control Protocol (TCP). IP packets are similar to Ethernet frames in the sense that they contain addresses and data payloads, but the IP addressing scheme is software-configurable and hardware-independent, which makes it suitable for supporting multiple network tiers. The IP payload is usually an encapsulated UDP or TCP packet. Both UDP and TCP use the abstraction of source and destination *ports*; this provision allows multiple, independent threads of execution to communicate cooperatively on the network. UDP is the simpler of the two; it adds only a checksum that ensures integrity of the data at the receiver. TCP provides a data checksum, as well as data retransmission support (for corrupted or lost packets) and a persistent, connection-oriented data transfer paradigm. UDP and TCP are both data transport protocols, and both are encapsulated in IP packets.

The protocol encapsulation paradigm, in which progressively more complex and distinct data structures are layered around the data, leads naturally to processing routines that are organized as a layered *stack*, with each layer handling a specific protocol. When a machine receives an Ethernet frame with an IP packet payload, for example, that payload is initially stored in an mbuf cluster which is passed "up" to the IP input routine. The IP input routine performs various checks, such as verifying that the packet destination address matches a local IP address, and then passes the IP data payload (in the mbuf cluster) either to the UDP input routine (if the encapsulated payload was a UDP packet) or the TCP input routine (if it was a TCP packet). The UDP and TCP routines then attach the mbuf cluster containing the user-level data to the data input queue of the appropriate thread. Delivery is completed when that thread reads its queue and the data is copied out of the mbuf cluster (a precious, limited resource) and into the user-level memory of the thread (a comparatively unlimited resource). Conversely, when a thread transmits data to the network, it also notifies the kernel, which then copies the data to a kernel buffer and calls the appropriate protocol output routine, where the buffer travels back down the layers until it is eventually transmitted onto the Ethernet.

It is important to note that the data is never copied until the final step out of the mbuf cluster and into user memory space; packet headers and tailers are 'removed' by changing the pointers into the mbuf cluster that indicate the beginning and end of the payloads. This delicate manipulation of data buffers necessitates a sophisticated kernel memory management system, discussed in the next section, which builds a complete picture of the relationship between mbufs, mbuf clusters, and the network.

5.1.2 BSD Network Memory Management

The BSD network buffer management system revolves around the use of *mbuf* structures and their data pages, *mbuf clusters*. An mbuf is a data structure, and an mbuf cluster is a contiguous, fixed-size area of unstructured memory. Mbufs and mbuf clusters are analogous to

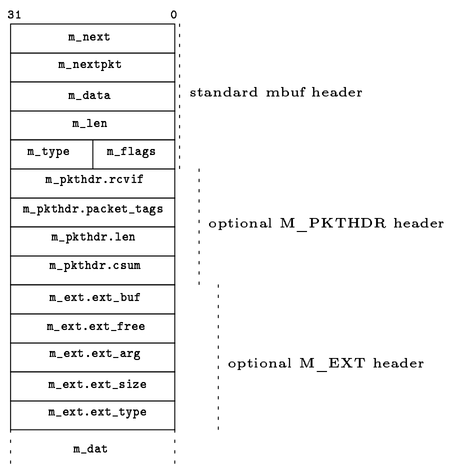


Figure 5.1: The mbuf header structure. The header is composed of three parts: a set of standard fields, a set of packet header fields, and external data fields. The standard fields are in every mbuf, whereas the packet header set is included only in mbufs that carry higher-level protocol packet headers, and the external data set is included only in mbufs that carry the data payloads of those packets.

the header and data payload of a network packet: the mbuf gives structure and meaning to the raw data in the mbuf cluster. Just as a network packet always has a packet header, an mbuf is always associated with an mbuf cluster, since without the data structure provided by the mbuf, the data in the mbuf cluster has no meaning. This structure includes information on the size, offset, and type of data in an mbuf cluster, as well as pointers to facilitate use of lists and chains that allow related data to be grouped. Mbufs and mbuf clusters are used to store Ethernet frames, TCP/IP packets, and unix *socket* structures that are involved in transferring data to and from the network. As packets arrive from or are transmitted onto the network, their data is split into chains of mbufs and mbuf clusters and then passed through the layers of protocol processing. In this section, I describe the organization of mbufs, how they are used to manage mbuf clusters, and how mbufs and mbuf clusters are manipulated as they are used to handle network data. This information is a prerequisite to a description of how an OpenBSD machine interfaces to an Ethernet network, which in turn is required to design the Markov Birth/Death Chain model of the mbuf clusters in the network subsystem.

An mbuf is a 256-byte structure consisting of a header and a small data area. The mbuf header is composed of three sets of fields: the **standard** set, the **packet header** set, and the **external data** set. The standard header is always present in every mbuf; it provides information on the size and type of data in the mbuf, the offset of the valid data in the data area, and pointers used to link mbufs together in *chains* and chains together in *lists*. The packet header set is included if the data area contains an IP, UDP or TCP protocol header, and the external data set is included if an mbuf cluster is used as the data area; this is usually the case, since mbuf clusters are many times larger than the mbuf data area. Hence the mbuf header changes according to the type of network data the mbuf carries; each set of header fields serves a specific purpose that I describe now in greater detail.

The standard header set contains basic information used in all mbufs. The memory address of the start of the data area—whether in the mbuf itself or in an mbuf cluster—is indicated by the `m_data` field, and the amount of valid data is given by the `m_len` field. Two or more mbufs may be linked into *mbuf chains* to hold an arbitrary amount of data using the `m_next` field, and related mbuf chains may be grouped into *mbuf queues* using the `m_nextpkt` field. The final fields in the standard header, `m_type` and `m_flags`, provide information to higher-level protocol handlers on the type and organization of data carried by the mbuf. The two flags of interest for TCP/IP are the M_PKTHDR and M_EXT flags, which necessitate the addition of the packet header and external data mbuf headers, respectively.

An mbuf with the M_PKTHDR flag set is an mbuf that contains the header of a network packet, and in the mbuf header the packet header set is included in addition to the standard mbuf header. The `m_pkthdr.rcvif` field points to a structure that describes the network interface on which the packet was received. The `m_pkthdr.packet_tags` field is a pointer to a linked list of arbitrary *tags* which may be used for selectively marking and filtering certain packets, among other things. Finally, the `m_pkthdr.len` and `m_pkthdr.csum` fields record the total packet length (not just the header) and the protocol checksum (if present), respectively. Network packets are normally divided into chains beginning with an M_PKTHDR mbuf that holds the protocol header and one or more successive network-data mbufs that carry the packet payload.

Mbufs that carry packet payloads always use an mbuf cluster as the external data area, with the M_EXT flag set in `m_flags` and the additional external data header set in the mbuf header. In this header set, `m_ext.ext_buf` is a pointer to the start of the external data area (which is not necessarily the same as the start of valid data within that area). In OpenBSD version 3.3, the mbuf cluster is the only valid external data area that may be associated with an mbuf, but the provision exists for using different data types: the `m_ext.ext_free` field in the mbuf external data header can be used to provide a pointer to a routine to free the memory in the external data area, should a dynamic buffer be desired instead of a statically-allocated mbuf cluster. The `m_ext.ext_arg` field is an argument to the `m_ext.ext_free` routine, the `m_ext.ext_size` field indicates the size of that argument, and the `m_ext.ext_type` field provides a way to determine what type of external buffer is in use.

The M_EXT and M_PKTHDR flags are not mutually exclusive, and in fact the packet headers are also kept in mbuf clusters, necessitating the use of both flags in the first mbuf of an mbuf chain. Every network packet is composed of a chain of mbufs in which the first points to the beginning of the packet header and the remainder point to packet data. However, in OpenBSD/x86,[4] only one mbuf cluster is used since the maximum size of an Ethernet frame is smaller than the size of an mbuf cluster; the various mbufs here merely point into different sections of the single mbuf cluster. Hence, on reception of a network packet, the OpenBSD/x86 kernel places the Ethernet frame in a single mbuf cluster, creates a chain of mbufs that indicate the relevant sections of data within that mbuf cluster, and passes it up through the layers of network procotol processing. The mbufs in the chain and their associated mbuf clusters remain allocated and unavailable for new packets as the chain passes up through the layers until the kernel finally copies the data into a destination user-space buffer. This means that all mbufs and mbuf clusters in an mbuf chain are not returned to their respective kernel buffer free pools until they are completely processed. This processing can require long periods of time (minutes, hours, days, or even years).

[4] OpenBSD running on an Intel 80x86 or Pentium processor.

