Exploiting Highly Reliable Networks with Careful Protocols

David Mosberger, Charles J. Turner, and Larry L. Peterson [1]
{davidm,cjt,llp}@cs.arizona.edu

TR 94-14

Abstract

Optical interconnects often exhibit excellent reliability. This paper studies the potential for exploiting this reliability via network protocols that no longer assume unreliable links. Such protocols are termed "careful protocols" as they have to be careful to preserve the reliability of the link to the application level. After giving a definition of careful protocols and discussing related work, the paper presents in detail a case study of a networking service that has been implemented, both as a traditional and as a careful protocol. The case study suggests that careful protocols can achieve considerably better performance than traditional ones but also shows that this is not trivial to achieve.

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by ARPA Contract DABT63-91-C-0030 and by Hewlett Packard Company.

1 Introduction

Communication protocols for workstations have traditionally been designed based on the assumption that the physical link is unreliable. That is, a packet sent over the network could be duplicated, lost, damaged, or arbitrarily delayed. Traditional protocols often take advantage of this assumption by dropping packets whenever they encounter resource shortages. In other words, unreliability is a self-fulfilling prophecy.

However, many current networking technologies provide service with outstanding reliability. For example, on token-ring based networks, there are no out-of-order packets, as long as the ring operates correctly. With the advent of optical links, bit-error rates became so small that for most practical purposes they can be ignored.

Protocol designers sometimes exploit this reliability by implementing so-called "optimistic" protocols. While they make optimistic assumptions about the reliability of the network, they still implement reliability in the case the optimistic assumptions are violated. So far, however, little work has been done to take the next logical step to trust *completely* in the physical link.[1]

This research investigates what the influence of reliable physical links would have on protocol design. It is not clear at all whether specialized protocols could achieve considerably better networking performance compared to traditional protocols. For brevity, we call protocols that have been designed under the assumption of a reliable physical link *careful* protocols.

In the following Section, we give a definition for careful protocols, explain their "typical" operation, and determine potential advantages and shortcomings. Section 3 then presents related work. Section 4 discusses miscellaneous issues that arise due to practical constraints, like the desire for interoperability. To test our ideas, we re-implemented an existing network protocol as a careful protocol and compared its performance with the existing traditional one. Both protocols are described in Section 5 and their performance is reported in 6. Section 7 lists areas that have been left unexplored by this research. Section 8 concludes the paper with some final remarks.

2 Careful Protocols

A *careful protocol* is defined to be a protocol offering a reliable service, provided that the lower-level protocol is reliable. It is careful in the sense that it *preserves* the reliability of the lower-level protocol in the service it provides to the higher-level protocol. A protocol is said to be *reliable* if the probability of lost, damaged, duplicated, or out-of-order delivered packets is negligibly small.

The definition of "negligibly small" is intentionally left unspecified, as it depends on the nature of the application using a careful protocol. For the kind of applications we will discuss later on, a protocol could be considered reliable even if fails on the order of once a week.

2.1 Operation

We illustrate the operation of a typical asynchronous careful protocol by comparing the major protocol processing events with the ones of traditional protocols (notice that this ignores flow-control, which would be present in both types of protocols).

[1] The interconnect is often trusted in an multicomputer, but to the best of our knowledge, not on clusters of workstations connected by local area networks.

Message Send:

1. allocate message buffer
2. fill data into buffer
3. send message
4. start timeout timer

Arrival of ACK:

1. free message buffer

Message Arrival:

1. deal with duplicated, dropped, and out-of-order messages
2. send ACK
3. deliver message to recipient

Message Processed:

1. free message buffer

The "start timeout timer" action sets up a timer such that the message would be retransmitted if an acknowledgement (ACK) has not arrived in due time. Of course, details may vary. For example, some protocols put the burden of requesting retransmissions on the recipient side. However, all of the above steps usually have to be performed in one form or another at some point. In contrast, a typical careful protocol would operate as follows:

Message Send:

1. allocate message buffer
2. fill data into buffer
3. send message
4. free message buffer

Message Arrival:

1. deliver message to recipient

Message Processed:

1. free message buffer

The first obvious difference is that a careful protocol can free a message as soon as it has been sent. If we regard the network adapter as a queueing system, this translates into a small service time for outgoing messages. For traditional protocols, the service time is essentially one round-trip time, while for careful protocols it is determined by the time it takes to send the message. Notice that the the round-trip time has a lower-bound imposed by the speed of light and the physical extent of the link. In contrast, the send time is determined by the link bandwidth alone. Considering the vast bandwidth of optical links, it is therefore perfectly possible that we will see networks where these two times differ by orders of magnitude.

A relatively minor point is that careful protocols do not require any timers. While not tremendously expensive, this does help to simplify the protocol somewhat and streamline protocol processing. More importantly, if there are no timers, then there are no timeouts that need to be tuned. As exemplified by TCP's congestion control, finding good timeout algorithms can be a rather challenging task [13].

Finally, as no messages can be lost, there is no need for acknowledging the arrival of a message with a careful protocol. However, notice that a careful protocol still needs to propagate flow control information back to the sender. Thus, ACKs are essentially replaced by flow control updates in a careful protocol.

2.2 Advantages

The advantage of careful protocols shows if a whole protocol stack solely consists of careful protocols. In such a stack, no retransmission scheme is required. Neither does it have to deal with duplicate or out-of-order packets, thus considerably reducing the number of problems the protocol has to solve.

A careful protocol stack has to be built on top of a physical connection guaranteeing ordered delivery and very low loss-rates. For example, optical rings like FDDI have these properties. Interconnects of multi-computers usually provide similar guarantees. Even if the physical connection is not sufficiently reliable, it might be worthwhile to add a

low-level protocol that ensures it, e.g., by adding error correction. For example, satellite connections often use forward error correction to improve on bit-error rate. That is, guaranteed delivery could be implemented at the very lowest level. As there would be no need for end-to-end retransmission, the advantages would be similar to a network that is physically reliable.

There are two potential advantages of careful protocols: first, the reduced service time for outgoing packets can improve throughput. Second, the algorithmic simplicity of careful protocols might yield lower processing overheads and therefore reduce latency. Observe that these hold even when a traditional protocol is used over a reliable link. This is because traditional protocols unavoidably have to do more work in order to be prepared for retransmissions, even though these may never occur. In short, we would expect careful protocols to show better performance than even so-called optimistic protocols. Naively, one would also expect that the absence of a retransmission scheme in careful protocols leads to more stable throughput and less jitter in transmission latency. However, it is easy to see that while traditional protocols introduce variance mainly via dropped packets, careful protocols do so via flow-control. Variance is simply a manifestation of the fact that a fixed bandwidth link is shared by a potentially unlimited number of users.

Lower service time implies that a sender has to buffer fewer messages. In fact, if a sender blocks until the network adapter has transmitted or copied a message, it does not need to buffer anything. Depending on the network adapter, this can translate in to fewer message copies and therefore higher throughput (see Section 4 for a discussion of the influence of different network adapter designs).

To illustrate the potential advantages of careful protocol stack we give two examples. Suppose a cluster of workstations is connected over a reliable high-speed, low-latency network. In such an environment, protocol and copy overheads are likely to dominate end-to-end latency. The algorithmic simplicity of careful protocols would help reduce protocol overheads. Furthermore, if the data being sent is volatile, a conventional protocol would need to make a backup copy of it, in the case retransmission is needed. This copy can often be avoided in careful protocols, which improves, both latency and throughput.

Conversely, in a high-latency environment, like a hypothetical[2] reliable, high-speed satellite link, the latency incurred by the protocol stack is negligible. However, if we consider a 1 Gbps connection with a 600 ms delay, then we can see that the bandwidth delay product is roughly 72 MB. Thus, a sender would have to buffer at least 144 MB in order to keep the pipe full! With a careful protocol stack, no sender side buffering at all would be needed.

2.3 Difficulties

The difficulty in designing and implementing careful protocols is in assuring that each incoming packet will find the resources it needs. A well-known technique for ensuring this in point-to-point communication is the sliding window. While the technique is well-understood, there is a question of exactly what resource a sliding-window controls. TCP, for example, controls the amount of buffer space expressed in bytes. This choice is not necessarily appropriate or even sufficient for a careful protocol, because the remaining buffer space is just one of many resources an incoming packet usually occupies[3]. Another problem is deadlock avoidance. A careful protocol has to guarantee that any given packet will be transmitted eventually (provided that the lower-level protocol also gives this guarantee).

When attempting to extend this idea to group communication, a multitude of design and implementation choices arises. For example, if host *A* knows that it is going to receive mostly from host *B* then it would make sense for it to advertise a large window to *B* and small windows to all other hosts. Thus, to achieve optimal performance and resource utilization, it is desirable to advertise windows large enough to ensure that no sender ever has to block due to filling up the window, but small enough to keep the number of unused resources low. As communication patterns are likely to change over time, optimal advertised windows will have to do so, too. Depending on the application, it might

[2] Current satellite connections do not appear to provide the required reliability.

[3] In fact, in a predecessors of TCP, both the number of packets and the amount of data was controlled [5]

be possible to *predict* future communication demands. In general, however, estimating communication demands based on past behavior might be the best one can do. It appears that arbitrarily sophisticated schemes could be developed in the search for optimal window sizes. Of course, the complexity of this scheme competes with the goal of keeping the protocol induced latency smaller than that of traditional protocols.

2.4 Limitations

A strength of traditional protocols is that they can achieve receiver resource utilizations arbitrarily close to one. If packets are received faster than they can be processed, the available buffer space will fill up and the receiver will eventually drop incoming packets. This operating point would lead to a low network utilization due to the high number of retransmitted packets. But it does illustrates that any operating point between the two extremes of the receiver always having most of the resources free and the receiver always having most of the resources in use can be reached with traditional protocols.

Careful protocols, on the other hand, always have to stay on the safe side. A single dropped packet is as severe as a host-crash in a traditional environment. In general, this will make it difficult to achieve high resource utilization. It appears, however, that this is a minor point considering how many resources traditional protocols occupy for sender-side buffering.

A final issue is what should happen in the case disaster strikes and the lower-level protocol violates its reliability promises. Clearly, any fault-tolerating technique such as check-pointing or active redundancy could be employed. If the probability of a lost packet is on the order of the probability that a host crashes, this is actually no different than what would be needed for traditional protocols anyway. That is, even though traditional protocols deal with retransmissions, they usually do not handle host-crashes transparently. So, it is sufficient to have loss probabilities that are small enough to make fault-tolerance techniques reasonably cheap. In fact, we would expect that for many realistic applications, a fault-rate on the order of one per week would be acceptable. While careful protocols attempt to provide a service as reliable as possible, the ultimate guarantee for proper operation should be implemented at the end-to-end level.

3 Related Work

As mentioned before, we are unaware of any work in the networking field that would have trusted in a physical link completely. For multicomputers, however, quite the opposite is true: just like the bus in a multiprocessor system, the interconnect in a multicomputer is usually assumed to work reliably. The system software of early multicomputers provided communication with either no or very simple flow-control only. For example, Sylvan [4] provided rendez-vous communication, where both, the send and receive operations are blocking. The Reactive Kernel [16, 17, 18] provided an interface in which the receive operation returns the first queued or incoming message. There is essentially no buffering and flow-control had to be implemented in higher layers. It is difficult to determine what protocols have been used in higher layers of the various multicomputers, but it appears that the rendez-vous has been one of the more attractive styles of interaction. Hoare's Communicating Sequential Processes (CSP), Occam, and Ada use it as their communication model (see [1], for example). It is easy to see that such protocols are trivial in the presence of reliable physical links: per communication channel, there is at most one outstanding message. Flow-control, therefore, is not an issue. Notice that even though flow-control is trivial, congestion-control still might be an issue. For example, in an Ethernet-style network, if too many stations attempt to send messages at the same time, a packet may be lost even though each individual communication channel never had more than one outstanding message. However, in the networks under consideration (multicomputer interconnects and token-rings), congestion is generally not an issue.

