Adding Adaptive Flow Control to Swift/RAID

Chane L. Fullmer, Darrell D. E. Long[†]
Computer and Information Sciences
University of California, Santa Cruz

Luis-Felipe Cabrera
Computer Science Department
IBM Almaden Research Center

January 12, 1995

Abstract

We discuss an adaptive flow control mechanism for the Swift/RAID distributed file system. Our goal is to achieve near-optimal performance on heterogeneous networks where available load capacity varies due to other network traffic. The original Swift/RAID prototype used synchronous communication, achieving throughput considerably less than available network capacity. We designed and implemented an adaptive flow control mechanism that provides greatly improved performance.

Our design uses a simple automatic repeat request (ARQ) go back N protocol coupled with the congestion avoidance and control mechanism developed for the Transmission Control Protocol (TCP). The Swift/RAID implementation contains a *transfer plan executor* to isolate all of the communications code from the rest of Swift. The adaptive flow control design was implemented entirely in this module.

Results from experimental data show the adaptive design achieving an increase in throughput for reads from 671 KB/s for the original synchronous implementation to 927 KB/s (a 38% increase) for the adaptive prototype, and an increase from 375 KB/s to 559 KB/s (a 49% increase) in write throughput.

1 Introduction

Multimedia and scientific visualization require huge files containing images or digitized sounds. Current systems can only offer a fraction of the data rates required by these applications. The Swift distributed file system architecture [4] was introduced to address this problem. Here we consider the problem of using the system in an efficient manner to maximize network throughput.

Due to network flow problems, synchronous operation of Swift/RAID was thought to be necessary. However, this resulted in diminished throughput due to the large waiting times. To achieve a higher throughput an adaptive flow control scheme has been designed and implemented. The design is based on an *ARQ go back N* protocol and includes the adaptive congestion avoidance and control techniques used for the Transmission Control Protocol (TCP) [7]. This design has allowed asynchronous operation of the prototype with an increase in throughput from 671 KB/s to 927 KB/s over the synchronous operation for RAID level 4 reads and from 380 KB/s to 482 KB/s for writes, and RAID level 5 shows an increase in throughput from 729 KB/s to 896 KB/s for reads and from 375 KB/s to 559 KB/s for writes. Our design has also attained an increase of 28% over the reported throughput of the original *non-redundant* Swift prototype [4, 9] (about 700 KB/s versus 927 KB/s for the new design) for read operations (both RAID levels 4 and 5), and has achieved one half of the write throughput for RAID level 4 (about 900 KB/s versus 482 KB/s for the new design) and 60% of the throughput for RAID level 5 write operations (about 900 KB/s versus 559 KB/s).

2 Swift Distributed File System Architecture

Our goal is to study how the Swift [4, 3] architecture can make the most effective use of available network capacity. Swift is designed to support high data rates in a general purpose distributed system. It is built on the notion of striping data over multiple storage agents and driving them in parallel. It assumes that data objects are produced and consumed by clients and that the objects are managed by the several components of Swift. In particular, the *distribution agents, storage mediators*, and *storage agents* are involved in planning and actual data transfer operations between the client and an array of disks, which are the principal storage media. We refer the reader to [4, 3] for details of the functionality of these components of Swift.

The communications protocols used in the original Swift system operated in a synchronous manner. Packets were transmitted one at a time and the sender waited until an acknowledgment was received from the destination before new packets were sent. This type of operation amounts to an automatic repeat request (ARQ) Stop-and-Wait protocol [1]. The protocol is simple and error free, but does not achieve as high a throughput as is possible with other protocols such as *ARQ Go back N* [1].

