Solving Capture in Switched Two-Node Ethernets by Changing Only One Node

Wayne Hayes
Department of Computer Science
University of Toronto, Toronto, Canada, M5S 1A4.
Phone: 416/978-7321, FAX: 416/978-4765, Internet: wayne@cs.utoronto.ca

Mart L. Molle
Computer Science Department
University of California, Riverside CA, 92521-0304
Phone: 909/787-7354, FAX: 909/787-4643, Internet: mart@cs.ucr.edu

To appear in *Proceedings of the 20th Annual Conference on Local Computer Networks*, Minneapolis, Minn. 1995 Oct 17-18.

Abstract

It is well known that the Ethernet medium access control protocol can cause significant short-term unfairness through a mechanism known as the Capture Effect, and that this unfairness condition is worst in heavily loaded Ethernets with a small number of active nodes. Recently, Ramakrishnan and Yang proposed Capture Avoidance Binary Exponential Backoff (CABEB) to provide 1-packet-per-turn round robin service in the important special case of a 2-node collision domain. In this paper, we introduce an equal time round-robin scheme, in which only one node needs to be modified. In our scheme, the modified node maintains a local copy of the attempts counter of the other node. It uses this information to trigger switching its medium access policy between the two extremes of aggressively persistent and completely passive. As a result, the modified node can control the actions of the other node in such a way that both enjoy fair, low delay, round-robin access to the shared channel.

1 Introduction

1.1 Why a two-node Ethernet?

Nowadays, a single node with a fast Ethernet interface can easily generate traffic, including all protocol processing overhead, fast enough to saturate a 10 megabits per second (Mb/s) Ethernet. Furthermore, network traffic-intensive software — such as remote file systems, remote graphical user interfaces, and World Wide Web — is rapidly gaining in popularity. In the near future, we can expect even more growth due to multimedia, as interactive audio and video conferencing systems become widespread.

An inexpensive way to increase the network bandwidth available to all nodes, while preserving the investment in existing Ethernet equipment, is to segment the network into smaller and smaller collision domains until, in the limit, we have a switched Ethernet system [9]. Here each node has a direct connection to a switch port, so that each collision domain consists of only two nodes: the attached user device and its associated switch port. In this case, the users will no doubt expect to be able to send lots of traffic over their dedicated Ethernet connection. Thus, it is important that a two node Ethernet provides an acceptable quality of service, even under heavy load.

1.2 The Ethernet Capture Effect

Although the medium access control protocol in Ethernet is usually described as 1-persistent CSMA/CD, in reality it is the Binary Exponential Backoff (BEB) algorithm, used to reschedule the next attempt after each collision, that is most responsible for its behaviour under periods of heavy load — especially when the number of active nodes is small. As a result, we now know that heavily loaded Ethernets are generally stable, but can exhibit severe short-term unfairness as one busy node [10] (or a small group of busy nodes, if each one is too slow to saturate the network by itself [6]) "captures" complete control of the network for considerable periods of time.

Omitting inessential details, the Ethernet medium access control protocol is summarized in Figure 1. Notice that the retransmission delay after each collision varies (dramatically!) with the value of the *attempts* counter, and that *attempts* is a local variable to each node. Furthermore, the node pays no attention to what else is happening on the network when it is *not* transmitting: the most important point is that a successful transmission resets the *attempts* counter of the transmitting node, but does not alter the *attempts* counter of any other node.

To see how the Capture Effect works, and for simplicity of argument, we consider a pair of active nodes, each with a long queue of packets to send. We assume that, at time 0, each tries to send its first packet, and both *attempts* counters are 0. Obviously, a collision will occur, and both nodes will increment their

0. Wait to be given a packet to send.
1. Set *attempts* := 0.
2. Wait for silence on the network.
3. Wait 96 bit-times, then attempt transmission. If successful, proceed to Step 0.
4. A collision has occurred. Increment *attempts* by 1. If *attempts* = 16 return failure.
5. Choose a uniform random integer *delay* between 0 and $2^{\min(10,attempts)} - 1$ inclusive.
6. Backoff (sleep) for *delay* slot-times, where a slot is 512 bit times.
7. Proceed to step 2.

Figure 1: *The essential details of the standard Ethernet protocol.*

attempts counters to 1, and independently choose a uniform random delay of either 0 or 1 slots. If both nodes happen to pick the same backoff, another collision will occur and both will choose a new uniform random delay of $0, 1, 2$ or 3 slots, and so on. This pattern continues until the winning node, which we denote **W**, succeeds in transmitting its first packet after the kth collision. In this case, **W** selects the second packet from its queue and resets its *attempts* counter, whereas the losing node, denoted by **L**, retains the value k in its *attempts* counter. If k is large, **L**'s backoff delay may be longer than the transmission time for **W**'s first packet, allowing **W** to transmit another $p - 1$ packets before **L** is even able to make another attempt.

