Computer Networks Design – Lab 1

Project Goal: Implement a mini-network simulator to evaluate the performance of layer 2 switches

The main goal of this lab (and the following) is to create the *infrastructure* required for the network simulator. In the following labs, we will improve the infrastructure, add more functionality, such as scheduling disciplines (later in the course), and evaluate the performance of the switches.

Before we start, we have a few remarks:

- The parts should be done sequentially. Part A in Lab 1 (A1 for short) is implemented first, then Part B in Lab 1 (B1). These parts *will be the baseline for* A2, B2, etc., so ensure they work properly.
  Each part will have *defenses* to ensure it works "in the bigger picture." This is similar to other lab courses you might have had in the past.
- Ensure you have a properly functioning test code. ***Each test should use a separate file.***
- While not obligatory, we highly recommend planning your code before writing it (even if it is a small flow chart of an algorithm). The order inside each part is irrelevant, so reading each part entirely can help you plan your work better.
- The instructions are not *super specific,* unlike other assignments you might have had in other courses. This is not to confuse you but rather to let you program in any convenient ***but readable*** way you wish.
- ***Document your code.***
- You can use Python, C++, C, MATLAB, or Wolfram Mathematica for your simulations. We recommend using Object-Oriented Programming (OOP) to ease the testing and writing process.
- Each "important" component should use a "prints ON/OFF" variable. Some may use more than one. Having the option to turn off prints is a good practice since printing, although convenient for testing and debugging, substantially slows your simulation.
- Plagiarism is unacceptable under any circumstances. Code copied from an answer in Stack Exchange, GitHub, AI Chats, etc., should be credited. Using another student's work for reference (in this course or past courses) is prohibited.

Part A: Host-to-Host Link

In this part, you are tasked to write a simulation of a Layer 2 link between two hosts. See the following image:

We are not implementing the whole internet stack; instead, we implement only layer 2 (Link Layer) with some hints of layer 1 (Physical Layer). ***ARP is not used***.

We list the basic guidelines below; you may use additional variables (or different names) and parameters as you see fit. However, the listed values must be present in your code.

Timeline, Events, and Object IDs

Since real-time simulations can be tedious, many simulations use their own timelines. For this purpose, many libraries exist, e.g., Python's SimPy (do not confuse it with Python's symbolic calculations, SymPy). Such libraries are much more powerful than needed for this project, so we implement a slightly more primitive timeline instead. The main idea is to put all the necessary information to parse events into an Event Object, including its simulation time, creator Object, and target Object.

Each object should have its *unique* Object ID. Using the natural number for this is recommended, where the first IDs are used for all Hosts, Links, etc. L2 Messages are created "on the fly" and should use the following IDs. For example, in Part A's Host-to-Host topology, Host 1 gets ID0, Host 2 gets ID1, and the Link gets ID2. All L2 Messages start with ID3+.

All objects should be stored in a dynamic list/array. This would allow you to accelerate your code.
Remark: *For OOP in C++, we recommend creating a base class whose only fields are ID and type (Host, Link, L2 Message, etc.). All other classes inherit from it. This would ease the dynamic array handling and casting.*

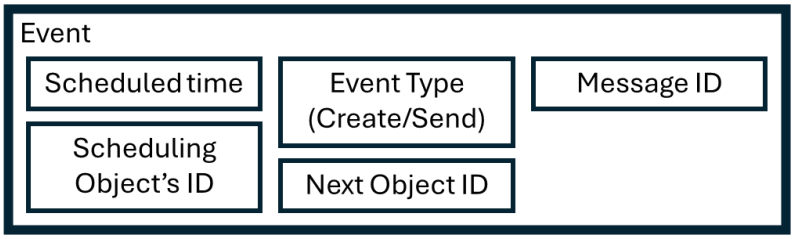
Events must have:

1. **Scheduled Time:** the (simulation) time the event occurs. All events are sorted and inserted into the timeline by their scheduled time. The creation of the timeline is discussed later.
2. **Event Type:** A variable that classifies the event type. It can be set to "create a message" or "send a message."
3. **Scheduling Object's ID:** The event creator's ID. In other words, it is the ID of the object that is scheduled to create or send a message at the time given in "Scheduled Time."
4. **Target Object's ID:** The ID of the object that the event ends at.
5. **Message ID:** The relevant L2 Message's ID if it exists. It is set to "None" if the Event Type is "create a message."

