Clock module implementation

Guillermo Aguirre and Ludovic Thomas

Introduction

In this document we present the main characteristics of the `clock` module that allows the implementation of different time notions within the ns-3 simulations. We believe that the `clock` module extends the capabilities of the simulator, allowing a large variety of new scenarios. In particular, the development was motivated by the need for simulating IEEE Time Sensitive Networks (TSN). In such networks, time-synchronization protocols distribute a unique time notion across the nodes and end-stations, but with some inaccuracy. Simulating the effects of time-synchronization protocols at large, their interaction with other network mechanisms and their performance on different types of networks could now be facilitated using the `clock` module. We propose a set of clear interfaces and classes that would allow researchers to future add different clock behavior models in ns-3.

Related work and contributions

The idea of different time notions in ns-3 has been prowling for a long time. 5 years ago, as a part of a work in GSoC (1), Matthieu developed a first approach in order to implement a clock per node. In this first approach, the node base class is rewritten to behave as a scheduler. The *Node* class extends from the simulator interface and $Simulator :: Schedule()$ is substituted by $node- > Schedule()$. This would probably have led to an efficient implementation but it required that any ns-3 model issuing $Simulator :: Schedule()$ calls shall replace them by $node- > Schedule()$ calls. It represents a huge amount of work in re-writing many ns-3 modules, in addition to changing the *Node* class itself. We believe that the notion of local clocks should be enabled without changing any of the already existing ns-3 modules.

Another project that introduced the concept of local clocks was presented in (2). According to their paper, the authors focused on simulating IEEE 1588 synchronized networks, with very little flexibility on the clock models or on the applications that benefit from the local-time mechanism.

In the present project, we rely on both the ideas and the limitations of (1) and (2) and we propose an open interface that:

• allows to introduce independent local clocks, one per *Node*

• comes as an independent module and does not require any change in already-existing modules. After successful configuration of the simulation, all the $Simulator :: Schedule()$ calls issued by already-existing ns-3 models will automatically be interpreted with respect to the local clock of the considered *Node*.

• provides an interface for designing more complex clock behavioral models.

• provides an interface for updating the clock behavioral model during simulation time. For example, in an IEEE 1588 node, when a synchronization message is received, the synchronization function typically correct the frequency of the node's clock, based on the received message. In ns-3, this can be simulated by having an IEEE1588 *Application* that triggers an update of the clock model (with the new frequency), using the interface that we provide. This interface is the basis for analyzing the interaction between local clocks and network events/mechanisms.

The document is organized as follows. We first describe the model developed in the module. We then show an example that validates the model and explains how to setup a simulation. Last, we evaluate the processing overhead introduced by the module.

Model

The model is depicted in figure 1. The `clock` module is composed of two main classes that are *LocalTimeSimulatorImpl* and *LocalClock* and of one interface, *ClockModel*.

The *LocalTimeSimulatorImpl* substitute the default simulator implementation and it slightly diverts from it to include new functionalities that allows to simulate time notions in ns-3. Additionally, a *LocalClock* object is aggregated to each *Node*. It manages the local clock and provides access to the clock model.

Clock Model. *ClockModel* is the interface for the clock behavioral model. It provides the minimum set of functions that a clock model should implement. Any clock model developed should extend from this interface. In order to introduce the notion of local time in ns-3, the clock model must provide a mapping between the local time and the simulator time (also called global time, or true time). The relative time-function can capture clock non-idealities in order to introduce realistic clocks. We are aware about the difficulty of creating realistic clock models as mentioned in (3). However, the development of realistic clock models its out of the scope of this project. We present the interface that a clock model should use in order to be introduced in ns-3 and we provide a basic implementation: *PerfectClockModelImpl*. We believe that this paves the way towards achieving realistic simulations with realistic clock models.

