Project: Analysis and Simulation of Asynchronous Traffic Shaping (ATS) in Time-Sensitive Networking (TSN)

Paul Pop, Lars Emil Tvernø Halling, Juri Borren and Lei Zhang
DTU Compute, Technical University of Denmark
Email: paupo@dtu.dk

Final version, Sept. 25

Updates:

- 2023-09-25: Updated the notations and the ATS delay analysis to match the TBE variant of the UBS paper.
- 2023-09-18: Completed the OMNeT++ guide and updated the test cases guide.
- 2023-09-10: Added a guide on creating test cases.
- 2023-09-10: Added the OMNeT++ guide.
- 2023-09-04: Document created.

Abstract

This document presents the project for the "02226 Networked Embedded Systems" course at DTU Compute. All the required materials mentioned in the text, including the test cases, are available via DTU Learn. IEEE 802.1 Time-Sensitive Networking (TSN) is a communication standard used in many application areas. Given a set of communication streams, and a TSN network topology consisting of switches and communication links, the problem is to determine the delays for the streams considering the Asynchronous Traffic Shaping (ATS) mechanism in TSN. You will use OMNeT++ to simulate the network and you will develop a software tool that calculates analytically the worst-case delays of the streams.

Contents

1 Introduction

Motivation: *Distributed* cyber-physical embedded systems have their functions distributed in a network. For example, a network backbone in a modern vehicle has to integrate Advanced Driver Assistance Systems (ADAS) functions, which rely on high-bandwidth data from sensors, such as video cameras and Light Detection And Ranging (LIDAR), with powertrain functions that have tight timing constraints but use small frame sizes, and diagnostic services, which are not time-critical. Owing to the increase in complexity, and the need to reduce costs, such mixed-criticality applications are currently implemented in *distributed architectures*, where the functions of different criticality share the same distributed platform.

There are various communication protocols on the market, depending on the application area, e.g., FlexRay for automotive, ARINC 664 p7 for avionics [1], and EtherCAT for industrial automation [13]. However, emerging applications, such as ADAS, autonomous driving, or Industry 4.0, have increasing bandwidth demands. For instance, autonomous driving requires data rates of at least 100 Mbps for graphical computing based on camera, radar, and LIDAR data, whereas CAN and FlexRay only provide data rates of up to 1 Mbps and 10 Mbps, respectively. Furthermore, there have been many safety-critical protocols proposed, though only a few of them can support the separation required by mixed-criticality messages [16].

TSN: The well-known networking standard IEEE 802.3 Ethernet [5] meets the emerging bandwidth requirements for safety-critical networks, besides remaining scalable and cost-effective. However, Ethernet is unsuitable for real-time and safety-critical applications [3]. Many extensions, such as EtherCAT [13], PROFINET [4], ARINC 664p7 [1], and TTEthernet [17], have been suggested and used in the industry. Although they satisfy the timing requirements, they are mutually incompatible, and hence, they cannot operate on the same physical links in a network without losing real-time guarantees. Consequently, the IEEE 802.1 Time-Sensitive Networking task group [6] has been working since 2012 to standardize the real-time and safety-critical enhancements for Ethernet. TSN primarily consists of amendments ("sub-standards") to IEEE 802.1Q[1]. TSN is quickly becoming the de facto standard in several areas, e.g., industrial, automotive, avionics, space, with a wide industry adoption and several vendors developing TSN switches.

Analysis problem: In this project, we are interested in cyber-physical distributed embedded systems that use IEEE 802.1 Time-Sensitive Networking (TSN) for communication, see [15] for a high-level presentation of TSN. TSN-based systems are composed of end systems (ESes) interconnected by network switches (SWs) and duplex physical links, see Section 3.1 for the model of a TSN network.

TSN supports the convergence of multiple traffic types, i.e., critical, real-time, and regular "best-effort" traffic within a single network, and hence, is suitable for mixed-criticality applications. We consider that the applications send messages using the Asynchronous Traffic Shaping (ATS) mechanism [2, 18], which allows for flexible management of traffic priorities, adjusting bandwidth allocation dynamically. ATS is advantageous as it adapts to varying network conditions, ensuring reliable real-time communication without static configurations, thereby simplifying network management while still providing strong real-time guarantees.

SWs are forwarding messages from their input (ingress) ports to their output (egress). Each egress port has eight priority queues where messages are placed before transmission. ATS utilizes these queues to manage traffic dynamically, see Section 2.3 for an explanation on how ATS works. The mixed-criticality real-time applications are modeled as a set of periodic streams. For each stream, we know its source and destination ESes, its period, and its deadline. See Section 3.2 for the details of the application model.

The analysis problem can be formulated as follows. Given a TSN network topology consisting of switches and communication links, and a set of communication streams, the objective is to determine the delays for each stream considering the ATS mechanism in TSN. The analysis involves using OMNeT++ to simulate the network behavior and developing a software tool to analytically calculate the worst-case delays of the streams. This process ensures we address both the practical and theoretical aspects of network performance, aiming to verify that all streams meet their deadlines and maintain effective network utilization under ATS.

2 TSN and ATS

Ethernet is a wired networking technology first standardized in IEEE 802.3 in 1983 for use in LANs. It has since evolved, and is now not only used for computer-to-computer networks, but also for appliances, automobiles, industrial applications, etc [14].

[1]The references for all sub-standards can be easily found based on their names. All the relevant materials for the project are also available via DTU Learn

Modern Ethernet uses switches to interconnect end-systems, which act as talkers and listeners. The connections are made up of shared full duplex physical links, which allow transmission in both directions.

End-systems communicate using streams, which are split into smaller frames before transmission on a link. The payload of a frame ranges from $46 - 1500$ bytes, where the maximum size (1500 bytes) is referred to as a *Maximum transmission unit (MTU)*. In addition to the payload, a 48 byte header is added.

2.1 General switch model

Figure 1 shows the general network model and switch architecture that will be used for the following sections. The figure depicts two end-systems (ES), *ES 1* and *ES 2*, and three switches (SW), *SW 1, SW 2* and *SW 3*. Links are depicted with arrows connecting switches and end-systems.

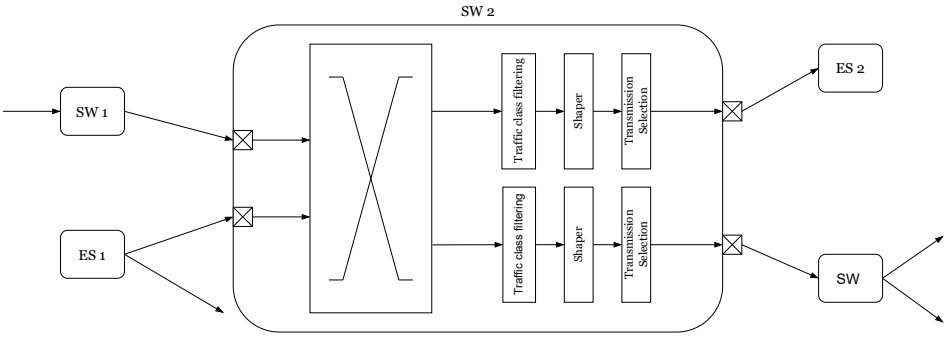


Figure 1: General network and switch architecture.

The architecture of *SW 2* is shown in detail. In the case of figure 1, frames move from left to right, and thus the two ports to the left of the switch are ingress ports for receiving frames, and the two ports to the right of the switch are egress ports for forwarding frames. Initially, the switching fabric, depicted as a square with a stretched cross, forwards frames from an ingress port to the frames' respective egress port.

This is followed by an optional traffic class filtering, which sorts traffic into queues depending on the frames traffic class. This step is not necessarily needed depending on the shaper and number of traffic classes. For the following sections summarizing specific switch architectures, it is assumed a traffic class filtering is always present.

Once traffic is sorted into traffic classes, it is regulated according to a traffic shaper. This shaper acts as congestion control and regulates the rate of traffic passing through it. The most simple shaper is *best effort*, which is the same as having no shaper. Best effort traffic doesn't provide any guarantees for quality-of-service and simply means "Forward data as fast as possible". Other shapers provide more sophisticated congestion control and will be described in later sections.

Finally, a transmission selection algorithm is used to select which frames are transmitted, from the ones that are ready. In the case of single-priority best-effort traffic, a transmission selection could simply be first-come-first-served. Another simple transmission selection algorithm is selecting frames based on strict priority, which is the algorithm used throughout this report.

2.2 Time-Sensitive Networking

Time-Sensitive Networking (TSN) builds on top of Ethernet and was created to minimize and create known bounds for delay, synchronization error, and jitter. It was originally set forth as IEEE 802.1 Audio-Video Bridging (AVB), as these properties were found especially important for Audio/Video (A/V) applications. For example, [19] mentions that *"... video can lead audio by as much as 25 ms but video may lag behind audio by only 15 μs; this is due to the way human brains are wired to perceive late audio as normal, but early audio as unnatural."* This gives a strict delay bound for A/V packets over a network, and gives a clear example of where best-effort traffic is not sufficient. To meet these requirements, the AVB task group working on these standards had three goals in mind [19]:

- Provide network-wide precision clock.
- Limit network delays to a known value.
- Keep best-effort traffic from interfering with TSN traffic.

The following protocols were created to meet these goals:

1. IEEE Std. 802.1AS - Generalized Precision Time Protocol (gPTP) [8]: A layer-2 protocol for synchronizing time over a network, built on top of IEEE 1588 Precision Time Protocol (PTP), which uses physical layer timestamps. The protocol works by using a single clock source, the *grand-master clock*, which all other network devices use as a time reference.

2. IEEE Std. 802.1Qat - Stream Reservation Protocol (SRP) [9]: A protocol used for announcing streams, reserving bandwidth for streams, and establishing paths through the network from talker to listener(s). This is done by flooding talker messages over the network announcing streams, and in return responding with listener messages to subscribe to said streams.

3. IEEE Std. 802.1Qav - Forwarding and Queuing of Time-Sensitive Streams (FQTSS) [10]: A specification for the credit-based shaper, which shapes traffic according to the leaky-bucket constraint. The shaper spaces out high-priority traffic as much as possible, to avoid long bursts of blocking traffic, which are responsible for a significant reduction in Quality-of-Service for lower-priority frames.

4. IEEE Std. 802.1BA - AVB Systems [11]: A specification of the overall system architecture.

As the scope of AVB has grown beyond that of A/V applications, such as the automotive industry or industrial automation, the AVB task group has been renamed to the TSN task group. This new group is actively working on improving the aforementioned protocols and specifications, as well as adding new standards for improving deterministic Ethernet networks.

One of the main topics where a lot of work has been done is adding and standardizing new traffic shapers, two of which will be the main focus of this report. The two shapers will be described in the following sections.

2.3 Asynchronous Traffic Shaper

The Asynchronous Traffic Shaper (ATS), originally introduced as the Urgency-Based Scheduler (UBS), was proposed in 2016 by Specht and Samii in [18]. Standardization of UBS has since been started in ”802.1Qcr Bridges and Bridged Networks Amendment 34: Asynchronous Traffic Shaping” [12], now under the Asynchronous Traffic Shaping name. The goal of ATS is to provide low and predictable worst-case delays at high link utilization, while being independent of a global time.

Figure 2 shows the architecture of an ATS switch with two ingress ports and one egress port. ATS provides per-flow shaping at every hop, a concept called *interleaved shaping*. This is done by implementing two layers of queues, one layer for the interleaved shaping itself, called shaped queues, and one for storing frames ready to be selected by the transmission selection algorithm, called shared queues.

Frames are forwarded from the traffic class filtering (TCF) into a shaped queue according to three rules. Let f_i, f_j be two frames being forwarded from the TCF. Then *QC1, QC2* and *QC3* specify when f_i and f_j can't be in the same queue:

QC1: Flows f_i and f_j cannot be assigned the same shaped queue if f_i and f_j are sent by different upstream servers.

QC2: Flows f_i and f_j cannot be assigned the same shaped queue if f_i and f_j are sent in different priority levels by the same upstream server.

QC3: Flows f_i and f_j cannot be assigned the same shaped queue if f_i and f_j are sent by the server in different priority levels.

QC1 maintains separation of flows being received at different ingress ports. This ensures that no malicious upstream nodes can completely block an ATS node, as traffic from other upstream nodes are allowed to overtake traffic arriving from the spamming node. QC2 and QC3 maintain separation of flows in different priorities. This simply ensures that higher-priority frames can overtake lower-priority frames.

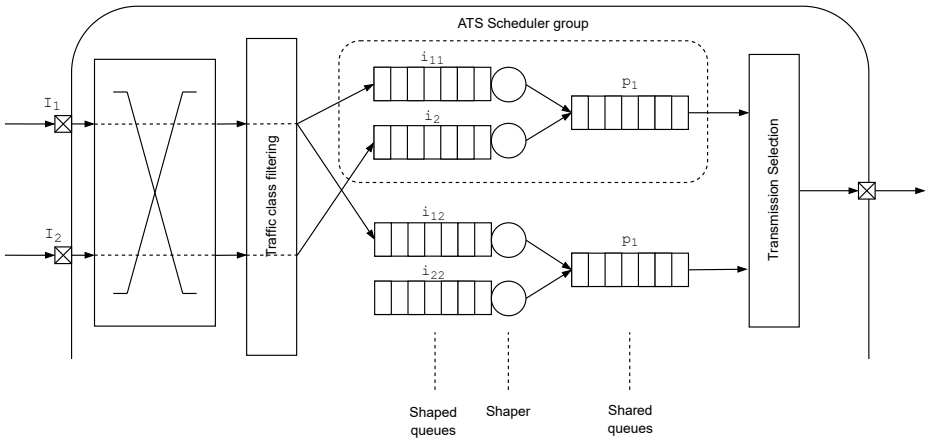


Figure 2: Architecture of an ATS switch with two ingress ports and one egress port.

What these three rules mean in practice is that there is going to be one shaped queue per ingress-priority pair, and one shared queue per priority. E.g. figure 2 shows two ingress ports, I_1 and I_2, two priorities, p_1 and p_2, and one egress port. The ingress port I_1 receives two frames, one in priority p_1 and one in priority p_2. These frames are forwarded to queue i_{11} and i_{12}, respectively, both of which don't have frames that have been received by any other ingress port, and don't contain any frames that have been received or are being sent in any other priority than p_1 and p_2, respectively. The ingress port I_2 just receives a single frame that belongs to priority p_1. This frame is simply forwarded to queue q_{21}, which again follows QC1, QC2, and QC3.

The ATS scheduler is what forwards frames from shaped queues to shared queues, and is depicted as circles in figure 2. There is one ATS scheduler per shaped queue, which is allowing for interleaved shaping. The shaping algorithm follows the principles of token-bucket emulation (TBE), which is depicted in figure 3. The figure shows a queue of message that is waiting to be forwarded by the TBE shaper. A message is forwarded by requesting the bucket to forward it. If the bucket contains enough tokens it can "pay" for the message to be forwarded. The price of forwarding a message is linear with the size of the message, thus larger messages are more expensive to forward than smaller messages. In case there are not enough tokens in the bucket, the message is queued into a FIFO queue, and is forwarded once the bucket has enough tokens.

