-- --

CITI Technical Report 93–8

The Rx Hex

D. Bachmann
bachmann@austin.ibm.com

P. Honeyman
honey@citi.umich.edu

L.B. Huston
lhuston@citi.umich.edu

ABSTRACT

At CITI, we run dataless AFS clients over dialup IP networks. Improving Rx performance is critical to that task. In this paper, we report on our progress in adapting Rx to networks characterized by low bandwidth, high delay, or variable round-trip time. Our focus is on adding facilities for congestion avoidance and control, and on compressing Rx headers. Although our work is ongoing, we have overcome several hurdles, and are now getting impressive data transfer rates over ordinary dialup lines.

November 2, 1993

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

-- --

The Rx Hex

D. Bachmann
bachmann@austin.ibm.com

P. Honeyman
honey@citi.umich.edu

L.B. Huston
lhuston@citi.umich.edu

1. Introduction

AFS [1] was designed with high-speed networking and continuous connectivity as a given, and this bias is reflected in the AFS transport layer, a UDP-based [2] remote procedure call (RPC) package, called Rx [3, 4].

Rx is a general-purpose remote procedure call (RPC) package used by AFS and its utilities. Rx accommodates the special needs of distributed file systems by offering a streaming interface to the service layer. This lets AFS transfer as much data as it wants in a single call; the amount of data in a single RPC can be over a gigabyte.

Rx uses a windowing strategy with packet-level acknowledgements. This avoids the ‘‘send-and-wait’’ behavior typical of Sun’s Open Network Computing Remote Procedure Call (ONC RPC) [5]: because ONC RPC limits the size of any RPC request, large files must be transferred in multiple, non-overlapping calls.

Because Rx is based on UDP, not TCP [6], it must provide its own mechanisms for flow-control, selective retransmit, and authentication. Necessarily, Rx has had to reinvent some of the mechanisms previously developed for TCP. As we shall see, in a number of ways, Rx did not copy TCP closely enough.

Like much in the UNIX† programming environment, the most accurate specification of Rx is that defined by the source code. Perusing source code, it’s clear that Rx is intended to accommodate the vagaries inherent in internetworking, such as networks with varying packet sizes and capacity. Yet the implementation is extremely rigid in its ability to cope with variability. In particular, Rx assigns constants to parameters that should be dynamically adjusted.

† UNIX is a Trademark of AT&T Bell Laboratories.

1.1. Frame size

Rx assumes a 1.5 KB maximum transmission unit (MTU). If the MTU is smaller than 1.5 KB, Rx implicitly employs IP fragmentation [7]. Yet, IP fragmentation is widely held to be inimical to robust internetworking [8].

1.2. Retransmission timeout

In AFS 3.1, the retransmission timeout (RTO) in Rx is set to two seconds in the source code. If not acknowledged within that interval, Rx assumes a packet is lost and retransmits it. Every fourth packet is tagged with the ‘‘please ACK’’ bit, so a clump of four packets has to transfer in less than two seconds or else the ACK is late and the whole clump is retransmitted. This process recapitulates the problems of IP fragmentation above the transport layer!

In our tests, we use a 9.6 Kbps SLIP [9] connection with a 576 byte MTU, so a single packet takes 0.6 seconds to traverse the network. Two seconds is not enough time for four packets to move across the SLIP link; without modifications to Rx, the RTO timer constantly fires prematurely, retransmissions are rampant, and throughput is abysmal, as we shall see.

1.3. Window size

Rx uses a 15 packet window with no support for adjusting the window in the face of congestion. Combined with the fixed retransmission timeout, this results in disaster.

With a 576 byte Rx packet, it takes over 9 seconds for a 15 packet window to traverse the SLIP network. (Our SLIP network uses RS-232 character framing, so a single byte requires ten bit times to transmit.) Nine seconds is more than four RTOs, so the last packet in the first window is retransmitted by the sender four times before it first arrives at the receiver!

Worse yet, with a 1.5 KB packet, a full window spends almost 24 seconds in transit, and exhausts the serial line buffers on our SLIP gateway.

2. Experimental Setup

To design and test our modifications of Rx, we configured three identical UNIX workstations as shown in Figure 1 to emulate a congested network.

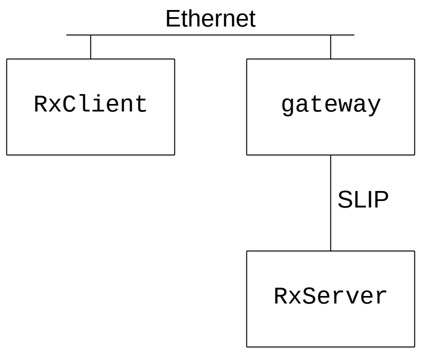

Figure 1. Experimental test bed. The three computers shown are IBM RT workstations running the AOS 4.3 BSD UNIX operating system. `RxClient` and `RxServer` are test programs that push Rx packets from the client to the server. At the same time, the client produces voluminous per-packet diagnostics, which we massage into the figures shown in the next sections.