Local Clock. A *LocalClock* object is aggregated to every *Node* using the ns-3 aggregation system. It represents the interface between the clock model and the node. It also tracks all the events that have been scheduled by this node with the purpose of rescheduling them if needed by an update of the clock model. Events are rescheduled when the *ClockModel*

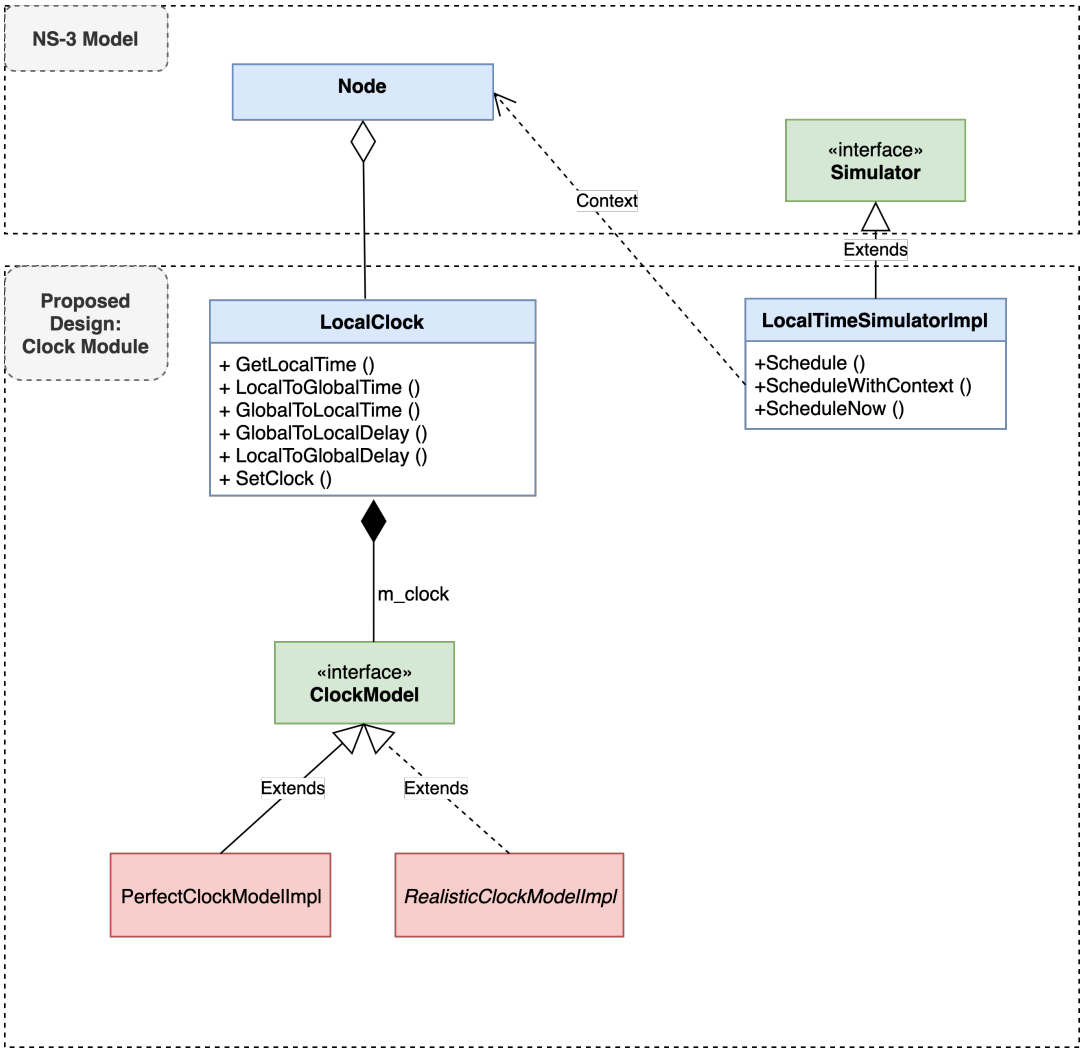

Fig. 1. UML design

for this *Node* is updated using the *SetClock()* function (to simulate an update of the clock's frequency for example). If some events have been scheduled using the previous clock model, then the mapping from local time to global time is no more valid for these events, so the *LocalClock* class reschedule them in accordance to the newly provided clock model.

