Library for Simplified Timer Implementation using Standard C++

Sérgio F. Lopes, Paulo Vicente and Ricardo Gomes
Centro Algoritmi, School of Engineering, University of Minho, Guimarães, Portugal

Keywords: Temporization, Timers, C++11, Computer-based Control, Event-driven programming.

Abstract: Temporization is a crucial aspects of control, automation and robotics systems. C++ is used in the development of such systems, especially if they are more complex and powerful. Because, the language and standard library do not support non-blocking timers with callbacks for event-driven programming, developers resort to libraries and frameworks that offer such functionality. However, their timer implementations are dependent on platform specificities and thus have more limited portability. C++11 has introduced features that enable standard implementations of timers. We propose a library that implements timers with simplified usage relatively to well-known libraries. The proposed library is contrasted with timers of two well know libraries, through a series of usage scenarios. We describe the design and provide performance measurements. The results show that it is faster and offers more accurate temporization.

1 INTRODUCTION

Time is a crucial aspect of control, automation and robotics systems. The complexity of such systems has increased, and many of them are increasingly based on more powerful computing platforms and software development environments that support C++ programming language. In fact, C++ has been used in embedded systems and other resource-constrained types of programming for a long time (Stroustrup, 2005), because it allows handling software complexity while retaining predictability and performance.

Time-related tasks can be divided in two kinds. One is to measure the time that it takes an activity to complete (or a phenomenon to occur). The result is a time interval that is calculated at the end, and we refer to it as (time) *counting*. The other is to wait a known amount of time before performing an action (or to wait for an event to happen). We hereafter designate it as *temporization*, and its result is the execution of the pre-configured action after (or the potential event reception during) the specified time interval.

Counting is widely supported by standard libraries' functions that read some form of clock, while temporization can be implemented in diverse ways. Simple approaches include: (1) stop the program/thread using some blocking sleep-like function, not being able to do anything else; and (2) to constantly pool and measure time wasting CPU time and energy. Naturally, these are unacceptable hypotheses. Suitable approaches include the use of system specific asynchronous IO, multithreading and synchronization primitives. These are not at the preferred level of abstraction for developing complex applications, and they do not allow the productivity levels necessary for large programs.

For the abovementioned reasons several C++ libraries, include a feature, usually called a *timer*, which supports temporization providing a simpler API. However, we have tried some well-known libraries and still wished for, and could think of, an easier to use interface and different functionalities. Moreover, those libraries depend on platform specific code and their usability is limited to the targeted platforms.

C++11 (ISO/IEC 2011) introduced many relevant features to C++, including the thread support and time utilities libraries. Yet, timers were not included, not even in the latest version of the language, C++14 (ISO/IEC 2014). From the time when most widely used compilers started to offer extensive support for C++11's standard library features, it became possible to start developing portable solutions for temporization in C++.

The main contribution of this paper is a new timer library, implemented exclusively in C++. Therefore, it can be used in all platforms for which there are C++ development environments, more

F. Lopes S., Vicente P. and Gomes R..
Library for Simplified Timer Implementation using Standard C++.
DOI: 10.5220/0005575205170524
In *Proceedings of the 12th International Conference on Informatics in Control, Automation and Robotics* (ICINCO-2015), pages 517-524
ISBN: 978-989-758-123-6

specifically those supporting C++11's thread and chrono libraries (Josuttis, 2012). We argue that this library's timers offer an easier and more advanced API comparatively to two other important timer libraries. Besides functional advantages, the proposed library also surpasses the others in terms of temporization performance.

In the following section we review related work. In section III, two widely used libraries are analysed from the perspective of a varied set of usage scenarios. At the same time, we present how those scenarios can be supported by an easier to use API. Section IV describes the proposed library and performance measurements. Finally, we present conclusions and future work.

2 RELATED WORK

We are interested in a timer that waits a given interval of time without blocking the program, and if it is not stopped before that interval expires (i.e., timeout event occurs) it calls a pre-configured function. We refer to such a function as *callback*, and it is executed asynchronously to the rest of the program. Therefore, it is an event-handler as found in event-driven programming environments.

