TCP retransmissions on very lossy networks

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

January 24, 1996

Abstract

In this paper we study the behaviour of the TCP retransmission code in BSD Unix in presence of large losses, and propose changes to the code, including an adaptive computation of retransmission timeouts, to improve the performance without increasing the congestion on the network. In particular, we point out three features of the TCP retransmission code which have a negative impact on the performance over lossy networks. First, there exist a poorly chosen upper bound on the number of retransmissions, which we show can cause any connection to fail provided it is sufficiently long. Second, retransmission timeouts are almost completely independent of the actual features of the network, and the values used are only suited to very slow networks. We advocate the use of a more adaptive choice of timeouts, and propose the use of bandwidth estimators in the computation of backoff times. Third, a subtle mistake in computing smoothed averages causes large errors in the case of small numbers (very common in the computation of SRTT and RTTVAR). We describe this problem in detail and indicate a correct implementation.

The proposed changes have been tested in FreeBSD 2.X, a 4.4 Lite derivative. Although not a definitive solution, they allows TCP data transfers to proceed even in presence of large losses, do not require extensions to the protocol or major rewrites of the code, do not interfere with the algorithms which control congestion, and, especially, only modify the sender side of TCP. Thus, by simply upgrading existing servers, all clients can benefit the improvements in performance.

1 Introduction

A great deal of work has been done to improve the behaviour of TCP over a wide range of operating conditions, including both low and high speed links. In TCP, flow control is left to the end nodes, and nodes with an anti-social behaviour may disrupt service on a network. As a consequence, many of the algorithms that have been devised and introduced in the implementations of TCP deal with the prevention, avoidance and recovery from congestion in the network. Most of these algorithms are present in the various releases of the BSD Unix Operating System, widely used as the base for several other commercial or experimental implementations (see [14, Chap.1.3]). We will refer here to the so-called “Net/3” release, corresponding to 4.4Lite, but the features we discuss in this paper are also present in older releases of the Berkeley code.

Generally speaking, TCP does not perform well on lossy networks. This is partly due to the structure of the protocol – the presence of a single, positive acknowledgment in the TCP header makes it hard to determine in a timely manner if a segment has been lost. Of course, a specific implementation can improve or worsen this situation. As we will see, the Net/3 implementation does not deal properly with lossy networks, due to some design choices. The fixes proposed in this paper can improve performance and can be incorporated with little effort into the existing code.

Achieving good performance with TCP in presence of large losses is not trivial, because TCP uses a single, cumulative ACK to notify the sender of the successful reception of the data. When one or more segments are lost, especially in presence of large losses, the sender has often to wait for the expiration of a timeout before realizing the loss and being able to retransmit the segment. To aggravate the problem further, segments received after the lost one cannot be acknowledged until the hole has been filled up, and thus they might be retransmitted needlessly.

Some researchers have proposed the introduction of *selective acknowledgements* (SACKs), both in TCP [1] and other protocols [15, 13, 5, 4] to report more exactly which segments have been delivered and which have not, thus avoiding unnecessary retransmissions. Work is currently in progress towards the definition of a SACK extension for TCP (see [12, 6, 7]).

One difficulty with SACKs is that they are an extension of the protocol, and as such require support on both ends of a connection to be effective. Because of the lack of a widely accepted specification of this option, implementations of TCP usually do not include SACKs at the time of this writing. Also, another important research issue is the interaction of SACKs with the retransmission strategy and the congestion control algorithms, in particular the management of the congestion and transmission windows.

The above problems will not be solved quickly. Thus we have explored alternative solutions that, while suboptimal, do not require protocol extensions and can be effective when interoperating with existing implementations. The solutions we propose can be implemented on just one of the peers of the communication, by introducing non-dangerous changes to the retransmission policy.

More aggressive retransmission and acking policies can yield some performance improvement. They can be implemented without protocol extensions, and on just one of the peers, but their use is strongly discouraged by the need of preventing congestion. Actually, the usual assumption in TCP implementations is that packet losses are always caused by congestion, either at the end nodes or at intermediate gateways. As a consequence, a node realizing that segments have been lost is extremely careful and conservative in doing retransmissions, to avoid unnecessary traffic which would aggravate the existing congestion.