To do so, events are retrieved from the event list of the *LocalClock* object and new execution times in the global time are calculated. New events are scheduled with the same *EventImpl* (using a copy of the pointer). Old events (which execution time does not correspond to the new clock) need to be removed from the scheduler. However, if events are cancelled using *Simulator::Cancel()*, it would be impossible to execute the event implementation provided to the new event. Indeed, in ns-3, *Simulator::Cancel()* cancels the *EventImpl*, not the *EventId*. To avoid this problem, old events along with new events are pushed to the *CancelEventsMap* map of *LocalTimeSimulatorImpl*.

The main attribute of the class *LocalClock* is m_clock, which is a pointer of type *ClockModelImpl* to the current clock model for this *Node*.

Simulator implementation. The main differences between *LocalTimeSimulatorImpl* and *DefaultSimulatorImpl* are:

- When an event is scheduled using the *Simulator::Schedule()* function, the delay is understood as being a local-time delay. The delay is then translated into a global-time delay before being inserted into the scheduler (the scheduler only operates in the global-time domain, and there is only one scheduler object, as in the current ns-3 code). See Figure 2.
- When a clock model is updated, *LocalTimeSimulatorImpl* keeps track of the events that have been rescheduled, and will not execute the old events.

When *Simulator::Schedule()* is called, the *Node* object is retrieved from the *NodeList* using the current context of the simulator. When the context do not correspond to any node, delays are considered to be in global time. Once the *Node* object is retrieved, *node->GetObject <LocalClock>()* is called to obtain the local clock, if aggregated[1]. Using the

[1]If the simulator doesn't find any aggregated *LocalClock*, it assumes that this *Node* relies on a perfect clock (local time = true time)

