Published in *Information Processing Letters*, 54 (3): 139-45, May 1995, Ed. Elsevier-North Holland.

# Efficient Management of Multiple Outstanding Timeouts

Guillermo A. Alvarez* Marcelo O. Fernández†

**Abstract:**
The timer management service maintains a set of independent timeout intervals. Client applications initiate and terminate timers, and are notified when timeouts occur. We present and prove the correctness of a scheme that implements this service, and compares favorably with previously existing solutions.
*Key words:* Algorithms; Concurrency; Operating Systems; Real-time Systems

## 1 Introduction

In the context of computer systems, there is a frequent need of maintaining a possibly large set of independent timeout intervals. As an example of this fact, consider the sliding window Go-Back-N network protocol [7, 6]. In this case each frame not yet acknowledged has its own timeout period, that triggers the retransmissions made by the sender side at the data link level. A similar policy was adopted for the TCP protocol [3], yielding an adaptive behavior based on dynamic adjustment of timeout periods. Another example can be found in the UNIX operating system [5], where the kernel schedules functions that must be invoked at certain times. A Clock-Handler process [2], activated periodically, determines which functions (if any) need to be started during the current clock tick. The UNIX `alarm` system call relies on this mechanism to send a signal to the calling process after the specified number of real-time seconds.

The *timer management service* allows client applications to start timers initially set to arbitrary intervals, and notifies their expiration back to the clients when the timeout intervals elapse. This service is a general building block for the implementation of software systems whose behavior depends on the passage of time.

In this work we present a group of algorithms and data structures that solve the general timer management problem. For a set of $n$ simultaneously outstanding timeouts, the asymptotic worst-case running time is $O(\log n)$ using $O(n)$ space. Our scheme stores timeout intervals in the data structure, as opposed to timestamps (moments of expiration) that grow without bound along the lifetime of the system. No assumptions are made about the usage pattern from the client applications. The core of this work was devised during the definition of the communication and synchronization protocols for the DREAM parallel programming environment [1].

The structure of this paper is as follows. Basic notation and preliminary assumptions are introduced in Section 2, together with some motivation. Section 3 contains the timer management algorithms, and the analysis of their running times in terms of the proposed set of data structures. Section 4 contains the formal proofs of correctness of the algorithms. A comparison with previously existing solutions is reported in Section 5. Finally, in Section 6 we draw some conclusions on the preceding results.

*Dept. of Computer Science and Engineering, University of California – San Diego, La Jolla, CA 92093-0114. Email: galvarez@cs.ucsd.edu

†IBM Corporation, Ing.Butty 275, 1300 Buenos Aires, Argentina.

## 2 Motivation and Basic Definitions

A global *Clock* keeps track of the time in units called *ticks*. The *Clock* is incremented by one at the end of each time quantum. A timer $t$ consists of two components: $t_{Exp}$ and $t_{Value}$. The former is the value that *Clock* will have at the timeout of $t$, i.e. the value of *Clock* when $t$ is started plus the number of ticks that the timer must sleep. The latter component is used in our algorithms to evaluate the time remaining before $t$ times out. It turns out that $t_{Exp}$ needs not be maintained in an actual implementation of the algorithms, as explained below in Section 3. However, just for the sake of clarity, it will be used when proving their correctness.

All timers are contained in a system-wide pool. A pool $P$ is defined by $P_{Timers}$ and $P_{Com}$, where $P_{Timers}$ is the set of timers present in the pool. $P_{Com}$ is a scalar pool-wide value that keeps track of the relative offset of the timers contained in the pool with respect to the real-time clock. Timers with the same $t_{Exp}$ value are clustered in groups within the pool.

Two main operations are defined on pools. `Insert(Time, P)` starts a timer that will expire in `Time` ticks into a pool `P`. `Update(P)` reflects on the data structures the passage of a time quantum; it returns the timers that have just reached the timeout. For now, we assume that `Update` is invoked at every clock tick by the underlying operating system or clock service, and that `Insert` is invoked by each client that wants to start a timer. We do not address the system-specific issues of how to guarantee that `Update` is invoked in a timely manner, or how clients are notified of the expiration of timeouts. This work is centered on the algorithmic aspects of the problem.

