DETERMINISTIC6G

Digest on Second 6GDetCom Simulator & Emulator Release

D4.4

The DETERMINISTIC6G project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement no 1010965604.

Digest on Second 6GDetCom Simulator & Emulator Release

Grant agreement number:	101096504
Project title:	Deterministic E2E communication with 6G
Project acronym:	DETERMINISTIC6G
Project website:	Deterministic6g.eu
Programme:	EU JU SNS Phase 1

Deliverable type:	Public Software Release
Deliverable reference number:	D4.4
Contributing workpackages:	WP4
Dissemination level:	PUBLIC
Due date:	30-04-2025
Actual submission date:	30-04-2025
Responsible organization:	USTUTT
Editor(s):	Lucas Haug Frank Dürr
Version number:	v1.0
Status:	Draft
Short abstract:	This deliverable describes the extensions of the simulator framework, which is used to validate the concepts that are developed in the project. It provides an overview of the enhancements since the first release and describes the design rationales behind these implementations. The core features of the second release of the simulation framework are: (1) an enhanced and more flexible architecture; (2) enhanced simulation models of time synchronization mechanisms; (3) simulation models for the Packet Delay Correction (PDC) mechanism; (4) network control plane interfaces to configure the network data plane supporting the simulation of dynamic scenarios. Besides the simulation framework, we also present two novel emulation frameworks: One framework allows to emulate the characteristic packet delay in existing TSN networks using Linux Qdiscs for converged 6G/TSN networks. The other framework enables the evaluation of security solutions for PTP-based time synchronization.
Keywords:	5G, 6G, software, validation, simulator, emulator, packet delay, processing delay, packet delay variation, dependable communication, TSN, PTP, time synchronization, gPTP, security

Contributor(s):	Lucas Haug (USTUTT) Frank Dürr (USTUTT) Simon Egger (USTUTT) Mahin Ahmed (SAL) James Gross (KTH)

	Samie Mostafavi (KTH) Gourav Prateek Sharma (KTH) Huu-Nghia Nguyen (MI) Edgardo Montes de Oca (MI) Marilet de Andrade Jardim (EAB) Oliver Höftberger (B&R) Emilio Trigili (SSSA)

Reviewers:	Marilet de Andrade Jardim Emilio Trigili

Revision History

v0.1	Initial Draft
v0.2	First complete version for internal review.
v0.3	Revised version for PMT review.
v0.4	Revised version including PMT review comments.
v1.0	Final version (submitted).

Disclaimer

This work has been performed in the framework of the Horizon Europe project DETERMINISTIC6G co-funded by the EU. This information reflects the consortium's view, but the consortium is not liable for any use that may be made of any of the information contained therein. This deliverable has been submitted to the EU commission, but it has not been reviewed and it has not been accepted by the EU commission yet.

Executive summary

This digest provides an overview of our open-source validation software for converged 6G/TSN networks released as D4.4 “DETERMINISTIC6G DetCom simulator framework release 2”. It describes the additions and new main features added since the first release of the simulation framework in D4.1 “DETERMINISTIC6G DetCom simulator framework release 1” [DET23-D41]. Even though title of D4.4 only mentions the simulator framework, this second release consists of three main frameworks: the 6GDetCom Simulator, the 6GDetCom Emulator, and the Secure PTP Emulation Framework. This digest is not intended as a programmer’s or user’s guide, but instead it gives an overview of the functionality and design rationales, as well as providing exemplary use-cases. A detailed description of the open-source releases can be found in the corresponding GitHub repositories (see Table 1). Links to specific documentation sites are provided in this document wherever necessary.

The core functionality of both, the 6GDetCom Simulator and 6GDetCom Emulator is the simulation/emulation of the characteristic Packet Delay (PD) of the virtual 6G TSN bridge (called DetCom node). The 6GDetCom Simulator already provides this functionality in the first release. This new release contains a refined and more flexible architecture as well as the option to replay delay traces instead of only histograms. Furthermore, we extend the simulator with the following additional features:

1. A full time-synchronization model for converged 6G/TSN networks based on gPTP [IEEE 802.1AS];
2. Simulation models for PDC, as described in Deliverable D2.3 [DET25-D23];
3. Two network control plane interfaces to support the simulation of dynamic scenarios: The first option adds a NETCONF [RFC6241] interface to the simulator, which allows for connecting an external TSN centralized network controller (CNC) to the simulated data plane. The other provides a direct scheduler interface within the simulator. These extensions allow for the simulation of adaptive schedules in scenarios with changing stream sets and dynamic PD distributions.

To facilitate the validation of real applications under test, such as the exo-skeleton, we developed the 6GDetCom Emulator as a second validation tool. Specifically, it can be used to analyze and evaluate the effect or characteristic 6G PD distributions onto real applications. The 6GDetCom Emulator is a Linux-based application, which delays packets passing through the emulator using characteristic 5G/6G PD distributions from delay measurements captured in the Deterministic6G testbed [DET24-D42].

Finally, we provide a Mininet-based [Min2025] emulation framework for the validation of secure PTP time synchronization, which is used to monitor, detect and localize Time-Delay Attacks (TDA).

Contents

1 Introduction

This digest describes Deliverable D4.4, the second release of the 6GDetCom validation frameworks, including extensions and enhancements over the first release of the 6GDetCom Simulator [DET23-D41] as well as two new emulation frameworks, namely the 6GDetCom Emulator and the emulator for the validation of secure PTP time synchronization.

The purpose of this digest is to highlight the updates since the first simulator release, the features of the new emulation frameworks, as well as to provide an intuitive description of the design principles applied to support the explanation of the software releases. This document is not a comprehensive technical documentation, nor a user manual. A detailed software documentation, including source code explanations, class descriptions, example showcases, and configuration guidelines, is included within the software package itself. Where appropriate, this digest links to these parts of the repository. The complete software and related resources, such as container images and configuration files, are accessible through the project's GitHub repository, with an archived version available on Zenodo. Links to these resources are provided in Table 1.

Component name	License	Link
6GDetCom Simulator	LGPL v3	GitHub: DETERMINISTIC6G/6GDetCom_Simulator Zenodo DOI: https://doi.org/10.5281/zenodo.10401976
6GDetCom Emulator	GPL v3	GitHub: DETERMINISTIC6G/6GDetCom_Emulator Zenodo DOI: https://doi.org/10.5281/zenodo.15305264
PD Datasets	Multiple	GitHub: DETERMINISTIC6G/deterministic6g_data Zenodo DOI: https://doi.org/10.5281/zenodo.10405084
NETCONF interface for INET	LGPL v3 and partially BSD-3-Clause	GitHub: https://github.com/DETERMINISTIC6G/netconf-for-inet Zenodo DOI: https://doi.org/10.5281/zenodo.15305259
Secure PTP Timesync Emulator	Apache 2, MIT	GitHub: https://github.com/deterministic6g/ptp-in-mininet Zenodo DOI: https://doi.org/10.5281/zenodo.15305270

Table 1: Overview over subcomponents of this deliverable.

1.1 Relation to Other Work Packages

Within the technical work packages of the DETERMINISTIC6G project, Deliverable D4.4 is part of WP4 “6G deterministic communication validation framework”. The relation of WP4 to the other work packages of this project is depicted in Figure 1. D4.4 takes features and concepts from WP2 “6G centric enablers for deterministic communication services” and WP3 “6G convergence enablers for

deterministic communication" and develops validation frameworks to validate these features and concepts through simulation and emulation models of these features. The validation is based on use-cases defined in WP1 "Vision, architecture and system aspects for deterministic e2e communication with 6G".

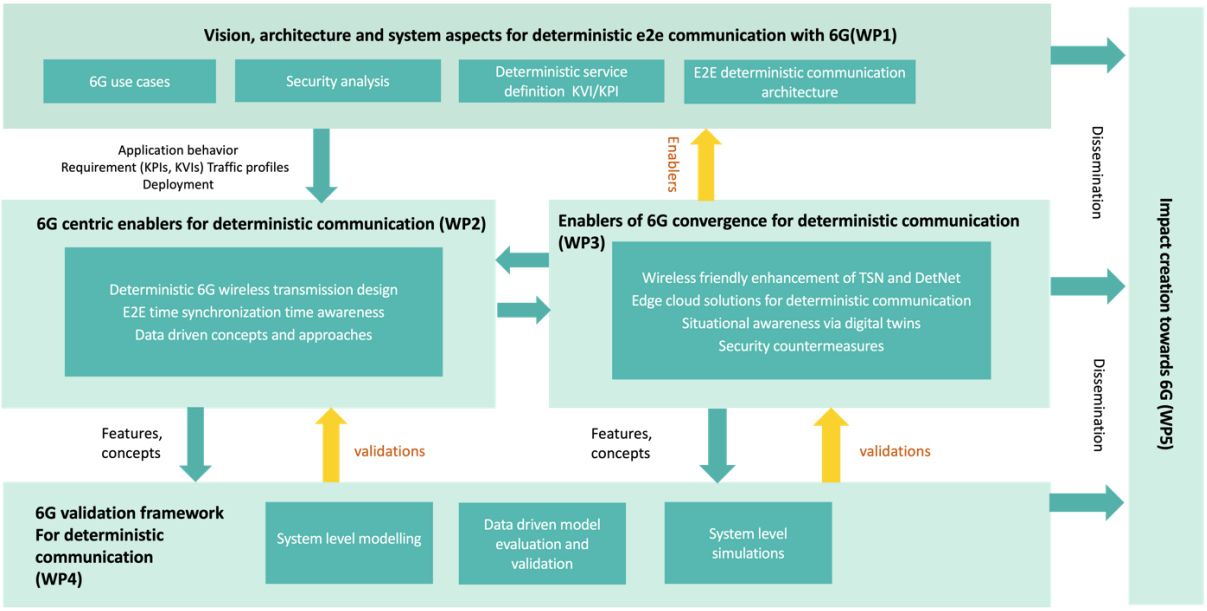


Figure 1: Relation of WP4 to other work packages.

In more detail, Deliverable D4.4 is tightly coupled with the following other deliverables:

- D1.1 "DETERMINISTIC6G use cases and architecture principles" [DET23-D11] provides use cases and applications, such as the exo-skeleton, which will utilize the validation framework for validation.
- D2.1 "First report on 6G centric enablers" [DET23-D21] introduces the concept of PDC. An implementation of PDC in the simulator is explained as part of this deliverable.
- D2.2 "First Report on the time synchronization for E2E time awareness" [DET23-D22] describes the time-synchronization architecture, which we implement in the 6GDetCom Simulator as part of this deliverable.
- D2.3 "Second report on 6G centric enablers" [DET25-D23] includes a validation analysis of PDC methods using WP4's validation framework.
- D2.4 "Report on the time synchronization for E2E time awareness" [DET25-D24] contains a description of time-synchronization architectures for converged 6G/TSN networks, which we simulate using our framework, as well as the validation results of these concepts. Results based on our emulator for the validation of secure PTP time synchronization are also provided in D2.4.
- D3.1 "Report on 6G convergence enablers towards deterministic communication standards" [DET23-D31] and D3.4 "Optimized deterministic end-to-end schedules for dynamic systems" [DET24-D34] define end-to-end scheduling concepts that are simulated in the simulation framework.

- In D3.2 “Report on the Security Solutions” [DET23-D32], a security-by-design approach was introduced. This deliverable contains an emulation framework that implements this approach to secure End-to-End (E2E) time synchronization using PTP.
- D4.1 “DETERMINISTIC6G DetCom simulator framework release 1” [DET23-D41] described the initial simulation framework release, which is extended in the second release of D4.4.
- D4.2 “Latency measurement framework” [DET24-D42] and D4.3 “Latency measurement data and characterization of RAN latency from experimental trials” [DET25-D43] provide the essential latency measurement from experiments in a 5G/6G testbed that are utilized by the simulation and emulation framework to simulate and emulate the characteristic packet delay.

1.2 Objective of the Document

The major objective of WP4 is the development of tools and performing evaluations to validate the features and concepts developed in WP2 and WP3, based on use-cases from WP1. This deliverable focuses on the first part of the major objective: The development of tools. The objective of this document is to provide insights on how the recent developments of our software tools – the 6GDetCom Simulator, 6GDetCom Emulator and Secure PTP Emulation Framework – enable this validation and evaluation of features and concepts.

While the major contribution of this deliverable towards WP4’s objectives are the software tools (see Table 1), the purpose of this document is to complement the open-source code and documentation by providing a high-level description of design choices and functionality. Where necessary, exemplary showcases are provided to support the description of the features. A final validation of use-cases and concepts is scheduled for the upcoming deliverable D4.5 “Validation Results”.

1.3 Structure and Scope of the Document

The content of this digest is structured as follows:

First, in Section 2, we present the extensions and enhancement of the 6GDetCom Simulator since the first release. We start this section with a description of an enhanced and more flexible architecture. This is followed by the description of newly added functionality, including the following: A DelayReplayer, allowing the replay of delay traces inside of the simulation, Simulation models for PDC, gPTP-based time synchronization, and two approaches for the evaluation of dynamic scenarios including network control plane features.

In Section 3, we present our new 6GDetCom Emulator. We start this section with a description of the main feature – emulating the characteristic PD of a virtual 6G bridge – and a description of its Linux QDisc based architecture. Following that, we evaluate the performance and limitations of this emulation framework.

In Section 4, we provide a technical description of a Mininet-based emulation framework for analyzing PTP-based time synchronization in converged 6G/TSN networks to monitor and detect Time-Delay Attacks (TDA).

Finally, we provide a conclusion of this digest in Section 5 including an outlook into future work.

2 6GDetCom Simulator Framework

In the first release of the 6GDetCom Simulator [DET23-D41], we proposed an OMNeT++ [OMN25] and INET [INE25] based simulator for simulating and evaluating TSN concepts in converged 6G/TSN networks. This second release is based on this first release and refines the initial architecture and provides additional functionality as described in the upcoming sections.

2.1 DetCom Node Architecture

The core component of the 6GDetCom Simulator is the so-called 6GDetCom node, which models a logical (wireless) 5G/6G TSN bridge. In our initial design to simulate the characteristic PD of a 6GDetCom node, we extended the implementation of an INET *TsnSwitch*[1] with a Delayer module to simulate the characteristic PD of logical 5G/6G TSN bridges. With the integration of new features into the simulator, such as PDC and Time Synchronization, it became obvious that the initial design of the 6GDetCom node needs to be further enhanced to facilitate the integration of these and other new features. To this end, we implemented a more elaborate and flexible approach in the following.

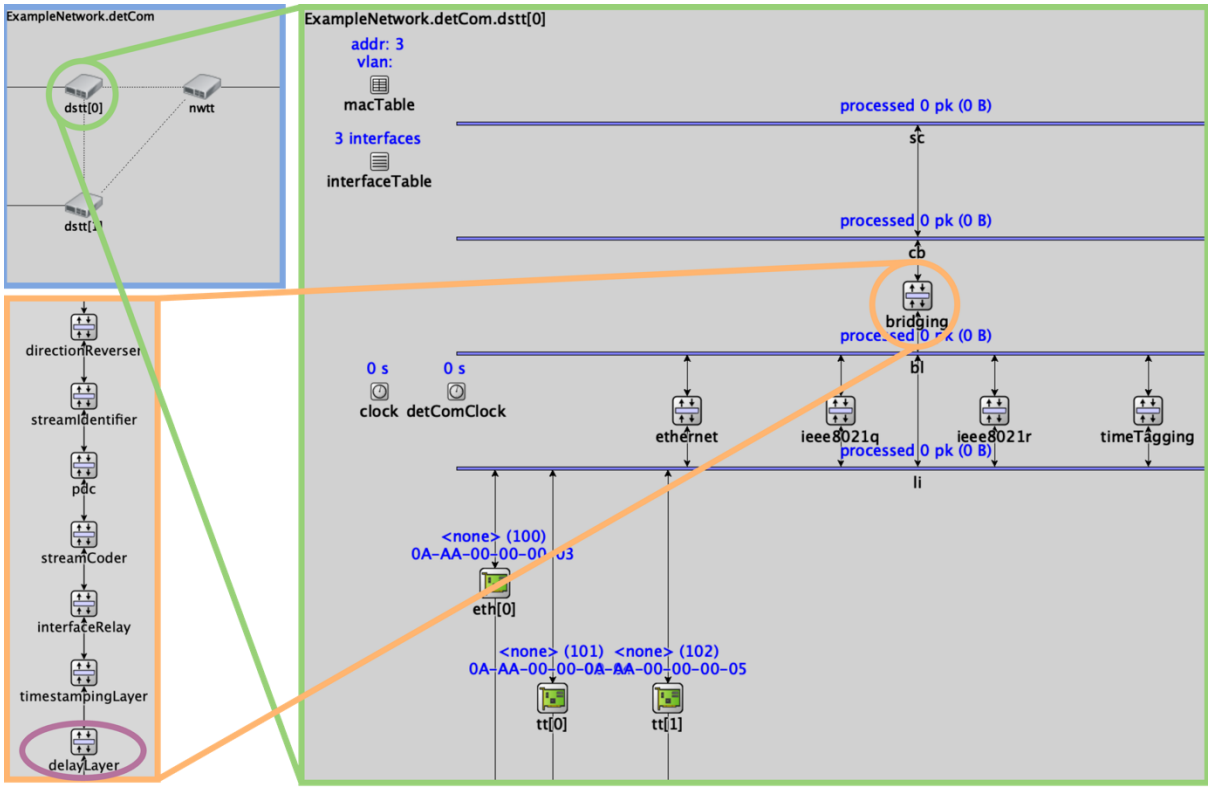


Figure 2: New architecture of the 6GDetCom node.

