


**A Fair, Robust DQDB Protocol
with Preemptive Priorities**

Scott D. Sterner 

Technical Report #1159

May 1993

A Fair, Robust DQDB Protocol with Preemptive Priorities

Scott D. Sterner
Computer Sciences Department
University of Wisconsin-Madison¹
26 May 1993

ABSTRACT

The IEEE standard for **Distributed Queue Dual Bus**, or **DQDB**, metropolitan area network (MAN) communication supports neither preemptive priorities nor fair allocation of bandwidth among same-priority hosts. Liebeherr, Akyildiz, and Tantawi have proposed an improvement on the standard that solves these problems, but with their protocol the loss of even *one* slot of data can cause starvation. This paper proposes a revision of their protocol that performs almost identically but can survive slot loss. Simulation studies of both single-priority and multiple-priority versions of all three protocols clearly demonstrate the nearly identical performance of the two newer protocols, as well as the advantages of this paper's protocol over the standard.

1. INTRODUCTION

This paper describes and compares three protocols for **Distributed Queue Dual Bus**, or **DQDB**, metropolitan area network (MAN) communication: the IEEE standard [1], an improvement on this standard [2], and a modified form of the second approach that accounts for loss of information due to noise or host restarts (e.g., after a crash).

Simulation studies demonstrate the superiority of the newer protocols over the IEEE standard in terms of (1) appropriate distribution of bandwidth among priorities in a multi-priority system, (2) fair distribution of bandwidth among hosts transmitting at the same priority, and (3) robustness of (1) and (2) when noise causes high data corruption. The standard only outperforms the newer protocols in total bandwidth utilization, but the standard accomplishes this by inappropriate bandwidth allocation among contending

1. Author's current address: Motorola Codex Corporation, 65 Dan Road, Canton, Massachusetts, 02021.

hosts.

2. *STANDARD DQDB*

The IEEE 802.6 working group has standardized the DQDB protocol in [1]. This standard, which this paper calls **Standard DQDB**, specifies physical-layer network communication among various hosts, or **stations**, in a network. Two unidirectional buses connect these stations, each sending data in opposite directions. The stations control access to these buses by means of a distributed protocol, where each station monitors the traffic on the buses to determine when it should transmit data.

At one end of a bus, a hardware device (possibly one of the stations) transmits empty **slots**, which divide the transmission medium into 53-byte units. Each slot contains the usual header information (e.g., destination address), a section for data, a *BUSY* bit, and bits that specify **requests** for slots to transmit on the other bus. When a station has a slot of data to transmit on one bus, it marks a request bit in a slot on the other bus. This tells **upstream** stations (i.e., stations closer to the slot generator) to leave one slot free, so that the requesting station may transmit its data. Each station monitors these requests and free slots, and each station repeats this process of requesting and sending for each slot of data that it has to send.

This paper only examines transmission of data in one direction, because transmission in the other direction works in an identical but independent manner. The remainder of this paper refers to the bus used for transmitting data as the **data bus** (*DB*), the bus used for transmitting requests as the **request bus** (*RB*), slots transmitted on the data bus as **data**

slots, and slots transmitted on the request bus as **request slots**.

DQDB allows for transmission of data at several priority levels. Conceptually, a single station consists of several **modules**, where each module processes all outgoing **messages** (transmissions of unlimited size) at one particular priority level. In other words, each station has a module for transmitting priority 1 data, another module for transmitting priority 2 data, and so on. Conceptually, these modules are connected to the data bus in decreasing order by priority number; the high-priority modules of a station are upstream from the low-priority modules. (Higher priority numbers correspond to more important data.) Each request slot contains one request bit for each priority level; thus, one slot could carry several requests.

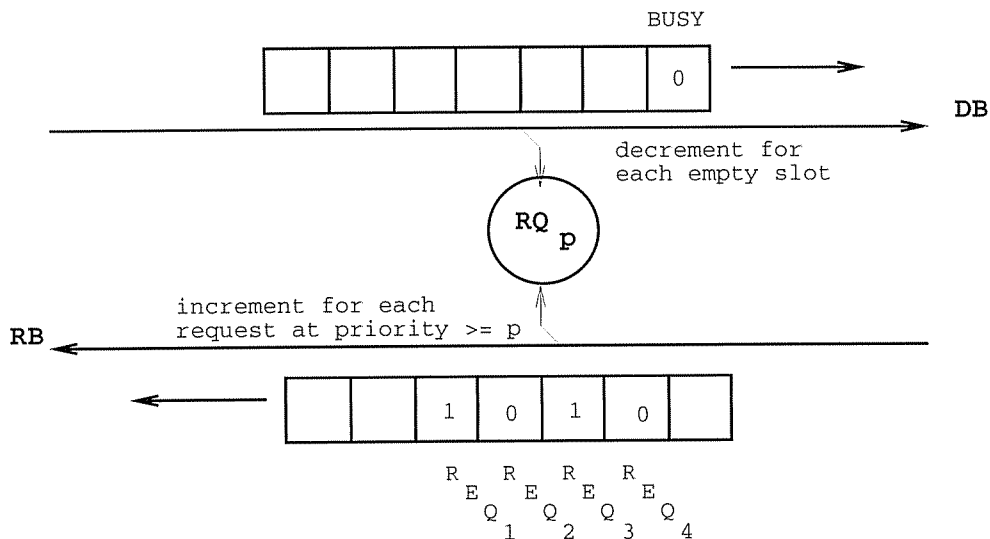


Figure 1. Standard DQDB with No Waiting Segments

Figure 1 shows the protocol used by a priority- p module that does not have data waiting for transmission. Modules send data on the *DB* bus (*data bus*) and requests on the *RB* bus (*request bus*). RQ_p keeps track of the number of unsatisfied requests that have passed the module on the request bus. The module increments this counter for each

request at equal or higher priority and decrements this counter for each free data slot.

(No DQDB counters ever hold negative values.)

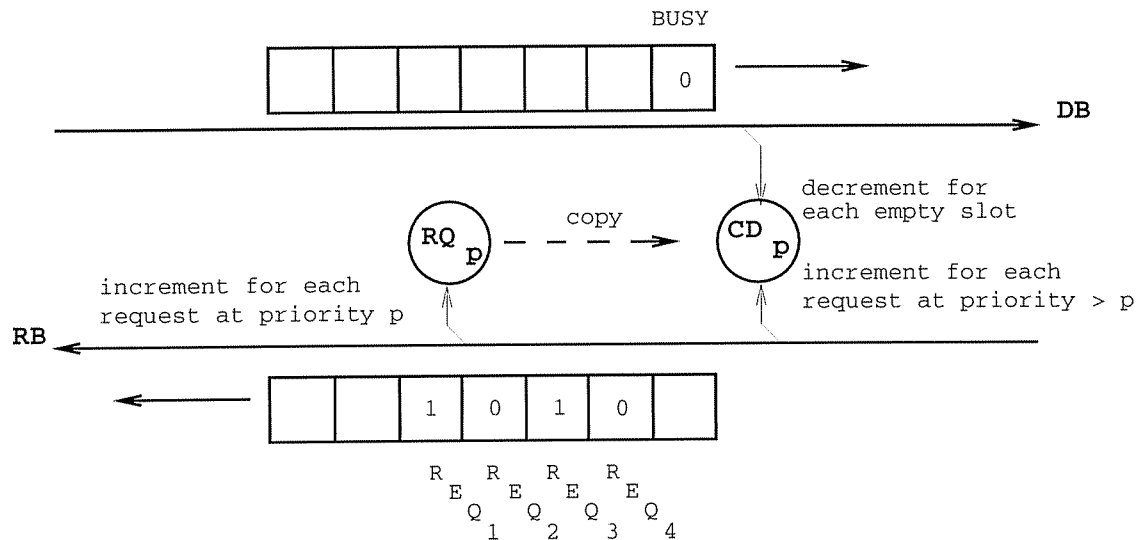


Figure 2. Standard DQDB with Waiting Segments

Figure 2 shows the protocol used by a priority- p module that has data waiting for transmission. When a message arrives at the module for transmission, the module divides it into **segments**, where each segment fits into one data slot. Then, the module sends a request on the request bus and (in parallel) copies RQ_p into a second counter named CD_p and resets RQ_p to zero. The module uses CD_p to count down how many data slots it must let pass before it can transmit its slot. From this point on, the module continues to increment RQ_p for each passing priority- p request. The module also increments CD_p for each passing request at priority greater than p and decrements CD_p for each free passing data slot. The module may transmit its slot when CD_p contains zero and a free data slot reaches it. In this way, RQ_p contains the number of free data slots that must pass before the module can transmit the *next* slot, and CD_p contains the number of free data slots that must pass before the module can transmit the *current* slot,

taking into account requests by other modules to transmit higher priority data. The module executes this same transmission protocol for each slot of data waiting for transmission, one slot at a time.

