高 级 计 算 机 网 络
实 验 报 告

学　　员：陈阳　　学　　号：24023175

专　　业：计算机科学与技术　年　　级：2024 级

所属学院：计算机学院　　授课教员：蔡志平、夏竟、吕高锋

国防科技大学计算机学院
College of Computer Science and Technology

目　录

【摘　要】本实验通过编程构建了一个简化的网络模型，模拟了一个包含多个主机和路由器的环形网络环境。在该环境中，成功复现了网络拥塞和死锁现象，并设计实现了一种基于发送速率动态调整的拥塞控制机制。实验结果表明，该机制能够有效缓解网络拥塞，显著降低丢包率，提高网络吞吐量，并优化数据传输延迟。本实验进一步分析了不同网络条件下拥塞和死锁的触发机制，对拥塞控制算法的设计进行了验证。实验的完成不仅加深了对网络拥塞及其控制方法的理解，还为网络优化和流量管理提供了参考。

【关键词】 网络拥塞，死锁，拥塞控制

1　引 言

在计算机网络中，拥塞是一种常见且不可避免的现象，当网络中传输的数据量超过了链路或节点的处理能力时，就会导致性能下降甚至数据丢失。拥塞问题不仅影响数据的传输效率，还可能导致网络死锁，进一步加剧通信延迟和服务质量的下降。因此，如何检测、分析并有效控制网络拥塞一直是网络技术研究中的重要课题。

随着互联网和大规模分布式系统的迅速发展，数据流量呈指数级增长，网络拥塞的发生变得愈加频繁。传统的拥塞控制方法（如 TCP 拥塞控制）尽管在一般情况下表现优异，但面对高复杂度、高动态性的网络环境仍存在优化空间。通过模拟实验研究拥塞现象及其控制方法，能够帮助我们更深入地理解网络拥塞的产生机制以及其对网络性能的具体影响。

本实验旨在通过编程模拟一个简化的网络环境，复现网络拥塞现象，研究拥塞产生的原因及其控制方法的效果。实验中，我们还将设计并实现一种拥塞控制算法，对算法的有效性进行验证和分析。本实验的意义不仅在于验证经典网络理论的正确性，更在于为实际网络优化和拥塞管理提供理论参考和实践基础。同时，通过实验数据的量化分析，可为复杂网络环境中的资源调度与流量控制提供更为精确的优化建议。

本次实验的重要目标包括：通过编程复现网络拥塞现象，观察并分析拥塞如何产生；实现一种拥塞控制算法并验证其效果；最后，对实验结果进行数据分析，为网络拥塞控制提供系统性的解决方案。

2　实验内容

2.1 进度安排

本实验计划分为以下几个阶段进行，确保实验内容的顺利推进和完成：

1. 需求分析

目标：理解网络拥塞和死锁现象的原理，研究相关的网络理论。

内容：分析拥塞控制机制以及常见的网络拥塞现象；明确实验目标及需求。

预计时间：1 周。

2. 网络模型设计

目标：设计一个简化的网络模型，用于模拟拥塞和死锁。

内容：构建包含多个主机和路由器的网络拓扑，规划数据流路径，设计路由器的缓冲区行为及重试机制。

预计时间：1 周。

3. 拥塞控制机制设计

目标：根据拥塞控制的思路和原理，设计一种符合实验环境的拥塞控制算法。

内容：实现发送速率调整机制，包括基于丢包率动态调整发送速率的算法。

预计时间：2 周。

4. 程序实现：

目标：编写实验代码，完成网络模型的模拟和拥塞控制算法的实现。

内容：用 Python 完成网络拓扑、数据包生成与传输、拥塞控制机制等的程序设计。

预计时间：2 周。

5. 实验测试与优化：

目标：验证实验功能，观察并记录网络行为，调整参数优化实验效果。

内容：通过设置不同的初始发包率和缓冲区大小，测试拥塞和死锁现象；验证拥塞控制的效果。

预计时间：2 周。

6. 实验报告撰写：

目标：整理实验数据，撰写实验报告。

内容：总结实验过程和结果，对实验数据进行分析，并提出优化方案和展望。

预计时间：1 周。

#	任务	周期	起	止	周						
					1	2	3	4	5	6	7
1	需求分析	1 w	1	1	■						
2	网络模型设计	1 w	2	2		■					
3	拥塞控制机制设计	2 w	2	3		■	■				
4	程序实现	2 w	3	4			■	■			
5	实验测试与优化	2 w	5	6					■	■	
6	实验报告撰写	1 w	7	7							■

