Timer Lawn: Solving the Timer Wheel Overflow Problem for Large Scale and High Throughput Systems

ADAM LEV-LIBFELD

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

***Abstract*—As the usage of Real-Time applications and algorithms rises, so does the importance of enabling large-scale, unbound algorithms to solve conventional problems with low to no latency becomes critical for product viability[1], [2]. Timer algorithms are prevalent in the core mechanisms behind operating systems[3], network protocol implementation, real-time and stream processing, and several database capabilities. This paper presents a field-tested algorithm for low latency, unbound timer data structure, that improves upon the well excepted Timing Wheel algorithm. Using a set of queues mapped to by TTL instead of expiration time, the algorithm allows for a simpler implementation, minimal overhead no overflow and no performance degradation in comparison to the current state of the art.**

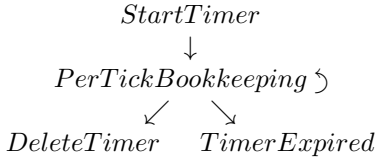
***Keywords*—Stream Processing, Timing Wheel, Dehydrator, Callout facilities, protocol implementations, Timers, Timer Facilities, Lawn.**

I. INTRODUCTION

This paper presents a theoretical analysis of a timer data-structure designed for use with hi-throughput computer systems called Lawn. In this paper, it will be shown that although the current state of the art algorithm is theoretically optimal, under some use cases (namely where max TTL is unpredictable, or the needed Tick resolution may change) it is under-performing due to the overflow problem, which the algorithm presented here addresses. Utilizing Lawn may assist in improving overall performance and flexibility in TTL and Tick resolution with no need for any prior knowledge of the using system apart from it not utilizing non-discrete stochastic values for timer TTLs.

A. Model

In a similar manner to previous work[4], [5], [6], 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:



*a) **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.

*b) **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.

*c) **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 *PerTick-Bookkeeping* had marked it to be expired.

*d) **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.

B. callback run-time complexity

Since payload and callback run-time complexity 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.

II. CURRENT SOLUTIONS

A. List and Tree Based Schemes

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 over-simplification and are appropriate for very unique use cases - where the number of timers it fairly small or the ticks are

Operation	List Based	Tree Based
StartTimer	$O(n)$	$O(log(n))$
PerTick	$O(1)$	$O(1)$
DeleteTimer	$O(n)$	$O(log(n))$

TABLE I
RUNTIME COMPLEXITY FOR COMMON DATA STRUCTURE BASED SCHEMES

far enough from one another. Using such implementations for large scale applications will require the grouping of timer producers and consumers into groups small

B. Hashed Timing Wheel

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

TABLE II
RUNTIME COMPLEXITY FOR THE HASHED TIMING WHEEL SCHEME

The Hashed Timing Wheel was designed to be an all-purpose timer storage solution for a unified system of known size and resolution[7], [4], [8], [9], [10]. While previous work has 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 [5] it is referred to as "the overflow problem"). For large numbers of timers, producers, or consumers as is common in large scale operations, the simplest and mose effective solution is to overestimate the needed resolution and/or TTL so to abstain from rebuilding (resizing) for as long as possible.

C. Overflow Algorithms

in order to handle overflows correctly there are two valid options:

1) increase current wheel slots (Alg. 1): Resize the overflowing wheel so it would include a larger number of slots into the future. This change entails a worst case cost of $O(n)$ upon any addition of an overflowing timer.

2) in place addition of cycle count for each item (Alg. 2): Each element in a slot is also stored with the number of wheel cycles needed before element expiration. This change entails a worst case cost of $O(n)$ for all *PerTickBookkeeping* operation.

III. THE LAWN DATA STRUCTURE

A. Intended Use Cases

This algorithm was first developed during the writing of a large scale, Stream Processing geographic intersection product[11] using a FastData[12] 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.