Standard DQDB also has **bandwidth balancing**, which attempts to correct for the inherent unfairness in DQDB. Because downstream modules must explicitly request a free slot for transmission, upstream modules may utilize more bandwidth than downstream modules. With bandwidth balancing, each module must increment RQ_p after transmitting β slots, thus allowing an extra data slot to pass unused "just in case another module needs it." ([1] uses 8 for β .)

3. *LIEBEHERR-AKYILDIZ-TANTAWI (LAT) DQDB*

Despite the addition of bandwidth balancing, Standard DQDB still suffers from unfairness. This unfairness affects the network in two main ways. First, Standard DQDB lacks **preemptive priorities**. Protocols that implement preemptive priorities allocate network bandwidth such that the modules transmit highest priority data at the exclusion of lower priority data. In other words, no module stops sending data because of bandwidth needs for lower priorities. High priority modules take as much bandwidth as they need, up to the limits of the network, even if that completely prevents lower priority modules from transmitting. In Standard DQDB, low priority modules *can* affect the bandwidth used by high priority modules. Standard DQDB also unfairly allocates bandwidth among modules at the same priority level. As described in the last section, this second bias in allocation favors modules by their nearness to the slot generator on the data bus.

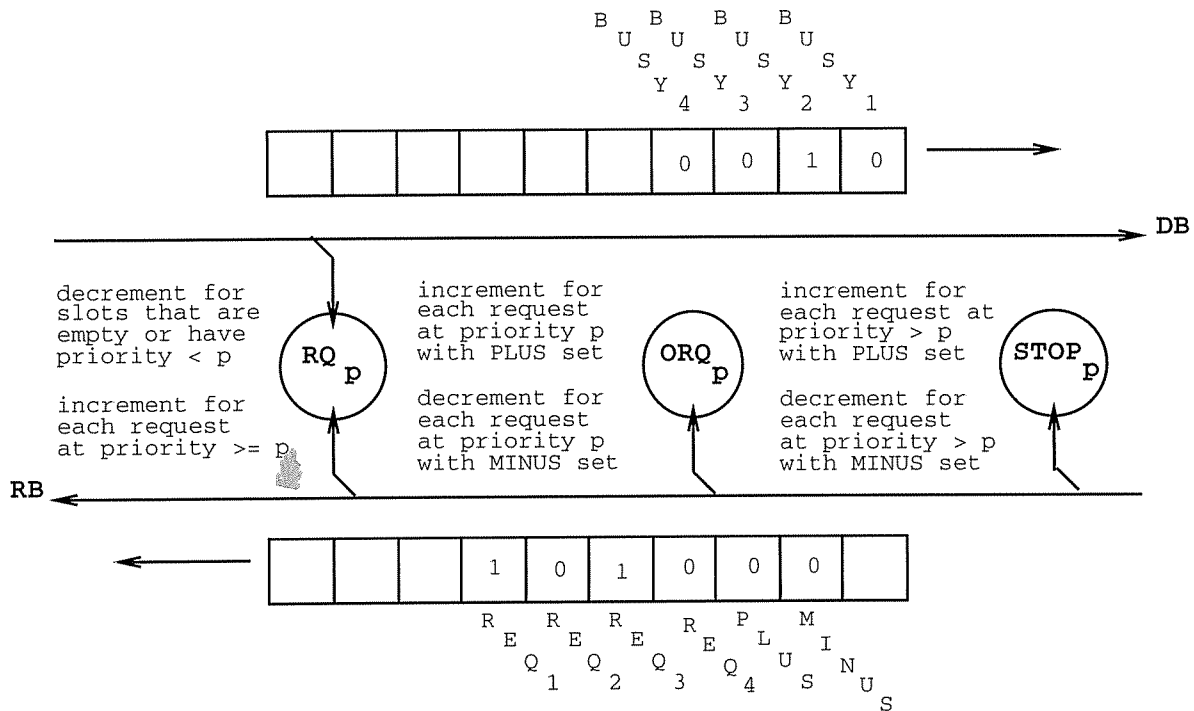


Figure 3. LAT DQDB with No Waiting Segments

In [2], Liebeherr, Akyildiz, and Tantawi define several different qualities of DQDB service in terms of priority preemption and bandwidth utilization. They not only note the theoretical weaknesses of Standard DQDB, but they provide theorems describing the nature of an optimal DQDB protocol. This optimal DQDB protocol perfectly implements preemptive priorities and equal sharing of bandwidth among same-priority modules without wasting any bandwidth. The modules using the optimal protocol decide how much bandwidth they can use in the following manner. A module is **overloaded** if it needs more bandwidth than it currently can get. A module is **underloaded** if it can get all of the bandwidth that it needs. For a particular priority- p module, if any downstream modules with priority greater than p are overloaded, then the module does not transmit at all. Otherwise, the priority- p module evenly divides the unused bandwidth among itself and all of the overloaded downstream priority- p modules. If this does not satisfy the

module's needs, then the module is overloaded. Note how this protocol preempts lower-priority modules and evenly divides bandwidth among same-priority modules as appropriate.

The authors of [2] then propose an implementation of DQDB that would approximate the optimal, which this paper calls **LAT DQDB** (after the creators' initials). (The optimal DQDB protocol cannot be directly implemented because it assumes immediate reception of all transmissions, without concern for timing delays.) Figure 3 shows the protocol used by a priority- p module that does not have data waiting for transmission. Note the addition of two bits to the request slot, *PLUS* and *MINUS*, and the addition of a *BUSY* bit for each priority to the data slot. As in Standard DQDB, the module increments the counter RQ_p for each request at priority equal to or greater than p and decrements RQ_p for each passing free data slot. In addition, LAT DQDB modules decrement RQ_p for each passing data slot with priority less than p . Modules also have two counters for monitoring overloaded downstream modules: ORQ_p and $STOP_p$. ORQ_p contains the number of overloaded downstream modules at priority p . $STOP_p$ contains the number of overloaded downstream modules at priority greater than p . When a module becomes overloaded, it transmits a special request slot in which (1) the only request bit set to 1 is the request bit for the same priority as the module, and (2) the *PLUS* bit is also set. When a module becomes underloaded, it transmits a special request slot in which (1) the only request bit set to 1 is the request bit for the same priority as the module, and (2) the *MINUS* bit is also set. It cannot transmit either special request slot until it receives a request slot with no request bits set. Other than these special requests, overloaded

modules do not transmit request slots. Thus, a module monitors the overloaded downstream modules by incrementing ORQ_p for each passing priority- p request with the *PLUS* bit set, decrementing ORQ_p for each priority- p request with the *MINUS* bit set, incrementing $STOP_p$ for each request at priority greater than p with the *PLUS* bit set, and decrementing $STOP_p$ for each request at priority greater than p with the *MINUS* bit set.

