OPTIMALLY SELECTING THE PARAMETERS OF ADAPTIVE BACKOFF ALGORITHMS FOR COMPUTER NETWORKS AND MULTIPROCESSORS

DISSERTATION
SUBMITTED IN PARTIAL SATISFACTION OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN COMPUTER SCIENCE
IN THE GRADUATE DIVISION
OF THE UNIVERSITY OF CALIFORNIA, BERKELEY

Peter B. Danzig

December 1, 1989

OPTIMALLY SELECTING THE PARAMETERS OF ADAPTIVE BACKOFF

ALGORITHMS FOR COMPUTER NETWORKS AND MULTIPROCESSORS

Optimally Selecting the Parameters of Adaptive Backoff Algorithms for Computer Networks and Multiprocessors

by

Peter B. Danzig

ABSTRACT

This dissertation examines the solutions, based on software, adaptive backoff algorithms, to two computer network problems and a shared-memory multiprocessor problem. For each problem we define a cost metric against which we adjust the adaptive backoff algorithm. This dissertation's unifying theme is that, when possible, backoff algorithms should tune themselves to their environment. The algorithms that we propose are software remedies to hardware shortcomings; they do not require changes to the hardware, however warranted these changes might be, but they may require that we periodically measure certain expected values or probability distributions.

We devote the majority of this dissertation to studying buffer overflow as it occurs during reliable, local area network, multicast. We develop a multiple round, soft real-time algorithm that trades latency for computational overhead: an n-round multicast is slower but suffers less computational overhead than an (n+1)-round multicast. Our prototype system measures the buffer service time distribution and employs it to calculate the algorithm's retransmission timeouts. We develop a preemptive, limited buffer queueing model that accurately models an operating system's communication protocol processes.

We study a memory contention problem that occurs during synchronization of bus-oriented, shared-memory multiprocessors with snoopy, invalidation-based caches. The contention occurs when such multiprocessors cache lock variables, lack advanced synchronization instructions, and synchronize with a test-and-set instruction embedded in a busy waiting loop. This type of synchronization structure has been dubbed a spin-lock. When a spin-lock is released, the cache invalidation signal can cause a burst of memory activity that we call an invalidation storm. Remedies for invalidation storms can waste memory cycles. Our spin-lock backoff algorithm wastes twenty to fifty percent fewer cycles than a recently proposed algorithm.

We consider how to calculate remote procedure call retransmission timeouts on lossy networks and on tariff-bearing networks with selectable grades of service. We develop an expression to calculate the optimal retransmission timeout and network service grade that minimizes a cost function composed of computational overhead, round trip service time, and network tariffs.

Domenico Ferrari, Committee Chair

''For a successful technology, reality must take precedence
over public relations, for Nature can not be fooled''.
--Richard P. Feynmann [30]

ACKNOWLEDGEMENTS

The last time I wrote a very long report I was an innocent sixth grader. I wrote an eighty page report on Colombia, which, to a great extent, was simply a paraphrase of the relevant entries from the Encyclopedia Britanica. At the time I was engaged in an unspoken sibling rivalry with my eldest sister who had written a lengthy report on Bolivia the previous year. Writing this long report I competed with no one but myself, and I wish to thank the many people who contributed to it.

Domenico Ferrari's encouragement, subtle advice, and complete academic freedom make him a model research advisor. I must also thank him for his continuous financial support for four and a half years, through both productive and arid seasons. Fyodor Dostoyevski wrote in his *Grand Inquisitor* that few people want absolute freedom; rather they prefer to live within the confines of the rules that others set for them. This also holds for academic freedom. The academic freedom that Domenico lends his students makes them better researchers. The confidence the he displays in his students helps them through the frustrating times that absolute freedom entails.

I want to thank Luis-Felipe Cabrera, George Shanthikumar, and Alan J. Smith for reading this dissertation and participating on my qualifying and dissertation committees. In particular, Alan's brutal comments contributed significantly to the clarity of Chapters 2 and 4.

Many fellow graduate students, too many to mention individually, contributed to my sojourn in Berkeley. A handful of these deserve special mention. For their friendship and their technical suggestions, I wish to thank Ramon Caceres, Fred Douglis, Riccardo Gusella, Corinna Lee, Steve Melvin, Luis Miguel, Carl Ponder, Venkat Rangan, Stuart Sechrest, Mark Sullivan, Shin-Yuan Tzou, and Songnian Zhou. I also wish to thank Kathryn Crabtree, Teresa Diaz, Liza Gabota, and Jean Root for their friendship. I wish to thank Riccardo Gusella for encouraging me to build the microsecond resolution clock with which the data for Chapter 3 was taken, and Steve Melvin for his partnership in their design, debugging, fabrication, and distribution. Distributing these timers to other research groups has been extremely rewarding. I must thank LTH for listening to my daily status reports, my woes, and my successes. It is to her support that I attribute this dissertation.

The work that appears here has absorbed most of my time for two years, including much of the time that I would ordinarily have spent with my wife. For this, I beg her pardon.

I thank the American Electronics Association for the Faculty Development Fellowship that it awarded me, and Measurex Corporation for providing the fellowship's funds and for sponsoring my two trips to SOSP.

This work has been supported in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 4871, monitored by the Naval Electronic Systems Command under Contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the author and should not be interpreted as representing official policies, either expressed or implied, of any of the sponsoring organizations or of the U.S. Government.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This dissertation examines three problems and their solutions, based on software, adaptive backoff algorithms. Of these problems, two deal with computer networks and one with multiprocessor computer architectures. We consider how to tune each problem's adaptive backoff algorithm to its environment. This dissertation's unifying theme is that, when possible, backoff algorithms should tune themselves to their environment, which may require that we periodically measure certain expected values or probability distributions. This principle is not new; for example, countless papers have suggested that we change CSMA/CD hardware to measure the number of colliding stations and to select the backoff interval accordingly. The algorithms that we propose are software remedies to hardware shortcomings; they do not require changes to the hardware, however warranted these changes might be.

Colloquially, as a consequence of the Ethernet's success, we say that the binary, exponential backoff algorithm is adaptive. Much effort has been spent showing that it meets various definitions of stability [35]. One common definition of stability is that the system's state eventually renews. The other common definition is that, regardless of the system's load, the system efficiently makes progress. System builders now incorporate binary exponential backoff into their software communication protocols [4, 12] because binary exponential backoff's stability assures that their systems will avoid server or network congestion. However, an algorithm that is adaptive in one situation, may not be adaptive, or may adapt incorrectly, in another. When, for example, we know a priori the number of participants that obey the backoff algorithm, stability is less of an issue than efficiency, which, like stability, requires definition.

We tune, or optimize, our backoff algorithms with respect to a cost metric that we define, and for whose quality we are alone responsible. The metrics used in this dissertation are functions of an operation's elapsed time and computational overhead. There exists no single, ideal operating point, appropriate for every problem. As, for example, at times we endure paying a premium for overnight delivery of our conference submissions that we would not pay for our annual new year's cards. Therefore, for each problem, we must define its cost metric and assign relative weights to its elapsed time and computational overhead.

This dissertation consists of seven chapters. The next three chapters, 2, 3, and 4, address the three problems mentioned above. Chapter 5 discusses how to make high resolution timing measurements. Chapter 6 proposes avenues for future work, and Chapter 7 reviews this dissertation's contributions. Chapters 2 through 6 are self-contained. Each chapter consists of an introduction to the problem, a survey of previous work, and conclusions. Below, we briefly highlight each chapter.

1.1. Outline

In Chapter 2, we examine memory bus contention due to spin-lock synchronization on shared-memory multiprocessors with snoopy, invalidation-based, caches [54]. For critical sections guarded by spin-locks, we improve the performance of a recently published backoff algorithm [3,4,6,7]. When implementing barrier synchronization with spin-locks, we give an algorithm which performs much better than any published algorithm to date. For instance, it yields a three-fold performance improvement for a sixteen processor barrier.

In Chapter 3, the dissertation's central and longest chapter, we develop a LAN, multiple round, reliable multicast algorithm, and derive and experimentally justify an analytical expression for the distribution of buffer overflows suffered by the multicast's initiator. The algorithm's latency decreases with the number of rounds, but the algorithm's overhead increases with it. We replace the sender's traditional retransmission timeout with a timeout table, tuned to the sender's measured service time distribution, which the sender transmits with the multicast. The multicast's recipients backoff over this timeout to reduce the severity of buffer overflow.

In Chapter 4, we investigate the effects of network service grade on protocol retransmission timeouts. The era of slow, free[1], long-haul computer networks is coming to a close, and the era of fast, tariff-bearing networks offered by public providers is emerging [31]. These new networks will offer a spectrum of service grades, and will charge tariffs accordingly. Certainly, two principal service grade parameters will be the network's transmission delay and probability of message loss. For such an environment, we propose a cost metric and derive an expression for simultaneously selecting remote procedure call retransmission timeouts and network service grade that optimize the cost metric.

In Chapter 5, we document a microsecond resolution timer that we built to take certain measurements presented in Chapter 3, and we derive the number of measurements needed to report performance times to a given precision as a function of the clock resolution with which measurements are taken, as well as of the desired degree of confidence. The derivation is based on the DeMoivre-Laplace limit theorem.

In Chapter 6, we propose extending the multiple round LAN multicast algorithm that we developed in Chapter 3 to internetwork multicast, and we propose extending the work developed in Chapter 4 to rate-based flow control algorithms for gigabit, tariff-bearing, virtual circuit networks.

In Chapter 7, we consider our work's relevance, recognizing that our problems arose from recognized hardware shortcomings: advanced hardware synchronization support could eliminate the contention caused by spin-waiting on shared-memory multiprocessors with snoopy, invalidation-based caches; additional buffer memory could eliminate buffer overflow due to LAN multicast; and faster and more reliable wide area networks could diminish congestion and the importance of protocol retransmission timeouts.

[1]Free, at least, to the academic world.

CHAPTER 2

BUS CONTENTION DUE TO SPIN-LOCKS

Broadcasts can initiate contention, regardless of whether the medium on which they propagate is a hardware bus or a computer network. The broadcasts' recipients, upon trying to respond, compete for exclusive access to the medium. On CSMA/CD networks, for example, a broadcast can cause a flurry of collisions, which has been called a *broadcast storm.* In this chapter we investigate a similar bus contention problem suffered by shared-memory multiprocessors.

We restrict our attention to bus-oriented, shared-memory multiprocessors with per processor snoopy caches [25,54] that cache synchronization variables and that, upon updating an item shared across more than one cache, achieve consistency by invalidating the item from all other caches but the cache at which it is written. (It does not matter whether the caches are write-through or write-back). With such caches, when a processor writes a shared item, it broadcasts an invalidation signal to the other processors' snoopy cache controllers which then invalidate, rather than update, their copies of the item.

Since it is this invalidation signal that initiates the contention problem we are studying, we call it the *invalidation storm* problem. Processors that lack advanced synchronization support such as fetch_and_op instructions [24] implement synchronization locks from a busy waiting loop that exits when a test-and-set instruction succeeds. Such locks have been called *spin-locks.* Busy waiting processors generate very little memory bus traffic while the lock is locked; their own caches satisfy their memory references. When a processor releases the lock, and invalidates the spinning processors' cached copy of the lock variable, each spinning processor reads, or *faults*, the new value of the lock variable into its cache, discovers that the lock is unlocked, and attempts to lock it. Except for the first attempt, which succeeds, these attempts fail. However each attempt invalidates the lock variable, and causes those processors that have already resumed spinning to fault the lock value again. Given that n processors were spinning, this can lead to $(n-2)(n-1)/2$ extra reads. We call any read or write to a lock variable that passes over the shared-bus a *contention cycle.*

A backoff algorithm proposed by Anderson [4] significantly reduces the number of contention cycles. Anderson's algorithm, however, introduces wasted, or *idle cycles*, during which no spinning processor attempts to acquire an available lock. We describe a spin-lock backoff algorithm that achieves a similar reduction in contention cycles, and demonstrate through simulation that it reduces these idle cycles twenty to fifty percent. We propose a barrier synchronization algorithm, and demonstrate that, for a sixteen processor barrier, it can yield a three-fold improvement in contention and idle cycles over Anderson's solution. We also identify a possible problem with Anderson's performance evaluation.

2.1. Introduction

In this chapter we investigate a synchronization related bus contention problem suffered by bus-oriented, shared-memory multiprocessors that employ snoopy caches that achieve consistency by invalidating, rather than updating, items shared across caches, that cache synchronization variables, but provide only a test_and_set (or atomic_exchange) synchronization instruction. Such processors, like the Sequent Symmetry [54], implement synchronization locks from a busy-waiting loop that exits when a test-and-set instruction succeeds, indicating that the lock has been acquired. This type of lock is called a spin-lock, and permits two operations: *Acquire_Lock* and *Release_Lock* (see Figure 2.1).

```
Acquire_Lock ( Lock *L )
{
    for (;;) {
       while ( *L == LOCKED ); /* spin - do nothing */
       if ( test_and_set  ( L, LOCKED ) == UNLOCKED ) break;
    }
}

Release_Lock ( Lock *L )
{
  Lock = UNLOCKED:
}
```

Figure 2.1. Implementation of a spin-lock on a bus-based, shared-memory, multiprocessor with only a test_and_set instruction for synchronization. A processor spins within its cache while the lock is LOCKED, reflected by the *while* loop. Executing *test_and_set* invalidates the lock variable from the other processors' caches, regardless of whether or not the lock's value changes. Busy waiting processor generate very little memory bus traffic while the lock is locked; their own caches satisfy their own memory references. However, the processor that releases the lock invalidates the spinning processors' cached copies of the lock variable, which all fault in the lock's new value.

2.1.1. Invalidation Storms

When a processor tests-and-sets a spin-lock, the cache consistency protocol invalidates the other processors' cached copies of the lock variable. Due to the cache consistency protocol we are considering (see above), processors that are busy-waiting, attempting to acquire the lock, fault the lock's new value into their caches and consume a bus cycle each. As we explain below, releasing a lock can initiate a flurry of invalidations and faults which we dub an *invalidation storm.* We show that if n processors are executing *Acquire_Lock*, an invalidation storm can consume up to $(n-2)(n-1)/2$ bus cycles. We call any reference to a lock variable that passes over the shared-bus a *contention cycle*.

Consider what happens when n processors are spinning inside *Acquire_Lock* as coded in Figure 2.1, and the lock holder releases the lock, invalidating the other n processors' copies of the cached lock variable. All n fault in the lock variable's new value, see that the lock is unlocked, and execute *test_and_set*. One of these successfully acquires the lock, invalidates the other processors' caches, and enters its critical section. One of the remaining $n-1$ processors, the next to execute *test_and_set*, and resumes spinning. One of the remaining $n-2$ processors, the next to execute *test_and_set*, invalidates the cached lock variables again and resumes spinning. This invalidation causes the processor that had resumed spinning to fault in the lock variable. One of the remaining $n-3$ processors, the next to execute *test_and_set*, invalidates the other processors' caches once again. This causes the two processors that have resumed spinning to fault in the lock variable. This storm of invalidations and faults continues until all the processors have executed *test_and_set* once. Along the way, besides the n initial faults and *test_and_set* instructions, S_n more faults may occur:

$$S_n = 1 + 2 + \cdots + (n-2) = (n-2)(n-1)/2 .$$

Suppose, for example, a dozen processors are spinning inside *Acquire_Lock* as might occur during barrier synchronization (see Figure 2.6). Each time the lock is released an invalidation storm ensues. Therefore the sum of the number of bus cycles lost to this barrier is $S_{12} + \cdots + S_2 = 220$, and grows cubicly.

2.1.2. Related Work

Recently, two research groups independently proposed [3] and implemented [4] Ethernet-like backoff algorithms to reduce the severity of invalidation storms. Their algorithms introduce a certain number of cycles during which, even though many processors are trying to acquire the lock, no processor executes test_and_set. We call these wasted cycles *idle cycles*. In this chapter we introduce a backoff algorithm of our own, and compare it to the recently published backoff algorithm of Anderson [4, 7].

2.1.3. Outline

In the next section we review Anderson's algorithm and introduce ours. We contrast their relative performance via simulation in Section 2.3. In Section 2.4 we review implementing barrier synchronization from spin-locks[1], explain why one should not depend on spin-lock backoff alone, explain our barrier backoff algorithm, and examine its performance. We discuss a possible flaw in Anderson's performance metric in Section 2.5, and, in Section 2.6 we review our conclusions.

2.2. The Two Backoff Algorithms

In this section we present Anderson's backoff algorithm and our backoff algorithm and explain the rationale behind them. We start with Anderson's algorithm.

2.2.1. Anderson's Backoff Algorithm

Contrast Anderson's *Acquire_Lock* algorithm, Figure 2.2, with the basic *Acquire_Lock* algorithm presented in Figure 2.1. Anderson implements, in essence, the CSMA/CD collision backoff algorithm which has been proven stable under any load [35]. (Of course, the basic algorithm also meets this definition of stability because some processor acquires the lock whenever it is released). A processor spins within its own cache while the lock is locked. When the lock is released, the processor waits a random number of cycles, chosen uniformly over the backoff range m. On one hand, if some other processor acquires the lock during this time, the processor faults in the lock's new value and does not execute the *test_and_set*, nipping the invalidation storm in the bud. It then doubles its backoff range m and resumes spin-waiting. On the other hand, if the lock is still free after backoff, the processor executes *test_and_set*, and, either acquires the lock and exits the loop, or fails to acquire it and doubles its backoff range m, and resumes spin-waiting. The algorithm eliminates propagating invalidations at the expense of the extra test before the *test_and_set*. Since $m-1$ is zero on the first time through the loop, the algorithm is partial to new arrivals, and, like all exponential backoff algorithms, is unfair [58].

The backoff range m grows with the number of processors that are simultaneously executing *Acquire_Lock*. The number of processors is a random variable that can not exceed the number of processors. Regardless of the number of processors that simultaneously execute *Acquire_Lock*, the stability of exponential backoff guarantees that, after the backoff range has grown sufficiently, the invalidation storm will be eliminated.

[1]Barrier synchronization on the Sequent Symmetry does not need spin-locks because its special atomic synchronization instruction *lock* permits one to construct a barrier that does not suffer from invalidation storms. In this regard, certain comments in [4] were misleading.

2.2.2. Our Backoff Algorithm

Contrast our backoff algorithm, Figure 2.3, with Anderson's algorithm, Figure 2.2. Again, a processor spins within its own cache while the lock is locked. Instead of backing off, it immediately attempts the *test_and_set*, only backing off if this fails. We also employ exponential backoff, and benefit from its stability guarantee.

We limit the maximum value of the backoff range to promote fairness between new and old arrivals, and direct the compiler to set *mMin* to an estimate of the minimum duration of the critical section that the lock guards. The intent of *mMin* is that, all processors spin silently during the time that invalidations take place.

Our algorithm backs off after executing *test_and_set*, rather than before executing it, to reduce the number of idle cycles wasted before a processor acquires the lock. Reducing idle cycles, in turn, reduces the effective duration of the critical section that the lock guards and decreases contention for the lock.

```
Acquire_Lock ( Lock *L )
{
    int m = 1;

    for (;;) {
       while ( *L == LOCKED );  /* spin - do nothing */
       backoff ( uniformInteger ( 0, m-1) );
       if ( *L == UNLOCKED )
            if ( test_and_set( L, LOCKED ) == UNLOCKED ) break;
       m *= 2;
    }
}
```

Figure 2.2. Anderson's lock backoff algorithm [4].

```
Acquire_Lock ( Lock *L )
{
    int m = mMin;

    for (;;) {
       while ( *L == LOCKED );  /* spin - do nothing */
       if ( test_and_set( L, LOCKED ) == UNLOCKED ) break;
       backoff ( uniformInteger( 0, m) );
       if (m <= mMax) m *= 2;
    }
}
```

Figure 2.3. Our lock backoff algorithm.

2.3. Contrasting the two Backoff Algorithms via Simulation

In this section we report our simulation-based performance comparison between our algorithm and Anderson's algorithm. We assume that processors generate lock requests independently of each other, with identical geometricly distributed interarrival times, and that processors hold locks for identical geometricly distributed periods. (We do not present our results for uniformly distributed periods, but they differ little from the results for geometricly distributed periods). Below, we describe our model of the background memory bus traffic, and define the term *critical section load.* When processors are executing parallel programs (e.g., FFT), lock requests may arrive in clumps separated by long periods during which no processor tries to acquire the lock [3]. Anderson called this self-organizing program behavior [4,7]. We consider spin-lock performance to this type of non-independent lock requests in Section 2.4 where we examine barrier synchronization.

We simulated a representative cache coherency protocol (the Sequent Symmetry's [54]) and counted the number of contention and idle cycles generated by *Acquire_Lock* for fifty thousand *Acquire_Lock/Release_Lock* pairs. (Note that an *Acquire_Lock — Release_Lock* pair, in the absence of contention, contributes three contention cycles: one to fault in the lock, one to lock it, and one to release it). It was necessary to model the memory bus cycles generated by processors not spin-waiting. Because this is program dependent, we modeled that any given memory bus cycle is consumed by these other processors with a probability that we call the bus load ν.

2.3.1. Critical Section Load

We define a critical section's load ρ, $0 \le \rho \le 1$, as the expected fraction of time that a processor is executing within it:

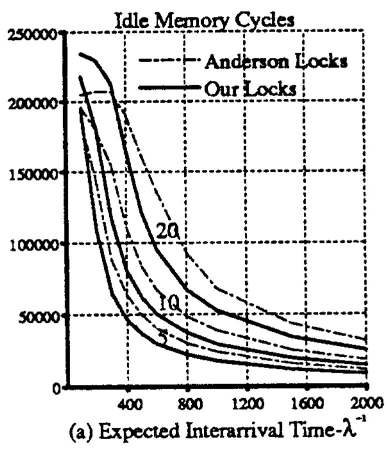
$$\rho = \min\left[\frac{n\mu}{\lambda(1-\nu)}, 1 \right], \tag{2.1}$$

where μ is the critical section's expected duration, n/λ is the net arrival rate of processors at the critical section, and ν is the fraction of shared-memory bus cycles consumed by other processors. When a critical section's load is near one, lock contention is high; backoff values grow large; and the unfairness of exponential backoff [58] may cause some processors to experience starvation. We call a critical section with $\rho = 1$ highly loaded. However, highly loaded critical sections indicate that either the degree of parallelism is too high or that the grain of locking should be made finer. Since highly loaded critical sections limit parallelism, in practice they are recognized as performance bottlenecks and tuned until they are lightly loaded, usually done by splitting locks that guard resources into two or more locks.

We say that a flurry of lock requests followed by a long period of no lock requests is a *transiently* loaded critical section. We will use this term when discussing self-organizing programs in Section 2.4.

2.3.2. Simulation Study

We now report the results of our simulation study. In Figure 2.4 (a) we contrast idle cycles for the two algorithms as a function of the arrival rate of lock requests λ. The three pairs of curves correspond to three different critical section durations. Notice how our algorithm experiences fewer idle cycles when the is lightly loaded. In Figure 2.4 (b) we contrast the algorithms as a function of the critical section's duration μ. Notice how our algorithm suffers fewer idle cycles well past the point where the critical section becomes highly loaded at duration 56. We suffer more contention cycles above duration 40, but at high loads, it is desirable to decrease the number of idle cycles at the expense of contention cycles because limited parallelism has already decreased the shared-bus load. In Figure 2.5 (a) we see that moderate shared-memory bus loads $\nu < 0.75$ do not affect the relative performance of the two algorithms, and, in Figure 2.5 (b) we examine the dependence on the number of processors that share the critical section when we hold the net arrival rate constant. Our algorithm performs slightly better as the number of processors increases.



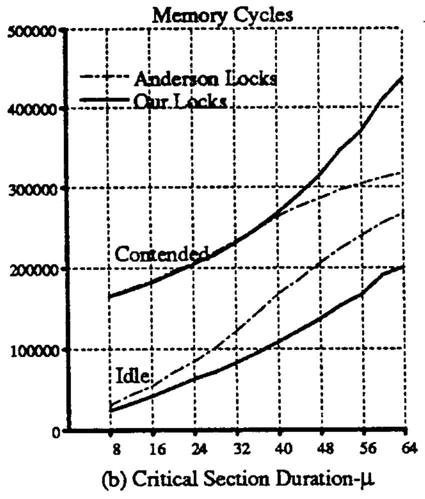
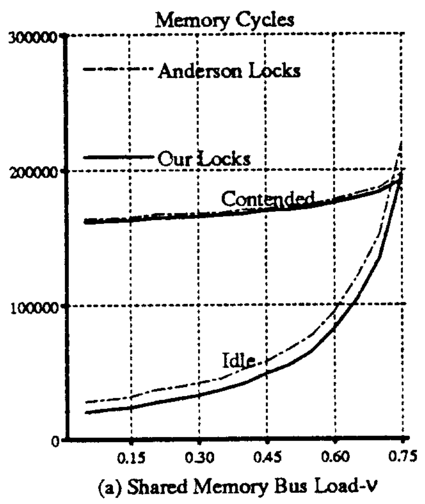


Figure 2.4. (a) Contrasting idle cycles between our algorithm and Anderson's algorithm as a function of interarrival time λ^{-1} (n = 16 processors, bus load $\nu = 0.10$). Critical section duration μ of 5, 10, and 20 cause the critical section to become heavily loaded at interarrival times of 89, 178, and 356. (b) Contrasting idle and contention cycles as a function of critical section duration μ. At duration $\mu = 56$, the critical section becomes heavily loaded (n = 16 processors). Expected interarrival time $\lambda^{-1} = 1000$ (bus load $\nu = 0.10$).



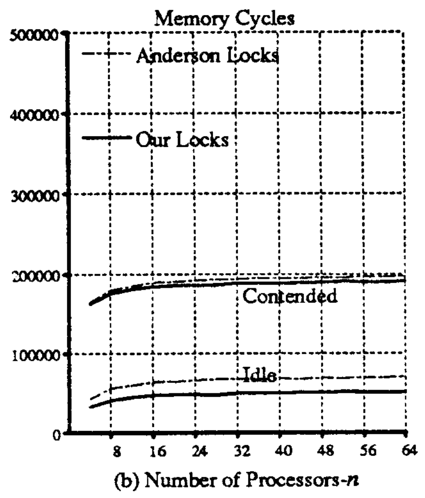


Figure 2.5. (a) Contrasting idle and contention cycles as a function of shared-bus load ν (n = 16 processors, critical section duration $\mu = 5$, expected interarrival time $\lambda^{-1} = 800$). (b) Contrasting idle and contention cycles as a function of the number of processors sharing the critical section n (net interarrival time $n\lambda^{-1} = 100$, bus load $\nu = 0.10$, critical section duration $\mu = 5$, critical section load $\rho = 0.22$).

