

Network Working Group
Request for Comments: 3390
Obsoletes: 2414
Updates: 2581
Category: Standards Track

M. Allman
BBN/NASA GRC
S. Floyd
ICIR
C. Partridge
BBN Technologies
October 2002

Increasing TCP's Initial Window

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2002). All Rights Reserved.

Abstract

This document specifies an optional standard for TCP to increase the permitted initial window from one or two segment(s) to roughly 4K bytes, replacing RFC 2414. It discusses the advantages and disadvantages of the higher initial window, and includes discussion of experiments and simulations showing that the higher initial window does not lead to congestion collapse. Finally, this document provides guidance on implementation issues.

Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

1. TCP Modification

This document obsoletes [RFC2414] and updates [RFC2581] and specifies an increase in the permitted upper bound for TCP's initial window from one or two segment(s) to between two and four segments. In most cases, this change results in an upper bound on the initial window of roughly 4K bytes (although given a large segment size, the permitted initial window of two segments may be significantly larger than 4K bytes).

The upper bound for the initial window is given more precisely in (1):

$$\min (4 * \text{MSS}, \max (2 * \text{MSS}, 4380 \text{ bytes})) \quad (1)$$

Note: Sending a 1500 byte packet indicates a maximum segment size (MSS) of 1460 bytes (assuming no IP or TCP options). Therefore, limiting the initial window's MSS to 4380 bytes allows the sender to transmit three segments initially in the common case when using 1500 byte packets.

Equivalently, the upper bound for the initial window size is based on the MSS, as follows:

```
If (MSS <= 1095 bytes)
    then win <= 4 * MSS;
If (1095 bytes < MSS < 2190 bytes)
    then win <= 4380;
If (2190 bytes <= MSS)
    then win <= 2 * MSS;
```

This increased initial window is optional: a TCP MAY start with a larger initial window. However, we expect that most general-purpose TCP implementations would choose to use the larger initial congestion window given in equation (1) above.

This upper bound for the initial window size represents a change from RFC 2581 [RFC2581], which specified that the congestion window be initialized to one or two segments.

This change applies to the initial window of the connection in the first round trip time (RTT) of data transmission following the TCP three-way handshake. Neither the SYN/ACK nor its acknowledgment (ACK) in the three-way handshake should increase the initial window size above that outlined in equation (1). If the SYN or SYN/ACK is lost, the initial window used by a sender after a correctly transmitted SYN MUST be one segment consisting of MSS bytes.

TCP implementations use slow start in as many as three different ways: (1) to start a new connection (the initial window); (2) to restart transmission after a long idle period (the restart window); and (3) to restart transmission after a retransmit timeout (the loss window). The change specified in this document affects the value of the initial window. Optionally, a TCP MAY set the restart window to the minimum of the value used for the initial window and the current value of cwnd (in other words, using a larger value for the restart window should never increase the size of cwnd). These changes do NOT change the loss window, which must remain 1 segment of MSS bytes (to

permit the lowest possible window size in the case of severe congestion).

2. Implementation Issues

When larger initial windows are implemented along with Path MTU Discovery [RFC1191], and the MSS being used is found to be too large, the congestion window ``cwnd'` SHOULD be reduced to prevent large bursts of smaller segments. Specifically, ``cwnd'` SHOULD be reduced by the ratio of the old segment size to the new segment size.

When larger initial windows are implemented along with Path MTU Discovery [RFC1191], alternatives are to set the "Don't Fragment" (DF) bit in all segments in the initial window, or to set the "Don't Fragment" (DF) bit in one of the segments. It is an open question as to which of these two alternatives is best; we would hope that implementation experiences will shed light on this question. In the first case of setting the DF bit in all segments, if the initial packets are too large, then all of the initial packets will be dropped in the network. In the second case of setting the DF bit in only one segment, if the initial packets are too large, then all but one of the initial packets will be fragmented in the network. When the second case is followed, setting the DF bit in the last segment in the initial window provides the least chance for needless retransmissions when the initial segment size is found to be too large, because it minimizes the chances of duplicate ACKs triggering a Fast Retransmit. However, more attention needs to be paid to the interaction between larger initial windows and Path MTU Discovery.

The larger initial window specified in this document is not intended as encouragement for web browsers to open multiple simultaneous TCP connections, all with large initial windows. When web browsers open simultaneous TCP connections to the same destination, they are working against TCP's congestion control mechanisms [FF99], regardless of the size of the initial window. Combining this behavior with larger initial windows further increases the unfairness to other traffic in the network. We suggest the use of HTTP/1.1 [RFC2068] (persistent TCP connections and pipelining) as a way to achieve better performance of web transfers.

3. Advantages of Larger Initial Windows

1. When the initial window is one segment, a receiver employing delayed ACKs [RFC1122] is forced to wait for a timeout before generating an ACK. With an initial window of at least two segments, the receiver will generate an ACK after the second data segment arrives. This eliminates the wait on the timeout (often up to 200 msec, and possibly up to 500 msec [RFC1122]).

2. For connections transmitting only a small amount of data, a larger initial window reduces the transmission time (assuming at most moderate segment drop rates). For many email (SMTP [Pos82]) and web page (HTTP [RFC1945, RFC2068]) transfers that are less than 4K bytes, the larger initial window would reduce the data transfer time to a single RTT.
3. For connections that will be able to use large congestion windows, this modification eliminates up to three RTTs and a delayed ACK timeout during the initial slow-start phase. This will be of particular benefit for high-bandwidth large-propagation-delay TCP connections, such as those over satellite links.
4. Disadvantages of Larger Initial Windows for the Individual Connection

In high-congestion environments, particularly for routers that have a bias against bursty traffic (as in the typical Drop Tail router queues), a TCP connection can sometimes be better off starting with an initial window of one segment. There are scenarios where a TCP connection slow-starting from an initial window of one segment might not have segments dropped, while a TCP connection starting with an initial window of four segments might experience unnecessary retransmits due to the inability of the router to handle small bursts. This could result in an unnecessary retransmit timeout. For a large-window connection that is able to recover without a retransmit timeout, this could result in an unnecessarily-early transition from the slow-start to the congestion-avoidance phase of the window increase algorithm. These premature segment drops are unlikely to occur in uncongested networks with sufficient buffering or in moderately-congested networks where the congested router uses active queue management (such as Random Early Detection [FJ93, RFC2309]).

Some TCP connections will receive better performance with the larger initial window even if the burstiness of the initial window results in premature segment drops. This will be true if (1) the TCP connection recovers from the segment drop without a retransmit timeout, and (2) the TCP connection is ultimately limited to a small congestion window by either network congestion or by the receiver's advertised window.

5. Disadvantages of Larger Initial Windows for the Network

In terms of the potential for congestion collapse, we consider two separate potential dangers for the network. The first danger would be a scenario where a large number of segments on congested links

were duplicate segments that had already been received at the receiver. The second danger would be a scenario where a large number of segments on congested links were segments that would be dropped later in the network before reaching their final destination.

In terms of the negative effect on other traffic in the network, a potential disadvantage of larger initial windows would be that they increase the general packet drop rate in the network. We discuss these three issues below.

Duplicate segments:

As described in the previous section, the larger initial window could occasionally result in a segment dropped from the initial window, when that segment might not have been dropped if the sender had slow-started from an initial window of one segment. However, Appendix A shows that even in this case, the larger initial window would not result in the transmission of a large number of duplicate segments.

Segments dropped later in the network:

How much would the larger initial window for TCP increase the number of segments on congested links that would be dropped before reaching their final destination? This is a problem that can only occur for connections with multiple congested links, where some segments might use scarce bandwidth on the first congested link along the path, only to be dropped later along the path.

First, many of the TCP connections will have only one congested link along the path. Segments dropped from these connections do not "waste" scarce bandwidth, and do not contribute to congestion collapse.

However, some network paths will have multiple congested links, and segments dropped from the initial window could use scarce bandwidth along the earlier congested links before ultimately being dropped on subsequent congested links. To the extent that the drop rate is independent of the initial window used by TCP segments, the problem of congested links carrying segments that will be dropped before reaching their destination will be similar for TCP connections that start by sending four segments or one segment.

An increased packet drop rate:

For a network with a high segment drop rate, increasing the TCP initial window could increase the segment drop rate even further. This is in part because routers with Drop Tail queue management have difficulties with bursty traffic in times of congestion. However, given uncorrelated arrivals for TCP connections, the larger TCP initial window should not significantly increase the segment drop rate. Simulation-based explorations of these issues are discussed in Section 7.2.

These potential dangers for the network are explored in simulations and experiments described in the section below. Our judgment is that while there are dangers of congestion collapse in the current Internet (see [FF99] for a discussion of the dangers of congestion collapse from an increased deployment of UDP connections without end-to-end congestion control), there is no such danger to the network from increasing the TCP initial window to 4K bytes.

6. Interactions with the Retransmission Timer

Using a larger initial burst of data can exacerbate existing problems with spurious retransmit timeouts on low-bandwidth paths, assuming the standard algorithm for determining the TCP retransmission timeout (RTO) [RFC2988]. The problem is that across low-bandwidth network paths on which the transmission time of a packet is a large portion of the round-trip time, the small packets used to establish a TCP connection do not seed the RTO estimator appropriately. When the first window of data packets is transmitted, the sender's retransmit timer could expire before the acknowledgments for those packets are received. As each acknowledgment arrives, the retransmit timer is generally reset. Thus, the retransmit timer will not expire as long as an acknowledgment arrives at least once a second, given the one-second minimum on the RTO recommended in RFC 2988.

For instance, consider a 9.6 Kbps link. The initial RTT measurement will be on the order of 67 msec, if we simply consider the transmission time of 2 packets (the SYN and SYN-ACK), each consisting of 40 bytes. Using the RTO estimator given in [RFC2988], this yields an initial RTO of 201 msec ($67 + 4 \cdot (67/2)$). However, we round the RTO to 1 second as specified in RFC 2988. Then assume we send an initial window of one or more 1500-byte packets (1460 data bytes plus overhead). Each packet will take on the order of 1.25 seconds to transmit. Therefore, the RTO will fire before the ACK for the first packet returns, causing a spurious timeout. In this case, a larger initial window of three or four packets exacerbates the problems caused by this spurious timeout.

One way to deal with this problem is to make the RTO algorithm more conservative. During the initial window of data, for instance, the RTO could be updated for each acknowledgment received. In addition, if the retransmit timer expires for some packet lost in the first window of data, we could leave the exponential-backoff of the retransmit timer engaged until at least one valid RTT measurement, that involves a data packet, is received.

Another method would be to refrain from taking an RTT sample during connection establishment, leaving the default RTO in place until TCP takes a sample from a data segment and the corresponding ACK. While this method likely helps prevent spurious retransmits, it also may slow the data transfer down if loss occurs before the RTO is seeded. The use of limited transmit [RFC3042] to aid a TCP connection in recovering from loss using fast retransmit rather than the RTO timer mitigates the performance degradation caused by using the high default RTO during the initial window of data transmission.

This specification leaves the decision about what to do (if anything) with regards to the RTO, when using a larger initial window, to the implementer. However, the RECOMMENDED approach is to refrain from sampling the RTT during the three-way handshake, keeping the default RTO in place until an RTT sample involving a data packet is taken. In addition, it is RECOMMENDED that TCPs use limited transmit [RFC3042].

7. Typical Levels of Burstiness for TCP Traffic.

Larger TCP initial windows would not dramatically increase the burstiness of TCP traffic in the Internet today, because such traffic is already fairly bursty. Bursts of two and three segments are already typical of TCP [Flo97]; a delayed ACK (covering two previously unacknowledged segments) received during congestion avoidance causes the congestion window to slide and two segments to be sent. The same delayed ACK received during slow start causes the window to slide by two segments and then be incremented by one segment, resulting in a three-segment burst. While not necessarily typical, bursts of four and five segments for TCP are not rare. Assuming delayed ACKs, a single dropped ACK causes the subsequent ACK to cover four previously unacknowledged segments. During congestion avoidance this leads to a four-segment burst, and during slow start a five-segment burst is generated.

There are also changes in progress that reduce the performance problems posed by moderate traffic bursts. One such change is the deployment of higher-speed links in some parts of the network, where a burst of 4K bytes can represent a small quantity of data. A second change, for routers with sufficient buffering, is the deployment of

queue management mechanisms such as RED, which is designed to be tolerant of transient traffic bursts.

8. Simulations and Experimental Results

8.1 Studies of TCP Connections using that Larger Initial Window

This section surveys simulations and experiments that explore the effect of larger initial windows on TCP connections. The first set of experiments explores performance over satellite links. Larger initial windows have been shown to improve the performance of TCP connections over satellite channels [All97b]. In this study, an initial window of four segments (512 byte MSS) resulted in throughput improvements of up to 30% (depending upon transfer size). [KAGT98] shows that the use of larger initial windows results in a decrease in transfer time in HTTP tests over the ACTS satellite system. A study involving simulations of a large number of HTTP transactions over hybrid fiber coax (HFC) indicates that the use of larger initial windows decreases the time required to load WWW pages [Nic98].

A second set of experiments explored TCP performance over dialup modem links. In experiments over a 28.8 bps dialup channel [All97a, AHO98], a four-segment initial window decreased the transfer time of a 16KB file by roughly 10%, with no accompanying increase in the drop rate. A simulation study [RFC2416] investigated the effects of using a larger initial window on a host connected by a slow modem link and a router with a 3 packet buffer. The study concluded that for the scenario investigated, the use of larger initial windows was not harmful to TCP performance.

Finally, [All00] illustrates that the percentage of connections at a particular web server that experience loss in the initial window of data transmission increases with the size of the initial congestion window. However, the increase is in line with what would be expected from sending a larger burst into the network.

8.2 Studies of Networks using Larger Initial Windows

This section surveys simulations and experiments investigating the impact of the larger window on other TCP connections sharing the path. Experiments in [All97a, AHO98] show that for 16 KB transfers to 100 Internet hosts, four-segment initial windows resulted in a small increase in the drop rate of 0.04 segments/transfer. While the drop rate increased slightly, the transfer time was reduced by roughly 25% for transfers using the four-segment (512 byte MSS) initial window when compared to an initial window of one segment.

A simulation study in [RFC2415] explores the impact of a larger initial window on competing network traffic. In this investigation, HTTP and FTP flows share a single congested gateway (where the number of HTTP and FTP flows varies from one simulation set to another). For each simulation set, the paper examines aggregate link utilization and packet drop rates, median web page delay, and network power for the FTP transfers. The larger initial window generally resulted in increased throughput, slightly-increased packet drop rates, and an increase in overall network power. With the exception of one scenario, the larger initial window resulted in an increase in the drop rate of less than 1% above the loss rate experienced when using a one-segment initial window; in this scenario, the drop rate increased from 3.5% with one-segment initial windows, to 4.5% with four-segment initial windows. The overall conclusions were that increasing the TCP initial window to three packets (or 4380 bytes) helps to improve perceived performance.

Morris [Mor97] investigated larger initial windows in a highly congested network with transfers of 20K in size. The loss rate in networks where all TCP connections use an initial window of four segments is shown to be 1-2% greater than in a network where all connections use an initial window of one segment. This relationship held in scenarios where the loss rates with one-segment initial windows ranged from 1% to 11%. In addition, in networks where connections used an initial window of four segments, TCP connections spent more time waiting for the retransmit timer (RTO) to expire to resend a segment than was spent using an initial window of one segment. The time spent waiting for the RTO timer to expire represents idle time when no useful work was being accomplished for that connection. These results show that in a very congested environment, where each connection's share of the bottleneck bandwidth is close to one segment, using a larger initial window can cause a perceptible increase in both loss rates and retransmit timeouts.

9. Security Considerations

This document discusses the initial congestion window permitted for TCP connections. Changing this value does not raise any known new security issues with TCP.

10. Conclusion

This document specifies a small change to TCP that will likely be beneficial to short-lived TCP connections and those over links with long RTTs (saving several RTTs during the initial slow-start phase).

11. Acknowledgments

We would like to acknowledge Vern Paxson, Tim Shepard, members of the End-to-End-Interest Mailing List, and members of the IETF TCP Implementation Working Group for continuing discussions of these issues and for feedback on this document.

12. References

- [AH098] Mark Allman, Chris Hayes, and Shawn Ostermann, An Evaluation of TCP with Larger Initial Windows, March 1998. ACM Computer Communication Review, 28(3), July 1998. URL "<http://roland.lerc.nasa.gov/~mallman/papers/initwin.ps>".
- [All97a] Mark Allman. An Evaluation of TCP with Larger Initial Windows. 40th IETF Meeting -- TCP Implementations WG. December, 1997. Washington, DC.
- [All97b] Mark Allman. Improving TCP Performance Over Satellite Channels. Master's thesis, Ohio University, June 1997.
- [All00] Mark Allman. A Web Server's View of the Transport Layer. ACM Computer Communication Review, 30(5), October 2000.
- [FF96] Fall, K., and Floyd, S., Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. Computer Communication Review, 26(3), July 1996.
- [FF99] Sally Floyd, Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. IEEE/ACM Transactions on Networking, August 1999. URL "<http://www.icir.org/floyd/end2end-paper.html>".
- [FJ93] Floyd, S., and Jacobson, V., Random Early Detection gateways for Congestion Avoidance. IEEE/ACM Transactions on Networking, V.1 N.4, August 1993, p. 397-413.
- [Flo94] Floyd, S., TCP and Explicit Congestion Notification. Computer Communication Review, 24(5):10-23, October 1994.
- [Flo96] Floyd, S., Issues of TCP with SACK. Technical report, January 1996. Available from <http://www-nrg.ee.lbl.gov/floyd/>.
- [Flo97] Floyd, S., Increasing TCP's Initial Window. Viewgraphs, 40th IETF Meeting - TCP Implementations WG. December, 1997. URL "<ftp://ftp.ee.lbl.gov/talks/sf-tcp-ietf97.ps>".

- [KAGT98] Hans Kruse, Mark Allman, Jim Griner, Diepchi Tran. HTTP Page Transfer Rates Over Geo-Stationary Satellite Links. March 1998. Proceedings of the Sixth International Conference on Telecommunication Systems. URL "<http://roland.lerc.nasa.gov/~mallman/papers/nash98.ps>".
- [Mor97] Robert Morris. Private communication, 1997. Cited for acknowledgement purposes only.
- [Nic98] Kathleen Nichols. Improving Network Simulation With Feedback, Proceedings of LCN 98, October 1998. URL "<http://www.computer.org/proceedings/lcn/8810/8810toc.htm>".
- [Pos82] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, August 1982.
- [RFC1122] Braden, R., "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1122, October 1989.
- [RFC1191] Mogul, J. and S. Deering, "Path MTU Discovery", RFC 1191, November 1990.
- [RFC1945] Berners-Lee, T., Fielding, R. and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.
- [RFC2068] Fielding, R., Mogul, J., Gettys, J., Frystyk, H. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, January 1997.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2309] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J. and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, April 1998.
- [RFC2414] Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's Initial Window", RFC 2414, September 1998.
- [RFC2415] Poduri, K. and K. Nichols, "Simulation Studies of Increased Initial TCP Window Size", RFC 2415, September 1998.
- [RFC2416] Shepard, T. and C. Partridge, "When TCP Starts Up With Four Packets Into Only Three Buffers", RFC 2416, September 1998.

- [RFC2581] Allman, M., Paxson, V. and W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.
- [RFC2821] Klensin, J., "Simple Mail Transfer Protocol", RFC 2821, April 2001.
- [RFC2988] Paxson, V. and M. Allman, "Computing TCP's Retransmission Timer", RFC 2988, November 2000.
- [RFC3042] Allman, M., Balakrishnan, H. and S. Floyd, "Enhancing TCP's Loss Recovery Using Limited Transmit", RFC 3042, January 2001.
- [RFC3168] Ramakrishnan, K.K., Floyd, S. and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

Appendix A - Duplicate Segments

In the current environment (without Explicit Congestion Notification [Flo94] [RFC2481]), all TCPs use segment drops as indications from the network about the limits of available bandwidth. We argue here that the change to a larger initial window should not result in the sender retransmitting a large number of duplicate segments that have already arrived at the receiver.

If one segment is dropped from the initial window, there are three different ways for TCP to recover: (1) Slow-starting from a window of one segment, as is done after a retransmit timeout, or after Fast Retransmit in Tahoe TCP; (2) Fast Recovery without selective acknowledgments (SACK), as is done after three duplicate ACKs in Reno TCP; and (3) Fast Recovery with SACK, for TCP where both the sender and the receiver support the SACK option [MMFR96]. In all three cases, if a single segment is dropped from the initial window, no duplicate segments (i.e., segments that have already been received at the receiver) are transmitted. Note that for a TCP sending four 512-byte segments in the initial window, a single segment drop will not require a retransmit timeout, but can be recovered by using the Fast Retransmit algorithm (unless the retransmit timer expires prematurely). In addition, a single segment dropped from an initial window of three segments might be repaired using the fast retransmit algorithm, depending on which segment is dropped and whether or not delayed ACKs are used. For example, dropping the first segment of a three segment initial window will always require waiting for a timeout, in the absence of Limited Transmit [RFC3042]. However, dropping the third segment will always allow recovery via the fast retransmit algorithm, as long as no ACKs are lost.

Next we consider scenarios where the initial window contains two to four segments, and at least two of those segments are dropped. If all segments in the initial window are dropped, then clearly no duplicate segments are retransmitted, as the receiver has not yet received any segments. (It is still a possibility that these dropped segments used scarce bandwidth on the way to their drop point; this issue was discussed in Section 5.)

When two segments are dropped from an initial window of three segments, the sender will only send a duplicate segment if the first two of the three segments were dropped, and the sender does not receive a packet with the SACK option acknowledging the third segment.

When two segments are dropped from an initial window of four segments, an examination of the six possible scenarios (which we don't go through here) shows that, depending on the position of the

dropped packets, in the absence of SACK the sender might send one duplicate segment. There are no scenarios in which the sender sends two duplicate segments.

When three segments are dropped from an initial window of four segments, then, in the absence of SACK, it is possible that one duplicate segment will be sent, depending on the position of the dropped segments.

The summary is that in the absence of SACK, there are some scenarios with multiple segment drops from the initial window where one duplicate segment will be transmitted. There are no scenarios in which more than one duplicate segment will be transmitted. Our conclusion is that the number of duplicate segments transmitted as a result of a larger initial window should be small.

Author's Addresses

Mark Allman
BBN Technologies/NASA Glenn Research Center
21000 Brookpark Rd
MS 54-5
Cleveland, OH 44135
EMail: mallman@bbn.com
<http://roland.lerc.nasa.gov/~mallman/>

Sally Floyd
ICSI Center for Internet Research
1947 Center St, Suite 600
Berkeley, CA 94704
Phone: +1 (510) 666-2989
EMail: floyd@icir.org
<http://www.icir.org/floyd/>

Craig Partridge
BBN Technologies
10 Moulton St
Cambridge, MA 02138
EMail: craig@bbn.com

Full Copyright Statement

Copyright (C) The Internet Society (2002). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

[\[RFCs\]](#) [\[I-Ds\]](#) [\[Plain Text\]](#)

Versions: [00](#)

INTERNET-DRAFT
Expires: February 21, 1997
Inc.

Chi Chu
Research 2000,
August 1996

IP Cluster
draft-chu-ip-cluster-00.txt

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as ``work in progress.''

To learn the current status of any Internet-Draft, please check the ``l-id-abstracts.txt' listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

1. Abstract

This Internet-Draft is intended to provide a means for "IP Clustering" across multiple servers. It is meant as an improved alternative to the various solutions for distributing WWW traffic already attempted by the IETF DNS Working Group. In addition, the clustering method can be applied not only to a heavily visited web server, but also to any overloaded TCP/IP servers such as a domain name server. The IP Cluster provides two primary functions:
IP traffic distribution to multiple servers and fault-tolerance.

2. Introduction

The notion of distributing IP (Web) data traffic to multiple server machines has already been foray-ed by the various DNS methods mentioned or described in [RFC 1794](#) [1]. The basic drawbacks for all these methods are similar:

- * short or zero TTL for DNS records - this is not intended by the DNS specification and incurs a few unpleasant consequences;
- * heavy DNS traffic - since secondary or non-authoritative DNS servers cannot effectively cache the data, all these methods generate heavy DNS queries across the global Internet, bombarding a chain of servers in the name space;

[Page
1]

INTERNET DRAFT IP Cluster August
1996

- * potentially high delay - if any server in the DNS chain experiences outage or bottleneck, the response to the initial query would be significantly delayed if an alternate DNS server were required to process the query.
- * the primary DNS server becomes the single point of failure - since the TTL is very small or zero, outage of the primary server for even a small period of time results in failed DNS lookup;
- * easier to spoof - a DNS record can be easily "spoof-ed" to mislead a client to a bogus host name to IP address mapping;

and among other drawbacks that are method specific. In short, these DNS methods solve one problem that may be beneficial to a single site, but create another that can be quite undesirable for the Internet at large. Just imagine what would happen if every website decides to implement a DNS method for distributing its web traffic.

Clearly, it is imperative and highly desirable that an alternative solution be established that does not suffer the same drawbacks discussed above, and yet the new solution not introduce a new problem equal in its severity to the network at large.

3. The Alternative

3.1 Applicable Topology

The proposed method requires an IP router connecting to multiple servers in a switch-like configuration. That is, each server machine is directly connected to a unique physical port/interface on the router. Since a router interface can be a LAN or serial interface, this IP cluster formation can span locally or be distributed via wide-area network.

3.2 Description

The nature of IP load balancing requires that for a given IP host name, typically corresponding to some network services, the data traffic and thus the processing be actually distributed among

several server machines, with some control. This idea of a "virtual server" provides transparent services to a remote client.

The virtual server itself consists a number of host machines, or cluster members, each performing a set of services. In a true cluster environment, a cluster member performs a set of services or functions that may be different from that of another member within the cluster.

Similar to all the DNS load balancing methods, the proposed method

described in this document assumes symmetric host processing.

Namely, the so-called "IP Cluster" consists of a number of cluster

members each of which performs the same set of services (although strictly speaking, it does not have to be necessarily so).

[Page

2]

INTERNET DRAFT
1996

IP Cluster

August

The alternative method does not rely on the dynamic host name to IP address mapping. Instead, it relies on the concept of a Virtual IP (VIP) address. This VIP address is configured as the host IP address for all cluster members. Each cluster member is directly connected to a unique router interface port, much like an Ether-Switch configuration, topologically.

The VIP address appears to the outside world as just another unique IP address, with the usual DNS host name to IP address mapping in the traditional static sense. To each of the IP cluster members, it thinks that this VIP address is its globally unique host address.

However, this VIP address appears very differently to the IP router to which all the cluster members are directly connected to. With careful and deliberate choice of the VIP address (e.g., xx.xx.xx.63 for a Class C network), and with the appropriate subnet (or variable subnet) mask enabled in the router interface ports, this unique host IP address is in effect a broadcasting address as far as the router is concerned. Consequently, upon receiving an IP packet with destination address equal to this VIP

address, the router will attempt to, assuming configured properly, broadcast the packet to all its relevant interfaces.

Each of the "broadcasted" interface, however, is configured with a simple filter. This simple filter basically filters on the IP source address of the incoming packet. Thus with each interface filter permitting only a unique and non-overlapping portion of the IP address space to route through, we have effectively achieved high-performance "IP-Switching".

Furthermore, since this portioning of the IP address space can be

4. A Scalable Model

In short, this IP clustering model should scale quite linearly.

[Page

August

While I will not delve into all the relevant issues of building a Fault Tolerant (FT) IP Cluster, suffice to say, however, with

this IP cluster model, one may easily build a Fault Tolerant IP Cluster against any single point of host-or-network failure within the cluster.

6. Implementation

The implementation of this IP cluster assumes that a router used for IP switching is capable of forwarding IP broadcast packets. While most routers have limited broadcast forwarding capability (e.g., some may not forward TCP/IP broadcast packets), this limitation should be easily removed by a perspective router vendor

by relaxing the artificially imposed transport-layer filtering (which is not entirely a router's business to begin with).

Reconfiguration of the IP-Switch/router filters for achieving better load balance should be performed by an automated script. Since this type of reconfiguration is considered system down-time

for the IP cluster, the implementation of such a script should minimize the down-time by, for instance, separating logging into the router from actually modifying the filters with human control.

As for communications between cluster members (i.e., heartbeat, etc.), any number of protocols can be used. It may be as simple as ping and tcp-echo, or as sophisticated as a new multicast protocol.

7. Performance

[Page

4]

INTERNET DRAFT
1996

IP Cluster

August

As already mentioned in [Section 4](#), the IP clustering method described in this proposal should be extremely fast. The so-called

IP cluster here is essentially an IP-Switch (as opposed to an Ether or FastEther Switch) connecting to number of cluster members each taking full advantage of the underlying transmission

medium without the usual network contention.

Assuming that one is to configure a "Super IP-Switch" with maximum IO ports and each port is connected to the highest bandwidth technology and server machine available, the only issue

with regard to performance then is the router's routing capability, particularly the router's CPU required to perform the interface filtering.

We can rest assured, however, that this interface filtering or the router's routing performance cannot realistically be an issue

for two reasons. Reason one, because of bitmasking and wildcarding, each interface filter list should be very short and compact. (I don't see more than six lines in each access list unless the same router is also used for firewalling, etc.) Reason two, long before one reaches such routing performance issues, any reasonable organization would want to add a second router into the same IP cluster. The VIP clustering model supports multiple routers as an integral part of a single IP cluster. In fact, building such an IP cluster with multiple routers is one step towards building a fault-tolerant IP cluster.

One question remains: How effective is the load balancing scheme based on the IP source address filtering, which if not effective, would defeat a lot of this high-performance claim. I would say: pretty effective, especially if the client base is very large (which is what this proposal is intended to accomplish to begin with).

This is simply a basic principle of statistical analysis: when there is a large number of statistical samples, with each sample behaving randomly and wildly, the overall statistical distribution is often predictable and well behaved. In fact, the larger the number, the more predictable and better behaved the statistical envelope would be. Thus, this statistical property works greatly in favor of this Internet-Draft's intent to use the IP cluster to support very large client base.

Assuming one has setup the proposed IP cluster with multiple servers. It makes no sense to talk about how good the load balance actually is when the traffic is light enough that if all the traffic gets distributed to a single cluster member that that member server is still not overloaded. Good load balance becomes relevant when traffic is heavy enough that some or all of the cluster members must share significant (but still not necessarily equal) portions of the traffic load. It is important to keep the

[Page

5]

INTERNET DRAFT
1996

IP Cluster

August

perspective that the real purpose of clustering is to avoid server overloading and not to artificially maintain equal load balance at all time. The beauty of this IP clustering model is that the more traffic and the larger the client base grows, the better and more evenly the cluster distributes the load without incurring any processing overhead.

The above load analysis simply means that an effective IP cluster

does not require fully dynamic load balancing per IP packet. In fact, a truly dynamic load balancing scheme on per packet basis would adversely affect the performance of such an IP cluster.

How often (e.g., once a month, etc.) and what criteria (e.g., CPU, memory, IO) the load balance sampling and analyzing should be performed in order to re-tune, if necessary, the IP-Switch/router access filter lists are application dependent.

8. Security Considerations

While the DNS methods for IP clustering relies on dynamic host name to IP address mapping, which can easily be "spoof-ed", the Virtual IP method does not suffer the same level of security issues for the simple reason that it is more difficult to spoof (and spoof it well) the routing topology of the Internet than to spoof a DNS record.

Additionally, this Virtual IP clustering model does not preclude any security schemes that are available under a non-cluster single server environment, firewalls included.

9. Acknowledgments

Much appreciation is due to Mike Lee and Josh Sierles for enlightening me with the DNS load balancing methods, and to Josh again for referring me to [RFC 1794](#).

10. References

- [1] Brisco, T., "DNS Support for Load Balancing", [RFC 1794](#), Rutgers University, April 1995.

11. Author's Address

Chi Chu
Research 2000, Inc.
265 Cherry Street, 16G
New York, New York 10002
USA

Phone: 212-598-9455
Email: chi@soho.ios.com
URL: <http://soho.ios.com/~chi>

This document expires February 21, 1997.

Network Working Group
Request for Comments: 2391
Category: Informational

P. Srisuresh
Lucent Technologies
D. Gan
Juniper Networks, Inc.
August 1998

Load Sharing using IP Network Address Translation (LSNAT)

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1998). All Rights Reserved.

Preface

This document combines the idea of address translation described in RFC 1631 with real-time load share algorithms to introduce Load Share Network Address Translators(or, simply LSNATs). LSNATs would transparently offload network load on a single server and distribute the load across a pool of servers.

Abstract

Network Address Translators (NATs) translate IP addresses in a datagram, transparent to end nodes, while routing the datagram. NATs have traditionally been used to allow private network domains to connect to Global networks using as few as one globally unique IP address. In this document, we extend the use of NATs to offer Load share feature, where session load can be distributed across a pool of servers, instead of directing to a single server. Load sharing is beneficial to service providers and system administrators alike in grappling with scalability of servers with increasing session load.

1. Introduction

Traditionally, Network Address Translators, or simply NATs were used to connect private network domains to globally unique public domain IP networks. Applications originate in private domains and NATs would transparently translate datagrams belonging to these applications in

either direction. This document combines the characteristic of transparent address translation with real-time load share algorithms to introduce Load Share Network Address Translators.

The problem of Load sharing or Load balancing is not new and goes back many years. A variety of techniques were applied to address the problem. Some very ad-hoc and platform specific and some employing clever schemes to reorder DNS resource records. REF [11] uses DNS zone transfer program in name servers to periodically shuffle the order of resource records for server nodes based on a pre-determined load balancing algorithm. The problem with this approach is that reordering time periods can be very large on the order of minutes and does not reflect real-time load variations on the servers. Secondly, all hosts in the server pool are assumed to have equal capability to offer all services. This may not often be the case. In addition, there may be requirement to support load balancing for a few specific services only. The load share approach outlined in this document addresses both these concerns and offers a solution that does not require changes to clients or servers and one that can be tailored to individual services or for all services.

For the reminder of this document, we will refer to NAT routers that provide load sharing support as LSNATs. Unlike traditional NATs, LSNATs are not required to operate between private and public domain routing realms alone. LSNATs also operate in a single routing realm and provide load sharing functionality.

The need for Load sharing arises when a single server is not able to cope with increasing demand for multiple sessions simultaneously. Clearly, load sharing across multiple servers would enhance responsiveness and scale well with session load. Popular applications inundating servers would include Web browsers, remote login, file transfer and mail applications.

When a client attempts to access a server through an LSNAT router, the router selects a node in server pool, based on a load share algorithm and redirect the request to that node. LSNATs pose no restriction on the organization and rearrangement of nodes in server pool. Nodes in a pool may be replaced, new nodes may be added and others may be in transition. Changes of this kind to server pool can be shielded from client nodes by making LSNAT router the focal point for change management.

There are limitations to using LSNATs. Firstly, it is mandatory that all requests and responses pertaining to a session between a client and server be routed via the same LSNAT router. For this reason, we recommend LSNATs to be operated on a single border router to a stub domain in which the server pool would be confined. This would ensure

that all traffic directed to servers from clients outside the domain and vice versa would necessarily traverse the LSNAT border router. Later in the document, we will examine a special case of LSNAT setup, which gets around the topological constraint on server pool. Another limitation of LSNATs is the inability to switch loads between hosts in the midst of sessions. This is because LSNATs measure load in granularity of sessions. Once a session is assigned to a host, the session cannot be moved to a different host till the end of that session. Other limitations, inherent to NATs, as outlined in REF [1] are also applicable to LSNATs.

As with traditional NATs, LSNATs have the disadvantage of taking away the end-to-end significance of an IP address. The major advantage, however, is that it can be installed without changes to clients or servers.

2. Terminology and concepts used

2.1. TU ports, Server ports, Client ports

For the remainder of this document, we will refer TCP/UDP ports associated with an IP address simply as "TU ports".

For most TCP/IP hosts, TU port range 0-1023 is used by servers listening for incoming connections. Clients trying to initiate a connection typically select a TU port in the range of 1024-65535. However, this convention is not universal and not always followed. It is possible for client nodes to initiate connections using a TU port number in the range of 0-1023, and there are applications listening on TU port numbers in the range of 1024-65535.

A complete list of TU port services may be found in REF [2]. The TU ports used by servers to listen for incoming connections are called "Server Ports" and the TU ports used by clients to initiate a connection to server are called "Client Ports".

2.2. Session flow vs. Packet flow

Connection or session flows are different from packet flows. A session flow indicates the direction in which the session was initiated with reference to a network port. Packet flow is the direction in which the packet has traversed with reference to a network port. A session flow is uniquely identified by the direction in which the first packet of that session traversed.

Take for example, a telnet session. The telnet session consists of packet flows in both inbound and outbound directions. Outbound telnet packets carry terminal keystrokes from the client and inbound telnet

packets carry screen displays from the telnet server. Performing address translation for a telnet session would involve translation of incoming as well as outgoing packets belonging to that session.

Packets belonging to a TCP/UDP session are uniquely identified by the tuple of (source IP address, source TU port, target IP address, target TU port). ICMP sessions that correlate queries and responses using query id are uniquely identified by the tuple of (source IP address, ICMP Query Identifier, target IP address). For lack of well-known ways to distinguish, all other types of sessions are lumped together and distinguished by the tuple of (source IP address, IP protocol, target IP address).

2.3. Start of session for TCP, UDP and others

The first packet of every TCP session tries to establish a session and contains connection startup information. The first packet of a TCP session may be recognized by the presence of SYN bit and absence of ACK bit in the TCP flags. All TCP packets, with the exception of the first packet must have the ACK bit set.

The first packet of every session, be it a TCP session, UDP session, ICMP query session or any other session, tries to establish a session. However, there is no deterministic way of recognizing the start of a UDP session or any other non-TCP session.

Start of session is significant with NATs, as a state describing translation parameters for the session is established at the start of session. Packets pertaining to the session cannot undergo translation, unless a state is established by NAT at the start of session.

2.4. End of session for TCP, UDP and others

The end of a TCP session is detected when FIN is acknowledged by both halves of the session or when either half receives RST bit in TCP flags field. Within a short period (say, a couple of seconds) after one of the session partners sets RST bit, the session can be safely assumed to have been terminated.

For all other types of session, there is no deterministic way of determining the end of session unless you know the application protocol. Many heuristic approaches are used to terminate sessions. You can make the assumption that TCP sessions that have not been used for say, 24 hours, and non-TCP sessions that have not been used for say, 1 minute, are terminated. Often this assumption works, but sometimes it doesn't. These idle period session timeouts may vary considerably across the board and may be made user configurable.

Another way to handle session terminations is to timestamp sessions and keep them as long as possible and retire the longest idle session when it becomes necessary.

2.5. Basic Network Address Translation (Basic NAT)

Basic NAT is a method by which hosts in a private network domain are allowed access to hosts in the external network transparently. A block of external addresses are set aside for translating addresses of private hosts as the private hosts originate sessions to applications in external domain. Once an external address is bound by the NAT device to a specific private address, that address binding remains in place for all subsequent sessions originating from the same private host. This binding may be terminated when there are no sessions left to use the binding.

2.6. Network Address Port Translation (NAPT)

Network Address Port Translation(NAPT) is a method by which hosts in a private network domain are allowed simultaneous access to hosts in the external network transparently using a single registered address. This is made possible by multiplexing transport layer identifiers of private hosts into the transport identifiers of the single assigned external address. For this reason, only the applications based on TCP and UDP protocols are supported by NAPT. ICMP query based applications are also supported as the ICMP header carries a query identifier that is used to correlate responses with requests. Sessions other than TCP, UDP and ICMP query type are simply not permitted from local nodes, serviced by a NAPT router.

2.7. Load share

Load sharing for the purpose of this document is defined as the spread of session load amongst a cluster of servers which are functionally similar or the same. In other words, each of the nodes in cluster can support a client session equally well with no discernible difference in functionality. Once a node is assigned to service a session, that session is bound to that node till termination. Sessions are not allowed to swap between nodes in the midst of session.

Load sharing may be applicable for all services, if all hosts in server cluster carry the capability to carry out all services. Alternately, load sharing may be limited to one or more specific services alone and not to others.

Note, the term "Session load" used in the context of load share is different from the term "system load" attributed to hosts by way of CPU, memory and other resource usage on the system.

3. Overview of Load sharing

While both traditional NATs and LSNATs perform address translations, and provide transparent connectivity between end nodes, there are distinctions between the two. Traditional NATs initiate translations on outbound sessions, by binding a private address to a global address (basic NAT) or by binding a tuple of private address and transport identifier (such as TCP/UDP port or ICPM query ID) to a tuple of global address and transport identifier. LSNATs, on the other hand, initiate translations on inbound sessions, by binding each session represented by a tuple such as (client address, client TU port, virtual server address, server TU port) to one of server pool nodes, selected based on a real-time load-share algorithm. A virtual server address is a globally unique IP address that identifies a physical server or a group of servers that can provide similar or same functionality.

For the remainder of this document, we will refer traditional NATs simply as NATs and refer LSNATs exclusively in the context of load share, without implying traditional NAT functionality.

LSNATs are not limited to operate between private and public domain routing realms. LSNATs may operate within a single routing realm with globally unique IP addresses, just as well as between private and public network domains. The only requirement is that server pool be confined to a stub domain, accessible to clients outside the domain through a single LSNAT border router. However, as you will notice later, this topology limitation on server pool can be overcome under certain configurations.

Load Share NAT operates as follows. A client attempts to access a server by using the server virtual address. The LSNAT router transparently redirects the request to one of the hosts in server pool, selected using a real-time load sharing algorithm. Multiple sessions may be initiated from the same client, and each session could be directed to a different host based on load balance across server pool hosts at the time. If load share is desired for just a few specific services, the configuration on LSNAT could be defined to restrict load share for just the services desired.

In the case where virtual server address is same as the interface address of an LSNAT router, server applications (such as telnet) on LSNAT router must be disabled for external access on that address. This is the limitation to using address owned by LSNAT router as the virtual server address.

Load share NAT operation is also applicable during individual server upgrades as follows. Say, a server, that needs to be upgraded is statically mapped to a backup server on the inbound. Subsequent to this mapping, new session requests to the original server would be redirected by LSNAT to the backup server. As an extension, it is also possible to statically map a specific TU port service on a server to that of backup sever.

We illustrate the operation of LSNAT in the following subsections, where (a) servers are confined to a stub domain, and belong to globally unique address space as shared by clients, (b) servers are confined to private address space stub domain, and (c) servers are not restrained by any topological limitations.

3.1 Operation of LSNAT in a globally unique address space

In this section, we will illustrate the operation of LSNAT in a globally unique address space. The border router with LSNAT enabled on WAN link would perform load sharing and address translations for inbound sessions. However, sessions outbound from the hosts in server pool will not be subject to any type of translation, as all nodes have globally unique IP addresses.

In the example below, servers S1 (172.85.0.1), S2(172.85.0.2) and S3(172.85.0.3) form a server pool, confined to a stub domain. LSNAT on the border router is enabled on the WAN link, such that the virtual server address S(172.87.0.100) is mapped to the server pool consisting of hosts S1, S2 and S3. When a client 198.76.29.7 initiates a HTTP session to the virtual server S, the LSNAT router examines the load on hosts in server pool and selects a host, say S1 to service the request. The transparent address and TU port translations performed by the LSNAT router become apparent as you follow the down arrow line. IP packets on the return path go through similar address translation. Suppose, we have another client 198.23.47.2 initiating telnet session to the same virtual server S. The LSNAT would determine that host S3 is a better choice to service this session as S1 is busy with a session and redirect the session to S3. The second session redirection path is delineated with colons. The procedure continues for any number of sessions the same way.

In addition, say the organization has servers S1 (10.0.0.1), S2(10.0.0.2) and S3 (10.0.0.3) that form a pool to provide inbound access to external clients. This is made possible by enabling LSNAT on the WAN link of the border router, such that virtual server address S(198.76.28.4) is mapped to the server pool consisting of hosts S1, S2 and S3. When an external client 198.76.29.7 initiates a HTTP session to the virtual server S, the LSNAT router examines load on hosts in server pool and selects a host, say S1 to service the request. The transparent address and TU port translations performed by the LSNAT router are apparent as you follow the down arrow line. IP packets on the return path go through similar address translation. Suppose, we have another client 198.23.47.2 initiating telnet session to the same address. The LSNAT would determine that host S3 is a better choice to service this session as S1 is busy with a session and redirect the session to S3. The second session redirection path is delineated with colons. The procedure continues for any number of sessions the same way.

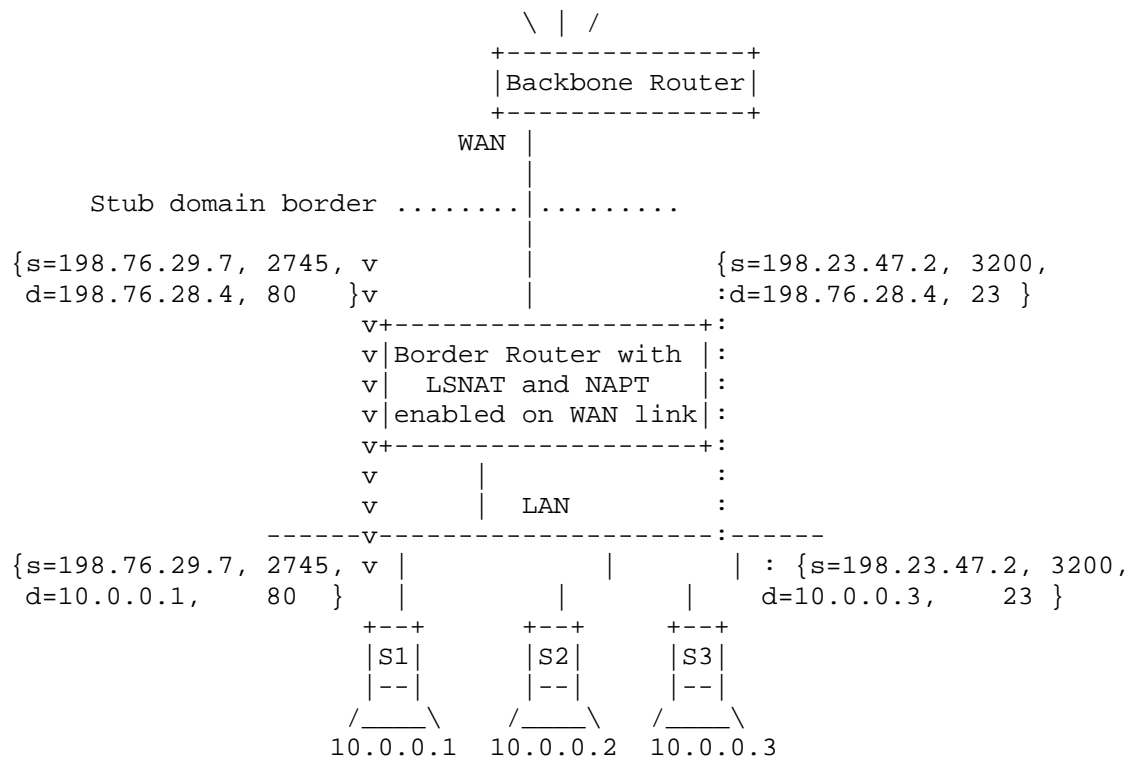


Figure 2: Operation of LSNAT, in coexistence with NATP

Once again, notice that this requires no changes to clients or servers. The translation is completely transparent to end nodes. Address mapping on the LSNAT performs load sharing and address translations for inbound sessions. Sessions outbound from hosts in server pool are subject to NAT. Both NAT and LSNAT co-exist with each other in the same router.

3.3. Load Sharing with no topological restraints on servers

In this section, we will illustrate a configuration in which load sharing can be accomplished on a router without enforcing topological limitations on servers. In this configuration, virtual server address will be owned by the router that supports load sharing. I.e., virtual server address will be same as address of one of the interfaces of load share router. We will distinguish this configuration from LSNAT by referring this as "Load Share Network Address Port Translation" (LS-NAPT). Routers that support the LS-NAPT configuration will be termed "LS-NAPT routers", or simply LS-NAPTs.

In an LSNAT router, inbound TCP/UDP sessions, represented by the tuple of (client address, client TU port, virtual server address, service port) are translated into a tuple of (client address, client TU port, selected server address, service port). Translation is carried out on all datagrams pertaining to the same session, in either direction. Whereas, LS-NAPT router would translate the same session into a tuple of (virtual server address, virtual server TU port, selected server, service port). Notice that LS-NAPT router translates the client address and TU port with the address and TU port of virtual server, which is same as the address of one of its interfaces. By doing this, datagrams from clients as well as servers are forced to bear the address of LS-NAPT router as the destination address, thereby guaranteeing that the datagrams would necessarily traverse the LS-NAPT router. As a result, there is no need to require servers to be under topological constraints.

Take for example, figure 1 in section 3.1. Let us say the router on which load sharing is enabled is not just a border router, but can be any kind of router. Let us also say that the virtual server address S (172.87.0.100) is same as the address of WAN link and LS-NAPT is enabled on the WAN interface. Figure 3 summarizes the new router configuration.

When a client 198.76.29.7 initiates a HTTP session to the virtual server address S (i.e., address of the WAN interface), the LS-NAPT router examines load on hosts in server pool and selects a host, say S1 to service the request. Appropriately, the destination address is translated to be S1 (172.85.0.1). Further, original client address and TU port are replaced with the address and TU port of the WAN

link. As a result, destination addresses as well as source address and source TU port are translated when the packet reaches S1, as can be noticed from the down-arrow path. IP packets on the return path go through similar translation. The second client 198.23.47.2 initiating telnet session to the same virtual server address S is load share directed to S3. This packet once again undergoes LS-NAPT translation, just as with the first client. The data path and translations can be noticed following the colon line. The procedure continues for any number of sessions the same way. The translations made to datagrams in either direction are completely transparent to end nodes.