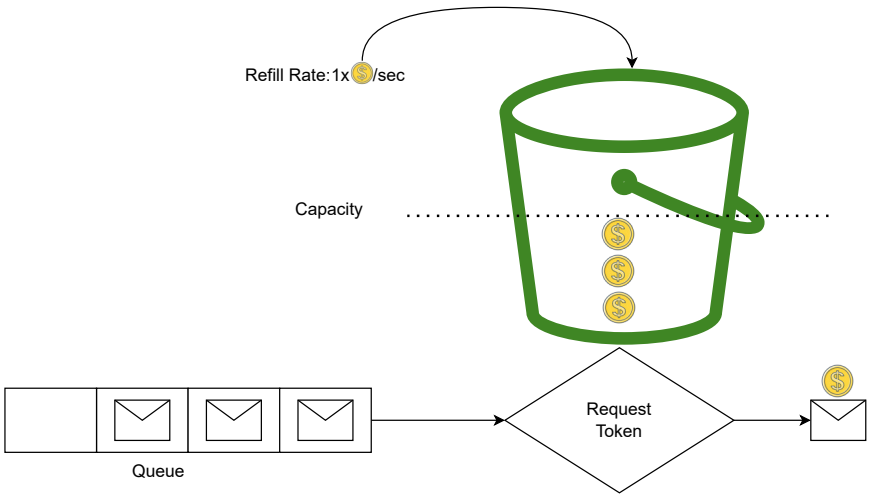


Figure 3: Token Bucket Emulation example

The refill rate, or simply referred to as the *rate*, is the frequency at which the bucket is filled with tokens. For example, in figure 3 the rate is simply one token per second.

The capacity, also called the *burst*, is the max capacity of the bucket, above which any incoming tokens are discarded. The capacity of the bucket corresponds to how many messages can be sent in rapid succession, hence the name burst. E.g. in the example given in figure 3, the capacity is three tokens, thus three messages of size one

can be sent in succession.

Given the rate r and the burst b, the maximum size of all messages forwarded $|w|$ in a time interval Δt is given by equation 1. This equation will later be used for the timing analysis of ATS frames.

$$|w| \leq b + r \cdot \Delta t \tag{1}$$

The Token-Bucket Emulation procedure is described in [12] under the name `ProcessFrame`. As opposed to the original UBS suggestion, the `ProcessFrame` procedure doesn't explicitly use tokens, but instead indirectly calculates a token state based on the timing of previous events. The algorithm works by assigning an eligibility time to incoming frames in FIFO order based on a local clock. Once a frame's eligibility time is passed, it is forwarded to its corresponding shared queue and is ready to be selected for transmission. This procedure is shown in Algorithm 1.

The algorithm first calculates the `lengthRecoveryDuration`, which is the time it takes for the token bucket to gain tokens in an amount equivalent to the length of the frames. This is followed by computing the `emptyToFullDuration`, which is the time it takes to fill up the bucket from empty. The `schedulerEligibilityTime` is the earliest time at which the shaper can forward the frame, equivalent to the time at which there are enough tokens to forward the frame. This timestamp is calculated as the sum of the last time the bucket was empty, `BucketEmptyTime`, and the time it takes for the bucket to regain enough tokens to send the frame, `EmptyToFullDuration`.

The eligibility time of the frame, `eligibilityTime`, is then the max of the arrival time of the frame, the eligibility time of the scheduler, and the eligibility time of the ATS scheduler group. An ATS scheduler group is depicted in figure 2, and is simply a grouping of all the shaped queues and schedulers which forward frames into the same shared queue. The `GroupEligiblityTime` is the latest `eligiblityTime` calculated by a shaper in an ATS scheduler group.

Finally, it is checked if the found eligibility time of the frame exceeds some predefined time, `MaxResidenceTime`, which specifies how long a frame can stay in a shaped queue. If the frame exceeds the `MaxResidenceTime`, the frame is discarded. Otherwise, the frame is assigned the calculated eligibility time, and the `GroupEligibilityTime` and `BucketEmptyTime` are updated according to the calculated eligibility time.

Data: *frame*
`lengthRecoveryDuration` ← `|frame| / rate;`
`emptyToFullDuration` ← `burst / rate;`
`schedulerEligibilityTime` ← `BucektEmptyTime + lengthRecoveryDuration;`
`bucketFullTime` ← `BucketEmptyTime + emptyToFullDuration;`

$$\texttt{eligibilityTime} \leftarrow \max \begin{cases} \texttt{arrivalTime(frame)} \\ \texttt{GroupEligibilityTime} \\ \texttt{schedulerEligibilityTime} \end{cases}$$

if *eligibilityTime* ≤ *arrivalTime(frame) + MaxResidenceTime/1.0e9* **then**
  `/* Valid Frame */`
  `GroupEligibilityTime` ← `eligibilityTime;`
  if *eligibilityTime < bucketFullTime* **then**
    `BucketEmptyTime` ← `schedulerEligibilityTime;`
  else
    `BucketEmptyTime` ← `schedulerEligibilityTime + eligibilityTime − bucketFullTime;`
  end
  `AssignAndProceed(frame,eligibilityTime)`
else
  `/* Invalid Frame */`
  `Discard(frame)`
end

Algorithm 1: `ProcessFrame(frame)`

Once a frame is forwarded, it is ready to be transmitted by the transmission selection algorithm (TSA). The TSA considered for this project is the Strict Priority TSA, which chooses to transmit the frame with the highest priority out of the ones ready for transmission. Other transmission selection algorithms are available for ATS switches and are specified in [12].

3 System Models

The system model consists of a network model and an application model described in Section 3.1 and Section 3.2, respectively.

3.1 Network Model

The network is modeled as a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertices $\mathcal{V}$ and edges $\mathcal{E}$. Vertices represent network switches and end-systems. An edge going from $v_f \in \mathcal{V}$ to $v_t \in \mathcal{V}$ represents a one-way link between v_f and v_t. Thus, to represent a full-duplex link between v_f and v_t, an edge from v_f to v_t and an edge from v_t to v_f are needed.

Edges are represented by a tuple:

$$(v_f, v_t, r)$$

where:

- $v_f \in \mathcal{V}$: The source vertex (node) of the edge (the vertex from which the edge is outgoing).
- $v_t \in \mathcal{V}$: The destination vertex (node) of the edge (the vertex to which the edge is incoming).
- r: The data rate (bandwidth) of the egress port associated with the edge.

An edge $e \in \mathcal{E}$ going from v_f to v_t also represents the corresponding egress port of v_f and the corresponding ingress port of v_t associated with e. For the remainder of the examples in this project, it is assumed that no two links share the same v_f and v_t nodes. This allows for edges to be abbreviated as (v_f, v_t) for the remainder of this project.

A path in the network is represented by an ordered list of edges. For example, a path going from v_1 to v_2 to v_3, where $v_1, v_2, v_3 \in \mathcal{V}$, is represented as $[(v_1, v2), (v_2, v_3)]$. Every valid route in the network starts and ends at an end-system and may have one or more intermediate switches.

3.2 Application Model

The application model is described by a set of flows $\mathcal{F}$ sent through the network. An ATS flow $f \in \mathcal{F}$ can be described by the tuple:

$$(path_f, p_f, d_f, l_f, r_f, \hat{b}_f, \check{l}_f, \hat{l}_f)$$

where:

- $path_f$: The path of the flow from the sender to the receiver, as described in the network model.
- p_f: The priority of the flow.
- d_f: The deadline of the flow, which is the maximum time allowed for the frame to be transmitted from source to destination. For a flow to be valid, the worst-case delay of the frame must be less than or equal to the deadline.
- l_f: The nominal frame length of the flow, usually described in bits.
- r_f: The reserved data rate of the flow, as per the ATS traffic shaping requirements.
- $\hat{b}_f$: The burst size of the flow, representing the maximum amount of data that can be sent in a burst.
- $\check{l}_f$: The minimum frame length of the flow.
- $\hat{l}_f$: The maximum frame length of the flow.

The total number of flows that can be sent over an edge $e \in \mathcal{E}$ is constrained by the fact that the sum of the reserved rates of all flows sent through edge e must not exceed the total bandwidth r of e. That is:

$$\sum_{f_i \in \mathcal{F}_e} r_{f_i} \leq r$$

where $\mathcal{F}_e$ denotes the set of flows that are passing through edge e, and r_{f_i} denotes the reserved data rate of flow f_i.

3.2.1 Token Bucket Parameters

Each flow $f \in \mathcal{F}$ is characterized by its token bucket parameters:

- $\hat{b}_f$: The burst size of the flow.
- r_f: The reserved data rate of the flow.

These parameters define the maximum amount of data that can be transmitted over any interval of time. Specifically, the amount of data $w_f(d)$ sent by flow f over any interval of duration d satisfies:

$$w_f(d) \leq \hat{b}_f + r_f \cdot d \tag{2}$$

3.2.2 Priority Levels and Interference Sets

Flows are assigned to priority levels, which determine their ordering in the network's scheduling mechanism. For a flow f, we define:

- $\hat{r}_H$: The total reserved data rate of all flows with higher priority than f.
- $\hat{b}_H$: The total burst size of all flows with higher priority than f.
- $\hat{r}_{C(f)}$: The total reserved data rate of all flows with the same priority as f, excluding f itself.
- $\hat{b}_{C(f)}$: The total burst size of all flows with the same priority as f, excluding f itself.

These parameters are used in the delay analysis to compute the interference that flow f experiences due to other flows in the network.

4 Problem Formulation

We formulate the problem as follows: Given the set of all streams in the system and the network graph, we are interested to determine the worst-case delays of ATS frames for all streams and validate them using simulation. The ATS analysis to be used is presented in Sect. 4.1. The simulation should use the OMNeT++ simulator with the ATS module[2].

4.1 Delay Analysis of ATS Frames

The timing analysis of ATS frames is based on the original Urgency-Based Scheduler (UBS) paper [18], focusing on the Token Bucket Emulation (TBE) version, which aligns with ATS. This analysis provides an upper bound on the delay experienced by a single frame over a single hop. To determine the end-to-end delay of a stream, the per-hop delays can be summed due to the property of temporal composability.

The analysis considers key time points within a single shaper-to-shaper hop, as illustrated in Figure 4. The figure shows a frame f traversing from one ATS shaper to another, highlighting the different delay components.

[2] `https://inet.omnetpp.org/docs/showcases/tsn/trafficshaping/asynchronousshaper/doc/index.html`

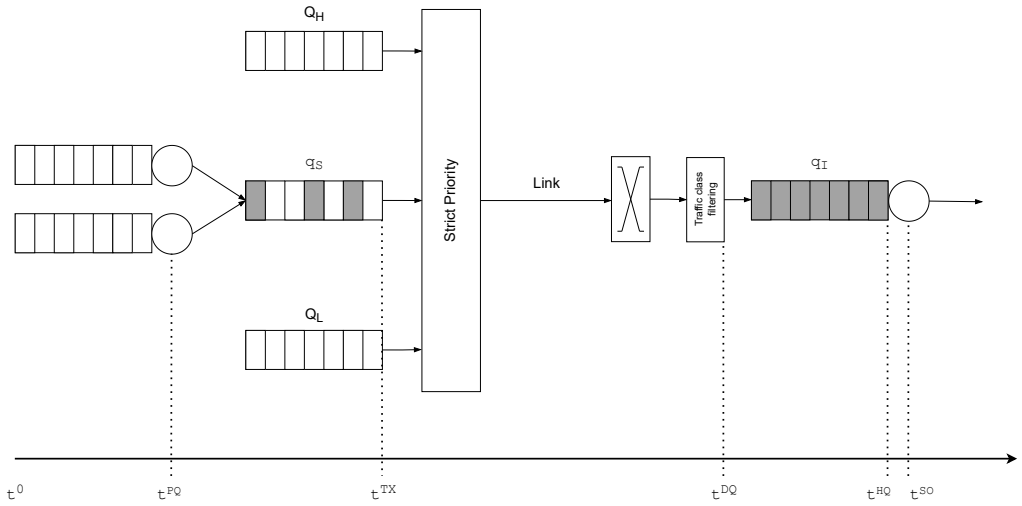


Figure 4: Shaper-to-shaper hop with relevant time points for a frame passing through the hop.

Let us define the following:

- q_S: The ready queue where frame f waits before transmission.
- q_I: The shaped queue at the next node where frame f arrives after transmission.
- Q_H: Ready queues of higher priority than f.
- Q_L: Ready queues of lower priority than f.

The six key time points are:

- t^0: The start of the busy period.
- t^{PQ}: The arrival time of frame f in its ready queue q_S.
- t^{TX}: The time when frame f starts transmission from q_S.
- t^{DQ}: The arrival time of frame f in the shaped queue q_I at the next node.
- t^{HQ}: The time when frame f becomes the head of q_I.
- t^{SO}: The time when frame f leaves q_I (after shaping delay), ready for transmission selection.

We define the following durations:

- $d^{PQ,TX} = t^{TX} - t^{PQ}$: Delay from arrival in q_S to start of transmission.
- $d^{TX,DQ} = t^{DQ} - t^{TX}$: Delay from start of transmission to arrival at q_I (transmission delay).
- $d^{DQ,SO} = t^{SO} - t^{DQ}$: Delay from arrival at q_I to departure from q_I (shaping delay).
- $d^{PQ,SO} = t^{SO} - t^{PQ}$: Total per-hop delay for frame f.

Our goal is to calculate $d^{PQ,SO}$, which is the total per-hop delay experienced by frame f. This total delay can be expressed as:

$$d^{PQ,SO} = d^{PQ,TX} + d^{TX,DQ} + d^{DQ,SO} \tag{3}$$

We will compute each component separately.

Upper Bound for Delay $d^{PQ,TX}$

The delay $d^{PQ,TX}$ is caused by the backlog of frames in the ready queues ahead of frame f. Frames that can block f in q_S are:

- **Higher Priority Frames (w_H):** Frames in Q_H that have arrived before or after f but have higher priority.
- **Same Priority Frames (w_C):** Frames in q_S that arrived before f. Due to FIFO ordering, frames arriving after f do not interfere.
- **Lower Priority Frame in Transmission (l_L):** A lower priority frame that was already in transmission when the busy period started.

The total backlog that can delay f is given by:

$$(d^{0,PQ} + d^{PQ,TX}) \cdot r \geq w_H + w_C + \hat{l}_L \tag{4}$$

Here, $d^{0,PQ}$ is the time from the start of the busy period to the arrival of f in q_S, and r is the link rate.
To bound w_H and w_C, we use the token bucket constraints of the interfering streams. Let:

- $\hat{b}_H$, $\hat{r}_H$: Total burst size and reserved rate of all higher priority streams.
- $\hat{b}_C$, $\hat{r}_C$: Total burst size and reserved rate of all same priority streams (excluding f).

The maximum backlog of higher priority frames is:

$$w_H \leq \hat{b}_H + (d^{0,PQ} + d^{PQ,TX})\hat{r}_H \tag{5}$$

Similarly, the maximum backlog of same priority frames is:

$$w_C \leq \hat{b}_C + d^{0,PQ}\hat{r}_C \tag{6}$$

Substituting equations 5 and 6 into equation 4:

$$(d^{0,PQ} + d^{PQ,TX})r \geq \hat{b}_H + (d^{0,PQ} + d^{PQ,TX})\hat{r}_H + \hat{b}_C + d^{0,PQ}\hat{r}_C + \hat{l}_L$$
$$(d^{0,PQ} + d^{PQ,TX})(r - \hat{r}_H) \geq \hat{b}_H + \hat{b}_C + \hat{l}_L + d^{0,PQ}(\hat{r}_H + \hat{r}_C)$$

Since $r \geq \hat{r}_H + \hat{r}_C + r_f$, and $d^{0,PQ} \geq 0$, the term $d^{0,PQ}(\hat{r}_H + \hat{r}_C)$ is non-negative. To find an upper bound on $d^{PQ,TX}$, we can conservatively assume $d^{0,PQ} = 0$ (worst case). This simplifies the inequality to:

$$d^{PQ,TX}(r - \hat{r}_H) \geq \hat{b}_H + \hat{b}_C + \hat{l}_L \tag{7}$$

Therefore, the upper bound on $d^{PQ,TX}$ is:

$$d^{PQ,TX} \leq \frac{\hat{b}_H + \hat{b}_C + \hat{l}_L}{r - \hat{r}_H} \tag{8}$$

Delay $d^{TX,DQ}$

The delay $d^{TX,DQ}$ is the transmission delay of frame f over the link:

$$d^{TX,DQ} = \frac{\hat{l}_f}{r} \tag{9}$$

We assume that the propagation delay is negligible and set $d^{\text{prop}} = 0$.

