Tweaking TCP's Timers
CUED/F-INFENG/TR.487

Kieran Mansley

Laboratory for Communication Engineering,
Cambridge University Engineering Department,
William Gates Building, 15 J.J.Thomson Avenue,
Cambridge CB3 0FD. UK
kjm25@cam.ac.uk

July 6, 2004

Abstract

This paper presents an architecture for implementing TCP timers efficiently at user-level, as part of the Cambridge User-Level TCP (CULT). The unusual architecture that CULT presents (as opposed to the normal in-kernel arrangement) makes the use of a separate thread for timers undesirable. In addition, the facilities for accurate time measurement and scheduling at user-level are limited in comparison with the kernel.

This leads to a number of changes, illustrated by considering the delayed-acknowledgement timer. In particular: (*i*) timers are processed "in-line" by the data thread to which they correspond, avoiding the need to lock due to a separate timer thread; and (*ii*) the upper and lower limits on the delayed-acknowledgement timer are changed by the use of "timer-buckets" to ensure no unnecessary delayed acknowledgements are sent while also reducing the maximum expected delay.

The new architecture results in a more scalable solution as no lists of connections need to be searched on timer clock ticks at the cost of slightly increased complexity to set and clear timers. Should the period of timers decrease to cope with faster networks this trade-off will become more important.

1 Introduction

1.1 Cambridge User-Level TCP

Cambridge User-Level TCP (CULT) [1] is an attempt to create a high performance implementation of TCP at user-level. It is targeted at server clusters, and makes use of facilities provided by RDMA cluster networks such as CLAN [2] to increase efficiency in a number of areas including: (*i*) low overhead demultiplex of incoming packets due to the connection oriented nature of the underlying networks; (*ii*) true zero-copy transmission facilitated by buffering for retransmission being performed at the network gateway; (*iii*) advanced load-balancing based on the network buffer usage of the application; and (*iv*) removal of context-switch and locking overheads by the avoidance of any dedicated threads in the protocol stack (all protocol processing is performed using the application thread that requests it).

The efficient implementation of TCP timers is one aspect of this work, and the CULT architecture makes this implementation particularly interesting: it presents unusual constraints and problems when compared to more traditional kernel-based TCPs. The rest of this paper describes these challenges in more detail, and the solution that has been devised to address them.

1.2 TCP Timers

TCP uses timers to cope with inactivity. Whenever an action or response is expected from a remote node, the local TCP will set a timeout to recover in case that action or response is not received. For example, when a segment is transmitted, TCP sets a *retransmission* timer which will resend the segment if it is not acknowledged.

In total there are seven different timers:

Connection-establishment timer: set when a SYN is transmitted, and aborts the connection if no response is received within 75 seconds.

Retransmission timer: set when data are transmitted, and retransmits the data if no acknowledgement is received.

Delayed-acknowledgement timer: set when data are received that do not need to be acknowledged immediately. If, after 200ms, the acknowledgement is still pending (i.e. it has been unable to piggy-back on an outgoing data segment) it is then sent.

Persist timer: set when the remote node advertises a window size of zero, thus preventing any data being sent. If the window has not been opened when the timer expires, 1 byte is sent in case the window update was lost. (Window updates, like acknowledgements, are not sent reliably).

Keep-alive timer: expires after 2 hours of inactivity (if requested by the application) and sends a special segment to keep the connection open.

`FIN WAIT 2` **timer:** set if the connection is in the `FIN WAIT 2` state, and cannot receive any more data. If after 11 minutes 15 seconds a FIN has not been received the connection is dropped.

`TIME WAIT` **(or 2MSL) timer:** set when a connection enters the `TIME WAIT` state, and expires after twice the Maximum Segment Lifetime (MSL). The state for the connection is deleted allowing the socket to be reused.

The duration of the delayed-acknowledgement and persist timers depends on the measured round trip time of the connection. The reasoning behind all these timers is discussed in detail by Stevens in TCP/IP Illustrated. [3, Chapter 25]

1.3 Historical Constraints

When TCP was specified in RFC 793 [4] in 1981 the support for time measurement in operating systems, when compared to today, was poor. TCP was also designed to be easily portable to a large number of operating systems, and so made very few assumptions and demands about what was available, even if some systems could offer much more. In addition, TCP did not need as accurate time measurements, as the time intervals being measured were larger due to slower networks resulting in lower bandwidths and longer round trip times.[1]

As a result, to implement all the timers TCP only requires that two functions are called periodically: (*i*) the *fast timer* is called every 200ms and (*ii*) the *slow timer* every 500ms. TCP uses these two periodic “ticks” to schedule and check all the timers described above, as well as measuring round trip times.

As network speeds have increased this relatively coarse-grained ability to measure time and schedule actions has become increasingly noticeable, and although operating systems can now measure time very accurately with very little overhead (particularly in the kernel), TCP has not adapted to make full

[1]For example, the original TCP specification bounds a measured network RTT to be greater than 1 second. As this is used to time retransmission, it can lead to unnecessary delay in modern networks, which will commonly have RTTs of less than 100ms. This is one of the problems addressed by TCP Vegas [5].

use of this facility (although some modifications such as TCP Vegas [5] do go some way to improving matters). The problems that this leads to are widely known, with many proposed solutions which are discussed fully in Section 5.