图 1 实验时间安排

2.2 实验环境

为了模拟一个简化的网络环境，本实验使用 Python 编程语言，在本地开发和测试。实验环境的软硬件需求如下：

2.2.1 硬件环境

开发主机：

CPU：Intel(R) Core(TM) i5-14400 2.50 GHz

内存：32.0 GB

存储：1024GB SSD

操作系统：Windows 11

2.2.2 软件环境

编程语言：Python 3.10

开发环境：PyCharm、VSCode

依赖库：

queue：用于实现路由器缓冲区的队列机制。

threading：用于模拟并发的网络通信。

matplotlib：用于可视化实验结果。

2.2.3 实验网络拓扑

实验中采用的网络拓扑如图 2 所示。网络结构包含 4 个主机（A、B、C、D）和 4 个路由器（R1、R2、R3、R4）。每个主机发送的数据包都会经过两个路由器到达目标主机，构成一个环形网络结构，确保了数据流的循环性和复杂性，有助于模拟拥塞和死锁现象。

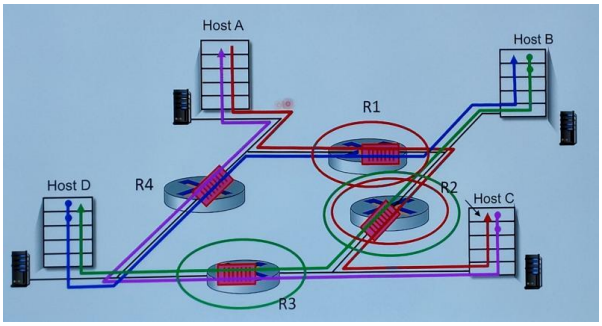


图 2 网络拓扑结构

3 网络拥塞与死锁的建模与实现

本章详细阐述了网络拥塞与死锁现象的建模与实现过程，包括网络拓扑设计、主机和路由器的行为建模、数据包的生成与传输机制、以及网络死锁的触发条件和检测方法。实验的设计充分考虑了网络通信中的核心问题，通过模拟网络环境中的资源竞争和数据流传输，成功复现了网络拥塞与死锁现象，为后续拥塞控制算法的设计和验证提供了坚实的基础。

3.1 实验网络模型设计

3.1.1 网络拓扑结构

实验设计了一个环形网络拓扑，包含 4 个主机（Host A、B、C、D）和 4 个路由器（Router R1、R2、R3、R4）。主机与路由器以及路由器之间的连接如下：

- Host A → Router R1 → Router R2 → Host C

- Host B → Router R2 → Router R3 → Host D
- Host C → Router R3 → Router R4 → Host A
- Host D → Router R4 → Router R1 → Host B

这种设计模拟了一个封闭的网络数据流路径，确保数据包在网络中循环传输。每条路径中均包含两个路由器，这种设计能够充分体现数据包在路由器之间竞争缓冲区资源的情况，同时为模拟死锁现象提供了环路等待的基础。

该环形拓扑具备以下特点：

1. **竞争性**：多个数据流同时通过相同的路由器，会导致缓冲区的资源竞争。
2. **环路依赖**：每条路径之间存在依赖关系，路由器需要等待下一跳路由器释放资源，从而可能形成循环等待。
3. **复杂性**：简化的环形拓扑结构使得问题易于分析，同时可以准确捕捉网络拥塞和死锁的本质。