To improve link utilization, more recent machines use communication protocols that are asynchronous and very

similar in nature to careful protocols. An example is the message passing interface of the node operating system NX/2 on Intel iPSC hypercubes [15, 3][4]. With asynchronous protocols, the number of outstanding messages is limited by the number of available resources only. Thus, flow-control becomes an important issue. For small messages of up to a hundred bytes, the NX/2 protocol works very much like our implementation of a careful protocol, while for large messages it operates more like a request/response protocol: first, a small message is sent requesting to allocate a suitably large buffer and once the recipient has acknowledged this, the actual (large) message is transferred. It is probably fair to say that the protocol used in NX/2 was the first careful protocol, according to our definition.

In summary, one way to look at careful protocols is that the advent of highly reliable physical links has blurred the distinction between multicomputers and clusters of workstations. As a consequence, the design space for workstation communication protocols has been extended by one dimension.

This work also appears to be the first in making an attempt to quantify the potential performance benefit of careful protocols versus traditional ones. This is not surprising. So far, there was not really a choice: communication outside of a multicomputer was unreliable and therefore *had* to employ traditional protocols. Similarly, the interconnect of a multicomputer was reliable by *design*, so there is no point in considering to employ a traditional protocol.

Active messages [19] appear to be similar to careful protocols. Unfortunately, it is unclear whether active messages should be regarded as an end-to-end protocol or as an implementation mechanism. In [19], Eicken et al. seem to imply the former. In this case, active messages are simply an instance of a careful protocol. Active messages are even simpler than the careful protocol we present in this paper because the communication model does not require any flow-control. If active messages are regarded as an implementation mechanism which is used to build more elaborate protocols, then they can by employed as a foundation for both careful and traditional protocols. Especially in the form of optimistic active message [11] they could help eliminate driver-level queueing overheads and a context switch per incoming message.

Advanced cross-domain data transfer facilities like Fbufs [8] and modern application program interfaces (APIs) achieve high throughputs by reducing the number of physical copies. This is similar to what careful protocols achieve. However, it is important to notice that the former are techniques that do not work in all cases. In particular, in parallel processing, it is often necessary to send the values of variables that will be modified later on. In such cases, these techniques still would require a physical copy while a careful protocol could transmit directly out of the application's data. Also, notice that even though physical copies might be eliminated, there is no reduction in the amount of sender side buffering.

4 Miscellaneous Issues

This Section discusses miscellaneous issues that influence the applicability of careful protocols.

4.1 Co-existence With Traditional Protocols

A system that uses careful protocols is likely to employ traditional protocols for interoperability reasons as well. Thus, there is a need to have careful and traditional protocols co-exist without interference. The only condition for interference free co-existence is that the arrival of a frame sent by a traditional protocol must never cause a frame from a careful protocol to be dropped. In our system, this is achieved by simply having two different network controllers. The Ethernet controller is used for traditional protocols like TCP/IP and the FDDI controller is used exclusively by the careful protocol. However, with a minimal amount of hardware, the two types of protocols can also be made to co-exist on the same network adaptor. All that is needed is a bit in the frame header telling whether the frame

[4]The protocol used on the Intel Paragon multicomputer is proprietary to Intel and we have no information regarding its design other than that it appears to be an extension of the protocol used in the iPSC's NX/2 operating system.

belongs to a traditional or a careful protocol. Based on this header bit, the network adaptor can then decide whether to allocate a frame buffer from the traditional or the careful protocol's buffer pool. Notice that virtual circuit identifiers (VCIs) and per-VCI frame buffer pools provide the same capability at an even finer granularity (per-session instead of per-protocol). Using VCIs is beneficial independent of careful protocols [7] and it is likely that future network adaptors will them.

4.2 Ideal Network-Adaptor Interface

Careful protocols can transmit data directly out of application level data-structures. Ideally, a network adaptor for careful protocols should therefore allow to transmit data via direct memory access (DMA) that supports a scattering/gathering of data. If the DMA facility is at least as fast as copying a frame and the network provides a sufficiently high bandwidth, all need for copying is eliminated. Of course, scatter/gather DMA operations generally have higher overheads than simple continuous transfers. However, as long as these overheads are lower than the time it takes to linearize a message, scatter/gather operations are beneficial.

5 Evaluation: Networking Protocols for C* Clusters

We implemented an asynchronous careful protocol as a proof of concept for a C* cluster. C* is an extension of C that adds support for data–parallel execution [9]. The cluster consists of a number of workstations connected over an FDDI token ring. The workstations (nodes) in the cluster work on the same application and use a networking protocol to communicate with each other. The networking protocol is optimized towards the needs of the C* run-time system. This Section discusses the design and implementation of these protocols. Section 6 gives details on the cluster configuration and on the performance of these protocols.

5.1 Protocol Interface

Each of the P nodes in a C* cluster runs a copy of the C* run-time system (RTS). The individual nodes communicate via the network layer in terms of *events* and *messages*. Events are totally ordered and subsume the sequence numbers found in ordinary reliable stream protocols. A node may or may not send a message for any given event and it can either send it to an individual destination or to all other cluster nodes (multicasting). Every node receives at least one message per event. If an event is a *crowd* (all-to-all) event, then all nodes receive $P-1$ messages, that is, one message from every other node. While the interface logically exchanges messages, the actual communication occurs in terms of *fragments*. A fragment has a fixed maximum size that depends on the network adapter. As messages may be of arbitrary size, the RTS potentially has to fragment and reassemble them into the appropriate number of fragments. This is discussed in more detail below.

The service provided by the network layer provides an asynchronous service that reliably delivers fragments. It is asynchronous in the sense that the send operation is non-blocking (pending resource shortages). Although the service mainly consists of a send and a receiver operation it is important to note that this service is not like more traditional networking services. First of all, while it delivers fragments reliably (every fragment destined to a particular node is delivered exactly once), it doesn't give any guarantees on order. C* communication is usually such that the order in which the fragments of a large message arrive does not matter. Any fragment can be processed as soon as it arrives. Consequently, while the network layer takes care of delivering each fragment exactly once, it does not do the actual fragmentation or reassembly. A second specialty is that network connections to the individual nodes are not treated independently. Assuming P nodes in a cluster, instead of having $P-1$ independent connections from each node, there is only one channel over which the RTS can communicate with any of the participating nodes. This reduces the total

amount of state needed for the network connections considerably. Of course, as the amount of state decreases, the time spent maintaining this state decreases, too. Third, the network layer provides a multicast service. In the current implementations, messages can be sent to either an individual node or to all other cluster nodes. Certain optimizations would be possible if it were possible to communicate with *any* subset of the participating nodes. However, it is not clear whether this would warrant the additional complexity in the network layer and we have not implemented this as of now.

The following is a functional discussion of the interface. While it fully describes the information flow between the RTS and the network layer, the details are different in the actual implementation. The send operation has the following interface:

send(*dst*, *event*, *frag*, *fraglen*, *flags*);
Node_Id *dst*;
Event_Id *event*;
Pointer *frag*;
int *fraglen*;
FlagSet *flags*;

The first parameter, *dst* is the id of the destination node or the broadcast address if the fragment should be sent to all other nodes. The *event* parameter specifies the event to which the fragment belongs and *frag*, together with *fraglen*, specifies the block of memory that constitutes the fragment. Finally, *flags* is a set consisting of the members Last_Frag, BCAST_Event, and Large_Event. The first indicates that *frag* is the last fragment from this node for the given event. Implicitly, it defines the length of the message that is formed by the sum of all fragments. BCAST_Event indicates that node *dst* will receive fragments not only from this node, but also from all other nodes (except *dst*). The final member, Large_Event is set when the fragment is part of a multi-fragment event. In the traditional protocol this is used as a hint for resource allocation decisions.

The receiver side of the interface is complementary:

(*rsrc*, *status*, *frag*, *fraglen*) = **receive**(*src*, *event*);
Node_Id *src*, *rsrc*;
Event_Id *event*;
StatSet *status*;
Pointer *frag*;
int *fraglen*;

The input parameters are the node and the id of the event from which a fragment should be received. However, unlike in most other network layers, it is possible to specify the broadcast address as source node. In this case, a fragment from any node will be returned. Return value *rsrc* indicates which node the fragment actually came from. Return value *status* can take on any of the values MORE, NODE_DONE, or EVENT_DONE. It reports whether there are (or will be) more fragments from node *rsrc* (MORE), whether all fragments from that node have been received (NODE_DONE), or whether there won't be any more fragments for this event, no matter what the source (EVENT_DONE). The fragment itself is returned via pointer *frag* and is of length *fraglen*.

5.2 Traditional Protocol Implementation

Most C* applications cannot be expected to perform reasonably over low-bandwidth or unreliable networks. As a consequence, the traditional protocol employs highly optimistic strategies with regard to failures. While it is capable

of dealing with dropped messages, duplicates, and out-of-order delivered messages, performance would deteriorate quickly and become unacceptable even with moderate loss rates.

The main considerations for the protocol are the acknowledgment/retransmission scheme and flow-control. Flow-control is tricky in general, but thanks to the application-level semantics of C*, it becomes rather simple. The point is that all nodes execute the same program with exactly the same flow of control. Furthermore, most events require message exchanges between all participating nodes. Events are therefore synchronization points. Thanks to this natural synchronism, it is very unlikely that explicit flow-control will ever be needed and is therefore omitted in the current implementation. Even so, the retransmission scheme has been designed to not add strain to a receiver that is already flooded with messages.

The default acknowledgment scheme is simple and cheap in terms of used networking resources. The guiding idea is that it is better to keep a fragment a little bit longer than strictly needed if it can improve the observed network latency. The associated cost is the memory needed for additional buffering. Under the assumption that memory is much cheaper than reducing the latency of the network directly, this is the right solution. While this works fine in the normal case of reasonably sized events, it is occasionally possible that an event carries so much data that the senders would overrun their send buffer space. In those cases, the network layer requests explicit, selective acknowledgments. Notice that this indirectly also works as a flow-control mechanism: if a receiver cannot keep up with the incoming messages, it will drop some of them. For those messages, it won't send an ACK until after it has requested retransmissions and received them a second time. Thus, a receiver can slow down a fast sender by delaying the sending of an ACK.

This protocol puts the burden of requesting retransmissions on the receiver side. As mentioned above, this was partly done for flow-control reasons. What is at least as important is that this also obviates the need for associating timers with messages. The protocol eagerly requests retransmission of a fragment as soon as it detects a hole in the receive sequence. So, under optimistic assumptions, even if a fragment has been lost, the eager retransmission request scheme is likely to ensure that the fragment is present by the time it is actually needed by the application[5]. In the case that even the second transmission of the fragment fails, no retransmission request will be issued until the receiver side of the network layer times out and asks for a retransmission. This is clearly feasible only if optimistic assumption about the network can be made. The receiver implements the time out with a simple busy-waiting loop.