Instead of implementing the *DetCom*[2] node as a single *TsnSwitch*, we now implement the *DetCom* node as a combined module of multiple *TsnTranslator*[3] (TT) modules. The example in Figure 2 contains

[1] https://doc.omnetpp.org/inet/api-current/neddoc/inet.node.tsn.TsnSwitch.html
[2] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.devices.DetCom.html
[3] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.devices.tsntranslator.TsnTranslator.html

the structure of this new *DetCom* node (blue). It consists of one network-side TSN Translator (*nwtt*) and two device-side TSN Translators (*dstt[0]* and *dstt[1]*).

Note: To stick with the naming convention and overcome the limitations of module naming in OMNeT++ and INET, the names of modules in Figure 2 and the following sections may deviate from the official names. For example, network-side TSN Translator typically named NW-TT is named *nwtt* within the software, DS-TTs are called *dstt*. And a TSN Translator is called *TsnTranslator*. To this end, the following sections always use *italic* formatting to denote a reference to the software naming.

Inside of the *DetCom* node, each TT (*dstt[0], dstt[1]* and *nwtt* in Figure 2) serves the purpose of a half-bridge, i.e. the ingress TT provides ingress TSN features, such as Per Stream Filtering and Policing (PSFP), and the egress TT provides egress TSN features, such as shaping mechanisms. To this end, every TT in our implementation extends an INET *TsnSwitch* already providing the necessary TSN features.

Inside of the *DetCom* node, all TTs are connected to each other using a tt-interface (*tt[0]* and *tt[1]* inside of *dstt[0]* in Figure 2). For example, *dstt[0]* connects to *nwtt* using it's tt-interface *tt[0]* and connects to *dstt[1]* using it's tt-interface *tt[1]*. A tt-interface is based on the Ethernet interface of INET but modified to transmit frames between TTs without any delay (i.e. no transmission and no propagation delay). Instead, similar to our first release, the core of our simulation framework is the Delayer component, which simulates the characteristic packet delay of the 6G system. In the new architecture this Delayer component (purple) resides inside the bridging layer (orange) of each TT. The delay configuration works by setting the uplink and downlink delay separately for every *dstt*. As shown in Listing 1, the configuration can be achieved by loading delay histograms, by specifying a static delay, or by using a built-in close form distribution (see Deliverable D4.1 [DET23-D41] for more details about the different options to specify delay distributions, including also our delay histograms from real measurements in our 5G/6G testbed).

```
*.histogramContainer.histograms = {uplink: "uplink.xml", downlink: "downlink.xml"}
*.detCom.dstt[0].delayUplink = rngProvider("histogramContainer","uplink")
*.detCom.dstt[0].delayDownlink = rngProvider("histogramContainer","downlink")
*.detCom.dstt[1].delayUplink = 5ms
*.detCom.dstt[1].delayDownlink = normal(5ms,1ms)
```

Listing 1: Example delay configuration

It is important to note, that the behavior of our simulated DetCom node did not change from a TSN perspective for the existing simulations. However, as previously mentioned, it lays the basis for the implementations explained in the next sections.

2.2 DelayReplayer

In our previous release, the only way to use the measurements was the usage of histograms. However, this approach has a few limitations. For example, the drawn delay values are independently identically distributed (i.i.d), which makes it hard to simulate the correlation between the delay of different TTs or to simulate slow fading effects of moving devices or an increased load on the network. To this end,

we extend our simulation framework by a so-called *DelayReplayer*[4], which allows to replay delay traces in two different modes: The first mode delays frames in the given order (cycle mode), the other delays frames based on timestamps (timestamped mode).

The configuration of the *DelayReplayer* works similarly to the configuration of delay histograms, as shown in Listing 2. In addition to the configuration of the filename, an offset can be applied. The meaning of this offset is described in the following.

```
*.delayreplayerContainer.delayreplayers = {
          uplink: {file: "trace_timestamp.csv", timestampOffset: 5ms},
          downlink: {file: "trace_cycle.csv", offset: 100}}

*.detCom.dstt[0].delayUplink = rngProvider("delayReplayerContainer","uplink")
*.detCom.dstt[0].delayDownlink = rngProvider("delayReplayerContainer ","downlink")
```

Listing 2: Delay Replayer configuration.

As mentioned above, the *DelayReplayer* supports two modes. The first mode, called cycle mode, requires a configuration file with one delay value per line. An example is given in Listing 3. In this mode, the delay values are picked in order, i.e., the first frame is delayed by the value given in the first line, the second frame is delayed by the value in the second line, and so on. An optional offset can be applied, this offset allows to start the cycle at a specific position in the file, e.g. 100 in the example above.

```
5.422592ms
5.051648ms
6.724608ms
6.25408ms
```

Listing 3: DelayReplayer configuration in cycle mode.

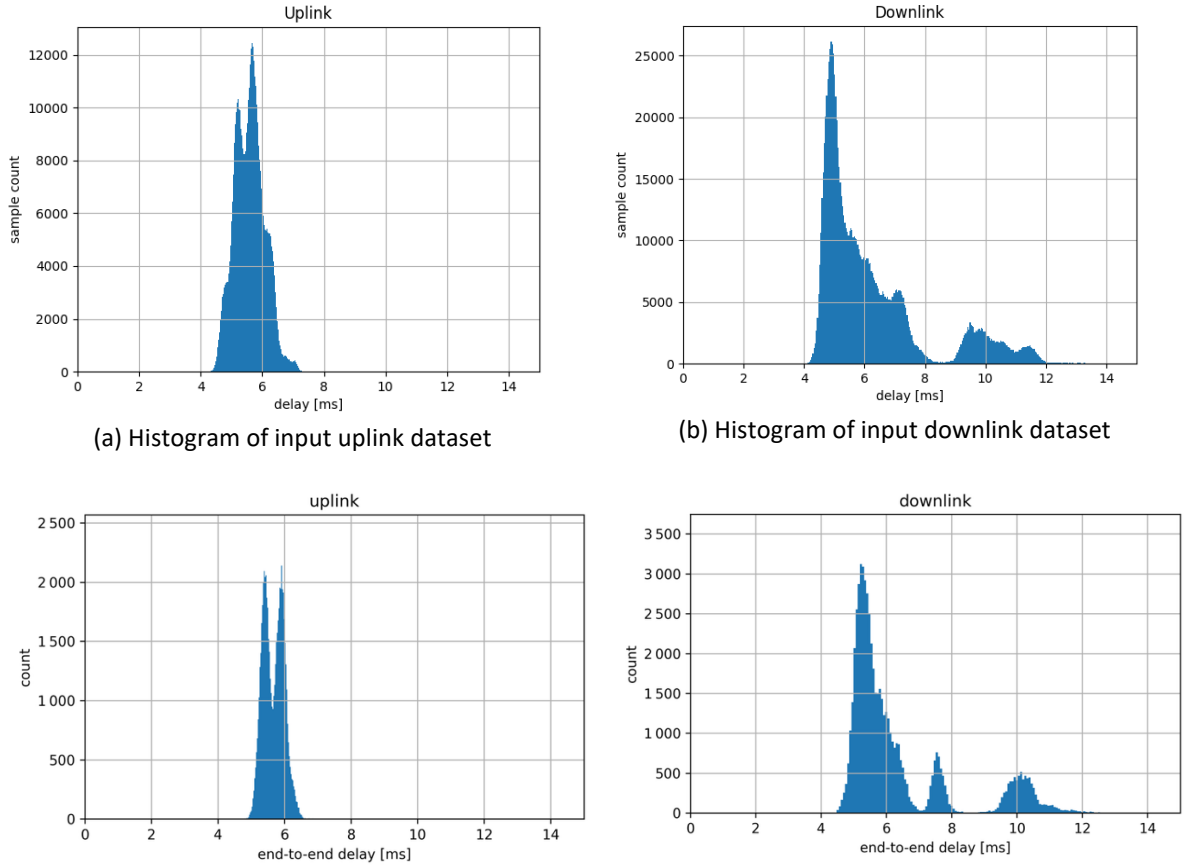
In the second mode, called timestamped mode, a file with two comma-separated values per line is required. The first value corresponds to a delay value, as before, and the second value corresponds to a timestamp. An example is given in Listing 4. In this case, if a frame arrives, the current simulation timestamp will be taken and the delay value of the closest smaller timestamp of the file will be taken. For example, for a frame arriving at time $17\ ms$, the next smaller delay value in the configuration is $10.435072\ ms$, so a delay of $5.825536\ ms$ is selected. As before, an offset can be specified. In the example configuration above, a time of $5\ ms$ would be added to the current simulation time before selecting the correct delay from the file.

```
4.88064ms,0ms
5.825536ms,10.435072ms
4.457472ms,19.926272ms
5.789696ms,30.438912ms
```

Listing 4: DelayReplayer configuration in timestamp mode.

[4] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.distribution.delayreplayer.DelayReplayer.html

Figure 3 shows the comparison of a histogram from the PD-Wireless-5G-1 dataset [HDM+23] on the top compared to the produced end-to-end packet delay of the simulation on the bottom using the raw measurement data of the same dataset to configure the *DelayReplayer*. This shows that the approach follows the delay traces and for a long-running simulation produces the same delay histograms. As expected, input and output histograms form a similar distribution, but as the simulation only covers a small timeframe of the whole dataset, they are not exactly the same.



(a) Histogram of input uplink dataset

(b) Histogram of input downlink dataset

(c) Resulting uplink end-to-end delay using timestamp mode

(d) Resulting downlink end-to-end delay using cycle mode

Figure 3: Comparison of input dataset (top) and resulting end-to-end delays of the *DelayReplayer* (bottom) using the timestamp mode (left) and cycle mode (right).

While the delay traces solve some problems introduced by histograms, they also have their own downsides. As the delay traces are not stateless (i.e., the current position in the delay file is stored), the same *DelayReplayer* key should not be used among multiple TTs. Instead, for every TT, the delay file needs to be loaded once, so a frame being delayed in one TT does not affect the delays of another TT. This increases the complexity of the simulation setup. Additionally, the delay trace files are usually bigger than their corresponding histograms leading to a greater memory footprint and simulation startup time.

In summary, both approaches have their own use-cases. On the one hand, if i.i.d. delay values from a histogram are sufficient for a simulation, they are easier and faster to setup. Additionally, some datasets are only provided as histograms without having access to the raw measurement data. On the

other hand, if the correlation of different distributions needs to be simulated or specific delay values should be mapped to specific timestamps in the simulation (e.g., when replaying delays from a captured history of delays), the *DelayReplayer* can be used to simulate these scenarios.

2.3 Packet Delay Correction (PDC)

In D2.1 "First report on 6G centric enablers" [DET23-D21] and D2.3 "Second report on 6G centric enablers" [DET25-D23], we present the approach of PDC. In summary, PDC allows to reduce the Packet Delay Variation (PDV) of a packet by utilizing the hold and forward buffering mechanism at the outgoing TT. As described in D2.1 and D2.3, there are multiple methods to implement PDC, for example the timestamp-based method and the virtual-timeslot-based method. Next, we describe the implementation of PDC in the simulator for both methods.

In the timestamp-based method, the ingress point (DS-TT or NW-TT) attaches an ingress timestamp t_i to the packet. The egress point (another DS-TT or NW-TT) then generates an egress timestamp t_e and uses both timestamps to generate a residence time $t_{res} = t_e - t_i$. The egress point then removes the ingress timestamp from the packet and holds the packet for time $t_{hold} = pdc - t_{res}$ where pdc refers to a predefined minimum packet delay. If pdc is equal to the maximum packet delay pd_{max}, all frames spent exactly pd_{max} inside of the 6GDetCom node. If, however, pdc is smaller than pd_{max}, the resulting Packet Delay Variation (PDV) is equal to the interval $[pdc, pd_{max}]$.

The virtual-timeslot-based method works similarly to the timestamp-based method but instead of using precise timestamps it uses timeslots of a predefined size T_{slot} and encodes the ingress and egress times using an integer number uniquely specifying a timeslot. The egress TT then calculates the number of timeslots to hold the frame and forwards the frame within the following timeslot. This approach is able to achieve a PDV equal to T_{slot}.

In the following we present two approaches on how to use these PDC approaches in our simulation framework. The first approach works by modifying the input delay histograms of the simulator. The second approach is an actual implementation of PDC into the simulator.

2.3.1 PDC Using Histogram Manipulation

As mentioned above, our first approach to mimic the behavior of PDC works by modifying the input histograms. To achieve this, we provide a script `histogram_manipulation.py` together with our histogram datasets, which is used to modify histograms as if PDC were active.

This script allows to take any input histogram and allows to specify a percentage p of frames that should be delay-corrected, e.g., if a percentage of $p = 0.7$ is chosen, that means the value of pdc is calculated such that $70\ \%$ of the histogram has a delay shorter than pdc and $30\ \%$ of the frames are contained in the interval $[pdc, pd_{max}]$.

An example usage of this this tool is shown in Listing 5 and the corresponding result is shown in Figure 4. As can be seen, the first $70\ \%$ of values are merged into a single bin (first orange bar) corresponding to the calculated pdc value.

```
$ python histogram_manipulation.py --hist "histogram.xml" --pdc 0.7
```

Listing 5: Example usage of the histogram manipulator.

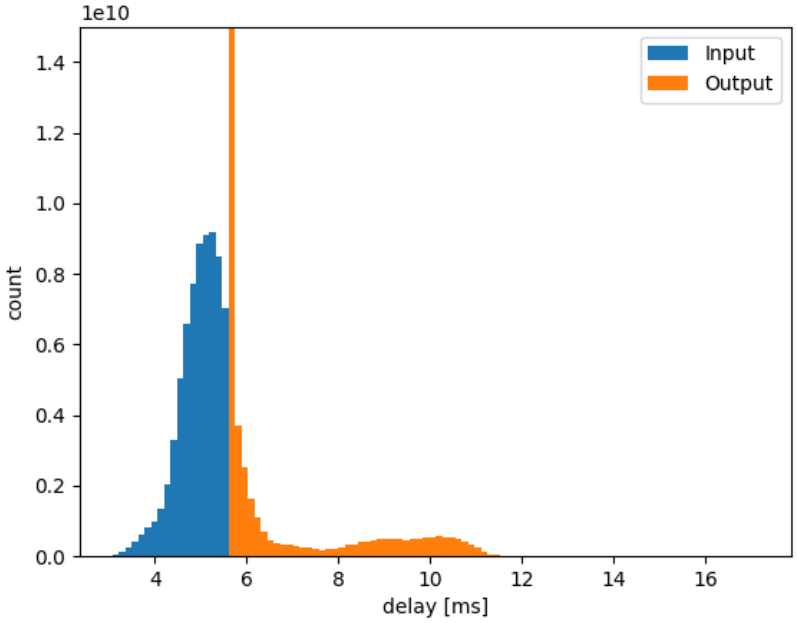


Figure 4: Original input histogram (blue) and resulting output histogram (orange) for 70% PDC.

This histogram modification approach is useful, when incorporating PDC into a scheduler which already supports delay histograms without requiring changes to the scheduler code.

2.3.2 PDC Implementation in the Simulator Framework

Additionally to the histogram manipulation method above, we also implement PDC as a functionality in the simulator. This allows for an easier setup and exploration of different PDC values without having to regenerate histograms as well as a support for PDC when using built-in probability density functions (like a normal distribution) instead of histograms.

Timestamping

Our PDC function in the simulator is focused on the timestamp-based approach. Thus, we extend our simulation framework with the timestamping mechanism explained above. All relevant components of this implementation are also shown in Figure 2.