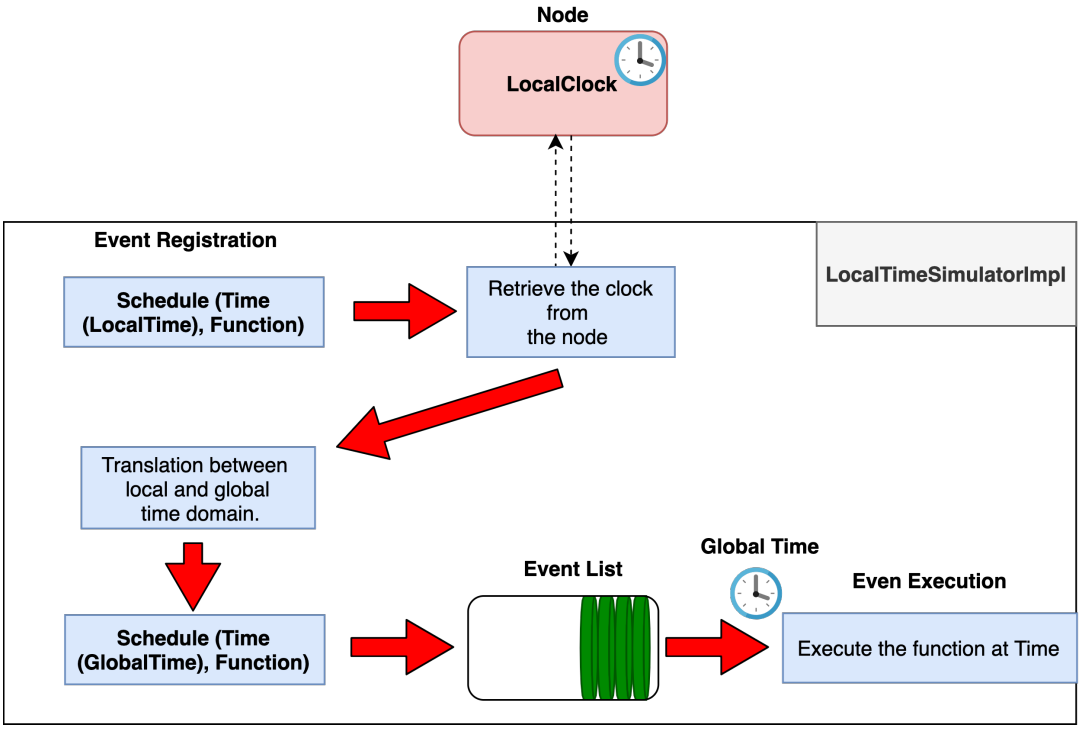

Fig. 2. Simulator steps for time translation when Schedule() is called

LocalClock object and its associated *ClockModel*, the local-time delay is converted into a global-time delay. The event is inserted in the scheduler using the global-time delay. Last, *LocalClock->InsertEvent()* is called in order to notify the *LocalClock* that the event has been scheduled. The *LocalClock* will keep track of it in case an update of the clock model requires to reschedule it.

When *Simulator::ScheduleWithContext()* is called, the same actions as in *DefaultSimulatorImpl* are taken. The delay is understood as being already a global-time delay. This is because:

• most of the *ScheduleWithContext()* calls in today's ns-3 modules are related to a packet transmission within a channel. The delay is normally[2] the channel propagation time (e.g. with *CsmaNetDevice*), which does not depend on the node clock. In particular, if a bit is 'on the wire' when a clock update is triggered on the sending node, the global-time value at which the bit is expected to reach the remote end is not changed. So we typically don't need to keep track of these events and we don't reschedule them after a clock update.

• the signature of *ScheduleWithContext()* does not allow to retrieve any *EventId*. Hence, even if we translate the time from local to global, we would never be able to reschedule or cancel the event. Is not recommended to change the signature of this function as discussed in (4).

When *Simulator::ScheduleNow()* is called, a call to the *Schedule()* function with local-time delay 0 is done.

One of the things to take into account is that *Simulator::Now()* returns the current global time and not the local time. This is because most of the *Simulator::Now()* calls in today's ns-3 are used to log or print the time at which an event occurs - and we definitively want the logs to be only prompted in the unique global-time domain. If the current local time is specifically required, it should be retrieved from the *LocalClock->GetLocalTime()* function.

Another particularity of this implementation resides in the *CancelEventsMap* map located in *LocalTimeSimulatorImpl*. When events are rescheduled, old events id (as key) with their corresponding new events (as value) are inserted in this map. As explained in the *LocalClock* section, we cannot use *Simulator::Cancel()* to remove old events or their implementation would be lost. In order not to execute events that should not be invoked (because the execution time attach to the event does not correspond to the new clock of the node), the *ProcessOneEvent()* function is slightly modified and checks the CancelEventsMap. If the EventId that is going to be executed is found as a key in the map, the event is skipped.

Other problem arises from the fact that, after some rescheduling of events, the original *EventId* is no more valid. Any process (i.e Applications) that schedule events will never realize about the change of the *EventId* due to the rescheduling. Therefore, there is a need to map between the original events and the rescheduled events. Suppose the following situation of figure 3. An event with id E_1 is scheduled by an application at some point in the future. Because the local clock runs slower, the event is moved even further in the global time. At some point, a clock update is triggered, creating a new *EventId* $E_1^{'}$ adjusted to the new local clock. The application only has the reference to E_1 and is completely unaware

[2]The Point to Point device is the only device where the transmission delay is not decoupled from the propagation delay. This can be solve calling to the channel transmission function after TransmitComplete has elapsed.