Notice that FDDI neither duplicates nor reorders packets. However, the protocol still has to deal with those failure modes because they follow from building a reliable protocol on top of a network that can drop messages (e.g., due to a checksum failure). Also, as stated earlier, if it is necessary to deal with dropped packets anyway, it is easier to implement a protocol using a best-effort rather than a guaranteed delivery scheme.

Great attention has been paid to efficiently integrate the network interface with the requirements of the run-time system. This vertical integration enables some optimizations that would not have been possible otherwise. For example, there is only one thread of control besides the interrupt handler. This reduces the need for locking and also keeps the number of context switches minimal. Other techniques used to attain an efficient implementation include a buffer management scheme taking advantage of the Medusa on-board memory, the use of lock free data-structures, and a single-level of demultiplexing.

5.3 Careful Protocol Implementation

The careful protocol version was built by adapting the traditional protocol. In essence, the retransmission scheme was removed, which simplified the code considerably. As the traditional protocol had no explicit flow-control scheme, window-based flow-control was added. We selected the most simple scheme we could imagine: each receive buffer is dedicated to a single node (static assignment) and the windows control only one resource, namely the number of frames that can be sent. This implies that all other resources (e.g., data-buffers and list-headers) must be abundant.

[5] Unordered delivery of fragments to the RTS makes this even more effective.

Every incoming frame must able to acquire whatever other resources it needs. This was relatively easy to guarantee as a fixed amount of additional resources are needed per frame. Furthermore, the window-sizes are static. At startup time the protocol determines the number of receive buffers R. Based on the number of nodes P in the cluster, it then advertises a window size of $\lfloor R/P \rfloor$ to all other nodes. Nodes have to be careful in using these windows. In particular, a node should never send data up to the point where its window closes completely. Otherwise, deadlock could occur if the node had to send a window update after that point. This deadlock can be avoided simply by blocking data transmissions if the window has a size of less than two frames. Clearly, R must at least $2P$ for proper operation. Below, a proof of this is given.

Theorem 1 *In a cluster with P nodes, each having $R \geq P$ receive buffers and advertising these buffers such that each of the $P-1$ other nodes has at least a window size of 1, deadlock will be avoided if a frame that will cause a sender's window to close is sent only if it is a window-update.*

Proof By definition, deadlock is avoided as long as at least one of the P nodes can send a frame at any given time. It is, however, easier to prove a stronger condition: we will prove that our protocol guarantees that least one node of any pair of nodes in the system can send a frame. The theorem then holds by implication. Without loss of generality, consider two nodes A and B and assume A's window W_A is at least one frame big. As long as $W_A > 1$ there is no problem. Now suppose $W_A = 1$. A sends a window update only if it can advertise a non-zero window. This will happen either after A finishes processing a (data) frame B sent or after processing a window-update from B. The former depends on the liveness of the consumer (application) on node A, about which we did not make any assumptions. The latter, however, depends on the protocol only and, in ours, liveness is guaranteed (i.e., any window update will be processed eventually). Thus, if A sends a window-update, its window may close ($W_A = 0$). However, B's window W_B will be at least one after it processed A's update, which is guaranteed to happen eventually. As $W_B > 0$, B can now send at least one frame, independent of its previous state. Thus, for any given pair, at least one node can always send a window update and absence of deadlock is guaranteed. ■

Notice that as long as $R \geq 2P$ the protocol will also avoid livelock. Lifelock would occur if the protocol would communicate window-updates only. Of course, this assumes that the consumer (application) on each node will eventually process any incoming frame. It should be clear from this discussion that an application can never have more frames outstanding than there are receive buffers. For example, if a C* application attempts to send more than a maximum window's worth of frames to all other nodes before consuming any if the incoming frames, deadlock would result with this protocol. With the traditional protocol, no deadlock would occur, but performance would degrade due to receivers dropping frames. So while it is *desirable* to avoid too many unprocessed frames in the traditional protocol, it is *necessary* to do so with the careful protocol. The Medusa board provides 128 frame buffers. As the number of nodes in a cluster grows, the maximum window size shrinks. This imposes an upper bound on the number of nodes in a cluster. Even in a non-maximally sized cluster, the windows quickly become small, so that applications with a moderate number of outstanding frames would deadlock already. To ameliorate this, the protocol monitors the number of available receive buffers. If this shrinks below some threshold, the protocol starts to copy out receive buffers into ordinary memory. The freed up receive buffers can then be advertised to the sender, thus avoiding deadlock.

Another issue is how and when to send a window-update. The tradeoff is between minimizing the number of window-updates and avoiding a sender having to block because its window closed. The current implementation chooses to send an update to node A once the number of frames received from A exceeds a certain threshold. This threshold is currently half of the maximum advertised window. Whenever possible, window-updates are piggy-backed on data-frames. With C* applications, there are relatively many broadcast frames. On such frames, a window-update is piggy-backed that advertises the minimum window-update among all other nodes [6].

[6] This was not done in an initial implementation, but turned out to be rather crucial to avoid sending frames containing window-updates only.

6 Experimental Results

This Section reports performance numbers for the careful and traditional protocol developed for our C* clusters. It is important to realize that it is dangerous to extrapolate these numbers and make general statements on the relative performance of traditional and careful protocols. The following simply performance analysis simply presents one data point in the spectrum of protocols.

Our C* cluster consists of eight HP 9000/720 Precision Architecture (PA-RISC) workstations running at 50 MHz. Each machine is equipped with 32 MB RAM, a Medusa FDDI and an Intel Ethernet controller. The Medusa board is an experimental FDDI adapter adhering to the Afterburner specification [6, 2]. The FDDI Medium Access (MAC) standard specifies a maximum frame length (MTU) of 4500 bytes [14]. However, the Medusa board is capable of dealing with frames up to 8 KB. As we were interested in the performance of our protocols and not in the performance of FDDI itself, we did not limit frames to the FDDI standard MTU. The board has one megabyte of on-board RAM which is accessed via programmed I/O (PIO). As the board connects to the high-speed graphics bus, it is also possible to use Venom, an HP proprietary graphics accelerator that can improve data-transfer speed. The Medusa board is capable of sustaining the full 100 Mbps offered by FDDI. All measurements were done with a ring exactly as big as there where nodes in the test (i.e., there are no machines in the ring that do not participate in a test). This is important because the latency to acquire a token increases with the number of stations in the ring. For example, when doing all tests in an eight node ring, the careful protocol is slightly faster, while the traditional one is often substantially slower. We attribute this behavior to changes in the FDDI token rotation timing: depending on whether protocol overheads cause transmission to be in synchrony with token rotation, the time to acquire the token changes dramatically. Measuring performance of n node tests in an n station ring disfavors the careful protocol somewhat and is therefore a conservative choice.

The C* cluster computing environment is based on the Mach micro-kernel. For optimal performance, we extended the Mach kernel with a C* network layer, a C* run-time system (RTS) and C* application programs. We also configured the *x*-kernel[12] into the micro-kernel in order to gain access to the Ethernet controller, which we use optionally for control and logging purposes. That is, a single monolithic kernel containing the Mach micro-kernel, the C* code, and the *x*-kernel has been built. Although our environment is currently based on the Mach micro-kernel, it is used only for booting, memory management, interrupt handling, and the creation of a single thread executing the C* applications. It would therefore be fairly straight-forward to port the environment to another operating system kernel. We wrote a "front-end" program that runs on top of Unix and allows controlling the entire cluster via TCP/IP. The front-end implements communication of the individual cluster members with the outside world (at present, we do not make use of the local disks on the cluster members). In particular, it provides the means to select a particular C* application program and pass "command-line" parameters. It also permits the logging of performance data to the front-end's disk. This has turned out to be extremely valuable as there is a considerable amount of data to be collected and processed.

6.1 Test Programs

We ran a number of tests in an attempt to quantify the relative benefits of each of the two protocols. Notice that there was no bias towards careful protocols. At the time the traditional one was designed and implemented, we did not even contemplate implementing a careful protocol (although there was some discussion as to whether a retransmission scheme will be needed in the given environment). It was only after running tests with the traditional protocol that we realized that the reliability of our FDDI ring is even better than we expected. Based on that, we verified the apparent reliability by running a test, transmitting over 1.57 billion frames at full FDDI speed of 100 Mbps (this is over 10^{14} bits or 12 days of transmission). Not a single frame was lost during this test, convincing us that a careful protocol would be feasible. Notice, that we did not attempt to measure the bit-error rate for FDDI. All we claim is that in our lab environment, we observed extremely low frame-loss rates.

We ran four different tests on two, four, and eight node clusters. In general, each test was run for five times. This results in $5P$ execution times, where P is the number of nodes in the cluster. We report the arithmetic mean of the $5P$ results. In the figures below, we also plot each individual measurement as a small circle. In most cases, observed execution times are together so closely that they are indistinguishable in the figures, indicating a very small deviation from the mean value. The minuscule magnitude of the standard deviation is also the reason why we omit it from the tables. Time was measured via an interval timer that is built into the processor. This timer is clocked at 50 MHz and therefore allows very high resolution measurements.

Latency: The first test stresses latency. It is a straight-forward generalization of the usual "ping-pong" test. The P nodes in the cluster are regarded as being connected in a ring fashion. A single message is sent around this ring. Measuring the time it takes to travel around the ring and normalizing this time to P then gives the time for a single hop (i.e., the latency to transmit a message). The payload in the exchanged messages varied from 1 to 2000 words (with 4 bytes per word). The length of the frame transmitted on the fiber is slightly longer due to protocol headers, which account for 52 additional bytes (including a 4 byte CRC checksum). Table 1 lists latency as a function of the number of nodes P, and the payload size. The results in the "No Processing" column are the latencies when the test program does not access the data. For both protocols this implies that the message data does not get accessed at all (except by the network adaptor, of course). Column "With Processing" reports the corresponding numbers when each word of payload is written once before the message is sent and read once after it has been received. The loops that access the data are not unrolled so that the resulting latencies are somewhat pessimistic. The latencies for the traditional protocol are listed in column "trad" and those for the careful one in column "care." Figure 1 shows the latency in the two node case that does not involve any data processing. Not surprisingly, the latency increases linearly with a slope of 10 ns/bit, which matches exactly the FDDI throughput of 100 Mbps. The figure also shows that the careful protocol is about 27 μs faster than the traditional one. Notice that latency for the traditional protocol increases much more dramatically with the number of nodes than for the careful one. Recall that the traditional protocol does not send explicit ACKs. In the latency test, this has the effect that it has to buffer and therefore copy more often. This is, of course, just due to the current implementation and not intrinsic to traditional protocols. Finally, it is interesting to observe that the one-word message latency for the traditional protocol when going from two to four nodes *decreases* by 2.4 μs. As explained above, this is most likely due to differences in the token rotation timing.

