**TCP源端拥塞控制和路由节点丢包控制组合对比实验**

小组成员：2151299 苏家铭 2152402 段婷婷 2051828 莫益萌 2051498 储岱泽

**引言**

|  |
| --- |
| 近年来，网络智能化已成为网络创新式发展和变革的突出特征，与传统网络不同，智能网络中存在着大量实时多媒体业务，如网页浏览、语音聊天、视频会议等，这些业务产生了巨大、多样的数据流量。网络转发设备的缓冲区在网络处于流量负荷大的情况下易被填满，产生网络拥塞问题，  基于源端的拥塞控制是缓解网络拥塞的有效办法，实现了诸如TCP New Reno、TCP High Speed、TCP Bic、Vegas、Illinois、westwood等算法。“端到端原则”提供的可扩展性在客观上促进了网络的快速发展，但基于源端的拥塞控制具有一定的滞后性，网络在检测到拥塞和拥塞发生的时间间隔内会一直处于拥塞状态。因此，为了保障服务质量quality of service(QoS）模型有效地运行，不仅需要在源端实现拥塞控制，网络中间节点也需要发挥作用。  队列管理是指网络出现拥塞时路由节点通过丢包来调节缓冲区的占有率，不同的队列管理机制直接影响到网络节点的拥塞控制能力和网络的QoS。总的来说，队列管理机制可以分为被动队列管理机制与主动队列管理机制两大类，与被动队列管理机制相比，主动队列管理机制能够使发送节点在中间节点缓存溢出前作出反应，有效减少和避免拥塞的发生。其中，RED(random early detection）算法作为主动队列管理机制中的代表性算法在拥塞控制方面起到了关键作用，该方法使路由节点缓冲区的队列在没有达到极值前就开始丢包，并向源端发送拥塞反馈信息，以此来缓解网络拥塞，但它存在参数配置敏感和全局同步现象等缺陷。  本实验实现了11种基于源端的TCP拥塞控制算法和3种基于中间节点的丢包控制算法，通过排列组合不同的算法，比较各种组合的丢包率、RTT、吞吐量、队列长度等性能指标，体现不同组合间的优劣。 |

**一、实验目的**

比较不同**TCP两端拥塞控制（End-to-End Congestion Control）**和**路由器丢包控制（Router-based Packet Loss Control）**策略组合在网络通信中的性能表现，深入了解各类组合的优势和劣势。

1. **探究拥塞控制策略的影响：**

* 比较TCP两端拥塞控制策略在网络拥塞环境下的表现，了解其对数据传输速率和稳定性的影响。
* 探究TCP两端拥塞控制策略在网络拥塞和瓶颈链路情况下的调整能力。

1. **分析路由器丢包控制策略的效果：**

* 研究路由器丢包控制策略在网络中的作用，了解其对数据包传输的影响。
* 考察在路由器级别进行丢包控制对网络整体性能的改善，以及对连接稳定性的影响。

1. **对比实验结果：**

* 进行一系列实验，模拟**哑铃型网络拓扑**和负载情境，分别应用不同TCP两端拥塞控制和路由器丢包控制策略进行组合。
* 通过收集实验数据，比较不同组合策略的网络性能，包括吞吐量、延迟、稳定性等方面的指标。

通过对不同TCP两端拥塞控制和路由器丢包控制策略组合的比较研究，深入了解它们在网络通信中的表现，为网络设计和优化提供有价值的参考和指导。

**二、实验原理**

**2.1 TCP拥塞算法简介**

目前，网络上已经获得广泛部署的TCP大多采用“慢启动”，“拥塞避免”，“快速重传”以及“快速恢复”这四种基本算法来实现拥塞控制，然而存在网络适应性差，效率低和公平性差等问题。因此后续的学者对此作出了有针对性的改进，提出了Tahoe、Reno、new Reno、BIC TCP、Vegas、Illinois、westwood等算法。

在本实验中，对于TCP拥塞控制算法的研究，本小组实现了以下TCP拥塞控制算法：

1. **TcpNewReno:**

* **基本思想：** TcpNewReno是Reno算法的扩展，引入了快速重传和快速恢复机制。当接收到三个重复的ACK时，它假设发生了丢包，快速重传丢失的数据段，并通过快速恢复机制迅速增加拥塞窗口。 因此能够区分出一次拥塞丢失多个包和多次拥塞两种不同的状况，解决了Reno算法过早退出快速恢复状态的问题，提高了网络性能。
* **作用：** 实现了快速重传和快速恢复，以更有效地处理数据包的丢失，提高TCP性能。
* **意义：** 在面对网络丢包时，通过快速的恢复机制，TcpNewReno能够减小数据传输的中断时间，提高网络的吞吐量。
* **适用条件：** 适用于需要快速适应网络丢包情况的场景，如高丢包率的环境。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if three\_duplicate\_ack\_received():  fast\_retransmit()  fast\_recovery()  on\_ack\_received():  if in\_fast\_recovery\_mode():  partial\_ack\_received()  else:  slow\_start()  congestion\_avoidance() |

* **算法性能指标**

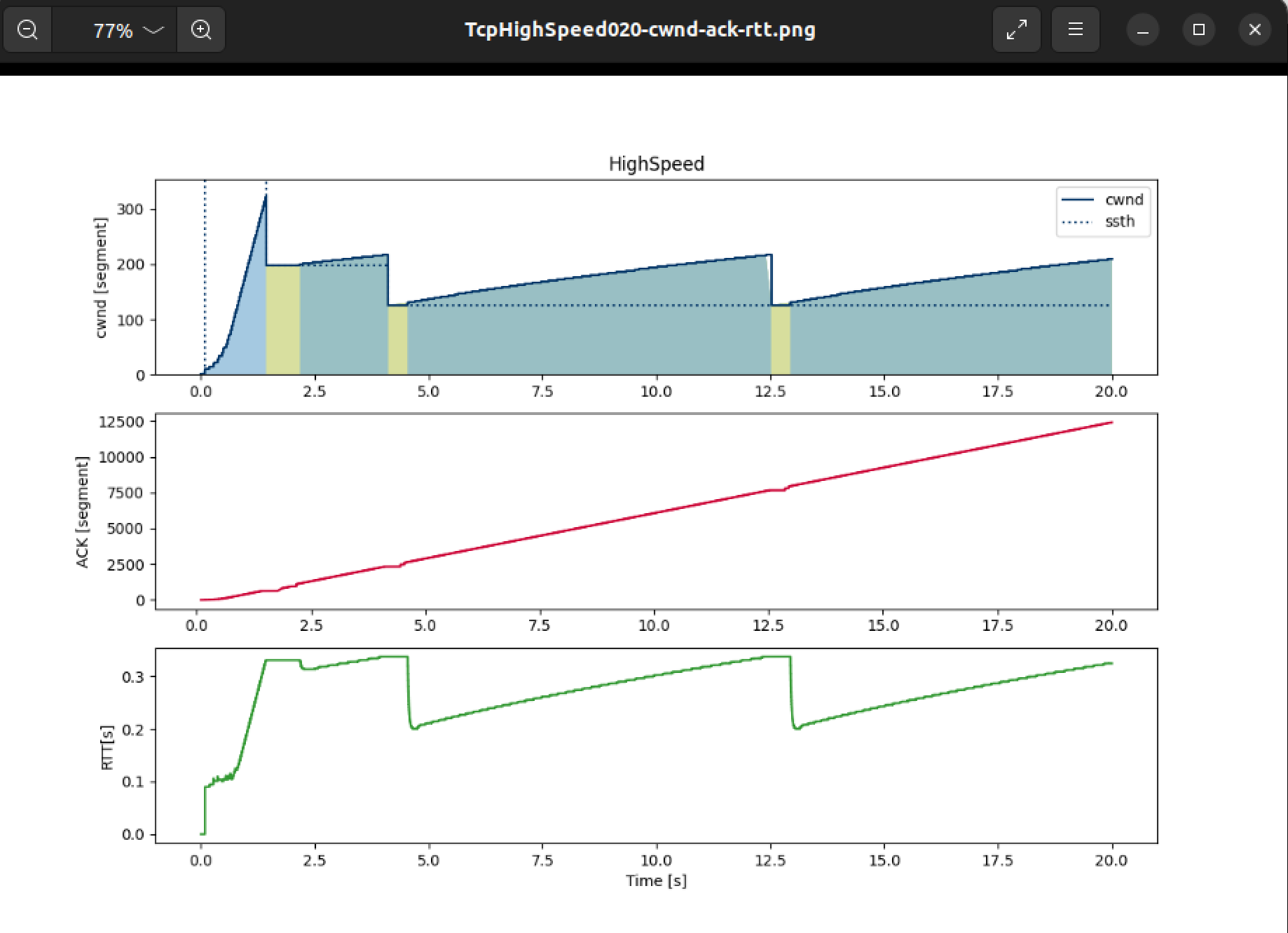


2. **TcpHighSpeed:**

* **基本思想：** TcpHighSpeed旨在提高TCP在高丢包环境中的性能。它通过动态调整拥塞窗口的增长速率和减小幅度，以更有效地适应网络拥塞，并降低拥塞对性能的影响。
* **作用：** 通过调整拥塞窗口的增长速率和减小幅度，提高TCP在高丢包环境中的性能。
* **意义：** 在高丢包率的网络中，TcpHighSpeed能够更灵活地调整发送速率，降低网络拥塞的影响。
* **适用条件：** 适用于高丢包率网络，如无线网络或不稳定的网络连接。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  increase\_gamma()  on\_ack\_received():  if increase\_window\_condition():  increase\_window\_size()  else:  congestion\_avoidance() |