Let $,(t)$ be the time needed for a timer $t$ to reach the timeout. Given a pool $P$, let $Min(P)$ be the group of timers that have the minimum value of $,$ among all the timers present in the pool. Every operation defined on a pool must preserve the invariant condition

$$(\forall t \in P_{Timers})(t_{Exp} - Clock = ,(t))$$

The function $,$ is defined as follows:

$$,(t) = \begin{cases} t_{Value} & \text{, if } t \in Min(P) \\ t_{Value} + Min(P)_{Value} - P_{Com} & \text{, if } t \notin Min(P) \end{cases}$$

where $Min(P)_{Value} = t'_{Value}$ for any $t' \in Min(P)$[1]. We define $Min(P)_{Exp}$ in the same way.

According to the previous definition, if $Min(P)_{Value}$ is decremented, then the value of $,$ on every element of $P$ also decreases by one tick (if no timer expires, the execution of `Update` does not change the identity of the minimum element of $P$). The main motivation for our definition of $,$ is to perform this decrement *only on* $Min(P)$, as opposed to on every element, so that the cost of the update operation depends only on the costs of finding the minimum element and (possibly) deleting it from the structure. Another important characteristic is that $,(t)$ does not depend on the *Value* of all the timers that will expire before $t$. Therefore, when a new timer is inserted, its *Value* can be computed efficiently (just in terms of the minimum element and of $P_{Com}$).

Now, the invariant condition becomes

$$\begin{aligned}(\forall t \in P_{Timers})\ ((t \in Min(P) \Rightarrow t_{Exp} - Clock = t_{Value}) \wedge \\ (t \notin Min(P) \Rightarrow t_{Exp} - Clock = t_{Value} + Min(P)_{Value} - P_{Com}))\end{aligned} \quad (1)$$

[1] Let $t$ and $t'$ be timers in $Min(P)$, then it holds that $t_{Value} = t'_{Value}$ and $t_{Exp} = t'_{Exp}$.

```
Update(P):

if (P_Timers is not EMPTY)
   then Min(P)_Value = Min(P)_Value - 1
        if (Min(P)_Value = 0)
           then /* Timer(s) Expired */
                Report_Timeout(Min(P))
                Erase_Min(P)
                if (P_Timers is EMPTY)
                   then P_Com = 0
                   else NewCom = Min(P)_Value
                        NewMin = Min(P)_Value - P_Com
                        P_Com = NewCom
                        Min(P)_Value = NewMin
```

Figure 1: Algorithm `Update`

As an example, consider the pool $< P_{Com} = 0, P_{Timers} = \{< t_{Exp} = 10, t_{Value} = 10 >\} >$ at instant $Clock = 0$, that contains a single timer that will expire in 10 time units. If a timeout of 5 time units is inserted, the pool will become $< P_{Com} = -5, P_{Timers} = \{< t_{Exp} = 5, t_{Value} = 5 > , < t_{Exp} = 10, t_{Value} = 0 >\} >$; here a new timer was inserted, and since its expiration will precede the expiration of the previous minimum, $P_{Com}$ and the previous minimum's $t_{Value}$ were updated to preserve the invariant.

## 3 Pseudocode and Analysis

In this section we describe the algorithms for timer management. We consider different data structures for their implementation, deriving the time and space costs for each of them. Two auxiliary functions are used in the pseudocode descriptions: the function `Min` retrieves from `P` all the timers $t$ such that $(t_{Exp} - Clock)$ is minimum, and `Erase_Min` deletes from `P` all the timers in `Min(P)`.

Figure 1 depicts the algorithm `Update`, which is executed periodically. Assume that no timer expires during a given execution of the algorithm; then, since only the minimum element is decremented, it will still be the minimum when the update is over. If there are expirations, the modifications made to the new minimum element do not change the minimum's identity either. Therefore, by definition of , and since (1) holds, timeouts occur after the amount of clock ticks originally specified when inserting the timers.

Figure 2 contains the code for the `Insert` operation. This function inserts a timer into the pool and returns a timer identifier (*timer_id*) as a handle for the new timer. Several auxiliary functions are referenced in Figure 2. `Search_Timers` finds the timers in the pool (if any) that will expire at the same time that the one being inserted. `Add_New_Timer` adds a timer into the pool, provided that the corresponding group already exists. `Insert_New_Timer` inserts a new timer into the pool, the second parameter in this function being the timer's *Value* component. This function might change $Min(P)_{Value}$ — when a timer with minimum , is inserted.