When a frame enters the *DetCom* node, the following happens in the incoming TT: The frame is received through the incoming eth-interface and is tagged with an internal *ingressTimestamp* tag containing the current ingress time. Tags in OMNeT++ are only valid inside of the current module (i.e. the internal *ingressTimestamp* tag is only accessible inside the current TT). To this end, when the packet passes through the *timestampingLayer*[5] inside of the *bridging* module, the *timestampingLayer* checks if the next hop of the packet is another TT. If this is the case, it ensures that the *timeTagging*[6]

[5] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.timestamping.DetComTimestampingLayer.html

[6] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.timestamping.DetComTimeTagging.html

module will encode the *ingressTimestamp* into the frame, so it is accessible at the next TT. We omit the details of this process here, as this behavior is highly OMNeT++/INET specific. Note: As the delay of frames inside the current *DetCom* node architecture is solely based on our Delayer, the increased packet size does not influence the transmission time of the frame inside of the *DetCom* node.

When the frame reaches the second (outgoing) TT, the *timeTagging* module removes the encoded timestamp from the frame and again adds the internal *ingressTimestamp* tag to the frame to make it available to other components inside of the current TT. As the frame reaches the *timestampingLayer*, the *ingressTimestamp* is used together with the current time to calculate the residence time t_{res}, which is then added as an internal *residenceTime*[7] tag, which can be used by the following modules.

Configuration

The configuration of PDC works on a per-stream basis and is based on the *streamIndentifier* principle, which is also used in INET to enable PSFP and to map streams to PCP values.

Listing 6 shows an example PDC configuration. It contains two streams. The PDC values of Stream 1 is chosen such that $70\ \%$ of the frames from the expected interval are corrected. Stream 2 is configured to correct $99.99\ \%$ of the streams. The jitter parameter allows to additionally specify a random delay that is added after correction. This can be used to simulate the behavior of timeslots. In this example, a uniform additional delay is selected from the interval $[0ms, 0.5ms]$ simulating a virtual-time-slot based approach with a time-slot length of $500\ \mu s$.

```
# default pdc
delayreplayerContainer
*.detCom.nwtt.bridging.pdc.defaultPdc = 1ms

# per stream pdc
*.detCom.nwtt.bridging.pdc.mapping = [
              {stream:"stream1", pdc:"5.84ms"}, # 70% PDC
              {stream:"stream2", pdc:"10.52ms", jitter:"uniform(0ms,0.5ms)"}] # 99.99% PDC
```

Listing 6: Example PDC configuration.

Based on this configuration, the *pdc*[8] component (as shown in Figure 2) first identifies the stream based on INETs *streamIdentifier* and loads the PDC configuration for the specific stream together with t_{res} calculates t_{hold}. If $t_{hold} > 0$, the frame is delayed by the *pdc* component for a duration of $t_{hold} + jitter$.

Results

The result of the above configuration is shown in Figure 5. The blue dots represent the delays of the frames of Stream 1, while the orange dots represent the delays of the frames of Stream 2. As expected, the delays of Stream 1 form a line at $5.82\ ms$ as configured with some outliers above the line corresponding to the expected remaining $30\ \%$ of the frames. The delays of Stream2 form a line of width $500\ \mu s$ corresponding to the configured time-slot size.

[7] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.timestamping.ResidenceTimeCalculator.html

[8] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.devices.pdc.PdcDelayer.html

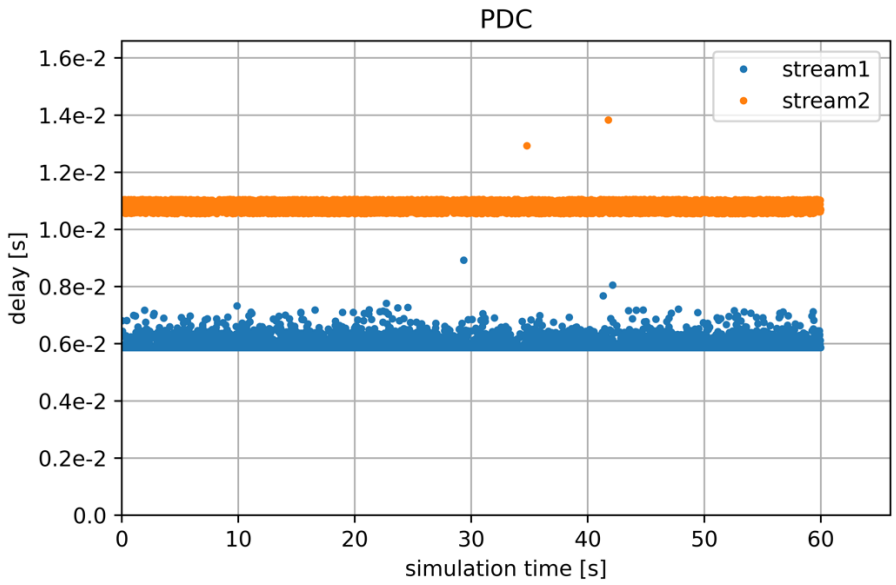


Figure 5: Simulation results of PDC configuration as in Listing 6.

2.4 Time Synchronization

In the first release of the simulator [DET23-D41], we gave a brief overview of the current implementation of time synchronization, pointing out required work to allow for the simulation of gPTP-based time synchronization in converged 6G/TSN networks. In this section, we present our changes and fixes to the gPTP implementation of the INET framework itself. These changes will be submitted to the INET codebase on GitHub as a pull request following the release of this deliverable. Furthermore, we provide the implementation details of gPTP inside the *DetCom* node.

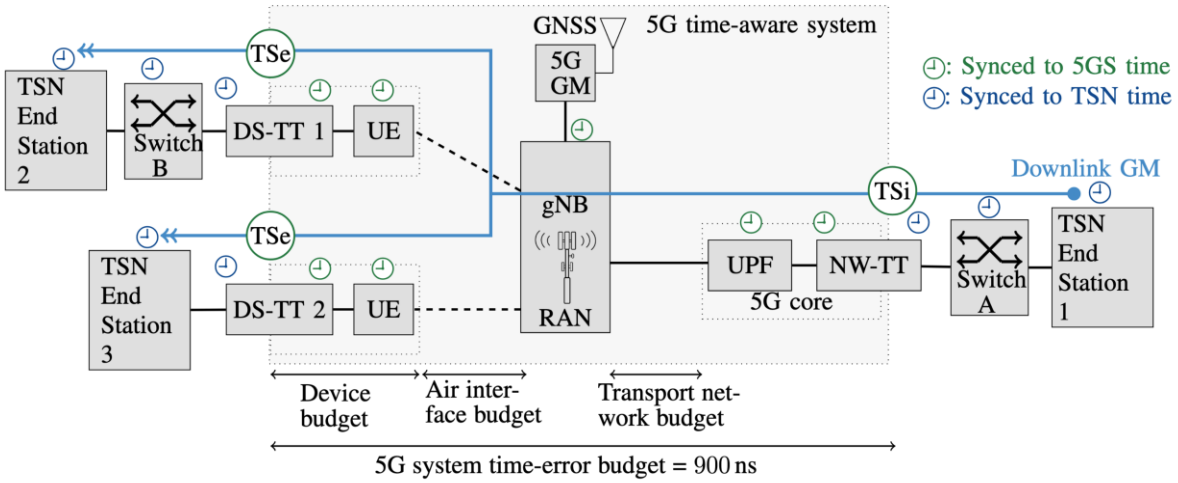


Figure 6: Architecture of gPTP-based time synchronization in converged networks.

The architecture of gPTP-based time synchronization in converged 6G/TSN networks has already been explained in detail in previous deliverables, such as D2.2 “First Report on the time synchronization for E2E time awareness” [DET23-D22] and D2.4 “Report on the time synchronization for E2E time

awareness" [DET25-D24]. Thus, we only provide a brief summary of the architecture here and refer the reader to these deliverables for a detailed description. Figure 6 shows the architectural overview to enable gPTP in converged networks. The figure contains two time domains, the 5G time domain (green) and the TSN time domain (blue), with the TSN translators (DS-TT, NW-TT) being on the boundary of the two time domains. These two time domains are independent to each other, and devices synchronize their clocks to their respective time domain – note that every TT maintains a 5G clock and a TSN clock. Most importantly, devices outside of the *DetCom* node are unaware of the 5G time domain, meaning the *DetCom* node as a whole needs to act like a (transparent) PTP relay instance.

In the following, we present how we implement this architecture in the simulator, starting with our modifications to the INET framework and continuing with the additions to our simulation framework.

2.4.1 Additions to the INET framework

The INET framework already includes a basic gPTP implementation for TsnDevices (ordinary clocks) and TsnSwitches (boundary clocks). As these basic features are insufficient to allow for a full evaluation of time synchronization in 6G/TSN networks, we extend the INET framework with additional features, namely an implementation of clock servos, a Best Transmitter Clock Algorithm (BTCA) implementation, and an extended Hot Standby implementation.

Clock Servos

The servo of a clock is responsible for correcting the clock drift of the time receiver (tR) compared to the time transmitter (tT). Currently, INET does not provide any clock servos. Instead, when a device receives all messages required to synchronize to the grand master (GM), it immediately jumps to the calculated time. A clock with this behavior is called a *SettableClock* in INET. This behavior is shown in green in Figure 7. However, this behavior introduces several challenges. On the one hand, a jump backward in time might result in events appearing to happen twice. Moreover, non-monotonically increasing clocks destroy the causal order of events (invalid "happens-before" relationship between events), which makes it impossible to correctly react to events based on their causal dependencies or analyze the causal dependency of recorded events ("did one event lead to another event?"). On the other hand, a jump forward in time might lead to events being skipped, or, if executed immediately lead to conflicts with other events. Therefore, in many cases, such jumps need to be avoided to prevent inconsistencies. Instead, the frequency (rate) of the clock being synchronized is controlled by the clock servo depending on the clock error (clock skew) to *gradually* converge to the time of the GM clock, also guaranteeing monotonically increasing clock values. To mitigate these issues, we implement a clock using a proportional-integral (PI) controller as a clock servo. Such PI controllers are a common choice for clock servos used in real-world implementations such as PTP for Linux (ptp4l) [Coc25]. Instead of jumping to the calculated time, the servo speeds up or slows down the clock to maintain the synchronization to the GM. The behavior of our *PiServoClock* is shown in orange in Figure 7.

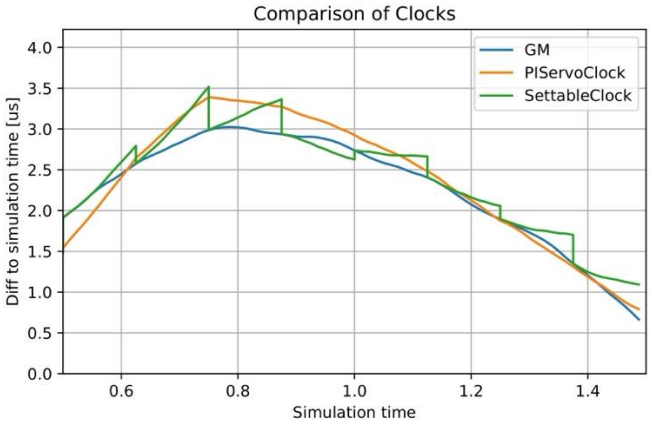
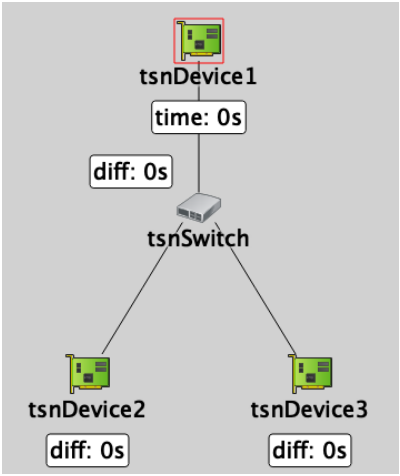


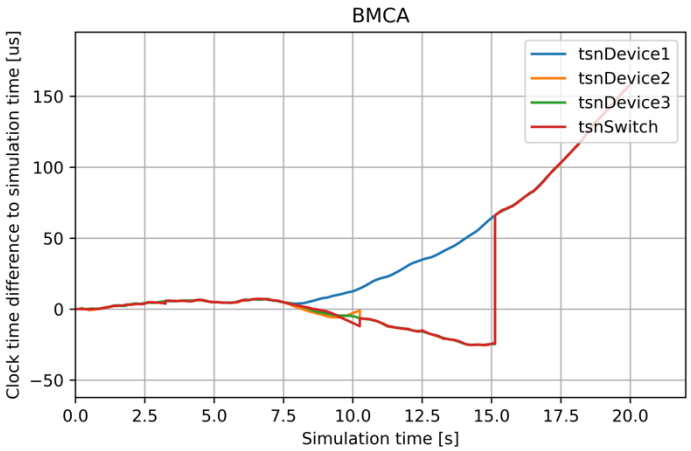
Figure 7: Comparison of a clock without a clock servo (SettableClock) vs a clock with a PI servo.

Best Transmitter Clock Algorithm (BTCA)

In the existing INET implementation, time synchronization domains are preconfigured in the configuration file of a simulation. This, however, is not suitable for changing dynamic environments. Instead, in a real network, the time synchronization architecture is typically configured automatically using the Best Transmitter Clock Algorithm (BTCA), previously known as the Best Master Clock Algorithm (BMCA). In order to select a GM in a network, devices periodically send Announce messages transmitting the state of their current GM. Upon reception of Announce messages, every device compares its own local clock to the received Announce messages based on user-defined parameters, such as the priority and parameters specifying the clock quality. Using this algorithm, the network eventually decides on a single GM in the network and establishes a synchronization tree.



(a) Example network



(b) Clock offset to master clock.

Figure 8: Simple BTCA scenario.

To allow for a simulation of time synchronization in dynamic environments, we extend the gPTP implementation of INET with an implementation of BTCA. As BTCA is responsible for modifying the

port states of gPTP, we implement BTCA directly into the existing Gptp[9] module. Our implementation allows the configuration of the BTCA relevant parameters, such as the grandmasterPriority and clockAccuracy. Additionally, the user needs to define the ports that should be considered for the BTCA evaluation, i.e., all ports that are not specified are disabled for time synchronization. An example configuration for the network shown in Figure 8a is provided in Listing 7.

```
*.tsnDevice*.gptp.bmcaPorts = ["eth0"]
*.tsnSwitch.gptp.bmcaPorts = ["eth0", "eth1", "eth2"]
*.tsnDevice1.gptp.grandmasterPriority1 = 2
*.tsnDevice3.gptp.grandmasterPriority1 = 3
```

Listing 7: Example BTCA configuration.

In our example scenario, we additionally disable the link between *tsnDevice1* and *tsnSwitch* in the interval $[7.5\ s, 14.5\ s]$ to evaluate the re-selection of a new GM. Figure 8b shows the resulting offset of the clocks to the simulation time. In the period between $0\ s$ and $7.5\ s$, all clocks are in sync. Then, after the link breaks, all clocks are starting to drift apart until $10\ s$, when *tsnDevice3* (as intended by the configuration) is selected as the new GM, from when on all devices of the intact network synchronize to *tsnDevice3*. Finally, when the link between *tsnDevice1* and *tsnSwitch* is re-established at time 14.5 s, all devices synchronize to *tsnDevice1* again.

Hot Standby

The INET framework already supports the synchronization of a single network device to multiple gPTP domains. However, in the current state, INET does not keep track of the synchronization state of these time domains, nor does it switch to the Hot Standby domain in case the primary domain becomes unavailable. Thus, we extend the already existing implementation by the missing features.

The multidomain support for gPTP in INET works by having a *MutliDomainGptp* module in a device, which consists of multiple single gPTP components, each uniquely mapped to gPTP domain number. Additionally, each device has a *multiClock* module consisting of multiple clocks modules. Each gPTP module (i.e. domain) is mapped to a clock within this *multiClock*. The *multiClock* has an *activeClock* parameter, which is used to specify, which of the clocks is used as the device clock.

We extend the *MultiDomainGptp* module with a *HotStandby* module keeping track of the synchronization state of the single gPTP domains. And, based on the specification in the IEEE 802.1ASdm Hot Standby amendment [IEEE 802.1ASdm], updates the *activeClockIndex* of the *multiClock* to switch between the primary and the Hot Standby domain.