The Swift/RAID prototype was developed to add redundancy to the original Swift system [9]. The prototype uses a *transfer plan executor* to execute a *transfer plan set* (one transfer plan set is generated per user request). Execution begins when the client transfer plan executor has sent the transfer

[†]Supported in part by the Office of Naval Research under Grant N00014-92-J-1807 and by the National Science Foundation under Grant NSF CCR-9111220.

plan to each server. Each transfer plan executor then steps through the transfer plan in parallel, executing the individual instructions. The instruction op-codes available allow all of the basic file operations and parity calculations. Also included are synchronization primitives to provide synchronization between the client and servers. The prototype implements the RAID-0 (no parity), RAID-4 (fixed parity node) and RAID-5 (distributed parity node) configurations.

The prototype handles errors in two ways – *time-out*, or *restart*. If a node is waiting for an instruction from its peer and the instruction is not received from the network within a fixed time, the waiting node issues a *restart* instruction to its peer. Or, if a node receives an instruction out of sequence a *restart* is also generated to re-synchronize the instruction plans. In either case, we assume a packet has been lost on the network, very likely due to congestion.

3 Improving Network Throughput

Our goal is to maximize the network throughput between a client making requests to the file system and the servers in the file system, and additionally, that of maintaining good utilization in a heterogeneous environment. To maximize the performance of the Swift/RAID prototype, the following three issues need to be addressed: saturation of client/server network buffers; expense of time-outs; and the variability and unpredictability of available network load capacity.

Burst mode operation can easily flood the network and overwhelm the Sun implementation of the User Datagram Protocol (UDP). The Swift/RAID prototype transaction driver, using a *burst mode* of operation, can push data to the UDP layer too quickly, causing data to be dropped and throughput to approach zero.

An examination of the Sun operating system kernel source code showed that packets are likely to be lost while waiting for transmission in the Ethernet queue. This queue is the final location for a datagram before being placed onto the physical network. The queue size is limited to 50 Ethernet packets.* When the queue exceeds the maximum allowed, it *randomly* removes packets from the queue and drops them without notifying any other part of the system. There is no obvious way to determine when this happens and it makes timely congestion detection extremely difficult. Consider that an 8 KB datagram packet is fragmented into five full Ethernet packets and one partial packet. When the client makes a write request to a three node system of over three packets per node in one pass, the client kernel Ethernet queue is immediately inundated with 54 packets to deliver. The kernel simply starts dropping the packets and throughput is seriously diminished.

*Increasing the kernel buffers is a temporary fix – it just masks the underlying problem.

3.1 Flow Control Mechanisms

The underlying system can be thought of as a group of buffers interconnected by a fixed capacity pipe. The servers each have a pair of buffers for the client: one for input and one for output. The client has one pair of buffers for each server that connect to another pair of buffers for the network. Each of these buffers have finite capacities in addition to the fixed capacity of the pipe. When any of these capacities are exceeded packets will be dropped. These buffers can also be thought of as *windows* providing an upper bound on the amount of data a transmitter can expect to send before receiving an acknowledgment from the destination for more data.

Because of the buffering needs of the Swift/RAID system, it lends itself to a class of flow control mechanisms known as *window flow control*. The communications primitives available to us at the client/server session level, the specific class of *end-to-end windowing* (also know as entry-to-exit flow control [6]) make it well suited for our prototype. In this scheme, the sender has knowledge of the destination's buffer capacity and only sends packets out in batches of sizes less up to the available buffer capacity. The destination sends permits (acknowledgments) of the packets received. Upon receiving the permit from the destination the sender continues by sending another batch of packets. This method is efficient, and can approach optimal performance of a given system [8]. A disadvantage of this scheme is that of choosing a window size. The choice of a window size is a trade-off: small window sizes limit the congestion and tend to avoid large delays, and large window sizes allow full-speed transmission and maximum throughput under lightly loaded conditions. One solution to the dynamic window adjustment was suggested by the congestion avoidance and control mechanisms in 4.3 BSD Reno TCP [7].

4 Implementation of the Adaptive Prototype

Our design addresses network buffer saturation, avoiding time-outs and adjusting to variable network capacity. We chose an *ARQ Go Back N* protocol [1] as our main mechanism. To handle the avoidance of time-outs and variable network capacity, we have added congestion avoidance similar to that used for TCP [7].