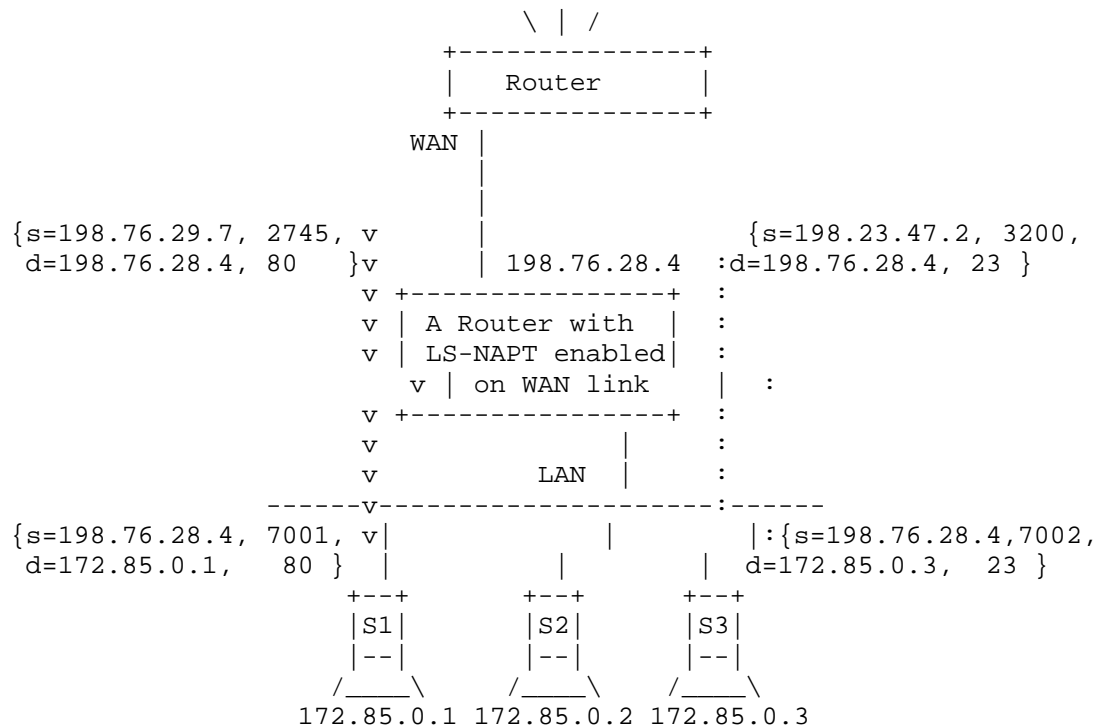


Figure 3: LS-NAPT configuration on a router

As you will notice, datagrams from clients as well as servers are forced to be directed to the router, because they use WAN interface address of router as the destination address in their datagrams. With the assurance that all packets from clients and servers would traverse the router, there is no longer a requirement for servers to be confined to a stub domain and for LSNAT to be enabled only on border router to the stub domain.

The LS-NAPT configuration described in this section involves more translations and hence is more complex compared to LSNAT configurations described in the previous sections. While the processing is complex, there are benefits to this configuration. Firstly, it breaks down restraints on server topology. Secondly, it scales with bandwidth expansion for client access. Even if Service providers have one link today for client access, the LS-NAPT configuration allows them to expand to more links in the future guaranteeing the same LS-NAPT load share service on newer links.

The configuration is not without its limitations. Server applications (such as telnet) on the router box would have to be disabled for the interface address assigned to be virtual server address. Load sharing would be limited to TCP and UDP applications only. Maximum concurrently allowed sessions would be limited by the maximum allowed TCP/UDP client ports on the same address. Assuming that ports 0-1023 must be set aside as well-known service ports, that would leave a maximum of 63K TCP client ports and 63K of UDP client ports on the LS-NAPT router to communicate with each load-share server. As a result, LS-NAPT routers will not be able to concurrently support more than a maximum of (63K * count of Load-share servers) TCP sessions and (63K * count of Load-share servers) UDP sessions.

4.0. Translation phases of a session in LSNAT router.

As with NATs, LSNATs must monitor the following three phases in relation to Address translation.

4.1. Session binding:

Session binding is the phase in which an incoming session is associated with the address of a host in server pool. This association essentially sets the translation parameters for all subsequent datagrams pertaining to the session. For addresses that have static mapping, the binding happens at startup time. Otherwise, each incoming session is dynamically bound to a different host based on a load sharing algorithm.

4.2. Address lookup and translation:

Once session binding is established for a connection setup, all subsequent packets belonging to the same connection will be subject to session lookup for translation purposes.

For outbound packets of a session, the source IP address (and source TU port, in case of TCP/UDP sessions) and related fields (such as IP, TCP, UDP and ICMP header checksums) will undergo translation. For inbound packets of a session, the destination IP address (and

destination TU port, in case of TCP/UDP sessions) and related fields such as IP, TCP, UDP and ICMP header checksums) will undergo translation.

The header and payload modifications made to IP datagrams subject to LSNAT will be exactly same as those subject to traditional NATs, described in section 5.0 of REF [1]. Hence, the reader is urged to refer REF [1] document for packet translation process.

4.3. Session unbinding:

Session unbinding is the phase in which a server node is no longer responsible for the session. Usually, session unbinding happens when the end of session is detected. As described in the terminology section, it is not always easy to determine end of session.

5. Load share algorithms

Many algorithms are available to select a host from a pool of servers to service a new session. The load distribution is based primarily on (a) cost of accessing the network on which a server resides and load on the network interface used to access the server, and (b) resource availability and system load on the server. A variety of policies can be adapted to distribute sessions across the servers in a server pool.

For simplicity, we will consider two types algorithms, based on proximity between server nodes and LSNAT router. The higher the cost of access to a sever, the farther the proximity of server is assumed to be. The first kind of algorithms will assume that all server pool members are at equal or nearly equal proximity to LSNAT router and hence the load distribution can be based solely on resource availability or system load on remote servers. Cost of network access will be considered irrelevant. The second kind would assume that all server pool members have equal resource availability and the criteria for selection would be proximity to servers. In other words, we consider algorithms which take into account the cost of network access.

5.1. Local Load share algorithms

Ideally speaking, the selection process would have precise knowledge of real-time resource availability and system load for each host in server pool, so that the selection of host with maximum unutilized capacity would be the obvious choice. However, this is not so easy to achieve.

We consider here two kinds of heuristic approaches to monitor session load on server pool members. The first kind is where the load share selector tracks system load on individual servers in non-intrusive way. The second kind is where the individual members actively participate in communicating with the load share selector, notifying the selector of their load capacity.

Listed below are the most common selection algorithms adapted in the non-intrusive category.

1. Round-Robin algorithm

This is the simplest scheme, where a host is selected simply on a round robin basis, without regard to load on the host.

2. Least Load first algorithm

This is an improvement over round-robin approach, in that, the host with least number of sessions bound to it is selected to service a new session. This approach is not without its caveats. Each session is assumed to be as resource consuming as any other session, independent of the type of service the session represents and all hosts in server pool are assumed to be equally resourceful.

3. Least traffic first algorithm

A further improvement over the previous algorithm would be to measure system load by tracking packet count or byte count directed from or to each of the member hosts over a period of time. Although packet count is not the same as system load, it is a reasonable approximation.

4. Least Weighted Load first approach

This would be an enhancement to the first two. This would allow administrators to assign (a) weights to sessions, based on likely resource consumption estimates of session types and (b) weights to hosts based on resource availability.

The sum of all session loads by weight assigned to a server, divided by weight of server would be evaluated to select the server with least weighted load to assign for each new session. Say, FTP sessions are assigned 5 times the weight($5x$) as a telnet session(x), and server S3 is assumed to be 3 times as resourceful as server S1. Let us also say that S1 is assigned 1 FTP session and 1 telnet session, whereas S3 is assigned 2 FTP sessions and 5 telnet sessions. When a new telnet session need assignment, the weighted load on S3 is evaluated to be $(2*5x+5*x)/3 = 5x$, and the load on S1 is evaluated to be $(1*5x+1*x) = 6x$. Server S3 is selected to bind the new telnet session, as the weighted load on S3 is smaller than that of S1.

5. Ping to find the most responsive host.

Till now, capacity of a member host is determined exclusively by the LSNAT using heuristic approaches. In reality, it is impossible to predict system capacity from remote, without interaction with member hosts. A prudent approach would be to periodically ping member hosts and measure the response time to determine how busy the hosts really are. Use the response time in conjunction with the heuristics to select the host most appropriate for the new session.

In the active category, we involve individual member hosts in resource utilization monitoring process. An agent software on each node would notify the monitoring agent on resource availability. Clearly, this would imply having an application program (one that does not consume significant resources, by itself) to run on each member node. This strategy of involving member hosts in system load monitoring is likely to yield the most optimal results in the selection process.

5.2. Distributed Load share algorithms

When server nodes are distributed geographically across different areas and cost to access them vary widely, the load share selector could use that information in selecting a server to service a new session. In order to do this, the load share selector would need to consult the routing tables maintained by routing protocols such as RIP and OSPF to find the cost of accessing a server.

All algorithms listed below would be non-intrusive kind where the server nodes do not actively participate in notifying the load share selector of their load capacity.

1. Weighted Least Load first algorithm

The selection criteria would be based on (a) cost of access to server, and (b) the number of sessions assigned to server. The product of cost and session load for each server would be evaluated to select the server with least weighted load for each new session. Say, cost of accessing server S1 is twice as much as that of server S2. In that case, S1 will be assigned twice as much load as that of S2 during the distribution process. When a server is not accessible due to network failure, the cost of access is set to infinity and hence no further load can be assigned to that server.

2. Weighted Least traffic first algorithm

An improvement over the previous algorithm would be to measure network load by tracking packet count or byte count directed from or to each of the member hosts over a

period of time. Although packet count is not the same as system load, it is a reasonable approximation. So, the product of cost and traffic load (over a fixed duration) for each server would be evaluated to select the server with least weighted traffic load for each new session.

6. Dead host detection

As sessions are assigned to hosts, it is important to detect the live-ness of the hosts. Otherwise, sessions could simply be black-holed into a dead host. Many heuristic approaches are adopted. Sending pings periodically would be one way to determine the live-ness. Another approach would be to track datagrams originating from a member host in response to new session assignments. If no response is detected in a few seconds, declare the server dead and do not assign new sessions to this host. The server can be monitored later again after a long pause (say, in the order of a few minutes) by periodically reassigning new sessions and monitoring response times and so on.

7. Miscellaneous

The IETF has been notified of potential intellectual Property Rights (IPR) issues with the technology described in this document. Interested people are requested to look in the IETF web page (<http://www.ietf.org>) under the Intellectual property Rights Notices section for the current information.

8. Security Considerations

All security considerations associated with NAT routers, described in REF [1] are applicable to LSNAT routers as well.

REFERENCES

- [1] Egevang, K. and P. Francis, "The IP Network Address Translator (NAT)", RFC 1631, May 1994.
- [2] Reynolds, J., and J. Postel, "Assigned Numbers", STD 2, RFC 1700, October 1994. See also: <http://www.iana.org/numbers.html>
- [3] Braden, R., "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1122, October 1989.
- [4] Braden, R., "Requirements for Internet Hosts -- Application and Support", STD 3, RFC 1123, October 1989.

- [5] Baker, F., "Requirements for IP Version 4 Routers", RFC 1812, June 1995.
- [6] Postel, J., and J. Reynolds, "File Transfer Protocol (FTP)", STD 9, RFC 959, October 1985.
- [7] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [8] Postel, J., "Internet Control Message (ICMP) Specification", STD 5, RFC 792, September 1981.
- [9] Postel, J., "User Datagram Protocol (UDP)", STD 6, RFC 768, August 1980.
- [10] Mogul, J., and J. Postel, "Internet Standard Subnetting Procedure", STD 5, RFC 950, August 1985.
- [11] Brisco, T., "DNS Support for Load Balancing", RFC 1794, April 1995.

Authors' Addresses

Pyda Srisuresh
Lucent Technologies
4464 Willow Road
Pleasanton, CA 94588-8519
U.S.A.

Voice: (925) 737-2153
Fax: (925) 737-2110
EMail: suresh@ra.lucent.com

Der-hwa Gan
Juniper Networks, Inc.
385 Ravensdale Drive.
Mountain View, CA 94043
U.S.A.

Voice: (650) 526-8074
Fax: (650) 526-8001
EMail: dhg@juniper.net

Full Copyright Statement

Copyright (C) The Internet Society (1998). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.



What is "mon"?

mon is a general-purpose scheduler and alert management tool used for monitoring service availability and triggering alerts upon failure detection. *mon* was designed to be open and extensible in the sense that it supports arbitrary monitoring facilities and alert methods via a common interface, all of which are easily implemented with programs in C, Perl, shell, etc., SNMP traps, and special *mon* traps.

mon views resource monitoring as two separate tasks: the testing of a condition, and triggering an action upon failure. *mon* was designed to implement the testing and action-taking tasks as separate, stand-alone programs. *mon* is fundamentally a scheduler which executes the monitors (each test a specific condition), and calls the appropriate alerts if the monitor fails. The decision to invoke an alert is governed by logic which offers various "squench" features and dependencies, all of which are configurable by the user.

Monitors and alerts are not a part of the core *mon* server, even though the distribution comes with a handful of them to get you started. This means that if a new service needs monitoring, or if a new alert is necessary, the *mon* server does not need to be changed. This makes *mon* easily extensible.

ABOUT [Some features that *mon* 0.99.2 offers.](#)

NEWS [The latest release notes and other news. Last updated Thu Jun 17 13:36:01 EDT 2004 .](#)

CONTRIB [Archive of user-submitted contributions \(monitors, alerts, clients, etc.\)](#)

FAQ [Frequently Asked Questions about *mon*](#)

LIST [Mailing list information](#)

BUGS [How to submit bugs and suggest features](#)

MONITORS [Monitor scripts which are included](#)

ALERTS [Alert scripts which are included](#)

CONFIG [An example configuration script to give you an idea of what *mon* can do.](#)

MANUAL [Manual pages for current release](#)

DEVELOPMENT [CVS access to *mon* source code](#)

FTP [Obtaining the current version via FTP](#)

Please use [a kernel.org mirror](#).

The path is /pub/software/admin/mon.

Stable: 0.99.2

Development: see [the latest code in CVS](#) on
sourceforge.net.

Jim Trocki

Network Engineer


trockij@linux.kernel.org



and generated with *m4*
6d5bbb4d

[HO](#) | [RESEAR](#) | **DAT** | [TOO](#) | [PUBLICATI](#) | [WORKSH](#) | [PROJE](#) | [FUNDI](#)
[ME](#) | [CH](#) | **A** | [LS](#) | [ONS](#) | [OPS](#) | [CTS](#) | [NG](#)

[by Topic](#) | [by Source](#) | [by Tool](#) | [by Accessibility](#) | [How-to](#) | [Statistics](#)

 **www.caida.org** > [data](#) : [visit](#) [contact](#) [search:](#)



Data Collection at CAIDA -- Data Sources

Recent Updates

DNS root/gTLD RTT Dataset (8/24/2006)	Backscatter-2006 Dataset (8/24/2006)	Code-Red Worms Dataset (6/15/2006)	Witty Worm Dataset (6/15/2006)	Autonomous System Relationships Dataset (4/27/2006)
---	--	--	--	---

As a part of CAIDA's mission to promote cooperative macroscopic measurement and analysis of Internet traffic and performance, collection of data for scientific analysis of network function is one of CAIDA's core objectives. CAIDA collects several different data types at geographically and topologically diverse locations, and makes this data available to the research community to the extent possible while preserving the privacy of individuals and organizations who donate data or network access. For questions about CAIDA data, contact data-info@caida.org.

To keep up to date on CAIDA datasets you can subscribe to data-announce@caida.org.

| View Caida Data by: [Topic](#) [Source](#) [Tool](#)
[Accessibility](#) [Statistics](#) |

For more information about using CAIDA data, please see the [CAIDA Data Usage FAQ](#).

CAIDA maintains a [list of non-CAIDA publications using CAIDA data](#).

CAIDA Data Collections by Source

Passive Data Collections

Many types of network measurements involve collecting data at strategic locations by replicating real traffic en route to its destination. No additional traffic is introduced into the network for these measurements; rather, a copy of all network information passing through a given location is recorded for future study. Since this type of data collection does not modify the links it measures, it is called *passive* data collection.

Data Sources

- [OC48 Commodity Link Traces](#)
- [DNS Data](#)
 - [DNS root/gTLD RTT Dataset](#)
 - DNS F Root Server (Palo Alto, CA)
 - DNS RFC 1918 Data
- [Network Telescope](#)
 - [Denial-of-Service Attack Backscatter:](#)
 - [Backscatter-2006 Dataset](#)
 - [Backscatter-2004-2005 Dataset](#)
 - [Backscatter-TOCS Dataset](#)



Cooperative Association for Internet Data Analysis (CAIDA)




Last Modified: Wed Dec-6-2006 12:14:15 PDT

Maintained by: Colleen Shannon

Page URL: http://www.caida.org/data/index_source.xml

measurement | [taxonomy](#) | [utilities](#) | [visualization](#)

 **www.caida.org** > [tools](#) :: measurement [visit](#) [contact](#) [search:](#)



CAIDA Measurement and Analysis Tools

CAIDA offers several tools for actively or passively measuring Internet traffic and flow patterns:

- [AutoFocus](#)

AutoFocus is a traffic analysis and visualization tool that describes the traffic mix of a link through textual reports and time series plots.

- [Beluga](#)

Beluga provides a real-time graph of RTTs and packet loss to an end host, showing both total round trip time and per-hop round trip time. It also shows a historical statistical breakdown of RTTs, to allow trend analysis.

- [cflowd](#)

cflowd is a flow analysis tool currently used for analyzing Cisco's NetFlow enabled switching method. The current release (described below) includes the collections, storage, and basic analysis modules for cflowd and for arts++ libraries. This analysis package permits data collection and analysis by ISPs and network engineers in support of capacity planning, trends analysis, and characterization of workloads in a network service provider environment. Other areas where cflowd may prove useful include usage tracking for Web hosting, accounting and billing, network planning and analysis, network monitoring, developing user profiles, data warehousing and mining, as well as security-related investigations.

- [CoralReef](#)

CoralReef is a comprehensive software suite developed by CAIDA to collect and analyze data from passive Internet traffic monitors, in real time or from trace files. Realtime monitoring support includes system network interfaces (via libpcap), FreeBSD drivers for Apptel POINT (OC12 and OC3 ATM) and FORE ATM (OC3 ATM) cards, and support for Linux and FreeBSD drivers for Endace DAG (OC3 and OC12, POS and ATM) cards. The package also includes programming APIs for C and perl, and applications for capture, analysis, and web report generation. This package is maintained by CAIDA developers with the support and collaboration of the Internet measurement community. CoralReef is the evolutionary successor of the Coral package and supersedes it.

- [iffinder](#)

The Skitter project discovers IP interfaces and how they're



Cooperative Association for Internet Data Analysis (CAIDA)



Last Modified: Sat Mar-4-2006 17:21:52 PDT

Maintained by: Alex Ma

Page URL: <http://www.caida.org/tools/measurement/index.xml>

PCAP

Section: C Library Functions (3)

Updated: 21 November 2003

[Index](#) [Return to Main Contents](#)

NAME

pcap - Packet Capture library

SYNOPSIS

```
#include <pcap.h>

char errbuf[PCAP_ERRBUF_SIZE];

pcap_t *pcap_open_live(const char *device, int snaplen,

    int promisc, int to_ms, char *errbuf)
pcap_t *pcap_open_dead(int linktype, int snaplen)
pcap_t *pcap_open_offline(const char *fname, char *errbuf)
pcap_dumper_t *pcap_dump_open(pcap_t *p, const char *fname)

int pcap_setnonblock(pcap_t *p, int nonblock, char *errbuf);
int pcap_getnonblock(pcap_t *p, char *errbuf);

int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)
void pcap_freealldevs(pcap_if_t *alldevs)
char *pcap_lookupdev(char *errbuf)
int pcap_lookupnet(const char *device, bpf_u_int32 *netp,

    bpf_u_int32 *maskp, char *errbuf)

int pcap_dispatch(pcap_t *p, int cnt,

    pcap_handler callback, u_char *user)
int pcap_loop(pcap_t *p, int cnt,

    pcap_handler callback, u_char *user)
void pcap_dump(u_char *user, struct pcap_pkthdr *h,

    u_char *sp)

int pcap_compile(pcap_t *p, struct bpf_program *fp,

    char *str, int optimize, bpf_u_int32 netmask)
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

```

void pcap_freecode(struct bpf_program *);

const u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header,

    const u_char **pkt_data)

void pcap_breakloop(pcap_t *)

int pcap_datalink(pcap_t *p)
int pcap_list_datalinks(pcap_t *p, int **dlt_buf);
int pcap_set_datalink(pcap_t *p, int dlt);
int pcap_datalink_name_to_val(const char *name);
const char *pcap_datalink_val_to_name(int dlt);
const char *pcap_datalink_val_to_description(int dlt);
int pcap_snapshot(pcap_t *p)
int pcap_is_swapped(pcap_t *p)
int pcap_major_version(pcap_t *p)
int pcap_minor_version(pcap_t *p)
int pcap_stats(pcap_t *p, struct pcap_stat *ps)
FILE *pcap_file(pcap_t *p)
int pcap_fileno(pcap_t *p)
void pcap_perror(pcap_t *p, char *prefix)
char *pcap_geterr(pcap_t *p)
char *pcap_strerror(int error)
const char *pcap_lib_version(void)