Algorithm 1 Timer Wheel Resize

```
1:  function StartTimer(id, ttl, payload)
2:      if ttl > len(Wheel) then
3:          MissingSlots ← ttl − len(Wheel)
4:          ExtWheel ← array of length len(Wheel) + MissingSlots
5:          for Slot,SlotIndex in Wheel do
6:              NewSlotIndex ← (len(Wheel)+SlotIndex) mod len(ExtWheel)
7:              ExtWheel[NewSlotIndex] ← Slot
8:          Wheel ← ExtWheel
9:      slot ← (current time+ttl) mod len(Wheel)
10:     append payload to Wheel[slot]
11:     TimerHash[id] ← (payload, slot)
```

Algorithm 2 Timer Wheel Multi-Pass (in-place)

```
1:  function StartTimer(id, ttl, payload)
2:      T ← (payload=payload, cycles=(current time+ttl) div len(Wheel))
3:      slot ← (current time+ttl) mod len(Wheel)
4:      append T to Wheel[slot]
5:      TimerHash[id] ← T

6:  function PerTickBookkeeping
7:      current slot ← current slot + 1
8:      for element in current slot do
9:          element.cycles ← element.cycles-1
10:         if element.cycles ¡= 0 then
11:             TimerExpired(T_id)
```

a) Assumptions and Constraints: As this algorithm was originally 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.

1) The Data Structure: Lawn is, at its 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 (as can be seen in fig. 1). 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 would be expired before the current one, since it is already in the queue and have the same TTL.

[1]These are the TTL 'buckets'

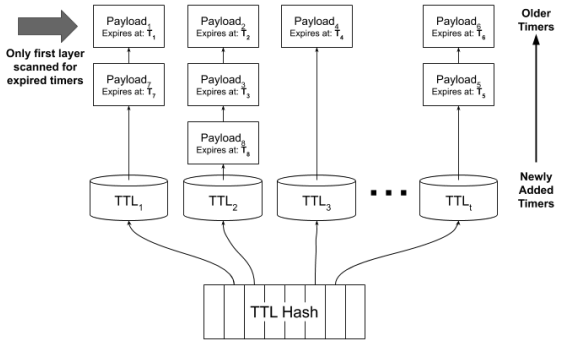


Fig. 1. A schematic view of the data structure components.

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 mowing the lawn to the desired level (current time).

B. Algorithm

1) Correctness & Completeness: To prove the algorithm's correctness, it should be demonstrated that for each Timer t with TTL ttl, *TimerExpired* operation is called on t within *Tick* of ttl. Since the algorithm pivots around the TTL bucketing concept, wherein each timer is stored exactly once in its 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.[4] 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:

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

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.

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 ($O(n)$).

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

Algorithm 3 The Lawn Data Store

Precondition:

- id - a unique identifier of a timer.
- ttl - a whole product of $TickResolution$ representing the amount of time to wait before triggering the given timer $payload$ action.
- $payload$ - the action to perform upon timer expiration.
- *current time* - the local time of the system as a whole product of $TickResolution$

function INITLAWN()
    TTLHash ← new empty hash set
    TimerHash ← new empty hash set
    $closest\ expiration \leftarrow 0$

function STARTTIMER(id, ttl, $payload$)
    $endtime \leftarrow$ current time + ttl
    $T \leftarrow (endtime, ttl, id, payload)$
    TimerHash[id] ← T
    if $ttl \notin$ TTLHash **then**
        TTLHash[ttl] ← new empty queue
    TTLHash[ttl].**insert**(T)
    if $endtime < closest\ expiration$ **then**
        $closest\ expiration \leftarrow endtime$

function PERTICKBOOKKEEPING()
    if current time $< closest\ expiration$ **then**
        return
    for $queue \in$TTLHash **do**
        $T \leftarrow$ **peek**($queue$)
        while $T_{endtime} <$ current time **do**
            TimerExpired(T_{id})
            $T \leftarrow$ **peek**($queue$)
        if $closest\ expiration$ = 0
    or $T_{endtime} < closest\ expiration$ **then**
            $closest\ expiration \leftarrow T_{endtime}$

function TIMEREXPIRED(id)
    $T \leftarrow$ TimerHash[id]
    DeleteTimer(T)
    do $T_{payload}$

function DELETETIMER(id)
    $T \leftarrow$ TimerHash[id]
    if $T_{endtime} = closest\ expiration$ **then**
        $closest\ expiration \leftarrow 0$
    TTLHash[T_{ttl}].**remove**(T)
    TimerHash.**remove**(T)
    if TTLHash[T_{ttl}] is empty **then**
        TTLHash.**remove**[ttl]

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

TABLE III
RUNTIME COMPLEXITY FOR THE LAWN SCHEME

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 are 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).

a) 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 Timing 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.