In general, our algorithm suffers fewer contention and idle cycles than Anderson's algorithm for lightly loaded critical section. While the difference in contention cycles does not warrant much attention, the difference in idle cycles is worth exploiting.

2.3.3. Explanation

For lightly loaded critical sections, few processors simultaneously attempt to acquire the lock. Think of the critical section as the service center of a lightly loaded, limited customer queuing system (with a distributed queue). The expected number of customers enqueued is small. We suffer fewer idle cycles because we backoff only after the test-and-set fails. Experiencing fewer idle cycles reduces the effective duration of the critical section, which decreases the critical section's load. Since contention cycles decrease with critical section load, we experience fewer contention cycles. Also, since our algorithm eliminates the second test performed by Anderson's algorithm, for lightly loaded critical sections, we save a handful of contention cycles. For highly loaded critical sections, Anderson's second test reduces the intensity of the invalidation storm, therefore suffering fewer contention cycles. If one must operate under high loads, increasing the minimum backoff period *mMin* eliminates unfairness and reduces both contention and idle cycles.

2.4. Barrier Synchronization

In this section we propose a barrier synchronization algorithm [62] based on spin-locks, contrast its performance with the algorithm to which [4] alludes, and use it to evaluate our spin-lock backoff algorithm's behavior to transiently loaded critical sections (as defined in Section 2.3). We begin by describing a spin-locked based barrier synchronization algorithm.

The purpose of a barrier is to synchronize some number of processors, n, such that they exit the barrier in unison. The algorithm in Figure 2.6 implements barrier synchronization. It employs spin-lock L for synchronization to the barrier variables. Barrier variable BV is set when the n processors can leave the barrier. Barrier variable Cnt counts the number of processors that need to enter the barrier before the n processors can be released. Barrier variable *race* prevents recently released processors from slipping back into the barrier before all n processors have left the barrier.

The n processors try to exit the barrier simultaneously when the *while* statement is satisfied. They compete with one another to acquire the barrier lock and create an invalidation storm. Since the spin-lock backoff range grows each time a processor acquires the lock, the last few processors through the barrier may experience a large number of idle cycles before they are released.

Our barrier synchronization algorithm consists of the original algorithm (Figure 2.6) plus the three commented-out lines. Each processor grabs a numbered *ticket* when it enters the barrier, and, when the barrier variable B is set, waits a backoff interval proportional to its ticket number, *ticket*DELAY*, before executing *Acquire_Lock*. This disperses the flurry of lock requests at the second critical section.

The second critical section's execution time depends on the shared-memory bus load ν and the critical section's duration μ. The compiler, with minimal effort, should set *DELAY* to the expected duration of the critical section, $\mu/(1-\nu)$. This serializes the lock requests of processors leaving the barrier and avoids an invalidation storm.

Conceivably, a lock holder could be descheduled within the critical section by the termination of a time slice or an interrupt, defeating our scheme. However, in this event, our algorithm performs no worse than the original.

2.4.1. Performance Analysis

We simulated one thousand iterations of our barrier synchronization algorithm with the same model of shared-bus background load used in Section 2.3. We plot the number of idle cycles,

```
Barrier( )
{
  static Lock *L;  /* to synchronize barrier variables */
  static BV = 0;   /* barrier gate */
  static Cnt = n;  /* number of outstanding processors */
  static race = 0; /* avoid race condition */
  /*int ticket;*/  /* processors assigned exiting position */

  while (race) spin(); /* wait until all processors
                              through previous barrier */
  Acquire_Lock(L);
      --Cnt;
      /** ticket = Cnt; **/
      if (Cnt == 0) {
         race = 1;     /* prevents race condition */
         Cnt = BV = n;  /* release processors */
      }
  Release_Lock(L);

  while (!BV) spin();   /* wait for all processors */

  /* nop( ticket * DELAY ); */

  Acquire_Lock(L);
     if ( --BV == 0 ) race = 0;  /* re-enable barrier */
  Release_Lock(L);
}
```

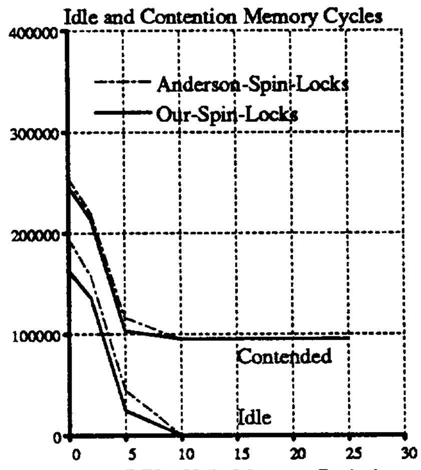
Figure 2.6. Implementing barrier synchronization out of spin-locks. Procedure *nop* (j) makes the processor wait j cycles. Enabling the commented-out lines generates our algorithm.

contended cycles, and the total time through an $n = 16$ processor barrier in Figure 2.7. The original algorithm is equivalent to $DELAY = 0$, which defeats the attempt to serialize the exiting processors. Notice the three-fold reduction in the time to pass through the barrier when $DELAY = 5.5$ (Figure 2.7 (b)). This performance improvement increases with the number of processors participating in the barrier, because the invalidation storm's intensity increases with the number of processors.

2.4.2. Transiently Loaded Critical Sections

In Figure 2.7 we plotted the performance of Anderson's and our spin-lock backoff algorithms to compare performance under high transient critical section load. Transient load is maximal when $DELAY = 0$, and decreases with increasing $DELAY$. We see that Anderson's backoff algorithm suffers more idle and contention cycles, just as it does with lightly loaded critical sections.

Since critical section durations are usually short, on the order of ten or twenty microseconds, critical section loads are ordinarily low. However, if an interrupt caused a lock holder to remain inside the critical section for a much longer time, a long queue of *Acquire_Lock* requests could develop. Since such a long queue resembles the queue of processors exiting a barrier, our spin lock backoff



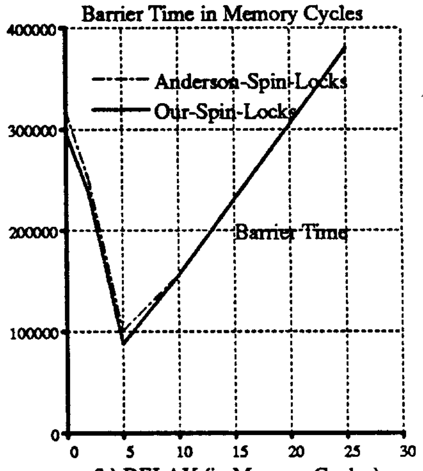


Figure 2.7. n = 16 processors. Bus load $\nu = 0.10$. Critical section duration $\mu = 5$. Our algorithm operates at *DELAY* = 5.5 while the original operates at *DELAY*=0. Note the reduction in idle and contended cycles that our spin-locks provide. (a) Contention and idle cycles for original and modified barrier algorithms. (b) Total number of cycles to complete barrier.

algorithm should still suffer fewer idle and contention cycles.

2.5. Highly Loaded Critical Sections and Performance Metric

In Section 2.3 we argued that highly loaded critical sections are less interesting than lightly loaded ones because they limit parallelism, degrade performance, and, in practice, get split into lightly loaded ones. Although Anderson [4] argues that his algorithm outperforms other backoff algorithms for large numbers of processors, his benchmark is suspect. In his early experiments, processors did nothing but compete for a lock, increment a shared variable, and release the lock. In his later experiments [6], he added idle time, equal to five times the critical section's duration, to this loop. Above five processors, of course, the critical section's load becomes one. Therefore, in all of his experiments, critical sections were highly loaded.

We argue for a diagnostic critical section mode in which each critical section tracks the average number of attempts a processor tries to acquire the lock. If this number grows, then the critical section's load is high and the program requires tuning.

2.6. Conclusions

We have demonstrated that our backoff algorithm eliminates invalidation storms at least as well as Anderson's algorithm and suffers twenty to fifty percent fewer idle cycles for both lightly loaded and transiently loaded critical sections. Our modified barrier algorithm can dramatically reduce the time to pass through a barrier.

More recently, Anderson extended his work to interconnection network-based shared-memory processors [6,7]. However, his performance comparison methodology retains this basic flaw: that algorithms are only compared for highly loaded critical sections. We believe that highly loaded locks occur infrequently in practice, and that we should optimize for the common case. Recently [7], Anderson recommended imposing a backoff limit *mMax* to achieve fairness. Our experience indicates that introducing a minimum backoff value, *mMin*, rather than limiting *mMax*, achieves fairness and reduces contention and idle cycles for heavily loaded critical sections. A hybrid algorithm, one that tests the lock before executing *test_and_set* when the backoff range *m* has grown sufficiently large, would perform well under both low and high loads.

Better support for synchronization such as fetch_and_op instructions and update-based rather than invalidation-based cache consistency make these spin-lock algorithms moot.

Backoff algorithms balance overhead and latency, which is this dissertation's common thread. We will see this in Chapter 3 when we consider multicast retransmission timeouts.

CHAPTER 3

FINITE BUFFERS AND MULTICAST TIMEOUTS

When many or all of the recipients of a multicast respond to the multicast's sender, their responses may overflow the sender's available buffer space. Buffer overflow can be a serious problem of broadcast-based protocols, and can be troublesome when as few as three or four recipients respond. We develop analytical models to calculate the distribution of the number of buffer overflows, which can be used to calculate the number of buffers that an application may need. We develop a backoff algorithm that recipients can use to randomly delay their responses, and consider how to tune its parameters to increase the minimum spacing between responses, reduce CSMA/CD collisions, and decrease the sender's buffering requirements. We define the multicast's latency as the elapsed time to reliably deliver the multicast to all of the recipients, including the time for retransmissions. The sender may need to retransmit if it fails to receive all of the responses before some timeout. Given the number of times it is willing to retransmit the multicast, the sender uses the algorithm to minimize the multicast's latency by selecting the recipient backoff times from a precalculated table. It transmits the backoff value with the multicast, and recipients backoff accordingly. The sender transmits the backoff value because this value is sender specific, load dependent, and changes with each additional retransmission. Recipients do not have sufficient knowledge of all the senders' states to calculate the timeouts. Once we accept that retransmissions are inevitable and explicitly choose to transmit more than once, we may use much shorter backoff intervals and significantly reduce a multicast's total latency. This is due to the backoff interval's dependency on the number of recipients. Permitting buffer overflow on the first transmission significantly reduces the expected duration of the second transmission's backoff interval, and the sum of the first and second backoff intervals is shorter than a much longer single backoff interval. On a purely theoretical note, we show that uniform backoff does not minimize buffer overflow over a given backoff interval.

3.1. Introduction

A multicast is a message broadcast to a selected group of recipients. The term multicast is a euphemism for broadcast implying that only the multicast group members receive the broadcast, although in practice this filtering is often implemented in software. When a recipient site receives a multicast, it formulates and forwards its response to the sender of the multicast. These numerous, closely spaced responses may overflow buffers in the sender's network interface or operating system, or, if the operating system implements protocol processing in the user's address space [18], in the user process that initiated the multicast. We illustrate these overflow points in Figure 3.1. The user process reading the responses may not empty the buffers quickly enough because, for example, it has suffered a page fault, or competing processes have prevented it from receiving adequate processor time.

Since real systems have finite memory, they have finite buffers. Most finite buffer queueing analyses apply only to stationary, homogeneous arrival processes; few analyses of finite buffer systems [32,50] deal with non-stationary, non-renewal, heterogeneous arrival processes. Standard blocking system analyses [42,52] do not apply to this problem because they deal with stationary arrival processes. The arrival process of a multicast's responses is neither stationary nor homogeneous. It is not homogeneous since the responses come from both fast and slow machines; it is non-stationary since the arrival rate changes as responses are received. In this chapter, we calculate the expected value and the distribution of the number of buffer overflows due to responses from multicast transmissions, and we describe a retransmission algorithm that minimizes the time to receive all the responses to a multicast given that the sender is willing to retransmit a specific number of times. In the remainder of this section we define terminology, present several situations for which multicast is appropriate, review related work, and describe our model of the network.

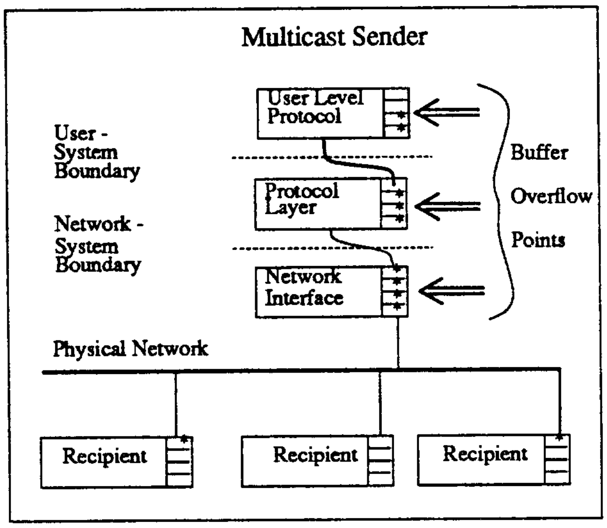


Figure 3.1. Buffer overflow may occur in the sender's network interface, operating system's protocol buffers, or user process protocol buffers.

We say that the *sender* sends a *multicast* to the *recipients* and that each recipient *responds* with its *response*. The elapsed time between the instant the sender transmits the multicast and the instant the recipient transmits its response is the recipient's *response time*. The probability distribution function of the recipient's response time is the *response time distribution*. The probability distribution function of the time devoted by the sender to process a response and free the buffer in which it is stored is the sender's *service time distribution*. The additional time that a recipient holds its already calculated response before sending it to the sender is the recipient's *backoff time*. The number of times the sender retransmits the message is the number of multicast *rounds*. The time window associated with each round during which the sender collects responses is the round's *timeout* (see Figure 3.2). We disregard the possibility that a response arrives after the round's timeout expires. The elapsed time between the sender's initial transmission of the multicast and the end of the last round is the multicast *latency*. We do not measure time in seconds, but treat it as a unitless quantity. However, the service time and response time distributions must be expressed in identical unit systems.

3.1.1. The Uses of Multicast

Multicast is an essential part of reliable, distributed computing. Its effect can be achieved through broadcast primitives implemented by the data-link layer hardware, or through sequential, unicast message transmission. Many common distributed algorithms depend on multicast, or can benefit from it. For example, to update replicated data managed by the available copies algorithm [10], a writer must acquire write locks at all sites which replicate the data, must write the update to all these sites, and must release the acquired locks. All three of these steps benefit from multicast. Distributed transaction processing and transaction-based file systems are an integral part of distributed computing [60,61,64,68]. Replicated files are commonly built atop transaction-based file systems [41,47,49]. The two-phase commit algorithm, used to implement transactions, contains two multicasts [46] (see Figure 3.3). The atomic broadcast algorithms of ISIS require two rounds of multicast [11]. The file name resolution and the load balancing algorithms within the V kernel depend on multicast [17,63]. Most clock synchronization tools depend on multicast [43]. Coordinator election algorithms, run after

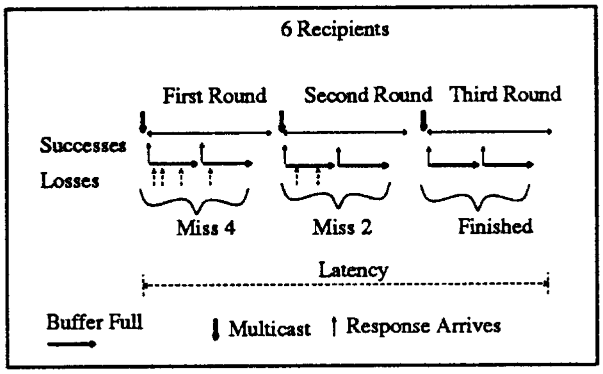


Figure 3.2. An example of a three round multicast from a single buffer sender. Responses that arrive while the buffer is full are lost and must be resent the next round. After the round's timeout expires, the sender retransmits if it has not yet received each recipient's response.

the detection of site failure, employ multicast [37]. Finally, the front-end processors of many database machines employ multicast to send their queries simultaneously to several back-end processors [21].

3.1.2. Model of the Operating System and Network

Our analyses apply to buffer overflow in the network interface and in communication protocol processes. We assume that messages may be corrupted and some recipients may not receive transmissions received at other sites. We assume that recipients transmit equally sized responses to the multicast's sender, and that all recipients respond if only to acknowledge receipt. Through Section 3.3.7 we assume no background traffic shares the buffers that we are studying. When applied to

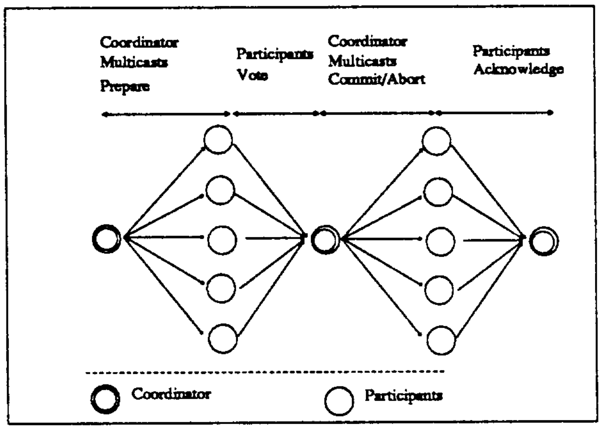

Coordinator Multicasts Prepare
Participants Vote
Coordinator Multicasts Commit/Abort
Participants Acknowledge
Coordinator
Participants

Figure 3.3. Database transactions can use multicast in both phases of the 2-phase commit algorithm. The multicasts must proceed quickly so that resource contention is minimized and database performance does not degrade.

protocol process buffers, we assume overflow does not occur at the network interface. In Section 3.6.2 we model protocol processing buffers and the impact of preemptive process scheduling on buffer overflow.

Through Section 3.3.8 our analyses implicitly assume messages do not collide excessively on CSMA/CD networks and the network possesses infinite bandwidth. These assumptions cause us to overestimate the number of buffer overflows experienced in practice. As discussed in Section 3.3.8, the probability that a CSMA/CD network drops a message due to excessive collisions is extremely low, and the CSMA/CD collision backoff algorithm increases the spacing between messages. The infinite network bandwidth assumption permits responses to arrive closer together than the network's message transmission time permits. Our measurements indicate that these assumptions do not degrade our calculations.

3.1.3. Why Overflow is Important

Speed mismatches at the interfaces of system layers cause buffer overflow. The bit rate of optical fiber networks exceeds the rate at which data can be copied into the memory of most computers and definitely exceeds the rate of computer communication protocol processes. These speed mismatches will always exist. Although various proponents of broadcast-based protocols believe that the problem of buffer overflows is solved in practice, this is not true. Let us review their arguments.

Although many existing network interfaces cannot keep pace with the network, some argue that interface technology is improving and buffer overflow at the interface will not be a problem in the near future. They cite the Ethernet LANCE chip as an example of a successful interface. While true for the slow Ethernet, we argue that, as interface technology advances, network speeds are advancing much faster, and these problems will soon reappear. FDDI [53], a 100 Mb/s ring network, exceeds the memory access speed of most existing computers. Since a multicast's responses come from many computers, and their arrival rate outstrips the rate at which a single computer can send messages, these responses will have to be buffered in the interface; consider, for example, AMD's Supernet FDDI network interface [1]. The cost of high speed buffer memory will limit the buffer size within these interfaces, and buffer overflow at the interface will again be a problem. Historically, interface technology has lagged behind new network technology (see Figure 3.4). Regardless of network bit-rate, the various computers on a network will always have disparate speeds and buffering capacities. Fast computers must communicate with slow computers.

Some argue that the number of acknowledgements can be reduced. The ordered broadcast algorithms of Chang and Maxemchuk [16] eliminate most acknowledgements because recipients do not respond to the sender. The number of acknowledgements is a tunable parameter. However, these algorithms suffer from long latency times, are complex, and do not apply to situations where the recipients must respond by transmitting data to the sender.

Others argue that acknowledgements can be missed. An early version of the V kernel deterministically missed two out of four and one out of three responses [17], though its author, Cheriton, argued that this was not a problem as he needed only the first response. Clearly, not all applications need only the first, or the first few responses.

We must understand the fundamental statistics behind buffer overflow to address the problem adequately, whether it occurs in network interfaces as reported by Cheriton, or in the operating system's protocol processing buffers. For example, the 4.3 BSD UNIX implementation of DARPA's TCP/IP [26] protocol devotes, by default, only four kilobytes to protocol buffer space [14], and such a small buffer can easily overflow. Throwing memory at the problem wastes resources and may not be possible in small systems found, for example, on factory floors.

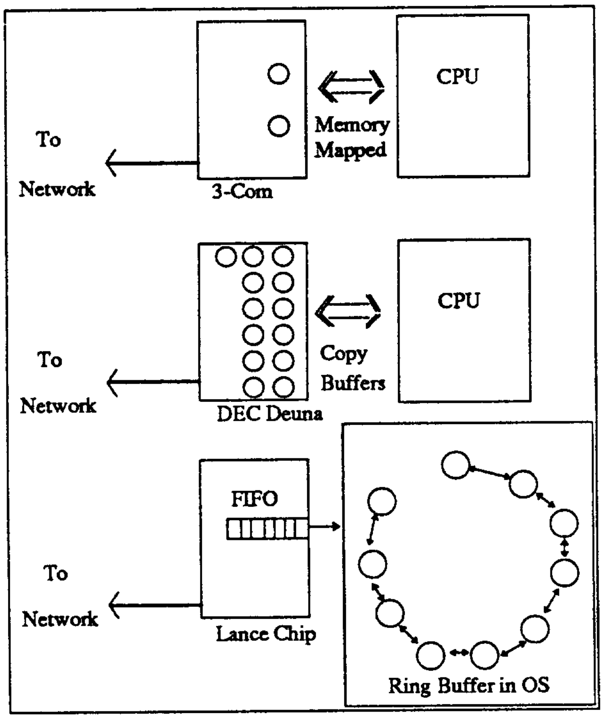


Figure 3.4. The evolution of the Ethernet interface took place over a decade. This figure represents three generations of interfaces. The original interfaces, like the 3-Com interface depicted here, had one or two receive-buffers. The next generation, like the DEC Deuna interface, had many buffers and was suitable for remote file systems. Finally, interfaces like the AMD Lance chip emerged that moves the bytes through a FIFO directly into computer memory.

3.1.4. Contrast to Previous Work

Our problem resembles the automatic repeat-request (ARQ) retransmission algorithms studied exhaustively in the communications literature [45,65,67]. This literature contrasts stream flow control algorithms based on Go-Back N and selective-repeat retransmission schemes. For example, the TCP/IP window based flow control protocol [26] is a Go-Back N scheme. Upon failing to receive an acknowledgement, a sender retransmits the N blocks subsequent to the last acknowledged block. The NETBLT protocol [19], a selective repeat protocol, only retransmits unacknowledged blocks. Our finite buffer, multicast retransmission algorithm differs sufficiently from the ARQ problem that it requires its own analysis.

Although many distributed systems employ broadcast and multicast, researchers have lent little attention to "backoff" algorithms. Cheriton [17] employed uniform backoff to eliminate buffer overflow at network interfaces, but did not study the underlying statistics. We analyze his problem, and develop an algorithm for selecting the backoff interval. We minimize the expected time to collect all responses to a multicast, explicitly introducing the number of times the sender may retransmit the multicast. The system designer chooses the number of broadcast-based retransmissions that the system must endure. We say "endure" because additional broadcasts extraneously interrupt recipients whose responses were not lost to overflow. Knowing the number of rounds, the number of buffers, and the sender's service time distribution, we derive each round's optimal timeout. Recipients choose their backoff functions so that, when added to their natural response time distribution, the effect is uniformly distributed. The sender transmits a bit-map of successfully acknowledged sites with each retransmission so that recipients can discard duplicates. We have implemented this process in a prototype system.

3.1.5. Multicast by Successive Unicast

When the network lacks physical broadcast, we can obtain the effect of broadcast by sending point-to-point, unicast, messages to each recipient. At times we may choose successive unicast over multicast because we do not wish to interrupt some of the group members. For example, dead-site detection usually proceeds by successive unicast (see Figure 3.5). A common misconception is that multicast addressing permits messages to be sent to a subset of the multicast group members. Transmitting to a subset of the multicast group would require either changing the group membership or creating a new group, operations that would require transmitting messages to each site[1]. Buffer overflow is not a problem with successive unicast when it involves few recipients. We discuss it no further.

3.1.6. Chapter's Outline

In the next section we consider overflow of single buffer systems, usually called back-to-back message loss. This illustrates the statistical methods we will use to analyze overflow of multiple buffer systems. In Section 3.3 we derive expressions for the exact distribution of overflows for several multiple buffer multicast systems, and derive an important approximation for uniformly distributed response times. We also consider background traffic, finite network bandwidth, and extensions to non-reliable multicast systems. In Section 3.4, we discuss our recipient backoff algorithms. In Section 3.5 we show how to find optimal round timeouts, and apply the concept to several example problems. In Section 3.6 we present our experimental validation of these ideas. In Section 3.7 we show that uniform backoff is not the optimal common arrival distribution that minimizes the expected number of buffer overflows during a round of multicast. For two recipients, exponential buffer service time, and a one buffer system, we derive the exact optimal response time distribution that minimizes the expected number of overflows for a given round timeout. Finally, in Section 3.8 we review our findings and outline avenues for future research.

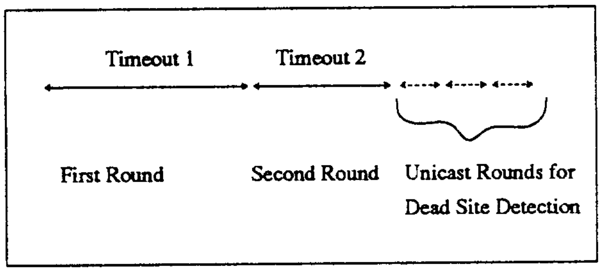


Figure 3.5. The rounds of broadcast-based multicast are occasionally followed by several rounds of unicast to detect failed sites.