* **算法性能指标**

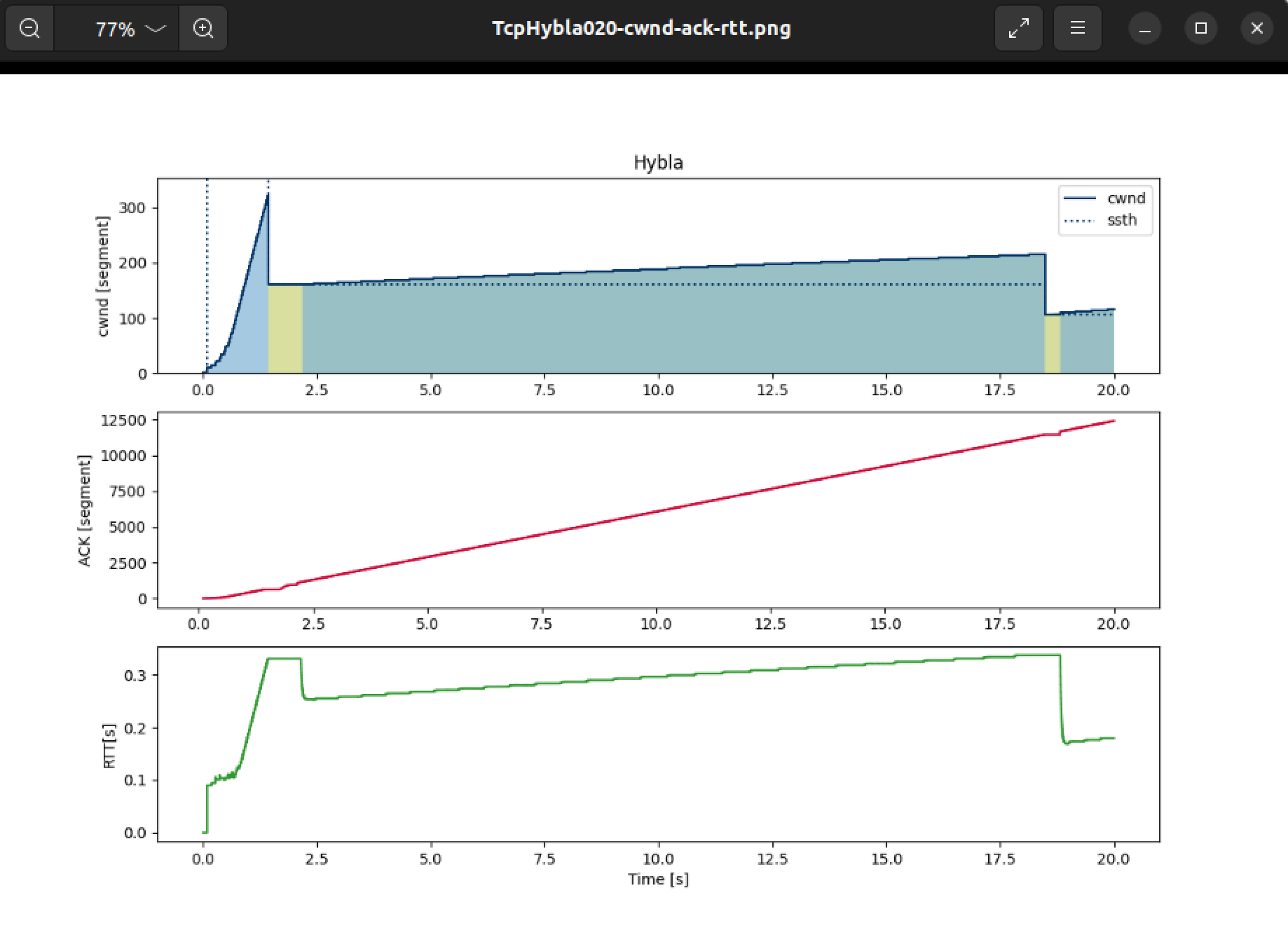


3. **TcpHybla:**

* **基本思想：**TcpHybla专注于解决高延迟网络的问题。它通过动态调整拥塞窗口以适应长延迟的特性，从而更好地利用网络资源，减少不必要的拥塞。
* **作用：** 专为高延迟网络设计，通过动态调整拥塞窗口以适应长延迟的特性。
* **意义：** 在卫星链路等高延迟网络中，TcpHybla通过考虑延迟的特性，提高了数据传输的效率。
* **适用条件：** 适用于高延迟网络，如卫星通信等。
* **算法伪代码：**

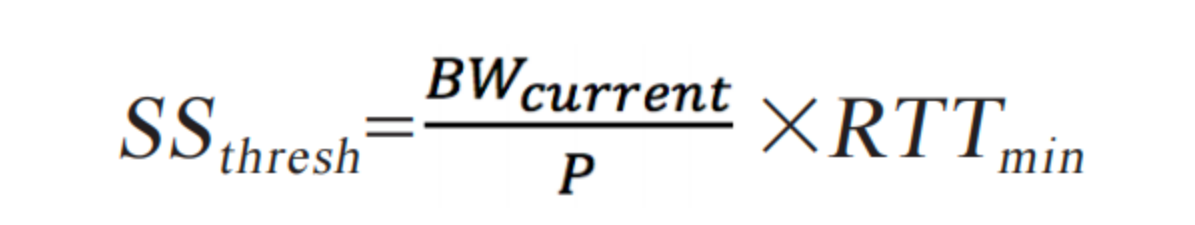
|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  on\_ack\_received():  if increase\_window\_condition():  increase\_window\_size()  else:  congestion\_avoidance() |

* **算法性能指标**



4. **TcpWestwood:**

* **基本思想：**通过计算接收端返回的ACK数据包的速率来估算端到端的可用带宽，然后再根据可用带宽来调整拥塞窗口。



当连续收到3个ACK后，就将SSthresh设置为拥塞窗口的当前值，这样算法可以根据可用带宽的变化，实时动态地调整拥塞窗口的大小，以解决缓冲区过度被占用问题。

另外，TCP Westwood算法还将丢包结合进来，一旦发现丢包，则根据一个估计数来设置慢启动，而不是直接将窗口减半，以避免在信道误码的情况下传输速率急剧下降。

* **作用：** 使用实时估计的网络带宽来调整拥塞窗口，提高网络利用率。
* **意义：** TcpWestwood通过更准确的网络带宽估计，更有效地适应网络状况，减少不必要的拥塞。
* **适用条件：** 适用于需要对网络带宽实时估计的场景，如流媒体传输。
* **算法伪代码：**

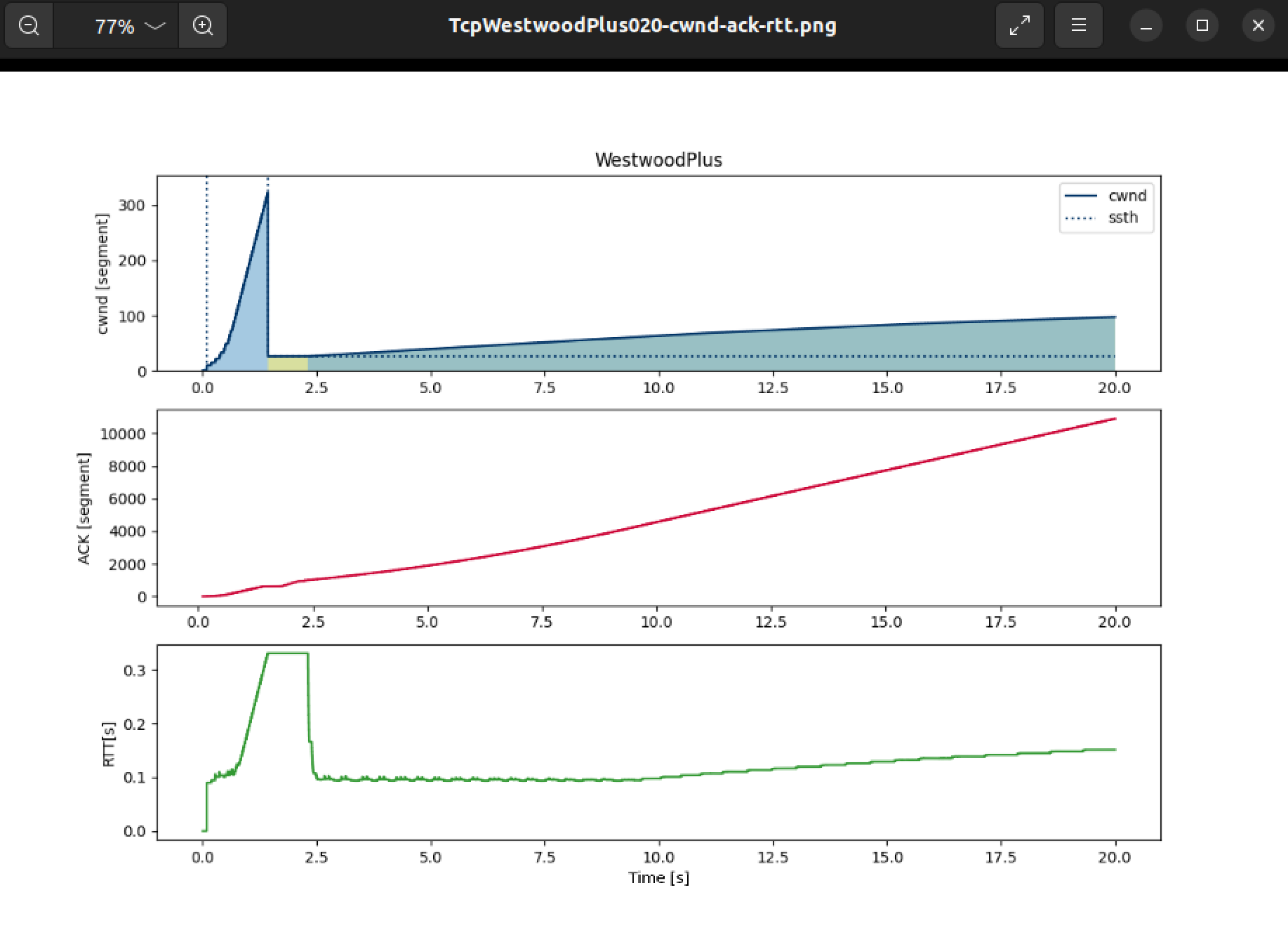
|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  update\_bandwidth\_estimate()  reduce\_window\_size()  on\_ack\_received():  increase\_window\_size() |

