Report Number: WUCS-96-09

1996-01-01

Reconsidering Fragmentation and Reassembly

Girish P. Chandranmenon and George Varghese

We reconsider several issues related to fragmentation and reassembly in IP. We first reconsider reassembly. We describe a simple expected case optimization that improves reassembly performance to 38 instructions per fragment if the fragments arrive in FIFO order (the same assumption made in header prediction) which has been implemented in the NetBSD kernel. Next, we introduce the new idea of Graceful Intermediate Reassembly (GIR), which is a generalization of the existing IP mechanisms of destination and hop-by-hop reassembly. In GIR, we coalesce the fragments at an intermediate router in order to use the largest sized packets on its outgoing... Read complete abstract on page 2.

Reconsidering Fragmentation and Reassembly

Girish P. Chandranmenon and George Varghese

Complete Abstract:

We reconsider several issues related to fragmentation and reassembly in IP. We first reconsider reassembly. We describe a simple expected case optimization that improves reassembly performance to 38 instructions per fragment if the fragments arrive in FIFO order (the same assumption made in header prediction) which has been implemented in the NetBSD kernel. Next, we introduce the new idea of Graceful Intermediate Reassembly (GIR), which is a generalization of the existing IP mechanisms of destination and hop-by-hop reassembly. In GIR, we coalesce the fragments at an intermediate router in order to use the largest sized packets on its outgoing interface. We show that GIR always outperforms hop-by-hop reassembly and can be implemented economically in routers with small processing and memory costs. We then reconsider fragmentation. We show that avoiding fragmentation has costs of its own in terms of increased packet processing and/or round-trip delays. We describe measurements in which TCP performance improves after turning on fragmentation. For examples, on Ethernet under NetBSDS using a 536 byte segment size for TCP we get a throughput of only about 5.5 Mb/s, whereas we get a throughput of 8.45 Mb/s using a segment size of 1460 bytes (without fragmentation) and a throughput of 8.82 Mb/s using a segment size of 16260 byes with fragmentation. We also describe a simple performance model that can be used to determine when fragmentation is beneficial. Finally, we address the major disadvantages of fragmentation. A major proglem is that when a fragment is lost, the entire TCP segment must be retransmitted, resulting in reduced or zero goodput under loss. We describe a mechanism - dynamic segment sizing, in which the segment size is dynamically reduced after loss - which address this problem. We describe simulations (using our modified NetBSD kernel) which shows that dynamic segment sizing keeps the goodput at reasonable levels even under extremely possy conditions. All our mechanisms (reassembly optimization, GIR, and dynamic segment sizing) are orthogonal and can be applied to other protocol suits besides TCP/IP.

Reconsidering Fragmentation and Reassembly

Girish P. Chandranmenon
George Varghese

WUCS-96-09

April 15, 1996

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

April 15, 1996

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

We reconsider several issues related to fragmentation and reassembly in IP. We first reconsider reassembly. We describe a simple expected case optimization that improves reassembly performance to 38 instructions per fragment if the fragments arrive in FIFO order (the same assumption made in header prediction) which has been implemented in the NetBSD kernel. Next, we introduce the new idea of *Graceful Intermediate Reassembly (GIR)*, which is a generalization of the existing IP mechanisms of destination and hop-by-hop reassembly. In GIR, we coalesce the fragments at an intermediate router in order to use the largest sized packets on its outgoing interface. We show that GIR always outperforms hop-by-hop reassembly and can be implemented economically in routers with small processing and memory costs. We then reconsider fragmentation. We show that avoiding fragmentation has costs of its own in terms of increased packet processing and/or round-trip delays. We describe measurements in which TCP performance *improves* after turning on fragmentation. For example, on Ethernet under NetBSD using a 536 byte segment size for TCP we get a throughput of only about 5.5Mb/s, whereas we get a throughput[1] of 8.45Mb/s using a segment size of 1460 bytes (without fragmentation) and a throughput of 8.82Mb/s using a segment size of 16260 bytes with fragmentation. We also describe a simple performance model that can be used to determine when fragmentation is beneficial. Finally, we address the major disadvantages of fragmentation. A major problem is that when a fragment is lost, the entire TCP segment must be retransmitted, resulting in reduced or zero goodput under loss. We describe a mechanism — dynamic segment sizing, in which the segment size is dynamically reduced after loss — which addresses this problem. We describe simulations (using our modified NetBSD kernel) which shows that dynamic segment sizing keeps the goodput at reasonable levels even under extremely lossy conditions. All our mechanisms (reassembly optimization, GIR, and dynamic segment sizing) are orthogonal and can be applied to other protocol suites besides TCP/IP.

[1] Mb/s = 1048576 bits/second

Reconsidering Fragmentation and Reassembly

Girish P. Chandranmenon
girish@cs.wustl.edu
+1 314 935 4163

George Varghese
varghese@askew.wustl.edu
+1 314 935 4963

1. Introduction

It is necessary to design network protocols that can incorporate diverse technologies because technology constantly evolves. An important aspect of diversity is the maximum packet size supported by each network technology. For reasons both technical[1] and political, most network technologies have different maximum packet size. For instance, X.25 uses 32768 bits, Ethernet uses 1500 bytes, FDDI uses 4500 bytes, and ATM uses 53 byte cells.

The Internet Protocol(IP)'s success for the last 15 years, during which several new technologies emerged, is due to its support for diversity. IP's solution to the use of different maximum packet sizes (MTU sizes) is to split datagrams that do not fit into the packet size of the link to be traversed, and to coalesce the pieces (fragments) at the receiving node. This is known as fragmentation and reassembly. The IP scheme is very general: it allows fragments to be further fragmented, and allows fragments to take different routes to the destination. Three fields in the IP header allow this versatility. Each datagram is given a packet identifier, which is copied to each of its fragments; each fragment also carries an offset field (i.e., where the fragment starts in the original packet) and a fragment length. If all fragments arrive at the receiver, the common packet ID allows the receiver to associate the fragments with a single packet, and the length and offset fields are used to paste together the fragments in the right order. The same solution is used by the OSI Routing Protocol [Per92].

Although fragmentation and reassembly has been a key tool for IP in dealing with diversity, there has been a perception that fragmentation is harmful for performance and useful only for interoperability. Kent and Mogul, in a very influential paper [KM87], provide a perceptive analysis of the disadvantages of fragmentation. Many protocol designers consider fragmentation to be at best a necessary evil; for example, recent proposals for IP version 6 [Hin94] appear to disallow fragmentation by routers (unless hop-by-hop fragmentation is used for links with small MTU sizes).

In this paper, we reexamine the advantages and disadvantages of fragmentation and reassembly. While the disadvantages cited in [KM87] remain valid, we describe mechanisms to avoid or alleviate the effect of these disadvantages. We also show (using a combination of analysis and measurement) that under certain circumstances fragmentation *can improve* performance. The performance improvement is aided by optimizing the fragmentation and reassembly code. We describe

[1] Technical issues that influence the choice of maximum packet size include the desire to limit the time occupied by one packet on a link, reducing header overhead, clock recovery and CRC coverage issues.

Algorithm	Deficiency
Arbitrarily small datagram size	Inefficient use of network; too many packets to process; bad performance in case of multicast.
MTU discovery	One round trip time delay in startup.
Using Don't Fragment flag	large delay in startup.
Help from routing protocols	does not work with hierarchical routing.

Table 2: Problems with algorithms that try to avoid fragmentation.