Upper Bound for Delay $d^{DQ,SO}$

The delay $d^{DQ,SO}$ is the shaping delay introduced by the token bucket at the next node's shaped queue q_I. In the TBE algorithm, a frame may need to wait in q_I until enough tokens have accumulated to allow its transmission.

However, the shaping delay of frame f can be influenced by the shaping delays of other frames in q_I that arrived before f. Specifically, the maximum shaping delay is bounded by the maximum $d_j^{PQ,DQ}$ of any frame in $F_I(f)$, the set of frames sharing the shaped queue with f, including f itself.

Let us denote I as the set of indices of flows in $F_I(f)$. Then, the per-hop delay for frame f is bounded by:

$$d_f^{PQ,SO} \leq \max_{j \in I} \left(\frac{\hat{b}_H + \hat{b}_{C(j)} + \hat{b}_j - \check{l}_j + \hat{l}_L}{r - \hat{r}_H} + \frac{\check{l}_j}{r} \right) \tag{10}$$

This expression matches equation (21) from the UBS paper and provides the upper bound for the total per-hop delay experienced by frame f.

Interpretation of the Delay Components

- **Burst Terms:**
  - $\hat{b}_H$: The total burst size of all higher priority streams. It accounts for the maximum amount of higher priority data that could be sent ahead of frame f, contributing to its delay.
  - $\hat{b}_{C(j)}$: The total burst size of all same priority streams excluding stream j. These streams share the same priority level as f and can cause additional queuing delay in q_S and shaping delay in q_I.
  - $\hat{b}_j - \check{l}_j$: The effective burstiness of stream j, adjusted by subtracting the minimum frame length $\check{l}_j$. This term represents the maximum additional burst that stream j can contribute to the backlog, beyond its minimum frame size.
- **Transmission Delay:** $\frac{\check{l}_j}{r}$ represents the minimum transmission delay of stream j's frame over the link. Using the minimum frame length provides a conservative estimate of the delay.
- **Denominator:** $r - \hat{r}_H$ is the remaining bandwidth after accounting for higher priority streams. This represents the effective bandwidth available to frame f and other same or lower priority streams.
- **Lower Priority Frame in Transmission ($\hat{l}_L$):** The maximum size $\hat{l}_L$ of a lower priority frame that could be in transmission when the busy period starts, potentially delaying frame f.

Total Per-Hop Delay $d^{PQ,SO}$

Thus, the total per-hop delay for frame f is given by equation 10:

$$d_f^{PQ,SO} \leq \max_{j \in I} \left(\frac{\hat{b}_H + \hat{b}_{C(j)} + \hat{b}_j - \check{l}_j + \hat{l}_L}{r - \hat{r}_H} + \frac{\check{l}_j}{r} \right) \tag{11}$$

This expression provides an upper bound on the per-hop delay experienced by frame f, considering the worst-case delays introduced by other frames in the shaped queue q_I.

End-to-End Delay

The end-to-end delay for stream f can be calculated by summing the per-hop delays over all hops in its path:

$$D_{\text{end-to-end}} = \sum_{\text{hops}} d_{(i)}^{PQ,SO} \tag{12}$$

Where $d_{(i)}^{PQ,SO}$ is the total per-hop delay at hop i, calculated using the method above.

Summary

By properly calculating each component of the per-hop delay and using the expression from the UBS paper (equation 10), we obtain an accurate upper bound for the delay experienced by a frame in an ATS network. The total per-hop delay $d^{PQ,SO}$ includes the effects of higher priority traffic, same priority traffic, and the shaping delay at the next node, considering the effective burstiness of interfering streams.

This detailed analysis ensures that we can accurately determine whether a stream meets its deadline constraints.

5 Evaluation

We evaluate the performance of the solution on several Test Cases (TCs). You can use the TCs provided by the course (via DTU Learn) or create your own TCs.

Output format: We expect that the solution generates an output file. You can use the suggested format or develop your own format. The generated output file should contain the following information.

- Runtime for generating the (analysis) solution
- Mean E2E delay of the solution
- Maximum E2E delay for each flow

The runtime for generating the solution is in seconds and represents the time takes for generating the solution. The mean E2E delay of the solution shows the mean E2E delay of all streams. To calculate the mean E2E delay of the solution, one should first calculate the mean E2E delay of each stream and then calculate the mean value of these delays.

References

[1] Aeronautical Radio, Inc. *ARINC 664P7: Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*. 2009.

[2] Marc Boyer. Equivalence between the urgency based shaper and asynchronous traffic shaping in time sensitive networking. 2024.

[3] J. D. Decotignie. Ethernet-based real-time and industrial communications. *Proceedings of the IEEE*, 93(6):1102–1117, 2005.

[4] Joachim Feld. Profinet—scalable factory communication for all applications. In *Factory Communication Systems, 2004. Proceedings. 2004 IEEE International Workshop on*, pages 33–38. IEEE, 2004.

[5] IEEE. *802.3 Standard for Ethernet*, 2015.

[6] IEEE. Official Website of the 802.1 Time-Sensitive Networking Task Group. `http://www.ieee802.org/1/pages/tsn.html`, 2016.

[7] IEC/IEEE 60802 TSN profile for industrial automation (Draft version 2.0). Standard, IEEE, 4 2023.

[8] IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications. Standard, IEEE, 3 2011.

[9] IEEE Standard for Local and metropolitan area networks–Virtual Bridged Local Area Networks Amendment 14: Stream Reservation Protocol (SRP). Standard, IEEE, 9 2010.

[10] IEEE Standard for Local and metropolitan area networks– Virtual Bridged Local Area Networks Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams. Standard, IEEE, 1 2010.

[11] IEEE Standard for Local and Metropolitan Area Networks–Audio Video Bridging (AVB) Systems. Standard, IEEE, 9 2011.

[12] IEEE Standard for Local and Metropolitan Area Networks–Bridges and Bridged Networks - Amendment 34:Asynchronous Traffic Shaping. Standard, IEEE, 11 2020.

[13] Dirk Jansen and Holger Buttner. Real-time ethernet: the ethercat solution. *Computing and Control Engineering*, 15(1):16–21, 2004.

[14] Oliver Kleinberg and Alex Schneider. *Time sensitive Networking for dummies*. Belden, 2018.

[15] AS Oliver Kleineberg and Axel Schneider. Time-sensitive networking for dummies, 2018.

[16] John Rushby. Bus architectures for safety-critical embedded systems. In *Embedded Software*, pages 306–323. Springer, 2001.

[17] SAE. *AS6802: Time-Triggered Ethernet*. SAE International, 2011.

[18] Johannes Specht and Soheil Samii. Urgency-based scheduler for time-sensitive switched ethernet networks. *Euromicro Conference on Real-Time Systems*, 28:75–85, 2016.

[19] Michael D. Johas Teener, Andre N. Fredett, Christian Boiger, Philippe Klein, Craig Gunther, David Olsen, and Kevin Stanton. Heterogeneous networks for audio and video: Using ieee 802.1 audio video bridging. *Proceedings of the IEEE*, 101(11):2339–2354, 2013.

A Appendix: OMNeT++ Guide

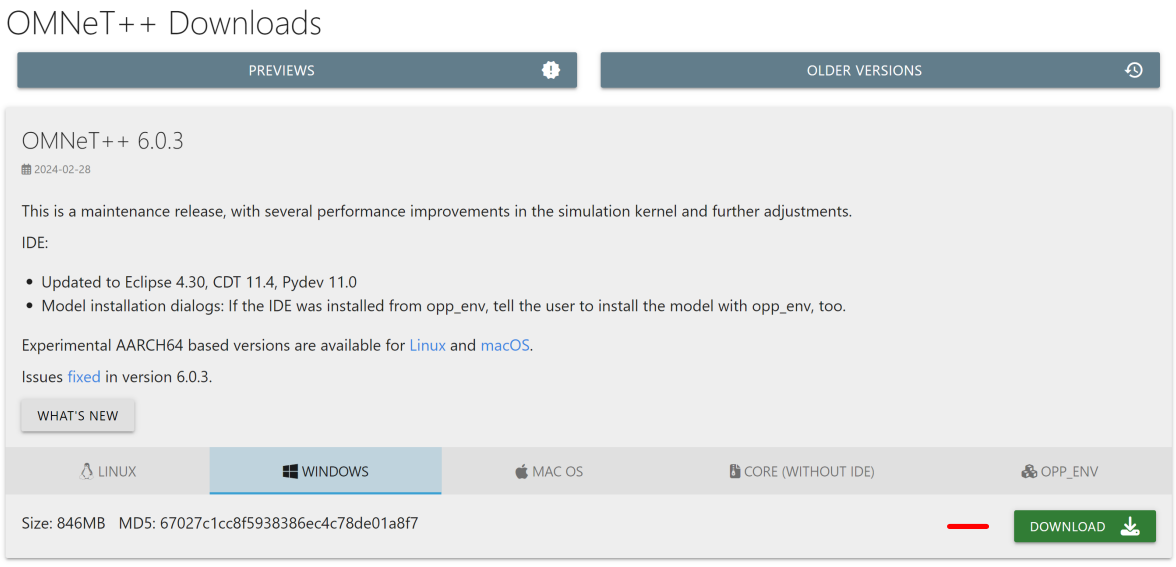
This guide covers the following:

- OMNeT++ Installation
  - Installing OMNeT++ on Windows
    - Downloading OMNeT++
    - Unpacking OMNeT++
    - Compiling and Installing OMNeT++
    - Launching OMNeT++ IDE
  - Installing OMNeT++ on Linux
  - Installing OMNeT++ on MAC OS
- First Project: Designing and Simulating a Small TSN Network
  - Introduction
  - Creating Workspace
  - Designing a Network
  - Configuring Simulation
  - Analyzing Simulation Results

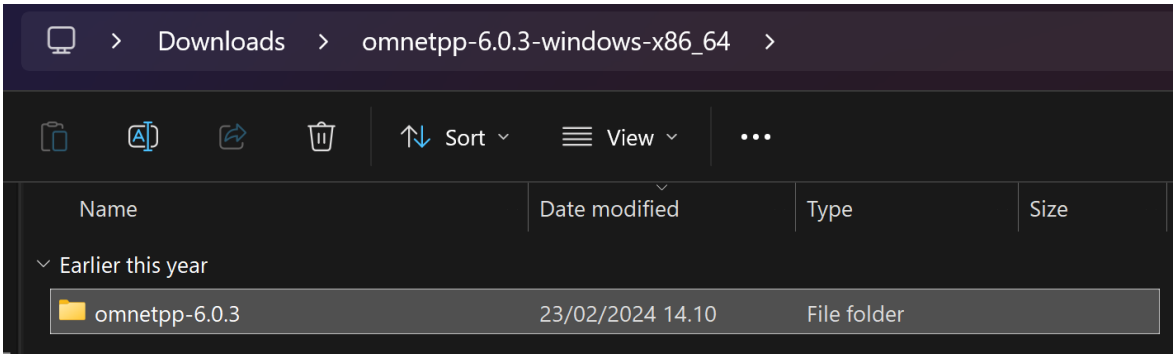
A.1 OMNeT++ Installation

A.1.1 Installing OMNeT++ on Windows

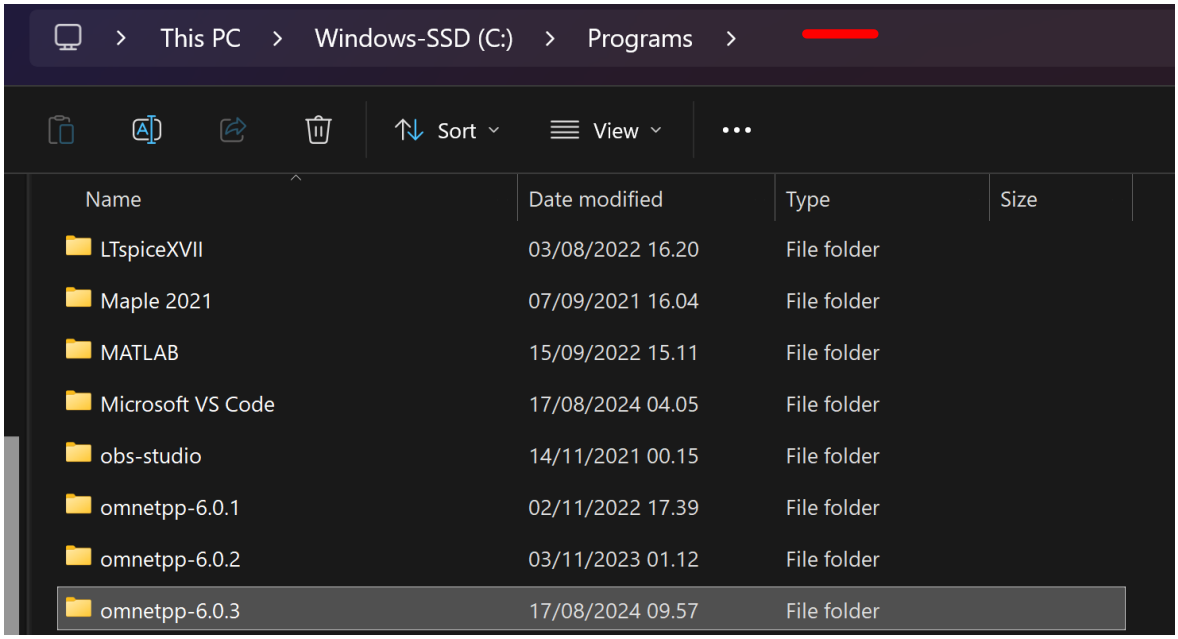
Downloading OMNeT++ OMNeT++ can be downloaded from the official site at `https://omnetpp.org/download/`



Unpacking OMNeT++ Inside the archive navigate one level down and find folder with OMNeT++ source files.