P	Payload [words]	Latency [μs]					
		No Processing			With Processing		
		trad	care	Δ%	trad	care	Δ%
2	1	112.5	84.9	-24.5	113.1	85.6	-24.3
	1000	431.7	404.1	-6.4	1094.6	1067.8	-2.5
	2000	751.6	724.2	-3.6	2081.1	2053.6	-1.3
4	1	110.1	86.2	-21.7	110.9	87.1	-21.5
	1000	549.7	406.2	-26.3	1192.3	1070.3	-10.2
	2000	979.5	726.5	-25.8	2264.3	2058.1	-9.1
8	1	117.8	95.6	-18.9	118.7	96.6	-18.6
	1000	551.9	414.7	-25.0	1196.2	1081.6	-9.6
	2000	981.7	735.6	-25.1	2265.6	2065.7	-8.8

Table 1: Latency Measurements

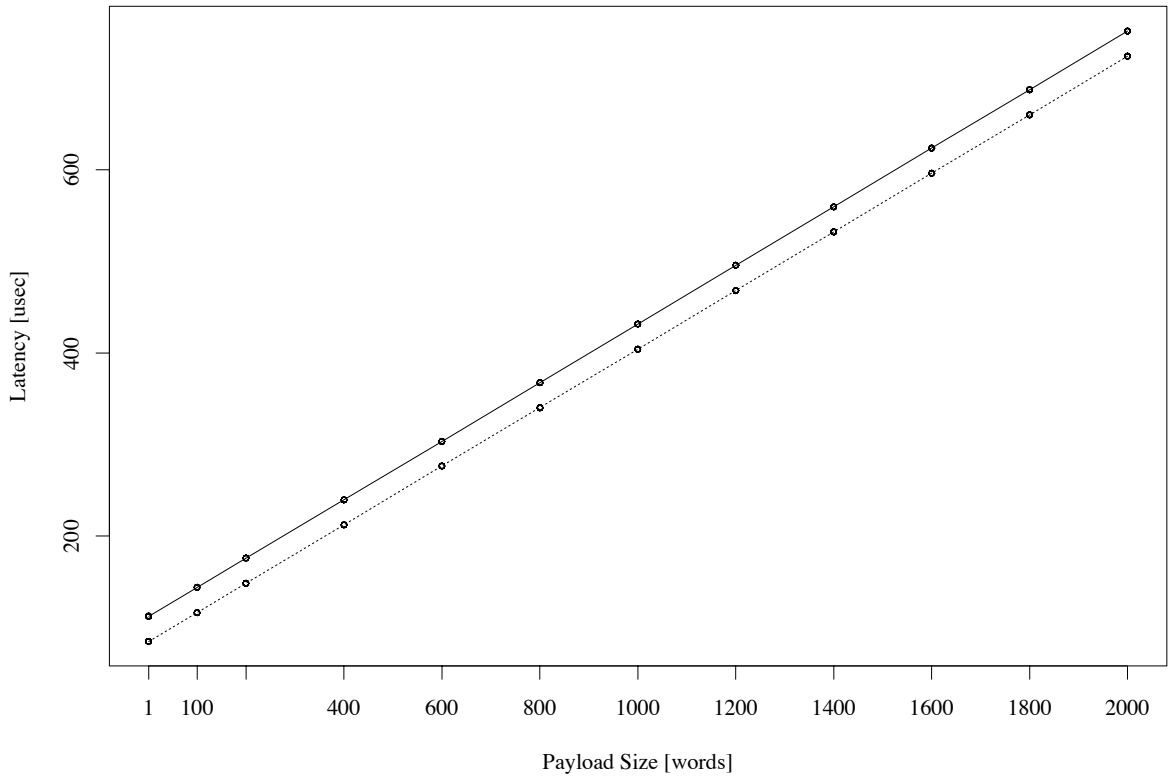


Figure 1: Latency Measurements

In order to get a better understanding of the difference in latency between the two protocols we performed a detailed comparison. In a first step, we listed all operations that one of the protocols has to do but the other can omit (under the assumption of sending a small fragment as in the latency test). We then instrumented the code to measure the time spent for each of these operations. As the interval timer in the PA-RISC CPU appears as a CPU register, the additional code does not distort the original code by much. The measured times were printed on the standard output, which was redirected to a log file. Even though the print statements were not in the critical path, they almost certainly worsen cache performance. However, unless special hardware is available, detailed performance measurements are always intrusive. The instrumented code was compiled with GCC version 2.4.5 and executed on two HP 9000/720 workstations running HP-UX version 8.07. For simplicity of measurement, we replaced the Medusa FDDI adaptor with a simulator that has an interface identical to the adaptor but is implemented via the Berkeley socket library. Overall, we found that the traditional protocol spends 28,36 μs (1418 cycles) on operations that can be omitted in the careful one. Conversely, the careful protocol spends 3.02 μs (151 cycles) on operations the traditional protocol can avoid. The difference of 25.34 μs matches well the 27 μs observed in the real system. Appendix A gives a detailed description of the results we obtained.

Throughput: The second test measures throughput. This is done by sending messages from node 1 to the last node $P-1$ while going through intermediate nodes 2 to $P-2$. Due to idiosyncrasies in the network layer, it is still

necessary to send a small message from node P to node 1. To avoid this extra message limiting performance of the test, its reception is delayed. That is, node 1 does not attempt to consume this message before it is very likely to have arrived already. Therefore, it is unlikely that the node has to block for this message and overheads are kept minimal. Throughput was calculated by dividing the total amount of payload sent by each node by the execution time of the test. This was then normalized by multiplying by the number of senders in the test (e.g., in an eight node throughput test, there are seven senders). Thus, the numbers reported are the total throughput as seen by the ring. Ideally they approach the FDDI bandwidth of 100 Mbps. Table 2 presents the results. Notice that in the absence of data processing, throughput gets very close to the FDDI limit. With 2000 word (8000 byte) messages, this is achieved to within 3.4 %, while with 1000 word it is within 5.2 %. Thus, the throughput loss incurred by restricting MTU to the standard FDDI value of 4500 bytes does not appear to be excessive. Just like for latency, processing the payload reduces performance considerably. With two nodes, throughput with processing is always less than half of that obtained when the data is not accessed at all. However, even with processing overheads, four nodes are already able to saturate the FDDI link. Figure 2 shows the throughput for the two node case. The careful protocol achieves at least a 10 % better throughput for messages with a payload up to 200 words. However, with larger message sizes this advantage quickly tails off as the network becomes saturated.

P	Pay-load [words]	Throughput [Mbps]					
		No Processing			With Processing		
		trad	care	Δ%	trad	care	Δ%
2	1	0.26	0.33	+27.28	0.26	0.32	+24.04
	200	49.8	65.4	+31.15	24.8	25.1	+1.04
	1000	95.3	95.5	+0.19	40.0	40.6	+1.66
	2000	97.6	97.7	+0.10	43.6	44.0	+0.98
4	1	0.53	0.68	+29.35	0.53	0.67	+26.02
	200	70.6	85.6	+21.33	50.4	62.6	+24.10
	1000	94.8	96.9	+2.26	85.5	91.4	+7.01
	2000	97.5	98.4	+0.97	95.1	95.1	+0.02
8	1	0.67	0.78	+17.34	0.67	0.77	+15.64
	200	76.7	84.4	+10.11	59.3	68.2	+15.04
	1000	96.1	96.6	+0.53	92.9	95.2	+2.44
	2000	98.2	98.3	+0.12	96.5	97.3	+0.84

Table 2: Throughput Measurements

It should be observed that depending on the network adapter interface, throughput with a careful protocol could be much better than with a traditional one due to reduced copying. Suppose that a high-bandwidth network adaptor uses DMA to transfer data between the host memory and the network. In this case, the careful protocol can transmit directly out of the application's data buffer obviating the need for a sender side copy. To simulate such a network adaptor, we measured throughput while pretending that there is only one on-board buffer. Thus, the careful protocol has to copy an out-going message exactly once, while the traditional one has to keep a backup copy in the case retransmission is needed. With two nodes and 2000 word large messages, latency for the traditional protocol was measured as 1.15 ms while the careful one had a latency of 0.72 ms. This corresponds to a 37 % decrease in latency, or, equivalently, a 59 % increase in throughput.

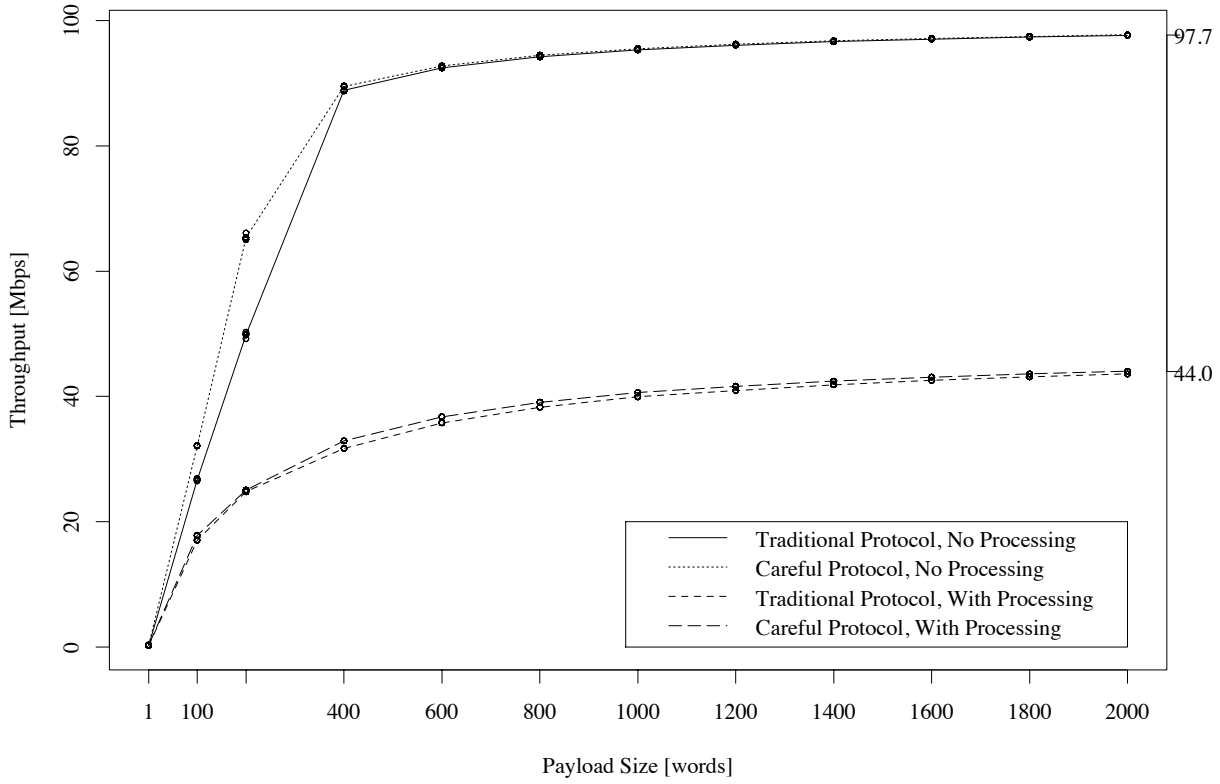


Figure 2: Throughput Measurements

Flooding: The third test stresses receivers by flooding them with messages. It broadcasts messages consisting of k fragments. Each node starts consuming messages after sending n messages (events). Thus, the product nk determines the maximum number of outstanding fragments. If nk is too big, the traditional protocol may get into a livelock, while the careful one will deadlock. It appears that while performing this test, the careful protocol encountered the rare event of a packet loss. This is why the number in the eight node, 2000 word case with processing for the careful protocol is missing. The traditional protocol has no explicit flow-control and it therefore performed very poorly on this test. With four or more nodes, it would almost immediately get into a livelock—forever retransmitting messages that immediately get dropped again (the traditional protocol by design is unable to cope with this situation well, so this is not unexpected). Except for one case, all four- and eight-node numbers consequently are missing for the traditional protocol. The careful protocol performs up to 12.1 % worse than the traditional one. This is because the deadlock avoidance algorithm in the careful protocol starts to copy messages out of the on-board memory into normal memory as soon as it detects a large number of outstanding messages. This copying does not occur in the traditional protocol as all messages can be kept in on-board memory. As would be expected, this is an extreme situation that never happened with any of the real C* applications we have available. With these applications, the careful protocol always performed at least as well as the traditional one. When data processing is included, the careful protocol performs better for large messages. Most likely, this is so because the data processing slows down the sender sufficiently such that the receiver can keep up with the incoming data and no extraneous copying is needed. The only four-node number available for the

traditional protocol is over 10 times slower than the corresponding number for the careful one. This certainly could be reduced by tuning retransmission timeouts. However, we deliberately kept the retransmission timeouts too large because packet loss is so rare in our lab environment.