Now consider what happens at the end of **W**'s pth packet transmission. Node **W** is ready to make the first attempt for its $p + 1$st packet (with *attempts*=0), and **L** is ready to make the $k + 1$st attempt for its first packet (with *attempts*=k). Obviously, another collision will occur, and both nodes will increment their respective *attempts* counters to 1 and $k + 1$ and execute another backoff delay. If both of them happen to choose the same value, they increment their respective *attempts* counters to 2 and $k + 2$, and so on. In the general case where their respective *attempts* counters are j and $k + j$, it can be shown that the probability of **W** winning again is

$$\frac{1}{2^j} \sum_{i=1}^{2^j} \frac{2^{k+j} - i}{2^{k+j}} = 1 - \frac{2^j + 1}{2^{k+j+1}},$$

whereas the probability of **L** getting its first win is

$$\frac{1}{2^j} \sum_{i=1}^{2^j} \frac{i - 1}{2^{k+j}} = \frac{2^j - 1}{2^{k+j+1}},$$

and the probability of another collision is $1/2^{k+j}$. Thus **W** is

$$\frac{2^{k+j+1} - 2^j - 1}{2^j - 1}$$

times as likely as **L** to win this contention, and **W**'s advantage grows exponentially with increasing k, becoming approximately 2^{k+1} larger for $k >> j$. For example, if $k = j = 1$, *i.e.,* **W** has transmitted only one packet, and its second packet encounters a collision with **L**'s first packet, then **W** is already 5 times more likely to win. If $k = 2$ and $j = 1$, **W** is 13 times more likely to win, and if $k = 3$ and $j = 1$ then **W** is 29 times more likely to win. In this situation, we say that **W** has *captured* the network. Eventually, **W** will probably empty its send queue and then the network will be completely idle for a time while **L** finishes backing off from the most recent collision. If more packets arrive in **W**'s transmit queue during this time, **L** may be forced to wait even longer, even though the network has seen some idle time. Some time later, **L** will finally begin transmission of its packet queue, probably capturing the network and locking out **W** for a long time.

We define a "run" of packets to be a sequence of packets transmitted by a single node until another node gets a successful transmission — *i.e.,* a collision only terminates a run if the next successful packet is from a different node. Figure 2 shows the mean run length and standard deviation of run length in a

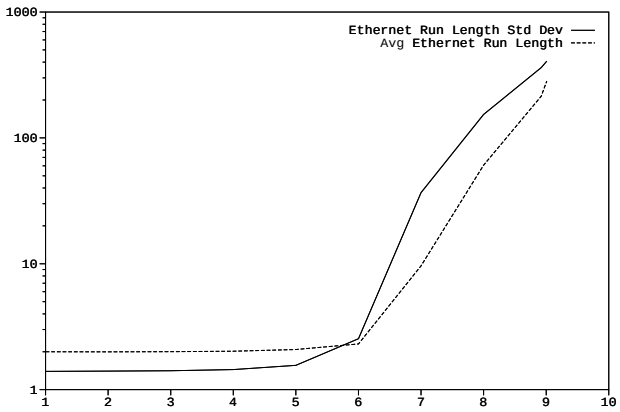

Figure 2: *The Ethernet Capture Effect: Mean and standard deviation (i.e., the square root of the variance) of run length for a Poisson-driven two-node system with constant sized 256 byte packets, as a function of throughput in Mb/s.*

two-node Ethernet as a function of throughput. (The statistics are the same for both nodes, taken from simulations described in the next section.) The system is a simple one with packets of constant length 256 bytes and a global Poisson arrival process with equal load from both nodes; experiments with other arrival processes and packet size distributions, including several derived from packet traces from a real Ethernet, did not change the qualitative nature of these results. At a throughput of 9 Mb/s, the mean run length is 214, indicating that the first packet of the deferring node must wait on average $(256 \times 8 + 96 + 64) \times 214 = 472512$

bit-times, or about 47 milliseconds (ms) at the front of its queue. The 96 bits per packet is the required inter-packet spacing of the Ethernet standard, and the 64 bits represent the preamble. The standard deviation of the run length is 363 (larger than the mean!), so run lengths routinely vary by several times the mean. Thus, on a highly loaded two-node Ethernet, it is not uncommon for the first packet of a run to wait *several tenths of a second* before transmission. Our simulations show that increasing the packet size causes the access delay to increase slightly less than linearly, and the mean run lengths to get shorter, simply because there are more chances (on average, per packet) for the losing node to jump in.

A side effect of the Capture Effect is that extremely high throughputs can be maintained under heavy traffic conditions, since few packets experience any collisions. The big disadvantage is that the variance of access delay is enormous: once **W** gets its first packet through, it can often transmit hundreds of subsequent packets with essentially zero access delay. However, there is no long term unfairness problem: **L**'s first packet may experience a huge access delay, but thereafter **L**, too, transmits many packets with essentially zero delay. Boggs, Mogul, and Kent [2] devised and ran several experiments that massively overloaded an Ethernet. The Capture Effect was not visible in their measurements because they only reported the mean time between a packet getting to the front of its local queue and the time it is transmitted, *i.e.,* the access time. Thus, the huge access delay experienced by the first packet in each run is masked when it is averaged with the negligible access delays experienced by the large number of other packets in the same run. Furthermore, their definition of "delay" does not include queueing effects. Even though only one out of the many packets sent in a single run spends a long time at the *front* of the local transmit queue, the remaining packets had to wait in the queue until that first packet "got out of the way". In fact, it would be possible for the deferring node to accumulate enough packets to overflow its packet queue, possibly resulting in data loss.

1.3 Simulation methodology

The results in this paper are obtained by carefully simulating an Ethernet system at the interface between the medium access control and physical layers. We chose simulation over analytical techniques to permit us to create a more faithful model of the system, without worrying about the ease and/or availability of suitable solution techniques. Furthermore, trace-driven simulation makes it easy to incorporate actual measured traffic patterns from a real network into our study. We chose simulation over direct experimentation because we did not have the means to construct hardware prototypes incorporating our proposed changes to the medium access control protocol.