2.2. Techniques for Avoiding Fragmentation

To avoid the problems cited above, Kent and Mogul suggested a number of schemes [KM87] to avoid fragmentation. We discuss the drawbacks of these schemes below. The discussion is summarized in Table 2.

There are two major techniques suggested in [KM87]: using a default MTU size or discovering (and using) the minimum MTU on the path. One proposed discovery method is to send a probe to the destination which each router updates to record the smallest MTU encountered. This requires router modifications. A second approach, which has been specified and implemented by some systems, is outlined in [KM87] and elaborated in RFC 1191 [MD81]. This approach uses the "don't fragment" flag and uses the MTU of the first hop as the initial datagram size. Whenever a router cannot forward this datagram because of a smaller MTU on the outgoing link, it discards the datagram and sends an ICMP message back to the sender. The source can then retry with a smaller sized packet. RFC 1191 suggests using a table of common MTU sizes so that the sender can converge quickly on a likely minimum MTU size.[2]

Two problems with these schemes are:

Extra Round Trip Delay for Discovery Schemes: The Path MTU scheme takes at least an extra round trip delay to start sending a message. The RFC 1191 scheme does not incur the extra round trip delay to start sending data, However, if we have n hops with decreasing MTU sizes, the first n packets will be dropped, leading to possibly more than n round trip delays for the first n packets to be retransmitted. Round trip delays are increasingly significant for high speed networks and applications that care about latency (e.g., HTTP and RPC traffic). Caching can help but may not be applicable for HTTP like connections that are unlikely to exhibit significant locality [Mog95]. These problems are exacerbated when using multicast communication.

Large number of Packets sent for a Data Transfer: Both schemes for avoiding fragmentation have the disadvantage that the sender must use the smallest MTU size on the path, even if there is only one link on the path (or tree, in the case of a multicast tree) that has a small size. This translates into a larger number of packets sent and processed by the hosts and routers. If the bottleneck is host protocol processing, this can adversely affect performance (see Section 4 for a detailed argument).

Despite these disadvantages, there are situations in which the schemes outlined in [KM87] and RFC 1191 are useful. The path discovery schemes are orthogonal to (and can be combined with) our main ideas; our dynamic segment sizing scheme is an elaboration of an idea suggested in [KM87].

[2]Routers that support this feature can also return the MTU of the outgoing link in the ICMP message so that the source can choose its segment size more intelligently.

2.3. AAL-5 and ATM

The new ATM (Asynchronous Transfer Mode) technology uses a very small packet(cell) size, of 53 bytes to reduce latency. Since ATM networks use a small cell size, larger data packets have to be fragmented into cells and reassembled at the boundaries of the network. The most popular standard for this is AAL-5 (ATM Adaptation Layer–5). In ATM, all cells are sent and received in FIFO order (but cells can be lost) and the last cell in a data packet is identified with a flag. The guaranteed FIFO delivery and the use of the flag makes it easy to reassemble at the destination. In the last few years, there have been several real implementations that implement AAL-5 at gigabit rates. When compared to ATM, the complexity of the IP reassembly algorithm is due to the fact that IP has to handle out of order fragments while ATM does not. In Section 3.1 we will use a simple expected case optimization that allows IP reassembly to be as efficient as AAL-5 in the case when fragments arrive in order.

2.4. Other Related Work

Routing protocols can help in figuring out the smallest MTU to the destination. We are not sure if this has been suggested in the literature but the following simple scheme can be used. In this method, each router keeps the smallest MTU to the destination along with the next hop to send the packets for that destination. (It is quite easy to modify Bellman-Ford or Link State Packet Routing to compute this, see appendix.) Whenever the source has to send a packet, it has to query the nearest router to find the smallest MTU to the destination. While this approach works well in interconnected LANs, it does not work as well with hierarchical routing, since all routers do not keep information about all destinations. Thus for packets traveling between areas the router closest to the source does not have information about the lowest MTU size in the portion of the path that lies in the destination area. Each area could advertise a default MTU size for its area that is equal to the smallest MTU size in the area; however, this can lead to using too small a packet size.

Feldmeier's paper on "*chunks*" [Fel93] describes a way to fragment packets such that each fragment describes itself completely. It also discusses algorithms for fragmentation and reassembly using *chunks*. These methods, though general, require extensive modifications to the packet formats. It seems better to optimize existing IP before designing a new protocol for ease of reassembly.

3. Reconsidering Reassembly

We begin by reconsidering reassembly. We first describe a simple expected case optimization that can be used to considerably speed up IP reassembly. Next, we introduce a novel idea, graceful intermediate reassembly: this is a new reassembly scheme for intermediate routers that always outperforms the older notion of hop-by-hop reassembly.

3.1. Optimized Reassembly

IP reassembly has to handle lost, duplicate and out of order fragments. This makes the complete implementation complex. We apply a common case optimization to the reassembly algorithm in the same way header prediction [Jac90] has been applied to TCP processing. In the expected case, fragments arrive in order, without loss or duplication. In this case reassembly is simple: we coalesce fragments as they arrive, until the last fragment arrives, and then pass the reassembled packet to the higher layer. We describe our implementation in detail after a brief introduction to the existing implementation.

Four fields in the IP header enable reassembly: a datagram identifier, a 'more fragments' flag, fragment length, and an offset indicating the offset from the start of the datagram of the first data byte in the current fragment. The NetBSD implementation keeps a reassembly list, which is a list of fragmented datagrams. Each element in reassembly list is itself a list of fragments received for that datagram, kept in increasing order of their offsets. When a new fragment F is received, the code steps through the reassembly list to find which datagram F belongs to, and then searches the fragment list of the datagram to find where F should be inserted.

Our optimized implementation keeps *two* reassembly lists. One is a 'good' list consisting of datagrams whose fragments have so far been received in order; the second list is for datagrams which have received at least one fragment out of order. We cache the following information about the datagram (as long as it remains in the 'good' list) whose fragment was received last: source IP address, datagram identifier and next expected offset. We also keep track of a pointer to the tail of the mbuf[3] chain. This helps us attach the new fragment quickly at the end of the chain. When a new fragment arrives, it is checked against the stored hints (expected source address, datagram identifier and offset) to check if it is the expected fragment.[4]

If we get a "cache hit", we append this fragment to the partially reassembled datagram, effectively coalescing fragments as they arrive.[5] On a cache miss, if the fragment is for a new datagram, we add it to the 'good' list if it is the first fragment; if the fragment is an out of order packet, we move it from the 'good' list to the old reassembly list as two fragments (one coalesced fragment plus the new fragment that just arrived), and let the older algorithm take care of it.

The expected path of the reassembly algorithm is further improved by keeping both the datagram identifier and the offset in the same integer hint. This works well for big endian machines; for little endian machines we have to work harder. On a sparc processor, the expected path (check for the expected case, append the fragment, update length fields in the fragment, and update the expected offset) takes 38 instructions per fragment.