P	Pay-load [words]	Out-standing messages	Time [ms]					
			No Processing			With Processing		
			trad	care	Δ%	trad	care	Δ%
2	1	10×16	2.1	2.2	+5.0	2.1	2.3	+5.8
	1000	10×16	10.5	11.7	+12.1	15.9	14.2	-10.1
	2000	10×16	20.7	23.0	+11.2	30.4	27.7	-9.0
4	1	10×8	42.9	2.5	-94.2	n/a	2.5	n/a
	1000	10×8	n/a	10.5	n/a	n/a	15.6	n/a
	2000	10×8	n/a	20.8	n/a	n/a	29.2	n/a
8	1	10×4	n/a	3.5	n/a	n/a	3.5	n/a
	1000	10×4	n/a	10.8	n/a	n/a	18.4	n/a
	2000	10×4	n/a	21.0	n/a	n/a	n/a	n/a

Table 3: Flooding Measurements

Jacobi Iteration: The fourth and final test is based on a real C* program. It performs a two-dimensional Jacobi iteration. This is an iterative algorithm to find an approximate solution to Laplace's heat equation. The problem can be formulated as: given a square surface and the temperature along the edges, find the steady-state temperature on the surface. To keep the computation finite, a grid is imposed on the surface and the temperatures are computed at the grid-points only. One iteration consists of setting each grid point to the average of the old values of its four neighbors. Often, the algorithm is run until the changes between two successive iterations have become sufficiently small (i.e., the solution has converged). However, for simplicity, a fixed number of iterations is used here. The program uses a rectangular grid with a width and height that must be a power of two. The grid points are uniformly distributed across the available nodes. For example, if there are eight nodes and the grid is of size 8×8, then each node computes one row. In each iteration, a node p sends the first row it owns to the node above and the last row it owns to the node below. It then receives a row from the node above and below and continues to compute the new values. To stress networking performance, small grid sizes have to be used as the amount of computation grows linearly in the number of grid points, while communication grows linearly in the number of rows only. The same C* program was also run on a single machine running HP-UX 8.07. Notice that even with a relatively small grid size of 16×16, it is already beneficial to run the problem on a multiple nodes. Of course, this is partly due to the relatively inefficient code generated by the available C* compiler. However, qualitatively the results should remain the same when switching to a better compiler. In the best case, the careful protocol is 33 % faster but with a grid size of 8×8, it is faster to compute it on a single node. However, a 16×16 grid, two nodes using the careful protocol are faster than a single node and still outperform the traditional protocol by 29 %.

6.2 Traffic Patterns

As described in Section 5, the careful protocol allocates windows statically, based on the number of nodes in the cluster. An obvious idea is to use an adaptive window management scheme instead. A node would advertise large windows to nodes that are expected to send a lot and vice versa. In order to understand whether such a scheme could work at all,

	Grid Size											
	8×8			16×16			32×32			64×64		
	trad	care		trad	care		trad	care		trad	care	
P	Time [s]		Δ%	Time [s]		Δ%	Time [s]		Δ%	Time [s]		Δ%
1	0.19		n/a	0.38		n/a	1.18		n/a	4.37		n/a
2	0.42	0.28	-33.3	0.50	0.36	-29.0	0.81	0.76	-6.8	2.40	2.39	-0.4
4	0.48	0.35	-27.6	0.53	0.40	-24.3	0.71	0.58	-17.4	1.53	1.44	-5.9
8	0.57	0.42	-26.4	0.63	0.47	-24.7	0.61	0.60	-20.3	1.25	1.12	-10.9

Table 4: Jacobi Measurements

we performed a preliminary study of traffic patterns in C* programs. The C* environment was augmented to generate tracing information in a format that can be read by the ParaGraph visualization tool [10].

An analysis of thirteen C* applications revealed highly regular traffic patterns. In fact, all communication was either broadcast messages or messages sent to the left or right neighbor (i.e., node p sends to node $p-1$ or $p+1$). This leads us to believe that an adaptive scheme could work well for these test. That is, an adaptive scheme could predict future communication behavior based on past behavior with a high degree of accuracy. However, there are two warnings in order. First, due to the absence of a file system in our experimental environment, all tests were run with a single set of inputs only. Thus, we were unable to analyze how traffic patterns would change as a function of input. Furthermore, even if an adaptive scheme could predict future communication perfectly and therefore make perfect allocation decisions, it is unclear whether performance would improve. In fact, we believe that an adaptive scheme would likely offset the latency advantage careful protocols have over traditional ones. For example, if a node decides to enlarge the window of one node then, with a finite amount of receive buffers, it would have to shrink the advertised window of another node. However, in a distributed environment it is very difficult to revoke a right, once granted. One possibility would be to associate an expiration time with a window update. The window update would be valid only if used before the expiration time. Thus, a node could wait until window updates to nodes that have no data to send expire and then use some of these additional resources to advertise larger windows to the nodes that are transmitting data. This is unlikely to be a mechanism that could be implemented such that the latency advantage is preserved, though.

7 Future Work

The present work investigated a single networking service to compare performance between a careful and a traditional implementation. It is, of course, nearly impossible to extrapolate these results and make general statements on the relative performance of careful and traditional protocols. It would be interesting to see how other reliable services, like the one presented by TCP, or RPC-like services would compare.

The careful protocol implemented is so simple because it could use a static window management scheme. This works well as long as the number of nodes is on the order of dozens. However, if this were to be scaled to thousands of nodes, resource usage would be too low as many receive buffers would have to be reserved for nodes that never send any data. Notice that this is a serious problem only if frames can be long (like 8 KB). That is, a static scheme might be usable even for thousands of nodes, if the maximum frame length (MTU) is limited to, say, 256 bytes. In essence, this trades bandwidth for simplicity in the flow-control scheme. As a careful protocol with a static window management has low latency, the bandwidth lost due to restricting the MTU might not be excessive. Also, bandwidth

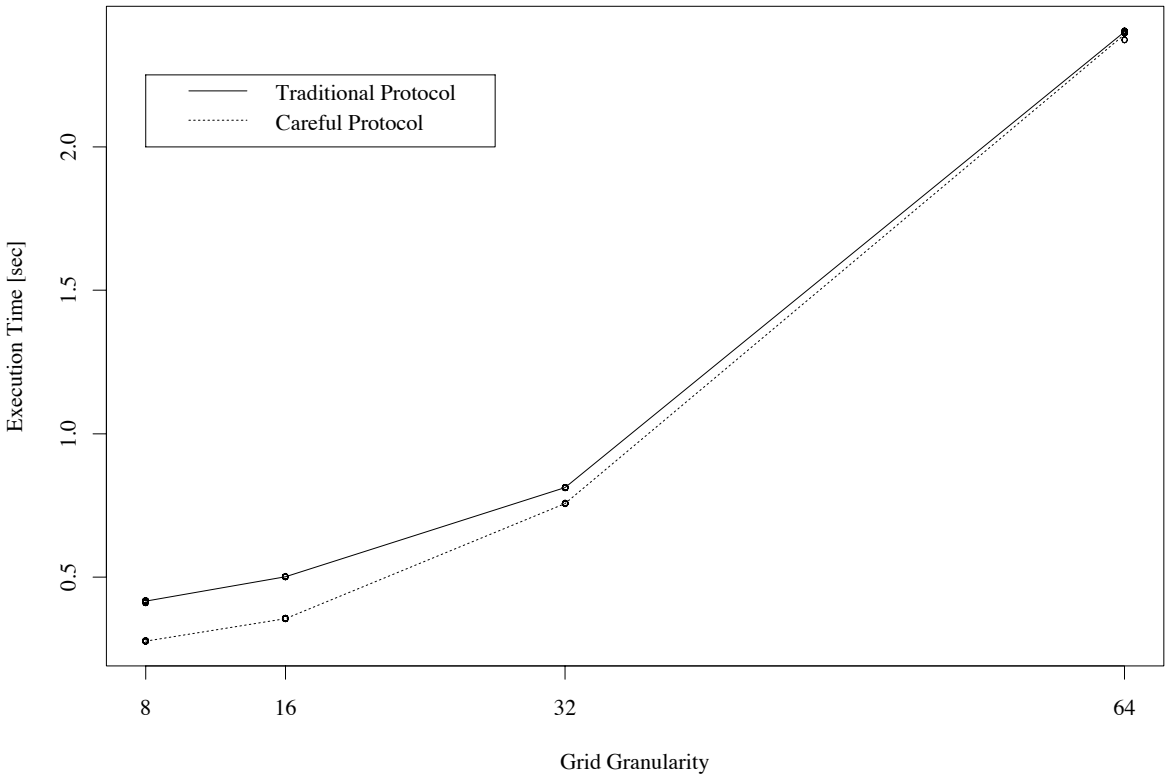


Figure 3: Jacobi Measurements

could be preserved by reserving receive buffers only immediately before they are needed. That is, before sending a large message, the sender would inform the recipient that a large message is present and request the allocation of an adequate number of receive buffers. This scheme is very similar to what is used in the NX/2 operating system [15]. Another approach would be to exploit the observed locality of communication. For example, windows could be advertised that are inversely proportional to the (logical) distance between the two nodes. This could be augmented by a forwarding scheme, such that distant nodes could communicate via intermediate nodes (greatly increasing the latency for this case, of course). In summary, we feel like the scalability problem could be solved while still achieving low latency for small messages and high throughput for large ones.

