Università degli Studi di Milano Bicocca

Department of Informatics, Systems and Communication DISCo

Master's degree in Information Technology

Definition and evaluation of a strategy for Real-Time Traffic Engineering in SD-WAN

Supervisor: Marco Savi, PhD

Thesis by:
Michele Beccari
856608

Academic Year 2023-2024

Abstract

Wide Area Networks (*WANs*) are adopted by large companies and organizations to overcome the potential reliability issues that may arise by just using general connectivity (i.e the standard connections offered by Internet Service Providers (*ISPs*), usually sold for personal use).
However, while offering greater guarantees, setting up a full WAN connectivity is massively more expensive than using ISPs; some WANs may even require renting actual cables from ISPs. SD-WANs (*Software Defined WAN*) offer an hybrid approach to find a middle ground between the reliability of standard WANs and the economic advantage of general connectivity.
When using both standard WAN connectivity solutions and general connectivity traffic may be routed through the different interfaces according to the requirements of the source of the traffic itself. (i.e the more expensive interfaces are only used when strictly necessary)
This paper tries to explore a strategy to route traffic across different kinds of interfaces to minimize the overall transmission cost, while still respecting the requirements of each source.

Contents

1 Background

The types of internet connections can be usually divided in two main categories, **Generalized connectivity** and **Dedicated connectivity**.

1.0.1 Generalized connectivity

The most popular kind of internet connection provided by an *ISP* (*Internet Service Provider*) , usually adopted by residential users.
It is based on an hierarchical structure where a set of autonomous system (*AS*) communicate between each other: any two user adopting this type of connection are able to communicate using different protocols (*IGP* protocol if the users are in the same AS and *EGP* otherwise)

1.0.2 Dedicated connectivity (WAN)

A specialized kind of connection, mainly used by businesses (i.e to connect the different points of sale of a large company or the different branches of a bank).

Organizations usually opt in into this kind of connection to ensure higher standards in regards to different metrics (i.e lower delays, higher availability) when compared to the *best-effort* approach of generalized connectivity.

Dedicated connectivity is still offered by internet service providers, but it is several orders of magnitude more expensive than generalized connectivity.
A network adopting this approach is called a *WAN* (Wide Area Network); A WAN is a network of LANs (Local Area Network, i.e the branches) where the distance between the LANs is usually in the tens or hundreds of kilometers range.
By using a *border router* in each LAN, it is possibile to communicate with the other LANs in the WAN.
The LANs can be connected in different ways:

1. *Dedicated physical WAN* : the business buys the full physical infrastructure that the networks is built on (the cables, the switches etc.).
   While the company is the owner of the whole WAN, the cost is extremely high and network management (e.g enforcing security) is the the company's responsibility.
2. *Leased Lines* : the business relies on an ISP to establish private circuits between the branches. (e.g branch 1 must be connected to branch 2, which must be connected to branches 3 and 4)
   The private circuit may be created by assigning a dedicated wavelength to a particular customer.
   This solution offers only partial control, but is cheaper and offsets some of the network's management issues to the ISP.
3. *MPLS networks*: The business still relies on an ISP to provide connectivity, but unlike leased lines there are no established circuits between the branches. Instead, a mesh connectivity is provided , with a certain QoS guarantee.
   For each branch a *customer edge router* is installed, allowing the branch to access the MPLS network via a *provider edge router*.
   The *provider edge router* (PE) is in between the MPLS network and the companies, one PE may manage connections for multiple customers.

4. *SD-WAN*: this kind of WANs will be discussed in detail in the next section.

2 Software defined networking and SD-WAN

2.1 Networking devices

A typical network device (e.g routers, firewalls...) is usually composed by two logical components:

1. A data plane
2. A control plane

The data plane is used to handle individual data packets locally. For example the data plane in a router is used to determine to which interface a packet should be forwarded to. The data plane usually applies a large amount of small, fast operations.
The control plane is used to handle control messages from other devices in the network, and allows the network device to work from a network wide perspective.
The messages from other devices may be used to configure policies which are then applied by the data plane (e.g the control plane receives a policy that determines which packets should be filtered, while the data place does the actual filtering).
Compared to the data plane, the control plane is used for more complex operations that usually require coordination along network devices.

2.2 Definition

The idea behind software defined networking (SDN) is to de-couple the control plane from the data plane, not only on a logical level but also physically.

The control plane is no longer part of the network devices and is instead moved to a centralized software component called a *controller*.

The basic architecture of a SDN is composed by the controller and by a series of network devices which in this context are called *switches*.

A SDN controller

1. Is able to communicate with the switches
2. Can be programmed: it is now possible to write software that defines how the whole network behaves.
   The switches keep executing their data plane locally, according to the policies received by the controller.
3. Is able to acquire information about the network it is managing, e.g it can discover the topology of the network to know where the switches are places and how they are connected.

The goal of the controller is to communicate with all the switches to inject into them the desired network behavior

The controller is logically placed between the network infrastructure and the application layer.

Using a series of high level programmable interfaces (also called *APIs* A*pplication P*rogrammable *I*interfaces) the controller is able to communicate with its neighboring layers.

Using the APIs connected with to the application layer (the *northbound*) interfaces various network devices can be implemented, e.g a network balancer, a firewall.

The communication between the controller and the switches is made possibile by a series of *southbound* interfaces.

Using these interfaces the controller can effectively apply the policies to the network, e.g it can send traffic rules to the switches connected to the controller.

Inside each switch there is a set of *generic tables* (also called *flow tables*) that are used to store the configuration sent by the southbound interfaces.

The southbound interfaces can also be used in "reverse": the switches are able to communicate with the controller using a series of predefined messages.

2.3 Openflow

Despite being relatively new, software defined networking became a popular approach for organizing new networks (e.g in 5G networks) Compared to other widespread network technologies such as the TCP/IP stack, software defined networking is a relatively new approach: the first paper defining protocol for SDN (the *Openflow Protocol*) was published in 2008.

The OpenFlow protocol defines a sets of API for remote communication with network plane devices, i.e. a southbound interface.

SDN switches inside the context of OpenFlow are also called *generic* switches; This is to avoid confusion with the usual meaning of the term switch, which usually refers to a device used to manage level 2 functionalities (i.e. the data link level, used to handle the physical addressing with MAC addresses). Openflow switches are able to interact up to the fourth layer (i.e. the transport layer, which may include protocols such as TCP or UDP).

