Issues in the implementation of selective acknowledgements for TCP

Luigi Rizzo
Dipartimento di Ingegneria dell'Informazione, Università di Pisa
via Diotisalvi 2 – 56126 Pisa (Italy)
email: `l.rizzo@iet.unipi.it`

DRAFT – January 24, 1996
Please get the updated version from `http://www.iet.unipi.it/~luigi/selack.ps`

Abstract

Researchers are investigating on the definition and the effectiveness of selective acknowledgement (SACK) options for TCP, as it is expected that a SACK mechanism, together with a suitable congestion control algorithm, can improve TCP performance over lossy networks.

Due to the widely variable traffic and congestion conditions on the Internet, we believe that work on the above subjects should be supported by extensive experiments in the field, in addition to theoretical analysis and trace-based simulations. We also believe that the design of new mechanisms, such as SACKs, should be done with special attention to their effectiveness when interoperating with older implementations.

In this paper we give two contributions in this direction. First, we show how the sender node can process incoming SACKs with a small, constant overhead and memory usage with respect to non-SACK implementations. Second, we show in detail how to transport SACK information in modified RFC1323 timestamps, which we call TSACKs. TSACKs are of trivial implementation, but allow the exploitation of the advantages of SACKs when sending data to RFC1323-compliant nodes. The patterns of traffic on the Internet (where relatively few, large FTP/WWW servers probably account for the majority of bytes sent), make this a very important feature.

An implementation of the two algorithms above is available from the author. Developed on FreeBSD, a 4.4BSD derivative, the code is compatible with the current IETF draft on the encoding of SACK options.

1 Introduction

The acknowledgement mechanism of TCP is not well suited to the use on lossy networks, as it does not allow a node to acknowledge data received out of sequence. This might cause unnecessary retransmissions of data segments, or, due to the congestion control mechanisms that are necessary for the Internet to operate, unnecessary delays in useful retransmissions.

Segment losses are particularly harmful on high bandwidth, long delay paths, because of the amount of outstanding data that must be sent before being able to detect a loss. Networks with such features are becoming more and more widespread nowadays. But even other networks suffer from losses. The Internet (and its traffic) has grown dramatically in recent years, because of the availability of information servers such as the Web, and the consequential interest of the commercial and home-users' world. Overloaded routers or congested links are more and more common these days, despite the congestion control algorithms currently implemented in TCP. Also, the availability of powerful, low cost, portable equipment and of a distributed communication infrastructure (cellular phones) makes it more and more common to use TCP over wireless (radio or infrared) channels. In the latter two situations there often exist relevant losses, and both the acknowledgement and the current congestion control algorithms of TCP do not give an adequate response.

There is some consensus on the fact that selective acknowledgements (SACKs) can improve TCP performance over lossy networks [6, 7]. SACKs have been proposed for TCP [1, 10] and other protocols [3, 4, 11, 12] to report to the sender which segments have been delivered and which have not, thus avoiding unnecessary retransmissions. If matched with suitably large windows, and a proper congestion control algorithm and retransmission strategy, SACKs also help in reducing the unavoidable delays caused by lost segments, thus improving the performance of the communication.

An effective exploitation of the information supplied by SACKs will probably require some changes to the congestion control algorithms currently in use. Due to the deep modifications that the Internet has incurred in recent years, and the variety of different operating environments in which TCP is used, we believe that work on SACKs and the related mechanisms should be supported by extensive experiments in the field, in addition to theoretical analysis and trace-based simulations. This is especially important in those cases where the loss rate is greater than a few percent, and the latter two techniques may become unpractical.

When introducing a new mechanism in a very large system such as the Internet, great care must be taken to its effectiveness when interoperating with old implementations (perhaps sacrificing something in terms of performance in these cases). Otherwise, the long time required to upgrade the majority of systems might render the new mechanism ineffective.

In this paper we give two contributions to hopefully shorten the time required to have a working SACK mechanism on the Internet.

First, we analyze the sender-side processing of SACK information, and show how this can be done efficiently, and without overheads in the case of reliable communications. A possible implementation is also shown, with reference to the TCP code in BSD Unix. Our proposal (and the code we have developed) is compatible with both the SACK option described in [10], and other implementations of SACKs.