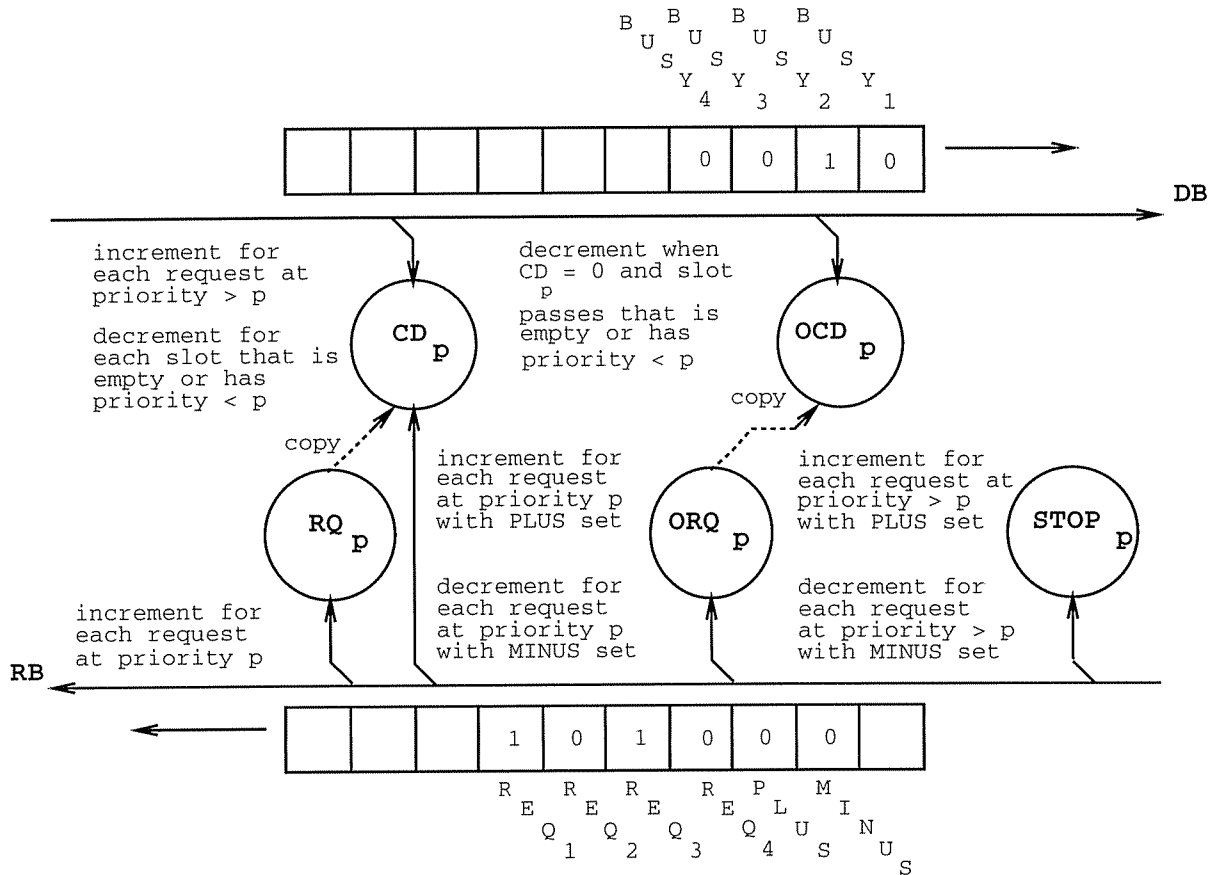


Figure 4. LAT DQDB with Waiting Segments

Figure 4 shows the protocol used by a module that has data waiting for transmission. In addition to the operations performed by Standard DQDB, a LAT DQDB module copies ORQ_p into OCD_p . (Note that, unlike RQ_p , ORQ_p retains its value at this point.) Similar to CD_p , OCD_p counts down free data slots before transmission can commence, one slot per overloaded downstream priority- p module. The module modifies RQ_p and CD_p in

the same manner as in Standard DQDB except that, similar to when the module does not have any slots to transmit, the module decrements CD_p for each busy data slot that passes at priority less than p . It modifies ORQ_p and $STOP_p$ in the same manner as in Figure 3. Once CD_p reaches zero, the module decrements OCD_p for each passing data slot that is either free or contains data with priority less than p . When CD_p , OCD_p , and $STOP_p$ all contain zero, the module may transmit its slot. As in Standard DQDB, this entire process begins when the module prepares to transmit the next slot.

Each module has additional counters for calculating how much bandwidth it can use. A module stores the number of slots queued for it to transmit in $NumSeg_p$. It also increments $SlotCtr_p$ for each passing data slot and increments $Busy_p$ for each passing busy data slot with priority greater than or equal to p . When $SlotCtr_p$ reaches the value $Basis$ (a predefined constant), the module uses these counters to determine $Quota_p$, the maximum number of slots the module can transmit out of the next $Basis$ passing data slots. It then resets $SlotCtr_p$ and $Busy_p$ to zero and continues with normal operation.

If $STOP_p$ does not contain zero, then the module sets $Quota_p$ to zero. This means that the module cannot transmit any data in the next $Basis$ slots, regardless of the values in its other counters. If $STOP_p$ contains zero, the module uses the following formula to calculate $Quota_p$:

$$Quota_p = \frac{Basis - Busy_p - RQ_p - CD_p}{ORQ_p + 1}.$$

This formula reasonably approximates the optimal DQDB protocol, as defined earlier.

At this point, if $NumSeg_p$ is greater than $Quota_p$, the module is overloaded and responds

as described above. Otherwise, the module is underloaded and responds as described above if it just was overloaded. In summary, a module can only transmit data when CD_p , OCD_p , and $STOP_p$ all contain zero and the module has transmitted less than $Quota_p$ data slots since it last calculated $Quota_p$. Note that a module will not transmit *request* slots either when $Quota_p$ or $STOP_p$ contains zero.

Unfortunately, LAT DQDB has two pathological flaws when implemented. First, loss of "I'm underloaded" request slots can cause lower-priority modules to starve and same-priority modules to waste bandwidth. This occurs because the $STOP_p$ counter will never reach zero in lower-priority modules and the ORQ_p counter will always be one greater than the correct value in same-priority modules. Similarly, loss of "I'm overloaded" request slots inhibits the operation of preemptive priorities, increasing delays during data transmission. Solutions to these problems involving resending "I'm overloaded" and "I'm underloaded" request slots clearly cannot succeed. In addition to this problem, if a station crashes and restarts, its modules cannot determine the appropriate values for their counters.

Standard DQDB is inherently unfair and wasteful. LAT DQDB should approach optimality in fairness and bandwidth utilization, but it does not handle slot loss or station restarts well. This paper proposes a modified version of LAT DQDB that also approaches optimality in fairness and bandwidth utilization, but that survives both restarts and even heavy slot losses. Because of its characteristics, this paper calls the new protocol **Sterner DQDB**.

4. STERNER DQDB

This section describes Sterner DQDB in terms of modifications to LAT DQDB.

Figure 5 shows the protocol used by a module that does not have data waiting for transmission. In place of LAT DQDB's *PLUS* and *MINUS* bits, Sterner DQDB request slots have one overload bit for each priority level. The module increments ORQ_p for each request slot that passes with the priority- p overload bit set and sets $STOP_p$ to two when a request slot passes with an overload bit set for priority greater than p .