[1]An interesting idea would be dynamic multicast groups where the network could permit a handful of destination addresses, filled in at transmission time. Would the performance gains outweigh the hardware costs?

3.2. Buffer Overflow of Single Buffer Systems

The distribution of the number of buffer overflows depends upon the recipients' response time and the sender's buffer service time distributions. We restrict our analysis to single buffer systems in this section because they are easier to analyze than multiple buffer systems, and because we can express their expected number of buffer overflows in closed form. We devote this section to analyzing multicast systems with independent, identically distributed (*i.i.d.*) response time distributions. Single buffer systems suffer overflow if the interarrival time between subsequent responses, the back-to-back inter-message time, is less than the buffer service time. Borrowing the notation of queueing theory, we denote a multicast to n recipients, that respond with *common response time distribution* R, to a b-buffer server with service time distribution S, as an $R^n/S/1/b$ system. In contrast, we denote a multicast to n recipients, that respond with *interarrival time distribution* I, as an $I_n/S/1/b$ system. This distinction is very important to understand. By common response time distribution, we mean that every recipient independently draws a response time from the same distribution. For these systems, the interarrival times between responses depends on the system's previous behavior. To illustrate this point, consider the common uniform response time distribution on $\overline{0, \tau}$, and suppose that the fourth of nine responses arrives at time $\tau/3$. The interarrival time distribution between the fourth and fifth responses is then

$$1 - \left[\frac{x}{(2\tau/3)} \right]^5 .$$

Specified interarrival time distributions systems, $I_n/S/1/b$, can occur when recipients transmit their responses in order. For example, a deterministic interarrival time distribution system results when we specify that recipient i transmit its response at time $i\ \Delta$.

We analyze constant buffer service time systems here, deferring the analysis of general buffer service time distributions to Section 3.3, where we discuss multiple buffer systems. Constant buffer service time accurately models network interfaces that buffer entire messages, because the time to copy a message from the network interface into memory is the sum of the interrupt service latency and the time for the DMA transfer. For equally sized responses, this time is nearly constant. (This argument does not hold for interfaces that do not buffer entire messages but buffer a few bytes at a time in a hardware FIFO [48].)

3.2.1. Upper Bound for M^n/D/1/1 Systems

An $M^n/D/1/1$ system results when n recipients independently draw their response times from a common, exponential distribution. Thus, each recipient i responds at instant y_i independently of other recipients, where y_i is drawn from the exponential distribution with mean $1/\lambda$:

$$Pr(y_i \le t) = 1 - e^{-\lambda t}.$$

We employ order statistics [52] to calculate the expected number of buffer overflows. Briefly, the order statistics $y_{(1)}, y_{(2)}, \ldots, y_{(n)}$ of n *i.i.d.* random variables $y_1, y_2, \ldots, y_n$ are the y_i's sorted in increasing order:

$$y_{(1)} < y_{(2)} < \cdots < y_{(n)}.$$

Recall that the minimum of exponentially distributed random variables is itself exponentially distributed with rate equal to the sum of the individual rates. Immediately after the multicast, but before any recipient responds, the responses arrive at rate $n\lambda$. After the first response, this drops to $(n-1)\lambda$, and each subsequent response decreases the future arrival rate by λ.

If the initial expected interresponse time, $1/n\lambda$, is not significantly larger than the constant buffer service time β, responses arrive closely spaced and overflow the buffer. The expected time to receive all n responses is the expected value of the slowest of the n independent response instants, the harmonic series

$$E[y_{(n)}] = \lambda^{-1}(1+2^{-1}+...+n^{-1}),$$

the maximum of n *i.i.d.*, exponentially distributed random variables.

Since responses require buffer service time β, the buffer overflows if any interarrival interval is shorter than β. The probability $F_0(n)$ that no two responses arrive within time β, that is the probability that no response overflows, is the product of the probabilities that all $n-1$ interarrival times exceed β. Since $y_{(i+1)}$ arrives with rate $(n-i)\lambda$, the probability that the interval between $y_{(i)}$ and $y_{(i+1)}$ exceeds the service time β is

$$Pr[y_{(i+1)} - y_{(i)} > \beta] = e^{-(n-i)\lambda\beta}. \tag{3.1}$$

Probability $F_0(n)$ is the product of the probabilities that each interarrival interval exceeds β:

$$F_0(n) = \prod_{j=1}^{n-1} e^{-\beta j \lambda} = e^{-\beta\lambda n(n-1)/2}. \tag{3.2}$$

We discover that the mean response time $1/\lambda$ must grow quadratically with n to prevent many overflows.

The expected number of interarrival intervals shorter than β bounds the expected number of buffer overflows, L_n. We employ the method of indicator variables to derive a convenient upper bound.

Let the indicator variable $I_i(\beta)$ be

$$I_i(\beta) = \begin{cases} 1 & \text{if } (y_{(i+1)} - y_{(i)}) \le \beta, \\ 0 & \textit{otherwise.} \end{cases} \tag{3.3}$$

The expected number of overflows, L_n, is bounded above by the expected number of short intervals:

$$L_n \le E[\sum_{i=1}^{n-1} I_i(\beta)] = \sum_{i=1}^{n-1} E[I_i(\beta)] = \sum_{i=1}^{n-1} Pr[y_{(i+1)} - y_{(i)} \le \beta]$$

$$\le \sum_{j=1}^{n-1} (1 - e^{-j\lambda\beta}) = n - 1 - \frac{e^{-\beta\lambda} - e^{-\beta n\lambda}}{1 - e^{-\beta\lambda}}. \tag{3.4}$$

Note that $Pr\ [(y_{(i+1)} - y_{(i)}) \le \beta]$ is given by (3.1). The upper bound in (3.4) grows tighter as the number of overflows decreases. Keep in mind that the exponential distribution's tail extends to infinity, and that the sender may treat as losses responses that arrive after the round's timeout expires.

3.2.2. Exact Distribution for M^n/D/1/1 Systems

We can calculate the exact number of overflows for the $M^n/D/1/1$ system, foreshadowing the techniques we will use in the next section. We start by finding the transition probabilities $p_{i,j}$, where $p_{i,j}$ indicates the probability that j of i responses arrive and are lost to buffer overflow during the current service interval. Because each response arrives independently of the others, these probabilities are distributed binomially:

$$p_{i,j} = \binom{i}{j} e^{-(i-j)\lambda\beta} \left[1 - e^{-\lambda\beta}\right]^{j} . \tag{3.5}$$

Denote the expected number of overflows by L_n when n responses remain outstanding and the buffer is unoccupied. During the interval β following the first arrival, during which the server is busy, more responses could arrive and be discarded. Summing over the number of responses i that are lost during the service interval, we arrive at a recursive expression for L_n:

$$L_n = \sum_{i=0}^{n-1} p_{n-1,i} \left[i + L_{n-i-1}\right] , \textit{and } L_0 = 0 .$$

We can also calculate $F_l(n)$, the distribution of overflows, for the $M^n/D/1/1$ system. (In Section 3.3.4 we calculate $F_l(n)$ for the $U^n/M/1/b$ system.) Denote by $F_l(n)$ the probability of losing l responses given n outstanding responses and no responses in service:

$$F_l(n) = \sum_{i=0}^{n-1} p_{n-1,i} \left[F_{l-i}(n-i-1)\right] , \textit{and } F_1(0) = 1 .$$

We plot this distribution in Figure 3.6 for a particular service time and arrival rate. Notice its similarity to the binomial distribution. The expected number of losses is so high because this is a one buffer system (check (3.4)).

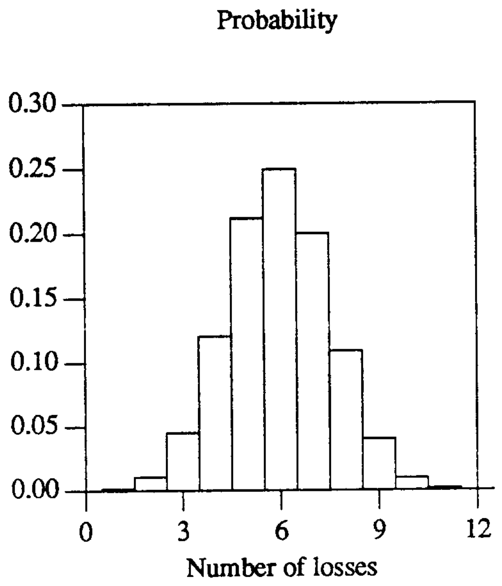


Figure 3.6. Notice how the distribution of the number of overflows suffered by the $M^{20}/D/1/1$ system appears binomial. This distribution corresponds to $\lambda\ \beta = .05$.

3.2.3. Upper Bound for U^n/D/1/1 Systems

In this case, each recipient i responds at instant y_i drawn independently from the uniform distribution on the interval $\overline{0,\tau}$. The expected time to receive all n responses is the expected value of the n^{th} order statistic $y_{(n)}$:

$$E[y_{(n)}] = \frac{n\,\tau}{n+1}.$$

The probability of no buffer overflow, $F_0(n,\tau)$, equals the probability that all $n-1$ interarrival intervals exceed β, and can be found by solving the recurrence relation[2]

$$F_0(n,\tau) = \begin{cases} \dfrac{n}{\tau} \displaystyle\int_{(n-1)\beta}^{\tau} F_0(n-1, x-\beta) \left[\frac{x-\beta}{\tau}\right]^{n-1} dx & \text{if } \beta < \tau/(n-1) \\ 0 & \textit{otherwise} \end{cases}$$

The variable of integration corresponds to τ — the arrival time of the last response. The probability that the remaining $n-1$ arrive no closer to the first than β accounts for the quantity raised to the $n-1$. From induction, the solution to this recurrence is

$$F_0(n,\tau) = \left[1 - \left[n-1\right]\frac{\beta}{\tau}\right]^n, \quad \beta < \tau/(n-1). \tag{3.6}$$

Employing indicator variable $I_i(\beta)$, defined in (3.3), we can bound the expected number of buffer overflows by the expected number of interarrival intervals shorter than β:

$$L_n \le E[\sum_{i=1}^{n-1} I_i(\beta)] = \sum_{i=1}^{n-1} Pr[\,y_{(i+1)} - y_{(i)} \le \beta\,]. \tag{3.7}$$

We exploit a trick based on the symmetry of a circle to evaluate these probabilities. Consider placing $n+1$ points uniformly on a circle of circumference τ. Cut the circle at an arbitrary point and unroll it into a line segment of length τ. This leaves n points on the line segment. The probability that any two of these points are separated by β is simply $(1-\beta/\tau)^n$. The probability that two points arrive more closely together than β follows immediately:

$$Pr[\,y_{(i+1)} - y_{(i)} \le \beta\,] = 1 - \left[1 - \frac{\beta}{\tau}\right]^n. \tag{3.8}$$

Substituting (3.8) into (3.7) and adding up terms, we find that the expected number of closely spaced responses, an upper bound on the number of overflows, is

$$L_n \le (n-1)\left[1 - \left[1 - \frac{\beta}{\tau}\right]^n\right]. \tag{3.9}$$

We calculate the exact, but extremely tedious, solution to the $U^n/D/1/b$ system in Section 3.3.5. In Figure 3.7, we contrast this upper bound with the exact solution and with the expected number of buffer overflows for the $U^5/M/1/1$ system. Notice the apparent paradox that deterministic service is lossier than exponential service with identical first moments. This occurs because there is a nonzero probability that the exponential service times are shorter than the the deterministic service times.

[2] This is an application of de Finetti's theorem (see [27], problem 24, Section I.13).

Under deterministic service, when the service time reaches one, only the first response is processed; the remaining four responses are lost. However, under exponential service, when the mean service time is one, the probability that the actual service time is less than one is $1 - e^{-1}$.

3.2.4. Summary of Single Buffer Systems

Let us interpret these expressions in terms of back-to-back message losses where n recipients respond to a multicast and each response is processed in constant time β. If the common response time is uniformly distributed on $\overline{0, \tau}$, then from (3.8) we obtain the probability that we lose at least one response to buffer overflow, and (3.9) gives an upper bound on the expected number of overflows. In Section 3.4 we associate the backoff interval and the round timeout with τ.

A few one buffer network interfaces exist, but these are historical remnants rather than the product of parsimonious design. (Note, however, the similarity between contention for single buffers and contention for a shared memory bus as analyzed in Chapter 2). We are now ready to analyze the more interesting multiple buffer systems.

3.3. Overflowing Multiple Buffers

We begin this section on multiple buffers by finding the expected number and distribution of overflows when both the recipient's response time and the sender's buffer service time are exponentially distributed. Next we derive recursive expressions for the expected number and distribution of overflows for general *i.i.d.* response time distribution, exponentially distributed buffer service times, and specialize this to the uniform response time distribution. We investigate a semi-Markov, approximation to uniform recipient response time distribution that we use throughout the rest of this chapter. Lastly, we extend the calculations to include background traffic.

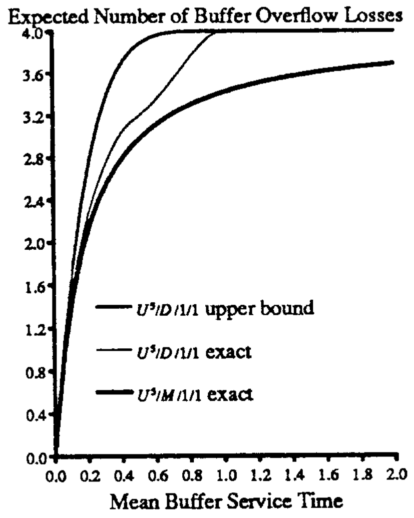


Figure 3.7. Contrasting (3.15), the expected number of overflows for $U^5/M/1/1$, and (3.16), the expected number of overflows for $U^5/D/1/1$ systems. Notice that upper bound (3.9) is nearly tight for less than 1.6 overflows. The five responses arrive uniformly on $\overline{0,1}$.

We devote this section to deriving expressions that predict curves like the ones in Figure 3.8, where we contrast the number of overflows suffered by $M^n/M/1/b$ and $U^n/M/1/b$ systems with rates selected such that the expected n^{th} order statistic $E[y_{(n)}]$ were equal:

$$\frac{n\tau}{n+1} = \frac{(1+\frac{1}{2}+\cdots+\frac{1}{n})}{\lambda}.$$

This operating point means that the expected time for the last recipient to reply is equal for both systems, and is, we believe, the fairest way to compare the systems. It is equivalent to comparing the systems under the same server load. In the figure, notice how the response time distribution significantly affects the expected number of overflows, and how this effect increases with buffer size. We begin with preassigned response times, where the recipients respond sequentially, and proceed to common, *i.i.d.* response times.

3.3.1. Preassigned Response Times, D_n/M/1/b

If we assign a group membership number to each site as it joins the group, and reassign these numbers when a member leaves the group, we can give recipient i a unique time $i\Delta$ at which it should respond to the multicast sender. Theoretically, the interarrival times should be a constant Δ, but, realistically, deadline scheduling is impossible at the necessary time scale, and we do not achieve the constant spacing we desire (see Section 3.4, Figure 3.13). (Preassigned response time is roughly equivalent to multicast by successive unicast, because in the latter case the recipients tend to respond in the order that they receive the unicast messages).

Besides requiring maintaining a group list, this scheme can be unduly restrictive. To illustrate this last point, consider the problem of finding a lightly-loaded site for load sharing. Theimer [63] suggests that sites backoff their responses for a time proportional to their loads and not respond if they are unwilling to accept more jobs. Despite these reservations, we now analyze this multiple buffer sys-

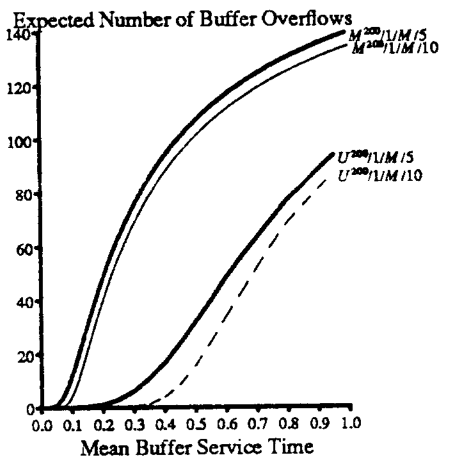


Figure 3.8. Expected buffer overflows for $M^{200}/M/1/5$, $M^{200}/M/1/10$, $U^{200}/M/1/5$, $U^{200}/M/1/10$ systems. Note that we selected the exponential arrival rate λ and the uniform interval τ to make the server's expected load equal.

tem[3] because, at the least, it serves as a lower bound.

We calculate the expected number of overflows suffered by b-buffer systems by observing that an overflow occurs if all b buffers are occupied when a recipient responds. Let $L_n(s)$ be the expected number of overflows when n recipients respond, s buffers are occupied, and the first of the n responses arrives immediately:

$$L_n(s) = \sum_{i=0}^{s+1} L_{n-1}(s+1-i)\, p_{s+1,i}(\Delta)\,,$$

$$L_n(b) = 1 + \sum_{i=0}^{b} L_{n-1}(b-i)\, p_{b,i}(\Delta)\,,$$

$$L_0(s) = 0.$$

$$p_{s,i}(x) = \begin{cases} (\mu x)^i e^{-\mu x}/i! & i<s, \\ 1 - \sum_{k=0}^{s-1} p_{s,k}(x) & i=s. \end{cases} \tag{3.10}$$

In these expressions, we sum over the number of buffers i that the server empties during the time interval Δ before the next arrival. When the next arrival occurs, $s+1-i$ buffers are full: $s+1$ for the number of full buffers at the start of the interval, and i for the number of buffers served during the interval. The probability that i, $i < s+1$, buffers are emptied is the probability that i Poisson events occur during interval Δ. The probability that all $s+1$ buffers are emptied is not the probability that $s+1$ Poisson events occur, but the probability that at least s Poisson events occur.

We can similarly calculate the distribution of losses, F_l.

$$F_l\,(n,s) = \sum_{i=0}^{s+1} F_l(n-1, s+1-i)\, p_{s+1,i}(\Delta)\,,$$

$$F_l\,(n,b) = \sum_{i=0}^{b} F_l(n-1, b-i)\, p_{b,i}(\Delta)\,,$$

$$F_l\,(0,s) = 1.$$

We should note that it is difficult to extend this calculation to general service time distributions [55], but since the Erlang distribution E_k approaches the constant distribution for large k, we can approximate the deterministic arrivals, general service case as closely as we like with an embedded Markov chain. We consider this further in Section 3.6.7.

3.3.2. M^n/M/1/b Systems

It is easy to find the number of buffer overflows given both exponential response and service time distributions (and general service time distribution as well). Consider a two-dimensional Markov chain, where state (i,j) means i responses are outstanding and j buffers are full (see Figure 3.9).

[3] From a renewal and indicator variable argument, a single buffer, exponentially distributed server with service rate μ experiences $\sum_{j=2}^{n} e^{-\Delta\mu} = (n-1)\,e^{-\Delta\mu}$ overflows. This is the expected number of buffer service time intervals longer than Δ.

When $b = 1$, we can write the expression for the expected number of buffer overflows by inspection:

$$L_n = \sum_{j=1}^{n-1} \frac{j\lambda}{\mu+j\lambda} .$$

When $b > 1$, we can write a recursive expression to count the expected number of buffer overflows.

$$L_n(s) = \left[p_{n,s}\right] L_{n-1}(s+1) + \left[1 - p_{n,s}\right] L_n(s-1) , \tag{3.11}$$

$$L_n(b) = \left[p_{n,b}\right] \left[1 + L_{n-1}(b)\right] + \left[1 - p_{n,b}\right] L_n(b-1) ,$$

$$L_0(s) = 0 .$$

$$p_{n,s} = \frac{n\lambda}{n\lambda + u(s-1)\mu} .$$

Note that $u(x)$ is the unit step function: one if $x \geq 0$ and zero otherwise. Of course, the distribution of overflows can be calculated similarly:

$$F_l(n, s) = \left[p_{n,s}\right] F_l(n-1, s+1) + \left[1 - p_{n,s}\right] F_l(n, s-1) ,$$

$$F_l(n, b) = \left[p_{n,b}\right] F_l(n-1, b) + \left[1 - p_{n,s}\right] F_l(n, b-1) ,$$

$$F_0(0, s) = 1.$$

3.3.3. G^n/M/1/b Systems

This calculation is reminiscent of the $G/M/1/b$ loss system analysis of queueing theory. Let $L_n\ (s, t)$ be the expected number of overflows given that s of the b buffers start full, the last arrival occurred at time t, and n additional arrivals remain outstanding. $L_n\ (s, t)$ is an integral over the arrival time, x, of the next interarrival time. Since the arrival times are drawn from a common response time distribution, we can easily express the probability density function, f_n, of the next arrival.

Since the remaining n responses remain independent, their arrival time distribution function conditioned on the arrival time t of the last arrival, $F(y \mid y > t)$, is

$$Pr(y \leq x \mid y > t) = \frac{P(y \leq x)}{P(y > t)} = \frac{F(x)}{1 - F(t)} .$$

Similarly:

$$Pr(y > x \mid y > t) = \frac{P(y > x)}{P(y > t)_.} = \frac{1 - F(x)}{1 - F(t)} .$$

The probability density function of the next arrival time is then the product of this density and the probability that the remaining $n-1$ responses arrive later, all multiplied by n, the number of ways to select the first arrival:

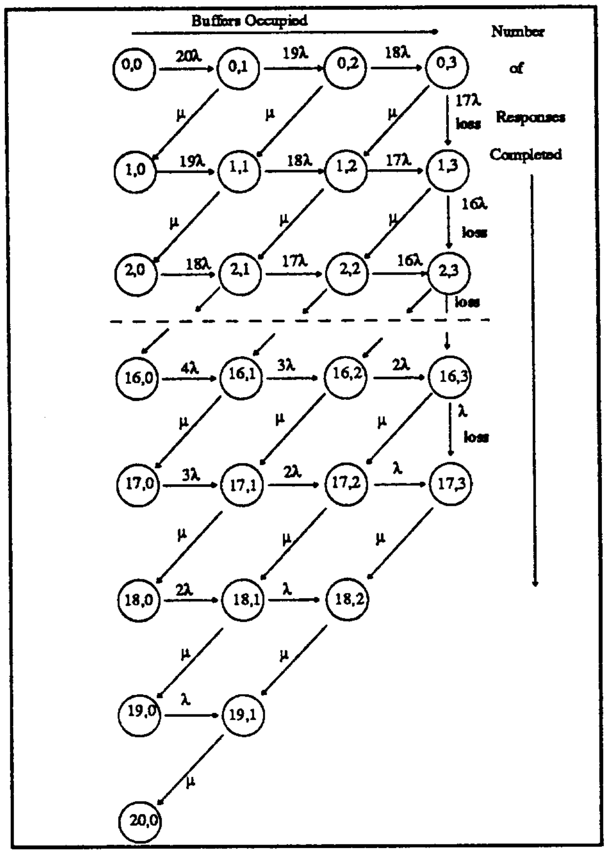


Figure 3.9. The Markov chain that corresponds to the $M^{20}/M/1/3$ system. Whenever the chain takes a transition labeled "loss", a buffer overflow occurs. This counts as a response. A round begins in state (0,0), and ends when the chain reaches state (20,0).

$$f_n(x) = n \left[\frac{1-F(x)}{1-F(t)} \right]^{n-1} \frac{f(x)}{1-F(t)} .$$

We insert this into the recursive integral expression of $L_n(s,t)$:

$$L_n(s,t) = \int_t^{\infty} f_n(x) \sum_{i=0}^{s} \left[L_{n-1}(s+1-i, x)\, p_{s+1,i}(x-t) \right] dx, \tag{3.12}$$

$$L_n(b,t) = \int_t^{\infty} \left[f_n(x) \sum_{i=1}^{b} \left[L_{n-1}(b+1-i, x)\, p_{b,i}(x-t) \right] + \right.$$

$$\left. \left[(L_{n-1}(b,x) + 1)\, p_{b,0} \right] \right] dx,$$

$$L_0(s,t) = 0.$$

We integrate over the next response's arrival time x, summing over i, the number of buffers serviced through time x. This leaves $n-1$ outstanding responses and $s+1-i$ buffers occupied. This number of buffers are occupied because s buffers were previously occupied, one buffer is occupied by the new arrival, and i buffers were freed during interarrival time x.

When all b buffers are full, an overflow occurs with probability $p_{b,0}$, the probability that no buffers are emptied before the next response arrives. Although formidable in appearance, we can integrate expression (3.12) for many response time distributions because the service time is exponential. For example, we now apply (3.12) to the case of uniformly distributed response times.