To enable experiments with SACKs while the standard for SACK options is formally defined, and, especially, to exploit SACKs when interoperating with old implementations of TCP, we suggest to embed SACK information into RFC1323 timestamps [9], which are already supported by several operating systems. We show in detail how RFC1323 timestamps can be modified to carry SACK-like information, which we call TSACK. This overloading of the meaning of timestamps is fully compatible with RFC1323 timestamps.

We believe that designing SACKs with this "backward compatibility" in mind can have very positive effects for the Internet, as it gives more nodes a chance to exploit the benefits of SACKs. Given the patterns of traffic on the Internet (where relatively few, large FTP/WWW servers probably account for the majority of bytes sent), this could also significantly reduce the time needed for a large scale exploitation of selective acknowledgements.

In addition to processing and generating SACK information, a complete SACK mechanism also has to implement a retransmission strategy and interact with congestion control mechanisms [8, 2]. In particular, the latter should not impose too restrictive limitations on the window size in presence of segment losses. The reader is referred to [8, 2, 6, 7, 5] for a discussion of this important topic. Here we only discuss briefly some retransmission strategies that we are currently investigating. We hope that the simple algorithms described in this paper should allow researchers to make significant experience on congestion control algorithms and retransmission strategies.

The code which implements SACKs and TSACKs as described in this paper is being developed on FreeBSD, a 4.4 BSD derivative, and is available from the author. The code should be easily portable to other BSD derivatives, provided the the `struct tcpcb` can accomodate the additional fields required by our implementation.

It has been pointed out to the author that the idea of using timestamps to carry additional information is being considered by other researchers[1], although no specific reference or implementation is known to the author at the time of this writing.

2 Sender side support for SACKs

In the following we call ACKed those blocks of data for which a TCP ACK has been received, SACKed those blocks of data for which a TCP ACK has not been received yet, but there is knowledge that the data have been received at the remote side. We will call *action* any action or processing which is performed *exclusively* to support SACKs.

Implementing SACKs requires the sender TCP to keep track of those blocks of data which have been successfully received at the other end of the connection. The BSD networking code has no provision for holding this information, at least in the network layer: unacknowledged outgoing data are held in the socket layer.

SACK information are usually built at the receiver side, and sent back to the sender, to give a possibly accurate picture of the status of the reassembly queue. The exact operation can be constrained by external factors such as the amount of space available in TCP options, the encoding of SACK information, or the amount of processing and storage that can be dedicated to the support of SACKs. Some of the above subjects are discussed elsewhere [6, 7, 10]. We only remark that this information can be gathered at very little cost while storing incoming data in the reassembly queue. As an extreme case, as we will see in Section 4, no action might be required at the receiver side to implement some form of selective acknowledgements.

The processing of SACK information must be reasonably cheap (in terms of memory usage and computation), so as not to add an excessive overhead in the normal case of very low losses. It must also be tolerant to inefficient behaviours of the other party (such as throwing away segments which are received out of sequence). It does not need to be exact, that is, information can be discarded with no influence on the correctness of the algorithm. This can be done if it pays back in terms of efficiency and speed.

We should also keep in mind that, as mentioned before, the receiver may arbitrarily drop segments or part of them. As a consequence, both ACKs and SACKs might not have the same boundaries as segments (although most implementation are likely to manage segments as a whole). This suggests that the SACK processing code should be able to deal with arbitrary blocks of data, not just entire segments.

[1] see the recent emails on the `tcplw` mailing list, available via `telnet` to port 23000 on `berserkly.cray.com`

Given the above requirements, we suggest the following algorithm for the processing of SACK information at the sender side. As we will show, the algorithm is easy to implement, requires a moderate, constant amount of storage and a introduces a small computational overhead only when segment losses occur.

SACK processing

- Information about SACKed blocks of data is held into a fixed length queue (the SACKed queue), sorted on sequence numbers. The queue is created as empty when the connection is established. The elements of the queue (see Figure 2) are non overlapping and non contiguous blocks of unacknowledged data.

- At the sender side, no action is performed when transmitting a segment. No action is performed when ACKs arrive, either. Any action is deferred until the arrival of SACKs. This means that SACKs are supported at no extra cost in the case of no losses.

- When a SACK arrives, it is inserted in the SACKed queue, sorted by sequence number. In the most common case, when no reordering of segments occurs, this just involves an insertion at the tail, a constant time operation. Otherwise, insertion requires roughly the same amount of pointer management that is done at the receiver to insert new segments into the reassembly queue.