Since mbuf clusters may persist in a busy state for extended periods of time, the mbuf cluster pool may drain to zero size if the system receives network traffic faster than it can recover mbuf clusters. In such a situation, network communications to and from the host *on every network interface* cease because the mbuf cluster pool is shared among all interfaces. Herein lies the crux of the Network DoS problem: because of this sharing, an attacker need only send attack packets to a single NIC on a victim (e.g., its Internet connection interface) in order to prevent it from communicating on any other (e.g., its Intranet connections). To mitigate these effects, OpenBSD uses the *pool(9)* buffer pool management system to allocate new virtual memory pages for mbufs and mbuf clusters when necessary. It is unwise to allow the pool manager to allocate memory without bound, however, since memory is finite and running out of virtual memory is worse than running out of mbufs or mbuf clusters. When out of memory, a system *thrashes* and becomes totally unusable; when out of mbuf clusters, it is usually only precluded from network communications. The total number of mbuf clusters is therefore limited so that no more than 2048 may exist in the system at any time in the default OpenBSD configuration.[5] Since each mbuf cluster must be associated with an mbuf, and because mbufs most often use an mbuf cluster for their data, this places an approximate limit on the number of available mbufs as well. By necessity, this hardcoded default value has increased as networks have become faster (since mbuf clusters can be consumed more rapidly on a faster network), and it is often manually increased when heavy network loads are expected (such as in World Wide Web servers).

I now describe the interface between OpenBSD and the Ethernet network in detail to show how mbufs and mbuf clusters are allocated, processed, and released back to their free pools, and then describe how a Denial of Service attack can drain the mbuf cluster pool. This description is the last piece of information needed in order to design a Markov Birth/Death Chain model of the mbuf cluster management system.

[5] The configuration is static: changing the number of mbuf clusters requires recompiling the kernel and rebooting the system.

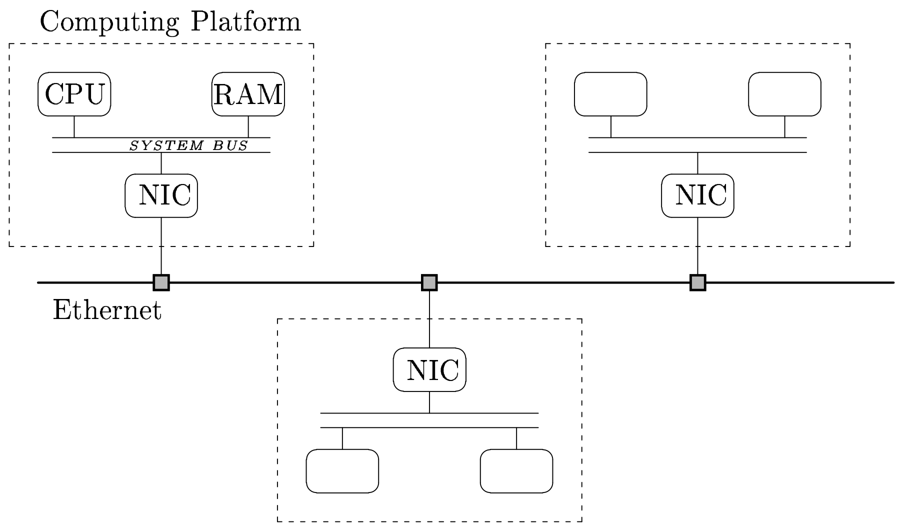


Figure 5.2: Physical relationship of NIC, CPU and RAM. On each computing platform, the NIC acts as the middleman between the network and the system data bus.

5.1.3 The Network Interface

In this section, I describe the hardware and software interface between the network and the operating system for a specific network interface controller, and then trace the path that data takes from the network into the kernel, and finally into user space, as well as the reverse path (user space back down to the network). These details are required in order to verify the validity of the Markov Birth/Death Chain model and determine its exact number of states.

The specific network device I study for this model is the 3Com 3C509 Network Interface Controller (NIC)[17] as deployed in an OpenBSD/x86 system. Technical documentation is readily available for this NIC, and it is widely deployed in general-purpose computers. This choice of networking hardware does not affect the generality of the mbuf cluster model I will develop. The design of the OpenBSD networking software guarantees that the model will be valid regardless of the exact network hardware configuration used because of the nonreentrant, single-threaded pool management routines by which OpenBSD/x86 mbuf clusters are allocated and deallocated.

The NIC acts as the interface between the operating system and the network (Figure 5.2).

It reads individual bits as they arrive from other nodes on the network, composes them into coherent frames, and passes them up for processing by the operating system. It also operates in reverse, accepting frames that are ready for transmission from the kernel and serializing them onto the network.

The 3C509C is a *bus mastering* NIC that can transfer Ethernet frames between the network and its host's internal system memory without the intervention of the host CPU. It does this by taking control of the host's internal bus (becoming the *bus master*) and writing a received frame directly into system memory; after doing so, it relinquishes control of the bus and generates an interrupt to the host CPU to inform it that data is ready for processing. The alternative for bus mastering is to require the CPU to copy the data from the NIC to its own memory; this is inefficient, since the CPU wastes cycles in the copy that could have been spent in user-level threads. Bus masters reduce the load on the CPU but at the expense of added complexity in the driver software.

Communication between the NIC and the OS occurs through a set of configuration registers in the NIC and dedicated blocks of RAM in the host called *uplists* and *downlists*. The configuration registers store operation statistics and pointers to the uplists and downlists in the host RAM, while the lists themselves are used to transfer frames between the NIC and the OS. After a system reset, the host reserves the uplist and downlist data areas in its RAM, and then initializes the NIC by writing the addresses of those areas into the NIC's *UplistPtr* and *DownlistPtr* registers. The host then enables frame reception and transmission on the NIC.

5.1.4 Ethernet Frame Reception and Transmission

The NIC transfers Ethernet frames between the network and the host memory by using the uplists and downlists, which must be initialized through proper configuration of their pointers. Each uplist (downlist) consists of a set of linked Upload Packet Descriptors or *UPDs* (Download Packet Descriptors, or *DPDs*), which can contain one or more pointers to *fragment buffers* (Figure 5.3). The upload and download procedures are quite similar from the standpoint

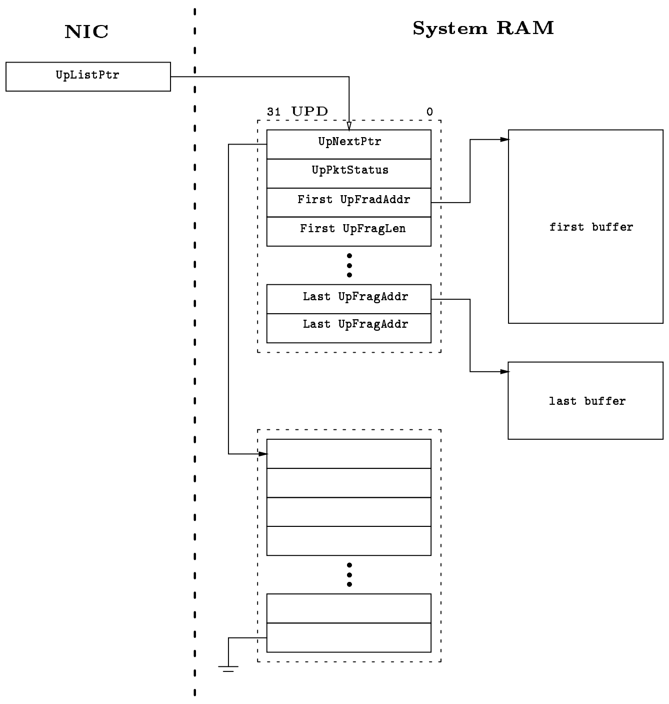
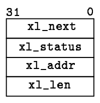


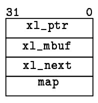
Figure 5.3: Data structure of the 3C509C Upload Packet Descriptor (UPD). The UpListPtr is a register in the NIC, and the Upload Packet Descriptors (UPDs) are data areas in the host's system RAM.

of the NIC, so I describe in detail only the sequence of events when a frame is received. When data arrives from the network, the NIC first assembles the serially-received bits into a coherent frame, and then uploads the frame by following the UpListPtr to find the first free UPD in the uplist. Each UPD contains a status field that is zero if the UPD is free and available, and nonzero after the NIC uploads a frame to it. The NIC transfers incoming frames to as many data buffers as are necessary to hold the whole frame; in OpenBSD/x86, this is never more than one since the buffers (mbuf clusters) are larger than the frames. After upload, the NIC automatically updates its UpListPtr to the UpListNext pointer in the (now occupied) UPD.

The kernel initializes an uplist for each NIC in the system at reset. To do this, it allocates 32 UPDs per uplist, each UPD containing one fragment pointer that is set to point to a single mbuf cluster. The kernel structure used to do this is the `xl_list` struct, which corresponds to a UPD with one fragment, and which the system internally organizes using `xl_chain` structs.



(a) `xl_list`



(b) `xl_chain`

Figure 5.4: Structures used by OpenBSD to manage 3C509C Uplist descriptors. `xl_list` is formatted to be the same as an Upload Packet descriptor (UPD) with one fragment; linked `xl_list` structures are provided to the NIC as its UpList. `xl_chain` is used by the kernel to manage the linked `xl_list` structures.