In our search we were able to find several C++ libraries that include timers. As mentioned in the previous section, all of them depende on platform specific code, i.e., they are not based on C++11 standard library.

In the low-level library (Mitchell, 2013), the developer is responsible for implementing the events loop. The specific event-driven design of (The Qt Company 2015) applies to timers, whose events are dispatched as all others, and timers can only be controlled from the thread that creates them. In (Robinson, 2013), timer callbacks are executed by the thread that dispatches events, thus blocking other (timer) events. These event processing limitations are not found in (Henning, 2004; King, 2009). In (Henning 2004), timers are implemented by deriving a base class and a thread is launched for each timer object. In (King 2009) there is a thread dedicated to process all timers, which launches one thread to execute each callback.

Two other libraries are reviewed in more detail since they are used in the following section. Boost is a large and important collection of C++ libraries that has had a major influence on C++ standard evolution. It includes Asio (Kohlhoff, 2014), a library that supports timers. When an Asio timer is created an io_service is associated to it. One or more callbacks are attached to a timer by using its async_wait function. The time interval starts at timer creation, and the io_service::run function blocks if invoked before the interval end, otherwise it executes the callbacks. Consequently, Asio timers requires the addition of multithreading to obtain asynchronous temporization as defined in the previous section. It is a lower-level library.

Poco (Applied Informatics Software Engineering 2010) is also a collection of libraries that offers two kinds of timers. One is the Poco::Util::Timer, which similarly to a Java timer works as a timed task scheduler, executing callbacks in sequence. The other one is Poco::Timer, from the main library, that is closer to the one proposed in this paper, whose model is described in the previous section. Although the former can be used according to that model, we hereafter consider the latter. The timer is created by specifying a start interval and a periodic interval. Callback objects are created from user classes using the TimerCallback template, which allows using any user function that has the first parameter of type Poco::Timer. The callback object is then passed to the timer using its start function, which begins the temporization. If the timer is not stopped before expiry, the callback is executed by an internal pool of threads.

3 TIMER USE CASES

This section discusses the implementation of solutions to temporization scenarios using Asio and Poco. At the same time, our view of an easier to use API is anticipated.

3.1 Single Temporization

In this scenario the timer is started once to execute a given callback when it expires.

In consequence of the general description of Asio timers previously given, it is necessary to create a thread to avoid blocking the main thread. Asio callbacks must be void and accept as first parameter an error code. The code in figure 1 considers a callback that also receives an integer number, for example purposes. Consequently, async_wait needs a binding of the extra arguments. After expiration and callback execution, the io_sevice::run function returns and ends the thread.

The Poco implementation is given in figure 2, in which timer arguments specify, respectively, the start and periodic intervals. The callback function must be void and have only one parameter of the

type Timer. If the developer needs more data in the callback, that data can be included as (a) member-variable(s) of UserClass, to be accessible from the member-function used as callback. In fact, the callback cannot be a global function.

While Asio uses C++11 standard time types that allow any units, Poco uses long integers in units of milliseconds. Another fundamental aspect is that Poco uses an object-oriented approach, requiring callbacks to be member-functions, while Asio allows also global functions. Clearly, Asio offers a lower-level of abstraction.

The periodic interval parameter of Poco Timer adds unnecessary complexity to non-periodic timers, and it has a special value (zero, meaning non-periodic) which must be remembered. Both interval parameters could be of double (instead of a long) type, and allow for higher resolution timers.

The instantiation of UserClass and TimerCallback to build a callback object can be simplified if the callback function's name is predefined. In our view, a callback object should clearly assume its role (Reenskaug 1996), which could simply be an interface with a member function that callback objects must implement. This way, only one object would need to be created. (Poco also allows this, but it requires the implementation of two functions.)