We will adhere to the conservative principles which are used in current TCP implementations, although we argue that loss of packets is not always caused by congestion (as in the case of wireless networks, where undetected – often undetectable – collisions or external interference can cause packet losses[1]). Also, there are cases in which overloaded routers/networks are shared by a very large user base (such as the link between a network provider and the rest of the Internet), so that the loss rate is consistently high, independently (within reasonable limits) of the traffic generated by each single connection.

Conventional congestion control techniques essentially try to limit the number of unsolicited or unacknowledged packets (retransmissions, but also ARP requests and SYN segments), and use exponential backoff in the computation of retransmission timeouts. In the two situations depicted above, these techniques (especially exponential backoff) have little influence on the congestion level. Moreover, the computation of retransmission timeouts is completely independent of the actual bandwidth available on the network. This forces the use of very conservative values which are unnecessarily large on most of today's networks.

The main contribution of this paper is the proposal of an adaptive computation of retransmission timeouts, tuned on the bandwidth of the network. This keeps the congestion on the network within reasonably low levels, yet greatly improving the performance especially in presence of lossy, but reasonably fast, networks. In addition, we point out two aspects of the implementation of the retransmission policy (namely the use of a limited number of retries and some algorithms in the computation of RTTs) whose effects are probably underestimated.

The work documented in this paper originates from the following scenario: the link between

[1] although these losses should be dealt with at a lower level in the protocol stack.

our site and the US has been very unreliable during 1995, with packet losses consistently ranging from 10 to 70%, as measured by pinging the remote hosts. Despite such large losses, a bandwidth in the 10KB/s range was available, if only TCP were able to exploit it.

Investigation on the phenomenon showed that these losses were almost completely independent from the traffic generated by our TCP connection, a clear symptom that the connection goes through one or more overloaded routers/links which are shared with many users. We know that several other Internet sites suffer from a similar problem. We believe that similar problems will be experienced by more and more users in the future, as the number of Internet providers is increasing and their connections with the rest of the Internet are not always adequate.

In such a situation, one has two choices: either give up and wait for better times to use the network[2], or accept the limitations and try to exploit the available bandwidth. Unfortunately, the impact on users of such losses is worse than one could expect: ftp sessions, even for files of moderate size, would almost invariably fail at some point or another. Shorter transactions, e.g. http connections, would randomly succeed. Besides being very disappointing, all this was quite surprising and contrasting with the common (among users) belief that “TCP is robust”. We thus studied the Net/3 sources trying to understand this behaviour, and identify the strategies that could help in keeping TCP connections alive in presence of large losses, to whatever reason they depend upon. Our investigations showed that a few, simple modifications to the retransmission and backoff policies of the sender’s TCP can serve to the purpose.

In the next Section we give some definitions and point out some deficiencies in the Net/3 code. Section 3 shows the solutions that we adopted to improve TCP performance over lossy networks, and discusses their effectiveness compared to the current code.

2 Problems in the Net/3 TCP code

In the following we assume that the reader is familiar with the Net/3 code. A detailed description of the code can be found in [14]. Net/3 is not exempt from bugs, some of which are pointed out in [2]. Also, Stevens [14, App.C] cites some weak points in the Net/3 code with reference to the requirements specified in RFC1122[8].

Throughout this paper, we call the unit of transmission for a TCP connection a *segment*. Segments are acknowledged by the receiver, so that the sender can determine the round trip time (RTT) of a connection by timing the interval between the transmission of a segment and the reception of the corresponding ACK.

[2]this includes changing provider or upgrading the bottlenecks; but this is not always feasible.

The RTT can be thought of as made of three parts. The first one, τ_1, approximately constant, depends on the (minimum) response time of all the nodes on the path between the sender and the receiver, and the time for the signals to physically traverse the network (this can be large, as in the case of satellite links). The second part, τ_2, depends on the length of the segment and on the speed of the various links in the connection; it can be considered constant because most segments tend to have constant size equal to the maximum allowed on the connection. The third part, τ_3, depends on the time that a segment (and the ACK) spends queued at intermediate nodes waiting to be transmitted: this part is highly variable and dependent on the actual load on the network.

Net/3 models the RTT with two parameters. The first one, called SRTT, is a smoothed mean of the RTT computed through a lowpass filter. In practice, SRTT includes τ_1, τ_2 (which, for the way SRTT is computed, is indistinguishable from τ_1), and that part of τ_3 corresponding to the average congestion at the nodes. The second parameter, RTTVAR, is an approximation of the variance of the RTT. It represents the variant part of τ_3.