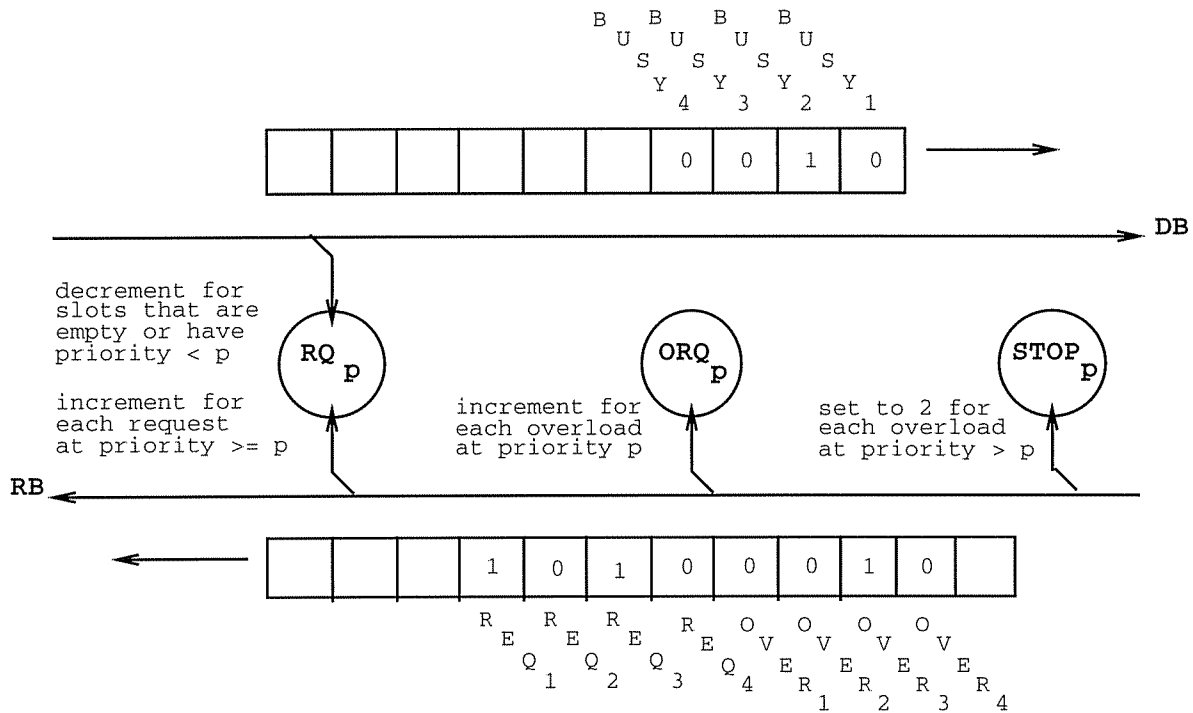


Figure 5. Sterner DQDB with No Waiting Segments

Figure 6 shows the protocol used by a module that has data waiting for transmission.

The module modifies ORQ_p and $STOP_p$ in the same manner as when it has no slots to transmit, and it uses the counters to determine if it can transmit in the same manner as LAT DQDB (i.e., the module cannot transmit until CD_p , OCD_p , and $STOP_p$ all contain

zero and the module has transmitted fewer than $Quota_p$ data slots since last calculating $Quota_p$). The remaining modifications to LAT DQDB involve the initial value for OCD_p and the operations performed when $SlotCtr_p$ equals $Basis$.

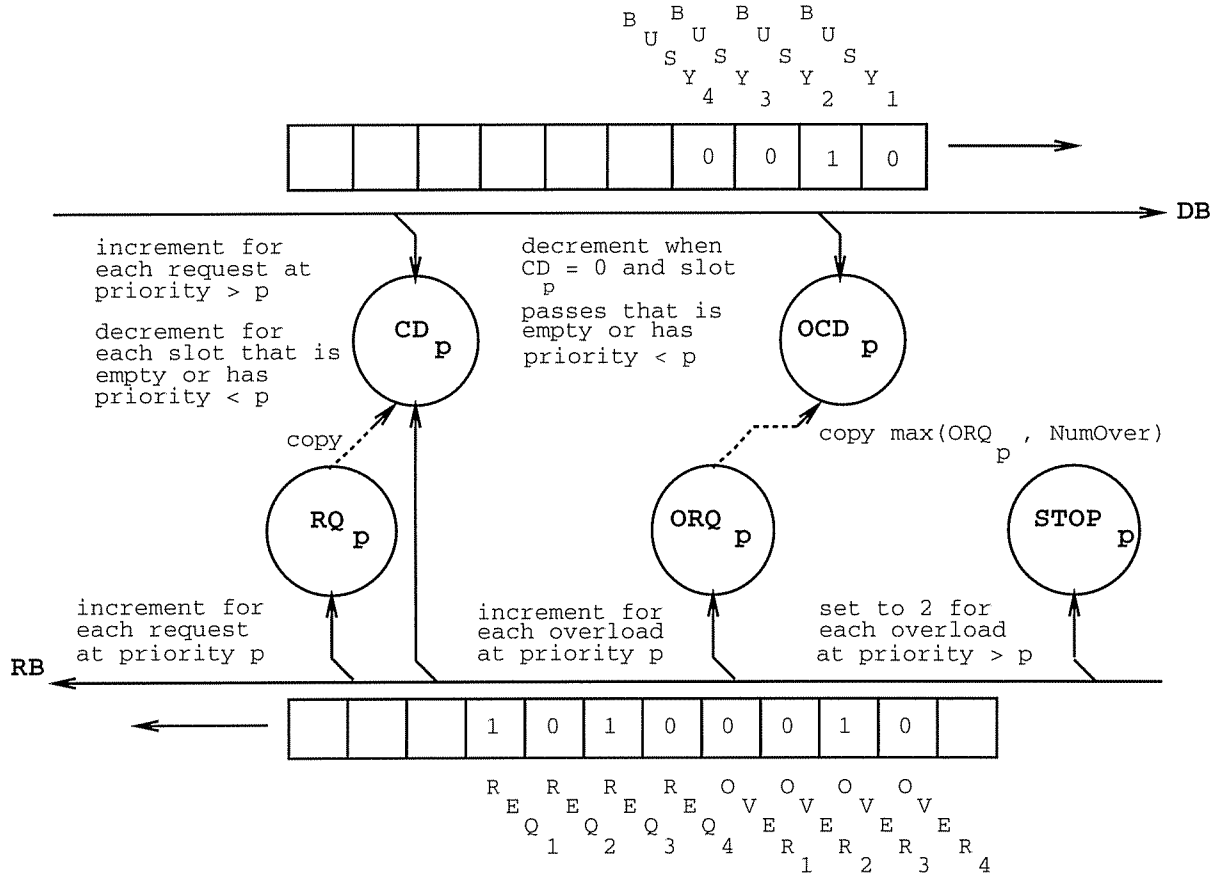


Figure 6. Stermer DQDB with Waiting Segments

Each time another $Basis$ data slots pass a module, it performs five operations. First, if $STOP_p$ is greater than zero, the module decrements it. Second, the module calculates $Quota_p$ in the same manner as in LAT DQDB. Third, if the module is overloaded and $STOP_p$ contains zero, the module marks the overload bit for its priority within a passing request slot. Fourth, the module copies the value in ORQ_p to the counter $NumOver_p$, which contains the number of overloaded downstream priority- p modules during the module's most recent calculation of $Quota_p$. Finally, the module sets ORQ_p , $SlotCtr_p$,

and $Busy_p$ to zero. When the module prepares to transmit a data slot, instead of copying ORQ_p into OCD_p as in LAT DQDB, it copies the larger of ORQ_p and $NumOver_p$ into OCD_p . This prevents a module from using an inappropriately small initial value for OCD_p if the module happens to prepare to transmit a slot shortly after calculating $Quota_p$.

Essentially, Sterner DQDB replaces LAT DQDB's explicit overload/underload requests with a timeout mechanism. In other words, it replaces the message, "I'm overloaded until I say otherwise," with the message, "I'm going to be overloaded for at least the next $Basis$ slots." Note that by using $NumOver_p$, a module must account for other overloaded downstream priority- p modules for at least the next $Basis$ slots after seeing their overload requests. In addition, by setting $STOP_p$ to two after seeing an overload request at priority greater than p , the module cannot transmit any slots for at least the next $Basis$ slots, even if the module calculates $Quota_p$ immediately after receiving the overload request.