We implemented this algorithm in the NetBSD kernel on Sun–4/300 series machines and measured the end to end throughput using the `ttcp` program. The results are shown in Figure 1. The x-axis (segment size) is in logarithmic scale. The total number of bytes transferred was 50 MBytes. One important observation is that throughput increases upto a certain segment size for both implementations and then it decreases. The reasons for this will be discussed in detail in Section 4.1.3. When the segment size used by TCP is smaller than Ethernet's MTU size, both implementations behave the same, since there is no fragmentation taking place. For larger segment sizes,[6] the efficient implementation gives better throughput and the difference between them increases as the segment size (or number of fragments) increases. This is because of the numerous linear list traversals in the old implementation — each of these traversals pay a price proportional to the number of fragments received so far. Also note that the segment sizes plotted are those which exactly fit into an integral number of Ethernet frames. Using a 536 byte segment size for TCP we get a throughput of only about 5.5Mb/s, whereas we get a throughput of 8.45Mb/s using a segment size of 1460 bytes (without fragmentation) and a throughput of 8.82Mb/s using a segment size of 16260 bytes with fragmentation. We use 16260 byte (1460 bytes of data in first fragment and 1480 bytes each in next

[3]Mbufs are memory management structures used in NetBSD.

[4]The IP specification [Pos81] says that a fragment is uniquely identified by source address, destination address, datagram identifier and protocol identifier. We assumed that sources use a unique global identifier for each IP datagram they send. Therefore, the <source address, datagram identifier> pair should be sufficient to uniquely identify the datagram of a fragment.

[5]The original implementation uses the pointer to the IP header to chain the fragments of a datagram. In our case we do not keep them as fragments but coalesce them to look like a single large fragment. We use the mbuf pointer itself instead of the data pointer that points to the IP header. This allows us to avoid a call to `m_pullup()`, an expensive call which ensures that the data pointer points to the data within the first mbuf.

[6]The NetBSD kernel was modified to allow us to set TCP_MAXSEG to a larger value in order to force fragmentation.

Disadvantage	Our Solutions
Inefficient Resource Usage	Intelligent choice of source segment size (Section 4.3) Graceful Intermediate Reassembly (Section 3.2)
Poor performance under loss	Dynamic Segment Sizing (Section 4.2)
Inefficient Reassembly	Optimized reassembly (Section 3.1). Deadlock prevention (Section 4.3)

Table 1: **Arguments against using fragmentation and our solutions.**

our optimized code (implemented in the NetBSD kernel) and measurements showing throughput improvements using fragmentation.

We also introduce a *novel* idea: a new form of reassembly called *Graceful Intermediate Reassembly* that allows intermediate routers to partially reassemble fragments, and show that such graceful reassembly always outperforms hop-by-hop reassembly. Finally, we describe mechanisms to avoid the disadvantages of fragmentation. In particular, we elaborate on and develop a scheme first suggested in [KM87], that we call *Dynamic Segment Sizing*. This allows us to cope with the throughput reduction caused by the fact that a fragment loss leads to the loss of the entire packet.

This paper is organized as follows. We start in Section 2 by reviewing previous work, including the arguments against fragmentation. Next, we reconsider reassembly in Section 3, where we describe our optimized implementations and Graceful Intermediate Reassembly. We reconsider fragmentation in Section 4, where we describe performance measurements, a performance model, and Dynamic Segment Sizing.

2. Previous Work

We review the arguments against fragmentation and the techniques suggested to avoid fragmentation or improve reassembly performance.

2.1. Disadvantages of Fragmentation

The main arguments against fragmentation in [KM87] remain valid and are discussed below.

Fragmentation causes inefficient resource usage: Bad choices of sender segment sizes can result in a large number of fragments which increases the number of packet headers to be processed. For instance, if datagrams of size 1010 are fragmented over a link with MTU size 1000, downstream nodes receive approximately twice the number of packets/fragments (than if the sender sent datagrams of size 1000).

Poor performance when fragments are lost: Fragments can be lost due to congestion, link errors, or idiosyncratic gateway behavior that results in the same fragment being continuously lost. Each fragment loss results in a segment retransmission, which can cause reduction in throughput.

Efficient Reassembly is difficult: Current implementations of fragmentation and reassembly are not optimized. Also, since IP fragments do not carry the length of the entire packet, it is possible to create reassembly deadlocks with a large number of partially reassembled fragments.

To anticipate, our solutions to these problems are described in Table 1.

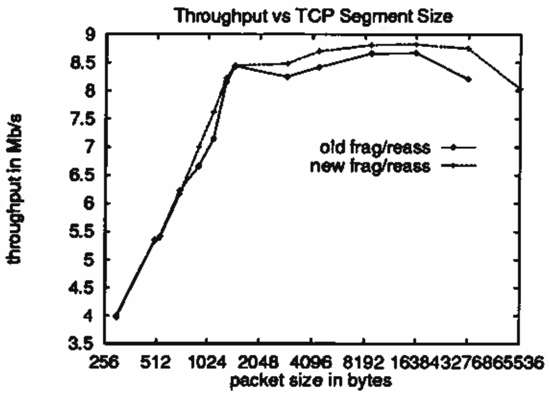

Figure 1: **Throughput in Mb/s seen by the application while using fragmentation. Optimized implementation gives better throughput than the older implementation; this effect will be more pronounced on a slower machine.**

10 fragments — only the first fragment contains the TCP header) segments instead of 16384 byte segments.

As an aside, we modified the fragmenting process as well. The NetBSD fragmenting algorithm creates a list of fragments and hands them to the Ethernet driver in a tight loop. We re-implemented this in a more pipelined fashion, feeding the interface driver as we create the fragments.

3.2. Graceful Intermediate Reassembly

IP reassembly is usually done only at the destination, although fragmentation can be done at any intermediate router. We call this *destination reassembly*. In some networks, it is common for a router to fragment a packet going over a single link and have a next-hop router (which is also on the same link) reassemble the packet. We call this *hop-by-hop reassembly* or HHR. In hop-by-hop reassembly or HHR, if a router does reassembly, then it only reassembles packets fragmented by the *previous hop router*.

HHR is useful to make links that have low MTU size appear to have a larger MTU size. For instance, two routers on a point-to-point link with MTU size 150 can make the link appear to effectively have an MTU size of 1500 by fragmenting packets of size larger than 150 and reassembling these packets on the other end of the link. HHR may be useful in a number of instances. Consider two FDDI rings (MTU 4500) connected by two routers and an Ethernet (MTU 1500). As a second example, consider a multicast tree that consists mostly of links of large MTU size and only a few links of small MTU size. Rather than have all packets be sent with the smaller MTU size, the routers bordering the small MTU size links can use HHR to boost the MTU size on those links.

In this section, we generalize these two extreme cases of reassembly (destination and hop-by-hop) to what we call *Graceful Intermediate Reassembly (GIR)*. GIR allows intermediate routers to reassemble fragments regardless of whether the previous hop router fragmented the packet; thus GIR is more general than HHR. GIR is also more discriminating than HHR because it can make its reassembly decision based on the next-hop link and can do partial reassembly of fragments. The contrast between the three schemes is illustrated in Figure 2.