We provide examples for points 1, 3, and 4:

1. An event with a scheduled time of t=15.1 occurs before an event with a scheduled time of t=15.2. It is possible, however, that the event at t=15.1 was created ***after*** the event scheduled at t=15.2.
3. Assume an event of type "create a message" comes. The scheduling object ID is 0, and the target ID is 2. Then, Host 1 has scheduled the event in the past and now needs to create an L2 Message to send to the Link (whose ID is 2). The Message ID is not checked.
4. Assume an event of type "send a message" comes. The scheduling object ID is 2, and the target ID is 1. Then, the Link has scheduled the event in the past and now needs to send an L2 Message (whose ID is given in Message ID) to Host 2 (whose ID is 1).

Visualization:



How to Create a Timeline?

Creating a timeline consists of several stages:

1. For each Host, generate the times between the L2 Messages. For example:

Host 1:	0.1	0.3	0.2	0.5
Host 2:	0.3	0.4	0.4	0.1

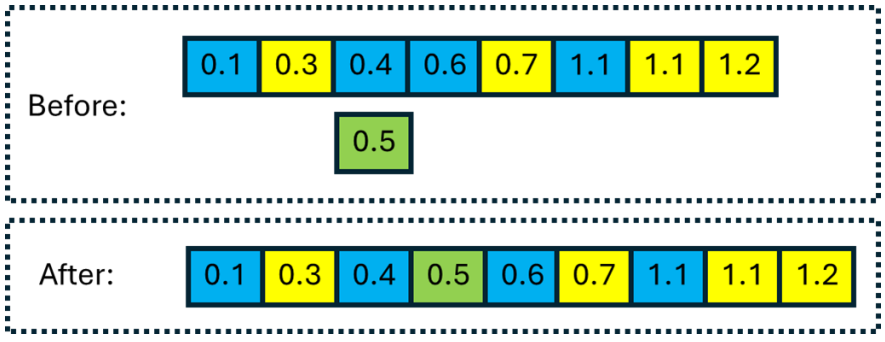
2. For each Host, create a cumulative array of times. For example:

Host 1:	0.1	0.4	0.6	1.1
Host 2:	0.3	0.7	1.1	1.2

3. Merge all arrays. Ties are broken by the lowest ID. For example:

0.1	0.3	0.4	0.6	0.7	1.1	1.1	1.2

Consider a case where a new event is scheduled to the existing timeline. In the above example, an event at t=0.5sec is scheduled. All we need is to insert it to the correct position:

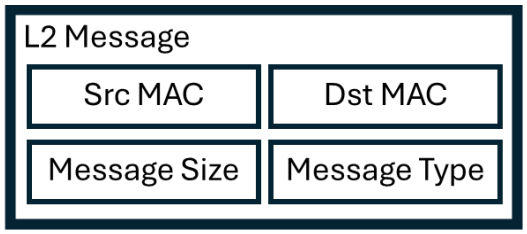


L2 Message

A message is an object that "passes" from one host to another and simulates information sent. A message is characterized by source MAC, destination MAC, message size (*in Bytes*), and message type (data/type). Any L2 Message must have:

1. **Source MAC Address:** The sender's MAC address
2. **Destination MAC Address:** The receiver's MAC address
3. **Message Size:** The L2 Message's size *in Bytes* (including ALL headers). Please refer to the Layer 2 Ethernet Frame for minimal and maximal message size. We do not use VLANs.
4. **Message Type:** A field always set to "data." Simulates the EtherType flag in an Ethernet Frame.

Visualization:

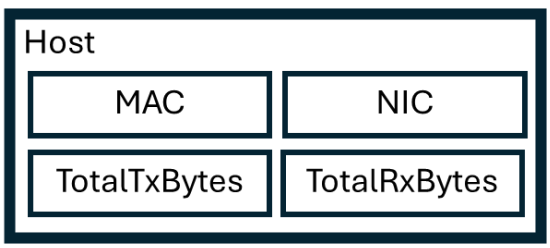


Host

A host is an object that both *creates and destroys* L2 messages. It is characterized by MAC address. The Host collects data for statistics and analysis. The Host must have:

1. **MAC Address:** The host's MAC address.
2. **Network Interface Card (NIC):** The link's ID.
3. **Total Bytes Sent:** The sum of all information bytes sent (without headers).
4. **Total Bytes Received:** The sum of all information bytes received (without headers).