The workstations are identical IBM RT workstations running 4.3 BSD and up-to-date networking software, including `cslipbeta` from Lawrence Berkeley Labs. The basic experiment consists of attempting to move 100 KB of data from the server to the client via a path that includes a 9.6 Kbps SLIP connection. To separate Rx performance from disk performance, the experiments were carried out using a pair of programs that talk directly to the Rx interface, `RxClient` and `RxServer`, a pair of diagnostic applications provided with AFS. To control variation in throughput due to the operating system, the machines were run in single-user mode.

In the experiments that follow, we vary Rx parameters, such as packet size and RTO, and process the output from `RxClient` to extract quantitative information.

3. IP Fragmentation

As shipped, Rx has a static packet size, 1.5 KB, and a constant window size, 15 packets. With every fourth packet, the sender requests an acknowledgement packet from the receiver; this packet acknowledges the sequence number of the packet that requested the ack, as well as all prior sequence numbers when things are going well. If a packet is not acknowledged before the two-second timeout expires, the sender retransmits. It's easy to see that any network not capable of delivering 90 Kbps will experience many pointless retransmissions.

Figure 2 shows what happens in a Rx session over a 9.6 Kbps network. The chart shows packet sequence number *vs.* the time the packet was sent; multiple dots on the same ordinate represent retransmissions. Although 70 packets are required to complete the transfer, the server never gets past packet 28. Obviously, something is very wrong here. Under these conditions, we found AFS service so poor that it was impossible to accomplish any work: even fetching a single file was precluded.

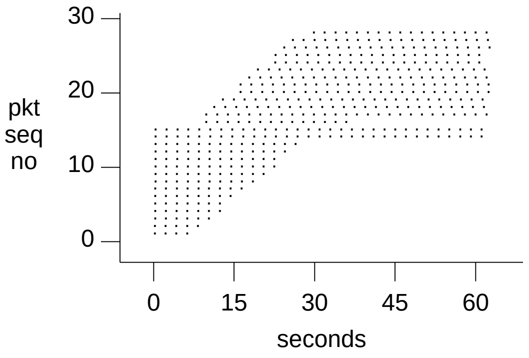

Figure 2. MTU = 1.5 KB, RTO = 2. This figure shows the ''two steps forward, one step back'' pattern, characteristic of congested networks with a broken packet retransmission strategy. (In actuality, it's more like ''15 steps forward, 14 steps back.'') In this case, after attempting over 1,400 times to get the first 28 packets through, the network is so clogged with retransmitted packets that the Rx connection times out.

This is the first problem encountered by Rx in our environment. For good interactive performance, SLIP links are usually set up with an MTU of 296 bytes [10]. A combined Rx/UDP/IP header size is 56-bytes (see Section 8), which is almost 19% of a 296-byte frame. We place a somewhat higher priority on file system throughput, so to we set our MTU to 576 bytes, a value commonly chosen for trans-Internet packets, and we modified Rx to assume this smaller packet size if the peer is on a different subnet, as is the case for SLIP peers.

With this change, Rx is able to get all 193 packets across the link, but it isn't pretty. Figure 3 plots packet sequence number against time after modifying Rx to avoid IP fragmentation.

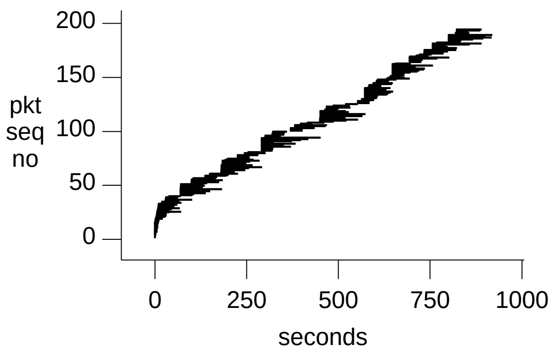

Figure 3. MTU = 576, RTO = 2. Of 4,798 packets transmitted, 4,605 (96%) were retransmissions. The average number of retransmissions was 25; packet number 93 was retransmitted 80 times. The total throughput for the 100 KB file was 873 bps, which is 11% of the peak rate.

Setting the packet size to the MTU of the path does more than eliminate IP fragmentation, it also reduces the total round-trip time (RTT): one window's worth of 1.5 KB packets takes almost three times as long to send as a window's worth of 576-byte packets, 24 seconds *vs.* 9 seconds. 9 seconds is much closer to the two second RTO than is 24 seconds. Adjusting the fixed RTO to 6 seconds (slightly less than the 9 second RTT) and 12 seconds (slightly greater than the 9 second RTT), we get the results shown in Figures 4 and 5, respectively.