void pcap_close(pcap_t *p)
int pcap_dump_flush(pcap_dumper_t *p)
FILE *pcap_dump_file(pcap_dumper_t *p)
void pcap_dump_close(pcap_dumper_t *p)

```

DESCRIPTION

The Packet Capture library provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this mechanism.

ROUTINES

NOTE: *errbuf* in **pcap_open_live()**, **pcap_open_dead()**, **pcap_open_offline()**, **pcap_setnonblock()**, **pcap_getnonblock()**, **pcap_findalldevs()**, **pcap_lookupdev()**, and **pcap_lookupnet()** is assumed to be able to hold at least **PCAP_ERRBUF_SIZE** chars.

pcap_open_live() is used to obtain a packet capture descriptor to look at packets on the network. *device* is a string that specifies the network device to open; on Linux systems with 2.2 or later kernels, a *device* argument of "any" or **NULL** can be used to capture packets from all interfaces. *snaplen* specifies the maximum number of bytes to capture. If this value is less than the size of a packet that is captured, only the first *snaplen* bytes of that packet will be captured and provided as packet data. A value of

65535 should be sufficient, on most if not all networks, to capture all the data available from the packet. *promisc* specifies if the interface is to be put into promiscuous mode. (Note that even if this parameter is false, the interface could well be in promiscuous mode for some other reason.) For now, this doesn't work on the "any" device; if an argument of "any" or NULL is supplied, the *promisc* flag is ignored. *to_ms* specifies the read timeout in milliseconds. The read timeout is used to arrange that the read not necessarily return immediately when a packet is seen, but that it wait for some amount of time to allow more packets to arrive and to read multiple packets from the OS kernel in one operation. Not all platforms support a read timeout; on platforms that don't, the read timeout is ignored. A zero value for *to_ms*, on platforms that support a read timeout, will cause a read to wait forever to allow enough packets to arrive, with no timeout. *errbuf* is used to return error or warning text. It will be set to error text when **pcap_open_live()** fails and returns **NULL**. *errbuf* may also be set to warning text when **pcap_open_live()** succeeds; to detect this case the caller should store a zero-length string in *errbuf* before calling **pcap_open_live()** and display the warning to the user if *errbuf* is no longer a zero-length string.

pcap_open_dead() is used for creating a **pcap_t** structure to use when calling the other functions in libpcap. It is typically used when just using libpcap for compiling BPF code.

pcap_open_offline() is called to open a "savefile" for reading. *fname* specifies the name of the file to open. The file has the same format as those used by [tcpdump\(1\)](#) and [tcpslice\(1\)](#). The name "-" is a synonym for **stdin**. *errbuf* is used to return error text and is only set when **pcap_open_offline()** fails and returns **NULL**.

pcap_dump_open() is called to open a "savefile" for writing. The name "-" is a synonym for **stdout**. **NULL** is returned on failure. *p* is a *pcap* struct as returned by **pcap_open_offline()** or **pcap_open_live()**. *fname* specifies the name of the file to open. If **NULL** is returned, **pcap_geterr()** can be used to get the error text.

pcap_setnonblock() puts a capture descriptor, opened with **pcap_open_live()**, into "non-blocking" mode, or takes it out of "non-blocking" mode, depending on whether the *nonblock* argument is non-zero or zero. It has no effect on "savefiles". If there is an error, -1 is returned and *errbuf* is filled in with an appropriate error message; otherwise, 0 is returned. In "non-blocking" mode, an attempt to read from the capture descriptor with **pcap_dispatch()** will, if no packets are currently available to be read, return 0 immediately rather than blocking waiting for packets to arrive. **pcap_loop()** and **pcap_next()** will not work in "non-blocking" mode.

pcap_getnonblock() returns the current "non-blocking" state of the capture descriptor; it always returns 0 on "savefiles". If there is an error, -1 is returned and *errbuf* is filled in with an appropriate error message.

pcap_findalldevs() constructs a list of network devices that can be opened with **pcap_open_live()**. (Note that there may be network devices that cannot be opened with **pcap_open_live()** by the process calling **pcap_findalldevs()**, because, for example, that process might not have sufficient privileges to open them for capturing; if so, those devices will not appear on the list.) *alldevsp* is set to point to the first

element of the list; each element of the list is of type **pcap_if_t**, and has the following members:

next

if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list

name

a pointer to a string giving a name for the device to pass to **pcap_open_live()**

description

if not **NULL**, a pointer to a string giving a human-readable description of the device

addresses

a pointer to the first element of a list of addresses for the interface

flags

interface flags:

PCAP_IF_LOOPBACK

set if the interface is a loopback interface

Each element of the list of addresses is of type **pcap_addr_t**, and has the following members:

next

if not **NULL**, a pointer to the next element in the list; **NULL** for the last element of the list

addr

a pointer to a **struct sockaddr** containing an address

netmask

if not **NULL**, a pointer to a **struct sockaddr** that contains the netmask corresponding to the address pointed to by **addr**

broadaddr

if not **NULL**, a pointer to a **struct sockaddr** that contains the broadcast address corresponding to the address pointed to by **addr**; may be null if the interface doesn't support broadcasts

dstaddr

if not **NULL**, a pointer to a **struct sockaddr** that contains the destination address corresponding to the address pointed to by **addr**; may be null if the interface isn't a point-to-point interface

-1 is returned on failure, in which case **errbuf** is filled in with an appropriate error message; **0** is returned on success.

pcap_freealldevs() is used to free a list allocated by **pcap_findalldevs()**.

pcap_lookupdev() returns a pointer to a network device suitable for use with **pcap_open_live()** and **pcap_lookupnet()**. If there is an error, **NULL** is returned and **errbuf** is filled in with an appropriate error message.

pcap_lookupnet() is used to determine the network number and mask associated with the network device **device**. Both *netp* and *maskp* are *bpf_u_int32* pointers. A return of

-1 indicates an error in which case *errbuf* is filled in with an appropriate error message.

pcap_dispatch() is used to collect and process packets. *cnt* specifies the maximum number of packets to process before returning. This is not a minimum number; when reading a live capture, only one bufferful of packets is read at a time, so fewer than *cnt* packets may be processed. A *cnt* of -1 processes all the packets received in one buffer when reading a live capture, or all the packets in the file when reading a ``savefile". *callback* specifies a routine to be called with three arguments: a *u_char* pointer which is passed in from **pcap_dispatch()**, a *const struct pcap_pkthdr* pointer to a structure with the following members:

ts

a *struct timeval* containing the time when the packet was captured

caplen

a *bpf_u_int32* giving the number of bytes of the packet that are available from the capture

len

a *bpf_u_int32* giving the length of the packet, in bytes (which might be more than the number of bytes available from the capture, if the length of the packet is larger than the maximum number of bytes to capture)

and a *const u_char* pointer to the first **caplen** (as given in the *struct pcap_pkthdr* a pointer to which is passed to the callback routine) bytes of data from the packet (which won't necessarily be the entire packet; to capture the entire packet, you will have to provide a value for *snaplen* in your call to **pcap_open_live()** that is sufficiently large to get all of the packet's data - a value of 65535 should be sufficient on most if not all networks).

The number of packets read is returned. 0 is returned if no packets were read from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read) or if no more packets are available in a ``savefile." A return of -1 indicates an error in which case **pcap_perror()** or **pcap_geterr()** may be used to display the error text. A return of -2 indicates that the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. **If your application uses pcap_breakloop(), make sure that you explicitly check for -1 and -2, rather than just checking for a return value < 0.**

NOTE: when reading a live capture, **pcap_dispatch()** will not necessarily return when the read times out; on some platforms, the read timeout isn't supported, and, on other platforms, the timer doesn't start until at least one packet arrives. This means that the read timeout should **NOT** be used in, for example, an interactive application, to allow the packet capture loop to ``poll" for user input periodically, as there's no guarantee that **pcap_dispatch()** will return after the timeout expires.

pcap_loop() is similar to **pcap_dispatch()** except it keeps reading packets until *cnt* packets are processed or an error occurs. It does **not** return when live read timeouts

occur. Rather, specifying a non-zero read timeout to **pcap_open_live()** and then calling **pcap_dispatch()** allows the reception and processing of any packets that arrive when the timeout occurs. A negative *cnt* causes **pcap_loop()** to loop forever (or at least until an error occurs). -1 is returned on an error; 0 is returned if *cnt* is exhausted; -2 is returned if the loop terminated due to a call to **pcap_breakloop()** before any packets were processed. **If your application uses pcap_breakloop(), make sure that you explicitly check for -1 and -2, rather than just checking for a return value < 0.**

pcap_next() reads the next packet (by calling **pcap_dispatch()** with a *cnt* of 1) and returns a *u_char* pointer to the data in that packet. (The *pcap_pkthdr* struct for that packet is not supplied.) **NULL** is returned if an error occurred, or if no packets were read from a live capture (if, for example, they were discarded because they didn't pass the packet filter, or if, on platforms that support a read timeout that starts before any packets arrive, the timeout expires before any packets arrive, or if the file descriptor for the capture device is in non-blocking mode and no packets were available to be read), or if no more packets are available in a ``savefile." Unfortunately, there is no way to determine whether an error occurred or not.

pcap_next_ex() reads the next packet and returns a success/failure indication:

1	the packet was read without problems
0	packets are being read from a live capture, and the timeout expired
-1	an error occurred while reading the packet
-2	packets are being read from a ``savefile", and there are no more packets to read from the savefile.

If the packet was read without problems, the pointer pointed to by the *pkt_header* argument is set to point to the *pcap_pkthdr* struct for the packet, and the pointer pointed to by the *pkt_data* argument is set to point to the data in the packet.

pcap_breakloop() sets a flag that will force **pcap_dispatch()** or **pcap_loop()** to return rather than looping; they will return the number of packets that have been processed so far, or -2 if no packets have been processed so far.

This routine is safe to use inside a signal handler on UNIX or a console control handler on Windows, as it merely sets a flag that is checked within the loop.

The flag is checked in loops reading packets from the OS - a signal by itself will not necessarily terminate those loops - as well as in loops processing a set of packets returned by the OS. **Note that if you are catching signals on UNIX systems that support restarting system calls after a signal, and calling pcap_breakloop() in the signal handler, you must specify, when catching those signals, that system calls should NOT be restarted by that signal. Otherwise, if the signal interrupted a call reading packets in a live capture, when your signal handler returns after**

calling `pcap_breakloop()`, the call will be restarted, and the loop will not terminate until more packets arrive and the call completes.

Note that `pcap_next()` will, on some platforms, loop reading packets from the OS; that loop will not necessarily be terminated by a signal, so `pcap_breakloop()` should be used to terminate packet processing even if `pcap_next()` is being used.

`pcap_breakloop()` does not guarantee that no further packets will be processed by `pcap_dispatch()` or `pcap_loop()` after it is called; at most one more packet might be processed.

If -2 is returned from `pcap_dispatch()` or `pcap_loop()`, the flag is cleared, so a subsequent call will resume reading packets. If a positive number is returned, the flag is not cleared, so a subsequent call will return -2 and clear the flag.

`pcap_dump()` outputs a packet to the ``savefile" opened with `pcap_dump_open()`. Note that its calling arguments are suitable for use with `pcap_dispatch()` or `pcap_loop()`. If called directly, the *user* parameter is of type *pcap_dumper_t* as returned by `pcap_dump_open()`.

`pcap_compile()` is used to compile the string *str* into a filter program. *program* is a pointer to a *bpf_program* struct and is filled in by `pcap_compile()`. *optimize* controls whether optimization on the resulting code is performed. *netmask* specifies the IPv4 netmask of the network on which packets are being captured; it is used only when checking for IPv4 broadcast addresses in the filter program. If the netmask of the network on which packets are being captured isn't known to the program, or if packets are being captured on the Linux "any" pseudo-interface that can capture on more than one network, a value of 0 can be supplied; tests for IPv4 broadcast addresses won't be done correctly, but all other tests in the filter program will be OK. A return of -1 indicates an error in which case `pcap_geterr()` may be used to display the error text.

`pcap_compile_nopcap()` is similar to `pcap_compile()` except that instead of passing a pcap structure, one passes the snaplen and linktype explicitly. It is intended to be used for compiling filters for direct BPF usage, without necessarily having called `pcap_open()`. A return of -1 indicates an error; the error text is unavailable. (`pcap_compile_nopcap()` is a wrapper around `pcap_open_dead()`, `pcap_compile()`, and `pcap_close()`; the latter three routines can be used directly in order to get the error text for a compilation error.)

`pcap_setfilter()` is used to specify a filter program. *fp* is a pointer to a *bpf_program* struct, usually the result of a call to `pcap_compile()`. -1 is returned on failure, in which case `pcap_geterr()` may be used to display the error text; 0 is returned on success.

`pcap_freecode()` is used to free up allocated memory pointed to by a *bpf_program* struct generated by `pcap_compile()` when that BPF program is no longer needed, for example after it has been made the filter program for a pcap structure by a call to `pcap_setfilter()`.

`pcap_datalink()` returns the link layer type; link layer types it can return include:

DLT_NULL

BSD loopback encapsulation; the link layer header is a 4-byte field, in *host* byte order, containing a PF_ value from **socket.h** for the network-layer protocol of the packet.

Note that ``host byte order" is the byte order of the machine on which the packets are captured, and the PF_ values are for the OS of the machine on which the packets are captured; if a live capture is being done, ``host byte order" is the byte order of the machine capturing the packets, and the PF_ values are those of the OS of the machine capturing the packets, but if a ``savefile" is being read, the byte order and PF_ values are *not* necessarily those of the machine reading the capture file.

DLT_EN10MB

Ethernet (10Mb, 100Mb, 1000Mb, and up)

DLT_IEEE802

IEEE 802.5 Token Ring

DLT_ARCNET

ARCNET

DLT_SLIP

SLIP; the link layer header contains, in order:

a 1-byte flag, which is 0 for packets received by the machine and 1 for packets sent by the machine;

a 1-byte field, the upper 4 bits of which indicate the type of packet, as per RFC 1144:

0x40

an unmodified IP datagram (TYPE_IP);

0x70

an uncompressed-TCP IP datagram (UNCOMPRESSED_TCP), with that byte being the first byte of the raw IP header on the wire, containing the connection number in the protocol field;

0x80

a compressed-TCP IP datagram (COMPRESSED_TCP), with that byte being the first byte of the compressed TCP/IP datagram header;

for UNCOMPRESSED_TCP, the rest of the modified IP header, and for COMPRESSED_TCP, the compressed TCP/IP datagram header;

for a total of 16 bytes; the uncompressed IP datagram follows the header.

DLT_PPP

PPP; if the first 2 bytes are 0xff and 0x03, it's PPP in HDLC-like framing, with the PPP header following those two bytes, otherwise it's PPP without framing, and the packet begins with the PPP header.

DLT_FDDI

FDDI

DLT_ATM_RFC1483

RFC 1483 LLC/SNAP-encapsulated ATM; the packet begins with an IEEE 802.2 LLC header.

DLT_RAW

raw IP; the packet begins with an IP header.

DLT_PPP_SERIAL

PPP in HDLC-like framing, as per RFC 1662, or Cisco PPP with HDLC framing, as per section 4.3.1 of RFC 1547; the first byte will be 0xFF for PPP in HDLC-like framing, and will be 0x0F or 0x8F for Cisco PPP with HDLC framing.

DLT_PPP_ETHER

PPPoE; the packet begins with a PPPoE header, as per RFC 2516.

DLT_C_HDLC

Cisco PPP with HDLC framing, as per section 4.3.1 of RFC 1547.

DLT_IEEE802_11

IEEE 802.11 wireless LAN

DLT_FRELAY

Frame Relay

DLT_LOOP

OpenBSD loopback encapsulation; the link layer header is a 4-byte field, in *network* byte order, containing a PF_ value from OpenBSD's **socket.h** for the network-layer protocol of the packet.

Note that, if a ``savefile" is being read, those PF_ values are *not* necessarily those of the machine reading the capture file.

DLT_LINUX_SLL

Linux "cooked" capture encapsulation; the link layer header contains, in order:

a 2-byte "packet type", in network byte order, which is one of:

0

packet was sent to us by somebody else

1

packet was broadcast by somebody else

2

packet was multicast, but not broadcast, by somebody else

3

packet was sent by somebody else to somebody else

4

packet was sent by us

a 2-byte field, in network byte order, containing a Linux ARPHRD_ value for the link layer device type;

a 2-byte field, in network byte order, containing the length of the link layer address of the sender of the packet (which could be 0);

an 8-byte field containing that number of bytes of the link layer header (if there are more than 8 bytes, only the first 8 are present);

a 2-byte field containing an Ethernet protocol type, in network byte order, or containing 1 for Novell 802.3 frames without an 802.2 LLC header or 4 for frames beginning with an 802.2 LLC header.

DLT_LTALK

Apple LocalTalk; the packet begins with an AppleTalk LLAP header.

DLT_PFLOG

OpenBSD pflog; the link layer header contains, in order:

a 4-byte PF_ value, in network byte order;

a 16-character interface name;

a 2-byte rule number, in network byte order;

a 2-byte reason code, in network byte order, which is one of:

0

match

1

bad offset

2

fragment

3

short

4

normalize

memory

a 2-byte action code, in network byte order, which is one of:

0

passed

1

dropped

2

scrubbed

a 2-byte direction, in network byte order, which is one of:

0

incoming or outgoing

1

incoming

2

outgoing

DLT_PRISM_HEADER

Prism monitor mode information followed by an 802.11 header.

DLT_IP_OVER_FC

RFC 2625 IP-over-Fibre Channel, with the link-layer header being the Network_Header as described in that RFC.

DLT_SUNATM

SunATM devices; the link layer header contains, in order:

a 1-byte flag field, containing a direction flag in the uppermost bit, which is set for packets transmitted by the machine and clear for packets received by the machine, and a 4-byte traffic type in the low-order 4 bits, which is one of:

0
raw traffic
1
LANE traffic
2
LLC-encapsulated traffic
3
MARS traffic
4
IFMP traffic
5
ILMI traffic
6
Q.2931 traffic

a 1-byte VPI value;

a 2-byte VCI field, in network byte order.

DLT_IEEE802_11_RADIO

link-layer information followed by an 802.11 header - see

<http://www.shaftnet.org/~pizza/software/capturefrm.txt> for a description of the link-layer information.

DLT_ARCNET_LINUX

ARCNET, with no exception frames, reassembled packets rather than raw frames, and an extra 16-bit offset field between the destination host and type bytes.

DLT_LINUX_IRDA

Linux-IrDA packets, with a **DLT_LINUX_SLL** header followed by the IrLAP header.

pcap_list_datalinks() is used to get a list of the supported data link types of the interface associated with the pcap descriptor. **pcap_list_datalinks()** allocates an array to hold the list and sets **dlt_buf*. The caller is responsible for freeing the array. **-1** is returned on failure; otherwise, the number of data link types in the array is returned.

pcap_set_datalink() is used to set the current data link type of the pcap descriptor to the type specified by *dlt*. **-1** is returned on failure.

pcap_datalink_name_to_val() translates a data link type name, which is a **DLT_** name with the **DLT_** removed, to the corresponding data link type value. The translation is case-insensitive. **-1** is returned on failure.

pcap_datalink_val_to_name() translates a data link type value to the corresponding data link type name. NULL is returned on failure.

pcap_datalink_val_to_description() translates a data link type value to a short description of that data link type. NULL is returned on failure.

pcap_snapshot() returns the snapshot length specified when **pcap_open_live()** was called.

pcap_is_swapped() returns true if the current ``savefile" uses a different byte order than the current system.

pcap_major_version() returns the major number of the file format of the savefile;
pcap_minor_version() returns the minor number of the file format of the savefile.
The version number is stored in the header of the savefile.

pcap_file() returns the standard I/O stream of the ``savefile," if a ``savefile" was opened with **pcap_open_offline()**, or NULL, if a network device was opened with **pcap_open_live()**.

pcap_stats() returns 0 and fills in a **pcap_stat** struct. The values represent packet statistics from the start of the run to the time of the call. If there is an error or the underlying packet capture doesn't support packet statistics, -1 is returned and the error text can be obtained with **pcap_perror()** or **pcap_geterr()**. **pcap_stats()** is supported only on live captures, not on ``savefiles"; no statistics are stored in ``savefiles", so no statistics are available when reading from a ``savefile".

pcap_fileno() returns the file descriptor number from which captured packets are read, if a network device was opened with **pcap_open_live()**, or -1, if a ``savefile" was opened with **pcap_open_offline()**.

pcap_perror() prints the text of the last pcap library error on **stderr**, prefixed by *prefix*.

pcap_geterr() returns the error text pertaining to the last pcap library error. **NOTE:** the pointer it returns will no longer point to a valid error message string after the **pcap_t** passed to it is closed; you must use or copy the string before closing the **pcap_t**.

pcap_strerror() is provided in case [strerror\(1\)](#) isn't available.

pcap_lib_version() returns a pointer to a string giving information about the version of the libpcap library being used; note that it contains more information than just a version number.

pcap_close() closes the files associated with *p* and deallocates resources.

pcap_dump_file() returns the standard I/O stream of the ``savefile" opened by **pcap_dump_open()**.

pcap_dump_flush() flushes the output buffer to the ``savefile," so that any packets written with **pcap_dump()** but not yet written to the ``savefile" will be written. **-1** is returned on error, 0 on success.

pcap_dump_close() closes the ``savefile."

SEE ALSO

[tcpdump\(1\)](#), [tcplice\(1\)](#)

AUTHORS

The original authors are:

Van Jacobson, Craig Leres and Steven McCanne, all of the Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

The current version is available from "The Tcpdump Group"'s Web site at

<http://www.tcpdump.org/>

BUGS

Please send problems, bugs, questions, desirable enhancements, etc. to:

tcpdump-workers@tcpdump.org

Please send source code contributions, etc. to:

patches@tcpdump.org

Index

[NAME](#)

[SYNOPSIS](#)

[DESCRIPTION](#)

[ROUTINES](#)

[SEE ALSO](#)

[AUTHORS](#)

[BUGS](#)

This document was created by [man2html](#), using the manual pages.

Time: 21:43:43 GMT, January 15, 2004

Network Working Group
Request for Comments: 3954
Category: Informational

B. Claise, Ed.
Cisco Systems
October 2004

Cisco Systems NetFlow Services Export Version 9

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2004).

IESG Note

This RFC documents the NetFlow services export protocol Version 9 as it was when submitted to the IETF as a basis for further work in the IPFIX WG.

This RFC itself is not a candidate for any level of Internet Standard. The IETF disclaims any knowledge of the fitness of this RFC for any purpose, and in particular notes that it has not had complete IETF review for such things as security, congestion control, or inappropriate interaction with deployed protocols. The RFC Editor has chosen to publish this document at its discretion.

Abstract

This document specifies the data export format for version 9 of Cisco Systems' NetFlow services, for use by implementations on the network elements and/or matching collector programs. The version 9 export format uses templates to provide access to observations of IP packet flows in a flexible and extensible manner. A template defines a collection of fields, with corresponding descriptions of structure and semantics.

Table of Contents

1.	Introduction.	2
2.	Terminology	4
2.1.	Terminology Summary Table	6
3.	NetFlow High-Level Picture on the Exporter.	6
3.1.	The NetFlow Process on the Exporter	6
3.2.	Flow Expiration	7

3.3.	Transport Protocol	7
4.	Packet Layout	8
5.	Export Packet Format	9
5.1.	Header Format	9
5.2.	Template FlowSet Format	11
5.3.	Data FlowSet Format	13
6.	Options	14
6.1.	Options Template FlowSet Format	14
6.2.	Options Data Record Format	16
7.	Template Management	17
8.	Field Type Definitions	18
9.	The Collector Side	25
10.	Security Considerations	26
10.1.	Disclosure of Flow Information Data	26
10.2.	Forgery of Flow Records or Template Records	26
10.3.	Attacks on the NetFlow Collector	27
11.	Examples	27
11.1.	Packet Header Example	28
11.2.	Template FlowSet Example	28
11.3.	Data FlowSet Example	29
11.4.	Options Template FlowSet Example	30
11.5.	Data FlowSet with Options Data Records Example	30
12.	References	31
12.1.	Normative References	31
12.2.	Informative References	31
13.	Authors	31
14.	Acknowledgments	31
15.	Authors' Addresses	32
16.	Full Copyright Statement	33

1. Introduction

Cisco Systems' NetFlow services provide network administrators with access to IP flow information from their data networks. Network elements (routers and switches) gather flow data and export it to collectors. The collected data provides fine-grained metering for highly flexible and detailed resource usage accounting.

A flow is defined as a unidirectional sequence of packets with some common properties that pass through a network device. These collected flows are exported to an external device, the NetFlow collector. Network flows are highly granular; for example, flow records include details such as IP addresses, packet and byte counts, timestamps, Type of Service (ToS), application ports, input and output interfaces, etc.

Exported NetFlow data is used for a variety of purposes, including enterprise accounting and departmental chargebacks, ISP billing, data

warehousing, network monitoring, capacity planning, application monitoring and profiling, user monitoring and profiling, security analysis, and data mining for marketing purposes.

This document specifies NetFlow version 9. It describes the implementation specifications both from network element and NetFlow collector points of view. These specifications should help the deployment of NetFlow version 9 across different platforms and different vendors by limiting the interoperability risks. The NetFlow export format version 9 uses templates to provide access to observations of IP packet flows in a flexible and extensible manner.

A template defines a collection of fields, with corresponding descriptions of structure and semantics.

The template-based approach provides the following advantages:

- New fields can be added to NetFlow flow records without changing the structure of the export record format. With previous NetFlow versions, adding a new field in the flow record implied a new version of the export protocol format and a new version of the NetFlow collector that supported the parsing of the new export protocol format.
- Templates that are sent to the NetFlow collector contain the structural information about the exported flow record fields; therefore, if the NetFlow collector does not understand the semantics of new fields, it can still interpret the flow record.
- Because the template mechanism is flexible, it allows the export of only the required fields from the flows to the NetFlow collector. This helps to reduce the exported flow data volume and provides possible memory savings for the exporter and NetFlow collector. Sending only the required information can also reduce network load.

The IETF IPFIX Working Group (IP Flow Information eXport) is developing a new protocol, based on the version 9 of Cisco Systems' NetFlow services. Some enhancements in different domains (congestion aware transport protocol, built-in security, etc...) have been incorporated in this new IPFIX protocol. Refer to the IPFIX Working Group documents for more details.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14, RFC 2119 [RFC2119].

2. Terminology

Various terms used in this document are described in this section. Note that the terminology summary table in Section 2.1 gives a quick overview of the relationships between some of the different terms defined.

Observation Point

An Observation Point is a location in the network where IP packets can be observed; for example, one or a set of interfaces on a network device like a router. Every Observation Point is associated with an Observation Domain.

Observation Domain

The set of Observation Points that is the largest aggregatable set of flow information at the network device with NetFlow services enabled is termed an Observation Domain. For example, a router line card composed of several interfaces with each interface being an Observation Point.

IP Flow or Flow

An IP Flow, also called a Flow, is defined as a set of IP packets passing an Observation Point in the network during a certain time interval. All packets that belong to a particular Flow have a set of common properties derived from the data contained in the packet and from the packet treatment at the Observation Point.

Flow Record

A Flow Record provides information about an IP Flow observed at an Observation Point. In this document, the Flow Data Records are also referred to as NetFlow services data and NetFlow data.

Exporter

A device (for example, a router) with the NetFlow services enabled, the Exporter monitors packets entering an Observation Point and creates Flows from these packets. The information from these Flows is exported in the form of Flow Records to the NetFlow Collector.

NetFlow Collector

The NetFlow Collector receives Flow Records from one or more Exporters. It processes the received Export Packet(s); that is, it parses and stores the Flow Record information. Flow Records can be optionally aggregated before being stored on the hard disk. The NetFlow Collector is also referred to as the Collector in this document.

Export Packet

An Export Packet is a packet originating at the Exporter that carries the Flow Records of this Exporter and whose destination is the NetFlow Collector.

Packet Header

The Packet Header is the first part of an Export Packet. The Packet Header provides basic information about the packet such as the NetFlow version, number of records contained within the packet, and sequence numbering.

Template Record

A Template Record defines the structure and interpretation of fields in a Flow Data Record.

Flow Data Record

A Flow Data Record is a data record that contains values of the Flow parameters corresponding to a Template Record.

Options Template Record

An Options Template Record defines the structure and interpretation of fields in an Options Data Record, including defining the scope within which the Options Data Record is relevant.

Options Data Record

The data record that contains values and scope information of the Flow measurement parameters, corresponding to an Options Template Record.

FlowSet

FlowSet is a generic term for a collection of Flow Records that have a similar structure. In an Export Packet, one or more FlowSets follow the Packet Header. There are three different types of FlowSets: Template FlowSet, Options Template FlowSet, and Data FlowSet.

Template FlowSet

A Template FlowSet is one or more Template Records that have been grouped together in an Export Packet.

Options Template FlowSet

An Options Template FlowSet is one or more Options Template Records that have been grouped together in an Export Packet.

Data FlowSet

A Data FlowSet is one or more records, of the same type, that are grouped together in an Export Packet. Each record is either a Flow Data Record or an Options Data Record previously defined by a Template Record or an Options Template Record.

2.1. Terminology Summary Table

FlowSet	Contents	
	Template Record	Data Record
Data FlowSet	/	Flow Data Record(s) or Options Data Record(s)
Template FlowSet	Template Record(s)	/
Options Template FlowSet	Options Template Record(s)	/

A Data FlowSet is composed of an Options Data Record(s) or Flow Data Record(s). No Template Record is included. A Template Record defines the Flow Data Record, and an Options Template Record defines the Options Data Record.

A Template FlowSet is composed of Template Record(s). No Flow or Options Data Record is included.

An Options Template FlowSet is composed of Options Template Record(s). No Flow or Options Data Record is included.

3. NetFlow High-Level Picture on the Exporter

3.1. The NetFlow Process on the Exporter

The NetFlow process on the Exporter is responsible for the creation of Flows from the observed IP packets. The details of this process are beyond the scope of this document.

3.2. Flow Expiration

A Flow is considered to be inactive if no packets belonging to the Flow have been observed at the Observation Point for a given timeout. If any packet is seen within the timeout, the flow is considered an active flow. A Flow can be exported under the following conditions:

1. If the Exporter can detect the end of a Flow. For example, if the FIN or RST bit is detected in a TCP [RFC793] connection, the Flow Record is exported.
2. If the Flow has been inactive for a certain period of time. This inactivity timeout SHOULD be configurable at the Exporter, with a minimum value of 0 for an immediate expiration.
3. For long-lasting Flows, the Exporter SHOULD export the Flow Records on a regular basis. This timeout SHOULD be configurable at the Exporter.
4. If the Exporter experiences internal constraints, a Flow MAY be forced to expire prematurely; for example, counters wrapping or low memory.

3.3. Transport Protocol

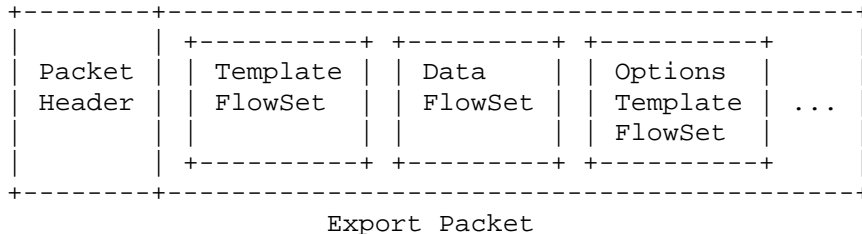
To achieve efficiency in terms of processing at the Exporter while handling high volumes of Export Packets, the NetFlow Export Packets are encapsulated into UDP [RFC768] datagrams for export to the NetFlow Collector. However, NetFlow version 9 has been designed to be transport protocol independent. Hence, it can also operate over congestion-aware protocols such as SCTP [RFC2960].

Note that the Exporter can export to multiple Collectors, using independent transport protocols.

UDP [RFC768] is a non congestion-aware protocol, so when deploying NetFlow version 9 in a congestion-sensitive environment, make the connection between Exporter and NetFlow Collector through a dedicated link. This ensures that any burstiness in the NetFlow traffic affects only this dedicated link. When the NetFlow Collector can not be placed within a one-hop distance from the Exporter or when the export path from the Exporter to the NetFlow Collector can not be exclusively used for the NetFlow Export Packets, the export path should be designed so that it can always sustain the maximum burstiness of NetFlow traffic from the Exporter. Note that the congestion can occur on the Exporter in case the export path speed is too low.

4. Packet Layout

An Export Packet consists of a Packet Header followed by one or more FlowSets. The FlowSets can be any of the possible three types: Template, Data, or Options Template.



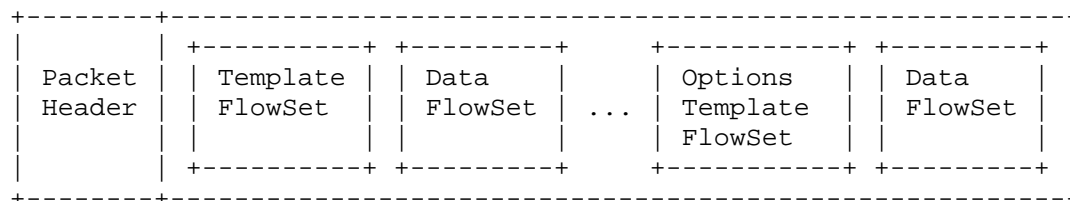
A FlowSet ID is used to distinguish the different types of FlowSets. FlowSet IDs lower than 256 are reserved for special FlowSets, such as the Template FlowSet (ID 0) and the Options Template FlowSet (ID 1). The Data FlowSets have a FlowSet ID greater than 255.

The format of the Template, Data, and Options Template FlowSets will be discussed later in this document. The Exporter MUST code all binary integers of the Packet Header and the different FlowSets in network byte order (also known as the big-endian byte ordering).

Following are some examples of export packets:

1. An Export Packet consisting of interleaved Template, Data, and Options Template FlowSets. Example: a newly created Template is exported as soon as possible. So if there is already an Export Packet with a Data FlowSet that is being prepared for export, the Template and Option FlowSets are also interleaved with this information, subject to availability of space.

Export Packet:



2. An Export Packet consisting entirely of Data FlowSets. Example: after the appropriate Template Records have been defined and transmitted to the NetFlow Collector device, the majority of Export Packets consists solely of Data FlowSets.

Export Packet:

+-----+										
 Packet Header 	+-----+			+-----+			+-----+			
	Data FlowSet			... FlowSet			... FlowSet			
	+-----+			+-----+			+-----+			
+-----+										

3. An Export Packet consisting entirely of Template and Options Template FlowSets. Example: the Exporter MAY transmit a packet containing Template and Options Template FlowSets periodically to help ensure that the NetFlow Collector has the correct Template Records and Options Template Records when the corresponding Flow Data records are received.

Export Packet:

+-----+									
	Packet Header	+-----+		+-----+		+-----+			
		Template FlowSet		... Template FlowSet		Options Template FlowSet			
+-----+									

5. Export Packet Format

5.1. Header Format

The Packet Header format is specified as:

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
+-----+										+-----+										+-----+										+-----+									
Version Number										Count																													
+-----+										+-----+										+-----+										+-----+									
sysUpTime																																							
+-----+										+-----+										+-----+										+-----+									
UNIX Secs																																							
+-----+										+-----+										+-----+										+-----+									
Sequence Number																																							
+-----+										+-----+										+-----+										+-----+									
Source ID																																							
+-----+										+-----+										+-----+										+-----+									

Packet Header Field Descriptions

Version

Version of Flow Record format exported in this packet. The value of this field is 9 for the current version.

Count

The total number of records in the Export Packet, which is the sum of Options FlowSet records, Template FlowSet records, and Data FlowSet records.

sysUpTime

Time in milliseconds since this device was first booted.

UNIX Secs

Time in seconds since 0000 UTC 1970, at which the Export Packet leaves the Exporter.

Sequence Number

Incremental sequence counter of all Export Packets sent from the current Observation Domain by the Exporter. This value MUST be cumulative, and SHOULD be used by the Collector to identify whether any Export Packets have been missed.

Source ID

A 32-bit value that identifies the Exporter Observation Domain. NetFlow Collectors SHOULD use the combination of the source IP address and the Source ID field to separate different export streams originating from the same Exporter.

5.2. Template FlowSet Format

One of the essential elements in the NetFlow format is the Template FlowSet. Templates greatly enhance the flexibility of the Flow Record format because they allow the NetFlow Collector to process Flow Records without necessarily knowing the interpretation of all the data in the Flow Record. The format of the Template FlowSet is as follows:

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
FlowSet ID = 0										Length																													
Template ID 256										Field Count																													
Field Type 1										Field Length 1																													
Field Type 2										Field Length 2																													
...										...																													
Field Type N										Field Length N																													
Template ID 257										Field Count																													
Field Type 1										Field Length 1																													
Field Type 2										Field Length 2																													
...										...																													
Field Type M										Field Length M																													
...										...																													
Template ID K										Field Count																													
...										...																													

Template FlowSet Field Descriptions

FlowSet ID

FlowSet ID value of 0 is reserved for the Template FlowSet.

Length

Total length of this FlowSet. Because an individual Template FlowSet MAY contain multiple Template Records, the Length value MUST be used to determine the position of the next FlowSet record, which could be any type of FlowSet. Length is the sum of the lengths of the FlowSet ID, the Length itself, and all Template Records within this FlowSet.

Template ID

Each of the newly generated Template Records is given a unique Template ID. This uniqueness is local to the Observation Domain that generated the Template ID. Template IDs 0-255 are reserved for Template FlowSets, Options FlowSets, and other reserved FlowSets yet to be created. Template IDs of Data FlowSets are numbered from 256 to 65535.

Field Count

Number of fields in this Template Record. Because a Template FlowSet usually contains multiple Template Records, this field allows the Collector to determine the end of the current Template Record and the start of the next.

Field Type

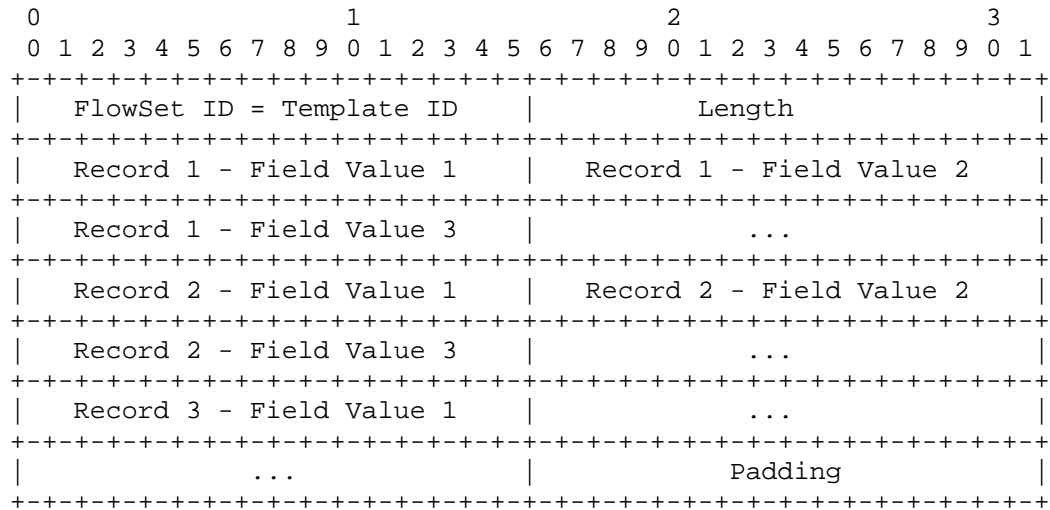
A numeric value that represents the type of the field. Refer to the "Field Type Definitions" section.

Field Length

The length of the corresponding Field Type, in bytes. Refer to the "Field Type Definitions" section.

5.3. Data FlowSet Format

The format of the Data FlowSet is as follows:



Data FlowSet Field Descriptions

FlowSet ID = Template ID

Each Data FlowSet is associated with a FlowSet ID. The FlowSet ID maps to a (previously generated) Template ID. The Collector MUST use the FlowSet ID to find the corresponding Template Record and decode the Flow Records from the FlowSet.

Length

The length of this FlowSet. Length is the sum of the lengths of the FlowSet ID, Length itself, all Flow Records within this FlowSet, and the padding bytes, if any.

Record N - Field Value M

The remainder of the Data FlowSet is a collection of Flow Data Record(s), each containing a set of field values. The Type and Length of the fields have been previously defined in the Template Record referenced by the FlowSet ID or Template ID.

Padding

The Exporter SHOULD insert some padding bytes so that the subsequent FlowSet starts at a 4-byte aligned boundary. It is important to note that the Length field includes the padding bytes. Padding SHOULD be using zeros.

Interpretation of the Data FlowSet format can be done only if the Template FlowSet corresponding to the Template ID is available at the Collector.

6. Options

6.1. Options Template FlowSet Format

The Options Template Record (and its corresponding Options Data Record) is used to supply information about the NetFlow process configuration or NetFlow process specific data, rather than supplying information about IP Flows.

For example, the Options Template FlowSet can report the sample rate of a specific interface, if sampling is supported, along with the sampling method used.

The format of the Options Template FlowSet follows.

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
FlowSet ID = 1										Length																													
Template ID										Option Scope Length																													
Option Length										Scope 1 Field Type																													
Scope 1 Field Length										...																													
Scope N Field Length										Option 1 Field Type																													
Option 1 Field Length										...																													
Option M Field Length										Padding																													

Options Template FlowSet Field Definitions

FlowSet ID = 1

A FlowSet ID value of 1 is reserved for the Options Template.

Length

Total length of this FlowSet. Each Options Template FlowSet MAY contain multiple Options Template Records. Thus, the Length value MUST be used to determine the position of the next FlowSet record, which could be either a Template FlowSet or Data FlowSet.

Length is the sum of the lengths of the FlowSet ID, the Length itself, and all Options Template Records within this FlowSet Template ID.

Template ID

Template ID of this Options Template. This value is greater than 255.

Option Scope Length

The length in bytes of any Scope field definition contained in the Options Template Record (The use of "Scope" is described below).

Option Length

The length (in bytes) of any options field definitions contained in this Options Template Record.

Scope 1 Field Type

The relevant portion of the Exporter/NetFlow process to which the Options Template Record refers.

Currently defined values are:

- 1 System
- 2 Interface
- 3 Line Card
- 4 Cache
- 5 Template

For example, the NetFlow process can be implemented on a per-interface basis, so if the Options Template Record were reporting on how the NetFlow process is configured, the Scope for the report would be 2 (interface). The associated interface ID would then be carried in the associated Options Data FlowSet. The Scope can be limited further by listing multiple scopes that all must match at the same time. Note that the Scope fields always precede the Option fields.

Scope 1 Field Length

The length (in bytes) of the Scope field, as it would appear in an Options Data Record.

Option 1 Field Type

A numeric value that represents the type of field that would appear in the Options Template Record. Refer to the Field Type Definitions section.

Option 1 Field Length

The length (in bytes) of the Option field.

Padding

The Exporter SHOULD insert some padding bytes so that the subsequent FlowSet starts at a 4-byte aligned boundary. It is important to note that the Length field includes the padding bytes. Padding SHOULD be using zeros.

6.2. Options Data Record Format

The Options Data Records are sent in Data FlowSets, on a regular basis, but not with every Flow Data Record. How frequently these Options Data Records are exported is configurable. See the "Templates Management" section for more details.

The format of the Data FlowSet containing Options Data Records follows.

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|  FlowSet ID = Template ID  |          Length          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  Record 1 - Scope 1 Value   |Record 1 - Option Field 1 Value|
+-----+-----+-----+-----+-----+-----+-----+-----+
|Record 1 - Option Field 2 Value|          ...          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  Record 2 - Scope 1 Value   |Record 2 - Option Field 1 Value|
+-----+-----+-----+-----+-----+-----+-----+-----+
|Record 2 - Option Field 2 Value|          ...          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|  Record 3 - Scope 1 Value   |Record 3 - Option Field 1 Value|
+-----+-----+-----+-----+-----+-----+-----+-----+
|Record 3 - Option Field 2 Value|          ...          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          ...          |          Padding          |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Options Data Records of the Data FlowSet Field Descriptions

FlowSet ID = Template ID

A FlowSet ID precedes each group of Options Data Records within a Data FlowSet. The FlowSet ID maps to a previously generated Template ID corresponding to this Options Template Record. The Collector MUST use the FlowSet ID to map the appropriate type and length to any field values that follow.

Length

The length of this FlowSet. Length is the sum of the lengths of the FlowSet ID, Length itself, all the Options Data Records within this FlowSet, and the padding bytes, if any.

Record N - Option Field M Value

The remainder of the Data FlowSet is a collection of Flow Records, each containing a set of scope and field values. The type and length of the fields were previously defined in the Options Template Record referenced by the FlowSet ID or Template ID.

Padding

The Exporter SHOULD insert some padding bytes so that the subsequent FlowSet starts at a 4-byte aligned boundary. It is important to note that the Length field includes the padding bytes. Padding SHOULD be using zeros.

The Data FlowSet format can be interpreted only if the Options Template FlowSet corresponding to the Template ID is available at the Collector.

7. Template Management

Flow Data records that correspond to a Template Record MAY appear in the same and/or subsequent Export Packets. The Template Record is not necessarily carried in every Export Packet. As such, the NetFlow Collector MUST store the Template Record to interpret the corresponding Flow Data Records that are received in subsequent data packets.

A NetFlow Collector that receives Export Packets from several Observation Domains from the same Exporter MUST be aware that the uniqueness of the Template ID is not guaranteed across Observation Domains.

The Template IDs must remain constant for the life of the NetFlow process on the Exporter. If the Exporter or the NetFlow process restarts for any reason, all information about Templates will be lost and new Template IDs will be created. Template IDs are thus not guaranteed to be consistent across an Exporter or NetFlow process restart.

A newly created Template record is assigned an unused Template ID from the Exporter. If the template configuration is changed, the current Template ID is abandoned and SHOULD NOT be reused until the

NetFlow process or Exporter restarts. If a Collector should receive a new definition for an already existing Template ID, it MUST discard the previous template definition and use the new one.

If a configured Template Record on the Exporter is deleted, and re-configured with exactly the same parameters, the same Template ID COULD be reused.

The Exporter sends the Template FlowSet and Options Template FlowSet under the following conditions:

1. After a NetFlow process restarts, the Exporter MUST NOT send any Data FlowSet without sending the corresponding Template FlowSet and the required Options Template FlowSet in a previous packet or including it in the same Export Packet. It MAY transmit the Template FlowSet and Options Template FlowSet, without any Data FlowSets, in advance to help ensure that the Collector will have the correct Template Record before receiving the first Flow or Options Data Record.
2. In the event of configuration changes, the Exporter SHOULD send the new template definitions at an accelerated rate. In such a case, it MAY transmit the changed Template Record(s) and Options Template Record(s), without any data, in advance to help ensure that the Collector will have the correct template information before receiving the first data.
3. On a regular basis, the Exporter MUST send all the Template Records and Options Template Records to refresh the Collector. Template IDs have a limited lifetime at the Collector and MUST be periodically refreshed. Two approaches are taken to make sure that Templates get refreshed at the Collector:
 - * Every N number of Export Packets.
 - * On a time basis, so every N number of minutes.Both options MUST be configurable by the user on the Exporter. When one of these expiry conditions is met, the Exporter MUST send the Template FlowSet and Options Template.
4. In the event of a clock configuration change on the Exporter, the Exporter SHOULD send the template definitions at an accelerated rate.

8. Field Type Definitions

The following table describes all the field type definitions that an Exporter MAY support. The fields are a selection of Packet Header fields, lookup results (for example, the autonomous system numbers or the subnet masks), and properties of the packet such as length.

Field Type	Value	Length (bytes)	Description
IN_BYTES	1	N	Incoming counter with length N x 8 bits for the number of bytes associated with an IP Flow. By default N is 4
IN_PKTS	2	N	Incoming counter with length N x 8 bits for the number of packets associated with an IP Flow. By default N is 4
FLows	3	N	Number of Flows that were aggregated; by default N is 4
PROTOCOL	4	1	IP protocol byte
TOS	5	1	Type of service byte setting when entering the incoming interface
TCP_FLAGS	6	1	TCP flags; cumulative of all the TCP flags seen in this Flow
L4_SRC_PORT	7	2	TCP/UDP source port number (for example, FTP, Telnet, or equivalent)
IPv4_SRC_ADDR	8	4	IPv4 source address
SRC_MASK	9	1	The number of contiguous bits in the source subnet mask (i.e., the mask in slash notation)
INPUT_SNMP	10	N	Input interface index. By default N is 2, but higher values can be used
L4_DST_PORT	11	2	TCP/UDP destination port number (for example, FTP, Telnet, or equivalent)

IPV4_DST_ADDR	12	4	IPv4 destination address
DST_MASK	13	1	The number of contiguous bits in the destination subnet mask (i.e., the mask in slash notation)
OUTPUT_SNMP	14	N	Output interface index. By default N is 2, but higher values can be used
IPV4_NEXT_HOP	15	4	IPv4 address of the next-hop router
SRC_AS	16	N	Source BGP autonomous system number where N could be 2 or 4. By default N is 2
DST_AS	17	N	Destination BGP autonomous system number where N could be 2 or 4. By default N is 2
BGP_IPV4_NEXT_HOP	18	4	Next-hop router's IP address in the BGP domain
MUL_DST_PKTS	19	N	IP multicast outgoing packet counter with length N x 8 bits for packets associated with the IP Flow. By default N is 4
MUL_DST_BYTES	20	N	IP multicast outgoing Octet (byte) counter with length N x 8 bits for the number of bytes associated with the IP Flow. By default N is 4
LAST_SWITCHED	21	4	sysUptime in msec at which the last packet of this Flow was switched
FIRST_SWITCHED	22	4	sysUptime in msec at which the first packet of this Flow was switched

OUT_BYTES	23	N	Outgoing counter with length N x 8 bits for the number of bytes associated with an IP Flow. By default N is 4
OUT_PKTS	24	N	Outgoing counter with length N x 8 bits for the number of packets associated with an IP Flow. By default N is 4
IPV6_SRC_ADDR	27	16	IPv6 source address
IPV6_DST_ADDR	28	16	IPv6 destination address
IPV6_SRC_MASK	29	1	Length of the IPv6 source mask in contiguous bits
IPV6_DST_MASK	30	1	Length of the IPv6 destination mask in contiguous bits
IPV6_FLOW_LABEL	31	3	IPv6 flow label as per RFC 2460 definition
ICMP_TYPE	32	2	Internet Control Message Protocol (ICMP) packet type; reported as ICMP Type * 256 + ICMP code
MUL_IGMP_TYPE	33	1	Internet Group Management Protocol (IGMP) packet type
SAMPLING_INTERVAL	34	4	When using sampled NetFlow, the rate at which packets are sampled; for example, a value of 100 indicates that one of every hundred packets is sampled
SAMPLING_ALGORITHM	35	1	For sampled NetFlow platform-wide: 0x01 deterministic sampling 0x02 random sampling Use in connection with SAMPLING_INTERVAL

			Timeout value (in seconds)
FLOW_ACTIVE_TIMEOUT	36	2	for active flow entries in the NetFlow cache
FLOW_INACTIVE_TIMEOUT	37	2	Timeout value (in seconds) for inactive Flow entries in the NetFlow cache
ENGINE_TYPE	38	1	Type of Flow switching engine (route processor, linecard, etc...)
ENGINE_ID	39	1	ID number of the Flow switching engine
TOTAL_BYTES_EXP	40	N	Counter with length N x 8 bits for the number of bytes exported by the Observation Domain. By default N is 4
TOTAL_PKTS_EXP	41	N	Counter with length N x 8 bits for the number of packets exported by the Observation Domain. By default N is 4
TOTAL_FLOWS_EXP	42	N	Counter with length N x 8 bits for the number of Flows exported by the Observation Domain. By default N is 4
MPLS_TOP_LABEL_TYPE	46	1	MPLS Top Label Type: 0x00 UNKNOWN 0x01 TE-MIDPT 0x02 ATOM 0x03 VPN 0x04 BGP 0x05 LDP
MPLS_TOP_LABEL_IP_ADDR	47	4	Forwarding Equivalent Class corresponding to the MPLS Top Label
FLOW_SAMPLER_ID	48	1	Identifier shown in "show flow-sampler"

FLOW_SAMPLER_MODE	49	1	The type of algorithm used for sampling data: 0x02 random sampling Use in connection with FLOW_SAMPLER_MODE
FLOW_SAMPLER_RANDOM_INTERVAL	50	4	Packet interval at which to sample. Use in connection with FLOW_SAMPLER_MODE
DST_TOS	55	1	Type of Service byte setting when exiting outgoing interface
SRC_MAC	56	6	Source MAC Address
DST_MAC	57	6	Destination MAC Address
SRC_VLAN	58	2	Virtual LAN identifier associated with ingress interface
DST_VLAN	59	2	Virtual LAN identifier associated with egress interface
IP_PROTOCOL_VERSION	60	1	Internet Protocol Version Set to 4 for IPv4, set to 6 for IPv6. If not present in the template, then version 4 is assumed
DIRECTION	61	1	Flow direction: 0 - ingress flow 1 - egress flow
IPV6_NEXT_HOP	62	16	IPv6 address of the next-hop router
BGP_IPV6_NEXT_HOP	63	16	Next-hop router in the BGP domain
IPV6_OPTION_HEADERS	64	4	Bit-encoded field identifying IPv6 option headers found in the flow
MPLS_LABEL_1	70	3	MPLS label at position 1 in the stack

MPLS_LABEL_2	71	3	MPLS label at position 2 in the stack
MPLS_LABEL_3	72	3	MPLS label at position 3 in the stack
MPLS_LABEL_4	73	3	MPLS label at position 4 in the stack
MPLS_LABEL_5	74	3	MPLS label at position 5 in the stack
MPLS_LABEL_6	75	3	MPLS label at position 6 in the stack
MPLS_LABEL_7	76	3	MPLS label at position 7 in the stack
MPLS_LABEL_8	77	3	MPLS label at position 8 in the stack
MPLS_LABEL_9	78	3	MPLS label at position 9 in the stack
MPLS_LABEL_10	79	3	MPLS label at position 10 in the stack

The value field is a numeric identifier for the field type. The following value fields are reserved for proprietary field types: 25, 26, 43 to 45, 51 to 54, and 65 to 69.

When extensibility is required, the new field types will be added to the list. The new field types have to be updated on the Exporter and Collector but the NetFlow export format would remain unchanged. Refer to the latest documentation at <http://www.cisco.com> for the newly updated list.

In some cases the size of a field type is fixed by definition, for example PROTOCOL, or IPV4_SRC_ADDR. However in other cases they are defined as a variant type. This improves the memory efficiency in the collector and reduces the network bandwidth requirement between the Exporter and the Collector. As an example, in the case IN_BYTES, on an access router it might be sufficient to use a 32 bit counter (N = 4), whilst on a core router a 64 bit counter (N = 8) would be required.

All counters and counter-like objects are unsigned integers of size N * 8 bits.

9. The Collector Side

The Collector receives Template Records from the Exporter, normally before receiving Flow Data Records (or Options Data Records). The Flow Data Records (or Options Data Records) can then be decoded and stored locally on the devices. If the Template Records have not been received at the time Flow Data Records (or Options Data Records) are received, the Collector SHOULD store the Flow Data Records (or Options Data Records) and decode them after the Template Records are received. A Collector device MUST NOT assume that the Data FlowSet and the associated Template FlowSet (or Options Template FlowSet) are exported in the same Export Packet.

The Collector MUST NOT assume that one and only one Template FlowSet is present in an Export Packet.

The life of a template at the Collector is limited to a fixed refresh timeout. Templates not refreshed from the Exporter within the timeout are expired at the Collector. The Collector MUST NOT attempt to decode the Flow or Options Data Records with an expired Template. At any given time the Collector SHOULD maintain the following for all the current Template Records and Options Template Records: Exporter, Observation Domain, Template ID, Template Definition, Last Received.

Note that the Observation Domain is identified by the Source ID field from the Export Packet.

In the event of a clock configuration change on the Exporter, the Collector SHOULD discard all Template Records and Options Template Records associated with that Exporter, in order for Collector to learn the new set of fields: Exporter, Observation Domain, Template ID, Template Definition, Last Received.

Template IDs are unique per Exporter and per Observation Domain.

If the Collector receives a new Template Record (for example, in the case of an Exporter restart) it MUST immediately override the existing Template Record.

Finally, note that the Collector MUST accept padding in the Data FlowSet and Options Template FlowSet, which means for the Flow Data Records, the Options Data Records and the Template Records. Refer to the terminology summary table in Section 2.1.

10. Security Considerations

The NetFlow version 9 protocol was designed with the expectation that the Exporter and Collector would remain within a single private network. However the NetFlow version 9 protocol might be used to transport Flow Records over the public Internet which exposes the Flow Records to a number of security risks. For example an attacker might capture, modify or insert Export Packets. There is therefore a risk that IP Flow information might be captured or forged, or that attacks might be directed at the NetFlow Collector.

The designers of NetFlow Version 9 did not impose any confidentiality, integrity or authentication requirements on the protocol because this reduced the efficiency of the implementation and it was believed at the time that the majority of deployments would confine the Flow Records to private networks, with the Collector(s) and Exporter(s) in close proximity.

The IPFIX protocol (IP Flow Information eXport), which has chosen the NetFlow version 9 protocol as the base protocol, addresses the security considerations discussed in this section. See the security section of IPFIX requirement draft [RFC3917] for more information.

10.1. Disclosure of Flow Information Data

Because the NetFlow Version 9 Export Packets are not encrypted, the observation of Flow Records can give an attacker information about the active flows in the network, communication endpoints and traffic patterns. This information can be used both to spy on user behavior and to plan and conceal future attacks.

The information that an attacker could derive from the interception of Flow Records depends on the Flow definition. For example, a Flow Record containing the source and destination IP addresses might reveal privacy sensitive information regarding the end user's activities, whilst a Flow Record only containing the source and destination IP network would be less revealing.

10.2. Forgery of Flow Records or Template Records

If Flow Records are used in accounting and/or security applications, there may be a strong incentive to forge exported Flow Records (for example to defraud the service provider, or to prevent the detection of an attack). This can be done either by altering the Flow Records on the path between the Observer and the Collector, or by injecting forged Flow Records that pretend to be originated by the Exporter.

An attacker could forge Templates and/or Options Templates and thereby try to confuse the NetFlow Collector, rendering it unable to decode the Export Packets.

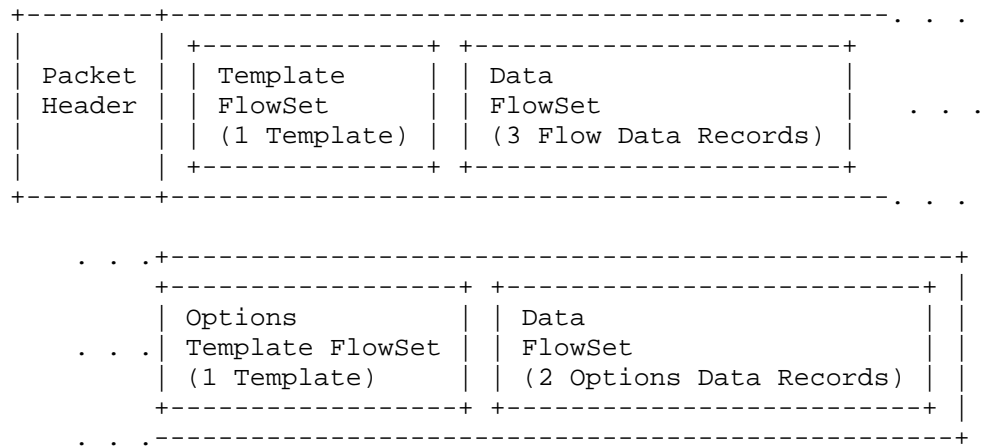
10.3. Attacks on the NetFlow Collector

Denial of service attacks on the NetFlow Collector can consume so many resources from the machine that, the Collector is unable to capture or decode some NetFlow Export Packets. Such hazards are not explicitly addressed by the NetFlow Version 9 protocol, although the normal methods used to protect a server from a DoS attack will mitigate the problem.

11. Examples

Let us consider the example of an Export Packet composed of a Template FlowSet, a Data FlowSet (which contains three Flow Data Records), an Options Template FlowSet, and a Data FlowSet (which contains two Options Data Records).

Export Packet:



11.1. Packet Header Example

The Packet Header is composed of:

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Version = 9      |      Count = 7      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               sysUpTime       |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               UNIX Secs       |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Sequence Number  |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Source ID        |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

11.2. Template FlowSet Example

We want to report the following Field Types:

- The source IP address (IPv4), so the length is 4
- The destination IP address (IPv4), so the length is 4
- The next-hop IP address (IPv4), so the length is 4
- The number of bytes of the Flow
- The number of packets of the Flow

Therefore, the Template FlowSet is composed of the following:

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|      FlowSet ID = 0      |      Length = 28 bytes      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Template ID 256     |      Field Count = 5        |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      IP_SRC_ADDR = 8     |      Field Length = 4      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      IP_DST_ADDR = 12    |      Field Length = 4      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      IP_NEXT_HOP = 15    |      Field Length = 4      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      IN_PKTS = 2         |      Field Length = 4      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      IN_BYTES = 1        |      Field Length = 4      |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

11.3. Data FlowSet Example

In this example, we report the following three Flow Records:

Src IP addr.	Dst IP addr.	Next Hop addr.	Packet Number	Bytes Number
-----	-----	-----	-----	-----
198.168.1.12	10.5.12.254	192.168.1.1	5009	5344385
192.168.1.27	10.5.12.23	192.168.1.1	748	388934
192.168.1.56	10.5.12.65	192.168.1.1	5	6534

0	1	2	3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1			
FlowSet ID = 256		Length = 64	
198.168.1.12			
10.5.12.254			
192.168.1.1			
5009			
5344385			
192.168.1.27			
10.5.12.23			
192.168.1.1			
748			
388934			
192.168.1.56			
10.5.12.65			
192.168.1.1			
5			
6534			

Note that padding was not necessary in this example.

11.4. Options Template FlowSet Example

Per line card (the Exporter is composed of two line cards), we want to report the following Field Types:

- Total number of Export Packets
- Total number of exported Flows

The format of the Options Template FlowSet is as follows:

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|          FlowSet ID = 1          |          Length = 24          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          Template ID 257         |          Option Scope Length = 4 |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          Option Length = 8       |          Scope 1 Field Type = 3 |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          Scope 1 Field Length = 2 |          TOTAL_EXP_PKTS_SENT = 41 |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          Field Length = 2        |          TOTAL_FLOWS_EXP = 42   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          Field Length = 2        |          Padding              |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

11.5. Data FlowSet with Options Data Records Example

In this example, we report the following two records:

Line Card ID	Export Packet	Export Flow
Line Card 1	345	10201
Line Card 2	690	20402

```

      0               1               2               3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   FlowSet ID = 257                               Length = 16   |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|               1                               345               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           10201                               2                 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           690                               20402                |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

12.2. Informative References

- [RFC768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.
- [RFC793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [RFC2960] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.
- [RFC3917] Quittek, J., Zseby, T., Claise, B., and S. Zander, "Requirements for IP Flow Information Export (IPFIX)", RFC 3917, October 2004.

13. Authors

This document was jointly written by Vamsidhar Valluri, Martin Djernaes, Ganesh Sadasivan, and Benoit Claise.

14. Acknowledgments

I would like to thank Pritam Shah, Paul Kohler, Dmitri Bouianovski, and Stewart Bryant for their valuable technical feedback.

15. Authors' Addresses

Benoit Claise (Editor)
Cisco Systems
De Kleetlaan 6a b1
1831 Diegem
Belgium

Phone: +32 2 704 5622
EMail: bclaise@cisco.com

Ganesh Sadasivan
Cisco Systems, Inc.
3750 Cisco Way
San Jose, CA 95134
USA

Phone: +1 408 527-0251
EMail: gsadasiv@cisco.com

Vamsi Valluri
Cisco Systems, Inc.
510 McCarthy Blvd.
San Jose, CA 95035
USA

Phone: +1 408 525-1835
EMail: vvalluri@cisco.com

Martin Djernaes
Cisco Systems, Inc.
510 McCarthy Blvd.
San Jose, CA 95035
USA

Phone: +1 408 853-1676
EMail: djernaes@cisco.com

Full Copyright Statement

Copyright (C) The Internet Society (2004).

This document is subject to the rights, licenses and restrictions contained in BCP 78, and at www.rfc-editor.org, and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the ISOC's procedures with respect to rights in ISOC Documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

- Navigate:
- mindrot.org
- [Projects](#)
- [People](#)
- [Contact](#)
- [eMail Lists](#)
- [BugZilla](#)
- [CVSWeb](#)

≤ November 2006 ≥

Su	Mo	Tu	We	Th	Fr	Sa
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30		

Softflowd

Softflowd is flow-based network traffic analyser capable of Cisco [NetFlow™](#) data export. Softflowd semi-statefully tracks traffic flows recorded by listening on a network interface or by reading a packet capture file. These flows may be reported via NetFlow™ to a collecting host or summarised within softflowd itself.

NB. If you are using OpenBSD, you may be interested in my [pfflowd](#) software instead. pfflowd uses the PF packet filter's stateful connection tracking to monitor flows rather than implementing it in software.

Mailing list

The [netflow-tools](#) mailing list is available for softflowd discussion, support, development and release announcements.

News

Thu, 02 Nov 2006: softflowd-0.9.8 released

It has been over a year since the last release of softflowd, but I'm happy to announce that [softflowd-0.9.8](#) has just been released. This release collects a number of small (but important) bugfixes that have accrued over the last year along with a couple of new features. See the [release notes](#) for details.

[\[permanent link\]](#)

Fri, 14 Jan 2005: softflowd-0.9.7 released

[softflowd-0.9.7](#) released. This release fixes some bugs and adds some options to facilitate export of flow records to multicast groups.

[\[permanent link\]](#)

Mon, 10 Jan 2005: Mailing list created

I have just created a new [mailing list](#) for the discussion of softflowd and the other NetFlow tools developed here. Development and support of the tools are on-topic and I will send announcements of new releases there too.

[\[permanent link\]](#)

Thu, 30 Sep 2004: softflowd-0.9.6 released

[softflowd-0.9.6](#) has just been released. This version adds support for the NetFlow v.9 export format and tracking of IPv6 flows.

[\[permanent link\]](#)

Fri, 27 Aug 2004: softflowd-0.9.2 released

I just released a bugfix version of softflowd: [0.9.2](#). This version fixes a one-line bug that crept in before the last release and prevents flow export from working. If you already have softflowd-0.9.1, you can download the small [patch](#) instead of picking up the entire tarball.

[\[permanent link\]](#)

Details

Softflowd semi-statefully tracks traffic flows. Upon expiry of a flow, its statistics are accumulated and reports them to a designated collector host using the standard NetFlow protocol. Currently the statistics collected are summaries only: min/max/avg/total bytes, packets on a aggregate or per-protocol basis.

Softflowd can export using NetFlow version 1, 5 or 9 datagrams and it is fully IPv6 capable: it can track and report on IPv6 traffic and flow export datagrams can be sent to an IPv6 host. Any standard NetFlow collector should be able to process the reports from softflowd.

As softflowd watches traffic promiscuously, it is likely to place additional load on hosts or gateways on which it is installed. However, this implementation has been designed to minimise this load as much as possible. Alternately, softflowd can read pcap save files recorded from tcpdump and friends.

Unless reading from a traffic dump, softflowd run as a daemon. A "remote control" program (softflowctl) is included which allows runtime control and extraction of statistics from a daemonised softflowd.

Softflowd is developed on Linux and OpenBSD. It requires libpcap and its associated headers to build, these are available from tcpdump.org, or from your operating system vendor. As of version 0.9, there is some support for Solaris but this is still experimental.

Download

softflowd is available here:

- [softflowd-0.9.8.tar.gz \(pgp signature\) \(relnotes\)](#)
- [TODO](#)
- [ChangeLog](#)
- [LICENSE](#)
- [Older versions](#)

RFC: 793

TRANSMISSION CONTROL PROTOCOL

DARPA INTERNET PROGRAM

PROTOCOL SPECIFICATION

September 1981

prepared for

Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, Virginia 22209

by

Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, California 90291

TABLE OF CONTENTS

PREFACE	iii
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Scope	2
1.3 About This Document	2
1.4 Interfaces	3
1.5 Operation	3
2. PHILOSOPHY	7
2.1 Elements of the Internetwork System	7
2.2 Model of Operation	7
2.3 The Host Environment	8
2.4 Interfaces	9
2.5 Relation to Other Protocols	9
2.6 Reliable Communication	9
2.7 Connection Establishment and Clearing	10
2.8 Data Communication	12
2.9 Precedence and Security	13
2.10 Robustness Principle	13
3. FUNCTIONAL SPECIFICATION	15
3.1 Header Format	15
3.2 Terminology	19
3.3 Sequence Numbers	24
3.4 Establishing a connection	30
3.5 Closing a Connection	37
3.6 Precedence and Security	40
3.7 Data Communication	40
3.8 Interfaces	44
3.9 Event Processing	52
GLOSSARY	79
REFERENCES	85

September 1981

Transmission Control Protocol

PREFACE

This document describes the DoD Standard Transmission Control Protocol (TCP). There have been nine earlier editions of the ARPA TCP specification on which this standard is based, and the present text draws heavily from them. There have been many contributors to this work both in terms of concepts and in terms of text. This edition clarifies several details and removes the end-of-letter buffer-size adjustments, and redescibes the letter mechanism as a push function.

Jon Postel

Editor

RFC: 793
Replaces: RFC 761
IENs: 129, 124, 112, 81,
55, 44, 40, 27, 21, 5

TRANSMISSION CONTROL PROTOCOL

DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION

1. INTRODUCTION

The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks.

This document describes the functions to be performed by the Transmission Control Protocol, the program that implements it, and its interface to programs or users that require its services.

1.1. Motivation

Computer communication systems are playing an increasingly important role in military, government, and civilian environments. This document focuses its attention primarily on military computer communication requirements, especially robustness in the presence of communication unreliability and availability in the presence of congestion, but many of these problems are found in the civilian and government sector as well.

As strategic and tactical computer communication networks are developed and deployed, it is essential to provide means of interconnecting them and to provide standard interprocess communication protocols which can support a broad range of applications. In anticipation of the need for such standards, the Deputy Undersecretary of Defense for Research and Engineering has declared the Transmission Control Protocol (TCP) described herein to be a basis for DoD-wide inter-process communication protocol standardization.

TCP is a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below the TCP layer. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, the TCP should be able to operate above a wide spectrum of communication systems ranging from hard-wired connections to packet-switched or circuit-switched networks.

Transmission Control Protocol

Introduction

TCP is based on concepts first described by Cerf and Kahn in [1]. The TCP fits into a layered protocol architecture just above a basic Internet Protocol [2] which provides a way for the TCP to send and receive variable-length segments of information enclosed in internet datagram "envelopes". The internet datagram provides a means for addressing source and destination TCPs in different networks. The internet protocol also deals with any fragmentation or reassembly of the TCP segments required to achieve transport and delivery through multiple networks and interconnecting gateways. The internet protocol also carries information on the precedence, security classification and compartmentation of the TCP segments, so this information can be communicated end-to-end across multiple networks.

Protocol Layering

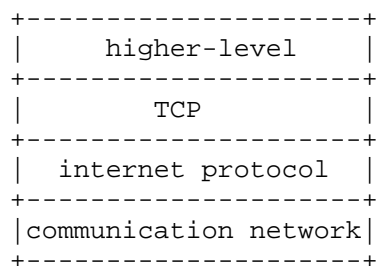


Figure 1

Much of this document is written in the context of TCP implementations which are co-resident with higher level protocols in the host computer. Some computer systems will be connected to networks via front-end computers which house the TCP and internet protocol layers, as well as network specific software. The TCP specification describes an interface to the higher level protocols which appears to be implementable even for the front-end case, as long as a suitable host-to-front end protocol is implemented.

1.2. Scope

The TCP is intended to provide a reliable process-to-process communication service in a multinet environment. The TCP is intended to be a host-to-host protocol in common use in multiple networks.

1.3. About this Document

This document represents a specification of the behavior required of any TCP implementation, both in its interactions with higher level protocols and in its interactions with other TCPs. The rest of this

section offers a very brief view of the protocol interfaces and operation. Section 2 summarizes the philosophical basis for the TCP design. Section 3 offers both a detailed description of the actions required of TCP when various events occur (arrival of new segments, user calls, errors, etc.) and the details of the formats of TCP segments.

1.4. Interfaces

The TCP interfaces on one side to user or application processes and on the other side to a lower level protocol such as Internet Protocol.

The interface between an application process and the TCP is illustrated in reasonable detail. This interface consists of a set of calls much like the calls an operating system provides to an application process for manipulating files. For example, there are calls to open and close connections and to send and receive data on established connections. It is also expected that the TCP can asynchronously communicate with application programs. Although considerable freedom is permitted to TCP implementors to design interfaces which are appropriate to a particular operating system environment, a minimum functionality is required at the TCP/user interface for any valid implementation.

The interface between TCP and lower level protocol is essentially unspecified except that it is assumed there is a mechanism whereby the two levels can asynchronously pass information to each other. Typically, one expects the lower level protocol to specify this interface. TCP is designed to work in a very general environment of interconnected networks. The lower level protocol which is assumed throughout this document is the Internet Protocol [2].

1.5. Operation

As noted above, the primary purpose of the TCP is to provide reliable, securable logical circuit or connection service between pairs of processes. To provide this service on top of a less reliable internet communication system requires facilities in the following areas:

- Basic Data Transfer
- Reliability
- Flow Control
- Multiplexing
- Connections
- Precedence and Security

The basic operation of the TCP in each of these areas is described in the following paragraphs.

Transmission Control Protocol Introduction

Basic Data Transfer:

The TCP is able to transfer a continuous stream of octets in each direction between its users by packaging some number of octets into segments for transmission through the internet system. In general, the TCPs decide when to block and forward data at their own convenience.

Sometimes users need to be sure that all the data they have submitted to the TCP has been transmitted. For this purpose a push function is defined. To assure that data submitted to a TCP is actually transmitted the sending user indicates that it should be pushed through to the receiving user. A push causes the TCPs to promptly forward and deliver data up to that point to the receiver. The exact push point might not be visible to the receiving user and the push function does not supply a record boundary marker.

Reliability:

The TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. At the receiver, the sequence numbers are used to correctly order segments that may be received out of order and to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments.

As long as the TCPs continue to function properly and the internet system does not become completely partitioned, no transmission errors will affect the correct delivery of data. TCP recovers from internet communication system errors.

Flow Control:

TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning a "window" with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission.

Multiplexing:

To allow for many processes within a single Host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets uniquely identifies each connection. That is, a socket may be simultaneously used in multiple connections.

The binding of ports to processes is handled independently by each Host. However, it proves useful to attach frequently used processes (e.g., a "logger" or timesharing service) to fixed sockets which are made known to the public. These services can then be accessed through the known addresses. Establishing and learning the port addresses of other processes may involve more dynamic mechanisms.

Connections:

The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream. The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection. Each connection is uniquely specified by a pair of sockets identifying its two sides.

When two processes wish to communicate, their TCP's must first establish a connection (initialize the status information on each side). When their communication is complete, the connection is terminated or closed to free the resources for other uses.

Since connections must be established between unreliable hosts and over the unreliable internet communication system, a handshake mechanism with clock-based sequence numbers is used to avoid erroneous initialization of connections.

Precedence and Security:

The users of TCP may indicate the security and precedence of their communication. Provision is made for default values to be used when these features are not needed.

September 1981

Transmission Control Protocol

2. PHILOSOPHY

2.1. Elements of the Internetwork System

The internetwork environment consists of hosts connected to networks which are in turn interconnected via gateways. It is assumed here that the networks may be either local networks (e.g., the ETHERNET) or large networks (e.g., the ARPANET), but in any case are based on packet switching technology. The active agents that produce and consume messages are processes. Various levels of protocols in the networks, the gateways, and the hosts support an interprocess communication system that provides two-way data flow on logical connections between process ports.

The term packet is used generically here to mean the data of one transaction between a host and its network. The format of data blocks exchanged within the a network will generally not be of concern to us.

Hosts are computers attached to a network, and from the communication network's point of view, are the sources and destinations of packets. Processes are viewed as the active elements in host computers (in accordance with the fairly common definition of a process as a program in execution). Even terminals and files or other I/O devices are viewed as communicating with each other through the use of processes. Thus, all communication is viewed as inter-process communication.

Since a process may need to distinguish among several communication streams between itself and another process (or processes), we imagine that each process may have a number of ports through which it communicates with the ports of other processes.

2.2. Model of Operation

Processes transmit data by calling on the TCP and passing buffers of data as arguments. The TCP packages the data from these buffers into segments and calls on the internet module to transmit each segment to the destination TCP. The receiving TCP places the data from a segment into the receiving user's buffer and notifies the receiving user. The TCPs include control information in the segments which they use to ensure reliable ordered data transmission.

The model of internet communication is that there is an internet protocol module associated with each TCP which provides an interface to the local network. This internet module packages TCP segments inside internet datagrams and routes these datagrams to a destination internet module or intermediate gateway. To transmit the datagram through the local network, it is embedded in a local network packet.

The packet switches may perform further packaging, fragmentation, or

Transmission Control Protocol
Philosophy

other operations to achieve the delivery of the local packet to the destination internet module.