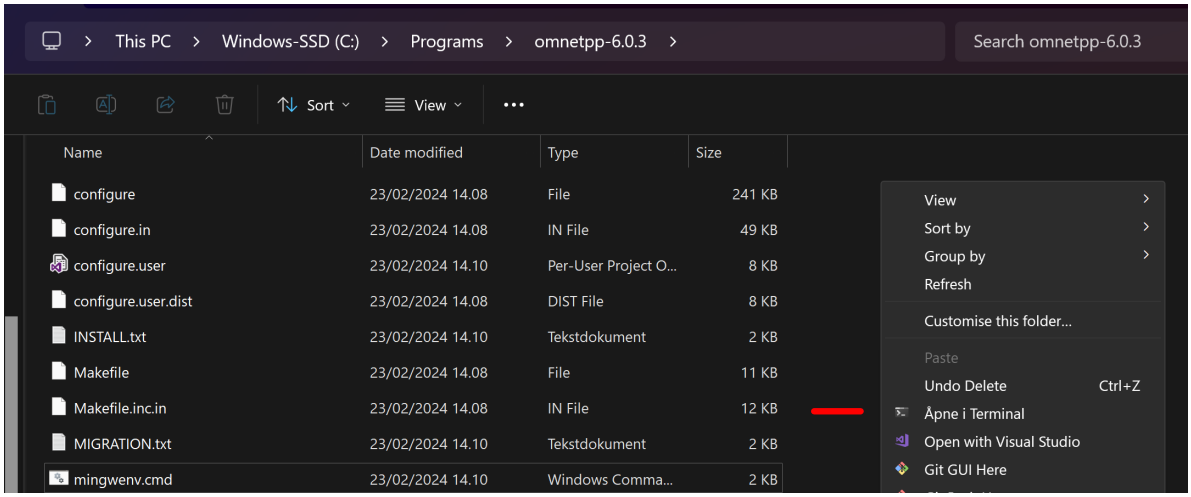
Use 7-Zip to extract the folder to your desired OMNeT++ installation location, such as C:/Programs/. While Windows' built-in extraction tool can open archived source files, using 7-zip is significantly faster. **Important**: On Windows, choose a directory whose full path **does not contain any spaces**; for example, do **not** put OMNeT++ under Program Files



Compiling and Installing OMNeT++ The steps below provide a quick installation guide using default settings. For a comprehensive installation guide, refer to:

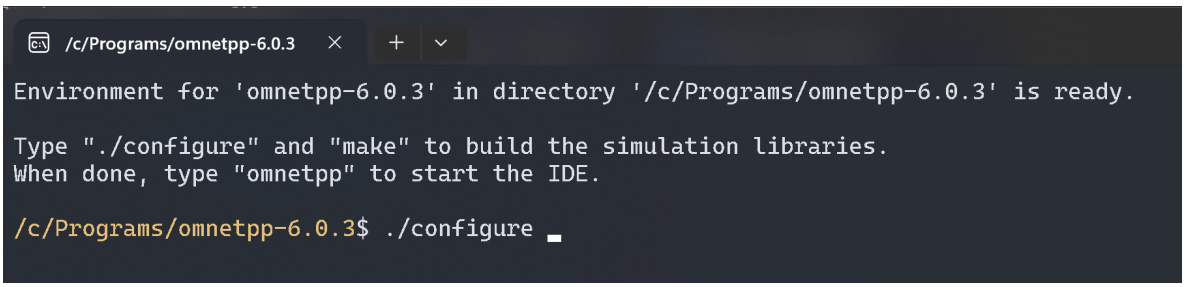
`C:/Programs/omnetpp-6.0.3/doc/InstallGuide.pdf`

Open a terminal in your chosen installation folder

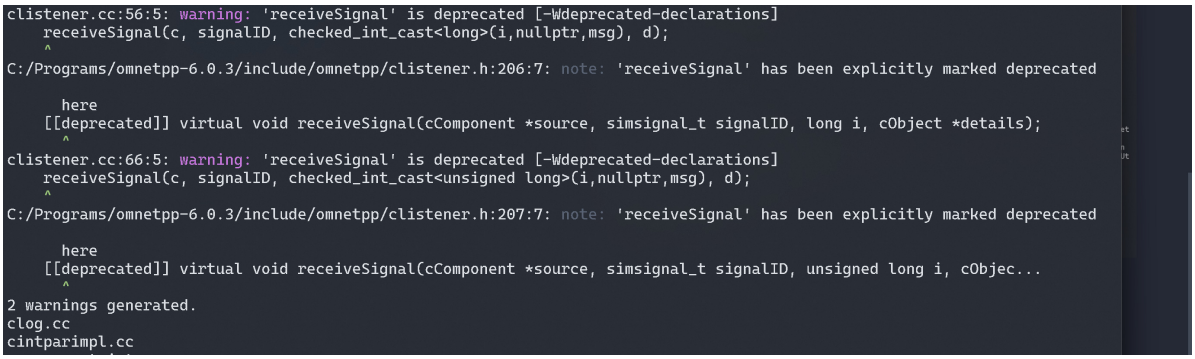


Execute the `mingwenv.cmd` script to set up the MinGW environment. This process typically takes 1-3 minutes.

After the process completes, a command prompt window will open. In this window, type: `./configure`



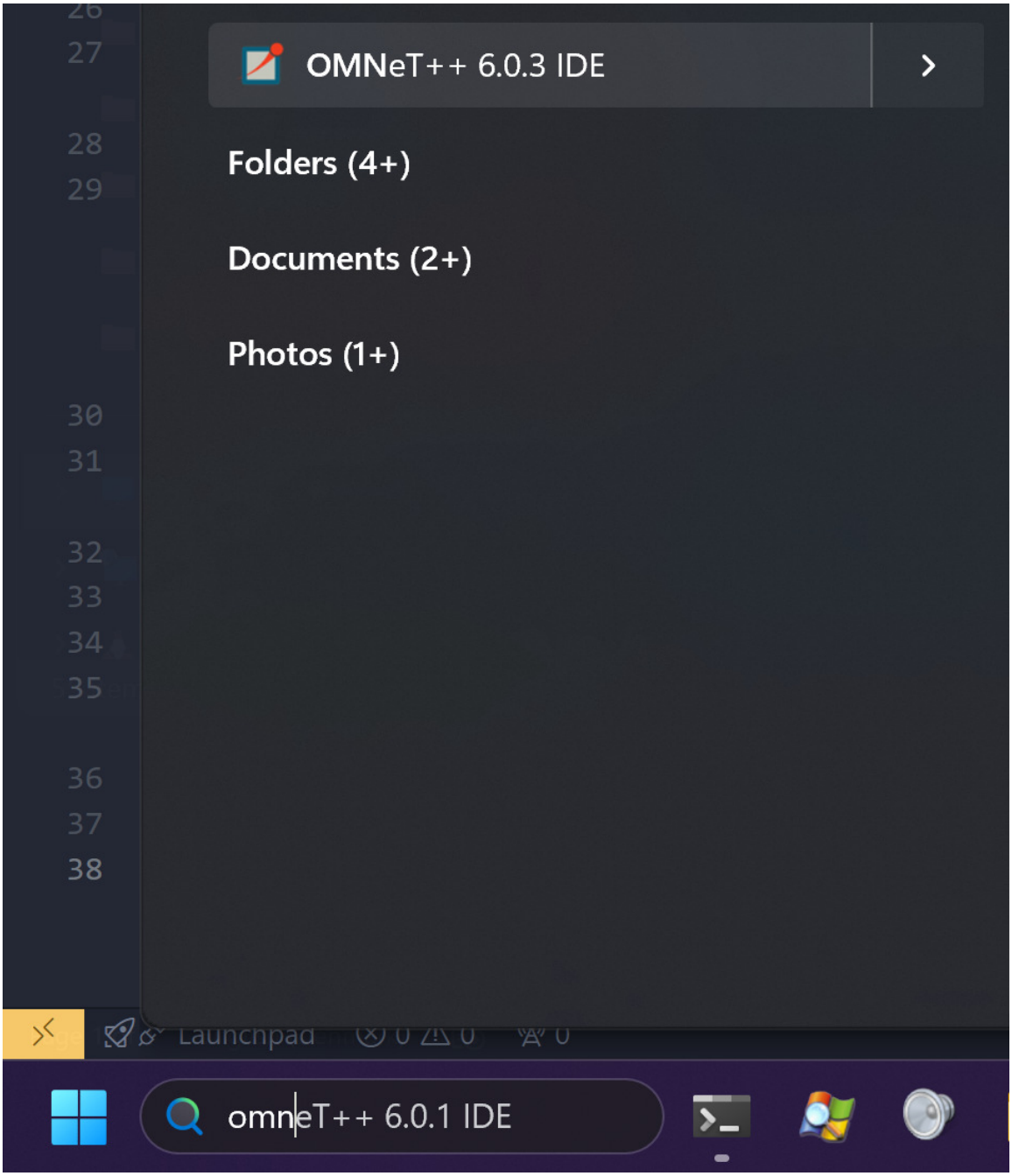
After the process completes, a command prompt window will open. In this window, type: `make`. This command initiates the build process, compiling and installing OMNeT++ from the source files. The process may take up to **60 minutes to complete**. For reference: A system with a Ryzen 7 5800H processor completed this step in 30 minutes. You may see warnings during the build process. These are normal and can be safely ignored.



Launching OMNeT++ IDE After installation, an OMNeT++ IDE shortcut will be added to your Start menu. To quickly access OMNeT++, copy the Start menu shortcut to your desktop and use it to launch the IDE by following

these steps:

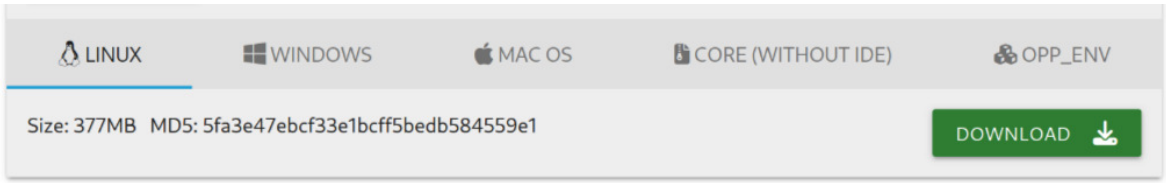
- Use Windows search bar to find OMNeT++ IDE



-
- Right click the shortcut and choose ”Open file location”
- Copy the shortcut to your desktop

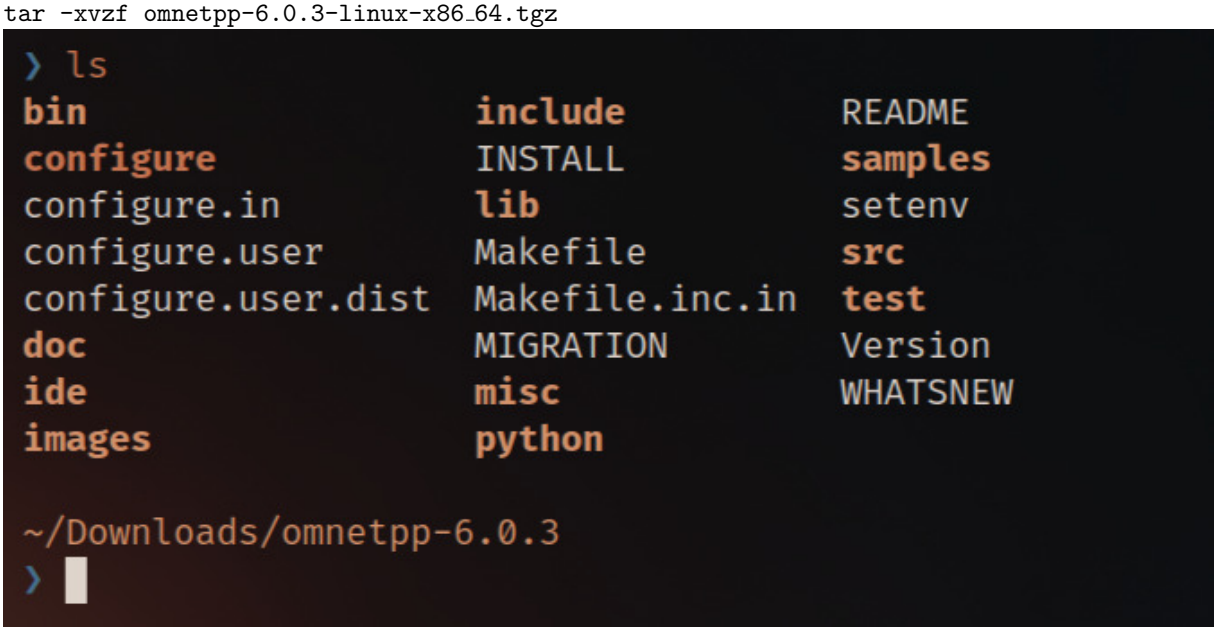
A.1.2 Installing OMNeT++ on Linux

Downloading OMNeT++ OMNeT++ can be downloaded from the official site at `https://omnetpp.org/download/`



Unpacking OMNeT++ To be able to unpack the archive use this command:

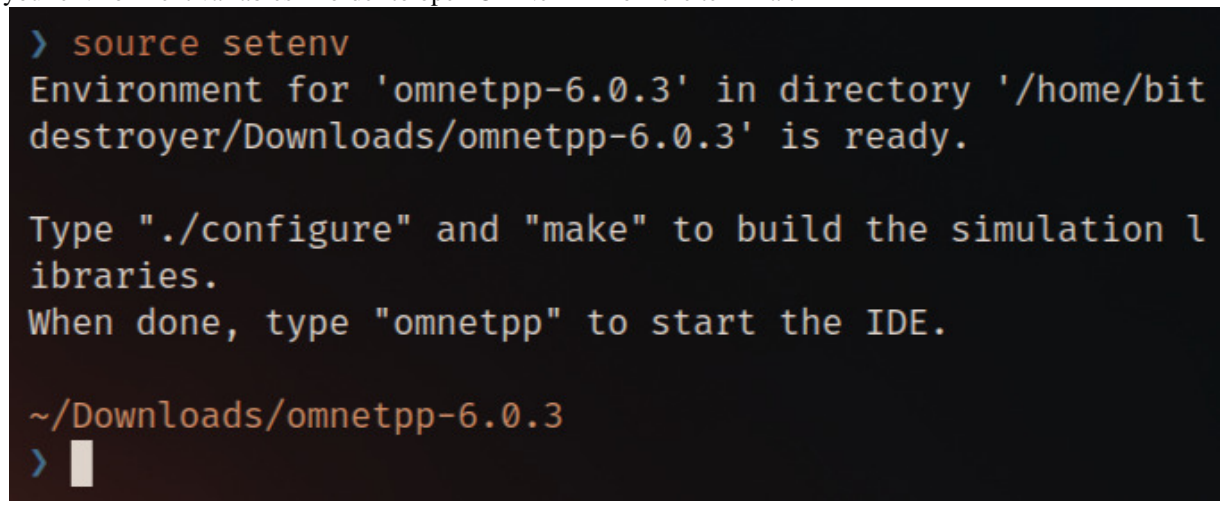
`tar -xvzf omnetpp-6.0.3-linux-x86_64.tgz`



Compiling and Installing OMNeT++ The steps below provide a quick installation guide using default settings and with the assumption that the running distribution is Ubuntu. For a comprehensive installation guide, refer to:

`/omnetpp-6.0.3/doc/InstallGuide.pdf`

Open a terminal in your chosen installation folder then execute the `source setenv` command in order to set up your environment variables in order to open OMNeT++ from the terminal.



First you need to install a lot of dependencies using the following commands in order:

```
sudo apt-get update
sudo apt-get install build-essential clang lld gdb bison flex perl  python3 python3-pip
qtbase5-dev qtchooser qt5-qmake qtbase5-dev-tools  libqt5opengl5-dev libxml2-dev zlib1g-dev
doxygen graphviz  libwebkit2gtk-4.0-37 xdg-utils
sudo apt-get install openscenegraph-plugin-osgearth libosgearth-dev
```

After that you need to install the following Python modules:

- scipy
- pandas
- matplotlib
- posix_ipc

The command to install them system-wide is: `python3 -m pip install scipy pandas matplotlib posix_ipc`

After the process completes type: `./configure`

There is a possibility of the command failing meaning that probably some of the dependencies were not properly installed or were missing. In this case try searching for the error on Google and installing them manually until the command completes successfully.

```
checking for icpc... no
checking for g++... g++
checking whether the compiler supports GNU C++... yes
checking whether g++ accepts -g... yes
checking for g++ option to enable C++11 features... none needed
checking for g++... g++
checking for c++14 support... yes
checking for ranlib... ranlib
checking whether LLD linker is available... no
checking whether g++ supports -fno-omit-frame-pointer... yes
checking whether g++ supports -glldb... no
checking whether g++ supports -ggdb3... yes
checking whether g++ supports -fstandalone-debug... no
checking whether g++ supports -fno-limit-debug-info... no
checking whether g++ supports -Wl,--no-as-needed... yes
checking whether g++ supports -Wl,--as-needed... yes
checking for swapcontext... yes
checking if shared libs need -fPIC... yes
checking for dlopen with CFLAGS="" LIBS=""... yes
checking if --export-dynamic linker option is supported/needed... both
checking for flags needed to link with static libs containing simple modules... --w
hole-archive
configure: NOTE: Use the following syntax when linking with static libraries
configure: containing simple modules and other dynamically registered components:
configure:    g++ ... -Wl,--whole-archive <libs> -Wl,--no-whole-archive ...
checking whether linker supports -rpath... yes
checking for bison... bison -y
checking for make... make
checking for perl... perl
checking for swig... not found
checking for python3... python3
checking if all necessary Python modules are available... yes
checking for math with CFLAGS=" -fPIC" LIBS=""... yes
checking for standard C++ lib with CFLAGS=" -fPIC" LIBS="-lstdc++"... yes
checking for dlopen with CFLAGS=" -fPIC" LIBS=""... yes
checking for qmake-qt5... /usr/bin/qmake-qt5
checking for moc-qt5... /usr/bin/moc-qt5
checking for uic-qt5... /usr/bin/uic-qt5
checking for rcc-qt5... /usr/bin/rcc-qt5
checking for Qt5 with CFLAGS="  -fPIC -I /usr/include/qt  -fPIC -I /usr/include/qt"
 LIBS="-lQt5Gui -lQt5Core -lQt5Widgets -lQt5PrintSupport -lQt5OpenGL -L/usr/lib"...
 yes
checking for OpenSceneGraph with CFLAGS=" -fPIC  -fno-omit-frame-pointer  " LIBS="
-Wl,-rpath,$(OMNETPP_LIB_DIR) -Wl,-rpath,. -Wl,-rpath,$(OMNETPP_TOOLS_DIR)/lib  -Wl
,--export-dynamic  -losg -losgDB -losgGA -losgViewer -losgUtil -lOpenThreads"... ye
s
checking for g++ option to support OpenMP... -fopenmp
checking for PTHREAD with CFLAGS=" -fPIC " LIBS="-lpthread"... yes
configure: creating ./config.status
config.status: creating Makefile.inc
config.status: creating src/qtenv/qtenv.pri

Configuration phase finished. Use 'make' to build OMNeT++.
```

After the process completes you can safely type: `make`. This command initiates the build process, compiling and installing OMNeT++ from the source files. The process may take up to **60 minutes to complete**. For reference: A system with a Ryzen 7 5800H processor completed this step in 30 minutes. You may see warnings during the build process. These are normal and can be safely ignored.

