MONITORING RULES

Not Congested Mode

C: Go into congested mode

S, E: Remain in not congested mode

Congested Mode, Successful Send

S: Remove top bucket from queue
    If this bucket was marked as DQPAIR (if so, it will be from the high priority queue), then
    Find the first bucket in the low priority queue marked as DQPAIR and unmark it

Congested Mode, Ordering Slot

E: No change

H, L: Add a new bucket on the end of the appropriate queue
    If we were the sender for this event, then
        Put our packet in this bucket

C: Add a new bucket on the end of each queue
    Mark each bucket as DQPAIR
    If we participated in this event, then
        Put our packet in the new bucket with the same priority as the packet

Congested Mode, Collision Period Slot

C: Remove the bucket from the front of the highest priority, non-empty queue
    Add two new buckets to the front of this queue
    Mark the top bucket as TICP
    If we participated in the collision, then
        Randomly pick one of the top two buckets and put our packet into it

E: If the top bucket of the highest priority non-empty queue is marked as TICP, then
    If we have a packet in the bucket immediately following this TICP bucket
        Remove our packet from this bucket
        Randomly pick either our old bucket or the TICP bucket and put our packet into it
    Else if the top bucket is marked as DQPAIR (if so, it is in the high priority queue), then
        Find the first bucket in the low priority queue marked as DQPAIR
        Unmark it
        Split it into two buckets
        If we have a packet in this low priority bucket, then
            Randomly pick one of the two buckets to put our packet into

Figure 5.2: The FDDQ algorithm

must see the bus be empty for two times the maximum propagation time plus the minimum length of time needed to detect a packet plus the minimum space between packets. This makes an empty slot always cost 64 bytes. The third diagram shows a full slot, which incurs the cost of sending a new minimum length packet (35 bytes) plus its associated propagation delay and minimum space, for a total cost of between 47 and 76 bytes.

The last diagram shows a collision slot. In this diagram, B actually stands for a set of multiple senders. One of the members of B detects the collision within one propagation time plus the minimum space between packets and thensends out a JAM signal. After the collision is finished, there is another propagation delay to the next sender which takes at most the maximum propagation delay of the bus plus another minimum space between the end of the JAM signal and the beginning of the new packet, for a total of between 18 and 47 bytes. These calculations assume that the slot is being used between two packet sends. If multiple slots occur in a row, then the extra minimum space and propagation delay costs (shown in the first diagram) must be amortized over these slots.

5.2 Basic algorithm

TRANSMISSION RULES

Not Congested Mode

If we have a packet to send, then
    Wait until the bus is free and attempt to send it
    If a collision occurs, then
        Stop transmission, but do not discard packet
        Send a JAM signal, as per 802.3 rules

Congested Mode, Ordering Slot

If we have a packet to send that is not in a bucket, then
    Send a minimum packet of 35 bytes, with the data field containing our requested priority
    If a collision occurs, then
        Stop transmission, discard minimum length packet
        Send a JAM signal, as per 802.3 rules

Congested Mode, Collision Period Slot

If we have a packet to send that is in the top bucket in the highest priority queue, then
    Attempt to send it.
    If a collision occurs, then
        Stop transmission, but do not discard packet
        Send a JAM signal, as per 802.3 rules

When in the congested mode, controllers place a packet of theirs into a bucket on one of the queues by participating in the first ordering slot that starts after the arrival of that packet. If only one controller sends in an ordering slot, then all controllers add a new bucket to the end of the queue of the packet's priority, and that sender places its packet in it. If more than one controller sends in an ordering slot, a collision occurs and all of the controllers add one new bucket to the end of each queue. The senders that were involved in that collision then put themselves into the new bucket in the queue with their priority. A controller may have more than one packet in the queues at a time, although it can only have one per bucket. The number of packets that a controller may have in the queues at a time should be limited by a higher level protocol.

When a controller has a packet in the top bucket, it waits until the network is empty and an ordering slot is no in progress, and them tries to send this packet. Since a bucket may have more than one packet in it, a collision resolution scheme must be used to resolve the order of the senders within a bucket. When a collision occurs, the FDDQ algorithm divides the top bucket into two buckets and each sender that was involved in the collision randomly puts itself into one of the two buckets. In this way, the number of senders in the top bucket is reduced by an average of one half with each collision, although some empty buckets may be created with this policy.