Visualization:



The host must support the following functionality (*printed text can be edited to your liking*):

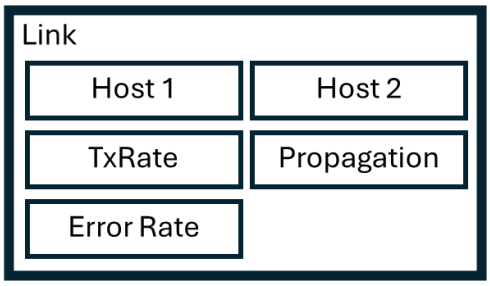
1. **Create an L2 Message:** Create an L2 Message and put its MAC address as the L2 Message's Source MAC Address. The Destination is drawn uniformly from all the *other* hosts.
   The payload size is a uniform random integer between the minimal and maximal payload size in Ethernet, drawn with each creation.
   Don't forget to update the total bytes sent!
   The L2 Message's Message Type is set to "data."
   If the printing flag is ON, print "Host <MAC Address> created an L2 Message (size: <Size in Bytes>)"
2. **Receive an L2 Message:** The Host receives an L2 Message and updates its statistics. Afterwards, it destroys the L2 Message.
   If the printing flag is ON, print "Host <MAC Address> destroyed an L2 Message (size: <Size in Bytes>)"
3. **Print statistics at the end:** At the end of the simulation, if the printing flag is ON, the Host prints how many bytes it sent and received.

Link

A link has 2 end-points and is characterized by rate. The links are Full-Duplex, so there is no need to worry about two Hosts simultaneously sending L2 Messages. In other words, any link must have the following:

1. **A first end-point:** Some host's ID
2. **A second end-point:** The other host's ID
3. **Transmission rate:** The transmission rate in *Bits Per Second*. This is used to calculate the transmission time based on the L2 Message's Message Size.
4. **Propagation delay:** Some constant delay added to the previous result. Set it to 0.
5. **Error rate:** The error probability of the link. Set it to 0.

Visualization:



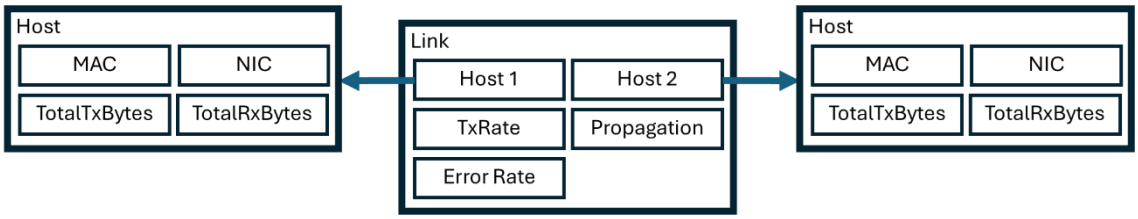
Remarks:

i. Should there be flags to disable sending two or more messages simultaneously? Fluid models (later on) will allow this. Do you think there should be an "Is Fluid?" flag?
ii. If the model is not fluid, where do we put the older message? Queue?

Testing Part A

Test 1 – Can Host 1 Reach Host 2?

Recreate the Hots-to-Host topology:



*The arrows are not necessarily C++ pointers.

Now, put the print flag ON on each object and do the proceeding test:

1. At t=0, let Host 1 schedule itself to create a new L2 Message at an exponentially distributed random time, likewise for Host 2.
2. Check the timeline for the first event, and let the relevant Host create an L2 Message and send it (i.e., schedule it immediately) to the Link. Print the simulation creation time of each message.
3. Have the Link relay the L2 Message to the other Host by scheduling it in the current time + propagation and transmission delays.
4. Continue the simulation until termination. Each Host destroys the message it receives. Print the destruction time of each message.