To run OMNeT++ just go into the bin folder and then type `./omnetpp` in the terminal window.

A.1.3 Installing OMNeT++ on macOS

If you run into problems compiling OMNeT++ on macOS, consider using `opp_env` and Nix as an alternative. `opp_env` is an automated tool for installing OMNeT++ simulation frameworks and models, while Nix is a powerful package manager ensuring consistent and reproducible builds.

1. Install Nix:

```
$ sh <(curl -L https://nixos.org/nix/install)
```

2. Install `opp_env`:

```
$ pip3 install opp-env
```

3. Create an `opp_env` workspace:

```
$ mkdir workspace && cd workspace && opp_env init
```

4. Install OMNeT++ and INET:

```
$ opp_env install omnetpp-6.0.3
$ opp_env install inet-4.5.2
```

5. Start an `opp_env` shell:

```
$ opp_env shell inet-latest
```

Note: running "opp_env shell omnetpp-6.0.3" sets up the environment for OMNeT++ version 6.0.3 only.

6. Launch OMNeT++ IDE:

```
$ omnetpp
```

Note: OMNeT++ must be started through the Nix environment using the `opp_env shell` command.
For more information, visit:

- `opp_env`: `https://github.com/omnetpp/opp_env`
- Nix: `https://nixos.org/download/`

A.2 First Project: Designing and Simulating a Small TSN Network

A.2.1 Introduction

The primary objective of this project is to familiarize oneself with OMNeT++ and gain insight into the simulation and analysis of network traffic. Through this endeavor, an understanding of how network scenarios can be modeled, simulated, and evaluated using OMNeT++ is intended to be developed. The project aims to provide hands-on experience with creating network topologies, configuring network elements, and analyzing the resultant traffic patterns.

Following the project objectives, it should be understood that an OMNeT++ simulation is comprised of two primary parts:

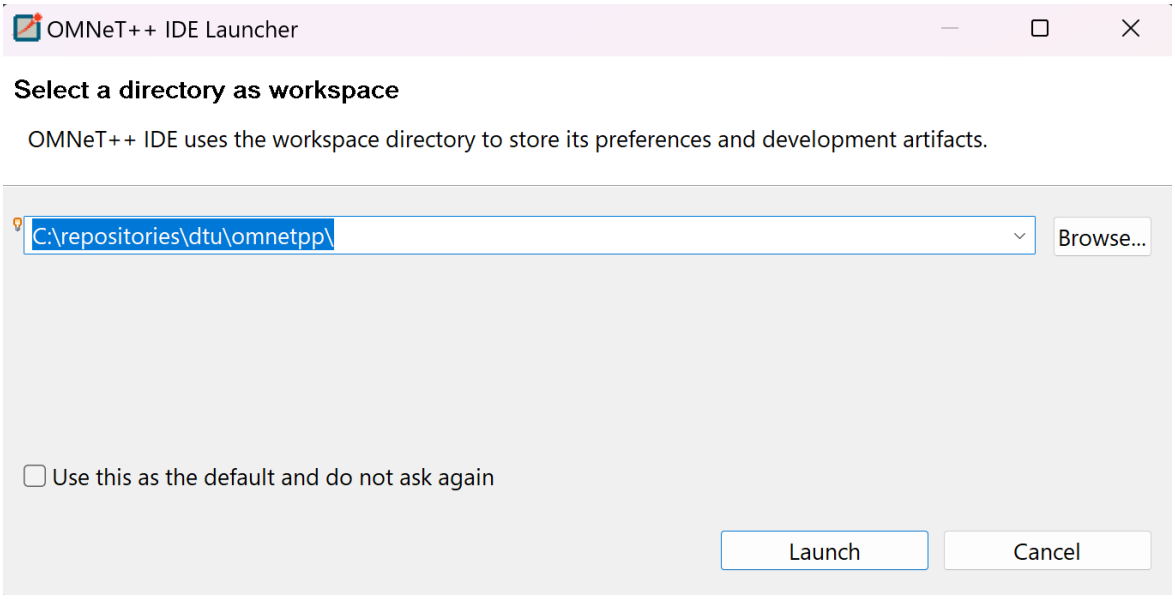
1. **Network Description (NED) file**: Expresses the arrangement and interconnection (topology) of network components, primarily switches and nodes. These network components are modular and self-contained, hence are referred to as network modules.

2. **Simulation Configuration (INI) file**: Configures the various network modules present in the NED file. The ini file also specifies traffic patterns and other attributes of the simulation, such as the traffic flow visualization.

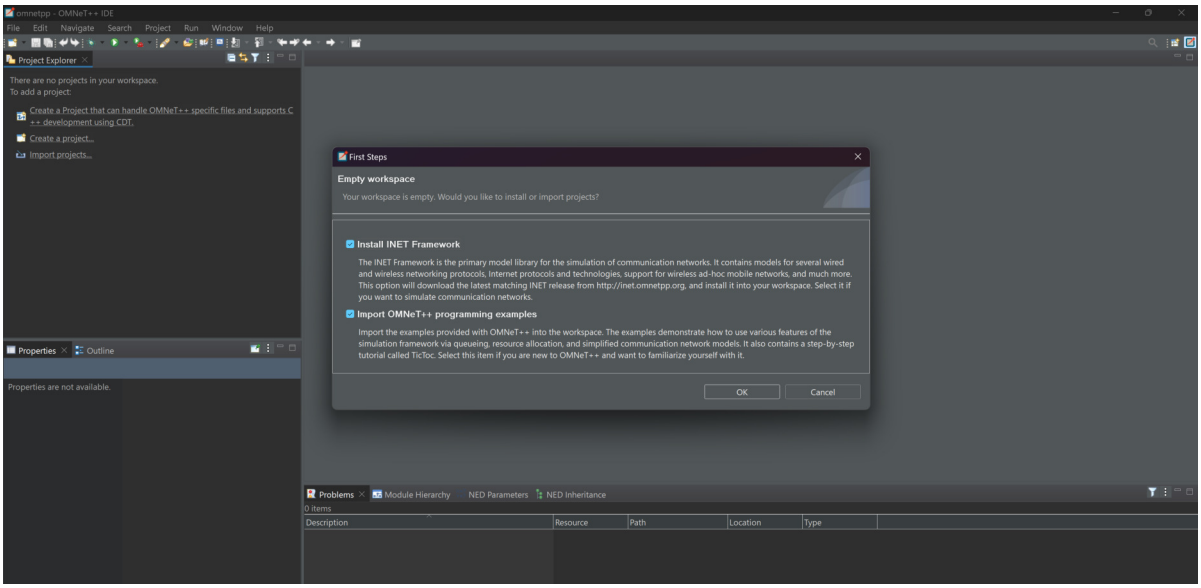
These two components work in tandem to create a simulation environment. The NED file provides the structure and layout of the network, while the configuration file determines how the network behaves during the simulation.

A.2.2 Creating Workspace

An OMNeT++ workspace contains all necessary files and libraries to design and simulate a network. Upon launching OMNeT++, a prompt will be given to choose a location for your workspace. A location and name for your workspace directory should be chosen. For example, the following might be used:
`C:/repositories/dtu/omnetpp/`

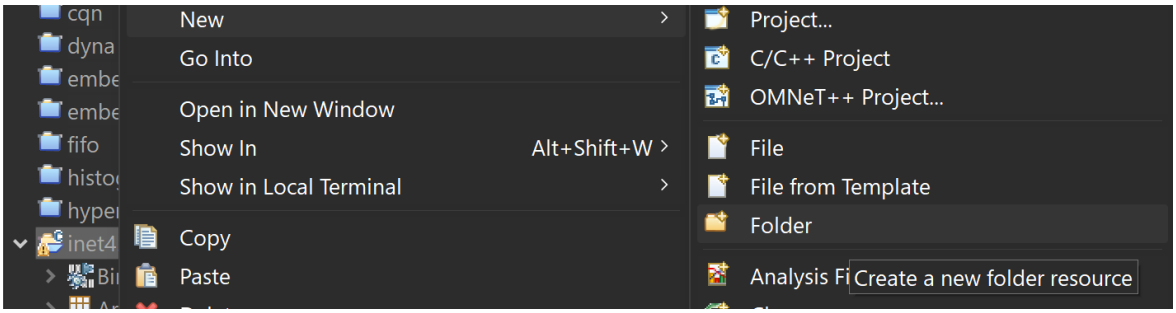


Upon the first launch of OMNeT++, the welcome window should be closed to reveal the main interface. The following window will be displayed for an empty workspace:

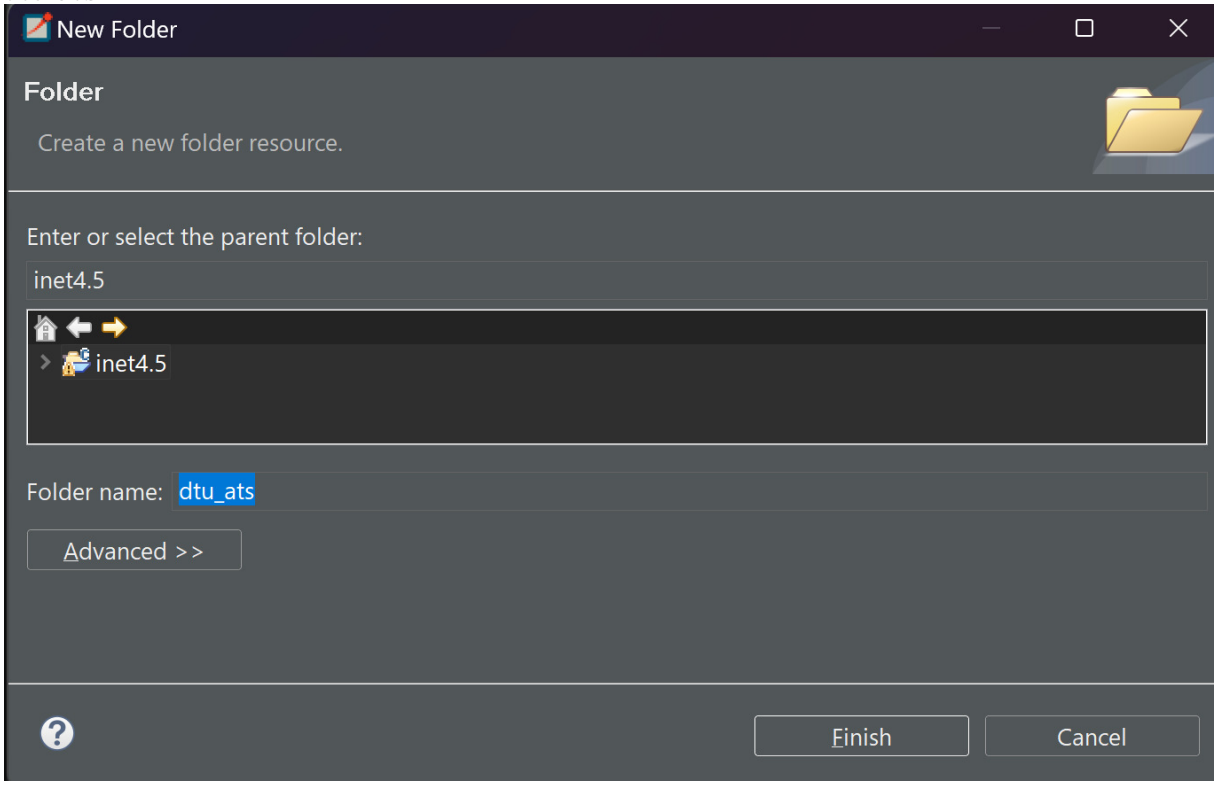


Both boxes should be checked, and OK should be clicked. This action initializes the INET Framework, which contains necessary models. The INET Framework contains switch and node TSN models that will be utilized in this project; therefore, it must be included in the workspace. It should be noted that this process may take up to **10 minutes** to complete.

Within the inet4.5 folder, a new folder for the project should be created. This newly created folder will serve as the container for the simulation configuration file for the project. RECAP: The simulation configuration file, which will be placed in this folder, will be used to specify the various parameters and settings for the network simulation.



A descriptive name should be given to the folder. For example, it could be named:
`dtu_ats`



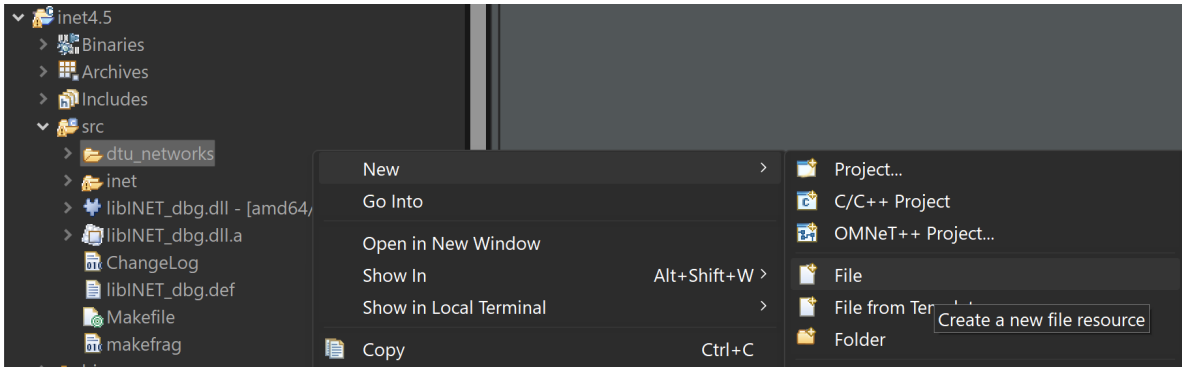
A.2.3 Designing a Network

Within the `inet4.5/src` folder, a new folder should be created for the network that is about to be designed. A descriptive name should be given to this folder, such as:

`dtu_networks`

Subsequently, within the newly created `dtu_networks` folder, a new **NE**twork **D**iscription (NED) file should be created. This file should be named:

`ring_3sw_network.ned`



NED file defines the network and all the modules contained within it. Let's now design our first network.