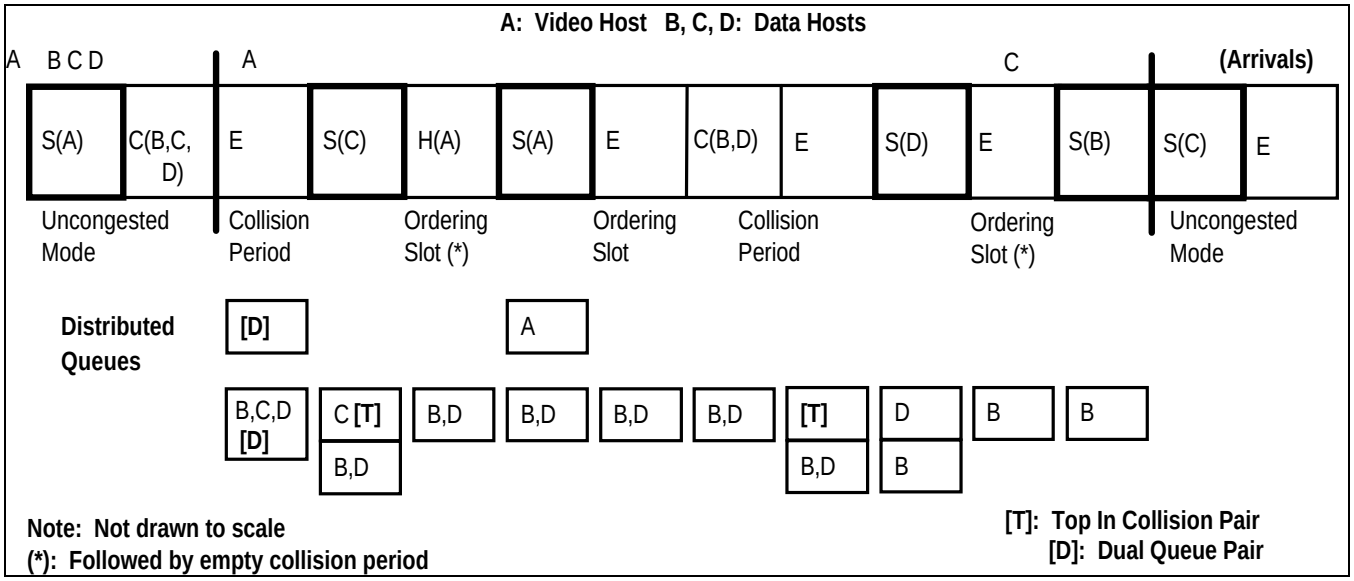
A bucket is removed after it is the top bucket and it is detected to be empty. If it is empty when it becomes the top bucket, then it will leave an empty event in a collision resolution slot and be removed. If it contains a single sender when it becomes the top bucket, then this sender will successfully send its packet and the bucket will be removed. When both queues are reduced to zero buckets, the controllers all enter uncongested mode again, and stay in this mode until the next collision occurs.

5.3: Optimizations

The basic protocol performs quite well with only two integer variables required per controller. When multiple senders collide and redistribute themselves into two buckets, they may all choose the same bucket, in which case an empty bucket is created. When this empty bucket is processed, it will create an empty collision resolution slot. If this empty bucket is the higher priority of the two buckets, the occurrence of the empty slot indicates that the lower priority bucket has at least two packets in it. The best thing to do in this case is to divide the lower priority bucket into two buckets, and have the members of this bucket randomly pick one of the two new buckets.

This optimization requires keeping track of two flags per bucket, one for each type of slot that can create this case. If two buckets are created due to a collision in a ordering slot, they are both marked as members of a Dual Queue Pair (DQPAIR). When the top bucket in the high priority queue is removed and was marked as DQPAIR, the top bucket in the low priority queue that is marked as DQPAIR is unmarked. This keeps the pairs associated together, so that the top DQPAIR bucket in the high priority queue is associated with the top DQPAIR bucket in the low priority queue. If this unmarking is due to an empty bucket, then the low priority DQPAIR bucket is split into two.