The Swift/RAID architecture implementation is modular, with all of the communications code placed in one highly cohesive module, the transaction driver module. All modifications to accomplish the flow-control/congestion avoidance were applied to this module: all versions of the prototype remained operational. The original implementation of the RAID-0, RAID-4 and RAID-5 systems were used, the only difference being increased throughput.

The transaction driver implementation uses two main data structures to control its operations. These contain the set of instructions for the current plan being executed. The client and each server keep them until the successful completion of each plan. They also contain information about the individual communicating entities on both sides: the client has one structure for each server it is using, and each server has a structure for each client it is serving.

The data structures were modified to store window and congestion information for each communications link. Counters in each structure reflect the last packet each server has acknowledged, and the last packet acknowledged by the client to the particular server. At any time if the difference between the counters is greater than the allowable window size, further transmissions are delayed until additional acknowledgments are received that reflect available buffer space at the receiver.

A few simple lines of code were added to the instruction transmission routine to do the difference calculation and comparison with the available window size. The code transmits a small burst of packets equal to the available window size. The same code was added to a similar section of the instruction execution routine and is executed after the system receives a new packet.

A fast retransmit mechanism was added through a separate routine in the transaction driver module. We use a similar technique to that found in TCP for approximating the variance and determining the new *round trip time* value. The calculated round trip time is used by the respective sender in a session waiting on an acknowledgment from its peer for the last packet sent. If the sender does not receive an acknowledgment within the round trip time, packets are resent from the last acknowledged packet through the current available window.

We deviate significantly from the actual implementation of the algorithms in TCP. First, we do not allow our congestion avoidance algorithm to probe past the known window size. Our Swift/RAID system has one link between the transmitter and destination nodes (and there are no intermediate links or gateways to buffer packets while in transit). Probing past the window size would only cause packets to be lost and throughput to drop accordingly. Second, the Reno TCP implementation waits for either a time-out or three duplicate acknowledgments from the destination to react to the apparently lost packet and trigger the retransmission of the packet. We use the *restart* mechanism in the transaction driver to send a *restart* packet for time-outs or receipt of an out-of-order (dropped) packet condition, and we trigger retransmission on the first receipt of this packet. Finally, in calculating the round trip times we use a much finer granularity of timer than the actual TCP implementation. The 4.3 BSD Reno TCP uses a coarse grained timer of around 500ms. Our implementation reads the system clock as each packet is sent from the transaction driver, and again when each acknowledgment is received, using the difference as our measured round trip time. We then use a round trip time estimator to compute our retransmit timer. This gives us a more accurate time-out calculation for retransmissions.

5 Results

Experiments were performed for all available versions of the system, including RAID-0, RAID-4 and RAID-5. Read and write throughput was measured for file transfers up to one megabyte and compared with the original non-redundant prototype as well as the Swift/RAID prototype. The throughput measurements performed to evaluate the prototype were essentially the same as those reported in [9]. In fact, the identical test programs and RAID-4 and RAID-5 modules were used.

5.1 Methodology

The block size was changed from 8192 bytes (in the Swift/RAID prototype) to 7340 bytes to avoid fragmentation of the Ethernet packets. Because we were interested in network performance, the files were preallocated on the servers so that file creation was not reflected in the results. The individual experiments were repeated 50 times each and averaged to obtain the results reported here.

The Swift/RAID architecture uses a 60 byte header on the datagram in addition to the data block being transferred. Therefore a data block size of 7340 bytes was used, which when added to the 60 byte header gives a 7400 byte datagram that fragments into exactly five 1480 byte Ethernet packets with no internal fragmentation. This maximizes use of the network resource and increases throughput slightly over the original 8192 byte block size used in previous experiments.