A large portion of an OpenFlow Switch is composed by functionalities related to the data plane, the only portion dedicated to the control plane is delegated to the *control channel*.

The control channel includes one or more OpenFlow channel that allows the switches to communicate with one or more controllers (multiple controllers may be used to avoid a single point of failure in the infrastructure).

OpenFlow switches include three kind of tables, used for handling data plane functionalities:

- *Flow tables*: the most important kind of table, they can be chained in a pipeline.
  They contain the rules that are applied to the incoming and outgoing traffic.
- *Group tables*: Users of a SDN can be assigned to different group; group tables allow traffic to be sent in multicast to users of a specific group (e.g. group tables can be used to send outgoing traffic to multiple ports if necessary)
- *Meter table*: table used to collects statistic about the current network.
  The gathered data can then be used to handle QoS (e.g rate limiting for traffic directed to a specific port)

2.3.1 Configuring the tables

To work properly, the flow tables inside the switches have to be configured by the controllers in the network. The solution is usually an hybrid of two types of configuration:

1. *Proactive configuration*: When the switch is first inserted into the network, the controller proactively fills the tables.
   The entries in the tables are static and do not change across the lifespan of the switch in the network.

This kind of approach is obviously unfit for rapid changing scenarios such as those encountered in managing networks.

2. *Reactive configuration*: Initially there no entries in the flow tables; the tables are fully configured at runtime according to the network state.
   If a switch does not known where a package should be send to, it forwards the packet to the controller.
   The controller is aware of the full topology of the network, and is then able to edit the received package to include the routing information, before sending the package back to the switch..
   This leads to increased round trip time for the first packets sent across the network.
   However, communication between the switches and controller usually represent a bottleneck for the network's performance; a large amount of packets sent to and from the controller may cause congestion in the network and limit the available bandwidth.

Usually the preferred solution is based on an hybrid approach.
A starting set of rules is used at the start up of the network, to cover the expected data flows in the network.
For any unknown data flow that may arise during the lifespan of the network new routing rules will be issued by the controller.

2.4 SD-WAN

2.4.1 SD-WAN

The principles of software defined networking can be applied when building a WAN; While SD-WAN were introduced recently (in 2014) their popularity increased due to the possible decrease in costs when compared to traditional WAN solutions.

SD-WAN works by stipulating multiple contracts of generalized internet connections (e.g 4G, POM, fiber) and then combining the different networks to guarantee a certain quality of service level.

Like the MPLS approach, a CPE (*Customer Premises Equipment*) is installed in each branch of the business.

The CPEs is connected to the branch's LAN and to multiple generalized connectivity networks.

The traffic generated by the LAN may then travel in different networks, potentially even from different ISPs.

An SDN controller, in this specific context called an *SD-WAN Controller* is responsible to communicate to each CPE how the outgoing traffic should be managed (e.g two branches may be connected using a fiber cable, if the cable connection fails the controller will instruct the branches to communicate using a 4G network).

MPLS may still be used in a SD-WAN context, a company could decide to use other (usually cheaper) kind of connections for most use cases and reserve a MPLS connection only exceptionally.

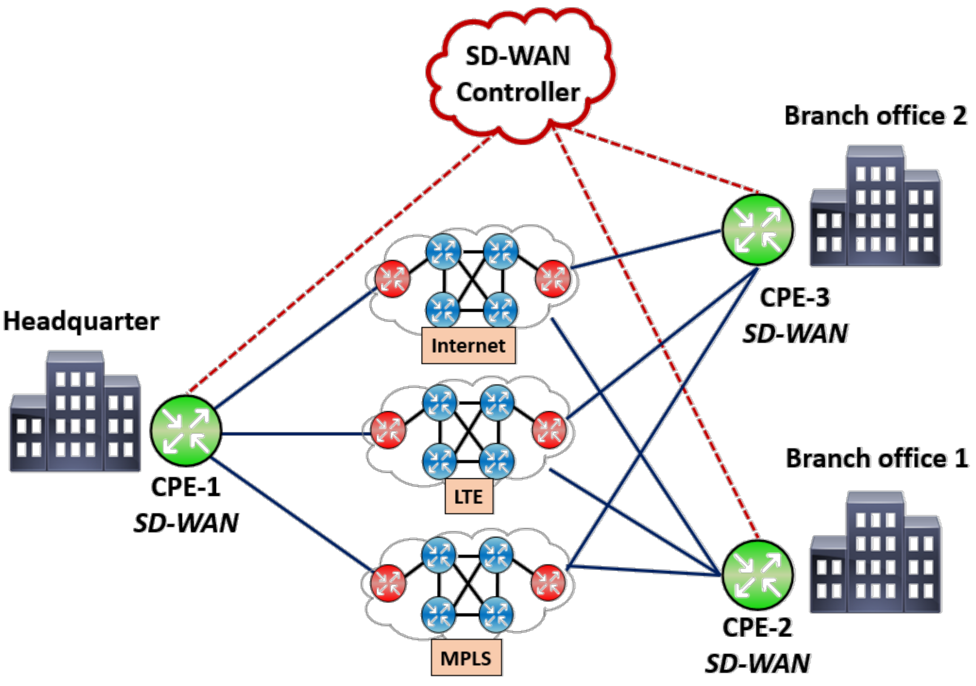

Figure 1: A sample network of three branches adopting SD-WAN

2.4.2 Load balancing and SLA in SD-WANs

As we previously discussed, in SD-WANs a controller's task is to instruct CPEs on how they should distribute outgoing traffic among the available links.

The traffic handled in a SD-WAN context is often heterogeneous and generated by different applications (i.e VoIP, emails, video streaming . . .).

Different applications have usually varying requirements in regards to the acceptable delay and the packet loss that can happen during the transmission of data.

Two important terms have to be introduced to describe the needs of each network application:

1. *QoS (Quality of Service)*: a quality indicator used to measure the service level of a given network communication. It is defined with a series of guarantees that must be respected during a network communication.
   In this paper QoS will always be defined in absolute terms (e.g. end to end latency is less than 5 milliseconds, 2% of packets have been dropped), however it can also be defined in relative tems (i.e how a given traffic flow is treated with respect to the others)

2. *SLA (Service Level Agreement)*: the SLA specifies the QoS that has to be guaranteed for a given application. It provides a series of bound on various measurable parameters (i.e the latency *must* be less than 5 milliseconds)

3 Problem definition

3.1 Load balancing as a linear optimization problem

On of the goals of SD-WAN is to minimize costs while still maintaining a set of requirements for a given set of applications.