Let $n$ be the total number of timers present in the pool. Let $N$ be the total number of timer groups present in the pool (recall that timers with the same moment of expiration are clustered in the same group, whether they are inserted during the same tick or not). There are several choices for implementing the algorithms in an efficient way.

```
Insert(Time, P):

if (P_Timers is EMPTY)
   then MinVal = 0
   else MinVal = Min(P)_Value
NewId = Generate_Timer_Id()
case
   Time > MinVal:  InsVal = Time - MinVal + P_Com
                   Position = Search_Timers(InsVal, P_Timers)
                   if (Position ≠ NIL)
                     then Add_New_Timer(NewId, Position)
                     else Insert_New_Timer(NewId, InsVal, P_Timers)
   Time = MinVal:  Add_New_Timer(NewId, Min(P))
   Time < MinVal:  Min(P)_Value = P_Com
                   P_Com = Time - MinVal + P_Com
                   Insert_New_Timer(NewId, Time, P_Timers)
endcase
return(NewId)
```

Figure 2: Algorithm `Insert`

A well-known result states that a search tree can be augmented with a $\Theta(1)$ `Min()` operation, without affecting the logarithmic worst-case cost of insertions, searches and deletions. A Red-Black tree [4] with a `Min()` operation can be used to keep a group of timers at each node. Linked lists can store the members of each group to allow for $\Theta(1)$ insertions into a given group (it is not necessary to keep the timers sorted within a group). Conceptually, the key used to sort the nodes of the tree is the value of the , function for the timers contained in each node. Since , $(t) = t_{Exp} - Clock$ and *Clock* is the same for all the timers, we can keep only $t_{Exp}$ in each node and use this value like the key. However, , can be evaluated at each visited node during searches, insertions or deletions: if the node is $Min(P)$ then , is $t_{Value}$, otherwise we evaluate , in constant time using $Min(P)_{Value}$ and $P_{Com}$. Therefore, $t_{Exp}$ needs not be stored at all in the nodes of the tree. Since $t_{Exp}$ grows continuously during the life of the system (we assume no clock wraparound), $t_{Value}$ can be stored using less bits, particularly for timers with a high resolution but comparatively short lifespan. Thus, the total amount of memory used by the data structures depends on the actual intervals present in them, and not on the moments in which the timers were started.

With these data structures, `Update` runs in $\Theta(1)$ if no timer expires. The worst case occurs only when some timer group expires, which implies the deletion of $Min(P)$ from the tree. This can be done in $\Theta(\log N)$ time for the Red-Black tree. Regarding the `Insert` operation, when $Time = Minval$ a new timer_id is added to `Min(P)` in $\Theta(1)$. Otherwise, the tree must be traversed in order to add the timer into an existent group or to create a new group with it. These operations run in $\Theta(\log N)$.

Another operation that can be defined on a pool is `Delete(TimerId, P)`, to stop a timer given its timer_id. The most efficient alternative for accommodating deletions is to define the timer_id so that it contains a pointer to the timer cell within the corresponding group. Timers in the same group are organized as a doubly linked list, to allow deletions in $\Theta(1)$ given a pointer to a cell. If the only timer of a group is erased, the corresponding node has to be deleted from the tree in $\Theta(\log N)$. The `Delete` operation is not essential, but can be implemented for completeness if we don't want the timeouts to occur (instead of ignoring them).

The particular case in which no two timers ever expire at the same time (i.e. all the timer groups are singletons) admits an efficient implementation using Fibonacci heaps [4]. This situation can occur when the timeout period is constant and at most one timer is inserted between two clock ticks. For this choice of data structures, the running time of all the cases of `Insert` drops to $\Theta(1)$, because we no longer need to search the whole pool for a timer with the same $t_{Exp}$. However, this is not worst-case cost, but amortized over a sequence of operations.