8 Conclusions

As the measurements indicate, careful protocols *can* perform significantly better than their traditional peers. However, it is important to bear in mind that this does not necessarily imply that careful protocols always perform better. With each protocol and each implementation, there are a multitude of decisions to be made that can influence the resulting performance one way or the other. We feel that the presented performance comparison is valid as the traditional protocol was developed before we had the careful one in mind and also because we spent roughly the same amount

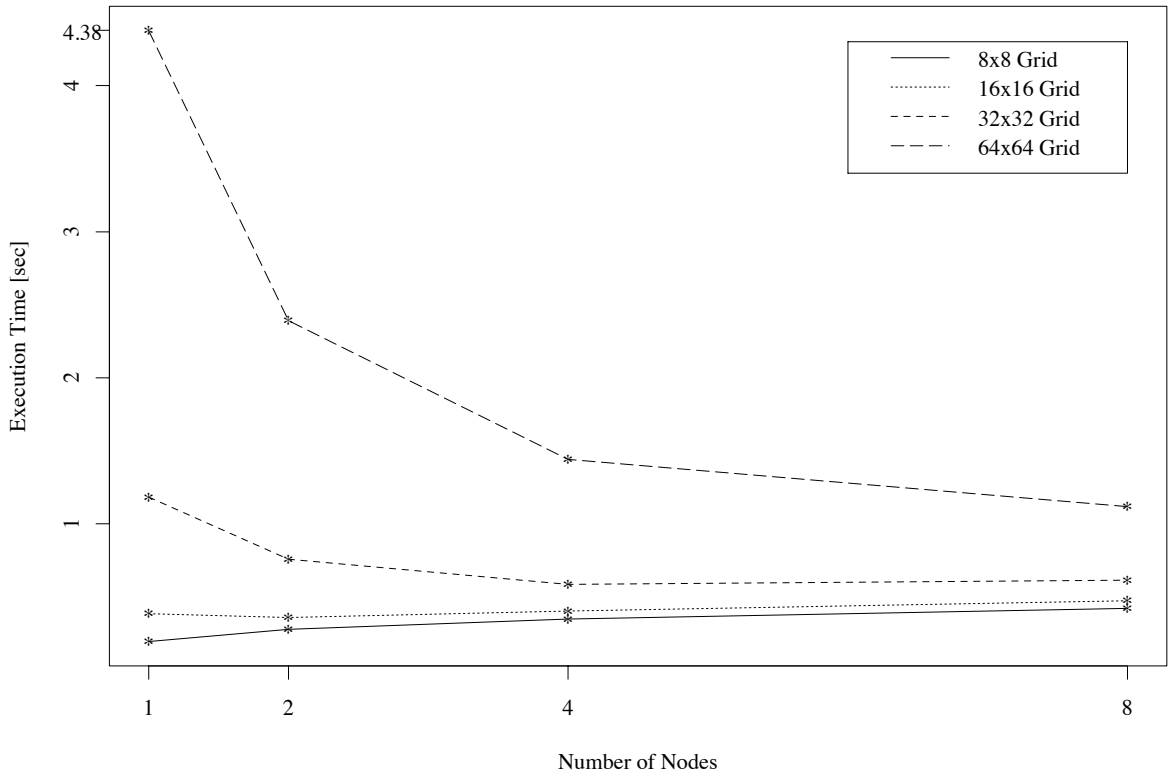


Figure 4: Jacobi Times for Careful Protocol as a Function of Number of Nodes

of time and effort on each implementation. Undoubtedly, there are missed opportunities for optimizations in both implementation.

Although it was not a simple task to get flow-control working, once this was done, the resulting careful protocol was much easier to understand and maintain than the traditional one. This is especially true if we consider that our traditional protocol has essentially no flow-control. We expect this to be true for all careful protocols that can employ a static window management. However, with dynamic window management, a careful protocol is likely to be at least as complex as a traditional one.

References

[1] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Menlo Park, 1991.

[2] D. Banks and M. Prudence. A high performance network architecture for a PA-RISC workstation. *IEEE Journal on Selected Areas in Communication*, 11(2):191–202, Feb. 1993.

[3] R. Berrendorf and J. Helin. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concurrency: Practice and Experience*, 4(3):223–240, May 1992.

[4] F. J. Burkowski, G. V. Cormack, J. D. Dyment, and J. K. Pachl. A message-based architecture for high concurrency. In *Proceedings of the First Conference on Hypercube Multiprocessors*, pages 27–37, Knoxville, Tennessee, Aug. 1985. SIAM.

[5] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *Proceedings of SIGCOMM '88 Symposium*, pages 106–114, Aug. 1988.

[6] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley. Afterburner. *IEEE Network*, 7(4):36–43, July 1993.

[7] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson. Network subsystem design. *IEEE Network*, 7(4):8–17, July 1993.

[8] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, Asheville, North Carolina, Dec. 1993. ACM.

[9] J. L. Frankel. A reference description of the C* language. Technical Report TR-253, Thinking Machines Corporation, Cambridge, Massachusetts, May 1991.

[10] M. T. Heath and J. E. Finger. ParaGraph: A tool for visualizing performance of parallel programs. Available from netlib@ornl.gov, 1993. Distributed with ParaGraph source.

[11] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl. The persistent relevance of IPC performance: New techniques for reducing the IPC penalty. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 186–190, Napa, California, Oct. 1993. IEEE.

[12] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991.

[13] V. Jacobson. Congestion avoidance and control. In *Proceedings of SIGCOMM '88 Symposium*, pages 314–329. ACM, Aug. 1988.

[14] D. Katz. RFC-1103: IP datagrams over FDDI networks. Available via ftp from ftp.nisc.sri.com, June 1989.

[15] P. Pierce. The NX/2 operating system. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume I, pages 384–390, Pasadena, California, Jan. 1988. ACM.

[16] C. L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, Jan. 1985.

[17] C. L. Seitz, W. C. Athas, C. M. Flaig, A. J. Martin, J. Seizovic, C. S. Steele, and W.-K. Su. The architecture and programming of the Ametek Series 2010 multicomputer. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, volume I, pages 33–36, Pasadena, California, Jan. 1988. ACM.

[18] C. L. Seitz, J. Seizovic, and W.-K. Su. The C programmer's abbreviated guide to multicomputer programming. Technical Report CS-TR-88-1, California Institute of Technology, Pasadena, CA 91125, Jan. 1988. Revision 1, 17 April 1989.

[19] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. Technical Report UCB/CSD 92/675, University of California, Berkeley, California, Mar. 1992.

A Detailed Measurements

This Appendix presents a detailed analysis of the cost of each operation that one of the protocols has to do but the other can omit. This analysis is restricted to the case where a short frame (4 bytes of payload) is sent from one node to another (i.e., like in the two node latency test with a small payload). Thus, if we sum up the execution times of these "extra" operations per protocol and take the difference, we should end up with a number that is close to the 27 μs latency difference measured via the latency test.

First, we list and shortly describe all the relevant differences between the two protocols. This is followed by histograms of the timing data we collected. Notice that the measurements were done via a simulator of the Medusa FDDI. The simulator was run on identical machines as the real cluster, but with HP-UX instead of Mach. Furthermore, in this configuration everything is in user-space, instead of being in the kernel.

Given these differences between the actual cluster and the simulator, cache effects are almost certainly different between the two systems. Nevertheless, the overall results of the simulator agree well with the latency difference observed on the real cluster. Thus, if not quantitatively, we would expect that the timing results are at least qualitatively accurate.

In order to keep the histograms readable, we cut off outliers. They are very obvious as they have execution times more than ten times bigger than the mode of the histogram. Such outliers are usually the result of pre-emption in the middle of a timed operation. Similarly, the histograms often show unusually small execution times. These are due to control messages that are not part of the actual latency test. They are needed due to idiosyncrasies in the current implementation of the traditional protocol and should be ignored.

It is important to notice that we did not attempt to optimize the protocols in response to these detailed measurements. We spent approximately the same amount of time in optimizing the two protocols, but "micro-optimization" were consciously excluded as performing them is almost prohibitively time-consuming and their benefit is strongly architecture and machine dependent.

The description below is organized as follows: the lists are nested according to the level of detail they provide. At the outermost nesting level, a per-protocol total cost of "extraneous" operations is given. In the next deeper nesting level, these costs are broken down into costs that occur during the send operation and the interrupt handler, respectively (the receive operations are identical in both protocols). With each additional nesting level, more detailed timing information is provided. The time reported at nesting level n is the sum of the times of the following list at nesting level $n+1$. Execution times are reported in number of cycles (20 ns/cycle).

Traditional Protocol (1418 cycles)**:**

- **Send** (318 cycles)**:**
  - **Decide not to set ACK flag** (7 cycles)**:** As only small fragments are sent, explicit acknowledgements are never requested and the ACK flag in the fragment header does not need to be set.
  - **Allocate and fill fragment descriptor** (109 cycles)**:** The traditional protocol needs to keep track of all fragments sent in order to be able to retransmit them. A fragment descriptor holds all the information necessary to retransmit a fragment.
  - **Set ACK mask** (24 cycles)**:** This operation determines how many ACKs are needed for the fragment being transmitted. In the latency test, only point-to-point messages are sent, so only one ACK is needed. This operation involves 5 instructions only but takes 24 cycles implying that it involves a cache miss on a clean cache line (cost: 18 cycles).
  - **Get TX event slot pointer** (32 cycles)**:** Events are kept track of via a cyclic buffer. Getting a pointer to an event slot involves checking the event number for having a valid value. This operation also stores in the event slot whether the fragment is a point-to-point or broadcast message.

Push fragment descriptor on stack (58 cycles)**:** Once the fragment descriptor is filled in, it is stored on a stack.

Collect TX buffers (62 cycles)**:** Once a buffer has been transmitted, it must be reclaimed to the free buffer pool if it has been deallocated in the meantime. In order to keep a count of available on-board buffers, transmitted buffers are collected in each send operation. The time of 231 cycles as given in the histogram includes a call to enforce mutual exclusion which adds 169 cycles that do not exist in the in-kernel version.

Check and update bit set (26 cycles)**:** The traditional protocol maintains two bit sets to keep track of which on-board transmit buffer is in what state (transmitting, allocated, or available). As can be seen in the corresponding histogram, each set operation takes roughly 13 cycles of which two are needed in the send operation.

Interrupt (1100 cycles)**:**

Maintain on-board buffer counter (14 cycles)**:** In order to guide resource allocation, the traditional protocol needs to keep a count of the number of free on-board buffers. This involves three instructions: a load, a decrement, and a store. The high cost of 14 cycles appears to indicate a cache miss.

Determine fragment priority (69 cycles)**:** Before accepting an incoming fragment, the interrupt handler checks whether it can accept it at all. This avoids deadlock if a receiver is heavily loaded.

Check for having to copy out fragment (32 cycles)**:** This checks whether a frame should be kept in on-board memory or copied out into normal RAM. This appears to usually involve a cache miss of 19 cycles.

Handle implicit ACK (779 cycles)**:** Under normal circumstances, the traditional protocol does not use explicit acknowledgments. Instead, it acknowledges fragments implicitly based on what is known to have arrived properly. This operation is very costly as it potentially has to update a lot of state. In the latency test, there is not really that much work to be done, but the generality of the operation still incurs a rather large performance penalty. We analyzed this operation in more detail, but presenting all results are beyond the scope of this paper. In essence, we found that due to the modularity of the code this operation involves many function calls (on the order of 10 calls). Furthermore, one call is via a function pointer, which costs 37 cycles instead of the 13 cycles for a direct function call. Thus, this is a classical instance of a tradeoff between modularity and performance.

We found that deallocating a fragment descriptor and its associated data takes 256 cycles. We would expect that optimizing for the special case of a single point-to-point fragment, it would be possible to get close to this time for handling an implicit ACK. In the best possible case, this would improve the latency of the traditional protocol by 10 μs.

Check and update fragments number (66 cycles)**:** This operation checks that the incoming fragment has the correct sequence number, updates the expected sequence number and also allocates a fragment descriptor.

Check for explicit ACK (6 cycles)**:** Check whether sender requested an explicit acknowledgment. In the latency test this never happens.

Check for retransmission (36 cycles)**:** This checks whether it is necessary to eagerly request the retransmission of a fragment. It involves roughly 30 instructions if no retransmission is need.

Check for all-to-all event (9 cycles)**:** Retransmissions are requested by the receiver so it needs to know who is expected to send fragments for a given event. This operation determines which nodes need to send fragments for the event the current fragment belongs to.

Check if complete (89 cycles)**:** If all fragments have been received for the event to which the received fragment belongs to, some state needs to be updated. In the latency test, every event consists of a single fragment, so the state always needs to be updated.

