A Distributed Scheduling Algorithm for Quality of Service Support in Multiaccess Networks

by

Craig Ian Barrack

S.B. in Mathematics, Massachusetts Institute of Technology (1996)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1998

Author .....
Department of Electrical Engineering and Computer Science
July 31, 1998

Certified by ....
Kai-Yeung Siu
Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

A Distributed Scheduling Algorithm for Quality of Service Support in Multiaccess Networks

by

Craig Ian Barrack

Submitted to the Department of Electrical Engineering and Computer Science
on July 31, 1998, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents a distributed scheduling algorithm for the support of quality of service in multiaccess networks. Unlike most contention-based multiaccess protocols which offer no quality of service guarantee and suffer the problems of fairness and low throughput at high load, our algorithm provides fairness and bandwidth reservation in an integrated services environment and at the same time achieves high throughput. Moreover, while most reservation-based multiaccess protocols require a centralized scheduler and a separate channel for arbitration, our algorithm is truly distributed in the sense that network nodes coordinate their transmissions only via headers in the packets. We derive theoretical bounds illustrating how our distributed algorithm approximates the optimal centralized algorithm. Simulation results are also presented to justify our claims.

Thesis Supervisor: Kai-Yeung Siu
Title: Assistant Professor

Acknowledgments

First and foremost, I would like to thank my thesis supervisor, Professor Kai-Yeung Siu, for his guidance and support in producing the research results that comprise this thesis. Over the last thirteen months, I have had many hours of helpful discussions with Professor Siu, and many of the ideas which arose in these meetings led me to strive for a tighter and more practical algorithm, with more useful theory at its foundation. I also want to thank Professor Siu for offering me funding during my months of working with him.

I am grateful to all of the members of Professor Siu's group at the d'Arbeloff Laboratory for Information Systems and Technology. In particular, Paolo L. Narvaez Guarnieri was always available for assistance with C, UNIX, LaTeX, and just about anything related to data networks. Anthony Kam helped me with the bursty traffic model, and provided me with insight into the performance metrics that matter in distributed multi-queue systems.

I want to thank Professors Charles E. Leiserson and Michel X. Goemans for allowing me to be a teaching assistant for the class Introduction to Algorithms, during the fall and spring terms of 1997. Teaching MIT juniors the art and science of algorithm design was one of my most enjoyable and fulfilling experiences during my six years at MIT. I learned a lot not only about algorithms, but also about public presentation and pedagogy from Professors Leiserson and Goemans.

Last and most of all, I would like to thank Mom, Dad, Jennifer, and Mr. Gray for their love and support.

Contents

Chapter 1

Multiaccess Communication

One of the most familiar and fundamental problems in the field of data communication networks is the problem of allocating a multiaccess channel among a set of nodes competing for its use. When most people think of data networks, they imagine nodes joined by point-to-point communication links. Each such link might consist physically of a pair of twisted wires, a coaxial cable, an optical fiber, or a microwave radio link. Developing efficient algorithms for detecting errors in transmitted data, calculating shortest paths, and controlling the flow of data traffic are some of the research aims in networks with point-to-point links. A good feature of point-to-point links is that the received signal on a link depends only on the transmitted signal on that link.

However, there are many practical communication media for which the received signal at one node depends on the transmitted signal from two or more nodes. In such media, the received signal is typically a weighted sum of the transmitted signals. These *multiaccess media* form the basis for local area networks, metropolitan area networks, satellite networks, and radio networks. In applications such as these, the summing effect of the multiaccess medium when two or more nodes attempt to transmit data simultaneously is undesirable. To efficiently use the medium, be it a satellite system, radio broadcast, multidrop telephone line, or multitap bus, we require a protocol that allocates the multiaccess medium among the nodes one at a time.

I first became interested in multiaccess communication in the fall of 1997, when I

was taking two classes: Distributed Algorithms and Data Communication Networks. In my class on distributed algorithms, I was intrigued by the fact that even for a simple protocol, that many processors execute the code in parallel with steps interleaved in some undetermined way implies that there are many different ways that the algorithm can behave. Because knowledge about the global state of the system is distributed, special problems of communication and coordination arise. Some of the most easily stated and useful problems — for example, getting a network of m distributed processors to agree — are either provably impossible or else require high time and message complexity to solve. I studied the mutual exclusion problem, in which access to a single, indivisible resource that can only support one user at a time must be arbitrated.[1] The problem of handling a sequence of user requests for a single, shared resource reminded me of the related problem with multiaccess media.

In my class on data networks, I studied existing multiaccess communication algorithms. Although a plethora of strategies for multiaccess communication has been proposed, I found most of these protocols to be ad hoc and intellectually unsatisfying. Many protocols adopt a "free-for-all" approach, in which nodes send data along the multiaccess channel whenever they wish, hoping for no interference from other nodes [4]. When interference does occur, the data needs to be retransmitted. Because few performance guarantees can be made for such protocols, I was more curious about "perfect scheduling" algorithms, in which interference cannot occur, because nodes receive reserved intervals for channel use. Unfortunately, I was disappointed to see that existing scheduling algorithms typically polled the nodes in cyclic order, allowing packets to be transmitted as soon as they were discovered by the scheduler. Little emphasis was placed on reducing packet delay and increasing channel utilization through intelligent, dynamic scheduling, in which the multiaccess communication algorithm adjusts to the location of the packets waiting in the system.

I soon began thinking about inventing a new distributed scheduling algorithm for multiaccess communication. My new multiaccess communication algorithm, the

[1]For example, the mutual exclusion problem arises in a computer cluster with a single printer. Mutual exclusion will be discussed further in Section 1.2.1.

GlobalTime algorithm, is the topic of this thesis, and as we will see in coming chapters, much theory can be set forth describing *GlobalTime*'s performance. In this chapter, we introduce the multiaccess communication problem. In Section 1.1, we discuss the model for multiaccess communication that will be used in this thesis. Section 1.2 explores some previous work in the field of multiaccess communication. In Section 1.3, we describe the chapter layout of the remainder of this thesis.

1.1 Model

We have stated that managing access to a multiaccess channel among a set of m nodes competing for its use is a fundamental problem in the field of data communication networks. In this *multiaccess communication problem*, we have m nodes $1, 2, \ldots, m$, each connected to a multiaccess channel C. Each node can be considered a *user* in the multiaccess network. A data packet that arrives at node i is inserted into a first-in-first-out (FIFO) queue at i, where the packet waits until it gains access to C, so that it can be transmitted to its destination. We assume that the time required to transmit a data packet is the same for all packets. The job of a *multiaccess communication algorithm* is to fairly allocate opportunities to transmit packets among the network's users.

In our multiaccess communication model, the single channel C is shared; every node can read the data sent by every other node. If multiple nodes attempt to transmit packets simultaneously, none of the attempts succeed. Moreover, any communication among users takes place along the single channel C. In particular, there is no separate control channel, and no centralized scheduler with global knowledge coordinating the users. Knowledge about the state of the queues is distributed among the nodes of the system. For simplicity, we assume zero propagation delay; that is, no time passes from the instant a bit is transmitted from the source node to the instant the same bit is received by all other nodes.[2] Finally, we assume that the nodes and

[2]In a local area network (LAN), this assumption is valid. Our work can be extended to a model with nonzero propagation delay between nodes, but in this thesis, unless otherwise noted, we assume zero propagation delay.

channel are fault-free.

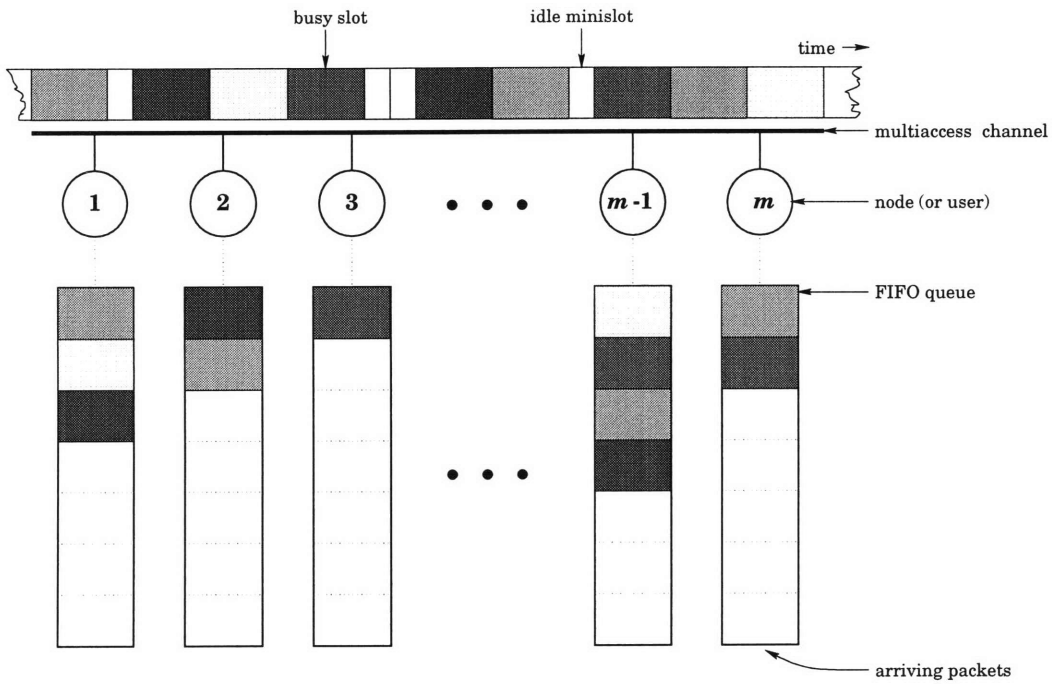


Figure 1-1: An illustration of the model for multiaccess communication used in this thesis.

Figure 1-1 illustrates the model for multiaccess communication that we employ in this thesis. Observe in the figure that the only link between the m separate users is the multiaccess channel. Although our focus in this thesis is a protocol for coordinating the users in which the multiaccess channel will serve as the sole means of communication, we should not forget that the channel is also a causeway for packets to move from source to destination. Ironically, we will never again refer to any packet's destination in this thesis, because destinations are irrelevant to coordinating channel access among the users. Transmitting packets to their destinations is simultaneously the whole point of the multiaccess communication problem, yet not the point at all.

Also, notice in Figure 1-1 that if the channel is idle, only a small fraction of the time required to transmit a packet need be wasted. This small period of wasted time is called an *idle minislot*. In practice, an idle minislot can be as small as 1% of the length of a busy slot in which an actual packet is transmitted.

We say that a distributed algorithm solves the multiaccess communication problem if it guarantees eventual transmission of all packets that enter the system. Though nearly any strategy one might think of can be said to "solve the multiaccess communication problem," our concern is algorithm performance. In our model, we are concerned with two performance metrics in particular:

Throughput How much data is transmitted in a given interval of time?

Delay How long does a packet have to wait for transmission?

With respect to delay, our focus will be geared toward delay variance and worst case delay rather than mean delay, because in a setting where users pay to receive performance guarantees, a low delay bound is more important than low mean delay [15]. High delay variance requires larger packet buffers and greater timing flexibility at the receiving end. With respect to throughput, our focus will be geared toward *maximum stable throughput.* By maximum stable throughput, we mean the maximum aggregate traffic arrival rate that can be supported by the system. In order for throughput to be stable, the algorithm must guarantee that packets will be transmitted at the given rate without queue sizes growing unbounded.

Throughout this thesis, we use a consistent set of notation. Let λ be the aggregate arrival rate to the system, normalized so that $\lambda = 1$ represents maximum utilization.[3] Let m be the number of users. We define β to be the length of an idle minislot. Finally, we define n as the length of a sequence of transmitted packets. Typically, when we analyze the performance of a multiaccess communication algorithm, we consider the output transmission sequence in the limit as n approaches infinity.

[3] If $\lambda > 1$, then the arrival rate exceeds the service rate, which is clearly unsustainable.

1.2 Background

1.2.1 The mutual exclusion problem

The multiaccess communication problem is related to the more general *mutual exclusion problem*, a problem of managing access to a single, indivisible resource that can only support one user at a time. Alternatively, the mutual exclusion problem can be viewed as the problem of ensuring that "certain portions of program code are executed within critical regions, where no two processes are permitted to be in critical regions at the same time" [8]. A central difficulty is that a user does not know which other users are going to request the resource nor when they will do so.

The mutual exclusion problem is qualitatively somewhat different from the multiaccess communication problem. In the mutual exclusion problem, the m users can communicate their needs via messages along point-to-point links, and the shared resource might be an output device or database. In the multiaccess communication problem, however, the users must communicate their needs via messages along a broadcast channel, yet the desired exclusive-access resource is this same channel. In other words, the multiaccess channel that the users require in order to transmit their packets also serves as the medium of communication and coordination. In order to request access to the channel, a user requires a priori channel access to place the request.

A distributed algorithm solves the mutual exclusion problem if the following two conditions are satisfied:

1. **Mutual exclusion** There is no reachable system state in which more than one user has access to the resource.

2. **Progress** If at least one user requests the resource and no user currently has it, then eventually some user will gain access to the resource.

A correct algorithm for mutual exclusion need not be "fair," in the sense that one user may hoard the resource, thus depriving other users. Therefore, we often desire

our mutual exclusion algorithms to satisfy a stronger fairness condition than ordinary progress:

3. **Lockout freedom** If all users always return the resource, then any user that requests the resource will eventually receive it.

In Chapter 4, we will return to the issue of lockout freedom, when we modify our original algorithm to support guaranteed quality of service; we will need to ensure that no user can flood the system with his own packets, locking out all other users.

Many algorithms have been proposed for mutual exclusion; [8] provides a recent survey of the area. Dijkstra's mutual exclusion algorithm (1965) requires at most $O(m)$ time between the time of a user's request and its access to the resource, where m is the number of users, but the algorithm is not lockout-free. Peterson's algorithm (1981) was among the first to guarantee lockout freedom, but its time complexity is exponential in the worst case. Lamport's algorithm (1986) guarantees lockout freedom with $O(m^2)$ time complexity.

1.2.2 Contention-based algorithms

Strategies for addressing the multiaccess communication problem have traditionally fallen into two categories. Many proposed strategies have been *contention-based* schemes, in which nodes greedily attempt to send new packets immediately. If two or more nodes attempt to transmit packets simultaneously, then a *collision* results, and none of the attempts succeed. Contention-based strategies generally require backlogged nodes to repeatedly reattempt transmission at random intervals until one node gains sole access to the channel.[4] Such algorithms satisfy the eventuality condition for multiaccess communication — that is, the condition that states that a correct algorithm must guarantee eventual transmission of all entering packets — with high probability, though not with certainty.

[4]The retransmission interval is random in order to prevent deadlock caused by the packets of contending users colliding every attempt.

In contention-based schemes, under heavy loading, collisions among contending packets reduce system throughput and increase packet delay. Furthermore, such schemes waste time on idle slots as well; because the retransmission interval is random, it is possible for channel bandwidth to go unused even if there are packets in the system waiting for transmission. Few quality of service guarantees can be made for contention-based schemes.

A good survey of contention-based algorithms for multiaccess communication is provided in [4] and [6]; see also [7], [10], and [12] for clear discussion. The Aloha system (1970) produces a maximum stable throughput of 0.1839, with delay substantially lower than other algorithms when $\lambda < 0.1839$. If nodes can detect and terminate idle periods and collisions quickly, a technology called carrier sense multiple access/collision detection (CSMA/CD), throughput can be improved further. CSMA/CD, also the protocol of the Ethernet (1976), is a popular example of the use of collision resolution to make implicit reservations. If a collision occurs, the transmitting node stops after one minislot; therefore, the first minislot of a packet's transmission initiates an implicit reservation for the rest of its transmission. The maximum stable throughput of the Ethernet is $[1 + \beta(e - 1)]^{-1}$, where β is the length of a minislot. Additional contention-based schemes include splitting algorithms (1978), in which the set of colliding nodes is split into subsets, one of which transmits in the next slot. If the collision is not resolved, then a further splitting into subsets occurs. [6] provides an overview of the work on splitting algorithms. The best splitting algorithms achieve a maximum stable throughput of 0.4878, with lower expected delay over a larger range of λ than Aloha.

1.2.3 Reservation-based algorithms

The second category of algorithm that has been proposed to solve the multiaccess communication problem is *reservation-based*, in which nodes receive reserved intervals for channel use. Here, in order to reserve a time slot for use of the channel, a node needs to communicate its need to the remaining nodes. Generally, such systems alternate short *reservation intervals* giving each node an opportunity to reserve a time

slot (using round robin or a contention-based approach), with longer *data intervals* where actual packet transmissions take place.

Existing reservation-based schemes generally guarantee few interesting delay properties, primarily because they typically follow a strict cyclic order in serving the users. In other words, the actual waiting time in queue is ignored; the last packet of a large data burst arriving in queue may have to wait a very long time to be transmitted.

A survey of reservation-based algorithms is presented in [4]; other worthwhile references on the subject are [3] and [5]. Most existing reservation-based schemes are variations on time-division multiplexing (TDM), in which users are given opportunities to transmit in cyclic order. In the *gated* system (1972), the rule is that only those packets that arrived prior to the user's turn are transmitted. By contrast, in the *exhaustive* system, those packets that arrive during the user's turn are transmitted as well. A variation on the gated system is *gated limited service* (1978), in which only the first packet waiting in queue is transmitted on a user's turn. Queuing delay analysis has been performed on all three of these reservation-based variations, and the interested reader is referred to [4]. Token rings (1969) constitute another popular approach to reservation-based multiaccess communication. The nodes are arranged logically in a ring, with each node transmitting to the next node around the ring. When a node completes its communication, it sends a "token." The next node around the ring either transmits its own data before forwarding the token, or if it has no data to send, relays the token to the next node. The maximum stable throughput of the token ring is 1, although no guarantees can be made regarding worst case delay.

1.3 Outline of thesis

This thesis presents a new distributed scheduling algorithm for quality of service support in multiaccess networks. In Chapter 2, we introduce a new concept called an inversion, and we apply it in conjunction with classical machine-job scheduling principles to inventing what we name the *GlobalTime* algorithm. Chapter 2 also states and proves two delay and throughput theorems which guarantee the theoretical per-

formance of our algorithm. In Chapter 3, we simulate the *GlobalTime* algorithm over time on random incoming traffic satisfying two different probability distributions. We also compare the performance of the *GlobalTime* algorithm to the performance of other reservation-based disciplines and the optimal centralized scheduler. In Chapter 4, we extend the *GlobalTime* algorithm to handle bandwidth reservation in an integrated services environment. The key innovation in Chapter 4 will be the notion of virtual timestamp, which will enable users that "follow the rules" to receive higher priority for channel access than overzealous users that attempt to "hog" the channel by flooding the system with their own packets. In Chapter 5, we summarize the results of the thesis.

Chapter 2

The *GlobalTime* Algorithm

In Chapter 1, we introduced the multiaccess communication problem, a fundamental problem in the field of networking. We outlined several proposed strategies for tackling this problem, from contention-based schemes like Aloha to reservation-based schemes like the token ring. Unfortunately, as we have seen, attempts at solving the multiaccess communication problem to date have been far from perfect. In contention-based schemes, collisions among contending packets drastically reduce system throughput and increase packet delay. Furthermore, existing reservation-based schemes rarely guarantee interesting delay or fairness properties. For example, under many existing reservation-based algorithms, no attempt is made to equalize the delay experienced by each user. A recently arriving packet at node i will be served quickly if i's queue size is small; meanwhile, a packet that arrived earlier at backlogged node j may have to wait a much longer time.[1]

In this chapter, we introduce a new reservation-based algorithm for multiaccess communication. In Section 2.1, we discuss the related problem of machine-job scheduling. In Section 2.2, we define the concept of an inversion, and we consider its application as a measure of the deviation of a transmission sequence from the optimal

[1]We consider fairness from the packet's perspective, in much the same way that we measure fairness in the checkout lines of a supermarket from the shopper's perspective. Fairness to the packet is equivalent to fairness to the user if we assume that users generate packets at or below their reserved rates. For example, if one user floods the system with packets, other users will be locked out, despite the system's effort to treat all packets equitably. Fairness will be discussed further in Chapter 4.

sequence. Section 2.3 introduces the main result of this thesis, a timestamp-based multiaccess communication protocol called the *GlobalTime* algorithm. In Sections 2.4 and 2.5, we analyze the theoretical performance of the *GlobalTime* algorithm with respect to throughput and worst case delay.

2.1 Machine-job scheduling

Recall our framework of m nodes $1, 2, \ldots, m$, each with an associated first-in-first-out (FIFO) queue, and each connected to a single multiaccess channel. A data packet that arrives at node i waits in the queue at i until it gains access to the channel, and can be successfully transmitted to its destination. The framework of the multiaccess communication problem is inherently on-line and distributed; users must decide in real time whether to attempt packet transmissions, and must do so without the coordinative assistance of a centralized scheduler.

To gain insight into this problem, we examine the related offline, centralized problem. In this variation, a system overseer schedules the packet transmissions of all the users in the order he sees fit. Furthermore, the overseer possesses knowledge of the future; he may schedule packet transmissions with a protocol that requires information about packets yet to arrive. The reason we examine this variant of the multiaccess communication problem is that we hope a solution to the offline, centralized problem provides needed intuition for the solution of the on-line, distributed one.

It turns out that the multiaccess communication problem in its offline, centralized incarnation is just the machine-job scheduling problem, a well-researched topic in the dynamic programming and discrete optimization literature. In machine-job scheduling, there are n equal length jobs $\mathcal{J} = J_1, J_2, \ldots, J_n$ to be scheduled on a single machine.[2] Job J_i becomes available at *release time* r_i.[3] Let the *completion time* of job J_i in schedule S be t_i^S, and define the *cost* of schedule S to be

[2]The "length" of a job is the time required for the machine to complete it.

[3]A release time might correspond in practice to a time at which an earlier task is completed (e.g. lumber must be sawed before it can be sanded), or at which required supplementary resources become available (e.g. a program cannot run until the necessary disk has been obtained).

$$C(S) = \max_{j \in \mathcal{J}} \left(t_j^S - r_j \right) \ .$$

Notice that the cost of a schedule is simply the waiting time between the release and the completion of the job whose waiting time is the longest. The *machine-job scheduling problem* is to find a feasible, cost-minimizing schedule for performing the n jobs on the single machine.

The machine-job scheduling problem has an optimal solution, described and proved in Theorem 2.1.

Theorem 2.1 *An algorithm A in which jobs are processed in order of increasing release times is cost-minimizing; that is,*

$$C(A(\mathcal{J})) = \min_S C(S) \ .$$

Proof: Let S be a cost-minimizing schedule such that jobs are not processed in order of increasing release times. Let J_a and J_b be two consecutive jobs in S such that $r_a > r_b$. We now argue that by transposing J_a and J_b in the task sequence, the cost of the new schedule can be no greater than $C(S)$, producing a contradiction. Without loss of generality, we assume unit-length jobs. Because $r_a > r_b$, we have

$$t_b^S - r_b > t_b^S - r_a \ . \tag{2.1}$$

Furthermore, because $t_a^S < t_b^S$ by assumption, we have

$$t_b^S - r_b > t_a^S - r_b \ . \tag{2.2}$$

Combining inequalities (2.1) and (2.2), we obtain

$$\max(t_a^S - r_b, t_b^S - r_a) < t_b^S - r_b < \max(t_a^S - r_a, t_b^S - r_b) \ , \tag{2.3}$$

which shows that the new schedule can cost no more than the original. □

This theorem certainly matches our intuition, proving that completing the jobs in order of increasing release times minimizes waiting time. In the next section, we will see how Theorem 2.1 provides motivation for a multiaccess communication algorithm based on machine-job scheduling principles.

2.2 Inversions

In Section 2.1, we saw that the machine-job scheduling problem is an offline, centralized variant of the multiaccess communication problem, where jobs take the place of data packets, and the machine takes the place of the multiaccess channel. We also proved a theorem that stated that serving the jobs in the order they are released minimizes the worst case wait.

Because of the correspondence between machine-job scheduling and multiaccess communication, Theorem 2.1 provides a benchmark to which on-line, distributed algorithms that solve the multiaccess communication problem can aspire. Though the optimal solution to the multiaccess communication problem may be unachievable in a distributed, real time environment, the machine-job scheduling bound provides the algorithm designer with a reasonable goal. Indeed, Theorem 2.1 suggests that the more closely the sequence of transmissions generated by a distributed algorithm approximates first-come-first-served (FCFS), the better the algorithm performs with respect to worst case delay.[4] To quantify this notion, we introduce the concept of an inversion.