We show that GIR is a generalization of HHR *and* destination reassembly, and a form of GIR can always outperform HHR. The general GIR algorithm is simple and is illustrated in Figure 3. The general algorithm keeps coalescing contiguous fragments until either a reassembly predicate RP is satisfied or a timer expires. When this happens, if P has been reassembled, P is "pushed" out (either

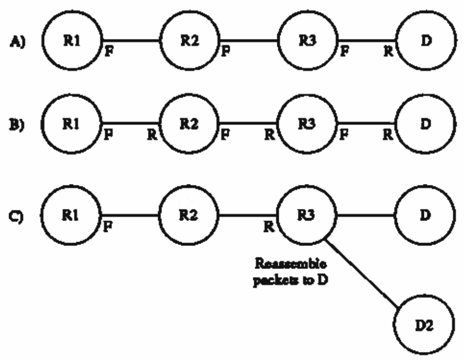

Figure 2: Destination Reassembly (A), Hop-by-Hop Reassembly (B), and Graceful Reassembly (C). F denotes fragmentation and R denotes reassembly. Note that Graceful Reassembly allows reassembly to occur at an intermediate node (e.g., $R3$) though fragmentation is done at any earlier node (e.g., $R1$, which is 2 hops before $R2$). GIR also allows the option of reassembling packets to some destinations (e.g., D) and not to others (e.g., $D2$).

```
When a fragment $F$ is received:
  Match fragment with other fragments for same packet, say $P$, if any.
  Coalesce contiguous fragments to make smaller fragments
  If (Reassembly Predicate $RP$ is satisfied) then
    $TerminateReassembly(P)$

Timer tick:
  For all partially reassembled packets $P$ that have not
    satisfied Reassembly Predicate $RP$ for time $T$ do
      $TerminateReassembly(P)$

$TerminateReassembly(P)$
  If ($P$ is complete) then
    Send $P$ to next hop (or to application, if at destination)
  Else (if at destination or hop-by-hop) then
    Destroy packet $P$ (*reclaim storage*)
  Else (*graceful intermediate reassembly*)
    Send coalesced fragments to next hop
```

Figure 3: Code for Graceful Reassembly

on the outgoing link or the receiving client if at the destination.) The most interesting difference occurs when P is not completely reassembled and the predicate RP fails. If we are simulating HHR or destination reassembly, we simply destroy the received fragments; however, in the general GIR algorithm, we "push" the coalesced fragments on to the outgoing link.

It is easy to see that the code in Figure 3 can simulate destination reassembly. However, as it stands the code is extremely difficult to implement in routers if the reassembly predicate is arbitrary.

Once again, we optimize for the common case where fragments arrive in order. There is a major problem in reassembling at the router. If the MTU size of the outgoing link is not large enough to carry the entire datagram, it will have to be fragmented again. Consider two cases: in the first, a packet of size 1500 bytes is fragmented into 500 byte fragments which arrive at a router that has an outbound MTU size of 1500; the second case is similar except that the outbound link size at the

Reassembly Predicate $RP(P)$ to simulate hop-by-hop and destination reassembly:
P has been completely reassembled

Reassembly Predicate $RP(P)$ for an efficiently implementable form of Graceful Reassembly
(P has been completely reassembled) OR
(No reassembly is being done for the destination of P) OR
(A fragment has been received out of FIFO order) OR
(If coalesced, fragment size exceeds or is close to exceeding the MTU size of the outgoing link for P)

Figure 4: Reassembly Predicates for Various forms of Reassembly

router is 1000 bytes. In the first case, the router can perform hop-by-hop reassembly. But what should the router do in the second case? And how can the router tell these two cases apart, since IP fragments do not carry the size of the original packet?

To avoid these issues we suggest a novel idea: *do partial reassembly on the fly*. The method is as follows: keep reassembling fragments until the coalesced size exceeds the outbound MTU or FIFO is violated. In either case, the coalesced partial datagram — a large fragment — is sent out on the link. This may seem complex, but the required code is simple in the expected FIFO arrival case; also the Data Link buffering can be used to store the partially reassembled packet. If fragments arrive in FIFO order, we do essentially the same algorithm as the end node reassembly algorithm described earlier. We confirm that this is another in-oder fragment for the same packet; if so, we coalesce it with the previous fragments. We might also want to do partial reassembly for certain destinations in the routing table and not for others. This leads to the efficiently implementable reassembly predicate shown in Figure 3.2.

The processing cost to the router is small (the 38 instruction estimate for expected case reassembly processing should apply to the router except for the additional cost of checking the outgoing MTU size against the coalesced fragment size). Notice also that IP forwarding need only be consulted when the *first* fragment arrives so that the outgoing link can be determined. The outgoing link can be cached along with the packet ID and expected offset. The remaining IP processing (e.g., adjusting TTL fields etc.) can be done after the coalesced fragments are ready to be pushed.

Memory costs would be large if we kept separate reassembly buffers for each concurrent flow. However, the simplest approach is to keep one reassembly buffer for a current packet P being reassembled. If another fragment for a different packet is received before the reassembly predicate for P is satisfied, P is "pushed". Thus the memory cost is also small. If support for QoS for flows in IPv6 becomes available, it is likely that per-flow buffering will be required to supported Weighted Fair Queuing Algorithms [DKS89]. The per-flow buffering that is required for QoS can then also be used for reassembly.

Comparison with Hop-by-Hop Reassembly

We compare the efficient Graceful Intermediate Reassembly (GIR) Algorithm with conventional Hop-by-Hop reassembly (HHR). GIR can simulate HHR but offers the following advantages over HHR:

More discrimination: Because GIR uses information about the next hop MTU, it can make better choices than HHR. For example, in Figure 2, suppose the link from $R2$ to $R3$, and the link from $R3$ to $D2$ are low-delay SLIP link with MTU size 296. Suppose further that the link from $R3$ to D is an

Ethernet. Using GIR, $R3$ can reassemble Ethernet packet fragments going to D and not reassemble any fragments destined to $D2$. HHR does not allow this discrimination because it does not "look ahead".

Partial Reassembly: GIR is graceful because it stops partial reassembly when it does not make sense to do more. Thus in Figure 2 if $R3$ receives a packet of size 4500 in fragments of 296 bytes and the link to D has MTU size 1500, $R3$ will push out 3 coalesced fragments of size ≈ 1500. HHR will completely reassemble the packet and then promptly undo its work on the next hop, increasing latency and processing.

No protocol changes: GIR uses the IP fragmentation headers. For layering purposes, some HHR protocols use their own fragmentation headers adding overhead and extra mechanisms.

No need for cooperation from previous hop: HHR requires cooperation from the previous hop. GIR can benefit from cooperation from the previous hop (e.g., if it sends all fragments to next hop without interleaving whenever possible) but does not require it.

Useful in other scenarios: GIR can be used in other scenarios where HHR is not applicable. For example, if a high-performance router fronts a LAN with a large number of low performance end nodes, it may be beneficial for the router to do GIR. This offloads the reassembly processing from the end nodes and allows them to receive larger datagrams, which can improve performance.

We will describe a model for assessing when GIR is useful in Section 4.

3.3. The FIFO assumption

To obtain efficient destination and intermediate reassembly, we have assumed that fragments will arrive in FIFO order in the expected case. How valid is this assumption? The only cases where FIFO would be violated are when the routing topology changes and network striping algorithms are used for load balancing. The first case is drastic and should be infrequent; whenever it happens, network traffic probably slows down anyway, because of routing loops. There are striping algorithms that preserve FIFO order. However, if striping algorithms in future IP networks routinely misorder packets, a number of other algorithms will also work inefficiently.

First, discovering MTU sizes will be much harder because packets routinely take different paths to the destination. Second, TCP Header prediction[Jac90] and UDP Header Prediction [PP93] will not work because they assume in-order delivery. Many other router and endnode caching optimizations rely on similar temporal locality. Thirdly, it will be hard to guarantee quality of service when datagrams are routed over multiple paths. Thus an optimization based on FIFO delivery of fragments seems reasonable.

4. Reconsidering Fragmentation

In the previous section, we showed that destination and intermediate reassembly could be done efficiently under the FIFO assumption, and we argued why the FIFO assumption was reasonable. But reassembly is useful only if we can show that fragmentation is useful! Most researchers agree that fragmentation is essential for interoperability; but if fragmentation can be avoided without performance loss, then there is no point optimizing reassembly. We begin this section by arguing why fragmentation (and notions like GIR) can improve performance; we end this section by showing that the disadvantages of fragmentation can be avoided or controlled.