Our simulator was written using a package called the *Smurph Protocol Modelling Environment* [7]. *Smurph* was designed specifically to allow easy implementation and testing of MAC layer protocols, by conveniently handling all details of the physical layer. *Smurph* allows realistic topologies to be defined, and can simulate many properties of real-world systems, like clock drift and physical layer noise. It uses software multi-word integers rather than floating-point numbers to model time and for counters. This is done because the absolute size of roundoff errors in floating point numbers increases with the magnitude of the number; this effect can cause errors in event-timing and in statistics taken during a long-running simulation. *Smurph* is written in C++, and requires the user to implement their protocol as C++ event routines implementing a finite-state machine. The entire system is compiled to execute at machine-code speeds. Thus, it is easy to simulate complex systems for long periods of simulated time with minimal CPU. *Smurph* is fast enough that we routinely ran our simulations for one hour of simulated time, simulating the transmission of millions of packets, in about an hour of CPU time on a Sun SPARCstation IPC.

Our finite state machine has been written to precisely conform to the IEEE 802.3 Ethernet Medium Access Layer [1] down to the bit level — including all the details of inter-frame spacing, preamble, Start Frame Delimiter, header and packet size limits. We validated our simulator by reproducing the setup of a famous and widely-distributed experiment on a real Ethernet by Boggs, Mogul, and Kent [2], and getting identical results in all measurements, within statistical uncertainties. See Molle [6] for a direct comparison of the published measurements by Boggs *et al.* with our simulation output.

In addition, we have taken packet traces of a local undergraduate computing facility. For each packet, we recorded the inter-arrival time (to the nearest microsecond), size (in bytes), source and destination addresses. These traces were manipulated (see section 3.2.1) to produce simulated two-node traffic, and in all cases the results of our experiments were qualitatively similar: the Ethernet Capture Effect always occurs in systems with two standard nodes, and the protocol introduced below eliminates the Capture Effect in all cases tested.

2 SHEP: The Switched Half-duplex Ethernet Protocol

2.1 Overview

The standard Binary Exponential Backoff algorithm was intended to work for systems with a large number of relatively inactive nodes. In systems with a small number of very active nodes, the losers get tricked by repeat collisions with the same winner into (drastically!) overestimating the number of active nodes. Consequently, they end up backing off too far and too quickly whenever congestion occurs, which leads to the Capture Effect.

Although other solutions to the Capture Effect have been proposed [6, 8], one must install the modification in all nodes of the network to obtain their full benefit. The Switched Half-duplex Ethernet Protocol is unique in that it requires changing only one side of the two-node system, allowing users to upgrade their entire network by adding one new piece of hardware (*i.e.,* a

SHEP compatible switch). This backwards compatibility is a significant advantage for our proposal in comparison to the Capture Avoidance Binary Exponential Backoff algorithm [8] (and even to Full-Duplex Ethernet), which gives little (or no) benefit without updating both ends of the link.

SHEP fulfills these requirements by following a strategy somewhat analogous to a trial judge having a discussion with a rude lawyer who incessantly interrupts the judge. The judge repeatedly asks the lawyer not to interrupt. Then, when the judge is finished, the lawyer is allowed to speak freely for a certain amount of time. In our case the changed node **H** at the switch hub (the judge), controls a round-robin service between itself and the standard Ethernet node **S** (the lawyer). **H** does this by alternately forcing its own packets onto the wire, and then leaving the network wide open. When **H** decides to have a turn, it transmits its packets and chooses a backoff delay of 0 for any collisions that occur during its turn. Since **S** is the only other node on the network, **H** can keep track of the *attempts* counter of **S**. After a stochastic amount of time T this counter will reach a predetermined maximum M, at which time **H** stops transmitting and allows **S** to transmit unhindered for the same amount of time T. Then the cycle is repeated, starting from **H**'s turn.

2.2 Detailed description

In this section, we present a state machine description of the transmit process used by the hub node **H**, which is running the Switched Half-duplex Ethernet Protocol (SHEP). The internal state variables used by the SHEP node are:

- *otherAttempts*: local copy of the *attempts* counter of the standard node **S**.
- *otherStartedWaitingAt*: the most recent time **S**'s *attempts* counter was incremented from 0 to 1. It marks the formal beginning of **H**'s turn; **H**'s turn finishes a short time after **S**'s collision counter reaches M.
- *weStoppedAt*: the time marking the end of the last packet of **H**'s turn. The difference between this and *otherStartedWaitingAt* is the time interval **H** was allowed to transmit, and is the length of time **S** will be allowed to transmit unhindered. This time-based round-robin is intended to be more fair than a packet-based round-robin service schedule in the case that the average size of one node's packets is different than the other's. (This is not uncommon depending, for example, upon whether a node is a file server, or a client.)
- *currentTime*: the wall clock time as it appears at any particular step. One of its uses is in the computation of when **S**'s turn will finish because, to be fair, we need to split the idle time that occurs between the end of **H**'s last packet and the beginning of **S**'s first packet.

The SHEP transmit state machine is presented in the following pseudocode. In addition, node **H**'s receive process should set *otherAttempts* to 0 whenever it receives a packet.

0. Wait until a packet arrives in the transmit queue.
1. If the network is busy, then the other host is transmitting, so set $otherAttempts := 0$. Wait for silence on the network plus the required inter-packet gap.
2. Attempt transmission. If a collision occurs:
   a) Increment *otherAttempts* by 1.
   b) If $otherAttempts = 1$, set *otherStartedWaitingAt* to the current time.
   c) Abort, jam and wait as in standard Ethernet, but then proceed to Step 2. (*i.e.,* choose a 0 backoff delay.)