Finally, dividing the temporization setup in two functions (constructor and start) is more error prone if the same timer is used for different temporizations. We prefer to unite the timer setup, namely configuring the callback along with the interval. An interface such as the one illustrated in figure 3 would be slightly simpler.

```
// callback function is:
// void func(const system::error_code& ec,
//       int* anArg);

io_service io;
high_resolution_timer timer1( io,
        std::chrono::milliseconds(500));
int anArg = 10;
auto bound_cb = bind( func,
        placeholders::error,
        &anArg);
timer1.async_wait(bound_cb);
auto bound_th = bind(
        &io_service::run,
        &io);
thread thread1(bound_th);
```

Figure 1: Start an Asio timer and wait for expiration.

```
Timer timer1(100, 0);
UserClass obj;
TimerCallback<UserClass> cb(obj,
        & UserClass::func);
timer1.start(cb);
```

Figure 2: Start a Poco timer and wait for expiration.

```
UserClass cbo;
Timer timer1(0.1, cbo);
timer1.start();
```

Figure 3: Preferred timer start and wait for expiry.

3.2 Periodic Temporization

In this scenario the timer executes a callback at a constant rate or frequency. This is relevant, for example, for cyber-physical systems' sensing and actuation tasks since digital control theory assumes fixed input/output rates.

Asio does not support the concept of a periodic timer. It can be implemented by repeatedly starting a timer after expiration. A good place to do this is in the callback itself, by passing to it as arguments the timer and period. The code is shown in figure 4, and the timer start up is achieved in the same way as in figure 1. To avoid accumulation of eventual delays between periods, the next expiration is calculated from the (previous) expiration time (using function expires_at without arguments), instead of from the present time. The timer is restarted using expires_at with the new expiration time as argument.

This use-case shows more clearly that Asio plays in a different abstraction level, involving considerably more work (i.e., calculate next timeout, restart timer and insert callback). Still, we continue to analyze it because C++11 imported many functionalities from Boost and that may probably happen again in future versions.

Poco offers timers that expire periodically and

```
void funcPeriodic (
        const system::error_code& error,
        high_resolution_timer* timer,
        std::chrono::milliseconds period)
{
    if (!error) {
        auto nextEnd =
            timer->expires_at() + period;
        timer->expires_at(nextEnd);
        timer->async_wait( bind(
                funcPeriodic,
                placeholders::error,
                timer,
                period));
        // callback actions go here
    } else if (error !=
            error::operation_aborted) {
        // handle error
    }
}
```

Figure 4: Callback for a periodic timer using Asio.

the code to implement one differs from figure 2 solely in the first line. In the example of figure 5, we consider the case wherein the timer has a first interval different from the period. In such case, the previously discussed complexity of Poco having a parameter that separates the period from the first interval pays off. However, when the developer needs a periodic timer with all intervals equal, it has to provide the same amount twice. An alternative could be to specify that the single temporization interval is to be repeated until the timer is stopped. This could be achieved in the preferred API of figure 3, by adding a Boolean argument to the second line.

3.3 Stop Temporization and Restart

This use-case reflects the need to detect the timeout of an event that should occur several times or repeatedly (e.g., a security keypad stroke). The timer is stopped because the expected event has occurred, and later on, after the handling work is done, the timer is started to repeat the temporization.

3.3.1 Non-Periodic Timer

To stop an Asio timer, its cancel function must be called, which triggers the execution of all pending callbacks passing them an error value that indicates the timer was stopped. It is up to the programmer to correctly handle this situation, namely to distinguish it from a common expiration. Since the callback(s) are executed in another thread, and depending on the amount of computation done between stop and restart, the io_service may or may have not stopped (i.e., completed the execution of callbacks(s)). If it does, it must be reset and executed in a new thread, as exemplified in figure 6.

The implementation with Poco is shown in figure 7. The timer restart requires providing once more to start function the correct/same callback object. This is the only hindrance to be straightforward. It would be preferable to memorize the callback object and avoid putting that responsibility on the programmer.

3.3.2 Conditional Stop and/or Restart

The previous section assumes that either the timer has not expired before the stop or the restart is unconditional. In another possible situation, the timer could only be (stopped and/or) restarted if it had not expired, because an expected event occurred in time.

To detect whether an Asio timer has expired or not, it is as simple as checking the positive value returned by the cancel function: if it is 0 the timer has already expired (and, naturally, cancel has no effect), otherwise it has not expired. So, the stop operation is automatically conditional, and to perform a conditional restart is straightforward.