4.1. How Fragmentation and GIR helps Performance

Fragmentation (together with reassembly) can allow the use of larger packets/segments by a bottleneck node. There are a number of fixed costs associated with processing a packet/segment including the cost of processing packet headers and interrupt overhead. In cases where the fixed costs restrict throughput, fragmentation and reassembly can improve performance by reducing the overall fixed costs for a transfer.

We can lump together the fixed packet processing costs at the source and destination into two categories: a) **Transport:** the Transport (e.g., TCP) cost to processing transport segments b) **Other:** The cost of Routing (e.g., IP), Data Link, and Interrupt service routines. If the end-system is compute bound and protocol processing is expensive, and we have a large data transfer, then it pays to reduce both Transport and other fixed costs. The fixed cost at each intermediate router is the cost of processing the routing header and the propagation delay on the next hop. In addition to fixed costs, there are variable costs that depend on packet length, for transmitting a packet on each intermediate link and for moving packet data across end-system and router busses.

Fragmentation can help in two orthogonal ways. Allowing the use of large transport segment sizes can reduce Transport costs (both number of headers processed as well as acks sent). Allowing the use of large MTU sizes on the first and last links, reduces the other costs. We start by demonstrating the first effect by the simple experiment referred to earlier, and then demonstrate the second effect by extrapolating some experiments with varying packet sizes. Then we describe a simple analysis that quantifies the advantage of using large packets.

4.1.1. How Super Fragmentation helps Performance. Super fragmentation is a technique by which we trade off small increase in IP processing due to fragmentation for larger reduction in TCP processing. This idea is suggested in [KM87] under the name "intentional fragmentation" and in [SGK+85]. We use large TCP segment sizes and let IP fragment them. When the cost of fragmenting and reassemby is less than the TCP processing required for a large number[7] of segments, this technique will provide increased throughput. This technique can be valuable in networks where there is very little loss. Later we will present dynamic segment sizing, which will keep throughput at reasonable values even when the loss rate is high.

As described in Section 3.1 our experiments on Ethernet between two Sun-4/300 series machines running the NetBSD operating system (using `ttcp`) verifies that throughput increases with fragmentation and reassembly until the cost of fragmentation and reassembly catches up with reduction in TCP processing. The important point is that throughput increases with super-fragmentation in both implementations. In both cases a segment size of around 16KBytes seems to give highest throughput.

While the performance increase we observed with super-fragmentation in this particular experiment is limited, we expect the gain in performance to be more pronounced as the link speed increases relative to the CPU speed.

4.1.2. How GIR helps Performance. Figure 5 shows a sample topology where GIR can help. Assume that the MTU size of the intermediate link, Y, is much less than X. By using GIR or HHR, the two end-nodes can send and receive packets of size X; these packets will then be fragmented by $R1$ and then reassembled at $R2$. When is this worth doing?

[7] In the current implementations TCP has to process as many segments as the number of fragments created with a large segment size.

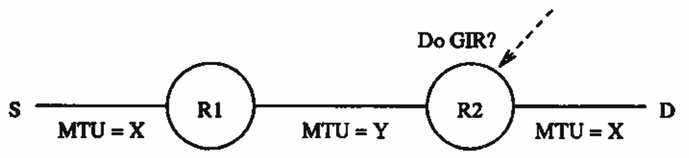

Figure 5: Two intermediate routers connected by a link with MTU size Y that is much smaller than the MTU size X of the first and last links. Doing GIR at $R2$ can help.

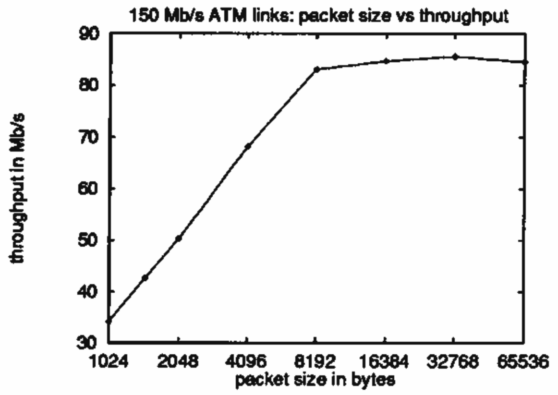

Figure 6: The throughput over an ATM 150 Mbs link as we vary the packet size used on the link. The receiving host is a Pentium machine.

Figure 6 shows the results of a simple experiment we conducted using `ttcp` to measure throughput between two Pentium PCs connected over an ATM 150Mb/s line. From the figure, as the packet size increases from 1024 to 8192, the performance increases by almost a factor of 3 from 30 Mbs to about 85 Mbs. Now suppose that in Figure 5, the first and last links are ATM links with MTU size 8192 and the middle link is an Ethernet with MTU size 1500. If we avoided fragmentation and used the smallest MTU size (i.e., 1500) we would expect to get no more than 42 Mb/s. If, however, $R2$ and $R1$ were sufficiently fast routers, we used an MTU of size 8192 on the first and last links, and $R2$ does GIR, we can hope for a throughput of 85 Mb/s.

Clearly, the improvement depends crucially on the endnodes being the bottlenecks and the routers being "fast enough". This seems like a reasonable trend for future routers, as routers are devoted to communication (and to other transfers) while endnodes should be doing other things besides communication. We note that in the two scenarios above, router $R1$ receives less packets when the source sends 8192 byte packets; this decrease should offset the increased cost for fragmentation. Router $R2$ receives the same number of packets in both cases, but does more work in the second case. If, as we have argued, GIR can be implemented efficiently in the expected case, this should not reduce throughput.

This improvement is also crucially dependent on fragmentation being implemented in the "fast-path" in high speed routers. Fragmentation is not a computationally expensive task, except for replication of headers in each fragment. Replication of headers can be aided by either software support for buffer reuse or by support from DMA engines.

Note also that the example given above could also be improved by using Hop-by-Hop Reassembly (HHR). However, GIR has the additional advantages cited at the end of Section 3.2. As a simple example, suppose that in Figure 5 router $R2$ had a link to another destination $D2$ with MTU size Y. Then GIR can distinguish between the two cases and not reassemble packets for $D2$ while continuing to reassemble packets for D; HHR cannot distinguish between these two cases.

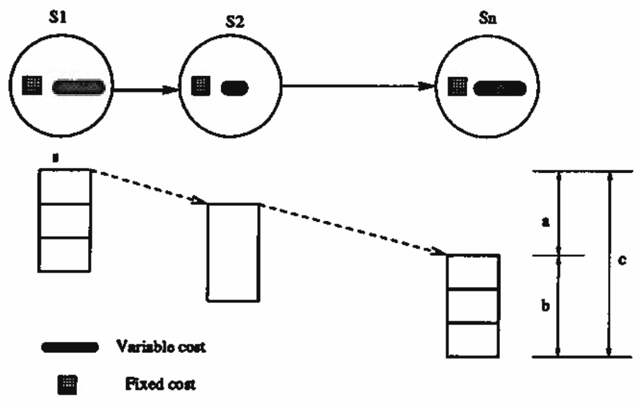

Figure 7: **The data transfer is modeled as a series of servers, each of which has two costs associated with it: a fixed cost for per packet processing and a variable cost proportional to the amount of data it handles. In the figure, the server S_2 is a high bandwidth server, and thus it has a very small variable cost. a is the latency or the time taken for the first bit to arrive at the destination; b is the time taken for the data transfer alone, and c is the total time spent in data transfer.**

4.1.3. When Larger Packet Sizes are Better: A Performance Model. In the last two subsections we have argued that allowing fragmentation can improve performance because it allows the use of larger packet sizes on the last link. This agrees with the findings of several researchers (e.g., [Mog92]). However, Stevens [Ste94] shows an example where using smaller packet sizes is better.[8] How can we reconcile these two views? The answer is that for short data transfers that are dominated by latency, short packet sizes allow better pipelining: for long data transfers that are dominated by throughput, large packets sizes can reduce fixed costs that improve overall throughput. We now present a simple performance model that reconciles these two extreme cases.