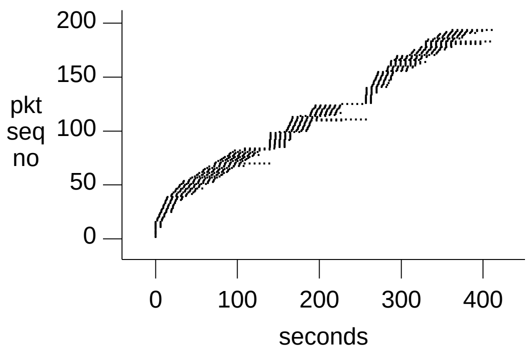

Figure 4. MTU = 576, RTO = 6. RTO is set to 6 seconds, slightly less than the 9 second RTT. While less extreme than Figure 3, the retransmission patterns are still observable here. The parallel threads of retransmissions, spaced 6 seconds apart, are visible, as is the occasional dropped packet, leading to a horizontal string of retransmissions bogging things down until a packet finally gets through and frees up a new window-full (the vertical bursts). The throughput in this experiment is 1.9 Kbps, 25% of the available bandwidth, after taking header overhead into account.

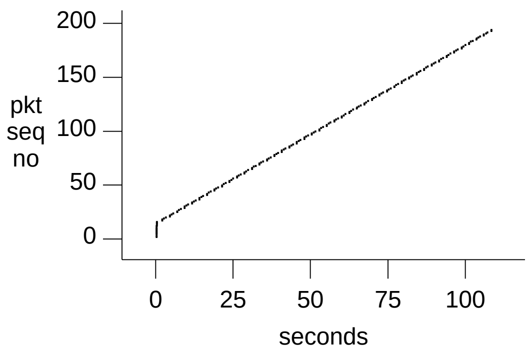

Figure 5. MTU = 576, RTO = 12. The 12 second RTO is slightly more than the 9 second RTT. Here we have achieved our goal of transmitting each packet exactly once. The initial window is evident as a vertical line on the left side, followed by smooth, self-clocking transfer of packets, uninterrupted by retransmissions. The throughput for this experiment is 7.4 Kbps, 96% of the available bandwidth, after taking header overhead into account.

Figure 5 proves that the RTO can be adjusted by hand to achieve good performance. But the parameters are very sensitive to prevailing network conditions. For example, if another transfer starts up in the middle, we are again overwhelmed by retransmissions, as shown in Figure 6.

So while the 12 second RTO works when a single session has the entire bandwidth to itself, it is unstable, and a bit of resource competition is enough to throw the network into a stable, congested state. To make Rx work for a single session, we can tune it for a specific link speed. But to run Rx over a connection that involves greater delay, *e.g.*, a satellite link or a 2.4 Kbps link, we are forced to fiddle with the RTO again. The solution to this problem is to teach Rx to adapt to the speed of the environment it encounters.

4. Congestion Control

Our strategy for improving performance in the face of congestion is to use careful measurement of packet round-trip times (RTT) to adjust the retransmit timeout (RTO). Following Jacobson's approach [11], we also estimate the variance of the RTT by calculating an estimator of the mean deviation of the RTT, and use this to adjust the RTO timer.

In the SLIP environment, the slow serial line contributes most to RTT. Because the SLIP line is dedicated, it has stable delay characteristics and the deviation of RTT is small. When the network

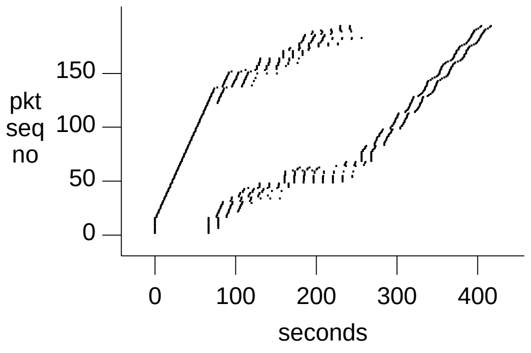

Figure 6. MTU = 576, RTO = 12. The second transfer begins at 66 seconds with a full window of packets, which increases the RTT seen by subsequent packets to 18 seconds, due to additional queueing delay at the gateway. Consequently, both sessions start retransmitting packets, as they are expecting acks within 12 seconds. The increase in the number of packets in circulation leads to a further increase in RTT and yet more retransmissions. At this point, we are seeing the now familiar behavior shown in Figure 4. Even after the first session finishes, the second session remains in a permanently congested state. The extra retransmissions lead to two windows, *i.e.*, 30 packets, in circulation, with a resulting RTT of 18 seconds. Each packet is retransmitted once (at 12 seconds) before it is acknowledged (at 18 seconds), continuing the cycle.

bottleneck is eliminated in an Ethernet workplace, other components become principal contributers to RTT delay, and the RTT deviation varies much more widely. Processing time on the AFS server becomes the major factor in RTT, and load variation on the server may have the effect of increasing RTT variance. To study these effects, our remaining experiments focus on transfers between a client and server running on multiprogrammed computers on the same Ethernet.

As Jacobson recommends, we modified Rx to keep estimators for the RTT and its mean deviation (MDEV). We then experimented with setting the RTO to the sum of the estimated RTT and the mean deviation, with results depicted in Figure 7.

While this is an improvement, it's apparent that we are not reacting quickly enough to congestion at the server. Almost two seconds into the experiment, RTT rises suddenly. Because the Mdev component does not play a large enough role in RTO calculation, service delay is accommodated as increased RTT; this condition is stable, resulting in the remainder of the packets being transmitted twice. Doubling the MDEV component in the RTO eliminates the stable congested state, with the results shown in Figure 8.

