A Platform for Time-Sensitive Networking in Converged IoT-Cloud Environments

George Papathanail, Ilias Sakellariou, Lefteris Mamatas, and Panagiotis Papadimitriou
University of Macedonia, Greece
{papathanail, iliass, emamatas, papadimitriou}@uom.edu.gr

***Abstract*—IoT-enabled applications benefit from various forms of digital twinning, such as the so-called Virtual Objects (VOs). A crucial requirement in the communication among IoT devices and their associated VOs is low latency. In this respect, we showcase a software platform for Time-Sensitive Networking (TSN), which enables the rapid computation of TSN schedules, based on flow demands, and the configuration of Gate Control Lists (GCLs) on the TSN bridges that reside between IoT Gateways and VOs.**

I. Introduction

Over the last years, we have seen an increasing need for the deployment of distributed applications that interact with Internet-of-Things (IoT) devices, paving the way for the convergence of IoT and edge computing technologies. In this respect, digital twinning is gaining traction across such environments, where various forms of IoT middleware, such as Virtual Objects (VOs), can be instantiated on-demand for the execution of IoT-specific or generic functions, alleviating their computational burden from IoT devices [1].

A crucial requirement in converged IoT-cloud environments is low latency in the communication between IoT devices and their associated VOs. Even if VOs are deployed in proximity to IoT devices, interference from cross-traffic may still cause undesirable implications, such as latency inflation and packet loss. To guarantee bounds on latency and jitter for IoT-VO traffic, we leverage on the principles of Time-Sensitive Networking (TSN), which provides a solid underpinning for sustaining bounded latency and reliability for high-priority traffic [2].

To this end, we showcase a software platform for TSN, which is oriented to the dynamicity of converged IoT-cloud environments, stemming from the relatively short lifetime of cloud-native applications and the fact that VOs are orchestratable (*i.e.,* they are subject to migration or scaling). As such, TSN schedules (that essentially designate the transmission periods for each traffic class) should be computed at short timescales, ensuring adaptability to fluctuations in terms of traffic patterns and demands. In the following, we briefly discuss the architecture of the TSN platform to be demonstrated, which is based on our previous work [1]–[4].

II. TSN Platform

Our TSN platform represents a holistic TSN approach that couples a Centralized Network Controller (CNC) with a Time-Aware Shaper (TAS) compliant software TSN bridge (Fig. 1). In a nutshell, the TSN platform supports the computation of TSN schedules and the population of Gate Control List (GCL) configurations onto the TSN bridges that reside between the IoT Gateway and the compute node that hosts the VO.

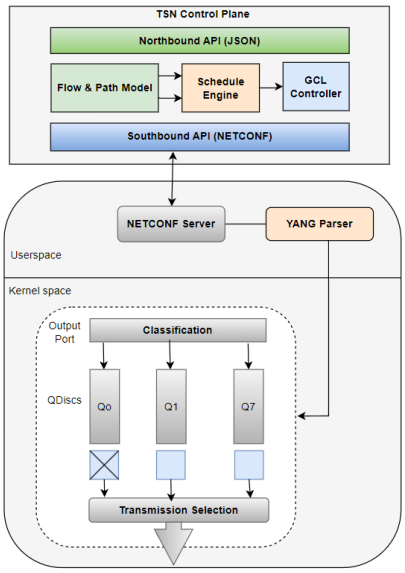

Fig. 1: TSN platform overview.

A. Control Plane

The CNC consists of three internal modules: (i) Flow & Path Model, (ii) Schedule Engine, and (iii) GCL Controller:

Flow & Path Model. The *Flow and Path Model* module supports two functionalities. The first one is the categorization of incoming flows into distinct traffic classes, such as high-priority and best-effort. This is achieved based on some predefined rules that can match applications' network requirements to traffic classes and determine whether the request should be categorized as critical or non-critical. The second functionality of this module is the path configuration, which corresponds to the sequence of TSN bridges between the IoT Gateway and the VO. The path is used as an input to the Schedule Engine module.

Schedule Engine. The main objective of the Schedule Engine is to determine a feasible scheduling pattern for a set of incoming flows with respect to their specified requirements.