With this approach, modules automatically retransmit lost overload/underload requests, but these requests only have value for a bounded (and brief) period of time. In addition, any station that crashes and restarts may determine its appropriate counter values after only seeing $Basis$ data slots pass. Even with these changes, Sterner DQDB nearly duplicates LAT DQDB in the calculation of $Quota_p$ and in the use of its counters for determining when a module can transmit a data slot.

Of course, starvation can still occur in Sterner DQDB if a module never sees a free data slot, and external noise damages all of the module's request slots immediately after it

transmits them. However, such a situation would cause starvation in *any* DQDB protocol because of DQDB's use of requests to alert upstream modules of a module's bandwidth needs.

5. SIMULATIONS

This section describes the parameters of the simulations discussed in the following two sections. An appendix to this paper describes the criteria for selecting these parameters and the mechanics of the simulations in greater detail.

5.1 Constants

- number of stations: 10, where station 0 is the furthest upstream on the data bus and station 9 is the furthest downstream (i.e., station 0 is adjacent to the data-bus slot generator, and station 9 is adjacent to the request-bus slot generator)
- length of simulation: 55 seconds simulated real time (about 20 million data-slot transmissions)
- β in Standard DQDB (for bandwidth balancing): 8 (as specified in [1])
- size of messages by priority level: 1 slot for priority 3, 1-2 slots for priority 2, 1-3 slots for priority 1, 100-1105 slots for priority 0 (Size was randomly selected within the specified ranges using a uniform probability distribution.)
- maximum number of messages queued for transmission at a module: 10
- offered load: 150 percent (100 percent was also simulated with similar results, so this paper only examines the 150 percent simulations)

- breakdown of 150 percent offered load by priority level: 7.5 percent for priority 3, 22.5 percent for priority 2, 45 percent for priority 1, and 75 percent for priority 0
- description of arrivals: Let

$s_i = \text{average slots per message at priority } i,$

$l_i = \text{offered load at priority } i,$

$p_i = \text{probability of a message arriving at priority } i,$ and

$P_i = \text{per-station probability of a message arriving at priority } i.$

The simulations selected p_i such that

$$p_i = \frac{l_i}{s_i} \text{ and}$$

$$P_i = \frac{p_i}{\text{number of stations}}.$$

5.2 Variables

- number of priorities: 1 or 4
- number of slots per bus: 10 (in the single-priority case only), 20, or 40
- slot loss: 0 percent, 0.75 percent, or 7.5 percent.
- number of slots lost at one time: 5-10 slots (randomly selected within this range using a uniform probability distribution)
- $\text{Basis} = 2 * (\text{number of slots per bus}) = 1 \text{ round trip time}$

5.3 Measurements

The simulations measure three quantities: bandwidth used per module, average delay experienced per message per module, and total network bandwidth utilization. These

measurements are used to calculate bandwidth utilization and average delay experienced per priority level. This paper expresses the bandwidth measurements as fractions of the total possible bandwidth and the delay measurements as multiples of message size, to compensate for variations in delay caused by differing message sizes (e.g., a message delay of 2 means that during transmission of the message, the module could not use $2 * (\text{number of slots in message})$ of the data slots that passed it).

6. SINGLE-PRIORITY PERFORMANCE

This section discusses simulations performed where modules transmit all data at only one priority level.

| simulation number | number of stations | percent slot loss |
|-------------------|--------------------|-------------------|
| 0 | 10 | 0 |
| 1 | 20 | 0 |
| 2 | 40 | 0 |
| 3 | 20 | 0.75 |
| 4 | 20 | 7.5 |

TABLE 1. Single-priority Simulation Parameters

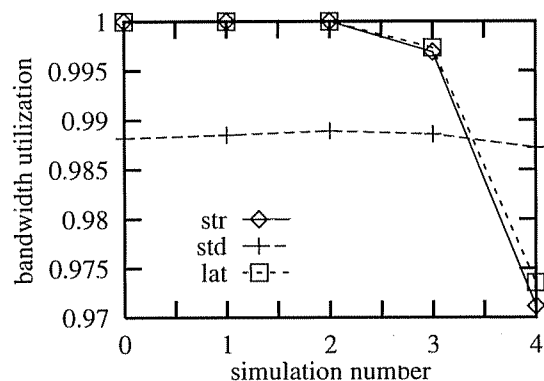


Figure 7. Bandwidth Utilization (one priority)

6.1 Total Bandwidth Utilization

Figure 7 shows total bandwidth utilization for the five sets of parameters given in Table 1. It shows the superiority of Sterner DQDB and LAT DQDB under a variety of situations, utilizing 100 percent of the bandwidth in three of the simulations. The dip in the Sterner DQDB and LAT DQDB graphs represent simulation of very high slot loss (7.5 percent). This effect occurs because these protocols sacrifice bandwidth in order to maintain fairness among the modules. Whenever one of these networks loses many slots, its modules remain overloaded longer because they cannot use slots marked as "lost."

6.2 Fairness Among Modules

Figures 8, 9, and 10 show the fairness of bandwidth allocation among the modules for a network with 10, 20, and 40 slots per bus respectively (with no slot loss). In all three cases, Sterner DQDB and LAT DQDB perfectly divide the bandwidth, whereas Standard DQDB favors the first two stations on the data bus. Note further that Sterner DQDB and LAT DQDB perform identically for all topologies, while Standard DQDB becomes more unfair with increasing inter-station distance, decreasing bandwidth and increasing delays for all but the first two stations.

The unfairness of Standard DQDB occurs because station 0 only allows slots to pass unused when it has seen an explicit request for a free slot. This means that station 0 can often send a burst of data slots, followed by allowing one slot to pass unused, followed by another burst of data slots.

