Lawn: an Unbound Low Latency Timer Data Structure for Large Scale, High Throughput Systems

Adam Lev-Libfeld

Tamar Labs
Tel-Aviv, Israel
adam@tamarlabs.com

June 25, 2019

Abstract

As demand for Real-Time applications rise among the general public, the importance of enabling large-scale, unbound algorithms to solve conventional problems with low to no latency is critical for product viability[7]. Timer algorithms are prevalent in the core mechanisms behind of operating systems[2], network protocol implementation, stream processing, and several database capabilities. This paper presents a field tested algorithm for a low latency, unbound range timer structure, based upon the well excepted Timing Wheel algorithm. Using a set of queues hashed by TTL, the algorithm allows for a simpler implementation, minimal overhead no overflow and no performance degradation in comparison to the current state of the algorithms under typical use cases.

Index Terms **- Stream Processing, Timing Wheel, Dehydrator, Callout facilities, protocol implementations, Timers, Timer Facilities.**

1 Introduction

2 Model

In similar manner to previous work[8][6][4], the model discussed in this paper shall consist of the following components, each corresponding with a different stage in the life cycle of a timer in the data store:

$$\begin{tikzcd}
 & StartTimer \arrow[d] & \\
 & PerTickBookkeeping \arrow[loop right] \arrow[dl] \arrow[dr] & \\
DeleteTimer & & TimerExpired
\end{tikzcd}$$

StartTimer(TTL,timerId{,Payload}): This routine is called by the client to start a timer that will expire in after the TTL has passed. The client is also expected to supply a *timer ID* in order to distinguish it from other timers in the data store. Some implementations also allow the client to provide a *Payload*, usually some form of a callback action to be performed or data to be returned on timer expiration.

PerTickBookkeeping(): This routine encompasses all the actions, operation and callbacks to be performed as part of timer management and expiration check every interval as determined by the data store granularity. Upon discovery of an outstanding timer to expire *TimerExpired* will be initiated by this routine.

DeleteTimer(timerId): The client may call this utility routine in order to remove from the data store an outstanding timer (corresponding with a given *timer ID*), this is done by calling *TimerExpired* for the requested timer before *PerTickBookkeeping* had marked it to be expired.

TimerExpired(timerId): Internally invoked by either *PerTickBookkeeping* or *DeleteTimer* this routine entails all actions and operations needed in order to remove all traces of the timer corresponding with a given *timer ID* from the data store and invoking the any callbacks that were provided as *Payload* during the *StartTimer* routine.

Since payload and callback behavior varies significantly between different data store implementations, the store of such data can be achieved for $O(1)$ using a simple hash map, and the handling of such callbacks can be done in a discrete, highly (or even embarrassingly) parallel this paper will disregard this aspect of timer stores.

3 Current Solutions

3.1 Lists, Queues & Hash Maps

Being included as an integral part of almost any modern programming language, these basic data structures enable convenient and simple addition of timer management to any software. That said, such simple structures suffer from oversimplification and are appropriate for very unique use cases - where the number of timers it fairly small or the ticks are far enough from one another. Using such implementations for large scale applications will require the grouping of timer producers and consumers into groups small

Operation	**List/Queue**	**Hash Map**
StartTimer	$O(log(n))$	$O(1)$
PerTick	$O(1)$	$O(n)$
DeleteTimer	$O(n)$	$O(1)$
TimerExpired	$O(1)$	$O(1)$

Table 1: Mean Runtime Complexity for timing schemes based on common data structures

3.2 Hashed Timing Wheel

The Hashed Timing Wheel was designed to be an all purpose timer storage solution for a unified system of known size and resolution[9][8][10]. While previous work have shown that Hashed Timing Wheels have optimal run-time complexity, and in ideal conditions are in fact, optimal, real-world implementations would suffer from either being bound by maximal TTL and resolution combination, or would require a costly ($O(n)$) run-time re-build upon of the data structure upon reaching such limits (in [6] it is referred to as "the overflow problem"). For large numbers of timers, producers, or consumers as is common in large scale operations, the solution is commonly to over estimate the needed resolution and/or TTL so to abstain from rebuilding for as long as possible.

Operation	**Worst**	**Mean**
StartTimer	$O(n)$	$O(1)$
PerTick	$O(n)$	$O(1)$
DeleteTimer	$O(1)$	$O(1)$
TimerExpired	$O(1)$	$O(1)$

Table 2: Runtime Complexity for the Hashed Timing Wheel scheme

4 The Lawn Data Structure

4.1 Intended Use Cases

This algorithm was first developed during the writing of a large scale, Stream Processing geographic intersection product using a FastData[5] model. The data structure was to receive inputs from one or more systems that make use of a very limited range of TTLs in proportion to the number of concurrent timers they use.