Definition 2.1 *Given n packets $P_1, P_2, \ldots, P_n$, where packet P_i arrived in the system at time a_i and was transmitted at time t_i, we say two packets P_i and P_j are* **inverted** *if $t_i < t_j$ but $a_i > a_j$.*

In other words, two packets are inverted in a transmission sequence if they "should

[4]This is a practical observation, because worst case delay (or a guaranteed "delay bound") is the most important measure of a user's satisfaction — in particular, of what the user is willing to pay.

have" been transmitted in the opposite order according to the FCFS criterion. Inversions enable the algorithm designer to simply count how far off from optimal his multiaccess communication algorithm is on a given input.

Before we overstate the case for inversions as a measure of worst case delay, we need to be careful. It is not strictly correct to claim that 43 inversions are worse than 42, or more generally, that "the more inversions in a transmission sequence, the higher its worst case delay." However, the asymptotic number of inversions, calculated as the number of packet transmissions approaches infinity, provides a good measure of the algorithm's delay performance. By dividing this asymptotic number of inversions by the number of packet transmissions n, we obtain a quantity representing the average displacement of a packet from its rightful FCFS spot in the transmission sequence. If as n approaches infinity, this amortized quantity approaches $\Omega(n)$, then the average packet is unfairly displaced by a substantial fraction of its peers; thus, delay variance (and all higher moments of delay) suffer. On the other hand, if the average packet is unfairly displaced by only $O(1)$ other packets, then the output only differs from the optimal FCFS order within small local neighborhoods of the transmission sequence; thus, the higher moments of delay are kept small. As we have already observed, ultimately it is these higher moments of delay that matter most to paying customers, because higher variation in packet delay requires larger packet buffers and greater timing flexibility in receiving end protocols. In fact, we can prove that under special circumstances, mean packet delay is unaffected by the particular choice of multiaccess communication algorithm.

Theorem 2.2 *Consider any two algorithms A and B that solve the multiaccess communication problem, and suppose that the channel is fully utilized. Then on any input, A and B produce transmission sequences with the same mean delay.*

Proof: We consider the transmission sequences $A(i)$ and $B(i)$ generated by algorithms A and B on input i. If $A(i) = B(i)$, then we are done. If not, then there exist two packets p and q such that $t_p^{A(i)} < t_q^{A(i)}$, but $t_q^{B(i)} < t_p^{B(i)}$. Without loss of

generality, we consider the case in which there is only one such pair.[5]

Suppose that the mean delay of sequence $A(i)$ is $\mu_{A(i)}$; thus, the *total* delay of sequence $A(i)$ is $n\mu_{A(i)}$. But the total delay $n\mu_{B(i)}$ of sequence $B(i)$ is

$$\begin{aligned} n\mu_{B(i)} &= n\mu_{A(i)} - (t_p^{A(i)} - a_p) - (t_q^{A(i)} - a_q) + (t_p^{B(i)} - a_p) + (t_q^{B(i)} - a_q) \\ &= n\mu_{A(i)} - t_p^{A(i)} - t_q^{A(i)} + t_p^{B(i)} + t_q^{B(i)} \\ &= n\mu_{A(i)} \ . \end{aligned}$$

Thus, A and B produce transmission sequences with the same total delay, and hence the same mean delay. □

Although Theorem 2.2 no longer strictly holds when the channel is not fully utilized, the spirit of the theorem is still correct. In particular, under high throughput conditions, the order in which packets are transmitted hardly affects mean packet delay. The intuition is that transposing two packets in the transmission sequence simply reduces the delay of one packet at the expense of the other, the effects canceling each other on average. Therefore, we apply the concept of inversion to measure worst case delay, a performance metric that is sensitive to choice of algorithm.

To our knowledge, existing schemes that solve the multiaccess communication problem guarantee nothing about the worst case number of inversions. Certainly, any of the contention-based schemes described in Chapter 1 cannot guarantee anything, nor can any of the canonical reservation-based protocols. In fact, existing algorithms generate $\Omega(n^2)$ inversions in the worst case, where n is the number of packet transmissions. Theorem 2.3 demonstrates this property for a gated limited service or "round robin" protocol.

Theorem 2.3 *Round robin generates a sequence of transmissions with $\Omega(n^2)$ inversions in the worst case.*

[5]If there exists more than one pair of packets p and q such that $t_p^{A(i)} < t_q^{A(i)}$ but $t_q^{B(i)} < t_p^{B(i)}$, then the proof of Theorem 2.2 can be applied repeatedly to each such pair to easily obtain the same result.

Proof: Consider the sequence of packets $\mathcal{P} = \langle P_1, P_2, \ldots, P_n \rangle$, listed in the order they were transmitted. Also, consider the associated sequence of arrival times $\mathcal{A} = \langle a_1, a_2, \ldots, a_n \rangle$. We claim $\mathcal{A}$ can have $\Omega(n^2)$ inversions in the worst case. Because there are m nodes (where $m > 1$), each with its own FIFO queue, the sequence $\mathcal{A}$ is guaranteed to be *m-ordered.*[6] Other than m-orderedness, there are no restrictions on $\mathcal{A}$, since the relative arrival times of packets at different nodes is unconstrained. It follows by induction on m and n that an m-ordered sequence can have as many as $\Omega(n^2)$ inversions. □

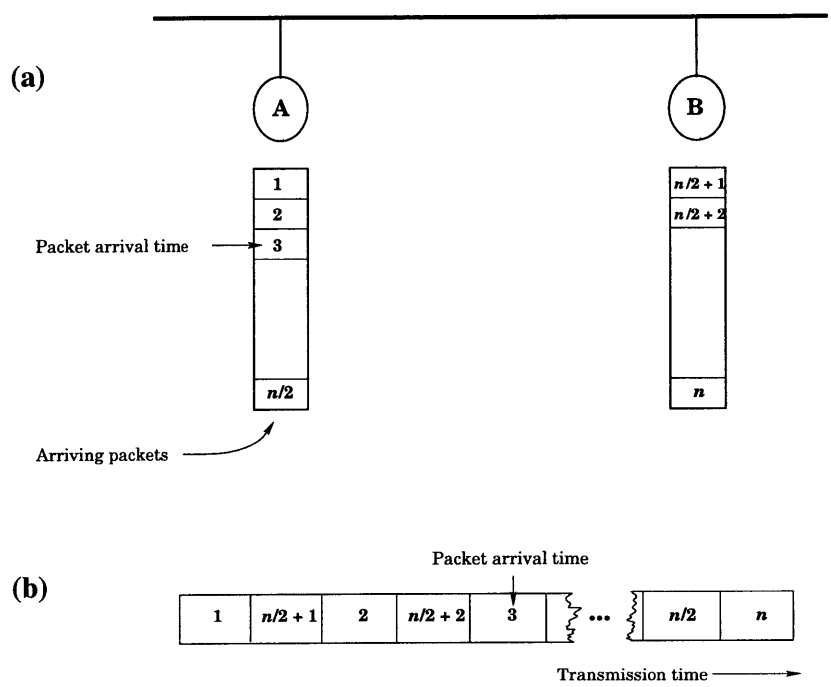


Figure 2-1: An illustration of the theorem in the case where $m = 2$. In (a), packets are labeled with their relative arrival times; all of node A's packets have arrived in the system before any of node B's. In (b), the packets are interleaved in the transmission sequence, producing $\Omega(n^2)$ inversions.

Figure 2-1 illustrates the idea behind the proof of Theorem 2.3 in the case where $m = 2$. Under a round robin protocol, the sequence $\mathcal{A}$ will be 1, $n/2+1$, 2, $n/2+$

[6]The term m-ordered is used in the sorting literature to mean that the m interleaved subsequences $\langle a_i, a_{i+m}, a_{i+2m}, \ldots \rangle$ for $i = 1, 2, \ldots, m$ are each sorted.

2, 3, ..., $n/2$, n. Therefore, the number of inversions in $\mathcal{A}$ is

$$\sum_{i=1}^{n/2}\left(\frac{n}{2}-i\right)=\sum_{i=0}^{n/2-1} i=\Omega(n^2)\ .$$

The example extends naturally to $m > 2$, simply by interleaving m subsequences rather than 2.

Observe that $\Omega(n^2)$ inversions signifies that on average, each packet is displaced from its optimal FCFS position by a length proportional to that of the entire transmission sequence. In theory, this makes sense, but as the system evolves in time, n approaches infinity. Does this mean the average packet will be delayed indefinitely? In practice, delays are finite because data traffic would have to be strikingly out of the ordinary for events like the one in Figure 2-1 to occur. On the other hand, the fact that delay is theoretically unbounded strongly suggests the need for a better algorithm.

2.3 The *GlobalTime* algorithm

In the previous section, we introduced the concept of an inversion as a way of measuring the deviation of a transmission sequence from the optimal sequence. We also implied the desirability of an algorithm that could guarantee a constant number of inversions per packet on average.

In this section, we produce a new algorithm called *GlobalTime*. As we will see, the *GlobalTime* algorithm generates a sequence of transmissions with fewer than mn inversions in the worst case — at most m per packet — where m is the number of users in the system. Thus, the total number of inversions is linear (not quadratic) in n, the number of packets transmitted. Indeed, the *GlobalTime* algorithm guarantees that packets are transmitted nearly in the order they arrived. This follows immediately from the fact that each packet is displaced from its correct FCFS position by at most m other packets, independent of n.

The *GlobalTime* algorithm works as follows:

1. Each node $1, 2, \ldots, m$ has a clock. The nodes' clocks are synchronized and are initialized to 0.

2. When a packet P arrives in the system, it is immediately timestamped with the current time. This timestamp records the arrival time of P.

3. Each node i has a set of m states *known-time*$[j]$, one for each node j. All the *known-time* states are initialized to 0.

4. At the beginning of a time slot, node i determines the value of j such that *known-time*$[j] \leq$ *known-time*$[k]$ for all k. If $i = j$, then node i has gained access to the channel. (Ties are broken according to predetermined rules.)

5. On its turn, node i transmits the first data packet in its queue. In addition to actual data, the transmitted frame also contains piggybacked information — namely, the *timestamp of the next packet* in node i's queue. However, if the transmission has exhausted node i's queue, the algorithm instead inserts the *current time* in this extra field. Finally, if on node i's turn, i has no packets to send at all, then i sends a dummy packet (augmented with the current time, as before).

6. When a packet P is being transmitted from node i, all nodes read the timestamp piggybacked on P, and then update their local value of *known-time*$[i]$.

The crucial idea behind the *GlobalTime* algorithm is that on their turn, users not only transmit the usual data, but also broadcast the arrival time of the packet waiting next in queue. Because all users maintain a table of the most recent *known-time* declaration by each other user, transmission can proceed in nearly FCFS order. Notice how the use of timestamp broadcast imposes a global notion of time on the system, making the name *GlobalTime* appropriate.

A perceived problem with piggybacking *known-time* declarations on data packets is that if a node has no data to send initially, it might never get an opportunity to tell the other users when it *does* have a packet to transmit. Indeed, any algorithm that

solves the multiaccess communication problem must give frequent opportunities to all users to reserve channel bandwidth; otherwise, the worst case result is user lockout. The *GlobalTime* algorithm avoids this problem by broadcasting the current time in lieu of a real packet's arrival time if a user has no next-in-line packet on its turn. Furthermore, this dummy timestamp is treated no differently from a real timestamp during the execution of *GlobalTime*, ensuring that the idle user will eventually get another turn. On the other hand, because the dummy timestamp carries the current time, we are guaranteed that all real data packets presently in the system will be cleared out before the idle user's next turn.[7]

Figure 2-2 shows a sample execution fragment of the *GlobalTime* algorithm in the case where packet transmission time is 5 units. At the left, the state of the queues (with arrival times of waiting packets) is depicted, and at the right, the common *known-time* table is shown. Notice that at $t = 40$, the head-of-line packet at node A (with arrival time 26) is transmitted, but there is no waiting next-in-line packet at A. Therefore, a dummy timestamp indicating the current time (in this case, $t = 40$) is sent, and is recorded in the state *known-time*$[A]$ at each node. The existence of dummy timestamps leads to a slight deviation from strict FCFS order in the transmission sequence. The arrival times of the packets in the order they are transmitted are 25, 26, 27, 29, 48, 47, 46. However, packets are displaced from their proper FCFS positions only by a few time slots; inversions are local.

What is interesting about the *GlobalTime* algorithm is the way we exploit the full power of the multiaccess channel architecture; the ability of all nodes to "snoop the bus" is paramount here. In particular, after node i has won the competition for channel access, the other nodes do not simply wait passively for the next round of competition. Instead, all nodes are active at all rounds — if not transmitting actual data, then performing local update operations.

[7]The cleverness of this strategy is subtle, and will be discussed further in Section 2.4, when we see how the combination of real and dummy timestamps guarantees $O(mn)$ inversions in the worst case.

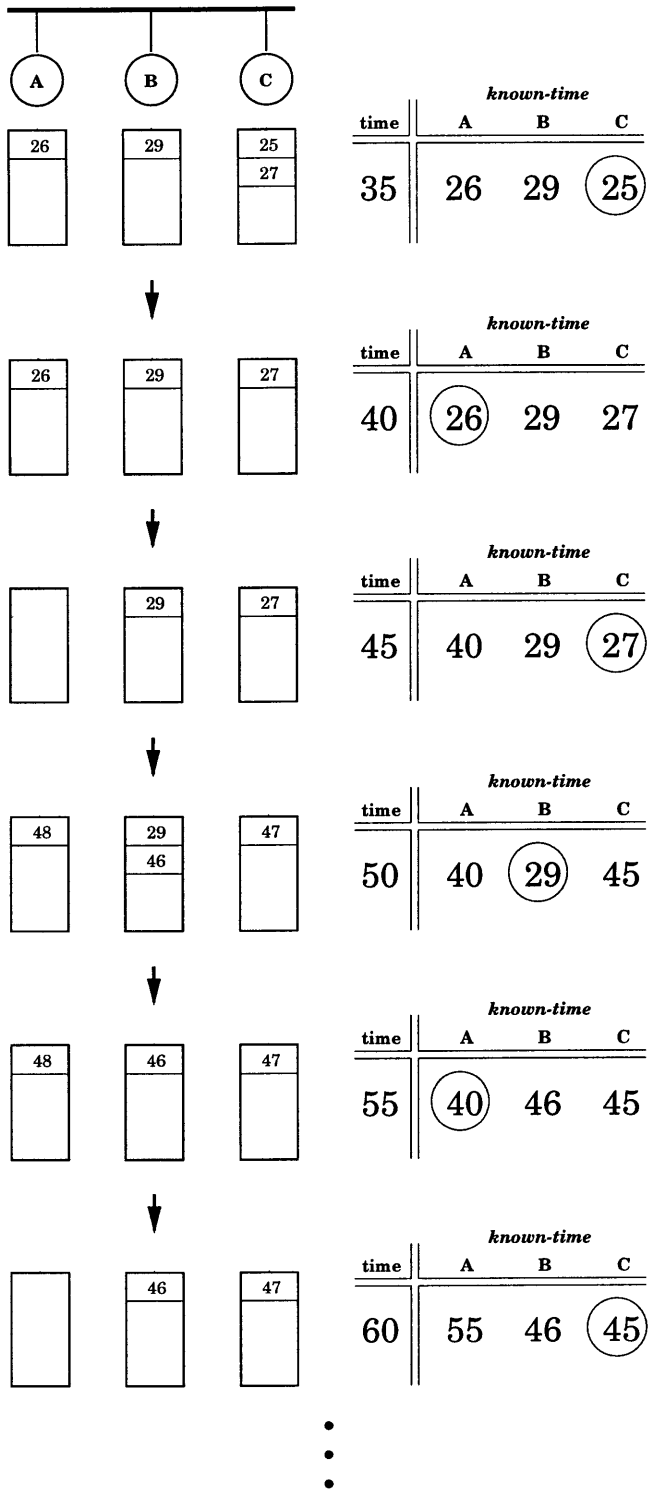


Figure 2-2: A sample execution fragment of the *GlobalTime* algorithm. For each time step, the state of the queues and the common *known-time* table is shown. In this example, the time required to transmit a packet is 5 units.

2.4 Worst case delay

In Section 2.2, we introduced the notion of an inversion in a transmission sequence, and in Section 2.3, we described a new algorithm designed to reduce inversions. In this section, we prove the result we alluded to earlier, that the *GlobalTime* algorithm indeed addresses the issue of eliminating inversions.

Theorem 2.4 *In a sequence of n transmissions generated by the* GlobalTime *algorithm, a packet P with arrival time a_P will be transmitted after at most $m-1$ packets with later arrival times.*

Corollary 2.5 *A sequence of n transmissions generated by the* GlobalTime *algorithm has $O(mn)$ inversions in the worst case.*

Before we prove Theorem 2.4, let us consider the importance of this result. Theorem 2.4 demonstrates that we can approximate a FCFS discipline in a distributed setting. The theorem guarantees that a given packet P will be delayed by no more than $m-1$ time slots than is fair, in the sense that only $m-1$ packets can "cut ahead" of P with respect to the globally optimal order. Put another way, we can divide worst case packet delay into two parts: a *natural delay* associated with the optimal transmission sequence described in Theorem 2.1, and a *residual delay*, the additional delay incurred by an on-line, distributed algorithm. The *GlobalTime* algorithm has an $O(1)$-bounded residual delay — in particular, at most $m-1$ time slots.[8]

Proof of Theorem 2.4: First, we define some notation. Let a_P be the arrival time of packet P at node X, and let τ_P be the value of *known-time*$[X]$ at the time packet P is transmitted. Now consider a packet i that arrives at node B. By step 5 of the *GlobalTime* algorithm, clearly $\tau_i \leq a_i$. Next, consider a packet j that is transmitted before packet i, but arrived at node $A \neq B$ at time $a_j > a_i$. By step 4 of the *GlobalTime* algorithm, we conclude $\tau_j < \tau_i$.

[8]In practice, the $m-1$ bound is not strictly accurate, because idle minislots may add slightly to the residual delay.

We now argue that there cannot exist another packet k at node A, transmitted between the transmissions of packets j and i, such that $a_k > a_i$. If true, the theorem and its corollary follow immediately. Assume for the purpose of contradiction that $a_k > a_i$, but k is transmitted before i. If the extra field in packet j contained the arrival time of the next packet in node A's queue, then

$$\tau_k = a_k > a_i \geq \tau_i \ ,$$

which, by step 4 of the *GlobalTime* algorithm, is a contradiction. On the other hand, if the extra field in packet j contained the current time c, then

$$\tau_k = c > a_j > a_i \geq \tau_i \ ,$$

again a contradiction. □

Suppose that packet x at node X arrives in the system *after* packet y at node Y, but that x is transmitted *before* y. The idea behind the proof of Theorem 2.4 is that no packet originating at X, other than x, has this property. Observe the role of both real and dummy timestamps in the proof in ensuring that all inversions remain localized within small neighborhoods in the transmission sequence. As we have explained earlier, by reducing inversions in the output, the *GlobalTime* algorithm minimizes worst case delay.

2.5 Throughput

One remaining question about the *GlobalTime* algorithm is the maximum traffic intensity it can sustain while ensuring bounded queues. For example, we have seen in Chapter 1 that Aloha cannot guarantee throughput higher than 0.1839. Furthermore, Figure 2-3 illustrates that a traditional TDM protocol produces throughput that in the worst case can be $1/m$, where m is the number of users.

The presence of dummy packets in the *GlobalTime* algorithm causes one to wonder

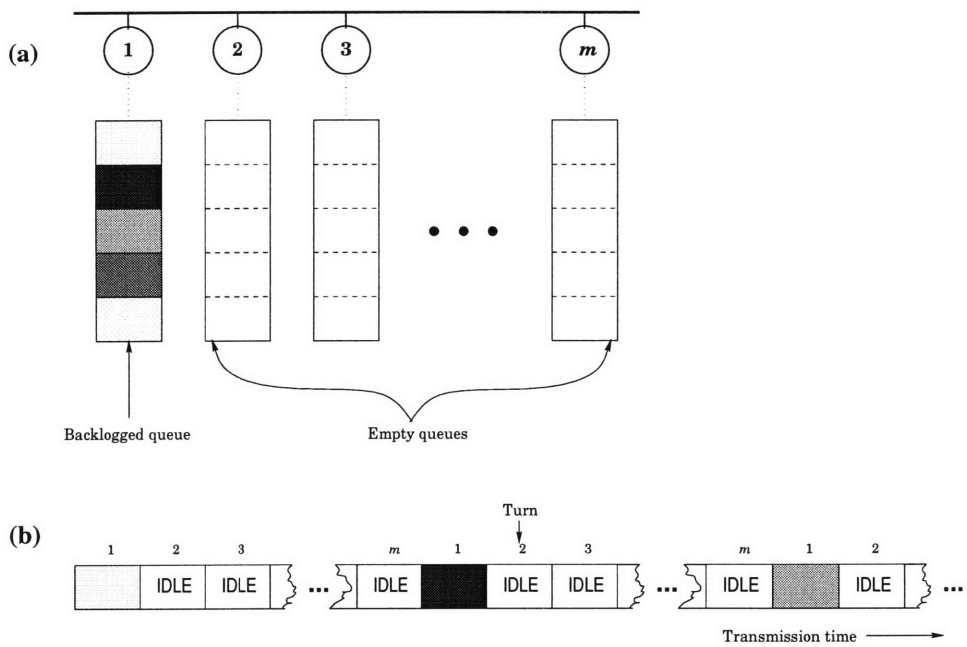


Figure 2-3: An illustration of the maximum stable throughput of a traditional TDM protocol. If all traffic in the system enters the same queue, that queue will remain bounded in size only if the arrival rate does not exceed $1/m$.

if a similar low-throughput phenomenon can occur here, if the input is sufficiently tweaked to force the algorithm's worst case behavior. Theorem 2.6 states that this is not the case.

Theorem 2.6 *The maximum sustainable throughput of the* GlobalTime *algorithm is 100%.*

In the remainder of this section, we provide a convincing argument for the validity of Theorem 2.6.[9] Throughout the argument, the underlying intuition is that the *GlobalTime* algorithm quickly adjusts to the location of waiting packets in the system, thereby limiting the number of dummy packets transmitted and promoting channel utilization.

First, we consider the following property of a reservation-based algorithm that solves the multiaccess communication problem.

[9] A rigorous proof of Theorem 2.6 is more difficult. For further corroborative evidence of the theorem's validity, see the simulation results in Chapter 3.

Property P: If on user A's turn, A has no packets to send at time t_{idle}, then all packets that arrived in the system at times less than t_{idle} must have already been transmitted.

In a single-user system with one FIFO queue, property P clearly holds. In this scenario, if the single user has no packet to transmit at time t_{idle}, then all previously arriving packets have already been transmitted at times less than t_{idle}. Furthermore, if property P indeed holds, then the corresponding algorithm supports 100% throughput. Certainly, if the only time an idle occurs is when there are no packets in the entire system, the channel can be fully utilized if enough data traffic exists to fill it.

In a multi-queue system, however, property P is unsatisfiable. Figure 2-4 sketches the proof of this assertion. At time $t = 56$, there is only one packet in the three-queue system; thus, to obey property P, user A must be served at this time. During the transmission of A's packet, a new packet arrives at time $t = 56.4$ in either user B's or user C's queue. Because executions (1) and (2) are indistinguishable (that is, in both executions, the presence of the newly arriving packet is unknown to the remaining users), the user whose turn follows A's must be the same in both cases.[10] Therefore, either execution (1) or (2) must serve as a counterexample to property P; for example, if B's turn is next, then an adversary can simply place the new packet at node C, as in execution (2).

Although the *GlobalTime* algorithm (and indeed any multi-queue algorithm) does not satisfy property P, it does satisfy a weaker condition expressed in Lemma 2.1.

