Lawn: an Unbound Low Latency Timer Data Structures for Large Scale Systems

Adam Lev-Libfeld

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

January 11, 2018

Abstract

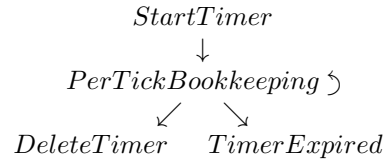
As demand for Real-Time applications rise among the general public, the importance of enabling large scale, unbound algorithms to solve conventional problems with low to no latency is critical for product viability Timers algorithms are prevalent in the core mechanisms behind of operating systems, network protocol implementation, stream processing and several data base 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 simpler implementation, minimal overhead and no degradation in performance in comparison to current state of the algorithms under typical use cases.

***Index Terms* - Stream Processing, Timer 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 routing 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.

2.1 Performance Notation

the following notation would be used for all runtime and space complexity demarcations:

n : The total number of timers in the data store.

t : The total number of different TTLs in the data store.

for breverty and readabuility sake this paper will notate the

3 Method

This paper will present results to compare the algorithms calculated using both Python and C[1]:

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

1. Algorithmic view
2. Performance testing - using Python code
3. Performance testing - using C code

this is how i will compare things

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

4 Previous Work in light of Scale

4.1 Lists, Queues & Hash Maps

simple and popular

4.2 Timer Wheel

best, but bounded

4.2.1 Assumptions and Constraints :

The

4.2.2 Algorithm :

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

Table 1: Runtime Complexity for the Hashed Timer Wheel scheme

4.2.3 Variants :

Hashed : prevalent for high load systems, including Unix kernel

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 system only that has a very limited range of TTLs.

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 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.

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:

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

[2]These are the TTL 'buckets'

Algorithm 1 The Lawn Data Store

Precondition: *id* - a unique identifier of the Timer.
ttl - a whole product of *TickResolution* representing the amount of time to wait before triggering the given timer *Payload* action.

```
1:  function StartTimer(id, ttl, payload)
2:      endtime ← current time + ttl
3:      T ← (endtime, ttl, id, payload)
4:      TimerHash[id] ← T
5:      if ttl ∉ TTLHash then
6:          TTLHash[ttl] ← empty queue
7:      TTLHash[ttl].insert(T)

8:  function PerTickBookkeeping()
9:      for q ∈TTLHash do
10:         T ← peek(q)
11:         while T_endtime < current time do
12:             TimerExpired(T_id)
13:             T ← peek(q)

14: function DeleteTimer(id)
15:     T ← TimerHash[id]
16:     TTLHash[T_ttl].remove(T)
17:     TimerHash.remove(T)
18:     if TTLHash[T_ttl] is empty then
19:         TTLHash.remove[ttl]

20: function TimerExpired(id)
21:     T ← TimerHash[id]
22:     DeleteTimer(T)
23:     return T_payload
```

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

As this algorithm iterates over all active TTLs (TTLs that are used by timers which did not expire yet) it's works case runtime is lacking even in comparison to more primitive implementations of timer storage. That said, with the added assumption that the TTL set size asymptotically smaller then the number of timers (in many cases millions of timers and tens of TTLs), we can regard this operation as constant time. Moreover, using an active TTLs and known timers list (containing pointers to the allocated memory of the trigger) and a doubly linked list implementation for the queues, it is possible to get runtime complexity to be:

Operation	**Worst**	**Average**
StartTimer	$O(1)$	$O(1)$
PerTick	$O(t)$	$O(1)$
DeleteTimer	$O(1)$	$O(1)$
TimerExpired	$O(1)$	$O(1)$

Table 2: Runtime Complexity for the Lawn scheme

Space complexity is $O(n)$ as the data structure footprint grows linearly with the number of timers, since every new trigger adds at most a single new queue and possibly an entry in the timer hash.

6 Comparison and Reflection

7 An Algorithmic View

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.