Timeouts: Understanding and applying absolute and relative methods

Table of Contents

Working with timers in applications – especially in Lua environments such as Emilua – understanding the difference between **absolute timeout** and **relative timeout** is essential for controlling task execution with precision. This distinction is reminiscent of the differentiation found in linguistics, where "Absolute tense situates events in a fixed timeline, while relative tense expresses temporal relations relative to the moment of speaking". This analogy helps us understand that, just like in language, timeout methods have distinct approaches to managing time.

1. Basic concepts

Absolute timeout

In an absolute timeout, you define a **fixed point** in time when an operation should expire. This method is ideal when a task must occur at a specific time, regardless of any delays in execution. Much like absolute tense in linguistics, the event is placed on an immutable timeline.

Relative timeout

In a relative timeout, the expiration is defined as a **time interval** from the current moment – that is, time is measured relative to the moment of execution. This approach is practical for recurring

tasks, although minor delays may accumulate if previous executions take longer than expected. This concept is similar to relative tense in language, which describes events based on the current moment.

2. The problem of jitter in relative timeouts

When using a relative timeout, small delays in each iteration (for example, due to I/O operations or additional processing time) can accumulate over multiple cycles, leading to **jitter**. This means the actual elapsed time drifts away from the expected schedule.

Below is an example of a relative timeout in Emilua. Notice that even though the code asks for 1 second per iteration, slight variations may lead to a progressive difference between expected and actual timing.

3. Practical example: Relative timeout

```
local time = require 'time'

local timer = time.steady_timer.new()
local start_time = time.steady_clock.now()
local i = 0

while true do
  i = i + 1

  timer:expires_after(1)
  timer:wait()

  local current_time = time.steady_clock.now()
  local elapsed = current_time - start_time

  local formatted_message = format("Iteration {0} completed after {1:.3f} seconds", i,
elapsed)
  print(formatted_message)
end
```

Relative timeout: Output example

Due to the nature of relative timeouts, minor delays may accumulate over iterations. Thus, the output might be something like:

```
Iteration 1 completed after: 1.000 seconds
Iteration 2 completed after: 2.012 seconds
Iteration 3 completed after: 3.018 seconds
Iteration 4 completed after: 4.025 seconds
```

Even though each cycle attempts to wait 1 second, occasional delays (e.g., in the print execution or the wait itself) can cause slight deviations that add up over time.

4. Practical example: Absolute timeout

To address jitter, an absolute timeout defines a **fixed point** in time at which each iteration should complete. This way, the timer remains synchronized with an initial reference, rather than shifting each subsequent iteration based on the last.

```
local time = require 'time'

local start_time = time.steady_clock.now()
local timer = time.steady_timer.new()

local i = 0
while true do
  i = i + 1

  timer:expires_at(start_time + i)
  timer:wait()

  local current_time = time.steady_clock.now()
  local elapsed = current_time - start_time

  local formatted_message = format("Iteration {0} completed after {1:.3f} seconds", i,
elapsed)
  print(formatted_message)
end
```

Absolute timeout: Output example

Assuming there are no unexpected delays, the output might be:

```
Iteration 1 completed after 1.000 seconds
Iteration 2 completed after 2.000 seconds
Iteration 3 completed after 3.000 seconds
Iteration 4 completed after 4.000 seconds
```

In this approach, the timer remains synchronized with the fixed point, ensuring each iteration occurs exactly at the expected time.

5. Comparison between absolute and relative timeouts

Aspect	Absolute timeout	Relative timeout
Definition	Fixed point in time (e.g., start_time + i seconds)	Time interval (e.g., wait 1 second)
Precision	Maintains synchronization with an exact schedule	May accumulate delays if each iteration takes too long
Ideal use	Precise scheduling (e.g., daily events)	Recurring tasks without strict alignment
Output example	Iteration 1: 1.000 s; Iteration 2: 2.000 s; ...	Iteration 1: 1.000 s; Iteration 2: 2.012 s; ...

Final considerations

The choice between **absolute timeout** or **relative timeout** depends on the context and specific timing requirements. Below is an enhanced summary of the characteristics of each approach:

Absolute timeout

- **Advantages:**
- Ensures operations occur at fixed points on the timeline;
- Ideal for precise scheduling (e.g., daily tasks at specific times).
- **Ideal Output:**
- Each iteration shows an elapsed time exactly as expected (1s, 2s, 3s, etc.).

Relative Timeout

- **Advantages:**
- Simple to implement for recurring tasks;
- Each cycle resets the count from the current moment.
- **Disadvantages:**
- Minor delays can accumulate, causing divergence between expected and actual elapsed time.
- **Example Output:**
- Iteration 1: 1.000 s; Iteration 2: 2.012 s; Iteration 3: 3.018 s; Iteration 4: 4.025 s.

Just as linguistic concepts of time allow us to situate events absolutely or relatively,

> timeout methods provide distinct strategies for managing time. The choice of method should be determined by the context in which they are applied.