Poco does not provide information about the timer state, and thus another mechanism is necessary to measure the time elapsed since start. This can be implemented using C++11 chrono library or another Poco utility, the Stopwatch class, which is illustrated in figure 8. With the exception of the conversion of Stopwatch units to milliseconds, all code should be self-explanatory.

Since a separate time measurement mechanism must be used with Poco, it is not synchronized with timer functions and the precision of timer state detection is degraded. More importantly, a race condition arises between the thread "running" the timer (expiration event) and the thread controlling it (stop invocation), preventing correct operation. In contrast, Asio cancel function offers both functionalities atomically, and, consequently, it corresponds to the preferred API.

3.3.3 Periodic Timer

Periodic timers can also be subject of this use-case, whenever their action needs to be halted for some time.

In Asio, implementation concerns are roughly the same as for the non-periodic timer (in section 3.3.1). The difference is that if the stop instruction (cancel function invocation) occurs in between the expiration and periodic restart (made in the callback, see figure 4), the timer will not stop. This code span

```
Timer timer2(50, 100);
// create callback and start as in fig. 2
```

Figure 5: Periodic timer using Poco.

```
timer1.cancel();
// computations between stop and restart
timer1.expires_from_now(
        std::chrono::milliseconds(500));
timer1.async_wait(bound_cb);
if(io.stopped()) {
    io.reset();
    thread1.join();
    thread1 = thread(bound_th);
}
```

Figure 6: Stop and restart the Asio timer of figure 1.

```
timer1.stop();
// computations between stop and restart
timer1.start(cb);
```

Figure 7: Stop and restart Poco timer of figure 2.

forms a critical section that cannot be protected using synchronization primitives, because it includes both code internal to Asio (io_service loop dispatching callback execution) and user code (timer restart in the callback). To ensure the timer is stopped, it is necessary to stop the io_service and wait for its thread exit. Consequently and relatively to section 3.3.1, the stop procedure is more complex, but it puts the io_service and thread in a known state, making the restart procedure simpler, as shown in figure 9.

When a Poco timer is stopped, it loses the periodic interval setting and, thus, it is necessary to reset it, as illustrated in figure 10. The programmer has to perform that extra step and to use/maintain the same interval in two code places. Like in section 3.3.1 for the callback setting, it would be preferable to remember the periodic interval, simplifying the job and reducing the liabilities of the programmer. The resulting code would be the same as for the non-periodic timer in figure 7.

3.4 Restart (without Stopping)

A use-case similar to the previous is to restart the timer without stopping it, or conversely the stop is immediately followed by the start. In this case, we consider only the variant wherein a non-periodic timer is restarted if not expired.

```
Stopwatch counter;
counter.start();  // next to timer start
// ...
// next to timer stop
long elapsed_tm = counter.elapsed()/1000;
if(elapsed_tm <
        timer1.getStartInterval()) {
    // timer has not expired yet
```

Figure 8: Detect if the Poco timer of figure 2 has expired or not.

```
timer1.cancel();
if (!io.stopped()) {
    io.stop();
    thread1.join();
}
// computations between stop and restart
timer1.expires_from_now(
        std::chrono::milliseconds(500));
timer1.async_wait(bound_func);
io.reset();
thread1 = thread(bound_th);
```

Figure 9: Stop and restart an Asio periodic timer (i.e., with the callback of figure 4).

```
timer2.stop();
timer2.setPeriodicInterval(300);
timer2.start(cb);
```

Figure 10: Stop and restart the periodic Poco timer of figure 5.

Using Asio, the implementation is almost identical to the one given for the restart procedure in figure 6: the difference is that the value returned by expires_from_now invocation (which has the same meaning as that of cancel function) is used to make all other steps conditional.

As explained in section 3.3.2, a supplementary mechanism is necessary to find out if a Poco timer is still running. Moreover, the start function has no effect on timers in running state, and restart function is only applicable to periodic timers. Therefore, as shown in figure 11 (assuming the counter is started as illustrated in figure 8), it is necessary to stop the timer and then start it again.

In contrast with Poco, a preferable API would allow to simply call a restart function without having to provide settings (callback and interval) that the timer already has. To implement a conditional restart, timer state detection must be done atomically with it (as explained in section 3.3.2 for the stop operation). Therefore, a restartIfRunning function returning success/failure information could be provided, as illustrated in figure 12.

