1. 課題

参考文献 [1] の通り，次の 3 つの課題に取り組む.

1.1. 課題 1

1.2. 演習課題 2：ネットワークの遅延とスループットの測定

かきくけこ

1.3. 演習課題 3：パケットロス率とネットワークパフォーマンスの関係

さしすせそ

2. 実験設定

以下の実験環境で実行した.

表 1: 実験環境

実行環境	Google Colob
言語	Python

3. 解答

3.1. 演習課題 1

まず，実行したプログラムを以下に示す.

```python
import matplotlib.pyplot as plt
import networkx as nx
import heapq
import uuid
import random
import numpy as np
from collections import defaultdict
import sys
from sec2.NetworkEventScheduler import NetworkEventScheduler
from sec2.Node import Node
from sec2.Link import Link

class NetworkEventScheduler:
    def __init__(self, log_enabled=False, verbose=False):
        self.current_time = 0
        self.events = []
        self.event_id = 0
        self.packet_logs = {}
        self.log_enabled = log_enabled
        self.verbose = verbose
        self.graph = nx.Graph()

    def add_node(self, node_id, label):
        self.graph.add_node(node_id, label=label)

    def add_link(self, node1_id, node2_id, label, bandwidth, delay):
```

```
        self.graph.add_edge(node1_id, node2_id, label=label,
bandwidth=bandwidth, delay=delay)

    def draw(self):
        def get_edge_width(bandwidth):
            return np.log10(bandwidth) + 1

        def get_edge_color(delay):
            if delay <= 0.001:  # <= 1ms
                return 'green'
            elif delay <= 0.01:  # 1-10ms
                return 'yellow'
            else:  # >= 10ms
                return 'red'

        pos = nx.spring_layout(self.graph)
        edge_widths = [get_edge_width(self.graph[u][v]['bandwidth']) for u,
v in self.graph.edges()]
        edge_colors = [get_edge_color(self.graph[u][v]['delay']) for u, v in
self.graph.edges()]

        nx.draw(self.graph, pos, with_labels=False, node_color='lightblue',
node_size=2000, width=edge_widths, edge_color=edge_colors)
        nx.draw_networkx_labels(self.graph, pos,
labels=nx.get_node_attributes(self.graph, 'label'))
        nx.draw_networkx_edge_labels(self.graph, pos,
edge_labels=nx.get_edge_attributes(self.graph, 'label'))
        plt.show()

    def schedule_event(self, event_time, callback, *args):
        event = (event_time, self.event_id, callback, args)
        heapq.heappush(self.events, event)
        self.event_id += 1

    def log_packet_info(self, packet, event_type, node_id=None):
        if self.log_enabled:
            if packet.id not in self.packet_logs:
                self.packet_logs[packet.id] = {
                    "source": packet.header["source"],
                    "destination": packet.header["destination"],
                    "size": packet.size,
                    "creation_time": packet.creation_time,
                    "arrival_time": packet.arrival_time,
                    "events": []
                }

            if event_type == "arrived":
                self.packet_logs[packet.id]["arrival_time"] =
self.current_time

            event_info = {
                "time": self.current_time,
                "event": event_type,
                "node_id": node_id,
                "packet_id": packet.id,
                "src": packet.header["source"],
```

```
                "dst": packet.header["destination"]
            }
            self.packet_logs[packet.id]["events"].append(event_info)

            if self.verbose:
                print(f"Time: {self.current_time} Node: {node_id}, Event:
{event_type}, Packet: {packet.id}, Src: {packet.header['source']}, Dst:
{packet.header['destination']}")


    def print_packet_logs(self):
        for packet_id, log in self.packet_logs.items():
            print(f"Packet ID: {packet_id} Src: {log['source']}
{log['creation_time']} -> Dst: {log['destination']} {log['arrival_time']}")
            for event in log['events']:
                print(f"Time: {event['time']}, Event: {event['event']}")

    def generate_summary(self, packet_logs):
        summary_data = defaultdict(lambda: {"sent_packets": 0, "sent_bytes":
0, "received_packets": 0, "received_bytes": 0, "total_delay": 0,
"lost_packets": 0, "min_creation_time": float('inf'), "max_arrival_time":
0})

        for packet_id, log in packet_logs.items():
            src_dst_pair = (log["source"], log["destination"])
            summary_data[src_dst_pair]["sent_packets"] += 1
            summary_data[src_dst_pair]["sent_bytes"] += log["size"]
            summary_data[src_dst_pair]["min_creation_time"] =
min(summary_data[src_dst_pair]["min_creation_time"], log["creation_time"])