The `xl_list` contains the exact structure expected by the 3C509C for its UPD: a pointer to the next UPD in a linked list (`xl_next`), a status field (`xl_status`), a pointer to the upload fragment (`xl_addr`), and a field indicating the length of the data fragment buffer (`xl_len`) (Figure 5.4(a)). The kernel creates its own linked framework of `xl_chain` structures in order to manage the UPDs, each of which contains a pointer to its associated `xl_list` (the `xl_ptr` pointer), a pointer to the mbuf associated with the mbuf cluster (`xl_mbuf`), a pointer to the next `xl_chain` (`xl_next`), and a pointer to the direct memory access map state structure that OpenBSD uses to control the NIC's bus mastering and direct memory access capabilities for the mbuf cluster associated with the list and chain (Figure 5.4(b)). The 32 list and chain structures are linked together in a ring, with a head pointer (`xl_rx_head`) maintained by the kernel to keep track of where the next available UPD is at all times (Figure 5.1.4). Recall that the NIC must assemble each frame in its internal memory before uploading it; it does so using its own 2048-byte RAM. The 32 UPDs per uplist that OpenBSD allocates in the ring structure represent the maximum number of frames that the NIC can buffer in its RAM if it receives the smallest possible ethernet frame (64 bytes). By providing as many buffers as necessary for the NIC to completely drain its internal buffer, the kernel minimizes the chance of a receiver overrun (in which a frame arrives but no free UPD exists) in normal operating conditions.

Once an upload is complete, the NIC sets its UpListPtr pointer to the UpNext pointer value in the just-loaded UPD, and then generates an interrupt to the host CPU, causing the kernel to block the upload of more frames and to enter the network interrupt handler for the 3C509C. This interrupt handler, `xl_intr()`, blocks new network interrupts, performs houskeep-

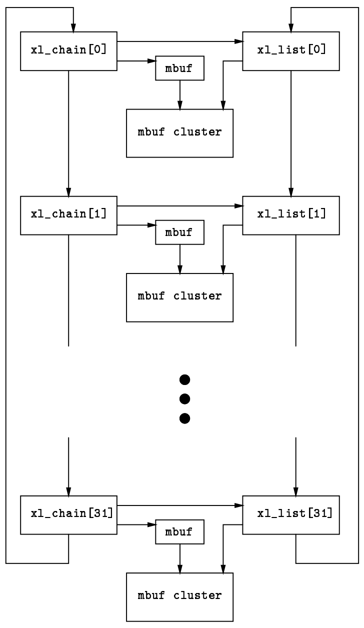


Figure 5.5: OpenBSD ring organization of list and chain structures for Upload Packet Descriptors. Each `xl_list` corresponds to a UPD and is accessible by both the NIC and the kernel. The `xl_chain` structures are used by the kernel to manage the mbufs associated with the UPDs and mbuf clusters.

ing tasks such as updating the network statistics for the interface, and then begins processing the uploaded frame by clearing the interrupt status in the NIC and walking through the linked list of UPDs. For each UPD, the interrupt handler attempts to allocate a replacement mbuf cluster by calling the *pool_get()* function; this function either returns a pointer to a single new mbuf cluster or fails if no mbuf clusters are available. If there are *no free mbuf clusters*, the interrupt handler simply clears the status field in the UPD (`xl_status`). This informs the NIC that the mbuf cluster is available once again for upload; since the cluster data is never processed, the ethernet frame contained in the mbuf cluster is lost, and the data contained therein never reaches its destination. For TCP packets, the data will eventually be retransmitted by the sender; for all other types of network packets, the data is irrevocably lost. In the case that free mbuf clusters do exist, the interrupt handler sets the `xl_addr` pointer in the UPD to point to the new mbuf cluster, and then passes the old mbuf cluster (with its uploaded ethernet frame) to the `ether_input()` routine.

The `ether_input()` handler performs basic sanity checks on the incoming Ethernet frames and then directs the frame to the next layer of protocol processing. If an IP packet is encapsulated in the frame—the most relevant case for the Denial of Service attacks with which I am concerned—the handler enqueues the mbuf cluster onto the IP input queue, schedules an IP software interrupt, and exits.

The `ip_input()` routine processes the IP input queue, and `tcp_input()` or `udp_input()` continue the processing if a TCP or UDP packet is encapsulated in the IP packet. For each mbuf cluster in the queue, `ip_input()` performs sanity checks on the IP packet in the mbuf cluster (e.g., ensures that the packet is destined for the local host) and then routes it to the next protocol layer. The `tcp_input()` (or `udp_input()`) function processes the TCP (UDP) packet within the mbuf cluster and enqueues the mbuf cluster at the socket for the user thread awaiting the data. Note that until this time, there have been no actual data copies: only the pointer to the mbuf cluster is passed from function to function and layer to layer. Regardless of how much data is in the mbuf cluster, only a few pointer bytes are moved. After the mbuf cluster is enqueued at the socket, it awaits a read on the socket by the user process via the `soreceive()` function. When this user process is eventually given cpu time by a kernel context switch, the `soreceive()` can copy the data portion of the TCP packet (UDP packet) in the mbuf cluster from the cluster to a virtual memory page that has been mapped into the user thread's memory space; the kernel and then calls the `mfree()` function to release the mbuf cluster back to the kernel pool. At that point, the cluster is available to be reallocated and reattached to a UPD fragment pointer.

Frame transmission occurs in an analogous but opposite manner. When data is written to a network socket by a user thread it is initially broken into mbuf clusters by the `ether_output()` routine and then distributed into the NIC's Download Packet Descriptors (DPDs). The DPD and UDP data structures are nearly identical, but OpenBSD-3.3 provides 63 fragment pointers for each DPD. Unlike the UPD fragments, however, the DPD fragments are not automatically associated with mbuf clusters; instead, they are attached only when a transmission is necessary.

The method by which resource-starvation DoS attacks work should now be clear, since such an attack need only allocate mbuf clusters faster than they can be freed by the user-level thread's `soreceive()`. This can occur in a variety of ways; a common one is to force each mbuf cluster to contain a request that can only be satisfied through a slowly occurring physical event. For example, each mbuf cluster could contain a WWW request for a specific image that is on a hard disk drive; requests for this image can arrive from the network very quickly, but the data can only be read from the disk drive very slowly. This also reveals why network performance guarantees cannot solve the problem: the attacker need only send requests faster than the victim can service them, not faster than an arbitrary performance metric. If the victim requires one second to satisfy a request then an attacker has nearly 34 minutes to send only a few thousand packets and completely drain the mbuf cluster pool.

5.1.5 Markov Model for mbuf clusters

I now use this background to justify the use of a Markov Birth/Death Chain to model the mbuf cluster state, and then describe how to use such a model in the network subsystem. I start by showing that the system state is discrete, that mbuf clusters are accessed serially, and that they are allocated and freed in unit increments.

The *state of the mbuf cluster system* in this model represents the number of mbuf clusters in use *and associated with a particular network interface* in the system. Hence, the state of the model applies only to received packets on a single network interface. State zero means that no free mbuf clusters are in use and associated with that network interface; state N will represent the maximum state, in which the maximum number of mbuf clusters are in use. System-wide, OpenBSD predefines the maximum number of mbuf clusters to be NMBCLUSTERS, a kernel constant set to 2048 in OpenBSD-3.3, and each 3C509C interface uses 32 mbuf clusters at all times. I will develop a controller for one NIC in a system with three 3C509C NICs, hence state zero is the state in which 32 mbuf clusters are allocated, and state $N \leq$ NMBCLUSTERS.

Transitions in the state of the mbuf cluster system (and hence in the model as well) occur when mbuf clusters are allocated or deallocated. Upward transitions between the states occur upon allocation. This occurs in the Ethernet receiver interrupt routine `xl_intr()`, and in the `sendmsg()` and `sendto` packet transmission routines, and when setting up a network connection: through a call to `setsockopt()` (which sets options on network sockets); or a call to `bind()` (which associates a network address with a network socket); or a call to `connect()` (which establishes a stateful connection between two sockets). In every case, allocations occur only through the `pool_get()` call, which executes at a high interrupt level to ensure that its operation is not interrupted, making it non-reentrant. Downward transitions occur upon deallocation. This happens when mbuf clusters are released back to the free pool: when the `setsockopt()`, `bind()`, and `connect()` calls return, and when a user-level thread calls `so_receive()` to transfer received data from the mbuf cluster into the thread's own memory space.