Careful Protocol (151 cycles)**:**

Send (42 cycles)**:**

Shrink window (21 cycles)**:** When sending a fragment, the careful protocol has to shrink the appropriate window by one. If the window is closed, it blocks, but this does never happen in the latency test. In the histogram, this operation takes 190 cycles, however 169 cycles are due to a mutex operation which is very cheap in the in-kernel version, so we can subtract 169 cycles.

Fill in window update (21 cycles)**:** When sending a fragment, a window-update is piggy-backed into the fragment. Just as above, the histogram gives the cost of this operation as 190 cycles. Again 169 cycles are due to a mutex operation which we can ignore.

Interrupt (109 cycles)**:**

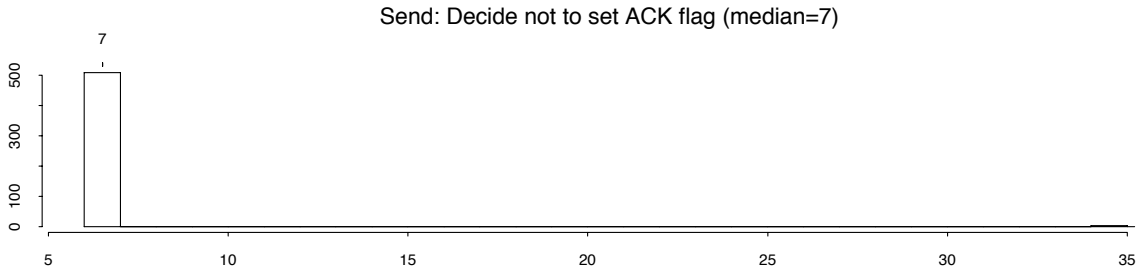
Process window update (52 cycles)**:** The interrupt handler updates the window according to the window update that has been piggy-backed in the fragment. The histogram indicates an execution cost of 52 cycles. Roughly 23 cycles are due to a cache miss, leaving 29 cycles for the 11 instructions in this operation.

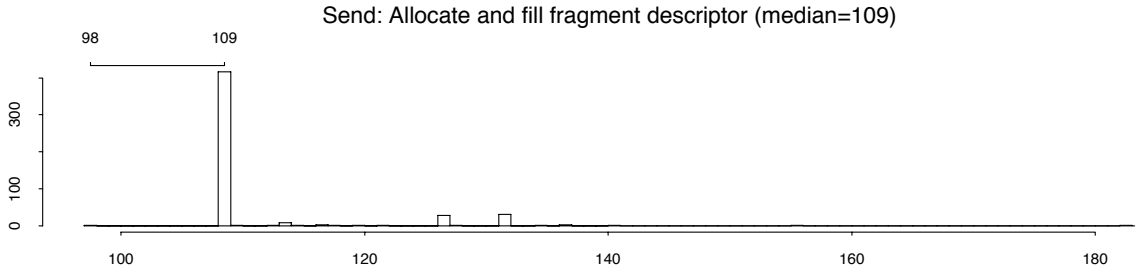
Track last event number (24 cycles)**:** The careful protocol needs to keep track of the last even for which a fragment has been received. It involves 8 instructions only, so the cost of 24 cycles is rather high.

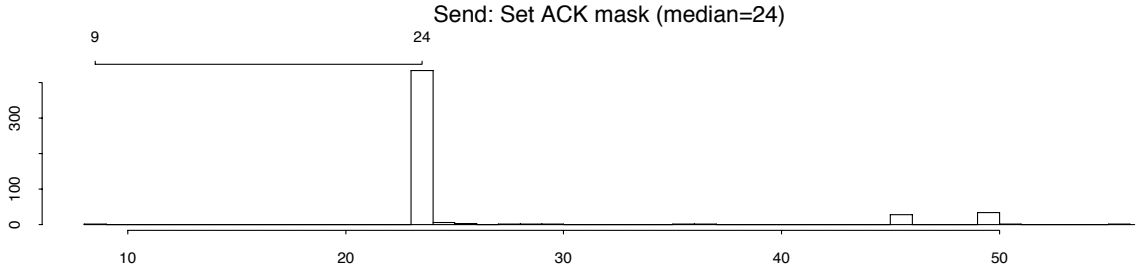
Decrement sender's window (29 cycles)**:** The protocol needs to keep track of the sender's window size. The cycles per instruction (CPI) for this operation is again very high (3.2)..

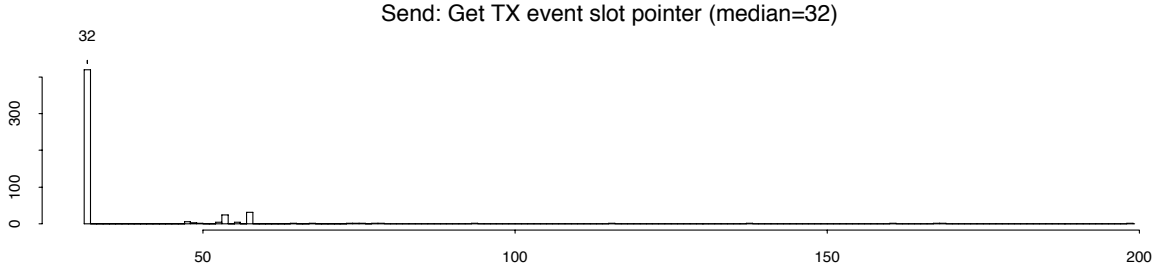
Skip copy-out (4 cycles)**:** This operation checks whether the received frame needs to be copied from on-board to normal RAM. With the latency test, this never happens.