3.5 Suspend and Resume

In this use-case a timer is suspended and later resumed to wait for the remainder of the interval. This is pertinent whenever the waiting operation must not be restarted every time an expected event occurs, but the time spent processing those events must be subtracted from the total waiting time. Naturally, this scenario only applies to timers that are running. Moreover, it is more relevant for non-periodic timers and, therefore, we analyze only that circumstance.

The Asio implementation is similar to the stop and restart use-case (in section 3.3.1, figure 6): the difference is the need to measure the remaining time "when" the timer is stopped, and pass it as a new interval to expires_from_now (see figure 13).

```
long elapsed = counter.elapsed()/1000;
if(elapsed < timer1.getStartInterval()) {
    timer1.stop();
    timer1.start(cb);
}
```

Figure 11: Restarting the Poco timer of figure 2, using the counter of figure 8.

```
timer1.restartIfRunning();
```

Figure 12: Preferable API for conditionally restarting the timer of figure 3.

The Poco implementation differs from stop and restart use-case (see figure 7) in the same way Asio does. The remaining time is configured by the setStartInterval function, as exemplified in figure 14. The separate counter necessary to measure it the elapsed/remaining time (explained in section 3.3.2), may (due to unavoidable imprecision) incorrectly restart a timer that has just expired.

Neither Asio nor Poco distinguish pause from stop, not offering direct support to suspend timer waiting. With both libraries it is necessary to resort to stop and start, incurring in the complications discussed in section 3.3.1, in addition to the ones discussed in this section. In a preferable API, suspending and resuming a timer should be as simple as it is shown in figure 15.

4 PROPOSED TIMER LIBRARY

4.1 Alignment with Previous Discussion

The proposed timer is based on the preferable API that is described in previous section. The timer library is defined in the namespace UM, and consists of Timer class and the TimerCallback interface, as depicted in UML class diagram of figure 16.

```
if(timer1.cancel() > 0) {
    auto remaining =
        timer1.expires_from_now();
    // compts. between suspend and resume
    timer1.expires_from_now(remaining);
    timer1.async_wait(bound_cb);
    if(io.stopped()) {
        io.reset();
        thread1.join();
        thread1 = thread(bound_th);
    }
}
```

Figure 13: Suspend and resume Asio timer of figure 1.

```
timer1.stop();
long remaining_tm =
        timer1.getStartInterval() -
        counter.elapsed()/1000;
if(remaining_tm > 0) {
    // compts between suspend and resume
    timer1.setStartInterval(remaining_tm);
    timer1.start(cb);
}
```

Figure 14: Suspend and resume the Poco timer of figure 2, using the counter of figure 8.

Any user class can be used to create callback objects, by implementing the TimerCallback interface and putting the timeout handling code in the execute function. Multiple callbacks supported by Asio can also be implemented within the execute function of the callback object, namely calling functions from other objects, either synchronously or asynchronously launching threads.

The Timer class provides methods to support all use-cases analyzed in the previous section. Concretely, it offers:

- constructors that allow to create timers using any C++ time units, including the helper duration types already offered in the standard library (i.e., microseconds, milliseconds, etc.);
- start and stop functions supporting single and periodic temporizations (respectively, in sections 3.1 and 3.2), and stop and restart use-cases (in section 3.3); and,
- pause function supporting the suspend action of the suspend and resume use-case (in section 3.5).

```
if (timer1.suspend()) {
    // compts between suspend and resume
    timer1.resume();
}
```

Figure 15: Preferable API to suspend and resume timer of figure 3.