Make sure you understand the following:

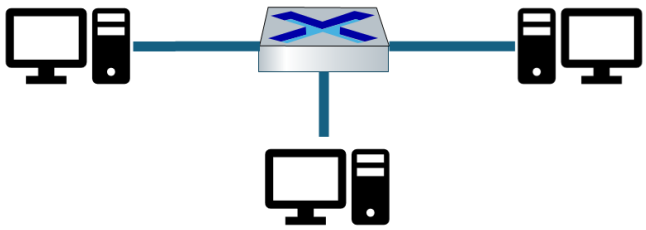
- Why is ***this*** test bound to terminate? Is it possible for it to never end?
- How can we ensure that a simulation stops at t=10sec?
- Which Host receives an L2 Message first? Does this Host always obtain a message first? Why?
- Why do we immediately schedule the L2 Message's transmission to the Link in step 3?

Test 2 – Let it Flow

Extend test 1 so that each Host creates a flow (i.e., sends multiple messages) of 10000 L2 Messages. Do not print anything except the **average** amount of bytes received.

Part B: Switch Baseline

In this part, you are tasked to write a simulation of a LAN with three hosts connected via a learning switch. See the following image:



The main focus of this part is the L2 Learning Switch, or Switch for short. The Switch object learns MAC addresses and associates them with its ports, floods the network when needed, and, most importantly, relays messages from one host to another.

Switch

The Switch must have:

1. **(Numbered) I/O Ports:** The Switch's number of ports. The convention is that this parameter is a power of 2. This is an array of link IDs and "None"s (in case a port is unused).
2. **MAC Table:** A table with entries "Used? | MAC Address | Port | TTL" updated in run-time. When an L2 Message is received from port N, the MAC Table registers the Source MAC Address, port N, and the simulation time *the message was received* (see optimization tip: Efficient TTL Testing). For example, L2 Message from Port 2 with source 02:00:FE:E1:90:0D at simulation time t=11.33 is saved as:

True	02:00:FE:E1:90:0D	2	11.33

The main challenge of this part is handling the MAC Table. Here, we provide the basic functionality (and tips) of the flooding and MAC Table:

1. **MAC Table Size:** The table must be small (e.g., up to 10 entries) since the port look-up process is essentially linear probing.
2. **Efficient TTL Testing:** Whenever an L2 Message is received, the Switch looks for the Source MAC Address in its MAC Table. The search is treated as failed if it is not present, OR is present, and **|simulation time – time_at_table| >= TTL**.

3. **Adding and Updating New Entries:** The above calculation applies to finding the first expired/empty entry when putting a new entry into the table.
   If an L2 Message is received, and its entry has yet to expire, update its reception time (in the MAC Table) to refresh the entry. *Remember to update the input port, too.*
4. **Flooding:** When flooding, duplicate (in OOP, the Copy Constructor is your friend) the received L2 Message and schedule it for transmission immediately.
   Remember to update the message's IDs so they remain unique.
   Remember that when the Switch floods the network, it does not send the received L2 Message from the port it came from.
5. **Uninitialized Entries:** If an entry was never used, set the "Used?" flag as False. An entry is empty if this flag is False. In other words, test this flag *first* when looking for the first empty or expired entry.

Here are some questions to help you design your Switch better:

- Aside from the "Used" flag, should you bother initializing the other MAC Table fields?
- When should we print the MAC Table for debugging? When printing, should we print the TTL or the "is valid" test result (for each entry) from the optimization tip "Efficient TTL Testing?"
  Should empty or invalid entries printed?
- How does a Switch know which port is connected to some network component and which is not?
  This is crucial when handling flooding.
- How does the Switch know which from which port an L2 Message came from?
  This is crucial when handling flooding.
- In the future, you will implement queues in the switch (either at the input or output), so plan your code accordingly.

Network Visualization

Write a function to visualize the network. Using Python's NetworkX or C++'s Qt is recommended.

Testing Part B

Test 1 – Basic LAN

With all printing flags ON, recreate “Test 1 – Can Host 1 Reach Host 2?” but with three hosts connected to a switch like in the topology attached at the beginning of Part B. Show us the evolution of the Switch’s MAC Table.

Test 2 – Entries Edge Cases

Create the following topology:

Here, clouds represent a uniform random number of hosts (between 2 and 4), each with its dedicated Link. Set S1’s MAC Table size as the **total** number of hosts -2. Repeat the previous test, and show us the evolution of S1’s (the right switch) MAC Table.