3.3.4. U^n/M/1/b Systems

It is easier to calculate the solution to (3.12) for uniformly distributed response times than for any other response time distribution. Exploiting the fact that the arrival time distribution of the remaining responses is uniform when it is conditioned on knowing the arrival time of the most recent response, we redefine $L_n(s,t)$ to be the expected number of overflows when n responses remain outstanding, given that s buffers start full and the remaining responses arrive uniformly on $\overline{0,t}$.

$$L_n(s,t) = \frac{n}{t} \int_0^t \left[\left[\frac{t-x}{t} \right]^{n-1} \sum_{i=0}^{s} L_{n-1}(s-i+1, t-x)\, p_{s,i}(x) \right] dx, \tag{3.13}$$

$$L_n(b,t) = \frac{n}{t} \int_0^t \left[\left[\frac{t-x}{t} \right]^{n-1} \sum_{i=1}^{b} L_{n-1}(b-i+1, t-x)\, p_{b,i}(x) \right.$$

$$\left. + \left[1 + L_{n-1}(b, t-x)\, p_{b,0}(x) \right] \right] dx,$$

$$L_0(s,t) = 0.$$

We can similarly calculate the exact distribution of overflows:

$$F_l(n,s,t) = \frac{n}{t}\int_0^t \left[\left[\frac{t-x}{t}\right]^{n-1} \sum_{i=0}^{s} F_l(n-1, s+1-i, t-x)\, p_{s,i}(x) \right] dx\ , \tag{3.14}$$

$$F_l(n,b,t) = \frac{n}{t}\int_0^t \left[\left[\frac{t-x}{t}\right]^{n-1} \sum_{i=1}^{b} F_l(n-1, b-i+1, t-x)\, p_{b,i}(x) \right.$$

$$\left. + F_{l-1}(n-1, b, t-x)\, p_{b,0}(x) \right] dx\ ,$$

$$F_0(0,s,t) = 1.$$

Although (3.13) does not have closed form solutions for arbitrary numbers of buffers, we can always integrate it in polynomial time. For example, the expected number of overflows experienced by the $U^4/M/1/b$ system on $\overline{0,1}$ are

$$b{=}1:\ 12\left[(\mu^{-1}-3\mu^{-2}+6\mu^{-3}-6\mu^{-4}) + e^{-\mu}\,(6\mu^{-4}) \right].$$

$$b{=}2:\ 12\left[(2\mu^{-2}-6\mu^{-3}+4\mu^{-4}) + e^{-\mu}\,((3\mu)^{-1}+2\mu^{-2}+2\mu^{-3}-4\mu^{-4}) \right].$$

$$b{=}3:\ 12\left[(2\mu^{-3}-6\mu^{-4}) + e^{-\mu}\,(\mu^{-2}+4\mu^{-3}+6\mu^{-4}) \right].$$

$$b{=}4:\ 0.$$

Equation (3.13) has closed form solutions for two systems. The single buffer system on $\overline{0,\tau}$ has solution

$$n!(n-1)\left\{ (\mu\tau)^{-1} \sum_{i=0}^{n} \frac{(\mu\tau)^{-i}\left[-1\right]^{i}}{(n-1-i)!} + (-1)^n\, e^{-\mu\tau}\,(\mu\tau)^{-n} \right\}. \tag{3.15}$$

The $n-1$ buffer system on $\overline{0,\tau}$ has solution

$$n!(n-1)\left\{ (n-1)(n-2)\,(\mu\tau)^{-(n-1)} - (n-1)!\,(\mu\tau)^{-n} \right.$$

$$\left. + (\mu\tau)^{-1}\, e^{-\mu\tau} \sum_{i=1}^{n-1} (\mu\tau)^{-i}\, i!\, a_{n-1-i,n-1} \right\},$$

where

$$a_{i,j} = \begin{cases} a_{i-1,j} + a_{i,j-1} & \text{if } i\ ,j \ge 1, \\ 1 & \textit{otherwise.} \end{cases}$$

Although we observe and are confident that it can be proved that the uniform response time distribution suffers fewer overflows than the exponential distribution with identical $E[y_n]$, it does not minimize buffer overflow. The *i.i.d.* response time distribution that minimizes the number of buffer overflows depends on the service distribution, the number of buffers, the number of responses, and the number of buffers that start full. We consider this further in Section 3.7.

3.3.5. U^n/D/1/b Systems

We include this section to demonstrate that exact analysis of these problems is tricky. Denote the number of overflows that this system suffers by $L_n(\tau, s)$, where s indicates the number of *empty* buffers, and service has not yet begun.

$$L_n(\tau, s) = \int_0^{(\tau-\beta)_+} \left\{ \sum_{i=0}^{s-1} P_i(x)\, L_{n-1-i}(\tau-x-\beta, s-i) + \sum_{i=s}^{n-1} P_i(x) \left[L_{n-1-i}(\tau-x-\beta, 1) + (i+1-s) \right] \right\} dx + (n-s) \left[\frac{\beta}{\tau} \right]^n . \tag{3.16}$$

where

$$P_i(x)\,dx = \left[\frac{n!}{(n-1-i)!\, i!\, 1!} \right] \left[\frac{dx}{\tau} \right] \left[\frac{\beta}{\tau} \right]^i \left[\frac{\tau-x-\beta}{\tau} \right]^{n-1-i} .$$

Note that $(x)_+ = max(x, 0)$. The variable of integration x represents the first response's arrival time. Probability $P_i(x)dx$ corresponds to the event that the first response arrives at time x, i more responses arrive during deterministic service time β, and $n-i-1$ remaining responses arrive after service completes. This can be computed using an $O[(\frac{\tau}{\beta})^4]$ algorithm given by Ott and Shanthikumar [57].

The term outside of the integral is a necessary endpoint condition, accounting for overflows that occur when all n responses arrive within β of τ. The first sum inside the integral accounts for responses that arrive and occupy available buffers during the service time β. The second sum accounts for responses that overflow the buffer.

Although this recursive computation is exact, it is difficult to evaluate. Approximating the constant service time with an Erlang distribution, or the uniform arrival time distribution with an exponential interarrival time distribution works well. We consider this second approximation next.

3.3.6. M$_n$/G/1/b System with Exceptional First Service

When the interarrival time between the recipients' responses is exponentially distributed we can efficiently calculate both the expected number and the distribution of overflows. Recall that the notation M_n indicates that the responses' interarrival times are exponentially distributed. We have seen that calculating the $U^n/G/1/b$ system's expected number or distribution of overflows requires solving (3.12), a difficult system of equations. Fortunately, the $M_n/G/1/b$ system behaves remarkably like the $U^n/G/1/b$ system. We now introduce the key approximation that we will use from now on. We can approximate the $U^n/G/1/b$ system's distribution and expected number of overflows with the $M_n/G/1/b$ system's distribution and expected number of overflows. We must simply set the arrival rate equal to n/τ.

Denote the sender's normal buffer service time distribution by $G_1(x)$ and its first service time distribution by $G_o(x)$. We calculate $L_n(s)$, the expected number of overflows given that n recipients respond with a Poisson interarrival time distribution, that s buffers are occupied, and that service has just begun.

$$L_n(s) = \sum_{i=0}^{b-s} L_{n-i}\,(s-1+i)\,p_{n,i} + \sum_{b-s+1}^{n} p_{n,i}\left[L_{n-i}(b-1)+i-(b-s)\right], \tag{3.17}$$

$$L_n(0) = \sum_{i=0}^{b-1} L_{n-i-1}\,(i)\,q_{n-1,i} + \sum_{b}^{n-1} q_{n-1,i}\left[L_{n-i-1}(b-1)+i-(b-1)\right],$$

$$L_n(s) = 0, \qquad \text{if } n+s \le b\ .$$

$$p_{n,i} = \int_0^{\infty} \frac{\left[\lambda x\right]^i e^{-\lambda x}}{i!}\, d_1\, G(x), \qquad q_{n,i} = \int_0^{\infty} \frac{\left[\lambda x\right]^i e^{-\lambda x}}{i!}\, dG_o\,(x), \tag{3.18}$$

$$p_{n,n} = 1 - \sum_{i=0}^{n-1} p_{n,i}, \qquad q_{n,n} = 1 - \sum_{i=0}^{n-1} q_{n,i}\ . \tag{3.19}$$

The distribution of overflows follows immediately:

$$F_l(n,s) = \sum_{i=0}^{b-s} p_{n,i}\, F_l(n-i, s+i-1) + \sum_{i=b-s+1}^{n} p_{n,i} F_{l-(i-(b-s))}(n-i, b-1), \tag{3.20}$$

$$F_l(n,0) = \sum_{i=0}^{b-1} q_{n-1,i}\, F_l(n-1-i, i) + \sum_{i=b}^{n-1} q_{n-1,i} F_{l-(i-(b-1))}(n-1-i, b-1),$$

$$F_0(0,0) = 1\ .$$

In practice (and we believe it can be shown analytically), the approximation's expected value is slightly high; its distribution is slightly skewed towards the higher number of buffer overflows. The differences, for our purposes, are immaterial. We contrast the approximation, (3.17), with the exact solution, (3.16), for the $U^{20}/D/1/3$ system in Figure 3.10. (Note that we don't need all of this machinery to approximate the $U^n/M/1/b$ system; a simple modification to (3.11) suffices.)

3.3.7. Including Poisson Background Traffic

We now address the effect of background traffic incident to the same buffers that the multicast's responses occupy. Background traffic is unavoidable when considering overflow at network interfaces. We continue to assume that the network possesses infinite bandwidth, and to denote the buffer service time distribution as G. Since characterizing the background traffic's interarrival time distribution is a significant feat in itself, and as we have already made several approximations, we model it as Poisson with rate γ. We define the buffer occupancy distribution at the time the multicast is sent as B_k. Although this is a quantity easily measured in practice, here we model B_k as the equilibrium queue length distribution of the $M/G/1/b$ queue [42] composed of the Poisson background traffic and general service time distribution with mean μ.

3.3.7.1. U^n/M/1/b Systems

When the buffer service time distribution is Poisson, we denote the load of the background traffic by $\rho = \gamma/\mu$, and calculate the buffer occupancy distribution as distribution of an $M/M/1/b$ queue:

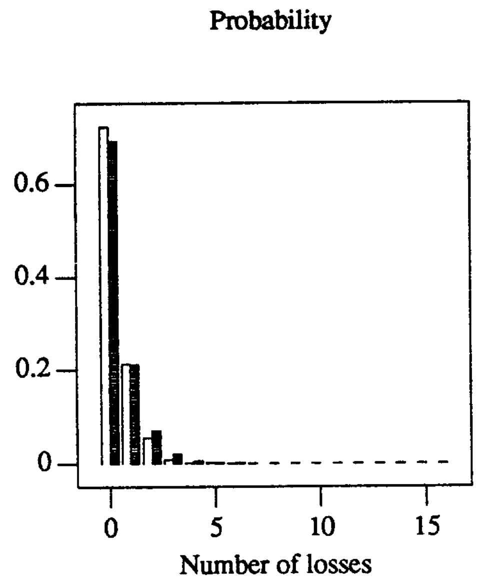


Figure 3.10. Probability distribution, $M_{20}/D/1/3$, (3.16), arrival rate λ=1, expected service time μ=.025. The approximation, (3.17), is shaded. In this example the first service and normal service distributions are identical.

$$B_k = \frac{1-\rho}{1-\rho^{b+1}}\rho^k .$$

The multicast's n responses arrive over some interval $\overline{0,\tau}$. Since the background traffic is Poisson with rate γ, we can express the probability Γ_i that i background messages arrive during interval τ:

$$\Gamma_i = (\gamma\tau)^i \, e^{-\gamma\tau}/i\,!.$$

We condition the calculation of the number of overflows on knowing i, the number of background messages that arrive with the n responses over interval τ. Having conditioned the i Poisson events to occur over an interval τ, from elementary statistics, we know these events are uniformly distributed on $\overline{0,\tau}$. We multiply $L_{n+i}(k,\tau)$ (3.13), the total number of overflows (responses plus background), by $n/(n+i)$, the fraction of the total arrivals that are responses.

$$L_n = \sum_{k=0}^{b} B_k \sum_{i=0}^{\infty} \Gamma_i(\tau)\, L_{n+i}\;(k,\tau) \left[\frac{n}{i+n}\right], \tag{3.21}$$

Under practical loads ($\rho \le 0.5$), the Γ_i approach zero quickly as i grows. (In practice, finite network bandwidth and minimum network packet size limit the number of background arrivals.)

3.3.7.2. M$_n$/G/1/b Systems

It is easy to add background traffic to our $M_n/G/1/b$, exceptional first service approximation to the $U^n/G/1/b$ system on $\overline{0,\tau}$. To do so, substitute (3.21) for (3.13), and calculate B_k from the $M/G/1/b$ queue composed by the Poisson background traffic and general server.

3.3.7.3. Distribution of Overflows with Background Traffic

We can calculate the distribution of overflows given background traffic. Recall the hypergeometric distribution describes the distribution of choosing r balls such that l of them are red from a set of n red balls and i white balls:

$$\frac{\binom{n}{l}\binom{i}{r-l}}{\binom{n+i}{r}}.$$

Denote the multicast's n responses by red balls and the i background messages by white balls. The probability that of l of r overflows are recipient responses is exactly the hypergeometric probability just given. Employing (3.20) for the total distribution of overflows, then the distribution of overflows that correspond to recipient responses is:

$$F_l = \sum_{k=0}^{b} B_k \sum_{i=0}^{\infty} \Gamma_i(\tau) \sum_{r=l}^{n+i} F_r\,(n+i,k)\,\frac{\binom{n}{l}\binom{i}{r-l}}{\binom{n+i}{r}}.$$

Let it suffice to say, we have taken these analyses to their practical limits.

3.3.8. Including Finite Network Bandwidth

Recall our other assumption, that the network possesses infinite bandwidth. This is equivalent to assuming instantaneous, collisionless, message transmission. In this section we relax this assumption by solving the tandem system illustrated in Figure 3.11. We approximate the expected number and distribution of overflows for the simplest, Markov case. More sophisticated models of the network are best evaluated through simulation. Since CSMA/CD collisions are resolved in microseconds and waste negligible network bandwidth, we will continue to ignore them here and focus on message transmission time and competing network background traffic.

Other network traffic competes with the responses for exclusive access to the network. A multicast recipient may have to hold its response until the network goes idle. We model this as variations in

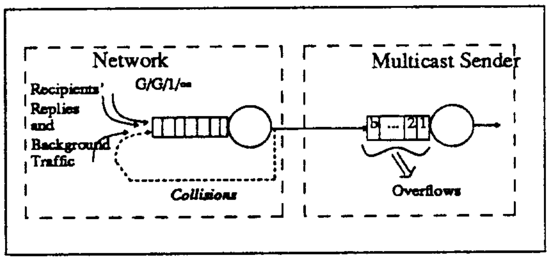


Figure 3.11. Accurately modeling the network's bit-rate is difficult, as is modeling how collisions affect response interarrival times. Here we attempt to model the network's characteristics with a tandem queue.

the responses' transmission time. Given that the network carries background traffic some fraction ρ of the time and that responses have average transmission time x, we model message transmission time as exponentially distributed with rate

$$\lambda = \frac{1-\rho}{x}.$$

Let the n responses arrive with rate $\gamma = n/\tau$ and the buffers be serviced at rate μ. The state space (n, r, s), indicates that n responses have yet to enqueue at the network, r responses are enqueued at the network, and s responses occupy buffers at the multicast sender. The expected number of overflows L_n is:

$$L_n(r,s) = \left[\frac{(1-u_{s,b})\,\lambda L_n(r-1,s+1)}{u(r)\,\lambda + u(s)\,\mu + u(n)\,\gamma}\right] + \left[\frac{u_{s,b}\,\lambda\left[L_n(r-1,b)+1\right]}{u(r)\,\lambda + \mu + u(n)\,\gamma}\right] +$$

$$\left[\frac{u(s)\,\mu L_n(r,s-1)}{u(r)\,\lambda + \mu + u(n)\,\gamma}\right] + \left[\frac{u(n)\,\gamma L_{n-1}(r+1,s)}{u(r)\,\lambda + u(s)\,\mu + \gamma}\right].$$

$$L_0(0,s) = 0.$$

Note that $u(x)$ is the unit step function: one if $x \geq 0$ and zero otherwise; and $u_{s,b}$ is the Kronecker delta function: one if $s=b$ and zero otherwise. We compute the derivation of the distribution of overflows similarly.

$$F_l(n,r,s) = \left[\frac{(1-u_{s,b})\,\lambda F_l(n,r-1,s+1)}{u(r)\,\lambda + u(s)\,\mu + u(n)\,\gamma}\right] + \left[\frac{u_{s,b}\,\lambda F_{l-1}(n,r-1,b)}{u(r)\,\lambda + \mu + u(n)\,\gamma}\right] +$$

$$\left[\frac{u(s)\,\mu F_l(n,r,s-1)}{u(r)\,\lambda + \mu + u(n)\,\gamma}\right] + \left[\frac{u(n)\,\gamma F_l(n-1,r+1,s)}{u(r)\,\lambda + u(s)\,\mu + \gamma}\right].$$

$$F_0(0,0,s) = 1.$$

3.3.9. Conclusions

Throughout this section we derived analytic expressions for the distribution of buffer overflows which we will use in subsequent sections. Some of these expressions were easily evaluated, while others were not. The remainder of this chapter is much more experimental in nature.

3.4. Two Recipient Backoff Algorithms

Frequently a multicast's recipients transmit their responses to the sender nearly in unison, overflowing the sender's buffers. In this section, we consider two recipient backoff algorithms that significantly reduce buffer overflow. The first algorithm attempts to transform each recipient's response time distribution onto the uniform[4] distribution on $\overline{0,\tau}$. The second algorithm preassigns a response time on $\overline{0,\tau}$ to each recipient. We first present measurements of the response time distribution taken from several computer systems without backoff, then discuss the two backoff methods, and finally present measurements of the response time distribution with backoff. In Section 3.5 we identify

[4] In Section 3.7 we show that the uniform distribution does not quite minimize the expected number of overflows on $\overline{0,\tau}$.

τ as the round's timeout and consider how to select it.

Recall our definition of the recipients' response time in Section 3.1. We assume that each recipient i can measure its response time y_i and schedule its response with clock granularity Δ milliseconds. We implemented both algorithm on a network of Sun workstations. We will see that, because existing workstations tend to have coarse clock granularity, the deterministic backoff algorithm does not attain its intended effect unless the backoff interval is large.

3.4.1. Measuring the Response Time Distribution

Even when similarly configured recipients perform identical calculations, each recipient's response time distribution is unique, depending on its workload. The distribution approaches a constant when the node is lightly loaded, but its tail can grow quite long when the node is heavily loaded. We illustrate this with representative measurements of the response time distribution of several UNIX computers (see Figure 3.12 (a)). We should note that a given machine's response times are essentially *i.i.d.* and uncorrelated. We plot their autocorrelation function in Figure 3.12 (b). We see that even one lag away (see Figure 3.12 (b)). We do note that the distribution drifts with load; compare ucbernie's response time distribution when its load is less than three (the line labeled ernie.lt3) with its response time distribution when its load is greater than three (the line labeled ernie.gt3).

The response time distribution is sensitive to changes in the system load. We plot *ucbernie's* response time distribution when the load average was under three and over three separately. Node *snow* was always lightly loaded, so its response time distribution was nearly a constant. When the response times of many machines are nearly constant, their responses arrive closely spaced, and buffer overflow is likely.

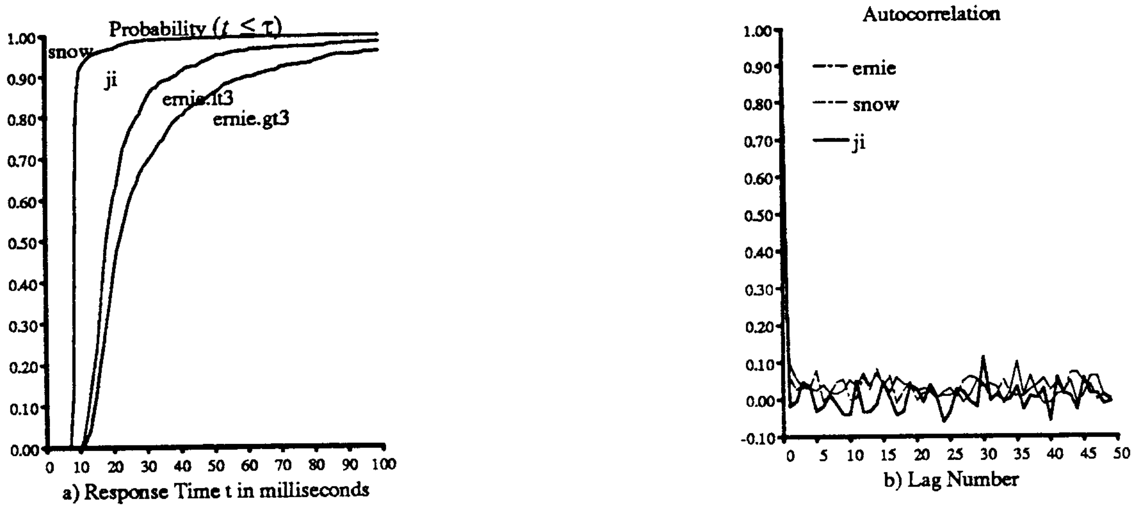


Figure 3.12. (a) Recipient response time distributions of several machines sending 592 byte responses. Note how ucbernie's response time depends on its processor load. Lines ernie.gt3 and ernie.lt3 corresponds to service times when the load is greater than 3 and less than 3. Snow is a lightly loaded Sun file server. Ucbernie and Ji are moderately loaded VAXes. (b) Autocorrelation function of the response time distributions.

3.4.2. Predetermined Backoff

The obvious backoff algorithm is to divide the backoff interval into $n-1$ equally spaced intervals and assign response times accordingly. This is easily done when we assign each recipient its group membership number $i \in 0,..,n-1$. We assign response time $\tau\, i/(n-1)$ to member i. Member i, after receiving a multicast and computing its response at time y_i, waits an additional time Y_i:

$$Y_i = maximum\ (\frac{i}{n}\tau - y_i\ , 0\ ). \tag{3.22}$$

We implemented this backoff strategy and measured the interarrival times it generates. Notice that the interarrival times are not constant, but closer to exponential[5] (see Figure 3.13). We computed the standard deviation σ and found it to be 81% of the mean ($\bar{x} = 8.94$, $\sigma = 7.23$). This is due to the twenty millisecond clock and scheduling granularity of Sun workstations.

As noted in Section 3.3.1, this scheme requires that each site be assigned a group membership number upon joining a group, and that sites change these numbers when members leave the group.

3.4.3. Uniform Backoff

We now describe a backoff algorithm that transforms a recipient's response time distribution into the uniform distribution on $\overline{0, \tau}$. We cannot do this perfectly because, on occasion, a site's response time may exceed τ, and it is never less than some minimum. Also, coarse timing and scheduling granularity Δ distorts our transformation.

We assume that each site has measured its response time distribution and placed it in an array H_k:

$$H_k = Pr\left[y_i \le k\,\Delta\right] .$$

We exploit our observation that response times are nearly *i.i.d.* as follows. Each recipient measures its response time before backoff. This falls into bucket k if its value is between $\overline{(k-1)\,\Delta,\ k\,\Delta}$. The fraction $H_k - H_{k-1}$ of response times fall into bucket k. We compute backoff time $Y_i(k)$ so that, after backoff, responses cover this same fraction of the uniform interval $\overline{0, \tau}$:

$$Y_i(k) = uniform\ \left[\tau\, H_{k-1} - (k-1)\Delta, \tau\, H_k - k\,\Delta\right] . \tag{3.23}$$

We note that, if the recipients' are unloaded, then their response times fall into a single bucket, and our uniform backoff algorithm is equivalent to always adding a uniform $\overline{0, \tau}$ backoff value.

3.4.4. Conclusions

In Figure 3.13 we plot the interarrival time distributions as collected from 13 Sun 3/50 recipients with predetermined and uniform backoff on $\overline{0, 100mS}$. Most of the workstations were idle during the experiment. Despite our attempt to preorder the responses so that the interarrival times were constant, the workstations' coarse clock granularity randomizes them. The conclusion that we draw from this experiment is that predetermined backoff does not lead to constant interarrival times (although we will see later that it does suffer fewer buffer overflows than uniform backoff).

Now that we know how to make recipients arrive uniformly over an interval $\overline{0, \tau}$, we consider next how to best choose τ.

[5]What is the variance of this scheme's interarrival times? We would like to know the variance value to model the interarrival time with an Erlang distribution.

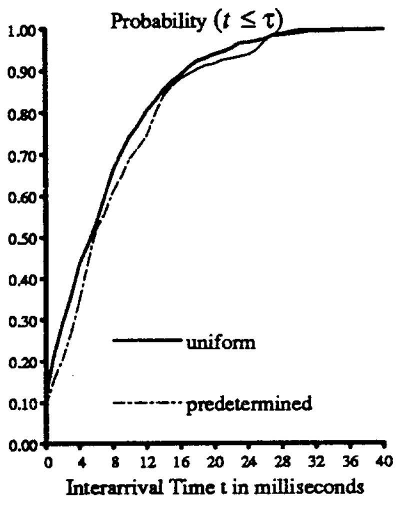


Figure 3.13. Interarrival time distribution of thirteen Sun 3/50 recipients after backoff onto $\overline{0, 100mS}$ transmitting 592 byte responses. The Sun 3/50's scheduling resolution is 20 milliseconds.

3.5. Multiple Round Multicasts and Multicast Timeouts

When communicating by multicast, computers with slow network interfaces, slow CPU's, and limited memory may suffer buffer overflow. Because sites usually consist of older and newer machines and lower-end and upper-end machines, we need a multicast protocol whose parameters each machine can tune. In this section we describe a multiple round multicast algorithm, and consider criteria for selecting the round timeouts. Each sender chooses its timeouts to minimize the latency, the expected time to reliably deliver the multicast to all the recipients, given that it must meet a certain constraint with probability Φ. Each sender must also choose the number of rounds. The multicast's latency, the time between the initial transmission and the end of the last round, drops with each additional round and approaches the expected time to service n responses. The computational overhead imposed by the extra interrupts and retransmissions increases with each additional round. A sender seeks to limit overhead or to achieve a given latency, and selects the number of rounds accordingly. Soft real-time systems are systems that meet a deadline with some probability. When this probability is one, the system is called a hard real-time system. Ours is a soft real-time algorithm. With probability Φ the sender meets its constraint by the end of the last round, and with probability $1 - \Phi$ it fails to meet it. In practice, senders usually select one or two rounds, and, on occasion, three rounds. Each multicast sender selects the current round's timeout from its own, precalculated, timeout table. We precalculate the timeout table from the sender's measured service time distribution to meet the real-time constraint with the desired degree of confidence. The sender transmits the timeout in the message header, and the recipients use it as their uniform backoff interval.

In this section we consider the algorithm's details, describe how to calculate the timeout table, briefly consider how to set timeouts for dead-site detection, and conclude by extending the algorithm to where we only know the distribution rather than the exact number of recipients that respond.

3.5.1. Discarding Duplicates

Given that the multicast and responses may be transmitted more than once, we must take measures so that recipients can discard duplicate multicasts. Consider the message format:

Serial Number	Sender-Id	Timeout	Count	Bit-map	Data

The message's bit-map field indicates which group members should reply to this transmission. Using a bit-map rather than a list requires that the group members agree on their group membership numbers. Before transmitting a round, the sender adjusts the bit-map. If the sender does not receive all the required responses after the last round, it enters the protocol's dead-site detection phase which we discuss later in this section. As an optimization, the sender need only transmit the message's data field during the first round. If a recipient receives a subsequent round but not the initial round, it can request that the sender retransmit the data. As a further optimization, if few recipients remain outstanding, the sender can transmit to these via successive unicast rather than broadcast, eliminating extraneous interrupts at the other recipients.