1. Begin the file with a package declaration. The package name should match your folder structure. For instance, if your NED file is in `src/dtu_networks/`, your package declaration should be `package dtu_networks;`

   ```
   package dtu_networks;
   ```

   The package declaration serves several important purposes in OMNeT++:

   - Namespace Management: It creates a unique namespace for your network components, helping to avoid naming conflicts with other modules in larger projects.
   - Module Organization: It reflects the folder structure of your project, making it easier to organize and locate files.
   - Import Simplification: Other NED files in the same package can refer to this network without needing to use the full package name.
   - Build System Integration: The OMNeT++ build system uses this information to correctly compile and link your modules.

2. Next, import the modules the network will contain. In total, 4 modules must be imported:

   ```
   import inet.networks.base.TsnNetworkBase;
   import inet.node.ethernet.Eth1G;
   import inet.node.tsn.TsnDevice;
   import inet.node.tsn.TsnSwitch;
   ```

   Let's break down each import and explain its purpose:

   (a) **TsnNetworkBase**:
   - This is the base class for Time-Sensitive Networking (TSN) networks.
   - It provides fundamental TSN functionalities and configurations.
   - Our custom network will extend this base class to inherit its properties.

   (b) **Eth1G**:
   - This represents a 1 Gigabit Ethernet connection.
   - It will be used to define the links between network devices.

   (c) **TsnDevice**:
   - This module represents a TSN-capable end device or node in the network.
   - It can be used to model sensors, actuators, or other endpoints in a TSN system.

   (d) **TsnSwitch**:
   - This module represents a TSN-capable switch.
   - It's used to model the switching elements in our TSN network topology.

By importing these modules, we gain access to their functionalities and can use them to construct our custom TSN network topology in the subsequent steps.

3. Next define the network that extends `TsnNetworkBase` as follows

```
network ring_3sw_network extends TsnNetworkBase {

}
```

It's important to note that OMNeT++ is highly hierarchical in nature. Most modules, including networks, inherit from multiple levels of modules using the `extends` keyword. In this case, our `ring_3sw_network` extends from `TsnNetworkBase`, thereby inheriting its properties and default configurations.

This hierarchical structure allows for:

- Reusability of code and configurations
- Easier management of complex network structures
- Gradual specialization of modules from general to specific

For instance, `TsnNetworkBase` itself extend from a more generic network base class, creating a chain of inheritance:

```
cModule -> NetworkBase -> TsnNetworkBase -> ring_3sw_network
```

Each level in this hierarchy adds or refines properties and behaviors specific to its purpose, allowing you to create sophisticated network models while maintaining a clear and organized structure.

4. NED files contain both textual definitions and visual representations. To define the visual container for your network, set the canvas size using the `@display()` as follows

```
network ring_3sw_network extends TsnNetworkBase
{
    @display("bgb=1500,1000");
}
```

5. After defining the network module, we proceed to include instances of INET nodes and switches as submodules. Here's how to define these submodules::

```
network ring_3sw_network extends TsnNetworkBase {
    @display("bgb=1500,1000");
    submodules:
        sw_0: TsnSwitch {
            @display("p=729,444");
        }
        node_0_0: TsnDevice {
            @display("p=633,344");
        }
}
```

Let's break down the submodule definition:

(a) **Submodule Declaration**:

- Each submodule is given a unique name (e.g., `sw_0`, `node_0_0`).
- The type of each submodule is specified (e.g., `TsnSwitch`, `TsnDevice`).

(b) **Submodule Positioning**:

- The `@display` property is used to set the position of each submodule on the network canvas.
- The syntax `"p=x,y"` determines the x and y coordinates of the submodule.
- For example, `@display("p=729,444")` positions the switch at coordinates (729, 444).

(c) **Importance of Positioning**:

- Proper positioning enhances the visual representation of your network.
- It helps in understanding the network topology at a glance.
- Well-organized layouts can make complex networks easier to comprehend and debug.
- The positions affect only the visual representation and do not impact the network's logical structure or performance.

(d) **Naming Convention**:

- In this example, we use `sw_0` for the switch and `node_0_0` for the node.
- This naming scheme can be extended for larger networks (e.g., `sw_1`, `sw_2`, `node_1_0`, etc.).
- Consistent naming helps in organizing and referencing submodules in larger network configurations.

Note: You can adjust the positions of submodules later in the OMNeT++ IDE's graphical editor for fine-tuning the network layout.

6. After defining the submodules, the next step is to connect them using a medium. Here's how to establish connections between submodules:

```
package dtu_networks;

import inet.networks.base.TsnNetworkBase;
import inet.node.ethernet.Eth1G;
import inet.node.tsn.TsnDevice;
import inet.node.tsn.TsnSwitch;

network ring_3sw_network extends TsnNetworkBase {
 @display("bgb=1500,1000");
 submodules:
     sw_0: TsnSwitch {
         @display("p=729,444");
     }
     node_0_0: TsnDevice {
         @display("p=633,344");
     }
 connections:
     sw_0.ethg++ <--> Eth1G <--> node_0_0.ethg++;
 }
```

Let's break down the connection syntax and its components:

(a) **Connection Declaration**:

- Connections are declared within the `connections:` block of the network.
- Each connection defines how two submodules are linked together.

(b) **Connection Syntax**:

- The basic syntax is: `submodule1.gate <--> medium <--> submodule2.gate`
- `sw_0` and `node_0_0` are the submodule names we defined earlier.
- `.ethg++` refers to the Ethernet gate of the submodule.
- `<-->` represents a bidirectional connection.
- `Eth1G` is the medium (1 Gigabit Ethernet) through which the submodules are connected.

(c) **Gate Specification**:
- `ethg` stands for Ethernet gate.
- The `++` notation is used for vector gates, allowing multiple connections to the same gate.
- OMNeT++ automatically assigns the next available gate index when using `++`.

(d) **Medium Specification**:
- `Eth1G` specifies a 1 Gigabit Ethernet connection between the submodules.
- This medium was imported in step 2 (`import inet.node.ethernet.Eth1G;`).
- Different media can be used based on the network requirements (e.g., `Eth100M` for 100 Mbps Ethernet).

(e) **Importance of Proper Connections**:
- Correct connections ensure that data can flow between submodules as intended.
- They define the network topology and affect how packets are routed in the simulation.
- Misconfigurations in connections can lead to unrealistic network behavior or simulation errors.

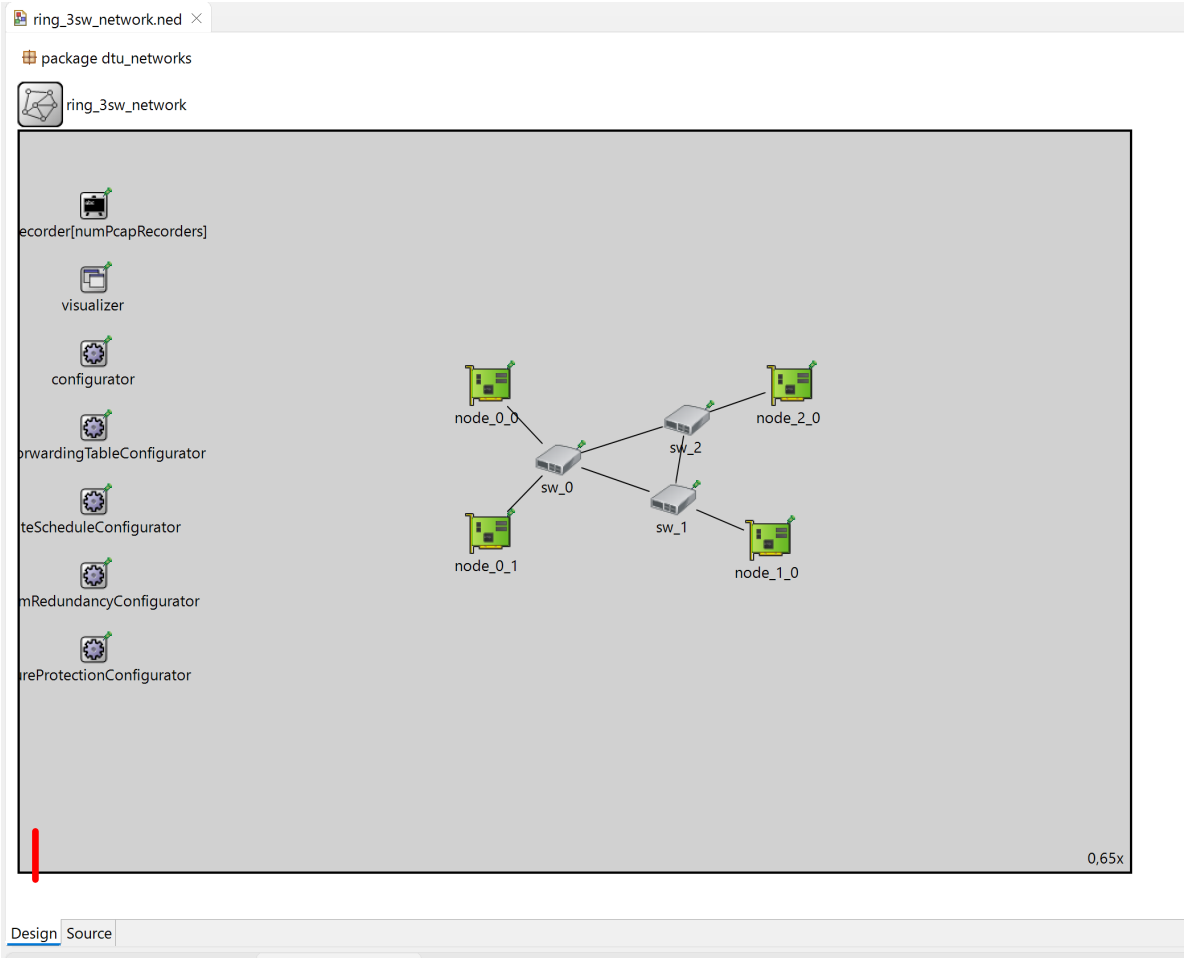
Note: As you add more submodules to your network, you'll need to define additional connections to link them together appropriately.

7. To complete our network design, we'll expand our previous example to create a 3-switch ring network. Follow these steps:

(a) **Expand the Submodules**: Add two more switches and two more nodes to create a 3-switch ring topology.

(b) **Add Connections**: Connect the new submodules to form a ring.

(c) **Adjust Positioning**: Position the submodules to create a visually clear ring layout.



To view and interact with your network in the OMNeT++ IDE:

(a) **Open the NED file**: Double-click on your `ring_3sw_network.ned` file in the Project Explorer.

(b) **Switch to Design view**: Click on the 'Design' tab at the bottom of the editor window. This will show you a graphical representation of your network.

(c) **Adjust submodule positions**:
- Click and drag submodules to reposition them within the Design view.
- As you move submodules, their `@display` coordinates in the source code will automatically update.

(d) **Examine connections**:
- The lines between submodules represent the connections you defined.
- Hover over a connection to see details about the link (e.g., connection type).

(e) **Explore submodule properties**:
- Double-click on a submodule to open its property sheet.
- Here you can view and modify various parameters of the submodule.

(f) **Validate the network**:
- OMNeT++ will automatically check for errors in your network definition.
- Any issues will be highlighted in the Problems view at the bottom of the IDE.

Note: The Design view provides a way to visualize and interact with your network topology. It's useful for complex networks where manual positioning in code might be cumbersome.

A.2.4 Configuring the Simulation

In this section, we'll configure our simulation to generate and manage three types of TSN traffic using the Asynchronous Traffic Shaper (ATS). We'll build our configuration step-by-step, referring to the provided `3sw_ats.ini` file and the node and switch model diagrams (corresponding to the `3sw_network.ned` file).

Before we delve into the specifics of configuring a simulation using an INI file in OMNeT++, it's crucial to understand two fundamental aspects of TSN networks:

- How traffic is categorized in a TSN network
- The node and switch model structure that we will be configuring

Traffic categorization in TSN Table 1 presents an overview of the eight traffic types in Time-Sensitive Networking (TSN), each characterized by several distinct attributes. In TSN, traffic flows are categorized into these 8 distinct classes, each associated with a specific Priority Code Point (PCP). The PCP, a 3-bit field in the IEEE 802.1Q tag of Ethernet frames, assigns a priority level to each traffic type, ranging from 0 (lowest) to 7 (highest). The table also shows the cycle time, which indicates the typical frequency or pattern of data transmission for each type. Packet sizes, representing the payload or data portion of the Ethernet frame, vary across traffic types and can be fixed or variable. The criticality column denotes the relative importance of each traffic type within the network, while the data delivery guarantee specifies the primary performance metric for each traffic type, such as bandwidth, latency, or deadline adherence. It's worth noting that as packets move through the network, various headers (including the 802.1Q tag containing the PCP) are added and removed at different layers of the network stack, which affects the total frame size beyond the payload sizes listed in the table.

Traffic Type	PCP	Cycle Time	Packet Size (bytes)	Criticality	Data Delivery Guarantee
Network Control	7	50ms - 1s	50 - 500 (variable)	High	Bandwidth (1-2 Mbps)
Isochronous	6	$<$ 2ms	30 - 100 (fixed)	High	Deadline
Cyclic	5	2μs - 20ms	50 - 1000 (fixed)	High	Latency (100μs - 2ms)
Events - Control	4	Sporadic	100 - 200 (variable)	High	Latency (10-50ms)
Events - Alarms & Commands	3	Sporadic	100 - 1500 (variable)	Medium	Latency ($\sim$2s)
Configuration & Diagnostics	2	Sporadic	500 - 1500 (variable)	Medium	Latency ($\sim$100ms)
Video, Audio, Voice	1	Frame/Rate	1000 - 1500 (variable)	Low	Bandwidth
Best Effort	0	Sporadic	30 - 1500 (variable)	Low	None

Table 1: TSN Traffic Types

To fully understand the role of PCP in TSN networks, let's examine its key aspects and functions:

- **Purpose**: PCP determines the priority of a frame as it travels through the network, influencing how switches and other network devices handle the frame.
- **Prioritization**: Higher PCP values generally indicate higher priority. For example, PCP 7 (Network Control) has the highest priority, while PCP 0 (Best Effort) has the lowest.
- **Queue Assignment**: In TSN switches, frames are assigned to specific queues based on their PCP value. Higher priority queues are serviced more frequently and with stricter timing guarantees.
- **Traffic Shaping**: The PCP influences traffic shaping mechanisms, like the ATS.
  - **ATS Configuration**: Each PCP is associated with a specific ATS queue, each with its own shaping parameters (e.g., committed information rate, eligibility time).
  - **Bandwidth Allocation**: Higher PCP values often correspond to ATS queues with higher committed information rates, ensuring more bandwidth for critical traffic.
  - **Latency Control**: ATS uses PCP to determine how aggressively to shape traffic. Higher priority flows (higher PCP) should have shorter eligibility times, reducing their latency through the network.
  - **Burst Control**: PCP can influence the maximum burst size allowed for each traffic class, helping to prevent congestion from high-priority bursty traffic.

Node and switch model structure The network traffic is produced and consumed by nodes (end-points), while switches facilitate its transmission through the network. To simulate TSN-capable nodes and switches in OMNeT++, we imported specific modules from the INET framework using the commands:

```
import inet.node.tsn.TsnSwitch;
import inet.node.tsn.TsnDevice;
```

These imported modules define the TSN-capable node (`TsnDevice`) and switch (`TsnSwitch`) models that we will configure and use in our simulation. Figures 5 and 6 illustrate the internal structures of these INET models, demonstrating their layered architectures. These models provide detailed representations of how TSN nodes and switches operate within the simulation environment and serve as the basis for our network configurations.

As in all layered network models, traffic is always produced and consumed in the application layer of the node. The process of sending data through the network involves the following steps:

1. The traffic stream payload originates in the application layer.
2. It travels down through the layers of the node.
3. At each layer, a new header and/or tail is prepended or appended to the payload.
4. This process continues until the fully encapsulated packet reaches the physical layer for transmission.