3.2 数据包与路由器行为建模

3.2.1 数据包结构设计

为了模拟网络中数据包的生成、传输和处理，实验中设计了一个 Packet 类，用于描述数据包的基本属性和行为。数据包的主要属性包括：

- **src** 和 **dest**：数据包的源主机和目标主机。
- **path_list**：数据包在网络中经过的路由器路径列表。
- **path**：当前数据包所在路径的索引，用于标识数据包的传输进度。
- **seq_num**：数据包的唯一序列号，便于统计和跟踪。
- **timestamp**：数据包生成的时间戳，用于计算网络延迟。

数据包的路径列表设计为一个静态属性，由主机在生成数据包时设定，确保数据包能够准确地传递至目标主机。

3.2.2 路由器行为建模

路由器是网络中的关键节点，用于处理和转发数据包。每个路由器的行为分为以下几部分：

1. **缓冲区管理：**

路由器内部使用有限大小的缓冲区（基于 Python 的 queue.Queue 实现）存储等待处理的数据包。

如果缓冲区未满，则成功接收数据包；如果缓冲区已满，则根据实验设计采取不同的策略（如重试或丢弃数据包）。

2. **数据包处理：**

路由器对每个数据包的处理时间引入了随机延迟，延迟值服从正态分布（均值为 DELAY_MEAN，标准差为 DELAY_STDDEV）。这种延迟模拟了网络中由于路由器硬件处理能力和网络带宽限制而产生的波动。

3. **数据包转发：**

路由器检查数据包的路径表，将其转发至下一跳路由器或目标主机。如果目标路由器的缓冲区已满，则数据包进入重试机制，直到缓冲区释放为止。

以下是路由器处理数据包的核心代码片段（不是完整的原代码）：

```
def process_packets(self):
    while True:
        try:
            packet = self.buffer.get(timeout=0.1)
            delay = max(0, random.gauss(DELAY_MEAN,
DELAY_STDDEV))
            time.sleep(delay)  # 模拟路由器处理时间
            if packet.path >= len(packet.path_list):
                print(f"Host {packet.dest} received packet
{packet.packet_id}")
            else:
                next_router = packet.path_list[packet.path]
                next_router.receive_packet(packet)
        except queue.Empty:
            continue
```

3.3 网络拥塞与死锁的模拟

3.3.1 拥塞现象的复现

在实验中，我们通过以下方式成功复现了网络拥塞现象：

1. **路由器缓冲区限制：**

每个路由器的缓冲区容量设置为固定值（如 10）。当缓冲区满时，数据包无法立即被处理或转发。

2. **高发送速率：**

主机的初始数据包生成速率设置为每秒 1.2

个，这一速率远高于路由器的处理能力。随着数据流的增加，路由器的缓冲区逐渐被填满，最终引发网络拥塞。

3. **重试机制：**

当数据包无法进入路由器的缓冲区时，会触发重试机制。重试机制增加了路由器间的资源竞争，进一步加剧了网络拥塞。

3.3.2 死锁现象的触发

死锁的发生需要以下条件：

1. **资源竞争：**所有路由器的缓冲区均满载。
2. **循环等待：**路由器之间的资源依赖形成一个闭环（如 R1 → R2 → R3 → R4 → R1）。
3. **不可剥夺：**数据包无法被中途移除或重新调度，所有数据包均等待缓冲区释放。

通过逐步提高主机的发送速率（如从 1.2 提高至 3.0），实验中所有路由器的缓冲区最终达到满载状态，导致死锁现象的出现。

3.4 延迟的设置及合理性分析

```
# 参数设置
BUFFER_SIZE = 10
SIMULATION_TIME = 100  # 模拟时间（秒）
PACKET_GEN_RATE_START = 0.8  # 初始数据包生成率（每秒）
PACKET_INTERVAL_STDDEV = 0.5  # 生成数据包间隔时间的标准差
DELAY_MEAN = 0.5  # 路由器处理数据包的延迟的数学期望（秒）
DELAY_STDDEV = 0.2  # 网络延迟的震荡幅度（标准差）
RETRY_TIME = 0.5
```

图 3 无拥塞情况

实验的主要参数配置如图 3 所示。在本实验中，延迟的值设置为较大的数量级（基本都以秒为单位），这是基于以下原因：

1. **进程间交互的隐式延迟：**

实验环境基于进程间的线程调度和通信。由于 Python 的多线程受全局解释器锁（GIL）影响，以及操作系统调度的延迟，实际的进程间交互时间已经隐式地模拟了网络传输的一部分延迟。