Lemma 2.1 *In the* GlobalTime *algorithm, if* known-time$[A] = k$, *and the timestamp k is a dummy timestamp recording the last time A had a turn, then all packets with arrival time less than k will be transmitted before A's next turn.*

Proof: At the time *known-time*$[A] = k$ is recorded, k must be the largest value in the common *known-time* table. Therefore, all users will get at least one opportunity

[10]The notion of user "turns" is specific to reservation-based protocols. However, property P also fails under any contention-based multiaccess communication algorithm, because idle slots result from collisions.

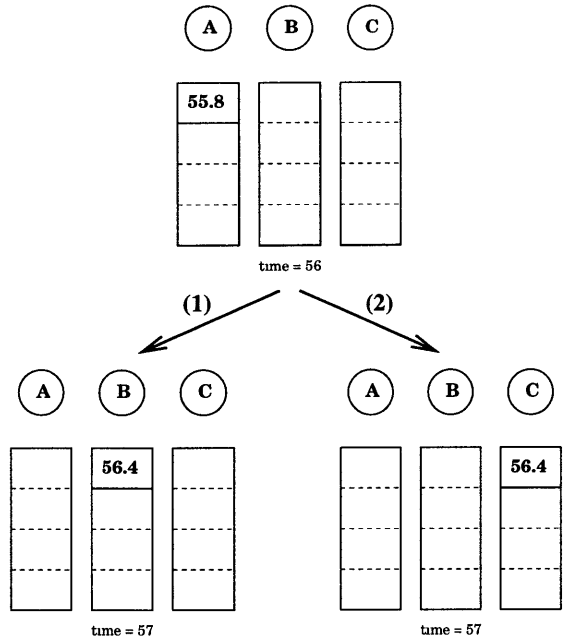


Figure 2-4: Evidence that property P is unsatisfiable in a multi-queue system. If the packet at A is served at time $t = 56$, the arrival of the new packet at either B or C is unknown to the remaining users. Executions (1) and (2) are indistinguishable.

before A's next turn. It follows that all head-of-line packets waiting at time k will be transmitted before A's next turn. By induction on packet position in queue, any other packet waiting at time k will be announced by the piggybacked timestamp on the packet preceding it before A's next turn. We conclude that all packets in the system at time k will be transmitted before A's next turn. □

Property P and Lemma 2.1 both describe the relationship between any idle slot and the packets that must have been transmitted prior to it. Property P states that if user A is idle on its turn, then all packets that had arrived in the system *before the idle turn* must have already been transmitted. By contrast, the *GlobalTime* algorithm satisfies the weaker condition that if user A is idle on its turn, then all packets that had arrived *before A's previous idle turn* must have already been transmitted. We now can prove an essential corollary of Lemma 2.1.

Corollary 2.2 *(Lagging Window Corollary)* *Let t_{idle} be any time at which user A*

is idle on its turn, and let t_{idle} be the kth idle slot in the transmission sequence of length n. Furthermore, let t_{lag} be the time of the $(k-m)$th idle slot, where m is the number of users in the system. Then at time t_{idle}, all packets that arrived in the system before time t_{lag} must have been transmitted.

Proof: Some user X must have generated two of the $m+1$ idle slots between t_{lag} and t_{idle}, by the pigeonhole principle. Applying Lemma 2.1 to user X proves the corollary. □

The lagging window property can be used to show that the *GlobalTime* algorithm has a maximum stable throughput of nearly 100%. The intuition is that if the expected time between consecutive idle slots in the transmission sequence is high, then by definition, the throughput is high. On the other hand, if the expected time between consecutive idle slots in the transmission sequence is low, then the expected lagging window size $E[t_{\text{idle}} - t_{\text{lag}}]$ is small; therefore, by Corollary 2.2 the throughput is again nearly optimal.

Now we proceed to prove Theorem 2.6. Consider a segment of the transmission sequence of length N, between $t_0 = i - N/2$ and $t_N = i + N/2 - 1$ for some i. Define ρ_i to be $1/N \cdot$ (number of transmissions between t_0 and t_N). Finally, we define the throughput ρ:

$$\rho = \lim_{t\to\infty} \frac{1}{t} \sum_{i=N/2}^{t} \rho_i \ .$$

Figure 2-5 illustrates the setup of the proof of Theorem 2.6. Notice that t_{last} is defined as the time of the last idle slot in the transmission sequence fragment, and t_{lag} occurs m idle slots earlier, as in Corollary 2.2. Then

$$\begin{aligned} \rho &= \frac{1}{N} \lim_{t\to\infty} E\left[\text{number of transmissions between } t_0 \text{ and } t_N\right] \\ &\geq \frac{1}{N} \lim_{t\to\infty} \left(E\left[t_N - t_{\text{last}}\right] + E\left[\text{number of arrivals in range } [t_0, t_{\text{lag}}]\right]\right) \ , \end{aligned}$$

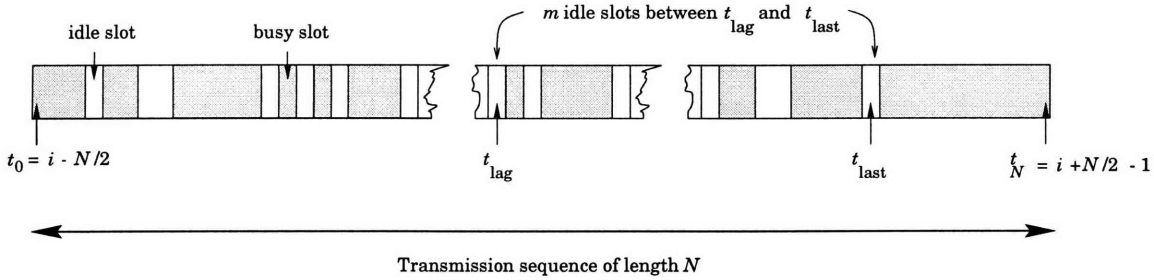


Figure 2-5: An illustration of a transmission sequence of length N with t_{lag} and t_{last} labeled, as described in the proof of the theorem.

the latter term following from the Lagging Window Corollary. Thus,

$$\rho \geq \frac{1}{N} \lim_{t \to \infty} \left(E[t_N - t_{\text{last}}] + \lambda \cdot E[t_{\text{lag}} - t_0] \right) ,$$

where λ is the arrival rate of the system.[11] It follows that

$$\begin{aligned} \rho &\geq \frac{1}{N} \lim_{t \to \infty} \left(\lambda N + (1-\lambda) E[t_N - t_{\text{last}}] - \lambda \cdot E[t_{\text{last}} - t_{\text{lag}}] \right) \\ &\geq \frac{1}{N} \lim_{t \to \infty} \left(\lambda N - \lambda \cdot E[t_{\text{last}} - t_{\text{lag}}] \right) \\ &= \frac{1}{N} \left(\lambda N - \lambda \cdot m \cdot \frac{1}{1-\rho} \right) , \end{aligned}$$

where $m \cdot (1-\rho)^{-1}$ is equal to the expected length of time between t_{last} (the last idle slot) and t_{lag} (m idle slots before t_{last}) in a transmission sequence fragment of length N, the limit taken as time approaches infinity. The factor $(1-\rho)^{-1}$ can be thought of as the expected spacing between consecutive idle slots in the transmission sequence, a function of the throughput ρ. Thus, the maximum stable throughput ρ satisfies

[11] Strictly speaking, the validity of this last transformation requires Poisson traffic; in particular, the expected number of arrivals in an interval must be proportional to the length of the interval. This is why the argument is suggestive and convincing, but not rigorous.

$$\rho(1-\rho) = \lambda(1-\rho) - \lambda\frac{m}{N} \ .$$

Because $m \ll N$, we eliminate the negligible $\lambda m/N$ term, and we obtain

$$\rho^2 - (1+\lambda)\rho + \lambda = 0 \ ,$$

with roots λ and 1.

In conclusion, the *GlobalTime* algorithm can support as much throughput as there is arriving traffic, as long as $\lambda < 1$. Clearly, this is a significant improvement over traditional TDM, where the maximum stable throughput is $1/m$. In the next chapter, we examine the performance of the *GlobalTime* algorithm in practice.

Chapter 3

Simulations

3.1 Objective

In Chapter 2, we investigated the *GlobalTime* algorithm as a theoretical solution to the multiaccess communication problem. Our main objective in that discussion was the reduction of the number of inversions in a transmission sequence, so as to more closely approximate first-in-first-out (FIFO) service in a distributed, multi-queue setting. We proved a theorem bounding the number of inversions in any transmission sequence generated by the *GlobalTime* algorithm, and we stated that this bound improved upon the performance of existing algorithms.

In developing the theory behind the *GlobalTime* algorithm, our underlying assumption was that inversions can serve as a useful intermediary between the actual sequence of transmissions produced and the performance measures we really care about: delay and throughput. Certainly, if this assumption were valid, then we have greatly simplified the problem; by mapping an input set of real number arrival times to an associated set of integer ordinals, the space of possible input permutations collapses. Furthermore, we provided theoretical evidence that inversions are indeed a useful measure. The more inverted the transmission sequence is, the longer some packet in the system will have to wait for service relative to another packet; thus, inversions are strongly correlated with end-to-end delay and delay jitter, both highly undesirable properties [15].

Our objective in running simulations is to observe and measure our algorithm's performance in practice. We say "in practice" loosely, because we neglect many minor issues like clock skew, propagation delay, and node failures, all of which would need to be handled in a real implementation of the *GlobalTime* algorithm. What we really mean is that our simulation will first generate a set of random traffic data collected over a long time frame. Then the simulation will run this input set through the *GlobalTime* algorithm under ideal conditions, collecting data on delay and throughput. Viewed in this way, the *GlobalTime* simulation is less a network protocol than it is an application of an abstract mathematical function, as illustrated in Figure 3-1.

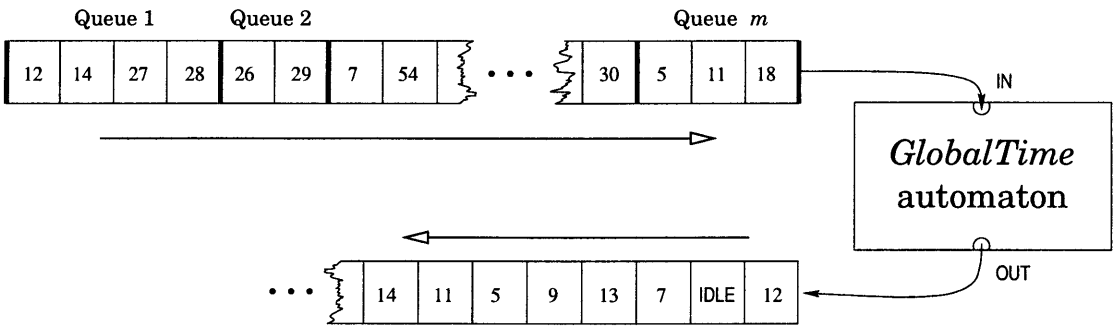


Figure 3-1: The *GlobalTime* algorithm, viewed as an automaton. The automaton takes as input an array of arrival times, and returns as output a permuted version of the same array with intervening idle slots. Although the algorithm is a distributed protocol, we can simulate it as an abstract mathematical "function."

In this chapter, we describe in detail the simulation of the *GlobalTime* algorithm. In Section 3.2, we explain the basic layout of the code and the role played by each of the main functions. In Section 3.3, we delineate the computational procedures required to generate interesting traffic patterns. Section 3.4 discusses the implementation of the *GlobalTime* algorithm and three other reservation-based algorithms; in particular, Section 3.4 emphasizes the cleverness needed to implement the *GlobalTime* algorithm in code efficiently. In Section 3.5, we present the results of the simulations. Applying the familiar yardsticks of throughput and delay, we will see that the *GlobalTime* algorithm performs admirably in comparison with other canonical

reservation-based disciplines.

3.2 Overview of the code

The code for the *GlobalTime* simulation is written in the programming language ANSI C, selected for its efficiency and power. The code was expected to be computationally intensive; we would require huge data structures to hold the simulated arrivals, and we would need millions of arithmetic calculations to keep track of performance statistics.[1] In addition, C is well-supported in MIT's Athena computing environment. The code was edited in Emacs and compiled with cc. Appendix A contains the complete code listing for the *GlobalTime* simulation. Figure 3-2 depicts the graph of dependencies among the main functions.

At the root of the simulation is the `main` procedure. In lines 5–20 of the code, the user specifies a range of m (the number of nodes), N (the time of the simulation), and λ (the aggregate arrival rate) that he wishes to simulate. In addition, the user sets the desired increments for m, N, and λ; for example, he can choose to test all values of λ between 0.05 and 0.55 at increments of 0.05. If the user desires a bursty traffic simulation, he can also set a range for the mean burst size. The `main` procedure is simply a set of nested loops that runs the simulation through all the possible combinations of requested parameter settings. For example, if the user requests data for $m = 128$, $10^4 \leq N \leq 5 \times 10^4$ at increments of 10^4, and $0.6 \leq \lambda \leq 0.8$ at increments of 0.02 on Poisson traffic, then the `main` procedure will call `run_simulation` 55 times, producing 55 data points.

The chief "administrator" of the code is the `run_simulation` procedure. As shown in Figure 3-2, the code for `run_simulation` calls all the functions necessary for the entire simulation, both those involved in traffic generation and in the actual multi-access communication algorithms. The arguments to `run_simulation` are a single value each of m, N, λ, and mean burst size, passed to it by `main`. In line 14 of the

[1]The author had no bias for C, because he had no prior experience programming in C, or programming at all in the last five years. However, the language was surprisingly easy to learn in just a couple of days, given a strong background in algorithms and data structures.

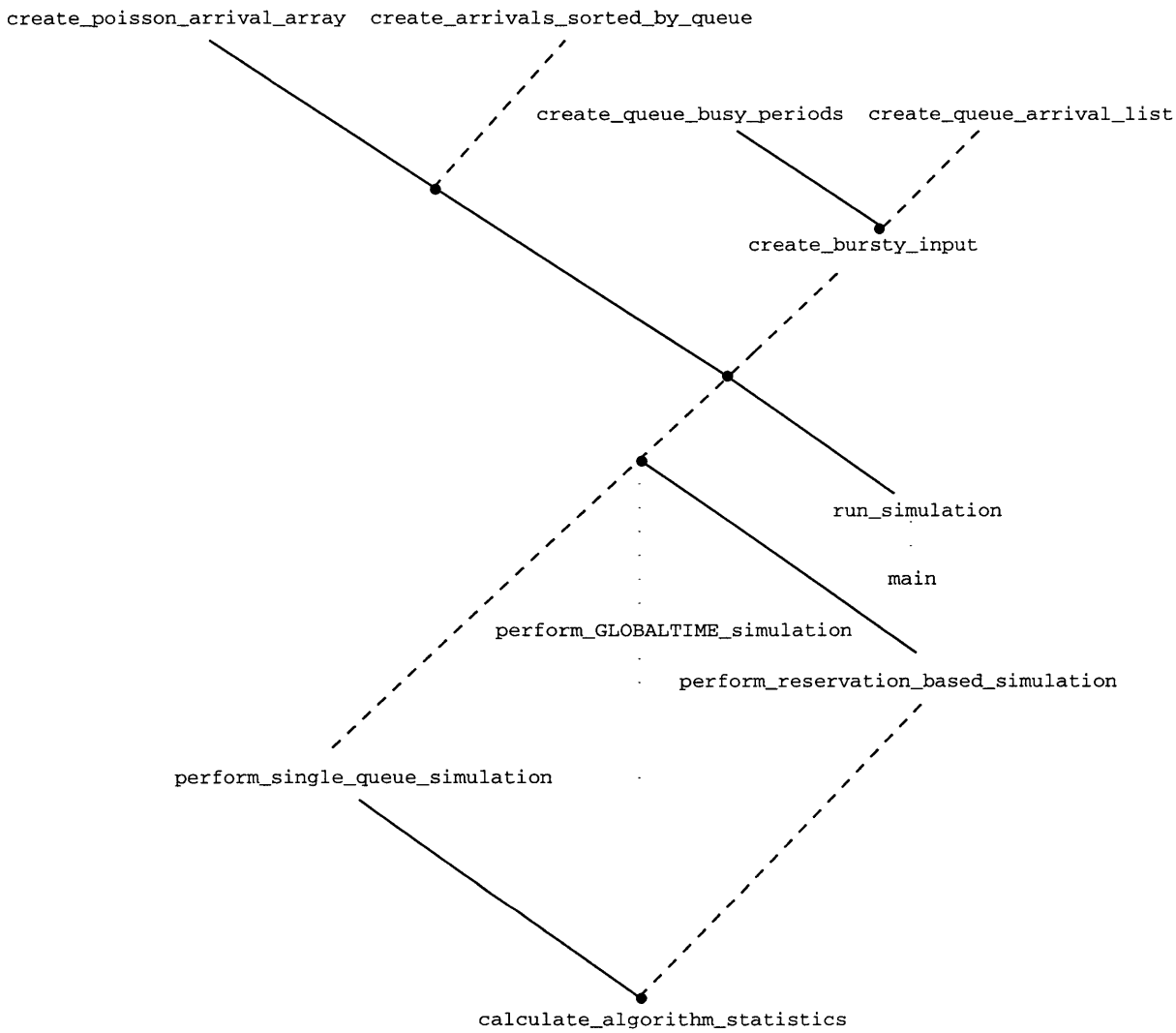


Figure 3-2: Graph of the dependencies among the primary functions in the code for the *GlobalTime* simulation.

code, the user indicates the desired number of trials per simulation; the performance statistics from the trials are simply averaged in producing a single data point.

Within a single trial, a set of random arrivals is first created according to the desired traffic distribution. Then this input set is run through both the *GlobalTime* algorithm and three other canonical reservation-based algorithms: gated limited service, gated unlimited service, and exhaustive.[2] Furthermore, for the sake of comparison, the simulation runs the same set of arrivals through a single FIFO queue. As `run_simulation` loops through the assigned number of trials, the performance data is accumulated and later divided by the total number of trials. At the completion of a single call to `run_simulation`, the results are written to the file specified by the user in the format of 34 comma-separated values per line, constituting a single data point. The 34 values are m, N, λ, mean burst size, and mean delay, delay variance, and worst case delay for both the average and worst case users in each of the five scheduling algorithms ($4 + 3 \cdot 2 \cdot 5 = 34$).

In the next two sections, we describe the procedures for traffic generation and algorithm implementation in more detail.

3.3 Generation of traffic

3.3.1 Poisson arrivals

The first traffic pattern we consider is the Poisson process, a distribution "generally considered to be a good model for the aggregate traffic of a large number of similar and independent users" [4]. Regardless of whether this assertion is actually true, the Poisson process is well understood and is widely employed in the simulation of network protocols.

The procedure `create_poisson_arrival_array` generates a sequence of Poisson arrivals between 0 and N, with aggregate arrival rate λ. To create this input, C's pseudorandom number generator first outputs an integer between 0 and $2^{31}-1$. Sec-

[2]See Chapter 1 for further discussion of canonical reservation-based disciplines.

ond, this integer is divided by $2^{31}-1$ to produce a random decimal between 0 and 1. Next, `create_poisson_arrival_array` needs to map the uniform distribution on $(0,1)$ to the exponential distribution on $(0,\infty)$, in order to generate a random interarrival time from the random decimal; the appropriate mapping function is

$$\texttt{random\_interarrival} = -\frac{\ln(1-\texttt{random\_decimal})}{\lambda}\,. \tag{3.1}$$

By iteratively producing random interarrival times, we easily build a list of Poisson arrivals by adding each new interarrival time to the time of the last arrival to generate the next arrival. When complete, this list is copied into an array whose first element is the array's length; this array is returned to the caller.

The `poisson_arrival_array` created by `create_poisson_arrival_array` serves as an input to `perform_single_queue_simulation`. However, we still require another procedure to generate m Poisson traffic streams for the multi-queue simulations. Fulfilling this requirement, the procedure `create_arrivals_sorted_by_queue` generates an array of random queue assignments for each arrival. The independent random choices for each queue assignment are again created with C's pseudorandom number generator, but this time, the resulting integer between 1 and $2^{31}-1$ is evaluated modulo m. As illustrated in Figure 3-3, the m resulting streams are themselves each Poisson with rate λ/m [4]. The m queues are placed in order into one long array the same size as the original `poisson_arrival_array`. `Queue_offsets`, an auxiliary array of length m, keeps track of where one queue ends and the next begins.

3.3.2 Bursty arrivals

In practice, real data traffic patterns exhibit bursts of activity followed by long idle periods. In this section, we describe a reasonable model for bursty traffic and the code used to generate it.

In our bursty traffic model, time is divided into busy periods and idle periods independently for each user. During busy periods, arrivals are Poisson distributed with rate 1; during idle periods, no traffic arrives. The length of a busy period is an

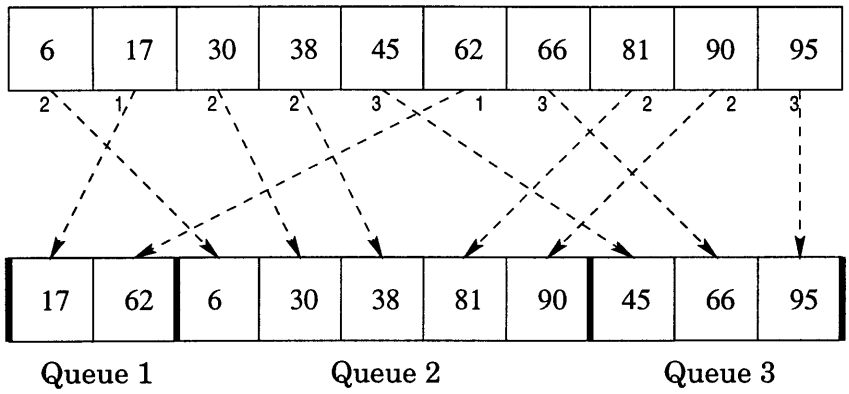


Figure 3-3: A small example of the procedure `create_arrivals_sorted_by_queue`. Each arriving packet in the system is randomly sent to one of the three queues. The result is that the arrivals at each queue will be Poisson distributed with rate $\lambda/3$.

exponential random variable with given mean burst size. Similarly, the length of an idle period is exponentially distributed, where

$$\texttt{mean\_idle\_size} = \frac{\texttt{mean\_burst\_size}}{\texttt{arrival\_rate}} - \texttt{mean\_burst\_size} \ . \tag{3.2}$$

Of course, in this case, $\texttt{arrival\_rate} = \lambda/m$, where λ is the aggregate arrival rate to the system and m is the number of users, because we assume that the time-averaged arrival rate per user is identical. Figure 3-4 illustrates the model for bursty traffic.

Implementing bursty traffic in code requires first that the busy periods be generated, and then the arrivals within the busy periods. Exactly as its name indicates, the procedure `create_queue_busy_periods` produces a list of busy periods given a user's packet arrival rate and mean burst size and a value for N. Because busy periods are time intervals, each element of this list is a triple consisting of a starting time, an ending time, and a pointer to the next element. As in Section 3.3.1, the procedure generates exponentially distributed busy and idle period lengths by first producing a decimal uniformly distributed on $(0, 1)$, and then mapping it to the range $(0, \infty)$:

$$\texttt{random\_busy\_period} = -\ln(1 - \texttt{random\_decimal}) \cdot \texttt{mean\_burst\_size} \tag{3.3}$$

$$\texttt{random\_idle\_period} = -\ln(1 - \texttt{random\_decimal}) \cdot \texttt{mean\_idle\_size} \tag{3.4}$$

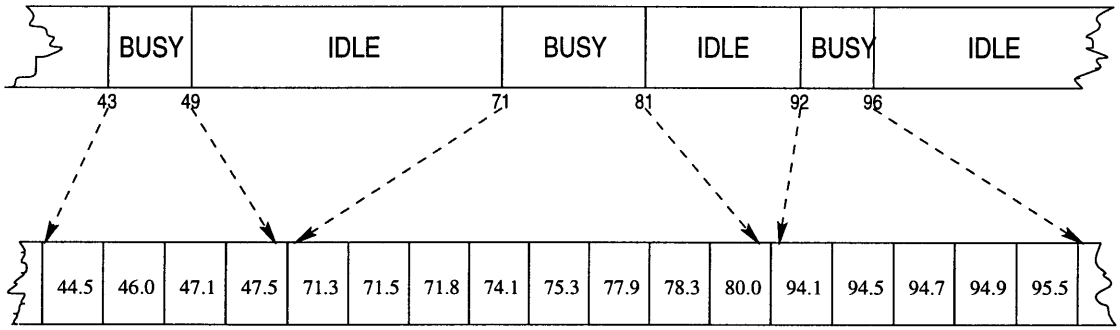


Figure 3-4: An illustration of the model for bursty traffic in the *GlobalTime* simulation. Busy periods and idle periods alternate, each with exponentially distributed length. Within a busy period, arrivals are Poisson distributed with rate 1. No packets arrive during idle periods.