With *application* we mean any kind of source of internet traffic with a defined set of requirements.
An application has a *life cycle*: it starts, it generates data according to arbitrary rules and it terminates when no more data has to be sent.

As a first step in formalizing the problem, we can picture a scenario in which we have a CPE sending an arbitrary amount of packets to a sink.

The CPE is connected to the sink through a set of distinct links.

The goal of the CPE is to allow an *application* to transmit an arbitrary amount of packets to the sink. The transmission of data has some bounds in regards to:

- Average bandwidth
- Average delay
- Average Packet loss

An important assumption on the application requirements is that they only apply to the average:
if an application requires the average delay to be 100ms some of the packets can be sent with an higher delay , as long as the overall delay stays under the desired threshold.
Managing applications that have stricter requirements (e.g all packets must always be sent with
a maximum a delay of 100ms) go beyond the scope of our strategy.

Each link has different characteristics in regard to:

- Available bandwidth *(i.e how many bytes can be sent per second through the link)*
- Delay *(i.e how long it takes for a packet to reach the sink)*
- Cost *(i.e a metric used to track how convenient is to use a link instead of another, the section 3.1.1 goes into further details*
- Packet loss *(i.e how many packets are lost and/or need to be sent multiple times?)*

3.1.1 Measuring the cost of links

Each link as a cost associated with sending a packet on the link itself.
The cost of sending a packet is not a monetary cost, rather it is a measure of how convenient it is to use a given link.
Monetary costs for different kinds of connections are measured in multiple ways: i.e MPLS

connections charge increasing amount of money depending on the amount of traffic, while typical consumer ISPs charge a fixed amount regardless of the amount of traffic.
This cost for each packet is fixed for each link, and it does not change through the lifetime of any application.
This means that when assigning the packet cost to a given link we assign lower costs to the links that have a fixed monetary cost, and an higher cost to the link the have a pay-as-you-go model.

3.1.2 The problem as a linear optimization problem

We can formalize the problem as a linear optimization problem; As a starting point we consider an application sending an amount N of packets of fixed size using three different links.

The amount of packets is fixed and generated by a single application: our goal is to minimize the overall cost of the transmission while respecting the different requirements.

To simplify the problem we set the bandwidth of the links in a way that it is always a multiple of the size of a single packet (i.e the bandwith of a link can only be an integer, 1 packet per second, 5 packet per second ...)
We consider a set of links L, to visualize the example we can set $L = \{l_1, l_2, l_3\}$.

For each one we define

- The link delay: d_1, d_2, d_3, measured in ms
- The link bandwidth: b_1, b_2, b_3, measured in *packets per second (pps)*
- The cost for sending a single packet c_1, c_2, c_3
- The probability of losing a packet, expressed as a number in the range $[0, 1)$

The total cost time required to send the packets can be expressed as:

$$N_1 \cdot c_1 + N_2 \cdot c_2 + N_3 \cdot c_3$$

Where N_i is the amount of packets sent on the the ith interface.
We also have to consider the bound on average delay:

$$\frac{N_1 * d_1 + N_2 * d_2 + N_3 * d_3}{N} \leq \text{delay}$$

The bound on average bandwidth:

$$\frac{N_1 * b_1 + N_2 * b_2 + N_3 * b_3}{N} \leq \text{bandwidth}$$

The bound on packet loss, expressed as a percentage:

$$\frac{N_1 \cdot p_1 + N_2 \cdot p_2 + N_3 \cdot p_3}{N} \leq \text{packet loss}$$

Finally, we need to express a bound to make sure every packet is actually sent.

$$N_1 + N_2 + N_3 \geq N$$

This simplified version of the problem can be expressed as:

$$
\begin{cases}
\text{min} & N_1 \cdot c_1 + N_2 \cdot c_2 + N_3 \cdot c_3 \\
\text{subject to:} & \dfrac{N_1 \cdot d_1 + N_2 \cdot d_2 + N_3 \cdot d_3}{N} \leq \text{delay} \\
& \dfrac{N_1 \cdot b_1 + N_2 \cdot b_2 + N_3 \cdot b_3}{N} \leq \text{bandwidth} \\
& \dfrac{N_1 \cdot p_1 + N_2 \cdot p_2 + N_3 \cdot p_3}{N} \leq \text{packet loss} \\
& N_1 + N_2 + N_3 \geq N
\end{cases}
\tag{1}
$$

3.2 Extending the problem

While the linear optimization problem allows for an easy and clear solution, it can only work if some assumptions are made:

- Only a single application is sending data at any given time
- The amount of traffic generated by the application must be known beforehand (this is usually unrealistic, unless for specific situations, i.e file transfer)

This linear approach does also not take into account delays introduced by the outgoing traffic; the theoretical delay on a given link is only a lower bound.

When sending multiple packages over an interface the time required to send the packages that are already into the queue must also be taken into account (e.g if a package is sent through an interface where n packages are already queued, the n packages must be sent first)

The linear optimization strategy is still useful to set a lower bound on the lowest possible overall transmission cost.

To allow for modeling of more complex scenarios, we extend the problem definition to allow the presence of multiple applications.

In this new scenario, each application can now generate traffic at any point of its lifetime: the total amount of traffic can not be determined until the end of the application lifecylce.

Examples of different applications may include some that generates:

- A fixed amount of traffic every second.
- A random amount of traffic every hundred milliseconds
- A burst of traffic when starting the application, then no more.
- Any kind of commmonly studied traffic model (i.e Poisson distribution traffic model, Pareto distribution model)
- A simplified model of traffic (i.e sinusoidal traffic)

Any proposed solution must:

- Be able to account for multiple sources of traffic
- Be able to account for the delay caused by the queuing of packages on a given interface

4 An existing strategy

4.1 Intent-Based Routing Policy

In the paper *Intent-Based Routing Policy Optimization in SD-WAN* [6] a strategy is discussed to optimize different metrics in a scenario with multiple applications (called "flow groups" in the original paper) with different QoS requirements; the traffic generated from a set of applications has to be distributed on a set of available links.

The goal of the strategy is to provide the "best" possible split ratio for the traffic generated by the different applications among the available links.

What is the "best" split ratio can be arbitrarily decided: in the original paper multiple metrics are proposed such as minimizing link utilization, or even improving the quality of the transmission beyond what is strictly required.