We can model the various stages in the path of packet from sender to receiver by a sequence of servers (see Figure 7). Let these servers be $s_{1..n}$ where the sender represented by server s_0 and the receiver represented by s_{n+1}. Every server has two stages: a fixed delay stage corresponding to the per packet processing cost independent of the packet length and a variable delay stage which is proportional to the packet length. Clearly, the fixed part models the protocol processing and propagation delays and the variable part models the data movement and data transmission costs. Let F_i represent the fixed delay through stage i, and let r_i represent the 'bandwidth' (the rate at which the data movers can move data through stage i) of stage i. Stage i uses a packet size of ℓ_i. Then, the delay of D bytes of data through stage i is

$$T_i(D) = F_i * D/\ell_i + (D + D/\ell_i * h)/r_i,$$

where h is the length of the per packet header and $D + (D/\ell_i) * h$ represents the total number of bytes processed at i.

The time taken for the entire data transfer from stage s_0 to s_{n+1} is:

$$\begin{aligned} T_{transfer} &= \textit{time for first bit of first segment to reach } s_n \\ &\quad + \textit{time time for the data to get through the slowest stage.} \\ &\leq \textstyle\sum_{i=1}^{i=n} \max_i T_i(\ell_i) + \max_{1 \leq i \leq n} T_i(D), \end{aligned}$$

The first term in this equation represents the latency for the first bit of first segment of data to get to the receiver, and the second term represents the delay due to throughput limitations of the path from sender to receiver. The first term is actually quite complicated and depends on whether

[8] Stevens attributes this observation to Bellovin.

reassembly is being done at intermediate stages; however it is always bounded by $\max_i T_i(\ell_i)$ at each stage.[9]

The second term is limited by the bottleneck stage (the slowest stage). As we change the segment size, two things happen. With smaller segment sizes, the fixed component of $T_i(D)$ increases since it depends on the number of fragments processed, but the initial latency (time taken by the first bit of the first segment to reach the receiver) reduces. With larger segment sizes, there is larger initial latency, but the number of packets processed at the servers is small and thus we reduce the throughput term. Various examples in the following sections illustrate which term dominates under specific constraints and recommend the appropriate segment sizes for such constraints.

Small Data Transfers:. For small data transfers the latency term dominates. The magnitude of this term depends on the size of the first segment. Assuming we are using a constant segment size throughout, as the segment size increases, overall time for transfer increases. This is illustrated in the following example taken from [Ste94]. We send 8192 bytes from a sender to a receiver, going through two intermediate routers. Assume constant segment sizes throughout the network; assume also that all links are T1 telephone links (1,544,000 bits/sec). Overall transfer time using 4096 byte packets is $2 * ((4096 + 40) * 8/1544000) + (8192 + 2 * 40) * 8/1544000 = 85.72ms$. Overall transfer time using 512 byte packets is $2 * ((512 + 40) * 8/1544000) + (8192 + 16 * 40) * 8/1544000 = 51.48ms$. In general, it is better to use smaller segment sizes for small transfers.

Super fragmentation:. Super fragmentation increases IP processing, trading it off for reduced TCP processing. Assume that the throughput bottleneck is the receiver node; we analyze processing costs at the receiver. Let T_{TCP} be the time taken for a packet to go through the TCP processing, and let T_{IP} be the corresponding time for a packet to get through IP processing. The total processing time (fixed cost) required for TCP and IP together is $T_{TCP} * D/\ell_{TCP} + T_{IP} * D/\ell_{IP}$, where ℓ_{TCP} is the amount of user data TCP passes to IP as one datagram, and ℓ_{IP} is the amount of data IP can pass to the interface driver as one packet. (Here we are ignoring the effects of added headers. Therefore we can assume that the variable costs are identical with and without fragmentation.)

Note that typically, $T_{IP} < T_{TCP}$. By enforcing $\ell_{TCP} = \ell_{IP}$, as the TCP implementation on NetBSD and many others do, we subject each small packet that goes out on the device interface to the entire TCP and IP processing. Super-fragmentation enables us to keep ℓ_{TCP} much larger than ℓ_{IP}, thus resulting in smaller TCP processing at the cost of slightly higher IP processing. Until $T_{TCP} * D/\ell_{TCP} = T_{IP} * D/\ell_{IP}$, we should get better throughput. For larger values of ℓ_{TCP}, the increased IP processing (due to fragmentation and reassembly) will become the bottleneck and the throughput will reduce.

Default MTU, Minimum MTU along the path *vs* Graceful Reassembly:. Currently TCP/IP implementations use a default small sized segment (536 bytes, corresponding to an MTU of 576 bytes) when a connection is non-local. This avoids fragmentation at the cost of processing a large number of small packets. Assume that the sender and receiver are compute bound and that the throughput bottleneck is at the receiver. Using path discovery or RFC 1191 techniques would limit the segment size used at the receiver link to $l_n = \min_i MTU_i$, where MTU_i is the MTU of link i. Using GIR can allow the segment size used at the receiver link to grow as large as $l'_n = \min(MTU_1, MTU_n)$ (i.e., the smaller of the MTU's of the first and last links.) If the data transfer is large enough and the throughput is dominated by $T_n(D)$ and by the fixed costs at the receiver, then the throughput can increase by a factor of l'_i/l_i.

[9] We are assuming that $D > \max_i l_i$. It is easy to modify the latency term if D is smaller.

To summarize, if a small amount of data is to be transferred over a large number of hops, then small packet sizes offer smaller latencies. If the amount of data to be transferred exceeds a critical value, and the data transfer rate is limited by fixed processing costs, then large packet sizes are better. In the latter case, super-fragmentation allows a reduction in transport processing by increasing the segment size; also GIR can potentially reduce endnode routing and driver costs by allowing the use of large packet sizes on the first and last links.

4.2. Dynamic Segment Sizing

The use of super-fragmentation can cause serious instability in the case of lossy links [KM87]. We have observed this by a simple experiment in which we did super-fragmentation and also dropped each fragment with probability p. We found, hardly surprisingly, that the throughput dropped down to 0 for small values of p. For example, using $16K$ segment sizes and $p = 0.1$ can result in essentially zero throughput (almost every segment is being lost and retransmitted because there are roughly 10 fragments per segment). Actually, the reduction in throughput occurred for smaller values of p, though we did not precisely determine the knee of the curve.

Of course the problem arises because the IP does not retransmit packets (IP does, after all, provide a datagram service). Thus transport is forced to retransmit large segments, leading to livelock. In order to guard against this, we suggest that the Transport segment size should be dynamically adjusted. A wide range of dynamic segment sizing schemes are possible, based on how segment sizes are increased and decreased, as well as when they are done. [KM87] suggests such a scheme (detecting the problem using increased retransmission rates) but states that it will not work well unless coupled with a congestion control scheme. At the time [KM87] was written, congestion control [Jac88] was not implemented in TCP.