Given a list of busy periods, the procedure `create_queue_arrival_list` now creates a single list of arrivals, as shown in Figure 3-4. The graph in Figure 3-2 demonstrates the relationship between the procedures `create_queue_busy_periods` and `create_queue_arrival_list`, and their caller, `create_bursty_input`. The procedure `create_bursty_input` first calls the two lower level procedures m times each, yielding m traffic streams, all inserted into `array_of_arrival_lists`. The remainder of the procedure simply converts the array of lists format to the required format: namely, a single long array with an accompanying table storing the queue offsets. Observe that the bursty traffic simulation, unlike the Poisson traffic simulation, starts by generating the multi-queue input. To provide the associated single queue input, `run_simulation` applies quicksort to the `arrivals_sorted_by_queue` array. An important benefit of the single array data structure for tracking m queues is this capacity to sort the arriving packets with no additional overhead from data shuffling.

3.4 Algorithm implementations

3.4.1 Single queue algorithm

The procedure `perform_single_queue_simulation` requires only a few lines of iterative code. The variable `time` is incremented by 1 if a packet is currently in the queue, and by `BETA` $\ll 1$ otherwise. The value of `BETA` represents the length of an idle minislot, and is a predefined global constant.[3] When a packet is indeed transmitted, its queuing delay is calculated; this computed packet delay is used to update `total_delay`, `total_square_delay`, and `worst_case_delay`. Finally, when `time` exceeds N, the `while` loop is exited, and the results are recorded in `single_queue_results`: mean delay, delay variance, and worst case delay.

3.4.2 Reservation-based algorithms

A generalized procedure for simulating canonical reservation-based multiaccess communication schemes, `perform_reservation_based_simulation` can emulate gated limited service, gated unlimited service, and exhaustive disciplines. As discussed in Chapter 1, all three are TDM-like protocols — that is, users receive turns in cyclic order. When it is node i's turn to transmit, i sends all its packets in both a gated unlimited and an exhaustive service discipline (`protocol` $= 1$ and `protocol` $= 2$, respectively). The only difference between the two is whether or not packets that arrive during node i's turn get served during that same turn; in an exhaustive scheme, they do. Based on the integer value of `protocol`, the auxiliary function `protocol_switch` decides whether `gate_time` (the time node i's turn began) or `time` (the current time) should determine if i's head-of-line packet is eligible to be transmitted.

On the other hand, if `protocol` $= 0$, then a gated limited service discipline is being employed; in this case, at most one packet per user per turn is transmitted. A single transmission by node i sets a flag which forces i to relinquish its turn the second time through the `while` loop. Regardless of the specific reservation-based scheme,

[3] A minislot is used whenever there is no data to transmit. In the *GlobalTime* algorithm, a minislot will also contain a dummy timestamp of a few bytes.

after each packet transmission, the delay statistics are updated as in the single queue simulation; in the multi-queue case, however, arrays must be maintained to keep track of the statistics *per user*. At the end of the simulation, these user statistics tables are analyzed by the procedure `calculate_algorithm_statistics` for the following numerical data:

- Mean delay of the average user
- Mean delay of the user with the highest mean delay
- Delay variance of the average user
- Delay variance of the user with the highest delay variance
- Worst case delay of the average user
- Worst case delay of the user with the highest worst case delay

By examining the performance of both the average and the worst case user, we not only observe the algorithm's delay characteristics, but also its fairness.

3.4.3 *GlobalTime* algorithm

The procedure `perform_GLOBALTIME_simulation` handles the implementation of the *GlobalTime* algorithm. As discussed in Chapter 2, each node in an execution of the *GlobalTime* algorithm maintains the state variables *known-time*$[i]$ for $i = 1, 2, \ldots, m$, each storing a time corresponding to either the arrival time of i's next-in-line packet, or the last time i received a turn in which i's queue contained no next-in-line packet. Furthermore, we observed that the main computational task in the *GlobalTime* algorithm is finding the minimum among the m *known-time* variables every time slot.[4] The node i whose *known-time* value is smallest receives a turn, and at the end of its turn, *known-time*$[i]$ must be updated in each node's local table. This repeated

[4]Assuming 2 μs time slots, finding the minimum among m real numbers may not be feasible with 1998 technology, despite our efficient implementation. It is quite likely that chips will continue to scale to higher speeds in coming years.

application of FIND-MIN and INCREASE-KEY operations to a dynamic set of values suggests a *known-time* priority queue.

For $m < 50$, an unsorted array performs fine, but for larger m, we employ a binary heap, as illustrated in Figure 3-5. The heap is keyed on the actual *known-times*, and a second field holds the associated node's unique identifier. As demonstrated in Figure 3-5, the structure of a binary heap implies that `turn = known_time[0].val`. After a node's turn, `known_time.key` is increased, and the array is reheapified. Of course, the asymptotic gain is that the aggregate cost of a node's turn is $O(\lg m)$ rather than $O(m)$, as would be required in linear search for $\min_i\{\textit{known-time}[i]\}$. This difference is substantial for large m, especially given the real implementation issue of time slots only a few microseconds in length. Because a binary heap and an unsorted array are maintained differently, an indicator variable `heap` switches between two sets of related functions in several parts of the code: for example, during the update of the *known-time* array.

As in Section 3.4.2, the procedure `calculate_algorithm_statistics` analyzes the data in the statistics arrays at the end of the simulation. Data is stored in these arrays (`total_delay`, `total_square_delay`, and `worst_case_delay`) throughout the execution.

3.5 Simulation results

As described in Section 3.2, the *GlobalTime* simulation writes data points to a file in the format of 34 comma-separated values per line. To analyze this data, we imported the `data.csv` file into Xess, a spreadsheet program supported on MIT's Athena workstations. The spreadsheet software was practical for sorting the data and performing additional calculations not computed by the original C code: for example, for taking the square root of delay variance to obtain delay standard deviation. Xess also provides aesthetically pleasing line and scatter graphs of specified data sets, the product of which comprises much of the remainder of this chapter.

In sketching the implementations of the five simulated algorithms in Section 3.4,

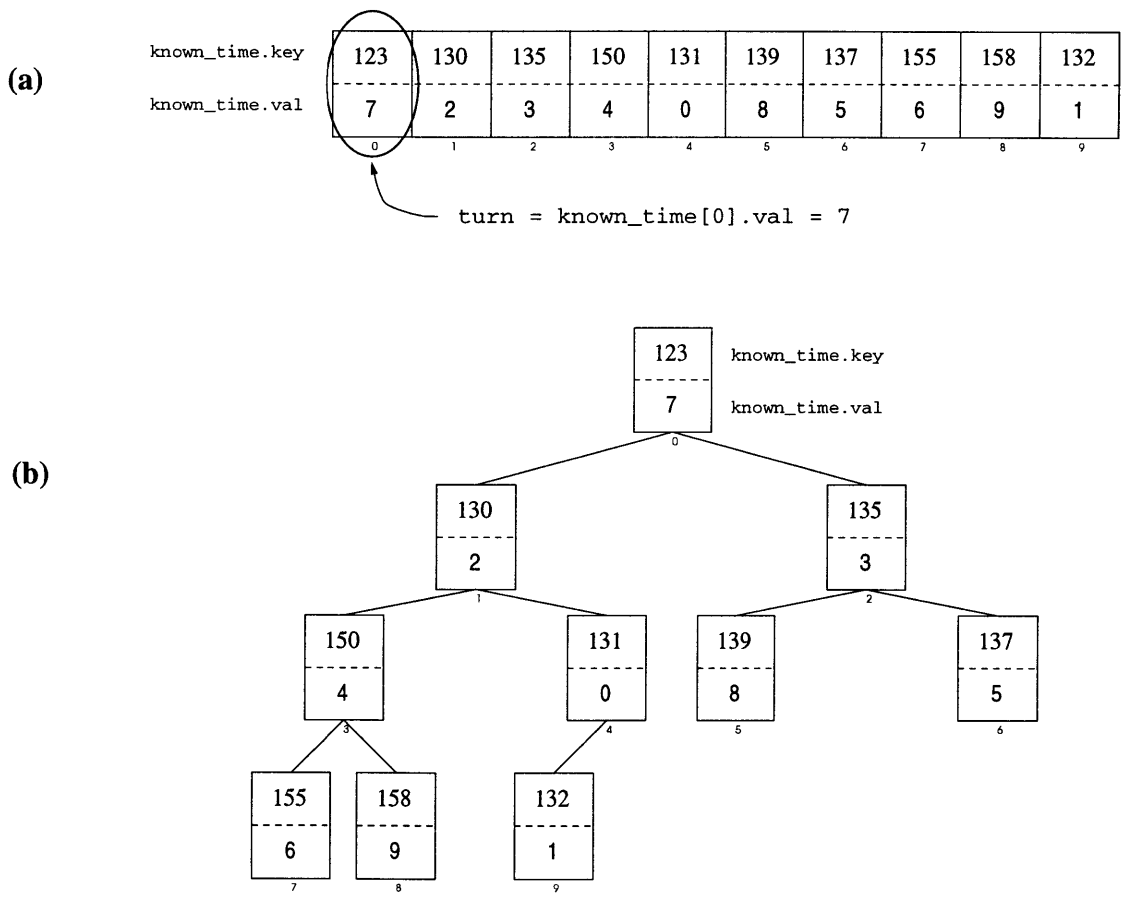


Figure 3-5: Handling of the `known_time` state variables using a binary heap as priority queue in the *GlobalTime* algorithm. In (a), the node whose turn it is to transmit is simply the number stored in `known_time[0].val`. In (b), the heap structure of the array is illustrated. Nodes are keyed by their `known_time`.

we alluded to six performance metrics; the six numbers returned by each algorithm's simulation correspond to these six performance criteria. Before discussing the results of the simulations, let us examine these six performance metrics in greater detail:

Mean delay

The mean delay of a user i is defined as the average queuing delay of i's packets. User i cares about both the mean delay of the average user and of the worst case user in the system. The reason for this dual concern is that although i certainly wants a low expected mean delay, more importantly, i wants a low *guaranteed* mean delay. If even one user experiences significantly higher mean delay than the others, then from i's standpoint, the algorithm is unacceptable — unless i could somehow be assured that he wouldn't be the unlucky one!

Worst case delay

The worst case delay of a user i is defined as the queuing delay of i's most delayed packet. In a setting where clients pay to receive performance guarantees, a low delay bound is more important than low mean delay. High variation in packet delay requires larger packet buffers and greater timing flexibility in receiving end protocols [15]. Of course, because our model does not deterministically bound the burstiness of incoming traffic, we say "worst case delay bound" loosely. What we really mean is that with very high probability, a packet's delay will obey the bound; indeed, this is the best we can do given a stochastic bounding traffic model as input (such as the models described in Section 3.3).

Delay variance

Delay variance is a softer measure of delay jitter than a worst case bound, and is therefore more likely to converge quickly in simulations. As with the previous two performance criteria, our interest in delay variance extends to both the average and the worst case user, the latter enabling us to provide a delay variance guarantee.

In running the simulations, we set several parameters. First, all simulations were run with $N = 2^{15}$, where one time slot is equal to the service time of a single packet.

Assuming 2 μs time slots, the simulations were run for approximately 66 ms.[5] Second, we set BETA, the length of a minislot, equal to 0.01.[6] Third, TRIALS was set to 32, so that all data points are the average of 32 executions.

In Sections 3.5.1 and 3.5.2, we analyze the results of the simulations on Poisson and bursty traffic, respectively.

3.5.1 Poisson traffic

The graphs in Figures 3-6 through 3-8 plot mean delay, delay variance, and worst case delay against system arrival rate for four different values of m in an execution of the *GlobalTime* algorithm.[7] The graphs have logarithmic y axes, so that the plots are easily discernible for both high and low arrival rates. Observe that the delays caused by the *GlobalTime* algorithm increase with m. For example, in Figure 3-6, at 80% throughput, if the simulation is run on 32 nodes, the mean queuing delay is slightly more than 2 time slots; on 128 nodes, the mean delay jumps to 5, and on 512 nodes, to approximately 20 time slots. This is not surprising, because for the same value of λ, more nodes mean more dispersion of packets throughout the system. Greater dispersion leads to greater time lag while the *GlobalTime* algorithm cycles through the nodes, looking for the small fraction of nodes that actually have a packet to transmit in any given time slot. As we discussed in Chapter 2 and will see again shortly, this problem is fundamental to reservation-based algorithms.

Figure 3-8 illustrates statistical worst case bounds in a simulation of 2^{15} time slots. At 80% throughput on 128 nodes, the worst case delay is approximately 35 time slots. That an execution run for over 32,000 time slots in an 128 node distributed system can guarantee end-to-end delay not exceeding 35 time slots is remarkable. Of course, Poisson traffic is forgiving to multiaccess communication algorithms, and a more

[5]Furthermore, assuming a 2 Gb/s channel and $\lambda = 0.8$, then approximately 13.2 MB of data are transmitted during one execution.

[6]The assumption BETA = 0.01 might be an underestimate with 2 μs time slots, but is typical in Ethernet local area networks [4]. In any case, both the *GlobalTime* algorithm and its competitors face the same problem, regardless of our choice of BETA.

[7]Unless otherwise noted, the discussion in this chapter focuses on the performance of the most poorly treated user in the system.

challenging test of performance is bursty traffic, to be discussed in the next section.

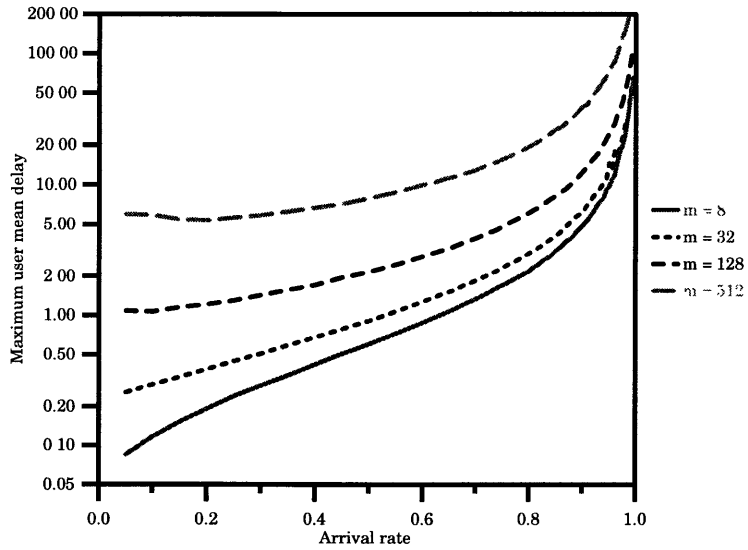


Figure 3-6: Mean delay of the user with the highest mean delay in an execution of the *GlobalTime* algorithm. All users generate Poisson traffic with equal rate.

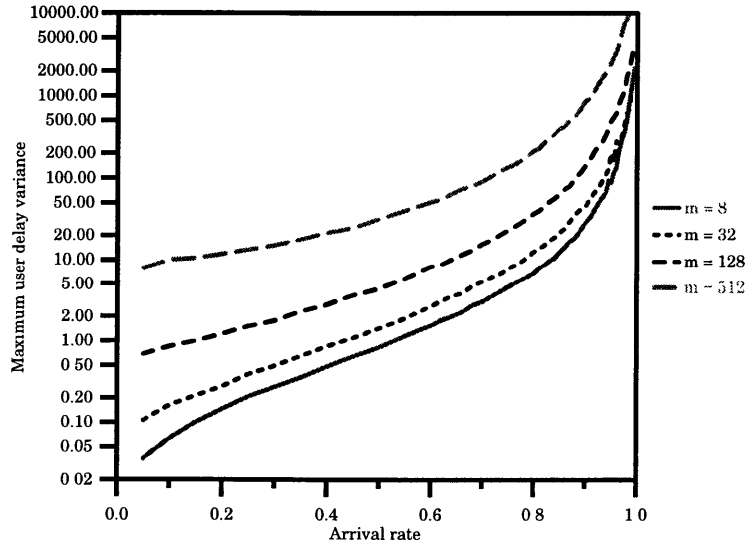


Figure 3-7: Delay variance of the user with the highest delay variance in an execution of the *GlobalTime* algorithm. All users generate Poisson traffic with equal rate.

In Figure 3-9, we compare delays for the average and worst case user in the

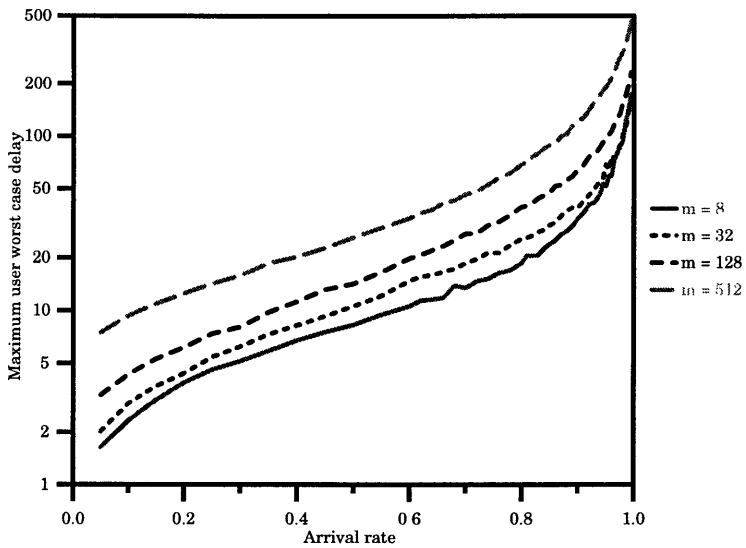


Figure 3-8: Worst case delay of the user with the highest worst case delay in an execution of the *GlobalTime* algorithm. All users generate Poisson traffic with equal rate.

GlobalTime algorithm on 128 nodes. Ideally, the disparity between the average and most poorly treated user should be minimized, since clients are more interested in performance guarantees than in luck. As Figure 3-9 illustrates, the average and worst case user are treated similarly. At 80% throughput, mean delay is nearly identical for the average and worst case user at approximately 5; as expected, worst case delay is slightly more disparate, at 20 time slots for the average user and 35 for the most poorly treated. Clearly, a key advantage of the *GlobalTime* algorithm is that it distributes system delay equitably among its users.

Like Figure 3-9, the remaining graphs in this section are 128 node simulations. In Figures 3-10 through 3-12, the mean delay, delay variance, and worst case delay of the *GlobalTime* algorithm are plotted on the same set of axes as the mean delay, delay variance, and worst case delay of the four other simulated algorithms cited in Section 3.4. Figure 3-10 illustrates that mean delay is nearly identical for the four multi-queue algorithms, even for high values of λ. The figure strongly suggests that there is something fundamental to the distributed nature of the multiaccess communication problem, because the centralized single queue algorithm is able to

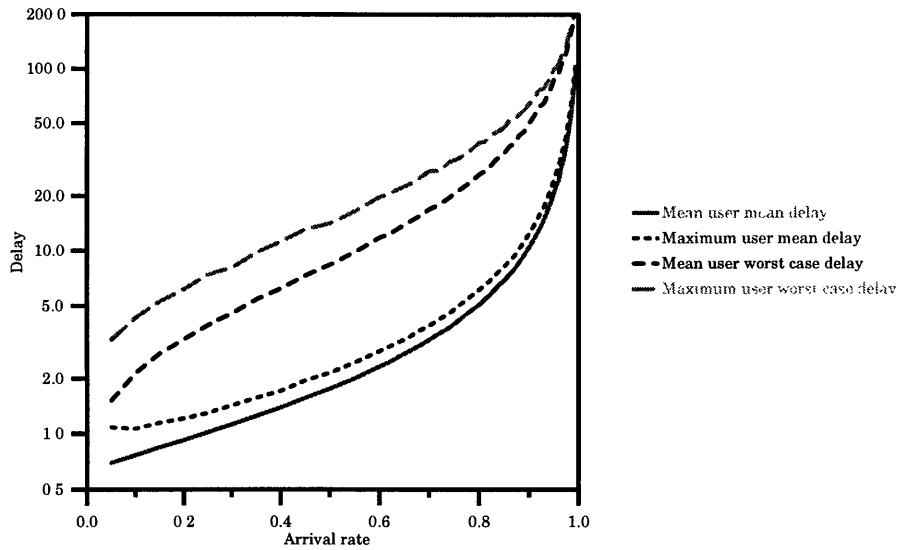


Figure 3-9: Comparison of mean and worst case delays in the average user and the most poorly treated user in an execution of the *GlobalTime* algorithm. All users generate Poisson traffic with equal rate.

achieve substantially lower mean delay. We can understand Figure 3-10 intuitively by observing that mean delay is insensitive to packet ordering; serving packet i ahead of packet j or vice versa has no effect on collective delay, and thus on mean delay. Therefore, whether an algorithm serves packets nearly in the order of their arrivals (as in the *GlobalTime* algorithm), or in a round robin fashion by user (as in gated limited service), or by some other scheme, the mean delay will not change much.

A more practical performance measure, worst case delay, is plotted in Figure 3-12. As demonstrated in the figure, for $\lambda > 0.6$, gated limited service is significantly inferior to the other three multi-queue algorithms. Furthermore, at $\lambda = 0.8$, the gated unlimited and exhaustive algorithms start to diverge from the *GlobalTime* algorithm. A close-up of this divergence is shown in Figure 3-13. At 90% throughput, the *GlobalTime* algorithm reduces the delay bound by approximately 25% over the unlimited service disciplines and by 60% over gated limited service.

What we can conclude from these graphs is that given Poisson traffic, the *GlobalTime* algorithm does not show markedly improved delay bounds compared to other canonical reservation-based schemes unless the throughput is very high. In other

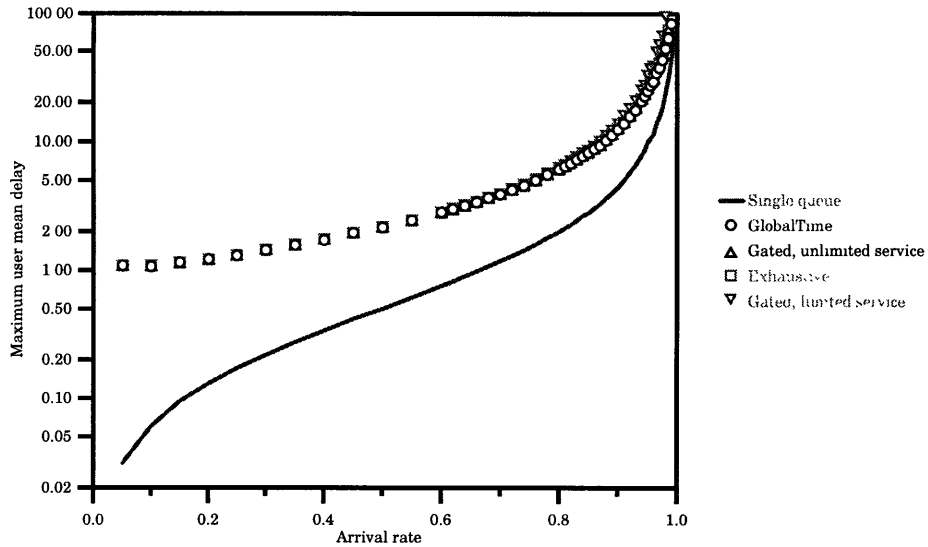


Figure 3-10: Mean delay of the user with the highest mean delay in each of 5 simulated algorithms. The *GlobalTime* algorithm is compared with gated unlimited service, gated limited service, and exhaustive service disciplines, and with a single FIFO queue for comparison. Simulation is run with Poisson traffic on 128 nodes.