Assumptions and Constraints : As this algorithm was designed to operate as the core of a dehydration utility for a single FastData application, where TTLs are usually discrete and variance is low it is intended for use under the assumptions that:

$$Unique\ TTL\ Count \ll Concurrent\ Timer\ Count$$

Assuming that most timers will have a TTL from within a small set of options will enable the application of the core concept behind the algorithm - TTL bucketing.

4.1.1 The Data Structure

Lawn is, in it's core, a hash of sorted sets[1], much like Timing Wheel. The main difference is the key used for hashing these sets is the timer TTL. Meaning different timers will be stored in the same set based only on their TTL regardless of arrival time. Within each set the timers are naturally sorted by time of arrival - effectively using the set as a queue. Using this queuing methodology based on TTL, we ensure that whenever a new timer is added to a queue, every other timer that is already there should be expired before the current one, since it is already in the queue and have the same TTL.

The data structure is analogous to blades of grass (hence the name) - each blade grows from the roots up, and periodically (in our case every *Tick*) the overgrown tops of the grass blades (the expired timers) are maintained by mewing the lawn to a desired level (current time).

4.2 Algorithm

4.2.1 Correctness & Completeness

To prove correctness of the algorithm, it should be demonstrated that for each Timer, it is expired (by the calling the *TimerExpired* operation or it) within *Tick* of it's intended TTL. As the algorithm pivots around the TTL bucketing concept, where in each timer is stored exactly once in it's corresponding bucket, and these buckets are independent of each other, it is sufficient to demonstrating correctness for all timers of a bucket, That is:

$$\forall \quad T^{start}, T^{ttl} \in \mathbb{N} \quad \exists \quad T^{stop} \in \mathbb{N} : T^{stop} - T^{start} \approx T^{ttl}$$

Alternatively, we can use the sorting analogy made by G. Varghese et al.[8] to show that given two triggers T_n, T_m:

$$\forall \quad T_n, T_m \quad | \quad T_n^{start} < T_m^{start}, T_n^{ttl} = T_m^{ttl} \quad \exists \quad T_n^{stop}, T_m^{stop} \Rightarrow T_n^{stop} < T_m^{stop}$$

Taking into account that each bucket only contains triggers with the same TTL we can simplify the above:

$$\forall \quad T_n, T_m \quad | \quad T_n^{start} < T_m^{start} \Rightarrow T_n^{stop} < T_m^{stop}$$

[1]These are the TTL 'buckets'

Algorithm 1 The Lawn Data Store

```
Precondition:
 1: $id$ - a unique identifier of a timer.
 2: $ttl$ - a whole product of $TickResolution$ representing the amount of time to wait before triggering the given timer $payload$ action.
 3: $payload$ - the action to perform upon timer expiration.
 4: $current\ time$ - the local time of the system as a whole product of $TickResolution$

 5: function InitLawn()
 6:     TTLHash ← new empty hash set
 7:     TimerHash ← new empty hash set
 8:     $closest\ expiration \leftarrow 0$

 9: function StartTimer($id$, $ttl$, $payload$)
10:     $endtime \leftarrow$ current time + $ttl$
11:     $T \leftarrow (endtime, ttl, id, payload)$
12:     TimerHash[$id$] ← $T$
13:     if $ttl \notin$ TTLHash then
14:         TTLHash[$ttl$] ← new empty queue
15:     TTLHash[$ttl$].insert($T$)
16:     if $endtime < closest\ expiration$ then
17:         $closest\ expiration \leftarrow endtime$

18: function PerTickBookkeeping()
19:     if current time < $closest\ expiration$ then
20:         return
21:     for $queue \in$TTLHash do
22:         $T \leftarrow$ peek($queue$)
23:         while $T_{endtime}$ < current time do
24:             TimerExpired($T_{id}$)
25:             $T \leftarrow$ peek($queue$)
26:         if $closest\ expiration = 0$
    or $T_{endtime} < closest\ expiration$ then
27:             $closest\ expiration \leftarrow T_{endtime}$

28: function TimerExpired($id$)
29:     $T \leftarrow$ TimerHash[$id$]
30:     DeleteTimer($T$)
31:     do $T_{payload}$

32: function DeleteTimer($id$)
33:     $T \leftarrow$ TimerHash[$id$]
34:     if $T_{endtime} = closest\ expiration$ then
35:         $closest\ expiration \leftarrow 0$
36:     TTLHash[$T_{ttl}$].remove($T$)
37:     TimerHash.remove($T$)
38:     if TTLHash[$T_{ttl}$] is empty then
39:         TTLHash.remove[$ttl$]
```

Which, due to the bucket being a sorted set, ordered by T^{start} and triggers being expired by bucket order from old to new is self-evident, and we arrived at a proof.

4.2.2 Space and Runtime Complexity

The Lawn data structure is dense by design, as every timer is stored exactly once, a new trigger will add at most a single TTL bucket and empty TTL buckets are always removed, the data structure footprint will only grow linearly with the number of timers. Hence, overall space complexity is linear to the number of timers ($O(n)$).