The excess retransmissions here are due to our

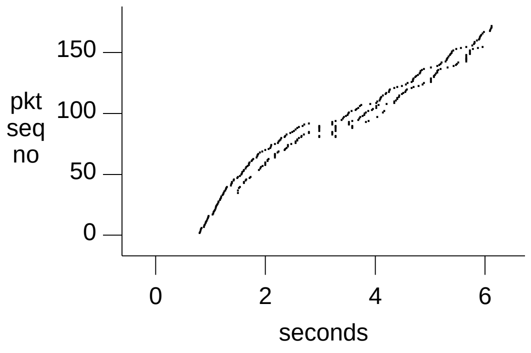

Figure 7. RTO = RTT + MDEV. While the situation is much improved over the static SLIP tests shown earlier, in this Ethernet experiment, we are not being aggressive enough in dealing with variations in delay. The parallel tracks evident in the figure indicate a stable congested state in which all packets are transmitted twice.

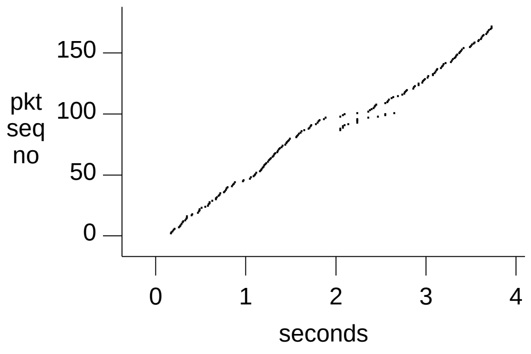

Figure 8. RTO = RTT + 2 × MDEV. Here the response to congestion is much improved over the previous figure.

incorrect assumption that packet RTTs are independent. If one packet is delayed, it is almost certain that the succeeding packets will also be delayed. Thus we modified Rx so that the RTO timers of outstanding packets were updated after every acknowledgement; if an ACK is delayed, we assume that ACKs for other outstanding packets will be similarly delayed. Figure 9 shows that we are approaching our goal of avoiding retransmissions.

In the next section, we consider the response to rapidly rising RTT, usually due to congestion.

5. To Karn or not to Karn

TCP ACK does not offer a way to tell whether the first or last (or other) instance of a retransmitted packet is being acknowledged. It is therefore impossible to determine the RTT from an ACK if the packet was retransmitted. Karn's algorithm [12] simply ignores the potential RTT information from such ACKs.

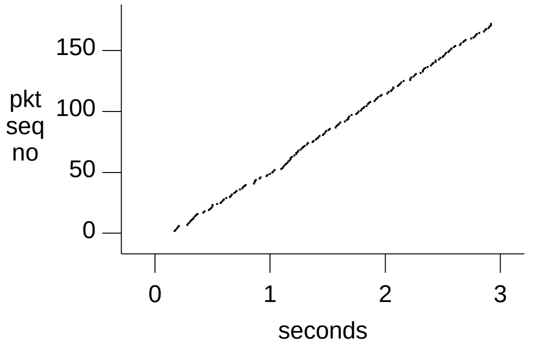

Figure 9. Timers updated after experiencing delay. In this experiment, the RTOs for outstanding packets are updated to reflect the delay experienced by earlier packets. The changing slope of the curve indicates that packets experienced unexpected delay at about the one second mark, yet Rx finally seems to be behaving well in the face of changing network conditions.

This behavior, while correct, is unfortunate. When an ACK is delayed because of congestion, its RTT is of critical importance for adjusting the RTO, otherwise unnecessary retransmissions ensue. Figure 10 shows the effect of ignoring RTT information in ACKs of retransmitted packets.

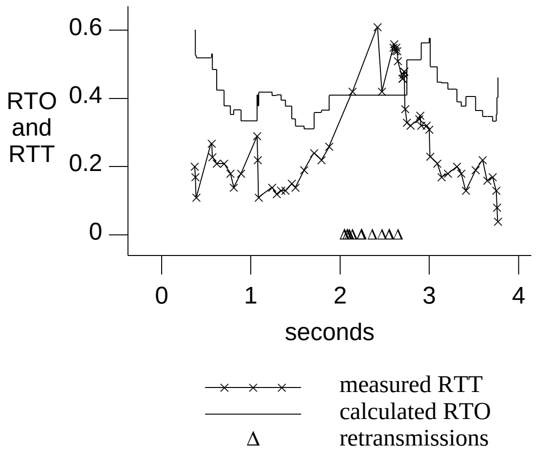

Figure 10. Using Karn's algorithm. If RTT exceeds RTO, acknowledgements time out and retransmissions begin. Using Karn's algorithm, we discard RTT information for retransmitted packets. Consequently, RTO is not updated until RTT falls below RTO and we begin getting RTT information for packets that were not retransmitted.