Segments (or ACKs) can be lost, dropped at some node because of excessive congestion, or misrouted to a wrong destination because of configuration problems. We call α the segment loss rate which can be experienced by "pinging" the other end of the connection with a frequency that does not significantly increase the congestion of the network. α can be thought of as a measure of the lossiness of the network, and is, for the way it is measured, independent from the traffic generated by our connection.

To be precise, noting that i) the forward and return paths are not guaranteed to have the same features, ii) often a single ACK serves multiple segments, and iii) losses can occur on both data and ack segments, we should use α_f and α_r to distinguish between the two directions (in the simple case of one ack per segment $\alpha = 1-(1-\alpha_f)(1-\alpha_r) = \alpha_f + \alpha_r - \alpha_f\alpha_r$), and consider the two values separately in our computations. However, given the uncertainty on the exact value of the α, and given that we want to give qualitative results, we will use a single value to summarize the losses on a round trip.

2.1 Number of retransmissions

The expiration of a timeout is one of the events which trigger the retransmission of a segment. The base value for the retransmission timeout, called RTO, is computed from SRTT and RTTVAR as

$$\mathrm{RTO} = \mathrm{SRTT} + 4\mathrm{RTTVAR}$$

The dependency on the variance is there to give segments queued at intermediate nodes enough time to be delivered, instead of being erroneously considered lost[9]. Net/3 places a hard limit of 13 on the total number of transmissions for a segment. This choice is likely to come from a reasoning such as the following:

> For a loss rate of α, each segment needs to be transmitted, on average, $r_A = \frac{1}{1-\alpha}$ times. Even for $\alpha = .5$ (a very lossy network), $r_A = 2$, well below the limit in Net/3. The probability of a segment being successfully transmitted within r (13 in Net/3) retries is $1 - \alpha^r$, reasonably large even with the above numbers.

The mistake lies in the fact that *even a single segment* exceeding the maximum number of retransmissions causes the connection to be dropped. For a connection involving the exchange of n segments, this happens with probability

$$p_d = 1 - (1 - \alpha^r)^n$$

which, being r a constant, can clearly be made arbitrarily close to unity for a sufficiently large n (i.e. for a sufficiently long connection). A few numbers help: for $\alpha = 0.1$, $r = 13$, $n = 8192$ (corresponding to a moderately lossy network, and a connection of 4 MB for a segment size of 512 bytes), we have $p_d < 10^{-9}$. However, things become rapidly worse for larger α: for $\alpha = .5$, we have $p_d = .632$ i.e. the connection is very likely to fail. Increasing α to .7, we see that even a 512-segment transfer will have less than 1% chance of success. Thus,

the number of retransmissions on each segment should not be bounded

(at least, not to a small value). RFC1122, Sec.4.2.3.5 suggests that applications should be able to request an unlimited number of retransmissions, but does not explain the consequences of not doing so. We suggest that retransmissions should continue for *both* a minimum number of attempts *and* a total timeout before giving up. Considering both total time and number of retries is important because makes connection failures more predictable.

There is an additional implementation detail in Net/3 which further limits the number of retransmissions for a single segment: when the ACK for a retransmitted segment is received, in some cases the retry counter is not reset. Thus, two or more segments might compete for the use of the 12 retransmissions, causing connections to break even in presence of moderate losses.

2.2 A note on the use of fixed point arithmetic

Smoothed means of various quantities, such as the RTT and RTTVAR in Net/3, are often computed by passing values through a lowpass filter, as follows:

$$y' = \frac{y(k-1)+x}{k} = y + \frac{x-y}{k}$$

y and y' are the old and new value of the smoothed mean, and x is the value being averaged. The value of k determines the corner frequency of the lowpass filter. The computation is usually done with fixed point arithmetic, in order to avoid the use of floating point operations in the kernel. As a consequence, the stored values y_s are scaled by a factor 2^d ($y_s = 2^d y$), in the hope of having d fractional binary digits. The equation above thus becomes

$$y'_s = y_s + \lfloor \frac{2^d x - y_s}{k} \rfloor$$

The replacement of the division with integer division, and the fact that x is often non negative (as in the case of times and segment lengths), sets a lower bound of $k-1$ on the value of y_s (once $y_s \geq k-1$, it cannot go below $k-1$ because the integer division gives a contribution which is always non negative). A common mistake is to choose the scaling factor so that $k = 2^d$. By doing this way, we have a large error on numbers smaller than unity. As an example, in Net/3 the *scaled* values of SRTT and RTTVAR can never go below 7 and 3 respectively.