   Otherwise, our packet was successfully transmitted: proceed to Step 3.
3. If $otherAttempts \geq M$, or **H**'s queue is empty and $otherAttempts > 0$, concede the end of **H**'s turn by proceeding to Step 4. Otherwise it is still **H**'s turn, so proceed to Step 0.
4. Concede control of the network to **S**. We know that **S** has a packet to transmit because $otherAttempts > 0$. Set *weStoppedAt* to the current time. Wait for the beginning of **S**'s first packet, then proceed to Step 5. Note that there may be some idle time while **S** finishes backing off.[1]
5. We are now hearing the first packet of **S**'s turn. Set $otherAttempts := 0$. We split the idle time we just saw between the turns of the two nodes, and compute when it will again be **H**'s turn as:
   $turnLength := weStoppedAt - otherStartedWaitingAt$
   $idleTime := currentTime - weStoppedAt$
   $ourTurnAgainAt := currentTime + turnLength + idleTime/2$.
   From steps 5, 6 and 7, we always watch to see if **S**'s turn is over, at which time we proceed to Step 0. Otherwise we wait for the end of **S**'s first packet, and proceed to Step 6.
6. There is silence on the network but it is still **S**'s turn. If the end of its turn arrives, go to Step 0. Otherwise wait to hear the next packet. If the inter-packet gap + some small grace period expires[2], without seeing another transmission from **S**, then assume **S** has no more packets to send, and proceed to Step 0. Otherwise we see a new packet, and proceed to Step 7.
7. We are now hearing a packet from **S** other than the first one of its turn. If our turn arrives, proceed to Step 0. Otherwise wait for the end of this packet and proceed to Step 6.

[1] In the unlikely event that **S** somehow doesn't transmit the packet, this step of the SHEP protocol should implement some sort of timeout.

[2] We used twice the 96 bit-time inter-packet gap as our grace period, as does Molle [6].

2.3 Discussion

Some qualitative observations of this protocol exhibit some of its properties. First, there is the obvious fairness of an equal-time round-robin service schedule. Second, although there is no strict upper bound on the length of a turn, it is clear that service is far more predictable than standard Ethernet, since the time bound on the length of a turn is much more strict than simply allowing long run lengths. Third, the "predictability" of delivery is controlled by the single parameter M, the predetermined maximum *attempts* counter of the standard node. Fourth, the protocol is striking in its simplicity.

There is a slight cost in throughput at high load due to the extra collisions and idle time in this protocol. The cost is highest with the smallest packets (maximum capacity of 5 Mb/s for 64 byte packets compared to about 7 Mb/s for standard Ethernet), since proportionally less time is spent servicing them than in inter-packet gaps and idle time between turns. This cost quickly becomes negligible with packets only 2 to 4 times the minimum size (maximum capacities of 7 *vs.* 8.5 Mb/s and 8.2 *vs.* 8.9 Mb/s, respectively) and is almost immeasurable with the largest packets (about 9.5 *vs.* 9.6 Mb/s). We believe the large cost for tiny packets is unimportant because a 64-byte Ethernet packet contains almost no user information, and it is difficult to imagine an application requiring continuous back-to-back packets containing little information.[3]

There is some choice as to exactly when node **H** should concede control of the network. If guaranteed access delays are important, then two possible choices are (1) concede immediately after M collisions, and (2) attempt transmission of the current packet one more time after the Mth collision. (The motivation for choice 2 is that there is likely to be some idle time anyway, so we should attempt to use it even though there is a slight chance of another collision.) A third choice is (3) Re-attempt transmission of the current packet as many times as necessary to get it through, even though this will occasionally raise *otherAttempts* significantly above M, causing an occasional long idle period after the end of **H**'s turn. Either of choices 1 or 2 will guarantee an upper bound on the length of a turn when the other node is waiting to transmit, since **S**'s collision counter will never get above M or $M+1$, respectively, and this puts an upper bound on the length of **H**'s turn, and thus **S**'s. If a completely predictable turn length is not a paramount issue, then analysis and experiment has shown that choice 3 is slightly better than the others, since it decreases the mean idle time between turns (and, therefore, increases the capacity) at little loss in predictability for the access times. The code in step 2 listed above implements the third choice, as do all simulations reported in this paper.

[3] Note that although 72-byte packets are common on modern networks, they are not common enough to come in long back-to-back streams that would saturate a 2-node network.

3 Simulation results

3.1 Poisson traffic

We ran thousands of long *Smurph* [7] simulations of two-node systems consuming several CPU-years on a collection of 70 SPARCstation IPCs. We did simulations duplicating the experiments of Boggs *et al.* [2], including varying the number of nodes running standard Ethernet protocol while validating our simulator. Our results accurately duplicated all their measures within statistical uncertainties. More validation details can be found in Molle [6], which used the same Smurph finite state machine for its standard Ethernet as was used for SHEP results. We also varied the distance between the two nodes, the load, the distribution of load between nodes, the time between back-to-back packets from the same node, the distribution of packet sizes, and the traffic patterns (including trace-driven packet sizes and arrival times, as described in section 3.2.1.) In all cases, the results were qualitatively identical to the results we present below: with two standard nodes, the Capture Effect is observable even at loads well below saturation and the effect gets much worse as the load approaches saturation; use of SHEP eliminates the Capture Effect and provides efficient round-robin service.