1.4 Timer Implementation

These historical constraints have to some extent shaped the way that most TCPs implement timer support.

Setting and clearing timers is a common operation (each sent and received segment will involve at least one or more timer operations), and so this must be cheap: usually just involving setting a flag in the protocol control block. However, this means that on each of the fast and slow clock ticks the list of connections must be searched to discover which require attention, which has time complexity of $O(n)$ where n is the number of connections. The drawbacks of this approach are well illustrated by the delayed-acknowledgement timer:

When the list of connections is searched for those that have delayed acknowledgements it is not known when, other than after the last fast timer tick, this acknowledgement was delayed. i.e. It could have been waiting for anything between 0ms and 200ms. This has two side-effects: (i) an acknowledgement could be sent unnecessarily if it had only been waiting for 0ms, and (ii) an acknowledgement could be delayed for much greater than the RTT of the network if it had been waiting for 200ms.

The TCP specification suggests that at least every other acknowledgement should be sent. i.e. If there is a request to delay an acknowledgement, and one is already pending on that connection, an acknowledgement should be sent straight away. (As acknowledgements are cumulative, a single one is sufficient.) This means that on a reasonably busy connection (even if it half-duplex, and so acknowledgements will never be able to piggy-back on data) it should never be necessary to send a delayed acknowledgement on a timer tick. However, in practice, on average every other fast timer tick will result in one being sent.[2]

As well as the scalability issue, searching the list of connections each clock tick has a second detrimental side effect. Timers are, in a user-level context, most easily implemented as a separate thread. Great effort has been expended to avoid the need for multiple threads per connection in the CULT stack. If a separate thread were used for timers, many of the benefits gained

[2]This can be demonstrated by noting that if an odd number of acknowledgements are requested between two ticks, an acknowledgement will be sent at the tick, but if an even number of acknowledgements are requested, no acknowledgement will be sent on the next tick.

from single threading would immediately be lost: locking would be required for access to the stack to ensure that the state for each connection is only used by a single thread at a time.

1.5 Time at User-Level

Most TCPs are kernel-based. As CULT is a user-level stack, it has a different set of problems when trying to efficiently implement timers. Accurate time measurement is a common operation within the kernel, and so is well supported: on Linux, for example, the “jiffies” variable is incremented whenever the timer interrupt occurs (typically 100 times a second). The kernel also provides “task-queues” or “tasklets” and “kernel timers” that can be used to schedule execution of code at a later time - ideal for implementing TCP’s timers.

As with many operating system resources, access to such features from user level requires a system call (e.g. gettimeofday()), which is relatively expensive. Solutions to this such as Soft Timers [6] are able to reduce this cost by executing the event handler at certain (common) entry points into the kernel, when the overhead is already being taken. There is also the possibility of using a real time scheduler to ensure that a process is executed at a specific time.

The approach taken by CULT is to use the free running processor cycle counter which is accessible on x86 systems using the “`rdtsc`” assembler instructions. This has very low overhead and provides high-resolution (of the order of nanoseconds) time information. It does require calibration however, to gain an accurate measure time in more standard units (seconds rather than processor cycles).

2 Cross-Thread Scheduling

An interesting way to view this problem of efficiently implementing the fast-timer is to consider it as a scheduling problem. Threads (or processes) can be in many scheduling states: (*i*) *Runnable* when it is only prevented from running because the processor is busy with something else; (*ii*) *Running* when it is actually executing on the processor; and (*iii*) *Waiting* when it is waiting for something (some I/O to complete, or a lock to be released for example) and so cannot currently continue. A typical state diagram for the life cycle of a process is shown in Figure 1.

Processes enter the Waiting state when they need some other process or task to complete an action before they can continue. There is generally

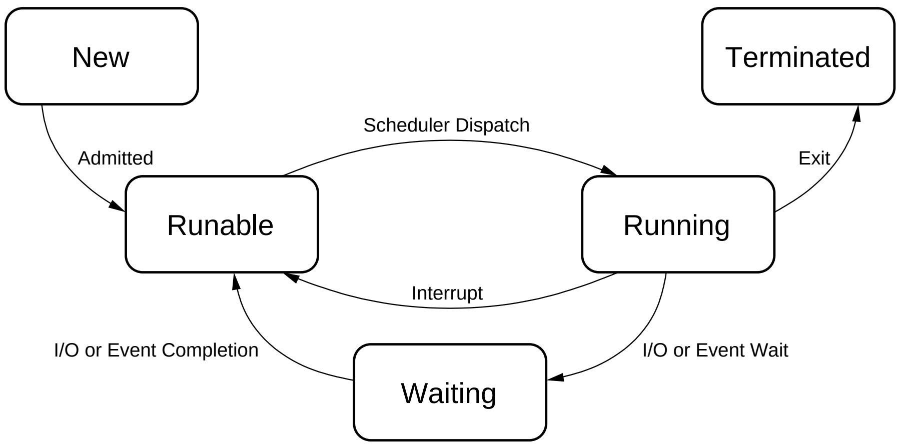


Figure 1: Scheduling States of a Process

no mechanism for the event that the processes is waiting to complete to be changed once it is in the Waiting state, or for a process to move from Runnable to Waiting without going through Running. This is because, in general, the process itself must decide when to enter the Waiting state, and the only way for it to change its mind about that is for that process to be executed: i.e. It must be in the Running state.