5. **TcpWestwoodPlus:**

* **基本思想：**TcpWestwoodPlus是TcpWestwood的改进版本，增加了对丢包事件的考虑，以更好地适应网络条件的变化。它进一步提高了网络的鲁棒性。
* **作用：** 是TcpWestwood的改进版本，更好地适应网络条件的变化，特别是在丢包事件发生时。
* **意义：** 在网络发生变化时更加灵活地调整，提高了网络的稳定性和性能。
* **适用条件：** 适用于网络环境动态变化的场景，如移动网络。
* **算法伪代码：**

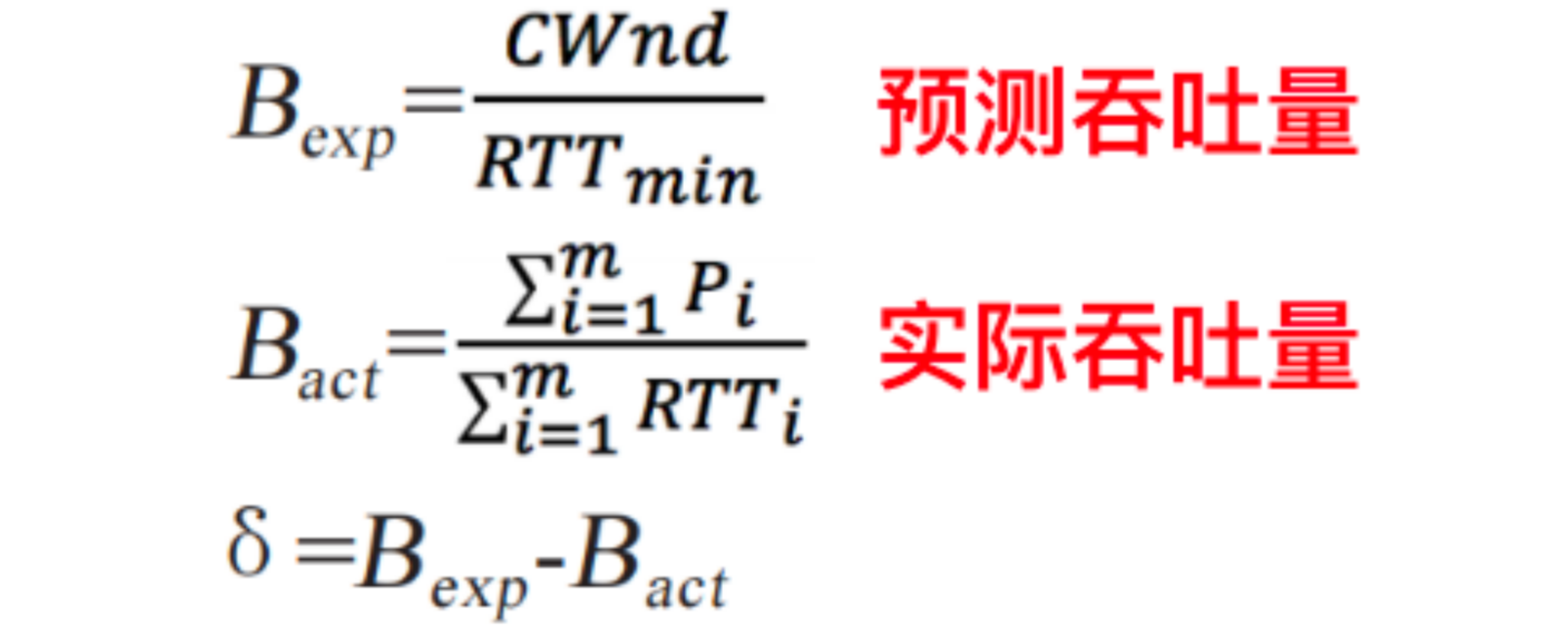
|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  update\_bandwidth\_estimate()  reduce\_window\_size()  on\_ack\_received():  if increase\_window\_condition():  increase\_window\_size()  else:  congestion\_avoidance() |

* **算法性能指标**



6. **TcpVegas:**

* **基本思想：**以往返时间为单位，分别计算单位时间内的预测吞吐量和实际吞吐量。

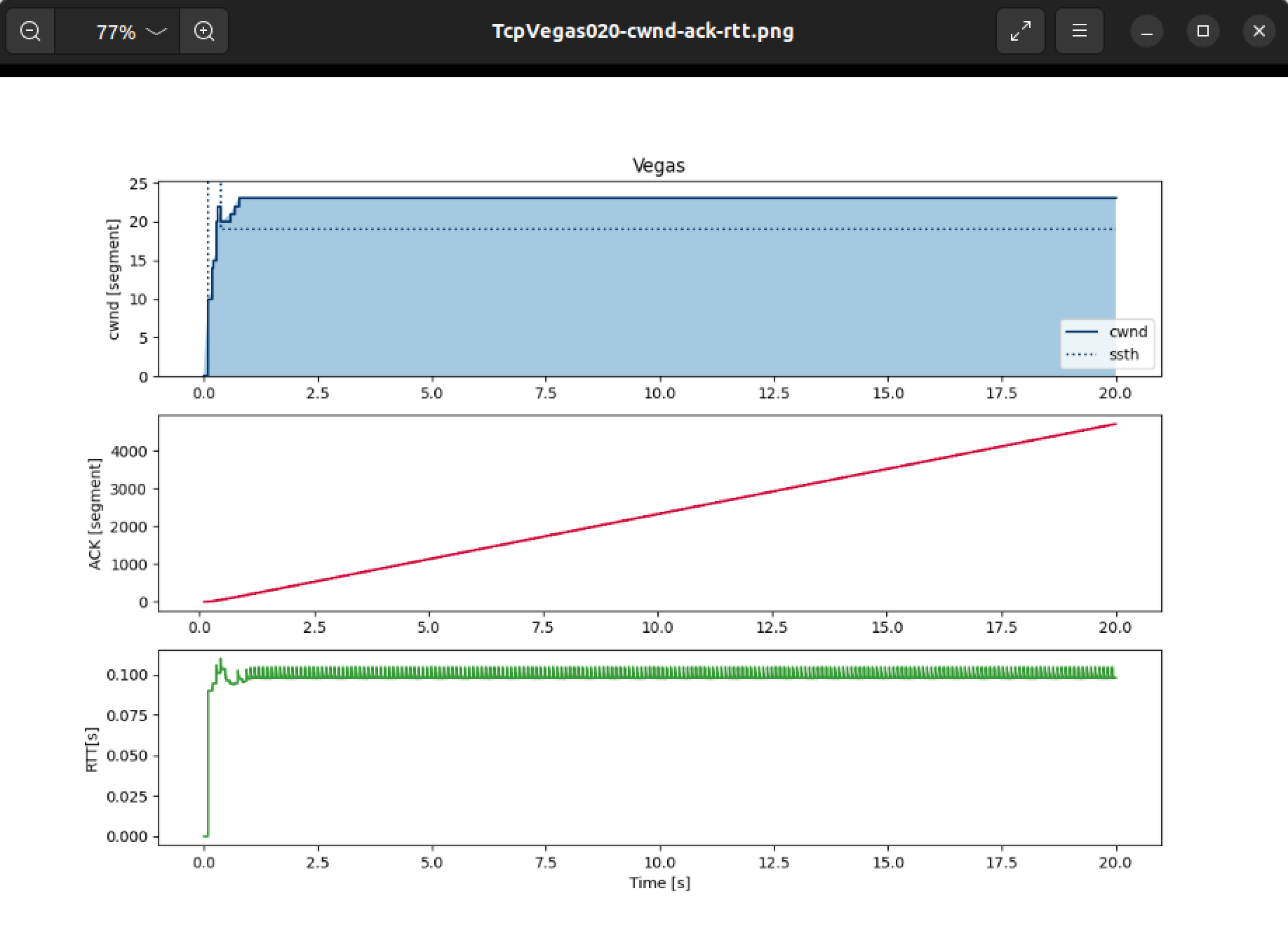


以δ值的大小判断路径拥挤状况，当δ大于高阈值时，表示该路径发生严重拥挤，因此需要重新降低发送速率，否则保持当前的发送速率。

* **作用：** 使用窗口变化率来判断网络拥塞，并相应地调整拥塞窗口。
* **意义：** TcpVegas通过实时观察拥塞窗口的变化率，更及时地响应网络拥塞，提高网络性能。
* **适用条件：** 适用于需要快速响应拥塞的场景，如实时通信系统。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  on\_ack\_received():  if increase\_window\_condition():  increase\_window\_size()  else:  congestion\_avoidance() |