Two seconds into the experiment, RTT rose rapidly, causing timers to expire as soon as RTT exceeded the current RTO, at which point a flurry of retransmissions ensued. Because Karn's algorithm doesn't allow the new RTT information (the ×'s) to be used, neither the estimated RTT nor the estimated MDEV could be updated. This is evident in the graph where the calculated RTO remains constant for a long time despite many ACKs and associated RTT samples. During this time retransmissions are rampant. With Karn's algorithm (absent exponential backoff for RTO), the condition is stable: until RTT falls below RTO, retransmissions are frequent and inevitable.

Unlike TCP, Rx ACKs for retransmitted packets can be distinguished by their serial number. We modified the Rx sender to remember the send times of the first and last transmission of a packet. This makes RTT information from delayed packets available in most cases, so that Rx can react quickly to congestion as new RTT information arrives, as shown in Figure 11.

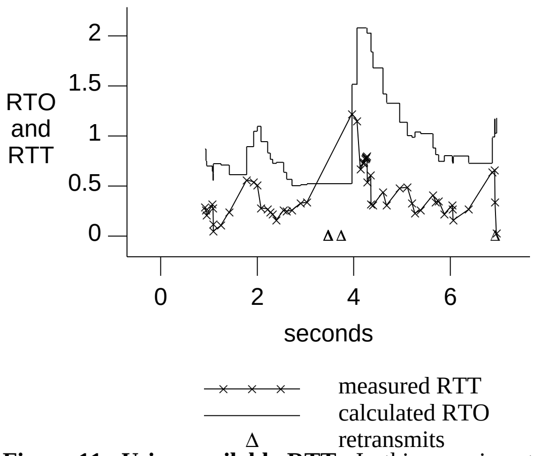

Figure 11. Using available RTT. In this experiment, we update the RTO, using the RTT available in retransmitted packets. When RTT exceeds RTO, packets are retransmitted. However, the increase in RTT, and especially in MDEV, allows Rx to adapt almost immediately with an increase in RTO.

Three seconds into this experiment, the RTT rose rapidly. In the absence of any further ACKs, two packets were retransmitted as their RTO (still set at 0.5 seconds from the last ACK) expired. When an ACK finally came in at four seconds the new RTT information was used to push the RTO to 1.5 seconds, eliminating further retransmissions.

6. Congestion Control Summary

By manually adjusting the constant RTO of the off-the-shelf AFS to 12 seconds (from two), we were able to get satisfactory throughput. But, as illustrated in Figure 6, we had to be careful not to disturb the data transfer (e.g., with TELNET traffic, or a simultaneous FETCH and STORE), or the whole thing would fall to pieces, and you'd be staring at the blinking lights on your modem wondering what the hell was going on.

Careful measurement and use of RTT and MDEV make SLIP Rx connections behave very well. Repeating the 100 KB SLIP transfer is much more satisfactory now, as illustrated in Figure 12. Data movement is nearly perfect, and the RTO calculation easily accommodates rapid changes in RTT.

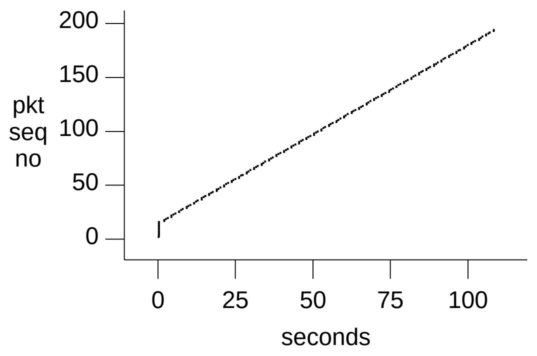

Figure 12. MTU = 576, RTO = 2. Performance of fully adaptive Rx with 576-byte packets in a dedicated SLIP line is nearly perfect.

To compare to Figure 6, if we start up a second adaptive Rx session a minute into the first, the results shown in Figure 13 ensue.

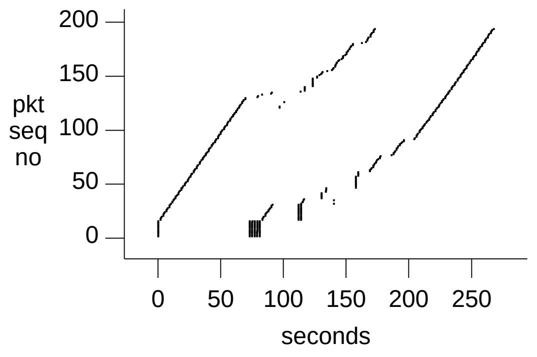

Figure 13. As the initial disruption due to the second session starting at 73 seconds works its way through the system, the first session adapts, retransmitting only two packets that were dropped from the router's queue when it overflowed, and continues smoothly to completion. In this experiment the first session had no unnecessary retransmissions. The second session retransmits every 2 seconds (the default timeout) until it gets its first acknowledgement, whereupon it increases its timeout to match the 18 second RTT experienced.

Figure 13 makes apparent that our work is not complete: the ''blob'' that corresponds to the start of the second connection can be addressed by improving our initial guess for the RTT, perhaps by using historical data. Slow-start or bandwidth estimation might also play a role here.

7. Header Compression

Having followed Van Jacobson's work as closely as possible so far, we continue with his TCP header compression algorithms [10]. Rx/UDP/IP headers, shown in Figure 14, consume almost 10% of AFS bandwidth with a 576-byte MTU.