[9] https://doc.omnetpp.org/inet/api-current/neddoc/inet.linklayer.ieee8021as.Gptp.html

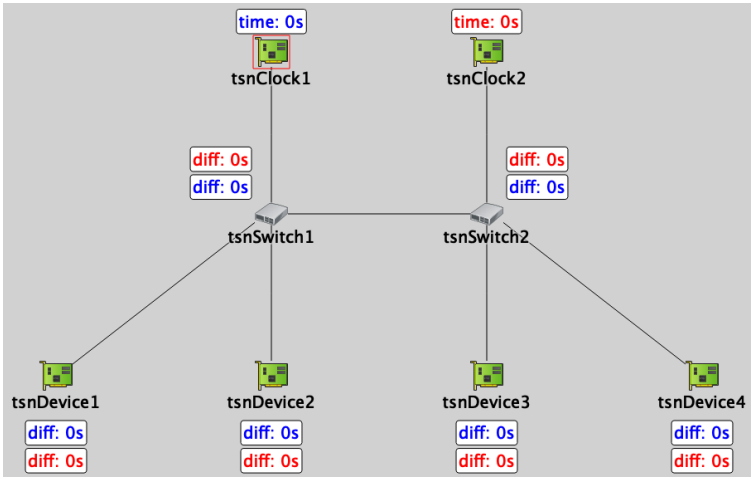


Figure 9: Network for the HotStandby showcase.

To showcase our Hot Standby implementation, we use the network as shown in Figure 9. This network consists of two time synchronization domains blue (Domain 1) and red (Domain 2), with *tsnClock1* and *tsnClock2* being the GMs of Domain 1 and 2, respectively. All other devices are synchronized two both domains with Domain 1 being the primary domain and Domain 2 being the secondary domain. We define two failure cases: In the first failure case in the interval *[3 s, 7 s]* the link between *tsnClock1* and *tsnSwitch1* is disabled, i.e. the GM of Domain 1 goes offline. In the second failure case in the interval $[10\ s, 14\ s]$, the link between *tsnSwitch1* and *tsnSwitch2* breaks, i.e. the network is partitioned into two parts.

Figure 10 shows the resulting clock drifts. In the first failure case, all network devices switch to the Hot Standby domain, diverging from the clock of the primary GM (*tsnSwitch1*). In the second failure case, the devices on the left side of the network remain synchronized to the primary GM, while the right side of the network switches to the Hot Standby GM (*tsnSwitch2*). In both failure cases, all devices return to the primary domain, as soon as it is synchronized again.

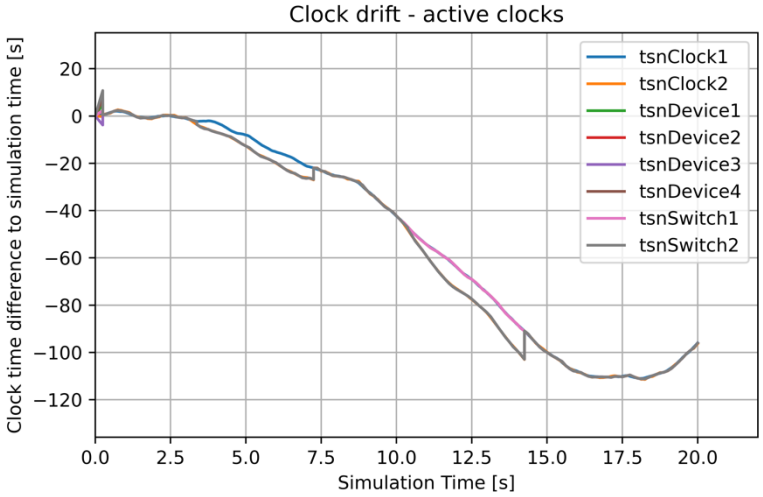


Figure 10: Clock drifts in the HotStandby showcase.

2.4.2 Additions to the 6GDetCom Simulator

Now, after having provided an overview of the new gPTP functions added to INET, we present in more detail the modifications to our simulation framework to support time synchronization in the context of converged 6G/TSN networks. This includes extensions to the implementation of the TTs as well as the implementation of the gPTP component within the TTs.

6G Clock Model

In order to enable the ingress and egress timestamping, we need to extend the TTs with a model for 6G clocks. To this end, we add the *detComClock* module to every TT. For these clocks, we provide a modified oscillator that allows to limit the maximum offset to the simulation time. We can use this oscillator to implement a simplified 6G clock model in the simulator. By default, the 6G clocks of the TTs are configured to drift apart by a maximum of $450\ ns$ to ensure compliance with 3GPP's budget of $900\ ns$ [3GPP-TS22104] between two TTs. An example of this configuration is given in Figure 11 for a DetCom node with three TTs.

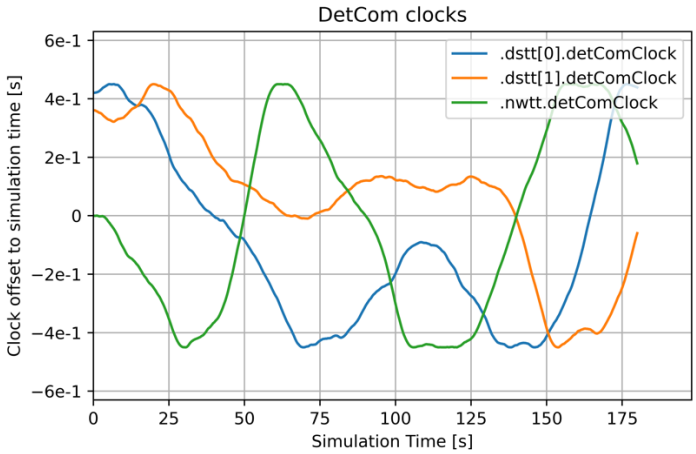


Figure 11: Clock drift of 6G clocks with a simplified model.

gPTP component modifications

As mentioned above, gPTP within TTs needs to provide additional functionality, mainly the ingress and egress timestamping mechanism. To this end, we extend the gPTP component in the following way.

For the Sync procedure, when a gPTP module inside a TT receives a Sync or FollowUp message from outside the DetCom node, it is handled as in any other TSN switch, and the TSN clock of the TT is synchronized according to the contents of the gPTP message. Afterwards, a Sync message is generated for transmission to the other TTs (given this TT is the time transmitter for any other TT in the synchronization topology). Together with the generation of this Sync message, the ingress timestamp *TSi* is generated for both time domains. We label these timestamps t_i^{tsn} and t_i^{6G} for the TSN and 6G time domain respectively. Timestamp t_i^{tsn} is used to calculate the residence time within the gPTP component before the generation of the Sync message, while t_i^{6G} is added as an internal tag to the gPTP FollowUp up message, so it can be encoded by the *timeTagging* module as described in Section 2.3.2. Upon reception of the Sync message on the second TT, the egress timestamps t_e^{TSN} and t_e^{6G} are generated for both time domains respectively. These two timestamps can be used to calculate the rate ratio between the TSN clocks and the 6G clocks. Upon reception of the corresponding FollowUp

message, the contained t_i^{6G} timestamp is decoded and used together with t_e^{6G} to calculate the residence time and correctly add it to the correction field. Afterwards, the gPTP module again behaves like inside a TSN switch and sends the Sync and Follow Up message to a device outside of the DetCom node.

Additionally, the gPTP module for TTs contains some more minor changes: As the link delay inside the DetCom node is calculated using the ingress and egress timestamping mechanism, no peer delay request and response messages are transmitted between TTs. If a TT is selected as the GM of a network, it generates the origin timestamps using the *detComClock* and directly synchronizes its own TSN clock to it.

A detailed analysis of time synchronization in converged 6G/TSN networks together with simulation results based on this implementation can be found in [DET25-D24].

2.5 Dynamic Scenarios

For a full validation of time-critical traffic in converged 6G/TSN networks, it is indispensable to also analyze the effect of dynamic behavior in a network. Dynamic behavior in a network for example includes:

Variable Stream Sets: In a dynamic setting, new devices might join or leave the network, or a device that is already part of the network wants to start or stop an application. Both cases result in the addition or removal of streams from the stream set, requiring the scheduler to re-calculate schedules. Additionally, a device might change its requirements, which might also require the adaptation or re-calculation of the schedule.

Changing Packet Delay Distributions: Slow and fast fading effects, often caused by device mobility or changes in the surrounding environment (e.g., obstacles, reflection, scattering), result in temporal variations of the wireless channel. Slow fading typically occurs due to large-scale movements (e.g., a user walking through a building), leading to gradual changes in signal strength and hence delay characteristics. In contrast, fast fading is induced by small-scale movements or multipath propagation, resulting in rapid fluctuations of the channel within short time intervals. Both fading effects can lead to a change in the delay distribution, eventually requiring an adaptation and/or re-calculation of the current schedule to adapt for these changes.

Variable Topology: The movement of devices or broken links might lead to a new network topology, also necessitating a re-calculation of the schedule, similar to the changing stream sets.

We designed two possible approaches to enable the simulation of dynamic scenarios in the context of our simulation framework. Our first approach is to extend the simulation framework with a real NETCONF interface, which ultimately enables to connect an existing CNC implementation to control the simulated TSN data plane. Obviously, a full implementation of all NETCONF features is complex and is, therefore, beyond the scope of our implementation. Instead, the provided implementation is an initial step towards such a NETCONF interface for the simulator. Our second approach is a direct interface between the simulation and a scheduler. We explain these approaches in detail in the following and discuss their respective advantages and disadvantages.

2.5.1 NETCONF Interface

In this section, we present our first approach, which extends INET with a NETCONF interface. The goal of this approach is to provide a NETCONF interface, so it can be used by a real Centralized Network Configuration (CNC). From the view of the CNC, the simulation should behave like a real network, whenever possible. This allows to evaluate the functionality of a CNC in a simulated environment without the need of adjusting its software. In the following, we first present the architecture of this approach, followed by a quick introduction to the usage and an example scenario.

Architecture

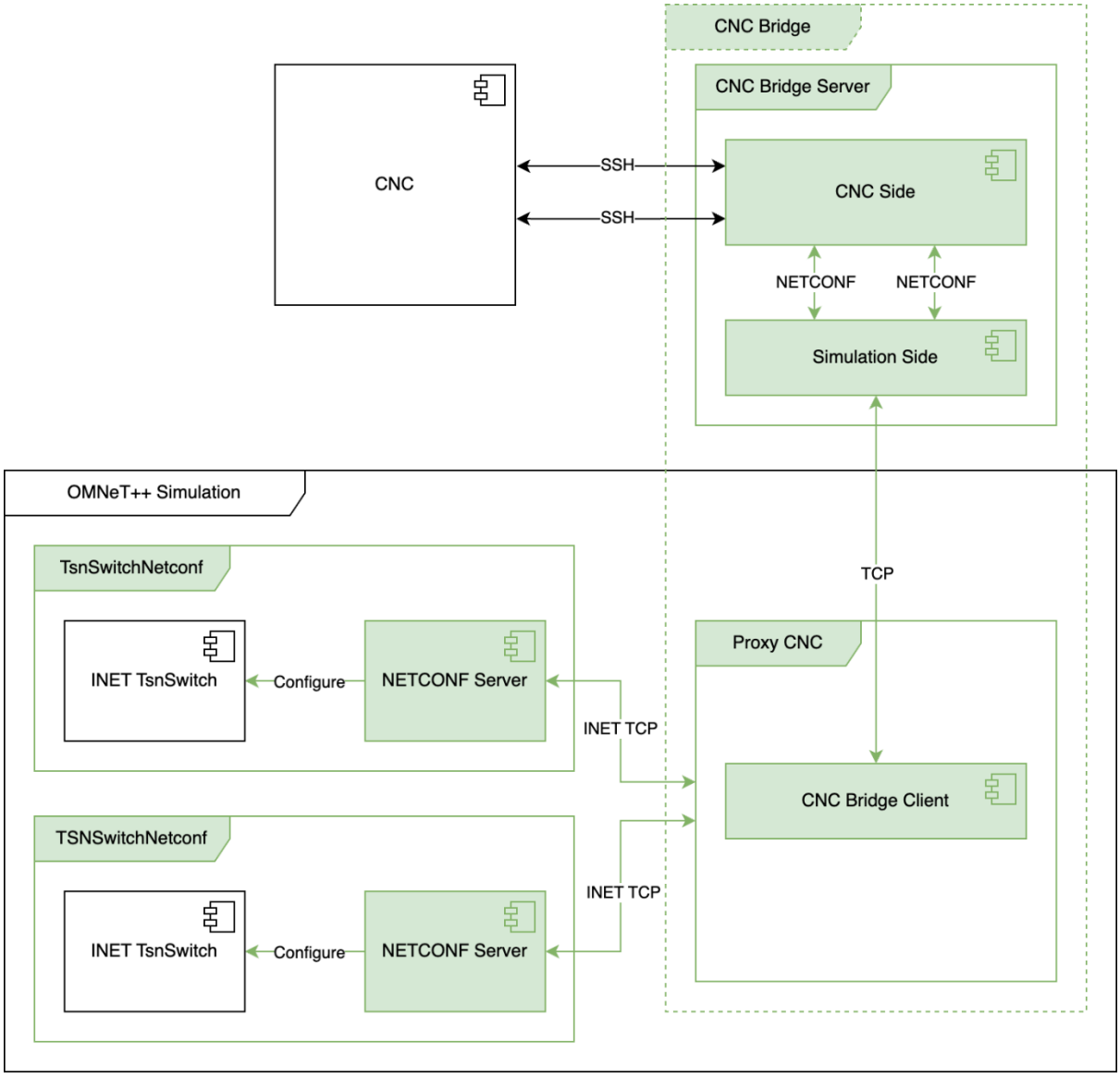


Figure 12: Architecture of the INET NETCONF interface.

Figure 12 presents an architectural overview of our NETCONF interface. The key-component of the interface between the real-world and the simulation is the *CNC Bridge*. This bridge component is responsible to "convert" NETCONF messages from the real CNC to the simulation and vice versa. To this end, the *CNC Bridge Server* provides an SSH endpoint on the same IP address as the NETCONF

Servers inside the simulation. This requires the user to bind these IP addresses to a loopback interface assuming that the CNC is executed on the same host as the simulator. A detailed description of this can be found in the documentation in the repository[10]. The *CNC Side* of the *CNC Bridge Server* also terminates the SSH session and extracts the received NETCONF messages from the *CNC* and forwards them to the *Simulation Side* of the *CNC Bridge Server*. In the other direction from the simulator to the CNC, when a NETCONF message is received from the *Simulation Side*, the *CNC Side* sends this NETONF message through the SSH connection to the *CNC*.

The *Proxy CNC* within the simulation is implemented as an OMNeT++ component and from the view of the simulation it is the CNC. The *CNC Bridge Client* is connected to the *Simulation Side* of the *CNC Bridge Server* through a TCP socket. NETCONF messages received from the *CNC Bridge Server* are encoded into INET frames and transmitted as TCP frames within the simulation. NETCONF messages received from a *NETCONF Server* by the *Proxy CNC* within the simulation are sent through the socket to the *CNC Bridge Server*.

In order to allow to read and edit the configuration of a *TsnSwitch* in INET, we introduce a new *TsnSwitchNetconf* module. This module contains a typical INET *TsnSwitch*, but additionally contains a *Netconf Server* responsible of handling incoming NETCONF messages and performing the intended actions based on them. The *Netconf Server* is based on existing libraries for NETCONF, such as Libnetconf2 [Lib25] and Sysrepo [Sys25].

Use-Cases

As already mentioned, a full NETCONF Server implementation is out-of-scope of this project due to its complexity. Therefore, we restrict our implementation to support two different exemplary use cases, which are of major importance for the validation in the Deterministic6G project. One use case is editing and querying the Time-Aware Shaper (TAS) configuration of a *TsnSwitchNetconf,* and the other use case is gathering information about connected neighbors using LLDP.

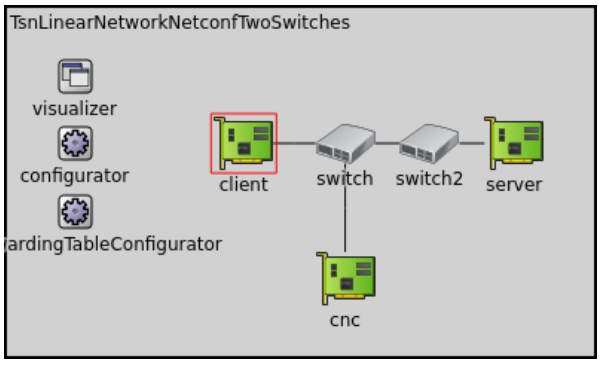


Figure 13: Network of the NETCONF showcase.

Figure 13 shows a simple network, which we use in the following to showcase the two use cases. In our first use case, we start the simulation with the initial TAS configuration shown in Listing 8.

