



# ERLANG TIMER WHEELS

Jade Allen @avocadoSuperFan

# Closing Walls and Ticking Clocks

- Let's talk about how weird time is
- Fact or crap: time always advances.
- Fact or crap: NTP is a thing, time is a solved problem.
- What even is a leap second?

# Love in the time of cholera

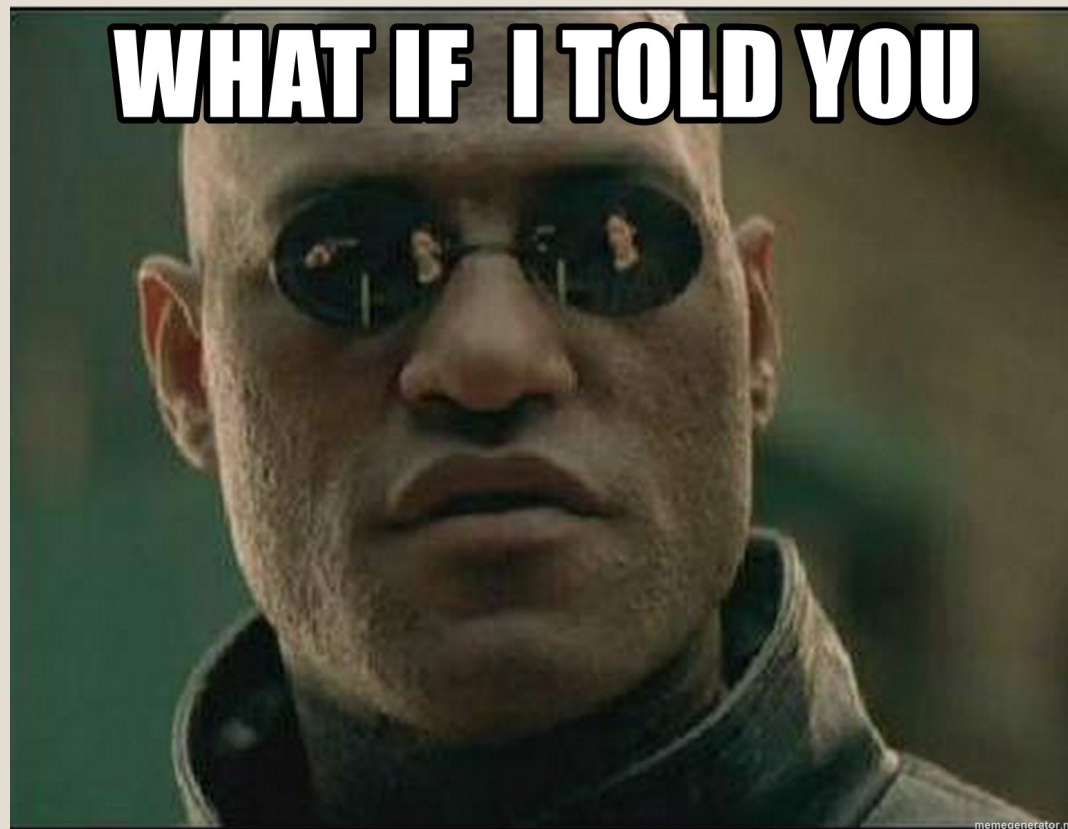
- Distributed systems need reliable timers for all kinds of reasons
  - Coordination of state replication (linearization)
  - Failure recovery/retries
  - Failure detection
  - Traffic control algorithms

# How can we implement timers?

- Method 1: Direct access
- Method 2: A list (possibly ordered by expiry)
- Method 3: Ignore wall clock time and “tick” on an event.

What's wrong with these implementations?

But wait, we can do better!