of $E_1^{'}$. Thus, when the *IsExpired(E_1)* function is called by the application, the answer of the simulator should be based on the new event $E_1^{'}$ as shown on figure 3. In order to cope with the problem, the *CancelEventsMap* is used to obtain $E_1^{'}$ from E_1, and *IsExpired(E_1)* returns *IsExpired($E_1^{'}$)*. Moreover, this is also useful when cancelling events. When the *Cancel()* function is called, the first step is to verify if the *EventId* is expired. The event is cancelled only if it is not expired. In such a manner, if the application wants to cancel event E_1, the simulator will first check if $E_1^{'}$ is expired. Assuming $E_1^{'}$ is not expired, the simulator will cancel the event implementation of E_1, which is the same than $E_1^{'}$. So, neither E_1 nor $E_1^{'}$ will be executed.

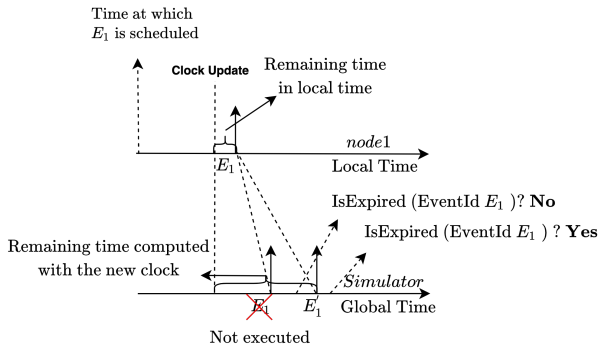

Fig. 3. IsExpired () function when event rescheduling

Validation

The clock model validation have been done using both examples and test. Unitary tests have been written to verify the internals of *LocalTimeSimulatorImpl* and *LocalClock* classes using *PerfectClockModelImpl*. They can be found in src/test/clock-test.cc. Here we present the main characteristics of the two-clocks-simple.cc available in src/clock/example.

The scenario model is shown in figure 4 and the simulation parameters are in table 1.

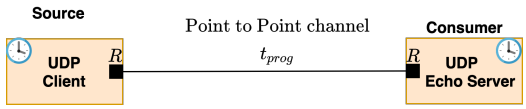

Fig. 4. Simulation Scenario

Scenario Settings	Node 1	Node 2
Application	UDPClient	UDPEchoServer
Interval Time (s)	3	
Net Device	Point to Point	Point to Point
Data Rate	5mbps	5mbps
Packet Size (bytes)	1024	1024
Propagation Delay	2ms	2ms
Simulation (s) Time	100	100
Clock	Non-ideal	Ideal

Table 1. Validation scenario parameters

The scenario uses two nodes connected by point-to-point net devices. *UdpClient* and *UdpEchoServer* applications are installed on the nodes to generate the traffic. The client has a non-ideal clock using the *PerfectClockModelImpl* class. This class allows to define the relative time-function between the global time and the local time. The relative time-function, shown in figure 5,is defined by a set of affine functions with a frequency offset (the slope of the function) as well as an initial offset (initial value when global time equals 0). When the fractional frequency offset is bigger than one, the local clock runs faster than the global time. On the other hand, when the fractional frequency offset is below one, the local clock runs slower. Likewise, when set to one, the clock runs at the same speed as the global time and is considered to be a perfect clock.

In order to build the shape of the function, we use several *PerfectClockModelImpl* objects and we trigger several clock updates to switch from one to the other: every 20 seconds, the clock update changes the frequency offset between clocks. In order to provide time continuity at each update, the initial offset of the function has to be re-computed. The relative time-function could also have been written as a single function, using a single clock model object, in which case we don't need the clock updates. Conversely, the server node runs a perfect clock.