To establish credibility of the data and the data gathering techniques a typical experimental run was analyzed to determine the standard deviation and confidence intervals. The experiment was run for each block size using the Swift/RAID adaptive prototype running on a three node RAID-5 system. The 90% confidence intervals ranged from $\pm 0.4\%$ to $\pm 8.11\%$, with a mean interval of $\pm 1.91\%$ for reads and $\pm 3.17\%$ for writes. All of the numbers for the adaptive Swift/RAID prototype in this report are mean values with similar confidence intervals. Figure 1 shows the throughput for both reads and writes with error bars for the 90% confidence intervals.

The testing platform was a heterogeneous local area network consisting of Sun SparcStations, including a SparcStation 2, a SparcStation IPX, a SparcStation IPC and three SparcStation SLCs. The interconnection medium was a 10 Mb/s Ethernet. The SparcStation 2 was used as the client in all of the measurements that follow. The balance of the machines were used as the Swift/RAID servers, with the SparcStation IPX and SparcStation IPC always included, and one or more of the SparcStation SLCs added in as necessary for the experiment. The client machine was also the NFS file server and gateway for the subnet described. This accentuated the inabilities of the workstation to handle the loads presented, but the workstation was used for historical reasons (comparison with previous results). In addition, the network was an active network heavily used by researchers and with a fluctuating load. These activities and fluctua-

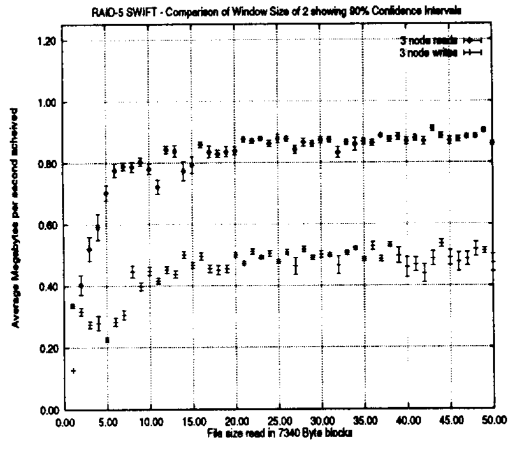

Figure 1: Swift/RAID-5 prototype performance with 90% confidence intervals shown.

tions are reflected in the data collected and appear in the graphs as small bumps and occasionally more severe dips. To capture this activity and loading in a more direct manner we used a Network General Distributed Sniffer System to monitor network utilization.

5.2 Performance Evaluation

The throughput obtained for reads for the RAID-0 prototype using the *Stop and Wait* protocol ranged from 651 to 758 KB/s. Results for the *ARQ Go Back N* protocol with a window size of one are very similar to those of the *Stop and Wait*, as expected (Stop and Wait is the same as ARQ Go Back 1). When the window is increased to two, reads improve slightly to 689 – 830 KB/s. As the window size is increased up to five, the read operations level off to 910 KB/s. Results for the RAID-4 and RAID-5 prototypes are similar and push the throughput on the network to 927 KB/s and 896 KB/s, respectively. (Figures 2, 3 and 4 show read and write throughput of a typical file transfer (40 blocks – 293.6 KB) for window sizes from one to five.)

For the RAID-0 prototype it can be seen that the addition of the adaptive *ARQ Go Back N* control improved read operations by as much as 40% for a two node system. The improvement is less dramatic for larger configurations. However, it should be noted that in all cases the read operation is brought up to about 900 KB/s, which is very close to the usable bandwidth of the network [2], making further improvements difficult. RAID-4 read operations show an improvement of 35% over the *Stop and Wait* protocol (from 671 KB/s to 927 KB/s) and RAID-5 improved up to 23% (from 729 KB/s to 896 KB/s).

Write operations for the RAID-0 *Stop and Wait* prototype ranged from 616 to 766 KB/s. For our new prototype writes achieved throughputs from 768 to 800 KB/s. Write operations continue at about 800 KB/s, except for the case of four nodes. Here the client is sending based on a window size of at least three Swift data unit packets to four servers at once. One Swift data unit is 7400 bytes with header, which is five Ethernet packets on our network configuration. This, times three for the window, times four for the servers equals 60 Ethernet packets. Write throughput starts to drop off as the number of blocks transmitted by the client reaches, or exceeds, the capacity of the kernel buffers. The Sun implementation of UDP randomly discards packets from its kernel network buffers when it receives more packets than it can handle, and our kernel network buffers were set at 50 packets (the original configuration from Sun). When the client sends the 60 packets to be transmitted, the kernel is throwing some of them away causing the throughput to drop accordingly. These can be seen in Figure 2.