If two buckets are created due to a collision in a collision resolution slot, the top one is marked as Top In Collision Pair (TICP). Both packets do not need to be marked in this case because if the top bucket of the pair is empty (the optimized case) then the lower bucket must



always immediately follow it in the queue. So when an empty slot occurs as the result of a bucket that is marked as TICP, the next bucket in that queue is split into two.

5.4 FDDQ rules

Figure 5.2 lists the rules of the FDDQ algorithm. These rules assume that FDDQ or some other protocol has already split up the actual signals from the bus into a series of events. Each returned event is either a full packet send from a single controller (S), a slot containing a collision between two or more controllers (C), an ordering slot filled with one sender (H or L) or an empty slot (E). When the net is in uncongested mode, any event can be returned, with C events putting the controllers into congested mode. Collision period slots can be of type E or C. Ordering slots can be of type E, C, H, or L.

A sample set of these events is shown in figure 5.3, along with the resulting queues. The three classes of events (ordering slots, collision period slots, and successful sends) are demarcated by the successful sends, marked in bold. Immediately succeeding each send is an ordering slot. The events, if any, immediately after the ordering slot and before the next send are collision period slots.

5.5 Implementation of FDDQ

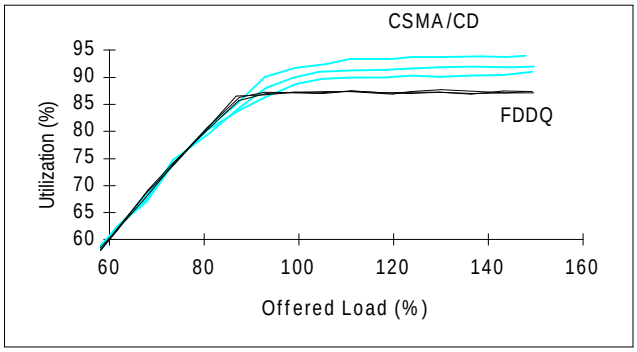


Figure 6.1: Utilization for 20, 40, and 60 Data Hosts

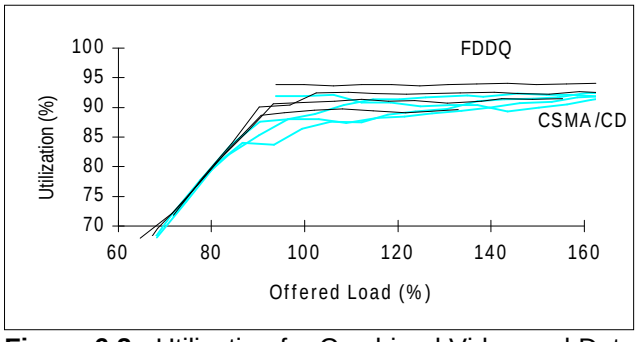


Figure 6.2: Utilization for Combined Video and Data Sources

As shown by the previous set of rules, FDDQ is very simple and should execute very quickly. The amount of state needed is minimal. Each controller needs to know the size of the queue (if any), the position in the queue of its bucket (if any), and a list of which buckets are marked as TICP or DQPAIR. The first two pieces of information are just integers, and while the last piece requires a more complicated data structure, this can be removed if necessary, with only a mild average packet latency penalty. The maximum utilization numbers are not affected by these optimizations. Because many of the current Ethernet chipsets allow the collision detection signal to be input from an external source, FDDQ should be implementable in firmware in a single microcontroller or other chip added to an Ethernet board.

5.6 Analysis of FDDQ

The FDDQ algorithm guarantees FIFO sending behavior between buckets. There is one ordering slot per successful send, so the number of senders in one bucket is limited to the number that arrive between the ends of two consecutive successful sends. This is equal to the time it takes to send a maximum length packet (1.2 ms) plus the maximum length of the collision period between packets, which is easily bounded by 0.5 ms. This is equal to slightly more than 8 collision slots, and so roughly 256 senders would have to send at the same time to cause this many collisions in a row. So even though the order of packets sent within a bucket is arbitrary, the FDQ algorithm guarantees FCFS behavior of packets that arrive at least 1.7 ms apart.

A very nice feature of FDDQ is that it actually behaves better as the offered load increases, as long as the number of senders and the number of packets they can offer at once stays constant. If the senders offer more than 100% load, and back up, FDDQ turns into a round robin scheduler. As one packet is successfully sent, if this controller is backed up, it will immediately request a spot in the next ordering slot. Since all of the other senders are in the queues, it will get into the bucket by itself. This is the ideal case for FDDQ, where exactly one slot is spent per packet sent. Because of this, FDDQ is stable for a constant number of senders.