At a gateway between networks, the internet datagram is "unwrapped" from its local packet and examined to determine through which network the internet datagram should travel next. The internet datagram is then "wrapped" in a local packet suitable to the next network and routed to the next gateway, or to the final destination.

A gateway is permitted to break up an internet datagram into smaller internet datagram fragments if this is necessary for transmission through the next network. To do this, the gateway produces a set of internet datagrams; each carrying a fragment. Fragments may be further broken into smaller fragments at subsequent gateways. The internet datagram fragment format is designed so that the destination internet module can reassemble fragments into internet datagrams.

A destination internet module unwraps the segment from the datagram (after reassembling the datagram, if necessary) and passes it to the destination TCP.

This simple model of the operation glosses over many details. One important feature is the type of service. This provides information to the gateway (or internet module) to guide it in selecting the service parameters to be used in traversing the next network. Included in the type of service information is the precedence of the datagram. Datagrams may also carry security information to permit host and gateways that operate in multilevel secure environments to properly segregate datagrams for security considerations.

2.3. The Host Environment

The TCP is assumed to be a module in an operating system. The users access the TCP much like they would access the file system. The TCP may call on other operating system functions, for example, to manage data structures. The actual interface to the network is assumed to be controlled by a device driver module. The TCP does not call on the network device driver directly, but rather calls on the internet datagram protocol module which may in turn call on the device driver.

The mechanisms of TCP do not preclude implementation of the TCP in a front-end processor. However, in such an implementation, a host-to-front-end protocol must provide the functionality to support the type of TCP-user interface described in this document.

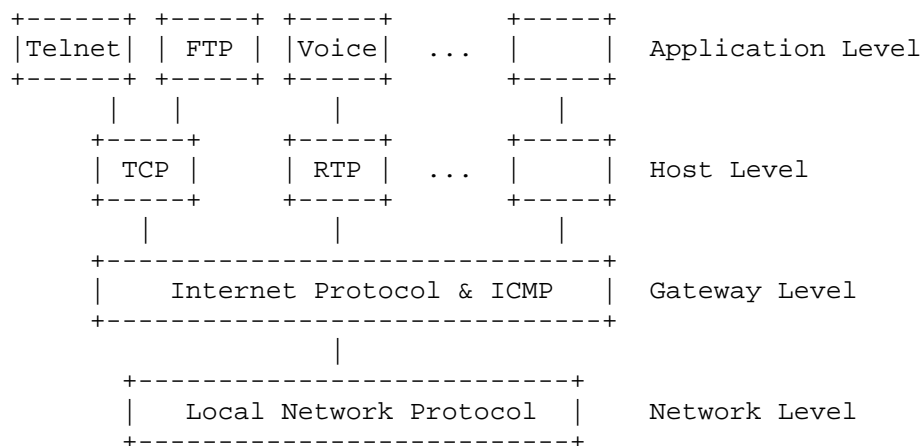
2.4. Interfaces

The TCP/user interface provides for calls made by the user on the TCP to OPEN or CLOSE a connection, to SEND or RECEIVE data, or to obtain STATUS about a connection. These calls are like other calls from user programs on the operating system, for example, the calls to open, read from, and close a file.

The TCP/internet interface provides calls to send and receive datagrams addressed to TCP modules in hosts anywhere in the internet system. These calls have parameters for passing the address, type of service, precedence, security, and other control information.

2.5. Relation to Other Protocols

The following diagram illustrates the place of the TCP in the protocol hierarchy:



Protocol Relationships

Figure 2.

It is expected that the TCP will be able to support higher level protocols efficiently. It should be easy to interface higher level protocols like the ARPANET Telnet or AUTODIN II THP to the TCP.

2.6. Reliable Communication

A stream of data sent on a TCP connection is delivered reliably and in order at the destination.

Transmission Control Protocol Philosophy

Transmission is made reliable via the use of sequence numbers and acknowledgments. Conceptually, each octet of data is assigned a sequence number. The sequence number of the first octet of data in a segment is transmitted with that segment and is called the segment sequence number. Segments also carry an acknowledgment number which is the sequence number of the next expected data octet of transmissions in the reverse direction. When the TCP transmits a segment containing data, it puts a copy on a retransmission queue and starts a timer; when the acknowledgment for that data is received, the segment is deleted from the queue. If the acknowledgment is not received before the timer runs out, the segment is retransmitted.

An acknowledgment by TCP does not guarantee that the data has been delivered to the end user, but only that the receiving TCP has taken the responsibility to do so.

To govern the flow of data between TCPs, a flow control mechanism is employed. The receiving TCP reports a "window" to the sending TCP. This window specifies the number of octets, starting with the acknowledgment number, that the receiving TCP is currently prepared to receive.

2.7. Connection Establishment and Clearing

To identify the separate data streams that a TCP may handle, the TCP provides a port identifier. Since port identifiers are selected independently by each TCP they might not be unique. To provide for unique addresses within each TCP, we concatenate an internet address identifying the TCP with a port identifier to create a socket which will be unique throughout all networks connected together.

A connection is fully specified by the pair of sockets at the ends. A local socket may participate in many connections to different foreign sockets. A connection can be used to carry data in both directions, that is, it is "full duplex".

TCPs are free to associate ports with processes however they choose. However, several basic concepts are necessary in any implementation. There must be well-known sockets which the TCP associates only with the "appropriate" processes by some means. We envision that processes may "own" ports, and that processes can initiate connections only on the ports they own. (Means for implementing ownership is a local issue, but we envision a Request Port user command, or a method of uniquely allocating a group of ports to a given process, e.g., by associating the high order bits of a port name with a given process.)

A connection is specified in the OPEN call by the local port and foreign socket arguments. In return, the TCP supplies a (short) local

connection name by which the user refers to the connection in subsequent calls. There are several things that must be remembered about a connection. To store this information we imagine that there is a data structure called a Transmission Control Block (TCB). One implementation strategy would have the local connection name be a pointer to the TCB for this connection. The OPEN call also specifies whether the connection establishment is to be actively pursued, or to be passively waited for.

A passive OPEN request means that the process wants to accept incoming connection requests rather than attempting to initiate a connection. Often the process requesting a passive OPEN will accept a connection request from any caller. In this case a foreign socket of all zeros is used to denote an unspecified socket. Unspecified foreign sockets are allowed only on passive OPENs.

A service process that wished to provide services for unknown other processes would issue a passive OPEN request with an unspecified foreign socket. Then a connection could be made with any process that requested a connection to this local socket. It would help if this local socket were known to be associated with this service.

Well-known sockets are a convenient mechanism for a priori associating a socket address with a standard service. For instance, the "Telnet-Server" process is permanently assigned to a particular socket, and other sockets are reserved for File Transfer, Remote Job Entry, Text Generator, Echoer, and Sink processes (the last three being for test purposes). A socket address might be reserved for access to a "Look-Up" service which would return the specific socket at which a newly created service would be provided. The concept of a well-known socket is part of the TCP specification, but the assignment of sockets to services is outside this specification. (See [4].)

Processes can issue passive OPENs and wait for matching active OPENs from other processes and be informed by the TCP when connections have been established. Two processes which issue active OPENs to each other at the same time will be correctly connected. This flexibility is critical for the support of distributed computing in which components act asynchronously with respect to each other.

There are two principal cases for matching the sockets in the local passive OPENs and an foreign active OPENs. In the first case, the local passive OPENs has fully specified the foreign socket. In this case, the match must be exact. In the second case, the local passive OPENs has left the foreign socket unspecified. In this case, any foreign socket is acceptable as long as the local sockets match. Other possibilities include partially restricted matches.

Transmission Control Protocol Philosophy

If there are several pending passive OPENs (recorded in TCBs) with the same local socket, an foreign active OPEN will be matched to a TCB with the specific foreign socket in the foreign active OPEN, if such a TCB exists, before selecting a TCB with an unspecified foreign socket.

The procedures to establish connections utilize the synchronize (SYN) control flag and involves an exchange of three messages. This exchange has been termed a three-way hand shake [3].

A connection is initiated by the rendezvous of an arriving segment containing a SYN and a waiting TCB entry each created by a user OPEN command. The matching of local and foreign sockets determines when a connection has been initiated. The connection becomes "established" when sequence numbers have been synchronized in both directions.

The clearing of a connection also involves the exchange of segments, in this case carrying the FIN control flag.

2.8. Data Communication

The data that flows on a connection may be thought of as a stream of octets. The sending user indicates in each SEND call whether the data in that call (and any preceeding calls) should be immediately pushed through to the receiving user by the setting of the PUSH flag.

A sending TCP is allowed to collect data from the sending user and to send that data in segments at its own convenience, until the push function is signaled, then it must send all unsent data. When a receiving TCP sees the PUSH flag, it must not wait for more data from the sending TCP before passing the data to the receiving process.

There is no necessary relationship between push functions and segment boundaries. The data in any particular segment may be the result of a single SEND call, in whole or part, or of multiple SEND calls.

The purpose of push function and the PUSH flag is to push data through from the sending user to the receiving user. It does not provide a record service.

There is a coupling between the push function and the use of buffers of data that cross the TCP/user interface. Each time a PUSH flag is associated with data placed into the receiving user's buffer, the buffer is returned to the user for processing even if the buffer is not filled. If data arrives that fills the user's buffer before a PUSH is seen, the data is passed to the user in buffer size units.

TCP also provides a means to communicate to the receiver of data that at some point further along in the data stream than the receiver is

currently reading there is urgent data. TCP does not attempt to define what the user specifically does upon being notified of pending urgent data, but the general notion is that the receiving process will take action to process the urgent data quickly.

2.9. Precedence and Security

The TCP makes use of the internet protocol type of service field and security option to provide precedence and security on a per connection basis to TCP users. Not all TCP modules will necessarily function in a multilevel secure environment; some may be limited to unclassified use only, and others may operate at only one security level and compartment. Consequently, some TCP implementations and services to users may be limited to a subset of the multilevel secure case.

TCP modules which operate in a multilevel secure environment must properly mark outgoing segments with the security, compartment, and precedence. Such TCP modules must also provide to their users or higher level protocols such as Telnet or THP an interface to allow them to specify the desired security level, compartment, and precedence of connections.

2.10. Robustness Principle

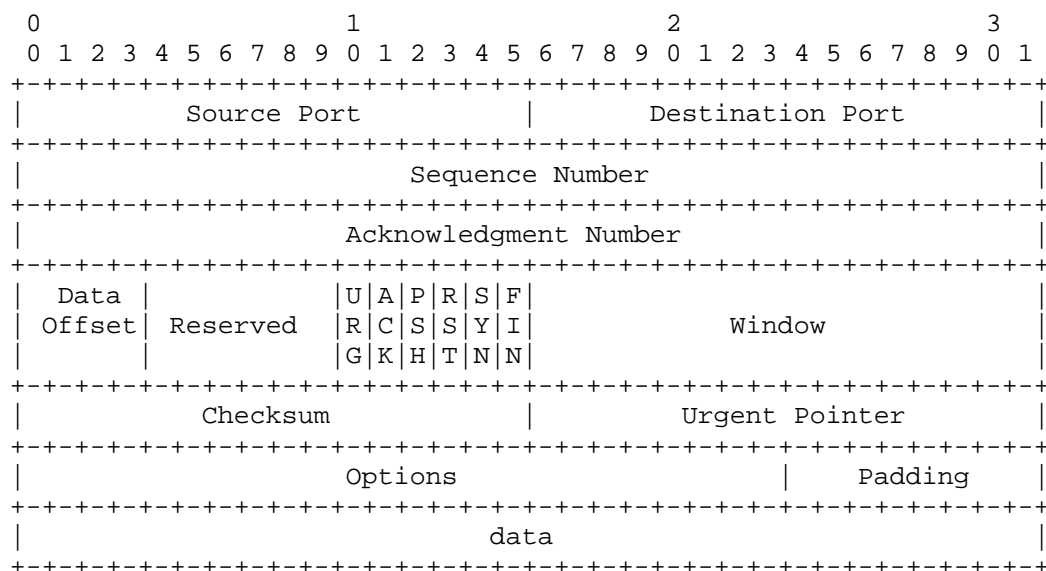
TCP implementations will follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others.

3. FUNCTIONAL SPECIFICATION

3.1. Header Format

TCP segments are sent as internet datagrams. The Internet Protocol header carries several information fields, including the source and destination host addresses [2]. A TCP header follows the internet header, supplying information specific to the TCP protocol. This division allows for the existence of host level protocols other than TCP.

TCP Header Format



TCP Header Format

Note that one tick mark represents one bit position.

Figure 3.

Source Port: 16 bits

The source port number.

Destination Port: 16 bits

The destination port number.

September 1981

Transmission Control Protocol
Functional Specification

Sequence Number: 32 bits

The sequence number of the first data octet in this segment (except when SYN is present). If SYN is present the sequence number is the initial sequence number (ISN) and the first data octet is ISN+1.

Acknowledgment Number: 32 bits

If the ACK control bit is set this field contains the value of the next sequence number the sender of the segment is expecting to receive. Once a connection is established this is always sent.

Data Offset: 4 bits

The number of 32 bit words in the TCP Header. This indicates where the data begins. The TCP header (even one including options) is an integral number of 32 bits long.

Reserved: 6 bits

Reserved for future use. Must be zero.

Control Bits: 6 bits (from left to right):

URG: Urgent Pointer field significant
ACK: Acknowledgment field significant
PSH: Push Function
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: No more data from sender

Window: 16 bits

The number of data octets beginning with the one indicated in the acknowledgment field which the sender of this segment is willing to accept.

Checksum: 16 bits

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header and text. If a segment contains an odd number of header and text octets to be checksummed, the last octet is padded on the right with zeros to form a 16 bit word for checksum purposes. The pad is not transmitted as part of the segment. While computing the checksum, the checksum field itself is replaced with zeros.

The checksum also covers a 96 bit pseudo header conceptually

prefixed to the TCP header. This pseudo header contains the Source Address, the Destination Address, the Protocol, and TCP length. This gives the TCP protection against misrouted segments. This information is carried in the Internet Protocol and is transferred across the TCP/Network interface in the arguments or results of calls by the TCP on the IP.

```

+-----+-----+-----+-----+
|           Source Address           |
+-----+-----+-----+-----+
|           Destination Address      |
+-----+-----+-----+-----+
| zero | PTCL |   TCP Length   |
+-----+-----+-----+-----+

```

The TCP Length is the TCP header length plus the data length in octets (this is not an explicitly transmitted quantity, but is computed), and it does not count the 12 octets of the pseudo header.

Urgent Pointer: 16 bits

This field communicates the current value of the urgent pointer as a positive offset from the sequence number in this segment. The urgent pointer points to the sequence number of the octet following the urgent data. This field is only be interpreted in segments with the URG control bit set.

Options: variable

Options may occupy space at the end of the TCP header and are a multiple of 8 bits in length. All options are included in the checksum. An option may begin on any octet boundary. There are two cases for the format of an option:

Case 1: A single octet of option-kind.

Case 2: An octet of option-kind, an octet of option-length, and the actual option-data octets.

The option-length counts the two octets of option-kind and option-length as well as the option-data octets.

Note that the list of options may be shorter than the data offset field might imply. The content of the header beyond the End-of-Option option must be header padding (i.e., zero).

A TCP must implement all options.

Transmission Control Protocol Functional Specification

Currently defined options include (kind indicated in octal):

Kind	Length	Meaning
----	-----	-----
0	-	End of option list.
1	-	No-Operation.
2	4	Maximum Segment Size.

Specific Option Definitions

End of Option List

```
+-----+
|00000000|
+-----+
Kind=0
```

This option code indicates the end of the option list. This might not coincide with the end of the TCP header according to the Data Offset field. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the TCP header.

No-Operation

```
+-----+
|00000001|
+-----+
Kind=1
```

This option code may be used between options, for example, to align the beginning of a subsequent option on a word boundary. There is no guarantee that senders will use this option, so receivers must be prepared to process options even if they do not begin on a word boundary.

Maximum Segment Size

```
+-----+-----+-----+-----+
|00000010|00000100|   max seg size   |
+-----+-----+-----+-----+
Kind=2   Length=4
```

Maximum Segment Size Option Data: 16 bits

If this option is present, then it communicates the maximum receive segment size at the TCP which sends this segment. This field must only be sent in the initial connection request (i.e., in segments with the SYN control bit set). If this option is not used, any segment size is allowed.

Padding: variable

The TCP header padding is used to ensure that the TCP header ends and data begins on a 32 bit boundary. The padding is composed of zeros.

3.2. Terminology

Before we can discuss very much about the operation of the TCP we need to introduce some detailed terminology. The maintenance of a TCP connection requires the remembering of several variables. We conceive of these variables being stored in a connection record called a Transmission Control Block or TCB. Among the variables stored in the TCB are the local and remote socket numbers, the security and precedence of the connection, pointers to the user's send and receive buffers, pointers to the retransmit queue and to the current segment. In addition several variables relating to the send and receive sequence numbers are stored in the TCB.

Send Sequence Variables

SND.UNA - send unacknowledged
SND.NXT - send next
SND.WND - send window
SND.UP - send urgent pointer
SND.WL1 - segment sequence number used for last window update
SND.WL2 - segment acknowledgment number used for last window update
ISS - initial send sequence number

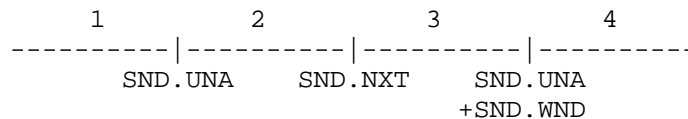
Receive Sequence Variables

RCV.NXT - receive next
RCV.WND - receive window
RCV.UP - receive urgent pointer
IRS - initial receive sequence number

Transmission Control Protocol
Functional Specification

The following diagrams may help to relate some of these variables to the sequence space.

Send Sequence Space



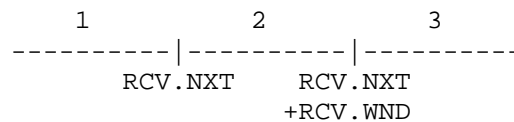
- 1 - old sequence numbers which have been acknowledged
- 2 - sequence numbers of unacknowledged data
- 3 - sequence numbers allowed for new data transmission
- 4 - future sequence numbers which are not yet allowed

Send Sequence Space

Figure 4.

The send window is the portion of the sequence space labeled 3 in figure 4.

Receive Sequence Space



- 1 - old sequence numbers which have been acknowledged
- 2 - sequence numbers allowed for new reception
- 3 - future sequence numbers which are not yet allowed

Receive Sequence Space

Figure 5.

The receive window is the portion of the sequence space labeled 2 in figure 5.

There are also some variables used frequently in the discussion that take their values from the fields of the current segment.

Current Segment Variables

SEG.SEQ - segment sequence number
SEG.ACK - segment acknowledgment number
SEG.LEN - segment length
SEG.WND - segment window
SEG.UP - segment urgent pointer
SEG.PRC - segment precedence value

A connection progresses through a series of states during its lifetime. The states are: LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, and the fictional state CLOSED. CLOSED is fictional because it represents the state when there is no TCB, and therefore, no connection. Briefly the meanings of the states are:

LISTEN - represents waiting for a connection request from any remote TCP and port.

SYN-SENT - represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED - represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED - represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 - represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2 - represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT - represents waiting for a connection termination request from the local user.

CLOSING - represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK - represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

September 1981

Transmission Control Protocol
Functional Specification

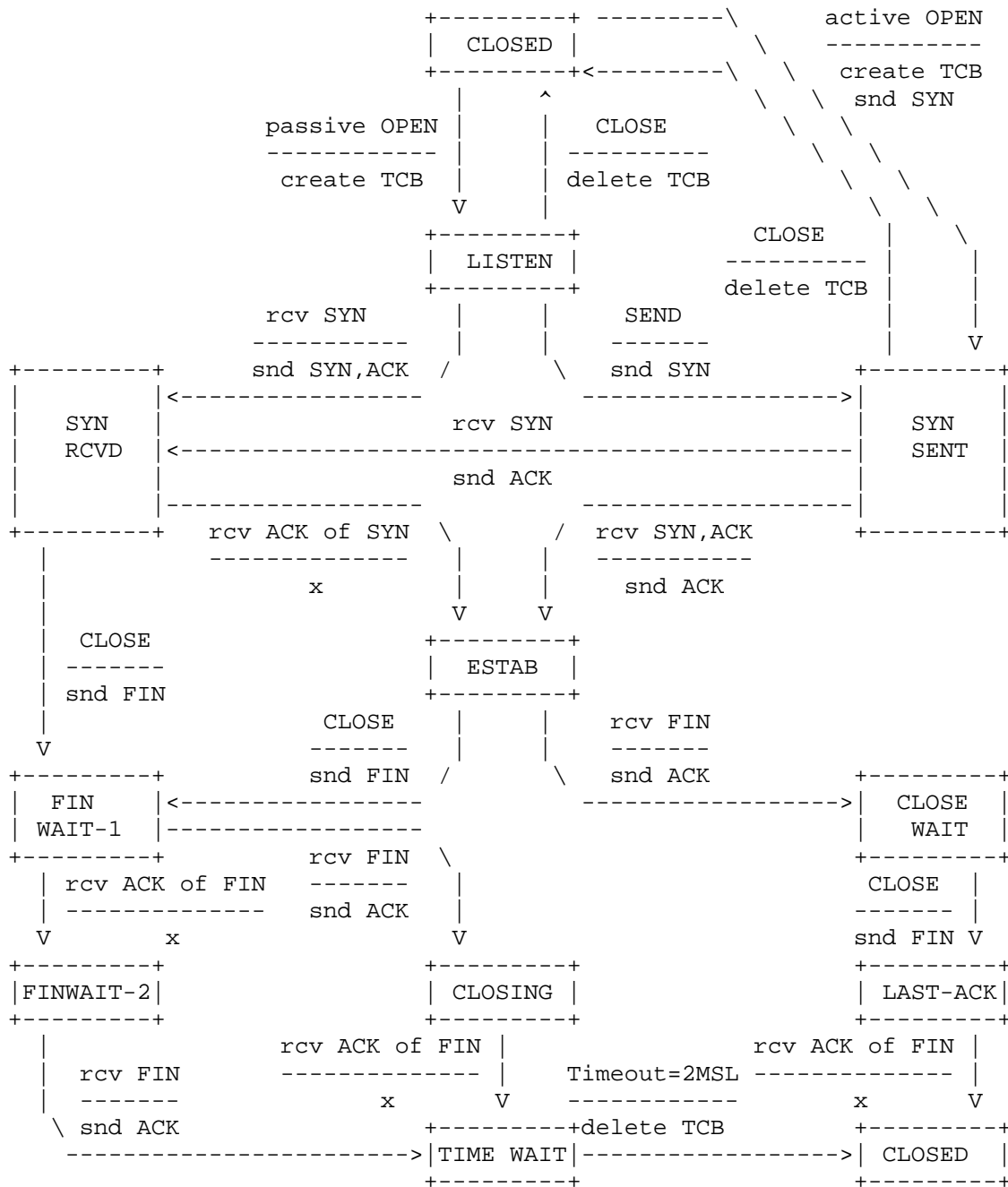
TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

CLOSED - represents no connection state at all.

A TCP connection progresses from one state to another in response to events. The events are the user calls, OPEN, SEND, RECEIVE, CLOSE, ABORT, and STATUS; the incoming segments, particularly those containing the SYN, ACK, RST and FIN flags; and timeouts.

The state diagram in figure 6 illustrates only state changes, together with the causing events and resulting actions, but addresses neither error conditions nor actions which are not connected with state changes. In a later section, more detail is offered with respect to the reaction of the TCP to events.

NOTE BENE: this diagram is only a summary and must not be taken as the total specification.

Transmission Control Protocol
Functional SpecificationTCP Connection State Diagram
Figure 6.

Transmission Control Protocol
Functional Specification

3.3. Sequence Numbers

A fundamental notion in the design is that every octet of data sent over a TCP connection has a sequence number. Since every octet is sequenced, each of them can be acknowledged. The acknowledgment mechanism employed is cumulative so that an acknowledgment of sequence number X indicates that all octets up to but not including X have been received. This mechanism allows for straight-forward duplicate detection in the presence of retransmission. Numbering of octets within a segment is that the first data octet immediately following the header is the lowest numbered, and the following octets are numbered consecutively.

It is essential to remember that the actual sequence number space is finite, though very large. This space ranges from 0 to $2^{32} - 1$. Since the space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2^{32} . This unsigned arithmetic preserves the relationship of sequence numbers as they cycle from $2^{32} - 1$ to 0 again. There are some subtleties to computer modulo arithmetic, so great care should be taken in programming the comparison of such values. The symbol " $=$ " means "less than or equal" (modulo 2^{32}).

The typical kinds of sequence number comparisons which the TCP must perform include:

- (a) Determining that an acknowledgment refers to some sequence number sent but not yet acknowledged.
- (b) Determining that all sequence numbers occupied by a segment have been acknowledged (e.g., to remove the segment from a retransmission queue).
- (c) Determining that an incoming segment contains sequence numbers which are expected (i.e., that the segment "overlaps" the receive window).

In response to sending data the TCP will receive acknowledgments. The following comparisons are needed to process the acknowledgments.

SND.UNA = oldest unacknowledged sequence number

SND.NXT = next sequence number to be sent

SEG.ACK = acknowledgment from the receiving TCP (next sequence number expected by the receiving TCP)

SEG.SEQ = first sequence number of a segment

SEG.LEN = the number of octets occupied by the data in the segment (counting SYN and FIN)

SEG.SEQ+SEG.LEN-1 = last sequence number of a segment

A new acknowledgment (called an "acceptable ack"), is one for which the inequality below holds:

$\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$

A segment on the retransmission queue is fully acknowledged if the sum of its sequence number and length is less or equal than the acknowledgment value in the incoming segment.

When data is received the following comparisons are needed:

RCV.NXT = next sequence number expected on an incoming segments, and is the left or lower edge of the receive window

RCV.NXT+RCV.WND-1 = last sequence number expected on an incoming segment, and is the right or upper edge of the receive window

SEG.SEQ = first sequence number occupied by the incoming segment

SEG.SEQ+SEG.LEN-1 = last sequence number occupied by the incoming segment

A segment is judged to occupy a portion of valid receive sequence space if

$\text{RCV.NXT} \leq \text{SEG.SEQ} < \text{RCV.NXT} + \text{RCV.WND}$

or

$\text{RCV.NXT} \leq \text{SEG.SEQ} + \text{SEG.LEN} - 1 < \text{RCV.NXT} + \text{RCV.WND}$

Transmission Control Protocol
Functional Specification

The first part of this test checks to see if the beginning of the segment falls in the window, the second part of the test checks to see if the end of the segment falls in the window; if the segment passes either part of the test it contains data in the window.

Actually, it is a little more complicated than this. Due to zero windows and zero length segments, we have four cases for the acceptability of an incoming segment:

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT
0	>0	RCV.NXT ≤ SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT ≤ SEG.SEQ < RCV.NXT+RCV.WND or RCV.NXT ≤ SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND

Note that when the receive window is zero no segments should be acceptable except ACK segments. Thus, it is possible for a TCP to maintain a zero receive window while transmitting data and receiving ACKs. However, even when the receive window is zero, a TCP must process the RST and URG fields of all incoming segments.

We have taken advantage of the numbering scheme to protect certain control information as well. This is achieved by implicitly including some control flags in the sequence space so they can be retransmitted and acknowledged without confusion (i.e., one and only one copy of the control will be acted upon). Control information is not physically carried in the segment data space. Consequently, we must adopt rules for implicitly assigning sequence numbers to control. The SYN and FIN are the only controls requiring this protection, and these controls are used only at connection opening and closing. For sequence number purposes, the SYN is considered to occur before the first actual data octet of the segment in which it occurs, while the FIN is considered to occur after the last actual data octet in a segment in which it occurs. The segment length (SEG.LEN) includes both data and sequence space occupying controls. When a SYN is present then SEG.SEQ is the sequence number of the SYN.

Initial Sequence Number Selection

The protocol places no restriction on a particular connection being used over and over again. A connection is defined by a pair of sockets. New instances of a connection will be referred to as incarnations of the connection. The problem that arises from this is -- "how does the TCP identify duplicate segments from previous incarnations of the connection?" This problem becomes apparent if the connection is being opened and closed in quick succession, or if the connection breaks with loss of memory and is then reestablished.

To avoid confusion we must prevent segments from one incarnation of a connection from being used while the same sequence numbers may still be present in the network from an earlier incarnation. We want to assure this, even if a TCP crashes and loses all knowledge of the sequence numbers it has been using. When new connections are created, an initial sequence number (ISN) generator is employed which selects a new 32 bit ISN. The generator is bound to a (possibly fictitious) 32 bit clock whose low order bit is incremented roughly every 4 microseconds. Thus, the ISN cycles approximately every 4.55 hours. Since we assume that segments will stay in the network no more than the Maximum Segment Lifetime (MSL) and that the MSL is less than 4.55 hours we can reasonably assume that ISN's will be unique.

For each connection there is a send sequence number and a receive sequence number. The initial send sequence number (ISS) is chosen by the data sending TCP, and the initial receive sequence number (IRS) is learned during the connection establishing procedure.

For a connection to be established or initialized, the two TCPs must synchronize on each other's initial sequence numbers. This is done in an exchange of connection establishing segments carrying a control bit called "SYN" (for synchronize) and the initial sequence numbers. As a shorthand, segments carrying the SYN bit are also called "SYNs". Hence, the solution requires a suitable mechanism for picking an initial sequence number and a slightly involved handshake to exchange the ISN's.

The synchronization requires each side to send it's own initial sequence number and to receive a confirmation of it in acknowledgment from the other side. Each side must also receive the other side's initial sequence number and send a confirming acknowledgment.

- 1) A --> B SYN my sequence number is X
- 2) A <-- B ACK your sequence number is X
- 3) A <-- B SYN my sequence number is Y
- 4) A --> B ACK your sequence number is Y

Transmission Control Protocol Functional Specification

Because steps 2 and 3 can be combined in a single message this is called the three way (or three message) handshake.

A three way handshake is necessary because sequence numbers are not tied to a global clock in the network, and TCPs may have different mechanisms for picking the ISN's. The receiver of the first SYN has no way of knowing whether the segment was an old delayed one or not, unless it remembers the last sequence number used on the connection (which is not always possible), and so it must ask the sender to verify this SYN. The three way handshake and the advantages of a clock-driven scheme are discussed in [3].

Knowing When to Keep Quiet

To be sure that a TCP does not create a segment that carries a sequence number which may be duplicated by an old segment remaining in the network, the TCP must keep quiet for a maximum segment lifetime (MSL) before assigning any sequence numbers upon starting up or recovering from a crash in which memory of sequence numbers in use was lost. For this specification the MSL is taken to be 2 minutes. This is an engineering choice, and may be changed if experience indicates it is desirable to do so. Note that if a TCP is reinitialized in some sense, yet retains its memory of sequence numbers in use, then it need not wait at all; it must only be sure to use sequence numbers larger than those recently used.

The TCP Quiet Time Concept

This specification provides that hosts which "crash" without retaining any knowledge of the last sequence numbers transmitted on each active (i.e., not closed) connection shall delay emitting any TCP segments for at least the agreed Maximum Segment Lifetime (MSL) in the internet system of which the host is a part. In the paragraphs below, an explanation for this specification is given. TCP implementors may violate the "quiet time" restriction, but only at the risk of causing some old data to be accepted as new or new data rejected as old duplicated by some receivers in the internet system.

TCPs consume sequence number space each time a segment is formed and entered into the network output queue at a source host. The duplicate detection and sequencing algorithm in the TCP protocol relies on the unique binding of segment data to sequence space to the extent that sequence numbers will not cycle through all 2^{32} values before the segment data bound to those sequence numbers has been delivered and acknowledged by the receiver and all duplicate copies of the segments have "drained" from the internet. Without such an assumption, two distinct TCP segments could conceivably be

assigned the same or overlapping sequence numbers, causing confusion at the receiver as to which data is new and which is old. Remember that each segment is bound to as many consecutive sequence numbers as there are octets of data in the segment.

Under normal conditions, TCPs keep track of the next sequence number to emit and the oldest awaiting acknowledgment so as to avoid mistakenly using a sequence number over before its first use has been acknowledged. This alone does not guarantee that old duplicate data is drained from the net, so the sequence space has been made very large to reduce the probability that a wandering duplicate will cause trouble upon arrival. At 2 megabits/sec. it takes 4.5 hours to use up 2^{32} octets of sequence space. Since the maximum segment lifetime in the net is not likely to exceed a few tens of seconds, this is deemed ample protection for foreseeable nets, even if data rates escalate to 10's of megabits/sec. At 100 megabits/sec, the cycle time is 5.4 minutes which may be a little short, but still within reason.

The basic duplicate detection and sequencing algorithm in TCP can be defeated, however, if a source TCP does not have any memory of the sequence numbers it last used on a given connection. For example, if the TCP were to start all connections with sequence number 0, then upon crashing and restarting, a TCP might re-form an earlier connection (possibly after half-open connection resolution) and emit packets with sequence numbers identical to or overlapping with packets still in the network which were emitted on an earlier incarnation of the same connection. In the absence of knowledge about the sequence numbers used on a particular connection, the TCP specification recommends that the source delay for MSL seconds before emitting segments on the connection, to allow time for segments from the earlier connection incarnation to drain from the system.

Even hosts which can remember the time of day and used it to select initial sequence number values are not immune from this problem (i.e., even if time of day is used to select an initial sequence number for each new connection incarnation).

Suppose, for example, that a connection is opened starting with sequence number S. Suppose that this connection is not used much and that eventually the initial sequence number function (ISN(t)) takes on a value equal to the sequence number, say S1, of the last segment sent by this TCP on a particular connection. Now suppose, at this instant, the host crashes, recovers, and establishes a new incarnation of the connection. The initial sequence number chosen is $S1 = \text{ISN}(t)$ -- last used sequence number on old incarnation of connection! If the recovery occurs quickly enough, any old

Transmission Control Protocol
Functional Specification

duplicates in the net bearing sequence numbers in the neighborhood of S1 may arrive and be treated as new packets by the receiver of the new incarnation of the connection.

The problem is that the recovering host may not know for how long it crashed nor does it know whether there are still old duplicates in the system from earlier connection incarnations.

One way to deal with this problem is to deliberately delay emitting segments for one MSL after recovery from a crash- this is the "quite time" specification. Hosts which prefer to avoid waiting are willing to risk possible confusion of old and new packets at a given destination may choose not to wait for the "quite time".

Implementors may provide TCP users with the ability to select on a connection by connection basis whether to wait after a crash, or may informally implement the "quite time" for all connections. Obviously, even where a user selects to "wait," this is not necessary after the host has been "up" for at least MSL seconds.

To summarize: every segment emitted occupies one or more sequence numbers in the sequence space, the numbers occupied by a segment are "busy" or "in use" until MSL seconds have passed, upon crashing a block of space-time is occupied by the octets of the last emitted segment, if a new connection is started too soon and uses any of the sequence numbers in the space-time footprint of the last segment of the previous connection incarnation, there is a potential sequence number overlap area which could cause confusion at the receiver.

3.4. Establishing a connection

The "three-way handshake" is the procedure used to establish a connection. This procedure normally is initiated by one TCP and responded to by another TCP. The procedure also works if two TCP simultaneously initiate the procedure. When simultaneous attempt occurs, each TCP receives a "SYN" segment which carries no acknowledgment after it has sent a "SYN". Of course, the arrival of an old duplicate "SYN" segment can potentially make it appear, to the recipient, that a simultaneous connection initiation is in progress. Proper use of "reset" segments can disambiguate these cases.

Several examples of connection initiation follow. Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate, so long as the receiving TCP doesn't deliver the data to the user until it is clear the data is valid (i.e., the data must be buffered at the receiver until the connection reaches the ESTABLISHED state). The three-way handshake reduces the possibility of false connections. It is the

implementation of a trade-off between memory and messages to provide information for this checking.

The simplest three-way handshake is shown in figure 7 below. The figures should be interpreted in the following way. Each line is numbered for reference purposes. Right arrows (-->) indicate departure of a TCP segment from TCP A to TCP B, or arrival of a segment at B from A. Left arrows (<--), indicate the reverse. Ellipsis (...) indicates a segment which is still in the network (delayed). An "XXX" indicates a segment which is lost or rejected. Comments appear in parentheses. TCP states represent the state AFTER the departure or arrival of the segment (whose contents are shown in the center of each line). Segment contents are shown in abbreviated form, with sequence number, control flags, and ACK field. Other fields such as window, addresses, lengths, and text have been left out in the interest of clarity.

TCP A		TCP B
1. CLOSED		LISTEN
2. SYN-SENT	--> <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
3. ESTABLISHED	<-- <SEQ=300><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
4. ESTABLISHED	--> <SEQ=101><ACK=301><CTL=ACK>	--> ESTABLISHED
5. ESTABLISHED	--> <SEQ=101><ACK=301><CTL=ACK><DATA>	--> ESTABLISHED

Basic 3-Way Handshake for Connection Synchronization

Figure 7.

In line 2 of figure 7, TCP A begins by sending a SYN segment indicating that it will use sequence numbers starting with sequence number 100. In line 3, TCP B sends a SYN and acknowledges the SYN it received from TCP A. Note that the acknowledgment field indicates TCP B is now expecting to hear sequence 101, acknowledging the SYN which occupied sequence 100.

At line 4, TCP A responds with an empty segment containing an ACK for TCP B's SYN; and in line 5, TCP A sends some data. Note that the sequence number of the segment in line 5 is the same as in line 4 because the ACK does not occupy sequence number space (if it did, we would wind up ACKing ACK's!).

Transmission Control Protocol
Functional Specification

Simultaneous initiation is only slightly more complex, as is shown in figure 8. Each TCP cycles from CLOSED to SYN-SENT to SYN-RECEIVED to ESTABLISHED.

TCP A		TCP B
1. CLOSED		CLOSED
2. SYN-SENT	--> <SEQ=100><CTL=SYN>	...
3. SYN-RECEIVED	<-- <SEQ=300><CTL=SYN>	<-- SYN-SENT
4.	... <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
5. SYN-RECEIVED	--> <SEQ=100><ACK=301><CTL=SYN,ACK>	...
6. ESTABLISHED	<-- <SEQ=300><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
7.	... <SEQ=101><ACK=301><CTL=ACK>	--> ESTABLISHED

Simultaneous Connection Synchronization

Figure 8.

The principle reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion. To deal with this, a special control message, reset, has been devised. If the receiving TCP is in a non-synchronized state (i.e., SYN-SENT, SYN-RECEIVED), it returns to LISTEN on receiving an acceptable reset. If the TCP is in one of the synchronized states (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), it aborts the connection and informs its user. We discuss this latter case under "half-open" connections below.

TCP A		TCP B
1. CLOSED		LISTEN
2. SYN-SENT	--> <SEQ=100><CTL=SYN>	...
3. (duplicate)	... <SEQ=90><CTL=SYN>	--> SYN-RECEIVED
4. SYN-SENT	<-- <SEQ=300><ACK=91><CTL=SYN,ACK>	<-- SYN-RECEIVED
5. SYN-SENT	--> <SEQ=91><CTL=RST>	--> LISTEN
6.	... <SEQ=100><CTL=SYN>	--> SYN-RECEIVED
7. SYN-SENT	<-- <SEQ=400><ACK=101><CTL=SYN,ACK>	<-- SYN-RECEIVED
8. ESTABLISHED	--> <SEQ=101><ACK=401><CTL=ACK>	--> ESTABLISHED

Recovery from Old Duplicate SYN

Figure 9.

As a simple example of recovery from old duplicates, consider figure 9. At line 3, an old duplicate SYN arrives at TCP B. TCP B cannot tell that this is an old duplicate, so it responds normally (line 4). TCP A detects that the ACK field is incorrect and returns a RST (reset) with its SEQ field selected to make the segment believable. TCP B, on receiving the RST, returns to the LISTEN state. When the original SYN (pun intended) finally arrives at line 6, the synchronization proceeds normally. If the SYN at line 6 had arrived before the RST, a more complex exchange might have occurred with RST's sent in both directions.

Half-Open Connections and Other Anomalies

An established connection is said to be "half-open" if one of the TCPs has closed or aborted the connection at its end without the knowledge of the other, or if the two ends of the connection have become desynchronized owing to a crash that resulted in loss of memory. Such connections will automatically become reset if an attempt is made to send data in either direction. However, half-open connections are expected to be unusual, and the recovery procedure is mildly involved.

If at site A the connection no longer exists, then an attempt by the

Transmission Control Protocol
Functional Specification

user at site B to send any data on it will result in the site B TCP receiving a reset control message. Such a message indicates to the site B TCP that something is wrong, and it is expected to abort the connection.

Assume that two user processes A and B are communicating with one another when a crash occurs causing loss of memory to A's TCP. Depending on the operating system supporting A's TCP, it is likely that some error recovery mechanism exists. When the TCP is up again, A is likely to start again from the beginning or from a recovery point. As a result, A will probably try to OPEN the connection again or try to SEND on the connection it believes open. In the latter case, it receives the error message "connection not open" from the local (A's) TCP. In an attempt to establish the connection, A's TCP will send a segment containing SYN. This scenario leads to the example shown in figure 10. After TCP A crashes, the user attempts to re-open the connection. TCP B, in the meantime, thinks the connection is open.

TCP A	TCP B
1. (CRASH)	(send 300, receive 100)
2. CLOSED	ESTABLISHED
3. SYN-SENT --> <SEQ=400><CTL=SYN>	--> (??)
4. (!!) <-- <SEQ=300><ACK=100><CTL=ACK>	<-- ESTABLISHED
5. SYN-SENT --> <SEQ=100><CTL=RST>	--> (Abort!!)
6. SYN-SENT	CLOSED
7. SYN-SENT --> <SEQ=400><CTL=SYN>	-->

Half-Open Connection Discovery

Figure 10.

When the SYN arrives at line 3, TCP B, being in a synchronized state, and the incoming segment outside the window, responds with an acknowledgment indicating what sequence it next expects to hear (ACK 100). TCP A sees that this segment does not acknowledge anything it sent and, being unsynchronized, sends a reset (RST) because it has detected a half-open connection. TCP B aborts at line 5. TCP A will

continue to try to establish the connection; the problem is now reduced to the basic 3-way handshake of figure 7.

An interesting alternative case occurs when TCP A crashes and TCP B tries to send data on what it thinks is a synchronized connection. This is illustrated in figure 11. In this case, the data arriving at TCP A from TCP B (line 2) is unacceptable because no such connection exists, so TCP A sends a RST. The RST is acceptable so TCP B processes it and aborts the connection.

TCP A	TCP B
1. (CRASH)	(send 300, receive 100)
2. (??) <-- <SEQ=300><ACK=100><DATA=10><CTL=ACK>	<-- ESTABLISHED
3. --> <SEQ=100><CTL=RST>	--> (ABORT!!)

Active Side Causes Half-Open Connection Discovery

Figure 11.

In figure 12, we find the two TCPs A and B with passive connections waiting for SYN. An old duplicate arriving at TCP B (line 2) stirs B into action. A SYN-ACK is returned (line 3) and causes TCP A to generate a RST (the ACK in line 3 is not acceptable). TCP B accepts the reset and returns to its passive LISTEN state.

TCP A	TCP B
1. LISTEN	LISTEN
2. ... <SEQ=Z><CTL=SYN>	--> SYN-RECEIVED
3. (??) <-- <SEQ=X><ACK=Z+1><CTL=SYN,ACK>	<-- SYN-RECEIVED
4. --> <SEQ=Z+1><CTL=RST>	--> (return to LISTEN!)
5. LISTEN	LISTEN

Old Duplicate SYN Initiates a Reset on two Passive Sockets

Figure 12.

Transmission Control Protocol
Functional Specification

A variety of other cases are possible, all of which are accounted for by the following rules for RST generation and processing.

Reset Generation

As a general rule, reset (RST) must be sent whenever a segment arrives which apparently is not intended for the current connection. A reset must not be sent if it is not clear that this is the case.

There are three groups of states:

1. If the connection does not exist (CLOSED) then a reset is sent in response to any incoming segment except another reset. In particular, SYNs addressed to a non-existent connection are rejected by this means.

If the incoming segment has an ACK field, the reset takes its sequence number from the ACK field of the segment, otherwise the reset has sequence number zero and the ACK field is set to the sum of the sequence number and segment length of the incoming segment. The connection remains in the CLOSED state.

2. If the connection is in any non-synchronized state (LISTEN, SYN-SENT, SYN-RECEIVED), and the incoming segment acknowledges something not yet sent (the segment carries an unacceptable ACK), or if an incoming segment has a security level or compartment which does not exactly match the level and compartment requested for the connection, a reset is sent.

If our SYN has not been acknowledged and the precedence level of the incoming segment is higher than the precedence level requested then either raise the local precedence level (if allowed by the user and the system) or send a reset; or if the precedence level of the incoming segment is lower than the precedence level requested then continue as if the precedence matched exactly (if the remote TCP cannot raise the precedence level to match ours this will be detected in the next segment it sends, and the connection will be terminated then). If our SYN has been acknowledged (perhaps in this incoming segment) the precedence level of the incoming segment must match the local precedence level exactly, if it does not a reset must be sent.

If the incoming segment has an ACK field, the reset takes its sequence number from the ACK field of the segment, otherwise the reset has sequence number zero and the ACK field is set to the sum of the sequence number and segment length of the incoming segment. The connection remains in the same state.

3. If the connection is in a synchronized state (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT), any unacceptable segment (out of window sequence number or unacceptable acknowledgment number) must elicit only an empty acknowledgment segment containing the current send-sequence number and an acknowledgment indicating the next sequence number expected to be received, and the connection remains in the same state.

If an incoming segment has a security level, or compartment, or precedence which does not exactly match the level, and compartment, and precedence requested for the connection, a reset is sent and connection goes to the CLOSED state. The reset takes its sequence number from the ACK field of the incoming segment.

Reset Processing

In all states except SYN-SENT, all reset (RST) segments are validated by checking their SEQ-fields. A reset is valid if its sequence number is in the window. In the SYN-SENT state (a RST received in response to an initial SYN), the RST is acceptable if the ACK field acknowledges the SYN.

The receiver of a RST first validates it, then changes state. If the receiver was in the LISTEN state, it ignores it. If the receiver was in SYN-RECEIVED state and had previously been in the LISTEN state, then the receiver returns to the LISTEN state, otherwise the receiver aborts the connection and goes to the CLOSED state. If the receiver was in any other state, it aborts the connection and advises the user and goes to the CLOSED state.

3.5. Closing a Connection

CLOSE is an operation meaning "I have no more data to send." The notion of closing a full-duplex connection is subject to ambiguous interpretation, of course, since it may not be obvious how to treat the receiving side of the connection. We have chosen to treat CLOSE in a simplex fashion. The user who CLOSEs may continue to RECEIVE until he is told that the other side has CLOSED also. Thus, a program could initiate several SENDs followed by a CLOSE, and then continue to RECEIVE until signaled that a RECEIVE failed because the other side has CLOSED. We assume that the TCP will signal a user, even if no RECEIVES are outstanding, that the other side has closed, so the user can terminate his side gracefully. A TCP will reliably deliver all buffers SENT before the connection was CLOSED so a user who expects no data in return need only wait to hear the connection was CLOSED successfully to know that all his data was received at the destination TCP. Users must keep reading connections they close for sending until the TCP says no more data.

Transmission Control Protocol
Functional Specification

There are essentially three cases:

- 1) The user initiates by telling the TCP to CLOSE the connection
- 2) The remote TCP initiates by sending a FIN control signal
- 3) Both users CLOSE simultaneously

Case 1: Local user initiates the close

In this case, a FIN segment can be constructed and placed on the outgoing segment queue. No further SENDs from the user will be accepted by the TCP, and it enters the FIN-WAIT-1 state. RECEIVES are allowed in this state. All segments preceding and including FIN will be retransmitted until acknowledged. When the other TCP has both acknowledged the FIN and sent a FIN of its own, the first TCP can ACK this FIN. Note that a TCP receiving a FIN will ACK but not send its own FIN until its user has CLOSED the connection also.

Case 2: TCP receives a FIN from the network

If an unsolicited FIN arrives from the network, the receiving TCP can ACK it and tell the user that the connection is closing. The user will respond with a CLOSE, upon which the TCP can send a FIN to the other TCP after sending any remaining data. The TCP then waits until its own FIN is acknowledged whereupon it deletes the connection. If an ACK is not forthcoming, after the user timeout the connection is aborted and the user is told.

Case 3: both users close simultaneously

A simultaneous CLOSE by users at both ends of a connection causes FIN segments to be exchanged. When all segments preceding the FINs have been processed and acknowledged, each TCP can ACK the FIN it has received. Both will, upon receiving these ACKs, delete the connection.

TCP A		TCP B
1. ESTABLISHED		ESTABLISHED
2. (Close) FIN-WAIT-1	--> <SEQ=100><ACK=300><CTL=FIN,ACK>	--> CLOSE-WAIT
3. FIN-WAIT-2	<-- <SEQ=300><ACK=101><CTL=ACK>	<-- CLOSE-WAIT
4. TIME-WAIT	<-- <SEQ=300><ACK=101><CTL=FIN,ACK>	(Close) <-- LAST-ACK
5. TIME-WAIT	--> <SEQ=101><ACK=301><CTL=ACK>	--> CLOSED
6. (2 MSL) CLOSED		

Normal Close Sequence

Figure 13.