* **算法性能指标**

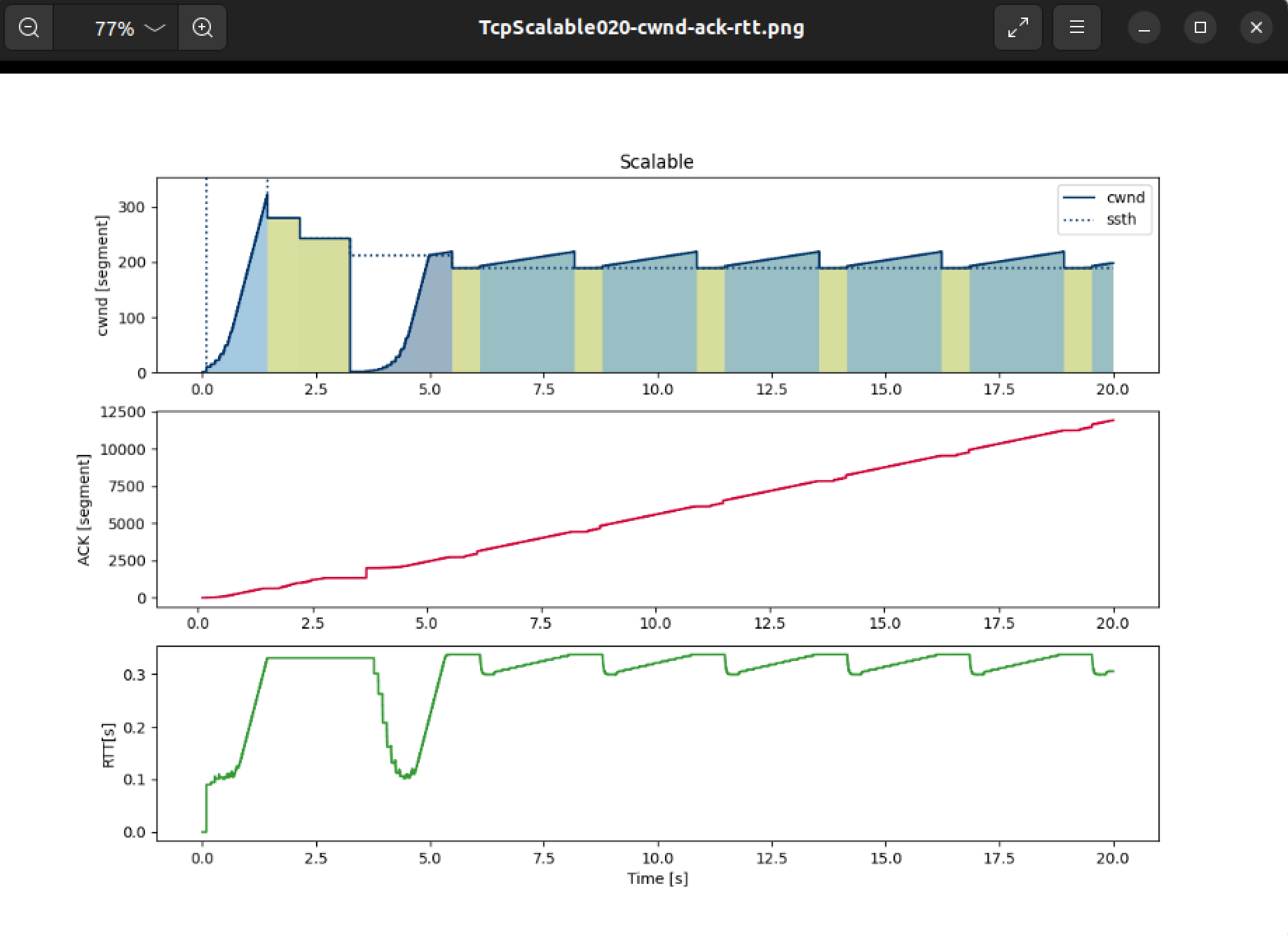


7. **TcpScalable:**

* **基本思想：**TcpScalable具有自适应性，能够根据网络状况调整发送速率。它采用窗口自适应机制，旨在处理拥塞并维护较高的性能。
* **作用：** 具有自适应性，根据网络状况调整发送速率，并采用窗口自适应机制来处理拥塞。
* **意义：** TcpScalable通过自适应的机制，能够在不同网络条件下保持较高的性能。
* **适用条件：** 适用于需要在不同网络环境中保持高性能的场景，如企业网络。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  decrease\_alpha()  on\_ack\_received():  increase\_window\_size() |

* **算法性能指标**

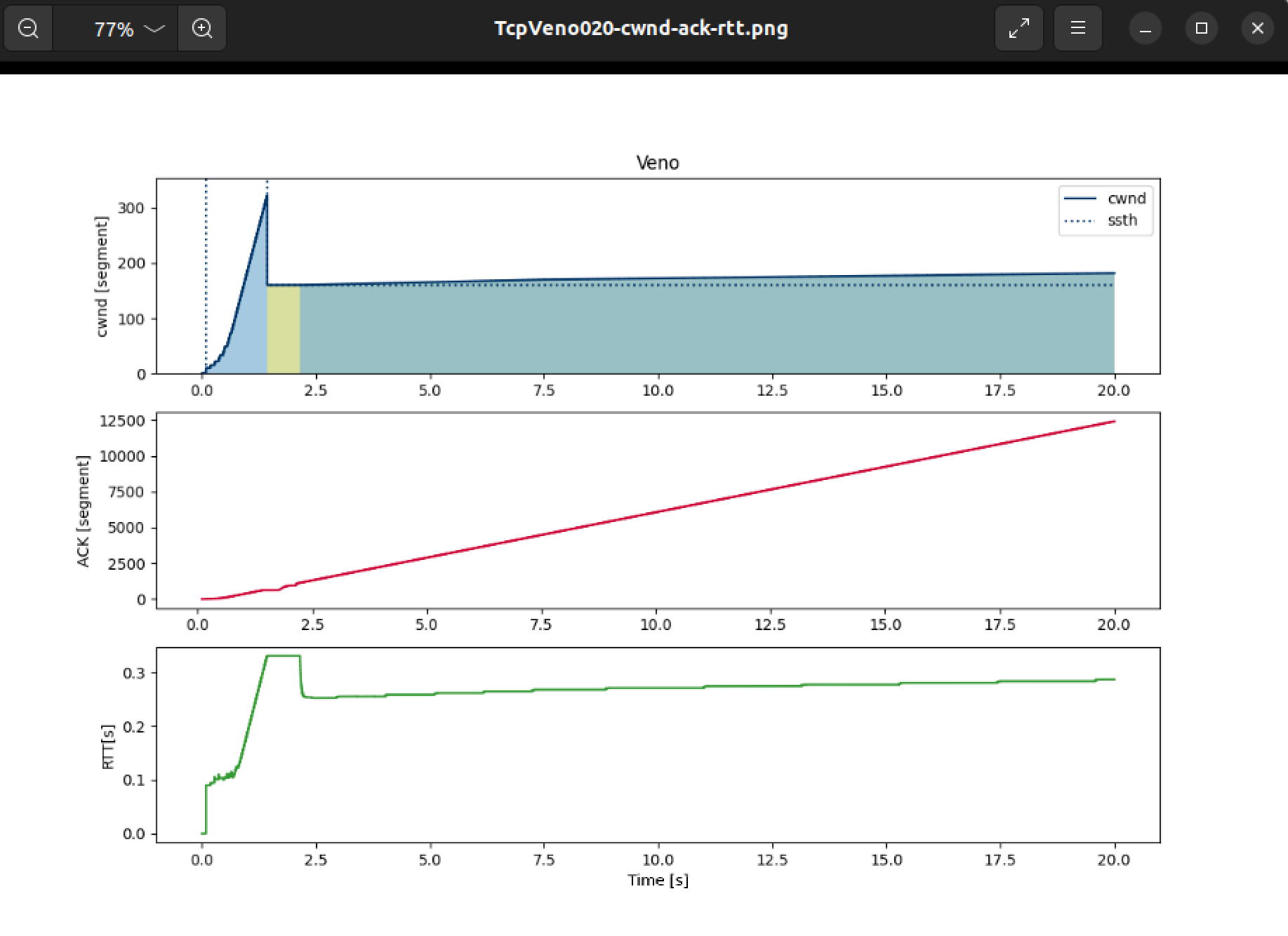


8. **TcpVeno:**

* **基本思想：**TcpVeno结合了TCP的拥塞控制和显式拥塞通知（ECN）机制。通过ECN，它能够提前感知网络拥塞，并采取相应的拥塞控制措施。
* **作用：** 结合了TCP的拥塞控制和显式拥塞通知（ECN）机制，通过ECN来提前发现网络拥塞并进行调整。
* **意义：** TcpVeno通过结合ECN机制，更及时地发现拥塞并调整，提高了网络的敏感性。
* **适用条件：** 适用于支持ECN机制的网络环境，如高级数据中心网络。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  on\_ack\_received():  if ecn\_marked\_packet\_received():  decrease\_alpha()  else:  increase\_window\_size() |