- When doing retransmissions, the sender can skip those blocks of data that have been already SACKed. To take care of the possibility that the receiver drops some out-of-sequence segments after SACKing them, blocks can be removed from the queue after some time.

- There are three mechanisms which allow the SACKed queue to shrink. First, when inserting a new block, all blocks that have been ACKed can be removed. These blocks are located at the head of the queue. Second, new blocks might merge with one or more blocks already present in the queue. Third, blocks might be removed during retransmissions, when the associated information is considered stale.

While the above processing looks time consuming, the SACKed queue does not need to be very long. To show this, we have computed the distribution function of the number of blocks which are present in the reassembly queue, for various window lengths and loss probabilities, in the case of uniform losses (burst losses should yield lower values for the same loss probability). In

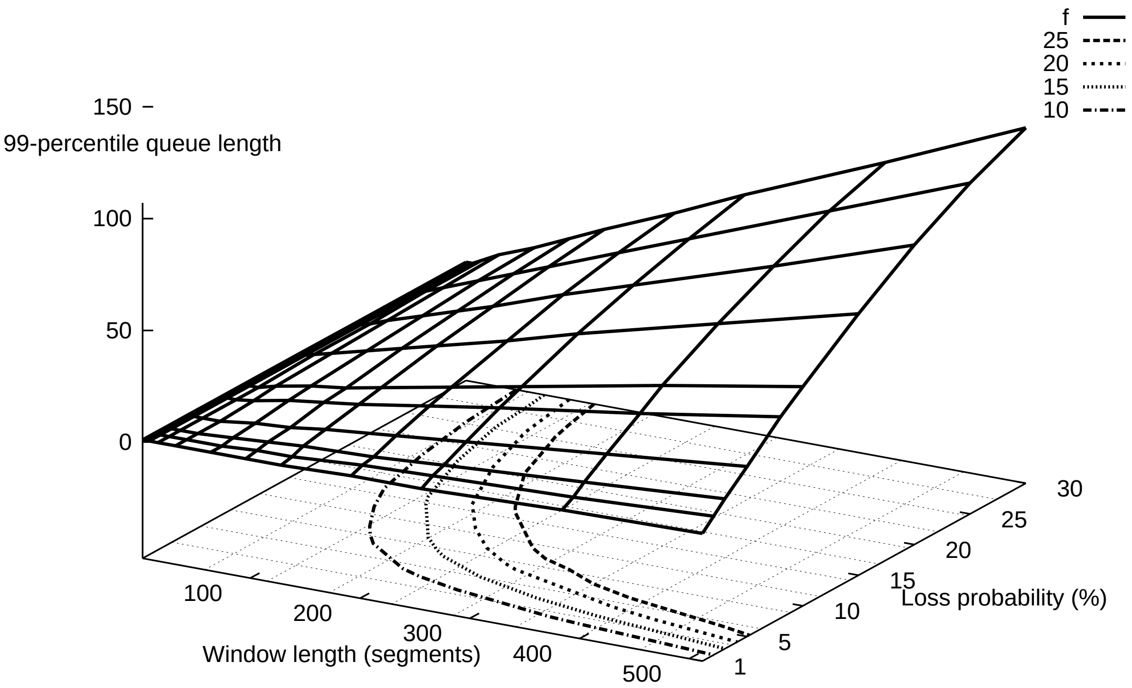

Figure 1: The 99-percentile length of the reassembly queue at the receiver. The contour lines at the bottom delimit the regions where the function is below 10, 15, 20 or 25 respectively.

Figure 1 we plot the 99-percentile length (in blocks) of the reassembly queue: from the contour lines at the bottom it appears clearly that even a modest size (e.g. 10 elements) covers the vast majority of common situations, even for large windows. This allows us to allocate a constant number of entries for the SACKed queue. A constant size also sets an upper bound on the overhead of processing SACKs, with the only possible side effect of causing some unnecessary retransmissions when we run out of space and the retransmission timeout expires.