In the case of our TCP timers, the timer thread is usually in the Waiting state, waiting for 200ms to pass. At the end of this time it will become Runnable, and eventually Running when it reaches the head of the Runnable queue. However, because there it usually no work for it to do, it will quickly return to the Waiting state. In other words, it is polling.

It is possible to determine if the timer thread has any work to do. i.e. if there are any acknowledgements that have delayed for more than 200ms (or whatever threshold time is set), and this calculation *can be performed by any other TCP thread* as follows:

Each TCP connection has either a time at which it would like the fast timer to be run (200ms since it delayed an acknowledgement) or no requirement for the fast timer (if it does not currently have any acknowledgements pending). If any of the times for the connections that have delayed acknowledgements are less than the current time, then the timer thread needs to be run. However, these times are constantly shifting as acknowledgements are sent, and other acknowledgements are delayed. As a result, the time at which the timer thread should next become Runnable is constantly updating. However, as mentioned above, there is currently no mechanism whereby the condition that a process is waiting on can be updated. If support for this were available, it would mean one group of threads or processes (the data-sending threads) preventing another (the timer thread) from becoming Runnable, which would be a new scheduling concept.

Processes do, of course, already influence when others can be Runnable through, for example, condition variables (and it may be possible to produce the desired effect through the use of these), but there is currently no way to explicitly alter the condition that another process is waiting on; only whether or not that condition is true. Condition variables would also require one of the other threads to be active at the right time in order to signal the condition variable as met, whereas the suggestion above would still operate correctly if none of the data threads are currently running. Scheduler Activations [7] (where the kernel schedules processors to a group of threads, which then use a user level scheduler to divide the processor among them, so allowing for an efficient N:M threading model) are more promising, and could be extended to perform this role.

To support this would require significant changes to either the POSIX thread library or the kernel (depending on which scheduler is being used) to provide the necessary API. However, due to the complexity of this, and the fact that for CULT it is hoped that the timer thread can be avoided completely for active connections (as discussed below) this has not been done. It would however constitute a very interesting area of further work, and as scheduling is so key to many aspects of operating systems, it could have many other applications.

3 New Timer Architecture

This dissertation presents a different approach to TCP timers, developed to counter the problems described (namely, (*i*) the inaccurate delay of acknowledgements, and (*ii*) the scalability and thread-consequences of searching lists of connections by timers).

3.1 Fast Timer Buckets

Timer buckets are a means of providing the upper and lower time limits on a delay. There are two buckets, and events that require a delay are marshalled into one of them. The target bucket that events are placed in is swapped periodically, and each bucket is emptied with the same periodicity but out of phase with the switching of the target bucket. This is illustrated by Figure 2.

The difference in phase between switching the target bucket and emptying that bucket is responsible for ensuring a non-zero lower limit. i.e. There is

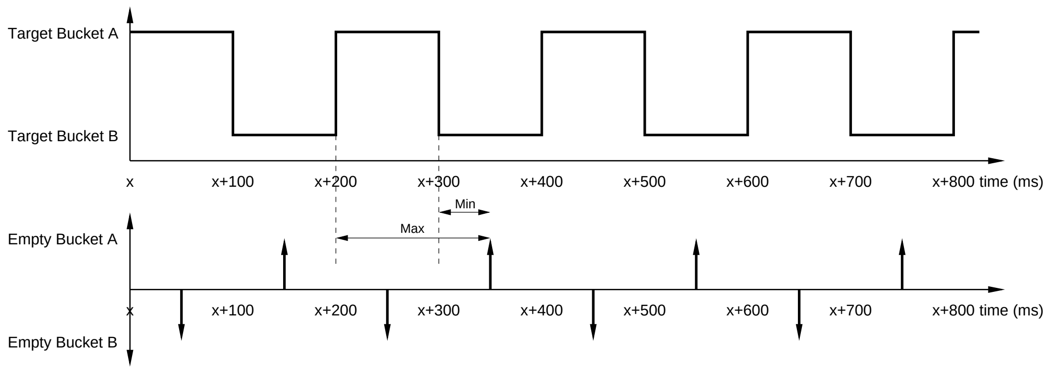


Figure 2: Timer Buckets

a delay between a bucket no longer being the target, and that bucket being emptied. The upper limit is, obviously, enforced by the fact that the buckets are regularly emptied.

In the case of delayed acknowledgements, it is convenient to set the lower limit at 50 ms and the upper limit at 150 ms. This maintains the average delay from the standard 0-200 ms range, but both reduces the maximum delay and introduces a guaranteed minimum delay to ensure that piggybacking of acknowledgements is encouraged. To achieve these limits, the period of the bucket switching and emptying is 200 ms, with the emptying 90° (i.e. 50 ms) out of phase, as shown in Figure 2.

3.2 Lazy Implementation

This work focuses on how to efficiently implement protocols. For this aspect of TCP we need a way to regularly switch between the two buckets and empty them appropriately, without requiring a separate thread to control it.