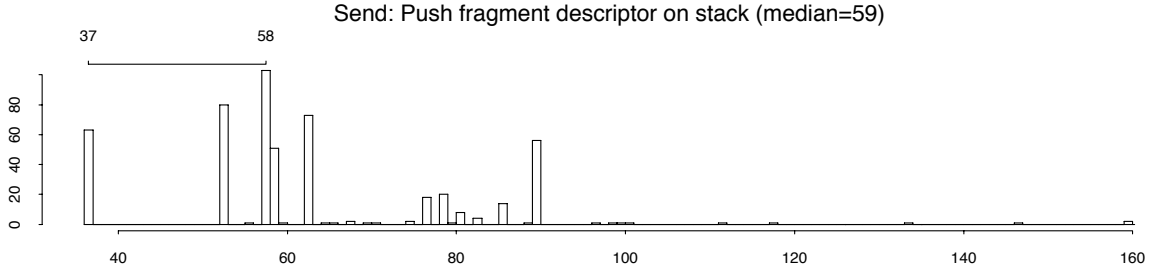
A.1 Traditional Protocol Histograms

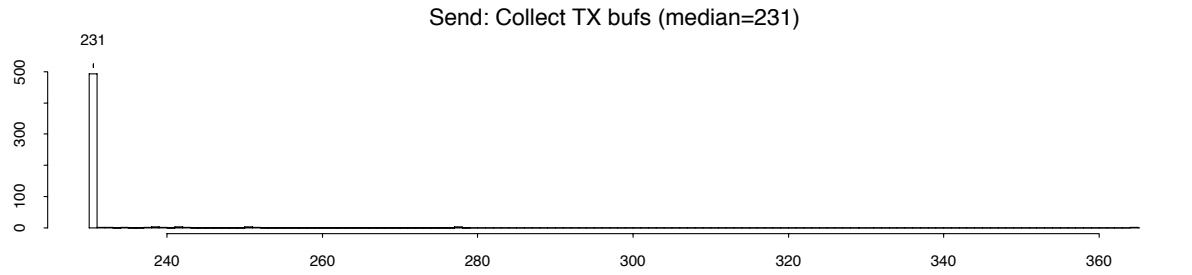


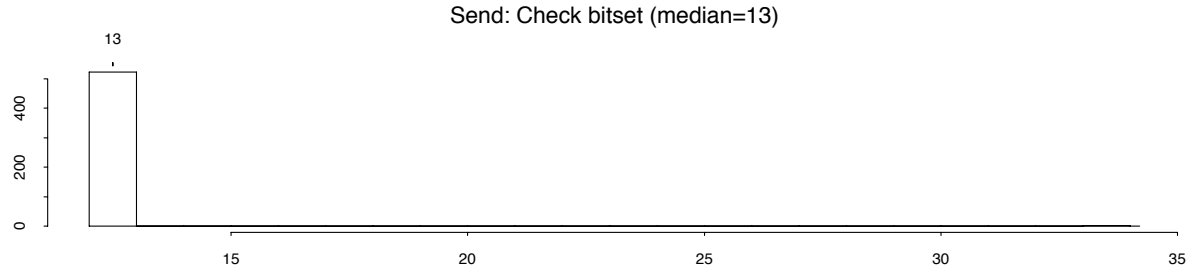


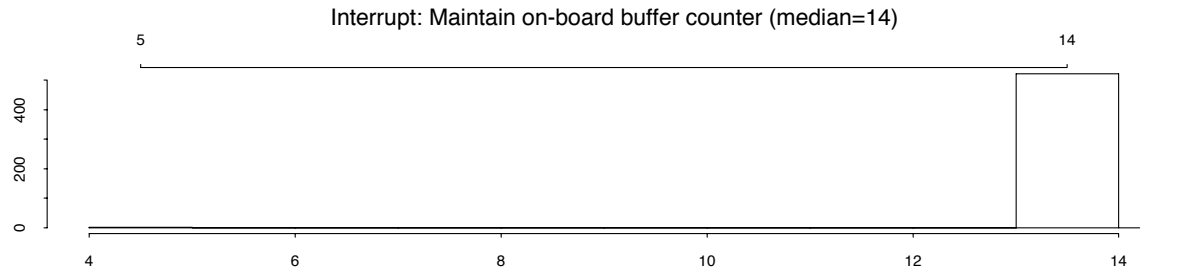


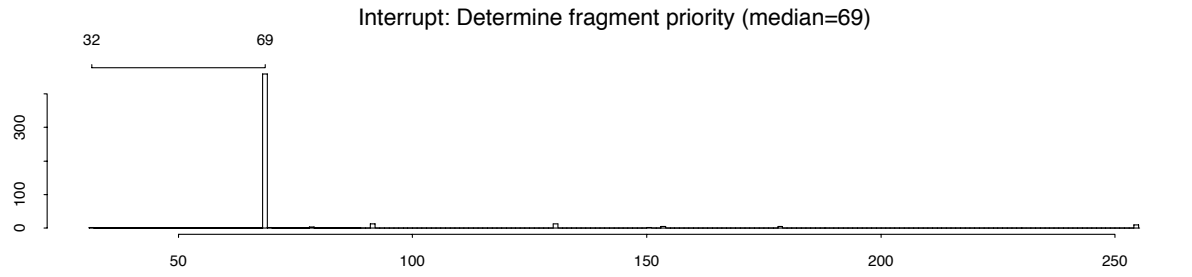


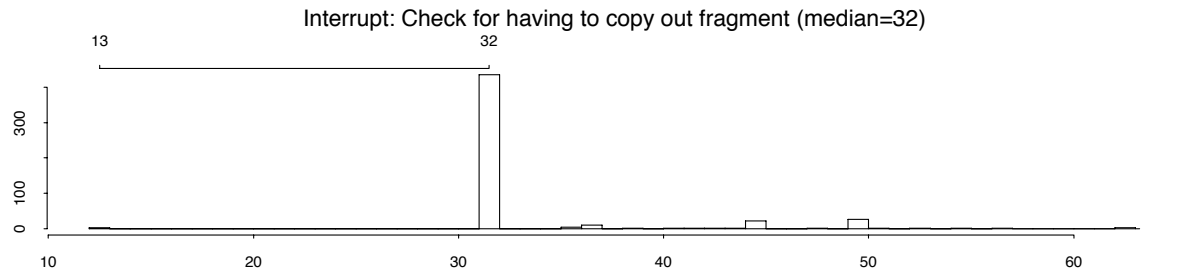


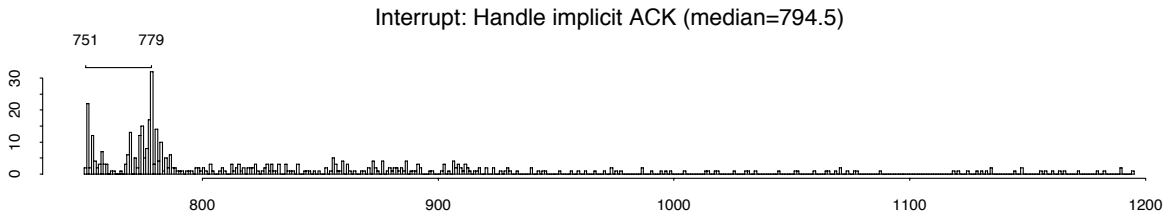

Send: Collect TX bufs (median=231)
231
0
100
300
500
240
260
280
300
320
340
360

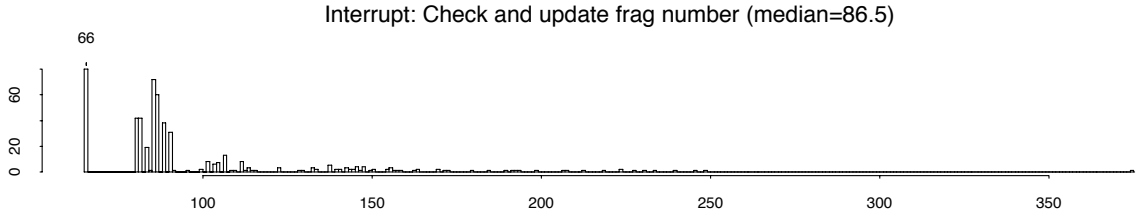

Send: Check bitset (median=13)
13
0
200
400
15
20
25
30
35

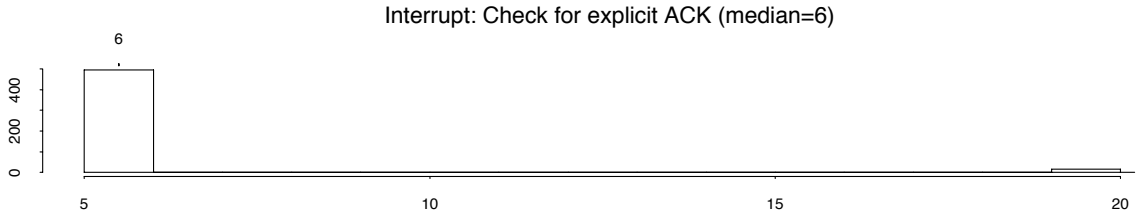

Interrupt: Maintain on-board buffer counter (median=14)
5
14
0
200
400
4
6
8
10
12
14

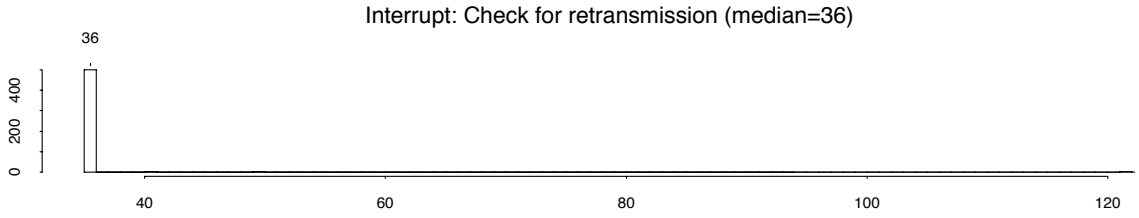

Interrupt: Determine fragment priority (median=69)
32
69
0
100
300
50
100
150
200
250

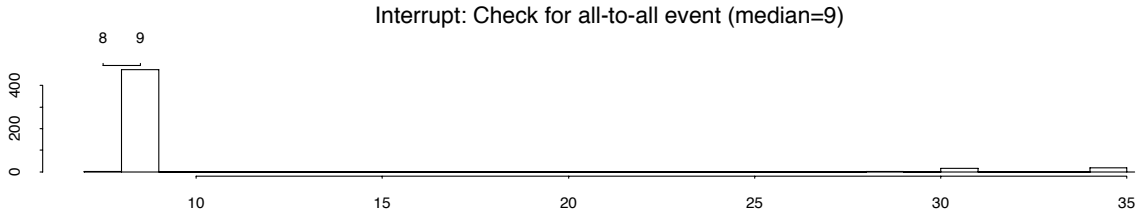

Interrupt: Check for having to copy out fragment (median=32)
13
32
0
100
300
10
20
30
40
50
60

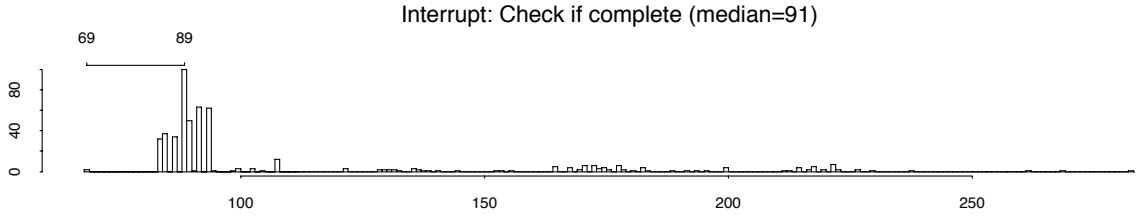

Interrupt: Handle implicit ACK (median=794.5)
751
779
0
10
20
30
800
900
1000
1100
1200


Interrupt: Check and update frag number (median=86.5)
66
0
20
60
100
150
200
250
300
350

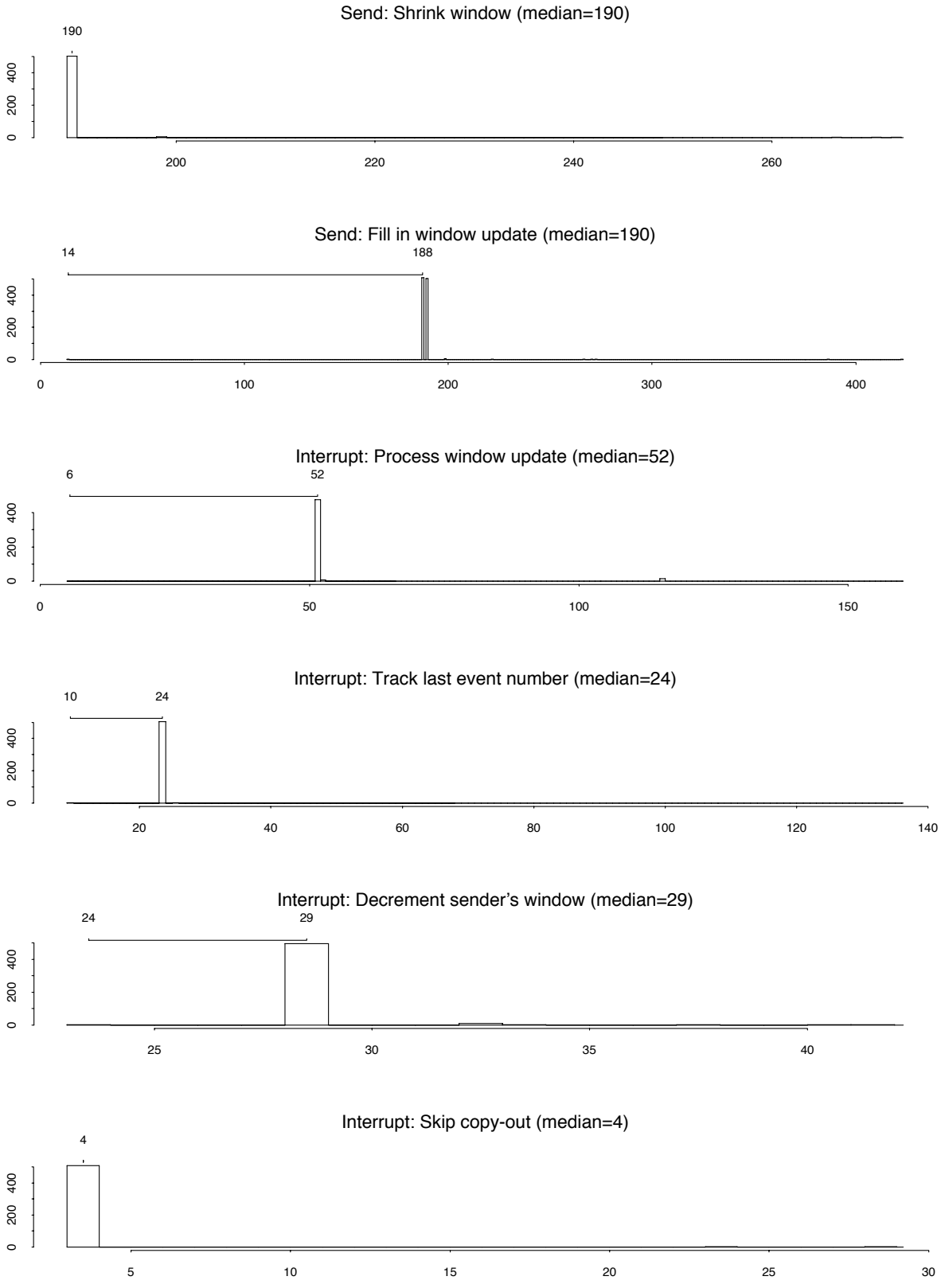

Interrupt: Check for explicit ACK (median=6)
6
0
200
400
5
10
15
20


Interrupt: Check for retransmission (median=36)
36
0
200
400
40
60
80
100
120


Interrupt: Check for all-to-all event (median=9)
8
9
0
200
400
10
15
20
25
30
35


Interrupt: Check if complete (median=91)
69
89
0
40
80
100
150
200
250

A.2 Careful Protocol Histograms



A.3 Lessons Learned

Clearly, the above results are specific to the given protocols, their implementations, the operating system, as well as the precise hardware configuration. As such they are difficult to generalized to different situations. However, there are a few lessons to be learned.

First, it is clearly not a good idea to count instructions in order to estimate execution time. This may have worked with earlier, simpler machines but clearly is inappropriate for modern RISC architectures.

Second, in our measurements we found that cache performance was often consistently bad. This can be partially explained by the fact that all buffer pools are operated under a FIFO policy. This was done because with this choice and the constraint that a pool always has at least one buffer there is no need for locking. We do not know whether the cost for locking would have exceeded the cost of the bad cache performance we observe now.

Finally, in the traditional protocol, the choice to eagerly request retransmission was a bad choice in the light of the excellent reliability of the FDDI ring. After all, if dropped packets are so rare, than there is no need to speed up this rare case. Considering the reliability of the FDDI link, it would have been more than appropriate to simply wait a few seconds until a lost packet got retransmitted.