3.5.2. The Timeout Table

Whether we choose the number of rounds to meet a real-time deadline or to limit the overhead, we must calculate the timeout table's entries. The table's columns correspond to rounds; the table's rows correspond to the number of unacknowledged recipients. We compute the last round's entries to meet the sender's constraint with probability Φ. The constraint that we use is that with probability Φ, after the last round, we have received the responses from all sites, which we express in terms of F_0, the probability we experience no overflows:

$$F_0(n, 0, \tau) \geq \Phi . \tag{3.24}$$

In Figure 3.14 we see a two-round timeout table calculated for a multicast sender with five buffers and exponentially distributed buffer service time. The table's last round timeout is computed by solving (3.24), where F_0 comes from (3.13) and constraint Φ is set to 0.99. We will discuss how to calculate the next-to-last round's timeout shortly (see (3.25)). The table's fourth column lists the expected two-round latency, which is the sum of the first round's latency and the expected value of the second round's latency.

3.5.3. Selecting Multiple Round Timeouts

Selecting multiple round timeouts is a standard optimization problem. On one hand, if we set the timeouts too short we suffer lots of overflows. On the other hand, if we set them too long, we needlessly increase the multicast's latency. Below, we demonstrate how to select good multiple round timeouts, and apply the procedure to several examples. We call the last round's timeout τ_1, the next-to-last round's timeout τ_2, ..., and the first round of a k-round multicast τ_k.

We must satisfy the constraint during the last round with probability Φ. Therefore, regardless of how we arrived to the last round with a certain number of outstanding responses, we choose the last round's timeout solely on the number of outstanding responses. To minimize the latency, we select the next-to-last round's timeouts so as to

$$\textit{minimize } E\left\{ \tau_2 + \tau_1 \right\}, \tag{3.25}$$

where the last round timeouts τ_1 are fixed, chosen to satisfy the constraint. This problem reduces to selecting the optimal next-to-last round timeouts τ_2 so as to:

$$\textit{minimize} \left\{ \tau_2(n) + \sum_{k=0}^{n} F_k(n, s, \tau_2(n))\, \tau_1(k) \right\}, \tag{3.26}$$

Number of Recipients	Next-to-last Round Timeout τ_2	Last Round Timeout τ_1	Expected Latency
6	0	7.7	0.0
7	0	12.0	0.0
8	0	16.8	0.0
9	0	21.0	0.0
10	0	25.0	0.0
11	3.3	29.4	3.8
12	3.7	34.3	5.1
13	4.8	38.5	6.8
14	6.2	44.2	8.5
15	7.5	49.0	10.1
16	8.8	53.2	11.7
17	10.1	58.4	13.2
.	.	.	.
.	.	.	.
97	115.0	562.5	137.7
98	116.2	570.4	139.3
99	117.4	577.3	140.9
100	118.6	584.2	142.6

Figure 3.14. This two-round timeout table was calculated for a 5-buffer, exponential, unit mean server and is good for up to 100 recipients. The table is calculated so that with probability ninety-nine percent responses do not overflow the second round.

where $F_k(n, s, \tau_2)$ is the probability that k responses overflow, e.g. (3.20). Intuitively, a three-round multicast consists of an initial round followed by an optimal two-round multicast. Whatever the number of overflows experienced during this initial round, we must still satisfy the constraint as quickly as possible in the remaining two rounds, which is the problem we just finished solving. We calculate τ_3 given the solution to (3.26), the expected latency of a two-round multicast, just as we calculated τ_2 from τ_1. We proceed like this until all of the timeout table's columns are complete.

Except in the simplest systems where we can employ calculus to find the solutions to (3.24) and (3.26), ordinarily we find their solution by searching.

3.5.3.1. Multiple-Round Multicast, $U^{100}/M/1/5$

We compute this system's distribution of overflows, from either the exact solution, (3.14), or from the approximation discussed in Section 3.3.6. We set the arbitrary degree of confidence to ninety-nine percent. The table is computed for unit expected service time. The lower the degree of confidence, the smaller the timeouts. In Figure 3.15 we plot the necessary one-round latency, τ_1, that satisfies constraint (3.25), and the minimum two-round and three-round latencies. We have already presented this system's two-round timeout table in Figure 3.14. The latency does not significantly drop beyond three rounds, but approaches the expected minimum: the expected service time of n responses.

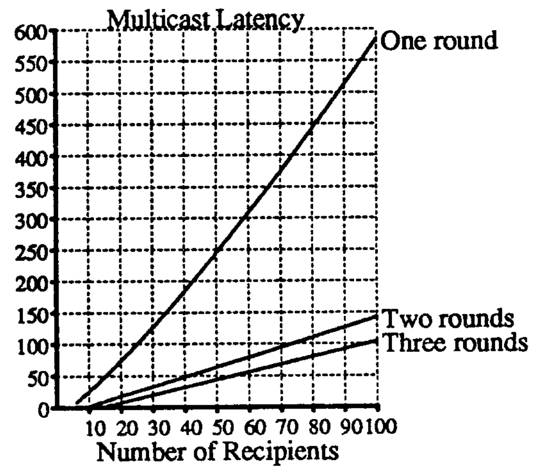


Figure 3.15. Minimum latency one-round, two-round, and three-round multicasts, $U^n/M/1/5$, $\Phi = .99$, $\mu = 1$. The common uniform arrival distribution is achieved through backoff.

3.5.4. Collecting a Subset of Replies

A related problem, practically solved already, is collecting a given fraction or number of responses. This could occur, for example, while searching for an idle machine. Suppose we want to collect m responses from n recipients. We simply substitute constraint equation (3.24) with

$$\sum_{k=0}^{n-m} F_k\ (n, s, \tau_2) \geq \Phi,$$

and proceed as before.

3.5.5. Dead-Site Detection

The multicast's sender may not receive a recipient's response for several reasons: the recipient did not correctly receive the multicast; its response was lost; or it failed before responding. The sender could conclude that any recipient that fails to respond to several unicast transmissions has failed, although this conclusion might be erroneous.

We say a multicast has completed when we have received responses from all functioning recipients. If a recipient fails to respond to several transmissions, the sender assumes that the recipient has failed and removes it from the group membership list. Since site failures are infrequent, dead-site detection timeouts are not terribly interesting. With no knowledge of the response time distribution $H(t)$, these timeouts should be set to a few seconds and be increased with each retransmission. However, when we know the response time distribution, this section shows how we can identify dead sites more quickly.

The timeouts for these rounds depend on the response time distribution's tail. We choose the number η of dead-site detection rounds such that we are satisfied with probability $1 - \zeta$ that the recipient has failed:

$$\prod_{i=1}^{\eta} \left[1 - H(\tau_i) \right] \geq 1 - \zeta\ .$$

For example, consider the response time distributions plotted in Figure 3.12. Given the distribution of response times that we collected, three 200 millisecond dead-site detection rounds would misidentify a working site as failed once in five million times.

3.5.6. Multicast to an Unknown Number of Recipients

Can we apply our method when we do not know the identity or number of recipients that will reply? Yes, but we must take steps to convince ourselves that we have collected responses from every operational site, because we do not know which recipients will respond. We must retransmit the multicast until no more recipients respond to satisfy ourselves that all operational recipients have responded. Then, we must undergo several dead-site detection rounds to assure ourselves, with probability ζ, that all operational sites have responded. We assume that we know P_i, the distribution of the total number of recipients, but not the exact number. We consider two cases below. In the first case, we assume that the operating system and network interface count and return the number of overflows experienced during a round. In the second case, we assume they do not.

Our goal is to minimize the latency of a k-round multicast given that we know the distribution of the number of recipients. After the k primary rounds, we must issue several dead-site detection rounds. Paralleling reliable multicast, we calculate the last-round timeouts from the constraint equation.

We select the optimal next-to-last round timeout such that the sum of the next-to-last timeout and the expected value of the last timeout is minimized:

$$\mathit{minimize} \left\{ \tau_2 + E[\tau_1] \right\} . \tag{3.27}$$

Since the network interface and the operating system tell us how many responses were lost during a round, we know the number of responses we must receive during the last round. Therefore we use the last round's timeout table that we have calculated in (3.24), and we need only optimize over the next-to-last round's timeout. Denoting the probability of losing j responses to buffer overflow by F_j, we must minimize

$$\tau_2 + \sum_{i=1}^{n} P_i \sum_{j=0}^{i} \tau_1(j) F_j(i, s, \tau_2).$$

This expression's only unknown is τ_2 (we know P_i and τ_2 and can calculate F_j). We minimize this expression with respect to τ_2 and construct a timeout table. We ignore the possibility that a recipient might fail to respond on the first round but responds to subsequent rounds, because this occurs infrequently.

When the hardware and operating system do not count the number of overflows, then we do not know how many recipients remain outstanding, and we must calculate the last round's timeouts differently. Define $P_k^{(1)}(j)$ as the distribution of k, the number of recipients that overflowed during the next-to-last round given that j responses were received:

$$P_k^{(1)}(j) = \frac{P_{k+j}}{1 - \sum_{i=1}^{j} P_i} .$$

With this we can express the last round's timeout as a function of the number of successfully received replies during the next-to-last round:

$$\sum_{k=0}^{n-j} P_k^{(1)}(j)\, F_0(k, s, \tau_1(j)) \geq \Phi\,.$$

We can perform a binary search to quickly find the value of $\tau_1(j)$ satisfying this expression. With these timeouts in hand, we reconsider equation (3.27). The probability that we successfully receive j of i responses is identical to the probability of losing $i-j$ responses given i recipients responded:

$$minimize \left\{ \tau_2 + \sum_{i=1}^{n} P_i \sum_{j=0}^{i} \tau_1(j)\, F_{i-j}\,(i, s, \tau_2) \right\}.$$

We operate as we did above, searching for the value of τ_2 that minimizes this expression. These timeout calculations depend on the service time distribution, which we must measure. We investigate this next.

3.6. Implementation, Instrumentation, and Experiments

We implemented our multiple round multicast algorithm and instrumented the UNIX operating systems of several DEC and Sun computers to measure and model real buffer service time distributions, and to explore the accuracy of our calculations. In this section we discuss how to model the buffer service time, present our service time measurements, and we describe our UDP buffer overflow experiments. But first, we review our instrumentation and experimental setup.

3.6.1. UDP Protocol Layer Instrumentation

Our multicast sender runs as a user process. It broadcasts a UDP datagram, and reads and discards the recipients' responses, also UDP datagrams. Since the sender process may compete with other processes for timeslices, DMA transfers may absorb part of the backplane's bandwidth, and device interrupts may preempt the processor, UDP buffer service times are random and workload dependent. Their minimum value depends on the message's size and processor's speed.

What happens when a recipient responds? Once the sender's network interface receives the response, it posts a processor interrupt. When the sender's interrupt priority level is sufficiently low, the sender fields the interrupt, and if space permits enqueues the message on the IP protocol input queue, schedules a software interrupt and returns from interrupt. When it fields the software interrupt, it dequeues the message, passes it to the IP layer which, in turn, passes it to the UDP layer. The UDP layer, if space permits, enqueues the message at the appropriate UDP socket where it can be read by the sender process. Newly arriving responses preempt the sender process, are processed by IP and UDP, and eventually are enqueued at the socket. The sender does not run during this processing. We will need to account for this preemption in our model of the service time. Figure 3.16 illustrates the path responses take through the operating system.

We instrumented the Ethernet device driver to record a time stamp (TS_1) the moment a packet arrival interrupt occurs. We reserved a particular UDP port number for our experiments. For UDP datagrams destined to this experimental port we write TS_1 into one of the message's fields. We instrumented the UDP network code to write a timestamp (TS_2) into another of the message's fields just before the call to the scheduler that indicates that a message had been placed in the UDP socket's buffer and awaits the sender process. Hence the sender process finds these two time stamps in the response that it reads. It determines the response's residence time by subtracting the current time (TS_3) from the datagram's first timestamp (this is the service time if the buffer was previously empty).

The Sun's poor clock resolution hampered our efforts to measure the buffer service time distribution; our Sun 3/50 workstation's clock has twenty millisecond resolution. To accurately calculate the service time distribution we designed, built, and installed a microsecond resolution clock in one workstation. The clock can be read both inside and outside the kernel, and is documented in Chapter

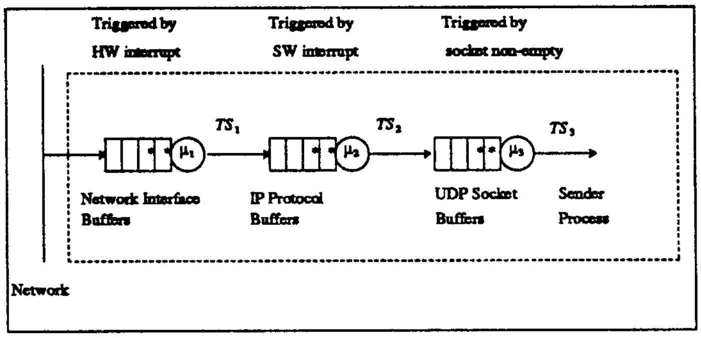


Figure 3.16. The responses flow from queue to queue within the operating system. Server μ_3 is preempted while server μ_1 and μ_2 run. In this section, we are studying the UDP socket buffers.

User Program (TS_3)	Socket Wakeup (TS_2)	Network Interface (TS_1)	Waiting Time (TS_3–TS_1)
83114	79568	78479	4635
81202	74434	73340	7862
69991	64344	60523	9468
68294	63430	59036	9258
66368	62401	57961	8407
52639	47005	45910	6729
48505	44238	43148	5357
29768	21440	20348	9420
28018	18057	15982	12036
26243	17051	14750	11493
22430	14225	12726	9704

Figure 3.17. Microsecond timestamps generated when recipients send a 592 byte response to a multicast. The fourth column is the responses' resident time in system, and is composed of its service time and delay waiting for service to start.

5. We used our clock to gather the data shown in Figure 3.17.

We added a device driver option to the Sun's kernel that sets the amount of buffer space allocated to a UDP socket, a quantity normally compiled into the kernel. We treat this as if it set the number of buffers rather than the number of bytes in the buffer, because each response consumes a fixed number of bytes. For example, if responses are 1,024 bytes and we allocate 4,500 bytes of memory, we say that we allocated four buffers (protocol layering adds about a hundred bytes to responses).

3.6.3. Modeling the OS's Service Time

Let us examine the UDP buffer service time. It consists of the time for the UDP packet to traverse the operating system's networking code, the sender process's scheduling latency and service time, and the service time lost to preemption by high priority peripheral device events. The time to traverse the networking code and sender process depends on the contention for the backplane's finite bandwidth. The scheduling latency depends on the length of the run queue and the processor's scheduling policy. Time lost to preemptive service depends on the peripheral devices and their workload. When the processor is lightly loaded, the scheduling latency and preemption times disappear leaving constant buffer service time. When it is heavily loaded, both scheduling latency and device servicing can cause large variations in the service time. Scheduling latency only affects the first response served during a busy period. Therefore a busy period's first service time is drawn from a different, more widely varying distribution than other service times. We account for this by employing the exceptional first service model[6] introduced in Section 3.3.6.

Since the Ethernet driver runs at hardware interrupt priority and UDP protocol processing runs at software interrupt priority, newly arriving responses preempt the sender process. The sender process resumes only when every message enqueued at the network interface has wound its way through the operating system and into the UDP socket's buffer. One can see this by reviewing the timestamps in Figure 3.17. Notice how the first response does not reach the user program (TS_3) until the first four responses have all gotten to the UDP socket's buffer (TS_2). We must model this queueing process and find an expression for its distribution of overflows given that the n recipients employ uniform backoff. We invoke the Poisson approximation from Section 3.3.6, and treat the interarrival times of the n responses as Poisson with parameter

$$\lambda = n/\tau .$$

We analyze the UDP protocol with (3.17) and (3.20). To do so, we must first find new expressions for the transition probabilities $p_{n,i}$ and $q_{n,i}$ that model the effects of preemption, and model the service time distribution. Recall that $p_{n,i}$ and $q_{n,i}$ are the probabilities that i of n responses arrive during a regular or first service interval respectively. These new arrivals occupy and possibly overflow the UDP socket's buffers. First service times, whose distribution is $G_o(x)$, consist of a constant time D to traverse the networking code plus a hyperexponentially distributed time that accounts for the variations in the service time which are mainly due to processor scheduling. Regular service times, whose distribution is $G_1(x)$, consist of a constant time D and a hyperexponentially distributed time to account for variations in the service time that are mainly due to interrupts and DMA devices [56]. Regular service times exclude the time spent traversing the networking code since we account for this time as a preemption of time Δ to the current service. These distributions are:

$$G(x) = u(x-D)\left[1 - \alpha\, e^{-(x-D)/x_1} - (1-\alpha)\, e^{-(x-D)/x_2}\right] \tag{3.28}$$

where $u(x)$ is the unit step function: one if $x \geq 0$ and zero otherwise. We discuss how we evaluate G_o and G_1 in Section 3.6.6.

How many responses arrive during a service interval? This number depends on the length of the service interval. Since arriving responses extend the service interval's length by time Δ, each subsequent arrival spawns its own M/D/1 busy period during which other responses arrive. Let ν^{bp} be the distribution of the number of arrivals during an M/D/1/ busy period, and let ν be the distribution of the number of arrivals that occur during the service interval, excluding preemption. The distribution that we need to know is the convolution of ν and ν^{bp} (see Figure 3.18).

[6]Alternatively, this is equivalent to exhaustive service with service vacations and preemptions.

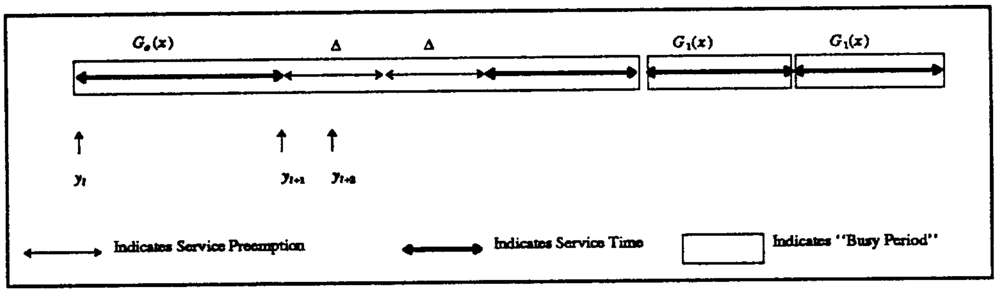


Figure 3.18. Consider what happens when responses y_{l+1} and y_{l+2} arrive during the service interval of response y_l. These two responses each preempt y_l's service interval for time Δ, and, since the remaining responses interarrival times are Poisson, spawn their own M/D/1 busy periods during which more preemptions may occur.

Let $v_{n,i}\,(G)$ be the probability that i of n responses arrive due to a service interval with distribution G:

$$v_{n,i}\,(G) = \sum_{k=0}^{i} \frac{\left[\lambda D\right]^{k} e^{-\lambda D}}{k!} \sum_{j=1}^{2} \left[\alpha_j \left[\frac{\lambda}{1/x_j + \lambda} \right]^{i-k} \left[\frac{1/x_j}{1/x_j + \lambda} \right] \right] .$$

$$v_{n,n}\,(G) = 1 - \sum_{i=0}^{n-1} v_{n,i}\,(G) .$$

Note that we have defined $\alpha = \alpha_1$ and $\alpha_2 = 1 - \alpha$.

Each of these i responses engenders its own M/D/1 busy period during which still more responses may arrive. Recall the distribution of the number served during an M/D/1 busy period. Denote the probability that i responses are served during the busy period by $v_{n,i}^{bp}\,(\Delta)$:

$$v_{n,i}^{bp}\,(\Delta) = \frac{(i\lambda\Delta)^{i-1} e^{-i\lambda\Delta}}{i!}, \tag{3.29}$$

where $i \geq 1$ since a busy period contains at least one job. Let $v^{bp\,(k)}$ be the distribution of the kth convolution of (3.29), and $r_{n,i}$ denote the probability that i of the n outstanding responses arrive during a service interval with distribution G:

$$r_{n,i}(G) = \sum_{v=0}^{i} v_{n,v}\,(G)\; v_{n,i}^{bp\,(v)}\,(\Delta) \quad , \quad r_{n,n}(G) = 1 - \sum_{i=0}^{n-1} r_{n,i}\,(G). \tag{3.30}$$

The transition probabilities we seek are $q_{n,i} = r_{n,i}\,(G_0)$ and $p_{n,i} = r_{n,i}\,(G_1)$, and we can apply (3.20) to compute the distribution of overflows. In the next section we relate our experience with this model's predictions.

3.6.4. Overflow Experiments

How well can we predict the expected number and distribution of overflows with this model of the service time distribution? To investigate this we collected first and normal service time distributions and measured Δ for ucbarpa, ucbernie, monet, and figaro. In Figure 3.19 and Figure 3.20 we

Host	Preemption	First Service G_o				Normal Service G_1			
Name	Δ $[mS]$	D $[mS]$	α	x_1 $[mS]$	x_2 $[mS]$	D $[mS]$	α	x_1 $[mS]$	x_2 $[mS]$
ucbarpa	1.3	3.39	.973	1.29	37.5	2.08	.927	.013	10.4
ucbernie	2.3	4.38	.880	2.25	20.2	2.10	.980	1.25	70.4
ucbernie	2.3	4.38	.917	2.43	34.1	2.10	.989	3.22	196
ucbernie	2.3	4.38	.967	6.25	246	2.10	.981	3.63	205
figaro	2.05	2.89	.987	.403	40.0	1.56	.998	.22	15.2
monet	3.5	7.93	.958	0.62	49.1	4.43	.974	.413	16.7

Figure 3.19. UDP buffer service model parameters for 592 byte responses. Time Δ is the duration of the service preemption caused by a new arrival. The three different ucbernie models correspond to three different days of the week. Notice the difference in variability between first and normal service times.

Host	Preemption	First Service G_o				Normal Service G_1			
Name	Δ $[mS]$	D $[mS]$	α	x_1 $[mS]$	x_2 $[mS]$	D $[mS]$	α	x_1 $[mS]$	x_2 $[mS]$
ucbernie	2.8	5.95	.920	1.98	30.3	3.20	.990	3.05	79.4
ucbernie	2.8	5.95	.913	5.76	114	3.20	.956	1.26	99.6
figaro	3.1	3.10	.993	2.16	54.4	1.75	.992	.719	11.3

Figure 3.20. UDP buffer service model parameters for 1392 byte responses. The ucbernie parameters were taken for different days of the week.

tabulate the preemption time Δ, first service distribution G_o, and normal service distribution G_1. For example, we plot ucbernie's measured first service distribution and our model of it in Figure 3.21.

We predicted the expected number of buffer overflows through our service time model and conducted a series of experiments using *figaro*, our instrumented Sun 3/50. Examine Figure 3.22; notice the close agreement between our predictions and measurements. The model and measurements diverge near zero backoff time because the Ethernet's exponential backoff policy disperses the responses over an interval longer than the backoff time. At asymptotically high backoff times, we find another discrepancy. The expected number of losses that we measured was asymptotically about one half the value that we predicted. We conjecture that the recipient's coarse, 20 millisecond scheduling granularity prevents us from achieving perfect uniform backoff.

Our measurements of *ucbernie* diverge badly from the model (see Figure 3.28) because *ucbernie's* network interface is excruciatingly slow. Messages are first buffered in the interface's large buffers, and then, as backplane bandwidth permits, are copied into memory. This requires that we explicitly model the network interface, although the rest of the model remains intact. We discuss this further in Section 3.6.7.

3.6.5. Fragmentation

If the recipients' responses become fragmented by the network, then each fragment preempts the server. Because other network traffic and other responses may be interleaved with a fragment, the total preemption time is much more variable than with unfragmented responses. Our measurements of four kilobyte responses, transmitted as four fragments, show the total preemption time to be uniformly

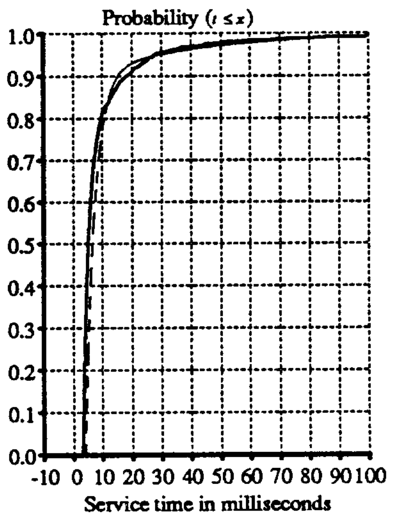


Figure 3.21. Measured (dashed) and modeled distributions of first service times for ucbernie. This distribution depends on processor type and workload (response size 592 bytes).

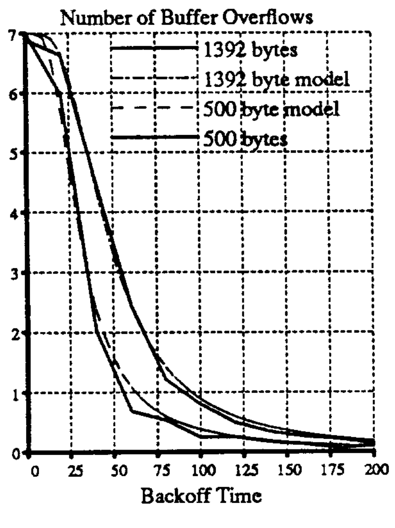


Figure 3.22. Figaro's measured and modeled expected number of buffer overflows depends on the backoff time. The UDP socket contained buffer space for three responses. Ten recipients replied.

distributed between some non-zero minimum and a maximum time. The IP layer buffers fragments until reassembly is complete, after which it passes them to the UDP layer, where they may cause buffer overflow.

We treat the sum of the preemption times as a constant, and reuse our M/D/1 busy period model. In Figure 3.23 we contrast the theoretical and measured numbers of overflows for the measured parameters listed in Figure 3.24.

3.6.6. Timeout Table's Sensitivity to Workload Variations

As our timeouts are statically calculated, one is prompted to ask about their sensitivity to variations in the workload, e.g., the three entries for *ucbernie* in Figure 3.19. In this section we suggest two approaches to calculating timeouts.

In production multicast systems, a ***timeout calculation server*** could periodically recalculate the timeout table based on the recently measured service time distribution. Since the service time's constant portion and preemption time do not drift with workload, they need be measured once. The sender could estimate and periodically send G_1 and G_o to a timeout calculation server (see Figure 3.25). The server could calculate a new timeout table, and return it to the sender. In the next section we show how to evaluate the parameters of the service time distribution's hyperexponential portion from the service time's measured moments.