In order to get correct results on small values, we should choose d so that $2^d >> k$. In this way, the maximum error on y is reduced to $\frac{k}{2^d}$, and small values can be represented with sufficient precision.

2.3 Backoff policy

In this section we point out a series of problems which are present in the computation of the timeouts for the retransmission of segments. This is commonly called the *backoff policy*.

Retransmissions of a lost segment are not done at a fixed rate; rather, a truncated exponential backoff policy is used to compute successive timeouts[9]. The motivation for such a policy is the following: a lost segment probably indicates that the network cannot keep up with the current transmission rate, and its queues are saturating. As a consequence, the sender has to wait that the network recovers (i.e. flushes the queues) before retransmitting the lost segments. Repeated losses mean that the current retransmission timeout is not sufficient, and should be increased.

Exponential backoff has a problem – it grows too quickly – which is usually circumvented by truncating the exponential growth at some point. In Net/3, exponential backoff is implemented by using RTO as the base timeout and doubling the timeout at each retransmission. The resulting value is then clamped to keep it within reasonable limits (1-64 s). In practice clamping occurs after very few retries, as shown in [14, Sec.25.11], so the exponential backoff quickly becomes a constant backoff.

We asked ourselves if this way of computing the timeout (i.e. basing on the RTO only) is reasonable. The recovery time (what the exponential backoff tries to guess), should depend largely on the size of the queues on the path between the sender and the receiver, the speed of the communication links, and the congestion level. It certainly depends very little on the time needed for the physical propagation of signals across the network, which is constant and independent of the congestion.

Nevertheless, Net/3 applies exponential backoff to the full RTO, while there is no knowledge, hence no dependency, on the actual speed of the network. This is likely to generate much larger timeouts than necessary, because the timeouts must then be calibrated for the slowest networks. In fact, the largest possible timeout (64 sec) corresponds to 115 KB of data on a 14.4Kb/s link, and 80 MB on a 10 Mb/s link. We argue that such long timeouts only make sense on a slow network, while they are practically pointless on fast networks, where it is unlikely (and unpractical) to have such a huge amount of buffering on a path.

Note that the computation of the timeouts is affected by two other problems, which have already been noticed in the literature. First, the minimum timeout is bound to 2 ticks, because a measured RTT of 0 ticks can be as large as $(1 - \epsilon)$ ticks, and a timeout of 2 ticks can expire after $(1 + \epsilon)$ ticks (see [9]). There is no way to overcome this limitation other than using a finer granularity clock. Second, as also noted in [2, Sec.3.2], and discussed in more detail in the previous section, there is a minimum value for the RTO estimate of 1.5 s, no matter how fast the network is.

3 Proposed changes to Net/3 code

We have pointed out three problems in the handling of retransmissions in Net/3. The first one (bounded number of retries) has a simple fix, i.e. continue retransmissions for a sufficiently large number of times and interval of time. The second problem is even easier to solve, as it only requires a change in the scaling factors for SRTT and RTTVAR.

The last problem lies in the computation of retransmit timeouts, which are often too large. When segments need to be retransmitted frequently, these large delays make performance decay dramatically. To achieve the best performance, one should realize that the network is lossy, and retransmit segments more aggressively. Of course this cannot be done, otherwise the network would collapse under the flood of retransmissions.

An alternative approach would be the use of large windows and, more importantly, selective acks, together with a modified congestion control algorithm. With a large window, we allow a large number of segments to be sent without needing an immediate response from the receiver. As a consequence, even if the network is lossy, a smaller, but still large, number of segments can be received at the remote side. At this point, if selective acks are used, the receiver can report which segments have been received and which one have been lost, so that the sender only needs to retransmit the latter. The condition for this mechanism to be effective is that the congestion control algorithm do not react to losses by shrinking the send window too much, otherwise the throughput would be limited to little more than a single segment per RTT.

The reason why we do not focus on selective acks in this paper is twofold. First, the various congestion control algorithms base their action on the presence of duplicate or missing acks. The implementation of selective acks would interfere deeply with these algorithms, thus requiring extensive changes to the code and a careful study of the side effects that are introduced by the new code. Second, and more important, we are possibly looking for a solution that can work effectively with existing TCP implementations. Selective acks require changes both to the sender and the receiver side, and thus it would be of little use until all systems have upgraded their software[3].