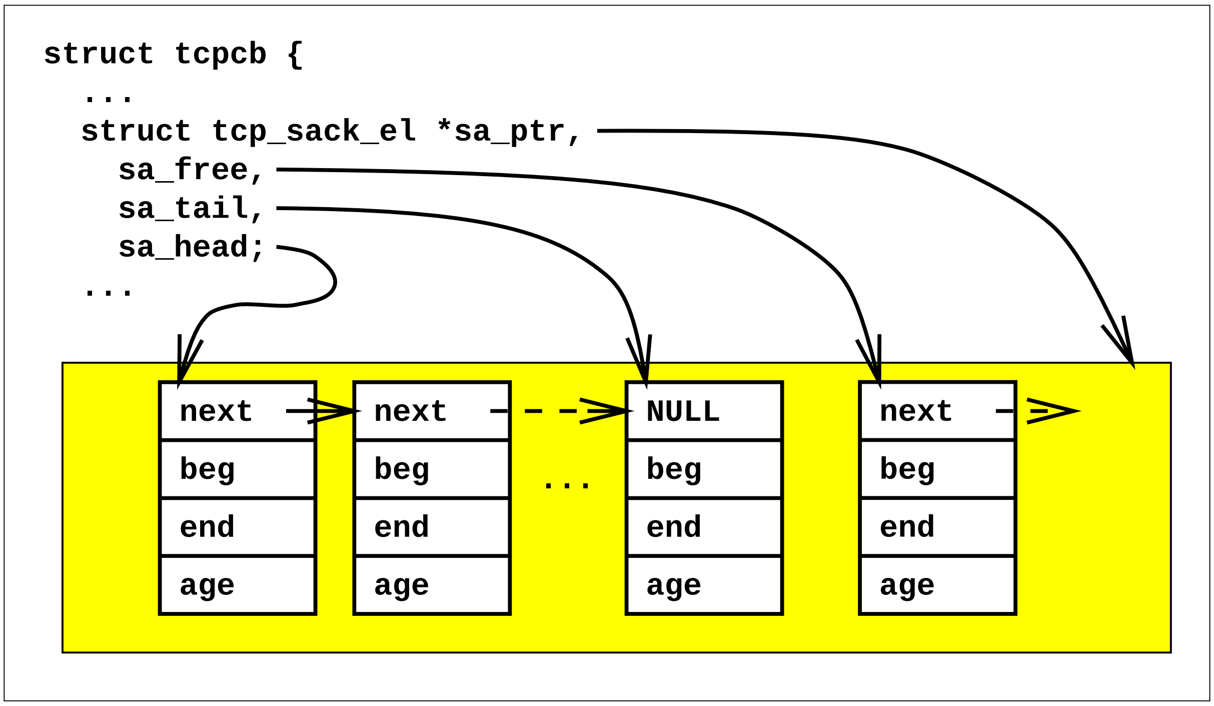

Figure 2: The data structures required to implement SACKs at the sender side

2.1 Implementation

The data structures involved in our implementation of SACKs are shown in Figure 2. A fixed amount of memory is allocated to hold the elements of the SACKed queue. The size is chosen as a reasonable compromise between memory occupation and performance. The elements of the queue have room for the sequence numbers of the boundaries of each block (easier to use than block lengths), and an `age` field which can be used to implement an expiration policy for the information held in the queue.

The TCP control block (`struct tcpcb`) holds four new pointers, to the block of memory containing the elements of the queue, the head and the tail of the queue, and a free list, respectively.

The code to implement the required changes is mostly made of the functions which manage the queue, about 150 lines located in `tcp_subr.c`. They handle the various cases of overlap between SACKed blocks, and also provide fast paths for the most common cases. Other small changes occur in `tcp_var.h` for the definition of the relevant data structures, in `tcp_input.c` to invoke the processing of incoming SACKs, and in other places where the segments to be retransmitted are selected.

3 Retransmissions

In this section we briefly discuss possible strategies on how to schedule retransmissions. We do not indicate a preferred solution as this subject is still under investigation. For the same reason, we do not discuss or suggest modifications to the congestion control algorithms. Congestion control algorithms are described in detail in [8, 2, 13]; their influence on the handling of segment losses is discussed in [5, 6, 7].

In the following discussion we assume that all segments have maximum size (MSS), the round trip time is RTT, the available bandwidth for the connection is BW, and that data are always available at the sender side.

Under steady state conditions, segments arrive at the receiver every $t_s = \frac{MSS}{BW}$ seconds. Lost segments, if followed by successfully received ones, cause the generation of duplicate ACKs.