We found through experiment that the best choice for the maximum collision counter M is 1, except on very long networks where an M of 2 or 3 may be used. We were quite surprised because this value of M seems rather small, but our simulations clearly show that using values greater than 1 only negligibly increases the maximum throughput while substantially increasing variance of delay. Both these effects are due to the increased idle time between the end of **H**'s turn and the beginning of **S**'s turn, while **S** finishes its final backoff. To substantially increase throughput requires setting M greater than about 10, which simply results in high mean run lengths and high variance of delay, similar to the Capture Effect.

Figure 3 is the same as Figure 2 with the addition of statistics for SHEP. SHEP's fairness is such that the two nodes have identical statistics, so we only show the statistics for one node. The mean run lengths are of order 2, rather than 200. The standard deviation *decreases* as the load increases, signifying almost perfect round-robin service. Figure 4 shows the mean access delay for both the standard protocol and SHEP. SHEP offers slightly lower access delay until the very highest loads. Furthermore, the delay graph is the same for both nodes. Figure 5 shows the global standard deviation of delay for both protocols. The standard deviation of delay for SHEP is more than an order of magnitude smaller than that of the standard protocol. Figure 6 shows the single largest observed access delay for the standard protocol and for the SHEP protocol. The difference is about two orders of magnitude. It is the rare but huge access delays in the standard protocol that cause the most trouble to higher-level protocols, and to users. SHEP completely eliminates the effect.

Finally, we mention two other related results. First, if there is more than one standard node on a network

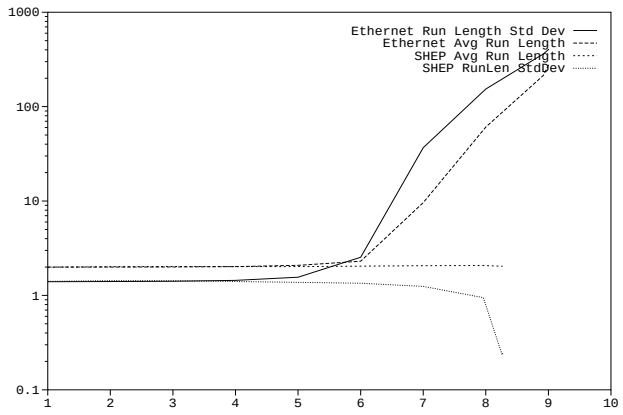

Figure 3: *This is Figure 2 with the same run length statistics included for when one node runs the SHEP protocol, vs. throughput. Note the vertical axis is logarithmic, as are all figures in this paper. The mean run length and its standard deviation are almost 2 orders of magnitude smaller at high load. For both protocols, the statistics are only shown for one node, because for a given protocol the statistics are identical for both nodes. In the case of two nodes running the standard Ethernet protocol, this is obvious by symmetry; it is by design in SHEP. Note the slightly lower capacity of SHEP (8.3 Mb/s) vs. the standard protocol (9.0 Mb/s).*

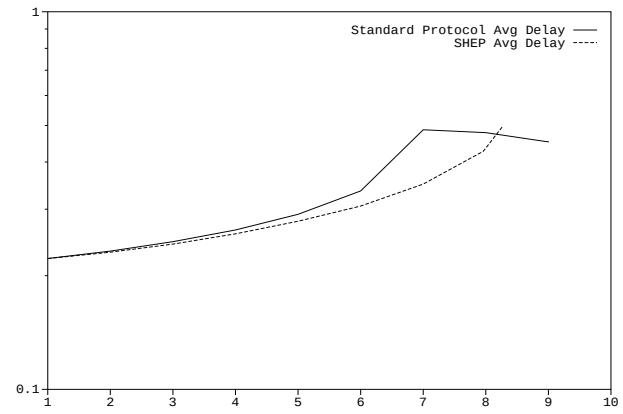

Figure 4: *Comparison between the mean access delay in milliseconds for the standard protocol and SHEP. SHEP gives lower delay until the very highest loads. The graph is also the same for both nodes.*

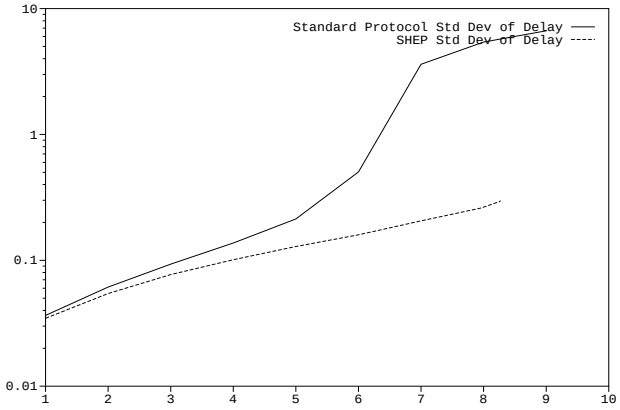

Figure 5: *Comparison of the standard deviation of access delay in milliseconds for both the standard Ethernet protocol and SHEP. SHEP gives an order of magnitude smaller standard deviation of delay. Again, this graph is the same for both nodes.*