* **算法性能指标**

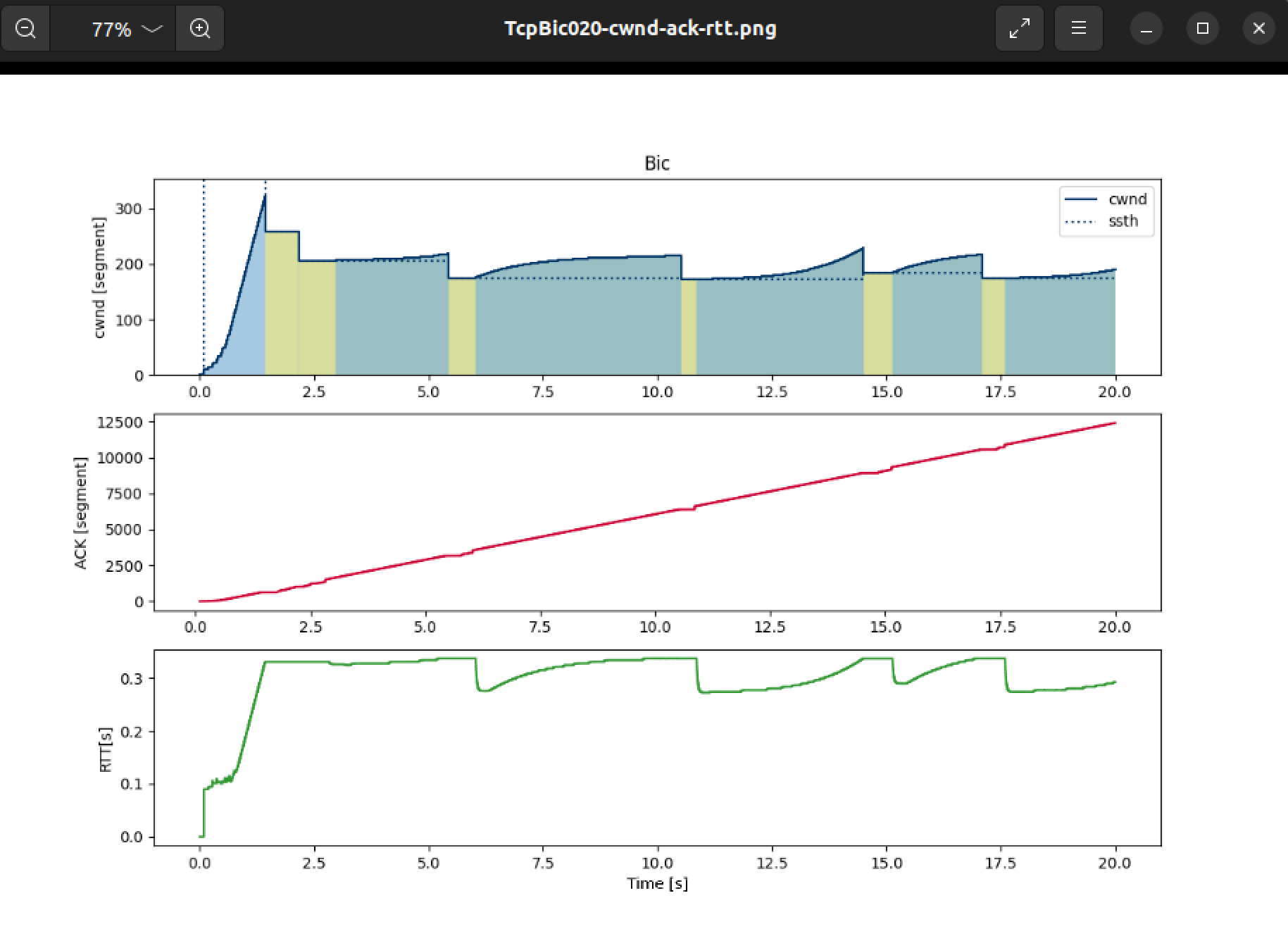


9. **TcpBic:**

* **基本思想：**将拥塞窗口控制转化成实际值搜索问题，采用二分查找法，通过取所允许窗口范围中点的方式来调整拥塞窗口，以加快最大拥塞窗口的恢复速度，提高资源利。
* **作用：** 设计用于高速长距离网络，通过快速调整拥塞窗口以提高网络利用率。
* **意义：** TcpBic专注于高速网络，通过快速调整，提高了在长距离网络中的数据传输效率。
* **适用条件：** 适用于高速长距离网络，如数据中心互联网络。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  on\_ack\_received():  increase\_window\_size() |

* **算法性能指标**

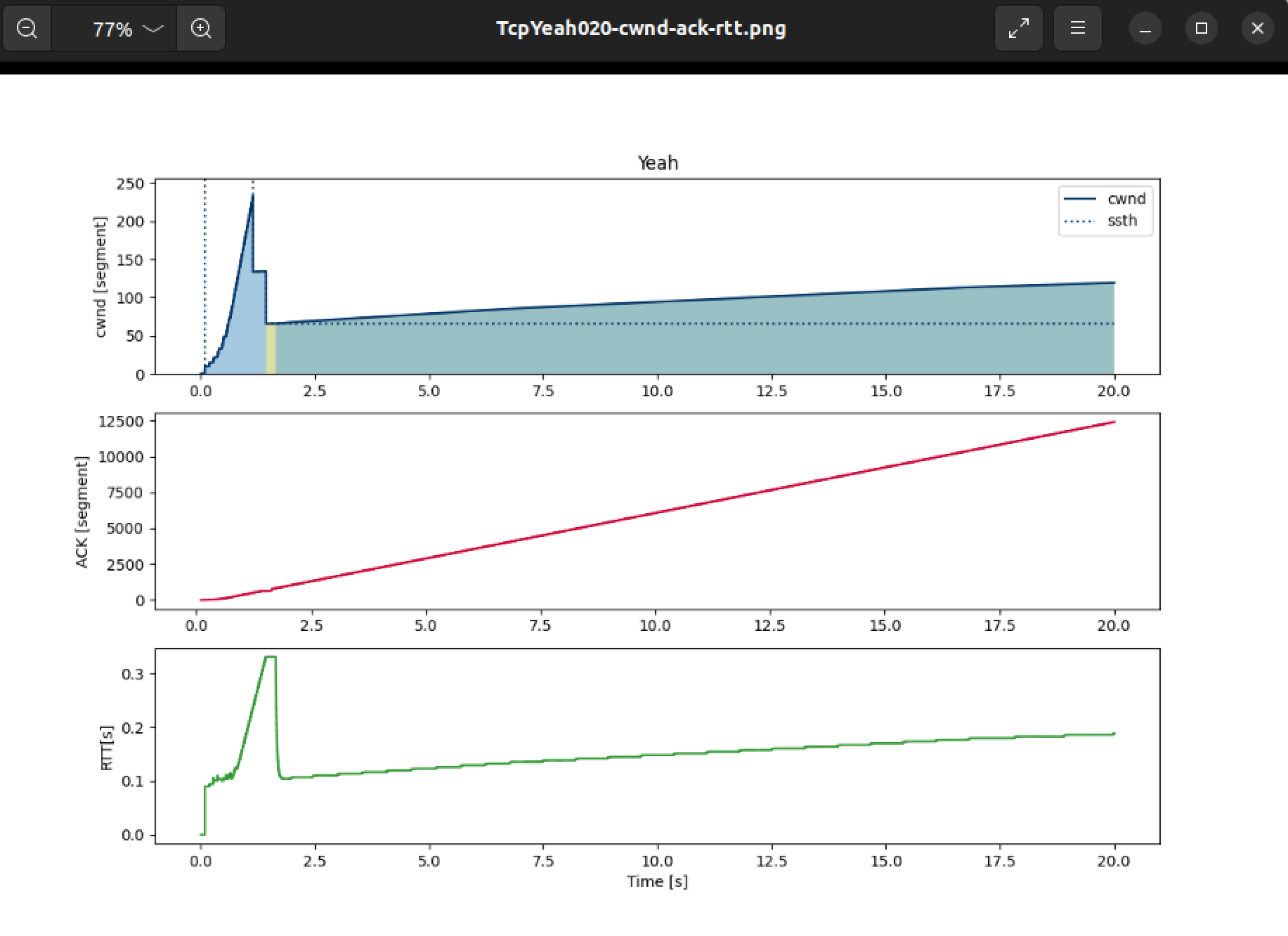


10. **TcpYeah:**

* **基本思想：** TcpYeah主要关注在高速网络中提高性能。通过调整拥塞控制参数，它致力于在高速网络环境中实现更高的吞吐量。
* **作用：** 通过调整拥塞控制参数以适应高速网络的特性。
* **意义：** TcpYeah通过调整参数，使其更适应高速网络，提高了数据传输的效率。
* **适用条件：** 适用于高速网络，如高性能计算集群。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  on\_ack\_received():  if increase\_window\_condition():  increase\_window\_size()  else:  congestion\_avoidance() |