At the sender side, transmissions to fill up the window are triggered by incoming ACKs, and obey to the principle of "conservation of packets" [8]. The first few duplicate ACKs are recorded, but they not cause any retransmission or new transmission: they might be the result of some reordering of packets in the network, and are not, per se, a sure indication that segments have been lost. The current heuristic is to assume three duplicate acks as an indication of a segment loss, and do an immediate retransmit of the single segment that it is assumed to have been lost. As a consequence, the retransmission of a segment occurs after a minimum time

$$t_r = RTT + t_k + 3t_s$$

where t_k is the amount of time the network has been busy dealing with the k lost segments. t_k can have any value between 0 (if segments are dropped at the sender) and kt_s (if they are dropped at the receiver), or it can be even larger if several retries are done by the lower layers before dropping a segment. For simplicity, let us assume $t_k = 0$.

Provided that the transmission window remains sufficiently large, t_r can be partly or completely amortized because other segments can be sent in the meantime. However, since new transmissions are triggered by incoming ACKs (new or duplicate ones), it is likely that fast retransmissions are done after a delay of $3t_s$ or more from the previous transmission. These pauses add up to the total time for the connection, but for low loss rates the factor of 3 does not change significantly the effective bandwidth.

Unfortunately, things might go worse than this, particularly when the combination of window size and loss rate make it more likely that multiple segments from the same window are lost. As shown in [5, 6, 7] there is a chance that multiple losses in a window require the expiration of a timeout to be detected. Moreover, as the loss rate becomes larger, the chance that a segment is lost more than once becomes non negligible; also, the congestion window might become so small to prevent the generation of the three duplicate acks which should trigger the retransmission.

The expiration of a timeout is a symptom that something unforeseeable has occurred. Since no new information on what happened is available (except that some time has elapsed), the response to timeouts can only be very conservative, and, as such, has a great impact on performance.

SACK information should be used to prevent the occurrence of such situations, and, in particular, to reduce the recourse to timeouts to restart the flow of data. Of course, the usual congestion control principles should be be followed. We intend to evaluate the following strategies:

- still delay retransmissions until three duplicate acks are received, but only if we know that such an event can actually happen. If fewer segments are in transit, we should reduce the threshold to avoid the occurrence of a retransmit timeout.

- during fast retransmits, resend the non-SACKed data before the new ones. Current implementations of TCP can only send the first unacknowledged segment, being unaware if more than one segment has been lost.

- obeying to the principle of conservation of packets, add the amount of SACKed data (and subtract the retransmitted ones) to the size of the congestion window when determining if a segment can be sent. In the current BSD code, the first two duplicate ACKs do not trigger any transmission, even if they are an indication that some packets have actually left the network.

As we have mentioned before, there is a possibility that the receiver drops parts or whole segments after having SACKed them. Such actions can be detected only in a few cases, e.g. when a previously SACKed segment is adjacent (on the right side) to a newer ACKed one. As a consequence, the fact that a segment has been SACKed should not prevent its retransmission at a later time. One possible strategy is to remove segments from the SACKed queue after the preceding lost segments have been retransmitted. Another possibility is to flush the queue after the occurrence of one or more retransmit timeout, or after the SACKed segment has remained in the queue for some time.

We believe that a receiver should drop SACKed segments only in extreme cases, so that we should consider this an exception and tolerate a less efficient behaviour in these cases. Nevertheless, the best strategy is strictly related to the effectiveness of the retransmit strategy: if the latter is too conservative, it might not prevent the occurrence of timeouts and force us to discard valid information. This is especially true in those cases where large losses exist.

4 RFC1323-based SACKs

As we have previously pointed out, it is very useful that new mechanisms can be exploited even when operating with old implementations of TCP. The extent to which this can be done is of course highly variable, and it is quite possible that "backward-compatible" operating modes are somewhat limited with respect to a full implementation. In the case of SACKs, it is possible to convey SACK information in RFC1323 timestamps, although this is limited to *one segment* instead of *one or more arbitrary blocks* of data that can be SACKed by using a dedicated option.

RFC1323 does not pose particular requirements on timestamps, except that they should be a monotonic non-decreasing sequence, and should not be recycled faster than sequence numbers. Timestamps are not subject to any special computation on the receiving side (except the so-called PAWS test to determine if a segment is too old). As a consequence, it is possible to use RFC1323 timestamps to convey additional information other than the timestamp of the received segment. We will call TSACKs these modified timestamps, and show a possible implementation which is simple and efficient.

TSACK algorithm

- On the sender side, for every connection, we use a counter which supplies a unique serial number (`id`) for each sent segments, and a table (`sent[]`, implemented as an array) which holds, for each segment, its serial number, real timestamp, and sequence numbers (see Figure 3).