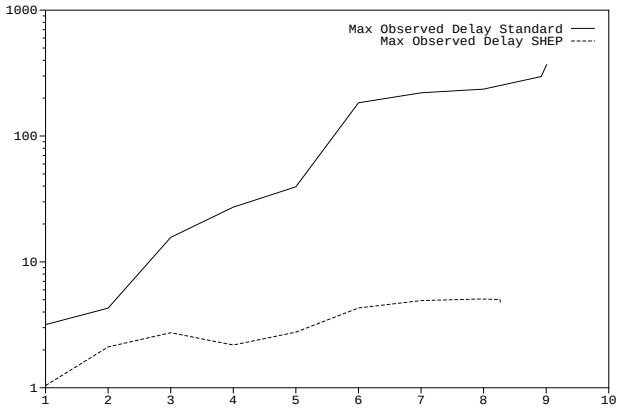

Figure 6: *Single maximum observed access delay during our simulation, in milliseconds, for both protocols. With SHEP, the maximum observed delay at a throughput of 8.3 Mbits/s was only 4 milliseconds, as compared to about 300 ms for the standard protocol.*

with 1 SHEP node, the system will not crash, and in fact the presence of the SHEP node slightly decreases delay variances of all nodes, although the effect is far less than in an actual two-node system. See Molle [6] for a protocol which allows good fairness effects for more than 2 nodes.

Second, extremely poor performance would result if more than one SHEP node attempted to control the same collision domain: both will quickly get into the "attempt transmission" state and, thereafter, little progress will be made as each tries to beat the other into submission with repeated 0-delay backoffs. Although we could modify SHEP to be less aggressive (for example, in SHEP.5 the node transmits with probability 0.5 at every slot), a better solution would be to ensure that only one host is using SHEP, using either the Spanning Tree Algorithm for bridges or the link based auto-negotiation protocol in the 100BaseT standard. This is because our experiments have shown that modifying SHEP to SHEP.5 is a poor compromise: using SHEP.5 instead of SHEP in combination with a standard Ethernet node hurts performance, whereas if both nodes can be non-standard then better performance is available using other protocols, such as CABEB [8], that are specifically optimized for 2-node co-operation.

3.2 Packet trace driven traffic

3.2.1 Packet trace environment

The traces were taken on the University of Toronto's Undergraduate *Computing Disciplines Facility* (*CDF*), which consists of two separate Ethernets on two floors. Our trace machine was on the second floor Ethernet, which connects 28 Sparc IPC workstations, one SPARCserver 490 file server, one SPARCserver 690 model 140 compute server, a smart console attached to the file server, plus the trace machine, which was passive the vast majority of the time. Any other addresses appearing in the traces (including all the machines downstairs) were mapped into one node called the "Outside" node. Each workstation has a 200MB local disk containing /tmp, a swap partition and all of the standard Unix binaries. Student home directories are on the file server, accessed via NFS. Typical traffic on the network consists of NFS accesses, remote interactive logins, remote X-Window traffic, and occasional distributed ray tracing using PVM [4]. Traces were taken intermittently over many weeks using the *tcpdump* program, tuned for low processing overhead. Usages on the network varied from the facility full of students doing programming assignments, to having 35 remote *Matlab*s [5] and *Xmaple*s [3] on the compute server, to days of many students playing Xtrek and NetTrek games. The average packet size was usually about 140 bytes, roughly bimodal between the smallest and largest packet sizes. Each trace lasted 1 hour, with the load average ranging from 2% one evening to over 60% on occasion. A typical trace with a 1-hour average load of 7% would have 1-minute averages varying usually between 1% and 20%, and never observed over 70%. Loads were distributed across nodes very asymmetrically, with the file server being the source and destination of the greatest number of packets, followed by either the compute server or the NetTrek game server. Most other nodes had negligible loads in comparison. We do not attempt to correct for propagation delay, because our network is small — the maximum delay between any two nodes does not exceed about 200 feet (less than 4 bit times).

These traces are still not ideal, because they only include *successful* transmissions, *i.e.,* packet *departures*. Even though we can subtract the transmission time from the packet departure time to get its beginning-of-transmission time, there is still the possibility that the packet participated in deferrals or collisions before its final successful transmission. Thus, the calculated beginning-of-transmission time is only an upper bound on the arrival time. We attempted to minimize this effect by discarding entire traces whose average load was above 10% Ethernet capacity. Even so, many of the remaining traces were during hours of interesting student traffic.

Since the traces only include successful transmissions (*i.e.,* the *final output* of the medium access control algorithm), driving the *inputs* to the model from a single trace file will produce *no* collisions in the simulation. We also want to produce simulated high load traffic. Our solution was to "overlay", or *merge* multiple traces into one to achieve higher loads. Finally, to produce two-node traffic, the file server was node 0, and the compute server plus all the workstations were wrapped into node 1.

3.3 Trace driven simulation results

Figure 7 shows the throughput as a function of offered load for this simulated two-node system. For both protocols, it shows the total throughput from both nodes, as well as the individual throughputs of the two nodes. The same traffic pattern (*i.e.,* same set of merged traces) was used in both tests. In the SHEP case, node 0 (the file server) ran the SHEP protocol, while node 1 ran the standard Ethernet protocol. The file server consistently has more traffic to send than the other node. It can also be seen that at high load, it is the file server that accounts for most of SHEP's lost capacity. This is probably because it has more data to send, and an equal-time round-robin service favours the less busy node. Figure 8 compares the mean run length of the two protocols, showing both nodes in each case. The standard protocol averages up to 100 packets per run, while SHEP averages about 3 for the file server, and 2 for the (less busy) other node. Figure 9 compares the standard deviation of the run length for both nodes and both protocols, clearly showing that the standard protocol consistently has runs lengths well into the hundreds of packets, while SHEP maintains a fair, round-robin schedule at any offered load, limiting run lengths to about 3 for the file server, and 2 for the other node.