[10] https://github.com/DETERMINISTIC6G/netconf-for-inet/blob/master/cnc-bridge-server/README.md

```
**.switch...transmissionGate[0].offset = 0ms
**.switch...transmissionGate[0].durations = [4ms, 6ms]
**.switch...transmissionGate[1].offset = 6ms
**.switch...transmissionGate[1].durations = [2ms, 8ms]
```

Listing 8: Initial TAS configuration.

We then perform a NETCONF get request to query for the current TAS configuration. The output is shown in Listing 9. As can be seen, the YANG module contains a different format than the simulation configuration due to the way NETCONF and INET specify the configuration of GCLs. Our implementation performs the necessary conversion. In the YANG module, each entry corresponding to a gate state of all gates and a duration. In the first entry, only gate 0 is open (gate state $00000001_2 = 1_{10}$) for $4\ ms$. After that, gate 1 is open for $2\ ms$ (gate state $00000010_2 = 2_{10}$). Lastly, all gates are closed for another $4\ ms$. This matches our initial simulation configuration.

We then perform an edit-config request (shown in Listing 10) to modify the GCL of the switch. By setting the *admin-base-time* parameter, we can define the time when the configuration should be applied by the switch. In our example we set this value to $67\ ms$. The new GCL configuration has a period of $15\ ms$ of gate 1 followed by a $2\ ms$ period with both gates closed, and finally a *13 ms* period with only gate 0 open.

Figure 14 shows the results of this scenario. The period after reconfiguration is highlighted in green from $67\ ms$ on. The gate states of gate 0 and gate 1 are shown on blue and brown, respectively. As can be seen, the re-configuration of the gates works as expected.

To showcase the LLDP implementation for retrieving the network topology (an essential information to perform stream routing tasks by the CNC), we launch another simulation and request LLDP information from the *switch* module. The LLDP response is shown in Listing 11. As can be seen, the switch module correctly returns the (simulated) MAC addresses of the connected devices together with their name and for *switch2* additionally returns the IP address of its NETCONF server.

Discussion

Our two showcases above show that it is feasible to extend the OMNeT++ simulator with a NETCONF interface to get and edit the config of TsnSwitches in INET. However, this approach also has a few downsides.

First, OMNeT++ and INET were not developed with NETCONF or YANG in mind. Thus, the way of configuring modules in the simulation is inherently different from NETCONF. To this end, providing a fully-fledged NETCONF interface for INET requires to add a whole new interface of configuration for every supported YANG module. Additionally, INET does not support the concept of different datastores (pending, running, …) by default. By adding these datastores to the simulation, for every YANG module, consistency between the actual simulation configuration and the datastores needs to be ensured, introducing new room for error and additional development overhead.

Secondly, the simulation time and real-world time are not only different in their absolute value, but additionally the simulation time does not necessarily proceed in the same speed as the real-world time. While OMNeT++ allows to limit the simulation speed to the real-world time (i.e. the time in the simulation never runs faster than the real-world time), bigger simulations typically run slower than the real-world time due to their complexity. As especially in time-critical applications a correct sense

of time is required by the CNC, providing a fully transparent interface between the CNC and the simulation is complex and requires sophisticated synchronization concepts.

Lastly, the setup complexity of the interface is quite high, as it requires the setup of virtual network interfaces and an additional simulation-external module to be started along with the simulation to allow for a communication between a CNC and the simulation.

Therefore, a full-fledged implementation of a NETCONF interface for the simulator is beyond the scope of this project. The given implementation can be considered a first step towards such an implementation. However, in order to reduce complexity and focus on the validation targets of the project, we next propose an alternative interface, which can be integrated with INET more easily.

```
<gate-parameter-table xmlns="urn:ieee:std:802.1Q:yang:ieee802-dot1q-sched-bridge">
    <gate-enabled>true</gate-enabled>
    <admin-gate-states>1</admin-gate-states>
    <oper-gate-states>1</oper-gate-states>
    <admin-control-list>
        <gate-control-entry>
            <index>0</index>
            <operation-name xmlns:sched="urn:ieee:std:802.1Q:yang:ieee802-dot1q-
          sched">sched:set-gate-states
            </operation-name>
            <time-interval-value>4000000</time-interval-value>
            <gate-states-value>1</gate-states-value>
        </gate-control-entry>
        <gate-control-entry>
            <index>1</index>
            <operation-name xmlns:sched="urn:ieee:std:802.1Q:yang:ieee802-dot1q-
          sched">sched:set-gate-states
            </operation-name>
            <time-interval-value>2000000</time-interval-value>
            <gate-states-value>2</gate-states-value>
        </gate-control-entry>
        <gate-control-entry>
            <index>2</index>
            <operation-name xmlns:sched="urn:ieee:std:802.1Q:yang:ieee802-dot1q-
          sched">sched:set-gate-states
            </operation-name>
            <time-interval-value>4000000</time-interval-value>
            <gate-states-value>0</gate-states-value>
        </gate-control-entry>
    </admin-control-list>
    <admin-cycle-time>
        <numerator>10000000</numerator><denominator>1000000000</denominator>
    </admin-cycle-time>
    <oper-cycle-time>
        <numerator>10000000</numerator><denominator>1000000000</denominator>
    </oper-cycle-time>
    <admin-base-time>
        <seconds>0</seconds><nanoseconds>0</nanoseconds>
    </admin-base-time>
    <oper-base-time>
        <seconds>0</seconds><nanoseconds>0</nanoseconds>
    </oper-base-time>
    <config-change>false</config-change>
    <config-change-time>
        <seconds>0</seconds><nanoseconds>0</nanoseconds>
    </config-change-time>
    <current-time>
        <seconds>0</seconds><nanoseconds>10028850</nanoseconds>
    </current-time>
    <config-pending>false</config-pending>
    <config-change-error>0</config-change-error>
    <supported-list-max>4294967295</supported-list-max>
    <supported-cycle-max>
        <numerator>4294967295</numerator><denominator>1</denominator>
    </supported-cycle-max>
    <supported-interval-max>4294967295</supported-interval-max>
</gate-parameter-table>
```

Listing 9: Examplary get-config response.

```
<interfaces
        xmlns="urn:ietf:params:xml:ns:yang:ietf-interfaces">
    <interface>
        <name>eth1</name>
        <type xmlns:ianaif="urn:ietf:params:xml:ns:yang:iana-if-
             type">ianaif:ethernetCsmacd</type>
        <bridge-port
                xmlns="urn:ieee:std:802.1Q:yang:ieee802-dot1q-bridge">
            <gate-parameter-table
                    xmlns="urn:ieee:std:802.1Q:yang:ieee802-dot1q-sched-bridge">
                <config-change>true</config-change>
                <gate-enabled>true</gate-enabled>
                <admin-base-time>
                    <seconds>0</seconds>
                    <nanoseconds>67000000</nanoseconds>
                </admin-base-time>
                <admin-cycle-time>
                    <numerator>30000000</numerator>
                    <denominator>1000000000</denominator>
                </admin-cycle-time>
                <admin-gate-states>2</admin-gate-states>
                <admin-control-list
                        xmlns:xc="urn:ietf:params:xml:ns:netconf:base:1.0"
                        xc:operation="replace">
                    <gate-control-entry>
                        <index>0</index>
                        <operation-name
                                xmlns:sched="urn:ieee:std:802.1Q:yang:ieee802-dot1q-
            sched">sched:set-gate-states</operation-name>
                        <time-interval-value>15000000</time-interval-value>
                        <gate-states-value>2</gate-states-value>
                    </gate-control-entry>
                    <gate-control-entry>
                        <index>1</index>
                        <operation-name
                                xmlns:sched="urn:ieee:std:802.1Q:yang:ieee802-dot1q-
            sched">sched:set-gate-states</operation-name>
                        <time-interval-value>2000000</time-interval-value>
                        <gate-states-value>0</gate-states-value>
                    </gate-control-entry>
                    <gate-control-entry>
                        <index>2</index>
                        <operation-name
                                xmlns:sched="urn:ieee:std:802.1Q:yang:ieee802-dot1q-
            sched">sched:set-gate-states</operation-name>
                        <time-interval-value>13000000</time-interval-value>
                        <gate-states-value>1</gate-states-value>
                    </gate-control-entry>
                </admin-control-list>
            </gate-parameter-table>
        </bridge-port>
    </interface>
</interfaces>
```

Listing 10: Exemplary edit-config to modify the gates of switch.

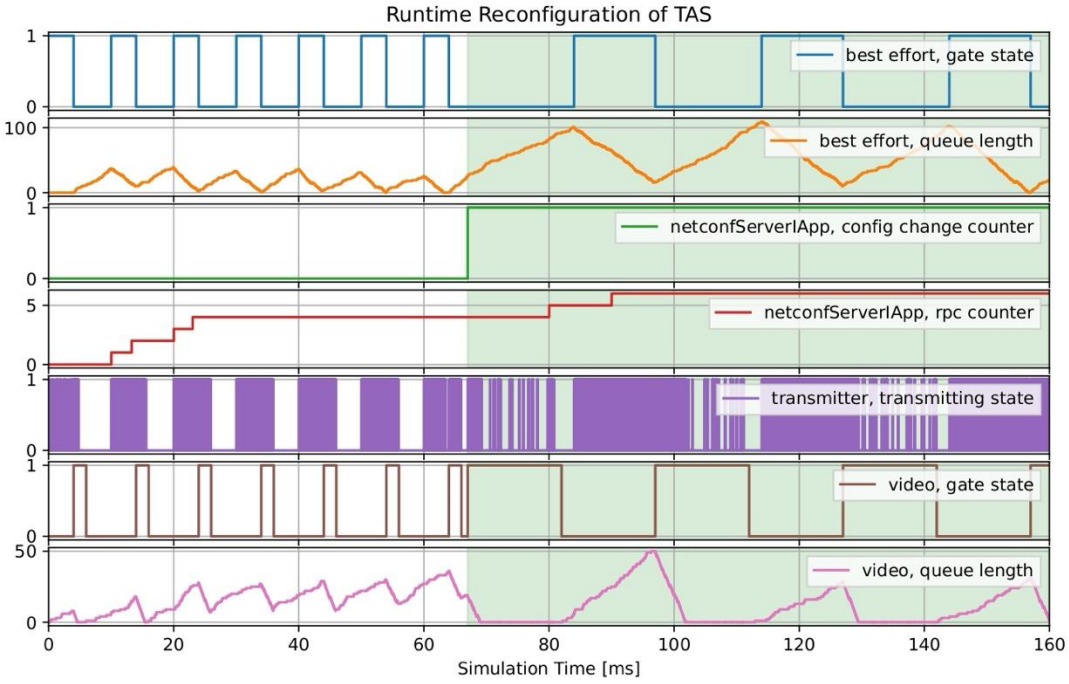


Figure 14: Reconfiguration of TAS during simulation using NETCONF.

```
<lldp xmlns="urn:ieee:std:802.1AB:yang:ieee802-dot1ab-lldp">
    <port>
        <name>eth0</name>
        <dest-mac-address>0A-AA-00-00-00-02</dest-mac-address>
        <port-id>100</port-id>
        <remote-systems-data>
            <!-- omitted some less important elements -->
            <system-name>TsnLinearNetworkNetconfTwoSwitches.client</system-name>
            <system-description>TsnDevice</system-description>
        </remote-systems-data>
    </port>
    <port>
        <name>eth1</name>
        <dest-mac-address>0A-AA-00-00-00-03</dest-mac-address>
        <port-id>101</port-id>
        <remote-systems-data>
            <time-mark>0</time-mark>
            <remote-index>101</remote-index>
            <chassis-id-subtype>mac-address</chassis-id-subtype>
            <chassis-id>0A-AA-00-00-00-07</chassis-id>
            <system-name>TsnLinearNetworkNetconfTwoSwitches.switch2.switch</system-name>
            <system-description>TsnSwitchManaged</system-description>
            <management-address>
                <address-subtype xmlns:rt="urn:ietf:params:xml:ns:yang:ietf-
              routing">rt:ipv4</address-subtype>
                <address>010A0B0C02</address>
            </management-address>
        </remote-systems-data>
    </port>
    <port>
        <name>eth2</name>
        <dest-mac-address>0A-AA-00-00-00-04</dest-mac-address>
        <port-id>102</port-id>
        <remote-systems-data>
            <!-- omitted some less important elements -->
            <system-name>TsnLinearNetworkNetconfTwoSwitches.cnc</system-name>
            <system-description>CentralizedNetworkController</system-description>
        </remote-systems-data>
    </port>
    <port>
        <name>eth3</name>
        <dest-mac-address>0A-AA-00-00-00-05</dest-mac-address>
        <port-id>103</port-id>
        <remote-systems-data>
            <!-- omitted some less important elements -->
            <system-name>TsnLinearNetworkNetconfTwoSwitches.switch.netconfServer</system-
              name>
            <system-description>NetconfServer</system-description>
        </remote-systems-data>
    </port>
</lldp>
```

Listing 11: LLDP response for example network.

2.5.2 Direct Scheduler Interface

Our second approach aims to overcome the issues of the first approach, by providing a direct and dynamic scheduler interface to INET. We base our implementation on an already existing INET implementation to configure schedules at startup[11] and extend it with dynamic scheduler calls at simulation time and with features required to schedule streams in converged 6G/TSN networks. We first explain our newly introduced modules, followed by an example showcase.

Dynamic Scheduling Modules

Our dynamic scheduling implementation consists of three main modules, the *DynamicPacketSource*, *ChangeMonitor,* and the *ExternalGateScheduleConfigurator*, which we describe in the following.

The *DynamicPacketSource*[12] module extends INETs *ActivePacketSource*[13], which is typically part of an *UdpApp*[14] in INET and can be used to generate UDP packets. Our extension of this module adds more parameters to allow for a dynamic reconfiguration of the sending behavior of an app. Our new parameters are described in detail in Table 2. In general, for most parameters there is an equivalent *pending* parameter which indicates that the app intends to change this parameter upon the confirmation of the scheduler. Default values marked with an "@" default to the defined value of the referenced parameter.

Parameter	Type	Default	Explanation
enabled	bool	$true$	Defines whether the app is currently enabled (generating UDP packets) or not.
pendingEnabled	bool	@enabled	Changing this parameter calls the scheduler and afterwards updates the *enabled* parameter.
packetLength	byte	*none*	Defines the length of the generated packet (excluding headers added by following modules).
pendingPacketLength	byte	@packet Length	Changing this parameter calls the scheduler and afterwards updates the *packetLength* parameter.
productionInterval	seconds	$-1s$ (once at startup)	Defines the interval at which packets are generated.
pendingProductionInterval	seconds	@production Interval	Changing this parameter calls the scheduler and afterwards updates the *productionInterval* parameter.
pcp	int	0	Defines the PCP value of the stream.
reliability	double	1.0	Defines the required reliability of the stream.

[11] https://inet.omnetpp.org/docs/showcases/tsn/gatescheduling/index.html

[12] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.apps.dynamicsource.DynamicPacketSource.html

[13] https://doc.omnetpp.org/inet/api-current/neddoc/inet.queueing.source.ActivePacketSource.html

[14] https://doc.omnetpp.org/inet/api-4.4.0/neddoc/inet.applications.udpapp.UdpApp.html

maxLatency	ms	@productionInterval	Defines the maximum allowed end-to-end delay of this stream.
maxJitter	ms	$0.05 *$ @productionInterval	Defines the maximum allowed jitter of this stream.
productionOffsets	seconds[]	$[0s]$	Defines an array of offsets relative to the productionInterval. This is typically set by the scheduler to configure different offsets of multiple cycles within an hypercycle.

Table 2: Configuration parameters of the DynamicPacketSource module.

The parameters of the *DynamicPacketSource* module can be changed using INETs ScenarioManager[15], which allows to make changes at defined points in simulation time. The same applies to the *delayUplink* and *delayDownlink* paramaters of TTs inside of the DetCom node (see Section 2.1). The *ChangeMonitor*[16] is responsible for tracking the changes, mainly the changes of delay distributions and parameters of apps using the *DynamicPacketSource* module. The *ChangeMonitor* is also responsible of gathering all important information for the scheduler, such as histograms. In case the histogram is not provided using our *Histogram* module (see Section 2.1), it samples the defined delay distribution. The important parameters of this module are presented in Table 3.

Parameter	**Type**	**Default**	**Explanation**
schedulerCallDelay	seconds	$0s$	After detecting a change, the ChangeMonitor wait for the specified time before performing a call to the scheduler. This can be used to gather multiple simultaneous changes before calling the scheduler.
numberOfSamples	int	$1{,}000{,}000$	Defines the number of samples to generate a histogram, if the delay distribution is not defined using our *Histogram* module.