This has been achieved using a lazy approach: whenever a data thread performs an timer operation, or an operation that is blocked, it determines if a timer bucket action is required (i.e. switch or empty bucket).

Each bucket is represented using a simple counter. It records the number of delayed acknowledgements that are currently pending. When an acknowledgement is delayed, the current bucket's counter is increased. When an acknowledgement is sent, the counter is set to zero.

Two variables are used to track which bucket to use: (*i*) the current bucket to use if delaying an acknowledgement, and (*ii*) the bucket to examine when "emptying" on the next timer tick. They are updated as follows:

Current Target Bucket: The time at which the next bucket switch should take place is also recorded. Whenever an acknowledgement is delayed (just before the timer bucket counter is increased) this time is compared to the current time[3], and if it has passed the buckets are switched. This has the side effect that, because it is done lazily, the buckets do not switch exactly every 100 ms, and in the absence of any acknowledgements being sent a bucket could span considerably more than that. This is, however, safe in that it only happens to an empty (inactive) bucket: as soon as something tries to add to a bucket, the switch occurs, and the "next switch time" is updated accordingly.

Bucket to Empty: Whenever a bucket is emptied, a "`struct timeval`" timeout structure is set to 100 ms from the current time. This structure is passed to any blocking operation involving that connection, so that once 100 ms has passed the timer handler is executed, without requiring a separate thread. The timer handler examines the current bucket, sends any acknowledgements that have been delayed, and updates the timeout structure.

This approach separates the timing of the bucket switch and the bucket emptying, and so the two may drift, resulting in one not falling exactly half way between the other. This effectively alters the phase difference between them, and is not ideal: it will alter the limits on the time for which acknowledgements are delayed. However, the fact that they are not in phase is enough to ensure that there is usually sufficient delay to enable piggybacking.

3.2.1 Implications

Although at first glance the benefit of not sending a few unnecessary acknowledgements is small, this has wider implications. It has been suggested that TCPs' timers' frequencies should be increased to allow for more accurate time measurement, and lower delays in responding to inactivity on modern networks. As the period of the timers decreases, the proportion of acknowledgements that would occur immediately before a clock tick would increase, and so the number of unnecessary acknowledgements would also rise. By using this bucket approach you will always maintain a gap between a delayed acknowledgement being requested, and the timer that sends it if nothing else has.

By maintaining a bucket for each connection the scheme also removes the need to search all connections on each timer tick: each connection is now responsible for administering its own delayed acknowledgement timers.

[3]For performance, the processor cycle counter (rather than a system call) is used to obtain a measurement of time.

This comes at the cost of increasing the complexity of requesting and cancelling a delayed acknowledgement, mostly as a result of the need to compare times to determine when to switch buckets. An implementation in C of the new algorithm on the Intel x86 architecture (making use of the "Read Time Stamp Counter" processor cycle counter for measuring time) requires an additional 28 assembler instructions (assembled without optimisations on a dual Pentium 4 Xeon architecture). Whether this trade off is beneficial will depend on the frequency of the fast timer, the number of active connections, and the traffic pattern experienced; this is discussed in Section 4.

While these timer bucket changes can be implemented on their own, additional benefits can be realised by extending the concept of making each data thread responsible for the timers of that connection to avoid the possibility of the data thread coming into conflict with the timer thread.

3.3 Timer Threads

The need for a separate timer thread arises from the requirement that some timers occur during periods when the application will not perform any operations on that connection.[4] As a result, the data thread will not be executing any protocol code, and so it can not be used to perform timer operations. This in turn leads to locks being required both by the timer thread, and the application thread, to ensure exclusive access to a connection.

As the timer thread is required for some cases where the data thread is not active, to avoid locking we need to ensure that only one of the threads will attempt to use a connection at any one time. To do this we ensure that those connections which are active, and so likely to receive attention by a data thread are not touched by the timer thread. The timer thread only looks after the timers of those connections which are inactive.

The split of connections is managed by maintaining lists of those in each group. Timers on connections in the `CLOSED`, `LISTEN`, or `TIME WAIT` states are managed by the timer thread. The rest are managed by the application data threads.

The application data thread manages timers as described in Section 3.2: it compares the current time against the next required timer whenever it enters a blocking operation. If 100ms have passed since the last "tick", the fast timer (and if 500ms have passed the slow timer) routine is called, but it only operates on that connection.

The timer thread operates much as it would in a normal environment,

[4]For example, the `TIME WAIT` timer is used to clean up connections after they have been closed.

waking every 100ms, looping over the list of connections it is responsible for and calling the fast and slow timer functions for each one.

When the connection lists are manipulated they must be locked to ensure consistency, but this only happens at connection setup and tear-down, and so is not on the data path.

This change has the following properties:

- No locks are required on the data path as only a single thread can operate on each connection at any one time. This represents a significant efficiency saving.
- Clock ticks on active connections do not require iteration over all connections to determine which need attention.
- Timers on active connections occur with lower priority than data operations. This is because timers occur when an operation would block, i.e. when a data operation cannot immediately complete. This is justified by the use of timers to react to *inactivity*: if there is data activity, then the timers are not required. This property is of particular interest to a loaded server where checking timers (but not doing anything useful as nothing is required) could take a significant portion of CPU time.
- The accuracy of timers on active connections is reduced, and we cannot *guarantee* they will be called in a timely manner, although the probability of them being unduly delayed is low.