Therefore, we conclude that with augmented Red-Black trees `Insert`, `Update`, and `Delete` run in worst-case time $O(\log N)$, using $O(n)$ total space. The average size of timer groups is difficult to estimate, for it depends on both the timeout periods inserted and the moment of each insertion. However, since $n \geq N$, the running time is also $O(\log n)$ in the worst case. For the special case $N = n$, Fibonacci heaps give an amortized cost $\Theta(1)$ for `Insert`. It is possible to use certain hardware features if the platform provides them, to reduce the workload on the main processor and the interrupt overhead [8]. For instance, if the hardware supports a single timer, we can use it for the minimum timer in the pool; when the interval expires, the timer is reset to the new minimum. Another alternative is to decrement $Min(P)_{Value}$ by hardware (first instruction of `Update`), interrupting the main processor only if a timer has expired. These solutions have less overhead than interrupting the processor every single tick, particularly with a high-precision clock or long timeout intervals.

# 4 Correctness

In this section we prove formally that the basic algorithms for timer management, `Insert` and `Update`, are correct. That is to say, each execution of `Update` carries all the timers an additional step towards expiration, and they expire when their value reaches zero. Moreover, the execution of both algorithms preserves the validity of condition (1).

For proving the correctness of `Update`, we assume without loss of generality that `Update` is invoked on a nonempty $P$, and that executions of `Insert` and `Update` are mutually excluding in time.

**Theorem 1 [Correctness of `Update`]:** Let $i$ be the (integer) number of clock ticks elapsed since the first timer insertion. If for each $i > 0$, $Clock^i + 1 = Clock^{(i+1)}$, then every execution of `Update` preserves (1).

**Proof:** Let $i$ be the number of previous invocations of `Update` since the first insertion. We use superscripts to denote the values of the variables after the $j$-th update, as in $P^j_{Com}$. The two feasible branches that the $(i+1)$-th execution of the algorithm can take are considered separately.

First we discuss the case $Min(P)_{Value} > 1$. In this case, no timer expires at the $(i+1)$-th clock tick; the only operation performed is a decrement on $Min(P)_{Value}$. Assume that (1) holds before the execution of the algorithm, i.e. $Min(P)_{Exp} - Clock^i = Min(P)^i_{Value}$. From the algorithm we have,

$$Min(P)^i_{Value} = 1 + Min(P)^{(i+1)}_{Value} \tag{2}$$

the theorem assumptions and equation (2) imply that $Min(P)_{Exp} - Clock^{(i+1)} = Min(P)^{(i+1)}_{Value}$. We thus have that the first part of (1) holds after the execution of `Update`.

Let $x$, $x \notin Min(P)$ be a timer in $P$. Before the execution

$$x_{Exp} - Clock^i = x^i_{Value} + Min(P)^i_{Value} - P^i_{Com}$$

Given equation (2) and the theorem assumptions, since $x_{Value}$ and $P_{Com}$ are not modified we conclude that

$$x_{Exp} - Clock^{(i+1)} = x^{(i+1)}_{Value} + Min(P)^{(i+1)}_{Value} - P^{(i+1)}_{Com}$$

so (1) holds when the execution is finished.

The second case to be discussed is $Min(P)_{Value} = 1$. In this case, the timers in $Min(P)$ expire and must be removed from $P$. Let $y \in Min(Erase\_Min(P))$, the new minimum after the deletion of the previous one (if $P$ becomes empty after this extraction, then (1) is trivially valid).

$$y_{Exp} - Clock^i = y^i_{Value} + Min(P)^i_{Value} - P^i_{Com}$$

is true because (1) is true before `Update` is executed. Given $Min(P)^i_{Value} = 1$ and $Min(P)^{(i+1)}_{Value} = y^{(i+1)}_{Value} = y^i_{Value} - P^i_{Com}$ (from the algorithm), the theorem assumptions imply that

$$\begin{aligned} y_{Exp} - Clock^{(i+1)} &= (y^i_{Value} - P^i_{Com} + 1) - 1 \\ &= y^i_{Value} - P^i_{Com} \\ &= Min(P)^{(i+1)}_{Value} \end{aligned}$$

so the first part of (1) is true after the update.