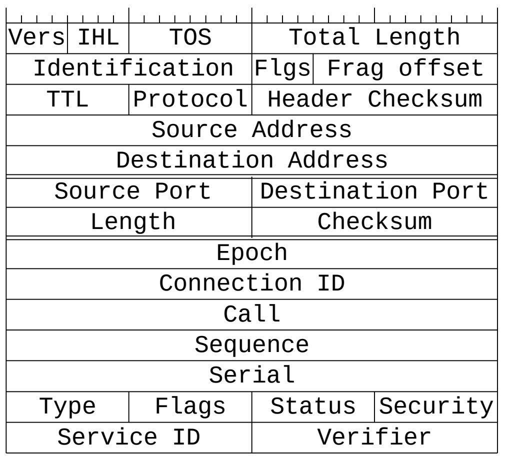

Figure 14. IP, UDP, and Rx headers. The headers for the network, transport, and RPC layers are shown, separated by double lines. Most of the fields in the combined headers can be predicted from an earlier packet or are constant.

Just as TCP/IP headers can be compressed by predicting the expected case, so can Rx/UDP/IP headers. The Van Jacobson approach is to elide header fields that remain constant, such as destination address, or that can be predicted most of the time, such as RPC sequence number.

An Rx connection can be uniquely characterized by the combination of its source and destination IP addresses, source and destination UDP ports, Rx connection ID, and Rx epoch. We refer to these fields as the Rx connection's *signature*.

When the sender encounters a new signature, it stashes a private copy of the combined headers, and forwards the (uncompressed) packet to the receiver. Thereafter, the private copy is referenced by a one-byte index. The sender alerts the receiver with a bogus IP version number and tucks the index into the IP protocol field. The receiver fixes the IP header and stashes it for later use.

When the sender encounters a packet with a known signature, it uses a different bogus IP version number, and sends the index of the packet in lieu of the signature fields. The receiver fills in the signature fields from its stashed copy of the uncompressed headers. This alone replaces the 20 signature bytes with a one-byte index.

A 12-bit wide mask is used to control the remaining fields. A bit indicating the absence of a field indicates that the value can be calculated from the stashed copy. Values that change are encoded in the compressed headers in the Van Jacobson style, so that small changes, denoted Δ, can be sent in one byte. We had some bits left over, so we coded some common, special cases, such as the Rx call number increasing by one.

Rx compression is indicated with the IP version number. This four-bit field is usually set to `4`. TCP header compression uses the values from `7` on. We use `5` and `6`. Figure 15 shows the common two-byte preamble of compressed Rx headers.

IP Vers.				A	B	C	D
E	F	G	H	I	J	K	L

Figure 15. Compressed Rx/UDP/IP header preamble. The four-bit IP version number and a 12-bit mask are sent with each packet. Fields that are not predictable from a previous packet are appended. The bit mask indicates precisely which fields are appended and which should be calculated.

7.1. Bit mask rules

The bit mask rules are as follows.

`A` If the Rx sequence number bit (bit `F`) is set and Δ is one, then this bit is set. This eliminates the need to send Δ for a common case.

`B` If the Rx call number bit (bit `E`) is set and Δ is one, then this bit is set. This eliminates the need to send Δ for a common case.

`C` If the packet signature differs from the previous packet, then this bit is set and the Rx index is included in the compressed header.

`D` If the IP packet ID is not one greater than the value seen in the last packet with the same signature, then this bit is set and Δ is included in the compressed header.

`E` If the Rx call number differs from the call number in the previous packet with the same signature, then this bit is set. If Δ is one, bit `B` is set, otherwise Δ is included in the compressed header.

`F` If the Rx sequence number differs from the sequence number in the previous packet with the same signature, then this bit is set. If Δ is one or if the new sequence number is zero, bit `A` or `L` is set, respectively. Otherwise Δ is included in the compressed header.

`G` If the Rx serial number is not one greater than the value seen in the last packet with this signature, then this bit is set and Δ is included in the compressed header.

`H` If the Rx type field is not the same as the value seen in the last packet with this signature, then this bit is set and the type is included in the compressed header.

`I` If the Rx flags field is not the same as the value seen in the last packet with this signature, then this bit is set and the flags are included in the compressed header.

`J` If the Rx status field is not the same as the value seen in the last packet with this signature, then this bit is set and the status is included in the compressed header.

`K` If the packet has a non-zero Rx verifier, this bit is set and the two-byte verifier is included in the compressed header.

`L` If the Rx sequence number bit (bit `F`) is set and the new sequence number is zero, then this bit is set. This eliminates the need to send Δ for a common case.

For example, the IP packet ID is almost always one greater than that sent on the last packet. If that is the case, bit `D` is set and the IP packet ID is not sent. Otherwise, the sender includes the encoded IP packet ID in the compressed header.

Figure 16 shows the order format of the remaining fields in a compressed Rx/UDP/IP header, most of which are optional and under the control of the bit mask.