4 Measurements and Results

The exact trade-off between increased complexity of queueing an acknowledgement versus decreased likelihood of sending an unnecessary delayed acknowledgement is complex. It is highly dependent on the traffic pattern, number of (and any interaction between) connections, the application that is using the stack, the cost of obtaining a mutex, and many other possible variables. The suggested improvements will clearly not improve matters in all cases. This section attempts to quantify these trade-offs.

4.1 Cost of Queueing an Acknowledgement

The proposed bucket scheme for the delayed acknowledgement timer increases the complexity of queueing an acknowledgement. This is a common operation (almost every received packet will result in this operation being called), so even a small increase in complexity could be detrimental.

To measure this effect profiling code was inserted into the function that queues and dispatches acknowledgements. This profiling measured and recorded the number of processor cycles that were spent in the added section of code, and the number of cycles spent in the old (non-bucket) equivalent. The latter of these is just a single bitwise OR operation to set a flag. The results of this are shown in Figure 3.

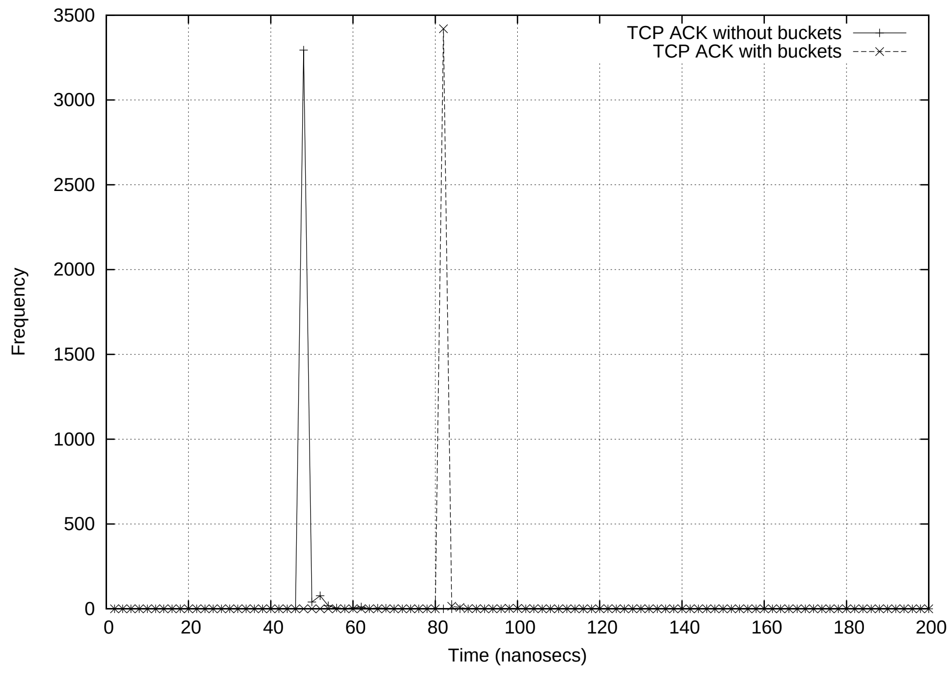


Figure 3: Time Taken to Queue a Delayed ACK

As these are very small sections of code there is a significant "probe-effect": the timings include the time spent measuring the time. However, this is constant for both versions, so a comparison is valid. The times (in seconds) are calculated offline from the processor cycle count using the bogomips measure. The test used to obtain these results was a simple "ttcp" benchmark, running on a Dual Intel P4 Xeon 2.4GHz server.

From Figure 3 the added complexity of dealing with buckets adds 34 ns to each acknowledgement request. This overhead will be taken each time the quantity of unacknowledged data exceeds twice the TCP Maximum Segment Size (MSS), and the frequency of this will clearly depend on the traffic pattern. If we consider a single connection using the full theoretical 1Gbps of bandwidth, and a MSS of 1460 bytes, it will happen every 11.7us. The additional overhead will therefore amount to 0.29% of a single CPU. In practise the overhead is considerably less than this as the theoretical maximum bandwidth is rarely achieved[5] and so the acknowledgements are requested less frequently.