There is a considerable analogy between dynamic segment sizing and congestion control algorithms. Both algorithms are characterized by an Increase Policy (when and how to increase) and a Decrease policy (when and how to decrease). For example, a congestion control scheme may increase its window size (or rate) when it has not observed any retransmissions for time T, and may increase the window size by a constant; it may also decrease its window, whenever a retransmission is done, and may decrease the window size by halving the window [Jac88, RJ88].

The RFC 1191 [MD81] for path MTU also provides a dynamic segment sizing policy by beginning with the MTU of the first link, and decreasing the MTU (using, for example, a table of common packet sizes) whenever an ICMP message is received. RFC 1191 also recommends an increase policy of increasing the segment size after 10 minutes.

Thus rather than invent another set of policies and heuristics we can leverage these existing schemes. For the wide area the RFC 1191 scheme seems suitable. However, whereas RFC 1191 schemes decrease the segment size after receiving an ICMP message, we decrease the segment size after a transport timeout (or timeouts). However, note that decreasing the segment size further reduces the transport window (since congestion control also decreases the number of segments in the window). Thus it may be better to scale up the congestion window to compensate for the reduction in segment size.

For the local area, we implemented a simple scheme that is similar but slightly different from the existing schemes. In our scheme, the segment size increase policy is quite aggressive: it is done when a successful acknowledgment is received. Segment sizes are increased in multiples of the default segment size for the interface. We decrease the segment size by half (with the minimum anchored at the default minimum segment size) when a retransmission timeout occurs. This can also be varied by waiting for more than k retransmissions.

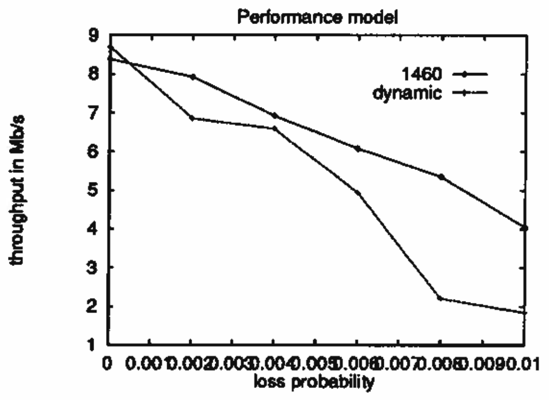

Figure 8: Variation in throughput with error rate for two schemes: ordinary TCP without dynamic segment sizing and a TCP that does super fragmentation and dynamic segment sizing. The performance of a TCP that does super fragmentation *without* dynamic segment sizing is much worse, and is not shown.

The results, for transfers between two Sun–4/300 series workstations on an Ethernet, are shown in Figure 8. We compare our dynamic segment sizing algorithm to ordinary TCP (whose segment size remains fixed); clearly we would expect ordinary TCP to do better as the error rate increases. We can see that dynamic segment sizing offers reasonable throughput (about 2 Mbs versus 4 Mbps for ordinary TCP) when the loss rate is 0.01. Clearly, we can make the difference between the two curves much smaller by employing much more conservative increase policies. For example, we could increase the segment size only if c acks are successfully received; we can increase c if errors continue and reduce c at a much more conservative rate. For a large transfer, this should provide performance that tracks the behavior of ordinary TCP because it will eventually result in using the default TCP size. We are currently experimenting with other variations of dynamic segment sizing.

We note that dynamic segment sizing is not just useful as a guard against fragmentation loss: *dynamic segment sizing is also a guard against a drop in performance in other lossy environments.* For example, consider a TCP connection over a telephone line that has a high bit error rate b. If any bit in a TCP segment is corrupted, the segment will be discarded (hopefully) by checksums at the Data Link or Transport level. If the bit error rate is as high as 0.0005 and we use an MTU of size 296 bytes (roughly 2400 bits), then almost every segment will be lost and throughput will go down to zero. A dynamic segment sizing algorithm that tries lower segment sizes may achieve some goodput even in such a lossy environment.

Notice the analogy between bit loss and fragment loss: they both lead to the loss of the corresponding segment. This effect is more pronounced when we have a uniform bit error rate model; correlated and burst errors will reduce this effect somewhat. Congestion loss can also fall into this category. Consider an endnode sending large segments of size 1500 through a router that is so congested that it never has spare buffers of size larger than 500 bytes. Reducing the window size to 1 will not help improve goodput, but reduction of segment size to 500 might. Thus segment size reduction may be a further fallback mechanism for congestion control when losses are heavy and the window size is already reduced to 1.

4.3. Avoiding other Disadvantages of Fragmentation

Dynamic Segment Sizing helps avoid or lessen the major disadvantage of fragmentation pointed out in [KM87] — i.e., poor performance under loss. We describe techniques to alleviate the other disadvantages of fragmentation.

Poor Choice of Fragment Size:. [KM87] points out that a packet of size $x + \epsilon$ that is to be fragmented over a link of size x can cause roughly twice the optimal number of fragments for small ϵ. Thus it is necessary to have the source intelligently choose the segment size. A simple heuristic is to avoid using a segment size that is only a small amount over a possible MTU size. For instance, when two MTU sizes are close to each other (e.g., 1492 for 802.3 and 1500 for Ethernet) we use the lower value. This idea is incorporated into the "plateau table" in RFC 1191. The same idea, however, can be used with Dynamic Segment Sizing (especially for off-LAN traffic) to avoid suboptimal fragmentation in most cases.

Reassembly Deadlocks/Livelocks:. [KM87] points out that it is hard to have a reassembly algorithm that guarantees progress even if fragments are being delivered at some rate because buffers could fill up with partially reassembled fragments. Even though buffers are released after a time period, this could keep recurring, resulting in livelocks.

However, if Dynamic Segment Sizing is used, livelocks cannot persist as the application will begin (after loss is detected) to use segment sizes that are small enough to avoid fragmentation. This, together with the timed release of geriatric fragment buffers, can avoid livelocks. In practice, a useful heuristic is to keep separate buffering for complete packets and fragments that have arrived in FIFO order, and separate buffering for fragments that have arrived out-of-order.[10] If we always reserve some buffering for fragments that arrive in-order, then most fragments will avoid even temporary deadlock/livelock problems, if the mostly-FIFO assumption is correct.

5. Discussion

If history is any indicator of the future, every network technology of the future will have a different maximum packet size. Interoperating among these diverse technologies is a challenge at which IP has been very successful, using fragmentation and reassembly and other techniques. Recent trends in the IP community have suggested [KM87] removing or limiting the need for IP fragmentation and reassembly by discovering the smallest MTU size. While these mechanisms are useful they also have problems: they lead to extra round trip delays, require extra mechanisms, are not guaranteed to work (e.g., when paths change), and do not minimize the number of packets sent.

Contributions:. We have shown, using a combination of measurement and analysis, that there are situations when fragmentation can improve performance because it reduces the number of packets/segments that must be processed for a large data transfer. We discussed the notion of super-fragmentation and showed that it can improve performance (at least for local communication) because of reduced transport protocol processing. Moreover, we have implemented an optimized reassembly algorithm in NetBSD that takes 38 instructions per fragment in the expected case and shown that it performs well. We also implemented dynamic segment sizing to adjust the transport segment size upon packet loss. We demonstrated, by experiment, that this allows throughput to stay reasonable when the fragment loss rate is high, while allowing good performance when the loss rate is low.