The idea behind the strategy is run a linear optimization problem periodically so that it can adjust to changing demands by the different applications.

The linear optimization resolution relies on an estimate to determine what the delay on a given link for a given application will be in the future; a similar approach could be used to estimate an error rate for a given application on a given link, as a starting point thought we suppose the error rate is always fixed given an link.

4.2 The linear optimization problem

The original paper provides a formulation with a linear programming model that may include different objectives, including:

1. An objective that measures if the SLA for a given application (in the original paper called a *flow group*) can be violated. We will not allow for any violation of SLA, so this objective can be safely ignored.
2. An objective function that minimizes the MLU (Maximum link utilization)
3. An objective function that can eventually be used to decrease the delay beyond SLA requirements.
4. An objective function that may be used to minimize costs.

For our purposes, we can modify the proposed model. While the importance of the various objective functions in the original formulation can be adjusted using weights, we can ignore most of the proposed functions (i.e setting their weight to 0).

We consider a SD-WAN composed by a CPE sending data to a sink; the network is composed by:

- A set of links L with bandwidth $B_l, \ \forall l \in L$
- A set of applications, A. For each application $a \in A$, the traffic demand is denoted with t_a, the required delay over all links is $\bar{D}_a$, the required bandwidth is $\bar{B}_a$ and the required error rate over all links is $\bar{E}_a$

The modified formulation is:

$$
\begin{aligned}
\min \quad & \sum_{l \in L} c_l \cdot LU_l & & & (1)\\
\text{s.t.} \quad & LU_l = \sum_{a \in A} t_a x_l^a \le C_l, & & \forall l \in L, & (2)\\
& f_l^a(x) \le D_a, & & \forall a \in A, \forall l \in L, & (3)\\
& D_a \le \bar{D}_a, & & \forall a \in A, & (4)\\
& e_l^a(x) \le E_a, & & \forall a \in A, \forall l \in L, & (5)\\
& E_a \le \bar{E}_a, & & \forall a \in A, & (6)\\
& b_l^a(x) \le B_a, & & \forall a \in A, \forall l \in L, & (7)\\
& B_a \le \bar{B}_a, & & \forall a \in A, & (8)\\
& \sum_{l \in L} x_l^a = 1, & & \forall a \in A & (9)
\end{aligned}
$$

The variables are:

Variable	Range	Notes
c	R	The cost coefficient for a link
x_l^a	$[0,1]$	The split ratio of each application $a \in A$ over each link $l \in L$
LU_l	$[0,1]$	The utilization in percentage for each link $l \in L$
D_a	R	The maximum delay estimated by application a over all overlay link
$f_l^a(x)$	R	The delay function that provides the delay of application a over overlay link l considering the assignment given by x. This function will be discussed in greater detail in the 4.3 section.
$e_l^a(x)$	R	The error rate function that provides the error rate of application a over overlay link l considering the assignment given by x. This function will be discussed in greater detail in the 4.4 section
$b_l^a(x)$	R	The bandwidth function that provides the required bandwidth rate of application a over overlay link l considering the assignment given by x. This function will be discussed in greater detail in the 4.5 section

The constraints are:

Constraint	Meaning
(2)	Ensures that the capacity of each link is satisfied
(3)	Computes the delay for each application over each link
(4)	Verifies the satisfaction of SLA delay requirements
(5)	Computes the error rate for each application over each link
(6)	Verifies the satisfaction of SLA error rate requirements
(7)	Computes the bandwidth usage for each application over each link
(8)	Verifies the satisfaction of SLA bandwidth requirements
(9)	Ensures that all pending packets are sent

4.3 Estimating the delay

The most intuitive approach for solving a linear optimization problem would be to keep using the cheapest links that meet the requirements to send the incoming packets.
The problem with this approach is that overusing an interface will inevitably lead to congestion. While we know the upper bound of the bandwidth of any given link (i.e $B_l, \forall l \in L$), we must also take into account the reduction of bandwidth caused by the packets we previously sent.

In this paper we will rely on a very basic delay function, that it is purely based on how many packets arrived on a link up to a point in time.

When estimating the delay of sending n packets on a given link, we can rely on the arrival rate up to that instant of time to approximate how much load is on a current link.

$$D_a = \frac{n}{\text{arrival rate}}$$

Where the arrival rate can be computed by considering how many packets were sent through the link up to a given point in time:

$$\text{arrival rate} = \frac{\text{total amount of packets}}{\text{current time}}$$

Estimating the delay constitutes a challenge that opens for a lot of possible extensions to the current implementation, this will be further discussed in the future work session.

4.4 Estimating the error rate

Each link has a set error rate, expressed as a percentage, that affects how many packets actually reach the destination. For the scope of this paper the error rate will be fixed for every application and for every link.
However the error rate can potentially be any user-defined function to account for possibile changes in the links (e.g the error rate on a given link could rise when incoming traffic goes over a certain threshold)

4.5 Estimating the bandwidth usage

Each application has a pre-determined bandwidth usage, so we for the purpose of this paper there is no need to estimate what the necessary bandwidth for an application is. (i.e our function always returns a fixed value that is set by the application, in the case of a sinusoidal traffic model we will be considering the peak of the bandwidth usage as the required bandwidth)
However in other scenarios this function could be redefined to try and guess the future required bandwidth of any given application.

5 Implementation

5.1 Emulating a network of computers

To test the strategy we defined we setup a simple network with two nodes:

1. A source node representing the LAN adopting a SD-WAN connectivity
2. An arbitrary node to which the data is sent: in a real life scenario it would be another LAN (e.g another branch); in our case it will be a sink node used to collect data about the strategy (i.e how many packets were received, from which application they were generated ...)

The two nodes will then have an arbitrary amount of links connecting them: each link represent an element of $l \in L$.

Once the nodes are connected we have to define the application set A. Each application is a piece of code that may generate any amount of packets at any point in time, until it finally stops.

At a very high level the code for the simulation is:

Algorithm 1 The simulation algorithm

```
while $ShouldBeRunning$ do
    $Result \leftarrow executeStrategy(A, L)$
    $enqueuePackages(Result)$
    if there are no longer pending packages and all applications have stopped then
        $ShouldBeRunning \leftarrow false$
    end if
end while
```

When running the experiments, the applications will stop after having generated a pre-determined amount of traffic. The applications will run (and therefore generate traffic) for a set amount of time and the simulation will stop when all the packets will have been sent.