Table 3: Configuration parameters of the ChangeMonitor module.

After detecting the changes and optionally waiting for the defined call delay, the *ChangeMonitor* calls the *ExternalGateScheduleConfigurator*[17]. This module is responsible for calling a configured scheduler and afterwards configuring the GCLs of all TSN devices as well as the production offsets and enabled state of all *DynamicPacketSource* apps. The configuration parameters are provided in Table 4. All file paths are relative to the *SCHEDULER_ROOT* environment variable, which needs to be defined before executing the simulation. When calling the scheduler, our simulation also transmits the current simulation time to the scheduler and requires the scheduler to respond with a *commitTime* in its result

[15] https://doc.omnetpp.org/inet/api-current/neddoc/inet.common.scenario.ScenarioManager.html

[16] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.dynamicscenario.ChangeMonitor.html

[17] https://deterministic6g.github.io/6GDetCom_Simulator/doc/neddoc/d6g.dynamicscenario.ExternalGateScheduleConfigurator.html

file. This commit time represents a simulation timestamp, when the returned configuration should be applied to the TSN devices. It is the scheduler's responsibility to ensure that this commit time does not break the existing schedule, e.g., by frames of the old schedule still being in transmission.

Parameter	Type	Explanation
command	string	Defines the command that is called to execute the scheduler. The first `%s` is replaced with the networkFile parameter, the second `%s` is replaced with the streamsFile parameter, etc. Example: `./scheduler.py --network %s --streams %s --dist %s --out %s`
networkFile	string	Defines the file path where the file defining the network topology is stored.
streamsFile	string	Defines the file path where the file defining the stream requirements is stored.
histogramsFile	string	Defines the file path where a file containing all delay distributions is stored.
configurationFile	string	Defines the file path where the scheduler outputs its calculated TSN schedule.

Table 4: Configuration parameters of the ExternalGateSchedulingConfigurator module.

Showcase

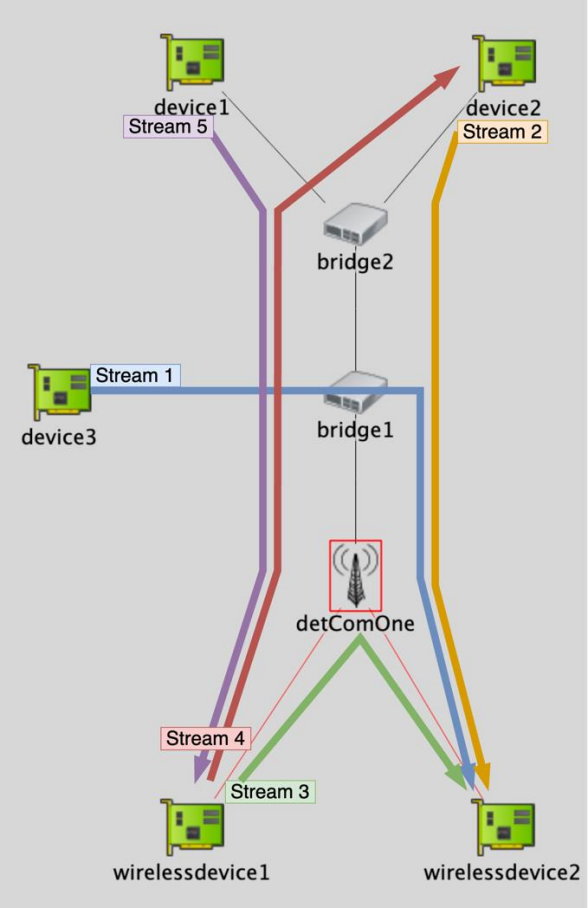


Figure 15: Network of the dynamic scenario showcase.

To showcase our dynamic scheduler interface, we set up a scenario containing different types of dynamic behavior. The network of this showcase is depicted in Figure 15. Device *wirelessdevice1* is connected to the DetCom node via dstt[0] and *wirelessdevice2* is connected to the DetCom node via dstt[1]. The network consists of five streams with the configuration provided in Table 5. Additionally, our simulation holds multiple histograms, as described in Table 6, with the initial configuration as provided in Listing 12.

stream	source	dest.	enabled	packetLength	productionInterval
1 (blue)	device3	wirelessdevice2	$false$	$1000\ B$	$30\ ms$
2 (orange)	device2	wirelessdevice2	$true$	$1000\ B$	$50\ ms$
3 (green)	wirelessdevice1	wirelessdevice2	$true$	$1000\ B$	$50\ ms$
4 (red)	wirelessdevice1	device2	$true$	$1000\ B$	$30\ ms$
5 (purple)	device1	wirelessdevice1	$true$	$1000\ B$	$30\ ms$

Table 5: Initial stream configuration.

histogram	description
Uplink	Uplink histogram of the PD-Wireless-5G-1 dataset [HDM+23] (See Figure 3a).
Uplink_improve	Uplink histogram shifted $1\ ms$ to the left.
Uplink_worse	Uplink histogram shifted $1\ ms$ to the right.
Downlink	Downlink histogram of the PD-Wireless-5G-1 dataset [HDM+23] (See Figure 3b).
Downlink_improve	Downlink histogram shifted $1\ ms$ to the left.
Downlink_worse	Downlink histogram shifted $1\ ms$ to the right.

Table 6: Available histograms.

```
**.dstt[0].delayDownlink = rngProvider("histogramContainer","Downlink_worse")
**.dstt[1].delayDownlink = rngProvider("histogramContainer","Downlink")
**.dstt[*].delayUplink = rngProvider("histogramContainer","Uplink")
```

Listing 12: Initial histogram configuration.

The dynamics of our scenario is defined using INETs *ScenarioManager* with the configuration as provided in Listing 13. The resulting end-to-end delays of this configuration are shown in Figure 16:

1. At $t = 20\ s$, *stream2* requests to stop its stream. After the execution of the scheduler, the stream is stopped, which is visible in the results by the ending orange line. At the same time the scheduler re-schedules *stream3* resulting in a change of its end-to-end delay.
2. At $t = 40\ s$, *stream3* wants to increase the production interval and packet size. This again leads to a re-scheduling of *stream3* resulting in the change of its end-to-end delay.
3. At $t = 50\ s$, the downlink histogram of *dstt1* improves (shifting $1\ ms$ to the left), while the uplink histogram of *dstt0* degrades (shifting $1\ ms$ to the right) resulting in rescheduling of *stream3* and *stream4*.
4. At $t = 60\ s$, *stream1* requests to start sending new data, while at the same time *stream3* immediately stops sending data. This becomes visible in the results by the ending green line and the newly starting blue line.

5. At $t = 80\ s$, the downlink histogram of *dstt0* improves (shifting $1\ ms$ to the right), resulting in a reduced end-to-end delay for *stream5.*

Discussion

Our direct scheduler interface for the 6GDetCom Simulator allows for the simulation of dynamic scenarios and a direct evaluation of the effects of adapted schedules. As the only perquisite to connect a scheduler to our framework is the conformity with our predefined file format, this approach is flexible and easy to set up.

However, the current implementation only provides basic features to update the GCL of devices. Further functionality, such as configuring PSFP, might be added in the future, e.g., if necessary for further evaluation in the upcoming deliverable D4.5 “Validation for DETERMINISTIC6G concepts”.

```
<!--stop stream2 -->
<at t="20">
    <set-param module="device2.app[0].source"
               par="pendingEnabled" value="false"/>
</at>

<!-- Change interval and packet length of stream3 -->
<at t="40">
    <set-param module="wirelessdevice1.app[2].source"
               par="pendingProductionInterval" value="300ms"/>
    <set-param module="wirelessdevice1.app[2].source"
               par="maxLatency" value="300ms"/>
    <set-param module="wirelessdevice1.app[2].source"
               par="pendingPacketLength" value="1200B"/>
</at>


<!-- downlink delay of dstt1 improves -->
<set-param t="50" module="detComOne.dstt[1]"
           par="delayDownlink"
           expr="rngProvider(&quot;histogramContainer&quot;,&quot;Downlink_improve&quot;)"
/>

<!-- uplink delay of dstt0 degrades -->
<set-param t="50" module="detComOne.dstt[0]"
           par="delayUplink"
           expr="rngProvider(&quot;histogramContainer&quot;,&quot;Uplink_worse&quot;)"/>

<!-- start stream1 and stop stream3 -->
<at t="60">
    <set-param module="device3.app[0].source"
               par="pendingEnabled" value="true"/>
    <set-param module="wirelessdevice1.app[2].source"
               par="enabled" value="false"/>
</at>

<!-- downlink delay of dstt0 improves -->
<set-param t="80" module="detComOne.dstt[0]"
           par="delayDownlink"
           expr="rngProvider(&quot;histogramContainer&quot;,&quot;Downlink&quot;)"/>
```

Listing 13: Dynamic scenario configuration.

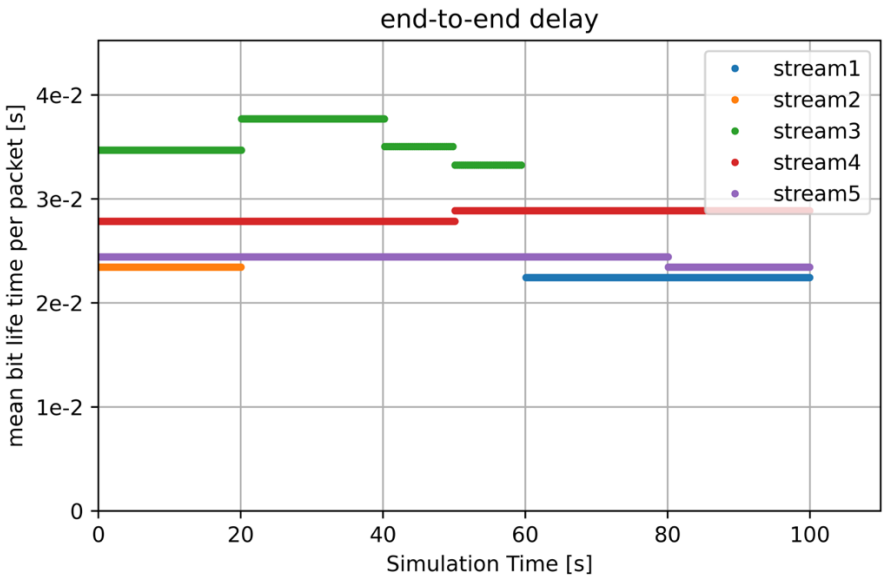


Figure 16: Resulting end-to-end delays in the dynamic scenario.

3 6GDetCom Emulator

While our simulator framework described above allows for the simulation and analysis of novel 6G features, it does not allow to analyze the behavior of *real* applications in converged 6G/TSN networks. Providing realistic simulation models for applications is often very complex and unattractive if the real application already exists. To facilitate the testing and evaluation of real applications under 5G/6G network conditions, we propose the novel 6GDetCom Emulator, which allows to emulate the characteristic PD of virtual 6G bridges in a real network, without the need of dedicated 5G/6G hardware. In the Deterministic6G project, we use this framework in particular to validate the exoskeleton applications. Results of this validation will be available in the upcoming Deliverable D4.5.

3.1 Architecture of the Network Delay Emulator

The core of the emulator is a Linux Queueing Discipline (QDisc) called sch_delay that can be assigned to network interfaces to add artificial delay to all packets leaving through this network interface.

Figure 17 shows the system architecture consisting of two major parts: the QDisc running in the kernel space, and a user-space application providing individual delays for each transmitted packet through a character device. The provided delays are buffered in the QDisc, such that delay values are available immediately when new packets arrive. Whenever a packet is to be transmitted through the network interface, the next delay value is dequeued and applied to the packet before passing it on to the network interface (TX queue).

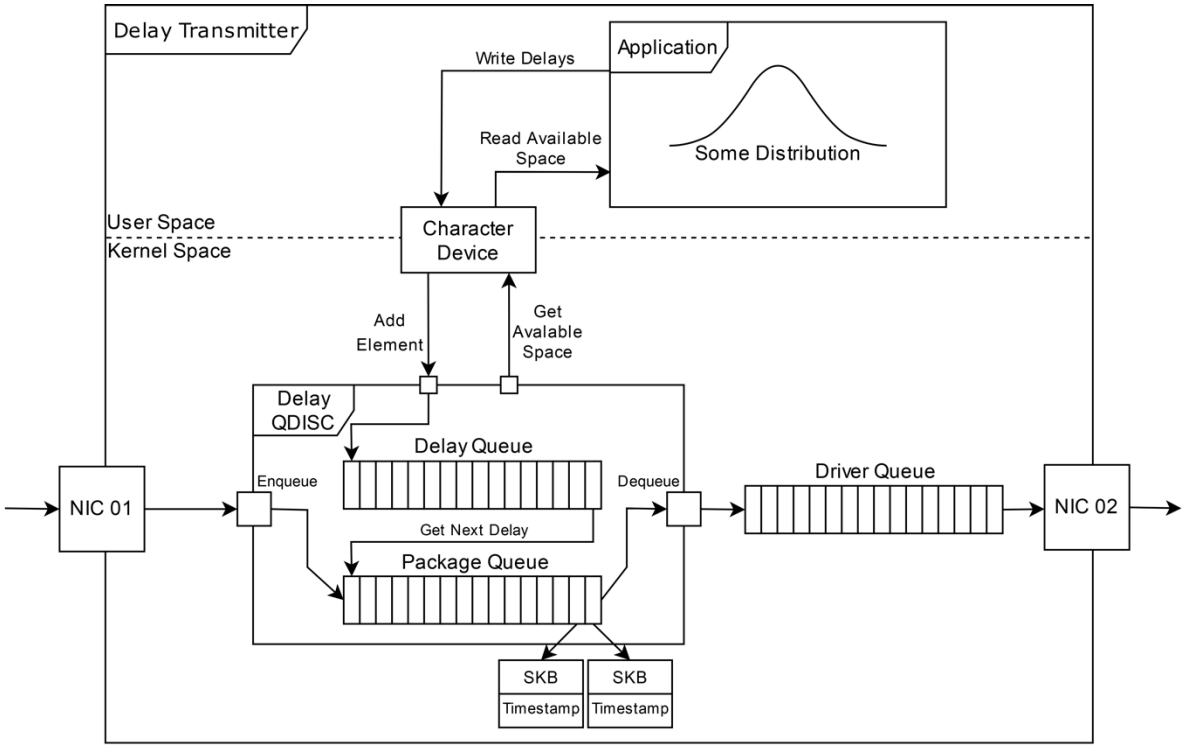


Figure 17: Architecture of the Network Delay Emulator

Providing delays through a user-space application allows for a flexible and convenient definition of delays without touching any kernel code. The project contains a sample user-space application implemented in Python to define delays as constant values, normal distributions (probability density

function), or histograms (as can be derived from our measurements in our 5G/6G testbed). This application can be easily extended to calculate other delay distributions.

The QDisc can also be applied to network interfaces that are assigned to a virtual bridge to apply individual delay distributions to packets forwarded through different egress interfaces. This allows for emulating the end-to-end network delay of a whole emulated network with a single Linux machine.

One limitation of this approach based on pre-calculating and buffering delays is that it is restricted to independent and identically distributed (i.i.d.) delays. If the delay of a specific packet depends on the delay of an earlier packet, this cannot be easily modelled by this approach since delays were already calculated and buffered possibly long before the packet to be delayed actually arrives. Also changing the delay distribution at runtime is not easily possible due to the buffering of delays from the old distribution.

QDiscs are typically configured with the tc (traffic control) command in Linux. Since the sch_delay QDisc requires specific parameters as shown next, a modified version of tc is required (a patch for tc is also included in the GitHub repository of the Network Delay Emulator). The following parameters are available to configure the QDisc:

Option	Type	Default	Explanation
limit	int	1000	The size of the internal queue for buffering delayed packets. If this queue overflows, packets will get dropped. For instance, if packets are delayed by a constant value of $10\ ms$ and arrive at a rate of $1000\frac{pkt}{s}$, then a queue of at least $1000\frac{pkt}{s} * \ 10 * (10^{-3}s) \ = \ 10\ pkt$ would be required. A warning will be posted to the kernel log if messages are dropped.
reorder	bool	$true$	Whether packet reordering is allowed to closely follow the given delay values, or keep packet order as received. If packet reordering is allowed, a packet with a smaller random delay might overtake an earlier packet with a larger random delay in the QDisc. If packet re-ordering is not allowed, additional delay might be added to the given delay values to avoid packet re-ordering.