4.2 Cost of Fast Timer Routine

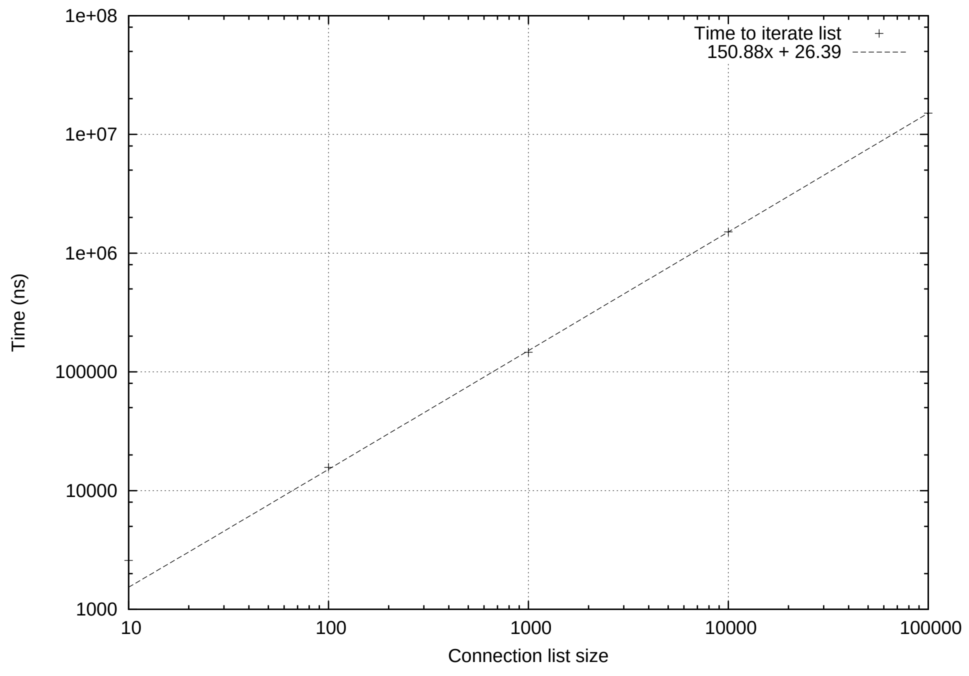
The benefit that the bucket scheme has is avoiding the need to call the fast timer routine. To measure the saving this represents a small piece of test code was created. This test wakes up every 200ms, and iterates over a list of "connections".[6] Each connection is locked with a mutex and a flag is checked, in the same way that would happen in the fast timer. In this way the test simulates the work needed to check a set of connections for a delayed acknowledgement, but not the work involved in dequeueing and sending any delayed acknowledgements found. As a result, the test measures just the work that has been avoided by the bucket scheme. It is also possible to easily vary the number of connections in a way that would be difficult to do on a "live" TCP stack.

The average time per iteration of the connection list, and the CPU usage this represents, is plotted in Figure 4 for different list sizes. The times plotted do not include the overhead of sleeping and waking this process every time the list needs to be walked. The times were calculated using both in-line profiling and the unix "time" utility (taking the ratio of user CPU time to elapsed real time to calculate the CPU percentage).

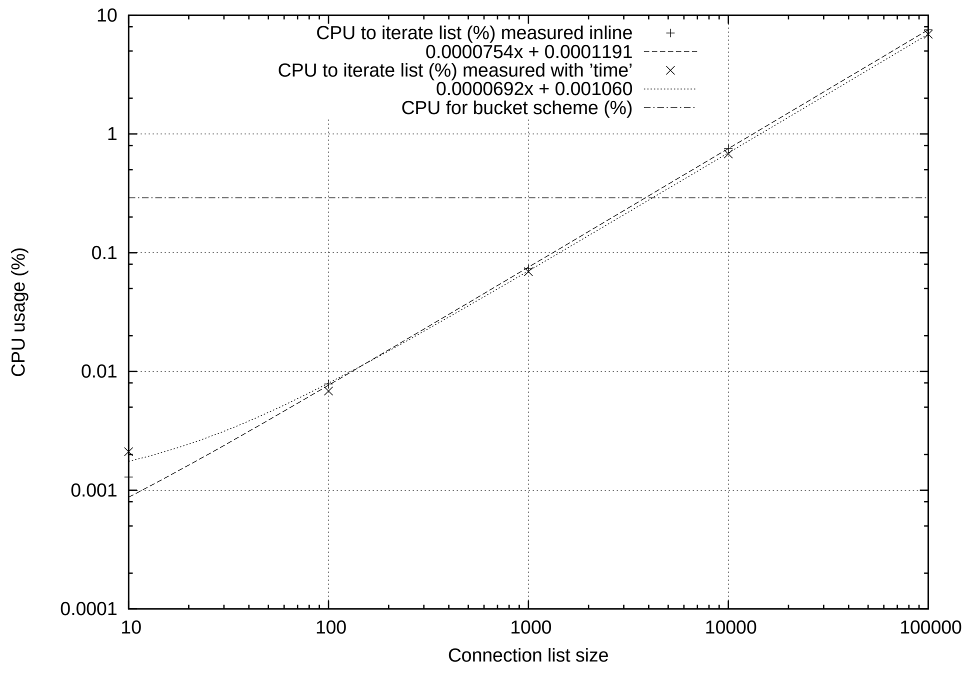
Figure 4 shows that the point at which the list iteration becomes more expensive than the bucket scheme is approximately 4000 connections (depending on which technique is used to measure the time taken). This may seem high, but it should be noted that this is the worst case for the bucket scheme, and an ideal case for the connection list iteration scheme. Other factors which would increase the cost of the list iteration (such as the thread switch overhead) were not taken into account. Also, if the period of the fast timer was reduced in the future (or this approach used for something other than TCP where the frequency is higher), this would result in the bucket scheme becoming more favourable. The bucket scheme does not have a direct dependence on the number of connections (it is capped by the number of acknowledgements sent, which is itself a function of the number of packets received), and so represents a more scalable solution.

[5]This analysis has ignored the bandwidth taken up by the network protocol headers.

[6]These "connections" are just data structures; there is no data transfer involved.