Downward transitions occur when mbuf clusters are released back to the free pool. Since mbuf clusters are used to hold socket options as well as network data, they are freed whenever a `setsockopt()` call returns or when a user-level thread reads its input queue via the `so_receive()` system call. Both calls return the mbuf cluster to the free pool via the `pool_put()` call. The `setsockopt()` call happens quickly (and rarely, since socket options are not often changed once the socket has been created), but the rate of calls to `so_receive()` for a particular input queue on a socket will depend entirely on the behavior of the user-level process. It might occur quickly—as fast as the system can process the input queue—or each read of the queue might be predicated upon very slow events such as data read from a disk drive or even human input. In either case, `pool_put()` is protected by a high interrupt level to ensure that it is non-reentrant and also only returns one mbuf cluster to the free pool per invocation; this ensures that mbuf clusters are released serially and in unit increments.

I can now summarize the evidence to support the use of a Birth/Death Markov Chain to model the mbuf cluster pool:

(1) **FINITENESS**: mbuf clusters are *discrete resources*. The state of the Markov Chain

is discrete and represents the number of mbuf clusters in use. It ranges between 32 and NMBCLUSTERS;

(2) **SERIALIZATION**: allocation and deallocation of mbuf clusters occurs only through the `pool()` interface, is non-reentrant and can only be called by one thread (including the kernel) at any time; and

(3) **UNIT CHANGES**: both `pool_get()` and `pool_put()` allow only one mbuf cluster to be manipulated at once.

The only variable of the model between systems is N, and it can be affected by changing NMBCLUSTERS, but this event cannot occur without a hardware reset. Thus, the model for this system is static between system reboots, and it a controller based upon it can determine all the necessary information it needs to use the model at that boot time.

I have now shown that the structure of this system admits the *Birth/Death Model* of Figure 1.2 with the appropriate choice of N. The next step is to choose the controller design parameters.

5.1.6 Mbuf Cluster Controller Design

I now describe the design of a Markov Birth/Death Chain controller to manage the mbuf cluster pool for a single network interface. This controller limits the number of mbuf clusters available to that interface, and degrades access to the cluster pool as the interface begins to reach its buffer limits. The design proceeds in the following steps:

(1) Determine n

(2) Choose the QoS specification

(3) Choose Δt

(4) Select a PID controller type

(5) Decide upon the PID gain constants

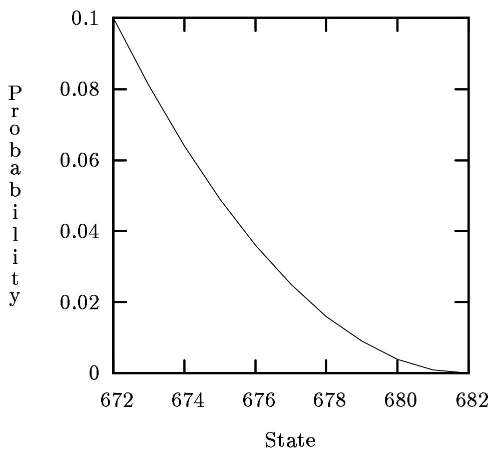


Figure 5.6: Parabolic QoS Specification for the mbuf cluster controller. For this specification, $n = 682$ and $\beta = 672$.

(6) Determine the filters to use

I now describe each of the design steps.

The first step is to determine n, the size of the Markov Birth/Death Chain that is to serve as the reference model for the system. The value of n represents the number of mbuf clusters from the system-wide pool that will be allocated exclusively to the controlled network interface. This is an essentially arbitrary value; since there are three network interfaces in the example machine, I choose NMBCLUSTERS/3 as a reasonable value. Since NMBCLUSTERS is defined to be 2048 under OpenBSD, this means $n = 682$.

The second step is to choose the QoS specification. The results of the numerical simulations from Chapter 4 indicate that the QoS specification should be as small as possible , in order to optimize the running time of the control algorithm, and that the conductance from state $\beta - 1$ to state β should be as high as possible, in order to minimize the mixing rate. In particular, I found that π_β should be 0.015 or larger. For these reasons, I selected a 10-state specification, implying that $\beta = 672$. The exact specification is the parabolic curve of Figure 5.6, which I chose because for the same amount of probability mass relative to a linear specification, the conductance between state $\beta - 1$ and state β is higher.

The third step is to choose Δt, the control time constant. It must be large enough that the mbuf cluster pool has a chance to converge to near-stationarity, but small enough to make the control system useful. Simulations on a Birth/Death Markov Chain using the parabolic QoS showed that convergence under an aggressive attack ($p = 0.9$, $q = 0.1$) was achieved in approximately 200,000 steps on average, with a standard deviation of approximately 300,000 steps. Since Ethernet packets can theoretically arrive every 6.4μs on a 10Mb/s network, this would imply that a convergence could be achieved in 1.3 ± 2.0 seconds under this aggressive attack. Hence a value of $\Delta t = 5$s should be adequate from a control perspective, and a long-enough time scale to ensure controller stability, as it is larger than 3.3 seconds.

The fourth and fifth steps are to choose a PID controller type and the associated controller gains. For this proof of concept example, I chose a simple P-type controller. The control step is applied every $\Delta t = 5$ seconds, when $\mathcal{R}_{\beta-1}$ is modified by the addition of the controller error scaled by K_p. The controller error is a number between -1.00 and 1.00, since it represents the difference in mass between the QoS specification and the empirical distribution of allocated mbuf clusters. The choice of the gain constant was somewhat arbitrary; because the value of $\mathcal{R}_{\beta-1}$ for the example (as computed by SIMPLE-BOTTLENECK-ESTIMATE(), assuming a DoS attack with $p = 0.9$ and $q = 0.1$) was similar in magnitude to the errors expected from the control system (100.00 versus ± 1.00), I chose $K_p = 5$. (Recall that the error, multiplied by K_p, is added to the value of $\mathcal{R}_{\beta-1}$ by the feedback control system.) I expected that this value would be low enough to result in a slow convergence to the correct bottleneck ratio without long-term oscillations, since it was close to an order of magnitude smaller than the ideal ratio for $p = 0.9$ and $q = 0.1$. A higher value would cause the controller to overshoot quickly (perhaps more quickly than the distribution could converge), while a lower value would result in even slower convergence. As described in Section 3.2.2, this kind of heuristic tuning is typical for PID design.

Finally, the sixth and final step is to determine what filters to use to calculate $\hat{p}_{in}$ and $\hat{q}$. I chose the EWMA filter of Chapter 3 with a decay constant $\eta = 0.9$, because the EWMA filter weights recent data more heavily than historical data, a characteristic that seems to be intuitively correct for filtering bursty network traffic. Following this conjecture, I chose a large decay constant so that the filter is highly responsive and quickly forgets historical data.

5.1.7 OpenBSD Controller Implementation

The OpenBSD implementation of the mbuf cluster controller comprises a small set of static kernel modifications, a dynamically-loaded kernel module, and a user-level program.[6] These three components serve as hooks into the kernel, the admission control mechanism, and the feedback control mechanism, respectively. Although the controller could be implemented completely in the static OpenBSD kernel code, this separation of tasks is convenient from a development and reusability perspective: the static kernel modifications contain the code that is specific to mbuf clusters, while the dynamic module contains only the remaining code that must be in the kernel, and the user-level program contains everything else. I now outline the overall controller design, followed by details on each of the components of the implementation.

In order to manage the mbuf cluster state via the methods described in this thesis, the controller must *observe* that state, and then *control* it by selectively dropping mbuf cluster allocation requests. Both of these tasks (observation and control) can be performed by instrumenting the OpenBSD kernel. The kernel is a static program like any other piece of software, with the exception that it can accept small dynamically-loadable modules to add features to its defaults. For example, the static kernel supports a range of popular network cards; if a specialized card were to be used, then the changes needed to support it might be added to either the static kernel or be put in a dynamically loaded module. Since the former requires compiling the full source code and then rebooting the machine, while the latter needs only a few seconds to load or unload a module, the dynamic approach is preferred whenever possible. I chose to instrument the OpenBSD kernel by adding only the mbuf-cluster-specific components to the static kernel, with the remainder of the kernel control code in a dynamic module. These *static kernel modifications* for the mbuf cluster controller are little more than "hooks" into a *dynamic*

[6] The controller implementation was made for the OpenBSD/x86-3.4 release of November 1, 2003. Details may differ for other OpenBSD releases.

kernel module. The dynamic module provides the actual ability to perform the observation and control tasks.

The dynamic kernel module uses four functions to realize the observation and control tasks:

- the `mbcl_hardclock_hook()` function, invoked from the static `hardclock()` interrupt service routine every 10 invocations (100 milliseconds on OpenBSD/x86), records the time-series data of mbuf cluster allocations and deallocations;
- the `mbcl_inalloc_hook()` function, invoked from the static `pool_get()` function whenever an mbuf cluster is allocated, records the fact that an allocation occurred;
- the `mbcl_dealloc_hook()` function, invoked from the static `pool_do_put()` function whenever an mbuf cluster is deallocated, records the fact that a deallocation occurred; and
- the `mbcl_drophook()` function, invoked whenever a new mbuf cluster is needed from the static device-driver function `xl_newbuf()`, decides whether the allocation should be allowed.