In order to make the *clock* model run in the example, few changes need to be done in the main file. Remark, that the *clock* module can run with the preexisting code of ns-3. A few changes need to be done while coding the scenario model in the main file. The changes are as follows:

- First, the global variable *SimulatorImplementationType* needs to be changed, pointing to the *LocalTimeSimulatorImpl*.

```
GlobalValue::Bind ("
    SimulatorImplementationType",
    StringValue ("ns3::
    LocalTimeSimulatorImpl"));
```

- Second, The *ClockModel* implementation has to be created. In the case of the example, we use the *PerfectClockModelImpl*.

```
Ptr<PerfectClockModelImpl> clockImpl =
    CreateObject <PerfectClockModelImpl> ();

clockImpl -> SetAttribute ("Frequency",
    DoubleValue (freq));

clockImpl -> SetAttribute ("Offset",
    TimeValue (init_offset));
```

- Third, the *LocalClock* object must be created, the attribute m_clock set, and the object aggregated to the node.

```
Ptr<LocalClock> clock = CreateObject<
    LocalClock> ();

clock -> SetAttribute ("ClockModelImpl",
    PointerValue (clockImpl));

node -> AggregateObject (clock);
```

The full code for the example can be found in /src/clock/examples/two-clocks-simple.cc

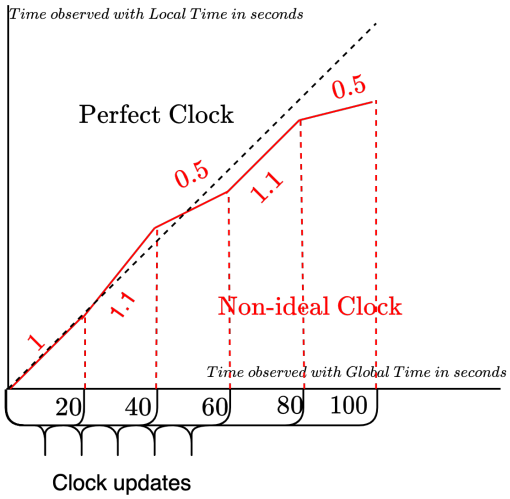

Fig. 5. Ideal and Non-Ideal clock model

Figure 6 shows the result obtained in the simulation. We represent the time at which the client node sends packets. As can be appreciated, the time at which packets are sent variates slightly depending on the frequency offset of the local clock. To compare the performance, we plot the same simulation, but without using the *clock* module. We can see how the clock on the client tends to go slower.

Between seconds 0 and 20, both clocks have same the frequency. However, after second 20 seconds, a clock update is triggered. This clock update modifies immediately the packets that are scheduled. As can be seen, straight after second 20 the slope increases, having the client clock running faster. The interval between packets change from 3 seconds to 2.7 seconds. Immediately after second 40 the slope decreases moving the interval time from 2.7 to 6 seconds. As we can see the frequency difference directly affects at which time packets are sent.

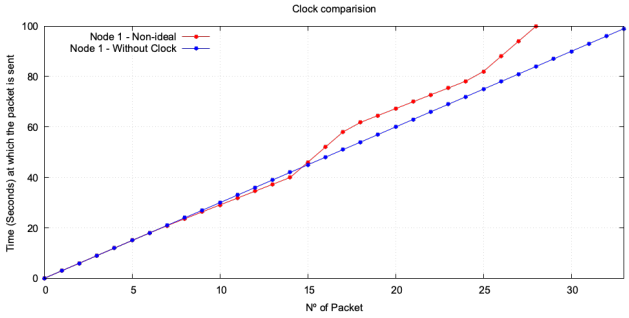

Fig. 6. Results obtained for Ideal and Non-Ideal clocks

Performance test

For the purpose of evaluating the overhead added by the module, the computation time has been taken into account. A flexible scenario has been considered, where the parameters such as the number of nodes and the clock models can be tuned. Half of the nodes in the network communicate with the other half through a switch. The application generates random data flows within some intervals of time. Nodes can be tuned without clocks, with clocks and with clock updates. In such a way that the three possible scenarios are tested. Remark that clock updates affect all the nodes of the network. Simulation have been performed with 50 nodes 7, 100 nodes 8, 200 nodes 9 and 500 nodes 10.