For inbound traffic, the process is reversed:

1. The packet is received at the physical layer.
2. It moves up through the layers of the node.
3. Each layer processes the packet, stripping away its respective header and tail information.
4. This continues until the original payload reaches the application layer for consumption.

This layered approach allows each level of the network stack to perform its specific functions while maintaining a standardized way of handling data across different systems and networks.

While nodes are responsible for generating and consuming network traffic, the data must traverse the network to reach its destination. This is where network switches come into play.

ATS in TSN switches The ATS process involves several key components that work together to shape traffic:

1. **Stream Classifier:** This component categorizes incoming packets based on their associated stream names. It uses StreamReq or StreamInd tags to identify the stream and maps it to a specific output path. In a standards-compliant TSN context, this classifier effectively maps streams to one of the 8 Priority Code Point (PCP) values defined by IEEE 802.1Q.

2. **Eligibility Time Meter:** Implemented by the EligibilityTimeMeter class, this component calculates when packets become eligible for transmission. It uses a token bucket algorithm characterized by a Committed Information Rate (CIR) and a Committed Burst Size (CBS). For each packet, it determines an eligibility time based on the packet's arrival time, the current state of the token bucket, and a group eligibility time.

3. **Eligibility Time Filter:** The EligibilityTimeFilter class controls packet flow based on the calculated eligibility times. It examines the EligibilityTimeTag of each packet and allows packets to proceed only if their eligibility time has been reached.

4. **PCP Classifier:** After ATS processing, packets are classified again based on their PCP values. This ensures that packets are directed to the appropriate queue corresponding to their priority level.

5. **EligibilityTimeQueue:** This specialized queue, implemented by the EligibilityTimeQueue class, orders packets based on their eligibility times. It uses a PacketEligibilityTimeComparator to maintain packets in ascending order of their eligibility times.

6. **EligibilityTimeGate:** The final component in the ATS mechanism, implemented by the EligibilityTimeGate class, controls the flow of packets from the queue based on their eligibility times. It continuously updates its open/closed state by comparing the eligibility time of the next packet in the queue with the current clock time.

The ATS shaping process works as follows:
1. Incoming packets are classified into streams and assigned to one of the 8 PCP values.
2. The Eligibility Time Meter calculates the eligibility time for each packet.
3. Packets are then filtered based on their eligibility times and placed into the EligibilityTimeQueue.
4. The queue maintains packets ordered by their eligibility times.
5. The EligibilityTimeGate releases packets from the queue only when their eligibility time has been reached.

This process ensures that traffic is shaped according to the configured ATS parameters, providing deterministic, low-latency communication for time-sensitive applications. The modular nature of the OMNeT++ implementation allows for flexible configuration of ATS on a per-PCP stream basis, enabling precise control over traffic shaping in TSN simulations.

Note: Figure 6 illustrates the modularity and flexibility of the INET switch model:

- The dashed line around ATS for PCP 0-2 is an example showcasing that traffic shaping can be enabled on a per-queue basis.

- Different queues can be managed by different traffic shaping mechanisms. For instance, some queues might use ATS, while others use Time Aware Shaping (TAS) or no shaping at all.

- Each queue can have multiple traffic shapers enabled, allowing for hybrid traffic shaping. For example, a queue could implement both ATS and TAS simultaneously.

- In hybrid-shaped queues, traffic is pulled only when all enabled shapers for that queue allow it. This means in an ATS + TAS hybrid configuration, both shapers must permit transmission for a packet to be sent.

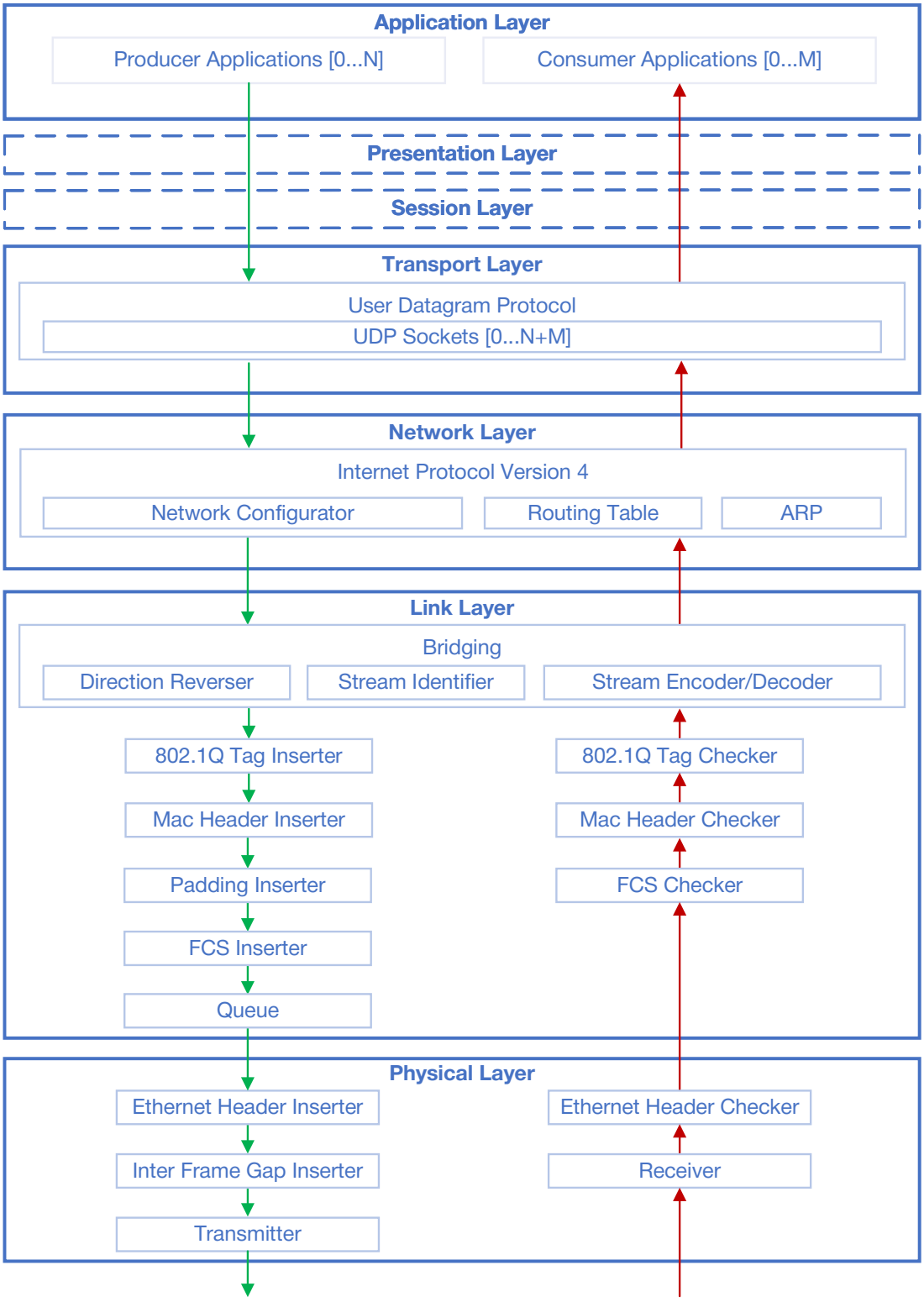


Figure 5: TSN Node Model

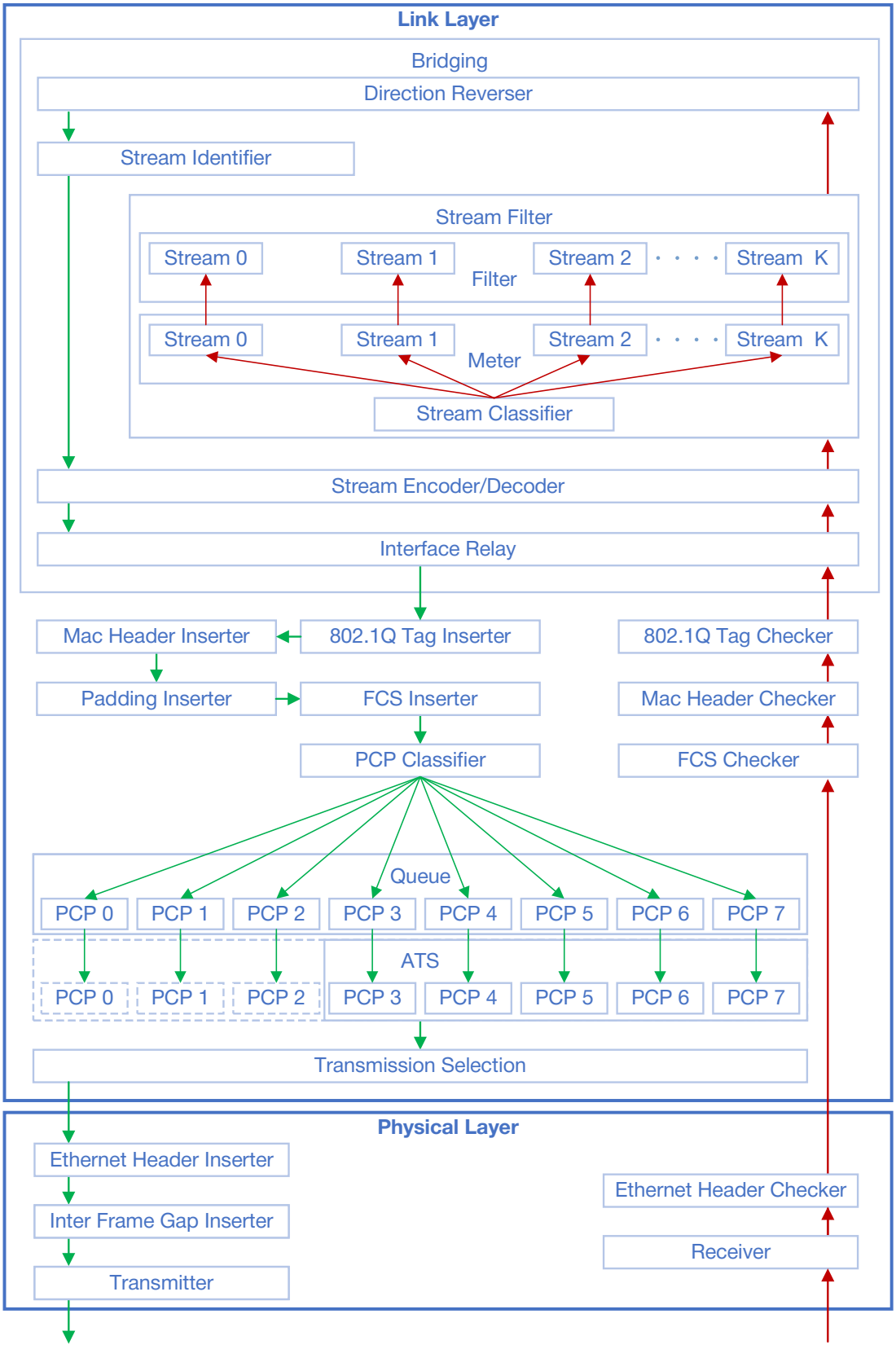


Figure 6: TSN Switch Model

Basic Simulation Setup Start by defining the network and simulation time:

```
[General]
network = dtu_networks.ring_3sw_network
sim-time-limit = 1.0s
```

This sets up a 1-second simulation of our 3-switch ring network.

Visualization Configuration To help visualize different traffic types, we'll set up route visualizers:

```
*.visualizer.typename = "IntegratedMultiCanvasVisualizer"
*.visualizer.numNetworkRouteVisualizers = 3
*.visualizer.networkRouteVisualizer[*].displayRoutes = true
*.visualizer.networkRouteVisualizer[0].packetFilter = "\"Isochronous*\""
*.visualizer.networkRouteVisualizer[0].lineColor = "red1"
*.visualizer.networkRouteVisualizer[1].packetFilter = "\"Cyclic*\""
*.visualizer.networkRouteVisualizer[1].lineColor = "blue4"
*.visualizer.networkRouteVisualizer[2].packetFilter = "\"Video_Audio_Voice*\""
*.visualizer.networkRouteVisualizer[2].lineColor = "green1"
```

This configuration creates three visualizers, each highlighting a different traffic type with a unique color.

Network Configuration Set the Ethernet bitrate and add a small processing delay:

```
*.*.eth[*].bitrate = 1Gbps
*.*.bridging.directionReverser.delayer.typename = "PacketDelayer"
*.*.bridging.directionReverser.delayer.delay = 8us
```

These configuration lines do the following:

- Set the bitrate for all Ethernet interfaces to 1 Gigabit per second.
- Specify a "PacketDelayer" as the type of delayer in the direction reverser.
- Set a processing delay of 8 microseconds in the direction reverser.

Note: The processing delay directly influences the end-to-end packet latency. This delay determines the amount of time each packet is held within the direction reverser before being forwarded. In a network with multiple switches, this delay accumulates, contributing to the overall latency experienced by packets traveling through the network.

Traffic Generation and ATS Configuration We'll configure three types of traffic: Isochronous, Cyclic, and Video/Audio/Voice. Each is set up as a UDP application. Referring to the node model in Figure 5, these applications are part of the Application Layer.

Here's a detailed explanation of the key ATS parameters:

- **committedInformationRate**: This is the guaranteed bandwidth for a traffic stream, measured in bits per second (bps). It represents the average data rate that the network commits to providing for this stream.
- **committedBurstSize**: This is the maximum amount of data that can be sent at once in a burst, measured in bytes. It allows for short-term variations in traffic while still maintaining the average rate specified by the committedInformationRate.

These parameters affect traffic shaping as follows:

- A higher committedInformationRate allows more data to be sent per second on average.
- A larger committedBurstSize allows for bigger short-term spikes in traffic, which can be useful for bursty data patterns.
- The combination of these parameters determines how strictly the traffic is shaped and how much variation is allowed in the data rate.