Figures 10 and 11 compare the mean access delay and its standard deviation, respectively. These Figures tell roughly the same story as Figures 4 and 5: SHEP has slightly lower mean access delay and drastically lower variance of access delay. Finally, Figure 12 shows the single maximum access delay for both protocols, and like Figure 6 shows that SHEP limits

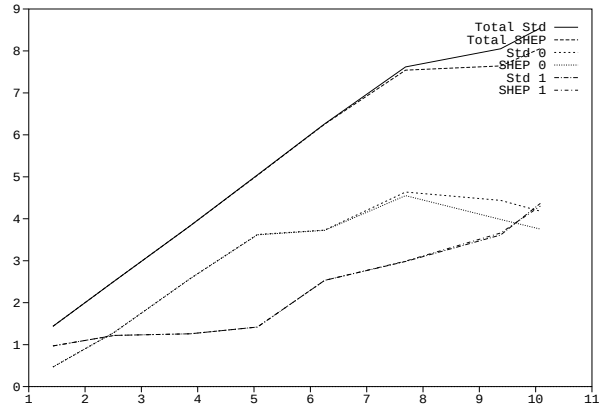

Figure 7: *Throughput as a function of offered load in Mb/s for trace-driven simulation. Curves labelled "Std" represent a standard 2-node Ethernet; ones labelled "SHEP" represent when node 0 runs the SHEP protocol, while node 1 runs the standard Ethernet protocol. Note that the load is asymmetric; node 0 (the file server) transmits more information than node 1 (the rest of the workstations, which form our simulated "other node" in our two-node system). Also, it is clear here that the SHEP protocol has a slightly lower throughput at high load.*

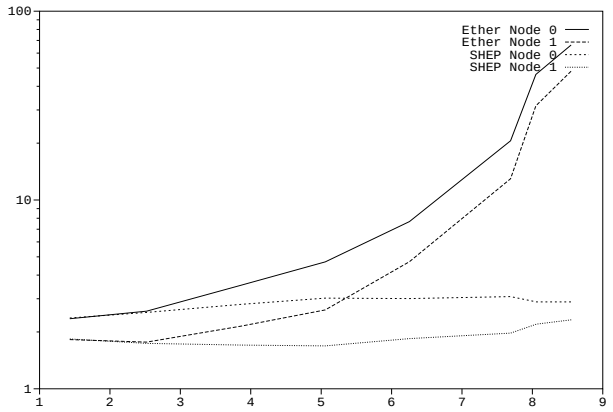

Figure 8: *Mean run lengths for each node using the standard Ethernet protocol, and SHEP.*

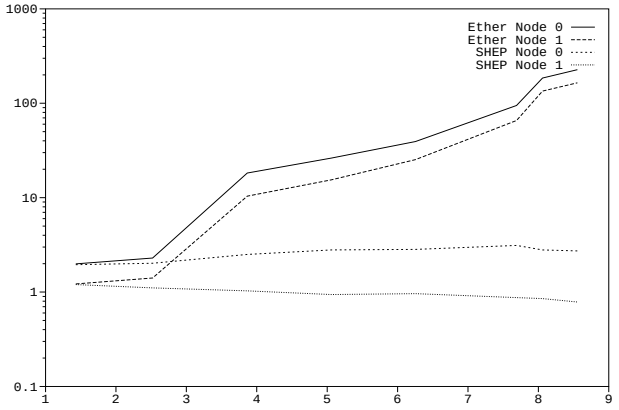

Figure 9: *Run length standard deviations of each node for both protocols.*

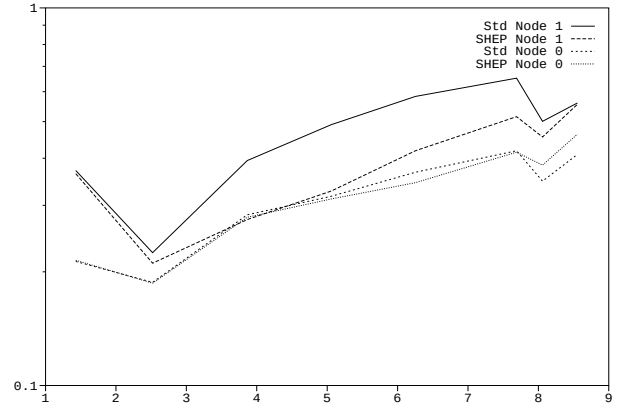

Figure 10: *Mean access delay in milliseconds (ms) for each node for both protocols.*

the maximum access delay to almost two orders of magnitude smaller than the standard protocol.

There are no qualitative changes to the figures if we swap protocols being run on the two nodes (*i.e.,* the file server as a standard node and the other node running SHEP). The only two non-negligible quantitative changes are that both run length curves (Figure 8) slope down, and the access delay standard deviations (Figure 11) reach a little higher to about 0.8ms at the highest loads.

4 Comparison with CABEB

Last year, Ramakrishnan and Yang introduced the Capture Avoidance Binary Exponential Backoff [8]. The most obvious and important difference from CABEB is that SHEP requires only one node be changed. Referring to Tables 6.1 through 6.3 of [8], if a similar update is attempted with CABEB, the CABEB node gains a significant throughput advantage against the standard node. With 64 byte packets, the CABEB node gets a throughput of 4.1 *vs.* 3.5 Mb/s for the standard node. Although SHEP has a capacity of only 5.0 Mb/s with 64 byte packets, we believe a network saturated both ways with 64-byte packets is unrealistic. Even with 1500 byte packets, the CABEB node gets 5.4 *vs.* 4.4 Mb/s for the standard node, whereas SHEP allows equal access. In the case of asymmetrical offered loads, their Table 6.3 indicates that a CABEB node will send most of its packets, at the expense of the standard node, *even if* the standard node has a higher offered load.