* **算法性能指标**

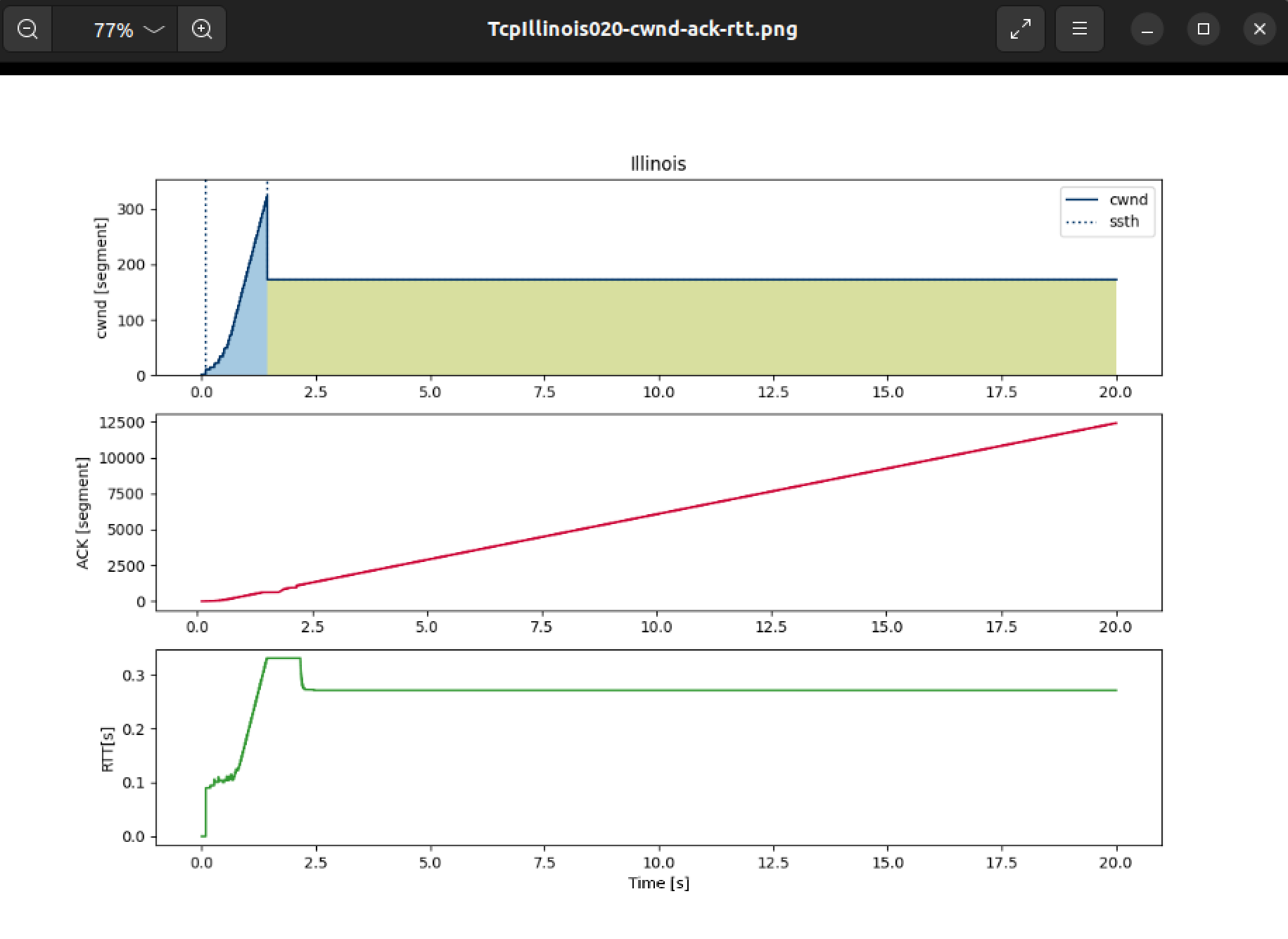


11. **TcpIllinois:**

* **基本思想：**TcpIllinois采用自适应窗口控制和加权平均RTT计算。通过自适应性和RTT加权，它在高速网络中提供相对较好的性能。
* **作用：** 采用自适应窗口控制和加权平均RTT计算，在高速网络中提供较好的性能。
* **意义：** TcpIllinois通过自适应窗口和加权RTT计算，适应网络条件，提高性能。
* **适用条件：** 适用于高速网络，如大规模科研计算环境。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  reduce\_window\_size()  decrease\_beta()  on\_ack\_received():  increase\_window\_size() |

* **算法性能指标**



12. **TcpHtcp:**

* **基本思想：**TcpHtcp基于混合拥塞控制机制，结合了Reno和Cubic算法的特性。它的目标是适用于不同网络条件，综合多种算法的优势。
* **作用：** 基于混合拥塞控制机制，结合了Reno和Cubic算法的特性，适用于不同网络条件。
* **意义：** TcpHtcp综合了不同拥塞控制算法的优点，更适用于多样化的网络环境。
* **适用条件：** 适用于需要适应不同网络条件的场景，如复杂的网络架构。
* **算法伪代码：**

|  |
| --- |
| Python on\_packet\_loss():  if loss\_event\_detected():  htcp\_behavior()  on\_ack\_received():  htcp\_behavior() |

* **算法性能指标**



这些TCP拥塞控制算法在实验中被实现和测试，旨在深入了解它们在特定网络条件下的性能表现，包括吞吐量、延迟、稳定性等方面。

以下是各类TCP拥塞控制算法的拥塞窗口变化情况汇总：



**2.2 路由丢包控制算法简介**

并且，对于路由器丢包控制的研究，本小组实现了以下控制算法：

1. **RED：**

* RED（Random Early Detection）是一种用于拥塞避免的算法，主要用于网络中的队列管理。它的目标是在网络流量接近容量时，通过提前丢弃一些数据包，以避免网络发生拥塞。RED算法的主要思想是在网络流量接近饱和时，通过随机选择一些数据包进行丢弃，向发送方发出拥塞信号，以降低发送速率，从而维持网络的稳定性。
* RED算法的关键参数包括最小门限（Min Threshold）、最大门限（Max Threshold）和最大概率（Max Probability）。当队列长度低于最小门限时，不进行丢包；当队列长度超过最小门限但低于最大门限时，以一定概率进行丢包；当队列长度超过最大门限时，以高概率进行丢包。
* 这个算法的优点之一是它能够在网络拥塞发生之前就开始采取行动，防止网络完全饱和。然而，RED算法的具体配置需要根据网络的特性进行调整，以达到最佳性能。

1. **NLRED：**

* 路由器中队列长度的变化是非线性的,针对原始随机早期检测RED算法在线性丢弃概率增长下的局限性,提出一种非线性高阶函数修正分组丢弃概率的改进RED算法(简称NLRED算法)
* 改进的算法NLRED在提高网络吞吐量、网络链路利用率,减小分组在路由器中的排队时延等方面性能均有所提高。

1. **FengAdaptive：**

* Feng Adaptive RED 是 RED 的一个变体，它适应最大丢弃概率。

通过实验结果的对比，可以评估不同算法组合在哑铃型网络拓扑结构下的优劣势，为网络性能优化提供参考。

**三、实验环境**

**1. 硬件**

三台装置ns3虚拟仿真软件的Windows系统主机和2台装置ns3虚拟仿真软件的MacOS系统主机

**2. 软件**

|  |  |
| --- | --- |
| **名称** | **备注** |
| **NS3** | 版本：3.40 |
| **NetAnim** | 模拟和可视化网络协议和通信行为的工具 |
| **python** | 基于numpy和matplotlib包用于数据可视化 |
| **Wireshark** | 用于查看运行后导出的pcap文件中的数据 |
| **Ubuntu** | 用于提供Linux平台运行算法 |
| **VScode** | 用于编辑C++代码以及python画图代码 |

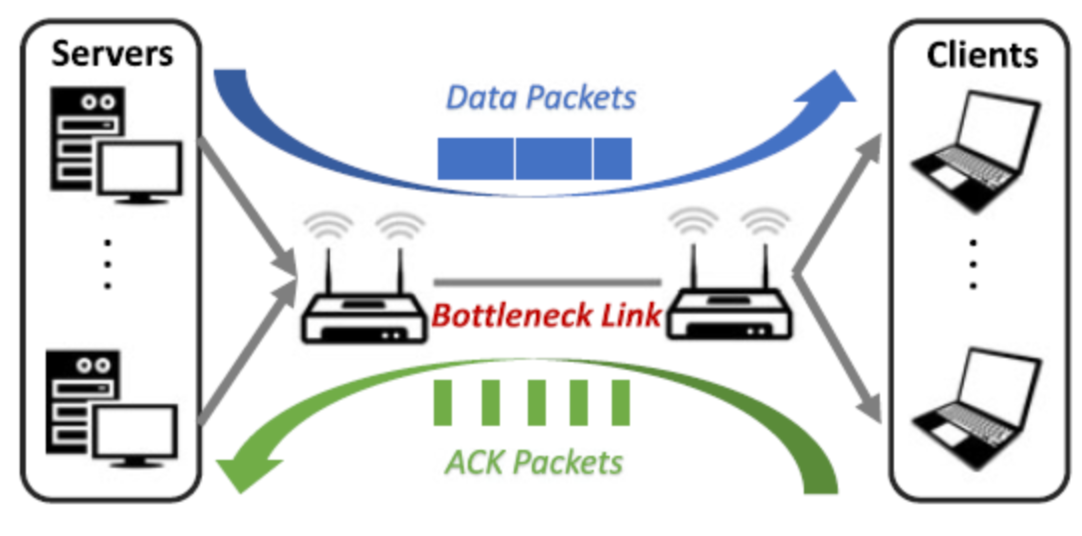
**四、实验网络拓扑结构**

本实验设置的网络拓扑结构如下：

有若干个服务器通过一个中间的路由器进行连接，该路由器扮演着中转和转发数据的角色。连接到路由器的是一个被称为“瓶颈链路”的网络链路，它是整个网络中的瓶颈点，即网络中传输速率较低或容量较小的部分。

瓶颈链路后面连接着另一个路由器，该路由器连接着若干个客户端，这些客户端可能是普通用户设备，如个人电脑、智能手机或其他终端设备。客户端通过该路由器与网络中的服务器进行通信。