Let $y$ be an element of the new minimum timer set as before, and let $x \neq y$ be an element of $Erase\_Min(P)$ not in $Min(Erase\_Min(P))$. Since (1) is true before the execution,

$$x_{Exp} - Clock^i = x^i_{Value} + 1 - P^i_{Com}$$

We also know that $P^{(i+1)}_{Com} = y^i_{Value}$, and $Min(P)^{(i+1)}_{Value} = y^i_{Value} - P^i_{Com}$ — from Figure 1. Given the theorem assumptions, we have that

$$\begin{aligned} x_{Exp} - Clock^{(i+1)} &= x^i_{Value} + 1 - P^i_{Com} - 1 \\ &= x^i_{Value} - P^{(i+1)}_{Com} + y^i_{Value} - P^i_{Com} \\ &= x^i_{Value} + (y^i_{Value} - P^i_{Com}) - P^{(i+1)}_{Com} \\ &= x^{(i+1)}_{Value} + Min(P)^{(i+1)}_{Value} - P^{(i+1)}_{Com} \end{aligned}$$

so (1) also holds after the execution for the second branch and the proof is finished. ■

**Theorem 2 [Correctness of `Insert`]:** If timers are always positive when inserted, the execution of `Insert` preserves the invariant condition (1).

**Proof:** If the pool $P$ is empty, the new timer will belong to $Min(P)$, and only the first part of (1) needs to be established. Let $t$ be the new timer inserted, then from the algorithm we have $t_{Value} = Time - MinVal + P_{Com}$. Whenever the pool is empty, $MinVal = 0$ and $P_{Com} = 0$. Moreover, $t_{Exp} = Time + Clock$, so

$$\begin{aligned} t_{Exp} - Clock &= Time \\ &= t_{Value} + MinVal - P_{Com} \\ &= t_{Value} \end{aligned}$$

thus (1) remains true after the insert operation.

For a non-empty pool, the proof is organized by cases according to the values of $Time$ and $Min(P)_{Value}$ as they are used in the algorithm.

First we discuss the case $Time > Min(P)_{Value}$. If there is another timer that will expire at the same time than the new one, no update is done to any timer nor to $P_{Com}$. The $Value$ and $Exp$ components of each timer remain equals, only a new timer_id is added to the pool. Since the condition (1) is true before the insert operation, it remains trivially valid after the insertion.

Otherwise, no previously inserted timer is modified, so it is only necessary to establish (1) for the new timer inserted into the pool. Since the new timer does not belong to $Min(P)$, the first part of (1) is true.

Let $t$ be the new timer inserted, then from the algorithm we have $t_{Value} = Time - Min(P)_{Value} + P_{Com}$. Moreover, $t_{Exp} = Time + Clock$,

$$\begin{aligned} t_{Exp} - Clock &= Time + Clock - Clock \\ &= t_{Value} + Min(P)_{Value} - P_{Com} \end{aligned}$$

and (1) holds after the insert operation.

The second case is $Time = Min(P)_{Value}$. In this case, since every timer and $P_{Com}$ remain unchanged (only a new timer_id is added to the pool), (1) is preserved by the execution of `Insert`.

In the third case, $Time < Min(P)_{Value}$, the $Min(P)$ timer set is redefined to contain only the new timer, and the timers previously in $Min(P)$ become part of the rest of the pool. The first part of (1) must be proven for the new timer, while the second part must be established for all the timers in the pool before the insertion.

Let $t$ be the new timer inserted, then from the algorithm we have $t_{Value} = Time$. We also know that $t_{Exp} = Time + Clock$, so

$$\begin{aligned} t_{Exp} - Clock &= Time + Clock - Clock \\ &= t_{Value} \end{aligned}$$

and the first part of (1) is true for $t$.

We use apostrophes to denote the values of variables after the timer insertion. Since $t$ belongs to $Min(P')$,

$$Min(P')_{Value} = t_{Value} \tag{3}$$

Let $x$ be a timer in $Min(P)$. Since the invariant condition is true before `Insert`, $x_{Exp} - Clock = x_{Value}$. Moreover, from the algorithm, $t_{Value} = Time$, $x'_{Value} = P_{Com}$, and $P'_{Com} = Time - x_{Value} + P_{Com}$. Using also equation (3) we have

$$\begin{aligned} x_{Exp} - Clock &= x_{Value} \\ &= Time + P_{Com} - P'_{Com} \\ &= t_{Value} + x'_{Value} - P'_{Com} \\ &= x'_{Value} + Min(P')_{Value} - P'_{Com} \end{aligned}$$