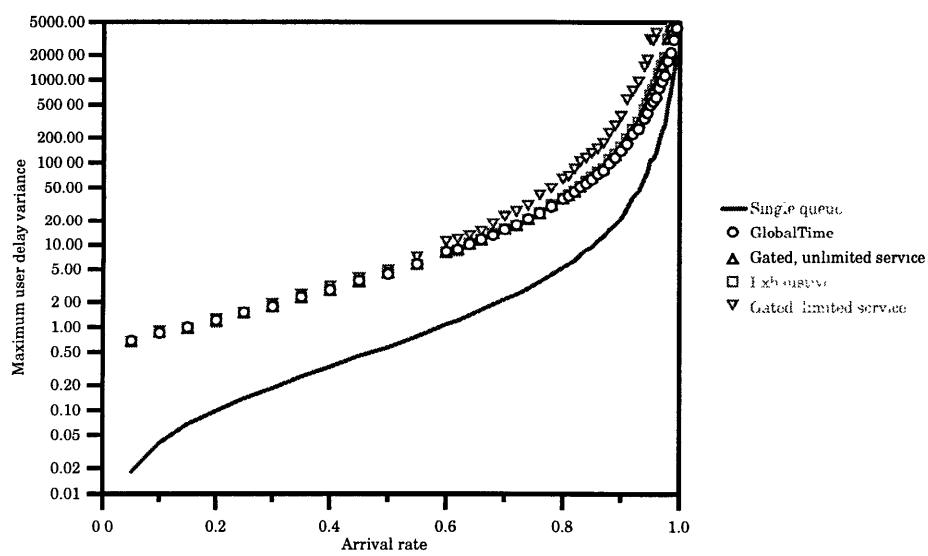


Figure 3-11: Delay variance of the user with the highest delay variance in each of 5 simulated algorithms. Simulation is run with Poisson traffic on 128 nodes.

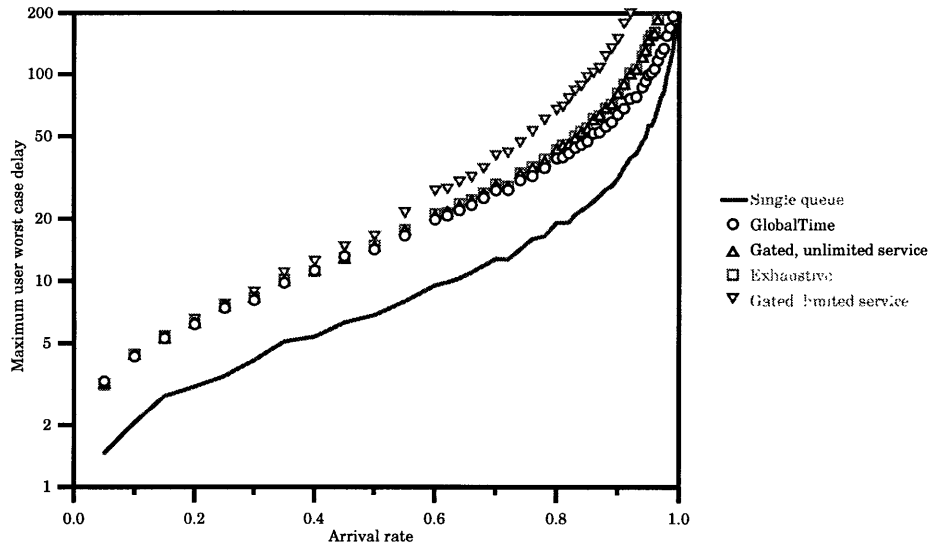


Figure 3-12: Worst case delay of the user with the highest worst case delay in each of 5 simulated algorithms. Simulation is run with Poisson traffic on 128 nodes.

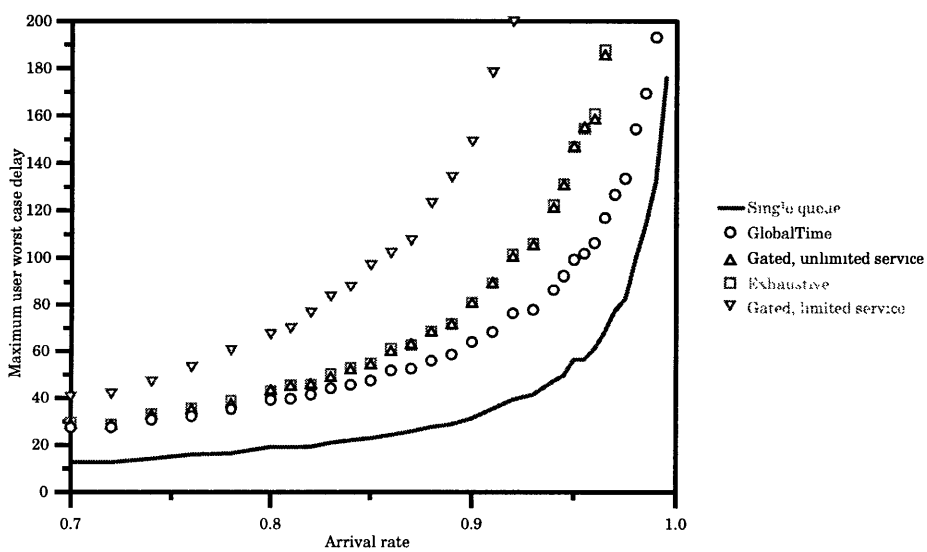


Figure 3-13: A closer look at the worst case delay of the most poorly treated user in the system in each of 5 simulated algorithms. With Poisson arrivals, there is little difference between the delay bound of the *GlobalTime* algorithm and that of other unlimited service schemes when $\lambda < 0.8$.

words, if the traffic is relatively well-behaved and not too intense, the particular multiaccess communication algorithm employed is unimportant. The best any multiaccess communication protocol can do is predict the location of packets queued in the system. If the traffic pattern is low-intensity and memoryless, there simply is not enough predictability to exploit.

3.5.2 Bursty traffic

In this section, we discuss the results of applying bursty traffic to the *GlobalTime* algorithm. Recall that in our bursty traffic model, busy and idle periods alternate independently at each node. During busy periods, packets arrive with Poisson rate 1. Furthermore, the length of a busy period is an exponential random variable with parameter `mean_burst_size`.

In the first set of simulations, depicted in Figures 3-14 through 3-17, mean delay, delay variance, and worst case delay are plotted against system arrival rate for the *GlobalTime* algorithm and three alternative distributed protocols. As in Section 3.5.1, all simulations are run with 128 nodes and 2^{15} time slots. In this first set of simulations, the mean burst size is set to 8. If the channel is 2 Gb/s, and the time to transmit a single packet is 2 μs, then a mean burst size of 8 corresponds to 4 KB. Certainly, 4 KB is a typical mean burst size, corresponding to the size of a small file or web document.

As Figure 3-14 illustrates, even under bursty traffic conditions, the *GlobalTime* algorithm performs similarly to both exhaustive and gated unlimited service disciplines with respect to mean delay. Roughly speaking, by reordering the transmission of packets, we reduce the delay of some packets only at the expense of others, thereby conserving the average delay of the system.[8] However, Figures 3-15 through 3-17

[8]While correct in the case of 100% throughput, this intuition is a little misleading when considering idle slots. For example, Figure 3-14 shows that gated limited service results in significantly greater mean delay than its alternatives. The reason is that limited service tends to push idles *forward* in the sequence of transmissions, delaying all subsequent packets. Similarly, as illustrated by the solid line in Figure 3-14, in a single queue, idle slots are pushed as far *back* as possible in the transmission sequence, resulting in lower mean delay. Why can't we achieve such low mean delay in a multi-queue system? The answer is that any distributed algorithm that minimizes worst case

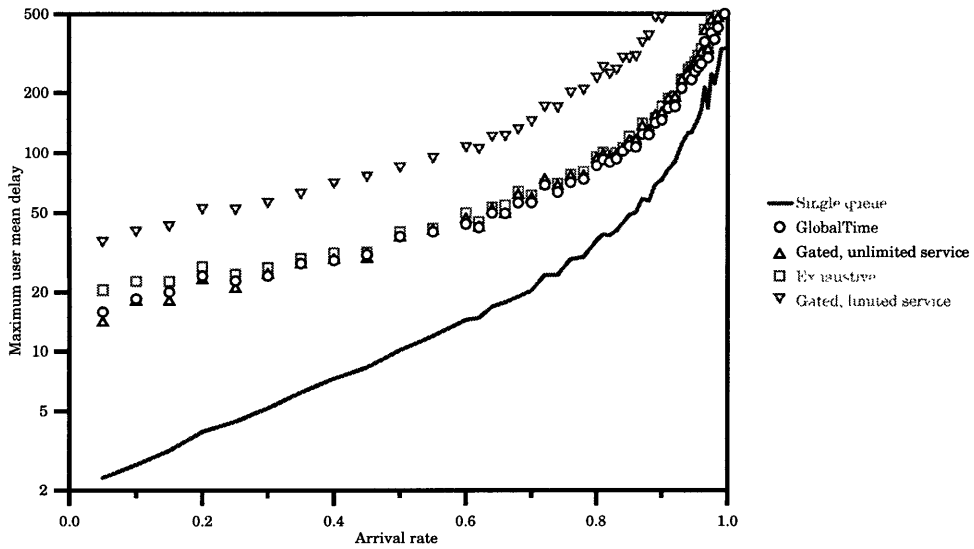


Figure 3-14: Mean delay of the user with the highest mean delay in each of 5 algorithms simulated under bursty traffic conditions with mean burst size of 8.

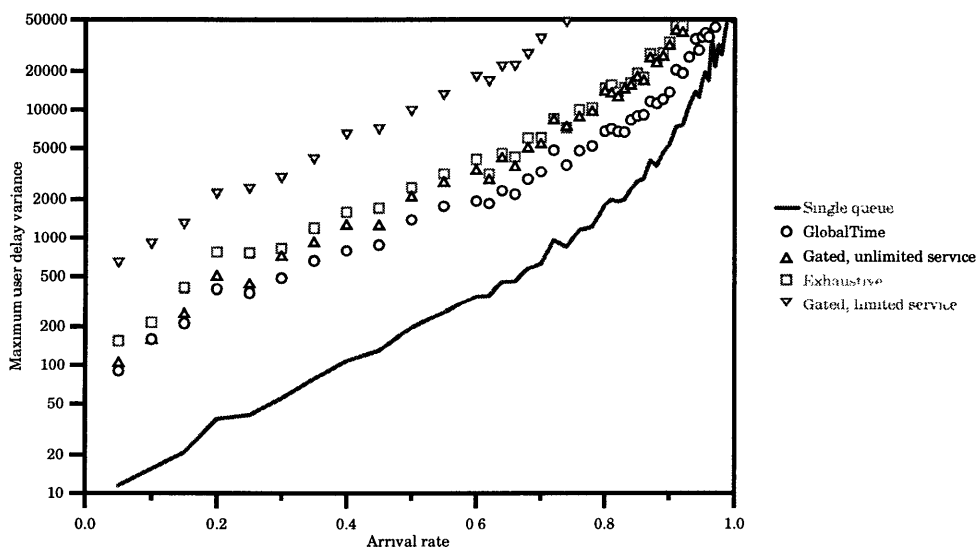


Figure 3-15: Delay variance of the user with the highest delay variance in each of 5 algorithms simulated under bursty traffic conditions with mean burst size of 8. The *GlobalTime* algorithm reduces delay variance by no less than 1/3 over a gated unlimited service discipline for $\lambda > 0.3$, and by no less than 1/2 for $\lambda > 0.8$.

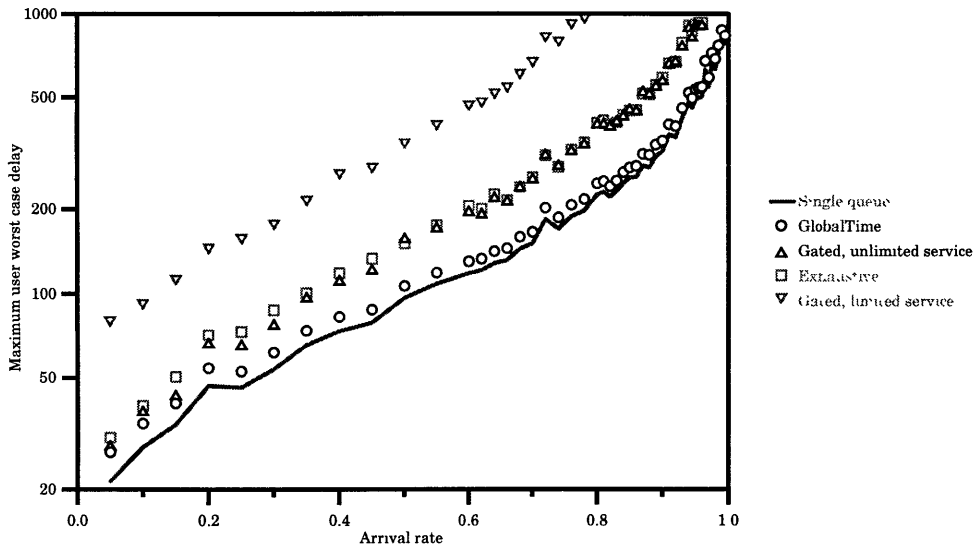


Figure 3-16: Worst case delay of the user with the highest worst case delay in each of 5 algorithms simulated under bursty traffic conditions with mean burst size of 8. The *GlobalTime* algorithm improves the delay bound over gated unlimited service by 25% or more when $\lambda > 0.3$ and by 40% or more when $\lambda > 0.8$.

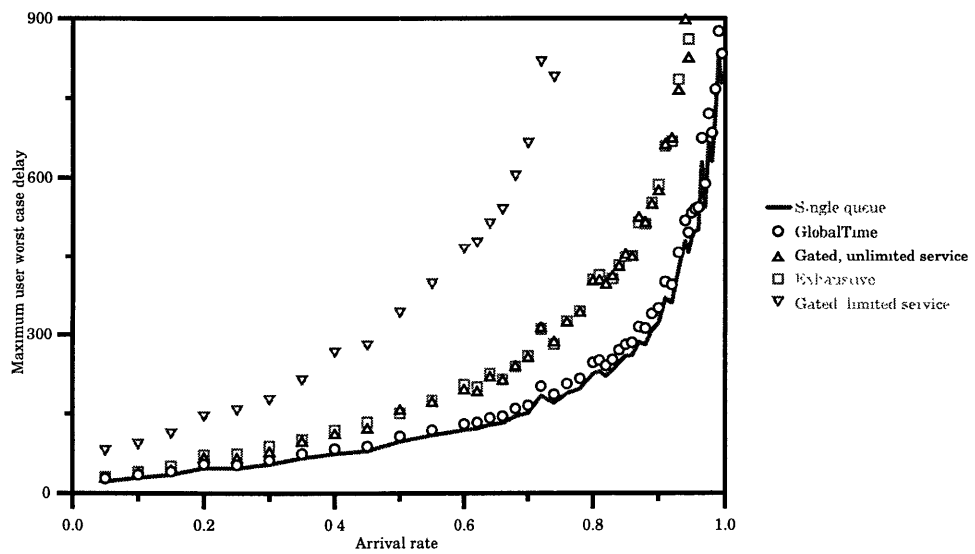


Figure 3-17: Another look at the worst case delay of the most poorly treated user in the system under bursty traffic conditions. The *GlobalTime* algorithm very nearly follows the lower bound of a single FIFO queue.

demonstrate the superiority of the *GlobalTime* algorithm over reservation-based alternatives with respect to delay variance and worst case delay. For utilization greater than 30%, the *GlobalTime* algorithm improves the delay bound over the next best strategy by at least 25%. At 80% utilization, this improvement percentage increases to at least 40%.

Observe that in Figures 3-16 and 3-17, the *GlobalTime* curve closely follows the single queue curve over all arrival rates. The single queue curve traces the *natural delay* of the system — that is, the worst case delay that would have been experienced by any packet, had all packets entered a single queue. Such natural delay results from the "natural" burstiness of the traffic; the last packet in a long data burst must experience a lengthy delay, even in a single first-in-first-out queue. The disparity between the multi-queue and single queue curves represents the *residual delay* of the distributed system — that is, any additional delay required for user coordination, or caused by a lack thereof. The graph in Figure 3-17 illustrates that with moderately bursty traffic, the residual delay of the *GlobalTime* system is near zero. In other words, the *GlobalTime* system performs as well as a single server in controlling worst case packet delay. This is surprising, because with 128 users, one might suspect that a packet could "get lost in the shuffle," suffering an enormous delay unbeknownst to the other 127 nodes. The *GlobalTime* protocol makes such an occurrence impossible. Of course, the cost of decentralization shows up in mean packet delay, as shown in Figure 3-14. However, as we have indicated, the *GlobalTime* algorithm performs no worse than other unlimited service schemes in this regard, because mean delay is largely unaffected by the particular transmission sequence permutation.

Finally, we investigate the behavior of the *GlobalTime* algorithm as the burstiness of the input stream varies. For this simulation, we again employ 128 nodes for 2^{15} time slots, and we fix channel utilization at 80%. The results of this simulation are plotted in Figures 3-18 through 3-20.

In Figure 3-18, we observe that the mean delay of the *GlobalTime* algorithm is

---

delay must periodically check each user for waiting packets; such an algorithm will produce many "early" idle slots.

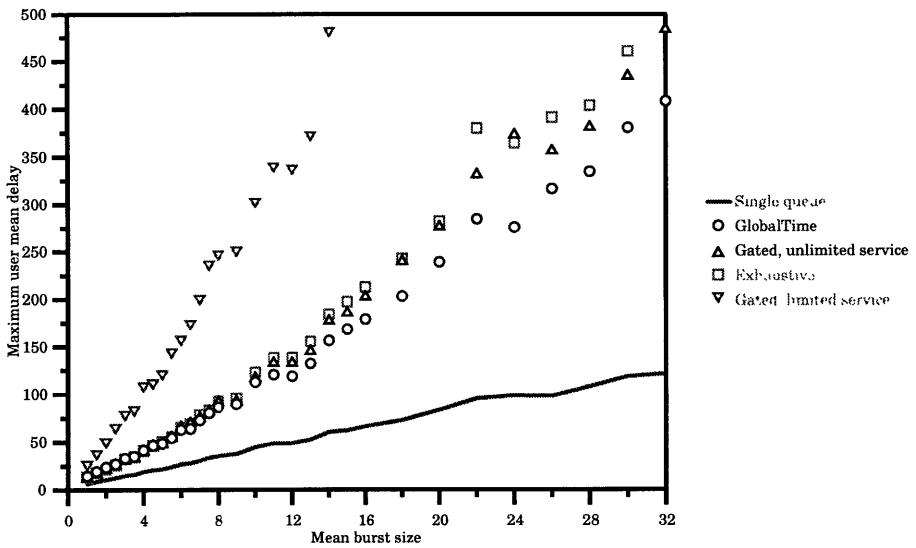


Figure 3-18: Mean delay of the user with the highest mean delay plotted against mean burst size of arriving traffic. Simulation is run on 128 nodes with $\lambda = 0.8$ for 2^{15} time units. Data points are the average of 32 trials.

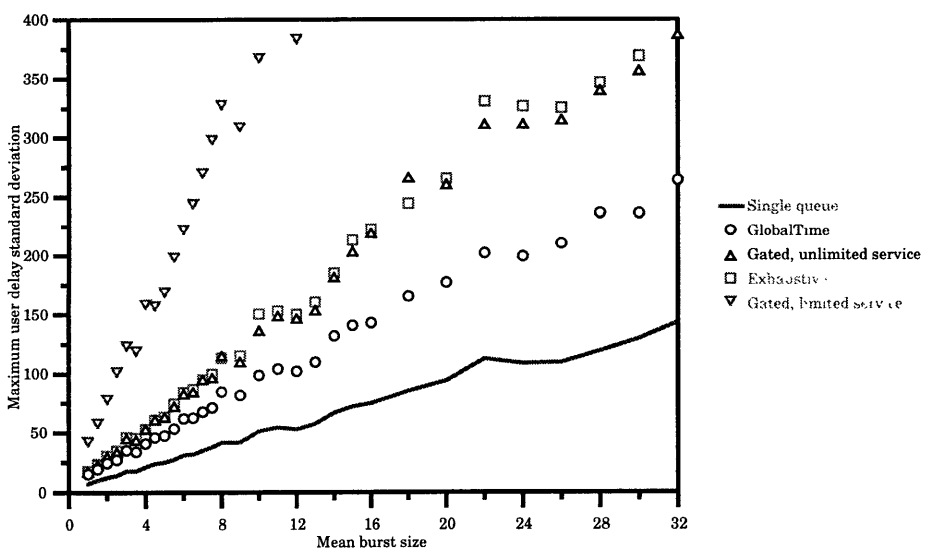


Figure 3-19: Delay standard deviation of the most poorly treated user plotted against mean burst size of arriving traffic. Simulation is run on 128 nodes with $\lambda = 0.8$.

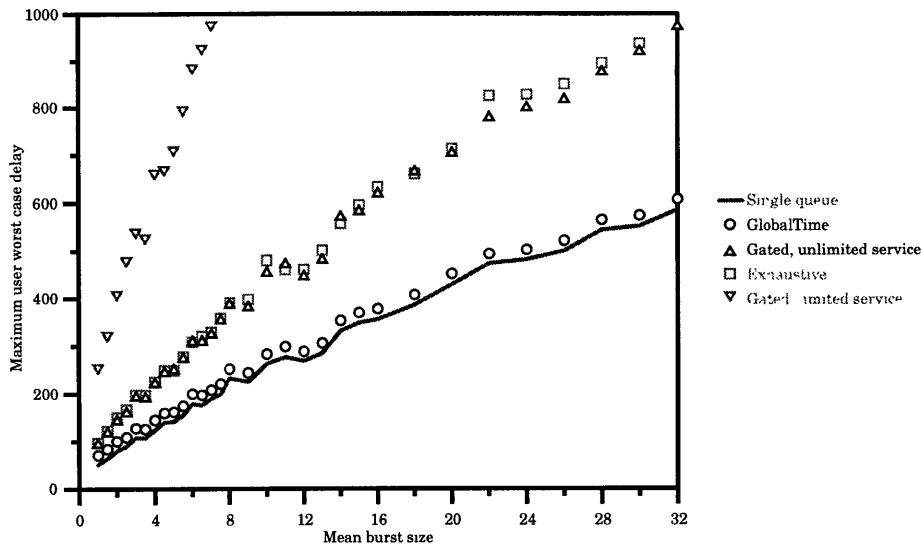


Figure 3-20: Worst case delay of the user with the highest worst case delay plotted against mean burst size of arriving traffic. Simulation is run on 128 nodes with $\lambda = 0.8$. The *GlobalTime* algorithm very nearly follows the lower bound of a single FIFO queue.

similar to the mean delay of the other unlimited service algorithms when the mean burst size is less than 12. As the mean burst size increases, however, so too does the dispersion between the three unlimited service curves; the *GlobalTime* algorithm demonstrates the best mean delay under highly bursty traffic conditions. The improvement is more marked when examining the higher moments: delay standard deviation in Figure 3-19 and worst case delay in Figure 3-20. Notice that for fixed λ, an increase in mean burst size leads to greater natural delay, represented by the solid line in the figures. Indeed, the burstier the traffic is, the greater the jitter in queue size, and thus the longer the worst case packet delay will be. However, the *GlobalTime* algorithm adds virtually no residual delay, as illustrated by the small circles tracing the path of the solid line in Figure 3-20. Though the *GlobalTime* algorithm does add some residual standard deviation, Figure 3-19 shows that *GlobalTime* improves even this second moment by about 50% over other unlimited service protocols.

Although it is tempting to claim that "The burstier the traffic, the bigger the win for the *GlobalTime* algorithm," a closer examination of the figures indicates that

Traffic type		Improvement in delay standard deviation	Improvement in worst case delay
Poisson	$\lambda = 0.6$	$< 1\%$	4%
	$\lambda = 0.8$	$< 1\%$	13%
Mean burst size = 8	$\lambda = 0.6$	27%	41%
	$\lambda = 0.8$	32%	47%

Table 3.1: Summary of some of the simulation results from this chapter. Delay is significantly lower for the *GlobalTime* algorithm compared to a gated unlimited service protocol.

such a claim is unfair. A more correct statement is that over a large range of mean burst size, the percentage improvement in delay of *GlobalTime* over its alternatives is constant. However, as we have already seen in the Poisson simulations, when the traffic is especially unbursty, any unlimited service algorithm performs fine.