TCP A		TCP B
1. ESTABLISHED		ESTABLISHED
2. (Close) FIN-WAIT-1	--> <SEQ=100><ACK=300><CTL=FIN,ACK> <-- <SEQ=300><ACK=100><CTL=FIN,ACK> ... <SEQ=100><ACK=300><CTL=FIN,ACK>	(Close) ... FIN-WAIT-1 <-- -->
3. CLOSING	--> <SEQ=101><ACK=301><CTL=ACK> <-- <SEQ=301><ACK=101><CTL=ACK> ... <SEQ=101><ACK=301><CTL=ACK>	... CLOSING <-- -->
4. TIME-WAIT (2 MSL) CLOSED		TIME-WAIT (2 MSL) CLOSED

Simultaneous Close Sequence

Figure 14.

Transmission Control Protocol
Functional Specification

3.6. Precedence and Security

The intent is that connection be allowed only between ports operating with exactly the same security and compartment values and at the higher of the precedence level requested by the two ports.

The precedence and security parameters used in TCP are exactly those defined in the Internet Protocol (IP) [2]. Throughout this TCP specification the term "security/compartment" is intended to indicate the security parameters used in IP including security, compartment, user group, and handling restriction.

A connection attempt with mismatched security/compartment values or a lower precedence value must be rejected by sending a reset. Rejecting a connection due to too low a precedence only occurs after an acknowledgment of the SYN has been received.

Note that TCP modules which operate only at the default value of precedence will still have to check the precedence of incoming segments and possibly raise the precedence level they use on the connection.

The security parameters may be used even in a non-secure environment (the values would indicate unclassified data), thus hosts in non-secure environments must be prepared to receive the security parameters, though they need not send them.

3.7. Data Communication

Once the connection is established data is communicated by the exchange of segments. Because segments may be lost due to errors (checksum test failure), or network congestion, TCP uses retransmission (after a timeout) to ensure delivery of every segment. Duplicate segments may arrive due to network or TCP retransmission. As discussed in the section on sequence numbers the TCP performs certain tests on the sequence and acknowledgment numbers in the segments to verify their acceptability.

The sender of data keeps track of the next sequence number to use in the variable SND.NXT. The receiver of data keeps track of the next sequence number to expect in the variable RCV.NXT. The sender of data keeps track of the oldest unacknowledged sequence number in the variable SND.UNA. If the data flow is momentarily idle and all data sent has been acknowledged then the three variables will be equal.

When the sender creates a segment and transmits it the sender advances SND.NXT. When the receiver accepts a segment it advances RCV.NXT and sends an acknowledgment. When the data sender receives an

acknowledgment it advances SND.UNA. The extent to which the values of these variables differ is a measure of the delay in the communication. The amount by which the variables are advanced is the length of the data in the segment. Note that once in the ESTABLISHED state all segments must carry current acknowledgment information.

The CLOSE user call implies a push function, as does the FIN control flag in an incoming segment.

Retransmission Timeout

Because of the variability of the networks that compose an internetwork system and the wide range of uses of TCP connections the retransmission timeout must be dynamically determined. One procedure for determining a retransmission time out is given here as an illustration.

An Example Retransmission Timeout Procedure

Measure the elapsed time between sending a data octet with a particular sequence number and receiving an acknowledgment that covers that sequence number (segments sent do not have to match segments received). This measured elapsed time is the Round Trip Time (RTT). Next compute a Smoothed Round Trip Time (SRTT) as:

$$\text{SRTT} = (\text{ALPHA} * \text{SRTT}) + ((1-\text{ALPHA}) * \text{RTT})$$

and based on this, compute the retransmission timeout (RTO) as:

$$\text{RTO} = \min[\text{UBOUND}, \max[\text{LBOUND}, (\text{BETA} * \text{SRTT})]]$$

where UBOUND is an upper bound on the timeout (e.g., 1 minute), LBOUND is a lower bound on the timeout (e.g., 1 second), ALPHA is a smoothing factor (e.g., .8 to .9), and BETA is a delay variance factor (e.g., 1.3 to 2.0).

The Communication of Urgent Information

The objective of the TCP urgent mechanism is to allow the sending user to stimulate the receiving user to accept some urgent data and to permit the receiving TCP to indicate to the receiving user when all the currently known urgent data has been received by the user.

This mechanism permits a point in the data stream to be designated as the end of urgent information. Whenever this point is in advance of the receive sequence number (RCV.NXT) at the receiving TCP, that TCP must tell the user to go into "urgent mode"; when the receive sequence number catches up to the urgent pointer, the TCP must tell user to go

Transmission Control Protocol
Functional Specification

into "normal mode". If the urgent pointer is updated while the user is in "urgent mode", the update will be invisible to the user.

The method employs a urgent field which is carried in all segments transmitted. The URG control flag indicates that the urgent field is meaningful and must be added to the segment sequence number to yield the urgent pointer. The absence of this flag indicates that there is no urgent data outstanding.

To send an urgent indication the user must also send at least one data octet. If the sending user also indicates a push, timely delivery of the urgent information to the destination process is enhanced.

Managing the Window

The window sent in each segment indicates the range of sequence numbers the sender of the window (the data receiver) is currently prepared to accept. There is an assumption that this is related to the currently available data buffer space available for this connection.

Indicating a large window encourages transmissions. If more data arrives than can be accepted, it will be discarded. This will result in excessive retransmissions, adding unnecessarily to the load on the network and the TCPs. Indicating a small window may restrict the transmission of data to the point of introducing a round trip delay between each new segment transmitted.

The mechanisms provided allow a TCP to advertise a large window and to subsequently advertise a much smaller window without having accepted that much data. This, so called "shrinking the window," is strongly discouraged. The robustness principle dictates that TCPs will not shrink the window themselves, but will be prepared for such behavior on the part of other TCPs.

The sending TCP must be prepared to accept from the user and send at least one octet of new data even if the send window is zero. The sending TCP must regularly retransmit to the receiving TCP even when the window is zero. Two minutes is recommended for the retransmission interval when the window is zero. This retransmission is essential to guarantee that when either TCP has a zero window the re-opening of the window will be reliably reported to the other.

When the receiving TCP has a zero window and a segment arrives it must still send an acknowledgment showing its next expected sequence number and current window (zero).

The sending TCP packages the data to be transmitted into segments

which fit the current window, and may repackage segments on the retransmission queue. Such repackaging is not required, but may be helpful.

In a connection with a one-way data flow, the window information will be carried in acknowledgment segments that all have the same sequence number so there will be no way to reorder them if they arrive out of order. This is not a serious problem, but it will allow the window information to be on occasion temporarily based on old reports from the data receiver. A refinement to avoid this problem is to act on the window information from segments that carry the highest acknowledgment number (that is segments with acknowledgment number equal or greater than the highest previously received).

The window management procedure has significant influence on the communication performance. The following comments are suggestions to implementers.

Window Management Suggestions

Allocating a very small window causes data to be transmitted in many small segments when better performance is achieved using fewer large segments.

One suggestion for avoiding small windows is for the receiver to defer updating a window until the additional allocation is at least X percent of the maximum allocation possible for the connection (where X might be 20 to 40).

Another suggestion is for the sender to avoid sending small segments by waiting until the window is large enough before sending data. If the the user signals a push function then the data must be sent even if it is a small segment.

Note that the acknowledgments should not be delayed or unnecessary retransmissions will result. One strategy would be to send an acknowledgment when a small segment arrives (with out updating the window information), and then to send another acknowledgment with new window information when the window is larger.

The segment sent to probe a zero window may also begin a break up of transmitted data into smaller and smaller segments. If a segment containing a single data octet sent to probe a zero window is accepted, it consumes one octet of the window now available. If the sending TCP simply sends as much as it can whenever the window is non zero, the transmitted data will be broken into alternating big and small segments. As time goes on, occasional pauses in the receiver making window allocation available will

Transmission Control Protocol Functional Specification

result in breaking the big segments into a small and not quite so big pair. And after a while the data transmission will be in mostly small segments.

The suggestion here is that the TCP implementations need to actively attempt to combine small window allocations into larger windows, since the mechanisms for managing the window tend to lead to many small windows in the simplest minded implementations.

3.8. Interfaces

There are of course two interfaces of concern: the user/TCP interface and the TCP/lower-level interface. We have a fairly elaborate model of the user/TCP interface, but the interface to the lower level protocol module is left unspecified here, since it will be specified in detail by the specification of the lower level protocol. For the case that the lower level is IP we note some of the parameter values that TCPs might use.

User/TCP Interface

The following functional description of user commands to the TCP is, at best, fictional, since every operating system will have different facilities. Consequently, we must warn readers that different TCP implementations may have different user interfaces. However, all TCPs must provide a certain minimum set of services to guarantee that all TCP implementations can support the same protocol hierarchy. This section specifies the functional interfaces required of all TCP implementations.

TCP User Commands

The following sections functionally characterize a USER/TCP interface. The notation used is similar to most procedure or function calls in high level languages, but this usage is not meant to rule out trap type service calls (e.g., SVCs, UUOs, EMTs).

The user commands described below specify the basic functions the TCP must perform to support interprocess communication. Individual implementations must define their own exact format, and may provide combinations or subsets of the basic functions in single calls. In particular, some implementations may wish to automatically OPEN a connection on the first SEND or RECEIVE issued by the user for a given connection.

In providing interprocess communication facilities, the TCP must not only accept commands, but must also return information to the processes it serves. The latter consists of:

- (a) general information about a connection (e.g., interrupts, remote close, binding of unspecified foreign socket).
- (b) replies to specific user commands indicating success or various types of failure.

Open

Format: OPEN (local port, foreign socket, active/passive
[, timeout] [, precedence] [, security/compartment] [, options])
-> local connection name

We assume that the local TCP is aware of the identity of the processes it serves and will check the authority of the process to use the connection specified. Depending upon the implementation of the TCP, the local network and TCP identifiers for the source address will either be supplied by the TCP or the lower level protocol (e.g., IP). These considerations are the result of concern about security, to the extent that no TCP be able to masquerade as another one, and so on. Similarly, no process can masquerade as another without the collusion of the TCP.

If the active/passive flag is set to passive, then this is a call to LISTEN for an incoming connection. A passive open may have either a fully specified foreign socket to wait for a particular connection or an unspecified foreign socket to wait for any call. A fully specified passive call can be made active by the subsequent execution of a SEND.

A transmission control block (TCB) is created and partially filled in with data from the OPEN command parameters.

On an active OPEN command, the TCP will begin the procedure to synchronize (i.e., establish) the connection at once.

The timeout, if present, permits the caller to set up a timeout for all data submitted to TCP. If data is not successfully delivered to the destination within the timeout period, the TCP will abort the connection. The present global default is five minutes.

The TCP or some component of the operating system will verify the users authority to open a connection with the specified

Transmission Control Protocol
Functional Specification

precedence or security/compartment. The absence of precedence or security/compartment specification in the OPEN call indicates the default values must be used.

TCP will accept incoming requests as matching only if the security/compartment information is exactly the same and only if the precedence is equal to or higher than the precedence requested in the OPEN call.

The precedence for the connection is the higher of the values requested in the OPEN call and received from the incoming request, and fixed at that value for the life of the connection. Implementers may want to give the user control of this precedence negotiation. For example, the user might be allowed to specify that the precedence must be exactly matched, or that any attempt to raise the precedence be confirmed by the user.

A local connection name will be returned to the user by the TCP. The local connection name can then be used as a short hand term for the connection defined by the <local socket, foreign socket> pair.

Send

Format: SEND (local connection name, buffer address, byte count, PUSH flag, URGENT flag [,timeout])

This call causes the data contained in the indicated user buffer to be sent on the indicated connection. If the connection has not been opened, the SEND is considered an error. Some implementations may allow users to SEND first; in which case, an automatic OPEN would be done. If the calling process is not authorized to use this connection, an error is returned.

If the PUSH flag is set, the data must be transmitted promptly to the receiver, and the PUSH bit will be set in the last TCP segment created from the buffer. If the PUSH flag is not set, the data may be combined with data from subsequent SENDs for transmission efficiency.

If the URGENT flag is set, segments sent to the destination TCP will have the urgent pointer set. The receiving TCP will signal the urgent condition to the receiving process if the urgent pointer indicates that data preceding the urgent pointer has not been consumed by the receiving process. The purpose of urgent is to stimulate the receiver to process the urgent data and to indicate to the receiver when all the currently known urgent

data has been received. The number of times the sending user's TCP signals urgent will not necessarily be equal to the number of times the receiving user will be notified of the presence of urgent data.

If no foreign socket was specified in the OPEN, but the connection is established (e.g., because a LISTENing connection has become specific due to a foreign segment arriving for the local socket), then the designated buffer is sent to the implied foreign socket. Users who make use of OPEN with an unspecified foreign socket can make use of SEND without ever explicitly knowing the foreign socket address.

However, if a SEND is attempted before the foreign socket becomes specified, an error will be returned. Users can use the STATUS call to determine the status of the connection. In some implementations the TCP may notify the user when an unspecified socket is bound.

If a timeout is specified, the current user timeout for this connection is changed to the new one.

In the simplest implementation, SEND would not return control to the sending process until either the transmission was complete or the timeout had been exceeded. However, this simple method is both subject to deadlocks (for example, both sides of the connection might try to do SENDs before doing any RECEIVES) and offers poor performance, so it is not recommended. A more sophisticated implementation would return immediately to allow the process to run concurrently with network I/O, and, furthermore, to allow multiple SENDs to be in progress. Multiple SENDs are served in first come, first served order, so the TCP will queue those it cannot service immediately.

We have implicitly assumed an asynchronous user interface in which a SEND later elicits some kind of SIGNAL or pseudo-interrupt from the serving TCP. An alternative is to return a response immediately. For instance, SENDs might return immediate local acknowledgment, even if the segment sent had not been acknowledged by the distant TCP. We could optimistically assume eventual success. If we are wrong, the connection will close anyway due to the timeout. In implementations of this kind (synchronous), there will still be some asynchronous signals, but these will deal with the connection itself, and not with specific segments or buffers.

In order for the process to distinguish among error or success indications for different SENDs, it might be appropriate for the

Transmission Control Protocol
Functional Specification

buffer address to be returned along with the coded response to the SEND request. TCP-to-user signals are discussed below, indicating the information which should be returned to the calling process.

Receive

Format: RECEIVE (local connection name, buffer address, byte count) -> byte count, urgent flag, push flag

This command allocates a receiving buffer associated with the specified connection. If no OPEN precedes this command or the calling process is not authorized to use this connection, an error is returned.

In the simplest implementation, control would not return to the calling program until either the buffer was filled, or some error occurred, but this scheme is highly subject to deadlocks. A more sophisticated implementation would permit several RECEIVES to be outstanding at once. These would be filled as segments arrive. This strategy permits increased throughput at the cost of a more elaborate scheme (possibly asynchronous) to notify the calling program that a PUSH has been seen or a buffer filled.

If enough data arrive to fill the buffer before a PUSH is seen, the PUSH flag will not be set in the response to the RECEIVE. The buffer will be filled with as much data as it can hold. If a PUSH is seen before the buffer is filled the buffer will be returned partially filled and PUSH indicated.

If there is urgent data the user will have been informed as soon as it arrived via a TCP-to-user signal. The receiving user should thus be in "urgent mode". If the URGENT flag is on, additional urgent data remains. If the URGENT flag is off, this call to RECEIVE has returned all the urgent data, and the user may now leave "urgent mode". Note that data following the urgent pointer (non-urgent data) cannot be delivered to the user in the same buffer with preceeding urgent data unless the boundary is clearly marked for the user.

To distinguish among several outstanding RECEIVES and to take care of the case that a buffer is not completely filled, the return code is accompanied by both a buffer pointer and a byte count indicating the actual length of the data received.

Alternative implementations of RECEIVE might have the TCP

allocate buffer storage, or the TCP might share a ring buffer with the user.

Close

Format: CLOSE (local connection name)

This command causes the connection specified to be closed. If the connection is not open or the calling process is not authorized to use this connection, an error is returned. Closing connections is intended to be a graceful operation in the sense that outstanding SENDs will be transmitted (and retransmitted), as flow control permits, until all have been serviced. Thus, it should be acceptable to make several SEND calls, followed by a CLOSE, and expect all the data to be sent to the destination. It should also be clear that users should continue to RECEIVE on CLOSING connections, since the other side may be trying to transmit the last of its data. Thus, CLOSE means "I have no more to send" but does not mean "I will not receive any more." It may happen (if the user level protocol is not well thought out) that the closing side is unable to get rid of all its data before timing out. In this event, CLOSE turns into ABORT, and the closing TCP gives up.

The user may CLOSE the connection at any time on his own initiative, or in response to various prompts from the TCP (e.g., remote close executed, transmission timeout exceeded, destination inaccessible).

Because closing a connection requires communication with the foreign TCP, connections may remain in the closing state for a short time. Attempts to reopen the connection before the TCP replies to the CLOSE command will result in error responses.

Close also implies push function.

Status

Format: STATUS (local connection name) -> status data

This is an implementation dependent user command and could be excluded without adverse effect. Information returned would typically come from the TCB associated with the connection.

This command returns a data block containing the following information:

local socket,

Transmission Control Protocol Functional Specification

foreign socket,
local connection name,
receive window,
send window,
connection state,
number of buffers awaiting acknowledgment,
number of buffers pending receipt,
urgent state,
precedence,
security/compartiment,
and transmission timeout.

Depending on the state of the connection, or on the implementation itself, some of this information may not be available or meaningful. If the calling process is not authorized to use this connection, an error is returned. This prevents unauthorized processes from gaining information about a connection.

Abort

Format: ABORT (local connection name)

This command causes all pending SENDs and RECEIVES to be aborted, the TCB to be removed, and a special RESET message to be sent to the TCP on the other side of the connection. Depending on the implementation, users may receive abort indications for each outstanding SEND or RECEIVE, or may simply receive an ABORT-acknowledgment.

TCP-to-User Messages

It is assumed that the operating system environment provides a means for the TCP to asynchronously signal the user program. When the TCP does signal a user program, certain information is passed to the user. Often in the specification the information will be an error message. In other cases there will be information relating to the completion of processing a SEND or RECEIVE or other user call.

The following information is provided:

Local Connection Name	Always
Response String	Always
Buffer Address	Send & Receive
Byte count (counts bytes received)	Receive
Push flag	Receive
Urgent flag	Receive

TCP/Lower-Level Interface

The TCP calls on a lower level protocol module to actually send and receive information over a network. One case is that of the ARPA internetwork system where the lower level module is the Internet Protocol (IP) [2].

If the lower level protocol is IP it provides arguments for a type of service and for a time to live. TCP uses the following settings for these parameters:

Type of Service = Precedence: routine, Delay: normal, Throughput: normal, Reliability: normal; or 00000000.

Time to Live = one minute, or 00111100.

Note that the assumed maximum segment lifetime is two minutes. Here we explicitly ask that a segment be destroyed if it cannot be delivered by the internet system within one minute.

If the lower level is IP (or other protocol that provides this feature) and source routing is used, the interface must allow the route information to be communicated. This is especially important so that the source and destination addresses used in the TCP checksum be the originating source and ultimate destination. It is also important to preserve the return route to answer connection requests.

Any lower level protocol will have to provide the source address, destination address, and protocol fields, and some way to determine the "TCP length", both to provide the functional equivalent service of IP and to be used in the TCP checksum.

Transmission Control Protocol
Functional Specification

3.9. Event Processing

The processing depicted in this section is an example of one possible implementation. Other implementations may have slightly different processing sequences, but they should differ from those in this section only in detail, not in substance.

The activity of the TCP can be characterized as responding to events. The events that occur can be cast into three categories: user calls, arriving segments, and timeouts. This section describes the processing the TCP does in response to each of the events. In many cases the processing required depends on the state of the connection.

Events that occur:

User Calls

OPEN
SEND
RECEIVE
CLOSE
ABORT
STATUS

Arriving Segments

SEGMENT ARRIVES

Timeouts

USER TIMEOUT
RETRANSMISSION TIMEOUT
TIME-WAIT TIMEOUT

The model of the TCP/user interface is that user commands receive an immediate return and possibly a delayed response via an event or pseudo interrupt. In the following descriptions, the term "signal" means cause a delayed response.

Error responses are given as character strings. For example, user commands referencing connections that do not exist receive "error: connection not open".

Please note in the following that all arithmetic on sequence numbers, acknowledgment numbers, windows, et cetera, is modulo 2^{32} the size of the sequence number space. Also note that " $=<$ " means less than or equal to (modulo 2^{32}).

September 1981

Transmission Control Protocol
Functional Specification

A natural way to think about processing incoming segments is to imagine that they are first tested for proper sequence number (i.e., that their contents lie in the range of the expected "receive window" in the sequence number space) and then that they are generally queued and processed in sequence number order.

When a segment overlaps other already received segments we reconstruct the segment to contain just the new data, and adjust the header fields to be consistent.

Note that if no state change is mentioned the TCP stays in the same state.

OPEN Call

CLOSED STATE (i.e., TCB does not exist)

Create a new transmission control block (TCB) to hold connection state information. Fill in local socket identifier, foreign socket, precedence, security/compartments, and user timeout information. Note that some parts of the foreign socket may be unspecified in a passive OPEN and are to be filled in by the parameters of the incoming SYN segment. Verify the security and precedence requested are allowed for this user, if not return "error: precedence not allowed" or "error: security/compartments not allowed." If passive enter the LISTEN state and return. If active and the foreign socket is unspecified, return "error: foreign socket unspecified"; if active and the foreign socket is specified, issue a SYN segment. An initial send sequence number (ISS) is selected. A SYN segment of the form <SEQ=ISS><CTL=SYN> is sent. Set SND.UNA to ISS, SND.NXT to ISS+1, enter SYN-SENT state, and return.

If the caller does not have access to the local socket specified, return "error: connection illegal for this process". If there is no room to create a new connection, return "error: insufficient resources".

LISTEN STATE

If active and the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If Foreign socket was not specified, then return "error: foreign socket unspecified".

September 1981

Transmission Control Protocol
Functional Specification

OPEN Call

SYN-SENT STATE
SYN-RECEIVED STATE
ESTABLISHED STATE
FIN-WAIT-1 STATE
FIN-WAIT-2 STATE
CLOSE-WAIT STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Return "error: connection already exists".

Transmission Control Protocol
Functional Specification

SEND Call

SEND Call

CLOSED STATE (i.e., TCB does not exist)

If the user does not have access to such a connection, then return "error: connection illegal for this process".

Otherwise, return "error: connection does not exist".

LISTEN STATE

If the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment, set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. Data associated with SEND may be sent with SYN segment or queued for transmission after entering ESTABLISHED state. The urgent bit if requested in the command must be sent with the data segments sent as a result of this command. If there is no room to queue the request, respond with "error: insufficient resources". If Foreign socket was not specified, then return "error: foreign socket unspecified".

SYN-SENT STATE

SYN-RECEIVED STATE

Queue the data for transmission after entering ESTABLISHED state. If no space to queue, respond with "error: insufficient resources".

ESTABLISHED STATE

CLOSE-WAIT STATE

Segmentize the buffer and send it with a piggybacked acknowledgment (acknowledgment value = RCV.NXT). If there is insufficient space to remember this buffer, simply return "error: insufficient resources".

If the urgent flag is set, then SND.UP <- SND.NXT-1 and set the urgent pointer in the outgoing segments.

September 1981

Transmission Control Protocol
Functional Specification

SEND Call

FIN-WAIT-1 STATE
FIN-WAIT-2 STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Return "error: connection closing" and do not service request.

RECEIVE Call

RECEIVE Call

CLOSED STATE (i.e., TCB does not exist)

If the user does not have access to such a connection, return
"error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

SYN-SENT STATE

SYN-RECEIVED STATE

Queue for processing after entering ESTABLISHED state. If there
is no room to queue this request, respond with "error:
insufficient resources".

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

If insufficient incoming segments are queued to satisfy the
request, queue the request. If there is no queue space to
remember the RECEIVE, respond with "error: insufficient
resources".

Reassemble queued incoming segments into receive buffer and return
to user. Mark "push seen" (PUSH) if this is the case.

If RCV.UP is in advance of the data currently being passed to the
user notify the user of the presence of urgent data.

When the TCP takes responsibility for delivering data to the user
that fact must be communicated to the sender via an
acknowledgment. The formation of such an acknowledgment is
described below in the discussion of processing an incoming
segment.

September 1981

Transmission Control Protocol
Functional Specification

RECEIVE Call

CLOSE-WAIT STATE

Since the remote side has already sent FIN, RECEIVES must be satisfied by text already on hand, but not yet delivered to the user. If no text is awaiting delivery, the RECEIVE will get a "error: connection closing" response. Otherwise, any remaining text can be used to satisfy the RECEIVE.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Return "error: connection closing".

CLOSE Call

CLOSED STATE (i.e., TCB does not exist)

If the user does not have access to such a connection, return "error: connection illegal for this process".

Otherwise, return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVES are returned with "error: closing" responses. Delete TCB, enter CLOSED state, and return.

SYN-SENT STATE

Delete the TCB and return "error: closing" responses to any queued SENDs, or RECEIVES.

SYN-RECEIVED STATE

If no SENDs have been issued and there is no pending data to send, then form a FIN segment and send it, and enter FIN-WAIT-1 state; otherwise queue for processing after entering ESTABLISHED state.

ESTABLISHED STATE

Queue this until all preceding SENDs have been segmentized, then form a FIN segment and send it. In any case, enter FIN-WAIT-1 state.

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

Strictly speaking, this is an error and should receive a "error: connection closing" response. An "ok" response would be acceptable, too, as long as a second FIN is not emitted (the first FIN may be retransmitted though).

September 1981

Transmission Control Protocol
Functional Specification

CLOSE Call

CLOSE-WAIT STATE

Queue this request until all preceding SENDs have been
segmentized; then send a FIN segment, enter CLOSING state.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Respond with "error: connection closing".

ABORT Call

CLOSED STATE (i.e., TCB does not exist)

If the user should not have access to such a connection, return "error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Any outstanding RECEIVES should be returned with "error: connection reset" responses. Delete TCB, enter CLOSED state, and return.

SYN-SENT STATE

All queued SENDs and RECEIVES should be given "connection reset" notification, delete the TCB, enter CLOSED state, and return.

SYN-RECEIVED STATE

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

CLOSE-WAIT STATE

Send a reset segment:

<SEQ=SND.NXT><CTL=RST>

All queued SENDs and RECEIVES should be given "connection reset" notification; all segments queued for transmission (except for the RST formed above) or retransmission should be flushed, delete the TCB, enter CLOSED state, and return.

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

Respond with "ok" and delete the TCB, enter CLOSED state, and return.

STATUS Call

STATUS Call

CLOSED STATE (i.e., TCB does not exist)

If the user should not have access to such a connection, return
"error: connection illegal for this process".

Otherwise return "error: connection does not exist".

LISTEN STATE

Return "state = LISTEN", and the TCB pointer.

SYN-SENT STATE

Return "state = SYN-SENT", and the TCB pointer.

SYN-RECEIVED STATE

Return "state = SYN-RECEIVED", and the TCB pointer.

ESTABLISHED STATE

Return "state = ESTABLISHED", and the TCB pointer.

FIN-WAIT-1 STATE

Return "state = FIN-WAIT-1", and the TCB pointer.

FIN-WAIT-2 STATE

Return "state = FIN-WAIT-2", and the TCB pointer.

CLOSE-WAIT STATE

Return "state = CLOSE-WAIT", and the TCB pointer.

CLOSING STATE

Return "state = CLOSING", and the TCB pointer.

LAST-ACK STATE

Return "state = LAST-ACK", and the TCB pointer.

TIME-WAIT STATE

Return "state = TIME-WAIT", and the TCB pointer.

September 1981

Transmission Control Protocol
Functional Specification

SEGMENT ARRIVES

SEGMENT ARRIVES

If the state is CLOSED (i.e., TCB does not exist) then

all data in the incoming segment is discarded. An incoming segment containing a RST is discarded. An incoming segment not containing a RST causes a RST to be sent in response. The acknowledgment and sequence field values are selected to make the reset sequence acceptable to the TCP that sent the offending segment.

If the ACK bit is off, sequence number zero is used,

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If the ACK bit is on,

<SEQ=SEG.ACK><CTL=RST>

Return.

If the state is LISTEN then

first check for an RST

An incoming RST should be ignored. Return.

second check for an ACK

Any acknowledgment is bad if it arrives on a connection still in the LISTEN state. An acceptable reset segment should be formed for any arriving ACK-bearing segment. The RST should be formatted as follows:

<SEQ=SEG.ACK><CTL=RST>

Return.

third check for a SYN

If the SYN bit is set, check the security. If the security/compartments on the incoming segment does not exactly match the security/compartments in the TCB then send a reset and return.

<SEQ=SEG.ACK><CTL=RST>

September 1981

SEGMENT ARRIVES

If the SEG.PRC is greater than the TCB.PRC then if allowed by the user and the system set TCB.PRC<-SEG.PRC, if not allowed send a reset and return.

<SEQ=SEG.ACK><CTL=RST>

If the SEG.PRC is less than the TCB.PRC then continue.

Set RCV.NXT to SEG.SEQ+1, IRS is set to SEG.SEQ and any other control or text should be queued for processing later. ISS should be selected and a SYN segment sent of the form:

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

SND.NXT is set to ISS+1 and SND.UNA to ISS. The connection state should be changed to SYN-RECEIVED. Note that any other incoming control or data (combined with SYN) will be processed in the SYN-RECEIVED state, but processing of SYN and ACK should not be repeated. If the listen was not fully specified (i.e., the foreign socket was not fully specified), then the unspecified fields should be filled in now.

fourth other text or control

Any other control or text-bearing segment (not containing SYN) must have an ACK and thus would be discarded by the ACK processing. An incoming RST segment could not be valid, since it could not have been sent in response to anything sent by this incarnation of the connection. So you are unlikely to get here, but if you do, drop the segment, and return.

If the state is SYN-SENT then

first check the ACK bit

If the ACK bit is set

If SEG.ACK =< ISS, or SEG.ACK > SND.NXT, send a reset (unless the RST bit is set, if so drop the segment and return)

<SEQ=SEG.ACK><CTL=RST>

and discard the segment. Return.

If SND.UNA =< SEG.ACK =< SND.NXT then the ACK is acceptable.

second check the RST bit

September 1981

Transmission Control Protocol
Functional Specification

SEGMENT ARRIVES

If the RST bit is set

If the ACK was acceptable then signal the user "error: connection reset", drop the segment, enter CLOSED state, delete TCB, and return. Otherwise (no ACK) drop the segment and return.

third check the security and precedence

If the security/compartiment in the segment does not exactly match the security/compartiment in the TCB, send a reset

If there is an ACK

<SEQ=SEG.ACK><CTL=RST>

Otherwise

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If there is an ACK

The precedence in the segment must match the precedence in the TCB, if not, send a reset

<SEQ=SEG.ACK><CTL=RST>

If there is no ACK

If the precedence in the segment is higher than the precedence in the TCB then if allowed by the user and the system raise the precedence in the TCB to that in the segment, if not allowed to raise the prec then send a reset.

<SEQ=0><ACK=SEG.SEQ+SEG.LEN><CTL=RST,ACK>

If the precedence in the segment is lower than the precedence in the TCB continue.

If a reset was sent, discard the segment and return.

fourth check the SYN bit

This step should be reached only if the ACK is ok, or there is no ACK, and if the segment did not contain a RST.

If the SYN bit is on and the security/compartiment and precedence

September 1981

SEGMENT ARRIVES

are acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ. SND.UNA should be advanced to equal SEG.ACK (if there is an ACK), and any segments on the retransmission queue which are thereby acknowledged should be removed.

If SND.UNA > ISS (our SYN has been ACKed), change the connection state to ESTABLISHED, form an ACK segment

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

and send it. Data or controls which were queued for transmission may be included. If there are other controls or text in the segment then continue processing at the sixth step below where the URG bit is checked, otherwise return.

Otherwise enter SYN-RECEIVED, form a SYN,ACK segment

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

and send it. If there are other controls or text in the segment, queue them for processing after the ESTABLISHED state has been reached, return.

fifth, if neither of the SYN or RST bits is set then drop the segment and return.

September 1981

Transmission Control Protocol
Functional Specification

SEGMENT ARRIVES

Otherwise,

first check sequence number

SYN-RECEIVED STATE
ESTABLISHED STATE
FIN-WAIT-1 STATE
FIN-WAIT-2 STATE
CLOSE-WAIT STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Segments are processed in sequence. Initial tests on arrival are used to discard old duplicates, but further processing is done in SEG.SEQ order. If a segment's contents straddle the boundary between old and new, only the new parts should be processed.

There are four cases for the acceptability test for an incoming segment:

Segment Length	Receive Window	Test
0	0	SEG.SEQ = RCV.NXT
0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND
>0	0	not acceptable
>0	>0	RCV.NXT =< SEG.SEQ < RCV.NXT+RCV.WND or RCV.NXT =< SEG.SEQ+SEG.LEN-1 < RCV.NXT+RCV.WND

If the RCV.WND is zero, no segments will be acceptable, but special allowance should be made to accept valid ACKs, URGs and RSTs.

If an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set, if so drop the segment and return):

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

After sending the acknowledgment, drop the unacceptable segment and return.

September 1981

SEGMENT ARRIVES

In the following it is assumed that the segment is the idealized segment that begins at RCV.NXT and does not exceed the window. One could tailor actual segments to fit this assumption by trimming off any portions that lie outside the window (including SYN and FIN), and only processing further if the segment then begins at RCV.NXT. Segments with higher beginning sequence numbers may be held for later processing.

second check the RST bit,

SYN-RECEIVED STATE

If the RST bit is set

If this connection was initiated with a passive OPEN (i.e., came from the LISTEN state), then return this connection to LISTEN state and return. The user need not be informed. If this connection was initiated with an active OPEN (i.e., came from SYN-SENT state) then the connection was refused, signal the user "connection refused". In either case, all segments on the retransmission queue should be removed. And in the active OPEN case, enter the CLOSED state and delete the TCB, and return.

ESTABLISHED
FIN-WAIT-1
FIN-WAIT-2
CLOSE-WAIT

If the RST bit is set then, any outstanding RECEIVES and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

CLOSING STATE
LAST-ACK STATE
TIME-WAIT

If the RST bit is set then, enter the CLOSED state, delete the TCB, and return.

September 1981

Transmission Control Protocol
Functional Specification

SEGMENT ARRIVES

third check security and precedence

SYN-RECEIVED

If the security/compartments and precedence in the segment do not exactly match the security/compartments and precedence in the TCB then send a reset, and return.

ESTABLISHED STATE

If the security/compartments and precedence in the segment do not exactly match the security/compartments and precedence in the TCB then send a reset, any outstanding RECEIVES and SEND should receive "reset" responses. All segment queues should be flushed. Users should also receive an unsolicited general "connection reset" signal. Enter the CLOSED state, delete the TCB, and return.

Note this check is placed following the sequence check to prevent a segment from an old connection between these ports with a different security or precedence from causing an abort of the current connection.

fourth, check the SYN bit,

SYN-RECEIVED

ESTABLISHED STATE

FIN-WAIT STATE-1

FIN-WAIT STATE-2

CLOSE-WAIT STATE

CLOSING STATE

LAST-ACK STATE

TIME-WAIT STATE

If the SYN is in the window it is an error, send a reset, any outstanding RECEIVES and SEND should receive "reset" responses, all segment queues should be flushed, the user should also receive an unsolicited general "connection reset" signal, enter the CLOSED state, delete the TCB, and return.

If the SYN is not in the window this step would not be reached and an ack would have been sent in the first step (sequence number check).

SEGMENT ARRIVES

fifth check the ACK field,

if the ACK bit is off drop the segment and return

if the ACK bit is on

SYN-RECEIVED STATE

If $\text{SND.UNA} \leq \text{SEG.ACK} \leq \text{SND.NXT}$ then enter ESTABLISHED state and continue processing.

If the segment acknowledgment is not acceptable, form a reset segment,

$\langle \text{SEQ} = \text{SEG.ACK} \rangle \langle \text{CTL} = \text{RST} \rangle$

and send it.

ESTABLISHED STATE

If $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$ then, set $\text{SND.UNA} \leftarrow \text{SEG.ACK}$. Any segments on the retransmission queue which are thereby entirely acknowledged are removed. Users should receive positive acknowledgments for buffers which have been SENT and fully acknowledged (i.e., SEND buffer should be returned with "ok" response). If the ACK is a duplicate ($\text{SEG.ACK} < \text{SND.UNA}$), it can be ignored. If the ACK acks something not yet sent ($\text{SEG.ACK} > \text{SND.NXT}$) then send an ACK, drop the segment, and return.

If $\text{SND.UNA} < \text{SEG.ACK} \leq \text{SND.NXT}$, the send window should be updated. If ($\text{SND.WL1} < \text{SEG.SEQ}$ or ($\text{SND.WL1} = \text{SEG.SEQ}$ and $\text{SND.WL2} \leq \text{SEG.ACK}$)), set $\text{SND.WND} \leftarrow \text{SEG.WND}$, set $\text{SND.WL1} \leftarrow \text{SEG.SEQ}$, and set $\text{SND.WL2} \leftarrow \text{SEG.ACK}$.

Note that SND.WND is an offset from SND.UNA , that SND.WL1 records the sequence number of the last segment used to update SND.WND , and that SND.WL2 records the acknowledgment number of the last segment used to update SND.WND . The check here prevents using old segments to update the window.

September 1981

Transmission Control Protocol
Functional Specification

SEGMENT ARRIVES

FIN-WAIT-1 STATE

In addition to the processing for the ESTABLISHED state, if our FIN is now acknowledged then enter FIN-WAIT-2 and continue processing in that state.

FIN-WAIT-2 STATE

In addition to the processing for the ESTABLISHED state, if the retransmission queue is empty, the user's CLOSE can be acknowledged ("ok") but do not delete the TCB.

CLOSE-WAIT STATE

Do the same processing as for the ESTABLISHED state.

CLOSING STATE

In addition to the processing for the ESTABLISHED state, if the ACK acknowledges our FIN then enter the TIME-WAIT state, otherwise ignore the segment.

LAST-ACK STATE

The only thing that can arrive in this state is an acknowledgment of our FIN. If our FIN is now acknowledged, delete the TCB, enter the CLOSED state, and return.

TIME-WAIT STATE

The only thing that can arrive in this state is a retransmission of the remote FIN. Acknowledge it, and restart the 2 MSL timeout.

sixth, check the URG bit,

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

If the URG bit is set, $RCV.UP \leftarrow \max(RCV.UP, SEG.UP)$, and signal the user that the remote side has urgent data if the urgent pointer (RCV.UP) is in advance of the data consumed. If the user has already been signaled (or is still in the "urgent mode") for this continuous sequence of urgent data, do not signal the user again.

September 1981

SEGMENT ARRIVES

CLOSE-WAIT STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT

This should not occur, since a FIN has been received from the remote side. Ignore the URG.

seventh, process the segment text,

ESTABLISHED STATE
FIN-WAIT-1 STATE
FIN-WAIT-2 STATE

Once in the ESTABLISHED state, it is possible to deliver segment text to user RECEIVE buffers. Text from segments can be moved into buffers until either the buffer is full or the segment is empty. If the segment empties and carries an PUSH flag, then the user is informed, when the buffer is returned, that a PUSH has been received.

When the TCP takes responsibility for delivering the data to the user it must also acknowledge the receipt of the data.

Once the TCP takes responsibility for the data it advances RCV.NXT over the data accepted, and adjusts RCV.WND as appropriate to the current buffer availability. The total of RCV.NXT and RCV.WND should not be reduced.

Please note the window management suggestions in section 3.7.

Send an acknowledgment of the form:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

This acknowledgment should be piggybacked on a segment being transmitted if possible without incurring undue delay.

September 1981

Transmission Control Protocol
Functional Specification

SEGMENT ARRIVES

CLOSE-WAIT STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

This should not occur, since a FIN has been received from the remote side. Ignore the segment text.

eighth, check the FIN bit,

Do not process the FIN if the state is CLOSED, LISTEN or SYN-SENT since the SEG.SEQ cannot be validated; drop the segment and return.

If the FIN bit is set, signal the user "connection closing" and return any pending RECEIVES with same message, advance RCV.NXT over the FIN, and send an acknowledgment for the FIN. Note that FIN implies PUSH for any segment text not yet delivered to the user.

SYN-RECEIVED STATE
ESTABLISHED STATE

Enter the CLOSE-WAIT state.

FIN-WAIT-1 STATE

If our FIN has been ACKed (perhaps in this segment), then enter TIME-WAIT, start the time-wait timer, turn off the other timers; otherwise enter the CLOSING state.

FIN-WAIT-2 STATE

Enter the TIME-WAIT state. Start the time-wait timer, turn off the other timers.

CLOSE-WAIT STATE

Remain in the CLOSE-WAIT state.

CLOSING STATE

Remain in the CLOSING state.

LAST-ACK STATE

Remain in the LAST-ACK state.

SEGMENT ARRIVES

TIME-WAIT STATE

Remain in the TIME-WAIT state. Restart the 2 MSL time-wait
timeout.

and return.

September 1981

Transmission Control Protocol
Functional Specification

USER TIMEOUT

USER TIMEOUT

For any state if the user timeout expires, flush all queues, signal the user "error: connection aborted due to user timeout" in general and for any outstanding calls, delete the TCB, enter the CLOSED state and return.

RETRANSMISSION TIMEOUT

For any state if the retransmission timeout expires on a segment in the retransmission queue, send the segment at the front of the retransmission queue again, reinitialize the retransmission timer, and return.

TIME-WAIT TIMEOUT

If the time-wait timeout expires on a connection delete the TCB, enter the CLOSED state and return.

September 1981

Transmission Control Protocol

GLOSSARY

1822

BBN Report 1822, "The Specification of the Interconnection of a Host and an IMP". The specification of interface between a host and the ARPANET.

ACK

A control bit (acknowledge) occupying no sequence space, which indicates that the acknowledgment field of this segment specifies the next sequence number the sender of this segment is expecting to receive, hence acknowledging receipt of all previous sequence numbers.

ARPANET message

The unit of transmission between a host and an IMP in the ARPANET. The maximum size is about 1012 octets (8096 bits).

ARPANET packet

A unit of transmission used internally in the ARPANET between IMPs. The maximum size is about 126 octets (1008 bits).

connection

A logical communication path identified by a pair of sockets.

datagram

A message sent in a packet switched computer communications network.

Destination Address

The destination address, usually the network and host identifiers.

FIN

A control bit (finis) occupying one sequence number, which indicates that the sender will send no more data or control occupying sequence space.

fragment

A portion of a logical unit of data, in particular an internet fragment is a portion of an internet datagram.

FTP

A file transfer protocol.

Transmission Control Protocol
Glossary

header	Control information at the beginning of a message, segment, fragment, packet or block of data.
host	A computer. In particular a source or destination of messages from the point of view of the communication network.
Identification	An Internet Protocol field. This identifying value assigned by the sender aids in assembling the fragments of a datagram.
IMP	The Interface Message Processor, the packet switch of the ARPANET.
internet address	A source or destination address specific to the host level.
internet datagram	The unit of data exchanged between an internet module and the higher level protocol together with the internet header.
internet fragment	A portion of the data of an internet datagram with an internet header.
IP	Internet Protocol.
IRS	The Initial Receive Sequence number. The first sequence number used by the sender on a connection.
ISN	The Initial Sequence Number. The first sequence number used on a connection, (either ISS or IRS). Selected on a clock based procedure.
ISS	The Initial Send Sequence number. The first sequence number used by the sender on a connection.
leader	Control information at the beginning of a message or block of data. In particular, in the ARPANET, the control information on an ARPANET message at the host-IMP interface.

left sequence

This is the next sequence number to be acknowledged by the data receiving TCP (or the lowest currently unacknowledged sequence number) and is sometimes referred to as the left edge of the send window.

local packet

The unit of transmission within a local network.

module

An implementation, usually in software, of a protocol or other procedure.

MSL

Maximum Segment Lifetime, the time a TCP segment can exist in the internetwork system. Arbitrarily defined to be 2 minutes.

octet

An eight bit byte.

Options

An Option field may contain several options, and each option may be several octets in length. The options are used primarily in testing situations; for example, to carry timestamps. Both the Internet Protocol and TCP provide for options fields.

packet

A package of data with a header which may or may not be logically complete. More often a physical packaging than a logical packaging of data.

port

The portion of a socket that specifies which logical input or output channel of a process is associated with the data.

process

A program in execution. A source or destination of data from the point of view of the TCP or other host-to-host protocol.

PUSH

A control bit occupying no sequence space, indicating that this segment contains data that must be pushed through to the receiving user.

RCV.NXT

receive next sequence number

Transmission Control Protocol
Glossary

RCV.UP	receive urgent pointer
RCV.WND	receive window
receive next sequence number	This is the next sequence number the local TCP is expecting to receive.
receive window	This represents the sequence numbers the local (receiving) TCP is willing to receive. Thus, the local TCP considers that segments overlapping the range RCV.NXT to RCV.NXT + RCV.WND - 1 carry acceptable data or control. Segments containing sequence numbers entirely outside of this range are considered duplicates and discarded.
RST	A control bit (reset), occupying no sequence space, indicating that the receiver should delete the connection without further interaction. The receiver can determine, based on the sequence number and acknowledgment fields of the incoming segment, whether it should honor the reset command or ignore it. In no case does receipt of a segment containing RST give rise to a RST in response.
RTP	Real Time Protocol: A host-to-host protocol for communication of time critical information.
SEG.ACK	segment acknowledgment
SEG.LEN	segment length
SEG.PRC	segment precedence value
SEG.SEQ	segment sequence
SEG.UP	segment urgent pointer field

SEG.WND	segment window field
segment	A logical unit of data, in particular a TCP segment is the unit of data transfered between a pair of TCP modules.
segment acknowledgment	The sequence number in the acknowledgment field of the arriving segment.
segment length	The amount of sequence number space occupied by a segment, including any controls which occupy sequence space.
segment sequence	The number in the sequence field of the arriving segment.
send sequence	This is the next sequence number the local (sending) TCP will use on the connection. It is initially selected from an initial sequence number curve (ISN) and is incremented for each octet of data or sequenced control transmitted.
send window	This represents the sequence numbers which the remote (receiving) TCP is willing to receive. It is the value of the window field specified in segments from the remote (data receiving) TCP. The range of new sequence numbers which may be emitted by a TCP lies between SND.NXT and $\text{SND.UNA} + \text{SND.WND} - 1$. (Retransmissions of sequence numbers between SND.UNA and SND.NXT are expected, of course.)
SND.NXT	send sequence
SND.UNA	left sequence
SND.UP	send urgent pointer
SND.WL1	segment sequence number at last window update
SND.WL2	segment acknowledgment number at last window update

Transmission Control Protocol
Glossary

SND.WND

send window

socket

An address which specifically includes a port identifier, that is, the concatenation of an Internet Address with a TCP port.

Source Address

The source address, usually the network and host identifiers.

SYN

A control bit in the incoming segment, occupying one sequence number, used at the initiation of a connection, to indicate where the sequence numbering will start.

TCB

Transmission control block, the data structure that records the state of a connection.

TCB.PRC

The precedence of the connection.

TCP

Transmission Control Protocol: A host-to-host protocol for reliable communication in internetwork environments.

TOS

Type of Service, an Internet Protocol field.

Type of Service

An Internet Protocol field which indicates the type of service for this internet fragment.

URG

A control bit (urgent), occupying no sequence space, used to indicate that the receiving user should be notified to do urgent processing as long as there is data to be consumed with sequence numbers less than the value indicated in the urgent pointer.

urgent pointer

A control field meaningful only when the URG bit is on. This field communicates the value of the urgent pointer which indicates the data octet associated with the sending user's urgent call.

REFERENCES

- [1] Cerf, V., and R. Kahn, "A Protocol for Packet Network Intercommunication", IEEE Transactions on Communications, Vol. COM-22, No. 5, pp 637-648, May 1974.
- [2] Postel, J. (ed.), "Internet Protocol - DARPA Internet Program Protocol Specification", RFC 791, USC/Information Sciences Institute, September 1981.
- [3] Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols", Computer Networks, Vol. 2, No. 6, pp. 454-473, December 1978.
- [4] Postel, J., "Assigned Numbers", RFC 790, USC/Information Sciences Institute, September 1981.

RFC: 791

INTERNET PROTOCOL

DARPA INTERNET PROGRAM

PROTOCOL SPECIFICATION

September 1981

prepared for

Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, Virginia 22209

by

Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, California 90291

TABLE OF CONTENTS

PREFACE	iii
1. INTRODUCTION	1
1.1 Motivation	1
1.2 Scope	1
1.3 Interfaces	1
1.4 Operation	2
2. OVERVIEW	5
2.1 Relation to Other Protocols	9
2.2 Model of Operation	5
2.3 Function Description	7
2.4 Gateways	9
3. SPECIFICATION	11
3.1 Internet Header Format	11
3.2 Discussion	23
3.3 Interfaces	31
APPENDIX A: Examples & Scenarios	34
APPENDIX B: Data Transmission Order	39
GLOSSARY	41
REFERENCES	45

September 1981

Internet Protocol

PREFACE

This document specifies the DoD Standard Internet Protocol. This document is based on six earlier editions of the ARPA Internet Protocol Specification, and the present text draws heavily from them. There have been many contributors to this work both in terms of concepts and in terms of text. This edition revises aspects of addressing, error handling, option codes, and the security, precedence, compartments, and handling restriction features of the internet protocol.