The first hook function, `mbcl_hardclock_hook()` records the allocation and deallocation counts from the last $100ms$ time period in a ring buffer. These allocation and deallocation counts are raw totals of the number of times that the corresponding event occurred in the last 100 millisecond timeframe; these counts are incremented by the second and third hook functions, `mbcl_inalloc_hook()` and `mbcl_dealloc_hook()`. The final hook function, `mbcl_drophook()`, computes the drop probability, based upon the transition ratio table; it returns a 1 if the allocation should be admitted or a 0 if it should be dropped. With the exception of the `mbcl_drophook()` function, each of these hooks updates the empirical mbuf cluster distribution as well. The empirical distribution of the mbuf cluster state is maintained as a 10-element array (corresponding to the 10 elements of the QoS specification) in which each element represents the total amount of time (in microseconds) spent in each QoS state.

I interfaced these functions to the static kernel by using function pointers that are accessed in that kernel, but initialized when the dynamic module loads. For example, to link the `mbcl_inalloc_hook()` function (in the dynamic module) to the `pool_get()` function (in the static kernel), I defined a function pointer `mbcl_inhook()`, initially null, in the static kernel code. I then added the following code to `pool_get()`, which is called whenever an mbuf cluster allocation occurs:

```
if (mbcl_inhook)
     (*mbcl_inhook)();
```

This code invokes the function pointed to by the `mbcl_inhook` pointer if that pointer is nonnull. When loaded, the dynamic module sets the value of `mbcl_inhook` equal to the function address of `mbcl_inalloc_hook()`; when unloaded, the module sets the value of `mbcl_inhook` back to null. Hence for each component in the dynamic module, the corresponding static kernel changes are minor and small (only one comparison and a function invocation), so the overall impact upon the standard OpenBSD kernel (in terms of additional lines of code and runtime complexity) is tiny.

In addition to the hook functions described above, the dynamic module provides an interface for accessing the data needed to control the mbuf cluster state. This interface is needed by a user-level program to set the values of the drop probabilities for each state in the QoS specification, and—in support of that task—to read the empirical distribution of mbuf cluster states, the raw time-series data of mbuf cluster allocations, and the time-series data of deallocations. I made no attempt to optimize these operations in this proof-of-concept example, and in fact the user-level code is inefficient. For example, in order to filter the time-series, the user-level code processes all the data points instead of using a recursive approach on only the most recent data. As another example, the control code needed to correlate mbuf clusters to the ethernet interface that allocated them, but I did not wish to modify the OpenBSD mbuf structure (since such a modification would affect a significant amount of the static kernel code).

I chose instead to maintain an array of *status* flags, indexed by mbuf cluster, that indicated whether a particular mbuf cluster were allocated or not. Each mbuf cluster allocation and deallocation therefore required a search of up to NMBCLUSTERS (2048) array elements. To improve performance, the mbuf structure itself could be modified with the addition of a network interface flag that would require only a single comparison—instead of the array search—which would reduce the overhead from $O(\text{NMBCLUSTERS})$ to $O(1)$.

In summary, the tasks performed by the three components of the controller implementation are:

- Static Kernel Code: mbuf-cluster-specific code that calls the functions in the Dynamic Module.
- Dynamic Module: generalized (non-specific) code to record time-sensitive data (the allocation and deallocation time-series plus the empirical distribution); also includes the the admission control stage.
- User-level Program: non-kernel code that filters the the time-series data and provides the feedback control stage.

5.1.8 Results

In this section I present the results of tests of the mbuf cluster controller. The tests were designed to measure networking performance with and without control (i.e. how well could the controller stop a Denial of Service attack?). The hardware used for these tests consisted of three machines:

- the *victim*: an OpenBSD/x86-3.4 machine with two 3C509C-based Ethernet interfaces that acted as the test platform;
- the *attacker*: an OpenBSD/x86-3.4 machine connected to the victim's first interface; and

- the *bystander*: an OpenBSD/x86-3.4 machine connected to the victim's second interface.

The two 10Mb/s Ethernets were isolated, i.e. the traffic from the first did not interfere with the traffic on the second, and vice versa. In addition, neither network was attached to any other. In order to simulate a Denial of Service flood, the attacker sent a stream of UDP packets to the victim at wire speed (i.e., as fast as the 10Mb/s ethernet could accomodate them) but the victim read those packets very slowly (one every 10ms).

Before testing the control system, I conducted baseline tests between the attacker and victim machines without the Denial of Service attack active. These tests consisted of *TCP stream benchmarks*,[7] which measured the TCP throughput over the network, as well as 10,000-packet *ping floods*, which measured how well IP packets reached the victim. The stream test is a measure of usability (higher throughput means that more data can be transferred in a unit time interval), while the ping flood test is a measure of reachability (low packet loss means that data will reach its intended recipient). The TCP stream test indicated a normal network throughput of 5.6 Mb/s on the victim's first network interface (upon which the DoS attack would be initiated in the following tests), and the same throughput on the victim's second network interface (the one that was connected to the bystander machine). The ping floods were a spray of 10,000 or more ICMP Echo Request packets, each with a 512 byte payload, and each of which generated an ICMP Echo Reply from the victim;[8] they indicated zero packet loss on both network interfaces. After loading the mbuf cluster controller module, I retested the victim machine and found the results to be identical. Hence the network performance of the victim with and without the controller loaded, in the absence of a DoS attack, was the same.

I next *unloaded* the controller module, initiated the Denial of Service attack, and performed the same tests a third time. The goal in this case was to determine the "normal" behavior of the victim under a DoS attack. As expected, the victim experienced high packet loss on both interfaces (96.9% and 97.0% on the first and second interfaces, respectively). In addition, the

[7] Using the *netperf* performance testing tool[9]

[8] The command **ping -s 512 -c 10000 -f <victim IP>** was used to generate the ping floods. Note that the **-c** option specifies the number of Echo Reply packets to wait for, not the number of Echo Request packets to send, contrary to the OpenBSD/x86-3.4 manual page for *ping*.

Test type	Interface #1	Interface #2
Baseline Tests Without Control (No DoS)		
TCP Stream Throughput	5.6 Mb/s	5.6 Mb/s
Ping flood packet loss	0.0 %	0.0 %
Baseline Tests With Control (No DoS)		
TCP Stream Throughput	5.6 Mb/s	5.6 Mb/s
Ping flood packet loss	0.0 %	0.0 %
DoS on Interface #1 (No controller)		
TCP Stream Throughput	–	–
Ping flood packet loss	96.9%	97.0 %
mbuf cluster control and DoS on Interface #1		
TCP Stream Throughput	–	2.9 Mb/s
Ping flood packet loss	93.4 %	0.0 %

Table 5.1: Mbuf cluster controller test results. Without the mbuf cluster controller, the test machine's network performance was severely affected by a simulated DoS attack. With the controller, throughput was reduced on its second interface during the attack, but the victim was nevertheless was able to maintain network communications.

TCP stream test was incomplete because its control connection could not be established with the victim on either interface. This does not mean that throughput to the victim was reduced to zero, but only that the test could not be performed due to the excessive packet loss.

The goal of the final test was to show that the mbuf cluster controller designed using the ideas proposed in this thesis could restore network connectivity on the unattacked network interface. To perform this test, I loaded the mbuf cluster control system and reinitiated the Denial of Service attack on the first network interface. Under these conditions, the controller on the attacked interface prevented the mbuf clusters from being completely consumed. In particular, the first network interface exhibited a high loss rate (93.4%), while the second interface showed no loss at all. However, TCP throughput on the second interface was reduced by almost 50%. This was a result of the (already noted) inefficiency of this proof-of-concept example; if the code were optimized, the throughput could be increased. The data obtained in the tests is summarized in Table 5.1.

In summary, the addition of the mbuf cluster controller is clearly beneficial in this Denial of Service case. Without the controller, a DoS attack crippled the test machine—to a point when it became unreachable for most practical purposes. With the controller, the machine was still crippled on its attacked network interface, but the remaining interface experienced nothing more than a drop in throughput.

5.2 Example #2: Virtual Memory Pages

In this section, I outline how to use the adaptive control techniques of this thesis in the OpenBSD/x86-3.4 virtual memory subsystem. Multitasking operating systems are not restricted in their memory use to semiconductor or *physical* memory, but can also use *virtual memory*, in which the entire addressable space of a machine is available to every thread of execution. This is done by partitioning the memory space into logical *pages*, with pages of running threads *memory resident* (i.e., in physical memory), and other pages stored on secondary storage such as disk drives. If a thread attempts to read from or write to an address in a nonresident page—a so-called *page fault*—the virtual memory subsystem suspends the thread's execution, reads the page from the disk into a free page of physical memory, maps that physical page into the thread's address space, and then resumes the thread. If no free page of physical memory is available, the virtual memory subsystem chooses an active physical page from another thread, writes the page to disk, and then reuses it for the thread that faulted. Hence disk drives play an important part of the virtual memory operation, since the amount of "memory" available to a system is limited only by the size of the attached disks.