Header index
UDP checksum
Δ IP ID
Δ Rx Call
Δ Rx Sequence
Δ Rx Serial
Rx Type
Rx flags
Rx Status
Rx Service ID
Verifier

Figure 16. Compressed Rx/UDP/IP header fields. Most of these fields are optional. The bit mask in the header preamble tells which fields are present in the packet.

To test the effectiveness of the Van Jacobson approach to Rx header compression, we monitored Rx headers in two sessions. The first session ran overnight, recording all Rx activity over a SLIP line to the home of one of the authors.

The second session monitored Rx activity on an Ethernet for five hours while a staff member went about her work, mostly building and testing UNIX kernels. The first trace has 6,435 compressed headers; the second trace has 14,416. We used a Network General Sniffer to collect the traces and wrote a simulator to calculate the average number of bits sent for each field of the compressed headers. Figure 17 shows the simulation results for the combined traces.

Field	Bits
IP Vers.	4
Bit mask	12
Header index	4.8
UDP checksum	16
Δ IP ID	2.4
Δ Rx Call	0.04
Δ Rx Sequence	0.99
Δ Rx Serial	0.076
Rx Type	1.3
Rx flags	3.4
Rx Status	0.086
Verifier	3.3
Long Δ	2.8
TOTAL	51.1

Figure 17. Average length of compressed Rx/UDP/IP header fields. Using a trace driven simulator, we calculated the number of bits used in each of the fields of compressed Rx/UDP/IP headers. Three fields — IP Version number, bit mask, and UDP checksum — are never compressed. Several fields associated with Rx, such as Call, Serial, and Status, are very predictable.

The entry labeled ‘‘`Long Δ`’’ accounts for differences that do not fit into one byte.

Although the UDP checksum can be derived from other information in the packet, it is our only assurance of end-to-end reliability, so it is never elided, and always takes up two bytes. So when things are going well, the compressed header size is six bytes for an authenticated connection — the two-byte preamble containing the bit map, the UDP checksum, and the Rx verifier — and four bytes for an unauthenticated connection.

The table shows that on average, the remaining fields add about two bytes to the total header length. In the traces we collected, over 95% of the headers were sent compressed; the weighted average length of all Rx headers, compressed and uncompressed, was 8.2 bytes. This represents a dramatic improvement over the 56-byte headers sent in uncompressed packets.

7.2. Synchronization

If a packet gets lost or corrupted, the sender and receiver lose the synchronization required to reconstitute compressed packets. In this case, the receiver uncompresses headers incorrectly and hands bogus packets to the IP layer. Because the UDP checksum is almost certainly incorrect for packets so constructed, the UDP layer later discards them. This continues until a new uncompressed packet is received, at which time the sender and receiver can restore synchronization.

The sender uses Rx retransmissions to detect synchronization error; the UDP layer on the receiver is tossing packets with abandon, so ACKs are not being generated. Eventually, the service provider will start retransmitting. This is apparent to the sender, as the Rx sequence number jumps backwards. Under these circumstances, the sender sends an uncompressed packet, and synchronization is restored.

Synchronization error can have a very detrimental effect on throughput. Suppose a large file is being fetched from the server to a client. If a single byte is lost or corrupted in the transfer across the SLIP line, the client begins discarding packets. More than likely, an entire window’s worth of packets is discarded. After the first discarded packet times out, the server begins to retransmit, restoring synchronization. In the mean time, though, a single bit error has been magnified into a $15 \times 576 \times 8$ bits retransmitted, *i.e.,* the error cascades almost 70,000-fold. For this reason, we put special effort into engineering the asynchronous device driver to avoid data overruns.

8. Throughput

We conducted a series of experiments to measure the performance of AFS servers running with our changes to Rx. In these experiments we fetched a 100 KB file from the server to the client. The SLIP endpoints are IBM RTs, running 4.3BSD (`rt_aos4`) as dataless AFS 3.1 clients. During the tests, they ran normal background tasks, but were otherwise idle. The modems are US Robotics Courier V.32*bis* locked at 14.4 Kbps link and 19.2 kbps interface speeds. The RT asynchronous interface is known to operate at 57.6 Kbps. AFS fetches are from cold cache to `/dev/null`. FTP is in binary mode, from server `/tmp` to client `/dev/null`.

The interface rate is 19.2K, allowing a maximum 1920 bytes/sec. (V.32*bis* over V.42 can support

1724 bytes/sec. [13]; 11% compressibility and V.42*bis* delivers up the remainder.)

There are no measurements for the unmodified AFS server. Prior to our congestion avoidance changes, AFS performance over Rx was so chaotic that it was difficult to measure reliably. The off-the shelf AFS product imposes both IP fragmentation and massive retransmissions when deployed over a low-speed network. Prior to lowering the Rx packet size to fit in a single IP frame, service was non-existent. With a smaller Rx frame, service remains poor, tending to be in the 100−200 bytes/sec. range.

The following table shows the effect of congestion avoidance and Rx header compression over a SLIP interface. The Rx columns show the throughput in bps for the improved Rx with (Rx/HDR) and without (Rx/RTT) header compression while transferring different types of files.† For comparison we also show throughput using FTP over TCP/IP with compressed headers. The values in parentheses show the fraction of the maximum rate.‡

In most cases, we achieve 95% or more of the available throughput. We can't explain the inferior performance when pushing uncompressible files, but we suspect V.42*bis* is confused.