3.6 Summary

The *GlobalTime* algorithm performs as well as we might have hoped. In practice, most data network traffic is bursty, and *GlobalTime* exploits this predictability in keeping the system delay bound close to a natural level. By "natural level," we mean a level necessitated by the particular timing of packet arrivals, even were the packets processed by a centralized scheduler. Future simulations might focus on increasing the number of users m, or on exploring the convergence behavior of *GlobalTime* as N approaches infinity. Table 3.1 summarizes some of the results of this chapter.

Chapter 4

Extending *GlobalTime*

In Chapter 2, we introduced a new algorithm for multiaccess communication that is practical, easily implementable, and provably efficient at transmitting data packets with high throughput and low delay. In Chapter 3, we simulated the *GlobalTime* algorithm with both Poisson and bursty traffic, and we observed up to 50% reductions in worst case delay when compared with a traditional TDM discipline in which users' queues are emptied one at a time in cyclic order. The innovation behind the *GlobalTime* algorithm is that in the process of transmitting packets, a user can also send a few bytes of piggybacked data, declaring the arrival time of its waiting head-of-line packet. Instead of passively waiting for a turn as in traditional reservation-based schemes, a user in the *GlobalTime* algorithm apprises itself of the states of the other users by actively listening to the channel and recording the piggybacked data in a local table. Because all users maintain an identical table, this simple data structure replaces the need for a centralized scheduler.

In our simulations, we assumed that the packet arrival rate at each user was the same, equal to a fraction $1/m$ of the aggregate arrival rate λ. Although this equal bandwidth assumption was made with the goal of simplicity in mind, it turns out that the particular apportionment of the channel bandwidth among the users is irrelevant to the *GlobalTime* algorithm's operation. Indeed, as we have seen, the *GlobalTime* algorithm adjusts to the location of the packets waiting in the system. Our discussion of delay and throughput in Theorems 2.4 and 2.6 never required an assumption about

the distribution of the packets in the system, only about the total number of packets, wherever they may be located. For example, if packets arrive in user A's queue twice as fast as they do in user B's, then A will receive twice as many opportunities as B to transmit during a *GlobalTime* execution; furthermore, the delay variances for users A and B will converge over time to be identical, a function of the total arrival rate of the system. The intuition behind this property of the *GlobalTime* algorithm is that because packets are served in the order they arrived (or are displaced from FCFS order by $O(1)$ time slots per packet on average), how much bandwidth each local user is using is immaterial, except as it adds to the global aggregate load. In other words, the global coordination achieved by the *GlobalTime* protocol eliminates a major roadblock in the distributed multiaccess problem: finding the waiting packets.[1]

Throughout our discussion of the *GlobalTime* algorithm, we ignored the human negotiations associated with a multiaccess network, during which customers pay for a fraction of the total bandwidth; in return, paying customers receive throughput and delay guarantees. The reason we disregarded reservations is that if users receive packets at or below their reserved rates, then all users are guaranteed service with the delay bounds plotted in Chapter 3. Because the specific distribution of the packets waiting in the system does not affect the operation of the *GlobalTime* algorithm, neither do bandwidth reservations affect its operation — assuming, of course, that users do not attempt to utilize more bandwidth than they have reserved. On the other hand, if all users receive packets at a rate much faster than they have reserved, then the system cannot guarantee anything. Worse yet, if a single user floods the system with his own packets, then all other users will suffer, as illustrated in Figure 4-1. In the figure, user X experiences an enormous burst of arriving packets, effectively locking out user Y from access to the channel. Because the *GlobalTime* algorithm

[1]For comparison, we reexamine Figure 2-3, which shows that the location of the packets greatly affects the performance of a traditional TDM protocol in the absence of idle minislots. Notice that if all arriving packets enter user 1's queue, then channel utilization is at most $1/m$. On the other hand, the *GlobalTime* algorithm quickly adjusts itself to this traffic pattern, granting user 1 the vast majority of the opportunities to transmit before long. Furthermore, by Theorem 2.6, user 1's queue will remain bounded even in the absence of idle minislots, and even if the arrival rate to user 1's queue is nearly 1.

ignores reservations, the algorithm treats user X as if it had reserved a huge fraction of the channel bandwidth, even if this is false. Therefore, one wayward user can effectively lock out his peers by severe overutilization of the channel.

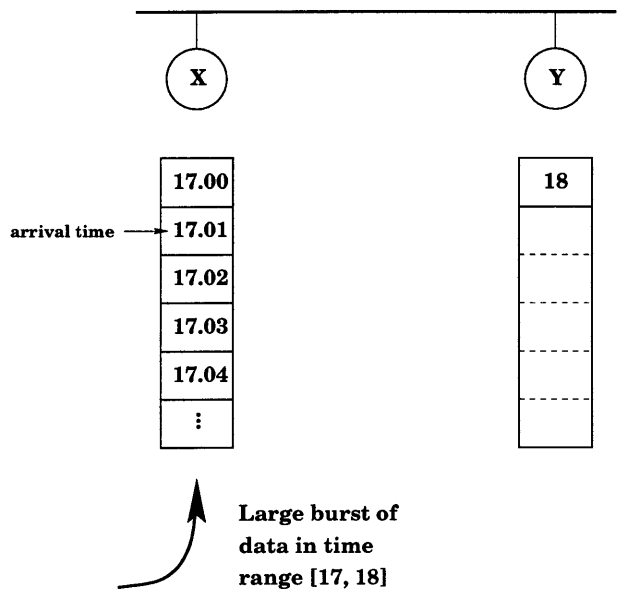


Figure 4-1: An example showing that one overzealous user X can lock out another user Y from accessing the channel, simply because all of X's packets have earlier arrival times. The extended *GlobalTime* algorithm handles this anomaly.

Although in theory the basic *GlobalTime* algorithm supports quality of service, users do not always "follow the rules" in practice. In this chapter, we propose an extension of the *GlobalTime* algorithm that safeguards performance guarantees even in the presence of misbehaving users. In Section 4.1, we invent a new virtual timestamp that will replace the arrival-based timestamps employed in the original *GlobalTime* algorithm. Section 4.2 extends the *GlobalTime* algorithm further by introducing a precaution against user lockout, a step that will also promote improved channel utilization. Finally, in Section 4.3, we list the steps of the extended *GlobalTime* algorithm more carefully, and we consider an example of the algorithm in action.

4.1 Virtual timestamps

In the original *GlobalTime* algorithm, packets are stamped with the current time as they arrive, and these timestamps, eventually declared via the multiaccess channel and recorded in a *known-time* table at each node, serve as the keys to the m identical priority queues in determining order of transmission. As we indicated in the motivation for this chapter, these arrival-based timestamps implicitly assume that the corresponding arrivals follow the rules, in the sense that the arrival rate at a node is bounded by the reserved rate at that node. In this section, we introduce an extension to the *GlobalTime* algorithm, in which packets are stamped when they arrive as before, but instead of employing a real timestamp that records the current time, the algorithm uses a virtual timestamp.

Let user A have a reserved rate λ_A, and let $T_A = 1/\lambda_A$. We can interpret T_A as the expected period between packet arrivals at user A, assuming packets enter A's queue at the reserved rate. In the extended *GlobalTime* algorithm, we will regulate the traffic transmitted from node A. We imagine *credits* periodically arriving at A, one every T_A time units. The incoming credits are stored in a *credit bucket* up to a maximum of W; any credits that arrive when the bucket is already full are discarded. Furthermore, we allow the bucket to contain an arbitrarily negative number of credits, signifying a credit shortage. The idea behind this credit scheme is that incoming credits will *validate* waiting packets; without a credit to validate its presence in the system, a packet receives lower priority for transmission.[2] This makes sense, because credits arrive at a node at the node's reserved rate; if user A attempts to overutilize the channel, A's unvalidated packets will receive lower priority, and as a result, the remaining users will not be locked out.

The circulation of credits is a theoretical fiction in the extended *GlobalTime* algorithm. In reality, user A maintains an additional state N^A_{credit}, where

$$-\infty < N^A_{\text{credit}} < W \ .$$

[2]Credit-based protocols are often used to shape traffic in data networks. In the field of network flow control, such approaches are often called "leaky bucket" schemes.

The value of N^A_{credit} is incremented every T_A time units, up to a maximum of W. Moreover, whenever a packet arrives at node A, N^A_{credit} is decremented. The value of N^A_{credit}, if positive, represents the number of packets that may be transmitted immediately while still keeping A at or below its reserved rate. On the other hand, if A is generating too many packets, N^A_{credit} may be negative as well. Notice that the fact that N_{credit} can never exceed W ensures that a user cannot stockpile credits indefinitely and eventually flood the system with validated packets. Indeed, the bucket threshold of W serves as a regulator of the burstiness of the transmitted traffic.[3]

In addition to N^A_{credit}, user A maintains a state t^A_{next}, which records the time at which the next virtual credit is set to arrive. Of course, t^A_{next} is initialized to 0 and is increased by T_A whenever t^A_{next} is equal to the current time; furthermore, whenever t^A_{next} is increased, N^A_{credit} is incremented. The point of all this discussion about implementation is that only two new states, the real number t_{next} and the integer N_{credit}, are required to maintain the fictional credit bucket scheme locally at each node.

When a packet P arrives at A, N^A_{credit} is decremented, as indicated earlier. Then P receives a virtual timestamp according to the following criterion:

1. If $N^A_{\text{credit}} \geq 0$, then set P's timestamp equal to the current time.
2. If $N^A_{\text{credit}} < 0$, then set P's timestamp equal to $t^A_{\text{next}} - T_A\left(1 + N^A_{\text{credit}}\right)$.

What is a virtual timestamp? Like a real timestamp, it is recorded immediately after a packet arrives in the system. In Case 1, at packet P's arrival time, we have $N^A_{\text{credit}} \geq 1$, indicating that P can be validated by a waiting credit instantaneously upon P's arrival. Therefore, Case 1 sets P's virtual timestamp to P's arrival time, as in the original *GlobalTime* algorithm. In fact, if packets were to always arrive at times

[3]In some multiaccess contexts, a user not only reserves a particular rate, but also specifies more descriptive arrival statistics: for example, maximum burst size. In this case, the extended *GlobalTime* algorithm could make W a function of each user's individual contract with the system, greater for higher paying customers. In this chapter, however, we assume W is a global constant.

when they can be instantly validated, then the behavior of the extended *GlobalTime* algorithm would be identical to that of the original. This makes sense, because we have already proved in Chapter 2 the near optimality of the original *GlobalTime* algorithm if all users receive packets at or below their reserved rates.

In Case 2, at packet P's arrival time, we have $N^A_{\text{credit}} \leq 0$, expressing that the credit bucket is currently empty. The fact that P cannot be validated immediately by a waiting credit means that P entered A's queue too early. In other words, packet P arrived in A's queue faster than expected, given A's reserved rate. With what time should we stamp packet P? Case 2 stamps packet P with the time P *should have arrived* in the system, so as to conform with A's reserved rate. The innovation here is that if P arrives too early, we measure P's delay not from its actual arrival time, but from its *virtual arrival time*: the time at which P's corresponding credit arrived at A.

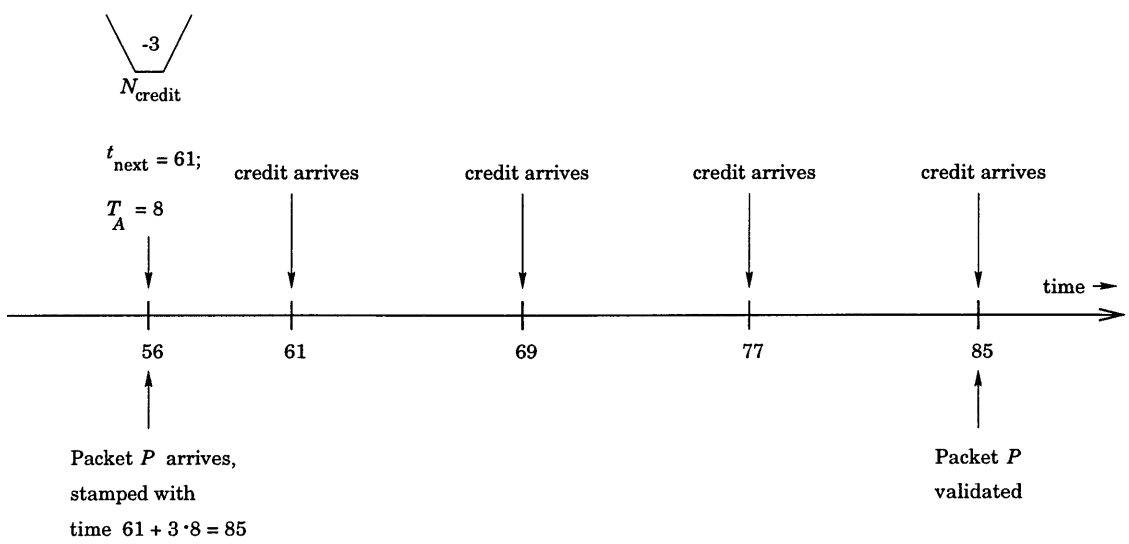


Figure 4-2: An illustration of Case 2 of the rule generating virtual timestamps. If packet P arrives too early, it is stamped with the time that it "should have" arrived, calculated from the reserved rate.

Suppose that at the arrival time a_P of packet P at node A, we have $N^A_{\text{credit}} = 0$. Then the virtual credit that validates P is scheduled to arrive at time t^A_{next}; thus, P

is stamped with t^A_{next}. The calculation gets slightly more complicated if $N^A_{\text{credit}} = -1$ at time a_P. In this case, the credit shortage indicates that one yet-to-arrive credit has already been assigned to a packet waiting ahead of P. Thus, P must wait for two credits to arrive before being validated. Because the credit interarrival period is T_A, P is stamped with $t^A_{\text{next}} + T_A$. In general, in Case 2, if $N^A_{\text{credit}} = -k$ at time a_P, then P is stamped with $t^A_{\text{next}} + kT_A$, the time at which P will be validated by an incoming credit.[4]

In the original *GlobalTime* algorithm, if on user A's turn, A's queue has no next-in-line packet, a dummy timestamp representing the current time is sent. The idea is that if A has no waiting packet to declare, A pretends as if it has a waiting packet that arrived at that very instant; thus, A sends the current time in lieu of a real packet arrival time. In the extended *GlobalTime* algorithm, dummy timestamps are generated similarly. If A has no next-in-line packet to declare on its turn and $N^A_{\text{credit}} \geq 0$, then A sends the current time as in original *GlobalTime*. On the other hand, if A has no next-in-line packet and $N^A_{\text{credit}} < 0$, then as before, A pretends as if it has a packet that just arrived. Such a packet could not be immediately validated, because the credit bucket is empty, and therefore would be stamped with the time at which the associated credit is scheduled to arrive. Thus, in this case, a dummy timestamp of $t^A_{\text{next}} - T_A\left(1 + N^A_{\text{credit}}\right)$ is sent.

The original *GlobalTime* algorithm minimized the deviation of a transmission sequence from the optimal sequence. Because we have not touched the basic *GlobalTime* framework, this minimization of inversions must remain true, though the optimal sequence is now different. We still want a sequence that minimizes the worst case delay of any packet. However, we now measure delay starting not from a packet's real arrival time, but from its virtual arrival time, the time at which the packet is validated by a credit. With respect to this new optimal sequence, the $O(mn)$ inversions bound clearly must hold as before.

[4]In Cases 1 and 2 for computing the virtual timestamp, note that N_{credit} is first decremented when a packet arrives, and then the rules are followed based on this updated value of N_{credit}.

4.2 User lockout

As we have seen, a virtual timestamp simply indicates P's arrival time if P entered A's queue on time or later than expected; on the other hand, it indicates the time at which P *should have arrived* if P entered A's queue too early. The notion of virtual timestamp circumvents the overzealous user by measuring packet delays not from their real arrival times but from their virtual arrival times, equal only if the user has obeyed the rules.

If we run the original *GlobalTime* algorithm using virtual timestamps in lieu of real timestamps, we obtain good delay performance. In particular, a transmission sequence generated by the extended *GlobalTime* algorithm has $O(mn)$ inversions with respect to the optimal sequence in the worst case. However, we may encounter a throughput anomaly depicted in Figure 4-3.

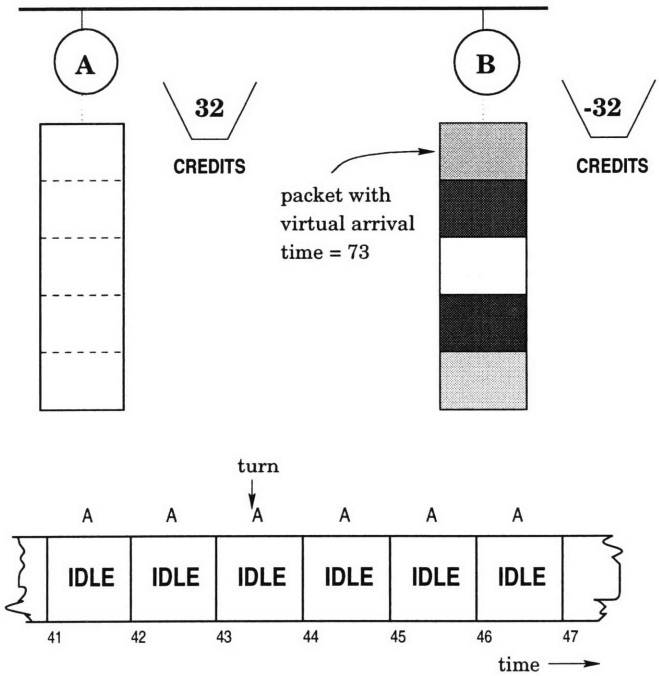


Figure 4-3: An example showing that the extended *GlobalTime* algorithm can experience user lockout. Although user A has an empty queue and user B is backlogged, user A continually gets turns.

In the figure, although node B is backlogged, node A — with no waiting packets — gets a turn every time slot. What causes this anomaly? Because node A has $N^A_{\text{credit}} > 0$, the next arriving packet will be validated immediately. If on its turn, A has no packet to transmit, then a dummy timestamp with the current time is sent. In the original *GlobalTime* algorithm, this strategy posed no problem because if the current time was sent, then *known-time*$[i]$ < *known-time*$[A]$ for all other users i immediately after A's turn; that is, the current time served as an upper bound on the values in the *known-time* table. The problem in the extended *GlobalTime* algorithm is that virtual timestamps can exceed the current time. In the figure, the head-of-line packet at node B perhaps "should have" arrived a long time from now, particularly if B's reserved rate is very low. Therefore, even after *known-time*$[A]$ is updated with the current time, it will still be the case that *known-time*$[A]$ < *known-time*$[B]$, and A will receive another turn. In the worst case, this throughput anomaly will continue until the current time "catches up" with the validation time of B's head-of-line packet; meanwhile, valuable channel bandwidth is wasted. Of course, from a delay perspective, it is fair for A to continually receive turns, because B's packet delays will be measured from the time the packets "should have" arrived anyway. On the other hand, the channel is clearly not being well-utilized.

In this section, we suggest a simple solution to the channel utilization problem. If on user A's turn, A has no next-in-line packet and sends a dummy timestamp, then the algorithm *freezes* user A for a period of time Δ.

Definition 4.1 *A user A has been* **frozen** *during an execution of the extended* GlobalTime *algorithm if the protocol ignores A when it uses the* known-time *table to decide which user's turn will be next.*

The idea is that if a user has no packets in its queue, it must wait at least time Δ before getting another turn. Once A has been unfrozen, its virtual timestamp again is active in the $\min_i\{$*known-time*$[i]\}$ phase of the *GlobalTime* algorithm. Because A is frozen for some time, user B is not locked out of the protocol.

How large should we make Δ? There is no single, definitive answer to this question.

The tradeoff between utilization and delay makes the choice situation-specific. The larger we make Δ, the greater the worst case number of inversions in the transmission sequence. The smaller we make Δ, however, the lower the channel utilization in the worst case.

One proposed solution to the choice of Δ that will *not* work is to make $\Delta = T_i$ (or some other function of λ_i) for each node i. The reason we might propose such an idea is that if user A reserves less bandwidth than user B, then we might expect to "get away with" freezing A for a long time more easily than freezing B for a long time. A problem with this idea is that it needlessly couples delay with throughput. If $\Delta = T_i$ for each node i, then in the worst case, A will experience greater delay than B, simply because its reserved rate is lower. By contrast, our goal is to equalize the delay experienced by the system's users; users pay for a guaranteed fraction of the channel bandwidth, not for a guaranteed delay bound.[5] Another problem with setting $\Delta = T_i$ is that it needlessly couples reserved rate with actual arrival rate. In our worst case analytical model, it is inappropriate to infer anything about A's actual arrival rate from A's reserved rate; an adversary can easily defeat any such inference by making A misbehave, either by receiving far too many or far too few packets for its reserved rate.

We conclude with an important observation about the choice of Δ.

Theorem 4.1 *Let β be the length of an idle minislot. Then if $\Delta \leq (m-1)\beta$, user lockout can occur in the worst case.*

Proof: If $\Delta \leq (m-1)\beta$, then the freeze time Δ is not large enough to prevent an execution in which $m-1$ of the users each obtain opportunities to transmit one after the other in cyclic fashion, but never have any packets to send. The remaining user may have a backlogged queue, but will be locked out. □

The proof of Theorem 4.1 is illustrated in Figure 4-4. Notice that backlogged user D never gets an opportunity to transmit, while the algorithm repeatedly cycles

[5]Delay is simply a function of the aggregate load, and is not reserved as is bandwidth in our model. Of course, customer dollars could pay for delay guarantees under a different set of assumptions, but in any case, the important point is that we want throughput and delay to be decoupled.

through users A, B, and C, giving them each many opportunities to transmit. The problem is that with $\Delta = (m-1)\beta$, the $m-1$ idle users effectively lock out the single backlogged user, needlessly wasting the channel bandwidth.

4.3 The extended *GlobalTime* algorithm

To summarize, the extended *GlobalTime* algorithm works as follows:

1. Each node $1, 2, \ldots, m$ has a clock. The nodes' clocks are synchronized and are initialized to 0.

2. Each node i has a predefined reserved rate λ_i. A state t^i_{next} is initialized to 0, and is increased by $T_i = 1/\lambda_i$ whenever the clock reads t^i_{next}. Moreover, each node i has an integer state N^i_{credit}, initialized to 0, and incremented whenever the clock reads t^i_{next}.

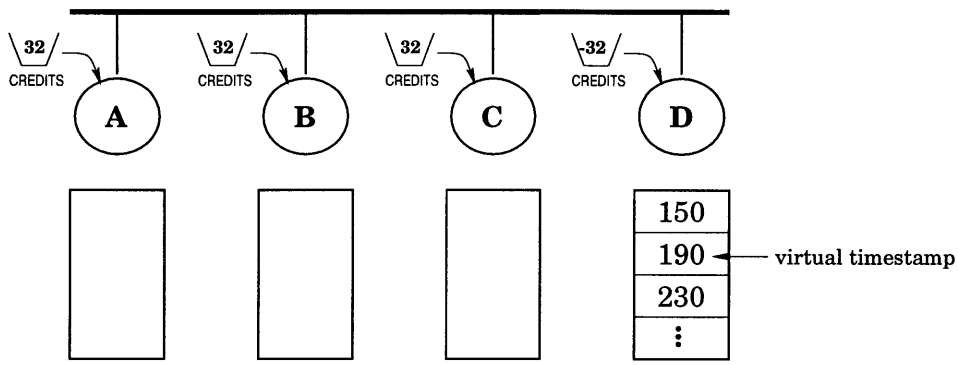
3. When a packet P arrives in node i's queue, N^i_{credit} is decremented. Then P is immediately timestamped according to the following criterion:

   - If $N^i_{\text{credit}} \geq 0$, then set P's timestamp equal to the current time.
   - If $N^i_{\text{credit}} < 0$, then set P's timestamp equal to $t^i_{\text{next}} - T_i\,(1 + N^i_{\text{credit}})$.

4. Each node i has a set of m states *known-time*$[j]$, one for each node j. Furthermore, each node i has a set of m states *unfreeze-time*$[j]$, one for each node j. All the *known-time* and *unfreeze-time* states are initialized to 0.