RAID-4 write operations did not improve over the *Stop and Wait* prototype except for the case where the window size is equal to two. In this case the adaptive prototype maintains a throughput of 482 KB/s compared to the 400 KB/s attained by the *Stop and Wait* prototype – an improvement of 20%. Write operations for the RAID-4 redundant systems suffer from some inherent problems. Parity block traffic, the extra traffic for small writes and the CPU overhead used for parity calculations all adversely impact the RAID-4 system. Parity block traffic is not serious for systems with many nodes to stripe across, but for systems with small numbers of nodes (like ours) it is a major contributing factor in limiting throughput. The impact of the small write traffic decreases as the number of blocks requested increases, but is a major contributor to poor performance for small block requests. Parity calculations have been shown to be a factor in limiting the ability of the workstations used to achieve better performance. Experiments have found the cost of parity to be 200KB/s in the Swift/RAID *Stop and Wait* prototype using a SparcStation 2 [9]. Additionally, the bottleneck created by the use of a single parity node becomes as issue for large writes, and tends to keep RAID-4 performance below others, such as RAID-5. Also, as node and window sizes increase, we run into the same buffer limitations as we did with RAID-0.

Results for RAID-5 operation show up to a 49% increase in write operation throughput (from 375 KB/s to 559 KB/s). RAID-5 write operations do best at the window sizes of two and three, and diminish above that, especially for the larger server populations. (Again, the Sun operating system kernel buffers are being exhausted and packets are being randomly dropped from the queue, causing retransmissions and lower throughput.)

RAID-5 writes don't do as well in all cases, but in general do as well as the *Stop and Wait* prototype, or better. As for RAID-4, RAID-5 has some similar inherent problems. The parity block traffic, small write traffic and CPU costs for the parity calculations all take their tolls on RAID-5 systems. The parity block traffic is the over-riding factor in limiting throughput for small numbers of nodes.

Results from the use of the network analyzer were interesting in that they show both read and write operations utilize the network close to maximum capacity. Why then do writes seem to perform so poorly in our experiments? The low performance of the writes is directly attributable to the

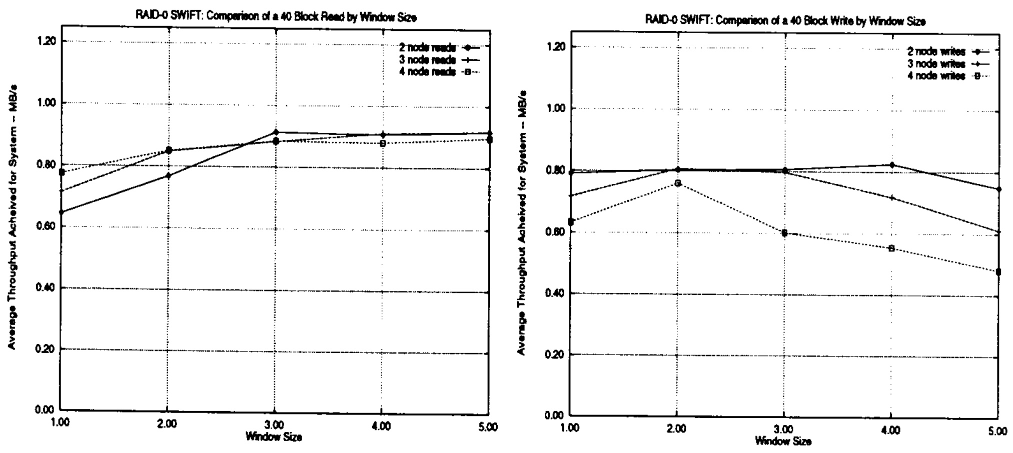

Figure 2: RAID-0 adaptive prototype performance.