Hop-by-hop reassembly (HHR) is sometimes advocated across small MTU links so that larger effective MTU sizes can be advertised. We have introduced a new form of reassembly known as Graceful Intermediate Reassembly (GIR) and shown that GIR can always perform at least as well as HHR, and can often perform better because it can discriminate between packets to destinations

[10] If a fragment is received out of order for a datagram that was in the in-order list, we can reclaim any (e.g., the oldest) out-of-order buffer to obtain space for future in-order fragments.

that can benefit from reassembly and those that cannot. GIR also has the unique property of coalescing fragments gracefully into larger size fragments when the outgoing MTU size does not permit complete reassembly; this avoids useless work and reduces latency. Using the expected FIFO reassembly optimization, GIR can be implemented in routers at low cost. We have argued that there are situations where GIR can improve performance by allowing the use of a larger packet size on most links in the data path.

We have the following specific recommendations for TCP/IP. We believe super-fragmentation (along with efficient reassembly and dynamic segment sizing as a safeguard) is worth investigating for improved performance, at least for local traffic. Applications that wish to should be able to request smaller segment sizes in order to reduce latency (see Section 4.1.3). We recommend that the IP version 6 specification not encourage all routers to implement fragmentation and reassembly in a low performance path. If the fragmentation and reassembly header (which is fixed size) is part of the required header (as opposed to an optional header), then some routers could choose to do high-performance fragmentation and graceful reassembly.

Relation to Other Approaches and Other Protocols:. Our results are complementary to and build upon the analysis and proposals in [KM87]. All the disadvantages of fragmentation outlined in [KM87] remain valid; we have proposed additional mechanisms (e.g., dynamic segment sizing, separate reassembly buffers for in-order fragment reassembly) to combat these problems. We agree with [KM87], that fragmentation without these mechanisms would still be "considered harmful". Our dynamic segment sizing algorithms are an elaboration of an idea in [KM87] and influenced by the dynamic MTU discovery algorithm in RFC 1191. However, we believe we have gone further in: a) implementing and measuring an optimized reassembly algorithm; b) showing the possible performance gains of fragmentation (especially super-fragmentation), c) introducing the new notion of graceful intermediate reassembly, and d) elaborating on and measuring the effects of dynamic segment sizing.

The three ideas advocated in this paper: optimized reassembly based on FIFO fragment arrivals, Graceful Intermediate Reassembly, and Dynamic Segment Sizing are all orthogonal and can be applied either separately or together, and to other protocol suites. The OSI protocol suite [Per92] uses a very similar fragmentation scheme and thus the first two ideas apply directly. However, the OSI Transport Protocol, TP-4 [osi], does not allow dynamic segment sizing because TP-4 numbers segments and not bytes as in TCP. Thus, without a change to the TP-4 mechanisms, segment size changes are not possible in TP-4 without setting up a new connection. In the context of fragmentation, this appears to be a significant advantage of TCP over TP-4.

Future Work:. There are several directions we wish to explore. The interactions between dynamic segment sizing and congestion control (especially for non-local traffic) need to be studied carefully before super-fragmentation can be safely recommended. Since TCP windows are based on the number of segments, reducing the segment size further affects the pipe size. It is possible to do dynamic segment sizing while leaving the TCP window sizes at the levels they would have been. The dynamics of these myriad variations merit further study.

We would like to implement GIR reassembly for a router and measure performance; much of the code is already written. Finally, the ideal solution to the reassembly problem would be an efficient reassembly algorithm that scales according to the degree of non-FIFOness. Our solution of coalescing all FIFO fragments received so far is a step in this direction, but does not go far enough. We believe that we can devise schemes that do so.

References

[DKS89] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *Proceedings of the Sigcomm '89 Symposium on Communications Archtectures and Protocols*, 19(4):1–12, September 1989.

[Fel93] D. C. Feldmeier. A Data Labelling Technique for High-Performance Protocol Processing and Its Consequences. *Computer Communications Review (SIGCOMM '93)*, 23(4):170–181, October 1993.

[Hin94] R. Hinden. Editor, Internet Protocol, Version 6 (IPv6) Specification. *Draft*, October 1994.

[Jac88] Van Jacobson. Congestion Avoidance and Control. *Computer Communications Review — Proceedings of SIGCOMM'88)*, 18(4), August 1988.

[Jac90] Van Jacobson. 4.3BSD TCP Header Prediction. *Computer Communications Review*, 20(2):13–15, April 1990.

[KM87] C. A. Kent and J. C. Mogul. Fragmentation Considered Harmful. *Computer Communications Review — Proceedings of SIGCOMM'87)*, 17(5):390–401, August 1987.

[MD81] J. C. Mogul and S. E. Deering. Path mtu discovery. Technical Report RFC 791, September 1981.

[Mog92] J. C. Mogul. *IP Network Performance, in Internet System Handbook, eds. D.C. Lynch and M.T. Rose.* Addison-Wesley, Reading,MA, 1992.

[Mog95] J. C. Mogul. The Case for Persistent HTTP Connections. In *In Proc. SIGCOMM '95*, pages 299–313, September 1995.

[osi] ISO 8073 – Information Processing Systems – Open Systems Interconnection – Transport Protocol Specification. Technical report.

[Per92] R. Perlman. *Interconnections: Bridges and Routers.* Addison-Wesley, Reading,MA, 1992.

[Pos81] J. B. Postel. Internet protocol. Technical Report RFC 791, Information Sciences Institute, September 1981.

[PP93] C. Partridge and S. Pink. A Faster UDP. *IEEE/ACM Tran. on Networking*, 1(4), August 1993.

[RJ88] K. K. Ramakrishnan and R. Jain. A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer. *Computer Communications Review — Proceedings of SIGCOMM'88)*, 18(4), August 1988.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. *Proc. Summer USENIX Conference*, pages 119–130, June 1985.

[Ste94] R. Stevens. *TCP/IP Illustrated, Volume 2.* Addison-Wesley, Reading,MA, 1994.

A. Finding Min MTU sizes using Routing Algorithms

We give a quick example of how to modify an existing routing protocol to calculate the min MTU size on the path to a destination. Many routing protocols (e.g., RIP) are based on the Bellman-Ford algorithm [Per92]. The modifications required for Bellman-Ford are as follows: Normally each router sends its shortest cost to each destination D to all its neighbors. Router R calculates its shortest path to D by taking the minimum of the cost to D through each neighbor. The cost through a neighbor N is just N's cost to D plus the cost from R to N. We modify the basic protocol so that *each neighbor reports its min MTU to a destination in addition to its cost to the destination.* Then each router calculates its min MTU to destination D as the smaller of the min MTU of the minimal cost neighbor N for that destination, and the MTU of the link from R to N.

Any link state routing protocol [Per92] can be similarly modified by having each router add the MTU of each outgoing link to the LSP. Then a simple traversal of the min cost Dijkstra tree can obtain the Min MTU sizes to all destinations in linear time.

Of course, this does not work well across hierarchical boundaries because each area cannot accurately summarize the Min MTU of all paths within the area for the purposes of area routing. Nevertheless, since most traffic is local this mechanisms may be preferable to the Path MTU discovery schemes which take additional round-trip delays to obtain information. The disadvantage of this scheme, however, is that it requires all routers to implement this modification which is infeasible in a large Internet owned and managed by disparate organizations.