Jon Postel

Editor

September 1981

RFC: 791
Replaces: RFC 760
IENs 128, 123, 111,
80, 54, 44, 41, 28, 26

INTERNET PROTOCOL

DARPA INTERNET PROGRAM PROTOCOL SPECIFICATION

1. INTRODUCTION

1.1. Motivation

The Internet Protocol is designed for use in interconnected systems of packet-switched computer communication networks. Such a system has been called a "catenet" [1]. The internet protocol provides for transmitting blocks of data called datagrams from sources to destinations, where sources and destinations are hosts identified by fixed length addresses. The internet protocol also provides for fragmentation and reassembly of long datagrams, if necessary, for transmission through "small packet" networks.

1.2. Scope

The internet protocol is specifically limited in scope to provide the functions necessary to deliver a package of bits (an internet datagram) from a source to a destination over an interconnected system of networks. There are no mechanisms to augment end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols. The internet protocol can capitalize on the services of its supporting networks to provide various types and qualities of service.

1.3. Interfaces

This protocol is called on by host-to-host protocols in an internet environment. This protocol calls on local network protocols to carry the internet datagram to the next gateway or destination host.

For example, a TCP module would call on the internet module to take a TCP segment (including the TCP header and user data) as the data portion of an internet datagram. The TCP module would provide the addresses and other parameters in the internet header to the internet module as arguments of the call. The internet module would then create an internet datagram and call on the local network interface to transmit the internet datagram.

In the ARPANET case, for example, the internet module would call on a

Internet Protocol Introduction

local net module which would add the 1822 leader [2] to the internet datagram creating an ARPANET message to transmit to the IMP. The ARPANET address would be derived from the internet address by the local network interface and would be the address of some host in the ARPANET, that host might be a gateway to other networks.

1.4. Operation

The internet protocol implements two basic functions: addressing and fragmentation.

The internet modules use the addresses carried in the internet header to transmit internet datagrams toward their destinations. The selection of a path for transmission is called routing.

The internet modules use fields in the internet header to fragment and reassemble internet datagrams when necessary for transmission through "small packet" networks.

The model of operation is that an internet module resides in each host engaged in internet communication and in each gateway that interconnects networks. These modules share common rules for interpreting address fields and for fragmenting and assembling internet datagrams. In addition, these modules (especially in gateways) have procedures for making routing decisions and other functions.

The internet protocol treats each internet datagram as an independent entity unrelated to any other internet datagram. There are no connections or logical circuits (virtual or otherwise).

The internet protocol uses four key mechanisms in providing its service: Type of Service, Time to Live, Options, and Header Checksum.

The Type of Service is used to indicate the quality of the service desired. The type of service is an abstract or generalized set of parameters which characterize the service choices provided in the networks that make up the internet. This type of service indication is to be used by gateways to select the actual transmission parameters for a particular network, the network to be used for the next hop, or the next gateway when routing an internet datagram.

The Time to Live is an indication of an upper bound on the lifetime of an internet datagram. It is set by the sender of the datagram and reduced at the points along the route where it is processed. If the time to live reaches zero before the internet datagram reaches its destination, the internet datagram is destroyed. The time to live can be thought of as a self destruct time limit.

September 1981

Internet Protocol
Introduction

The Options provide for control functions needed or useful in some situations but unnecessary for the most common communications. The options include provisions for timestamps, security, and special routing.

The Header Checksum provides a verification that the information used in processing internet datagram has been transmitted correctly. The data may contain errors. If the header checksum fails, the internet datagram is discarded at once by the entity which detects the error.

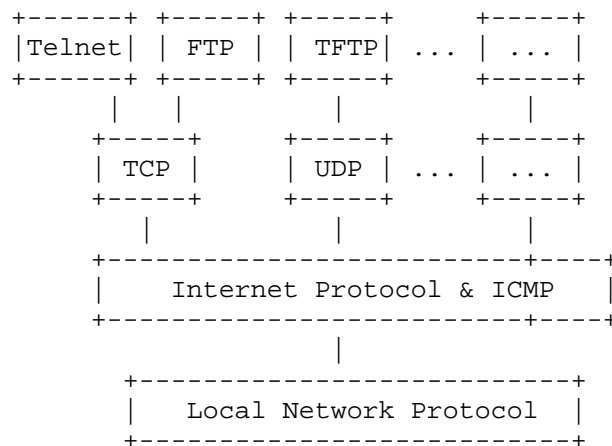
The internet protocol does not provide a reliable communication facility. There are no acknowledgments either end-to-end or hop-by-hop. There is no error control for data, only a header checksum. There are no retransmissions. There is no flow control.

Errors detected may be reported via the Internet Control Message Protocol (ICMP) [3] which is implemented in the internet protocol module.

2. OVERVIEW

2.1. Relation to Other Protocols

The following diagram illustrates the place of the internet protocol in the protocol hierarchy:



Protocol Relationships

Figure 1.

Internet protocol interfaces on one side to the higher level host-to-host protocols and on the other side to the local network protocol. In this context a "local network" may be a small network in a building or a large network such as the ARPANET.

2.2. Model of Operation

The model of operation for transmitting a datagram from one application program to another is illustrated by the following scenario:

We suppose that this transmission will involve one intermediate gateway.

The sending application program prepares its data and calls on its local internet module to send that data as a datagram and passes the destination address and other parameters as arguments of the call.

The internet module prepares a datagram header and attaches the data to it. The internet module determines a local network address for this internet address, in this case it is the address of a gateway.

Internet Protocol Overview

It sends this datagram and the local network address to the local network interface.

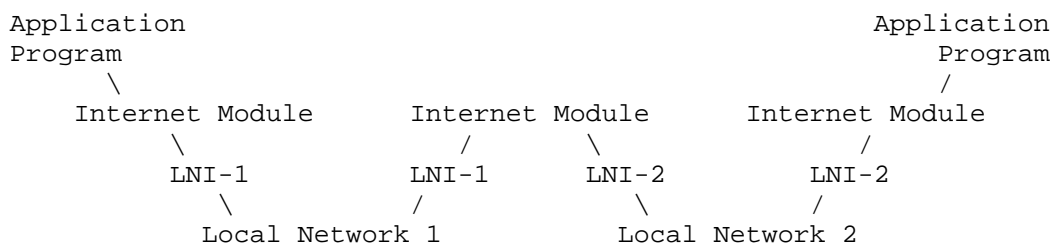
The local network interface creates a local network header, and attaches the datagram to it, then sends the result via the local network.

The datagram arrives at a gateway host wrapped in the local network header, the local network interface strips off this header, and turns the datagram over to the internet module. The internet module determines from the internet address that the datagram is to be forwarded to another host in a second network. The internet module determines a local net address for the destination host. It calls on the local network interface for that network to send the datagram.

This local network interface creates a local network header and attaches the datagram sending the result to the destination host.

At this destination host the datagram is stripped of the local net header by the local network interface and handed to the internet module.

The internet module determines that the datagram is for an application program in this host. It passes the data to the application program in response to a system call, passing the source address and other parameters as results of the call.



Transmission Path

Figure 2

2.3. Function Description

The function or purpose of Internet Protocol is to move datagrams through an interconnected set of networks. This is done by passing the datagrams from one internet module to another until the destination is reached. The internet modules reside in hosts and gateways in the internet system. The datagrams are routed from one internet module to another through individual networks based on the interpretation of an internet address. Thus, one important mechanism of the internet protocol is the internet address.

In the routing of messages from one internet module to another, datagrams may need to traverse a network whose maximum packet size is smaller than the size of the datagram. To overcome this difficulty, a fragmentation mechanism is provided in the internet protocol.

Addressing

A distinction is made between names, addresses, and routes [4]. A name indicates what we seek. An address indicates where it is. A route indicates how to get there. The internet protocol deals primarily with addresses. It is the task of higher level (i.e., host-to-host or application) protocols to make the mapping from names to addresses. The internet module maps internet addresses to local net addresses. It is the task of lower level (i.e., local net or gateways) procedures to make the mapping from local net addresses to routes.

Addresses are fixed length of four octets (32 bits). An address begins with a network number, followed by local address (called the "rest" field). There are three formats or classes of internet addresses: in class a, the high order bit is zero, the next 7 bits are the network, and the last 24 bits are the local address; in class b, the high order two bits are one-zero, the next 14 bits are the network and the last 16 bits are the local address; in class c, the high order three bits are one-one-zero, the next 21 bits are the network and the last 8 bits are the local address.

Care must be taken in mapping internet addresses to local net addresses; a single physical host must be able to act as if it were several distinct hosts to the extent of using several distinct internet addresses. Some hosts will also have several physical interfaces (multi-homing).

That is, provision must be made for a host to have several physical interfaces to the network with each having several logical internet addresses.

Internet Protocol Overview

Examples of address mappings may be found in "Address Mappings" [5].

Fragmentation

Fragmentation of an internet datagram is necessary when it originates in a local net that allows a large packet size and must traverse a local net that limits packets to a smaller size to reach its destination.

An internet datagram can be marked "don't fragment." Any internet datagram so marked is not to be internet fragmented under any circumstances. If internet datagram marked don't fragment cannot be delivered to its destination without fragmenting it, it is to be discarded instead.

Fragmentation, transmission and reassembly across a local network which is invisible to the internet protocol module is called intranet fragmentation and may be used [6].

The internet fragmentation and reassembly procedure needs to be able to break a datagram into an almost arbitrary number of pieces that can be later reassembled. The receiver of the fragments uses the identification field to ensure that fragments of different datagrams are not mixed. The fragment offset field tells the receiver the position of a fragment in the original datagram. The fragment offset and length determine the portion of the original datagram covered by this fragment. The more-fragments flag indicates (by being reset) the last fragment. These fields provide sufficient information to reassemble datagrams.

The identification field is used to distinguish the fragments of one datagram from those of another. The originating protocol module of an internet datagram sets the identification field to a value that must be unique for that source-destination pair and protocol for the time the datagram will be active in the internet system. The originating protocol module of a complete datagram sets the more-fragments flag to zero and the fragment offset to zero.

To fragment a long internet datagram, an internet protocol module (for example, in a gateway), creates two new internet datagrams and copies the contents of the internet header fields from the long datagram into both new internet headers. The data of the long datagram is divided into two portions on a 8 octet (64 bit) boundary (the second portion might not be an integral multiple of 8 octets, but the first must be). Call the number of 8 octet blocks in the first portion NFB (for Number of Fragment Blocks). The first portion of the data is placed in the first new internet datagram, and the total length field is set to the length of the first

datagram. The more-fragments flag is set to one. The second portion of the data is placed in the second new internet datagram, and the total length field is set to the length of the second datagram. The more-fragments flag carries the same value as the long datagram. The fragment offset field of the second new internet datagram is set to the value of that field in the long datagram plus NFB.

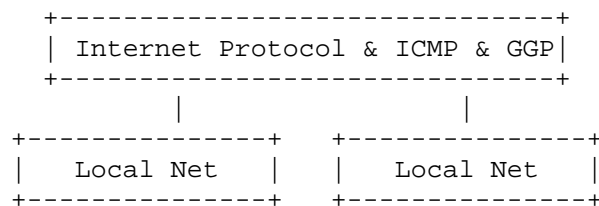
This procedure can be generalized for an n-way split, rather than the two-way split described.

To assemble the fragments of an internet datagram, an internet protocol module (for example at a destination host) combines internet datagrams that all have the same value for the four fields: identification, source, destination, and protocol. The combination is done by placing the data portion of each fragment in the relative position indicated by the fragment offset in that fragment's internet header. The first fragment will have the fragment offset zero, and the last fragment will have the more-fragments flag reset to zero.

2.4. Gateways

Gateways implement internet protocol to forward datagrams between networks. Gateways also implement the Gateway to Gateway Protocol (GGP) [7] to coordinate routing and other internet control information.

In a gateway the higher level protocols need not be implemented and the GGP functions are added to the IP module.



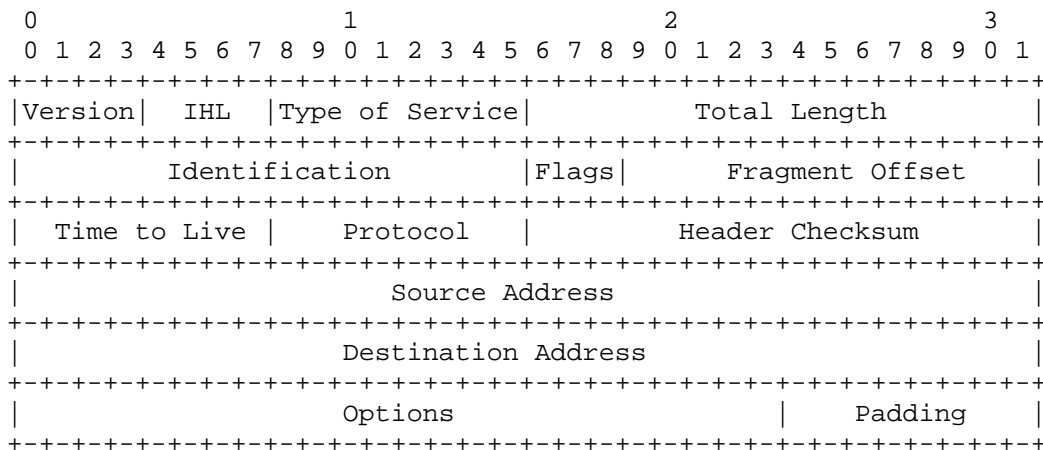
Gateway Protocols

Figure 3.

3. SPECIFICATION

3.1. Internet Header Format

A summary of the contents of the internet header follows:



Example Internet Datagram Header

Figure 4.

Note that each tick mark represents one bit position.

Version: 4 bits

The Version field indicates the format of the internet header. This document describes version 4.

IHL: 4 bits

Internet Header Length is the length of the internet header in 32 bit words, and thus points to the beginning of the data. Note that the minimum value for a correct header is 5.

Internet Protocol
Specification

Type of Service: 8 bits

The Type of Service provides an indication of the abstract parameters of the quality of service desired. These parameters are to be used to guide the selection of the actual service parameters when transmitting a datagram through a particular network. Several networks offer service precedence, which somehow treats high precedence traffic as more important than other traffic (generally by accepting only traffic above a certain precedence at time of high load). The major choice is a three way tradeoff between low-delay, high-reliability, and high-throughput.

Bits 0-2: Precedence.

Bit 3: 0 = Normal Delay, 1 = Low Delay.

Bits 4: 0 = Normal Throughput, 1 = High Throughput.

Bits 5: 0 = Normal Reliability, 1 = High Reliability.

Bit 6-7: Reserved for Future Use.

0	1	2	3	4	5	6	7
PRECEDENCE			D	T	R	0	0

Precedence

- 111 - Network Control
- 110 - Internetwork Control
- 101 - CRITIC/ECP
- 100 - Flash Override
- 011 - Flash
- 010 - Immediate
- 001 - Priority
- 000 - Routine

The use of the Delay, Throughput, and Reliability indications may increase the cost (in some sense) of the service. In many networks better performance for one of these parameters is coupled with worse performance on another. Except for very unusual cases at most two of these three indications should be set.

The type of service is used to specify the treatment of the datagram during its transmission through the internet system. Example mappings of the internet type of service to the actual service provided on networks such as AUTODIN II, ARPANET, SATNET, and PRNET is given in "Service Mappings" [8].

The Network Control precedence designation is intended to be used within a network only. The actual use and control of that designation is up to each network. The Internetwork Control designation is intended for use by gateway control originators only. If the actual use of these precedence designations is of concern to a particular network, it is the responsibility of that network to control the access to, and use of, those precedence designations.

Total Length: 16 bits

Total Length is the length of the datagram, measured in octets, including internet header and data. This field allows the length of a datagram to be up to 65,535 octets. Such long datagrams are impractical for most hosts and networks. All hosts must be prepared to accept datagrams of up to 576 octets (whether they arrive whole or in fragments). It is recommended that hosts only send datagrams larger than 576 octets if they have assurance that the destination is prepared to accept the larger datagrams.

The number 576 is selected to allow a reasonable sized data block to be transmitted in addition to the required header information. For example, this size allows a data block of 512 octets plus 64 header octets to fit in a datagram. The maximal internet header is 60 octets, and a typical internet header is 20 octets, allowing a margin for headers of higher level protocols.

Identification: 16 bits

An identifying value assigned by the sender to aid in assembling the fragments of a datagram.

Flags: 3 bits

Various Control Flags.

Bit 0: reserved, must be zero

Bit 1: (DF) 0 = May Fragment, 1 = Don't Fragment.

Bit 2: (MF) 0 = Last Fragment, 1 = More Fragments.

0	1	2
+---+---+---+		
	D	M
	0	F F
+---+---+---+		

Fragment Offset: 13 bits

This field indicates where in the datagram this fragment belongs.

Internet Protocol
Specification

The fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset zero.

Time to Live: 8 bits

This field indicates the maximum time the datagram is allowed to remain in the internet system. If this field contains the value zero, then the datagram must be destroyed. This field is modified in internet header processing. The time is measured in units of seconds, but since every module that processes a datagram must decrease the TTL by at least one even if it process the datagram in less than a second, the TTL must be thought of only as an upper bound on the time a datagram may exist. The intention is to cause undeliverable datagrams to be discarded, and to bound the maximum datagram lifetime.

Protocol: 8 bits

This field indicates the next level protocol used in the data portion of the internet datagram. The values for various protocols are specified in "Assigned Numbers" [9].

Header Checksum: 16 bits

A checksum on the header only. Since some header fields change (e.g., time to live), this is recomputed and verified at each point that the internet header is processed.

The checksum algorithm is:

The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

This is a simple to compute checksum and experimental evidence indicates it is adequate, but it is provisional and may be replaced by a CRC procedure, depending on further experience.

Source Address: 32 bits

The source address. See section 3.2.

Destination Address: 32 bits

The destination address. See section 3.2.

Options: variable

The options may appear or not in datagrams. They must be implemented by all IP modules (host and gateways). What is optional is their transmission in any particular datagram, not their implementation.

In some environments the security option may be required in all datagrams.

The option field is variable in length. There may be zero or more options. There are two cases for the format of an option:

Case 1: A single octet of option-type.

Case 2: An option-type octet, an option-length octet, and the actual option-data octets.

The option-length octet counts the option-type octet and the option-length octet as well as the option-data octets.

The option-type octet is viewed as having 3 fields:

- 1 bit copied flag,
- 2 bits option class,
- 5 bits option number.

The copied flag indicates that this option is copied into all fragments on fragmentation.

- 0 = not copied
- 1 = copied

The option classes are:

- 0 = control
- 1 = reserved for future use
- 2 = debugging and measurement
- 3 = reserved for future use

The following internet options are defined:

CLASS	NUMBER	LENGTH	DESCRIPTION
0	0	-	End of Option list. This option occupies only 1 octet; it has no length octet.
0	1	-	No Operation. This option occupies only 1 octet; it has no length octet.
0	2	11	Security. Used to carry Security, Compartmentation, User Group (TCC), and Handling Restriction Codes compatible with DOD requirements.
0	3	var.	Loose Source Routing. Used to route the internet datagram based on information supplied by the source.
0	9	var.	Strict Source Routing. Used to route the internet datagram based on information supplied by the source.
0	7	var.	Record Route. Used to trace the route an internet datagram takes.
0	8	4	Stream ID. Used to carry the stream identifier.
2	4	var.	Internet Timestamp.

Specific Option Definitions

End of Option List

```
+-----+
|00000000|
+-----+
Type=0
```

This option indicates the end of the option list. This might not coincide with the end of the internet header according to the internet header length. This is used at the end of all options, not the end of each option, and need only be used if the end of the options would not otherwise coincide with the end of the internet header.

May be copied, introduced, or deleted on fragmentation, or for any other reason.

No Operation

```
+-----+
|00000001|
+-----+
Type=1
```

This option may be used between options, for example, to align the beginning of a subsequent option on a 32 bit boundary.

May be copied, introduced, or deleted on fragmentation, or for any other reason.

Security

This option provides a way for hosts to send security, compartmentation, handling restrictions, and TCC (closed user group) parameters. The format for this option is as follows:

```
+-----+-----+---//---+---//---+---//---+---//---+
|10000010|00001011|SSS SSS|CCC CCC|HHH HHH| TCC  |
+-----+-----+---//---+---//---+---+---//---+---+
Type=130 Length=11
```

Security (S field): 16 bits

Specifies one of 16 levels of security (eight of which are reserved for future use).

00000000	00000000	- Unclassified
11110001	00110101	- Confidential
01111000	10011010	- EFTO
10111100	01001101	- MMMM
01011110	00100110	- PROG
10101111	00010011	- Restricted
11010111	10001000	- Secret
01101011	11000101	- Top Secret
00110101	11100010	- (Reserved for future use)
10011010	11110001	- (Reserved for future use)
01001101	01111000	- (Reserved for future use)
00100100	10111101	- (Reserved for future use)
00010011	01011110	- (Reserved for future use)
10001001	10101111	- (Reserved for future use)
11000100	11010110	- (Reserved for future use)
11100010	01101011	- (Reserved for future use)

Internet Protocol
Specification

Compartments (C field): 16 bits

An all zero value is used when the information transmitted is not compartmented. Other values for the compartments field may be obtained from the Defense Intelligence Agency.

Handling Restrictions (H field): 16 bits

The values for the control and release markings are alphanumeric digraphs and are defined in the Defense Intelligence Agency Manual DIAM 65-19, "Standard Security Markings".

Transmission Control Code (TCC field): 24 bits

Provides a means to segregate traffic and define controlled communities of interest among subscribers. The TCC values are trigraphs, and are available from HQ DCA Code 530.

Must be copied on fragmentation. This option appears at most once in a datagram.

Loose Source and Record Route

```
+-----+-----+-----+-----//-----+
|10000011| length | pointer|      route data      |
+-----+-----+-----+-----//-----+
Type=131
```

The loose source and record route (LSRR) option provides a means for the source of an internet datagram to supply routing information to be used by the gateways in forwarding the datagram to the destination, and to record the route information.

The option begins with the option type code. The second octet is the option length which includes the option type code and the length octet, the pointer octet, and length-3 octets of route data. The third octet is the pointer into the route data indicating the octet which begins the next source address to be processed. The pointer is relative to this option, and the smallest legal value for the pointer is 4.

A route data is composed of a series of internet addresses. Each internet address is 32 bits or 4 octets. If the pointer is greater than the length, the source route is empty (and the recorded route full) and the routing is to be based on the destination address field.

If the address in destination address field has been reached and the pointer is not greater than the length, the next address in the source route replaces the address in the destination address field, and the recorded route address replaces the source address just used, and pointer is increased by four.

The recorded route address is the internet module's own internet address as known in the environment into which this datagram is being forwarded.

This procedure of replacing the source route with the recorded route (though it is in the reverse of the order it must be in to be used as a source route) means the option (and the IP header as a whole) remains a constant length as the datagram progresses through the internet.

This option is a loose source route because the gateway or host IP is allowed to use any route of any number of other intermediate gateways to reach the next address in the route.

Must be copied on fragmentation. Appears at most once in a datagram.

Strict Source and Record Route

```
+-----+-----+-----+-----//-----+
|10001001| length | pointer|      route data      |
+-----+-----+-----+-----//-----+
Type=137
```

The strict source and record route (SSRR) option provides a means for the source of an internet datagram to supply routing information to be used by the gateways in forwarding the datagram to the destination, and to record the route information.

The option begins with the option type code. The second octet is the option length which includes the option type code and the length octet, the pointer octet, and length-3 octets of route data. The third octet is the pointer into the route data indicating the octet which begins the next source address to be processed. The pointer is relative to this option, and the smallest legal value for the pointer is 4.

A route data is composed of a series of internet addresses. Each internet address is 32 bits or 4 octets. If the pointer is greater than the length, the source route is empty (and the

Internet Protocol
Specification

recorded route full) and the routing is to be based on the destination address field.

If the address in destination address field has been reached and the pointer is not greater than the length, the next address in the source route replaces the address in the destination address field, and the recorded route address replaces the source address just used, and pointer is increased by four.

The recorded route address is the internet module's own internet address as known in the environment into which this datagram is being forwarded.

This procedure of replacing the source route with the recorded route (though it is in the reverse of the order it must be in to be used as a source route) means the option (and the IP header as a whole) remains a constant length as the datagram progresses through the internet.

This option is a strict source route because the gateway or host IP must send the datagram directly to the next address in the source route through only the directly connected network indicated in the next address to reach the next gateway or host specified in the route.

Must be copied on fragmentation. Appears at most once in a datagram.

Record Route

```
+-----+-----+-----+-----//-----+
|00000111| length | pointer|      route data      |
+-----+-----+-----+-----//-----+
      Type=7
```

The record route option provides a means to record the route of an internet datagram.

The option begins with the option type code. The second octet is the option length which includes the option type code and the length octet, the pointer octet, and length-3 octets of route data. The third octet is the pointer into the route data indicating the octet which begins the next area to store a route address. The pointer is relative to this option, and the smallest legal value for the pointer is 4.

A recorded route is composed of a series of internet addresses. Each internet address is 32 bits or 4 octets. If the pointer is

greater than the length, the recorded route data area is full. The originating host must compose this option with a large enough route data area to hold all the address expected. The size of the option does not change due to adding addresses. The initial contents of the route data area must be zero.

When an internet module routes a datagram it checks to see if the record route option is present. If it is, it inserts its own internet address as known in the environment into which this datagram is being forwarded into the recorded route beginning at the octet indicated by the pointer, and increments the pointer by four.

If the route data area is already full (the pointer exceeds the length) the datagram is forwarded without inserting the address into the recorded route. If there is some room but not enough room for a full address to be inserted, the original datagram is considered to be in error and is discarded. In either case an ICMP parameter problem message may be sent to the source host [3].

Not copied on fragmentation, goes in first fragment only.
Appears at most once in a datagram.

Stream Identifier

```
+-----+-----+-----+-----+
|10001000|00000010|      Stream ID      |
+-----+-----+-----+-----+
```

Type=136 Length=4

This option provides a way for the 16-bit SATNET stream identifier to be carried through networks that do not support the stream concept.

Must be copied on fragmentation. Appears at most once in a datagram.

Internet Timestamp

```
+-----+-----+-----+-----+
|01000100| length | pointer|oflw|flg|
+-----+-----+-----+-----+
|               internet address               |
+-----+-----+-----+-----+
|               timestamp                       |
+-----+-----+-----+-----+
|               .                               |
|               .                               |
|               .                               |
```

Type = 68

The Option Length is the number of octets in the option counting the type, length, pointer, and overflow/flag octets (maximum length 40).

The Pointer is the number of octets from the beginning of this option to the end of timestamps plus one (i.e., it points to the octet beginning the space for next timestamp). The smallest legal value is 5. The timestamp area is full when the pointer is greater than the length.

The Overflow (oflw) [4 bits] is the number of IP modules that cannot register timestamps due to lack of space.

The Flag (flg) [4 bits] values are

- 0 -- time stamps only, stored in consecutive 32-bit words,
- 1 -- each timestamp is preceded with internet address of the registering entity,
- 3 -- the internet address fields are prespecified. An IP module only registers its timestamp if it matches its own address with the next specified internet address.

The Timestamp is a right-justified, 32-bit timestamp in milliseconds since midnight UT. If the time is not available in milliseconds or cannot be provided with respect to midnight UT then any time may be inserted as a timestamp provided the high order bit of the timestamp field is set to one to indicate the use of a non-standard value.

The originating host must compose this option with a large enough timestamp data area to hold all the timestamp information expected. The size of the option does not change due to adding

timestamps. The initial contents of the timestamp data area must be zero or internet address/zero pairs.

If the timestamp data area is already full (the pointer exceeds the length) the datagram is forwarded without inserting the timestamp, but the overflow count is incremented by one.

If there is some room but not enough room for a full timestamp to be inserted, or the overflow count itself overflows, the original datagram is considered to be in error and is discarded. In either case an ICMP parameter problem message may be sent to the source host [3].

The timestamp option is not copied upon fragmentation. It is carried in the first fragment. Appears at most once in a datagram.

Padding: variable

The internet header padding is used to ensure that the internet header ends on a 32 bit boundary. The padding is zero.

3.2. Discussion

The implementation of a protocol must be robust. Each implementation must expect to interoperate with others created by different individuals. While the goal of this specification is to be explicit about the protocol there is the possibility of differing interpretations. In general, an implementation must be conservative in its sending behavior, and liberal in its receiving behavior. That is, it must be careful to send well-formed datagrams, but must accept any datagram that it can interpret (e.g., not object to technical errors where the meaning is still clear).

The basic internet service is datagram oriented and provides for the fragmentation of datagrams at gateways, with reassembly taking place at the destination internet protocol module in the destination host. Of course, fragmentation and reassembly of datagrams within a network or by private agreement between the gateways of a network is also allowed since this is transparent to the internet protocols and the higher-level protocols. This transparent type of fragmentation and reassembly is termed "network-dependent" (or intranet) fragmentation and is not discussed further here.

Internet addresses distinguish sources and destinations to the host level and provide a protocol field as well. It is assumed that each protocol will provide for whatever multiplexing is necessary within a host.

Addressing

To provide for flexibility in assigning address to networks and allow for the large number of small to intermediate sized networks the interpretation of the address field is coded to specify a small number of networks with a large number of host, a moderate number of networks with a moderate number of hosts, and a large number of networks with a small number of hosts. In addition there is an escape code for extended addressing mode.

Address Formats:

High Order Bits	Format	Class
-----	-----	-----
0	7 bits of net, 24 bits of host	a
10	14 bits of net, 16 bits of host	b
110	21 bits of net, 8 bits of host	c
111	escape to extended addressing mode	

A value of zero in the network field means this network. This is only used in certain ICMP messages. The extended addressing mode is undefined. Both of these features are reserved for future use.

The actual values assigned for network addresses is given in "Assigned Numbers" [9].

The local address, assigned by the local network, must allow for a single physical host to act as several distinct internet hosts. That is, there must be a mapping between internet host addresses and network/host interfaces that allows several internet addresses to correspond to one interface. It must also be allowed for a host to have several physical interfaces and to treat the datagrams from several of them as if they were all addressed to a single host.

Address mappings between internet addresses and addresses for ARPANET, SATNET, PRNET, and other networks are described in "Address Mappings" [5].

Fragmentation and Reassembly.

The internet identification field (ID) is used together with the source and destination address, and the protocol fields, to identify datagram fragments for reassembly.

The More Fragments flag bit (MF) is set if the datagram is not the last fragment. The Fragment Offset field identifies the fragment location, relative to the beginning of the original unfragmented datagram. Fragments are counted in units of 8 octets. The

fragmentation strategy is designed so that an unfragmented datagram has all zero fragmentation information (MF = 0, fragment offset = 0). If an internet datagram is fragmented, its data portion must be broken on 8 octet boundaries.

This format allows $2^{13} = 8192$ fragments of 8 octets each for a total of 65,536 octets. Note that this is consistent with the datagram total length field (of course, the header is counted in the total length and not in the fragments).

When fragmentation occurs, some options are copied, but others remain with the first fragment only.

Every internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an internet header may be up to 60 octets, and the minimum fragment is 8 octets.

Every internet destination must be able to receive a datagram of 576 octets either in one piece or in fragments to be reassembled.

The fields which may be affected by fragmentation include:

- (1) options field
- (2) more fragments flag
- (3) fragment offset
- (4) internet header length field
- (5) total length field
- (6) header checksum

If the Don't Fragment flag (DF) bit is set, then internet fragmentation of this datagram is NOT permitted, although it may be discarded. This can be used to prohibit fragmentation in cases where the receiving host does not have sufficient resources to reassemble internet fragments.

One example of use of the Don't Fragment feature is to download a small host. A small host could have a bootstrap program that accepts a datagram, stores it in memory, and then executes it.

The fragmentation and reassembly procedures are most easily described by examples. The following procedures are example implementations.

General notation in the following pseudo programs: " \leq " means "less than or equal", " \neq " means "not equal", " $=$ " means "equal", " \leftarrow " means "is set to". Also, " x to y " includes x and excludes y ; for example, "4 to 7" would include 4, 5, and 6 (but not 7).

Internet Protocol Specification

An Example Fragmentation Procedure

The maximum sized datagram that can be transmitted through the next network is called the maximum transmission unit (MTU).

If the total length is less than or equal the maximum transmission unit then submit this datagram to the next step in datagram processing; otherwise cut the datagram into two fragments, the first fragment being the maximum size, and the second fragment being the rest of the datagram. The first fragment is submitted to the next step in datagram processing, while the second fragment is submitted to this procedure in case it is still too large.

Notation:

FO	-	Fragment Offset
IHL	-	Internet Header Length
DF	-	Don't Fragment flag
MF	-	More Fragments flag
TL	-	Total Length
OFO	-	Old Fragment Offset
OIHL	-	Old Internet Header Length
OMF	-	Old More Fragments flag
OTL	-	Old Total Length
NFB	-	Number of Fragment Blocks
MTU	-	Maximum Transmission Unit

Procedure:

```

IF TL <= MTU THEN Submit this datagram to the next step
    in datagram processing ELSE IF DF = 1 THEN discard the
    datagram ELSE
To produce the first fragment:
(1) Copy the original internet header;
(2) OIHL <- IHL; OTL <- TL; OFO <- FO; OMF <- MF;
(3) NFB <- (MTU-IHL*4)/8;
(4) Attach the first NFB*8 data octets;
(5) Correct the header:
    MF <- 1; TL <- (IHL*4)+(NFB*8);
    Recompute Checksum;
(6) Submit this fragment to the next step in
    datagram processing;
To produce the second fragment:
(7) Selectively copy the internet header (some options
    are not copied, see option definitions);
(8) Append the remaining data;
(9) Correct the header:
    IHL <- ((OIHL*4)-(length of options not copied))+3)/4;
  
```



```
TL <- OTL - NFB*8 - (OIHL-IHL)*4);  
FO <- OFO + NFB; MF <- OMF; Recompute Checksum;  
(10) Submit this fragment to the fragmentation test; DONE.
```

In the above procedure each fragment (except the last) was made the maximum allowable size. An alternative might produce less than the maximum size datagrams. For example, one could implement a fragmentation procedure that repeatedly divided large datagrams in half until the resulting fragments were less than the maximum transmission unit size.

An Example Reassembly Procedure

For each datagram the buffer identifier is computed as the concatenation of the source, destination, protocol, and identification fields. If this is a whole datagram (that is both the fragment offset and the more fragments fields are zero), then any reassembly resources associated with this buffer identifier are released and the datagram is forwarded to the next step in datagram processing.

If no other fragment with this buffer identifier is on hand then reassembly resources are allocated. The reassembly resources consist of a data buffer, a header buffer, a fragment block bit table, a total data length field, and a timer. The data from the fragment is placed in the data buffer according to its fragment offset and length, and bits are set in the fragment block bit table corresponding to the fragment blocks received.

If this is the first fragment (that is the fragment offset is zero) this header is placed in the header buffer. If this is the last fragment (that is the more fragments field is zero) the total data length is computed. If this fragment completes the datagram (tested by checking the bits set in the fragment block table), then the datagram is sent to the next step in datagram processing; otherwise the timer is set to the maximum of the current timer value and the value of the time to live field from this fragment; and the reassembly routine gives up control.

If the timer runs out, the all reassembly resources for this buffer identifier are released. The initial setting of the timer is a lower bound on the reassembly waiting time. This is because the waiting time will be increased if the Time to Live in the arriving fragment is greater than the current timer value but will not be decreased if it is less. The maximum this timer value could reach is the maximum time to live (approximately 4.25 minutes). The current recommendation for the initial timer setting is 15 seconds. This may be changed as experience with

this protocol accumulates. Note that the choice of this parameter value is related to the buffer capacity available and the data rate of the transmission medium; that is, data rate times timer value equals buffer size (e.g., 10Kb/s X 15s = 150Kb).

Notation:

FO - Fragment Offset
IHL - Internet Header Length
MF - More Fragments flag
TTL - Time To Live
NFB - Number of Fragment Blocks
TL - Total Length
TDL - Total Data Length
BUFID - Buffer Identifier
RCVBT - Fragment Received Bit Table
TLB - Timer Lower Bound

Procedure:

```
(1)  BUFID <- source|destination|protocol|identification;
(2)  IF FO = 0 AND MF = 0
(3)    THEN IF buffer with BUFID is allocated
(4)      THEN flush all reassembly for this BUFID;
(5)      Submit datagram to next step; DONE.
(6)  ELSE IF no buffer with BUFID is allocated
(7)    THEN allocate reassembly resources
          with BUFID;
          TIMER <- TLB; TDL <- 0;
(8)    put data from fragment into data buffer with
          BUFID from octet FO*8 to
          octet (TL-(IHL*4))+FO*8;
(9)    set RCVBT bits from FO
          to FO+((TL-(IHL*4)+7)/8);
(10)   IF MF = 0 THEN TDL <- TL-(IHL*4)+(FO*8)
(11)   IF FO = 0 THEN put header in header buffer
(12)   IF TDL # 0
(13)     AND all RCVBT bits from 0
          to (TDL+7)/8 are set
(14)     THEN TL <- TDL+(IHL*4)
(15)     Submit datagram to next step;
(16)     free all reassembly resources
          for this BUFID; DONE.
(17)   TIMER <- MAX(TIMER,TTL);
(18)   give up until next fragment or timer expires;
(19) timer expires: flush all reassembly with this BUFID; DONE.
```

In the case that two or more fragments contain the same data

either identically or through a partial overlap, this procedure will use the more recently arrived copy in the data buffer and datagram delivered.

Identification

The choice of the Identifier for a datagram is based on the need to provide a way to uniquely identify the fragments of a particular datagram. The protocol module assembling fragments judges fragments to belong to the same datagram if they have the same source, destination, protocol, and Identifier. Thus, the sender must choose the Identifier to be unique for this source, destination pair and protocol for the time the datagram (or any fragment of it) could be alive in the internet.

It seems then that a sending protocol module needs to keep a table of Identifiers, one entry for each destination it has communicated with in the last maximum packet lifetime for the internet.

However, since the Identifier field allows 65,536 different values, some host may be able to simply use unique identifiers independent of destination.

It is appropriate for some higher level protocols to choose the identifier. For example, TCP protocol modules may retransmit an identical TCP segment, and the probability for correct reception would be enhanced if the retransmission carried the same identifier as the original transmission since fragments of either datagram could be used to construct a correct TCP segment.

Type of Service

The type of service (TOS) is for internet service quality selection. The type of service is specified along the abstract parameters precedence, delay, throughput, and reliability. These abstract parameters are to be mapped into the actual service parameters of the particular networks the datagram traverses.

Precedence. An independent measure of the importance of this datagram.

Delay. Prompt delivery is important for datagrams with this indication.

Throughput. High data rate is important for datagrams with this indication.

Internet Protocol
Specification

Reliability. A higher level of effort to ensure delivery is important for datagrams with this indication.

For example, the ARPANET has a priority bit, and a choice between "standard" messages (type 0) and "uncontrolled" messages (type 3), (the choice between single packet and multipacket messages can also be considered a service parameter). The uncontrolled messages tend to be less reliably delivered and suffer less delay. Suppose an internet datagram is to be sent through the ARPANET. Let the internet type of service be given as:

Precedence:	5
Delay:	0
Throughput:	1
Reliability:	1

In this example, the mapping of these parameters to those available for the ARPANET would be to set the ARPANET priority bit on since the Internet precedence is in the upper half of its range, to select standard messages since the throughput and reliability requirements are indicated and delay is not. More details are given on service mappings in "Service Mappings" [8].

Time to Live

The time to live is set by the sender to the maximum time the datagram is allowed to be in the internet system. If the datagram is in the internet system longer than the time to live, then the datagram must be destroyed.

This field must be decreased at each point that the internet header is processed to reflect the time spent processing the datagram. Even if no local information is available on the time actually spent, the field must be decremented by 1. The time is measured in units of seconds (i.e. the value 1 means one second). Thus, the maximum time to live is 255 seconds or 4.25 minutes. Since every module that processes a datagram must decrease the TTL by at least one even if it process the datagram in less than a second, the TTL must be thought of only as an upper bound on the time a datagram may exist. The intention is to cause undeliverable datagrams to be discarded, and to bound the maximum datagram lifetime.

Some higher level reliable connection protocols are based on assumptions that old duplicate datagrams will not arrive after a certain time elapses. The TTL is a way for such protocols to have an assurance that their assumption is met.

Options

The options are optional in each datagram, but required in implementations. That is, the presence or absence of an option is the choice of the sender, but each internet module must be able to parse every option. There can be several options present in the option field.

The options might not end on a 32-bit boundary. The internet header must be filled out with octets of zeros. The first of these would be interpreted as the end-of-options option, and the remainder as internet header padding.

Every internet module must be able to act on every option. The Security Option is required if classified, restricted, or compartmented traffic is to be passed.

Checksum

The internet header checksum is recomputed if the internet header is changed. For example, a reduction of the time to live, additions or changes to internet options, or due to fragmentation. This checksum at the internet level is intended to protect the internet header fields from transmission errors.

There are some applications where a few data bit errors are acceptable while retransmission delays are not. If the internet protocol enforced data correctness such applications could not be supported.

Errors

Internet protocol errors may be reported via the ICMP messages [3].

3.3. Interfaces

The functional description of user interfaces to the IP is, at best, fictional, since every operating system will have different facilities. Consequently, we must warn readers that different IP implementations may have different user interfaces. However, all IPs must provide a certain minimum set of services to guarantee that all IP implementations can support the same protocol hierarchy. This section specifies the functional interfaces required of all IP implementations.

Internet protocol interfaces on one side to the local network and on the other side to either a higher level protocol or an application program. In the following, the higher level protocol or application

Internet Protocol Specification

program (or even a gateway program) will be called the "user" since it is using the internet module. Since internet protocol is a datagram protocol, there is minimal memory or state maintained between datagram transmissions, and each call on the internet protocol module by the user supplies all information necessary for the IP to perform the service requested.

An Example Upper Level Interface

The following two example calls satisfy the requirements for the user to internet protocol module communication ("=>" means returns):

```
SEND (src, dst, prot, TOS, TTL, BufPTR, len, Id, DF, opt => result)
```

where:

```
src = source address
dst = destination address
prot = protocol
TOS = type of service
TTL = time to live
BufPTR = buffer pointer
len = length of buffer
Id = Identifier
DF = Don't Fragment
opt = option data
result = response
    OK = datagram sent ok
    Error = error in arguments or local network error
```

Note that the precedence is included in the TOS and the security/compartiment is passed as an option.

```
RECV (BufPTR, prot, => result, src, dst, TOS, len, opt)
```

where:

```
BufPTR = buffer pointer
prot = protocol
result = response
    OK = datagram received ok
    Error = error in arguments
len = length of buffer
src = source address
dst = destination address
TOS = type of service
opt = option data
```

When the user sends a datagram, it executes the SEND call supplying all the arguments. The internet protocol module, on receiving this call, checks the arguments and prepares and sends the message. If the arguments are good and the datagram is accepted by the local network, the call returns successfully. If either the arguments are bad, or the datagram is not accepted by the local network, the call returns unsuccessfully. On unsuccessful returns, a reasonable report must be made as to the cause of the problem, but the details of such reports are up to individual implementations.

When a datagram arrives at the internet protocol module from the local network, either there is a pending RECV call from the user addressed or there is not. In the first case, the pending call is satisfied by passing the information from the datagram to the user. In the second case, the user addressed is notified of a pending datagram. If the user addressed does not exist, an ICMP error message is returned to the sender, and the data is discarded.

The notification of a user may be via a pseudo interrupt or similar mechanism, as appropriate in the particular operating system environment of the implementation.

A user's RECV call may then either be immediately satisfied by a pending datagram, or the call may be pending until a datagram arrives.

The source address is included in the send call in case the sending host has several addresses (multiple physical connections or logical addresses). The internet module must check to see that the source address is one of the legal address for this host.

An implementation may also allow or require a call to the internet module to indicate interest in or reserve exclusive use of a class of datagrams (e.g., all those with a certain value in the protocol field).

This section functionally characterizes a USER/IP interface. The notation used is similar to most procedure of function calls in high level languages, but this usage is not meant to rule out trap type service calls (e.g., SVCs, UUOs, EMTs), or any other form of interprocess communication.

APPENDIX A: Examples & Scenarios

Example 1:

This is an example of the minimal data carrying internet datagram:

```

      0              1              2              3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|Ver= 4 |IHL= 5 |Type of Service|           Total Length = 21      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Identification = 111       |Flg=0|   Fragment Offset = 0   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|   Time = 123   |   Protocol = 1   |           header checksum           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     source address                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     destination address                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           data           |
+-----+-----+-----+-----+

```

Example Internet Datagram

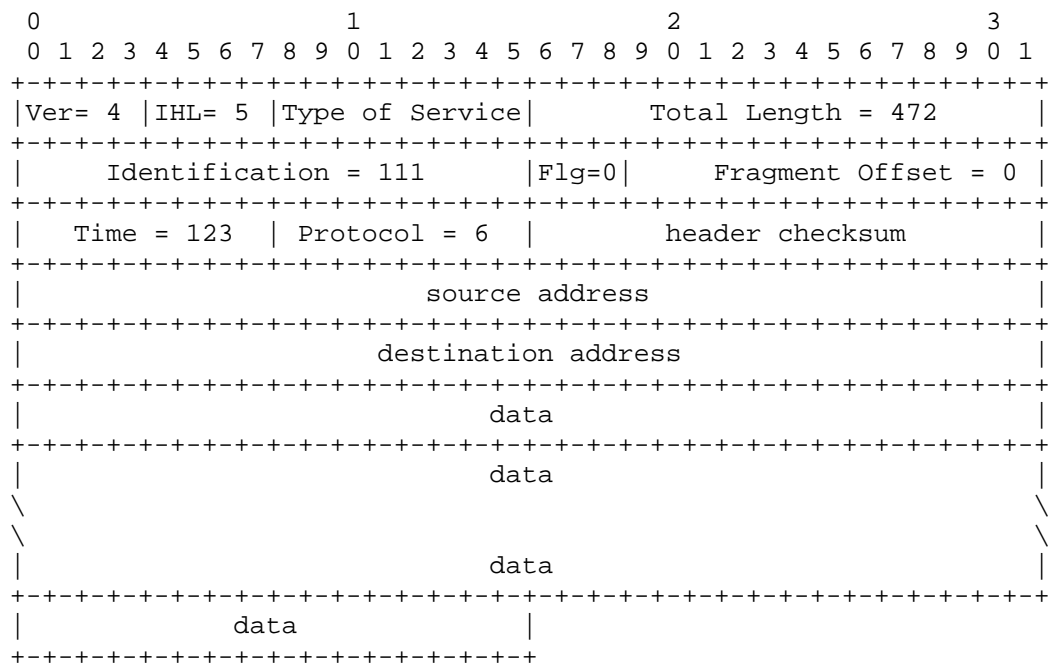
Figure 5.

Note that each tick mark represents one bit position.

This is a internet datagram in version 4 of internet protocol; the internet header consists of five 32 bit words, and the total length of the datagram is 21 octets. This datagram is a complete datagram (not a fragment).

Example 2:

In this example, we show first a moderate size internet datagram (452 data octets), then two internet fragments that might result from the fragmentation of this datagram if the maximum sized transmission allowed were 280 octets.

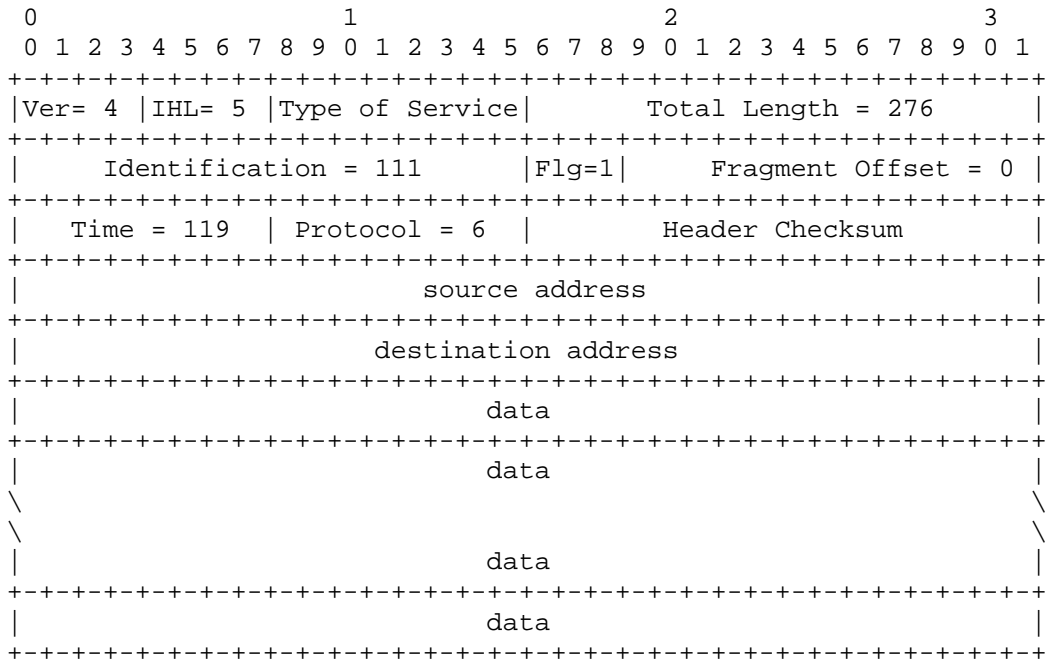


Example Internet Datagram

Figure 6.

Internet Protocol

Now the first fragment that results from splitting the datagram after 256 data octets.



Example Internet Fragment

Figure 7.