            if "arrival_time" in log and log["arrival_time"] is not None:
                summary_data[src_dst_pair]["received_packets"] += 1
                summary_data[src_dst_pair]["received_bytes"] += log["size"]
                summary_data[src_dst_pair]["total_delay"] +=
log["arrival_time"] - log["creation_time"]
                summary_data[src_dst_pair]["max_arrival_time"] =
max(summary_data[src_dst_pair]["max_arrival_time"], log["arrival_time"])
            else:
                summary_data[src_dst_pair]["lost_packets"] += 1

        for src_dst, data in summary_data.items():
            sent_packets = data["sent_packets"]
            sent_bytes = data["sent_bytes"]
            received_packets = data["received_packets"]
            received_bytes = data["received_bytes"]
            total_delay = data["total_delay"]
            lost_packets = data["lost_packets"]
            min_creation_time = data["min_creation_time"]
            max_arrival_time = data["max_arrival_time"]

            traffic_duration = max_arrival_time - min_creation_time
            avg_throughput = (received_bytes * 8 / traffic_duration) if
traffic_duration > 0 else 0
```

```
            avg_delay = total_delay / received_packets if received_packets > 0 else 0

            print(f"Src-Dst Pair: {src_dst}")
            print(f"Total Sent Packets: {sent_packets}")
            print(f"Total Sent Bytes: {sent_bytes}")
            print(f"Total Received Packets: {received_packets}")
            print(f"Total Received Bytes: {received_bytes}")
            print(f"Average Throughput (bps): {avg_throughput}")
            print(f"Average Delay (s): {avg_delay}")
            print(f"Lost Packets: {lost_packets}\n")

    def generate_throughput_graph(self, packet_logs):
        time_slot = 1.0  # 時間スロットを1秒に固定

        max_time = max(log['arrival_time'] for log in packet_logs.values() if log['arrival_time'] is not None)
        min_time = min(log['creation_time'] for log in packet_logs.values())
        num_slots = int((max_time - min_time) / time_slot) + 1  # スロットの総数を計算

        throughput_data = defaultdict(list)
        for packet_id, log in packet_logs.items():
            if log['arrival_time'] is not None:
                src_dst_pair = (log['source'], log['destination'])
                slot_index = int((log['arrival_time'] - min_time) / time_slot)
                throughput_data[src_dst_pair].append((slot_index, log['size']))

        aggregated_throughput = defaultdict(lambda: defaultdict(int))
        for src_dst, packets in throughput_data.items():
            for slot_index in range(num_slots):
                slot_throughput = sum(size * 8 for i, size in packets if i == slot_index)
                aggregated_throughput[src_dst][slot_index] = slot_throughput / time_slot

        for src_dst, slot_data in aggregated_throughput.items():
            time_slots = list(range(num_slots))
            throughputs = [slot_data[slot] for slot in time_slots]
            times = [min_time + slot * time_slot for slot in time_slots]
            plt.step(times, throughputs, label=f'{src_dst[0]} -> {src_dst[1]}', where='post', linestyle='-', alpha=0.5, marker='o')

        plt.xlabel('Time (s)')
        plt.ylabel('Throughput (bps)')
        plt.title('Throughput over time')
        plt.xlim(0, max_time)
        plt.legend()
        plt.show()

    def generate_delay_histogram(self, packet_logs):
        delay_data = defaultdict(list)
        for packet_id, log in packet_logs.items():
            if log['arrival_time'] is not None:
```

```
                src_dst_pair = (log['source'], log['destination'])
                delay = log['arrival_time'] - log['creation_time']
                delay_data[src_dst_pair].append(delay)

        num_plots = len(delay_data)
        num_bins = 20
        fig, axs = plt.subplots(num_plots, figsize=(6, 2 * num_plots))
        max_delay = max(max(delays) for delays in delay_data.values())
        bin_width = max_delay / num_bins

        for i, (src_dst, delays) in enumerate(delay_data.items()):
            ax = axs[i] if num_plots > 1 else axs
            ax.hist(delays, bins=np.arange(0, max_delay + bin_width,
bin_width), alpha=0.5, color='royalblue', label=f'{src_dst[0]} ->
{src_dst[1]}')
            ax.set_xlabel('Delay (s)')
            ax.set_ylabel('Frequency')
            ax.set_title(f'Delay histogram for {src_dst[0]} ->
{src_dst[1]}')