# Wheels in the sky emulator

## Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility

George Varghese and Tony Lauck  
Digital Equipment Corporation  
Littleton, MA 01460

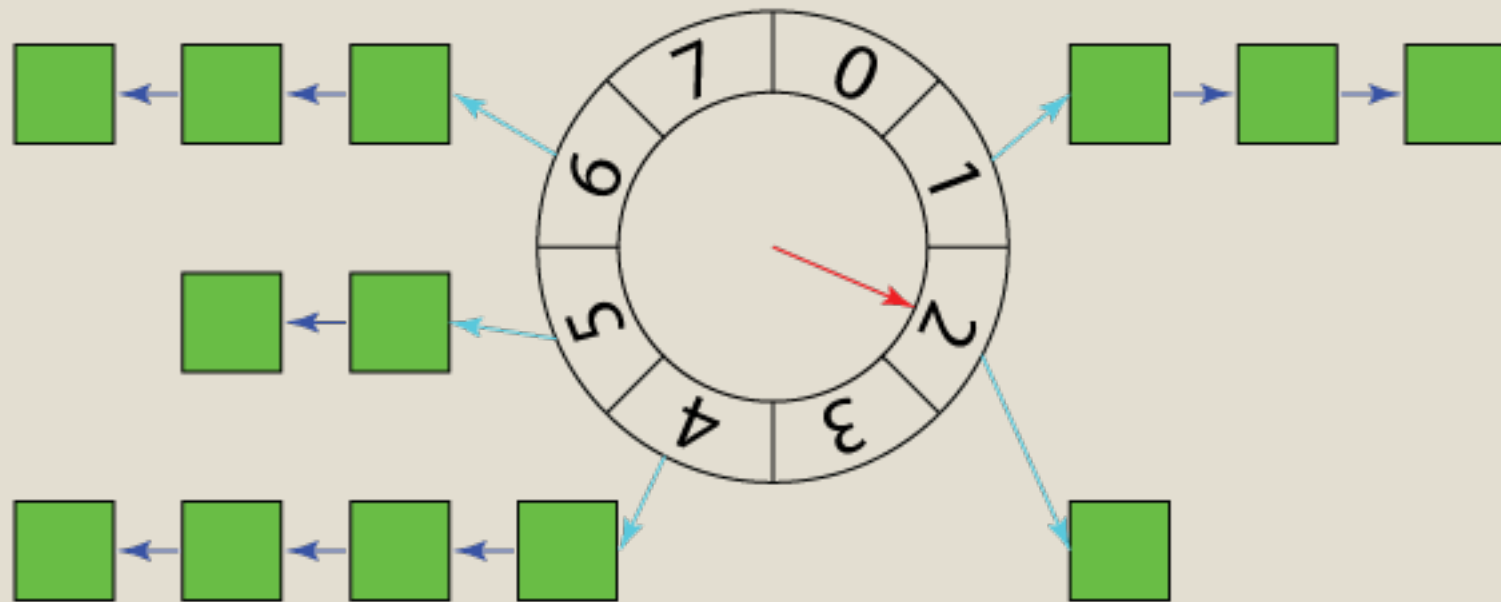
### Abstract

Conventional algorithms to implement an Operating System timer module take  $O(n)$  time to start or maintain a timer, where  $n$  is the number of outstanding timers: this is expensive for large  $n$ . This paper begins by exploring the relationship between timer algorithms, time flow mechanisms used in discrete event simulations, and sorting techniques. Next a timer algorithm for small timer intervals is presented that is similar to the timing wheel technique used in logic simulators. By using a circular buffer or timing wheel, it takes  $O(1)$  time to start, stop, and maintain timers within the range of the wheel.

be detected by periodic checking (e.g. memory corruption) and such timers always expire. Other failures can be only be inferred by the lack of some positive action (e.g. message acknowledgment) within a specified period. If failures are infrequent these timers rarely expire.

- Algorithms in which the notion of time or relative time is integral: Examples include algorithms that control the rate of production of some entity (process control, rate-based flow control in communications), scheduling algorithms, and algorithms to control packet lifetimes in computer networks. These timers al-

# A simple timer wheel



# How does Erlang deal with time?

- Before OTP 18
  - `now()`
  - The weird and terrible Erlang time tuple {MegaSeconds, Seconds, Milliseconds}
  - Terrible at dealing with OS level time changes
- OTP 18 and later (current release is OTP 26 – just released in May 2023)
  - `monotonic_time()` (managed by the Erlang run-time because reasons)
  - `system_time()`
  - `os:timestamp()` (also system time without interpretation)
  - Configurable ways to deal with underlying system time changes



```
apply_after(Time, Module, Function, Arguments) ->  
            {ok, TRef} | {error, Reason}
```

### Types

```
Time = time()  
Module = module()  
Function = atom()  
Arguments = [term()]  
TRef = tref()  
Reason = term()
```

Evaluates `spawn(Module, Function, Arguments)` after `Time` milliseconds.