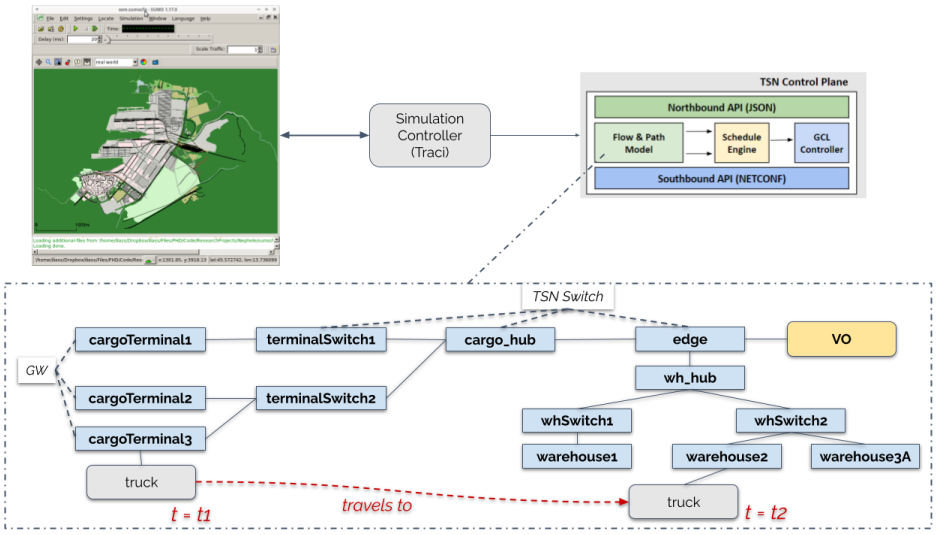

Fig. 2: Simulation of CNC interaction and port topology.

In this respect, we consider a set of periodic flows, each flow being associated with a valid path from a talker to a listener, and having a deadline, a packet size and period. The problem can be considered as a classic job-shop scheduling problem with a set of additional constraints (*i.e.,* restricting the transmission to a single packet at each time, ensuring proper ordering in the reception of frames in the queues, ensuring ordering constraints between transmissions of the same packet along the path). The current implementation of the model (described in detail in [4]) relies on the ECLiPSe Constraint Logic Programming system, harvesting on its support of the *disjunctive* global constraint for scheduling, reified constraints, flexible and efficient search strategies, and variable/value ordering general heuristics.

GCL Controller. The GCL Controller module receives the output of the Schedule Engine as its input and is responsible for configuring time intervals on the GCL, and determining the duration over which each queue is open for transmission. The GCL Controller utilizes YANG in order to populate the GCL configurations onto the TSN bridges.

B. Data Plane

For TSN data plane activation, we rely on Time-Aware Priority Packet Scheduler (TAPRIO), a powerful queuing discipline available in the Linux kernel's traffic control (tc) tool. TAPRIO plays a crucial role in simulating the behavior of IEEE 802.1Qbv, which is a standard for enhancing time-aware scheduling in Ethernet networks. By integrating TAPRIO, we allow the configuration of a series of gate states, each one responsible for enabling outgoing traffic for specific subsets of traffic classes based on the concept of time slices.

To ensure proper packet classification into the appropriate traffic class, TAPRIO uses the priority field of the socket buffer (*skb*) employed by the network stack of the Linux Kernel. This enables TAPRIO to effectively assign time-sensitive flows to their respective priority queues. In our implementation, we map traffic classes to queues by modifying the Differentiated Services Code Point (DSCP) field of the packet header. To achieve the modification of the *skb* priority field before packets are directed to the queuing discipline, we utilize *iptables* (see [3] for further details).

C. Interfaces and Interactions

The TSN control plane incorporates the following interfaces: (i) a Northbound API, implemented using a well-defined JSON schema, which is capable of processing requests related to application configuration and requirements (this information can be conveyed from a Service Orchestrator), and (ii) a technology-specific Southbound API, utilizing NETCONF, which is responsible for transmitting the GCL configuration to each TSN bridge via Remote Procedure Calls (RPC).

The interaction between the CNC and a TSN bridge using NETCONF is illustrated in Fig. 1. A NETCONF plugin runs at the CNC as a management client and establishes communication with the NETCONF server that is operational on each TSN bridge. CNC establishes communication through the NETCONF plugin by utilizing a YANG-TSN data model. A YANG Parser, deployed at the userspace of the TSN bridge, translates the YANG-TSN model to a set of actions that can be applied directly to the queuing disc layer of Linux kernel.

III. Demo Description

Our demonstration includes a simulation of IoT-VO communication in a port context, where we consider IoT devices mounted on trucks, the latter transporting cargo containers from cargo terminals to warehouses in the port. IoT devices are associated with a VO located at the edge and communicate via a path on a topology depicted in Fig. 2. The scenario involves dynamically re-configuring TSN switches on newly associated paths that depend on the location of the trucks (*e.g.,* cargo terminal or warehouse positions). For instance,

```
"type": "taprio",
"switches": [
    {
        "switch": "wh_hub",
        "interfaces": [
            {
                "src": "wh_hub",
                "dest": "edge",
                "schedEntries": [
                    {
                        "schedEntry": 1,
                        "gatemask": "01",
                        "interval": 52000
                    },
                    {
                        "schedEntry": 2,
                        "gatemask": "02",
                        "interval": 32000
                    },
                    {
                        "schedEntry": 3,
                        "gatemask": "01",
                        "interval": 716000
                    }
                ]
            }
        ]
```


Fig. 3: Schedule Engine output.

in Fig. 2, the truck IoT at time point $t1$ is connected to the `cargoTerminal3`, and the corresponding flow is forwarded via the `cargo_hub` switch, whereas at time point $t2$ connects to `warehouse2` with the associated path going through `wh_hub`. Truck locations are obtained from a port road traffic simulation model, implemented in Eclipse SUMO, as shown in Fig. 2. In the specific scenario, we consider six trucks transporting cargo containers from terminals to different warehouses.