(a) Time to iterate list



(b) CPU used to iterate list

Figure 4: Overhead of iterating over a list of connections

Note the log-log scale

4.3 Reduction in Data Path Locks

We have not been able to formally quantify what is perhaps the major benefit of this work: the removal of the need for many of the locks on the data path. A conservative estimate of this follows: with a stack operating at full 1Gbps network bandwidth, transferring data to or from the application 1KB at a time, will involve 122070 read or write operations each second. If we have avoided a single lock/unlock pair of operations on each of these calls (measured above to take approximately 150 ns) the total saving will amount to 1.8% of a CPU, ignoring other factors. This dwarfs the overhead (0.29%) of the bucket scheme.

4.4 Discussion and Comparison

There are currently no widely used user-level TCP stacks to compare to the approach suggested here. Although it has been noted that timers at user-level are a different implementation problem to those in the kernel, this section therefore examines two of the most common (open-source) kernel TCP stacks: Linux, and FreeBSD.

During the period of time that this work has been carried out, the Linux kernel TCP has improved markedly. At the time of writing, the latest Linux kernel (2.6.5) uses the generic (i.e. not specific to TCP) kernel timer support. This is based on a hierarchical timing wheel [8] and an evaluation (of an earlier, but largely similar, version: Linux 2.5) showed it performed and scaled well [9]. Timing wheels (in their simplest form) represent an ordered circular list of events, with each element in the list containing the events that should occur at a particular time. On each tick of the clock, the next element in the list is examined, and any events present are processed. This scheme is $O(1)$ (with respect to the number of timers) for setting timers, clearing timers, and performing each clock tick, but does require that all timers are set for periods less than a fixed upper bound (to prevent wrapping around the circular list). The timing wheel scheme has been extended [10, 11] to remove the need for the fixed upper bound, and dynamically size the “wheel”.

This approach has similarities with that described in this section: it replaces the need to search lists of connections on each timer tick with an individual, per-connection, timer. This avoids the need to bucket delayed acknowledgements, as each connection is dealt with individually, but this is enabled by the ease (and low cost) with which kernel timers can be modified (from within the kernel). This would be hard to replicate at user-level without the ability to change the scheduling condition that a process is waiting on, as outlined in Section 2. It also, in the forms currently used, requires a regular clock tick to service the wheel. At user-level this lends itself to a separate thread, which is something CULT needs to avoid. It may be possible to adapt it to use a lazy evaluation, and so allow it to be used in-line by the data threads as proposed in Section 3.2

Timing wheels are also used by the FreeBSD implementation, and an implementation for the BSD UNIX version of TCP has demonstrated their scalability [12]. In FreeBSD each connection has a separate delayed acknowledgement timer, and timers are managed centrally using a timing wheel. The timer is reset to 100ms (rather than 200ms) each time an acknowledgement is requested as a result of received segments, if there is no delayed acknowledgement timer in progress. This condition means that delayed acknowledgements are sent for every other received segment, rather than waiting for the outstanding unacknowledged data to exceed twice the maximum segment size. The timer is cancelled (if in progress) when a segment is sent.

As with Linux, this is made possible by the ability to manipulate timers with low overhead in the kernel. However, the fact that both Linux and FreeBSD have opted for an increased complexity timer set/clear operation and individual timers for each connection, in order to achieve greater scalability justifies the goals and approach taken in this paper.

5 Related Work

The approach of using timeouts to detect failures has been highlighted [13] for some time as sub-optimal. In this paper Zhang suggests that a decision based on a timeout is a guess and that for high performance external events should trigger failure recovery with timeouts used as a second line of defence.[7] He highlights how it is particularly difficult to choose the amount of time to wait before retransmitting an unacknowledged packet, as this must be based on an estimate of the round trip time. This raises another problem of how to accurately measure the round trip time. Zhang also points out that external events convey information about what has gone wrong (and why), so allowing a more informed response. Finally, to have some confidence that a failure really has occurred timeouts must be set conservatively (so as to avoid false detections of errors). This can often lead to unnecessary delays, particularly where the timeouts are overly conservative. This is increasingly an issue in modern, faster, networks where the times being measured are considerably less than their equivalents when TCP was designed.

[7]The TCP Vegas approach to retransmitting after three consecutive acknowledgements is a good example of this.

As mentioned in Section 1.3 the TCP slow timer is used both for scheduling timeouts and for measuring round trip times. Aron and Druschel criticise the use of this relatively coarse grained (500ms) clock for measurements. [14] They show how it leads to a high variance in the measured RTT, especially when the RTT is comparable to the clock period, which in turn leads to a very high estimation of the retransmission timeout (RTO). They propose separating the roles of scheduling and measurement, and using a more accurate clock (1ms) for measurement. Scheduling of events is still done using the slow timer, and the retransmission timeout is still restricted to be no less than 2 ticks (0.5–1 sec) to ensure correct behaviour. In related work on TCP Vegas [5], Brakmo, O'Malley and Peterson demonstrate how the algorithm for calculating the RTO often results in estimates as high as 5 ticks (a delay of 2-2.5secs) and never less than 3 ticks (1–1.5 secs) for a round-trip time of 100ms. They propose a more aggressive and responsive method for performing retransmissions, by retransmitting once a RTT has passed, even if the timeout has not yet expired. To do this they record the (much more accurate) system time whenever a packet is sent, and compare this to the system time when certain acknowledgements are received. This is very much in line with the approach that Zhang expounds, as described above. Both of these papers show how improved time measurement can also lead to improvements in the slow start phase of TCP.