2. **实验延迟平衡的合理性：**

进程间交互带来的延迟较大，为了保证整体延迟时间的平衡性，我们将路由器处理时间的延迟设置为相同的数量级（0.5 秒），以保持实验中各部分延迟的相对合理性。

3. **关注相对延迟：**

实验的目标是通过相对时间差分析网络行为，因此延迟的绝对值并不重要。只要各部分的延迟设置合理，实验结果的相对时间具有参考意义。这种延迟设置有效地平衡了模拟实验和实际网络环境之间的差异，使得实验结果更具可解释性。

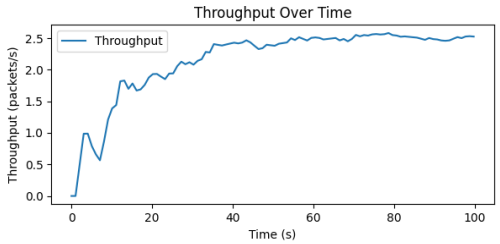
3.5 模拟实验结果

在实验中，我们通过调整主机的发送速率（PACKET_GEN_RATE_START）和路由器的缓冲区大小（BUFFER_SIZE），观察并分析了网络在不同条件下的表现。实验结果表明，网络的状态可以大致分为三种情况：流畅无拥塞、拥塞和死锁。以下是对这三种情况的详细介绍：

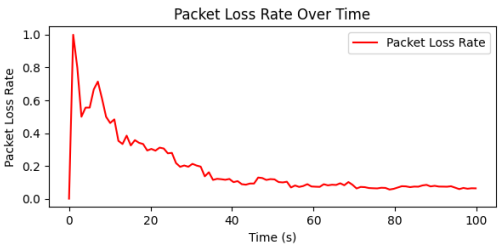
1. **无拥塞：**

当主机的发送速率较低（0.8 个数据包/秒），网络中的数据流量远低于路由器的处理能力和缓冲区的承载能力，系统处于流畅无拥塞状态。此时表现如下：

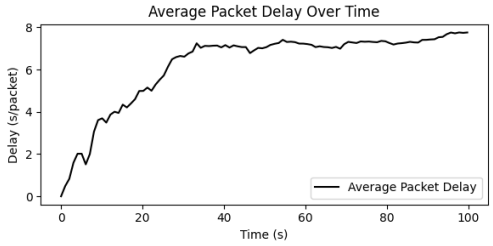
数据包能够在到达路由器时被立即处理或转发，缓冲区中几乎没有积压。由于没有等待和重试，数据包的传输延迟仅受网络中随机延迟（DELAY_MEAN 和 DELAY_STDDEV）的影响，传输过程非常高效。所有数据包均被成功传输至目标主机，丢包率接近 0。这种状态反映了网络资源充裕时的理想运行状态。



(a) 吞吐量



(b) 丢包率



(c) 平均延迟

图 4 无拥塞情况

图 4 展示了实验中的总吞吐量、丢包率和平均延迟随模拟时间的变化曲线。其中，丢包率会在模拟实验最开始的极短时间内上升到 1，是因为第一批传输的数据包已经发出，且在传输路径上没有到达目标节点，而本实验采取的丢包率的计算公式为：(已发送数据包-已接受数据包) / 已发送数据包。随着模拟继续进行，没有任何数据包在传输过程中因缓冲区溢出被丢失，丢包率会逐渐趋近于 0。而吞吐量和平均延迟逐渐趋于一个稳定值。

图 5 展示了各个节点的平均延迟和成功传输的数据包数量。

```
Host A average delay: 5.2036s over 65 packets
Host B average delay: 4.4297s over 65 packets
Host C average delay: 6.1009s over 60 packets
Host D average delay: 6.5080s over 67 packets
```