Operation	**Worst**	**Mean**
StartTimer	$O(1)$	$O(1)$
PerTick	$O(n)$	$O(t \sim 1)$
DeleteTimer	$O(1)$	$O(1)$
TimerExpired	$O(1)$	$O(1)$

Table 3: Runtime Complexity for the Lawn scheme

Since the *PerTickBookkeeping* routine of Lawn iterates over the top item of all known TTL buckets on every expiration cycle (where at least one timer is expected to expire), it's mean case runtime is linear to t (the number of different TTLs) and seems to be lacking even in comparison to more primitive implementations of timer storage. That said, with an added assumption that the TTL set size is roughly constant over time, or at worst asymptotically smaller then the number of timers, we can regard this operation as constant time.

This assumption is valid in our case as it is derived from the needs of the algorithm users, these being other computer systems, which often have either a single TTL used repeatedly, their TTLs chosen from a list of hard coded values or derived from a simple mathematical operation (sliding windows are a good example of this method, using fixed increments or powers of 2 to determine TTLs etc.). Computer systems which are using highly variable TTL values are suitable for usage with this timer algorithm only under specific circumstances (such as multi-worker expiration system as described below).

Space complexity is $O(n)$ since at worst case each timer is stored in its own bucket alongside a single entry in the timer hash. To compare, this spatial footprint is bound from above by that of the Hashed Hierarchical Time Wheel, as due to it's multi-level structure a single timer can be pointed at by a chain of hierarchical wheels, increasing its overall space requirement.

5 Comparison and Reflection

While general use systems, aggregating timers from several sources with, or applications with highly predictable needs may benefit from the relative stability of run-time provided by Timer Wheel (let alone the fact that it has been shown to be an optimal solution in terms of run-time complexity) Large scale machine serving systems would suffer from the overflow problem when faced with unpredictable scale of usage. This is handled in Lawn by a "slow and steady" approach, optimizing for specific use cases.

Designed for large scale, high throughput systems, Lawn has displayed beyond state of the art performance for systems complying with its core assumptions of multi worker, hi-frequency, hi-timer-count with low TTL variance applications.

Operation	**Timer Wheel**	**Lawn**
StartTimer	$O(1)$	$O(1)$
PerTick	$O(1)$	$\boldsymbol{O(t \sim 1)}$
DeleteTimer	$O(1)$	$O(1)$
TimerExpired	$O(1)$	$O(1)$
overflow	$\boldsymbol{O(n)}$	$O(1)$
space	$O(n)$	$O(n)$

Table 4: Mean Runtime Complexity comparison

5.0.1 An Algorithmic View of Multiprocessing

Unlike the state-of-the-art Timer Wheel algorithm, Lawn enables the simultaneous timer handling and bookkeeping by splitting the buckets between several worker processes/threads, adding or removing workers as needed. This method enables the usage of the Lawn algorithm in highly parallel applications and does not requires a use of semaphores than other synchronization mechanisms within the bucket level.

5.0.2 Known implementations of Lawn in the wild

1. Redis Internals [3] - a high performance in memory key value store - uses Lawn implementation for streams and other internal timers.

2. Mellanox RDMA timers for an undisclosed Infiniband subsystem.

3. User specific rate limiting timers for client device power consumption optimization[1].

4. *clib* Timer management utility lib.

6 Conclusion

Lawn is a simplified overflow-free algorithm that displays near-optimal results for use cases involving many (millions) concurrent timers from large scale (tens of thousands) of independent machine systems. The algorithm is currently deployed and in use by several organizations under real-world load, all reporting satisfactory results.

References

[1] Adam Matan, A. L.-L. Vioozer geotagging system, 2017. More information at https://www.vioozer.com

[2] Costello, A., Costello, A. M., Varghese, G., and Varghese, G. Redesigning the bsd callout and timer facilities. Tech. rep., 1995.

[3] Iberra, S. Redis. Available at https://redis.io/

[4] Jones, D. W. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM 29*, 4 (Apr. 1986), 300–311.

[5] Lev-Libfeld, A., and Margolin, A. Fast data - moving beyond from big datas mapreduce. *GeoPython 1* (June 2016), 3–6.

[6] Sleator, D., and Brown, R. Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem.

[7] Vany, A. D. Uncertainty, waiting time, and capacity utilization: A stochastic theory of product quality. *Journal of Political Economy 84*, 3 (1976), 523–541.

[8] Varghese, G., and Lauck, A. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM transactions on networking 5*, 6 (December 1997), 824–834.

[9] Varghese, G., and Lauck, T. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. *SIGOPS Oper. Syst. Rev. 21*, 5 (Nov. 1987), 25–38.

[10] Welch, N. timeout.c: Tickless hierarchical timing wheel implementation. Available at http://25thandclement.com/ william/projects/timeout.c.html