            ax.set_xlim(0, max_delay)
            ax.legend()

        plt.tight_layout()
        plt.show()

    def run(self):
        while self.events:
            event_time, _, callback, args = heapq.heappop(self.events)
            self.current_time = event_time
            callback(*args)

    def run_until(self, end_time):
        while self.events and self.events[0][0] <= end_time:
            event_time, callback, args = heapq.heappop(self.events)
            self.current_time = event_time
            callback(*args)

class Node:
    def __init__(self, node_id, address, network_event_scheduler):
        self.network_event_scheduler = network_event_scheduler
        self.node_id = node_id
        self.address = address
        self.links = []
        label = f'Node {node_id}\n{address}'
        self.network_event_scheduler.add_node(node_id, label)

    def add_link(self, link):
        if link not in self.links:
            self.links.append(link)

    def receive_packet(self, packet):
        if packet.arrival_time == -1:
            self.network_event_scheduler.log_packet_info(packet, "lost",
self.node_id)  # パケットロスをログに記録
            return
        if packet.header["destination"] == self.address:
```

```
                self.network_event_scheduler.log_packet_info(packet, "arrived",
self.node_id)  # パケット受信をログに記録
                packet.set_arrived(self.network_event_scheduler.current_time)
            else:
                self.network_event_scheduler.log_packet_info(packet, "received",
self.node_id)  # パケット受信をログに記録
                # パケットの宛先が自分自身でない場合の処理
                pass

    def send_packet(self, packet):
        self.network_event_scheduler.log_packet_info(packet, "sent",
self.node_id)  # パケット送信をログに記録
        if packet.header["destination"] == self.address:
            self.receive_packet(packet)
        else:
            for link in self.links:
                next_node = link.node_x if self != link.node_x else
link.node_y
                link.enqueue_packet(packet, self)
                break

    def create_packet(self, destination, header_size, payload_size):
        packet = Packet(source=self.address, destination=destination,
header_size=header_size, payload_size=payload_size,
network_event_scheduler=self.network_event_scheduler)
        self.network_event_scheduler.log_packet_info(packet, "created",
self.node_id)  # パケット生成をログに記録
        self.send_packet(packet)

    def set_traffic(self, destination, bitrate, start_time, duration,
header_size, payload_size, burstiness=1.0):
        end_time = start_time + duration
        def generate_packet():
            if self.network_event_scheduler.current_time < end_time:
                self.create_packet(destination, header_size, payload_size)
                packet_size = header_size + payload_size
                interval = (packet_size * 8) / bitrate * burstiness

self.network_event_scheduler.schedule_event(self.network_event_scheduler.current_time
+ interval, generate_packet)
        self.network_event_scheduler.schedule_event(start_time,
generate_packet)

    def __str__(self):
        connected_nodes = [link.node_x.node_id if self != link.node_x else
link.node_y.node_id for link in self.links]
        connected_nodes_str = ', '.join(map(str, connected_nodes))
        return f"ノード(ID: {self.node_id}, アドレス: {self.address}, 接続:
{connected_nodes_str})"

class Link:
```

```
    def __init__(self, node_x, node_y, bandwidth, delay, loss_rate,
network_event_scheduler):
        self.network_event_scheduler = network_event_scheduler
        self.node_x = node_x
        self.node_y = node_y
        self.bandwidth = bandwidth
        self.delay = delay
        self.loss_rate = loss_rate
        self.packet_queue_xy = []
        self.packet_queue_yx = []
        self.current_queue_time_xy = 0
        self.current_queue_time_yx = 0
        node_x.add_link(self)
        node_y.add_link(self)
        label = f'{bandwidth/1000000} Mbps, {delay} s'
        self.network_event_scheduler.add_link(node_x.node_id,
node_y.node_id, label, self.bandwidth, self.delay)

    def enqueue_packet(self, packet, from_node):
        if from_node == self.node_x:
            queue = self.packet_queue_xy
            current_queue_time = self.current_queue_time_xy
        else:
            queue = self.packet_queue_yx
            current_queue_time = self.current_queue_time_yx

        packet_transfer_time = (packet.size * 8) / self.bandwidth