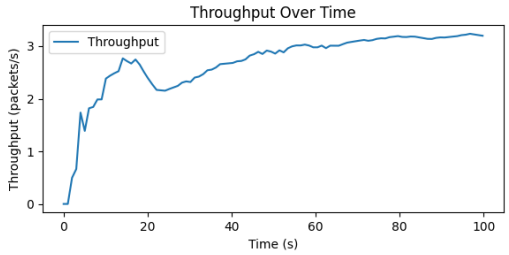
图 5 各节点情况（无拥塞）

2. **拥塞现象：**

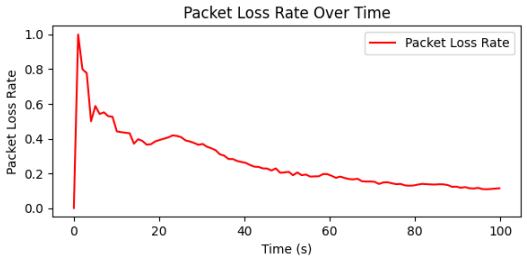
当主机的发送速率逐渐提高（1.2 个数据包/秒），网络流量接近或略微超过路由器的处理能力时，系统进入拥塞状态。此时表现如下：

部分路由器的缓冲区开始积压数据包，无法立即处理所有传入数据包。数据包需要在路由器的缓冲区中等待较长时间才能被处理或转发，传输延迟显著增加。当路由器的缓冲区满载时，某些数据包因无法进入缓冲区而被丢弃，丢包率显著上升。

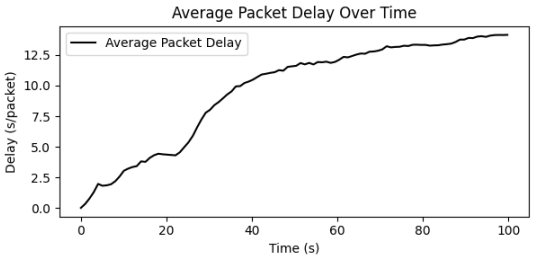
在此状态下，网络吞吐量仍然较高，但随着丢包率的增加，网络性能开始下降。具体性能如图 6 所示，吞吐量相比于无拥塞情况有所提高，因为情况 1 中的发送速率还没有完全利用到网络的性能。而丢包率平均延迟相比于情况 1 则都有了明显的上升。



(a) 吞吐量



(b) 丢包率



(c) 平均延迟

图 6 拥塞情况

图 7 展示了各个节点的平均延迟和成功传输的数据包数量。可以看到每个节点总的发送成功的数据包数量有所上升，但平均延迟也有了显著提高。

```
Host A average delay: 10.7974s over 80 packets
Host B average delay: 10.5223s over 75 packets
Host C average delay: 9.5951s over 80 packets
Host D average delay: 10.5388s over 85 packets
```

图 7 各节点情况（拥塞）

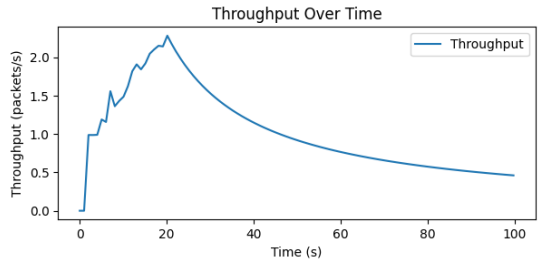
3. **死锁现象：**

当主机的发送速率进一步提高（ 2 个数据包/秒，远超出实验模拟网络的性能极限），所有路由器的缓冲区均达到满载状态，且没有足够的缓冲区空间释放以处理新的数据包时，系统进入死锁状态。此时表现如下：

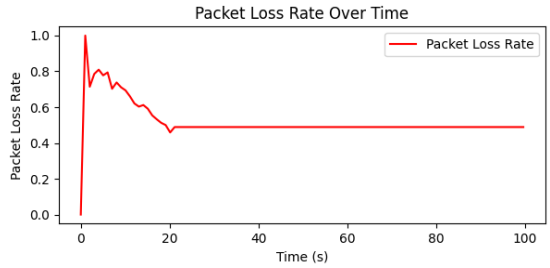
每个路由器的缓冲区中均存满了等待转发的

数据包，且这些数据包的目标路由器的缓冲区也已满，导致无法继续传输。新生成的数据包无法进入路由器的缓冲区，所有数据流停止，网络吞吐量降为 0。路由器之间形成了典型的循环等待，每个路由器都在等待下一个路由器释放缓冲区，从而无法处理队首数据包。从而出现了死锁的现象。