- Every time a segment is sent, it is tagged a unique `id` which is used as the sender's timestamp in the timestamp option. The `id` modulo the table size is used to select an entry in the `sent[]` table, which is filled with the relevant parameters for the segment being sent. This way of operating requires a small, constant time for each segment sent, and a fixed amount of storage.
- When an RFC1323 receiver produces an ACK, it also copies into the timestamp option one of the timestamps of the received segments (usually the most recent one).
- When the sender receives an ACK, it uses the value in the timestamp option to select an entry in the `sent[]` table. If the received timestamp matches the the `id` value from the selected entry, the other fields from the table entry are used as the timestamp value (`tv`) and the SACK parameters (`se_beg, se_end`) for further processing. The timestamp value is then processed as usual, while the SACK parameters can be processed as described in Section 2.

TSACKs only require modifications to the sender's TCP when talking to RFC1323-compliant receivers. By using a fixed size table, small, constant time operations are required both when sending and when retrieving information from the table.

It is important to realize that the SACK information reconstructed by the sender are possibly incorrect, as the receiver might have dropped part or the whole segment, while keeping the timestamp. We try to infer the most likely situation (the whole segment has been stored into the reassembly queue), but we might fail. The information contained in TSACKs are possibly less reliable than those supplied by dedicated SACK options, and this should be dealt with appropriately (e.g. by using a shorter lifetime for TSACK data).

There is a chance that the received timestamp and the `id` from the selected entry do not match. This occurs when the `sent[]` table has fewer slots than the number of outstanding segments. Unfortunately, the maximum number of outstanding segments is potentially limited only by the window size (i.e. we might have a huge amount of one-byte segments, although this is very unlikely and inefficient). However, in the common case of maximum-sized segments, a table sized on the $\frac{\text{window size}}{\text{MSS}}$ ratio minimizes the chance of overflows.

What are the consequences of an overflow of the `sent[]` table ? Of course we cannot use the received timestamp as a SACK. But this is not a real problem as SACKs can be imprecise without loss of correctness, and, in any case we have no guarantees that TSACKs (as well as regular ACKs) do not get lost in the network. What is more important is that we cannot use these timestamps to update the estimate of the RTT. This might lead to incorrect estimates if the connection has permanently more segments in transit than slots in the `sent[]` table. Other than using an unpractically large table, there are two workarounds to this problem:

- use the pre-RFC1323 timing algorithm as a fallback mechanism;
- only write in the `sent[]` table if the slot contains an already ACKed segment. To this purpose, all the slots are initialized with suitable values at the establishment of the connection. Afterwards, a simple sequence-number comparison is needed to test if a given slot is available.

The second approach is of straightforward implementation and corresponds to the use of the pre-RFC1323 timing algorithm, except that it works for many segments per RTT (equal to the table size) instead of just one.