As a consequence, we suggest a complementary mechanism, which simply aims to make the actual retransmit timeouts more related to the (estimated) speed of the network, and is of very simple implementation. This has the advantage of requiring changes only to the sender's TCP, which means that, for example, the upgrade of a few large servers can improve the performance for the entire user base. The actual backoff is thus computed by using a new function, `tcp_backoff()`, whose body is shown in Figure 1. For the computation of `tcp_backoff()`, we use an estimate of the bandwidth available on the connection. Methods to compute this estimate are given in the following section.

3.1 Estimating the bandwidth

We should first say that any measurement of the maximum bandwidth available on the connection is going to be incorrect and possibly biased, in one direction or another. First of all,

[3]Note that it is possible to implement selective acknowledgements by using modified RFC1323 timestamps. Anyways, the adaptive computation timeouts proposed in this paper are completely orthogonal and complementary to the use of selective acknowledgements

the communication channel is shared by many connections, thus making the measure different from time to time. Second, measurements based on a communication with the remote host will include the propagation delay of the channel and the effect of queueing at intermediate nodes, which can be highly variable. These effects can be minimized, but it takes many packet exchanges, and possibly some collaboration at the remote site, to produce accurate results.

This said, a simple way to measure the available bandwidth could be to compute the ratio

$$\frac{\text{segment size}}{\tau_2}$$

using the τ_2 defined in Sec.2. Unfortunately, there is no easy way to measure τ_2, and its closest approssimation is given by the RTT[4].

This estimate estimate of the bandwidth is not suited to our purposes for two reasons. First, it is limited by the resolution of the clock to at most one segment per clock tick. With the coarse (500 ms) timers used in Net/3, even when timestamps are used, this is three orders of magnitude lower than what is available on an Ethernet, and probably lower than the bandwidth on any practical network nowadays. It is possible to overcome this limitation by using a faster clock, or by accumulating several values before computing the ratio, thus improving the effective resolution.

The second, more important limitation derives the use of RTT in place of τ_2. This limits the estimate to at most one segment per RTT, which, on long links, is an exceedingly low value. The knowledge of the propagation delay would be necessary to overcome this limitation. A rough estimate of the propagation delay can be obtained by timing segments of different sizes. However, in practice this would require additional traffic because bulk data transfers tend to use maximum sized segments to improve the performance.

An alternative (and preferred) way to measure the bandwidth consists in the computation of the ratio

$$\frac{\text{total bytes sent}}{\Delta t - t_{idle} - t_{retran}}$$

over a suitable interval Δt. t_{idle} is the total idle time during Δt (defined as the sum of all intervals during which there are no unacknowledged data at the sender), and t_{retran} is the sum of all retransmission timeouts during which no ack arrive. By using a sufficiently large Δt, the limitations introduced by the low resolution of the clock can be overcome. If the connection does not have too many pauses due to retransmissions and timeouts, this measurement overcomes the

[4]unless we can determine, by other means, the propagation delay for the connection.

limitations introduced by long paths. Of course, if retransmissions are frequent, the estimate suffers from the influence of the propagation delays.

It is useful to pass the computed value through a lowpass filter so that it can adapt to the changes in the speed of the network If fixed point arithmetic is used in the computation, care must be taken to avoid the same mistakes which appear in the computation of SRTT and RTTVAR (see Section 2.2).

3.2 Computing `tcp_backoff()`

As it has been anticipated in Section 3, we use the estimate of the available bandwidth for an adaptive computation of backoff times. The basic idea in the computation of `tcp_backoff()` is to use exponential backoff, as "prescribed" by the theory, but with the following changes to the original algorithm:

1. only the variable part of the RTT (i.e. RTTVAR) is used as the base for the exponential growth. The constant part of the RTT, which can be represented by SRTT, appears only as an additive term in the computation, i.e. it is not multiplied by the exponential backoff;
2. the rate of growth and the upper limit for the backoff are reduced as the bandwidth increases. To this purpose, the estimated bandwidth is used, computed as shown in the Section 3.1.