When a disk drive fails, all the pages that might have been stored on that disk must then remain resident in the physical memory in order to run. If there are many different threads of execution running, they must all compete for the limited physical memory pages, and at a high enough load that some threads will fail to obtain enough memory to execute.

The goal of the control system, in this scenario, is to provide controlled access to the physical memory pages. For example, when a disk fails and memory suddenly becomes a critical resource, it is better for some threads to die (or become slowed) from insufficient memory than for all to be crippled. This is similar to the networking example of the previous section: when a machine experiences a Denial of Service attack on one ethernet interface, it is better for the control system to degrade performance on that interface and thereby preserve functionality on the other interfaces than for all network communications to be crippled on *all* the ethernet interfaces.

In this section, I identify the physical memory allocation routines in UVM Virtual Memory system running OpenBSD/x86-3.4, design the model for a memory controller, and then describe how the controller could be used to prevent the crippling effects of memory overloads. I do not go into as great detail as in the previous section, nor do I present experimental results; this example is intended only to illustrate that the control methods in this thesis can be easily translated from one domain (network buffer control) to another (virtual memory management).

5.2.1 Physical Memory Allocation under UVM

The UVM[9] Virtual Memory system is a complete open-source virtual memory system that was designed specifically for BSD operating systems[18]. It manages memory in an OpenBSD/x86-3.4 system by allocating free physical memory pages, handling page faults, and writing or reading from disk (the *backing store*). For this thesis, I am interested in how UVM manages and allocates physical memory pages, which are a limited resource in any computer system.

UVM maintains free physical memory pages on queues called *lists*. Two free lists exist in UVM under OpenBSD/x86-3.4: a list of zero-filled pages, and a list of nonzero but unknown pages. In addition, there is a list of active pages, which identifies those pages that have been allocated to running threads of execution and are presently in use. In general, pages will move between these lists in an unpredictable fashion, but all allocation requests are satisfied through a single point of entry in the uvm subsystem, the `uvm_pagealloc_strat()` function call, and all deallocation requests occur only through the `uvm_pagefree()` call. Furthermore, both functions allocate (or deallocate) only one page at a time, and both actions are protected with a locking semaphore to ensure that the free lists are not simultaneously modified (e.g., by individual processors in a multiprocessing system). Hence, the necessary and sufficient conditions for

[9] It is uncertain what the acronym 'UVM' stands for; its author leaves the explanation as "an exercise for the reader."[18]

using the Markov Birth/Death model are satisfied by the UVM physical memory management, namely:

(1) **FINITENESS**: the physical memory pool is composed of a finite number of *pages*;

(2) **SERIALIZATION**: locking semaphores guarantee that two or more changes to the free lists never happen at the same time; and

(3) **UNIT CHANGES**: the page allocation and deallocation routines handle only one physical page at a time.

To make a physical-page controller, then, I must identify how to instrument the OpenBSD/x86-3.4 kernel in a manner similar to the mbuf cluster instrumentation.

5.2.2 UVM/OpenBSD Kernel Instrumentation

Like the mbuf cluster controller, the kernel instrumentation for the physical-page controller consists of a set of static kernel modifications, a dynamic kernel module, and a user-level management program. In the case of the mbuf cluster controller, only the static kernel changes were mbuf cluster-specific—hence to create the physical-page controller, I would need only to identify the necessary static kernel changes, and could reuse the dynamic kernel module and user-level manager.

Recall that the mbuf cluster controller used the following "hook" function calls in the static kernel:

- the `mbcl_hardclock_hook()` function, invoked from the static `hardclock()` interrupt service routine every 100 ms, that recorded the time-series data of mbuf cluster allocations and deallocations;

- the `mbcl_inalloc_hook()` function, invoked from the static `pool_get()` function, that recorded the fact that an mbuf cluster allocation occurred;

- the `mbcl_dealloc_hook()` function, invoked from the static `pool_do_put()` function, that recorded the fact that an mbuf cluster deallocation occurred; and
- the `mbcl_drophook()` function, invoked from the static `xl_newbuf()` function, that provided the 3C509C network interface driver with the ability to decide when to drop a packet.

For a physical memory controller, the corresponding hook functions would be:

- a `hardclock_hook()` function, invoked from the static `hardclock()` interrupt service routine, to record the time-series data of physical page allocations and deallocations;
- a `inalloc_hook()` function, invoked from the static `uvm_pagealloc_strat()` function, to record physical page allocations;
- a `dealloc_hook()` function, invoked from the static `uvm_pagefree()` function, to record physical page deallocations; and
- a `drophook()` function, invoked from the `uvm_pagealloc_strat()` function, to decide whether or not to allow the page allocation.

Note that the static kernel code required for physical page allocation control is similar in both functionality and scope to those required by the mbuf cluster controller, so very few changes would be required in order to develop a control system for a new type of resource. In fact, a single dynamic module and user-level manager in OpenBSD/x86-3.4 system could support a large number of independently-controlled resource pools, both inside the kernel and outside it—e.g., at the application level, in order to support resources such as the number of connections allowed to email or web ports, as long as the necessary and sufficient control conditions could be satisfied. This generality and broad applicability demonstrates the power of the control approach.

Chapter 6

Conclusion and Future Work

In this thesis I have described a novel control method to protect against the ill effects of overload conditions that is *adaptive*, *nonlinear*, and that operates upon the entire *distribution* of resource states. The approach is

- **robust**, because it makes no assumptions of cooperation between the resource manager and any external agents;
- **powerful**, because it shapes entire probability *distributions* instead of attempting to control gaussian statistics such as the average;
- **lightweight**, because it is computationally inexpensive;
- **accessible**, because the underlying modeling technique corresponds to a straightforward process model (the Birth/Death Markov Chain); and
- **universal**, because the techniques can be used in a wide array of disparate applications.

Numerical simulations and the Denial of Service example prove the effectiveness of this approach. In this chapter, I summarize the contributions of this thesis, describe a current commercial application of the method, and outline some directions for future research.

6.1 Contributions

The research contributions of this thesis are four-part, consisting of:

- a novel set of *distribution-shaping algorithms* that use classical closed-loop feedback control to achieve control objectives;
- a way to *adapt* the controller to changing operating conditions that requires knowledge only of the ebb and flow of the resource under control;
- the use of *nonlinearity* to dramatically improve the control performance over a simple linear approach; and
- the application of *Quality of Service concepts* in a nontraditional way, viz. the manner in which service is **degraded** instead of the fashion in which it is **provided.**

Application of the methods described in this thesis can improve robustness and fault tolerance of computer systems. Because the observability and controllability requirements (section 4.2) are minimal, the approach can be used at multiple levels, ranging from operating systems, to high-level programs, and even in multiple hierarchies at the same level of application.

6.2 Future Work

Future work on these adaptive, nonlinear control methods comprises both applied as well as research objectives. From an application perspective, the Denial of Service protection mechanism of chapter 5 is impractical because its control granularity (at the level of the network interface) is often too coarse to be of use against a real DoS attack. The point of the attack is to prevent users from accessing the resource, a goal that often translates into shutting down a commercial website. Since businesses may only have a single network link to their customers, a DoS protection scheme must be able to distinguish attackers from valid customers on that single link. An obvious extension to the DoS controller would be a controller at the *protocol connection* level, e.g. at every TCP flow (so that DoS flows do not disrupt normal network flows, regardless of network interface). Such an extension would involve additional overhead, since up to $(65,535)^2$ TCP flows can exist between any two machines, but the lightweight nature of this approach makes the idea possible. An important associated research topic is determining how to dynamically control the allocation of buffer space between flows such that the number of flows and the data throughput over them are maximized. The algorithms in this thesis are the subject of a United States provisional patent filing made on June 26, 2004, and as a result, the DoS-flow controller extension and its concomitant buffer allocation problem are now being studied and commercialized (i.e, as of October, 2004). The application is a high-performance World Wide Web caching engine to be used for large on-line businesses, with planned deployment in 2005. Finally, because operating systems are essentially complicated software resource management systems, I plan to investigate and apply the methods in this thesis in operating system modules including memory managers (virtual memory, caching, etc.) and schedulers, although any resource management subsystem that satisfies the necessary and sufficient conditions in Chapter 4 is a potential control candidate.

From a wider research perspective, a limitation of the approach in this thesis is its dependence upon the Markov Birth/Death process model. Although this model is widely applicable, certain resource management systems do not satisfy the necessary and sufficient conditions outlined in section 4.2 for use of the control algorithms in section 4.3. Obtaining an understanding of how to use feedback control on more-complicated process models, with their complex stochastic dynamics, is my long-term goal.