6. Comparison of FDDQ with CSMA/CD

In this section, we use simulations to compare the utilization, average delay, and standard deviation of delay between CSMA/CD and FDDQ. We show that, for the tested loads, CSMA/CD only achieves higher utilizations than FDDQ at offered loads greater than 90%. Because the PSE makes this range of operation non-viable for

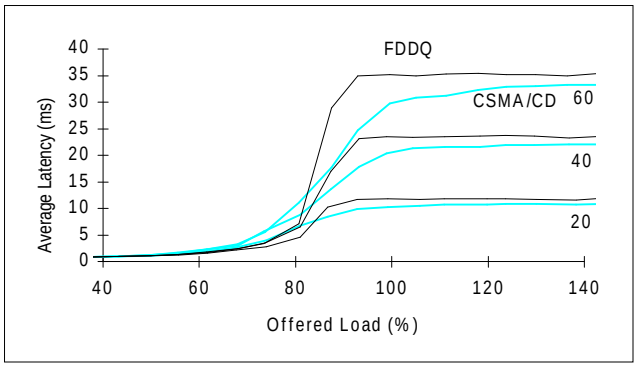


Figure 6.3 Average Latency for 20, 40, and 60 Data Hosts

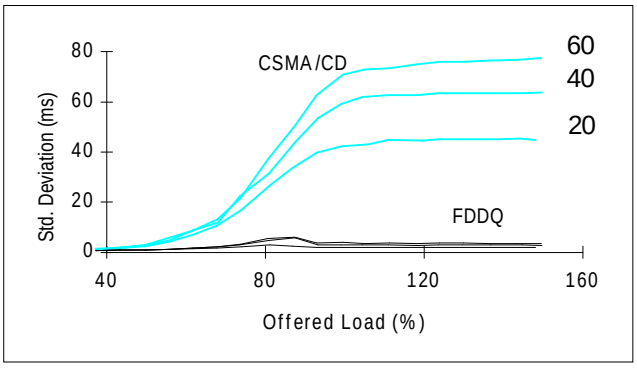


Figure 6.5 Standard Deviation of Latency for 20, 40, and 60 Data Hosts

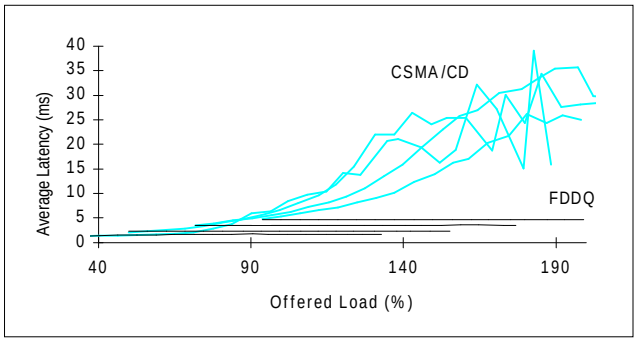


Figure 6.3 Average Latency of Video Packets (Part of Combined Load)

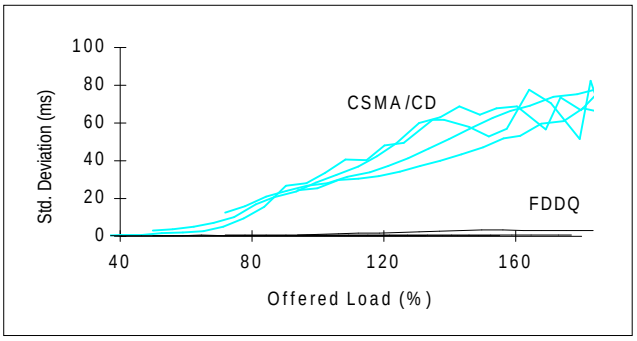


Figure 6.6: Standard Deviation of Video Packet Latency (Part of Combined Load)

CSMA/CD, FDDQ provides utilization at least as high as CSMA/CD for all practical loads. The comparison of average delay mirrors that of utilization. We show that the standard deviation of delay for FDDQ is much smaller than that of CSMA/CD, reflecting the fact that FDDQ completely eliminates the PSE and provides two priority FCFS access.