Table 7: Configuration Parameters

A detailed tutorial on how to use and setup the 6GDetCom Emulator can be found in the GitHub repository[18], as well as in the DETERMINISITC6G blog post[19] dedicated to the usage of the network delay emulator.

3.2 Evaluation of Delay Emulation Accuracy

To give an impression on the accuracy to be expected with the 6GDetCom Emulator, we performed measurements with the following virtual bridge setup as shown in Figure 18. We used network taps in fiber optic cables (red dots) and an FPGA network measurement card from Napatech (NT40E3-4-PTP) to capture the traffic from H1 (sender) to H2 (receiver) with nano-second precision.

[18] https://github.com/DETERMINISTIC6G/6GDetCom_Emulator
[19] https://blog.deterministic6g.eu/posts/2024/10/26/network_delay_emulator.html

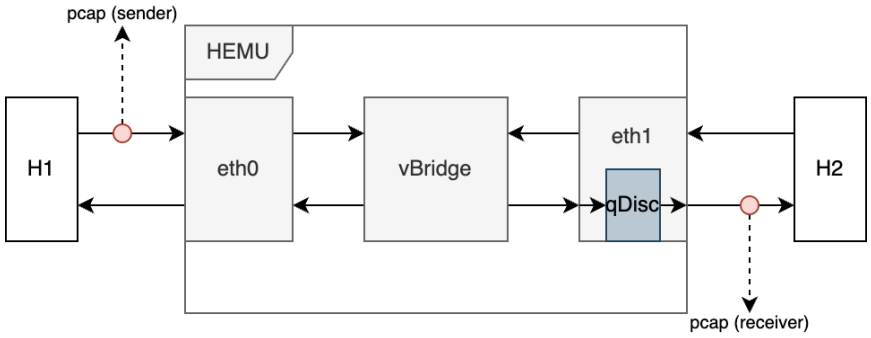


Figure 18: Evaluation setup.

The sender app on H1 sends minimum-size (64B) UDP packets at a rate of 100 pkt/s and at a speed of 10 Gbps to the receiver app on H2. The QDisc is configure with a normal distribution with mean = 10 ms and stddev = 1 ms.

As baseline, we also capture a trace with zero delay emulation (w/o QDisc on eth1). Traces were captured for about 10 min.

The specs of the Hemu host are:

- Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz
- 16 GB RAM

Figure 19 shows the histogram of the measured (emulated) end-to-end delay overlayed with the input histogram. We see that the emulated delay closely follows the input with a small offset, which could already be considered (subtracted) in the delay value generation.

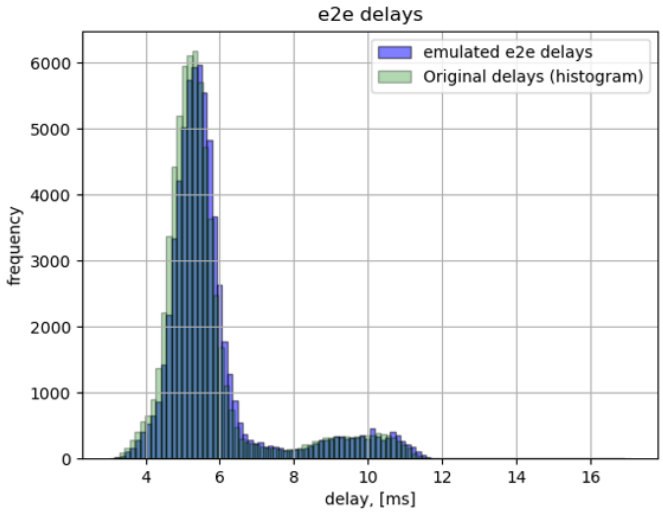
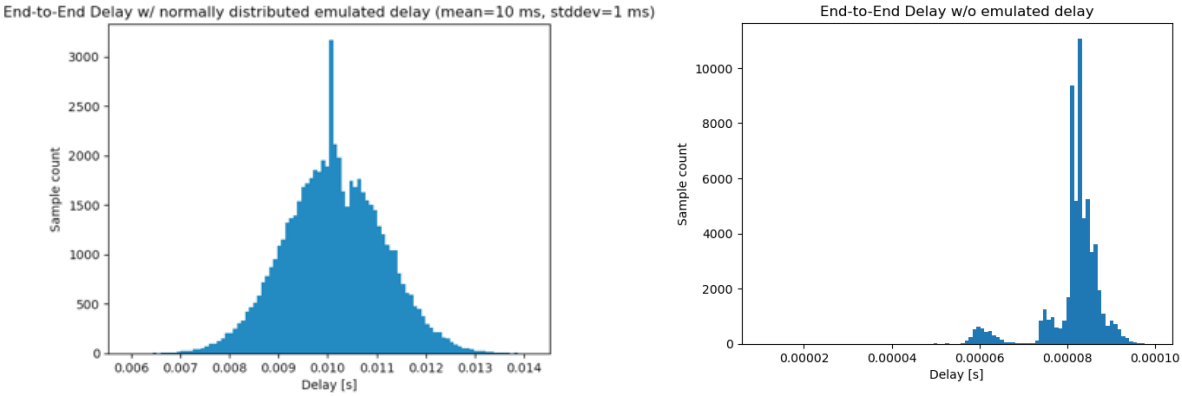


Figure 19: Comparison of emulated end-to-end delay and input histogram.

Figure 20 shows the histograms of the actual delay between the measurement points. Theoretically, we would need to subtract:

1. One transmission delay ($(64 * 8\,bit) / 10\,Gbps = \mathbf{51.2\,ns}$) since Hemu will only start processing the packet when it has been fully received, and the measurement card takes the timestamp at the header.
2. The propagation delay for about $6\,m$ fiber cable (about $6\,m / (\frac{2}{3} * 3 * 10^8 \frac{m}{s}) = \mathbf{30\,ns}$) connecting the tap to the measurement card.

However, since this delay is only in the range of microseconds or below, we report values as measured.



(a) Normal distribution

(b) No emulated delay

Figure 20: Accuracy analysis of emulated delays

For the normal distribution, the following values were measured:

- mean = $10.126314\,ms$
- stddev = $0.996269\,ms$
- 99 % confidence interval of the mean = $[0.010115941\,s, 0.010136687\,s]$
- min = $5.947590\,ms$
- max = $14.116764\,ms$

Without delay emulation, the delay was:

- mean = $81.662\,\mu s$
- 99 % confidence interval of the mean = $[81.593\,\mu s, 81.730\,\mu s]$
- min = $10.252\,\mu s$
- max = $99.421\,\mu s$

We see that the delay added without any emulated delay is around 81 us. This could be considered as an offset when creating delay distributions.

An analysis of a real-world application will be available in the following deliverable D4.5.

4 Secure PTP Time Synchronization Emulation Framework

4.1 Introduction

In D3.2 “Report on the Security Solutions” [DET23-D32], we introduced a security-by-design approach that integrates high-precision telemetry with a programmable data plane to enhance security management in deterministic network applications. This section presents an emulation framework which implements this approach to secure End-to-End (E2E) time synchronization using PTP, which is further utilized in D2.4 “Report on the time synchronization for E2E time awareness” [DET25-D24].

Leveraging programmable data planes, our framework employs In-band Network Telemetry (INT) to embed monitoring data within PTP extension fields, eliminating the need for additional probe packets, and reducing network traffic overhead. A collector module captures PTP packets, extracts telemetry data, and facilitates real-time TDA detection and localization. To validate this approach, we emulate a time synchronization network using Mininet, where each end-host functions as an Ordinary Clock (OC) and the switch operates as a Transparent Clock (TC). The OC is implemented by LinuxPTP [CM+10], while the TC is developed using the P4 (Programming Protocol-Independent Packet Processors) programming language. This emulation framework represents a step toward integrating time synchronization processes into software-defined and programmable networking architectures. The following section details the technical implementation of these components.

4.2 Framework Description

4.2.1 Emulation Network

We use Mininet to emulate a PTP time synchronization network. Figure 21 represents a time synchronization network consisting of two OCs and three TCs. Each clock is running in an isolated network node, which is either an end-host or a switch.

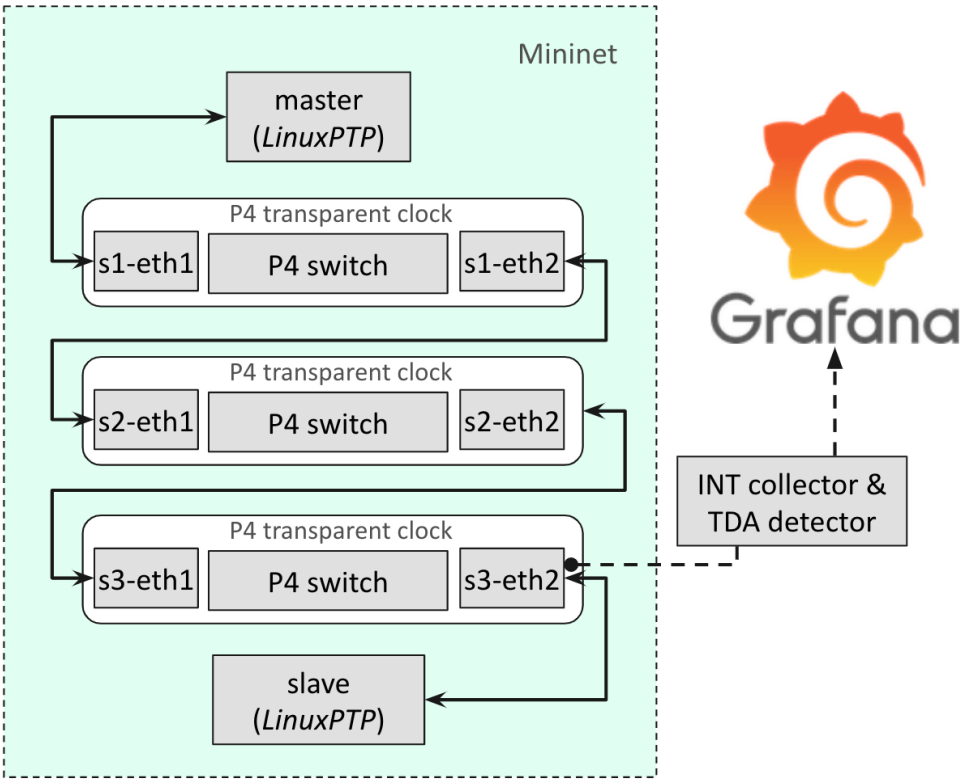


Figure 21: Security Emulation Framework of PTP Time Synchronization.

The emulation framework enables users to define the network topology and configure it through a JSON configuration file, streamlining the setup and deployment of the simulated environment. Listing 14 provides an example of this configuration, corresponding to the network illustrated in Figure 21. The configuration file consists of the following elements:

- hosts: list of end-hosts. Each host is configured by the following information:
  - ip: IP address of the host
  - mac: MAC address of the host
  - commands: list of Bash-based commands to be executed at startup
- switches: list of switches. Each switch is configured by the following information:
  - config: path to the file containing commands to configure the switch's IP routing table, INT.
  - commands: list of Bash-based commands to be executed at startup
  - override_ports: a table mapping host's NIC into the emulator's NIC
- link: connections between hosts and switches. A host can connect directly to another host or a port of a switch. A switch may have at least two ports.

```
{
  "hosts": {
    "h1": {
      "ip": "10.0.1.1/24", "mac": "08:00:00:00:01:11",
      "commands": [
        "route add default gw 10.0.1.10 dev eth0",
        "arp -i eth0 -s 10.0.1.10 08:00:00:00:01:00",
        "ptp4l -i eth0 -f ./configs/master.cfg -m"
      ]
    },
    "h2": {
      "ip": "10.0.2.2/24", "mac": "08:00:00:00:02:22",
      "commands": [
        "route add default gw 10.0.2.20 dev eth0",
        "arp -i eth0 -s 10.0.2.20 08:00:00:00:02:00",
        "ptp4l -i eth0 -f ./configs/slave.cfg -m"
      ]
    }
  },
  "switches": {
    "s1": {"config": "configs/s1.txt"},
    "s2": {"config": "configs/s2.txt"},
    "s3": {"config": "configs/s3.txt",
      "override_ports": {"2": "enp0s31f6"},
      "commands": ["python collector/collector.py --nic s3-eth2"]
    }
  },
  "links": [
    ["h1", "s1-p1"],
    ["s1-p2", "s2-p1"],
    ["s2-p2", "s3-p1"],
    ["s3-p2", "h2"]
  ]
}
```

Listing 14: Configuration File of the Emulator.

The emulation framework can be deployed on either a single machine or multiple machines. When executed on a single machine, it is necessary to configure the slave clock in free-running mode to prevent it from updating the system time. By default, each switch or host port is instantiated using a virtual network interface, meaning that clocks rely on software timestamps. However, the framework also supports binding a virtual port to a physical network interface, via the "override_ports" parameter, allowing clocks to utilize hardware timestamps provided by the underlying hardware. This significantly enhances time synchronization accuracy. Although this approach improves precision, it explicitly requires compatible hardware support to enable hardware timestamping.

4.2.2 P4-based Programmable Transparent Clock

The primary role of a transparent clock (TC) is to precisely measure residence time and adjust PTP messages accordingly to maintain accurate time synchronization across the network. Implementing this in P4 requires extending BMv2 to timestamp packets at both ingress and egress ports. These timestamps are used to calculate the residence time of the packet. The residence time is then carried in the correctionField of its associated packet. For example, the ingress and egress timestamps of a Sync (or Delay_Req) packet are captured and stored to later update the correctionField of its

associated Follow_Up (or Delay_Res) packet. This pairing of packets, such as Sync and Follow_Up, is identified using a 3-tuple (clockId, portId, sequenceId) to ensure proper correlation.

Listing 15 introduces four additional primitives that enable a P4 program to capture and retrieve ingress and egress timestamps of a PTP packet as it traverses a BMv2 virtual switch.

```
// capture ingress timestamps
extern void ptp_store_ingress_mac_tstamp(in bit<64> clockId, in bit<16> portId,
          in bit<16> seqId);
// retrieve ingress timestamps
extern void ptp_get_ingress_mac_tstamp(in bit<64> clockId, in bit<16> portId,
          in bit<16> seqId, out bit<64> rx_tstamp);

// capture egress timestamps
extern void ptp_capture_egress_mac_tstamp(in bit<64> clockId, in bit<16> portId,
          in bit<16> seqId);
// retrieve egress timestamps
extern void ptp_get_egress_mac_tstamp(in bit<64> clockId, in bit<16> portId,
          in bit<16> seqId, out bit<64> tx_tstamp);
```

Listing 15: Additional Primitives of BMv2.

Listing 16 presents a P4 skeleton that uses these primitives to implement a transparent clock. The implementation follows these steps:

- Capture ingress and egress timestamp of Sync and Delay_Req messages
- Upon detecting Follow_Up or Delay_Res message:
  - Retrieve ingress and egress timestamps of its corresponding message.
  - Update the telemetry information in TLV extension.
  - Adjust the correctionField value accordingly.
  - Update the PTP message length to account for the additional TLV.

```
if (hdr.ptp.isValid()) {
    // if we see a sync or delay_req message
    // Step 0: capture ingress and egress timestamps
    if (hdr.ptp.messageType == PTP_MSG_SYNC
          || hdr.ptp.messageType == PTP_MSG_DELAY_REQUEST) {
        // remember its arrival time
        ptp_store_ingress_mac_tstamp(hdr.ptp.clockId, hdr.ptp.portId, hdr.ptp.sequenceId);
        // require capturing and store its departure time
        ptp_capture_egress_mac_tstamp(hdr.ptp.clockId, hdr.ptp.portId, hdr.ptp.sequenceId);
    }
    // we see a follow_up or delay_response message
    if (hdr.ptp.messageType == PTP_MSG_FOLLOW_UP
          || hdr.ptp.messageType == PTP_MSG_DELAY_RESPONSE) {
        // Step 1: get ingress and egress timestamps of the corresponding packet
        // by default, we use the clockId and portId of the actual packet to correlate
        clockId = hdr.ptp.clockId;
        portId  = hdr.ptp.portId;

        // in case of delay_res message, the master will report clockId and
        //     portId of delay_req message at the end of Delay_res message.
        // => delay_res message contains 2 clockId values: one belonged to master,
        //     another (at the end of msg) belonged to the slave who requested
        if (hdr.ptp.messageType == PTP_MSG_DELAY_RESPONSE) {
            clockId = hdr.ptp_res.requestClockId;
            portId   = hdr.ptp_res.requestPortId;
        }
        // get timestamps
        ptp_get_ingress_mac_tstamp(clockId, portId, hdr.ptp.sequenceId, ingressNs);
        ptp_get_egress_mac_tstamp(clockId, portId, hdr.ptp.sequenceId, egressNs);

        // Step 2: add inband-network telemetry
        hdr.ptp_int.setValid();
        hdr.ptp_int.tlvType        = PTP_TLV_INT_TYPE;
        hdr.ptp_int.fieldLength    = PTP_TLV_INT_LENGTH;
        hdr.ptp_int.switchId       = switchId;
        hdr.ptp_int.ingressTstamp  = ingressNs;
        hdr.ptp_int.egressTstamp   = egressNs;
        hdr.ptp_int.correctionNs   = hdr.ptp.correctionNs;

        // Step 3: update the correctionField to reflex the delay
        correctionNs = egressNs - ingressNs;
        // add delay of its sync message to the correctionField
        // (currently we do not support subNano => no need to adjust this field)
        hdr.ptp.correctionNs = hdr.ptp.correctionNs + (bit<48>)correctionNs;

        // Step 4: update size of PTP message
        // +4: 4 bytes of header (2bytes of tlvType + 2bytes of fieldLength
        hdr.ptp.messageLength = hdr.ptp.messageLength + PTP_TLV_INT_LENGTH + 4;
    }
}
```

Listing 16: P4-Implementation Workflow of Transparence Clock.