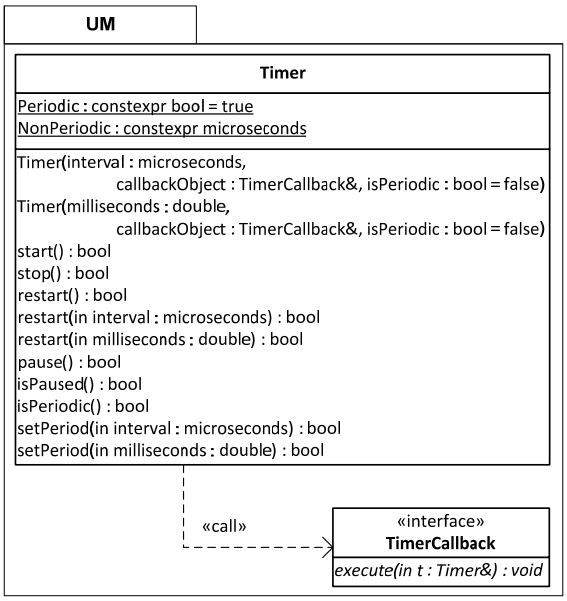

Figure 16: Class diagram of UM Timer library API.

4.2 Differences from Previous Discussion and Beyond

A constant expression (functionality introduced in C++11) was defined to allow an explicit creation of periodic timers, as exemplified in figure 17.

In section 3.4, a restartIfRunning function is suggested to atomically implementing the conditional restart. However, such a function is not a common approach in timer APIs, and the same effect can be achieved with existing functions. Concretely, it is implemented using the stop function, that (similarly to Asio cancel) returns true if the timer is running, and then doing a restart, as illustrated in figure 18.

Besides avoiding an additional function, the aforementioned solution is also related to another design choice, which is more fundamental. This choice is the cause for the absence of a resume function, to be used along with pause function. Figure 19 demonstrates how the suspend and resume use-case is supported using the start function to resume waiting for the remaining time. This is so because start distinguishes a paused timer from a stopped timer. More formally, start is a causal function, because it regards the timer state, whereas restart does not. In fact, the concept of restarting can be identically applied to a procedure whether it is stopped, paused or under way. From a causal perspective, starting a running timer makes no sense, and thus UM::Timer start has no effect. Therefore, the design decision to make start causal and restart non-causal is coherent and it is also aligned with the common notions of start and restart. Finally, both the causal and non-causal functions facilitate the management of timers that are shared among threads, by reducing the number of if-else statements that are necessary.

```
UserClass cbo;
Timer timer1(0.1, cbo, Timer::Periodic);
```

Figure 17: UM::Timer API to create a periodic timer.

```
if (timer1.stop()) {
    timer1.restart();
    // ...
```

Figure 18: UM::Timer API to restart a timer without a critical section.

```
if (timer1.pause()) {
    // compts between suspend and resume
    timer1.start();
}
```

Figure 19: Suspend and resume an UM::Timer.

UM::Timer also offers restart functions with an interval argument, which enable to either start a timer with an interval different from the preceding run or to change the interval of an ongoing temporization (which is not possible in Poco as explained in section 3.4). The former case is useful, for instance, to implement repetitive timers with variable periods, for example to wait a variable backoff period before retrying an authentication that has failed. Repetitive timers can be implemented easily by restarting them in the callback itself. It should be noted that such an approach is more complicated with both Asio (as discussed in section 3.2 for periodic timers) and Poco (which requires a periodic timer, whose callback restarts the timer with a different intervals).

Less importantly, but as a matter of flexibility, UM timers have two setPeriod functions that enable to convert them from non-periodic to periodic, and vice-versa. For example, the former conversion can be used to support periodic timers with a different first interval, as Poco does (see section 3.2). This is implemented with a non-periodic timer that is later converted to periodic. The latter conversion is supported using another constant expression – NonPeriodic – to avoid special values and make it explicit/clear.

5 PERFORMANCE RESULTS

We have done a series of comparative tests to measure the performance of the three discussed libraries, using Asio version 1.10.1 (Boost 1.55) and Poco version 1.3.6, with the GNU g++ 4.8.2 compiler, on a Pentium D915@2.8GHz machine running Linux kernel 3.16. The tests were repeated 100 times and the average values are presented in tables 1 - 3.

The first test consisted in starting a timer and stopping it after 50ms, measuring the time it took from before start to after stop (external interval), from after start to before stop (internal interval), and reading the elapsed interval "of the timer". In Poco, the elapsed interval cannot be obtained, and in Asio it is the difference from timer interval and remaining time. The results are shown in table 1. UM timer has almost no difference between all three readings, showing that both start and stop functions perform faster and the timer is more consistent with the externally measured values.