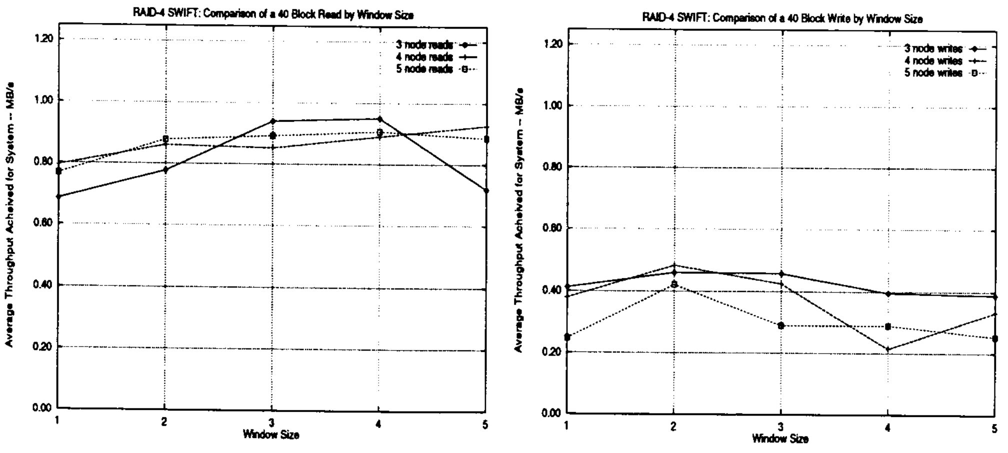

Figure 3: RAID-4 adaptive prototype performance.

cost of redundancy – the parity block that must be updated for each write operation. For a full stripe write $n - 1$ blocks are written to data nodes, and one parity block is written. The penalty here is that $1/n$ of the network traffic is being spent to preserve redundancy. When n is large this cost is small. However in our prototype system n is small (*i.e.*, 3,4 and 5 nodes were used) and therefore the cost is high. In addition, small writes impact writes that are not full stripes, and cost at least an extra two blocks (reading the old data block, and the old parity block) of traffic plus the rewriting of the parity block. For a three node system the stripe is two blocks in size, and small writes impact 50% of the block sizes used in our experiments. Similarly, in a four node system small writes impact 33% of the block sizes. As the number of blocks written becomes large the system writes to mostly full stripes, with one small write at the end if the total size does not fall on a full stripe boundary. This lessens the overall affect of the small writes so the throughput of the system is controlled by the costs of the single parity block traffic.

5.3 Degraded Mode Performance Evaluation: RAID-4 and RAID-5

For these experiments the Swift/RAID adaptive prototype was operated with a window size of two. An operating node was terminated before each experiment was started. The RAID-4 system was tested both with the terminated node as the parity node or as one of the data nodes. RAID-5 was tested with a random node failure. Figure 5 shows the results on both RAID-4 and RAID-5 for our prototype in degraded mode operation.

In degraded mode operation, RAID-4 and RAID-5 systems must reconstruct data on the failed node from the remaining nodes in the system. RAID-4 has two cases to consider: the case of a failed data node, and the case of the parity node. It can be seen that with a failed data node the read operation throughput drops from 927 KB/s to 760 KB/s, or 18%. For write operations the throughput increases by 26% from 482 KB/s to 608 KB/s. For the case where a RAID-4 parity node fails the read operations are unaffected, while write operations improve the same percentage over normal operation as for a data node failure. The improvement in the write performance is due to a significant decrease in