5. At the beginning of a time slot, let $\mathcal{F}$ be the set of nodes k such that *unfreeze-time*$[k]$ is no greater than the current time. Next, node i determines the value of $j \in \mathcal{F}$ such that *known-time*$[j] \leq$ *known-time*$[k]$ for all $k \in \mathcal{F}$. If $i = j$, then node i has gained access to the channel. (Ties are broken according to predetermined rules. Moreover, if $\mathcal{F} = \emptyset$, then predetermined rules decide which user gains access to the channel.)

(a)



(b)

time	A	B	C	D
	known-time			
39	36	37	38	150
40	39	37	38	150
41	39	40	38	150
42	39	40	41	150
43	42	40	41	150
44	42	43	41	150
⋮				

(c)

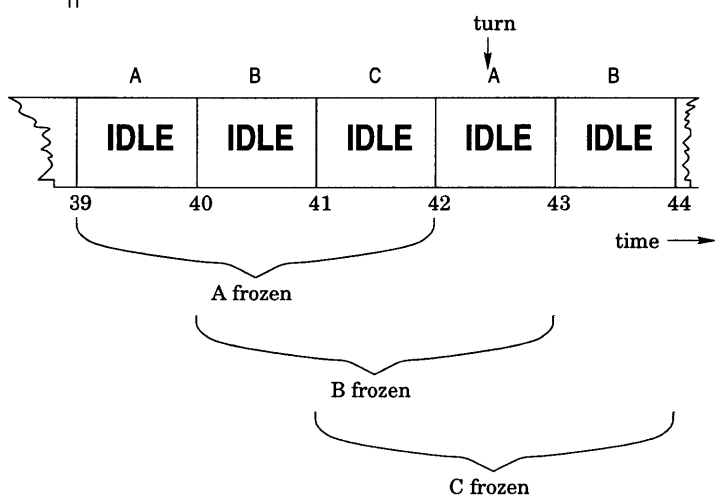


Figure 4-4: An illustration of the theorem in the case where $m = 4$, $\beta = 1$, and $\Delta = 3$. In (a), the queues at nodes A, B, and C are empty, but the queue at node D is backlogged. In (b), the evolution of the *known-time* table is shown. Notice that because D has a credit shortage, its *known-time* is never the minimum value in the table. In (c), a fragment of the transmission sequence is depicted. Because $\Delta = (m-1)\beta$, the three idle users are never all frozen at the beginning of a time slot, so D never receives an opportunity to transmit.

6. On its turn, node i transmits the first data packet in its queue. In addition to actual data, the transmitted frame also contains piggybacked information — namely, the *timestamp of the next packet* in node i's queue (if it exists). The transmitted frame also contains a *freeze* bit set to 0.

7. However, if the transmission has exhausted node i's queue, the algorithm instead inserts the current time in this extra field if $N^i_{\text{credit}} \geq 0$, and inserts $t^i_{\text{next}} - T_i\,(1 + N^i_{\text{credit}})$ otherwise. On the other hand, if on node i's turn, i has no packets to send at all, then i sends a dummy packet (augmented with either the current time or $t^i_{\text{next}} - T_i\,(1 + N^i_{\text{credit}})$, as before). In either case, the transmitted frame also contains a *freeze* bit set to 1.

8. When a packet P is being transmitted from node i, all nodes read the timestamp piggybacked on P, and then update their local value of *known-time*$[i]$. Furthermore, if the *freeze* bit is set to 1, then all nodes update *unfreeze-time*$[i]$ to be Δ added to the current time, where Δ is the globally defined *freeze period*.[6]

4.3.1 An example execution fragment

Figure 4-5 depicts an example execution fragment of the extended *GlobalTime* algorithm. In the figure, three users A, B, and C compete for access to the channel, and $\lambda_A = \lambda_C = 0.1$, and $\lambda_B = 0.5$. The time to transmit a single packet is 5 units, and the length of an idle minislot is $\beta = 1$. The freeze period Δ is 10.

The execution fragment starts at time $t = 40$. Because *known-time*$[A] = 31$ is the minimum value in the *known-time* table, user A receives a turn and transmits its head-of-line packet. Because this transmission exhausts A's queue and because $N^A_{\text{credit}} = 1$, the new value of *known-time*$[A]$ is 40, the current time. Furthermore, user A remains frozen for $\Delta = 10$ time units — that is, until $t = 50$.

At time $t = 45$, user B now receives a turn. B's head-of-line packet is transmitted, and because B has no next-in-line packet and $N^B_{\text{credit}} = 3$, the new value of *known-*

[6]For simplicity, we assume zero clock skew and propagation delay. In practice, we can easily modify the algorithm to handle these issues.

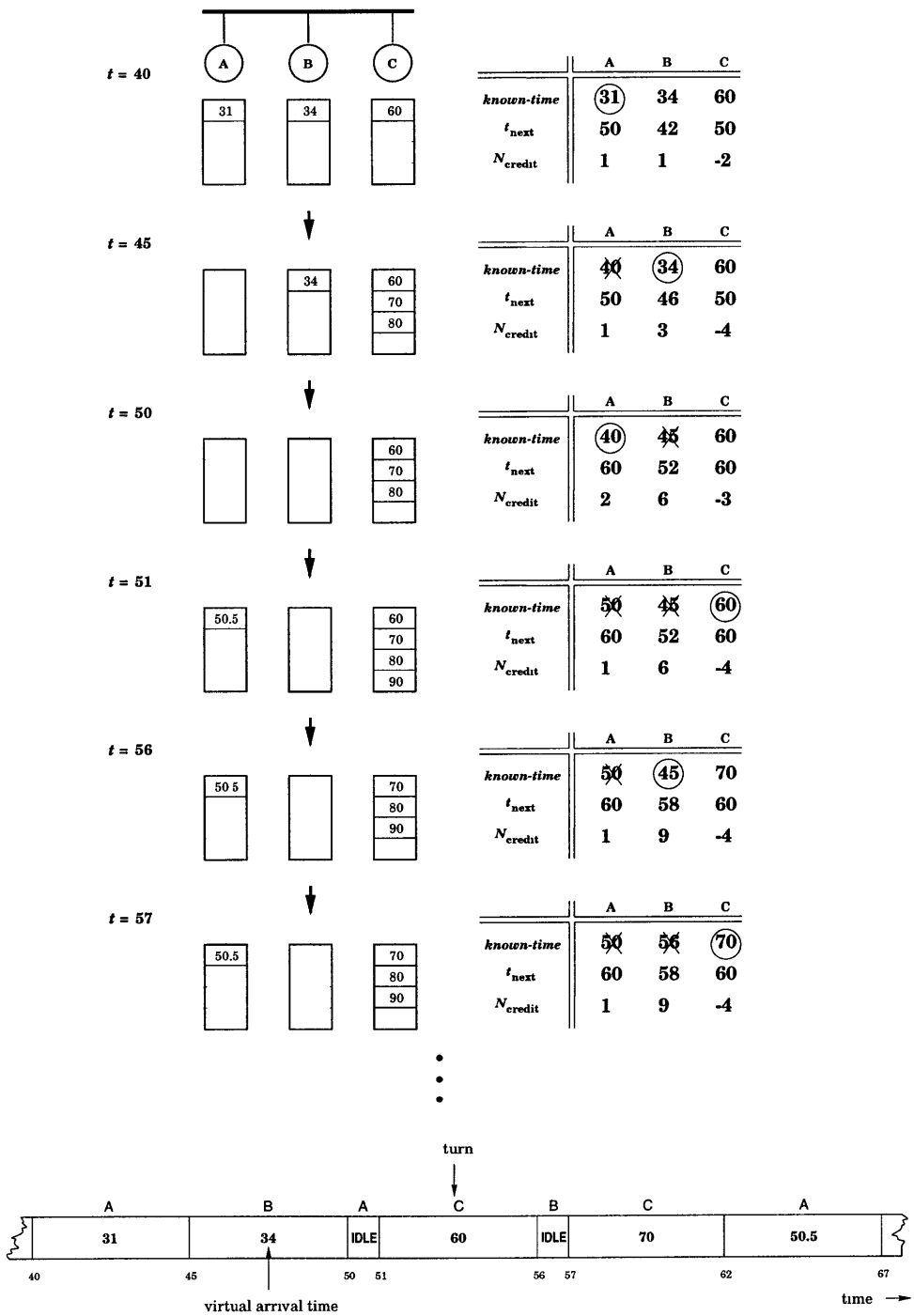


Figure 4-5: A sample execution fragment of the extended *GlobalTime* algorithm. For each time step, the current user states are shown. In this example, $\lambda_A = \lambda_C = 0.1$, and $\lambda_B = 0.5$. The time to transmit a single packet is 5 units, and the length of an idle minislot is $\beta = 1$. The freeze period Δ is 10. Packets are labeled with their virtual arrival times.

time[B] is set to the current time, 45. User B is frozen for $\Delta = 10$ time units, or until $t = 55$.

At time $t = 50$, node A is unfrozen. Because *known-time*[A] $= 40$ is the minimum value in the *known-time* table, A receives an opportunity to transmit. User A's queue is empty, however, so A sends a dummy packet augmented with a virtual timestamp. Because $N^A_{\text{credit}} = 2$, this virtual timestamp is simply the current time, 50. User A is frozen for $\Delta = 10$ time units.

The execution continues at $t = 51$, after the idle minislot of length $\beta = 1$. Because A and B are still frozen, node C gets an opportunity to transmit. Notice that the head-of-line packet at C has arrived too early, but the extended *GlobalTime* algorithm enables overzealous users like C to exploit excess bandwidth when other users are idle. Because the next-in-line packet at C has virtual timestamp 70, all users update *known-time*[C] to be 70.

At time $t = 56$, user B is unfrozen, and because *known-time*[B] $= 45$ is the minimum value in the *known-time* table, B receives a turn. Because B's queue is empty and $N^B_{\text{credit}} = 9$, *known-time*[B] is updated to 56, the current time. Furthermore, B is frozen. Finally, at time $t = 57$, both users A and B are again frozen, so C gets another opportunity to transmit from its backlogged queue.

Notice that the packet that arrived in user A's queue at time 50.5 will be inverted relative to two packets transmitted from node C with virtual timestamps 60 and 70. These inversions are a direct result of the freeze period discussed in Section 4.2. As we have stated, setting the freeze period Δ requires a tradeoff between worst case delay and channel utilization.

Chapter 5

Conclusion

In this thesis, we have presented a new distributed scheduling algorithm, *GlobalTime*, that supports quality of service in multiaccess networks. Our algorithm eliminates the need for a centralized scheduler and a separate control channel by coordinating transmissions via packet headers. The *GlobalTime* algorithm exploits the full power of the multiaccess channel architecture, by actually using the ability of nodes to hear all the information being transmitted on the channel. Furthermore, our algorithm provides fairness and bandwidth reservation in an integrated services environment, while also achieving high throughput. With moderately bursty traffic, *GlobalTime* has a worst case delay nearly identical to that produced by the optimal, centralized algorithm.

There are still many research questions that remain to be answered:

Variable packet size When packets have varying lengths, it is sometimes preferable to serve a small packet before a larger one with an earlier arrival time. How should we modify the *GlobalTime* algorithm to handle variable length packets efficiently?

Extension to multiple channels How can the *GlobalTime* algorithm be generalized to the multiple channel model? For example, this extension would apply to wavelength division multiplexing (WDM), where each wavelength corresponds to a single multiaccess channel.

Priority service How can we incorporate different classes of service into the *Global-Time* algorithm? What throughput and delay characteristics can be guaranteed to high and low priority customers?

Robustness As discussed in this thesis, the *GlobalTime* algorithm requires perfect synchronization among the users' clocks, and ceases to work properly if even one node fails. What modifications to the *GlobalTime* algorithm can be made in order to ensure robustness when faced with clock skew or node stopping failures?

Future investigation of the *GlobalTime* algorithm will proceed in the direction of answering these important remaining questions.

Appendix A

Simulation Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define M_MIN 64
#define M_MAX 128
#define M_INC 64
#define N_MIN 32768
#define N_MAX 32768
#define N_INC 1
#define LAMBDA_MIN 0.6
#define LAMBDA_MAX 0.9
#define LAMBDA_INC 0.1
#define TRIALS 32
#define BETA 0.01
#define BURST_MIN  4.0
#define BURST_MAX  8.0
#define BURST_INC  4.0
#define NULL 0
#define TRAFFIC_SWITCH 1

struct heap_element{
  int val;
  double key;
};
struct linked_list{
  double data;
  struct linked_list *next;
};
struct linked_list_2{
  double data1;
  double data2;
  struct linked_list_2 *next;
};

typedef struct linked_list ELEMENT;
typedef struct linked_list_2 ELEMENT_PAIR;
typedef ELEMENT *LINK;
```

```
typedef ELEMENT_PAIR *LINK2;

/*************************************************************************
 * The main() procedure is simply a set of nested loops that runs the simulation
 * through a range of m, N, lambda, and possibly mean_burst_size, depending on the
 * nature of the traffic. The user specifies (in the #define statements above) a
 * range of m, N, and lambda. He specifies whether he wants bursty or Poisson
 * traffic. If bursty, he specifies the mean burst size desired. The parameter m
 * represents the number of nodes in the simulation, N represents the amount of time
 * (in units of cell service time) for the simulation, and lambda represents the
 * normalized arrival rate, where lambda = 1.0 is the maximum stable arrival rate to
 * the system.
 *************************************************************************/

main(int argc, char *argv[])
{
 int m, n, lambda_count, burst_count;
 double lambda_count_max = ((LAMBDA_MAX - LAMBDA_MIN) / LAMBDA_INC) + 1.0;
 double burst_count_max = ((BURST_MAX - BURST_MIN) / BURST_INC) + 1.0;
 void run_simulation(int, double, int, double, char *);

 srandom(time(0) * getpid());
 for (lambda_count = 0; lambda_count < (int)lambda_count_max; ++lambda_count){
  for (m = M_MIN; m <= M_MAX; m += M_INC){
   for (n = N_MIN; n <= N_MAX; n += N_INC){
       if (TRAFFIC_SWITCH == 0)
        run_simulation(m, (LAMBDA_MIN + lambda_count * LAMBDA_INC), n,
                        (BURST_MIN + burst_count * BURST_INC), argv[1]);
       else{
        for (burst_count = 0; burst_count < (int)burst_count_max; ++burst_count)
         run_simulation(m, (LAMBDA_MIN + lambda_count * LAMBDA_INC), n,
                         (BURST_MIN + burst_count * BURST_INC), argv[1]);
       }
   }
  }
 }
}

/*************************************************************************
 * The run_simulation() procedure is the skeleton of the code. The code for
 * run_simulation() calls all the functions necessary for the simulation. It takes a
 * value of m, N, lambda, and mean_burst_size, and generates a single data point. The
 * single data point is generated by averaging together a fixed number of trials of
 * the simulation (stored in the constant TRIALS). In a single trial, a set of
 * arrivals is generated according to the desired traffic type (Poisson, or bursty
 * with a given mean burst size). The same set of traffic is queued in a single
 * queue, and also in m distinct queues. The single queue simulation is run, followed
 * by four different multiqueue simulations. One is the GlobalTime simulation, and
 * the other three are alternative reservation-based disciplines: gated limited
 * service, gated unlimited service, and exhaustive. Mean delay, delay variance, and
 * worst case delay are tracked for both the average user and the worst-case user in
 * all simulations. The run_simulation() procedure prints the results to the desired
 * file.
 *************************************************************************/
```

```
void run_simulation(int m, double lambda, int n, double mean_burst_size,
                    char *simulation_output)
{
  FILE *ofp;
  double *arrival_array, *arrivals_sorted_by_queue;
  double single_queue_results[3], cumulative_single_queue_results[3];
  double GLOBALTIME_results[6], cumulative_GLOBALTIME_results[6];                    100
  double limited_service_results[6], cumulative_limited_service_results[6];
  double gated_results[6], cumulative_gated_results[6];
  double exhaustive_results[6], cumulative_exhaustive_results[6];
  int *queue_offsets;
  int trial_number, number_of_arrivals, i;

  int compare_reals(const void *, const void *);
  double *create_poisson_arrival_array(double *, double, int);
  void create_arrivals_sorted_by_queue(double *, int *, double *, int, int);
  double *create_bursty_input(double *, int *, double, int, int, double);            110
  void perform_single_queue_simulation(double *, double *, int, int);
  void perform_GLOBALTIME_simulation(double *, int *, double *, int, int, int);
  void perform_reservation_based_simulation(double *, int *, double *, int, int, int, int);

  queue_offsets = calloc(m, sizeof(int));
  for (i = 0; i < 6; ++i){
    cumulative_gated_results[i] = cumulative_limited_service_results[i] = 0.0;
    cumulative_exhaustive_results[i] = cumulative_GLOBALTIME_results[i] = 0.0;
    cumulative_single_queue_results[i] = 0.0;
  }                                                                                   120

  for (trial_number = 0; trial_number < TRIALS; ++trial_number){
    if (TRAFFIC_SWITCH == 1){
      arrivals_sorted_by_queue = create_bursty_input(arrivals_sorted_by_queue, queue_offsets,
                                                     lambda, n, m, mean_burst_size);
      number_of_arrivals = (int)(arrivals_sorted_by_queue[0]);
      arrivals_sorted_by_queue += 1;
      arrival_array = calloc(number_of_arrivals, sizeof(double));
      for (i = 0; i < number_of_arrivals; ++i)
          arrival_array[i] = arrivals_sorted_by_queue[i];                             130
      qsort(arrival_array, number_of_arrivals, sizeof(double), compare_reals);
    }
    else{
      arrival_array = create_poisson_arrival_array(arrival_array, lambda, n);
      number_of_arrivals = (int)(arrival_array[0]);
      arrival_array += 1;
      arrivals_sorted_by_queue = calloc(number_of_arrivals, sizeof(double));
      create_arrivals_sorted_by_queue(arrival_array, queue_offsets, arrivals_sorted_by_queue,
                                      number_of_arrivals, m);
    }                                                                                 140
    perform_single_queue_simulation(arrival_array, single_queue_results, number_of_arrivals, n);
    for (i = 0; i < 3; ++i)
      cumulative_single_queue_results[i] += single_queue_results[i];
    perform_GLOBALTIME_simulation(arrivals_sorted_by_queue, queue_offsets,
                                  GLOBALTIME_results, number_of_arrivals, m, n);
    perform_reservation_based_simulation(arrivals_sorted_by_queue, queue_offsets,
```

```
                                        limited_service_results, number_of_arrivals,
                                        0, m, n);
  perform_reservation_based_simulation(arrivals_sorted_by_queue, queue_offsets,
                                        gated_results, number_of_arrivals, 1, m, n);
  perform_reservation_based_simulation(arrivals_sorted_by_queue, queue_offsets,
                                        exhaustive_results, number_of_arrivals, 2, m, n);
  for (i = 0; i < 6; ++i){
    cumulative_GLOBALTIME_results[i] += GLOBALTIME_results[i];
    cumulative_gated_results[i] += gated_results[i];
    cumulative_limited_service_results[i] += limited_service_results[i];
    cumulative_exhaustive_results[i] += exhaustive_results[i];
  }
  if (TRAFFIC_SWITCH == 1)
    arrivals_sorted_by_queue -= 1;
  else arrival_array -= 1;
  free(arrival_array);
  free(arrivals_sorted_by_queue);
}
free(queue_offsets);

ofp = fopen(simulation_output, "a");
if (TRAFFIC_SWITCH == 0){
  printf("%i, %i, %.3f, ", m, n, lambda);
  fprintf(ofp, "%i, %i, %.3f, ", m, n, lambda);
}
else{
  printf("%i, %i, %.3f, %.1f, ", m, n, lambda, mean_burst_size);
  fprintf(ofp, "%i, %i, %.2f, %.1f, ", m, n, lambda, mean_burst_size);
}
for (i = 0; i < 6; ++i){
  printf("%.3f, %.3f, %.3f, %.3f, %.3f, ",
          cumulative_single_queue_results[i % 3] / TRIALS,
          cumulative_GLOBALTIME_results[i] / TRIALS,
          cumulative_limited_service_results[i] / TRIALS,
          cumulative_gated_results[i] / TRIALS,
          cumulative_exhaustive_results[i] / TRIALS);
  fprintf(ofp, "%.3f, %.3f, %.3f, %.3f, %.3f, ",
           cumulative_single_queue_results[i % 3] / TRIALS,
           cumulative_GLOBALTIME_results[i] / TRIALS,
           cumulative_limited_service_results[i] / TRIALS,
           cumulative_gated_results[i] / TRIALS,
           cumulative_exhaustive_results[i] / TRIALS);
}
printf("\n");
fprintf(ofp, "\n");
fclose(ofp);
}
```

```
/*****************************************************************************
 *  The perform_reservation_based_simulation() procedure is a generalized procedure for
 *  all three of the alternative reservation-based schemes.  The integer argument
 *  "protocol" takes a value 0, 1, or 2, corresponding to gated limited, gated
 *  unlimited, or exhaustive.  In the gated limited version, at most one packet may be
 *  served on a node's turn.  In the gated unlimited version, all packets that arrived
 *  prior to the node's turn  may be served.  In the exhaustive version, all packets
 *  may be served until the queue is exhausted, at which time the node gives up its turn.
 *****************************************************************************/

void perform_reservation_based_simulation (double *arrivals_sorted_by_queue,
                                           int *queue_offsets, double *results,
                                           int number_of_arrivals,
                                           int protocol, int m, int n)
{
  int i, j, turn = 0;
  int *copy_of_queue_offsets, *number_of_busy_slots;
  double time = 0.0, gate_time, packet_delay;
  double *total_delay, *total_square_delay, *worst_case_delay;

  void calculate_algorithm_statistics(double *, double *, double *, int *, double *, int);
  void update_statistics_arrays(int *, double *, double *, double *, double, int *, int);
  double protocol_switch(double, double, int);

  copy_of_queue_offsets = calloc(m, sizeof(int));
  number_of_busy_slots = calloc(m, sizeof(int));
  total_delay = calloc(m, sizeof(double));
  total_square_delay = calloc(m, sizeof(double));
  worst_case_delay = calloc(m, sizeof(double));

  for (i = 0; i < m; ++i){
    copy_of_queue_offsets[i] = queue_offsets[i];
    total_delay[i] = total_square_delay[i] = worst_case_delay[i] = 0.0;
  }
  while (time <= (double) n - 1.0){
    gate_time = time;
    j = 0;
    while ((((turn == m - 1) && (queue_offsets[turn] < number_of_arrivals)) ||
            ((turn < m - 1) && (queue_offsets[turn] < copy_of_queue_offsets[turn + 1]))) &&
           (protocol_switch(time, gate_time, protocol) -
           arrivals_sorted_by_queue[queue_offsets[turn]] > 0.0) &&
           ((protocol != 0) || (j != 1))){
      packet_delay = time - arrivals_sorted_by_queue[queue_offsets[turn]];
      update_statistics_arrays(number_of_busy_slots, total_delay, total_square_delay,
                               worst_case_delay, packet_delay, queue_offsets, turn);
      time += 1.0;
      j = 1;
    }
    if ((protocol != 0) || ((protocol == 0) && (j == 0)))
      time += BETA;
    turn = (turn + 1) % m;
  }
  for (i = 0; i < m; ++i)
    queue_offsets[i] = copy_of_queue_offsets[i];
```

```
  calculate_algorithm_statistics(total_delay, total_square_delay, worst_case_delay,
                                 number_of_busy_slots, results, m);
  free(copy_of_queue_offsets);
  free(total_delay);
  free(worst_case_delay);
  free(total_square_delay);
  free(number_of_busy_slots);
}