Alternatively, for a two round multicast, one could avoid measuring the service time distribution by designing a control system that slowly adjusts the last round's timeouts so that the constraint is satisfied. This creates the difficulty that we can not calculate optimal next-to-last round's timeouts. From our experience, we believe that a good operating point for next-to-last round's timeouts is to

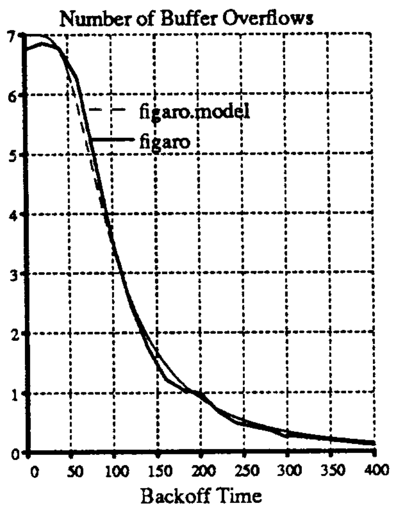


Figure 3.23. Figaro's measured and modeled expected number of buffer overflows for fragmented responses. We allocated 15 Kbytes of UDP socket buffer space, enough for three responses. Ten recipients sent 4184 byte responses.

Host Name	Preemption Δ [mS]	First Service G_o				Normal Service G_1			
		D [mS]	α	x_1 [mS]	x_2 [mS]	D [mS]	α	x_1 [mS]	x_2 [mS]
figaro	7.0	8.8	.88	1.11	8.8	3.3	.95	.85	10

Figure 3.24. UDP buffer service model parameters for 4184 byte fragmented responses.

select them so the server utilization is one. This simply requires that we measure the mean service time, rather than the distribution. We have not tried to evaluate the performance degradation that this entails.

3.6.7. Evaluating the Hyperexponential's Parameters

In the previous sections we modeled the UDP service times with a constant plus a hyperexponential distribution. We can easily measure D, the constant part that corresponds to the minimum service time, and $\overline{x^i}$, the moments of the service time's variable portion (the service time minus D). In this section we show how to calculate the parameters of the hyperexponential from these moments.

Hyperexponentials can model distributions with large variances. The density function of the hyperexponential distribution H_2 has three parameters x_1, x_2, and α:

$$Pr(t \le \tau) = 1 - \alpha e^{-\tau/x_1} - (1-\alpha)e^{-\tau/x_2}. \qquad (3.31)$$

We can use the first three moments of the measured service time $\bar{x}$, $\overline{x^2}$, and $\overline{x^3}$ to evaluate the H_2 distribution's three parameters by solving three nonlinear equations in three unknowns.

$$\bar{x} = 1! \left[\alpha x_1 + (1-\alpha) x_2\right]$$

$$\overline{x^2} = 2! \left[\alpha x_1^2 + (1-\alpha) x_2^2\right]$$

$$\overline{x^3} = 3! \left[\alpha x_1^3 + (1-\alpha) x_2^3\right]$$

We solve each equation for α; set the resulting equations equal to one another; and eliminate α.

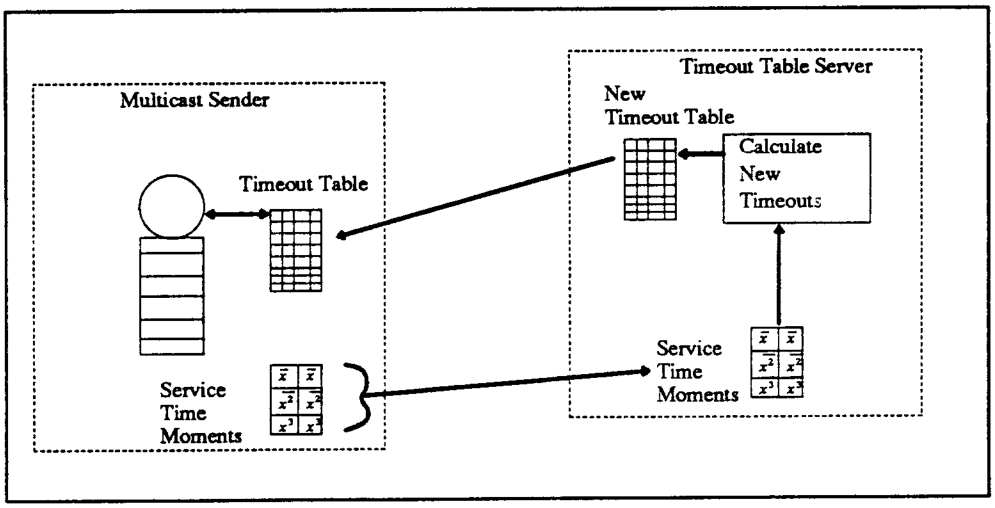


Figure 3.25. A timeout calculation server could compute a timeout table from the measured service time distribution.

$$\overline{x^2} - 2\bar{x}(x_1 + x_2) + 2x_1x_2 = 0$$

$$\overline{x^3}(x_1 + x_2) - 3\overline{x^2}(x_1{}^2 + x_1x_2 + x_2^2) + 6x_1{}^2x_2{}^2 = 0$$

Notice that these are equations in $x_1 + x_2$ and $x_1 \cdot x_2$, for which we substitute y_1 and y_2 respectively.

$$\overline{x^2} - 2\,\bar{x}\,y_1 + 2\,y_2 = 0$$

$$\overline{x^3}y_1 - 3\,\overline{x^2}\,(\,y_1{}^2 - y_2\,) + 6y_2{}^2 = 0$$

We easily solve this pair of equations for y_1 and y_2.

$$y_1 = \frac{3\,\bar{x}\,\overline{x^2} - \overline{x^3}}{6\,\bar{x}^2 - 3\,\overline{x^2}} \qquad y_2 = \frac{-\overline{x^2} + 2\,\bar{x}\,y_1}{2}$$

Substituting backwards we find x_1, x_2, and α,

$$x_1 = \frac{y_1 - \sqrt{y_1{}^2 - 4\,y_2}}{2} \qquad x_2 = \frac{y_1 + \sqrt{y_1{}^2 - 4\,y_2}}{2} \qquad \alpha = \frac{\bar{x} - x_2}{x_1 - x_2} \tag{3.32}$$

We can now derive hyperexponential parameters from the first three moments of the measured UDP buffer service time's variable portion. We tabulate the measured moments and calculated hyperexponential models of several computers in Figure 3.26.

3.6.8. Predetermined Backoff and Slow Interfaces

We must extend the model that we have just constructed to model buffer overflows as suffered when recipients employ predetermined backoff or when they employ uniform backoff but the sender's network interface is slow. It is possible to model these two problems similarly.

Let us first consider slow network interfaces. Our instrumentation of *ucbernie* and *ucbarpa* indicates that a 592 byte message spends 2.5 milliseconds buffered at the network interface before it is completely transferred into memory. A 1392 byte message spends 5.0 milliseconds. When added to the transmission time on a 10 Mb/s Ethernet, this gives 3.0 and 6.1 milliseconds respectively. These

Host		Measured Moments			Hyperexponential H_2		
Name	Type	$\bar{x}[mS]$	$\overline{x^2}[mS^2]$	$\overline{x^3}[mS^3]$	α	$x_1[mS]$	$x_2[mS]$
ucbarpa	VAX 785	2.24	78.2	8.43e3	.973	1.29	37.5
ucbernie	VAX 785	6.53	282	2.91e4	.901	3.27	36.5
figaro	Sun 3/50	1.08	310	4.26e3	.993	0.75	45.9
monet	VAX 750	2.68	206	3.02e4	.958	0.62	49.1

Figure 3.26. Moments of the UDP buffer service time's variable portion and hyperexponential model for 592 byte responses. Contrast this figure with Figure 3.19.

values are randomized by background traffic and the interrupt service latency. We see, therefore, that the network interface acts like an $M_n/D/1/b$ system (see Figure 3.27). Without recipient backoff, the interdeparture process of the network interface is roughly constant. With recipient backoff, the interdeparture times are either constant, or constant plus an exponential. Competing network traffic and background traffic addressed to the network interface randomize these interdeparture times.

With a few approximations, we can solve the problem analytically. Since the minimum interarrival spacing exceeds both the preemption time Δ and the constant portion of both G_o and G_1, we can eliminate both of these constants and the busy period calculation altogether. We can model the network interface's service time with an Erlang, determining its order from measurements. Its departure process drives the exceptional first service server. In summary, we must solve a tandem queue system, where the first queue has Poisson arrivals and Erlang service, and the second queue has hyperexponential, exceptional first and normal service, and is a system that we can solve exactly.

Predetermined backoff suffers fewer buffer overflows than uniform backoff (Figure 3.28). As a lower bound, we can ignore the variations in the responses' interarrival times that we observed in Section 3.4.2, and treat the interarrival time as constant. We can calculate the distribution of overflows by making a discrete approximating to the buffer service time distribution, and proceed to solve the $D_n/G/1/b$ system. Alternatively, we could try to model the responses' interarrival time with an Erlang distribution. The difficulty is calculating the variance of the arrival process from which we can calculate the Erlang's parameters. Recall that poor scheduling resolution adds variability to the recipients' response times. Recipient i's response arrives at time $i\tau/n + uniform(-\frac{\Delta}{2}, \frac{\Delta}{2})$. We but leave this problem unsolved.

3.7. The Dynasty Problem

Finding the common *i.i.d.* arrival time density of n recipients over an interval $\overline{0,\tau}$ that minimizes buffer overflow of a b-buffer, exponential server is, we believe, an unsolved problem. We call this the *Dynasty Problem* and begin this section by solving it for the special case of two recipients, one buffer, and exponential service time. The solution is composed of a uniform density and a bimodal density, and our studies suggest that the Dynasty problem's solution has this shape for any number of recipients and buffers. We conclude the section by summarizing our attempts to arrive at a general solution.

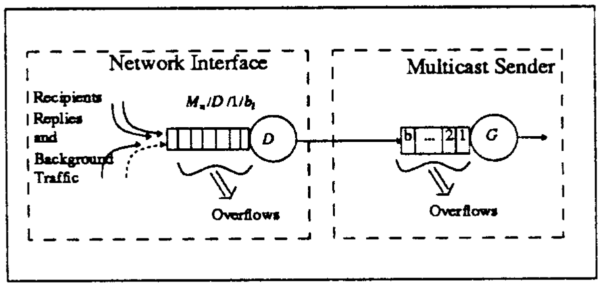
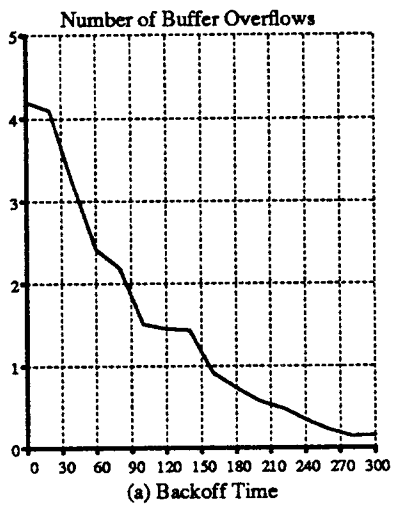


Figure 3.27. The VAX must copy the response from the network interface buffer across its slow bus into the operating system. Buffer overflow is possible at both the network interface and at the socket buffer.



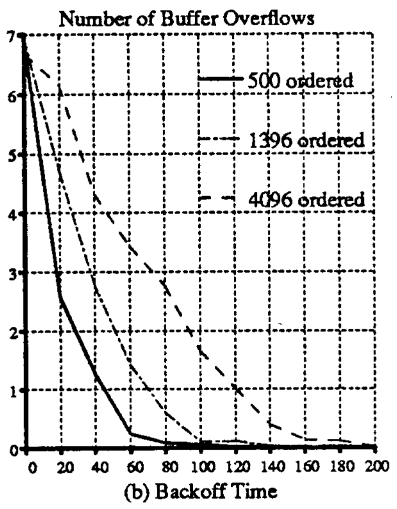


Figure 3.28. (a) Here we plot the number of UDP buffer overflows that ucbernie suffered. Ten recipients replied with 592 byte responses. We allocated three UDP buffers. Notice that, even at zero backoff time, the number of overflows does not reach seven. This is the result of the long time to copy messages from the Ethernet interface into memory. Contrast the shape of this curve with the shape of figaro's curve in Figure 3.22. The VAX must copy responses from the Ethernet interface across a slow backplane. (b) When recipients order their responses, the sender suffers fewer overflows. Here we plot the number of overflows suffered by a three buffer sender when the ten recipients order their responses. Contrast with Figure 3.22.

Consider the two response, single buffer, exponentially distributed server with mean $1/\mu$ Dynasty problem. Find the common *i.i.d.* response time density $h(y)$ on $\overline{0,1}$ that minimizes buffer overflow. Or equivalently, find $h(y)$ that minimizes the probability that the first response is still in service when the second arrives. For if it is, the second response overflows the buffer. We want to

$$\textit{minimize} \left\{ 2 \int_0^1 \int_{y_1}^1 e^{-(y_2 - y_1)\mu} h(y_2)\, h(y_1)\, dy_2\, dy_1 \right\}, \tag{3.33}$$

where $h(y)$ must satisfy

$$2 \int_0^1 \int_{y_1}^1 h(y_2)\, h(y_1)\, dy_1\, dy_2 = 1 ,$$

and

$$h(y) \ge 0 , \quad 0 \le y \le 1 .$$

Although this appears solvable by the calculus of variations [69], the optimal distribution $h(y)$ does not have continuous first and second derivatives, a requirement for that technique. Instead, we transform (3.33) into a discrete optimization problem. We subdivide $\overline{0,1}$ into M identical subintervals, and apply the method of Lagrange multipliers [8] to find the optimal, discrete distribution[7]. Denote the number of overflows by $L(\mathbf{p})$, where $\mathbf{p}$ is a vector composed of the probabilities p_i.

$$L(\mathbf{p}) = \sum_{i=1}^{M} p_i^2 + 2\sum_{i=1}^{M} \sum_{j=i+1}^{M} p_i\, p_j\, e^{-(j-i)\mu/M} .$$

We introduce Lagrange multipliers to incorporate the constraint equations:

$$l(\mathbf{p},\lambda) = L(\mathbf{p}) + \lambda \left[(\sum_{i=1}^{M} p_i) - 1 \right]$$

$$\nabla_{\mathbf{p}}\, l(\mathbf{p},\lambda) = \mathbf{0} .$$

$$\nabla_{\lambda}\, l(\mathbf{p},\lambda) = 0 .$$

We solve this system of $M+1$ linear equations for the discrete solution $\mathbf{p}$:

$$p_1 = p_M = \frac{e^{\mu/n}}{n(e^{\mu/n}-1)+2},$$

$$p_2 = \cdots = p_{M-1} = \frac{e^{\mu/M}-1}{M(e^{\mu/M}-1)+2}.$$

Taking the limit as integer M becomes large and substituting $(1+\mu/M)$ for $e^{\mu/M}$, we find the weight at the interval's endpoints is conserved:

$$p_1 = p_M = \frac{1}{\mu+2},$$

and the probability density of the interior points remains uniform. The optimal *i.i.d* continuous density function is

$$h(y) = \frac{\delta(y) + \mu + \delta(1-y)}{\mu+2}, \tag{3.34}$$

where $\delta(y)$ is the Dirac delta function. We call (3.34) the Dynasty distribution.

We see that $h(y)$ is the superposition of a uniform distribution and two impulses, one at either endpoint. As the mean service time $1/\mu$ decreases, the optimal distribution approaches the uniform distribution. As the mean service time increases, the uniform portion disappears, and the optimal distribution approaches the bimodal distribution with equal weights of one half at 0 and 1, the interval's endpoints.

We carry out integral (3.33), substituting (3.34) for $h(y)$, and find the expected number of overflows:

$$\frac{2}{2+\mu}.$$

We plot the expected number of overflows for the optimal distribution (3.34), the uniform distribution, and the bimodal distribution with probability one half at each end point in Figure 3.29.

[7] Simons solves a different problem similarly [59].

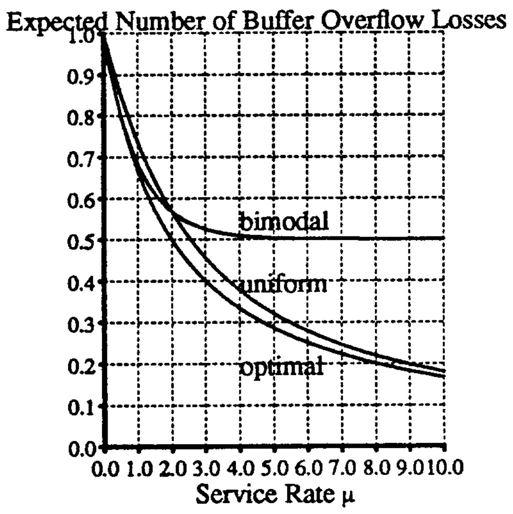


Figure 3.29. The Dynasty distribution minimizes overflows given two responses, one buffer, and exponential buffer service times. We plot overflows versus the Dynasty, the uniform, and the bimodal response time distributions.

3.7.1. Approximate Solutions to the Dynasty Problem

The Dynasty distribution outperforms the uniform distribution when $b > 1$, i.e., in multiple buffer systems, but we have not proved it is the optimal distribution. The Dynasty problem is of theoretical interest only; in practice, uniform backoff performs indistinguishably from Dynasty backoff.

We can try to solve the exponential service, multiple buffer Dynasty problem with Lagrange multipliers as we did above, treating the discrete result as a piecewise linear approximation of the continuous solution. Unfortunately this generates a system of nonlinear equations which we must solve to find the discrete solution. We investigated an iterative fixed point technique to approximate the solution to this system of nonlinear equations [20]. Unfortunately, the sufficient conditions for convergence were not satisfied, and convergence was not guaranteed. In the future we hope to find a general solution to the Dynasty problem, and extend it to the case where some buffers start non-empty.

3.8. Conclusions

When the sender of a multicast does not have enough buffer space for the responses that the multicast generates, some of these responses may overflow the available buffer space and be lost. In this chapter we propose a multiple round multicast algorithm (illustrated in Figure 3.2) that the sender executes to collect all the replies as quickly as possible. The sender chooses a round's timeout based on the number of outstanding responses, its buffer service time distribution, and its criterion (3.24) that the multicast completes by the last round. The criterion depends on the distribution of buffer overflows (3.20), which, in turn, depends on the buffer service time distribution. The sender calculates its table of timeout values, illustrated in Figure 3.14, from (3.26). The recipients, upon receiving a multicast, execute one of the two backoff algorithms given in (3.22) and (3.23).

Buffer overflow can occur at the network interface or within the protocol processing stack of the operating system and user program (see Figure 3.1). We illustrated the process of calculating the buffer service time distribution within the operating system by measuring and modeling it (see Figure 3.19) for the UDP protocol implementation of several VAX and Sun computer systems (3.28). This distribution accounts for both the minimum and variable time to process a response because it is the sum of a constant and an H_2 hyperexponential. It is also exceptional first service because first service times are highly variable due to processor scheduling latency. Process scheduling latency is the principal contributor to buffer overflow in the operating system, just as the processor's interrupt service latency is the principal cause of interface layer overflow. We used this model to calculate the number of responses that arrive during a service time. This is complicated by the fact that newly arriving responses preempt the processor, initiating their own busy period during which buffer service is suspended. The result of this model are the transition probabilities (3.30) needed to calculate the distribution of buffer overflows (3.20). Given the distribution of overflows, we can calculate a multiple round timeout table for UDP multicast. The close agreement between our theoretical model and measurements of UDP buffer overflow gives us confidence in our model (see Section 3.6.3).

We chose to model the highly variable part of UDP buffer service times with an H_2 hyperexponential distribution. To do this, we derived the relationship (3.32) between the first three moments of the measured service time distribution and the three parameters of the model H_2 distribution (3.31).

The distribution of buffer overflows is all that is necessary to calculate optimal timeout tables. We demonstrated how to calculate this distribution (and expected number) for various single buffer systems in Section 3.2 and for various multiple buffer systems Section 3.3. Single buffer systems yield simple expressions, and solving these systems lead to the techniques used with multiple buffer systems.

Our results can help system designers to choose the number of buffers to dedicate to a sender, and decide the cutoff point between hardware broadcast and successive unicast transmissions. They place selection of the retransmission timeout on firm mathematical ground.

In Chapter 6 we propose extending this work to study overflow at network gateways and LAN bridges caused by internet multicast [22].

The reader will have noticed the similarity of this problem to the invalidation storm problem presented in Chapter 2. A cache invalidation corresponds to the transmission of a multicast. The various processors attempting to acquire the spin-lock correspond to the multicast's recipients. In both cases these agents employ backoff to reduce computational overhead. We were better able to tune the multicast's backoff intervals because we knew, a priori, the number of multicast recipients. This leads us to propose two alternate ways to implement spin-lock backoff. We could track the number of spinning processors, and have them employ fixed backoff from a table, or have them order their lock requests in the order in which they executed *acquire_lock*. These strategies both require a secondary spin-lock, and should probably be reserved for highly loaded locks.

CHAPTER 4

SELECTING NETWORK GRADE OF SERVICE

We predict an explosion in the number of public utilities that sell virtual circuit digital network service (e.g., Telenet and Tymnet). When this happens, we will routinely make requests to servers across town and across the nation, paying for the messages that we send. Out of real time requirements borne by video and audio data, these networks will offer a spectrum of service grades from which to choose, permitting us to request specific bandwidths, loss rates and distributions of delay. We envision that nonreal-time traffic will absorb the residual bandwidth of virtual circuits established for real-time video and audio traffic, and that this residual bandwidth will be priced inexpensively because its service quality is relegated behind the service quality of higher tariff traffic. In this chapter we suggest that applications simultaneously tune their communication protocols and select their network service grade to minimize a cost function that they define. We illustrate this process by developing an algorithm to select remote procedure call (RPC) retransmission timeouts for lossy, tariff-bearing networks.

4.1. Introduction

In the near future, public utility operated networks will offer a spectrum of service grades and will charge us according to the number — and grade of service — of the messages we send [33,39]. In this chapter we investigate the relationship between tariffs, service grades, and communication protocol parameters. We consider how to simultaneously select protocol parameters and network service grade to minimize a cost function that we define. We illustrate this method by developing an expression for remote procedure call (RPC) retransmission timeouts for use over lossy, wide area, tariff-bearing networks.

4.1.1. Remote Procedure Call Timeouts

Consider an RPC between a *requestor* site and a *server* site. Since the network or the server may lose the request, and the network or the requestor may lose the reply, both the requestor and the server must be prepared to retransmit their respective messages. Most RPC implementations are designed for LAN environments and implicitly assume that message loss occurs at the server, rather than in the network. This assumption no longer holds: traffic in extended local area networks overflows bridge buffers; internetwork gateways drop packets due to congestion caused by stream protocols; satellite links lose messages due to noise; and, motivated by lower tariffs, we may purposely specify lossy grades of service.

Let us review how the Birrel-Nelson RPC protocol selects retransmission timeouts [12]. The Birrel-Nelson protocol sets the initial RPC retransmit timeout to exceed the expected value of the sum of the network's round trip message delay and the RPC server's service time, and doubles it after each retransmission (to a maximum of five minutes). Except for the initial transmission, the server acknowledges each subsequent retransmission to inform the requestor that it has not failed. It sends the results to the requestor as soon as they are computed. The server keeps a copy of the RPC's results in case the reply message is lost. These copies are eventually garbage collected and discarded.

Except for the way retransmission times are calculated, we employ the Birrel-Nelson protocol (see Figure 4.1). On lossy networks, doubling the retransmission timeout after each retransmission can work poorly because, as it may require many transmissions to successfully deliver a message, decreasing the frequency of retransmission can significantly raise the expected round trip service time. We modify the server so that its acknowledgements return an estimate of the RPC request's remaining service time. The requestor employs the server's new estimate of the remaining service time, x_r when

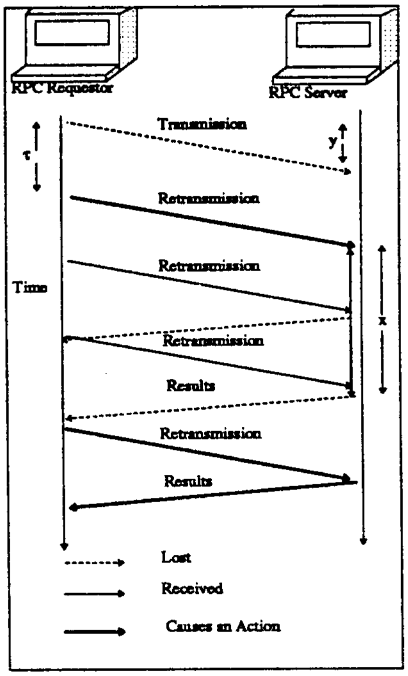


Figure 4.1. Our RPC algorithm's behavior.

calculating the next retransmission timeout. For backwards compatibility, if the server does not return this estimate with its acknowledgements, then the requestor defaults to doubling the remaining service time estimate after each retransmission.

4.1.2. Model of the Network and Server

We denote the bidirectional probability that the network loses a message by p, and assume that message losses are independent. For convenience, we define $q \equiv 1 - p$. We denote the operating system's cost of sending a message by K_s and the cost of receiving a message by K_r, e.g., if a message is sent and delivered the total cost is $K_s + K_r$. We denote the network's mean transmission delay by y, and assume that requests and responses experience the same delay. Regardless of whether or not a message is delivered, the network charges tariff $M(y, p)$ to send it. We denote the retransmission timeout before receiving the first acknowledgement by τ_{opt}, the retransmission timeout after receiving the first acknowledgement by τ'_{opt}, the server's service time probability density function by $f(x)$, and the server's expected service time by $\bar{x}$.