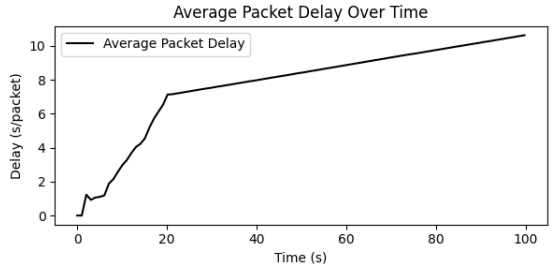
具体的性能曲线如图 8 所示。



(a) 吞吐量



(b) 丢包率



(c) 平均延迟

图 8 死锁情况

从图中可以看出，死锁大概发生在模拟时间推进到 20s 左右时。发生死锁后，所有数据包都停止了传输吗，吞吐量开始随时间持续下降，丢包率稳定在一个固定的值（因为此时主机也无法继续发送新的数据包），平均延迟随时间持续上升。

```
Time: 100.47s, R4 buffer full. Packet D22 dropped
Time: 100.52s, R3 buffer full. Packet C20 dropped
Time: 100.81s, R2 forwarded packet B20 to R3
Time: 100.81s, R3 buffer full. Packet B20 dropped
Time: 100.87s, R1 forwarded packet A15 to R2
Time: 100.87s, R2 buffer full. Packet A15 dropped
Time: 100.92s, R2 buffer full. Packet B27 dropped
Time: 100.98s, R4 forwarded packet D16 to R1
Time: 100.98s, R1 buffer full. Packet D16 dropped
Time: 100.98s, R3 forwarded packet C16 to R4
Time: 100.98s, R4 buffer full. Packet C16 dropped
Time: 101.19s, R3 buffer full. Packet C20 dropped
```

图 9 死锁情况的部分日志

图 9 是在死锁情况模拟中截取的部分日志信息，可以看到此时，R1 到 R4 四个路由器全部都被占满了缓冲区，他们都在不断尝试将数据包发送到下一个路由器，但是都会一直发送失败，从而形成死锁。

4 拥塞控制机制的设计与实现

4.1 拥塞控制机制设计思路

拥塞控制是为了在网络资源有限的情况下，防止网络陷入拥塞甚至死锁的状态，同时尽可能提高网络的吞吐量和资源利用率。本实验的拥塞控制机制参考了 TCP 的拥塞控制思想，并结合模拟网络的特点，设计了基于发送速率动态调整的简单算法。其设计思路如下：

1. 检测拥塞：

拥塞的主要表现是数据包在传输中被丢弃。因此，通过监控丢包数量，可以间接判断网络是否处于拥塞状态。主机每隔一段时间统计一次丢包数量的变化，并以此作为拥塞状态的判断依据。