6.1 Utilization

Figures 6.1 and 6.2 show the utilization of CSMA/CD and FDDQ for the tested data loads and combined data and video loads. The combined loads consisted of between 1 and 4 video streams with data traffic from 40 hosts added on incrementally. For both of these figures, FDDQ is almost identical to CSMA/CD up to an offered load of around 80%. From 80% to 90%, FDDQ has a higher utilization than CSMA/CD, probably due to the numerous starvations that occur in this range. Above 90%, CSMA/CD shows a utilization up to 5% higher than FDDQ for data packets, and for some combined loads. However, at this range, CSMA/CD is experiencing 2% to 15% starvations, depending on the exact offered load and the type of load. This level of starvations is unacceptable for most applications, so the extra utilization is not usable in practice. Note that for the loads with two or more video streams, FDDQ actually has higher utilization, but this is because it is giving prioirity to the longer video packets.

6.2 Average packet latency

Figures 6.3 and 6.4 show the average packet latency for CSMA/CD compared with FDDQ. Figure 6.3 shows the average latency for 20, 40, and 60 data hosts. In this case, FDDQ outperforms CSMA/CD from 60% offered load to 80% offered load, and is worse at levels above 85%. Figure 6.4 shows the average delay suffered by the video packets in the combined loads. Because of its two-priority FCFS access, FDDQ provides lower average latency to the video packets than CSMA/CD does.

6.3 Standard deviation

The main point of FDDQ is to provide dual-priority FCFS access to the bus This greatly reduces the standard deviation of packet latency, as shown by figures 6.5 and 6.6. The deviation that does exist for FDDQ in these graphs is primarily from the burstiness of the sources.

Although we do not provide graphs for it, we note that the PSE is completely eliminated by FDDQ. For the case

of 60 data sources and an offered load of 149% there were 0.06% of the packets which experienced delay of at least 50 ms, which is reasonable since the average delay for this case was 35.5 ms. This was the only tested case that had any packets experience a delay of at least 50 ms, and there was not a single packet starvation in all of the tests.

7. Conclusions and future work

We have shown that Ethernets with CSMA/CD experience the packet starvation effect (PSE) which causes some packets to experience very high delays at high offered loads. This is the primary reason that the standard deviation of packet latency is two to five times the mean for these loads. At high offered loads, the PSE causes real-time traffic to suffer unreasonable delays and loss rates, and limits the usable utilization of a CSMA/CD Ethernet. We quantified the point at where the PSE becomes significant, and concluded that 60% to 70% is probably a realistic offered load limit for current Ethernets unless only one or two sources are involved.

While the simulations in this paper were based on the 10Base5 version of the IEEE 802.3 specification, they will have at least limited applicability to the new 100 Mbps CSMA/CD standards being proposed by the 802.3 committee. There would have been three main differences to the simulations if the new 100 Mbps standards had been used instead of the 10 Mbps 10Base5 standard. First, all of the latency measurements would have been 1/10th what they were. This will decrease the cost of packet starvations when compared to human reaction time and video frame rates. Second, the traffic loads selected will not have as much direct applicability to these new LANs as to 10Base5 LANs, and so might need to be modified. Finally, the hub-based topology of these new LANs may produce different collision effects and costs than the straight-line bus topology used. Despite these differences, we expect the PSE to be a significant problem for real time traffic in these new networks. As real time traffic is a primary driving force behind the new standards, we feel that this is of high importance.

To solve these problems, we proposed the Fair Dual Distributed Queue algorithm. This algorithm provides two priority FCFS scheduling of packets with a scheduling resolution bounded by the maximum time it takes to send a single packet. It provides utilization equal or better than that of CSMA/CD in all but the highest loads, which are impractical for CSMA/CD to sustain because of the PSE. It has comparable average latency to CSMA/CD and eliminates the unfairness of CSMA/CD. FDDQ is stable for a constant number of senders and gracefully handles even extremely high offered loads.

This is a very powerful tool, for with the use of adaptive stream protocols (such as TCP/IP), a bus that utilizes FDDQ can be used at very close to its full capacity. Senders can set up audio and video streams that push the network close to 100% capacity, and they will still see low jitter. When bursty datagram traffic arrives at the network, the instantaneous offered load may exceed 100% for a short while, but this will not affect the real time traffic, and the data traffic will adjust itself so that it does not exceed the capacity of the bus.