TCP implementations originally (and in some cases still do) performed a linear search of all the current connections on each timer tick to check if anything needs to be done. Aron and Druschel show how a simple change can lead to dramatic increases in performance. [15] They sort the list of connections so that all those in the `TIME WAIT` state are at the end of the list, and that those in the `TIME WAIT` state are in the order in which they entered the `TIME WAIT` state. They can then terminate the loop that checks the timeouts of the connections as soon as one of the `TIME WAIT` state connections is encountered. (In the case of the slow timer they must also check the first few connections in the `TIME WAIT` state in case those connections have expired and should now be closed). As the `TIME WAIT` state can be long compared to the amount of time that a connection is active (particularly for short lived connections such as you might see in a web server) there can be many more connections in the `TIME WAIT` state than any other. This change has a large impact on the time spent scanning the lists. In the test case used for comparison they were able to reduce the maximum CPU overhead of the timers from 25% to 0.36%.

Aron and Druschel extend their findings above to create "Soft Timers" [6]: a facility whereby events can be efficiently scheduled with a granularity of tens of microseconds. This is achieved by executing the event handler at certain entry points into the kernel. These are accessed frequently in the normal course of program execution in response to operating system activity (system calls, page faults, interrupts, etc). The overhead of entering the kernel has already been made to perform the required operation, and if an event handler is also executed at this time the cost of the context switch is amortised over them both. The times at which the trigger events will be called is neither regular nor predictable, so this technique can only schedule events probabilistically. However, experiments show that these triggers happen sufficiently frequently for it to be suitable for supporting a number of network operations. The authors suggest that rate-based clocking and polling of the network could both be implemented with low CPU overhead using this technique. This is a similar approach to the lazy evaluation of timers suggested in Section 3.2, but done at entry to the kernel, rather than at blocking points within the TCP stack at user-level.

6 Conclusions

This paper has presented an alternative approach to implementing TCP's timers, and has in particular focused on the delayed-acknowledgement timer as an example. The changes involve using two "timer buckets" to provide upper and lower limits on the delay, and so remove the possibility of delayed acknowledgements being sent unnecessarily. By checking the timers "in-line" for active connections, the Cambridge User-Level TCP is able to avoid having to perform connection list searching on each timer clock tick. This in turn removes much of the need for locking on the TCP data path[8], leading to greater efficiency in other areas of the protocol implementation.

The use of this scheme (in common with other proposals such as timing wheels) increases the complexity of setting and clearing timers (tests have shown that this 34ns to each delayed-acknowledgement request), but makes the checking of timers a more scalable operation, and removes some locking overhead from the data path. The point for TCP where the new scheme starts to offer a benefit is around 4000 connections; this is high, but as the period of timers decreases (as is arguably necessary for modern fast networks) this figure drops. In addition, the removal of locks on the data path introduces a saving that outweighs this overhead.

[8]Some locks may still be necessary to access, for example, a memory pool

Acknowledgements

References

[1] Kieran Mansley. Engineering a User Level TCP for the CLAN Network. In *Proceedings of the Workshop on Network-I/O Convergence: Experience, Lessons, Implications (NICELI), part of ACM SIGCOMM*, 2003.

[2] David Riddoch, Steve Pope, Derek Roberts, Glenford Mapp, Dave Clarke, David Ingram, Kieran Mansley, and Andrew Hopper. Tripwire: A Synchronisation Primitive for Virtual Memory Mapped Computing. In *Proceedings of the 4th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2000.

[3] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley, 1995.

[4] Jon Postel. Transmission control protocol. RFC 791, 1981.

[5] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of ACM SIGCOMM*, pages 24–35, 1994.

[6] M. Aron and P. Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Transactions on Computer Science*, 18(3), August 2000.

[7] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[8] George Varghese and Anthony Lauck. Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility. *IEEE/ACM Transactions on Networking*, 5(6):824–834, 1996.

[9] Andy Pfiffer. A Study of Linux 2.5 Timer Scalability. White Paper, Open Source Development Labs, CGL Project, `http://developer.osdl.org/~andyp/timers/`.

[10] R. Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10), October 1988.

[11] G. Davison. Calendar P's and Queues. *Communications of the ACM*, 32(10):1241–1242, October 1989.

[12] A. Costello and G. Varghese. Redesigning the BSD Callout and Timer Facilities. Technical Report WUCS-95-23, Washington University in St. Louis, November 1995.

[13] L. Zhang. Why TCP timers don't work well. In *Proceedings of ACM SIGCOMM*, pages 397–405, 1986.

[14] Mohit Aron and Peter Druschel. TCP: Improving Startup Dynamics by Adaptive Timers and Congestion Control. Technical Report TR98-318, Dept. of Computer Science, Rice University, 1998.

[15] Mohit Aron and Peter Druschel. TCP Implementation Enhancements for Improving Webserver Performance. Technical Report TR99-335, Dept. of Computer Science, Rice University, 6, 1999.