5.2 Implementing the simulation

The simulations will be implemented using the C++ programming language, the ns3 simulator and the or-tools library.

The simulator code is available on GitHub. [7]

5.2.1 Discrete event simulation

Using a discrete-event simulation we can model all events in our system.
With event we generally mean any piece of code that has to be ran at a specific time and alters the state of the emulated system (i.e alters the state of the network).
The simulator keeps track of when each event should be executed and automatically updates the state of system accordingly.

For example, if the event is *send packet on link l_1 to the sink*, the simulator updates the state of the system with the information that there is a packet being sent on l_1.

The system will then react to the new state: to continue on with our example after the necessary time for the packet to traverse the link will have passed the state of the system will be updated with the information that the sink has received a packet.

It is important to note that the time used by the simulator is more often than not unrelated to actual physical time: we may want to emulate a really slow connection that takes up seconds to send a packet but the actual time to run the simulation will most likely be in milliseconds.

The opposite could also be true, we may want to emulate a really fast connection were gigabytes of data are sent through a link.

We could potentially then implement a sink that prints all the received data and takes minutes to do so, but no *simulator* time will have passed (printing data is something that is outside the bounds of the simulation)

Discrete event simulator are usually built to keep an internal timer based only on the events that alter the state of the simulated system.

5.2.2 NS3

One very popular choice for network discrete-event simulation in networks is *NS3* [8].
NS3 is a discrete-event simulator written with the C++ programming language which allows for easy simulation of networks. Most components that constitute a network can easily be emulated with NS3: in our case we can setup two nodes and connect them using an arbitrary amount of links.

All the technical details such as setting up the routing, defining the error rate of a given link, the IP addresses etc... can be handled with the tools offered by NS3.

We will be using NS3 for all the experiments: while the setup for our simulation is tailored to meet our use case, it can easily be modified for future works (e.g we use a static percentage-based error rate for our links, but NS3 allows defining any arbitrary error model that can then be applied to a link)

5.2.3 OR Tools

As we previously stated, the core of the strategy is solving a linear optimization problem.
A popular tools for handling linear optimization problem is *OR-Tools* [9].
According to its creators, OR-Tools is a *an open-source, fast and portable software suite for solving combinatorial optimization problems.*. Using OR-Tools we can offload the task of solving the linear optimization problem.

5.2.4 CMake

Such as with most moderately complex projects, code from multiple sources has to be organized and then compiled together.
Unlike other programming languages, C++ does not have an official tool for defining projects and their dependencies.
One of the most popular software build system is *CMake* [10]: we will rely on CMake to combine the custom code (i.e the strategy) and the dependencies (NS3 and or-tools). CMake relies on a project file (the *MakeFile*) that defines the project structure (i.e which files are needed for the compilation, what version of C++ to use).
What CMake outputs after being ran is not an executable or a library, but rather a series of files that can then be compiled in a variety of ways (e.g Visual Studio on Windows, XCode on a Mac...).
Our tools of choice will be *Ninja*, which is a program designed to work with files generated by other tools (such as CMake) that allows compiling the project on a variety of platforms;

5.2.5 Ninja

As we previously said *Ninja is a small build system with a focus on speed. It differs from other build systems in two major respects: it is designed to have its input files generated by a higher-level build system, and it is designed to run builds as fast as possible.* [11]
Using Ninja in conjunction with CMake allows for a quick development while style maintaining cross platform compatibility.

5.2.6 Ubuntu

While it is technically possibile to build and run NS3 on Windows, its creator suggests using Linux [12] .
Our distribution of choice will be Ubuntu [13], due to its user friendliness and ease of installing.

5.2.7 Random number generation

Random generation of integers will be used in the simulation:

1. To generate different traffic sinusoids to observe how the strategies behave in different scenarios.
2. To determine which packets won't be sent through an interface due to its error rate
3. To determine which interface to use in the random strategy.

The challenge in generating random values is in combining two opposite goals:

- Generating a sequence of values where it is not predictable to guess what the next number will be.
- Generating a sequence of values that can be easily replicated (i.e we may want to modify the strategy and test it again with the same set of data)

We will rely on the C++ standard library *default_random_engine* [14] to generate most of the random values (some of the random values are handled directly by the NS3 libraries).
To have control both on our random generated values and the ones generated by NS3 we will

use a *seed* that can be changed by the user (A *seed* is a value which when input into a random generator is guaranteed to generate the same sequence.).

Utils.h

```
const uint32_t SeedForRandomGeneration = 1;
```

main.cpp

```
std::default_random_engine gen;
gen.seed(Utils::SeedForRandomGeneration);
```

6 Testing strategy

6.1 High level view

To test the strategy we will consider a scenario with three applications generating data and we will compare how different strategies perform when trying to send data to a sink by using three interfaces.
We will run multiple simulations with different randomly generated characteristics; this will discussed in the 6.3 section. Multiple simulation will be ran on a fixed 1200 seconds timeframe. At each second in each simulation every application we will be sampled and any generated package will be queued according to the current strategy.
The testing report will be generated using the aggregated data coming from multiple application runs.

6.2 The interfaces

We will be using the following interfaces to test our strategy:

#	bandwidth (packets per second)	error rate (as a percentage)	delay (ms)	cost (per packet sent)
1	10	20%	150	10
2	20	10%	100	20
3	30	1%	30	100

By design, at any given time all the traffic generated from the various application can always be sent by using a subset of the available interfaces.
Discussing strategies where the interfaces are sometimes unable to handle all the incoming traffic goes beyond the goal of this paper

6.3 The applications

We will be using three applications that generate traffic in a sinusoidal pattern.
While the delay requirements of the applications will be fixed, other characteristics will be randomly generated for each run.

#	maximum allowed delay (ms)	peak bandwidth requirement (packets per second)	maximum allowed error rate (percentage)	maximum noise (packets)	shift (s)
1	200	randomly generated value (1-12)	20	randomly generated value (1-3)	randomly generated value (50-150)
2	150	randomly generated value (1-12)	10	randomly generated value (1-3)	randomly generated value (50-150)
3	100	randomly generated value (1-12)	5	randomly generated value (1-3)	randomly generated value (50-150)

The variation in shift, peak bandwidth requirement and noise can lead to significantly different traffic profiles.