这样的网络拓扑结构通常用于模拟或展示实际网络中的一些常见场景，其中服务器和客户端之间通过中间设备（如路由器）进行通信，而瓶颈链路会影响数据传输的性能。



**五、实验场景**

哑铃型网络拓扑结构在实际生活中可以适用于多种场景，尤其是在需要连接两个独立网络的情况下。以下是一些可能的应用场景：

1. **企业网络架构：** 在大型企业中，可能存在多个部门或分支机构，每个部门或分支机构都有其独立的网络。使用哑铃型网络结构，可以连接不同部门或分支机构的网络，同时通过瓶颈链路进行数据传输。
2. **数据中心互连：** 在云计算环境或大规模数据中心中，哑铃型网络结构可用于连接不同的服务器集群。这种拓扑允许数据中心内的服务器相互通信，同时通过瓶颈链路与其他数据中心或云服务提供商连接。
3. **校园网络：** 大学校园可能由多个学院、实验室和图书馆等组成。使用哑铃型网络结构，可以将这些独立的网络连接在一起，促进跨学院或跨部门之间的通信。
4. **分布式办公环境：** 在分布式办公环境中，哑铃型网络结构可用于连接位于不同地理位置的办公室或团队的网络。这样的连接有助于实现分布式团队之间的协作和数据共享。
5. **物联网（IoT）应用：** 在物联网场景中，不同类型的设备可能需要通过网络进行通信。哑铃型网络结构可以用于连接不同种类的传感器网络或物联网设备。

哑铃型网络结构在需要连接两个独立网络并通过独立链路进行通信的情况下非常有用。这种结构提供了一定的灵活性，使得不同网络之间可以相对独立地运行，同时通过瓶颈链路进行必要的通信。

**六、实验内容**

**TCP源端拥塞控制和路由节点丢包控制组合对比实验**

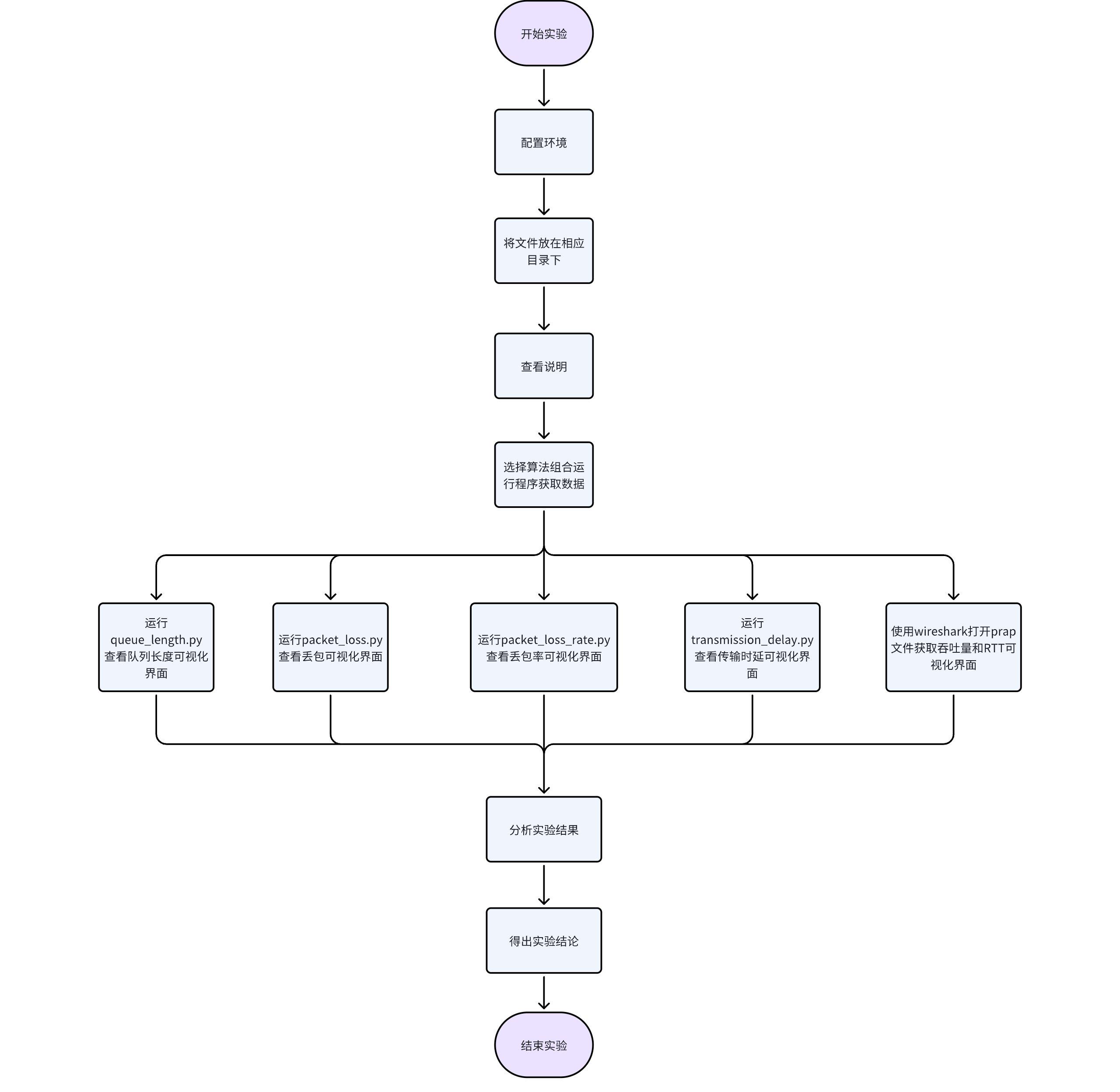
利用哑铃型实验拓扑结构，调整实验参数，模拟生活中企业中部门或校园网络的通信，通过组合11种TCP源端拥塞控制算法和3种路由节点丢包控制算法，研究它们对网络性能的影响。通过对比不同组合的时延、丢包率、队列长度和吞吐量等性能指标，并通过图表的形式展示实验结果，评估各组合的优劣，为网络优化和算法选择提供参考。

**实验参数：**

|  |  |  |
| --- | --- | --- |
| 参数英文名 | 参数大小 | 参数名称 |
| nLeaf | 15 | 叶节点数量 |
| maxPackets | 300 | 最大数据包数量 |
| queueDiscLimitPackets | 100 | 队列限制（数据包数量） |
| pktSize | 512 | 数据包大小 |
| num\_flows | 3 | 流数量 |
| duration | 150 | 持续时间 |
| minTh | 20 | RED算法的最小阈值 |
| maxTh | 60 | RED算法的最大阈值 |
| error\_p | 0.01 | 错误概率 |
| appDataRate | 100Mbps | 应用数据速率 |
| bottleNeckLinkBw | 10Mbps | 瓶颈链路带宽 |
| bottleNeckLinkDelay | 50ms | 瓶颈链路延迟 |
| leafNeckLinkBw | 100Mbps | 点对点链路带宽 |
| leafLinkDelay | 20ms | 点对点链路延迟 |

**七、实验步骤与现象**

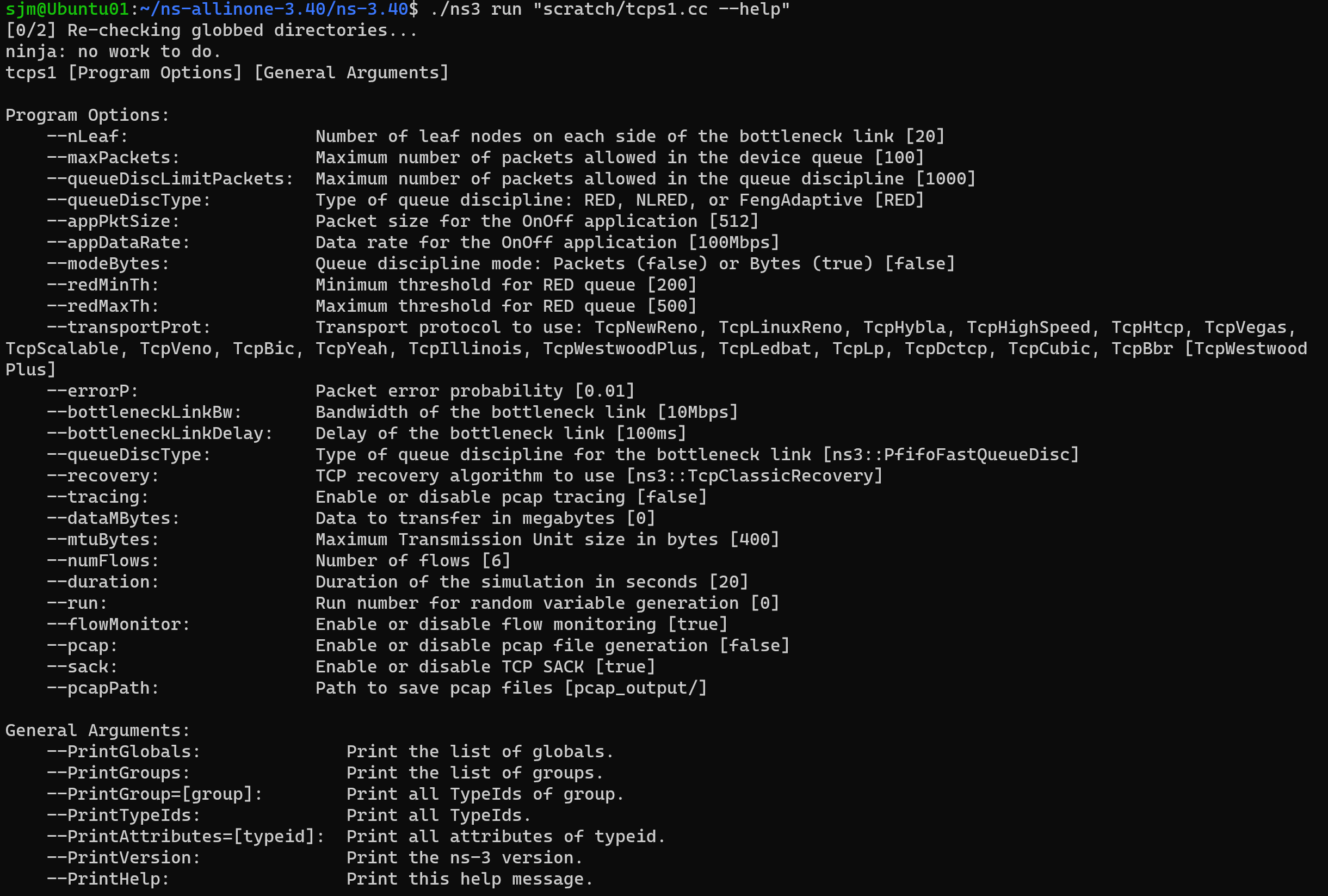
**实验总体流程图如下：**



1. 配置ns-3.40实验环境，下载python的numpy和matplotlib包
2. 将tcps1.cc文件置于/ns-allinone-3.40/ns-3.40/scratch路径下，并将packet\_loss.py,packet\_loss\_rate.py,queue\_length.py文件置于/ns-allinone-3.40/ns-3.40路径下
3. 进入/ns-allinone-3.40/ns-3.40路径，查看tcps1.cc的使用参数说明