An example of the implementation of `tcp_backoff()` is shown in Figure 1. The code is given just as a proof of concept, and serves us to evaluate the overhead introduced by the new code and the effectiveness of our proposal. First note that two fields, `bw_sent` and `bw_ticks`, have been added to the `struct tcpcb`. They are used to hold the smoothed means of the number of bytes sent, and the elapsed time (in ticks), using the last method described in Section 3.1. These fields are initialized appropriately when the connection is opened. This is done either with bandwidth information coming from the network layer, if available, or with values that produce a "slow-start" behaviour, i.e. a low bandwidth is assumed until sufficient samples have been accumulated.

The function first computes the bandwidth (`bw`) in KB/s, clamping the values between 1 and 128. The actual backoff is then computed as an approximation of

$$b(r, bw) = \mathrm{RTO} + \max\left(l(bw), v2^{r/k(bw)}\right)$$

where RTO is the retransmission timeout as computed by `TCP_REXMTVAL`, v is the variant part of the RTT (we use `t_rttvar - 2`), r is the number of retries so far, and $k(bw)$ varies between

```
short
inline tcp_backoff(struct tcpcb *tp)
{
    long r= tp->t_rxtshift;
    if (r !=0) {
        int bw= 1 + ((tp->bw_sent / (tp->bw_ticks + 1)) >> 9);

        if (bw > 128) bw=128;
        if (r > 20) r=20;
        r= r*127*(bw+15)/(556*bw+1476); /* compute and apply k(bw) */
        r= ( (tp->t_rttvar < 3) ? 1 : (tp->t_rttvar - 2)) << r;
        bw = 1024/(7+bw); /* compute l(bw) */
        if (r > bw) { /* apply l(bw) */
            r= bw;
        }
    }
    return r + TCP_REXMTVAL(tp);
}
```

Figure 1: An example of the body of `tcp_backoff()`

1 (for 1 KB/s or less) and 4 (for 128KB/s or more). The result is then upper bounded to $l(bw)$, which varies between 128 (for 1 KB/s or less) and 8 (for 128KB/s or more).

The actual code is obfuscated by the checks and roundings to avoid arithmetic overflows, the correction to the value of `t_rttvar` to account for the error discussed in Sec 2.2, and, especially, the approximation of the formula by using integer polynomials. The latter is done to achieve the desired output without using floating point computations in the kernel.

The behaviour of the code is shown more clearly by Figure 2, which plots the function for different values of `t_rxtshift` and bandwidths, in the (very common in practice) case of `t_rttvar=3`.

Note that a fast path is provided for the common case of $r = 0$, i.e. the first time a segment is sent. On a 486/66, the function, declared `inline` and compiled with `gcc -O2` takes approximately 0.18 μs to execute for $r = 0$, and 3.4 μs for other values of r, i.e. when a retransmission is needed. The computation of the bandwidth estimator has a negligible cost, as it requires just a few machine instructions per segment.

3.3 Performance

The solutions presented in this paper have been tested on FreeBSD 2.X (a Net/3 derivative). It was not possible to apply the same changes to Net/2 derivatives, because Net/2 stores `tcpcbs` into `mbufs`, and space was too tight to include the bandwidth estimator[5]. The only change that is easily applicable to the Net/2 code is the increase of the number of retransmissions.

It is straightforward to see that our changes allow TCP connections to survive to large losses, and do not break existing algorithms for congestion avoidance.

A precise evaluation of the effectiveness of the proposed backoff policy is hard, because of the presence of many variables and of the various algorithms involved in scheduling transmissions and retransmissions in TCP. However, the approximate analysis which follows can give sufficient information.

Let us assume that the connection involves the exchange of n segments, and that the total transmission time in absence of losses is T_w where w indicates the dependency on the window size. If the transmission window is just a single segment ($w = 1$), then it is simple to evaluate the effect of packet losses on performance: at every retransmission a segment would experience an additional delay $b(r, bw)$, where r is the retransmission number, bw is the estimated bandwidth

[5] unless one uses spare bits in various fields. For many variables, only small numeric values are meaningful, so that the remaining bits could be used for different purposes.