4.1.3. RPC Cost Function

One can gain fundamental insight into the requirements of an application by defining its cost function. Here we describe our RPC cost function. It is the weighted sum of the expected round trip service time or *latency*, the expected transmission and reception costs as borne by the requestor's and server's operating systems, and the charges that the network imposes for transmitting messages. We denote the latency by $\bar{l}$, the expected number of message sent by $\bar{n_s}$, the expected number of messages received by $\bar{n_r}$, and the cost function by L.

$$L = a \cdot \bar{l} + b \cdot (K_s \, \overline{n_s} + K_r \, \overline{n_r}) + c \cdot M(y,p) \cdot \overline{n_s} \, . \tag{4.1}$$

Notice that the weights a, b, and c scale each component of this sum into some common unit of cost, say dollars. With this definition, a and b depend on the rate at which we value our time. Since we can express K_s, K_r, $M(y,p)$, and $\bar{l}$ in the same units, without loss of generality we set $a=b=c=1$.

4.1.4. Outline of the Chapter

In Section 4.2 we calculate optimal timeouts for tariff-free, local area networks. In Section 4.3 we extend this to calculation to wide area networks. In Section 4.4 we construct an algorithm to select optimal network service grade and protocol retransmission timeout simultaneously. We illustrate this with a network with selectable loss and delay parameters. Assuming that slow messages cost less than fast messages and less reliable messages cost less than more reliable messages, we simultaneously minimize the RPC cost function over the retransmission timeout, network transmission delay, and network loss parameter. In Section 4.5 we discuss stability and how to incorporate our results into distributed systems, and we draw conclusions in Section 4.6.

4.2. LAN Retransmission Timeouts

In this section we calculate the RPC cost function for a local area network (zero network delay). We develop it piecewise. Since we assume that transmissions are delivered with probability q (and lost with probability p) and that network losses are independent, the distribution of the number of times the requestor must retransmit its RPC before the server receives it is geometricly distributed. Each lost retransmission increases the retransmission costs by K_s and the RPC's latency by τ (the period between retransmissions). The initial request costs K_s. When the server eventually receives the request, it experiences cost K_r. Together, this contributes

$$(K_s + \tau) \sum_{j=0}^{\infty} j \, p^j q \; + \; (K_s + K_r) = \frac{\tau p}{q} + (1 + \frac{p}{q}) K_s + K_r. \tag{4.2}$$

The server's first result transmission costs K_s, and the requestor receives it with probability q. This contributes

$$K_s + qK_r. \tag{4.3}$$

In the case that the server's results are delivered successfully on the first transmission, the expected latency is simply the RPC's expected service time, $\bar{x}$. If the service time is t, then the requestor retransmits $\lfloor t/\tau \rfloor$ times. The server, with probability q, receives a retransmission and generates an acknowledgement at cost $q(K_r + K_s)$, which, with probability q, reaches the sender at cost $q^2 K_r$. Together, this contributes

$$q \int_0^{\infty} f(t) \left[t + \left\lfloor \frac{t}{\tau} \right\rfloor (K_s + q(K_s + K_r) + q^2 K_r) \right] dt \approx q\bar{x} \left[1 + (K_s + qK_r)(1+q)/\tau \right] \tag{4.4}$$

The server's first result transmission is lost with probability q. The number of retransmissions send through time t, when the result transmission is sent and lost, is $\lceil t/\tau \rceil$. This contributes

$$p \int_0^{\infty} f(t) \left\lfloor \frac{t}{\tau} \right\rfloor \left[\tau + K_s + q(K_s + K_r) + q^2 K_r \right] dt \approx p\bar{x} \left[1 + (K_s + qK_r)(1+q)/\tau \right] \tag{4.5}$$

We now need to calculate the expected latency and retransmission costs that accrue between the the time that the result transmission is lost and the time that the requestor's retransmission and the server's reply are both successfully delivered (which occurs with probability q^2). The requestor must retransmit until it receives the server's reply. Each retransmission contributes latency τ and cost $K_s + q(K_r + K_s) + q^2 K_r$. This contributes

$$p \sum_{j=1}^{\infty} j \left[K_s + q(K_r + K_s) + q^2 K_r + \tau \right] (1 - q^2)^{j-1} q^2 = \left[(K_s + qK_r)(1+q) + \tau \right] \frac{p}{q^2}. \tag{4.6}$$

Putting together (4.2), (4.3), (4.4), (4.5) and (4.6), the local area network cost function L is

$$L \approx \left\{ \bar{x} + \tau\left(\frac{p}{q^2} + \frac{p}{q}\right) \right\} + K_s \left\{ 2 + \frac{p}{q} + \frac{(1+q)\bar{x}}{\tau} + \frac{(1+q)p}{q^2} \right\} + \tag{4.7}$$

$$K_r (1+q) \left\{ 1 + \frac{q\bar{x}}{\tau} + \frac{p}{q} \right\}.$$

We minimize this quantity by setting $\partial L / \partial \tau$ to zero and solving for τ. This value of τ is our optimal retransmission timeout τ_{opt}:

$$\tau_{opt} = q \sqrt{\frac{\bar{x}\,(K_s + qK_r)}{1-q}}. \tag{4.8}$$

After receiving the server's acknowledgement that it has received the request and is computing, we know that the can drop the component of the cost function that accounts for the cost to reliably send the request to the server (4.2). We employ the server's estimate of the remaining service time, x_r, transmitted with its acknowledgement. Our post-acknowledgement optimal timeout $\tau^r{}_{opt}$ is

$$\tau^r{}_{opt} = q \sqrt{\frac{(1+q)\bar{x_r}(K_s + qK_r)}{1-q}}. \tag{4.9}$$

This concludes our section on LAN retransmission timeouts.

4.3. WAN Retransmission Timeouts

We are now ready to account for the network transmission delay of wide area networks. We assume each message experiences constant message transmission delay y. To derive the optimal timeout τ_{opt}, we proceed piecewise as we did above.

We calculate the expected latency $\bar{l}$ first. Just as in the LAN case (4.2), it may require numerous retransmissions to successfully pass the request to the RPC server. This requires, on average, p/q retransmissions, contributing $p\tau/q$ to the RPC latency (and pK_s/q to the retransmission costs). When the server's result is not lost, the RPC's expected latency is the sum of the expected time to inform the server, the round trip network delay, and the expected service time. This contributes

$$q(\tau p/q + 2y + \bar{x}). \tag{4.10}$$

When the server's result transmission is lost, the RPC's expected latency is τ/q^2 (4.6) greater than the expected arrival time of the acknowledgement just prior to the lost result transmission, contributing

$$p(\tau p/q + \tau \int_0^{\infty} f(t) \left\lfloor \frac{2y+t}{\tau} \right\rfloor dt + \tau/q^2) \approx p(\tau p/q + 2y + \bar{x} + \tau/q^2). \tag{4.11}$$

The retransmission costs are the sum of the minimum transmission costs $2K_s + (1 = q)\,K_r$ (see (4.2) and (4.3)), the expected initial transmission cost $K_s p/q$, the retransmissions sent through the moment the server's response should arrive, and, in the case that this reply is lost, the expected number of retransmissions needed to collect the server's response (see (4.6)):

$$2K_s+(1+q)\,K_r + (p/q)K_s + (K_s + q(K_s + K_r) + q^2K_r)\left[\int_0^\infty f(t)\left\lfloor \frac{2y+t}{\tau}\right\rfloor dt + \frac{p}{q^2}\right] \qquad (4.12)$$

$$\approx K_s(2+p/q + (1+q)(\frac{2y+\bar{x}}{\tau} + \frac{p}{q^2})) + K_r(1+q)(1+q\;(\frac{2y+\bar{x}}{\tau} + \frac{p}{q^2})).$$

We collect (4.10), (4.11) and (4.12) to express the WAN RPC cost function:

$$L \approx \left[\frac{p\tau}{q}(1+\frac{1}{q}) + 2y + \bar{x}\right] + K_s\left[2+p/q + (1+q)(\frac{2y+\bar{x}}{\tau} + \frac{p}{q^2})\right] + \qquad (4.13)$$

$$K_r\left[(1+q)\,(1+q\;(\frac{2y+\bar{x}}{\tau} + \frac{p}{q^2}))\right].$$

In (4.13), the first parenthesized component corresponds to $\bar{t}$, the second to $\bar{n_s}$, and the third to $\bar{n_r}$ (see (4.1)). We set the $\partial L/\partial\tau$ to zero, replace p by $1-q$, and solve for the optimum timeout τ:

$$\tau_{opt} = q\sqrt{\frac{(2y+\bar{x})(K_s + qK_r)}{1-q}}\,. \qquad (4.14)$$

After receiving our first acknowledgement, we drop the component of the cost function that corresponds to reliably transmitting the request to the server and recalculate the post-acknowledgement optimal retransmission timeout τ^r_{opt}:

$$\tau^r_{opt} = q\sqrt{\frac{(1+q)\,(2y+\bar{x}_r)(K_s + qK_r)}{1-q}}\,. \qquad (4.15)$$

In Figure 4.2 (a) and (b) we plot the cost function and its standard deviation for several sets of parameters and service time distributions. Note that the standard deviation σ, calculated by simulation, grows when the timeout exceeds τ_{opt}. In Figure 4.2 (c) we plot the exact cost function L for uniform, exponential, and constant service time distributions, as evaluated numerically without ignoring the floor operators within the integrals. We indicate τ_{opt} as calculated from (4.14). We see that having ignored the floors within integrals (4.11) and (4.12) negligibly affects τ_{opt}.

4.4. Selectable Grade of Service and Retransmission Timeouts

In this section we investigate how to simultaneously select network service grade and the retransmission timeout. We assume that the network charges us for each message we send, charges more for service grades with decreased transmission delay and lower loss probability, and lets us specify our desired service grade parameters [5]. We assume the network lets us specify the mean transmission delay y, $y \ge y_0$, and loss probability p, $p \ge p_0$, and charges us a per message cost $M(y,p) \ge 0$, differentiable and decreasing in both y and p, where y_0 and p_0 are the minimum network delay and probability of message loss. Our retransmission timeouts depend on the service grade, because, for example, if we specify lossy service to reduce our per message costs, τ_{opt} decreases (4.13).

We now derive the optimal retransmission timeout τ_{opt} and service grade (y_{opt}, p_{opt}) that minimize our RPC cost function subject to the constraints $\tau > 0$, $y \ge y_0$, and $p \ge p_0$. Assume again that $M(y,p)$, K_s, K_r, and $\bar{t}$ are expressed in identical units. We form the cost function L (4.1) from

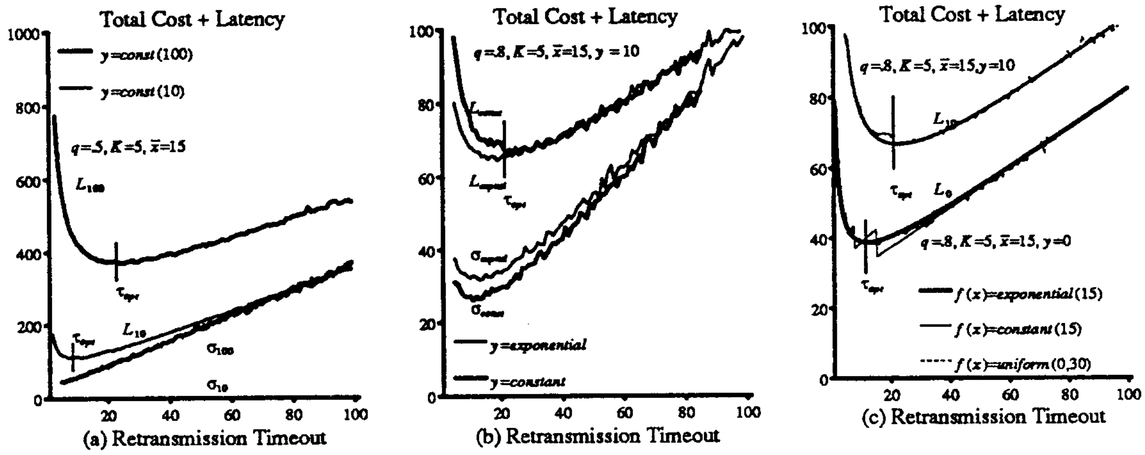


Figure 4.2. In these three figures, we plot the RPC cost function versus the retransmission timeout. We set $K_r = K_s \equiv K$. Contrasting graphs (a) and (b), notice that the cost function's standard deviation, as calculated by simulation, increases with decreasing loss rate. (a) The cost functions' standard deviations are indistinguishable despite the order of magnitude difference in the network transmission time. (b) The cost function's standard deviation is relatively insensitive to the network transmission delay distribution. (c) We plot the exact cost function for exponential, constant, and uniform service time distributions. The curves are nearly indistinguishable, indicating that ignoring the floor operators in the integrals negligibly affects the cost function.

$\bar{t}$, n_s, and n_r as identified in the paragraph following (4.13), and must minimize it over τ, y, and p.

Since we have assumed that $M(y,p)$ is differentiable, we can apply the Kuhn-Tucker algorithm [8] to find the service grade (y_{opt}, p_{opt}) and retransmission time τ_{opt} that minimize the cost function (note that the cost function's Hessian must be positive definite [8]). We make concrete by considering a possible message cost function:

$$M(y,p) = \frac{\alpha}{y}.$$

We must introduce constraint equation g_y and its their corresponding Lagrange multiplier λ_y. The Kuhn-Tucker conditions reduce to

$$g_y = y - y_0 \geq 0,$$

$$\frac{\partial}{\partial \tau}\left[L\right] = 0,$$

$$\frac{\partial}{\partial y}\left[L - \lambda_y\, g_y\right] = 0,$$

$$\lambda_y\, g_y = 0,$$

where $\tau \geq 0$, and $y \geq 0$.

Since p is given, the optimal pair (τ_{opt}, y_{opt}) is the solution to the unconstrained problem except for when this solution violates the constraint $y_{opt} \geq y_0$. In this case $y_{opt} = y_0$ and τ_{opt} is given by (4.14) and $\tau^r{}_{opt}$ by (4.15), substituting $M(y_0, p) + K_s$ for K_s.

In Figure 4.3 we plot the optimal timeout and transmission delay as a function of the proportionality constant α. We see that as operating system retransmission costs increase, the retransmission timeout increases and we request a lower delay, higher tariff network grade of service.

4.5. Stability

Our timeout calculation is easily implemented, but it requires that we cache estimates of servers' service time and network loss probabilities and round trip delays. We can update the cached estimates upon completing an RPC, so that the estimates for frequently used sites should be good. We guarantee stability by having the server return an estimate of the expected completion time with each acknowledgment. For example, if our request is enqueued pending the outcome of four other requests, the server acknowledges with an expected service time of $x = 5\,\bar{x}$, where $\bar{x}$ denotes the expected service time of a single request. The complication to the server, which needs to perform queue look-ahead, is balanced by the increased stability.

4.6. Conclusions

RPC developed as a local area network protocol and employed either large, constant retransmission timeouts or exponentially increasing timeouts values because they assumed that network losses were rare. Although exponential backoff is adaptive, it may not adapt correctly, and requires that system programmers exhibit excellent intuition. For example, Sun Microsystem's NFS file system suffered from premature timeouts because, although they employed Birrel-Nelson style retransmission timeouts, the initial timeout value was chosen too small [38]. Although billed as an adaptive backoff algorithm, it did not adapt the base of the retransmission timeout. Real systems should estimate the network's round trip time, server's service time, and network's loss probability and employ these estimates to calculate truly adaptive timeouts. Our expressions for the optimal retransmission timeout (4.14) and (4.15) behave correctly as the network loss rate increases.

Our retransmission timeout grows for two reasons. It grows after we receive the server's first acknowledgment because we know that the server is computing, and it grows as our estimates of the remaining service time grows. Our timeouts shrink as our estimate of the probability of message loss

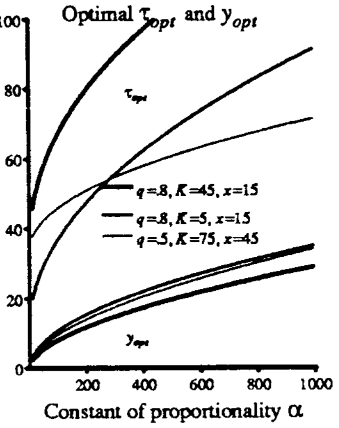


Figure 4.3. Retransmission timeout τ_{opt} and transmission delay y_{opt} are functions of the transmission delay cost function $M(y, p) = \alpha / y$. We have set $K_r = K_s \equiv K$.

increases. The effect of these three factors is that our timeout grows less quickly than Birrel-Nelson's timeout but quickly enough to remain stable despite server congestion, and to minimize excess computational overhead. The RPC cost function explicitly considers the load on the server that retransmissions generate. Our timeout selection can be made backwards compatible with existing RPC implementations.

We believe our RPC cost function reflects the tradeoffs that protocol designers make, but the optimal retransmission timeout that we calculate is only as good, of course, as the estimates that drive it. Reasonable values for K_s and K_r must be measured (on Sun 3/50 UNIX systems, this number is 2-5 milliseconds). Implementation should measure and cache the average service time $\bar{x}$, network loss probability p and round trip delay $2y$.

With selectable service grade networks we will have to select both the RPC retransmission times and the network service grade parameters, e.g., transmission delay and loss probability. We solve this problem by adding the cost per transmitted message into the RPC cost function and then minimizing the cost function over the timeout and selectable parameters.

Implicitly, we have assumed that RPCs are not the cause of network congestion and that message losses are independent of each other. Since our algorithm decreases the retransmission timeout when the loss probability p increases, we may add to network congestion. We believe this is reasonable since applications that employ RPC do not exchange large quantities of data, as do, for example, file transfer protocols. Tariffs will reduce the incentive to be greedy and cause congestion. An area for future research is understand the relationship between tariffs and congestion.

Our approach can be applied, for example, to rate-based bulk-data transfer protocols and to other request-reply protocols. In Chapter 6 we propose additional work along these lines.

CHAPTER 5

High Resolution Timing with Slowly Ticking Clocks

When tuning operating system and network code, profiling programs, analyzing message interarrival times, and accurately measuring device characteristics, a high resolution clock is often indispensable, as one cannot measure service time *distributions* without one. This chapter describes a microsecond clock that we designed and built for Sun 3 and Sun 4 workstations[1]. One can measure average service times without a high resolution clock. This chapter explains how to measure average times with high precision in the absence of such a clock. We pose and answer the question: "how many measurements are needed to report timing data to three significant digits?"

5.1. Introduction - Who Needs a Microsecond Clock

Beginning with its Sun 3 workstations, Sun Microsystems substituted an Intersil, battery backed up, time-of-day clock chip for the microsecond resolution clock chip present in their earlier models. The new clock interrupts the processor every ten milliseconds. By default, the Sun operating system discards every other interrupt, degrading the clock resolution from ten to twenty milliseconds. Sun kept an I.C. socket for a data encryption chip (DES), but chose to leave it empty as well as sockets for up to three machine-specific chips that are necessary to activate the DES socket. As we had no use for the DES chip, we designed a high resolution clock circuit board that plugs directly into the DES chip's socket. To install the board, one only needs to insert it into the DES chip's socket and any machine-specific support chips into their respective sockets[2] and to add a device driver to the operating system. In October 1989, we had three dozen of these clocks in use at U.C. Berkeley and other universities and laboratories.

In the next section we describe our clock's design. In Section 3 we derive the number of measurements needed to accurately report average timing data as a function of the clock's resolution. We show that without a microsecond clock, it may require several hours or days to report average timing data to three significant digits. We draw conclusions in Section 4.

5.2. Our Design

In this section we describe our clock's design. Because Sun guards its workstation's schematics, we designed our clock to meet the timing requirements and eight-bit interface of the Advanced Micro Devices (AMD) Am9518 DES chip (also known as the Zilog Z8068). We built the clock around AMD's Am9513a counter chip because its five 16-bit counters can be atomically saved with a single instruction yet read over an eight-bit bus. While other counter chips have multiple sixteen-bit timers, the Am9513a is the only chip that can save more than one timer with a single instruction. Although the timer and DES chips carry similar designations, their pin assignments, interface protocol, and timing needs are quite different.

Both chips have a data port and a control port which can be written or read. The DES chip selects the appropriate port with separate data-strobe and control-strobe pins. The timer chip's single strobe serves for both ports; its data/control pin selects between the two ports. The DES chip has a single read/write pin; the timer chip has separate read and write pins. We placed a programmable logic

[1] Contact us to obtain the schematic diagram, SunOS device driver, or completed timer boards. Note that we cannot support Sun 3/80, 386i, or Sparc Stations, but Sparc Stations have an internal microsecond timer.

[2] Here we list the missing, machine-specific chips: Sun 3/75, 3/140, 3/150, 3/160 systems require a 74ALS245 octal-buffer and a PAL22V10. Sun 3/260 and 3/280 systems require a 74ALS245, a PAL16R4, and a PAL16R8. Sun 4/110, 4/150, 4/260, 4/280 systems require a PAL22V10. Sun 3/50 and 3/60 systems need no additional support chips.

array (PAL) on our timer board that converts the DES chip's control signals into the timer chip's control signals. One cannot meet the timer chip's data/control pin's setup and hold requirements given the DES chip's published timing specifications. As we could neither modify the Sun's hardware nor firmware, yet wanted the board to work in all Sun 3 and Sun 4 processors, we chose a solution that adds a few instructions to the sequence of instructions necessary to read the timer. We drive the timer's data/control pin from a set-reset flipflop built from two of the PAL's gates. The data port is selected when this flipflop is set and the control port when it is reset. We precede accesses to both ports by appropriately setting or resetting this flipflop from the DES chip's data strobe and read/write signal.

The timer board's device driver sets the timer chip's fifth timer to divide the 4.0 megahertz oscillator frequency by four, concatenates the timer chip's lower four timers, and drives the lowest of these with the output of the fifth timer. The board can return a simple binary count or a 64-bit UNIX *timeval* structure. The *timeval* mode is useful for compatibility with the UNIX system call *gettimeofday()*. The clock appears as device */dev/tmr0* and can be read through the file system or through a system call. It can also be mapped into the user's address space, giving user programs quick access to the timer's registers and the microsecond time.

In Figure 5.1 we report the overhead associated with reading 32-bit timestamps and 64-bit *timevals*. Note that a 3/50's display steals cycles from main memory, as it does not have a separate frame buffer. Hence we give two sets of overhead figures for it, one for when the display is blanked and another for when it is active. When the display is active, the machine slows down by more than twenty percent. The overhead varies a few microseconds from read to read due to the speed of the memory and memory contention. Infrequently, when reading the clock from a user process, the process may be descheduled within the instrumented code, resulting in large times. This is, unfortunately, unavoidable, but easily detectable; any clock would suffer the same inconvenience. Interrupts can also increase the measured time. Since user programs cannot disable interrupts they cannot read the clock atomically when it is mapped into the user's address space, and it may return nonsensical values if other user processes or the operating system also read the clock. (User programs can always read the clock atomically through the system call). This occurs because the competing process may read the timer, which resets the timer chip's internal pointer. The original process, when it resumes, will continue reading bytes from where it left off, unaware that these are not the bytes that it wants.

5.3. Profiling Code with a Low Resolution Clock

Perusing the operating system's literature, we often see tables of performance measurements collected on computers with poor clock resolution [9, 15, 40, 51]. The highest possible resolution of an

Timestamp Type	3/50	3/50 Blanked	3/60	3/260
Kernel 32-bit	24.0	19.5	16.5	11.
Kernel 64-bit timeval	38.2	30.2	27.	18.
User 32-bit	14.0	11.3	11.1	7.
User 64-bit timeval	27.0	23.5	21.	13.
System call 32-bit	238.	179.	140.	87.
System call 64-bit timeval	254.	190.	162.	91.

Figure 5.1. Measured overhead in microseconds to read the high resolution clock (we do not state the precision and degree of confidence of the measured overhead because overhead is machine dependent).

IBM PC/RT is 125 microseconds; the clock resolution of microVAX-II workstations is 20 milliseconds, and, as we have mentioned, the highest possible resolution of Sun 3 and Sun 4 workstations is 10 milliseconds. Often practitioners report times as short as 10-300 microseconds to three decimal places based upon the average of a few dozen to a million iterations through the code. Instrumenting code and making measurements can be quite time consuming. For example, measurements of network code are usually repeated for several sizes of messages, and measurements of transaction systems are usually repeated for various numbers of participants. Let us consider the process by which we collect measurements and then pose the following question. How many iterations suffice to report our measurements to two or three significant digits given our hardware clock advances every Δ milliseconds?

We profile code by recording the difference in the clock's values upon entering and exiting each instrumented code segment. For example, these segments could correspond to the various layers of a communication protocol stack. Since the clock time only advances every Δ milliseconds, it may not advance between entering and exiting a code segment of duration less than Δ milliseconds. Without loss of generality, assume that we want to measure a code segment of duration $\delta \le \Delta$ milliseconds. First we must assure ourselves that the measurements are not initiated by (or otherwise synchronous to) clock ticks. During a code segment of duration δ the clock advances with probability $p = \delta/\Delta$. If the clock advances we record a one; if not, we record a zero. We define the event σ_i to be one if the clock advances during iteration i and zero otherwise. After n iterations the clock advances S_n ticks:

$$S_n = \sum_{i=1}^{n} \sigma_i \,.$$

After n iterations, the average time through this code segment is

$$\hat{\delta} = \hat{p}\,\Delta = \frac{S_n\,\Delta}{n}\,,$$

and, in our research papers, usually report this value to two or three decimal places.