IV. 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 Timing 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 a multi-worker, hi-frequency, hi-timer-count with low TTL variance applications.

Operation	TW (Resize)	TW (Multi-Pass)	Lawn
StartTimer	$\boldsymbol{O(n)}$[2]	$O(1)$	$O(1)$
PerTick	$O(1)$	$O(1)$	$\boldsymbol{O(t \sim 1)}$
DeleteTimer	$O(1)$	$O(1)$	$O(1)$
TimerExpired	$O(1)$	$\boldsymbol{O(n)}$[3]	$O(1)$
space	$O(n)$	$O(n)$	$O(n)$

TABLE IV
RUNTIME COMPLEXITY COMPARISON

[2]on overflow, else O(1)

[3]see footnote 2

A. A 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 or even different machines altogether, adding or removing workers as needed. This method enables usage of the Lawn algorithm in large scale or highly parallel applications and does not require the use of semaphores than other synchronization mechanisms within a single bucket.

B. Known implementations of Lawn

As mentioned in the body of this paper, the Lawn algorithm has already been tested and deployed in several programming languages by different organizations. Some of these implementations were developed by or in tandem with the author of this paper and some with his permission all with reported improvement in performance. The algorithm is free to use and the source code for many of these implementations has been published under an open source license.

1) Redis Internals [13] - a high performance in-memory key-value store - uses Lawn implementation for streams and other internal timers.
2) ReDe event dehydrator Redis module[14].
3) Mellanox RDMA timers for an Unified Communication X[15] .
4) User specific rate limiting-timers for client device power consumption optimization[11].
5) *clib* Timer management utility lib.

V. CONCLUSION

Lawn is a simplified overflow-free algorithm that displays near-optimal results for use cases involving many (tens of millions) concurrent timers from large scale (tens of thousands) independent machine systems. Using Lawn is an elegant solution to the overflow problem in Timer Wheel and other current algorithms, enabling simpler use of available cores and requiring less knowledge about the usage pattern with the only requirement is for the TTLs being discrete and their number being bound, but not knowing what the bound on max TTL is. The algorithm is currently deployed and in use by several organizations under real-world load, all reporting satisfactory results.

REFERENCES

[1] B. P. Douglass, *Real-Time Design Patterns*. Addison-Wesley, 2003. chapter 5: Concurrency Patterns (Rendezvous).

[2] A. D. Vany, "Uncertainty, waiting time, and capacity utilization: A stochastic theory of product quality," *Journal of Political Economy*, vol. 84, no. 3, pp. 523–541, 1976.

[3] A. Costello, A. M. Costello, G. Varghese, and G. Varghese, "Redesigning the bsd callout and timer facilities," tech. rep., Washington University in St Louis, 1995.

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

[5] D. Sleator and R. Brown, "Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem," *Comunications of the ACM*, vol. 31, Oct. 1988.

[6] D. W. Jones, "An empirical comparison of priority-queue and event-set implementations," *Commun. ACM*, vol. 29, pp. 300–311, Apr. 1986.

[7] G. Varghese and T. Lauck, “Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility,” *SIGOPS Oper. Syst. Rev.*, vol. 21, pp. 25–38, Nov. 1987.
[8] N. Welch, “timeout.c: Tickless hierarchical timing wheel implementation.” http://25thandclement.com/~william/projects/timeout.c.html.
[9] H. Hu, “Large-scale timer management.” US Patent no. US8307030B1.
[10] Z. ZhouIvan, “On-demand scalable timer wheel.” US Patent no. US20140298073A1.
[11] A. M. Adam Matan, Adam Lev-Libfeld, “Vioozer geo-tagging system.” http://www.vioozer.com/, 2017.
[12] A. Lev-Libfeld and A. Margolin, “Fast data - moving beyond from big datas map-reduce,” *GeoPython*, vol. 1, pp. 3–6, June 2016.
[13] S. Iberra, “Redis.” https://redis.io/, 2009.
[14] A. Lev-Libfeld, “A redis element dehydration module.” www.tamarlabs.com/ReDe/, 2017.
[15] Y. Ittigin et.al., “Unified Communication X” https://github.com/openucx/ucx/tree/master/src/ucs/time, 2017.