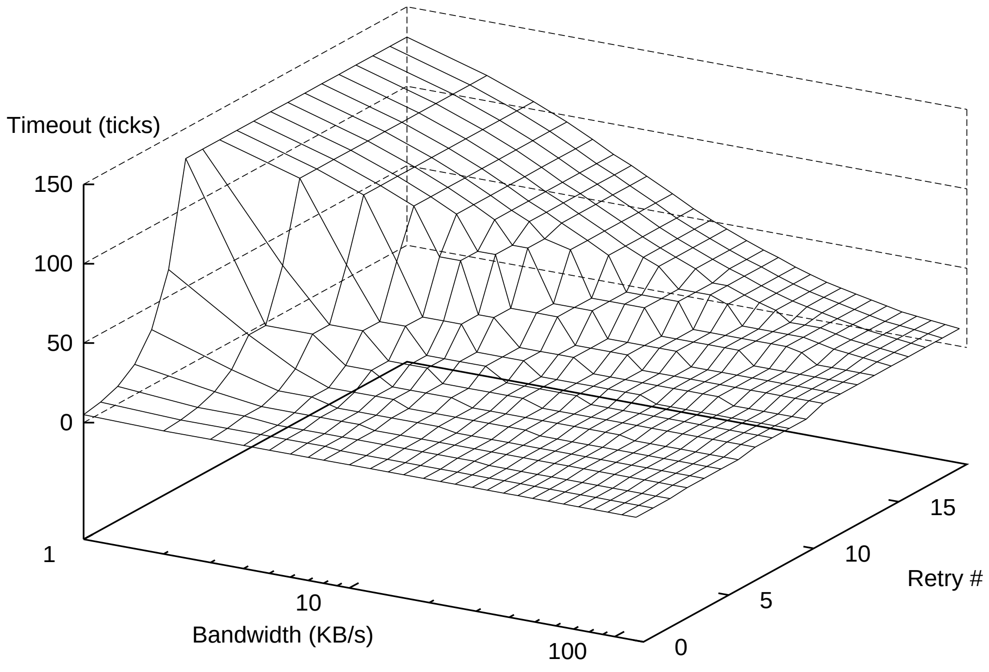

Figure 2: The variable part of the retransmission timeout for various bandwidths.

and $b(r, bw)$ is the value returned by the function which computes the backoff. As a consequence, for a segment loss rate of α, the total overhead for each segment is, on average,

$$ov = \sum_{r=0}^{\infty} b(r, bw)\alpha^{r+1}$$

and the total transmission time becomes $T_1 + n \cdot ov$.

If windows are larger than one segment, a successful retransmission might cause several segments to be acknowledged at once. Thus the total transmission time becomes lower than $T_w + n \cdot ov$, as the time to transmit the segments following the retransmitted ones is already accounted for in $n \cdot ov$. Note, however, that in presence of losses the sender might decide to reduce its transmission window. Thus, our approximation can still be considered valid, especially as the loss rate increases, and is certainly useful to compare the different backoff policies.

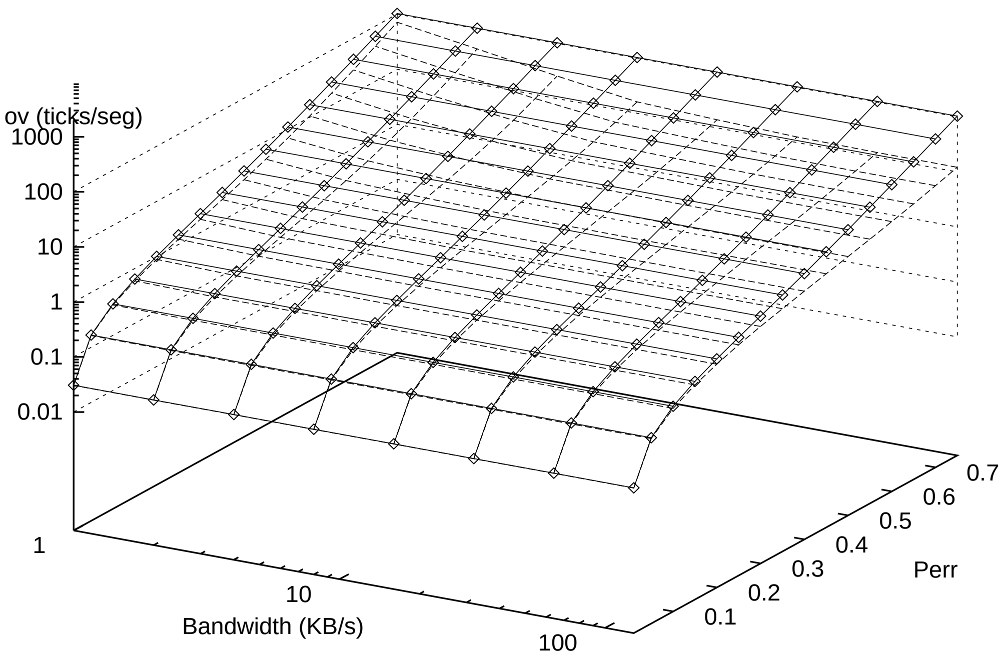

Figure 3: The overhead due to retransmissions, for variable bandwidths and loss rates