The second test consisted in letting the timer run to expiration and measure the elapsed time in the callback. More specifically, we measured the time

Table 1: Average intervals (in ms) for stopping a timer 50ms after start.

	External	Internal	Elapsed
Asio	53.6	50.7	51.5
Poco	55.6	51.2	–
UM	51.5	51.3	51.3

Table 2: Average intervals (in ms) for the expiration of a 100ms timer.

	Callback external	Callback internal	Timer elapsed
Asio	103.2	100.6	102.1
Poco	103.0	99.9	–
UM	101.8	101.7	100.9

Table 3: Average intervals (in μs) for restarting a timer.

	Asio	Poco	UM
Restart	351	1326	287

from before start to callback entry (callback-external interval), from after start to callback entry (callback-internal interval), and the elapsed interval reported by the timer (for Asio, it is the difference between the expires_at() value and the before start instant). The results are shown in table 2. UM::Timer offers the more precise temporization.

The third test consisted in restarting a timer if it was running. The time measured was the duration of the complete restart operation, and the results are shown in table 3. The results show once more that UM::Timer is faster and that Poco restart is more than 3 times slower.

6 CONCLUSION

Timers are an important feature for developing software interacting with the physical world. This paper proposes the UM::Timer library that offers a simpler API. To our knowledge, this is the first timer library implemented exclusively in standard C++ and, therefore, usable in all development environments supporting C++ and its threading library.

We have analysed a set of use-case scenarios, covering a wide variety of temporization needs, which show the functional advantages of UM timers. We also discuss non-functional characteristics of the proposed library. This includes a set of tests that show that it performs better than two widely known libraries, namely in terms of speed, precision and consistency.

Future work includes the study of an implementation based on a thread pool, aiming to augment its efficiency and possibly its performance. This will be validated by more in depth tests, namely measuring CPU time and memory usage. Since C++ provide the means to control its threads scheduling using platform specific mechanism, this work may also be extended with real-time dedicated functionalities.

ACKNOWLEDGEMENTS

This work has been supported by FCT - *Fundação para a Ciência e Tecnologia* in the scope of the project: UID/CEC/00319/2013.

REFERENCES

Applied Informatics Software Engineering GmbH 2010, *Multithreading: Doing things in parallel with POCO*, Available from: <http://pocoproject.org/slides/130-Threads.pdf>. [10 February 2015]

Henning, M. 2004. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing*.

ISO/IEC 2011, *Information technology – Programming languages – C++*, ISO/IEC 14882:2011.

ISO/IEC 2014, *Information technology – Programming languages – C++*, ISO/IEC 14882:2014.

Josuttis, N. M. 2012, *The C++ standard library: a tutorial and reference*, 2nd ed, Addison Wesley Longman.

King, D. E. 2009. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research*.

Kohlhoff, C.M. 2014, *Boost.Asio*. Available from: <http://www.boost.org/doc/libs/1_57_0/doc/html/boost_asio.html>. [9 February 2015]

Mathewson, N. 2012, *Fast portable non-blocking network programming with Libevent*. Available from: <http://www.wangafu.net/~nickm/libevent-book/TOC.html>. [9 February 2015]

Mitchell, S. 2013. *SDL Game Development*, Packt Publishing.

Reenskaug, T., Wold, P., Odd, A.L., 1996, *Working with objects: The OOram Software Engineering Method*, Manning Publications/Prentice Hall.

Robinson, M. 2013. *Getting started with JUCE*, Packt Publishing.

Schmidt, D. C. and Huston, S. D. 2003. *Systematic Reuse with ACE and Frameworks*, Addison-Wesley Longman.

Stroustrup, B. 2005, 'Abstraction and the C++ Machine Model', in *Embedded Software and Systems*. eds Z. Wu, C. Chen, M. Guo & J. Bu, Springer Berlin Heidelberg, LNCS, pp. 1-13.

Stroustrup, B. 2014, *Programming – Principles and Practice Using C++*, 2th Edition, Addison-Wesley.

The Qt Company 2015, *Qt*. Available from: < http://www.qt.io/application-development/>. [9 February 2015]