2. 动态调整发送速率：

减速策略：当检测到拥塞（丢包数量增加）时，主机减少发送速率，以缓解网络负载。

加速策略：当没有检测到拥塞（丢包数量没有增加）时，主机逐步增加发送速率，充分利用网络资源。

3. 自适应机制：

主机的发送速率调整幅度是动态的。拥塞时的减速较快，以便迅速恢复网络正常；而无拥塞

时的加速较慢，以防止发送速率过快导致新的拥塞。

4.2 拥塞控制机制的实现

主机的发送行为由 Host 类中的 send_packets 方法实现。在发送数据包的过程中，主机会根据丢包数量动态调整发送速率。具体实现包括以下几个步骤：

1. 发送数据包：

主机以当前的发送速率生成数据包，并尝试将其发送至第一个路由器。如果第一个路由器的缓冲区已满，数据包将被丢弃。

2. 丢包统计：

主机记录丢包数量的变化（packets_dropped），用于判断是否发生拥塞。

3. 发送速率调整：

若丢包数量增加，则主机降低发送速率；否则，主机缓慢提高发送速率。

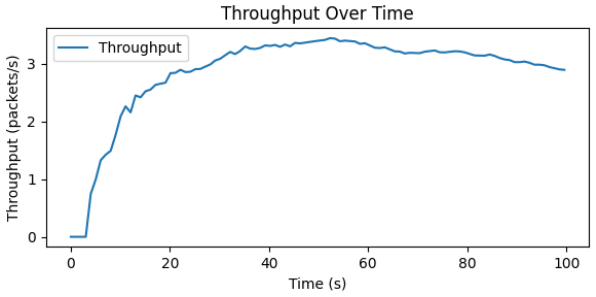
以下是主机调整发送速率的代码实现：

```
def adjust_send_rate(self):
    global packets_dropped
    delta = packets_dropped - self.pre_packets_dropped
    with self.lock:
        if delta > 0:  # 丢包数量增加，表明发生拥塞
            self.send_rate = max(0.2, self.send_rate - 0.1)  # 快速减速
            print(f"Host {self.name} detected congestion. Reducing send rate
to {self.send_rate:.2f} packets/s")
        else:  # 丢包数量未增加，逐步提高发送速率
            self.send_rate += 0.02  # 缓慢加速
            print(f"Host {self.name} increasing send rate to
{self.send_rate:.2f} packets/s")
    self.pre_packets_dropped = packets_dropped
```

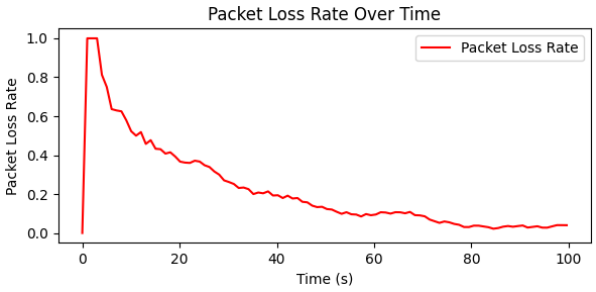
4.3 拥塞控制效果分析

在添加了上述的拥塞控制算法后，我们重新运行了第三章中的情况 2，也就是模拟拥塞的情况，数据包初始发送率为 1.2 个数据包/秒。

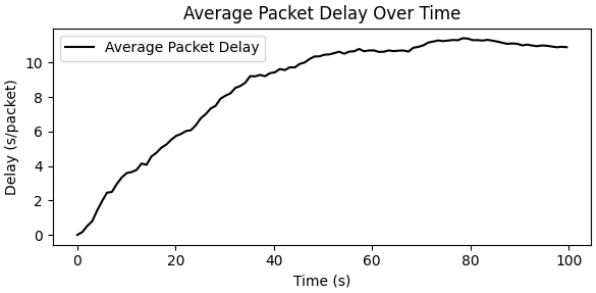
得到结果如图 10 所示。吞吐量相比于无拥塞控制时有所提高，且丢包率和平均延迟均有显著降低。拥塞控制算法在一定程度上得到了预期的结果。



(a) 吞吐量



(b) 丢包率



(c) 平均延迟

图 10 拥塞控制机制效果

5 结论

本实验通过编程的方式，成功地模拟了一个简化的网络环境，复现了网络拥塞和死锁这两种重要的网络现象。同时，我们在实验中设计并实现了一种基于发送速率动态调整的拥塞控制机制，对网络性能进行了有效的优化。通过本次实验，我们不仅加深了对网络拥塞产生原因及其控制方法的理解，还通过实验数据的分析和验证，进一步证明了所设计拥塞控制算法的有效性。该实验的完成为网络优化与拥塞管理提供了一个实践参考，同时也为后续更复杂的网络性能研究奠定了基础。

参考文献

[1] 赵梓铭，刘芳，蔡志平，肖侬. 边缘计算：平台、应用与挑战[J].计算机研究与发展, 2018: 327-337.

[2] 傅颖勋，罗圣美，舒继武. 安全云存储系统与关键技术综述[J].计算机研究与发展, 2013: 136-145.