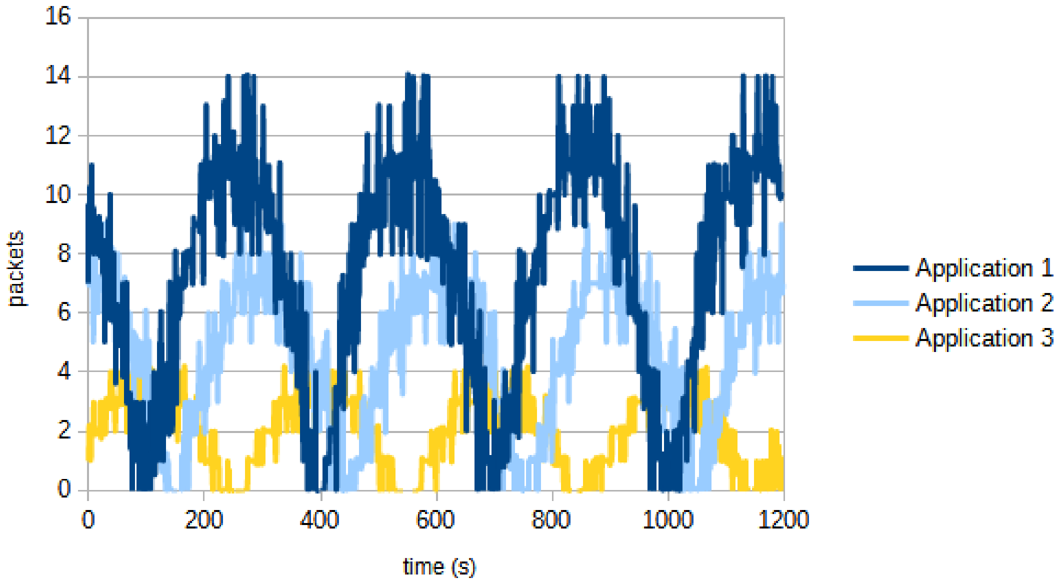

Figure 2: A sample traffic profile with noise and shift on all three applications

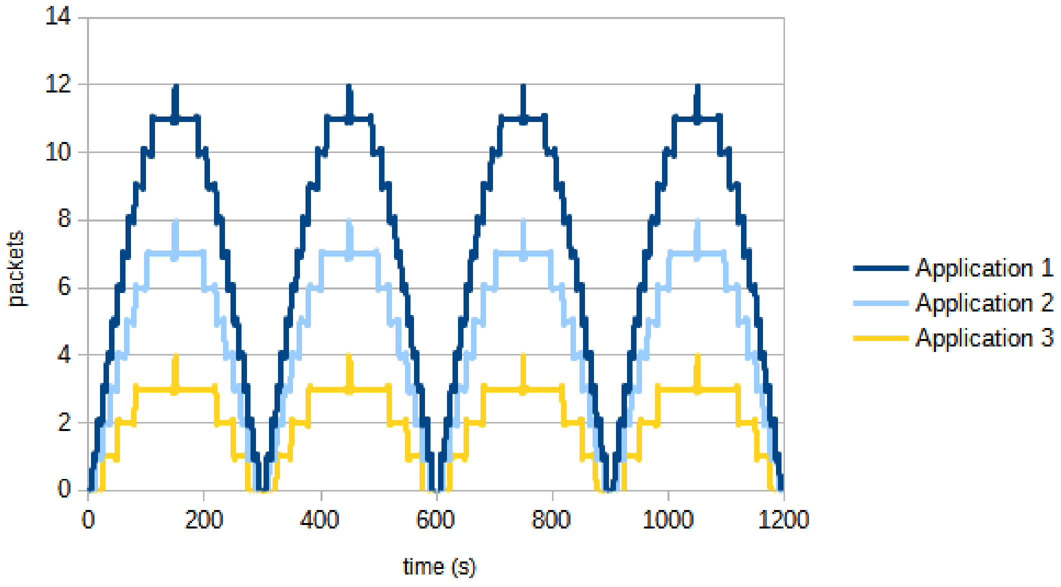

Figure 3: A sample traffic profile without any shift or noise

6.4 Comparison with baseline strategies

We will be comparing our strategy against three other strategies:

- Round robin strategy.
  This strategy will manage the packets generated by the different application by sending a packet to each interface, looping until there are no more packages. e.g if four packets are generated by an application:
  - The first packet will be sent on the first interface
  - The second packet will be sent on the second interface
  - The third packet will be sent on the third interface.
  - The fourth packet will be sent on the first interface.
  - etc ...
- Random strategy.
  This strategy will send any incoming to a randomly chosen interface. For more details about how random generation is handled, see section 5.2.7
- Static Strategy
  This strategy will use a modified version of the initial strategy discussed in section 3.1.2. This modified version will still not account for any delay caused by the excessive use of a given interface, but will be ran with all the three applications at the same time.

We will run each strategy for 500 times, each time with a randomized amount of shift, peak bandwidth usage and noise.

7 Results

7.1 Round robin strategy

Our reactive strategy performs better than a round robin strategy in most cases, guaranteeing a better cost per packet.

