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 15, 2018

Abstract

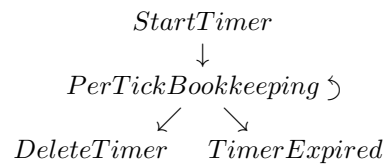
As demand for Real-Time applications rise among the general public, the importance of enabling large-scale, unbound algorithms to solve conventional problems with low to no latency is critical for product viability Timers algorithms are prevalent in the core mechanisms behind of operating systems, network protocol implementation, stream processing, and several database capabilities This paper presents an 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 and no degradation in performance 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[3], 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:



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 Method

In this paper we shall compare the algorithms in three manners:

1. Theoretical analysis
2. High level Python implementation (version 2.7.14)
3. Low level, highly optimized C implementation (compiled with gcc version 7.2.0)

Implemented using both Python and C[1]:

Results (Python 2.7 and C) schemes algorithms where compared using three distinct techniques:

this is how i will compare things

4 Previous Work in light of Scale

4.1 Lists, Queues & Hash Maps

simple and popular

4.2 Hashed Timing Wheel

best, but bounded

Assumptions and Constraints : The Hashed Timing Wheel was designed to be an all purpose timer storage solution for

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

Table 1: Runtime Complexity for the Hashed Timing Wheel scheme

5 The Lawn Data Structure

5.1 Intended Use Cases

This algorithm was first developed during the writing of a large scale, Stream Processing geographic intersection product using a FastData[2] 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.

5.1.1 The Data Structure

Lawn is, in it's core, a hash of sorted sets[2], 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.

[1] Source code is available here[1]. **Python** version 2.7.14 and **C** compiled with gcc version 7.2.0.

[2] These are the TTL 'buckets'

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).

5.2 Algorithm

5.2.1 Correctness

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.[3] 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.

5.2.2 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

Algorithm 1 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* action.
payload_action - 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 ← 0

function StartTimer(id, ttl, payload_action)
    endtime ← current time + ttl
    T ← (endtime, ttl, id, payload_action)
    TimerHash[id] ← T
    if ttl ∉ TTLHash then
        TTLHash[ttl] ← new empty queue
    TTLHash[ttl].insert(T)
    if endtime < closest expiration then
        closest expiration ← endtime

function PerTickBookkeeping()
    if current time < closest expiration then
        return
    for q ∈TTLHash do
        T ← peek(q)
        while T_endtime < current time do
            TimerExpired(T_id)
            T ← peek(q)
        if closest expiration = 0
    or T_endtime < closest expiration then
            closest expiration ← T_endtime

function TimerExpired(id)
    T ← TimerHash[id]
    DeleteTimer(T)
    do T_payload_action

function DeleteTimer(id)
    T ← TimerHash[id]
    if T_endtime = closest expiration then
        closest expiration ← 0
    TTLHash[T_ttl].remove(T)
    TimerHash.remove(T)
    if TTLHash[T_ttl] is empty then
        TTLHash.remove[ttl]
```

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 2: 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, or at worst asymptotically smaller then the number of timers[3], we can regard this operation as constant time.

Space complexity is $O(n)$ as , since and possibly an entry in the timer hash.

6 Comparison and Reflection

to compare: hashed heap vs Hashed Timing Wheel vs Lawn

7 An Algorithmic View

7.0.1 Multiprocessing

Highly parallel (splitting by queue) and does not requires a lighter use of semaphores than other queue based schemes

[3]The assumptions is derived from this data structure being designed around serving 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)

8 Conclusion

References

[1] Lawn github repository.

[2] Adam Lev-Libfeld and Alexander Margolin. Fast data - moving beyond from big datas map-reduce. *GeoPython*, 1:3–6, June 2016.

[3] 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, December 1997.