Returns `{ok, TRef}` or `{error, Reason}`.

```
send_after(Time, Message) -> {ok, TRef} | {error, Reason}
```

```
send_after(Time, Destination, Message) ->  
    {ok, TRef} | {error, Reason}
```

## Types

```
Time = time()  
Destination =  
    pid() |  
    (RegName :: atom()) |  
    {RegName :: atom(), Node :: node()}  
Message = term()  
TRef = tref()  
Reason = term()
```

### send\_after/3

Evaluates `Destination ! Message` after `Time` milliseconds. (`Destination` can be a remote or local process identifier, an atom of a registered name or a tuple `{RegName, Node}` for a registered name at another node.)

Returns `{ok, TRef}` or `{error, Reason}`.

See also [the Timer Module section in the Efficiency Guide](#).

### send\_after/2

Same as `send_after(Time, self(), Message)`.

Creating timers using `erlang:send_after/3` and `erlang:start_timer/3` is more efficient than using the timers provided by this module. However, the timer module has been improved in OTP 25, making it more efficient and less susceptible to being overloaded. See [the Timer Module section in the Efficiency Guide](#).



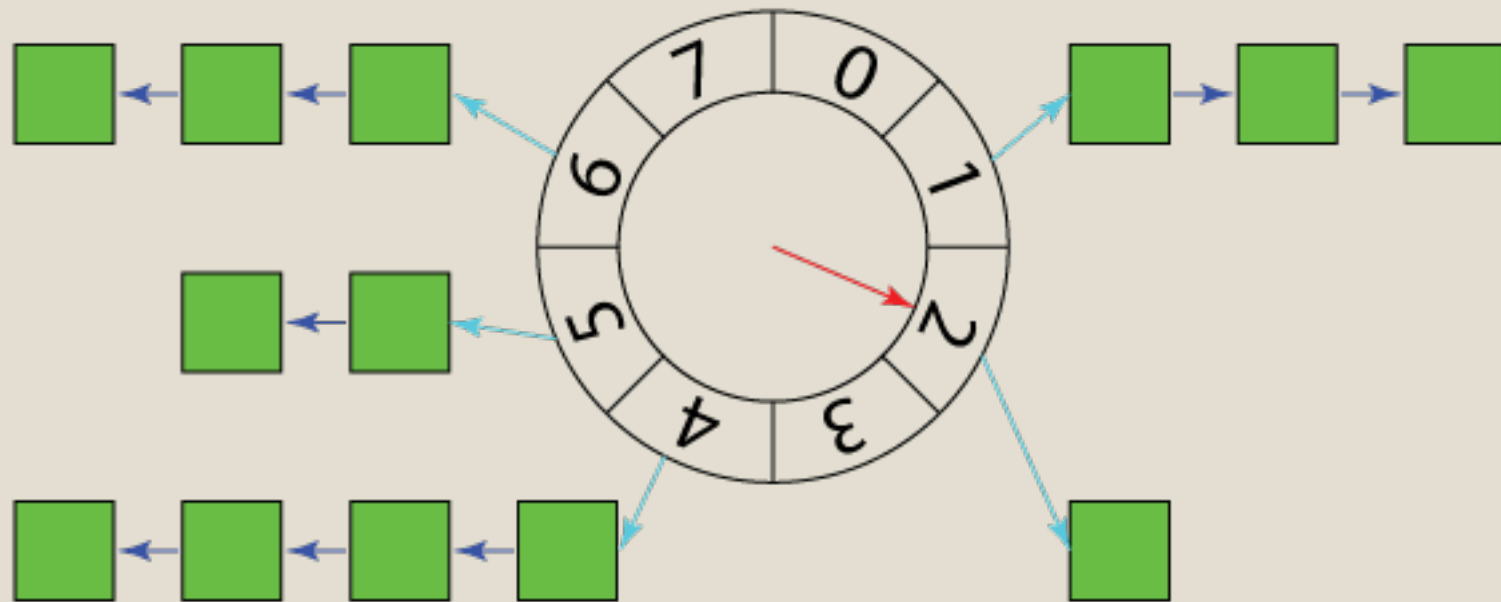
### 3.1 Timer Module

Creating timers using `erlang:send_after/3` and `erlang:start_timer/3`, is more efficient than using the timers provided by the `timer` module in STDLIB.

The `timer` module uses a separate process to manage the timers. Before OTP 25, this management overhead was substantial and increasing with the number of timers, especially when they were short-lived, so the timer server process could easily become overloaded and unresponsive. In OTP 25, the timer module was improved by removing most of the management overhead and the resulting performance penalty. Still, the timer server remains a single process, and it may at some point become a bottleneck of an application.

The functions in the `timer` module that do not manage timers (such as `timer:tc/3` or `timer:sleep/1`), do not call the timer-server process and are therefore harmless.

# A simple timer wheel, again



```
#ifdef ERTS_TW_DEBUG
/*
 * Soon wheel will handle about 1 seconds
 * Later wheel will handle about 8 minutes