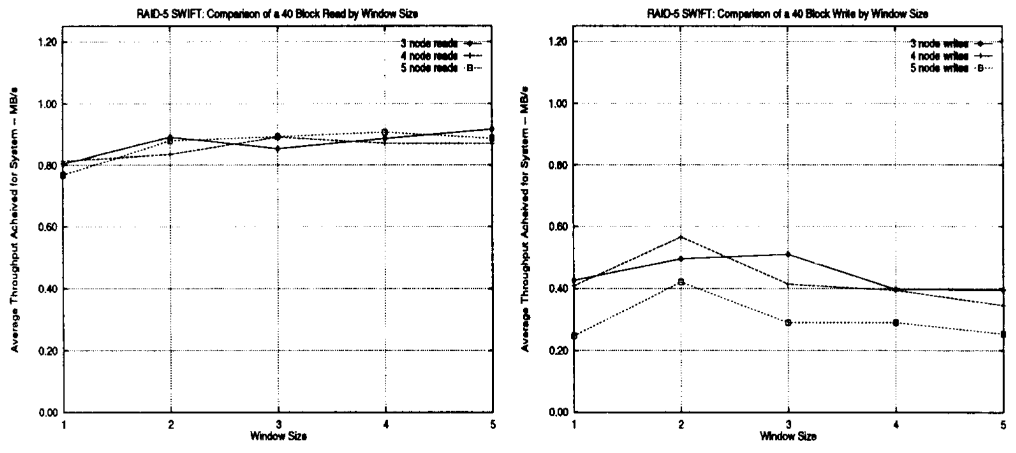

Figure 4: RAID-5 adaptive prototype performance.

network traffic due to lost redundancy. When a data node has failed, only the parity block is written; the write to the failed node is deferred until the failed node is reconstructed. When the parity node has failed, only the data blocks are written, and parity calculation and write are deferred until reconstruction.

For RAID-5 systems there is only one degraded mode since the parity information is distributed evenly across all nodes. Read operations show a decrease in throughput from 896 KB/s to 768 KB/s (a loss of 17%). Write operations improved from 559 KB/s to 714 KB/s, an increase of 27%. Writes improve for the same reasons as in RAID-4, and since RAID-5 nodes contain both data and parity information, both causes apply to a failed RAID-5 node.

Our results agree with other research that has shown the throughput for writes in systems of fewer than four nodes actually increase during degraded mode for both RAID-4 and RAID-5 systems [10]. These results show a decrease in total load due to writes for RAID-5 systems where the number of disks is less than eight, and average load decrease for fewer than four disks. It can similarly be shown for RAID-4 systems with a decrease in total load for less than eight disks, and a decrease in average load for less than five disks in the system.

6 Conclusions

An adaptive Flow Control Mechanism has been added to the prototype for the Swift/RAID distributed file system prototype. This mechanism has allowed the prototype to achieve a greater than 25% increase for read operation throughput, and up to a 50% increase for write operations over the previous Swift/RAID prototype. The adaptive RAID-4 and RAID-5 prototypes are both able to achieve read throughputs in excess of 900 KB/s.

Bertsekas and Gallager [1] have discussed that one of the limitations of end-to-end window flow control is the trade-off of choosing a window size – small window sizes keep packets in the subnet low and congestion to a minimum, but large windows allow higher rates of transmission and maximum throughput during light traffic conditions. They have also suggested that the value should be between n and $3n$, where n is the path length between the nodes. Our results agree and have shown that a window size of two (on our local network with a path length of one) provided the maximal throughput for write operations – on the order of a 50% increase over the *Stop and Wait* protocol. Other window sizes of 1, 3, 4, 5 showed little, or no improvement for writes and in the case for a window size of five the results tended to be below all others because of the swamping of the client Ethernet queue. This is also supported by Eldridge [5]. Read operations did better with higher window sizes, but not significantly better (approximately 5% in most cases).

Most of our improvement in throughput was gained by the simple "self-clocking" [7] of the data packets with the acknowledgments sent by the destination as packets were received. This is because once the packet traffic has stabilized, the acknowledgment packets are being returned at the rate at which the receiver is pulling packets off of the network. Likewise, the sender is receiving the acknowledgments at the same rate and putting new packets into the network. This has the affect that as the receiver is taking one packet off, the sender is simultaneously putting another one into the network.