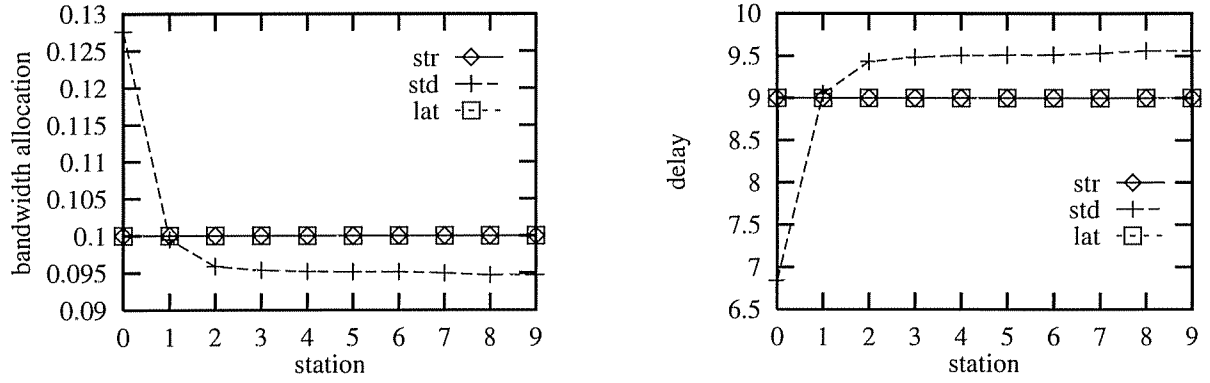


Figure 8. One Priority, 10 Slots Per Bus

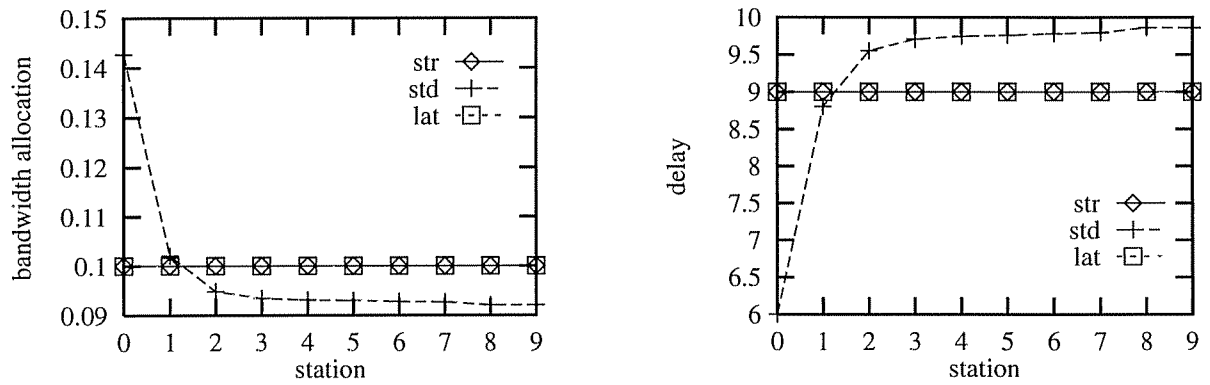


Figure 9. One Priority, 20 Slots Per Bus

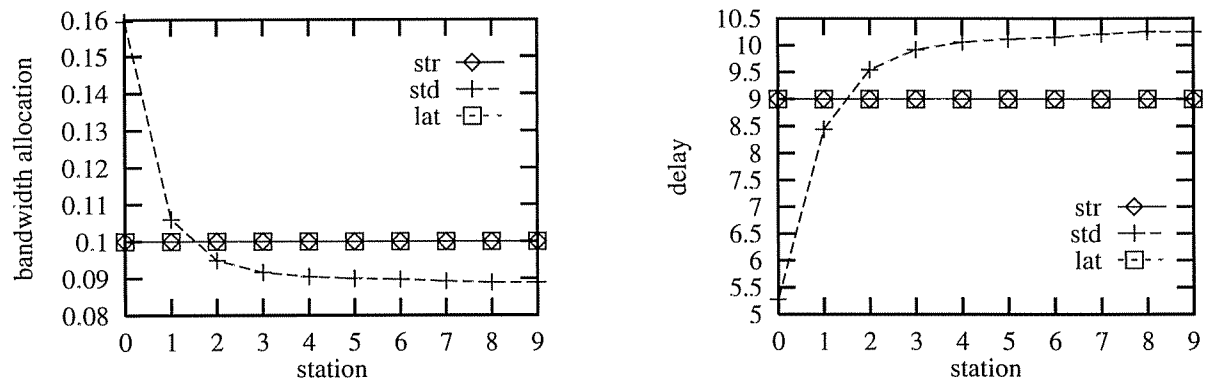


Figure 10. One Priority, 40 Slots Per Bus

7. MULTI-PRIORITY PERFORMANCE

This section discusses simulations performed where stations have modules for each of four priority levels.

7.1 Total Bandwidth Utilization

Figure 11 shows total bandwidth utilization for the four sets of parameters given in Table 2. It shows the slight superiority of Standard DQDB in overall bandwidth utilization for multi-priority systems. This results from how both Sterner DQDB and LAT DQDB waste some bandwidth in order to better distribute the bandwidth among the modules, as discussed below.

| simulation number | number of stations | percent slot loss |
|-------------------|--------------------|-------------------|
| 0 | 20 | 0 |
| 1 | 40 | 0 |
| 2 | 20 | 0.75 |
| 3 | 20 | 7.5 |

TABLE 2. Multi-priority Simulation Parameters

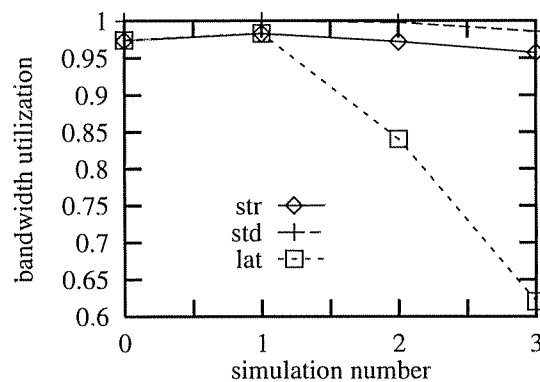


Figure 11. Bandwidth Utilization (four priorities)

With no slot loss, LAT DQDB and Sterner DQDB waste bandwidth when low-priority modules stop transmitting to free up bandwidth for higher-priority modules. This results

from the use of *Basis* by modules instead of modules instantaneously receiving request slots, as in the optimal DQDB. (Future simulation studies should explore the relationship between *Basis* and bandwidth utilization.) As slot loss increases, the performance of LAT DQDB degrades sharply.

7.2 Preemptive Priorities and Topology

Figure 12 compares the preemptiveness of the protocols for 20-slot and 40-slot buses (with no slot loss). These graphs show the nearly identical bandwidth and delay characteristics of Sterner DQDB and LAT DQDB. The graphs further show the superiority of these two protocols over Standard DQDB in both bandwidth allocation and transmission delay. (Recall that the optimal DQDB protocol would allocate 7.5 percent of the bandwidth to priority 3 traffic, 22.5 percent of the bandwidth to priority 2 traffic, 45 percent of the bandwidth to priority 1 traffic, and the remaining bandwidth to priority 0 traffic, with priority 0 traffic carrying most of the burden for delays.) Note also that topology had little effect on the performance of the protocols.

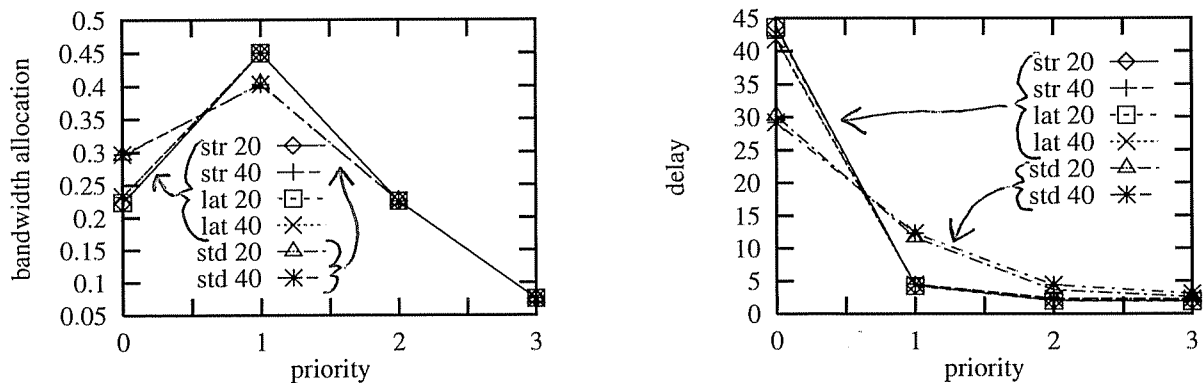


Figure 12. Four Priorities, Topology Variation (no slot loss)