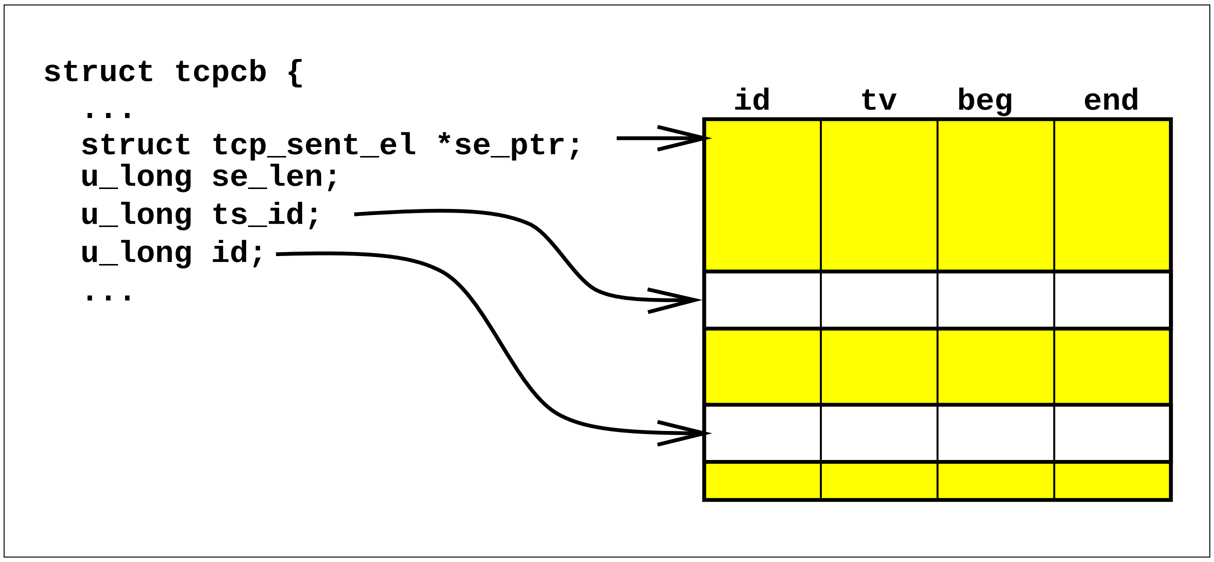

Figure 3: The data structures required to support TSACKs

4.1 Implementation

The data structures for the implementation of TSACKs are shown in Figure 3. The TCP control block is augmented with a counter (`id`), a pointer to the `sent[]` table, the size of the table, and a field to hold a copy of the received timestamp.

The table is allocated with a fixed size at the beginning of the connection, but might be easily resized as the send window changes. Flushing the table during reallocations can be done with little performance penalty, provided the operation is not too frequent. `tcp_subr.c` includes the (very short) routines to initialize the table and convert received TSACKs into timestamps. The conversion function is invoked in `tcp_input.c`, which also passes the SACK values to the functions described in the previous section. Finally, a few lines in `tcp_output.c` are used to build the outgoing TSACKs.

5 Conclusions and future work

We have illustrated how selective acknowledgements can be processed very efficiently at the sender side of TCP, and we have shown in detail how to embed selective acknowledgements into RFC1323 timestamps (TSACKs) with little performance penalties and no changes to the receiver side of TCP.

Some possible changes to the retransmission strategy have also been discussed briefly. The latter are still under investigation, with particular interest to their effectiveness on very lossy networks.

The possibility of exploiting selective acknowledgements with old (RFC1323 compliant) implementations of TCP is of particular interest, as it eases experiments on new congestion control algorithms and retransmission strategies, and also speeds up the diffusion of this new mechanism.

The algorithms described in this paper are being implemented in FreeBSD 2.1, a 4.4 derivative, and are available from the author (although their simplicity and the detailed description supplied certainly enable other researchers to implement them quickly).

References

[1] V. Jacobson, R. Braden, "RFC1185: TCP Extensions for Long-Delay paths", October 1988

[2] L.S.Brakmo, S.W.O'Malley, L.Peterson, "TCP Vegas: New Techniques for Congestion Detection and Avoidance", *Proceedings of SIGCOMM'94 Conference*, pp.24-35, Aug.94

[3] D.Cheriton, "RFC1045: VMTP: Versatile Message Transaction Protocol: Protocol specification", February 1988

[4] D.Clark, M.Lambert, L.Zhang, "RFC998: NETBLT a bulk data transfer protocol", March 1987

[5] S.Floyd, "TCP and Successive Fast Retransmits", Tech. Report, 1994, available via `ftp://ftp.ee.lbl.gov/fastretrans.ps`

[6] K. Fall, S.Floyd, "Comparison of Tahoe, Reno and SACK TCP", Tech. Report, 1995, available via `http://www-nrg.ee.lbl.gov/nrg-papers.html`

[7] S. Floyd, "Issues of TCP with SACK", Tech. Report, 1996, available via `ftp://www-nrg.ee.lbl.gov/nrg-papers.html`

[8] V.Jacobson, "Congestion Avoidance and Control", *Proceedings of SIGCOMM'88* (Standford, CA, Aug.88), ACM

[9] V. Jacobson, R. Braden, D. Borman, "RFC1323: TCP Extensions for High Performance", May 1992

[10] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgement Option", Internet Draft, work in progress, 1995.

[11] C.Partridge, R.Hinden, "RFC1151: Version 2 of the Reliable Data Protocol (RDP)", April 1990

[12] D.Velten, R.Hinden, J.Sax, "RFC908: Reliable Data Protocol", July 1984

[13] Z. Wang, J. Crowcroft, "Eliminating Periodic Packet Losses in the 4.3-Tahoe BSD TCP Congestion Control Algorithm", ACM Computer Communications Review, Apr '92.