In brief, the demo begins with a call received on the NorthBound API of the CNC. In practical terms, when a truck arrives at a new location, an appropriate call is generated to the CNC by the SUMO platform interface (Simulation Controller) to (re)compute schedules for the data flows, according to specified high-level intents of the overall application (for example, achieving latency below 1 ms). A flow is identified as either critical or best-effort by the *Flow and Path model* component. The latter, which also maintains information on the underlying network, transmits a JSON request with all the necessary information to the *Schedule Engine*, which is responsible for calculating the schedules of the flows.

The request contains details about the topology and flows. More specifically, the topology information includes: (i) the switches and end-points, and (ii) the links and their attributes, such as delay and bandwidth. Flow information comprises a distinct *flowId*, the *deadline* for each flow (*i.e.,* the time by which the packet must reach its destination), the *packetSize*, and the *period* which sets the frequency of transmission and is linked to a valid path. The *Schedule Engine* based on the provided data computes schedules for each flow and generates accordingly the corresponding GCL entries for each switch. The output of the scheduling model is illustrated in Fig. 3. In this instance, the scheduler generates an output of type TAPRIO, which is applied to the switch `wh_hub`. This instance further incorporates intervals for each schedule input.

The computed schedule is then transferred to the GCL Controller. The primary function of the GCL controller module


```
<?xml version='1.0' encoding='utf-8'?>
<tsn-taprio-structure
    xmlns="http://sssa.it/yang/tsn-taprio">
    <interface>
        <dev>ens4</dev>
        <parent>root</parent>
        <handle>10</handle>
        <num-tc>2</num-tc>
        <map>1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1</map>
        <base-time>1528743495910289987</base-time>
        <clockid>CLOCK_TAI</clockid>
        <sched-entries>
            <sched-entry>
                <schedEntry>1</schedEntry>
                <gatemask>01</gatemask>
                <interval>52000</interval>
            </sched-entry>
            <sched-entry>
                <schedEntry>2</schedEntry>
                <gatemask>02</gatemask>
                <interval>32000</interval>
            </sched-entry>
            <sched-entry>
                <schedEntry>3</schedEntry>
                <gatemask>01</gatemask>
                <interval>716000</interval>
            </sched-entry>
        </sched-entries>
    </interface>
</tsn-taprio-structure>
```


Fig. 4: Remote Procedure Call.

is to generate a RPC using the IETF TAPRIO YANG model and transmit it to the switch through the SouthBound API, which acts as a NETCONF client. An instance of the RPC is illustrated in Fig. 4. Eventually, this RPC is extracted from the YANG parser module, which functions within the userspace of the switch. This module is responsible for converting the YANG format into the *tc qdisc* command and subsequently applying this configuration to the egress interface of the switch. The above workflow is repeated each time a simulation truck arrival event is generated by the SUMO platform.

IV. Conclusions

In this paper, we presented the architecture of a software TSN platform, tailored to the needs of emerging IoT-enabled applications in conjunction with IoT VOs. The TSN platform couples TSN schedule computation with TAS-compliant scheduling using NETCONF, and it is showcased in the context of a smart port use case.

Acknowledgments

This work was funded by the European Union's Horizon Europe research and innovation program under grant agreement No. 101070487 (NEPHELE).

References

[1] G. Papathanail et. al, "A virtual object stack for iot-enabled applications across the compute continuum," in *IEEE/ACM CEICO*, 2023, pp. 1–6.

[2] G. N. Kumar, K. Katsalis, P. Papadimitriou, P. Pop, and G. Carle, "Failure handling for time-sensitive networks using sdn and source routing," in *7th IEEE International Conference on Network Softwarization (NetSoft)*, 2021, pp. 226–234.

[3] G. Papathanail, L. Mamatas, and P. Papadimitriou, "Towards the integration of taprio-based scheduling with centralized tsn control," in *IFIP Networking*, 2023, pp. 1–6.

[4] G. Papathanail, I. Sakellariou, L. Mamatas, and P. Papadimitriou, "Dynamic schedule computation for time-aware shaper in converged iot-cloud environments," in *IEEE ICIN*, 2024, pp. 1–8.