Now, let's configure our traffic types:

```
*.node_0_0.numApps = 2
*.node_0_0.app[0..1].typename = "UdpSourceApp"

# Isochronous traffic
*.node_0_0.app[0].display-name = "Isochronous"
*.node_0_0.app[0].io.destAddress = "node_1_0"
*.node_0_0.app[0].io.destPort = 1
*.node_0_0.app[0].source.productionInterval = 100us
*.node_0_0.app[0].source.packetLength = 86B

# Video/Audio/Voice traffic
*.node_0_0.app[1].display-name = "Video_Audio_Voice"
*.node_0_0.app[1].io.destAddress = "node_2_0"
*.node_0_0.app[1].io.destPort = 3
*.node_0_0.app[1].source.productionInterval = 10000us
*.node_0_0.app[1].source.packetLength = 1239B

# (Similar configuration for node_0_1 with Cyclic traffic)
```

Traffic Consumption Configure the receiving nodes:

```
*.node_1_0.app[0..1].typename = "UdpSinkApp"
*.node_1_0.app[0].io.localPort = 1
*.node_1_0.app[1].io.localPort = 2

*.node_2_0.app[0..1].typename = "UdpSinkApp"
*.node_2_0.app[0].io.localPort = 1
*.node_2_0.app[1].io.localPort = 3
```

Stream Identification and Encoding Configure how packets are identified and assigned Priority Code Points (PCPs). This corresponds to the Stream Identifier and Stream Encoder/Decoder in the Link Layer of the node model:

```
*.node*.bridging.streamIdentifier.identifier.mapping = [
    {stream: "Isochronous", packetFilter: expr(udp.destPort == 1)},
    {stream: "Cyclic", packetFilter: expr(udp.destPort == 2)},
    {stream: "Video_Audio_Voice", packetFilter: expr(udp.destPort == 3)}]

*.*.bridging.streamCoder.encoder.mapping = [
    {stream: "Isochronous", pcp: 6},
    {stream: "Cyclic", pcp: 5},
    {stream: "Video_Audio_Voice", pcp: 1}]
```

Switch Configuration Set up the switches to handle the different traffic types. Referring to the switch model in Figure 6, this configuration relates to the Stream Identifier, Stream Encoder/Decoder, and PCP Classifier components:

```
*.sw*.bridging.streamCoder.decoder.mapping = [
    {stream: "Isochronous", pcp: 6},
    {stream: "Cyclic", pcp: 5},
    {stream: "Video_Audio_Voice", pcp: 1}]

*.sw*.eth[*].macLayer.queue.classifier.mapping =
[[0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2], [0, 1, 2],
[0, 1, 1], [0, 0, 0], [0, 1, 2]]
```

Asynchronous Traffic Shaping Configuration Finally, set up the ATS. This configuration corresponds to the Stream Filter, Meter, and ATS components in the switch model:

```
*.sw*.hasIngressTrafficFiltering = true
*.*.hasEgressTrafficShaping = true

*.sw*.bridging.streamFilter.ingress.numStreams = 3
*.sw*.bridging.streamFilter.ingress.classifier.mapping = {
    "Isochronous": 0, "Cyclic": 1, "Video_Audio_Voice": 2 }

*.sw*.bridging.streamFilter.ingress.meter[*].typename = "EligibilityTimeMeter"
*.sw*.bridging.streamFilter.ingress.filter[*].typename = "EligibilityTimeFilter"

*.sw*.bridging.streamFilter.ingress.meter[0].committedInformationRate = 10Mbps
*.sw*.bridging.streamFilter.ingress.meter[0].committedBurstSize = 500B
*.sw*.bridging.streamFilter.ingress.meter[1].committedInformationRate = 10Mbps
*.sw*.bridging.streamFilter.ingress.meter[1].committedBurstSize = 5000B
*.sw*.bridging.streamFilter.ingress.meter[2].committedInformationRate = 10Mbps
*.sw*.bridging.streamFilter.ingress.meter[2].committedBurstSize = 5000B

*.sw*.eth[*].macLayer.queue.transmissionSelectionAlgorithm[*].typename =
"Ieee8021qAsynchronousShaper"
```

Note how the Isochronous traffic (meter[0]) has a smaller committedBurstSize compared to the other traffic types. This reflects its need for more consistent, less bursty transmission.

By following these steps, you can create an .ini file that simulates various TSN traffic types using the ATS shaper. This configuration allows you to observe how different traffic types are prioritized and shaped in a TSN environment.

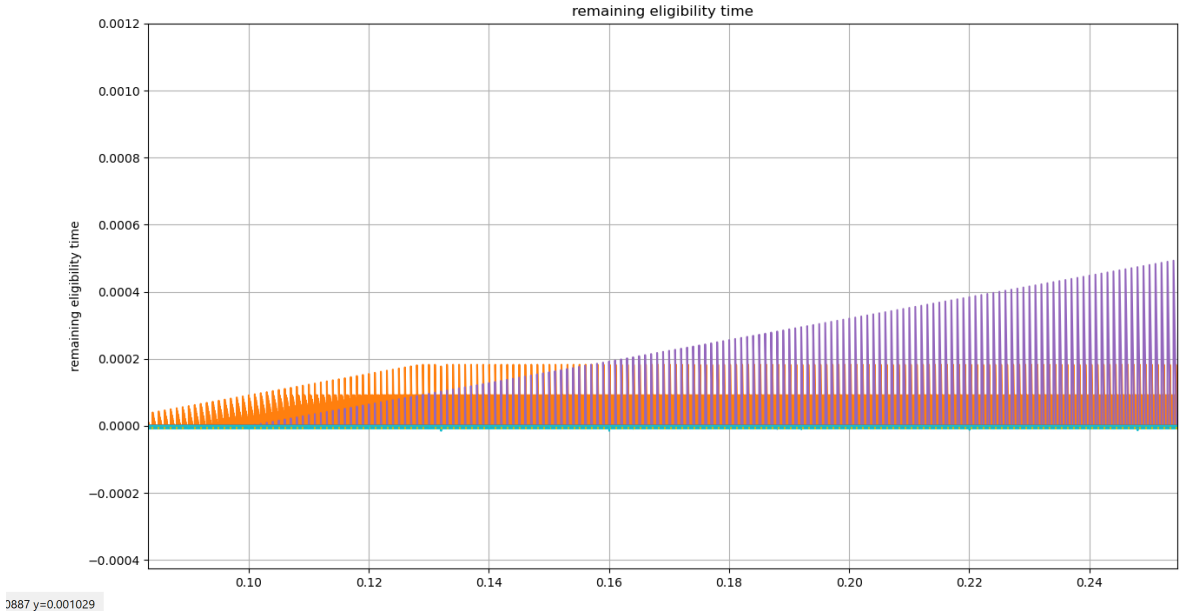
A.2.5 Running the Simulation and Accessing Results

To run the simulation and view the results:

- Right-click on the `3sw_ats.ini` file in the Project Explorer.
- Select **Run As** > **OMNeT++ Simulation**.
- In the launched simulation window, click the **Run** button to start the simulation.
- After the simulation completes, results will be stored in the "results" folder of your project.
- To view the results:
  - Right-click on your project folder
  - Select **New** > **Analysis File (anf)**
  - In the new analysis file, click **Inputs** and add the result files from your "results" folder
  - Use the **Browse Data** tab to select data for plotting

A.2.6 Analyzing Simulation Results

After running the simulation, you can analyze the results to understand how ATS affects different traffic types. Figure A.2.6 shows an example of simulation results:



By analyzing these results, you can gain insights into how the ATS algorithm manages different traffic types in your TSN network, ensuring that high-priority traffic (like Isochronous) meets its timing requirements while still allowing lower-priority traffic (like Video/Audio/Voice) to be transmitted.

B Appendix: Creating TSN Test Cases

Realistic test cases are crucial for meaningful network simulations. While simplified scenarios can be useful for initial testing and concept validation, they often fall short in capturing the complexity and challenges of real-world networks.

Using test cases that are too small or overly simplified may lead to overly optimistic results or fail to reveal critical issues that only emerge in more complex environments. So, it is a good idea to design test cases that are realistic and are not too simple. You could incorporate various topology types, multiple traffic classes with different requirements, and realistic network loads, see the discussion in this section.

By doing so, you can gain more accurate insights into network performance, identify potential bottlenecks, and evaluate the effectiveness of TSN mechanisms under conditions that closely resemble real-world scenarios. The following sections outline key considerations for creating robust and realistic test cases for TSN simulations.

B.1 Topology Types in Time-Sensitive Networks

The choice of topology can significantly impact the network's performance, reliability, and scalability. Common topologies used in industrial networks include line (bus) topology, ring topology, and mesh topology.

B.1.1 Line Topology

In a line topology, all devices are connected to a single central cable. It is simple to implement, cost-effective, and easy to extend. However, it is vulnerable to single point failures.

B.1.2 Ring Topology

In a ring topology, each device is connected to two other devices, forming a circular data path. This configuration ensures network resilience in case of a single connection failure.

For large-scale industrial environments, a hierarchical ring topology can be particularly useful. Figure 7 illustrates a hierarchical ring topology with primary (backbone) and secondary (local) rings.

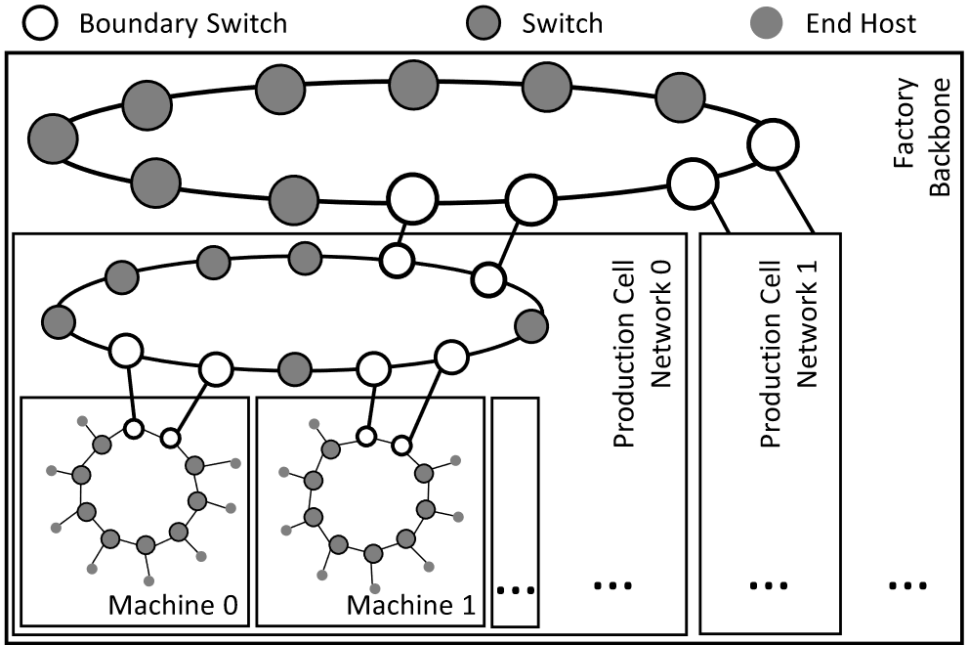


Figure 7: Hierarchical ring topology

B.1.3 Mesh Topology

In a mesh topology, each device is connected to multiple other devices, creating a web of interconnections. Mesh topologies can be generated by different graph models, such as random regular graph model, Erdős-Rényi (ER) model, and Barabási-Albert (BA) model. Figure 8 shows examples of these models.

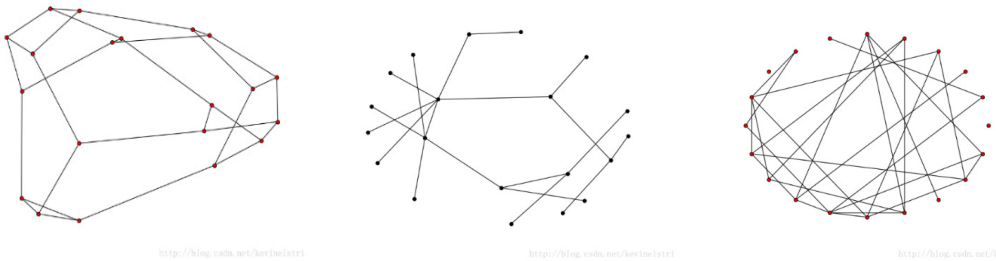


Figure 8: Mesh topologies: (a) Random model (b) ER model (c) BA model

B.2 Traffic Description for Industrial Automation Applications

In industrial automation systems, ”traffic” refers to the flow of data and communication between various devices, systems, and networks. Figure 9 illustrates how traffic flows within a tank control system.

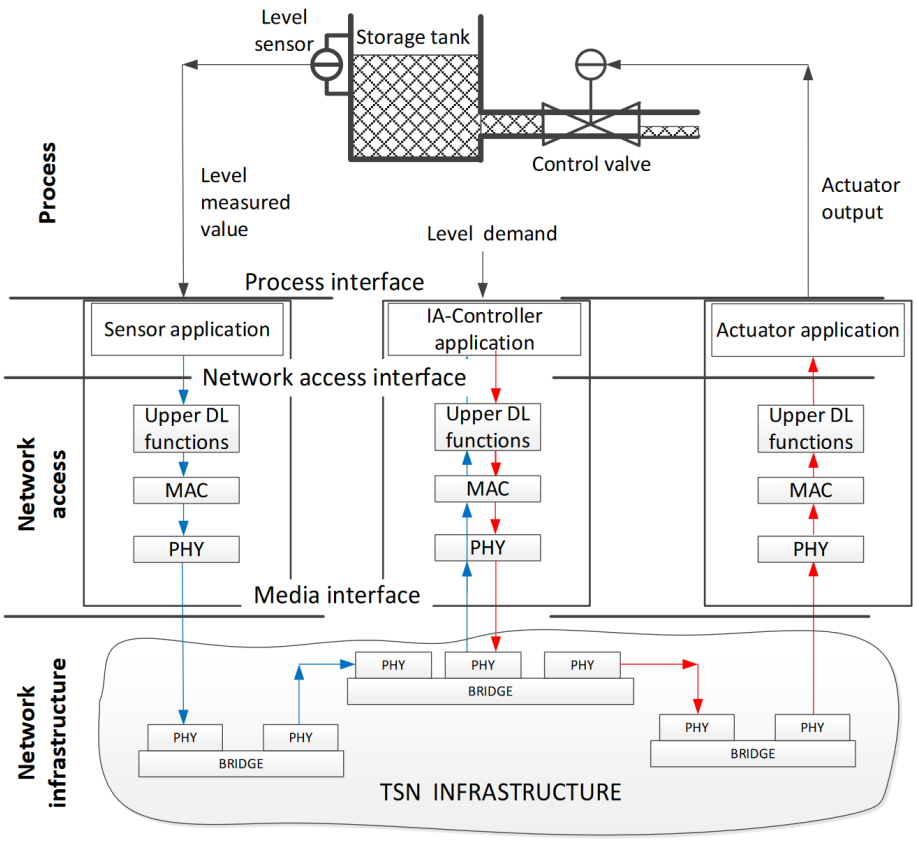


Figure 9: Networked control system [7]

B.2.1 Traffic Types in Industrial Automation Applications

Industrial automation (IA) applications use different traffic types for various functionalities. Table B.2.1 summarizes relevant industrial automation traffic types and their associated characteristics [7].

Traffic type name	Code	Cyclic	Data delivery	Time-triggered	Category
Isochronous	H	Required	Deadline	Required	IA time-aware-stream
Cyclic-Synchronous	G	Required	Frame Latency	Required	IA time-aware-stream
Cyclic-Asynchronous	F	Required	Frame Latency	Optional	IA stream
Alarms and Events	E	Optional	Flow Latency	Optional	IA traffic engineered non-stream
Configuration & Diagnostics	D	Optional	Flow Latency	Optional	IA traffic engineered non-stream
Network Control	C	Optional	Flow Latency	Optional	IA traffic engineered non-stream
Best Effort High	B	Optional	No	Optional	IA non-stream
Best Effort Low	A	Optional	No	Optional	IA non-stream

B.2.2 Required Elements in Traffic Description Profile

When defining a traffic profile for network simulation in industrial applications, the following elements should be included:

1. **PCP**: Priority Code Point assigned to the specific category of data.
2. **Size**: The size of the data packets being transmitted.
3. **Cycle**: The period of data transmission.
4. **Deadline**: The maximum allowable delay from data transmission to reception.
5. **Source**: The end station sending the data stream.
6. **Destination**: The end station receiving the data stream.
7. **Routing path**: The path the data stream takes through the network.

For example, a control message stream f_1 could be described as {PCP=7, 500 B, 1 ms, 1 ms, 5, 6, 5-1-2-3-6}. Table 2 provides an example of traffic in an automotive application.

Table 2: A traffic example in Automotive network

Flows	Flow count	Max. Payload (B)	Period (ms)	D (ms)	Type	Workload $L_{SW1,CC}$ (Mbps)
LiDAR	4	1248	1.3	1.3	Periodic	31.4
RADAR	4	625	2.5	2	Periodic	8.9
Ultrasonics	4	188	100	2	Periodic	0.4
Video	8	1500	16.667	16.667	Periodic	702.4
ADAS-Sensors1	4	1500	uniform (10,100)	1	Event-driven	Max 32.9
ADAS-Sensors2	4	1500	uniform (10,100)	1.5	Event-driven	Max 32.9
TelematicsData	1	1500	6	6	Periodic	2.1
HeartBeat	24	64	10	10	Periodic	2.0
DA-CAM	1	1500	0.166	0.166	Periodic	-

By incorporating these elements into the definition of a traffic profile, network simulations can more accurately reflect real-world conditions and requirements of industrial applications, ensuring that the network is designed to meet the specific needs of the industrial environment.