And the second fragment.

```

      0                               1                               2                               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|Ver= 4 |IHL= 5 |Type of Service|          Total Length = 216          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Identification = 111      |Flg=0|  Fragment Offset  =  32  |
+-----+-----+-----+-----+-----+-----+-----+-----+
|    Time = 119    | Protocol = 6  |          Header Checksum          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     source address                    |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     destination address                |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     data                                |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     data                                |
|                                     \                                  |
|                                     \                                  |
|                                     data                                |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     data                                |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Example Internet Fragment

Figure 8.

Internet Protocol

Example 3:

Here, we show an example of a datagram containing options:

```

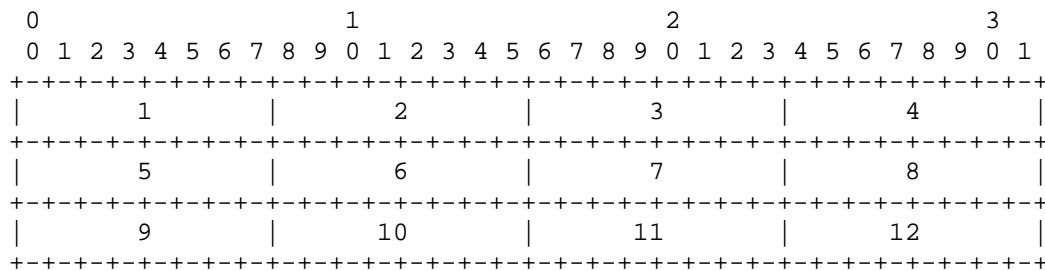
      0              1              2              3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
| Ver= 4 | IHL= 8 | Type of Service |           Total Length = 576 |
+-----+-----+-----+-----+-----+-----+-----+-----+
|           Identification = 111       | Flg=0 |   Fragment Offset = 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
|   Time = 123   | Protocol = 6 |           Header Checksum           |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     source address                    |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     destination address                |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Opt. Code = x | Opt. Len.= 3 | option value | Opt. Code = x |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Opt. Len. = 4 |           option value           | Opt. Code = 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+
| Opt. Code = y | Opt. Len. = 3 | option value | Opt. Code = 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     data                                |
\                                     \
\                                     \
|                                     data                                |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     data                                |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

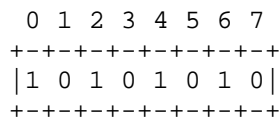
Example Internet Datagram

Figure 9.

The order of transmission of the header and data described in this document is resolved to the octet level. Whenever a diagram shows a group of octets, the order of transmission of those octets is the normal order in which they are read in English. For example, in the following diagram the octets are transmitted in the order they are numbered.



Whenever an octet represents a numeric quantity the left most bit in the diagram is the high order or most significant bit. That is, the bit labeled 0 is the most significant bit. For example, the following diagram represents the value 170 (decimal).



Similarly, whenever a multi-octet field represents a numeric quantity the left most bit of the whole field is the most significant bit. When a multi-octet quantity is transmitted the most significant octet is transmitted first.

GLOSSARY

1822

BBN Report 1822, "The Specification of the Interconnection of a Host and an IMP". The specification of interface between a host and the ARPANET.

ARPANET leader

The control information on an ARPANET message at the host-IMP interface.

ARPANET message

The unit of transmission between a host and an IMP in the ARPANET. The maximum size is about 1012 octets (8096 bits).

ARPANET packet

A unit of transmission used internally in the ARPANET between IMPs. The maximum size is about 126 octets (1008 bits).

Destination

The destination address, an internet header field.

DF

The Don't Fragment bit carried in the flags field.

Flags

An internet header field carrying various control flags.

Fragment Offset

This internet header field indicates where in the internet datagram a fragment belongs.

GGP

Gateway to Gateway Protocol, the protocol used primarily between gateways to control routing and other gateway functions.

header

Control information at the beginning of a message, segment, datagram, packet or block of data.

ICMP

Internet Control Message Protocol, implemented in the internet module, the ICMP is used from gateways to hosts and between hosts to report errors and make routing suggestions.

Internet Protocol
Glossary

Identification

An internet header field carrying the identifying value assigned by the sender to aid in assembling the fragments of a datagram.

IHL

The internet header field Internet Header Length is the length of the internet header measured in 32 bit words.

IMP

The Interface Message Processor, the packet switch of the ARPANET.

Internet Address

A four octet (32 bit) source or destination address consisting of a Network field and a Local Address field.

internet datagram

The unit of data exchanged between a pair of internet modules (includes the internet header).

internet fragment

A portion of the data of an internet datagram with an internet header.

Local Address

The address of a host within a network. The actual mapping of an internet local address on to the host addresses in a network is quite general, allowing for many to one mappings.

MF

The More-Fragments Flag carried in the internet header flags field.

module

An implementation, usually in software, of a protocol or other procedure.

more-fragments flag

A flag indicating whether or not this internet datagram contains the end of an internet datagram, carried in the internet header Flags field.

NFB

The Number of Fragment Blocks in a the data portion of an internet fragment. That is, the length of a portion of data measured in 8 octet units.

octet	An eight bit byte.
Options	The internet header Options field may contain several options, and each option may be several octets in length.
Padding	The internet header Padding field is used to ensure that the data begins on 32 bit word boundary. The padding is zero.
Protocol	In this document, the next higher level protocol identifier, an internet header field.
Rest	The local address portion of an Internet Address.
Source	The source address, an internet header field.
TCP	Transmission Control Protocol: A host-to-host protocol for reliable communication in internet environments.
TCP Segment	The unit of data exchanged between TCP modules (including the TCP header).
TFTP	Trivial File Transfer Protocol: A simple file transfer protocol built on UDP.
Time to Live	An internet header field which indicates the upper bound on how long this internet datagram may exist.
TOS	Type of Service
Total Length	The internet header field Total Length is the length of the datagram in octets including internet header and data.
TTL	Time to Live

Internet Protocol
Glossary

Type of Service

An internet header field which indicates the type (or quality) of service for this internet datagram.

UDP

User Datagram Protocol: A user level protocol for transaction oriented applications.

User

The user of the internet protocol. This may be a higher level protocol module, an application program, or a gateway program.

Version

The Version field indicates the format of the internet header.

REFERENCES

- [1] Cerf, V., "The Catenet Model for Internetworking," Information Processing Techniques Office, Defense Advanced Research Projects Agency, IEN 48, July 1978.
- [2] Bolt Beranek and Newman, "Specification for the Interconnection of a Host and an IMP," BBN Technical Report 1822, Revised May 1978.
- [3] Postel, J., "Internet Control Message Protocol - DARPA Internet Program Protocol Specification," RFC 792, USC/Information Sciences Institute, September 1981.
- [4] Shoch, J., "Inter-Network Naming, Addressing, and Routing," COMPCON, IEEE Computer Society, Fall 1978.
- [5] Postel, J., "Address Mappings," RFC 796, USC/Information Sciences Institute, September 1981.
- [6] Shoch, J., "Packet Fragmentation in Inter-Network Protocols," Computer Networks, v. 3, n. 1, February 1979.
- [7] Strazisar, V., "How to Build a Gateway", IEN 109, Bolt Beranek and Newman, August 1979.
- [8] Postel, J., "Service Mappings," RFC 795, USC/Information Sciences Institute, September 1981.
- [9] Postel, J., "Assigned Numbers," RFC 790, USC/Information Sciences Institute, September 1981.

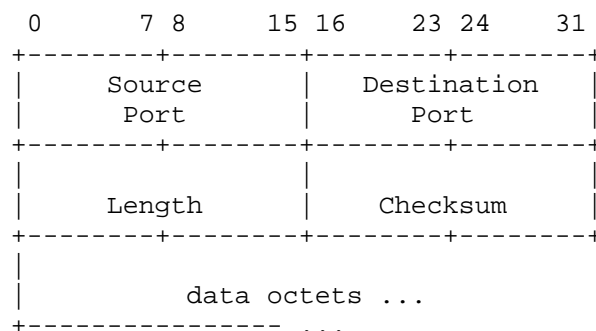
User Datagram Protocol

Introduction

This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

Format



User Datagram Header Format

Fields

Source Port is an optional field, when meaningful, it indicates the port of the sending process, and may be assumed to be the port to which a reply should be addressed in the absence of any other information. If not used, a value of zero is inserted.

Destination Port has a meaning within the context of a particular internet destination address.

Length is the length in octets of this user datagram including this header and the data. (This means the minimum value of the length is eight.)

Checksum is the 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.

The pseudo header conceptually prefixed to the UDP header contains the source address, the destination address, the protocol, and the UDP length. This information gives protection against misrouted datagrams. This checksum procedure is the same as is used in TCP.

0	7 8	15 16	23 24	31
+-----+-----+-----+-----+				
source address				
+-----+-----+-----+-----+				
destination address				
+-----+-----+-----+-----+				
zero protocol UDP length				
+-----+-----+-----+-----+				

If the computed checksum is zero, it is transmitted as all ones (the equivalent in one's complement arithmetic). An all zero transmitted checksum value means that the transmitter generated no checksum (for debugging or for higher level protocols that don't care).

User Interface

A user interface should allow

the creation of new receive ports,

receive operations on the receive ports that return the data octets and an indication of source port and source address,

and an operation that allows a datagram to be sent, specifying the data, source and destination ports and addresses to be sent.

IP Interface -----

The UDP module must be able to determine the source and destination internet addresses and the protocol field from the internet header. One possible UDP/IP interface would return the whole internet datagram including all of the internet header in response to a receive operation. Such an interface would also allow the UDP to pass a full internet datagram complete with header to the IP to send. The IP would verify certain fields for consistency and compute the internet header checksum.

Protocol Application -----

The major uses of this protocol is the Internet Name Server [3], and the Trivial File Transfer [4].

Protocol Number -----

This is protocol 17 (21 octal) when used in the Internet Protocol. Other protocol numbers are listed in [5].

References -----

- [1] Postel, J., "Internet Protocol," RFC 760, USC/Information Sciences Institute, January 1980.
- [2] Postel, J., "Transmission Control Protocol," RFC 761, USC/Information Sciences Institute, January 1980.
- [3] Postel, J., "Internet Name Server," USC/Information Sciences Institute, IEN 116, August 1979.
- [4] Sollins, K., "The TFTP Protocol," Massachusetts Institute of Technology, IEN 133, January 1980.
- [5] Postel, J., "Assigned Numbers," USC/Information Sciences Institute, RFC 762, January 1980.

Network Working Group
Request for Comments: 1323
Obsoletes: RFC 1072, RFC 1185

V. Jacobson
LBL
R. Braden
ISI
D. Borman
Cray Research
May 1992

TCP Extensions for High Performance

Status of This Memo

This RFC specifies an IAB standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "IAB Official Protocol Standards" for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

This memo presents a set of TCP extensions to improve performance over large bandwidth*delay product paths and to provide reliable operation over very high-speed paths. It defines new TCP options for scaled windows and timestamps, which are designed to provide compatible interworking with TCP's that do not implement the extensions. The timestamps are used for two distinct mechanisms: RTTM (Round Trip Time Measurement) and PAWS (Protect Against Wrapped Sequences). Selective acknowledgments are not included in this memo.

This memo combines and supersedes RFC-1072 and RFC-1185, adding additional clarification and more detailed specification. Appendix C summarizes the changes from the earlier RFCs.

TABLE OF CONTENTS

1. Introduction	2
2. TCP Window Scale Option	8
3. RTTM -- Round-Trip Time Measurement	11
4. PAWS -- Protect Against Wrapped Sequence Numbers	17
5. Conclusions and Acknowledgments	25
6. References	25
APPENDIX A: Implementation Suggestions	27
APPENDIX B: Duplicates from Earlier Connection Incarnations	27
APPENDIX C: Changes from RFC-1072, RFC-1185	30
APPENDIX D: Summary of Notation	31
APPENDIX E: Event Processing	32
Security Considerations	37

Authors' Addresses 37

1. INTRODUCTION

The TCP protocol [Postel81] was designed to operate reliably over almost any transmission medium regardless of transmission rate, delay, corruption, duplication, or reordering of segments. Production TCP implementations currently adapt to transfer rates in the range of 100 bps to 10^7 bps and round-trip delays in the range 1 ms to 100 seconds. Recent work on TCP performance has shown that TCP can work well over a variety of Internet paths, ranging from 800 Mbit/sec I/O channels to 300 bit/sec dial-up modems [Jacobson88a].

The introduction of fiber optics is resulting in ever-higher transmission speeds, and the fastest paths are moving out of the domain for which TCP was originally engineered. This memo defines a set of modest extensions to TCP to extend the domain of its application to match this increasing network capability. It is based upon and obsoletes RFC-1072 [Jacobson88b] and RFC-1185 [Jacobson90b].

There is no one-line answer to the question: "How fast can TCP go?". There are two separate kinds of issues, performance and reliability, and each depends upon different parameters. We discuss each in turn.

1.1 TCP Performance

TCP performance depends not upon the transfer rate itself, but rather upon the product of the transfer rate and the round-trip delay. This "bandwidth*delay product" measures the amount of data that would "fill the pipe"; it is the buffer space required at sender and receiver to obtain maximum throughput on the TCP connection over the path, i.e., the amount of unacknowledged data that TCP must handle in order to keep the pipeline full. TCP performance problems arise when the bandwidth*delay product is large. We refer to an Internet path operating in this region as a "long, fat pipe", and a network containing this path as an "LFN" (pronounced "elephan(t)").

High-capacity packet satellite channels (e.g., DARPA's Wideband Net) are LFN's. For example, a DS1-speed satellite channel has a bandwidth*delay product of 10^6 bits or more; this corresponds to 100 outstanding TCP segments of 1200 bytes each. Terrestrial fiber-optical paths will also fall into the LFN class; for example, a cross-country delay of 30 ms at a DS3 bandwidth (45Mbps) also exceeds 10^6 bits.

There are three fundamental performance problems with the current TCP over LFN paths:

(1) Window Size Limit

The TCP header uses a 16 bit field to report the receive window size to the sender. Therefore, the largest window that can be used is $2^{16} = 65K$ bytes.

To circumvent this problem, Section 2 of this memo defines a new TCP option, "Window Scale", to allow windows larger than 2^{16} . This option defines an implicit scale factor, which is used to multiply the window size value found in a TCP header to obtain the true window size.

(2) Recovery from Losses

Packet losses in an LFN can have a catastrophic effect on throughput. Until recently, properly-operating TCP implementations would cause the data pipeline to drain with every packet loss, and require a slow-start action to recover. Recently, the Fast Retransmit and Fast Recovery algorithms [Jacobson90c] have been introduced. Their combined effect is to recover from one packet loss per window, without draining the pipeline. However, more than one packet loss per window typically results in a retransmission timeout and the resulting pipeline drain and slow start.

Expanding the window size to match the capacity of an LFN results in a corresponding increase of the probability of more than one packet per window being dropped. This could have a devastating effect upon the throughput of TCP over an LFN. In addition, if a congestion control mechanism based upon some form of random dropping were introduced into gateways, randomly spaced packet drops would become common, possibly increasing the probability of dropping more than one packet per window.

To generalize the Fast Retransmit/Fast Recovery mechanism to handle multiple packets dropped per window, selective acknowledgments are required. Unlike the normal cumulative acknowledgments of TCP, selective acknowledgments give the sender a complete picture of which segments are queued at the receiver and which have not yet arrived. Some evidence in favor of selective acknowledgments has been published [NBS85], and selective acknowledgments have been included in a number of experimental Internet protocols -- VMTP [Cheriton88], NETBLT [Clark87], and RDP [Velten84], and proposed for OSI TP4 [NBS85]. However, in the non-LFN regime, selective acknowledgments reduce the number of

packets retransmitted but do not otherwise improve performance, making their complexity of questionable value. However, selective acknowledgments are expected to become much more important in the LFN regime.

RFC-1072 defined a new TCP "SACK" option to send a selective acknowledgment. However, there are important technical issues to be worked out concerning both the format and semantics of the SACK option. Therefore, SACK has been omitted from this package of extensions. It is hoped that SACK can "catch up" during the standardization process.

(3) Round-Trip Measurement

TCP implements reliable data delivery by retransmitting segments that are not acknowledged within some retransmission timeout (RTO) interval. Accurate dynamic determination of an appropriate RTO is essential to TCP performance. RTO is determined by estimating the mean and variance of the measured round-trip time (RTT), i.e., the time interval between sending a segment and receiving an acknowledgment for it [Jacobson88a].

Section 4 introduces a new TCP option, "Timestamps", and then defines a mechanism using this option that allows nearly every segment, including retransmissions, to be timed at negligible computational cost. We use the mnemonic RTTM (Round Trip Time Measurement) for this mechanism, to distinguish it from other uses of the Timestamps option.

1.2 TCP Reliability

Now we turn from performance to reliability. High transfer rate enters TCP performance through the bandwidth*delay product. However, high transfer rate alone can threaten TCP reliability by violating the assumptions behind the TCP mechanism for duplicate detection and sequencing.

An especially serious kind of error may result from an accidental reuse of TCP sequence numbers in data segments. Suppose that an "old duplicate segment", e.g., a duplicate data segment that was delayed in Internet queues, is delivered to the receiver at the wrong moment, so that its sequence numbers falls somewhere within the current window. There would be no checksum failure to warn of the error, and the result could be an undetected corruption of the data. Reception of an old duplicate ACK segment at the transmitter could be only slightly less serious: it is likely to

lock up the connection so that no further progress can be made, forcing an RST on the connection.

TCP reliability depends upon the existence of a bound on the lifetime of a segment: the "Maximum Segment Lifetime" or MSL. An MSL is generally required by any reliable transport protocol, since every sequence number field must be finite, and therefore any sequence number may eventually be reused. In the Internet protocol suite, the MSL bound is enforced by an IP-layer mechanism, the "Time-to-Live" or TTL field.

Duplication of sequence numbers might happen in either of two ways:

- (1) Sequence number wrap-around on the current connection

A TCP sequence number contains 32 bits. At a high enough transfer rate, the 32-bit sequence space may be "wrapped" (cycled) within the time that a segment is delayed in queues.

- (2) Earlier incarnation of the connection

Suppose that a connection terminates, either by a proper close sequence or due to a host crash, and the same connection (i.e., using the same pair of sockets) is immediately reopened. A delayed segment from the terminated connection could fall within the current window for the new incarnation and be accepted as valid.

Duplicates from earlier incarnations, Case (2), are avoided by enforcing the current fixed MSL of the TCP spec, as explained in Section 5.3 and Appendix B. However, case (1), avoiding the reuse of sequence numbers within the same connection, requires an MSL bound that depends upon the transfer rate, and at high enough rates, a new mechanism is required.

More specifically, if the maximum effective bandwidth at which TCP is able to transmit over a particular path is B bytes per second, then the following constraint must be satisfied for error-free operation:

$$2^{31} / B > \text{MSL (secs)} \quad [1]$$

The following table shows the value for $T_{\text{wrap}} = 2^{31}/B$ in seconds, for some important values of the bandwidth B:

Network	B*8 bits/sec	B bytes/sec	Twrap secs
ARPANET	56kbps	7KBps	$3 \cdot 10^{**5}$ (~3.6 days)
DS1	1.5Mbps	190KBps	10^{**4} (~3 hours)
Ethernet	10Mbps	1.25MBps	1700 (~30 mins)
DS3	45Mbps	5.6MBps	380
FDDI	100Mbps	12.5MBps	170
Gigabit	1Gbps	125MBps	17

It is clear that wrap-around of the sequence space is not a problem for 56kbps packet switching or even 10Mbps Ethernets. On the other hand, at DS3 and FDDI speeds, Twrap is comparable to the 2 minute MSL assumed by the TCP specification [Postel81]. Moving towards gigabit speeds, Twrap becomes too small for reliable enforcement by the Internet TTL mechanism.

The 16-bit window field of TCP limits the effective bandwidth B to $2^{**16}/RTT$, where RTT is the round-trip time in seconds [McKenzie89]. If the RTT is large enough, this limits B to a value that meets the constraint [1] for a large MSL value. For example, consider a transcontinental backbone with an RTT of 60ms (set by the laws of physics). With the bandwidth*delay product limited to 64KB by the TCP window size, B is then limited to 1.1MBps, no matter how high the theoretical transfer rate of the path. This corresponds to cycling the sequence number space in $Twrap = 2000$ secs, which is safe in today's Internet.

It is important to understand that the culprit is not the larger window but rather the high bandwidth. For example, consider a (very large) FDDI LAN with a diameter of 10km. Using the speed of light, we can compute the RTT across the ring as $(2 \cdot 10^{**4}) / (3 \cdot 10^{**8}) = 67$ microseconds, and the delay*bandwidth product is then 833 bytes. A TCP connection across this LAN using a window of only 833 bytes will run at the full 100mbps and can wrap the sequence space in about 3 minutes, very close to the MSL of TCP. Thus, high speed alone can cause a reliability problem with sequence number wrap-around, even without extended windows.

Watson's Delta-T protocol [Watson81] includes network-layer mechanisms for precise enforcement of an MSL. In contrast, the IP

mechanism for MSL enforcement is loosely defined and even more loosely implemented in the Internet. Therefore, it is unwise to depend upon active enforcement of MSL for TCP connections, and it is unrealistic to imagine setting MSL's smaller than the current values (e.g., 120 seconds specified for TCP).

A possible fix for the problem of cycling the sequence space would be to increase the size of the TCP sequence number field. For example, the sequence number field (and also the acknowledgment field) could be expanded to 64 bits. This could be done either by changing the TCP header or by means of an additional option.

Section 5 presents a different mechanism, which we call PAWS (Protect Against Wrapped Sequence numbers), to extend TCP reliability to transfer rates well beyond the foreseeable upper limit of network bandwidths. PAWS uses the TCP Timestamps option defined in Section 4 to protect against old duplicates from the same connection.

1.3 Using TCP options

The extensions defined in this memo all use new TCP options. We must address two possible issues concerning the use of TCP options: (1) compatibility and (2) overhead.

We must pay careful attention to compatibility, i.e., to interoperation with existing implementations. The only TCP option defined previously, MSS, may appear only on a SYN segment. Every implementation should (and we expect that most will) ignore unknown options on SYN segments. However, some buggy TCP implementation might be crashed by the first appearance of an option on a non-SYN segment. Therefore, for each of the extensions defined below, TCP options will be sent on non-SYN segments only when an exchange of options on the SYN segments has indicated that both sides understand the extension. Furthermore, an extension option will be sent in a <SYN,ACK> segment only if the corresponding option was received in the initial <SYN> segment.

A question may be raised about the bandwidth and processing overhead for TCP options. Those options that occur on SYN segments are not likely to cause a performance concern. Opening a TCP connection requires execution of significant special-case code, and the processing of options is unlikely to increase that cost significantly.

On the other hand, a Timestamps option may appear in any data or ACK segment, adding 12 bytes to the 20-byte TCP header. We

believe that the bandwidth saved by reducing unnecessary retransmissions will more than pay for the extra header bandwidth.

There is also an issue about the processing overhead for parsing the variable byte-aligned format of options, particularly with a RISC-architecture CPU. To meet this concern, Appendix A contains a recommended layout of the options in TCP headers to achieve reasonable data field alignment. In the spirit of Header Prediction, a TCP can quickly test for this layout and if it is verified then use a fast path. Hosts that use this canonical layout will effectively use the options as a set of fixed-format fields appended to the TCP header. However, to retain the philosophical and protocol framework of TCP options, a TCP must be prepared to parse an arbitrary options field, albeit with less efficiency.

Finally, we observe that most of the mechanisms defined in this memo are important for LFN's and/or very high-speed networks. For low-speed networks, it might be a performance optimization to NOT use these mechanisms. A TCP vendor concerned about optimal performance over low-speed paths might consider turning these extensions off for low-speed paths, or allow a user or installation manager to disable them.

2. TCP WINDOW SCALE OPTION

2.1 Introduction

The window scale extension expands the definition of the TCP window to 32 bits and then uses a scale factor to carry this 32-bit value in the 16-bit Window field of the TCP header (SEG.WND in RFC-793). The scale factor is carried in a new TCP option, Window Scale. This option is sent only in a SYN segment (a segment with the SYN bit on), hence the window scale is fixed in each direction when a connection is opened. (Another design choice would be to specify the window scale in every TCP segment. It would be incorrect to send a window scale option only when the scale factor changed, since a TCP option in an acknowledgement segment will not be delivered reliably (unless the ACK happens to be piggy-backed on data in the other direction). Fixing the scale when the connection is opened has the advantage of lower overhead but the disadvantage that the scale factor cannot be changed during the connection.)

The maximum receive window, and therefore the scale factor, is determined by the maximum receive buffer space. In a typical modern implementation, this maximum buffer space is set by default

but can be overridden by a user program before a TCP connection is opened. This determines the scale factor, and therefore no new user interface is needed for window scaling.

2.2 Window Scale Option

The three-byte Window Scale option may be sent in a SYN segment by a TCP. It has two purposes: (1) indicate that the TCP is prepared to do both send and receive window scaling, and (2) communicate a scale factor to be applied to its receive window. Thus, a TCP that is prepared to scale windows should send the option, even if its own scale factor is 1. The scale factor is limited to a power of two and encoded logarithmically, so it may be implemented by binary shift operations.

TCP Window Scale Option (WSopt):

Kind: 3 Length: 3 bytes

```
+-----+-----+-----+
| Kind=3 |Length=3 |shift.cnt|
+-----+-----+-----+
```

This option is an offer, not a promise; both sides must send Window Scale options in their SYN segments to enable window scaling in either direction. If window scaling is enabled, then the TCP that sent this option will right-shift its true receive-window values by 'shift.cnt' bits for transmission in SEG.WND. The value 'shift.cnt' may be zero (offering to scale, while applying a scale factor of 1 to the receive window).

This option may be sent in an initial <SYN> segment (i.e., a segment with the SYN bit on and the ACK bit off). It may also be sent in a <SYN,ACK> segment, but only if a Window Scale option was received in the initial <SYN> segment. A Window Scale option in a segment without a SYN bit should be ignored.

The Window field in a SYN (i.e., a <SYN> or <SYN,ACK>) segment itself is never scaled.

2.3 Using the Window Scale Option

A model implementation of window scaling is as follows, using the notation of RFC-793 [Postel81]:

- * All windows are treated as 32-bit quantities for storage in

the connection control block and for local calculations. This includes the send-window (SND.WND) and the receive-window (RCV.WND) values, as well as the congestion window.

- * The connection state is augmented by two window shift counts, Snd.Wind.Scale and Rcv.Wind.Scale, to be applied to the incoming and outgoing window fields, respectively.
- * If a TCP receives a <SYN> segment containing a Window Scale option, it sends its own Window Scale option in the <SYN,ACK> segment.
- * The Window Scale option is sent with shift.cnt = R, where R is the value that the TCP would like to use for its receive window.
- * Upon receiving a SYN segment with a Window Scale option containing shift.cnt = S, a TCP sets Snd.Wind.Scale to S and sets Rcv.Wind.Scale to R; otherwise, it sets both Snd.Wind.Scale and Rcv.Wind.Scale to zero.
- * The window field (SEG.WND) in the header of every incoming segment, with the exception of SYN segments, is left-shifted by Snd.Wind.Scale bits before updating SND.WND:

SND.WND = SEG.WND << Snd.Wind.Scale

(assuming the other conditions of RFC793 are met, and using the "C" notation "<<" for left-shift).

- * The window field (SEG.WND) of every outgoing segment, with the exception of SYN segments, is right-shifted by Rcv.Wind.Scale bits:

SEG.WND = RCV.WND >> Rcv.Wind.Scale.

TCP determines if a data segment is "old" or "new" by testing whether its sequence number is within 2^{31} bytes of the left edge of the window, and if it is not, discarding the data as "old". To insure that new data is never mistakenly considered old and vice-versa, the left edge of the sender's window has to be at most 2^{31} away from the right edge of the receiver's window. Similarly with the sender's right edge and receiver's left edge. Since the right and left edges of either the sender's or receiver's window differ by the window size, and since the sender and receiver windows can be out of phase by at most the window size, the above constraints imply that $2 \times$ the max window size

must be less than 2^{31} , or

$\text{max window} < 2^{30}$

Since the max window is 2^S (where S is the scaling shift count) times at most $2^{16} - 1$ (the maximum unscaled window), the maximum window is guaranteed to be $< 2^{30}$ if $S \leq 14$. Thus, the shift count must be limited to 14 (which allows windows of $2^{30} = 1$ Gbyte). If a Window Scale option is received with a `shift.cnt` value exceeding 14, the TCP should log the error but use 14 instead of the specified value.

The scale factor applies only to the Window field as transmitted in the TCP header; each TCP using extended windows will maintain the window values locally as 32-bit numbers. For example, the "congestion window" computed by Slow Start and Congestion Avoidance is not affected by the scale factor, so window scaling will not introduce quantization into the congestion window.

3. RTTM: ROUND-TRIP TIME MEASUREMENT

3.1 Introduction

Accurate and current RTT estimates are necessary to adapt to changing traffic conditions and to avoid an instability known as "congestion collapse" [Nagle84] in a busy network. However, accurate measurement of RTT may be difficult both in theory and in implementation.

Many TCP implementations base their RTT measurements upon a sample of only one packet per window. While this yields an adequate approximation to the RTT for small windows, it results in an unacceptably poor RTT estimate for an LFN. If we look at RTT estimation as a signal processing problem (which it is), a data signal at some frequency, the packet rate, is being sampled at a lower frequency, the window rate. This lower sampling frequency violates Nyquist's criteria and may therefore introduce "aliasing" artifacts into the estimated RTT [Hamming77].

A good RTT estimator with a conservative retransmission timeout calculation can tolerate aliasing when the sampling frequency is "close" to the data frequency. For example, with a window of 8 packets, the sample rate is $1/8$ the data frequency -- less than an order of magnitude different. However, when the window is tens or hundreds of packets, the RTT estimator may be seriously in error, resulting in spurious retransmissions.

If there are dropped packets, the problem becomes worse. Zhang

[Zhang86], Jain [Jain86] and Karn [Karn87] have shown that it is not possible to accumulate reliable RTT estimates if retransmitted segments are included in the estimate. Since a full window of data will have been transmitted prior to a retransmission, all of the segments in that window will have to be ACKed before the next RTT sample can be taken. This means at least an additional window's worth of time between RTT measurements and, as the error rate approaches one per window of data (e.g., 10^{-6} errors per bit for the Wideband satellite network), it becomes effectively impossible to obtain a valid RTT measurement.

A solution to these problems, which actually simplifies the sender substantially, is as follows: using TCP options, the sender places a timestamp in each data segment, and the receiver reflects these timestamps back in ACK segments. Then a single subtract gives the sender an accurate RTT measurement for every ACK segment (which will correspond to every other data segment, with a sensible receiver). We call this the RTTM (Round-Trip Time Measurement) mechanism.

It is vitally important to use the RTTM mechanism with big windows; otherwise, the door is opened to some dangerous instabilities due to aliasing. Furthermore, the option is probably useful for all TCP's, since it simplifies the sender.

3.2 TCP Timestamps Option

TCP is a symmetric protocol, allowing data to be sent at any time in either direction, and therefore timestamp echoing may occur in either direction. For simplicity and symmetry, we specify that timestamps always be sent and echoed in both directions. For efficiency, we combine the timestamp and timestamp reply fields into a single TCP Timestamps Option.

TCP Timestamps Option (TSopt):

Kind: 8

Length: 10 bytes

+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
Kind=8		10		TS Value (TSval)				TS Echo Reply (TSecr)	
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+									
1		1		4				4	

The Timestamps option carries two four-byte timestamp fields. The Timestamp Value field (TSval) contains the current value of the timestamp clock of the TCP sending the option.

The Timestamp Echo Reply field (TSecr) is only valid if the ACK bit is set in the TCP header; if it is valid, it echos a timestamp value that was sent by the remote TCP in the TSval field of a Timestamps option. When TSecr is not valid, its value must be zero. The TSecr value will generally be from the most recent Timestamp option that was received; however, there are exceptions that are explained below.

A TCP may send the Timestamps option (TSopt) in an initial <SYN> segment (i.e., segment containing a SYN bit and no ACK bit), and may send a TSopt in other segments only if it received a TSopt in the initial <SYN> segment for the connection.

3.3 The RTTM Mechanism

The timestamp value to be sent in TSval is to be obtained from a (virtual) clock that we call the "timestamp clock". Its values must be at least approximately proportional to real time, in order to measure actual RTT.

The following example illustrates a one-way data flow with segments arriving in sequence without loss. Here A, B, C... represent data blocks occupying successive blocks of sequence numbers, and ACK(A),... represent the corresponding cumulative acknowledgments. The two timestamp fields of the Timestamps option are shown symbolically as <TSval= x,TSecr=y>. Each TSecr field contains the value most recently received in a TSval field.

```

TCP  A                                     TCP  B

      <A,TSval=1,TSecr=120> ----->

<----- <ACK(A),TSval=127,TSecr=1>

      <B,TSval=5,TSecr=127> ----->

<----- <ACK(B),TSval=131,TSecr=5>

. . . . .

      <C,TSval=65,TSecr=131> ----->

<----- <ACK(C),TSval=191,TSecr=65>

      (etc)

```

The dotted line marks a pause (60 time units long) in which A had nothing to send. Note that this pause inflates the RTT which B could infer from receiving TSecr=131 in data segment C. Thus, in one-way data flows, RTTM in the reverse direction measures a value that is inflated by gaps in sending data. However, the following rule prevents a resulting inflation of the measured RTT:

A TSecr value received in a segment is used to update the averaged RTT measurement only if the segment acknowledges some new data, i.e., only if it advances the left edge of the send window.

Since TCP B is not sending data, the data segment C does not acknowledge any new data when it arrives at B. Thus, the inflated RTTM measurement is not used to update B's RTTM measurement.

3.4 Which Timestamp to Echo

If more than one Timestamps option is received before a reply segment is sent, the TCP must choose only one of the TSvals to echo, ignoring the others. To minimize the state kept in the receiver (i.e., the number of unprocessed TSvals), the receiver should be required to retain at most one timestamp in the connection control block.

There are three situations to consider:

(A) Delayed ACKs.

Many TCP's acknowledge only every Kth segment out of a group of segments arriving within a short time interval; this policy is known generally as "delayed ACKs". The data-sender TCP must measure the effective RTT, including the additional time due to delayed ACKs, or else it will retransmit unnecessarily. Thus, when delayed ACKs are in use, the receiver should reply with the TSval field from the earliest unacknowledged segment.

(B) A hole in the sequence space (segment(s) have been lost).

The sender will continue sending until the window is filled, and the receiver may be generating ACKs as these out-of-order segments arrive (e.g., to aid "fast retransmit").

The lost segment is probably a sign of congestion, and in that situation the sender should be conservative about retransmission. Furthermore, it is better to overestimate than underestimate the RTT. An ACK for an out-of-order segment should therefore contain the timestamp from the most recent segment that advanced the window.

The same situation occurs if segments are re-ordered by the network.

(C) A filled hole in the sequence space.

The segment that fills the hole represents the most recent measurement of the network characteristics. On the other hand, an RTT computed from an earlier segment would probably include the sender's retransmit time-out, badly biasing the sender's average RTT estimate. Thus, the timestamp from the latest segment (which filled the hole) must be echoed.

An algorithm that covers all three cases is described in the following rules for Timestamps option processing on a synchronized connection:

- (1) The connection state is augmented with two 32-bit slots: TS.Recent holds a timestamp to be echoed in TSecr whenever a segment is sent, and Last.ACK.sent holds the ACK field from the last segment sent. Last.ACK.sent will equal RCV.NXT except when ACKs have been delayed.

- (2) If Last.ACK.sent falls within the range of sequence numbers of an incoming segment:

$$\text{SEG.SEQ} \leq \text{Last.ACK.sent} < \text{SEG.SEQ} + \text{SEG.LEN}$$

then the TSval from the segment is copied to TS.Recent;
otherwise, the TSval is ignored.

- (3) When a TSOpt is sent, its TSecr field is set to the current TS.Recent value.

The following examples illustrate these rules. Here A, B, C... represent data segments occupying successive blocks of sequence numbers, and ACK(A),... represent the corresponding acknowledgment segments. Note that ACK(A) has the same sequence number as B. We show only one direction of timestamp echoing, for clarity.

- o Packets arrive in sequence, and some of the ACKs are delayed.

By Case (A), the timestamp from the oldest unacknowledged segment is echoed.

	TS.Recent
<A, TSval=1> ----->	1
<B, TSval=2> ----->	1
<C, TSval=3> ----->	1
<---- <ACK(C), TSecr=1>	
(etc)	

- o Packets arrive out of order, and every packet is acknowledged.

By Case (B), the timestamp from the last segment that advanced the left window edge is echoed, until the missing segment arrives; it is echoed according to Case (C). The same sequence would occur if segments B and D were lost and retransmitted..

	TS.Recent
<A, TSval=1> ----->	
<----- <ACK(A), TSecr=1>	1
	1
<C, TSval=3> ----->	
<----- <ACK(A), TSecr=1>	1
	1
<B, TSval=2> ----->	
<----- <ACK(C), TSecr=2>	2
	2
<E, TSval=5> ----->	
<----- <ACK(C), TSecr=2>	2
	2
<D, TSval=4> ----->	
<----- <ACK(E), TSecr=4>	4
(etc)	

4. PAWS: PROTECT AGAINST WRAPPED SEQUENCE NUMBERS

4.1 Introduction

Section 4.2 describes a simple mechanism to reject old duplicate segments that might corrupt an open TCP connection; we call this mechanism PAWS (Protect Against Wrapped Sequence numbers). PAWS operates within a single TCP connection, using state that is saved in the connection control block. Section 4.3 and Appendix C discuss the implications of the PAWS mechanism for avoiding old duplicates from previous incarnations of the same connection.

4.2 The PAWS Mechanism

PAWS uses the same TCP Timestamps option as the RTTM mechanism described earlier, and assumes that every received TCP segment (including data and ACK segments) contains a timestamp SEG.TSval whose values are monotone non-decreasing in time. The basic idea is that a segment can be discarded as an old duplicate if it is received with a timestamp SEG.TSval less than some timestamp recently received on this connection.

In both the PAWS and the RTTM mechanism, the "timestamps" are 32-

bit unsigned integers in a modular 32-bit space. Thus, "less than" is defined the same way it is for TCP sequence numbers, and the same implementation techniques apply. If s and t are timestamp values, $s < t$ if $0 < (t - s) < 2^{31}$, computed in unsigned 32-bit arithmetic.

The choice of incoming timestamps to be saved for this comparison must guarantee a value that is monotone increasing. For example, we might save the timestamp from the segment that last advanced the left edge of the receive window, i.e., the most recent in-sequence segment. Instead, we choose the value `TS.Recent` introduced in Section 3.4 for the RTTM mechanism, since using a common value for both PAWS and RTTM simplifies the implementation of both. As Section 3.4 explained, `TS.Recent` differs from the timestamp from the last in-sequence segment only in the case of delayed ACKs, and therefore by less than one window. Either choice will therefore protect against sequence number wrap-around.

RTTM was specified in a symmetrical manner, so that `TSval` timestamps are carried in both data and ACK segments and are echoed in `TSecr` fields carried in returning ACK or data segments. PAWS submits all incoming segments to the same test, and therefore protects against duplicate ACK segments as well as data segments. (An alternative un-symmetric algorithm would protect against old duplicate ACKs: the sender of data would reject incoming ACK segments whose `TSecr` values were less than the `TSecr` saved from the last segment whose ACK field advanced the left edge of the send window. This algorithm was deemed to lack economy of mechanism and symmetry.)

`TSval` timestamps sent on `{SYN}` and `{SYN,ACK}` segments are used to initialize PAWS. PAWS protects against old duplicate non-SYN segments, and duplicate SYN segments received while there is a synchronized connection. Duplicate `{SYN}` and `{SYN,ACK}` segments received when there is no connection will be discarded by the normal 3-way handshake and sequence number checks of TCP.

It is recommended that RST segments NOT carry timestamps, and that RST segments be acceptable regardless of their timestamp. Old duplicate RST segments should be exceedingly unlikely, and their cleanup function should take precedence over timestamps.

4.2.1 Basic PAWS Algorithm

The PAWS algorithm requires the following processing to be performed on all incoming segments for a synchronized connection:

- R1) If there is a Timestamps option in the arriving segment and `SEG.TSval < TS.Recent` and if `TS.Recent` is valid (see later discussion), then treat the arriving segment as not acceptable:

Send an acknowledgement in reply as specified in RFC-793 page 69 and drop the segment.

Note: it is necessary to send an ACK segment in order to retain TCP's mechanisms for detecting and recovering from half-open connections. For example, see Figure 10 of RFC-793.

- R2) If the segment is outside the window, reject it (normal TCP processing)
- R3) If an arriving segment satisfies: `SEG.SEQ <= Last.ACK.sent` (see Section 3.4), then record its timestamp in `TS.Recent`.
- R4) If an arriving segment is in-sequence (i.e., at the left window edge), then accept it normally.
- R5) Otherwise, treat the segment as a normal in-window, out-of-sequence TCP segment (e.g., queue it for later delivery to the user).

Steps R2, R4, and R5 are the normal TCP processing steps specified by RFC-793.

It is important to note that the timestamp is checked only when a segment first arrives at the receiver, regardless of whether it is in-sequence or it must be queued for later delivery. Consider the following example.

Suppose the segment sequence: A.1, B.1, C.1, ..., Z.1 has been sent, where the letter indicates the sequence number and the digit represents the timestamp. Suppose also that segment B.1 has been lost. The timestamp in `TS.TStamp` is 1 (from A.1), so C.1, ..., Z.1 are considered acceptable and are queued. When B is retransmitted as segment B.2 (using the latest timestamp), it fills the hole and causes all the segments through Z to be acknowledged and passed to the user. The timestamps of the queued segments are **not** inspected again at this time, since they have already been accepted. When B.2 is accepted, `TS.Stamp` is set to 2.

This rule allows reasonable performance under loss. A full

window of data is in transit at all times, and after a loss a full window less one packet will show up out-of-sequence to be queued at the receiver (e.g., up to 2^{30} bytes of data); the timestamp option must not result in discarding this data.

In certain unlikely circumstances, the algorithm of rules R1-R4 could lead to discarding some segments unnecessarily, as shown in the following example:

Suppose again that segments: A.1, B.1, C.1, ..., Z.1 have been sent in sequence and that segment B.1 has been lost. Furthermore, suppose delivery of some of C.1, ... Z.1 is delayed until AFTER the retransmission B.2 arrives at the receiver. These delayed segments will be discarded unnecessarily when they do arrive, since their timestamps are now out of date.

This case is very unlikely to occur. If the retransmission was triggered by a timeout, some of the segments C.1, ... Z.1 must have been delayed longer than the RTO time. This is presumably an unlikely event, or there would be many spurious timeouts and retransmissions. If B's retransmission was triggered by the "fast retransmit" algorithm, i.e., by duplicate ACKs, then the queued segments that caused these ACKs must have been received already.

Even if a segment were delayed past the RTO, the Fast Retransmit mechanism [Jacobson90c] will cause the delayed packets to be retransmitted at the same time as B.2, avoiding an extra RTT and therefore causing a very small performance penalty.

We know of no case with a significant probability of occurrence in which timestamps will cause performance degradation by unnecessarily discarding segments.

4.2.2 Timestamp Clock

It is important to understand that the PAWS algorithm does not require clock synchronization between sender and receiver. The sender's timestamp clock is used to stamp the segments, and the sender uses the echoed timestamp to measure RTT's. However, the receiver treats the timestamp as simply a monotone-increasing serial number, without any necessary connection to its clock. From the receiver's viewpoint, the timestamp is acting as a logical extension of the high-order bits of the sequence number.

The receiver algorithm does place some requirements on the frequency of the timestamp clock.

- (a) The timestamp clock must not be "too slow".

It must tick at least once for each 2^{31} bytes sent. In fact, in order to be useful to the sender for round trip timing, the clock should tick at least once per window's worth of data, and even with the RFC-1072 window extension, 2^{31} bytes must be at least two windows.

To make this more quantitative, any clock faster than 1 tick/sec will reject old duplicate segments for link speeds of ~8 Gbps. A 1ms timestamp clock will work at link speeds up to 8 Tbps ($8 \cdot 10^{12}$ bps!)

- (b) The timestamp clock must not be "too fast".

Its recycling time must be greater than MSL seconds. Since the clock (timestamp) is 32 bits and the worst-case MSL is 255 seconds, the maximum acceptable clock frequency is one tick every 59 ns.

However, it is desirable to establish a much longer recycle period, in order to handle outdated timestamps on idle connections (see Section 4.2.3), and to relax the MSL requirement for preventing sequence number wrap-around. With a 1 ms timestamp clock, the 32-bit timestamp will wrap its sign bit in 24.8 days. Thus, it will reject old duplicates on the same connection if MSL is 24.8 days or less. This appears to be a very safe figure; an MSL of 24.8 days or longer can probably be assumed by the gateway system without requiring precise MSL enforcement by the TTL value in the IP layer.

Based upon these considerations, we choose a timestamp clock frequency in the range 1 ms to 1 sec per tick. This range also matches the requirements of the RTTM mechanism, which does not need much more resolution than the granularity of the retransmit timer, e.g., tens or hundreds of milliseconds.

The PAWS mechanism also puts a strong monotonicity requirement on the sender's timestamp clock. The method of implementation of the timestamp clock to meet this requirement depends upon the system hardware and software.

- * Some hosts have a hardware clock that is guaranteed to be monotonic between hardware resets.

- * A clock interrupt may be used to simply increment a binary integer by 1 periodically.
- * The timestamp clock may be derived from a system clock that is subject to being abruptly changed, by adding a variable offset value. This offset is initialized to zero. When a new timestamp clock value is needed, the offset can be adjusted as necessary to make the new value equal to or larger than the previous value (which was saved for this purpose).

4.2.3 Outdated Timestamps

If a connection remains idle long enough for the timestamp clock of the other TCP to wrap its sign bit, then the value saved in TS.Recent will become too old; as a result, the PAWS mechanism will cause all subsequent segments to be rejected, freezing the connection (until the timestamp clock wraps its sign bit again).

With the chosen range of timestamp clock frequencies (1 sec to 1 ms), the time to wrap the sign bit will be between 24.8 days and 24800 days. A TCP connection that is idle for more than 24 days and then comes to life is exceedingly unusual. However, it is undesirable in principle to place any limitation on TCP connection lifetimes.

We therefore require that an implementation of PAWS include a mechanism to "invalidate" the TS.Recent value when a connection is idle for more than 24 days. (An alternative solution to the problem of outdated timestamps would be to send keepalive segments at a very low rate, but still more often than the wrap-around time for timestamps, e.g., once a day. This would impose negligible overhead. However, the TCP specification has never included keepalives, so the solution based upon invalidation was chosen.)

Note that a TCP does not know the frequency, and therefore, the wraparound time, of the other TCP, so it must assume the worst. The validity of TS.Recent needs to be checked only if the basic PAWS timestamp check fails, i.e., only if $SEG.TSval < TS.Recent$. If TS.Recent is found to be invalid, then the segment is accepted, regardless of the failure of the timestamp check, and rule R3 updates TS.Recent with the TSval from the new segment.

To detect how long the connection has been idle, the TCP may

update a clock or timestamp value associated with the connection whenever TS.Recent is updated, for example. The details will be implementation-dependent.

4.2.4 Header Prediction

"Header prediction" [Jacobson90a] is a high-performance transport protocol implementation technique that is most important for high-speed links. This technique optimizes the code for the most common case, receiving a segment correctly and in order. Using header prediction, the receiver asks the question, "Is this segment the next in sequence?" This question can be answered in fewer machine instructions than the question, "Is this segment within the window?"

Adding header prediction to our timestamp procedure leads to the following recommended sequence for processing an arriving TCP segment:

- H1) Check timestamp (same as step R1 above)
- H2) Do header prediction: if segment is next in sequence and if there are no special conditions requiring additional processing, accept the segment, record its timestamp, and skip H3.
- H3) Process the segment normally, as specified in RFC-793. This includes dropping segments that are outside the window and possibly sending acknowledgments, and queueing in-window, out-of-sequence segments.

Another possibility would be to interchange steps H1 and H2, i.e., to perform the header prediction step H2 FIRST, and perform H1 and H3 only when header prediction fails. This could be a performance improvement, since the timestamp check in step H1 is very unlikely to fail, and it requires interval arithmetic on a finite field, a relatively expensive operation. To perform this check on every single segment is contrary to the philosophy of header prediction. We believe that this change might reduce CPU time for TCP protocol processing by up to 5-10% on high-speed networks.

However, putting H2 first would create a hazard: a segment from 2^{32} bytes in the past might arrive at exactly the wrong time and be accepted mistakenly by the header-prediction step. The following reasoning has been introduced [Jacobson90b] to show that the probability of this failure is negligible.

If all segments are equally likely to show up as old duplicates, then the probability of an old duplicate exactly matching the left window edge is the maximum segment size (MSS) divided by the size of the sequence space. This ratio must be less than 2^{*-16} , since MSS must be $< 2^{*16}$; for example, it will be $(2^{*12})/(2^{*32}) = 2^{*-20}$ for an FDDI link. However, the older a segment is, the less likely it is to be retained in the Internet, and under any reasonable model of segment lifetime the probability of an old duplicate exactly at the left window edge must be much smaller than 2^{*-16} .

The 16 bit TCP checksum also allows a basic unreliability of one part in 2^{*16} . A protocol mechanism whose reliability exceeds the reliability of the TCP checksum should be considered "good enough", i.e., it won't contribute significantly to the overall error rate. We therefore believe we can ignore the problem of an old duplicate being accepted by doing header prediction before checking the timestamp.

However, this probabilistic argument is not universally accepted, and the consensus at present is that the performance gain does not justify the hazard in the general case. It is therefore recommended that H2 follow H1.