We embed custom telemetry data in PTPv2 extension TLV fields. This method minimizes overhead by avoiding additional probe packets and leverages the existing PTP network infrastructure for efficient telemetry collection.
The structure of an extension TLV field to carry INT data used in Listing 16 is detailed as below:

- tlvType: a configurable, unique identifier.

- fieldLength: fixed at 26 bytes, ensuring each TC adds a 26-byte PTP.
- switchID: a unique identifier assigned to each TC.
- ingressTstamp & egressTstamp: the timestamps recorded at the ingress and egress of the packet.
- correctionField: the value of the correctionField before the TC applied its adjustment.

We show in Figure 22 a Wireshark representation of a Follow_Up message that has three TLV extensions to carry on INT data of three switches.

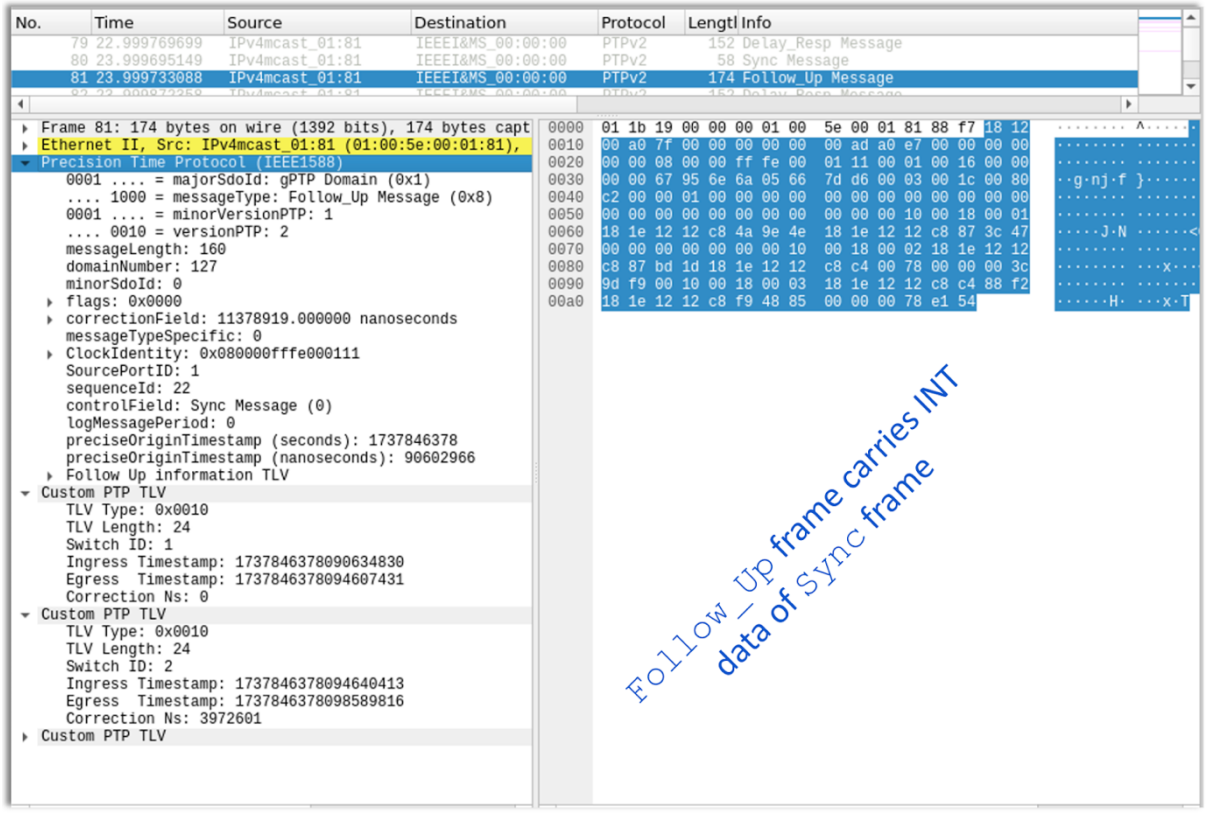


Figure 22: Wireshark Dissector Representation of INT Encapsulation.

4.2.3 INT Collector & TDA Detection

We implemented a collector to capture, analyze, and provide real-time statistics on PTPv2 packets with INT extensions. The collector consists of two threads to perform two different tasks:

- It first utilizes Scapy to sniff network traffic, parse PTP messages, including Sync, Follow_Up, Delay_Req, Delay_Resp, and extract timestamp data and telemetry fields. The script defines a custom PTP packet parser to interpret message types, timestamps, and TLV extensions, enabling accurate tracking of ingress and egress timestamps from TCs. The extracted data will then be used to detect time-delay attack (TDA) and to feed Grafana for a graphical representation. The TDA detection consists of two phases:
  - Learning phase: in this phase, we determine the time synchronization accuracy supported by the network. The duration of this phase is configured via the "--nb-learning-samples" parameter, which represents the number of INT messages.

- Monitoring phase: in this phase, we monitor the delay variation to detect TDA. The sensitive detection is configured via the "--sigma" parameter.

It then serves telemetry statistics through an HTTP server, allowing seamless integration with monitoring tools like Grafana.

Figure 23 provides a graphical Grafana representation of monitoring for the last 20 PTP messages in a time synchronization process consisting of a PTP client and server, and three TCs. The left side of the figure displays Sync messages, while the right side shows Delay_Req messages. The x-axis in each chart represents the sequence number of the messages.

In the first row, two charts illustrate the Inter-Arrival Time (IAT) variations of messages at both the server and each TC. The interval between consecutive Sync messages is consistently close to one second. Interestingly, Delay_Req messages generated by the LinuxPTP-based client exhibit variations ranging between 0 and $2^{\text{logMinDelayReqInterval}+1} = 2$ seconds, as detailed in Section 9.5.11.2 of [IEEE1588-2019]. Despite this variation, the IAT values observed at each monitored TC closely align with those recorded at the server, indicating consistent propagation behavior across the network.

The second-row charts depict the difference between the IAT values at each TC and those recorded at the server. These differences help identify deviations that may indicate potential TDAs.

The final row contains two charts: the left chart displays the arrival time of Sync packets at each TC, while the right chart shows the departure time of Delay_Req packets. Although the TCs may not be synchronized, these charts provide insight into packet propagation delays between them.

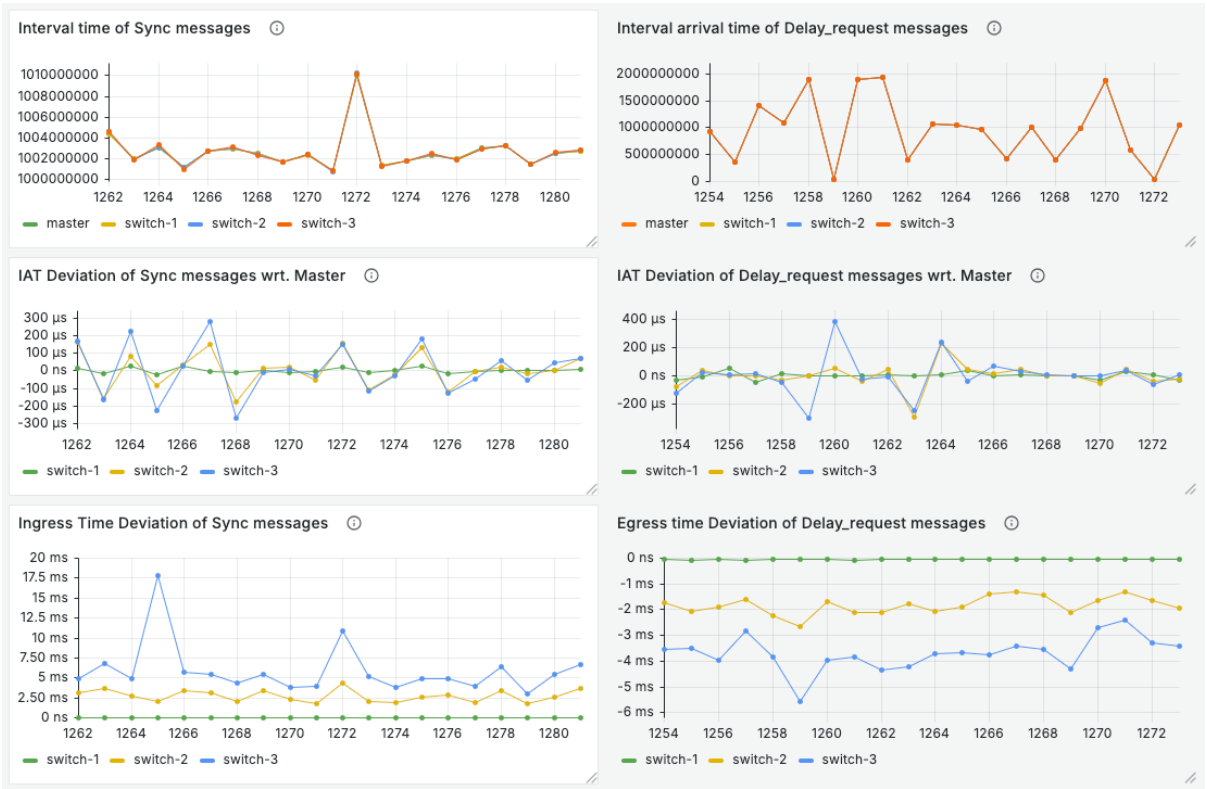


Figure 23: PTP monitoring.

5 Conclusion and Future Work

In this digest, we provided an overview of the latest enhancements of our open-source validation tools, including the following:

1. A description of the new architecture of the **6GDetCom Simulator**, as well as newly added concepts such as replaying delay traces, a Packet Delay Correction (PDC) simulation model, enhancements for the simulation of time synchronization, and network control plane interfaces supporting the simulation of dynamic scenarios. For all features, we provided an exemplary showcase to allow users to quickly understand and explore the features of the framework.
2. The introduction of our novel **6GDetCom Emulator** which enables the analysis of applications under the influence of the characteristic PD of 6G bridges without the need for dedicated 5G/6G hardware. This included an analysis of the capabilities and limitations of this emulator.
3. An **emulation framework for secure PTP time synchronization** allowing monitoring, detection, and localization of Time-Delay Attack (TDA) using a Mininet-based approach.

Throughout the project, the presented simulation and emulation frameworks already turned out to be useful for the validation of concepts developed in the Deterministic6G project, for instance, to validate wireless-friendly scheduling methods [DET24-D34], PDC mechanisms [DET23-D21, DET25-D23], time-synchronization mechanisms [DET23-D22, DET25-D24], or Time-Delay Attacks [DET25-D24]. Final validation results will be presented in Deliverable D4.5, which is dedicated to the final validation results.

Reference

[3GPP-TS22104]	3GPP, "TS 22.104: Service requirements for cyber-physical control applications in vertical domains", Release-18, v18.0.0, 2021
[CM+10]	R. Cochran and C. Marinescu, "Design and implementation of a PTP clock infrastructure for the Linux kernel," in Proc. of ISPCS, pp. 116–121, IEEE, Sept. 2010
[Coc25]	Richard Cochran, "User space PTP stack for the GNU/Linux operating system", https://github.com/richardcochran/linuxptp, last accessed April, 2025
[DET23-D11]	DETERMINISTIC6G, Deliverable 1.1, DETERMINISTIC6G Use Cases and Architecture Principles, June 2023
[DET23-D21]	DETERMINISTIC6G, Deliverable 2.1, First report on 6G centric enabler, Dec. 2023
[DET23-D22]	DETERMINISTIC6G, Deliverable 2.2, First report on time synchronization for E2E time awareness, Dec. 2023
[DET25-D23]	DETERMINISTIC6G, Deliverable 2.3, Second report on 6G centric enablers, Apr. 2025
[DET25-D24]	DETERMINISTIC6G, Deliverable 2.4, Report on the time synchronization for E2E time awareness, Apr. 2025
[DET23-D31]	DETERMINISTIC6G, Deliverable 3.1, Report on 6G convergence enablers towards deterministic communication standards, Dec. 2023
[DET23-D32]	DETERMINISTIC6G, Deliverable 3.2, Report on the Security solutions, Dec. 2023
[DET24-D34]	DETERMINISTIC6G, Deliverable D3.4, Report on Optimized Deterministic End-to-End Schedules for Dynamic Systems, Jun. 2024
[DET23-D41]	DETERMINISTIC6G, Deliverable 4.1, DETERMINISTIC6G DetCom simulator framework release 1, Dec. 2023
[DET24-D42]	DETERMINISTIC6G, Deliverable 4.2, Latency measurement framework, March 2024
[DET25-D43]	DETERMINISTIC6G, Deliverable 4.3, Latency measurement data and characterization of RAN latency from experimental trials, Apr. 2025
[HDM+23]	L. Haug, F. Dürr, S. S. Mostafavi, G. P. Sharma, J. Sachs, J. Harmatos, J. Costa-Requena, and J. Ansari, "Deterministic6g/deterministic6g_data: D4.1 - First DetCom simulator framework release (datasets)," Dec. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10405085
[IEEE1588-2019]	IEEE Standard 1588-2019, "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," tech. rep., 2020. (Revision of IEEE Std 1588-2008)
[IEEE 802.1AS]	IEEE, "IEEE Std 802.1AS-2020: IEEE standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications," 2020
[IEEE 802.1ASdm]	IEEE, "IEEE Std 802.1ASdm-2024: IEEE standard for local and metropolitan area networks–timing and synchronization for time-sensitive applications amendment 3: Hot standby and clock drift error reduction," 2024.
[INE25]	INET Framework (website), https://inet.omnetpp.org/, last accessed April, 2025

[Lib25]	Libnetconf2 contributor, “Libnetconf2 - The NETCONF protocol library”, https://github.com/CESNET/libnetconf2, last accessed April, 2025
[OMN25]	OMNeT++ Discrete Event Simulator (website), https://omnetpp.org/, last accessed April, 2025
[RFC6241]	R. Enns, M. Björklund, A. Bierman, J. Schönwälder, “Network Configuration Protocol (NETCONF)“, https://www.rfc-editor.org/info/rfc6241
[Sys25]	Sysrepo, “Sysrepo - Storing and managing YANG-based configurations for UNIX/Linux applications”, https://www.sysrepo.org/, last accessed April, 2025

List of abbreviations

5G	Fifth generation
5GS	5G system
6G	Sixth generation
BTCA	Best timeTransmitter clock algorithm
DS-TT	Device side TSN translator
E2E	End-to-End
GCL	Gate Control List
GM	Grandmaster
gNB	Next generation NodeB
gPTP	generalized Precision Time Protocol
INT	In-band Network Telemetry
HTTP	HyperText Transfer Protocol
NW-TT	Network side TSN translator
OC	Ordinaire Clock
P4	Programming Protocol-independent Packet Processor
PD	Packet Delay
PDC	Packet Delay Correction
PDV	Packet Delay Variation
PTP	Precision Time Protocol
TAS	Time-Aware Shaper (IEEE 802.1Qbv)
TC	Transparent Clock
TDA	Time-Delay Attack
TLV	Type-Length-Value
tR	time Receiver
TSe	Egress timestamp
TSi	Ingress timestamp
TSN	Time-sensitive Networking
tT	time Transmitter
TT	TSN Translator
UDP	User Datagram Protocol
UPF	User plane function

Table 8: List of abbreviations