/*************************************************************************
 * The heart of the code is the procedure perform_GLOBALTIME_simulation(). Here we
 * run the input in arrivals_sorted_by_queue through our GlobalTime algorithm, and
 * record the delay statistics. As discussed, the main computational task in the
 * GlobalTime algorithm is finding the min among the m known_time[] variables for each
 * time slot. Here, we simulate this process with linear search if m < 50, and we use
 * a binary heap otherwise, to cut down on computational complexity. Also as
 * discussed, we record either the next-in-line timestamp in the known_time[]
 * variable, or the last time at which a node had its turn and there was no
 * next-in-line packet.
 *************************************************************************/

void perform_GLOBALTIME_simulation(double *arrivals_sorted_by_queue,
                                   int *queue_offsets, double *GLOBALTIME_results,
                                   int number_of_arrivals, int m, int n)
{
  int i, turn, heap;
  struct heap_element *known_time_heap;
  double *known_time_linear;
  int *copy_of_queue_offsets, *number_of_busy_slots;
  double time = 0.0, min_known_time, packet_delay;
  double *total_delay, *total_square_delay, *worst_case_delay;

  void calculate_algorithm_statistics(double *, double *, double *, int *, double *, int);
  void update_statistics_arrays(int *, double *, double *, double *, double, int *, int);
  void heapify(struct heap_element *, int, int);

  if (m > 50)
    heap = 1;
  else heap = 0;
  if (heap == 1)
    known_time_heap = calloc(m, sizeof(struct heap_element));
  else
    known_time_linear = calloc(m, sizeof(double));
  copy_of_queue_offsets = calloc(m, sizeof(int));
  number_of_busy_slots = calloc(m, sizeof(int));
  total_delay = calloc(m, sizeof(double));
  total_square_delay = calloc(m, sizeof(double));
  worst_case_delay = calloc(m, sizeof(double));

  for (i = 0; i < m; ++i){
    if (heap == 1){
      known_time_heap[i].val = i;
      known_time_heap[i].key = 0.0;
```

```
    number_of_busy_slots[i] = 0;
  }                                                                          310
  else{
    known_time_linear[i] = 0.0;
    number_of_busy_slots[i] = 0;
  }
  copy_of_queue_offsets[i] = queue_offsets[i];
  total_delay[i] = total_square_delay[i] = worst_case_delay[i] = 0.0;
}
while (time <= (double) n - 1.0){
  if (heap == 1)
    turn = known_time_heap[0].val;                                           320
  else{
    min_known_time = known_time_linear[0];
    turn = 0;
    for (i = 1; i < m; ++i){
        if (known_time_linear[i] < min_known_time){
          min_known_time = known_time_linear[i];
          turn = i;
        }
    }
  }                                                                          330
  packet_delay = time - arrivals_sorted_by_queue[queue_offsets[turn]];
  if ((((turn == m - 1) && (queue_offsets[turn] < number_of_arrivals)) ||
        ((turn < m - 1) && (queue_offsets[turn] < copy_of_queue_offsets[turn + 1]))) &&
        (packet_delay > 0.0)){
    update_statistics_arrays(number_of_busy_slots, total_delay, total_square_delay,
                             worst_case_delay, packet_delay, queue_offsets, turn);
    packet_delay = time - arrivals_sorted_by_queue[queue_offsets[turn]];
    if ((((turn == m - 1)  && (queue_offsets[turn] < number_of_arrivals)) ||
          ((turn < m - 1) && (queue_offsets[turn] < copy_of_queue_offsets[turn + 1]))) &&
          (packet_delay >= 0.0))                                             340
        if (heap == 1)
          known_time_heap[0].key = arrivals_sorted_by_queue[queue_offsets[turn]];
        else known_time_linear[turn] = arrivals_sorted_by_queue[queue_offsets[turn]];
    else{
        if (heap == 1)
          known_time_heap[0].key = time;
        else known_time_linear[turn] = time;
    }
    time += 1.0;
    if (heap == 1)                                                           350
        heapify(known_time_heap, m, 0);
  }
  else{
    time += BETA;
    if (heap == 1){
        known_time_heap[0].key = time;
      heapify(known_time_heap, m, 0);
    }
    else known_time_linear[turn] = time;
  }                                                                          360
}
for (i = 0; i < m; ++i)
```

```
    queue_offsets[i] = copy_of_queue_offsets[i];
  calculate_algorithm_statistics(total_delay, total_square_delay, worst_case_delay,
                                 number_of_busy_slots, GLOBALTIME_results, m);
  if (heap == 1)
    free(known_time_heap);
  else free(known_time_linear);
  free(copy_of_queue_offsets);
  free(total_delay);                                                          370
  free(worst_case_delay);
  free(total_square_delay);
  free(number_of_busy_slots);
}


/*************************************************************************
 *  The procedure perform_single_queue_simulation() takes as input a sorted array of
 *  arriving packets, and serves them in order as if they all had entered a single
 *  queue.  If there is no packet to serve at the head of the queue, there is an idle   380
 *  minislot; otherwise, the packet at the head of the queue is served immediately.
 *************************************************************************/

void perform_single_queue_simulation(double *arrival_array, double *single_queue_results,
                                     int number_of_arrivals, int n)
{
  int number_of_busy_slots = 0;
  double time = 0.0, packet_delay, worst_case_delay = 0.0;
  double total_delay = 0.0, total_square_delay = 0.0;
                                                                              390
  while (time <= (double) n - 1.0){
    packet_delay = time - arrival_array[number_of_busy_slots];
    if ((packet_delay > 0.0) && (number_of_busy_slots < number_of_arrivals)){
      ++number_of_busy_slots;
      time += 1.0;
      total_delay += packet_delay;
      total_square_delay += packet_delay * packet_delay;
      if (packet_delay > worst_case_delay)
          worst_case_delay = packet_delay;
    }                                                                         400
    else time += BETA;
  }
  single_queue_results[0] = total_delay / number_of_busy_slots;
  single_queue_results[1] = total_square_delay / number_of_busy_slots -
    (single_queue_results[0] * single_queue_results[0]);
  single_queue_results[2] = worst_case_delay;
}


                                                                              410
```

```
/*****************************************************************************
 * The next several functions handle input generation.  The procedure
 * create_poisson_arrival_array() takes as input a value for lambda and a value for n
 * (the arrival rate of the entire system, and the total time of the simulation,
 * respectively).  The procedure creates a set of Poisson arrivals with parameter
 * lambda, in the time frame [0, n].  The arrivals are placed into a sorted array,
 * which is then returned.  The Poisson arrivals are generated by using C's
 * pseudorandom number generator to first generate a random decimal between 0 and 1.
 * This [0, 1] space is then mapped to the [0, inf) exponential interarrival
 * distribution, which is used to generate a set of Poisson arrivals.  As the arrivals
 * are first generated, they are linked into a growing list.  The arrival list is
 * later copied into an array.  Technical detail:  The first index in the arrival
 * array is reserved for recording the total number of arrivals, which is an
 * agreed-upon interface between this function and its caller.
 *****************************************************************************/


double *create_poisson_arrival_array(double *poisson_arrival_array, double lambda, int n)
{
  int i, length_of_list = 0;
  double random_decimal, random_interarrival, last_arrival, new_arrival;
  LINK head_of_poisson_arrival_list = NULL, tail_of_poisson_arrival_list;

  random_decimal = (double) random() / (double) 0x7fffffff;
  new_arrival = -((log (1.0 - random_decimal)) / lambda);
  head_of_poisson_arrival_list = malloc(sizeof(ELEMENT));
  head_of_poisson_arrival_list -> data = new_arrival;
  tail_of_poisson_arrival_list = head_of_poisson_arrival_list;
  last_arrival = new_arrival;
  length_of_list++;

  while (last_arrival < n){
    random_decimal = (double) random() / (double) 0x7fffffff;
    random_interarrival = -((log(1.0 - random_decimal))/ lambda);
    new_arrival = last_arrival + random_interarrival;
    tail_of_poisson_arrival_list -> next = malloc(sizeof(ELEMENT));
    tail_of_poisson_arrival_list = tail_of_poisson_arrival_list -> next;
    tail_of_poisson_arrival_list -> data = new_arrival;
    last_arrival = new_arrival;
    length_of_list++;
  }
  tail_of_poisson_arrival_list -> next = NULL;
  poisson_arrival_array = calloc(length_of_list + 1, sizeof(double));
  for (i = 0; head_of_poisson_arrival_list != NULL;
      head_of_poisson_arrival_list = head_of_poisson_arrival_list -> next){
    poisson_arrival_array[i + 1] = head_of_poisson_arrival_list -> data;
    free(head_of_poisson_arrival_list);
    i++;
  }
  poisson_arrival_array[0] = (double)length_of_list;
  return poisson_arrival_array;
}
```

```
/*************************************************************************
 * The procedure create_arrivals_sorted_by_queue() takes a set of Poisson arrivals
 * with rate lambda and divides it into m sets of Poisson arrivals, each of rate
 * lambda / m.  It does this by assigning each arrival to one of the m queues, where
 * the assigned queue is selected uniformly at random among the m.  This division
 * maintains the Poisson distribution at each user.  The m queues are all placed in a
 * single array the same size as the original sorted arrival array.  However, a
 * separate array called queue_offsets keeps track of the array index at which each of
 * the m queues begins within the large arrivals_sorted_by_queue array.
 *************************************************************************/

void create_arrivals_sorted_by_queue (double *arrival_array, int *queue_offsets,
                                      double *arrivals_sorted_by_queue,
                                      int number_of_arrivals, int m)
{
  int *random_queue_assignments;
  int *number_pkts_per_queue, *copy_of_queue_offsets;
  int i, offset = 0;

  number_pkts_per_queue = calloc(m, sizeof(int));
  copy_of_queue_offsets = calloc(m, sizeof(int));
  random_queue_assignments = calloc(number_of_arrivals, sizeof(int));

  for (i = 0; i < m; ++i)
    number_pkts_per_queue[i] = 0;
  for (i = 0; i < number_of_arrivals; ++i){
    random_queue_assignments[i] = random() % m;
    number_pkts_per_queue[random_queue_assignments[i]] += 1;
  }
  copy_of_queue_offsets[0] = queue_offsets[0] = 0;
  for (i = 0; i < m - 1; ++i){
    offset += number_pkts_per_queue[i];
    copy_of_queue_offsets[i + 1] = queue_offsets[i + 1] = offset;
  }
  for (i = 0; i < number_of_arrivals; ++i){
    arrivals_sorted_by_queue[queue_offsets[random_queue_assignments[i]]] = arrival_array[i];
    queue_offsets[random_queue_assignments[i]] += 1;
  }
  for (i = 0; i < m; ++i)
    queue_offsets[i] = copy_of_queue_offsets[i];
  free(number_pkts_per_queue);
  free(copy_of_queue_offsets);
  free(random_queue_assignments);
}

/*************************************************************************
 * The next few procedures handle generation of bursty, rather than pure Poisson
 * traffic.  The procedure create_bursty_input() is the highest-level procedure in
 * this hierarchy.  After calling two supplementary procedures m times each (to be
 * described in more detail later), the variable array_of_arrival_lists will contain
 * an array of m sorted lists of bursty arrivals.  The rest of the
 * procedure simply converts from that format to the format expected by the
 * simulations (i.e. a long array with a separate array of queue offsets).
 *************************************************************************/
```

```
double *create_bursty_input(double *arrivals_sorted_by_queue, int *queue_offsets,
                            double lambda, int n, int m, double mean_burst_size)
{
 double queue_arrival_rate = lambda / (double)m;
 LINK head_of_queue_arrival_list;
 LINK2 head_of_busy_period_list;
 LINK *array_of_arrival_lists;
 int *number_pkts_per_queue;
 int i, offset = 0, length_of_input = 0, index = 1;

 LINK2 create_queue_busy_periods(double, double, int);
 LINK create_queue_arrival_list(LINK2, int);
 void count_number_pkts_per_queue(int *, LINK *, int);

 array_of_arrival_lists = calloc(m, sizeof(LINK));
 for (i = 0; i < m; ++i){
   head_of_busy_period_list = create_queue_busy_periods(queue_arrival_rate, mean_burst_size, n);
   head_of_queue_arrival_list = create_queue_arrival_list(head_of_busy_period_list, n);
   array_of_arrival_lists[i] = head_of_queue_arrival_list;
 }
 number_pkts_per_queue = calloc(m, sizeof(int));
 count_number_pkts_per_queue(number_pkts_per_queue, array_of_arrival_lists, m);
 queue_offsets[0] = 0;
 for (i = 0; i < m - 1; ++i){
   offset += number_pkts_per_queue[i];
   queue_offsets[i + 1] = offset;
 }
 for (i = 0; i < m; ++i)
   length_of_input += number_pkts_per_queue[i];
 arrivals_sorted_by_queue = calloc(length_of_input + 1, sizeof(double));
 arrivals_sorted_by_queue[0] = (double)length_of_input;
 for (i = 0; i < m; ++i){
   head_of_queue_arrival_list = array_of_arrival_lists[i];
   while (head_of_queue_arrival_list != NULL){
     arrivals_sorted_by_queue[index] = head_of_queue_arrival_list -> data;
     index++;
     free(head_of_queue_arrival_list);
     head_of_queue_arrival_list = head_of_queue_arrival_list -> next;
   }
 }
 free(number_pkts_per_queue);
 free(array_of_arrival_lists);
 return arrivals_sorted_by_queue;
}

/*************************************************************************
 * The procedure create_queue_arrival_list() produces a list of bursty arrivals for a
 * queue.  As input, it requires a list of busy periods.  Each element in this input
 * list is a pair that indicates the beginning and the ending time of a single busy
 * period.  For each busy period, the procedure generates Poisson traffic with
 * parameter 1.  The resulting output list will contain bursts of arrivals only
 * during the busy periods, and no arrivals during the intermittent idle periods.
 *************************************************************************/
```

```
LINK create_queue_arrival_list(LINK2 head_of_busy_period_list, int n)
{
  double new_busy_period_start, new_busy_period_end;
  double random_decimal, random_interarrival, last_arrival, new_arrival;
  LINK head_of_arrival_list = NULL, tail_of_arrival_list;
  int i = 0;

  while (head_of_busy_period_list != NULL){
    last_arrival = new_busy_period_start = head_of_busy_period_list -> data1;
    if (new_busy_period_start < (double)n){
      if ((head_of_busy_period_list -> data2) < (double)n)
          new_busy_period_end = head_of_busy_period_list -> data2;
      else new_busy_period_end = (double)n;
      while (last_arrival < new_busy_period_end){
          random_decimal = (double) random() / (double) 0x7fffffff;
          random_interarrival = -(log(1.0 - random_decimal));
          new_arrival = last_arrival + random_interarrival;
          if (new_arrival < new_busy_period_end){
            if (i == 0){
              head_of_arrival_list = malloc(sizeof(ELEMENT));
              head_of_arrival_list -> data = new_arrival;
              tail_of_arrival_list = head_of_arrival_list;
              i = 1;
            }
            else{
              tail_of_arrival_list -> next = malloc(sizeof(ELEMENT));
              tail_of_arrival_list = tail_of_arrival_list -> next;
              tail_of_arrival_list -> data = new_arrival;
            }
          }
          last_arrival = new_arrival;
      }
    }
    free(head_of_busy_period_list);
    head_of_busy_period_list = head_of_busy_period_list -> next;
  }
  if (i == 1)
    tail_of_arrival_list -> next = NULL;
  return head_of_arrival_list;
}

/*****************************************************************************
 * The procedure create_queue_busy_periods() is the final helper procedure involved
 * in generating bursty traffic.  The procedure outputs a list of "busy periods."  A
 * busy period is represented as a pair of real numbers, corresponding to the "begin
 * time" and "end time" of the period.  It will later be the task of
 * create_queue_arrival_list() to actually generate the high-rate Poisson traffic
 * within these periods.  As input, the procedure requires the queue arrival rate
 * (that is, lambda / m), and the mean burst size (or mean busy period length).
 * [Technical point:  The arrival rate is only needed so as to calculate the mean idle
 * period length.]  Busy periods and idle periods are created in an alternating
 * fashion, and each busy or idle period length comes from an exponential distribution
 * with mean equal to mean_burst_size or mean_idle_size respectively.
 *****************************************************************************/
```

```
LINK2 create_queue_busy_periods(double arrival_rate, double mean_burst_size, int n)
{
 double mean_idle_size = (mean_burst_size / arrival_rate) − mean_burst_size;
 double random_decimal, random_busy_period, random_idle_period;
 double new_busy_period_start, new_busy_period_end, last_busy_period_end;
 LINK2 head_of_busy_period_list = NULL, tail_of_busy_period_list;

 random_decimal = (double) random() / (double) 0x7fffffff;
 new_busy_period_start = −((log (1.0 − random_decimal)) * mean_idle_size);
 head_of_busy_period_list = malloc(sizeof(ELEMENT_PAIR));
 head_of_busy_period_list −> data1 = new_busy_period_start;
 random_decimal = (double) random() / (double) 0x7fffffff;
 random_busy_period = −((log (1.0 − random_decimal)) * mean_burst_size);
 new_busy_period_end = new_busy_period_start + random_busy_period;
 head_of_busy_period_list −> data2 = new_busy_period_end;
 tail_of_busy_period_list = head_of_busy_period_list;
 last_busy_period_end = new_busy_period_end;

 while (last_busy_period_end < (double)n){
  random_decimal = (double) random() / (double) 0x7fffffff;
  random_idle_period = −((log(1.0 − random_decimal)) * mean_idle_size);
  new_busy_period_start = last_busy_period_end + random_idle_period;
  tail_of_busy_period_list −> next = malloc(sizeof(ELEMENT_PAIR));
  tail_of_busy_period_list = tail_of_busy_period_list −> next;
  tail_of_busy_period_list −> data1 = new_busy_period_start;
  random_decimal = (double) random() / (double) 0x7fffffff;
  random_busy_period = −((log(1.0 − random_decimal)) * mean_burst_size);
  new_busy_period_end = new_busy_period_start + random_busy_period;
  tail_of_busy_period_list −> data2 = new_busy_period_end;
  last_busy_period_end = new_busy_period_end;
 }
 tail_of_busy_period_list −> next = NULL;
 return head_of_busy_period_list;
}


/*****************************************************************************
 *  The procedure calculate_algorithm_statistics simply takes the accrued delay
 *  statistics from a given simulation, makes additional calculations (that is, computes
 *  means, variances, and worst cases), and then outputs the results into the desired
 *  results_array.
 *****************************************************************************/

void calculate_algorithm_statistics(double *total_delay, double *total_square_delay,
                                    double *worst_case_delay, int *number_of_busy_slots,
                                    double *results_array, int m)
{
 int i;
 double sum_of_user_means = 0.0, sum_of_user_variances = 0.0;
 double sum_of_user_maxes = 0.0, max_of_user_means = 0.0;
 double max_of_user_variances = 0.0, max_of_user_maxes = 0.0;

 for (i = 0; i < m; ++i){
```

```
    if (number_of_busy_slots[i] != 0){
      total_delay[i] = total_delay[i] / (double)number_of_busy_slots[i];
      total_square_delay[i] = total_square_delay[i] / (double)number_of_busy_slots[i] -
          (total_delay[i] * total_delay[i]);
    }
    sum_of_user_means += total_delay[i];
    sum_of_user_maxes += worst_case_delay[i];
    sum_of_user_variances += total_square_delay[i];
    if (worst_case_delay[i] > max_of_user_maxes)
      max_of_user_maxes = worst_case_delay[i];
    if (total_delay[i] > max_of_user_means)
      max_of_user_means = total_delay[i];
    if (total_square_delay[i] > max_of_user_variances)
      max_of_user_variances = total_square_delay[i];
  }
  results_array[0] = sum_of_user_means / (double)m;
  results_array[1] = sum_of_user_variances / (double)m;
  results_array[2] = sum_of_user_maxes / (double)m;
  results_array[3] = max_of_user_means;
  results_array[4] = max_of_user_variances;
  results_array[5] = max_of_user_maxes;
}


/*****************************************************************************
 * A simple procedure, update_statistics_arrays() just performs an update when a
 * packet is transmitted.  It adds the delay of the packet to the total delay of the
 * associated user, and adds the square delay of the packet to the total square delay
 * of the associated user.  It also updates worst_case_delay, number_of_busy_slots,
 * and queue_offsets.
 *****************************************************************************/

void update_statistics_arrays(int *number_of_busy_slots, double *total_delay,
                              double *total_square_delay, double *worst_case_delay,
                              double packet_delay, int *queue_offsets, int turn)
{
  number_of_busy_slots[turn] += 1;
  total_delay[turn] += packet_delay;
  total_square_delay[turn] += packet_delay * packet_delay;
  if (packet_delay > worst_case_delay[turn])
    worst_case_delay[turn] = packet_delay;
  queue_offsets[turn] += 1;
}


/*****************************************************************************
 * The procedure count_number_pkts_per_queue() is a simple helper procedure that takes
 * as input an array of m arrival lists, and returns as output an array of m integers.
 * Each integer corresponds to its associated arrival list's length.
 *****************************************************************************/

void count_number_pkts_per_queue(int *results, LINK *array_of_arrival_lists, int m)
{
  LINK head_of_arrival_list;
```

```
  int count, i;

  for (i = 0; i < m; ++i){
    head_of_arrival_list = array_of_arrival_lists[i];
    count = 0;
    while (head_of_arrival_list != NULL){
      count++;
      head_of_arrival_list = head_of_arrival_list -> next;
    }
    results[i] = count;
  }
}

/*************************************************************************
 * The function protocol_switch() simply switches between gate_time and time.  This is
 * an important distinction between a gated unlimited service discipline and an
 * exhaustive service discipline.
 *************************************************************************/

double protocol_switch(double time, double gate_time, int protocol)
{
  if (protocol == 1)
    return gate_time;
  else return time;
}

/*************************************************************************
 * The function compare_reals() is an auxiliary procedure required for qsort().
 *************************************************************************/

int compare_reals(const void *a, const void *b)
{
  double *a1 = a;
  double *b1 = b;
  if (*a1 < *b1)
    return -1;
  if (*b1 < *a1)
    return 1;
  if (*b1 = *a1)
    return 0;
}

/*************************************************************************
 * The procedure heapify() is the standard version of heapify for a binary min heap.
 *************************************************************************/

void heapify(struct heap_element *A, int heapsize, int i)
{
  int l, r, smallest;
  struct heap_element temp;

  l = 2 * i + 1;
  r = 2 * i + 2;
  if ((l <= heapsize - 1) && (A[l].key < A[i].key))
```

```
    smallest = l;
  else smallest = i;
  if ((r <= heapsize - 1) && (A[r].key < A[smallest].key))
    smallest = r;
  if (smallest != i){
    temp = A[i];
    A[i] = A[smallest];
    A[smallest] = temp;
    heapify(A, heapsize, smallest);
  }
}
```

Bibliography

[1] Awerbuch, B. and Azar, Y. "Local Optimization of Global Objectives: Competitive Distributed Deadlock Resolution and Resource Allocation." *Symposium on Foundations of Computer Science*, pp. 240–249. 1994.

[2] Bachem, A. et al. *Mathematical Programming: The State of the Art.* New York: Springer-Verlag. 1983.

[3] Barry, R. and others. "All-Optical Network Consortium — Ultrafast TDM Networks." *IEEE Journal on Selected Areas in Communications*, 14:999–1013. 1996.

[4] Bertsekas, D. and Gallager, R. *Data Networks.* New Jersey: Prentice Hall, 271–353. 1992.

[5] Cooper, R. "Queues Served In Cyclic Order: Waiting Times." em Bell Systems Journal, 49:399–413. 1970.

[6] Gallager, R. "A Perspective on Multiaccess Channels." *IEEE Transactions on Information Theory*, 31:124–142. 1985.

[7] Hajek, B. and Van Loon, T. "Decentralized Dynamic Control of a Multiaccess Broadcast Channel." *IEEE Transactions on Automatic Control*, 27:559–568. 1982.

[8] Lynch, N. *Distributed Algorithms.* San Francisco: Morgan Kaufmann Publishers, Inc. 1996.

[9] MacKenzie, P. and others. "On Contention Resolution Protocols and Associated Probabilistic Phenomena." *Journal of the ACM*, 45:324–378. 1998.

[10] Metcalfe, R. and Boggs, D. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM*, 395–404. 1976.

[11] Mukherjee, B. "WDM-Based Local Lightwave Networks Part I: Single-Hop Systems." *IEEE Network*, 12–27. 1992.

[12] Rivest, R. "Network Control By Bayesian Broadcast." Technical Report. MIT Laboratory for Computer Science. 1985.

[13] Sanchez, J. and others. "A Survey of MAC Protocols Proposed for Wireless ATM." *IEEE Network*, 52–62. 1997.

[14] Stallings, W. *Data Computer Communications.* New York: Macmillan. 1985.

[15] Zhang, H. "Service Disciplines For Guaranteed Performance Service in Packet-Switching Networks." *Proceedings of the IEEE*, 83:1–23. 1995.