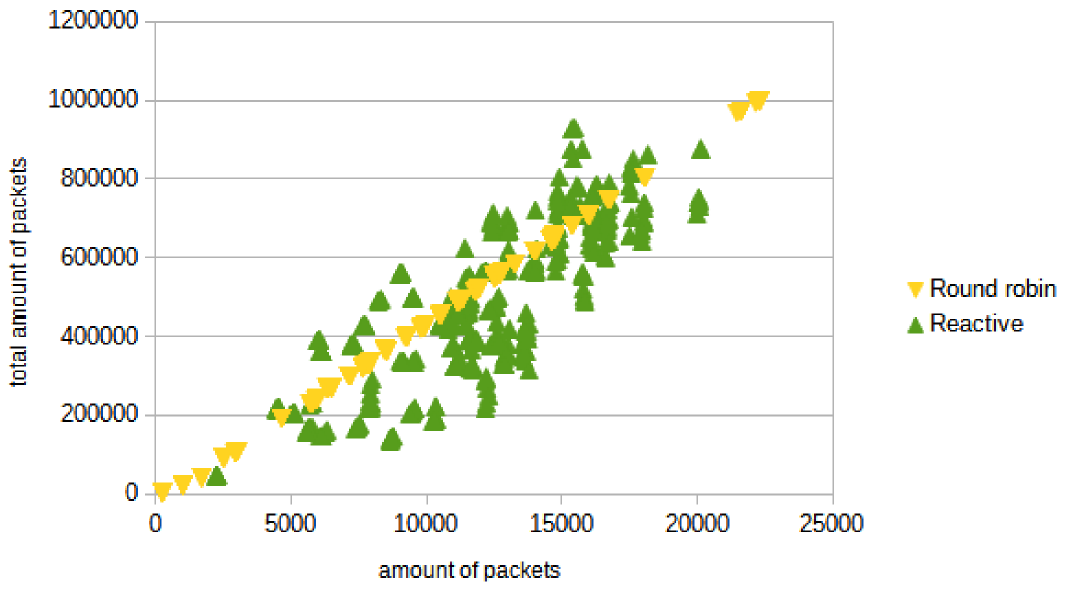

Figure 4: Comparison of round robin strategy in term of cost per packet

	Round Robin	Reactive strategy
Average packet cost	43.4	40.1

While the improvement in costs may not be dramatic, it is important to note that a round robin strategy does not offer any guarantee of satisfying SLA requirements.
As we previously said the round robin strategy sends a packet through each interface in a loop: for this reason we expected the average error rate and delay to be the average of the error rates and delays of the various interfaces.
This leads to failures in guaranteeing SLA requirements for any application for which the requirements are higher than the average error rate or delay.

The average error rate of the interfaces amounts to 11%; with this configuration the error rate requirements for the first application are easily satisfied; however for the second and third interface the SLA satisfaction rate drops sharply.

	Round Robin	Reactive strategy
SLA satisfied (Application 1)	99%	98%
SLA satisfied (Application 2)	73%	95%
SLA satisfied (Application 3)	0%	100%

7.2 Random strategy

By its nature, the random strategy will distribute the packets equally among all the available interfaces across a long enough time span.
This causes the random strategy to perform in a similar way to the round robin strategy (i.e each interface will get approximately one third of the total packages).

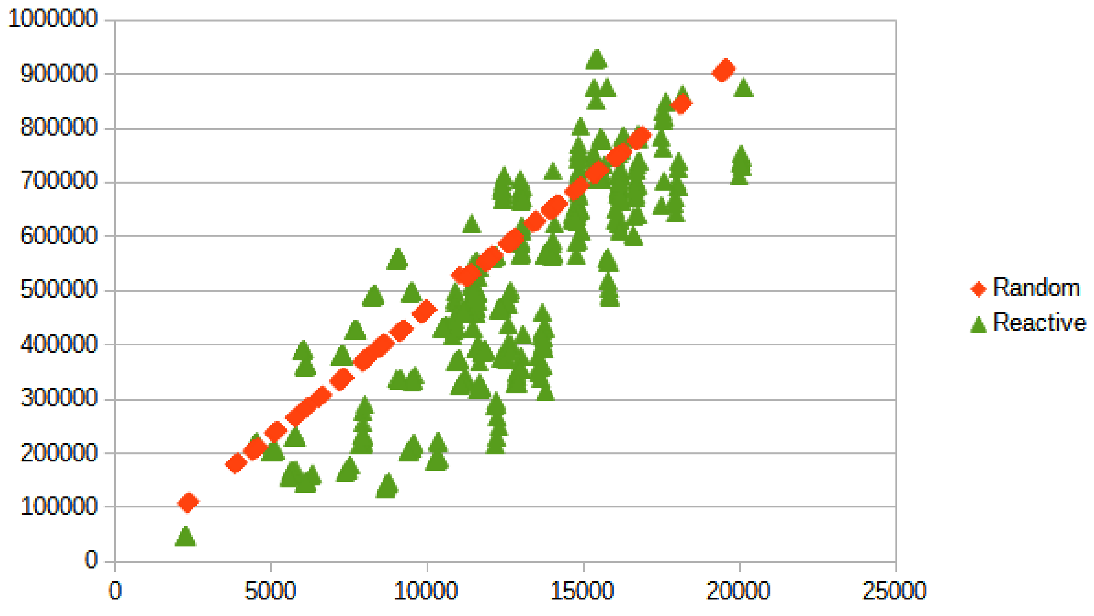

Figure 5: Comparison with random strategy in term of cost per packet

	Random	Reactive strategy
Average packet cost	46.5	40.1

With the current dataset we notice that the random strategy does not offer any improvement even if we just look at the cost.

	Random	Reactive strategy
SLA satisfied (Application 1)	99%	98%
SLA satisfied (Application 2)	90 %	95%
SLA satisfied (Application 3)	0%	100%

The SLA satisfaction results are in line with the expectations: the ability to satisfy requirements drops sharply when the application requirements are stricter than the average of the interfaces capabilities.

7.3 Static strategy

This strategy works with the very heavy assumption that all the traffic coming from an application is known beforehand.
One other important assumption is to that all the packets queued to a given interface are sent instantly; in fact we are assuming that all the interfaces have infinite bandwidth (the chosen interfaces still have to meet the SLA requirements: e.g if an application has a SLA requirements of 20 packet per second the strategy will still choose a set interfaces whose bandwidths averages out to 20 packet per second)