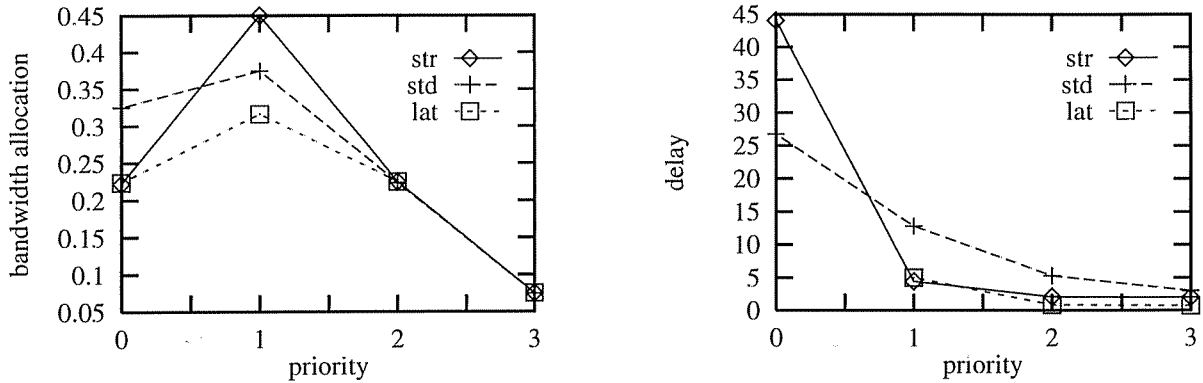


Figure 13. Four Priorities, 0.75 Percent Slot Loss (20 slots per bus)

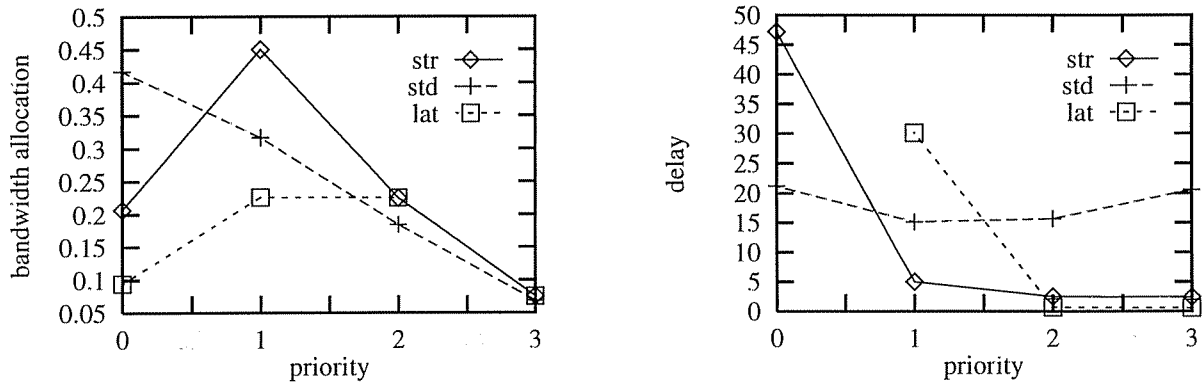


Figure 14. Four Priorities, 7.5 Percent Slot Loss (20 slots per bus)

7.3 Preemptive Priorities and Slot Loss

Figures 13 and 14 compare the preemptiveness of Standard DQDB with that of Sterner DQDB for each of the loss percentages. Note that Sterner DQDB approximates the optimal DQDB, while Standard DQDB becomes less and less preemptive as slot loss increases. When Standard DQDB suffers from 7.5 percent slot loss, it actually transmits *more* priority 0 data than for lesser amounts of loss. This results from the loss of request slots in Standard DQDB, which enables modules to transmit without allowing as many free slots to pass on the data bus. One might expect that Sterner DQDB would also suffer

greatly from increased slot loss (because of the loss of overload information), but Figures 13 and 14 contradict this. Why doesn't Sterner DQDB suffer like Standard DQDB?

High-priority Sterner DQDB modules become overloaded more easily with slot loss than without, so they transmit many more overload requests. Even with high levels of loss, modules are unlikely to lose two consecutive overload requests. Because the counter $STOP_p$ must count down from *two* to zero before transmission can continue, high-priority modules can still transmit, while priority 0 modules cannot transmit as often. Thus, the burden of additional slot loss rests almost solely on the lowest priority, as it should.

Figures 13 and 14 also show that Sterner DQDB suffers from little additional delay during high slot loss. Standard DQDB suffers from increasing delays, particularly once the slot loss reaches 7.5 percent.

These figures further verify the expected effect of slot loss on LAT DQDB. As slot loss increases, LAT DQDB's performance degrades in every way. In fact, the priority 0 points for LAT DQDB in Figures 13 and 14 are too large to graph without obscuring the differences between the other lines. (For priority 0 data with 0.75 percent slot loss, the LAT DQDB delay is 102; with 7.5 percent slot loss, the LAT DQDB delay is 4147.)

7.4 Fairness Among Modules

Figures 15, 16, 17, and 18 compare the fairness among same-priority modules using a 20-slot bus. (In these graphs the Sterner DQDB and LAT DQDB results are virtually identical, so the graphs each contain only one line for both protocols.) Note once again the unfairness of Standard DQDB in bandwidth allocation and delays for lower-priority

traffic. Both Sterner DQDB and Standard DQDB suffer from unfairness at priority 0, where upstream modules gain greater bandwidth (resulting from the same cause as the unfairness in single-priority Standard DQDB, where downstream modules have to send more requests to gain bandwidth than upstream modules), but Sterner DQDB's method of splitting bandwidth among overloaded modules greatly reduces this unfairness. Note further that the priority-0 delays at downstream modules for Standard DQDB are an order of magnitude greater than the corresponding delays for Sterner DQDB.

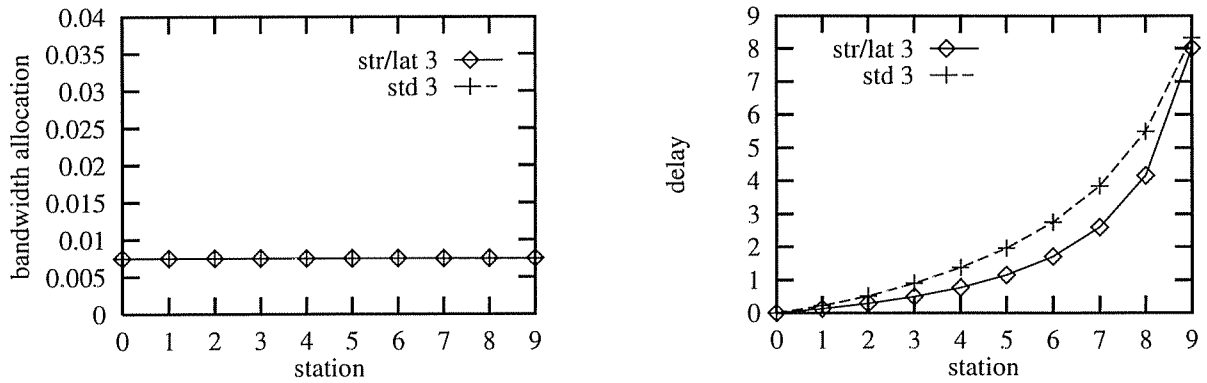


Figure 15. Priority Three, 20 Slots Per Bus (no slot loss)

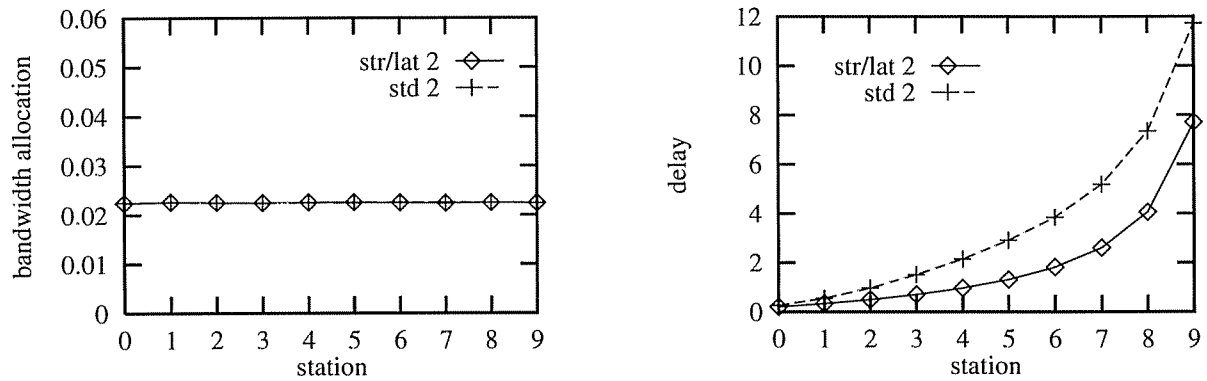


Figure 16. Priority Two, 20 Slots Per Bus (no slot loss)