        dequeue_time = self.network_event_scheduler.current_time +
current_queue_time
        heapq.heappush(queue, (dequeue_time, packet, from_node))
        self.add_to_queue_time(from_node, packet_transfer_time)
        if len(queue) == 1:
            self.network_event_scheduler.schedule_event(dequeue_time,
self.transfer_packet, from_node)

    def transfer_packet(self, from_node):
        if from_node == self.node_x:
            queue = self.packet_queue_xy
        else:
            queue = self.packet_queue_yx

        if queue:
            dequeue_time, packet, _ = heapq.heappop(queue)
            packet_transfer_time = (packet.size * 8) / self.bandwidth

            if random.random() < self.loss_rate:
                packet.set_arrived(-1)

            next_node = self.node_x if from_node != self.node_x else
self.node_y


self.network_event_scheduler.schedule_event(self.network_event_scheduler.current_time
+ self.delay, next_node.receive_packet, packet)
```

```
            self.network_event_scheduler.schedule_event(dequeue_time +
304 packet_transfer_time, self.subtract_from_queue_time, from_node,
    packet_transfer_time)
305
306             if queue:
307                 next_packet_time = queue[0][0]


308 self.network_event_scheduler.schedule_event(next_packet_time,
    self.transfer_packet, from_node)
309
310     def add_to_queue_time(self, from_node, packet_transfer_time):
311         if from_node == self.node_x:
312             self.current_queue_time_xy += packet_transfer_time
313         else:
314             self.current_queue_time_yx += packet_transfer_time
315
316     def subtract_from_queue_time(self, from_node, packet_transfer_time):
317         if from_node == self.node_x:
318             self.current_queue_time_xy -= packet_transfer_time
319         else:
320             self.current_queue_time_yx -= packet_transfer_time
321
322     def __str__(self):
        return f"リンク({self.node_x.node_id} ↔ {self.node_y.node_id}, 帯域幅:
323 {self.bandwidth}, 遅延: {self.delay}, パケットロス率: {self.packet_loss})"
324
325 class Packet:
    def __init__(self, source, destination, header_size, payload_size,
326 network_event_scheduler):
327         self.network_event_scheduler = network_event_scheduler
328         self.id = str(uuid.uuid4())
329         self.header = {
330             "source": source,
331             "destination": destination,
332         }
333         self.payload = 'X' * payload_size
334         self.size = header_size + payload_size
335         self.creation_time = self.network_event_scheduler.current_time
336         self.arrival_time = None
337
338     def set_arrived(self, arrival_time):
339         self.arrival_time = arrival_time
340
341     def __lt__(self, other):
342         return False  # heapqでの比較のため
343
344     def __str__(self):
        return f'パケット(送信元: {self.header["source"]}, 宛先:
345 {self.header["destination"]}, ペイロード: {self.payload})'
346
347 !git clone https://github.com/flyby-yunakayama/network-simulator.git
348 sys.path.insert(0,'/content/network-simulator')
349
350 # グローバルネットワークイベントスケジューラのインスタンスを作成
351 network_event_scheduler = NetworkEventScheduler(log_enabled=True)
352
353 # ノードとリンクの設定
```

```
    node1 = Node(node_id=1, address="00:01",
354
    network_event_scheduler=network_event_scheduler)
    node2 = Node(node_id=2, address="00:02",
355
    network_event_scheduler=network_event_scheduler)
    link1 = Link(node1, node2, bandwidth=10000, delay=0.001, loss_rate=0.0,
356
    network_event_scheduler=network_event_scheduler)
357
358 # 通信アプリケーションの設定
359 header_size = 40  # ヘッダサイズを40バイトとする
    payload_size = 85  # ペイロードサイズを設定（パケットサイズを 40 + 85 = 125バイト =
360
    1000ビット に設定）
    node1.set_traffic(destination="00:02", bitrate=1000, start_time=1.0,
361 duration=10.0, burstiness=1.0, header_size=header_size,
    payload_size=payload_size)
362
363 # イベントスケジューラを実行
364 network_event_scheduler.run()
365 # トラフィックのサマリを出力
366 network_event_scheduler.generate_summary(network_event_scheduler.packet_logs)
```

このプログラムの実行結果を以下に示す.

```
1 Src-Dst Pair: ('00:01', '00:02')
2 Total Sent Packets: 10
3 Total Sent Bytes: 1250
4 Total Received Packets: 10
5 Total Received Bytes: 1250
6 Average Throughput (bps): 1110.9876680368848
7 Average Delay (s): 0.0009999999999999343
8 Lost Packets: 0
```

参考文献

[1] 中山悠, *03_スイッチと MAC アドレス.ipynb*. 2025. [Online]. 入手先: https://colab.research.google.com/drive/1HzkwnrVWIBNE0deGMsrBNfjUNb_u33cx?usp=sharing