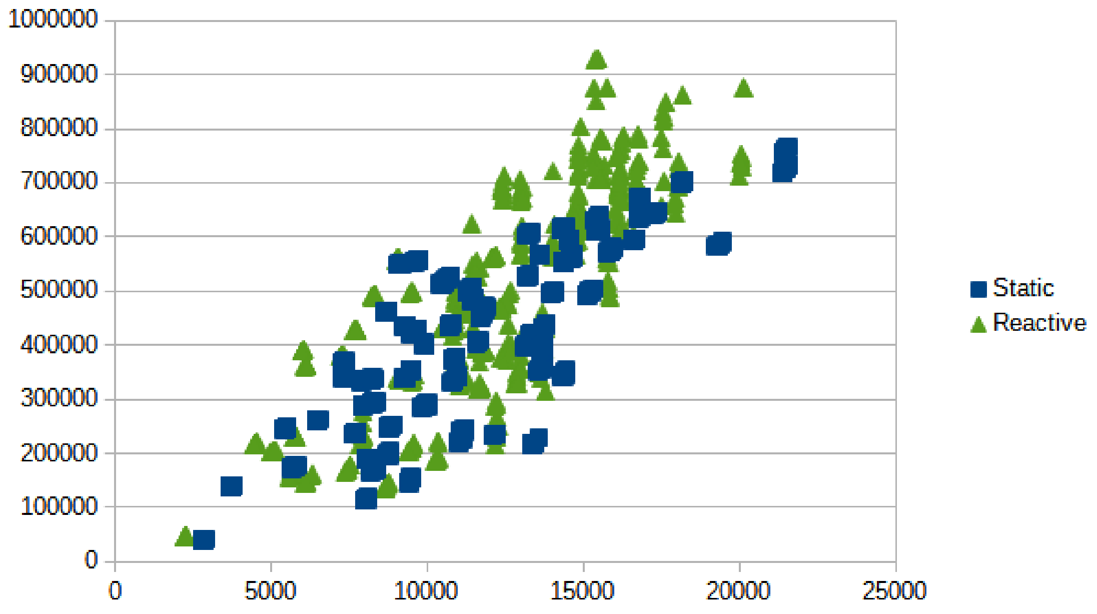

Figure 6: Comparison with the static strategy in term of cost per packet

	Static	Reactive strategy
Average packet cost	36.05	40.1

We notice that on average the reactive strategy is 11% more expensive than the ideal case.

	Static	Reactive
SLA satisfied (Application 1)	100%	98%
SLA satisfied (Application 2)	94 %	95%
SLA satisfied (Application 3)	0%	100%

Due to the randomness of determining which packets are not sent successfully (i.e applying the error rate) we notice that even with a theoretical perfect strategy SLA requirements may still not be respected; in practice this could be resolve by artificially decreasing the allowed error rate to account for variance.

8 Related work

Traffic engineering in SD-WAN is a widely researched topic, due to its potential in reducing cots and improving traffic efficiency.
These papers contain similar or alternative approaches to tackle some of the challenges in engineering traffic in SD-WAN.
While some of the topics treated in these papers are not strictly related to our goals (reducing costs) they still offer some extremely valuable information in term of how to approach problems when designing traffic engineering solutions

8.1 Machine learning

8.1.1 On Deep Reinforcement Learning for Traffic Engineering in SD-WAN [1]

This paper compares three different deep Reinforcement Learning (deep-RL) algorithms to try to overcome the limitations of on traditional approaches (e.g algorithms based on QoS thresholds) when managing traffic in a SD-WAN context. Specifically, three kinds of deep-RL algorithms are tested, which are: policy gradient, TD- and deep Q-learning. Results show that a deep-RL algorithm with a well-designed reward function is capable of increasing the overall network availability.

8.1.2 Load Balancing Optimization in Software-Defined Wide Area Networking (SD-WAN) using Deep Reinforcement Learning [2]

This paper tries using a gradient-based deep learning approach to improve load balancing under latency constraints. The balancing problem is formulated as a linear programming problem, where we want to minimize the load across the controllers and minimize the migration cost of CPEs to new controllers.

8.2 Heuristics

8.2.1 On the placement of controllers in software-Defined-WAN using meta-heuristic approach [3]

This paper explores an approach to the placing of nodes and controllers and how possible failures in links are handled in a SD-WAN context. The controller placement problem (CPP) is considered as a multi-objective combinatorial optimization problem and it is solved using two population-based meta-heuristic techniques such as: Particle Swarm Optimization (PSO) and FireFly Algorithm (FFA). The performance of the algorithms is evaluated on a set of publicly available network topologies in order to obtain the optimum number of controllers, and controller positions. By comparing the performance of the presented scheme to a competing scheme, it was found that the proposed scheme effectively improves the survivability of the control path and the performance of the network as well.

8.2.2 Multi-Controller Placement for Load Balancing in SDWAN [4]

This paper proposes a controller based algorithm called Simulated Annealing Partition-based K-Means (SAPKM). Two cost functions are used to assess the efficiency of the proposed algorithm from the perspective of topology structure and flow traffic distribution, respectively. This paper uses real world networks to test the algorithm performance (http://www.topology-zoo.org/explore.html)

8.3 Linear programming

8.3.1 Research of Improved Traffic Engineering Fault-Tolerant Routing Mechanism in SD-WAN [5]

This paper introduces a mathematical model used to describe an SD-WAN data plane to address possible fault tolerance issues while routing traffic. This model allows each access network to be switched simultaneously not to one but several border routers, increasing fault tolerance. The fault-tolerant routing problem is presented in optimization form as a linear programming problem with an optimality criterion and constraints

References

[1] Sebastian Troia, Federico Sapienza, Leonardo Varé, and Guido Maier. On deep reinforcement learning for traffic engineering in sd-wan. *IEEE Journal on Selected Areas in Communications*, 39(7):2198–2212, 2021.

[2] Mohamed Amine Ouamri, Gordana Barb, Daljeet Singh, and Florin Alexa. Load balancing optimization in software-defined wide area networking (sd-wan) using deep reinforcement learning, 2022.

[3] Kshira Sagar Sahoo, Deepak Puthal, Mohammad S. Obaidat, Anamay Sarkar, Sambit Kumar Mishra, and Bibhudatta Sahoo. On the placement of controllers in software-defined-wan using meta-heuristic approach. *Journal of Systems and Software*, 145:180–194, 2018.

[4] Kongzhe Yang, Daoxing Guo, Bangning Zhang, and Bing Zhao. Multi-controller placement for load balancing in sdwan. *IEEE Access*, 7:167278–167289, 2019.

[5] Oleksandr Lemeshko, Oleksandra Yeremenko, Maryna Yevdokymenko, Anna Zhuravlova, Anastasiia Kruhlova, and Valentyn Lemeshko. Research of improved traffic engineering fault-tolerant routing mechanism in sd-wan. pages 187–190, 2021.

[6] Pham Tran Anh Quang et al. Intent-based routing policy optimization in sd-wan. 2022.

[7] Simulator source code. `thesis_source_code`, last accessed on 04/03/2025.

[8] Ns3. `https://www.nsnam.org/`, last accessed on 04/03/2025.

[9] Or-tools. `https://developers.google.com/optimization?hl=it`, last accessed on 04/03/2025.

[10] Cmake. `https://cmake.org/`, last accessed on 04/03/2025.

[11] Ninja. `https://ninja-build.org/`, last accessed on 04/03/2025.

[12] Ns3 documentation. `https://www.nsnam.org/docs/installation/html/`, last accessed on 04/03/2025.

[13] Ubuntu. `https://ubuntu.com/`, last accessed on 04/03/2025.

[14] C++ documentation on default random engine. `https://cplusplus.com/reference/random/default_random_engine/`, last accessed on 04/03/2025.