4.3. Duplicates from Earlier Incarnations of Connection

The PAWS mechanism protects against errors due to sequence number wrap-around on high-speed connection. Segments from an earlier incarnation of the same connection are also a potential cause of old duplicate errors. In both cases, the TCP mechanisms to prevent such errors depend upon the enforcement of a maximum segment lifetime (MSL) by the Internet (IP) layer (see Appendix of RFC-1185 for a detailed discussion). Unlike the case of sequence space wrap-around, the MSL required to prevent old duplicate errors from earlier incarnations does not depend upon the transfer rate. If the IP layer enforces the recommended 2 minute MSL of TCP, and if the TCP rules are followed, TCP connections will be safe from earlier incarnations, no matter how high the network speed. Thus, the PAWS mechanism is not required for this case.

We may still ask whether the PAWS mechanism can provide additional security against old duplicates from earlier connections, allowing us to relax the enforcement of MSL by the IP layer. Appendix B explores this question, showing that further assumptions and/or mechanisms are required, beyond those of PAWS. This is not part of the current extension.

5. CONCLUSIONS AND ACKNOWLEDGMENTS

This memo presented a set of extensions to TCP to provide efficient operation over large-bandwidth*delay-product paths and reliable operation over very high-speed paths. These extensions are designed to provide compatible interworking with TCP's that do not implement the extensions.

These mechanisms are implemented using new TCP options for scaled windows and timestamps. The timestamps are used for two distinct mechanisms: RTTM (Round Trip Time Measurement) and PAWS (Protect Against Wrapped Sequences).

The Window Scale option was originally suggested by Mike St. Johns of USAF/DCA. The present form of the option was suggested by Mike Karels of UC Berkeley in response to a more cumbersome scheme defined by Van Jacobson. Lixia Zhang helped formulate the PAWS mechanism description in RFC-1185.

Finally, much of this work originated as the result of discussions within the End-to-End Task Force on the theoretical limitations of transport protocols in general and TCP in particular. More recently, task force members and other on the end2end-interest list have made valuable contributions by pointing out flaws in the algorithms and the documentation. The authors are grateful for all these contributions.

6. REFERENCES

- [Clark87] Clark, D., Lambert, M., and L. Zhang, "NETBLT: A Bulk Data Transfer Protocol", RFC 998, MIT, March 1987.
- [Garlick77] Garlick, L., R. Rom, and J. Postel, "Issues in Reliable Host-to-Host Protocols", Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, May 1977.
- [Hamming77] Hamming, R., "Digital Filters", ISBN 0-13-212571-4, Prentice Hall, Englewood Cliffs, N.J., 1977.
- [Cheriton88] Cheriton, D., "VMTP: Versatile Message Transaction Protocol", RFC 1045, Stanford University, February 1988.
- [Jacobson88a] Jacobson, V., "Congestion Avoidance and Control", SIGCOMM '88, Stanford, CA., August 1988.
- [Jacobson88b] Jacobson, V., and R. Braden, "TCP Extensions for Long-Delay Paths", RFC-1072, LBL and USC/Information Sciences Institute, October 1988.

- [Jacobson90a] Jacobson, V., "4BSD Header Prediction", ACM Computer Communication Review, April 1990.
- [Jacobson90b] Jacobson, V., Braden, R., and Zhang, L., "TCP Extension for High-Speed Paths", RFC-1185, LBL and USC/Information Sciences Institute, October 1990.
- [Jacobson90c] Jacobson, V., "Modified TCP congestion avoidance algorithm", Message to end2end-interest mailing list, April 1990.
- [Jain86] Jain, R., "Divergence of Timeout Algorithms for Packet Retransmissions", Proc. Fifth Phoenix Conf. on Comp. and Comm., Scottsdale, Arizona, March 1986.
- [Karn87] Karn, P. and C. Partridge, "Estimating Round-Trip Times in Reliable Transport Protocols", Proc. SIGCOMM '87, Stowe, VT, August 1987.
- [McKenzie89] McKenzie, A., "A Problem with the TCP Big Window Option", RFC 1110, BBN STC, August 1989.
- [Nagle84] Nagle, J., "Congestion Control in IP/TCP Internetworks", RFC 896, FACC, January 1984.
- [NBS85] Colella, R., Aronoff, R., and K. Mills, "Performance Improvements for ISO Transport", Ninth Data Comm Symposium, published in ACM SIGCOMM Comp Comm Review, vol. 15, no. 5, September 1985.
- [Postel81] Postel, J., "Transmission Control Protocol - DARPA Internet Program Protocol Specification", RFC 793, DARPA, September 1981.
- [Velten84] Velten, D., Hinden, R., and J. Sax, "Reliable Data Protocol", RFC 908, BBN, July 1984.
- [Watson81] Watson, R., "Timer-based Mechanisms in Reliable Transport Protocol Connection Management", Computer Networks, Vol. 5, 1981.
- [Zhang86] Zhang, L., "Why TCP Timers Don't Work Well", Proc. SIGCOMM '86, Stowe, Vt., August 1986.

APPENDIX A: IMPLEMENTATION SUGGESTIONS

The following layouts are recommended for sending options on non-SYN segments, to achieve maximum feasible alignment of 32-bit and 64-bit machines.

+	-----+	-----+	-----+	-----+
	NOP		NOP	
			TSopt	
			10	
+	-----+	-----+	-----+	-----+
			TSval	
			timestamp	
+	-----+	-----+	-----+	-----+
			TSecr	
			timestamp	
+	-----+	-----+	-----+	-----+

APPENDIX B: DUPLICATES FROM EARLIER CONNECTION INCARNATIONS

There are two cases to be considered: (1) a system crashing (and losing connection state) and restarting, and (2) the same connection being closed and reopened without a loss of host state. These will be described in the following two sections.

B.1 System Crash with Loss of State

TCP's quiet time of one MSL upon system startup handles the loss of connection state in a system crash/restart. For an explanation, see for example "When to Keep Quiet" in the TCP protocol specification [Postel81]. The MSL that is required here does not depend upon the transfer speed. The current TCP MSL of 2 minutes seems acceptable as an operational compromise, as many host systems take this long to boot after a crash.

However, the timestamp option may be used to ease the MSL requirements (or to provide additional security against data corruption). If timestamps are being used and if the timestamp clock can be guaranteed to be monotonic over a system crash/restart, i.e., if the first value of the sender's timestamp clock after a crash/restart can be guaranteed to be greater than the last value before the restart, then a quiet time will be unnecessary.

To dispense totally with the quiet time would require that the host clock be synchronized to a time source that is stable over the crash/restart period, with an accuracy of one timestamp clock tick or better. We can back off from this strict requirement to take advantage of approximate clock synchronization. Suppose that the clock is always re-synchronized to within N timestamp clock

ticks and that booting (extended with a quiet time, if necessary) takes more than N ticks. This will guarantee monotonicity of the timestamps, which can then be used to reject old duplicates even without an enforced MSL.

B.2 Closing and Reopening a Connection

When a TCP connection is closed, a delay of $2 \times \text{MSL}$ in TIME-WAIT state ties up the socket pair for 4 minutes (see Section 3.5 of [Postel81]). Applications built upon TCP that close one connection and open a new one (e.g., an FTP data transfer connection using Stream mode) must choose a new socket pair each time. The TIME-WAIT delay serves two different purposes:

- (a) Implement the full-duplex reliable close handshake of TCP.

The proper time to delay the final close step is not really related to the MSL; it depends instead upon the RTT for the FIN segments and therefore upon the RTT of the path. (It could be argued that the side that is sending a FIN knows what degree of reliability it needs, and therefore it should be able to determine the length of the TIME-WAIT delay for the FIN's recipient. This could be accomplished with an appropriate TCP option in FIN segments.)

Although there is no formal upper-bound on RTT, common network engineering practice makes an RTT greater than 1 minute very unlikely. Thus, the 4 minute delay in TIME-WAIT state works satisfactorily to provide a reliable full-duplex TCP close. Note again that this is independent of MSL enforcement and network speed.

The TIME-WAIT state could cause an indirect performance problem if an application needed to repeatedly close one connection and open another at a very high frequency, since the number of available TCP ports on a host is less than 2^{16} . However, high network speeds are not the major contributor to this problem; the RTT is the limiting factor in how quickly connections can be opened and closed. Therefore, this problem will be no worse at high transfer speeds.

- (b) Allow old duplicate segments to expire.

To replace this function of TIME-WAIT state, a mechanism would have to operate across connections. PAWS is defined strictly within a single connection; the last timestamp is TS.Recent is kept in the connection control block, and

discarded when a connection is closed.

An additional mechanism could be added to the TCP, a per-host cache of the last timestamp received from any connection. This value could then be used in the PAWS mechanism to reject old duplicate segments from earlier incarnations of the connection, if the timestamp clock can be guaranteed to have ticked at least once since the old connection was open. This would require that the TIME-WAIT delay plus the RTT together must be at least one tick of the sender's timestamp clock. Such an extension is not part of the proposal of this RFC.

Note that this is a variant on the mechanism proposed by Garlick, Rom, and Postel [Garlick77], which required each host to maintain connection records containing the highest sequence numbers on every connection. Using timestamps instead, it is only necessary to keep one quantity per remote host, regardless of the number of simultaneous connections to that host.

APPENDIX C: CHANGES FROM RFC-1072, RFC-1185

The protocol extensions defined in this document differ in several important ways from those defined in RFC-1072 and RFC-1185.

- (a) SACK has been deferred to a later memo.
- (b) The detailed rules for sending timestamp replies (see Section 3.4) differ in important ways. The earlier rules could result in an under-estimate of the RTT in certain cases (packets dropped or out of order).
- (c) The same value TS.Recent is now shared by the two distinct mechanisms RTTM and PAWS. This simplification became possible because of change (b).
- (d) An ambiguity in RFC-1185 was resolved in favor of putting timestamps on ACK as well as data segments. This supports the symmetry of the underlying TCP protocol.
- (e) The echo and echo reply options of RFC-1072 were combined into a single Timestamps option, to reflect the symmetry and to simplify processing.
- (f) The problem of outdated timestamps on long-idle connections, discussed in Section 4.2.2, was realized and resolved.
- (g) RFC-1185 recommended that header prediction take precedence over the timestamp check. Based upon some scepticism about the probabilistic arguments given in Section 4.2.4, it was decided to recommend that the timestamp check be performed first.
- (h) The spec was modified so that the extended options will be sent on <SYN,ACK> segments only when they are received in the corresponding <SYN> segments. This provides the most conservative possible conditions for interoperation with implementations without the extensions.

In addition to these substantive changes, the present RFC attempts to specify the algorithms unambiguously by presenting modifications to the Event Processing rules of RFC-793; see Appendix E.

APPENDIX D: SUMMARY OF NOTATION

The following notation has been used in this document.

Options

WSopt: TCP Window Scale Option
TSopt: TCP Timestamps Option

Option Fields

shift.cnt: Window scale byte in WSopt.
TSval: 32-bit Timestamp Value field in TSopt.
TSecr: 32-bit Timestamp Reply field in TSopt.

Option Fields in Current Segment

SEG.TSval: TSval field from TSopt in current segment.
SEG.TSecr: TSecr field from TSopt in current segment.
SEG.WSopt: 8-bit value in WSopt

Clock Values

my.TSclock: Local source of 32-bit timestamp values
my.TSclock.rate: Period of my.TSclock (1 ms to 1 sec).

Per-Connection State Variables

TS.Recent: Latest received Timestamp
Last.ACK.sent: Last ACK field sent

Snd.TS.OK: 1-bit flag
Snd.WS.OK: 1-bit flag

Rcv.Wind.Scale: Receive window scale power
Snd.Wind.Scale: Send window scale power

APPENDIX E: EVENT PROCESSING

Event Processing

OPEN Call

...

An initial send sequence number (ISS) is selected. Send a SYN segment of the form:

<SEQ=ISS><CTL=SYN><TSval=my.TSclock><WSopt=Rcv.Wind.Scale>

...

SEND Call

CLOSED STATE (i.e., TCB does not exist)

...

LISTEN STATE

If the foreign socket is specified, then change the connection from passive to active, select an ISS. Send a SYN segment containing the options: <TSval=my.TSclock> and <WSopt=Rcv.Wind.Scale>. Set SND.UNA to ISS, SND.NXT to ISS+1. Enter SYN-SENT state. ...

SYN-SENT STATE

SYN-RECEIVED STATE

...

ESTABLISHED STATE

CLOSE-WAIT STATE

Segmentize the buffer and send it with a piggybacked acknowledgment (acknowledgment value = RCV.NXT). ...

If the urgent flag is set ...

If the Snd.TS.OK flag is set, then include the TCP Timestamps option <TSval=my.TSclock,TSecr=TS.Recent> in each data segment.

Scale the receive window for transmission in the segment header:

SEG.WND = (SND.WND >> Rcv.Wind.Scale).

SEGMENT ARRIVES

...

If the state is LISTEN then

first check for an RST

...

second check for an ACK

...

third check for a SYN

if the SYN bit is set, check the security. If the ...

...

If the SEG.PRC is less than the TCB.PRC then continue.

Check for a Window Scale option (WSopt); if one is found, save SEG.WSopt in Snd.Wind.Scale and set Snd.WS.OK flag on. Otherwise, set both Snd.Wind.Scale and Rcv.Wind.Scale to zero and clear Snd.WS.OK flag.

Check for a TSopt option; if one is found, save SEG.TSval in the variable TS.Recent and turn on the Snd.TS.OK bit.

Set RCV.NXT to SEG.SEQ+1, IRS is set to SEG.SEQ and any other control or text should be queued for processing later. ISS should be selected and a SYN segment sent of the form:

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

If the Snd.WS.OK bit is on, include a WSopt option <WSopt=Rcv.Wind.Scale> in this segment. If the Snd.TS.OK bit is on, include a TSopt <TSval=my.TSclock,TSecr=TS.Recent> in this segment. Last.ACK.sent is set to RCV.NXT.

SND.NXT is set to ISS+1 and SND.UNA to ISS. The connection state should be changed to SYN-RECEIVED. Note that any other incoming control or data (combined with SYN) will be processed in the SYN-RECEIVED state, but processing of SYN and ACK should not be repeated. If the listen was not fully specified (i.e., the foreign socket was not fully specified), then the unspecified fields should be filled in now.

fourth other text or control

...

If the state is SYN-SENT then

first check the ACK bit

...

fourth check the SYN bit

...

If the SYN bit is on and the security/compartment and precedence are acceptable then, RCV.NXT is set to SEG.SEQ+1, IRS is set to SEG.SEQ, and any acknowledgements on the retransmission queue which are thereby acknowledged should be removed.

Check for a Window Scale option (WSopt); if is found, save SEG.WSopt in Snd.Wind.Scale; otherwise, set both Snd.Wind.Scale and Rcv.Wind.Scale to zero.

Check for a TSopt option; if one is found, save SEG.TSval in variable TS.Recent and turn on the Snd.TS.OK bit in the connection control block. If the ACK bit is set, use my.TSclock - SEG.TSecr as the initial RTT estimate.

If SND.UNA > ISS (our SYN has been ACKed), change the connection state to ESTABLISHED, form an ACK segment:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

and send it. If the Snd.Echo.OK bit is on, include a TSopt option <TSval=my.TSclock,TSecr=TS.Recent> in this ACK segment. Last.ACK.sent is set to RCV.NXT.

Data or controls which were queued for transmission may be included. If there are other controls or text in the segment then continue processing at the sixth step below where the URG bit is checked, otherwise return.

Otherwise enter SYN-RECEIVED, form a SYN,ACK segment:

<SEQ=ISS><ACK=RCV.NXT><CTL=SYN,ACK>

and send it. If the Snd.Echo.OK bit is on, include a TSopt option <TSval=my.TSclock,TSecr=TS.Recent> in this segment. If

the Snd.WS.OK bit is on, include a WSopt option
<WSopt=Rcv.Wind.Scale> in this segment. Last.ACK.sent is set to
RCV.NXT.

If there are other controls or text in the segment, queue them
for processing after the ESTABLISHED state has been reached,
return.

fifth, if neither of the SYN or RST bits is set then drop the
segment and return.

Otherwise,

First, check sequence number

SYN-RECEIVED STATE
ESTABLISHED STATE
FIN-WAIT-1 STATE
FIN-WAIT-2 STATE
CLOSE-WAIT STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE

Segments are processed in sequence. Initial tests on arrival
are used to discard old duplicates, but further processing is
done in SEG.SEQ order. If a segment's contents straddle the
boundary between old and new, only the new parts should be
processed.

Rescale the received window field:

TrueWindow = SEG.WND << Snd.Wind.Scale,

and use "TrueWindow" in place of SEG.WND in the following steps.

Check whether the segment contains a Timestamps option and bit
Snd.TS.OK is on. If so:

If SEG.TSval < TS.Recent, then test whether connection has
been idle less than 24 days; if both are true, then the
segment is not acceptable; follow steps below for an
unacceptable segment.

If SEG.SEQ is equal to Last.ACK.sent, then save SEG.ECopt in
variable TS.Recent.

There are four cases for the acceptability test for an incoming segment:

...

If an incoming segment is not acceptable, an acknowledgment should be sent in reply (unless the RST bit is set, if so drop the segment and return):

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

Last.ACK.sent is set to SEG.ACK of the acknowledgment. If the Snd.Echo.OK bit is on, include the Timestamps option <TSval=my.TSclock,TSecr=TS.Recent> in this ACK segment. Set Last.ACK.sent to SEG.ACK and send the ACK segment. After sending the acknowledgment, drop the unacceptable segment and return.

...

fifth check the ACK field.

if the ACK bit is off drop the segment and return.

if the ACK bit is on

...

ESTABLISHED STATE

If SND.UNA < SEG.ACK =< SND.NXT then, set SND.UNA <- SEG.ACK. Also compute a new estimate of round-trip time. If Snd.TS.OK bit is on, use my.TSclock - SEG.TSecr; otherwise use the elapsed time since the first segment in the retransmission queue was sent. Any segments on the retransmission queue which are thereby entirely acknowledged...

...

Seventh, process the segment text.

ESTABLISHED STATE

FIN-WAIT-1 STATE

FIN-WAIT-2 STATE

...

Send an acknowledgment of the form:

<SEQ=SND.NXT><ACK=RCV.NXT><CTL=ACK>

If the Snd.TS.OK bit is on, include Timestamps option
<TSval=my.TSclock,TSecr=TS.Recent> in this ACK segment. Set
Last.ACK.sent to SEG.ACK of the acknowledgment, and send it.
This acknowledgment should be piggy-backed on a segment being
transmitted if possible without incurring undue delay.

...

Security Considerations

Security issues are not discussed in this memo.

Authors' Addresses

Van Jacobson
University of California
Lawrence Berkeley Laboratory
Mail Stop 46A
Berkeley, CA 94720

Phone: (415) 486-6411
EMail: van@CSAM.LBL.GOV

Bob Braden
University of Southern California
Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

Phone: (310) 822-1511
EMail: Braden@ISI.EDU

Dave Borman
Cray Research
655-E Lone Oak Drive
Eagan, MN 55121

Phone: (612) 683-5571
Email: dab@cray.com

Network Working Group
Request for Comments: 2581
Obsoletes: 2001
Category: Standards Track

M. Allman
NASA Glenn/Sterling Software
V. Paxson
ACIRI / ICSI
W. Stevens
Consultant
April 1999

TCP Congestion Control

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (1999). All Rights Reserved.

Abstract

This document defines TCP's four intertwined congestion control algorithms: slow start, congestion avoidance, fast retransmit, and fast recovery. In addition, the document specifies how TCP should begin transmission after a relatively long idle period, as well as discussing various acknowledgment generation methods.

1. Introduction

This document specifies four TCP [Pos81] congestion control algorithms: slow start, congestion avoidance, fast retransmit and fast recovery. These algorithms were devised in [Jac88] and [Jac90]. Their use with TCP is standardized in [Bra89].

This document is an update of [Ste97]. In addition to specifying the congestion control algorithms, this document specifies what TCP connections should do after a relatively long idle period, as well as specifying and clarifying some of the issues pertaining to TCP ACK generation.

Note that [Ste94] provides examples of these algorithms in action and [WS95] provides an explanation of the source code for the BSD implementation of these algorithms.

This document is organized as follows. Section 2 provides various definitions which will be used throughout the document. Section 3 provides a specification of the congestion control algorithms. Section 4 outlines concerns related to the congestion control algorithms and finally, section 5 outlines security considerations.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [Bra97].

2. Definitions

This section provides the definition of several terms that will be used throughout the remainder of this document.

SEGMENT:

A segment is ANY TCP/IP data or acknowledgment packet (or both).

SENDER MAXIMUM SEGMENT SIZE (SMSS): The SMSS is the size of the largest segment that the sender can transmit. This value can be based on the maximum transmission unit of the network, the path MTU discovery [MD90] algorithm, RMSS (see next item), or other factors. The size does not include the TCP/IP headers and options.

RECEIVER MAXIMUM SEGMENT SIZE (RMSS): The RMSS is the size of the largest segment the receiver is willing to accept. This is the value specified in the MSS option sent by the receiver during connection startup. Or, if the MSS option is not used, 536 bytes [Bra89]. The size does not include the TCP/IP headers and options.

FULL-SIZED SEGMENT: A segment that contains the maximum number of data bytes permitted (i.e., a segment containing SMSS bytes of data).

RECEIVER WINDOW (rwnd) The most recently advertised receiver window.

CONGESTION WINDOW (cwnd): A TCP state variable that limits the amount of data a TCP can send. At any given time, a TCP MUST NOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and the minimum of cwnd and rwnd.

INITIAL WINDOW (IW): The initial window is the size of the sender's congestion window after the three-way handshake is completed.

LOSS WINDOW (LW): The loss window is the size of the congestion window after a TCP sender detects loss using its retransmission timer.

RESTART WINDOW (RW): The restart window is the size of the congestion window after a TCP restarts transmission after an idle period (if the slow start algorithm is used; see section 4.1 for more discussion).

FLIGHT SIZE: The amount of data that has been sent but not yet acknowledged.

3. Congestion Control Algorithms

This section defines the four congestion control algorithms: slow start, congestion avoidance, fast retransmit and fast recovery, developed in [Jac88] and [Jac90]. In some situations it may be beneficial for a TCP sender to be more conservative than the algorithms allow, however a TCP MUST NOT be more aggressive than the following algorithms allow (that is, MUST NOT send data when the value of cwnd computed by the following algorithms would not allow the data to be sent).

3.1 Slow Start and Congestion Avoidance

The slow start and congestion avoidance algorithms MUST be used by a TCP sender to control the amount of outstanding data being injected into the network. To implement these algorithms, two variables are added to the TCP per-connection state. The congestion window (cwnd) is a sender-side limit on the amount of data the sender can transmit into the network before receiving an acknowledgment (ACK), while the receiver's advertised window (rwnd) is a receiver-side limit on the amount of outstanding data. The minimum of cwnd and rwnd governs data transmission.

Another state variable, the slow start threshold (ssthresh), is used to determine whether the slow start or congestion avoidance algorithm is used to control data transmission, as discussed below.

Beginning transmission into a network with unknown conditions requires TCP to slowly probe the network to determine the available capacity, in order to avoid congesting the network with an inappropriately large burst of data. The slow start algorithm is used for this purpose at the beginning of a transfer, or after repairing loss detected by the retransmission timer.

IW, the initial value of cwnd, MUST be less than or equal to 2*SMSS bytes and MUST NOT be more than 2 segments.

We note that a non-standard, experimental TCP extension allows that a TCP MAY use a larger initial window (IW), as defined in equation 1 [AFP98]:

$$IW = \min (4*SMSS, \max (2*SMSS, 4380 \text{ bytes})) \quad (1)$$

With this extension, a TCP sender MAY use a 3 or 4 segment initial window, provided the combined size of the segments does not exceed 4380 bytes. We do NOT allow this change as part of the standard defined by this document. However, we include discussion of (1) in the remainder of this document as a guideline for those experimenting with the change, rather than conforming to the present standards for TCP congestion control.

The initial value of ssthresh MAY be arbitrarily high (for example, some implementations use the size of the advertised window), but it may be reduced in response to congestion. The slow start algorithm is used when $cwnd < ssthresh$, while the congestion avoidance algorithm is used when $cwnd > ssthresh$. When cwnd and ssthresh are equal the sender may use either slow start or congestion avoidance.

During slow start, a TCP increments cwnd by at most SMSS bytes for each ACK received that acknowledges new data. Slow start ends when cwnd exceeds ssthresh (or, optionally, when it reaches it, as noted above) or when congestion is observed.

During congestion avoidance, cwnd is incremented by 1 full-sized segment per round-trip time (RTT). Congestion avoidance continues until congestion is detected. One formula commonly used to update cwnd during congestion avoidance is given in equation 2:

$$cwnd += SMSS * SMSS / cwnd \quad (2)$$

This adjustment is executed on every incoming non-duplicate ACK. Equation (2) provides an acceptable approximation to the underlying principle of increasing cwnd by 1 full-sized segment per RTT. (Note that for a connection in which the receiver acknowledges every data segment, (2) proves slightly more aggressive than 1 segment per RTT, and for a receiver acknowledging every-other packet, (2) is less aggressive.)

Implementation Note: Since integer arithmetic is usually used in TCP implementations, the formula given in equation 2 can fail to increase cwnd when the congestion window is very large (larger than $SMSS \cdot SMSS$). If the above formula yields 0, the result SHOULD be rounded up to 1 byte.

Implementation Note: older implementations have an additional additive constant on the right-hand side of equation (2). This is incorrect and can actually lead to diminished performance [PAD+98].

Another acceptable way to increase cwnd during congestion avoidance is to count the number of bytes that have been acknowledged by ACKs for new data. (A drawback of this implementation is that it requires maintaining an additional state variable.) When the number of bytes acknowledged reaches cwnd, then cwnd can be incremented by up to SMSS bytes. Note that during congestion avoidance, cwnd MUST NOT be increased by more than the larger of either 1 full-sized segment per RTT, or the value computed using equation 2.

Implementation Note: some implementations maintain cwnd in units of bytes, while others in units of full-sized segments. The latter will find equation (2) difficult to use, and may prefer to use the counting approach discussed in the previous paragraph.

When a TCP sender detects segment loss using the retransmission timer, the value of ssthresh MUST be set to no more than the value given in equation 3:

$$ssthresh = \max (\text{FlightSize} / 2, 2 \cdot SMSS) \quad (3)$$

As discussed above, FlightSize is the amount of outstanding data in the network.

Implementation Note: an easy mistake to make is to simply use cwnd, rather than FlightSize, which in some implementations may incidentally increase well beyond rwnd.

Furthermore, upon a timeout cwnd MUST be set to no more than the loss window, LW, which equals 1 full-sized segment (regardless of the value of IW). Therefore, after retransmitting the dropped segment the TCP sender uses the slow start algorithm to increase the window from 1 full-sized segment to the new value of ssthresh, at which point congestion avoidance again takes over.

3.2 Fast Retransmit/Fast Recovery

A TCP receiver SHOULD send an immediate duplicate ACK when an out-of-order segment arrives. The purpose of this ACK is to inform the sender that a segment was received out-of-order and which sequence number is expected. From the sender's perspective, duplicate ACKs can be caused by a number of network problems. First, they can be caused by dropped segments. In this case, all segments after the dropped segment will trigger duplicate ACKs. Second, duplicate ACKs can be caused by the re-ordering of data segments by the network (not a rare event along some network paths [Pax97]). Finally, duplicate ACKs can be caused by replication of ACK or data segments by the network. In addition, a TCP receiver SHOULD send an immediate ACK when the incoming segment fills in all or part of a gap in the sequence space. This will generate more timely information for a sender recovering from a loss through a retransmission timeout, a fast retransmit, or an experimental loss recovery algorithm, such as NewReno [FH98].

The TCP sender SHOULD use the "fast retransmit" algorithm to detect and repair loss, based on incoming duplicate ACKs. The fast retransmit algorithm uses the arrival of 3 duplicate ACKs (4 identical ACKs without the arrival of any other intervening packets) as an indication that a segment has been lost. After receiving 3 duplicate ACKs, TCP performs a retransmission of what appears to be the missing segment, without waiting for the retransmission timer to expire.

After the fast retransmit algorithm sends what appears to be the missing segment, the "fast recovery" algorithm governs the transmission of new data until a non-duplicate ACK arrives. The reason for not performing slow start is that the receipt of the duplicate ACKs not only indicates that a segment has been lost, but also that segments are most likely leaving the network (although a massive segment duplication by the network can invalidate this conclusion). In other words, since the receiver can only generate a duplicate ACK when a segment has arrived, that segment has left the network and is in the receiver's buffer, so we know it is no longer consuming network resources. Furthermore, since the ACK "clock" [Jac88] is preserved, the TCP sender can continue to transmit new segments (although transmission must continue using a reduced cwnd).

The fast retransmit and fast recovery algorithms are usually implemented together as follows.

1. When the third duplicate ACK is received, set ssthresh to no more than the value given in equation 3.

2. Retransmit the lost segment and set `cwnd` to `ssthresh` plus $3 \times \text{SMSS}$. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.
3. For each additional duplicate ACK received, increment `cwnd` by `SMSS`. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.
4. Transmit a segment, if allowed by the new value of `cwnd` and the receiver's advertised window.
5. When the next ACK arrives that acknowledges new data, set `cwnd` to `ssthresh` (the value set in step 1). This is termed "deflating" the window.

This ACK should be the acknowledgment elicited by the retransmission from step 1, one RTT after the retransmission (though it may arrive sooner in the presence of significant out-of-order delivery of data segments at the receiver). Additionally, this ACK should acknowledge all the intermediate segments sent between the lost segment and the receipt of the third duplicate ACK, if none of these were lost.

Note: This algorithm is known to generally not recover very efficiently from multiple losses in a single flight of packets [FF96]. One proposed set of modifications to address this problem can be found in [FH98].

4. Additional Considerations

4.1 Re-starting Idle Connections

A known problem with the TCP congestion control algorithms described above is that they allow a potentially inappropriate burst of traffic to be transmitted after TCP has been idle for a relatively long period of time. After an idle period, TCP cannot use the ACK clock to strobe new segments into the network, as all the ACKs have drained from the network. Therefore, as specified above, TCP can potentially send a `cwnd`-size line-rate burst into the network after an idle period.

[Jac88] recommends that a TCP use slow start to restart transmission after a relatively long idle period. Slow start serves to restart the ACK clock, just as it does at the beginning of a transfer. This mechanism has been widely deployed in the following manner. When TCP has not received a segment for more than one retransmission timeout, `cwnd` is reduced to the value of the restart window (RW) before

transmission begins.

For the purposes of this standard, we define $RW = IW$.

We note that the non-standard experimental extension to TCP defined in [AFP98] defines $RW = \min(IW, cwnd)$, with the definition of IW adjusted per equation (1) above.

Using the last time a segment was received to determine whether or not to decrease $cwnd$ fails to deflate $cwnd$ in the common case of persistent HTTP connections [HTH98]. In this case, a WWW server receives a request before transmitting data to the WWW browser. The reception of the request makes the test for an idle connection fail, and allows the TCP to begin transmission with a possibly inappropriately large $cwnd$.

Therefore, a TCP SHOULD set $cwnd$ to no more than RW before beginning transmission if the TCP has not sent data in an interval exceeding the retransmission timeout.

4.2 Generating Acknowledgments

The delayed ACK algorithm specified in [Bra89] SHOULD be used by a TCP receiver. When used, a TCP receiver MUST NOT excessively delay acknowledgments. Specifically, an ACK SHOULD be generated for at least every second full-sized segment, and MUST be generated within 500 ms of the arrival of the first unacknowledged packet.

The requirement that an ACK "SHOULD" be generated for at least every second full-sized segment is listed in [Bra89] in one place as a SHOULD and another as a MUST. Here we unambiguously state it is a SHOULD. We also emphasize that this is a SHOULD, meaning that an implementor should indeed only deviate from this requirement after careful consideration of the implications. See the discussion of "Stretch ACK violation" in [PAD+98] and the references therein for a discussion of the possible performance problems with generating ACKs less frequently than every second full-sized segment.

In some cases, the sender and receiver may not agree on what constitutes a full-sized segment. An implementation is deemed to comply with this requirement if it sends at least one acknowledgment every time it receives $2 \cdot RMSS$ bytes of new data from the sender, where $RMSS$ is the Maximum Segment Size specified by the receiver to the sender (or the default value of 536 bytes, per [Bra89], if the receiver does not specify an MSS option during connection establishment). The sender may be forced to use a segment size less than $RMSS$ due to the maximum transmission unit (MTU), the path MTU discovery algorithm or other factors. For instance, consider the

case when the receiver announces an RMSS of X bytes but the sender ends up using a segment size of Y bytes ($Y < X$) due to path MTU discovery (or the sender's MTU size). The receiver will generate stretch ACKs if it waits for $2 \cdot X$ bytes to arrive before an ACK is sent. Clearly this will take more than 2 segments of size Y bytes. Therefore, while a specific algorithm is not defined, it is desirable for receivers to attempt to prevent this situation, for example by acknowledging at least every second segment, regardless of size. Finally, we repeat that an ACK MUST NOT be delayed for more than 500 ms waiting on a second full-sized segment to arrive.

Out-of-order data segments SHOULD be acknowledged immediately, in order to accelerate loss recovery. To trigger the fast retransmit algorithm, the receiver SHOULD send an immediate duplicate ACK when it receives a data segment above a gap in the sequence space. To provide feedback to senders recovering from losses, the receiver SHOULD send an immediate ACK when it receives a data segment that fills in all or part of a gap in the sequence space.

A TCP receiver MUST NOT generate more than one ACK for every incoming segment, other than to update the offered window as the receiving application consumes new data [page 42, Pos81][Cla82].

4.3 Loss Recovery Mechanisms

A number of loss recovery algorithms that augment fast retransmit and fast recovery have been suggested by TCP researchers. While some of these algorithms are based on the TCP selective acknowledgment (SACK) option [MMFR96], such as [FF96,MM96a,MM96b], others do not require SACKs [Hoe96,FF96,FH98]. The non-SACK algorithms use "partial acknowledgments" (ACKs which cover new data, but not all the data outstanding when loss was detected) to trigger retransmissions. While this document does not standardize any of the specific algorithms that may improve fast retransmit/fast recovery, these enhanced algorithms are implicitly allowed, as long as they follow the general principles of the basic four algorithms outlined above.

Therefore, when the first loss in a window of data is detected, ssthresh MUST be set to no more than the value given by equation (3). Second, until all lost segments in the window of data in question are repaired, the number of segments transmitted in each RTT MUST be no more than half the number of outstanding segments when the loss was detected. Finally, after all loss in the given window of segments has been successfully retransmitted, cwnd MUST be set to no more than ssthresh and congestion avoidance MUST be used to further increase cwnd. Loss in two successive windows of data, or the loss of a retransmission, should be taken as two indications of congestion and, therefore, cwnd (and ssthresh) MUST be lowered twice in this case.

The algorithms outlined in [Hoe96,FF96,MM96a,MM6b] follow the principles of the basic four congestion control algorithms outlined in this document.

5. Security Considerations

This document requires a TCP to diminish its sending rate in the presence of retransmission timeouts and the arrival of duplicate acknowledgments. An attacker can therefore impair the performance of a TCP connection by either causing data packets or their acknowledgments to be lost, or by forging excessive duplicate acknowledgments. Causing two congestion control events back-to-back will often cut ssthresh to its minimum value of $2 \cdot \text{SMSS}$, causing the connection to immediately enter the slower-performing congestion avoidance phase.

The Internet to a considerable degree relies on the correct implementation of these algorithms in order to preserve network stability and avoid congestion collapse. An attacker could cause TCP endpoints to respond more aggressively in the face of congestion by forging excessive duplicate acknowledgments or excessive acknowledgments for new data. Conceivably, such an attack could drive a portion of the network into congestion collapse.

6. Changes Relative to RFC 2001

This document has been extensively rewritten editorially and it is not feasible to itemize the list of changes between the two documents. The intention of this document is not to change any of the recommendations given in RFC 2001, but to further clarify cases that were not discussed in detail in 2001. Specifically, this document suggests what TCP connections should do after a relatively long idle period, as well as specifying and clarifying some of the issues pertaining to TCP ACK generation. Finally, the allowable upper bound for the initial congestion window has also been raised from one to two segments.

Acknowledgments

The four algorithms that are described were developed by Van Jacobson.

Some of the text from this document is taken from "TCP/IP Illustrated, Volume 1: The Protocols" by W. Richard Stevens (Addison-Wesley, 1994) and "TCP/IP Illustrated, Volume 2: The Implementation" by Gary R. Wright and W. Richard Stevens (Addison-Wesley, 1995). This material is used with the permission of Addison-Wesley.

Neal Cardwell, Sally Floyd, Craig Partridge and Joe Touch contributed a number of helpful suggestions.

References

- [AFP98] Allman, M., Floyd, S. and C. Partridge, "Increasing TCP's Initial Window Size, RFC 2414, September 1998.
- [Bra89] Braden, R., "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1122, October 1989.
- [Bra97] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [Cla82] Clark, D., "Window and Acknowledgment Strategy in TCP", RFC 813, July 1982.
- [FF96] Fall, K. and S. Floyd, "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", Computer Communication Review, July 1996. <ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z>.
- [FH98] Floyd, S. and T. Henderson, "The NewReno Modification to TCP's Fast Recovery Algorithm", RFC 2582, April 1999.
- [Flo94] Floyd, S., "TCP and Successive Fast Retransmits. Technical report", October 1994. <ftp://ftp.ee.lbl.gov/papers/fastretrans.ps>.
- [Hoe96] Hoe, J., "Improving the Start-up Behavior of a Congestion Control Scheme for TCP", In ACM SIGCOMM, August 1996.
- [HTH98] Hughes, A., Touch, J. and J. Heidemann, "Issues in TCP Slow-Start Restart After Idle", Work in Progress.
- [Jac88] Jacobson, V., "Congestion Avoidance and Control", Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988. <ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z>.
- [Jac90] Jacobson, V., "Modified TCP Congestion Avoidance Algorithm", end2end-interest mailing list, April 30, 1990. <ftp://ftp.isi.edu/end2end/end2end-interest-1990.mail>.
- [MD90] Mogul, J. and S. Deering, "Path MTU Discovery", RFC 1191, November 1990.

- [MM96a] Mathis, M. and J. Mahdavi, "Forward Acknowledgment: Refining TCP Congestion Control", Proceedings of SIGCOMM'96, August, 1996, Stanford, CA. Available from <http://www.psc.edu/networking/papers/papers.html>
- [MM96b] Mathis, M. and J. Mahdavi, "TCP Rate-Halving with Bounding Parameters", Technical report. Available from <http://www.psc.edu/networking/papers/FAcknotes/current>.
- [MMFR96] Mathis, M., Mahdavi, J., Floyd, S. and A. Romanow, "TCP Selective Acknowledgement Options", RFC 2018, October 1996.
- [PAD+98] Paxson, V., Allman, M., Dawson, S., Fenner, W., Griner, J., Heavens, I., Lahey, K., Semke, J. and B. Volz, "Known TCP Implementation Problems", RFC 2525, March 1999.
- [Pax97] Paxson, V., "End-to-End Internet Packet Dynamics", Proceedings of SIGCOMM '97, Cannes, France, Sep. 1997.
- [Pos81] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [Ste94] Stevens, W., "TCP/IP Illustrated, Volume 1: The Protocols", Addison-Wesley, 1994.
- [Ste97] Stevens, W., "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC 2001, January 1997.
- [WS95] Wright, G. and W. Stevens, "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley, 1995.

Authors' Addresses

Mark Allman
NASA Glenn Research Center/Sterling Software
Lewis Field
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135
216-433-6586

EMail: mallman@grc.nasa.gov
<http://roland.grc.nasa.gov/~mallman>

Vern Paxson
ACIRI / ICSI
1947 Center Street
Suite 600
Berkeley, CA 94704-1198

Phone: +1 510/642-4274 x302
EMail: vern@aciri.org

W. Richard Stevens
1202 E. Paseo del Zorro
Tucson, AZ 85718
520-297-9416

EMail: rstevens@kohala.com
<http://www.kohala.com/~rstevens>

Full Copyright Statement

Copyright (C) The Internet Society (1999). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Network Working Group
Request for Comments: 2988
Category: Standards Track

V. Paxson
ACIRI
M. Allman
NASA GRC/BBN
November 2000

Computing TCP's Retransmission Timer

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Copyright (C) The Internet Society (2000). All Rights Reserved.

Abstract

This document defines the standard algorithm that Transmission Control Protocol (TCP) senders are required to use to compute and manage their retransmission timer. It expands on the discussion in section 4.2.3.1 of RFC 1122 and upgrades the requirement of supporting the algorithm from a SHOULD to a MUST.

1 Introduction

The Transmission Control Protocol (TCP) [Pos81] uses a retransmission timer to ensure data delivery in the absence of any feedback from the remote data receiver. The duration of this timer is referred to as RTO (retransmission timeout). RFC 1122 [Bra89] specifies that the RTO should be calculated as outlined in [Jac88].

This document codifies the algorithm for setting the RTO. In addition, this document expands on the discussion in section 4.2.3.1 of RFC 1122 and upgrades the requirement of supporting the algorithm from a SHOULD to a MUST. RFC 2581 [APS99] outlines the algorithm TCP uses to begin sending after the RTO expires and a retransmission is sent. This document does not alter the behavior outlined in RFC 2581 [APS99].

In some situations it may be beneficial for a TCP sender to be more conservative than the algorithms detailed in this document allow. However, a TCP MUST NOT be more aggressive than the following algorithms allow.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [Bra97].

2 The Basic Algorithm

To compute the current RTO, a TCP sender maintains two state variables, SRTT (smoothed round-trip time) and RTTVAR (round-trip time variation). In addition, we assume a clock granularity of G seconds.

The rules governing the computation of SRTT, RTTVAR, and RTO are as follows:

- (2.1) Until a round-trip time (RTT) measurement has been made for a segment sent between the sender and receiver, the sender SHOULD set $RTO \leftarrow 3 \text{ seconds}$ (per RFC 1122 [Bra89]), though the "backing off" on repeated retransmission discussed in (5.5) still applies.

Note that some implementations may use a "heartbeat" timer that in fact yield a value between 2.5 seconds and 3 seconds. Accordingly, a lower bound of 2.5 seconds is also acceptable, providing that the timer will never expire faster than 2.5 seconds. Implementations using a heartbeat timer with a granularity of G SHOULD not set the timer below $2.5 + G$ seconds.

- (2.2) When the first RTT measurement R is made, the host MUST set

```
SRTT <- R
RTTVAR <- R/2
RTO <- SRTT + max (G, K*RTTVAR)
```

where $K = 4$.

- (2.3) When a subsequent RTT measurement R' is made, a host MUST set

```
RTTVAR <- (1 - beta) * RTTVAR + beta * |SRTT - R'|
SRTT <- (1 - alpha) * SRTT + alpha * R'
```

The value of SRTT used in the update to RTTVAR is its value before updating SRTT itself using the second assignment. That is, updating RTTVAR and SRTT MUST be computed in the above order.

The above SHOULD be computed using $\alpha=1/8$ and $\beta=1/4$ (as suggested in [JK88]).

After the computation, a host MUST update
 $RTO \leftarrow SRTT + \max(G, K \cdot RTTVAR)$

- (2.4) Whenever RTO is computed, if it is less than 1 second then the RTO SHOULD be rounded up to 1 second.

Traditionally, TCP implementations use coarse grain clocks to measure the RTT and trigger the RTO, which imposes a large minimum value on the RTO. Research suggests that a large minimum RTO is needed to keep TCP conservative and avoid spurious retransmissions [AP99]. Therefore, this specification requires a large minimum RTO as a conservative approach, while at the same time acknowledging that at some future point, research may show that a smaller minimum RTO is acceptable or superior.

- (2.5) A maximum value MAY be placed on RTO provided it is at least 60 seconds.

3 Taking RTT Samples

TCP MUST use Karn's algorithm [KP87] for taking RTT samples. That is, RTT samples MUST NOT be made using segments that were retransmitted (and thus for which it is ambiguous whether the reply was for the first instance of the packet or a later instance). The only case when TCP can safely take RTT samples from retransmitted segments is when the TCP timestamp option [JBB92] is employed, since the timestamp option removes the ambiguity regarding which instance of the data segment triggered the acknowledgment.

Traditionally, TCP implementations have taken one RTT measurement at a time (typically once per RTT). However, when using the timestamp option, each ACK can be used as an RTT sample. RFC 1323 [JBB92] suggests that TCP connections utilizing large congestion windows should take many RTT samples per window of data to avoid aliasing effects in the estimated RTT. A TCP implementation MUST take at least one RTT measurement per RTT (unless that is not possible per Karn's algorithm).

For fairly modest congestion window sizes research suggests that timing each segment does not lead to a better RTT estimator [AP99]. Additionally, when multiple samples are taken per RTT the alpha and beta defined in section 2 may keep an inadequate RTT history. A method for changing these constants is currently an open research question.

4 Clock Granularity

There is no requirement for the clock granularity G used for computing RTT measurements and the different state variables. However, if the $K \cdot \text{RTTVAR}$ term in the RTO calculation equals zero, the variance term **MUST** be rounded to G seconds (i.e., use the equation given in step 2.3).

$$\text{RTO} \leftarrow \text{SRTT} + \max(G, K \cdot \text{RTTVAR})$$

Experience has shown that finer clock granularities (≤ 100 msec) perform somewhat better than more coarse granularities.

Note that [Jac88] outlines several clever tricks that can be used to obtain better precision from coarse granularity timers. These changes are widely implemented in current TCP implementations.

5 Managing the RTO Timer

An implementation **MUST** manage the retransmission timer(s) in such a way that a segment is never retransmitted too early, i.e. less than one RTO after the previous transmission of that segment.

The following is the **RECOMMENDED** algorithm for managing the retransmission timer:

- (5.1) Every time a packet containing data is sent (including a retransmission), if the timer is not running, start it running so that it will expire after RTO seconds (for the current value of RTO).
- (5.2) When all outstanding data has been acknowledged, turn off the retransmission timer.
- (5.3) When an ACK is received that acknowledges new data, restart the retransmission timer so that it will expire after RTO seconds (for the current value of RTO).

When the retransmission timer expires, do the following:

- (5.4) Retransmit the earliest segment that has not been acknowledged by the TCP receiver.
- (5.5) The host MUST set $RTO \leftarrow RTO * 2$ ("back off the timer"). The maximum value discussed in (2.5) above may be used to provide an upper bound to this doubling operation.
- (5.6) Start the retransmission timer, such that it expires after RTO seconds (for the value of RTO after the doubling operation outlined in 5.5).

Note that after retransmitting, once a new RTT measurement is obtained (which can only happen when new data has been sent and acknowledged), the computations outlined in section 2 are performed, including the computation of RTO, which may result in "collapsing" RTO back down after it has been subject to exponential backoff (rule 5.5).

Note that a TCP implementation MAY clear SRTT and RTTVAR after backing off the timer multiple times as it is likely that the current SRTT and RTTVAR are bogus in this situation. Once SRTT and RTTVAR are cleared they should be initialized with the next RTT sample taken per (2.2) rather than using (2.3).

6 Security Considerations

This document requires a TCP to wait for a given interval before retransmitting an unacknowledged segment. An attacker could cause a TCP sender to compute a large value of RTO by adding delay to a timed packet's latency, or that of its acknowledgment. However, the ability to add delay to a packet's latency often coincides with the ability to cause the packet to be lost, so it is difficult to see what an attacker might gain from such an attack that could cause more damage than simply discarding some of the TCP connection's packets.

The Internet to a considerable degree relies on the correct implementation of the RTO algorithm (as well as those described in RFC 2581) in order to preserve network stability and avoid congestion collapse. An attacker could cause TCP endpoints to respond more aggressively in the face of congestion by forging acknowledgments for segments before the receiver has actually received the data, thus lowering RTO to an unsafe value. But to do so requires spoofing the acknowledgments correctly, which is difficult unless the attacker can monitor traffic along the path between the sender and the receiver. In addition, even if the

attacker can cause the sender's RTO to reach too small a value, it appears the attacker cannot leverage this into much of an attack (compared to the other damage they can do if they can spoof packets belonging to the connection), since the sending TCP will still back off its timer in the face of an incorrectly transmitted packet's loss due to actual congestion.

Acknowledgments

The RTO algorithm described in this memo was originated by Van Jacobson in [Jac88].

References

- [AP99] Allman, M. and V. Paxson, "On Estimating End-to-End Network Path Properties", SIGCOMM 99.
- [APS99] Allman, M., Paxson V. and W. Stevens, "TCP Congestion Control", RFC 2581, April 1999.
- [Bra89] Braden, R., "Requirements for Internet Hosts -- Communication Layers", STD 3, RFC 1122, October 1989.
- [Bra97] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [Jac88] Jacobson, V., "Congestion Avoidance and Control", Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988.
- [JK88] Jacobson, V. and M. Karels, "Congestion Avoidance and Control", <ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z>.
- [KP87] Karn, P. and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols", SIGCOMM 87.
- [Pos81] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

Author's Addresses

Vern Paxson
ACIRI / ICSI
1947 Center Street
Suite 600
Berkeley, CA 94704-1198

Phone: 510-666-2882
Fax: 510-643-7684
EMail: vern@aciri.org
<http://www.aciri.org/vern/>

Mark Allman
NASA Glenn Research Center/BBN Technologies
Lewis Field
21000 Brookpark Rd. MS 54-2
Cleveland, OH 44135

Phone: 216-433-6586
Fax: 216-433-8705
EMail: mallman@grc.nasa.gov
<http://roland.grc.nasa.gov/~mallman>

Full Copyright Statement

Copyright (C) The Internet Society (2000). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