Let $y$ be an element of $Erase\_Min(P)$ before `Insert`. Since (1) holds before the insertion, $y_{Exp} - Clock = y_{Value} + Min(P)_{Value} - P_{Com}$. Given equation (3), and knowing from the algorithm that $t_{Value} = Time$, $P'_{Com} = Time - Min(P)_{Value} + P_{Com}$, and that $y_{Value}$ is not modified, we conclude that

$$\begin{aligned} y_{Exp} - Clock &= y_{Value} + Min(P)_{Value} - P_{Com} \\ &= y_{Value} + Time - P'_{Com} \\ &= y'_{Value} + Min(P')_{Value} - P'_{Com} \end{aligned}$$

Therefore (1) holds after the execution of `Insert` for the three possible cases. ■

# 5 Related Work

An alternative solution (*delta-list*) is proposed in [2] as part of the actual design of the UNIX System V operating system, and also in [7] for network protocol implementation. A sorted linked list of elements is kept, in which each element contains the offset to its predecessor. We discuss worst-case running times below. When inserting a timer the correct place of insertion must be found sequentially. The insertion is thus $\Theta(n)$ for the delta-list. For the case of an update, our algorithm takes $\Theta(1)$ if no timer expires, and $\Theta(\log n)$ otherwise. The delta-list algorithms can update the pool uniformly in $\Theta(1)$. However, consider the running time for the insert-update-expire sequence for any given timer. Let $k$ be the timeout interval of the timer. The cost of such a sequence is $\Theta(n) + k \times \Theta(1) + \Theta(1) = \Theta(n)$ using the delta-list algorithm, as opposed to $\Theta(\log n) + k \times \Theta(1) + \Theta(\log n) = \Theta(\log n)$ using our algorithms. An analogous reasoning holds if the timer is deleted before expiring. Therefore, the delta-list suite of algorithms is asymptotically slower, independently of the usage pattern.

Several alternative approaches are examined in [8], including a family of tree-based algorithms with logarithmic running times. The approach simply stores the timestamps in a priority queue; thus the amount of memory used per timer grows as the system ages, and clients pay this penalty regardless of the intervals they insert. Other solutions presented in this paper use hashed and hierarchical variants of a circular buffer management scheme. The hashed versions have cost $\Theta(n)$ for insertions or updates. The hierarchical version has also cost $\Theta(n)$ for the update operation (when all timers are migrated from one level to the next), and it can only manage time intervals up to a fixed limit — longer timeout periods are not allowed.

# 6 Conclusions

The implementation of communication software often calls for a way of managing multiple outstanding timeout periods. Since efficiency is usual the main motivation for choosing this kind of algorithms, a fast timer management service is required. Similar requirements arise in the operating systems field.

The algorithms described in this work run in logarithmic worst-case time independently of the operations requested by the clients. They deal with the general timer management case in an efficient way, allowing the use of hardware support to reduce the interrupt load on the main processor. The other approaches surveyed in the literature are either asymptotically less efficient, or only suited for a limited set of usage patterns.

### Acknowledgements

We thank the anonymous reviewers for useful comments. Thanks are also due to Flaviu Cristian and George Polyzos, for proofreading the paper and providing suggestions for improving its quality.

# References

[1] G. Alvarez, M. Fernandez, R. Alvez, S. Rodriguez, J. Sanchez, and J. Sanz. Run-time support for asynchronous parallel computations. In *Proc. of the Ninth International Parallel Processing Symposium*, Santa Barbara, California, April 1995.

[2] M. Bach. *The Design of the UNIX Operating System.* Prentice-Hall International, 1986.

[3] D. Comer. *Internetworking with TCP/IP – 2nd. Edition.* Prentice-Hall International, 1993.

[4] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[5] D. Ritchie and K. Thompson. The UNIX time-sharing system. In *UNIX System Readings and Applications*, volume 1, pages 1–25. Prentice-Hall International, 1987.

[6] W. Stevens. *UNIX Network Programming.* Prentice-Hall International, 1990.

[7] A. Tanenbaum. *Computer Networks.* Prentice-Hall International, 1989.

[8] G. Varghese and T. Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *Proc. of the 11th Symp. on Operating System Principles*, pages 23–58, 1987.