Planned future work involves further simulation of FDDQ to quantify its performance in the face of noise on the line or controller faults. Work in progress has shown that FDDQ can provide very tight statistical guarantees on maximum packet latency when used in conjunction with admission control schemes such as the Tenet scheme[6]. Implementation of prototype 10 Mbps FDDQ Ethernet controllers is underway, with 100 Mbps prototypes eventually planned.

References

[1] Barnett, L. “Netsim User’s Manual”, University of Richmond Department of Math and Computer Science Technical Report TR-92-01.

[2] Boggs, D.; Mogul, J.; Kent, C. “Measured Capacity of an Ethernet: Myths and Reality”, *SIGCOMM*, 1988.

[3] Bux, W. "Local-Area Subnetworks: A Performance Comparison", *IEEE Transactions on Communications.* 29(10): 1465-1473, October 1981.

[4] Capetanakis, J. "Tree Algorithms for Packet Broadcast Channels", *IEEE Transactions on Information Theory.* 25(5): 505-515, September 1979.

[5] Coyle, J.; Liu, B. "Finite Population CSMA/CD Networks." *IEEE Transactions on Communications.* 31(11): 1247-1251, January 1985.

[6] Ferrari, D. "Real-time Communication in an Internetwork", *Journal of High Speed Networks.* 1(1):79-103, 1992.

[7] Gallager, R. "Conflict Resolution in Random Access Broadcast Networks", *in Proc. AFOSR Workship on Communication Theory Applications.* Provincetown, MA., Sept. 1978.

[8] Gusella, R. “A measurement study of diskless workstation traffic on an Ethernet”, *IEEE `Transactions on Communications*, Sept. 1990.

[9] Huang, J.; Berger, T. "Delay Analysis of Interval-Searching Contention Resolution Algorithms", *IEEE Transactions on Information Theory.* 31(2): 264-273, March 1985.

[10] Jacobson, V. "Congestion Avoidance and Control." *Proceedings of ACM SIGCOMM '88 Symposium.* Sept. 1988, 314-329.

[11] Jain, R.; Routhier, S. “Packet Trains — Measurements and a New Model for Computer Network Traffic”, *IEEE Journal on Communications*, 1986.

[12] Leland, W. E.; Taqqu, M. “On the Self-Similar Nature of Ethernet Traffic”, *SIGCOMM ‘93*, 1993.

[13] Massey, J. "Collision Resolution Algorithm and Random Access Communications", *in Multiuser Communication Sytems, G. Longo Ed.* New York: Springer-Verlag, 1981, 73-137.

[14] Melamed, B.; Sengupta, B. “TES-Based Traffic Modeling for Performance Evaluation of Integrated Networks”, *IEEE INFOCOM*, 1992.

[15] Metcalfe, R.; Boggs, D. “Ethernet: Distributed Packet Switching for Local Computer Networks”, *Communications of the ACM*, July, 1976.

[16] Metcalfe, R. “Ethernet versus Godzilla”, *Communication Week*, 1984.

[17] Nichols, K.M. “Network performance of packet video on a local area network”, *Eleventh Annual International Phoenix Conference on Computers and Communications*, 1992.

[18] Raychaudhuri, D. "Announced Retransmission Random Access Protocols", *IEEE Transactions on Communications*. 33(11): 1183-1190, November 1985.

[19] Schoch, J.; Hupp, J. “Measured Performance of an Ethernet Local Network”, *Communications of the ACM*, December, 1980.

[20] Smith, W.R.; Kain, R.Y. “Ethernet performance under actual and simulated loads”, *16th Conference on Local Computer Networks*, 1991.

[21] Tsybakov, B.; Mikhailov, V. "Random Multiple Packet Access: Part-and-try Algorithm", *Problems of Information Transmission.* 16(4): 305-317, Oct.-Dec. 1980.

[22] Xu, W.; Campbell, G. "A Distributed Queueing Random Access Protocol For a Broadcast Channel", *SIGCOMM 1993*. September, 1993.

[23] Yegenoglu, F.; Jabbari, B. “Modeling of Motion Classified VBR Video Codecs”, *IEEE INFOCOM*, 1992.