One way to improve small writes is through a server-server protocol. It can be shown that a small write can be accomplished with one data packet sent from the client to the server node for storage, and one interim parity packet sent directly to the parity node from the server receiving the new data packet. To take advantage of this technique a protocol for server to server communication needs to be designed and implemented. This would allow for the small write problem to be reduced from four network instruction transfers, all of which involve the client node, to two disk operation instruction packets sent over the network, only one of which would involve the client.

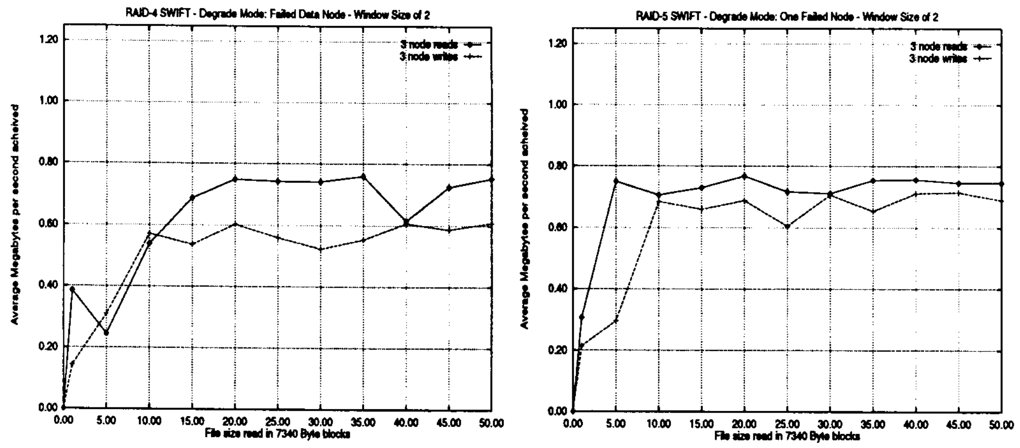

Figure 5: Comparison of Swift/RAID prototype read and write performance in degraded mode.

References

[1] D. Bertsekas and R. Gallager, *Data Networks, 2nd Edition.* Prentice-Hall,Inc., 1992.

[2] D. R. Boggs, J. C. Mogul, and C. A. Kent, "Measured capacity of an ethernet: Myths and reality," in *Proceedings of SIGCOMM 88*, pp. 222–234, ACM, 1988.

[3] L.-F. Cabrera and D. D. E. Long, "Swift: a storage architecture for large objects," in *Digest of papers, 11th IEEE Symposium on Mass Storage Systems*, pp. 123–8, IEEE, 1991.

[4] L.-F. Cabrera and D. D. E. Long, "Swift: Using distributed disk striping to provide high I/O data rates," *Computing Systems*, vol. 4, no. 4, pp. 405–36, 1991.

[5] C. A. Eldridge, "Rate controls in standard transport layer protocols," *ACM Computer Communication Review*, vol. 22, no. 3, 1992.

[6] M. Gerla and L. Kleinrock, "Flow control: A comparative survey," *IEEE Transactions on Communications*, vol. COM-28, no. 4, pp. 553–574, 1980.

[7] V. Jacobson, "Congestion avoidance and control," in *Proceedings of SIGCOMM 88*, ACM, 1988.

[8] D. D. Kouvatsos and A. T. Othman, "Optimal flow control of end-to-end packet-switched network with random routing," *IEEE Proceedings*, vol. 136 Pt E, no. 2, pp. 90–100, 1989.

[9] D. D. E. Long, B. R. Montague, and L.-F. Cabrera, "Swift/RAID: A distributed RAID system," *Computing Systems*, vol. 7, no. 3, pp. 333–59, 1994.

[10] S. W. Ng and R. L. Mattson, "Maintaining good performance in disk arrays during failure via uniform parity group distribution," in *Proceedings of the 5th International Symposium on High-Performance Distributed Computing*, pp. 260–69, IEEE Computer Society Press, 1992.