	Rx/RTT	Rx/HDR	FTP
V.42*bis* on	1727	1893	1903
V.42*bis* off	1551	1700	1709

Maximal bytes per second

	Rx/RTT	Rx/HDR	FTP
`vmunix.Z`	1030 (.6)	1234 (.71)	1420 (.75)
`vmunix`	1622 (.94)	1838 (.97)	1891 (.99)
`troffsrc`	1628 (.94)	1841 (.97)	1854 (.97)
`tenrisks`	1630 (.94)	1855 (.98)	1899 (.998)

Measured bytes per second

Figure 18. SLIP throughput. The tables show maximal and measured throughput for Rx transfers with and without header compression, and for FTP with header compression. Maximal throughput takes into account framing overhead at the various network layers.

9. Summary and Future Work

Our modifications to Rx have moved us from a system that was initially unusable in highly congested or low-speed environments, to one that easily accommodates prevailing network conditions. This effort pays off in high-speed nets as well, because Rx can now more quickly accommodate packet loss or corruption. Work remains to done, though. Rx still needs exponential backoff on successive retransmissions. Slow-start must be studied to determine whether a smoother approach to the equilibrium state is valuable in the context of a streaming RPC oriented toward bulk data transfer.

Other studies and improvements planned for Rx include window adjustments when congestion is detected (or suspected), and real IP MTU discovery. We are in search of a means to estimate end-to-end bandwidth, from which we can derive pipe size, which we hope to use to determine the optimal window size. Rx is used in environments that differ in pipe size by many orders of magnitude. This presents a real challenge to the implementation of a high-performance transport layer.

† `vmunix.Z` is a 697,180 byte, compressed file. `vmunix` is a 1,036,474 byte, binary executable. `troffsrc` is 266,949 bytes of C source code. `tenrisks` is 241,202 bytes of Email text.

‡ IP MTU is 576 bytes. SLIP framing adds two bytes, so the ''wire'' frame is 578 bytes. SLIP byte stuffing is accounted for in the file sizes shown. Uncompressed AFS headers are 56 bytes. For authenticated requests, compressed Rx headers are six bytes. Compressed TCP/IP headers are three bytes.

The maximum data rate is

$$1920 \times \frac{(578 - 2 - \text{hdrsize})}{578}$$

if the file is compressible with V.42*bis*; otherwise, replace 1920 with 1724.

Acknowledgements

Work at CITI was partially supported by IBM and Telebit.

References

1. J.H. Howard, ''An Overview of the Andrew File System,'' pp. 23–26 in *Winter 1988 USENIX Conf. Proc.*, Dallas (February, 1988).
2. J.B. Postel, ''User Datagram Protocol,'' RFC 768, Network Information Center, SRI International, Menlo Park, CA (August 1980).
3. Edward R. Zayas, ''AFS−3 Programmer's Reference: Specification for the *Rx* Remote Procedure Call Facility,'' Report FS-00-D164, Transarc Corporation (August, 1991).
4. R.N. Sidebotham, ''Rx: Extended Remote Procedure Call,'' in *Proceedings of the Nationwide File System Workshop*, Information Technology Center, Carnegie Mellon University, Pittsburgh (August, 1988).

5. Sun Microsystems, Inc., ‘‘RPC: Remote Procedure Call Protocol specification: Version 2,’’ RFC 1057, Network Information Center, SRI International, Menlo Park, CA (June 1988).

6. J.B. Postel, ‘‘Transmission Control Protocol,’’ RFC 793, Network Information Center, SRI International, Menlo Park, CA (September 1981).

7. J.B. Postel, ‘‘Internet Protocol,’’ RFC 791, Network Information Center, SRI International, Menlo Park, CA (September 1981).

8. Christopher A. Kent and Jeffrey C. Mogul, ‘‘Fragmentation Considered Harmful,’’ *Proc. SIGCOMM ’87 Workshop*, Stowe, VT, pp. 390–401 (August 1987).

9. J.L. Romkey, ‘‘Nonstandard for transmission of IP datagrams over serial lines: SLIP,’’ RFC 1055, Network Information Center, SRI International, Menlo Park, CA (June 1988).

10. V. Jacobson, ‘‘Compressing TCP/IP Headers for Low-Speed Serial Links,’’ RFC 1145, Network Information Center, SRI International, Menlo Park, CA (February 1990).

11. V. Jacobson, ‘‘Congestion Avoidance and Control,’’ *Proc. ACM SIGCOMM ’88*, Stanford, CA, pp. 314-329 (August 1988).

12. P. Karn and C. Partridge, ‘‘Improving Round-trip Time Estimates in Reliable Transport Protocols,’’ *Proc. ACM SIGCOMM ’87*, Stowe, Vermont, pp. 2–7 (1987).

13. Toby Nixon, ‘‘Estimate of V.32bis throughput,’’ `comp.dcom.modems` (24 Sep 91).