Let us consider the precision of measurements made in this manner. Notice that the σ_i's approximate the two possible outcomes of a sequence of Bernoulli trials [28]. We say approximate because the outcomes of subsequent trials may not be totally independent. That is, if we know that the clock advanced during the last measurement, this may increase the probability that the clock does not advance during this measurement. Assuming independence, we want to know how many iterations are required to report these times to two or three significant digits. Since our measurements result from randomized experiments, we can only make probabilistic statements about their precision. Conceivably, but with low probability, we may never observe a single clock tick. Let $\delta = p\;\Delta$ be the true interval length and $\hat{\delta}$ be our estimate of it based upon n observations. We would like to report our experimental data as

$$P\left[\,-\varepsilon \le (\hat{\delta} - \delta) \le \varepsilon\,\right] \ge \alpha\,. \tag{5.1}$$

That is, with probability α or greater our measurement is within ε of the true value. We can apply the DeMoivre-Laplace limit theorem to find the minimum number of iterations required to make such a statement. The DeMoivre-Laplace theorem states

$$P\left\{\, z_1\sqrt{npq} \le S_n - np \le z_2\sqrt{npq} \,\right\} \tag{5.2}$$

$$\rightarrow \Phi(z_2) - \Phi(z_1)\,,$$

where $q = 1 - p$ and $\Phi(x)$ is the normal distribution function. We will need the following identity concerning $\Phi(x)$:

$$\Phi(-x) = 1 - \Phi(x). \tag{5.3}$$

We can combine (5.1), (5.2), and (5.3) into a single expression:

$$P\left\{ \frac{-\varepsilon n}{\Delta} \le S_n - np \le \frac{\varepsilon n}{\Delta} \right\} \rightarrow 2\Phi(z_\alpha) - 1.$$

Equating $\varepsilon n/\Delta$ with $z_\alpha\sqrt{npq}$, we arrive at an expression for n:

$$n = \frac{z_\alpha{}^2\Delta^2 pq}{\varepsilon^2}, \tag{5.4}$$

for which we must evaluate z_α:

$$2\Phi(z_\alpha) - 1 = \alpha. \tag{5.5}$$

For the remainder of this chapter, we use an arbitrary confidence level of $\alpha = 0.95$. We invert (5.5) from tables of the normal distribution:

$$z_{.95} = \Phi^{-1}(0.975) = 1.96.$$

Below we summarize the number of iterations n_2 and n_3 required to report times to two and three decimal places such that the least significant digit is within one unit of the true value with probability $\alpha = 0.95$.

Consider the problem of instrumenting a UDP/IP protocol stack implementation. Conceivably each iteration could take about 10 milliseconds. In Figure 5.2 we see that measuring to three significant digits a code segment of about a millisecond duration requires about $7 \cdot 10^5$ iterations. At 10 milliseconds per iteration, we will need to run this experiment for about two hours. If some profiled section of this code is on the order of a hundred microseconds, then the experiment could have to run for an entire day!

5.4. Conclusions

Reporting performance data to two or three significant digits with high confidence may require many iterations, depending upon the clock resolution. When we report performance numbers of intervals shorter than our clock's resolution, we should perform enough iterations to achieve a high confidence level for the number of significant digits that we report (see (5.4)). Without a high resolution clock, this may require many hours of measurements.

When it is necessary to measure a distribution rather than an average, a microsecond resolution clock may be essential. The microsecond resolution clock described in this paper will work in all Sun

δ	n_3	n_2
$1 \cdot 10^1$ μSec	$8 \cdot 10^7$	$8 \cdot 10^5$
$1 \cdot 10^2$ μSec	$8 \cdot 10^6$	$8 \cdot 10^4$
$1 \cdot 10^3$ μSec	$7 \cdot 10^5$	$7 \cdot 10^3$
$1 \cdot 10^4$ μSec	$4 \cdot 10^4$	$4 \cdot 10^2$

Figure 5.2. Required number of iterations to measure code segments of duration δ to three and two significant digits given a clock with $\Delta = 20$ millisecond resolution to a confidence level of $\alpha = 0.95$.

workstations equipped with the DES socket. A rudimentary device driver for the clock can be written in three pages. Finally, we note that it is possible to build a clock like ours in place of non-essential chips in other computers.

CHAPTER 6

PROPOSALS FOR FUTURE WORK

This chapter outlines two independent problems that we did not pursue in this dissertation, but does not purport to offer their solutions. It can be read independently of the preceding chapters, and is intended to serve as the basis for research proposals on flow control for internetwork multicast and flow control for gigabit networks.

6.1. Internet Multicast

Researchers believe that the Internet should support multicast [44], and internetwork multicast has been an active research topic this decade [13,22,66]. The greater part of this research has dealt with multicast flooding algorithms and maintaining multicast group membership; therefore we assume that solutions to these problems have been implemented [23]. While flow control algorithms exist for communication streams, no flow control algorithm exists that the recipients of an internetwork multicast can employ. Our goal is to design a such an algorithm.

6.1.1. Flow Control for Internetwork Multicast

Internetwork multicast's principal users are distributed query and transaction systems, multimedia teleconferencing, and network management. The literature differentiates open groups from closed groups by whether or not non-group members can send multicasts to the group [44]. Without loss of generality we limit our attention to closed groups, as non-members can always post multicasts through a group member. Deering's work [23] applies to unreliable, Internet Protocol (IP) multicast, useful for resource location. In contrast, we are interested in reliable multicast, useful for text and graphic transmission and distributed transaction systems. Reliability means that each group member must acknowledge receipt.

One problem with internetwork multicast is that it may initiate an *implosion* of replies that congest the network. We say implosion rather than explosion because the replies are all sent to the multicast's sender. In contrast to the sender of a reliable stream that can reduce its window-size upon detecting network congestion, the sender of an internetwork multicast can do nothing, and the recipients' replies may consume an enormous number of network buffers.

Active agents can summarize recipients' responses if these consist of an acknowledgement bit and little or no data. On one hand, active agents cannot summarize lengthy, unique responses. On the other hand, since acknowledgements consume negligible buffer space, combining them may be neither necessary nor efficient. Our flow control algorithm should employ active agents only where appropriate.

6.1.2. Resource Reservations and Network Model

We must have a model of the network if we are to make quantitative statements about it. A control theory approach must adapt to our notion, which may be out of date, of the network's state. Instead of pursuing such an approach, we plan to pursue a resource reservation approach, because, if multicasts are infrequent, a control system approach would have a poor knowledge of the network's state. If multicasts are periodic, then resource reservations permit better management of limited network resources. Therefore we model an internetwork as a tree of limited-buffer gateways and nodes whose leafs are recipients and whose internal nodes are gateways (see Figure 6.1). We assume that gateway i agrees to provide j buffers with probability b_{ij}, and to service these buffers with service distribution $G_i(t)$. We know the delay τ_i through the link to its parent. We assume that stations on a LAN communicate instantaneously, and that we know the interarrival time distribution $A(t)$ between multicasts[1].

6.1.3. An Internetwork Flow Control Algorithm

Let us generalize the multiple round LAN multicast algorithm developed in Chapter 3. Our problem is to minimize the latency τ to inform and collect responses from every recipient while meeting some termination constraint ϕ. We assign to each local area network i a backoff time x_i, inserted into the message header by the multicast router [23], over which the recipients schedule their responses. Once sent, a response travels to its destination agent.

We must solve a nonlinear optimization problem with an equality constraint. We want to minimize the multicast latency $\tau(\mathbf{x})$ while meeting constraint $\phi(\mathbf{x})$, where $\mathbf{x}$ is a vector composed of each LAN's backoff times. Unfortunately, it is not possible to find analytical expressions for ϕ and τ. Wong and Gopal [36] needed several approximations to bound the latency of a broadcast to a tree of nodes with infinite buffers.

We intend to estimate τ and ϕ via simulation, and to assign $\mathbf{x}$ via a gradient optimization technique. Since simulation introduces noise into these functions' estimates, the gradient technique will need to be highly immune to noise.

6.1.4. Research Questions

We must answer several questions concerning congestion and resource reservations. For example, does network congestion affect more than one gateway? Should we measure joint gateway arrival time, service time, and queue length distributions to investigate their dependence? Should we employ an adaptive backoff algorithm that does not employ reservations?

Server and arrival rate variability cause queue lengths to grow. From the data that we collected in Chapter 3, we know that protocol service time distributions can be more variable than exponential, and are preemptive. This preemptive nature increases the probability of buffer overflow. We propose to measure the gateway interarrival process, queue length, and service time distributions at a set of

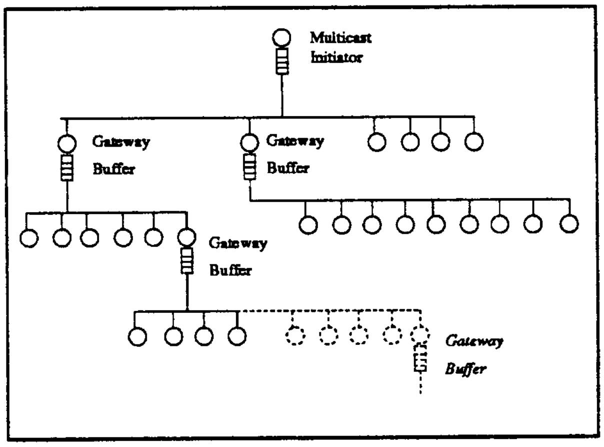


Figure 6.1. An internetwork viewed as a tree of destinations and gateways.

[1]A token based synchronization scheme can limit the number of concurrent multicasts.

gateways across a campus or regional network, synchronizing clocks from WWV radio signals. Such a set of joint distributions has not been published (hence very probably not collected) for the DARPA Internet or for a campus network.

6.2. Gigabit Networks

The backbone of future networks will consist of gigabit per second (Gb/s) links, limited only by the speed of electrical-optical interfaces [2] and the rate at which switches can forward packets. Round trip transmission delays in North America will be under sixty milliseconds, and wavelength division multiplexing leads us to anticipate essentially unbounded backbone bandwidth. Laboratory researchers have placed over twenty different gigabit channels on a single optical fiber, and expect this number will reach into the hundreds [34]. The backbone will be demultiplexed onto regional and local area networks operating in the tens to hundreds of Mb/s [53]. To exploit these high bandwidths and moderate delays in future internetworks, researchers must address several issues in switching, routing, flow control, network management, and security [44]. While we can predict infinitesimal bit-error rates and unlimited bandwidth within the network's backbone, it will be many years before this technology reaches the regional and local area networks (see Figure 6.2).

6.2.1. Virtual Circuit Networks

Video and audio transmission require, on one hand, that the network statistically bound its delay, delay jitter, and loss rate, properties of circuit switched networks. Efficiency requires, on the other hand, that these circuits be statistically multiplexed. Statistically multiplexed, packet switched networks have been dubbed virtual circuit networks. Gigabit rates may force a shift to virtual circuit networks to make routing quicker and probabilistic performance guarantees possible[2]. These network's bandwidth-delay product is huge; a three thousand mile link requires a ten megabyte buffer. We plan to investigate the combined performance of rate-based flow control [19] and statistical multiplexing to share virtual circuit buffers. Since future network traffic will combine audio, video, text, and graphics, each of which require different bounds on delay, jitter, and loss rate, a conversation's component traffic may travel different routes. We must design rapid, efficient connection establishment algorithms to create routes and reserve network bandwidth. This requires that we characterize the various traffic types' service grades and arrival processes [29, 33], but how can we characterize what does not

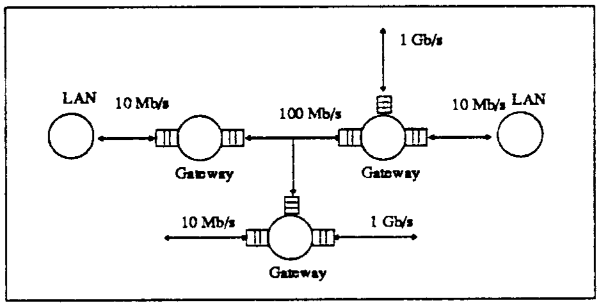


Figure 6.2. Although the long-haul portion of the network operates at gigabit rates, the regional and local networks will not.

[2]We should note that a virtual circuit architecture does not exclude connectionless traffic.

yet exist?

We must build multimedia prototype systems to begin characterizing the network's traffic, and experiment with communication software architectures and multimedia user interfaces. We hope to build a common, multimedia platform cooperatively with our colleagues at other research institutions, on top of which we will build tools for scientific visualization, cooperative problem solving, and very large database query processing.

On the analytical front, we plan to investigate routing and channel establishment algorithms that efficiently meet the requirements of gigabit networks. The channel establishment algorithm must route a channel's various text, graphic, video, and audio portions, and offer a discrete or continuous spectrum of service grades to meet the traffics' requirements efficiently. While telephony employs statistical multiplexing because telephone channels have been precisely characterized, and permissible telephonic loss rates are sufficiently high to yield efficient channel utilization, we do not yet know enough about multimedia interfaces to characterize the traffic's properties.

6.3. Conclusions

Flow control for internetwork multicast is a natural extension of the backoff algorithm developed in Chapter 3. Rate-based flow flow control for gigabit, virtual circuit networks is an extension of the minimum cost method applied to RPC retransmission timeouts in Chapter 4.

CHAPTER 7

CONCLUSIONS

We have examined three problems that benefit from adaptive backoff: cache invalidation storms due to contention for spin-locks, buffer overflow due to LAN multicast, and remote procedure call retransmissions due to lost messages and slow or congested servers. We have used a similar approach to each of these problems. In this concluding chapter we review and evaluate our research contributions and their relevance.

7.1. Relevance

Since limited buffer memory leads to buffer overflow and poor hardware synchronization support leads to cache invalidation storms, does this mean that our solutions to these problems have only historical relevance? On one hand, fantastically large memories and state of the art synchronization instructions will mitigate or eliminate these problems from the next generation of computer systems. On the other hand, current computer systems suffer from these problems, and benefit from our solutions. The bottom-end process control computers found in factories may survive ten or thirty years, and, if Birman is right [11], they may communicate by LAN multicast. Since a multiprocessor system's lifetime probably ranges from three to five years, and manufacturers are still selling them without additional synchronization support, our spin-lock backoff algorithm may be important for the coming decade.

7.2. Contributions

Chapter 2's contributions are a spin-lock backoff algorithm and a spin-lock based barrier synchronization algorithm that reduce memory contention in shared-memory multiprocessors with snoopy, invalidation-based caches. Our spin-lock algorithm loses slightly fewer cycles to memory contention and reduces the number of memory cycles left idle due to excessive backoff by twenty to fifty percent over a recently published algorithm [4]. Our spin-lock based barrier synchronization algorithm outperforms the published spin-lock based solutions by a factor of three.

Chapter 3's key contributions are its multiple round multicast timeout calculations and its UDP service time model. With limited buffers, service preemptions exacerbate buffer overflow because they extend the effective service time. Composing the service time distribution as a sum of a constant and a hyperexponential distribution models the constant minimum time through the code and the variations in the service time caused by device interrupts, scheduling latency, and memory contention.

Chapter 4 investigates the relationship between network service grade and protocol retransmission timeouts, and applies our approach to RPC retransmission timeouts. We anticipate that tariff-bearing, selectable service grade, long-haul network will eventually dominate the current Internet. We develop a method to simultaneously calculate protocol retransmission timeouts and select network service grade, and propose an adaptive retransmission timeout algorithm that behaves properly on lossy networks.

Chapter 5, a byproduct of the multicast work, makes both technical and methodological contributions. It documents the microsecond resolution timer board that we used to measure timing data and verify Chapter 3's model of UDP buffer overflow, and it derives the relationship between clock resolution, measurement precision, and degree of confidence. Our derivation leads us to speculate that few researchers perform enough measurements to report their performance data with a high degree of confidence.

7.3. Future Directions

In the future, we would like to extend our LAN result to internetwork multicast, instrument Internet gateways in a manner similar to the way we instrumented the UDP protocol to derive a more accurate model of the arrival process and protocol service times, build multimedia interfaces to characterize the network traffic that they generate, and investigate flow control and routing algorithms for virtual circuit, gigabit networks.

REFERENCES

1. AMD, The SUPERNET Family for FDDI, 1989.
2. Acampora, A. S. and Karol, M. J., "An Overview of Lightwave Packet Networks", *IEEE Networks 3*, 1 (Jan., 1989), 29-41.
3. Agarwal, A. and Cherian, M., "Adaptive Backoff Synchronization Techniques", *International Computer Architecture Conference*, Jerusalem, June, 1989, 396-406.
4. Anderson, T. E., Lazowska, E. D. and Levy, H. M., "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors", *1989 ACM SIGMETRICS Conference*, Berkeley, CA, May 23-26, 49-60.
5. Anderson, D., "A Software Architecture for Network Communication", *8th International Conference on Distributed Computing Systems*, June, 1988, 376-383.
6. Anderson, T. E., "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors", *Submitted to IEEE Transactions on Parallel and Distributed Systems*, 1989.
7. Anderson, T. E., "The Performance Implications of Spin-Waiting Alternatives for Shared-Memory Multiprocessors", *1989 ICPP II* (Aug., 1989), 170-174.
8. Avriel, M., *Nonlinear Programming: Analysis and Methods*, Prentice Hall, Englewood Cliffs, NJ, 1976.
9. Beilner, H., "Measuring with Slow Clocks", *ICSI-Technical Report-88-003*, Berkeley, California, July, 1988.
10. Bernstein, P. A., Hadzilacos, V. and Goodman, N., *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, MA, 1987.
11. Birman, K. P. and Joseph, T. A., "Reliable Communication in the Presence of Failures", *Trans. Computer Systems 5*, 1 (Feb., 1987), 47-76.
12. Birrel, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls", *Trans. Computer Systems 2*, 1 (Feb., 1984), 39-59.
13. Boggs, D. R., "Internet Broadcasting", *PhD. Dissertation, Stanford University*, 1982.
14. Cabrera, L. F., Hunter, E., Karels, M. J. and Mosher, D. A., "User-Process Communication Performance In Networks of Computers", *IEEE Trans. on Software Eng. 14*, 1 (Jan., 1988), 38-53.
15. Carter, J. B. and Zwaenepoel, W., "Optimistic Implementation of Bulk Data Tranfer Protocols", *1989 ACM SIGMETRICS Conference* , Berkeley, CA, May 23-26, 1989, 61-69.
16. Chang, J. and Maxemchuk, N. F., "Reliable Broadcast Protocols", *Trans. Computer Systems 2*, 3 (Aug., 1984), 251-273.
17. Cheriton, D. R. and Zwaenepoel, W., "Distributed Process Groups in the V Kernel", *Trans. Computer Systems 3*, 2 (May, 1985), 77-107.
18. Clark, D., "The Structuring of Systems Using Upcalls", *Tenth Symp. on Operating System Prin., Operating System Reviews 19*, 5 (Dec., 1985), 171-180.
19. Clark, D. D., Lambert, M. L. and Zhang, L., "NETBLT: A High Throughput Transport Protocol", *1987 ACM SIGCOMM Conference*, Aug., 1987, 353-359.

20. Dahlquist, G. and Bjorck, A., *Numerical Methods*, Prentice Hall, Englewood Cliffs, NJ, 1974.

21. DeWitt, D. J., "Gamma Database Machine", *Key Note Speaker, 1988 ACM SIGMETRICS Conference*, Santa Fe, New Mexico, May 24-27, 1988.

22. Deering, S., "Multicast Routing in Internetworks and Extended LANs", *1988 ACM SIGCOMM Symposium*, Aug. 16-19, 1988, 55-64.

23. Deering, S., "Host Extensions for IP Multicasting", *RFC 1112, Network Working Group*, Aug., 1989.

24. Dinning, A., "A Survey of Synchronization Methods for Parallel Computers", *Computer 22*, 7 (July 1989), 66-77, IEEE.

25. Dubois, M., Scheurich, C. and Briggs, F. A., "Synchronization, Coherence, and Event Ordering in Multiprocessors", *IEEE Computer 21*, 2 (Feb., 1988), 9-21.

26. Feinler, E. J., Jacobsen, O. J., Stahl, M. K. and Ward, C. A., *DDN Protocol Handbood*, Defense Communications Agency, Dec., 1985.

27. Feller, W., *An Introduction to Probability Theory and Its Applications, Vol. 2*, John Wiley & Sons, New York, NY, 1970.

28. Feller, W., *An Introduction to Probability Theory and Its Applications, Vol. 1*, John Wiley & Sons, New York, NY, 1970.

29. Ferrari, D., "Real-Time Communication in Packet-Switching Wide-Area Networks", *International Computer Science Institute Technical Report 89-002*, May, 1989.

30. Feynman, R. P. and Leighton, R., " *What Do You Care What Other People Think?" Further Adventures of a Curious Character*, Norton, 1988.

31. Fraser, A. G., Address Given at the XUNET Workshop, AT&T Bell Laboratories, Jan. 16-17, 1989.

32. Ganz, A. and Chlamtac, I., "Queueing Analysis of Finite Buffer Token Networks", *1988 ACM SIGMETRICS Conference*, Santa Fe, New Mexico, May 24-27, 1988, 30-36.

33. Gechter, J. and O'Reilly, P., "Conceptual Issues for ATM", *IEEE Network 3*, 1 (Jan., 1989), 14-16.

34. Goldstein, B., Personal Communication, Apr., 1989.

35. Goodman, J., Greenberg, A. G., Madras, N. and March, P., "Stability of Binary Exponential Backoff", *J. ACM 35*, 3 (July, 1988), 579-602.

36. Gopal, G. and Wong, J. W., "Delay Analysis of Broadcast Routing in Packet-Switching Networks", *IEEE Trans. on Computers c-30*, 12 (Dec., 1981), 915-922.

37. Gusella, R. and Zatti, S., "An Election Algorithm for a Distributed Clock Synchronization Program", *6th International Conference on Distributed Computing Systems*, Boston, May, 1986, 464-371.

38. Gusella, R., "The Analysis of Diskless Workstation Traffic on an Ethernet", *Computer Science Division, EECS, University of California Berkeley*, Berkeley, CA Technical Report 87/379, Nov., 1987.

39. Harita, B. R. and Leslie, I. M., "Dynamic Bandwidth Management of Primary Rate ISDN to Support ATM Access", *1989 ACM SIGCOMM Symposium*, Austin, Texas, September 19-22, 1989, 197-210.

40. Haskin, R., Malachi, Y., Sawdon, W. and Chan, G., "Recovery Management in QuickSilver", *Trans. Computer Systems 6*, 1 (Feb., 1988), 82-108.

41. Herlihy, M. P., "A Quorum-Consensus Replication Method for Abstract Data Types", *Trans. Computer Systems 4*, 1 (Feb., 1986), 32-53.

42. Kleinrock, L., *Queueing Systems Volume 1*, John Wiley & Sons, New York, NY, 1975.

43. Lamport, L. and Melliar-Smith, P. M., "Synchronizing Clocks in the Presence of Faults", *J. ACM 32*, 1 (Jan., 1985), 52-78.

44. Leiner, B., "Critical Issues in High Bandwidth Networking", *RFC 1077*, Nov., 1988.

45. Miller, M. J. and Lin, S., "The Analysis of Some Selective-Repeat ARQ Schemes with Finite Receiver Buffer", *IEEE Transactions on Communications COM-29*, 9 (Sep., 1981), 1307-1315.

46. Moss, J. E. B., *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, Cambridge, MA, 1985.

47. Noe, J. D. and Andreassian, A., "Effectiveness of Replication in Distributed Computer Networks", *7th International Conference on Distributed Computing Systems*, Berlin, 1987, 508-513.

48. Oskouy, R. M., Kyllonen, E. and Lau, H., *Local Area Network Controller Am7990 (LANCE) Technical Manual*, Advanced Micro Devices , 1986.

49. Paris, J. and Long, D. D. E., "Available Copies Algorithms", *Submitted to Performance Evaluation*, 1988.

50. Perros, H. G. and Altiok, T., "Approximate Analysis of Open Networks of Queues with Blocking: Tandem Configurations", *IEEE Trans. on Software Eng. SE-12* , 3 (Mar., 1986), 450-462.

51. Renessee, R., Staveren, H. and Tanenbaum, A. W., "Performance of the World's Fastest Distributed Operating System", *Operating System Reviews 22*, 4 (Oct., 1988), 25-34.

52. Ross, S. M., *Stochastic Processes*, John Wiley & Sons, New York, NY, 1983.

53. Ross, F. E., "FDDI Tutorial", *IEEE Communications Magazine*, May, 1986, 10-17.

54. Sequent, "The Symmetry Technical Summary", *Sequent Computer Systems, Inc.*, 1988.

55. Servi, L. D., "D/G/1 Queues with Vacations", *Operations Research 34*, 4 (July-August 1986), 619-629.

56. Shanthikumar, J. G., "On The Buffer Behavior with Poisson Arrivals, Priority Service, and Random Server Interruptions", *IEEE Trans. on Computers c-30*, 10 (Oct., 1981), 781-786.

57. Shanthikumar, J. G. and Ott, Personal Communication , 1989.

58. Shenker, S., "Some Conjectures on the behavior of acknowledgment based transmission control of random access communication channels.", *1987 ACM SIGMETRICS Conference*, Banff, Alberta, Canada, May 11-14, 1987, 245-255.

59. Simons, B. and Votta, L. G., "The Optimal Retry Distribution for Lightly Loaded Slotted Aloha Systems", *IEEE Transactions on Communications COM-33*, 7 (July, 1985), 724-725.

60. Spector, A. Z., Daniels, D., Duchamp, D., Eppinger, J. L. and Pausch, R., "Distributed Transactions for Reliable Systems", *Tenth Symp. on Operating System Prin., Operating System Reviews 19*, 5 (Dec., 1985), 127-146.

61. Spector *et al.*, A., "High Performance Distributed Transaction Processing in a General Purpose Computing Environment", Unpublished Technical Report, Department of Computer Science, CMU, Pittsburgh, PA, Sep. 9, 1987.

62. Tang, P. and Yew, P., "Processor Self-Scheduling for Multiple-Nested Parallel Loops", *1986 ICPP*, Aug., 1986, 528-535.

63. Theimer, M. M. and Lantz, K. A., "Finding Idle Machines in a Workstation-based Distributed System", *8th International Conference on Distributed Computing Systems*, San Jose, CA, June, 1988, 112-122.

64. Theimer, M., Cabrera, L. and Wyllie, J., "Quicksilver Support for Access to Data in Large, Geographically Dispersed Systems", *9th International Conference on Distributed Computing Systems*, Newport Beach, CA, June 5-9, 1989, 28-35.

65. Towsley, D. and Mithal, S., "A Selective Repeat ARQ Protocol for a Point to Multipoint Channel", *INFOCOMM 87*, 1987, 521-526.

66. Wall, D. W., "Mechanisms for Broadcast and Selective Broadcast", *Ph.D. Dissertation, Stanford University*, 1980.

67. Wang, J. L. and Silvester, J. A., "Throughput Optimization of the Adaptive Multi-Receiver Selective-Repeat ARQ Protocols Over Broadcast Links", *Proceedings of ICCC*, Tel Aviv, Israel, Oct., 1988.

68. Weinstein, M., Page, T., Livezey, B. and Popek, G., "Transactions and Synchronization in a Distributed Operating System", *Tenth Symp. on Operating System Prin., Operating System Reviews 19*, 5 (Dec., 1985), 115-126.

69. Weinstock, R., *Calculus of Variations*, McGraw-Hill, New York, NY, 1952.