 */
# define ERTS_TW_SOON_WHEEL_BITS 10
# define ERTS_TW_LATER_WHEEL_BITS 10
#else
# ifdef SMALL_MEMORY
/*
 * Soon wheel will handle about 4 seconds
 * Later wheel will handle about 2 hours and 19 minutes
 */
#   define ERTS_TW_SOON_WHEEL_BITS 12
#   define ERTS_TW_LATER_WHEEL_BITS 12
# else
/*
 * Soon wheel will handle about 16 seconds
 * Later wheel will handle about 37 hours and 16 minutes
 */
#   define ERTS_TW_SOON_WHEEL_BITS 14
#   define ERTS_TW_LATER_WHEEL_BITS 14
# endif
#endif
#endif
```

```

struct ErtsTimerWheel_ {
    ErtsTWheelTimer *slots[1
                        + ERTS_TW_SOON_WHEEL_SIZE /* Soon Wheel Slots */
                        + ERTS_TW_LATER_WHEEL_SIZE]; /* Later Wheel Slots */

    ErtsTWheelTimer **w;
    Sint scnt[ERTS_TW_SCNT_SIZE];
    Sint bump_scnt[ERTS_TW_SCNT_SIZE];
    ErtsMonotonicTime pos;
    Uint nto;
    struct {
        Uint nto;
    } at_once;
    struct {
        ErtsMonotonicTime min_tpos;
        Uint nto;
    } soon;
    struct {
        ErtsMonotonicTime min_tpos;
        int min_tpos_slot;
        ErtsMonotonicTime pos;
        Uint nto;
    } later;
    int yield_slot;
    int yield_slots_left;
    ErtsTWheelTimer sentinel;
    int true_next_timeout_time;
    ErtsMonotonicTime next_timeout_pos;
    ErtsMonotonicTime next_timeout_time;
};

```

# Say what now?

```
else {  
    /*  
     * Linux versions prior to 2.6.33 have a  
     * known bug that sometimes cause the NTP  
     * adjusted monotonic clock to take small  
     * steps backwards. Use raw monotonic clock  
     * if it is present; otherwise, fall back  
     * on locked verification of values.  
     */  
    init_resp->have_corrected_os_monotonic_time = 0;
```



# Even monotonic time?

```
/*  
 * Maximum drift of the OS monotonic clock expected.  
 *  
 * We use 1 milli second per second. If the monotonic  
 * clock drifts more than this we will fail to adjust for  
 * drift, and error correction will kick in instead.  
 * If it is larger than this, one could argue that the  
 * primitive is too poor to be used...  
 */  
#define ERTS_MAX_MONOTONIC_DRIFT ERTS_MSEC_TO_MONOTONIC(1)
```



# THANKS!

Questions?