The computer used for this testing uses an Intel Core i5 6200U Processor with 8GB RAM and Ubuntu 20.04. Ten simulations for each scenario have been carried out in order to have an average value of the computation time. In order to compute the values, *Time* linux command has been used.

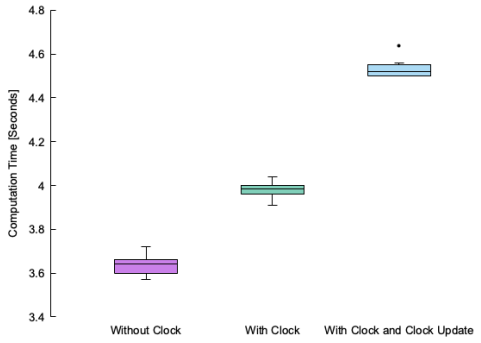

Fig. 7. Computation time with 50 nodes

As we could expect, the computation time increase with the number of nodes, as well as, with the different configurations of the clocks. Simulations show that in the case of using

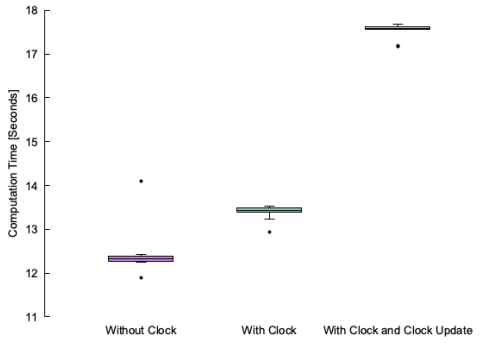

Fig. 8. Computation time with 100 nodes

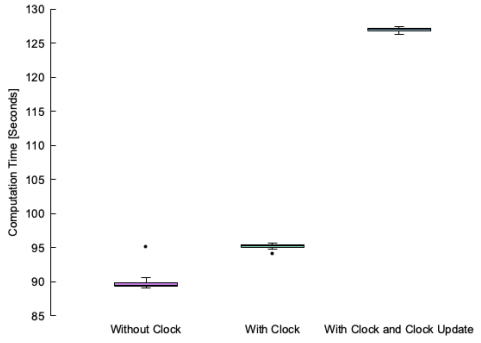

Fig. 9. Computation time with 200 nodes

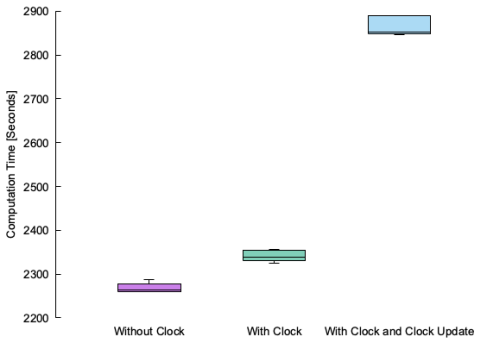

Fig. 10. Computation time with 500 nodes

clocks, the increment in time is around a 3% and 9%, with respect to the simulation without clocks. However, when using frequent clock updates, the simulation time skyrockets from 3% to 40% for simulations with bigger number of nodes. As we can see, this increase in time can be a drawback with big and complex scenarios that require constant clock updating and more computing resources.

Bibliography

1. Matthieu Coudron. Gsoc 2015 mptcp implementation. https://www.nsnam.org/wiki/GSOC2015MpTcpImplementation#On_the_per_node_clock, 2015.
2. Shouji Yoshida Mitsuyasu Kido Chikashi Komatsu Tatsuya Maruyama, Tsutomu Yamada. Ns-3 based ieee 1588 synchronization simulator for multi-hop network. https://ieeexplore.ieee.org/document/7324691, 2015.
3. Clock modelling difficulty discussion. https://www.nsnam.org/wiki/Local_Clocks.
4. Schedulewithcontex change of signature discussion. http://network-simulator-ns-2.7690.n7.nabble.com/Have-ScheduleWithContext-return-an-EventId-td30969.html.