Finally, use of process models necessitates a clear understanding of the models' convergence characteristics, about which little is presently known. The numerical results obtained in this thesis indicate that process models are promising, but formal proofs are needed. I also plan to investigate convergence characteristics under specific demand distributions, as I expect that certain distributions (e.g., Poisson) may result in faster convergence.

Bibliography

[1] T. Abdelzaher, K. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. IEEE Transactions on Parallel and Distributed Systems, 13(1):80–96, 2002.

[2] L. Abeni and G. Buttazzo. QoS Guarantee using Probabilistic Deadlines. In 11th Euromicro Conference on Real-Time Systems, York, England, June 1999.

[3] G. Alexander. "Hackers strike fear into net businesses". `http://web.lexis-nexis.com/universe`, 2000.

[4] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long. Managing Flash Crowds On the Internet. In IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2003), pages 246–249, 2003.

[5] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long. Modeling, analysis, and simulation of flash crowds on the internet. Technical Report UCSC-CRL-03-15, University of California, Santa Cruz, February 2004.

[6] K. Åström and T. Hägglund. Automatic Tuning of Simple Regulators with Specifications on Phase and Amplitude Margins. Automatica, 20, 1984.

[7] K. Åström and T. Hägglund. PID Controllers: Theory, Design, and Tuning. Instrument Society of America, Research Triangle Park, 1995.

[8] S. Banachowski, J. Wu, and S. Brandt. Missed deadline notification in best-effort schedulers. In Proceedings of the Conference on Multimedia Computing and Networking (MMCN 2004), jan 2004.

[9] The Netperf Benchmark. `http://freshmeat.net/redir/netperf/7067/url_homepage/NetperfPage.html`.

[10] D. Bernstein. SYN Cookies. `http://cr.yp.to/syncookies.html`, 1996.

[11] G. Box and G. Jenkins. Time Series Analysis, Forecasting and Control. Holden-Day, Incorporated, 1990.

[12] CERT Coordination Center. Denial-of-Service Attack via Ping. `http://www.cert.org/advisories/CA-1996-26.html`, 1996.

[13] CERT Coordination Center. TCP SYN flooding and IP spoofing attacks. `http://www.cert.org/advisories/CA-1996-21.html`, 1996.

[14] A. Cha and D. Streitfeld. "Microsoft Web Sites Attacked; Company Asks FBI For Assistance". `http://web.lexis-nexis.com/universe`, 2001.

[15] K. Cho. A framework for alternate queueing: towards traffic management by PC-UNIX based routers. In USENIX '98 Annual Technical Conference, pages 247–258, New Orleans, june 1998.

[16] D. Comer. Internetworking with TCP/IP. Prentice-Hall, Englewood Cliffs, New Jersey, 4 edition, 2000.

[17] 3Com Corporation. 3C90xC NICs Technical Reference. 3Com Corporation, Santa Clara, California, 1999.

[18] C. Cranor. Design and Implementation of the UVM Virtual Memory System. PhD thesis, Washington University, 1998.

[19] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. SIGCOMM '89, 1989.

[20] C. Derman. Finite State Markovian Decision Processes. Academic Press, San Diego, California, 1970.

[21] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. In Proceedings of Network Operations and Management Symposium, Florence, Italy, apr 2002.

[22] A. Elwalid and D. Mitra. Traffic Shaping at a Network Node: Theory, Optimum Design, Admission Control. In IEEE INFOCOM, 1997.

[23] B. Braden et al. Recommendations on Queue Management and Congestion Avoidance in the Internet. RFC 2309, 1998.

[24] E. Feinberg and A. Shwartz. Handbook of Markov Decision Processes. Kluwer Academic Publishers, Boston, Massachusetts, 2002.

[25] W. Feller. An Introduction to Probability Theory and Its Applications. John Wiley and Sons, Inc., New York, New York, 3 edition, 1968.

[26] W. Feng, D. Kandlur, D. Saha, and K. Shin. BLUE: A new class of active queue management algorithms.

[27] W. Feng, D. Kandlur, D. Saha, and K. Shin. A Self-Configuring RED Gateway. In IEEE INFOCOM, pages 1320–1328, 1999.

[28] W. Feng, D. Kandlur, D. Saha, and K. Shin. Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness. In IEEE INFOCOM, 2001.

[29] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, 2000.

[30] S. Floyd. TCP and explicit congestion notification. ACM Computer Communication Review, 24(5):10–23, 1994.

[31] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An Algorithm for Increasing the Robustness of RED, 2001.

[32] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. IEEE/ACM Transactions on Networking, 1(4):397–413, 1993.

[33] H. Fowler and W. Leland. Local Area Network Traffic Characteristics, with Implications for Broadband Network Congestion Management. IEEE J. Selected Areas in Communications, 9:1139–1149, September 1991.

[34] R. Gabel and R. Roberts. Signals and Linear Systems. John Wiley and Sons, Inc., New York, New York, 1987.

[35] N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Managing the Performance of Lotus Notes: A Control Theoretic Approach. In Proceedings of the Computer Measurement Group, Anaheim, California, 2001.

[36] A. Garg and A. Reddy. Mitigation of DoS attacks through QoS regulation. In Proceedings of IEEE International Workshop on Quality of Service (IWQoS), Miami Beach, may 2002.

[37] L. Georgiadis, R. Guérin, V. Peris, and K. Sivarajan. Efficient network QoS provisioning based on per node traffic shaping. IEEE/ACM Transactions on Networking, 4(4):482–501, 1996.

[38] G. Goodwin, S. Graebe, and M. Salgado. Control System Design. Prentice-Hall, Englewood Cliffs, New Jersey, 2001.

[39] S. Haykin. Introduction to Adaptive Filters. Macmillan, New York, New York, 1984.

[40] D. Henriksson, Y. Lu, and T. Abdelzaher. Improved Prediction for Web Server Delay Control. In Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04), pages 61–68, June 2004.

[41] O. Hernandez-Lerma and J. Lasserre. Discrete-Time Markov Control Processes. Springer-Verlag, New York, New York, 1996.

[42] C. Hollot, V. Misra, D. Towsley, and W. Gong. A Control Theoretic Analysis of RED. In IEEE INFOCOM, 2001.

[43] A. Householder, A. Minion, L. Pesante, G. Weaver, and R. Thomas. Managing the Threat of Denial-of-Service Attacks. CERT Coordination Center, 2001.

[44] J. Howard. An analysis of security incidents on the Internet 1989-1995. PhD thesis, Carnegie Mellon University, 1998.

[45] V. Jacobson. Minutes of the performance working group. In Proceedings of the Cocoa Beach Internet Engineering Task Force, Reston, Virginia, April 1989.

[46] M. Jerrum and A. Sinclair. Conductance and the rapid mixing property of Markov Chains: The approximation of the permanent resolved. In Proceedings of the ACM Symposium on Theory of Computing (STOC),, pages 235–244. ACM Press, 1988.

[47] M. Jerrum and A. Sinclair. Approximating the permanent. SIAM Journal on Computing, 18(6):1149–1178, 1989.

[48] M. Jerrum and A. Sinclair. Polynomial-time approximation algorithms for the Ising model. Society for Industrial and Applied Mathematics Journal on Computing, 22(5):1087–1116, 1993.

[49] N. Kahale. A semidefinite bound for mixing rates of markov chains. Technical Report 95–41, AT&T Bell Laboratories, September 1995.

[50] S. Karlin and H. Taylor. A First Course in Stochastic Processes. Academic Press, San Diego, California, 1975.

[51] M. Kitaev and V. Rykov. Controlled Queueing Systems. CRC Press, New York, New York, 1995.

[52] A. Kolmogorov. Foundations of the Theory of Probability. Chelsea Publishing Company, New York, New York, 1956.

[53] A. Kuzmanovic and E. Knightly. Low-rate TCP-targeted denial of service attacks: The shrew vs. the mice and elephants. In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, pages 75–86. ACM Press, 2003.

[54] R. Lebihan. "SCO Rides Out The Mydoom Storm". `http://web.lexis-nexis.com/universe`, 2004.

[55] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the Self-Similiar Nature of Ethernet Traffic. IEEE Transactions on Networking, 2(1):1–14, February 1994.

[56] J. Lemon. Resisting SYN flood DoS attacks with a SYN cache. In Proceedings of USENIX BSDCon 2002, San Francisco, February 2002.

[57] D. Lin and R. Morris. Dynamics of random early detection. In SIGCOMM '97, pages 127–137, Cannes, France, september 1997.

[58] C. Liu and R. Jain. Improving explicit congestion notification with the mark-front strategy. Computer Networks, 35(2–3):185–201, 2001.

[59] G. Lovasz. Combinatorial Problems and Exercises. North-Holland, 1 edition, 1979.

[60] L. Lovasz and R. Kannan. Faster mixing via average conductance. In Proceedings of the thirty-first ACM symposium on Theory of Computing, pages 282–287. ACM Press, 1999.