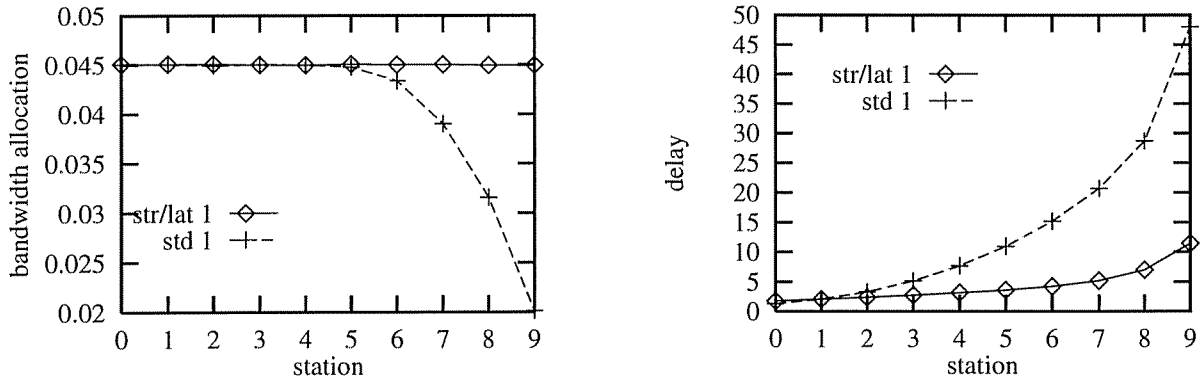


Figure 17. Priority One, 20 Slots Per Bus (no slot loss)

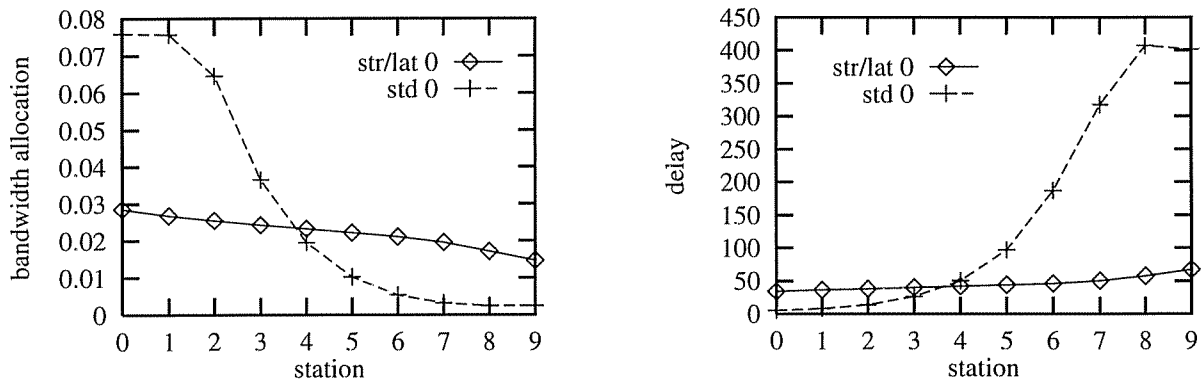


Figure 18. Priority Zero, 20 Slots Per Bus (no slot loss)

The 40-slot simulations have similar results, except for a slight increase in delay (as expected). For this reason, this paper does not present the 40-slot results in detail.

8. CONCLUSIONS

This paper demonstrates the advantages of Sterner DQDB over Standard DQDB for heavy-traffic environments, particularly for single-priority systems. Only in total bandwidth utilization does Standard DQDB surpass Sterner DQDB (by a small amount), but Standard DQDB achieves this by unfair and non-preemptive bandwidth allocation.

However, Sterner DQDB is more complex than Standard DQDB, and its behavior is more difficult to understand. Considering that these protocols would most likely be implemented in hardware, further study of both methods should be made to determine if the advantages of Sterner DQDB outweigh the anticipated additional costs.

This paper also demonstrates the robustness of Sterner DQDB over LAT DQDB for heavy-traffic, multi-priority environments. As slot loss increases, Sterner DQDB is virtually unaffected, while LAT DQDB's performance quickly and unacceptably degrades. These two approaches are nearly identical in complexity, making Sterner DQDB the best choice for implementation.

Until metropolitan area networks gain wide acceptance, the traffic loads and patterns that DQDB protocols must support cannot be reliably determined. The simulations described in this paper used one model of traffic, but future DQDB simulations should explore other traffic models.

REFERENCES

- [1] IEEE. **Std 802.6-1990, IEEE Standards for Local and Metropolitan Area Networks: Distributed Queue Dual Bus (DQDB) of a Metropolitan Area Network (MAN)**, July 1991.
- [2] J. Liebeherr, I. F. Akyildiz, and A. N. Tantawi. *An Effective Scheme for Pre-Emptive Priorities in Dual Bus Metropolitan Area Networks*, from **Proc. SIGCOMM'92**, August 1992, pp. 161-168.

APPENDIX A: BACKGROUND ON SIMULATIONS

This section provides additional background information on the simulations discussed in the body of the paper. The simulations use arrays of records to simulate the two buses.

The main loop of the simulations consists of the following:

for each iteration:

- randomly create messages for transmission
- move request slots ahead one position on the bus, creating a new slot in the most upstream position (relative to the request bus)
- process the request slot adjacent to each module
- move data slots ahead one position on the bus, creating a new slot in the most upstream position (relative to the data bus)
- process the data slot adjacent to each module
- randomly mark slots as "lost" (if simulating slot loss)

The number of iterations is fixed at 20 million. For a 155 Mb/sec bus, this simulates about 55 seconds of real time. Note that all of the modules within a single station are adjacent to the same slot on a bus. As described above, the modules each examine the adjacent data slot in decreasing order by priority and examine the adjacent request slot in increasing order by priority.

The random number generator uses uniform probability distribution.

Four-priority simulations were performed because all three protocols can support up to

four priorities in a slot with a two-byte access-control header field. All of the simulations use 53-byte slots [1]. If a network only uses three priorities, Standard DQDB can manage with only a one-byte access-control header field. The simulations do not explore this difference among the protocols because initial tests of the simulator indicated that the header-size differences would only noticeably affect the relative performance of the protocols if a common network application frequently sent messages containing *exactly* 52 bytes of data.

The message sizes and frequencies arise out of Internet traffic patterns. The simulations assume that modules use priority 0 for file transfer, priority 1 for remote terminal operations, priority 2 for high-priority user communication and low-priority network administration, and priority 3 for high-priority network administration. Note, however, that because the simulations randomly generate messages such that the overall bandwidth utilization approximates fixed percentages, the *size* of those messages matters little in the long run: if message size increases the frequency of message generation must decrease to maintain the fixed percentages.

Slot loss, specified in the body of this paper in terms of percentage, is simulated by means of a "slot loss constant" such that

$$\textit{slot loss percentage} = (\textit{slot loss constant}) * (\textit{average loss amount}),$$

where *average loss amount* is the average number of slots damaged during a slot loss situation. The slot loss constant indicates how often a simulator should lose slots; the location of this decision within the simulator was noted earlier in this section. When the simulator decides to lose slots, it randomly chooses the first slot to lose, randomly

chooses how many slots to lose (from 5 to 10), and marks the slots from the first through the $(\text{first} + \text{number lost} - 1)$ slot as "lost." All of these randomization decisions use a uniform probability distribution. Note that this assumes that modules neither read nor overwrite lost slots.