In Figure 3 we plot the value of ov for different bandwidths, and loss rates between .01 and .75. The upper graph shows the original backoff policy, the lower graph shows the new backoff policy. The summation has been limited to 12 elements. The values have been computed in the common case of `t_srtt=7` and `t_rttvar=3`, which yields a base value for the timeout of 3 ticks. As it can be seen, for low loss rates ov is negligible, while it becomes dominating as the loss rate increases. The figure also shows how the new algorithm for computing backoffs gives slightly better performance even in absence of bandwidth estimators, thanks to the removal of the constant part of the RTT from the exponential backoff. The advantage in using the bandwidth estimator appears clearly as the bandwidth and the loss rate increase.

During our experiments, we were able to send large (> 1 MB) files to remote systems running Net/2 and Net/3 derived kernels. Despite loss rates of up to 70% during the file transfers, the connections did not break, and the actual transfer speed was around 0.5 KB/s (but the peak available bandwidth, and the one estimated by our code, were higher). The low value of the effective bandwidth obviously comes from the timeouts that are incurred before being able to retransmit a lost segment. In this respect, the availability of selective acknowledgements would greatly reduce the occurrence of timeouts, thus improving the bandwidth estimate[6], and making the remaining timeouts slightly shorter.

One might think that our changes cause excessive congestion to the network. However, on a slow network (e.g. a dial-up point to point link) our code behaves essentially as the original Net/3 code. On fast networks, retransmissions become more frequent, but never more than 15 segments/minute, which is at least 300 times below the estimated capacity of the network. Actually, the original 64 s timeout might even be more demanding than this for slow networks. Thus we believe that our changes do not significantly increase the danger of congestion in the network.

4 Conclusions

We have discussed some consequences of the way retransmissions are handled in the BSD implementation of TCP. These features influence the performance of TCP over very lossy networks. Some simple changes to the retransmission policy have been proposed: among them, a large increase to the maximum number of retransmissions, and the use bandwidth estimator for a more adaptive computation of retransmission timeouts. These changes allow TCP connections to survive in presence of large losses and data transfer to proceed at an acceptable rate.

[6]The estimate is still guaranteed to be lower than the actual bandwidth, so there is no danger of instabilities.

It should be emphasized that the solutions given in this paper are not, by themselves, a definitive answer to the problem of lossy networks: they should be complemented by a selective ack mechanism to achieve much better performance. However, even without selective acks, our solutions can be effective, and have the significant advantage of requiring (simple) modifications only in the sender's TCP. This is very important in the case of large servers, because the simple upgrade of the servers would improve performance for the whole user base.

We have discussed the use of bandwidth estimators only in the computation of retransmission timeouts. There are, however, several other situations (ARP requests, SYN segments, window probes) in which the network software has to deal with packes losses and reacts by sending multiple datagrams. Currently, the sending rate for all the above cases is independent from the actual bandwidth available on the network or on the connection. This leads to very conservative choices on the actual values of the timeout, which (especially in the case of SYN segments) might be inadequate for the fast networks available nowadays. We believe that a generalised use of adaptive timeouts can have beneficial effects on the performance, especially in the interaction with human users, without increasing the risk of congestion on the network.

References

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

[2] L.S.Brakmo, L.Peterson, "Performance Problems in BSD4.4. TCP", 1994, available via `ftp://cs.arizona.edu/xkernel/Papers/tcp_problems.ps`

[3] 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

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

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

[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] IETF, "RFC1122: Requirements for Internet Hosts – Communication Layers", R. Braden (ed.), October 1989

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

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

[11] P.Karn, C.Partridge, "Estimating round-trip times in reliable transport protocols", *Proceedings of SIGCOMM'87* (Aug.1987), ACM

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

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

[14] W.R.Stevens, "TCP/IP Illustrated vol.2", Addison Wesley Publishing Co., New York, 1994

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