[61] C. Lu. Feedback Control, Real-Time Scheduling. PhD thesis, University of Virginia, 2001.

[62] C. Lu, T. Abdelzaher, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In Proceedings of the IEEE Real-Time Technology and Applications Symposium, pages 51–62, June 2001.

[63] C. Lu, G. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In USENIX Conference on File and Storage Technologies, 2002.

[64] C. Lu, J. Stankovic, G. Tao, and S. Son. Design and Evaluation of a Feedback Control EDF Scheduling Algorithm. In Proceedings of the 20th IEEE Real-Time Systems Symposium, 1999.

[65] C. Lu, J. Stankovic, G. Tao, and S. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. Journal of Real-Time Systems, 23(1/2):85–126, 2002.

[66] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services. In Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002), 2002.

[67] Y. Lu, A. Saxena, and T. Abdelzaher. Differentiated caching services: a control-theoretical approach. In Proceedings of the 21st International Conference on Distributed Computing Systems, pages 615–624, apr 2001.

[68] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron. Control-theoretic dynamic frequency and voltage scaling. In Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems, 2002.

[69] E. Luttwak. "Vandals threaten e-commerce". `http://web.lexis-nexis.com/universe`, 2000.

[70] A. Mankin. Random Drop Congestion Control. In Proc. ACM SIGCOMM '90; (Special Issue Computer Communication Review), pages 1–7, 1990.

[71] D. Marchette. A statistical method for profiling network traffic. In First USENIX Workshop on Intrusion Detection and Network Monitoring, pages 119–128, Santa Clara, CA, April 1999.

[72] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In USENIX Winter, pages 259–270, 1993.

[73] P. McKenney. Stochastic Fairness Queueing. In IEEE INFOCOM, pages 733–740, 1990.

[74] M. McKusick, K. Bostic, M. Karels, and J. Quaterman. The Design and Implementation of the 4.4BSD Operating System. Addison-Wesley, Reading, Massachusetts, 1996.

[75] T. McLaughlin. "Hackers attacking Internet sites with impunity". `http://web.lexis-nexis.com/universe`, 2000.

[76] J. Mirkovic. D-WARD: Source-End Defense Against Distributed Denial-of-Service Attacks. PhD thesis, University of California, Los Angeles, 2003.

[77] J. Mirkovic, J. Martin, and P. Reiher. A Taxonomy of DDoS Attacks and DDoS Defense Mechanisms. Technical Report 020018, University of California, Los Angeles, February 2001.

[78] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In Proceedings of the IEEE International Conference on Network Protocols, Paris, France, nov 2002.

[79] V. Misra, W. Gong, and D. Towsley. Fluid-based analysis of a network of AQM routers supporting TCP flows with an application to RED. In SIGCOMM, pages 151–160, 2000.

[80] D. Moore, G. Voelker, and S. Savage. Inferring internet Denial-of-Service activity. In Proceedings of the 2001 USENIX Security Symposium, pages 9–22, 2001.

[81] B. Morris and Y. Peres. Evolving sets and mixing. In Proceedings of the thirty-fifth ACM symposium on Theory of Computing, pages 279–286. ACM Press, 2003.

[82] R. Motwani and P. Raghavan. Randomized Algorithms. Cambridge University Press, San Diego, California, 1 edition, 1995.

[83] R. Nelson. Probability, Stochastic Processes and Queueing Theory. Springer-Verlag, New York, New York, 1995.

[84] J. Norris. Markov Chains. Cambridge University Press, San Diego, California, 1997.

[85] C. Nuttall. "Hackers blackmail internet bookies: Criminals believed to be targeting Grand National". `http://web.lexis-nexis.com/universe`, 2004.

[86] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In IFIP/IEEE International Symposium on Integrated Network Management, Seattle, Washington, May 2001.

[87] V. Paxson and S. Floyd. Wide Area Traffic: The Failure of Poisson Modelling. IEEE Transactions on Networking, 3(3):226–244, June 1995.

[88] T. Peng, C. Leckie, and K. Ramamohanarao. Defending against distributed denial of service attack using selective pushback. In Ninth IEEE International Conference on Telecommunications (ICT 2002), 2002.

[89] L. Phillips and R. Harbor. Feedback Control Systems. Prentice-Hall, Englewood Cliffs, New Jersey, 1999.

[90] The OpenBSD Project. `http://www.openbsd.org`.

[91] M. De Prycker. Asynchronous Transfer Mode. Ellis Horwood, New York, New York, 1993.

[92] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build DoS-Resistant Software. In Fifth ACM Symposium on Operating Systems Design and Implementation (OSDI '02), pages 45–60, 2002.

[93] K. Ramakrishnan and S. Floyd. A Proposal to add Explicit Congestion Notification (ECN) to IP, 1999.

[94] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP, 2001.

[95] M. Reiman and A. Shwartz. Call Admission: A New Approach to Quality of Service. Queueing Systems, 38:125–148, 2001.

[96] M. Reisslein, K. Ross, and S. Rajagopal. A framework for guaranteeing statistical QoS. Technical report, Arizona State University, Dept. of Electrical Engineering, September 2001.

[97] A. Robertsson, B. Wittenmark, and M. Kihl. Analysis and design of admission control in web-server systems. In Proceedings of ACC'03, 2003.

[98] S. Ross. Introduction to Stochastic Dynamic Programming. Academic Press, San Diego, California, 1983.

[99] S. Ross. Simulation. Academic Press, San Diego, California, 1997.

[100] S. Ross. Introduction to Probability Models. Academic Press, San Diego, California, 2000.

[101] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In Proceedings of SIGCOMM 2000, pages 295–306, 2000.

[102] Reuters News Service. "Blackmail latest scam for ambitious hackers". `http://web.lexis-nexis.com/universe`, November 2003.

[103] Reuters News Service. "Increase in web traffic not always a blessing". `http://web.lexis-nexis.com/universe`, 2003.

[104] S. Siewert. A Real-Time Execution Performance Agent Interface for Confidence-Based Scheduling. PhD thesis, University of Colorado, 2000.

[105] A. Sinclair and M. Jerrum. Approximate counting, uniform generation and rapidly mixing Markov Chains. Information and Computation, 82(1):93–133, 1989.

[106] K. Skadron, T. Abdelzaher, and M. Stan. Control-theoretic techniques and thermal-rc modeling for accurate and localized dynamic thermal management. In Proceedings of the 8th International Symposium on High-Performance Architecture (HPCA'02), 2002.

[107] J. Slotine and W. Li. Applied Nonlinear Control. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[108] J. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback Control Scheduling in Distributed Real-Time Systems. In IEEE Real-Time Systems Symposium, 2001.

[109] D. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A feedback-driven proportional allocator for real-rate scheduling. In Proceedings of Operating Systems Design and Implementation (OSDI '99), pages 145–158, 1999.

[110] R. Stevenson. "New computer virus targets Microsoft". `http://web.lexis-nexis.com/universe`, February 2004.

[111] D. Stroock. Probability Theory: An analytic view. Cambridge University Press, San Diego, California, 1994.

[112] Internet Security Systems. "Snork" Denial of Service Attack Against Windows NT RPC Service. `http://www.iss.net/security_center/alerts/advise9.php`.

[113] F. Tobagi, A. Markopoulou, and M. Karam. A statistical method for profiling network traffic. In Tyrrhenian International Workshop on Digital Communications (IWDC 2002), Capri, Italy, September 2002.

[114] I. Todhunter. A History of the Mathematical Theory of Probability From the Time of Pascal to that of Laplace. American Mathematical Society, Chelsea, 1949.

[115] J. Treichler, C. Johnson, and M. Larimore. Theory and Design of Adaptive Filters. John Wiley and Sons, Inc., New York, New York, 1987.

[116] S. Velusam, K. Sankaranarayanan, and D. Parikh. Adaptive cache decay using formal feedback control. In Proceedings of the 2002 Workshop on Memory Performance Issues, 2002.

[117] D. White. A Survey of Applications of Markov Decision Processes. Journal of Operational Research Society, 44(11):1073–1096, 1993.

[118] W. Willinger, M. Taqqu, R. Sherman, and D. Wilson. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. IEEE Transactions on Networking, 5(1):71–86, February 1997.

[119] D. Wu, Y. Hou, W. Zhu, Y. Zhang, and J. Peha. Streaming video over the internet: Approaches and directions. IEEE Transactions on Circuits and Systems for Video Technology, 11(1):1–20, February 2001.

[120] C. Yu. Autotuning of PID Controllers. Springer-Verlag, New York, New York, 2001.

[121] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In Proceedings of ICDCS, jul 2002.

[122] Y. Zhu and F. Mueller. Feedback dynamic voltage scaling dvs-edf scheduling: Correctness and pid-feedback. Technical Report TR-2003-13, North Caroline State University, June 2003.

[123] J. Ziegler and N. Nichols. Optimum Settings for Automatic Controllers. Trans. ASME, 12, 1942.