If both nodes are updated with CABEB, we believe

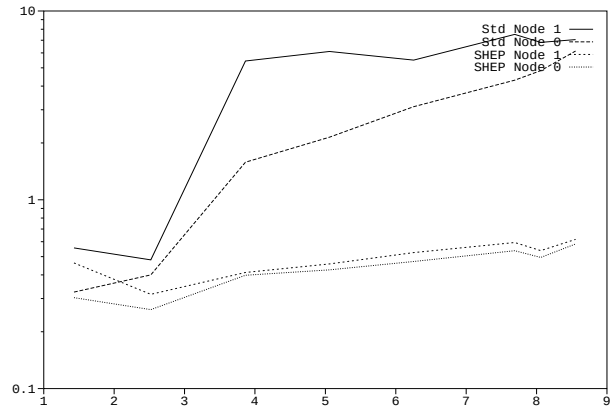

Figure 11: *Access delay standard deviation (ms) for each node, both protocols.*

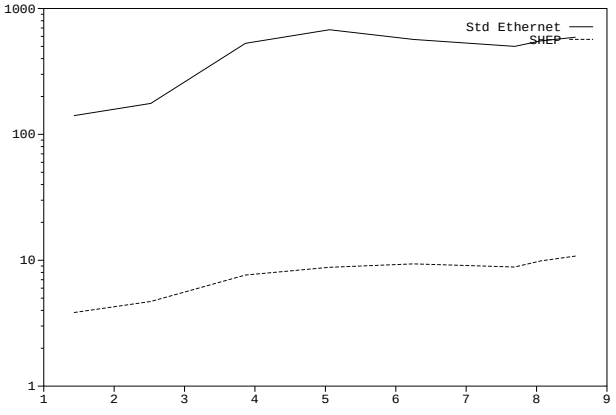

Figure 12: *Single maximum observed access delay (ms) over an hour for both protocols.*

SHEP still offers some advantages. Since CABEB implements a 1-packet-per-turn round-robin, a high but asymmetrical load with packet sizes s_1 and s_2 means that the ratio of the throughputs is forced to a constant of $\sim \frac{s_2}{s_1}$. On the other hand, SHEP is more dynamic, allowing each host to use as much bandwidth as it needs until the network becomes saturated, at which point it enforces equal time for each host.

5 Conclusions

We have presented a solution to the two-node Ethernet Capture Effect in which only one node needs to be changed. This protocol has the advantages of simplicity, fairness, slightly reduced delay, and drastically reduced variance of delay and maximum delay, all at a minor cost in capacity. We have simulated thousands of two-node systems with various combinations of many parameters and found these results to be consistent across all systems considered.

Acknowledgements

This research was conducted while both authors were at the University of Toronto, and was supported in part by the Natural Sciences and Engineering Research Council of Canada, under grant #A5517, and by the Information Technology Research Centre of the Province of Ontario, Canada. We would also like to thank the Computing Disciplines Facility (*CDF*) at the University of Toronto for allowing us to take packet traces, and for allowing us to consume vast quantities of CPU time for our simulations.

References

[1] "Carrier Sense Multiple Access with Collision Detection (CSMA/CD)". IEEE Std 802.3-1990 Edition (ISO/DIS 8802-3), IEEE, New York (1990).

[2] D. R. Boggs, J. C. Mogul, and C. A. Kent. "Measured Capacity of an Ethernet: Myths and Reality". *ACM SIGCOMM '88 on Communications Architectures & Protocols*, pp. 222-234 (Aug. 16-19, 1988).

[3] Bruce W. Char. *Maple user's guide.* WATCOM, Waterloo, Ontario, Canada. (1985)

[4] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, Vaidy Sunderam. *PVM: Parallel Virtual Machine:A Users' Guide and Tutorial for Network Parallel Computing.* Available via anonymous ftp from `netlib2.cs.utk.edu:/pvm3/book/pvm-book.ps`.

[5] *MATLAB, high-performance numeric computation and visualization software : user's guide : for UNIX workstations.* MathWorks, Natick, Mass. (1992)

[6] Mart L. Molle. "A New Binary Logarithmic Arbitration Method for Ethernet". Technical Report CSRI-298, available by *ftp* from `ftp.cs.utoronto.ca`. (July 1994)

[7] P. Gburzynski and P. Rudnicki. *The SMURPH Protocol Modelling Environment.* Department of Computing Science, University of Alberta, Edmonton, Canada. (October 1991). The author may be contacted as `pawel@cs.ualberta.ca`.

[8] K. K. Ramakrishnan and Henry Yang. "The Ethernet Capture Effect: Analysis and Solution". *Proc. 19th Conference on Local Computer Networks*, Minneapolis Minn, October 1994, pp.228-240.

[9] S. Salamone, "Ethernet Networking—Switching Hubs Get Their Due". *BYTE*, **19**:6, p. 28. (1994)

[10] S. Shenker. "Some Conjectures on the Behavior of Acknowledgement-Based Transmission Control of Random Access Communications Channels". *ACM SIGMETRICS '87 Conference on Measurement and Modeling of Computer Systems*, pp. 245-255 (May 1987).