|  |
| --- |
| Python ./ns3 run "scratch/tcps1.cc --help" |

参数类型及解释如下：



由上，可以通过手动输入参数值来修改默认值以贴合实际场景。

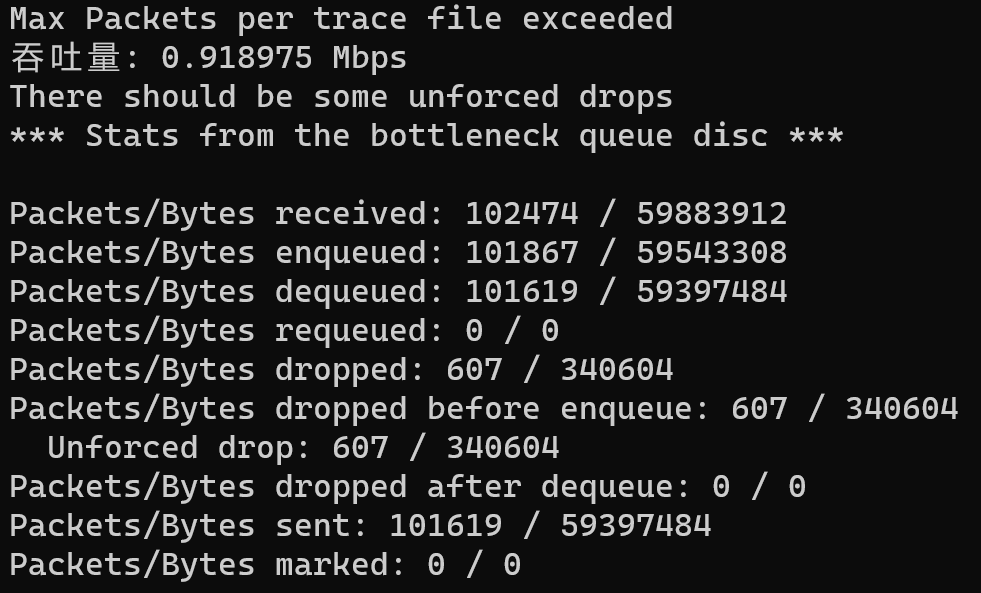
1. 从3种路由算法和11种TCP算法中选择一种组合，根据实际情况选择修改的参数，运行tcps1.cc脚本：

（以修改叶节点nLeaf为25，修改持续时间duration为50s，选择TCP算法为TcpNewReno，选择路由算法为NLRED为例）



|  |
| --- |
| Python ./ns3 run "scratch/tcps1.cc --nLeaf=25 --duration=50 --queueDiscType=NLRED --transportProt=TcpNewReno" |

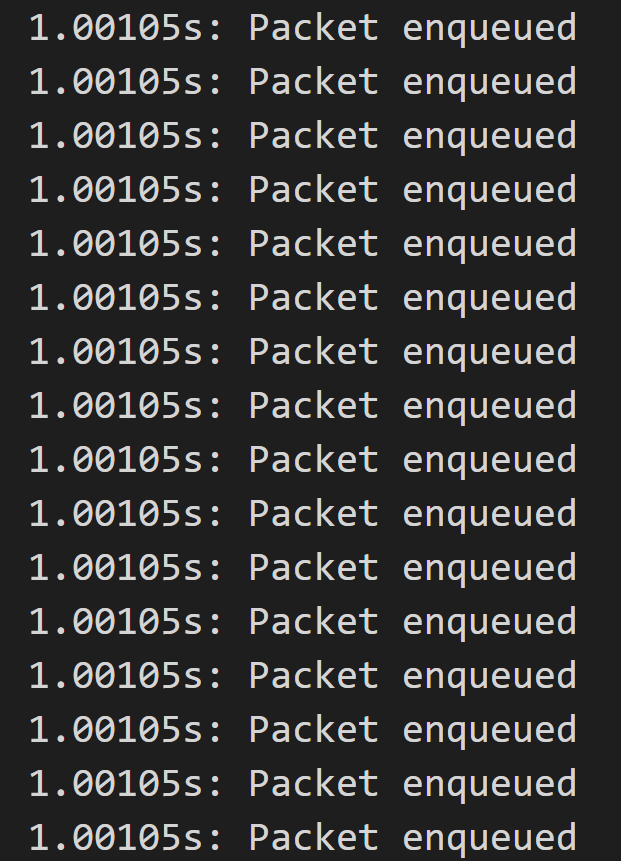
程序运行结束后获取以下输出结果：



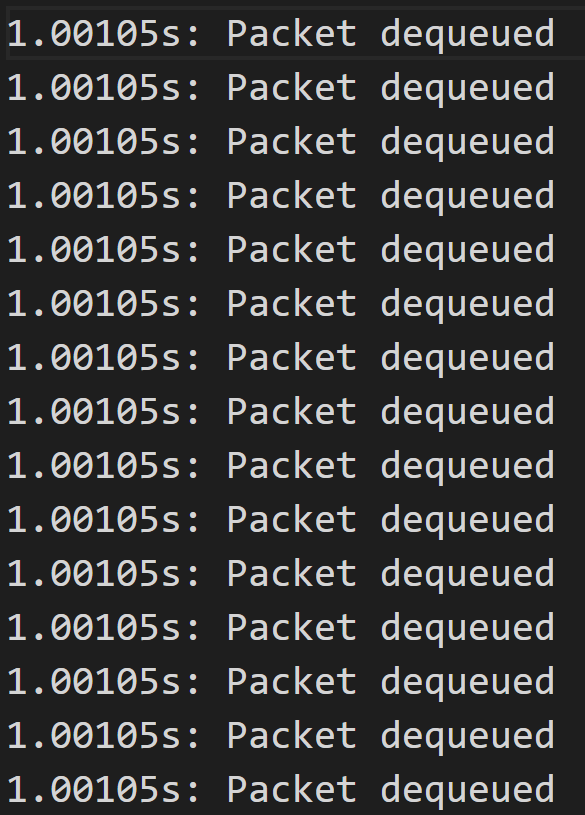
结果打印该算法组合在该情景下的吞吐量、数据包接收、进队列、出队列、丢包情况等信息。

并且在同级目录下生成enqueue.tr dequeue.tr queue\_drop.tr queue\_length.tr四个文件及pcap文件

其中，enqueue.tr文件中记录了数据包进队列的时间，以下是部分数据展示：



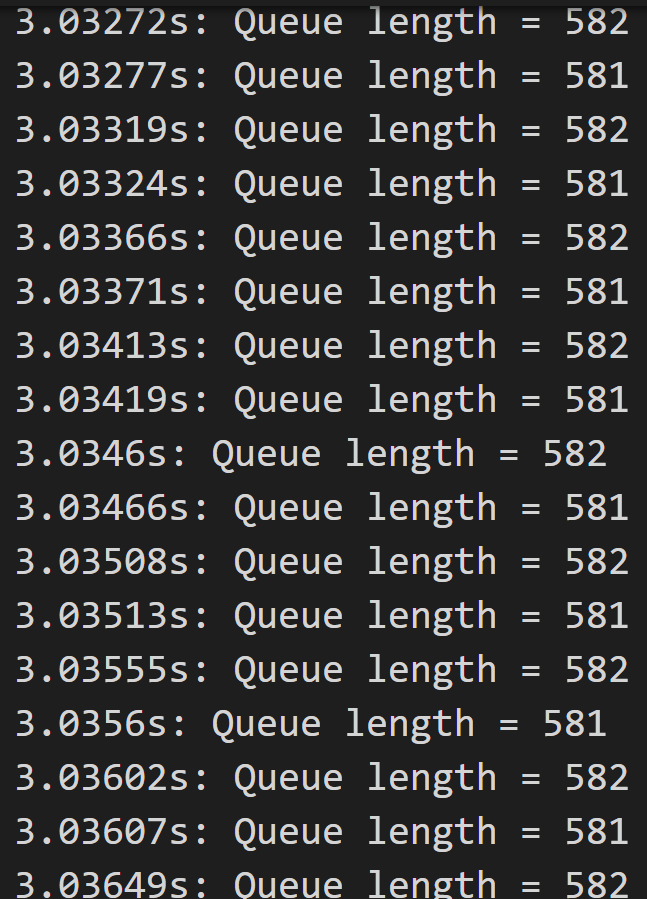
dequeue.tr文件记录了数据包离开队列的时间，以下是部分数据展示：



queue\_drop.tr文件记录了数据包丢包时间，以下是部分数据展示：

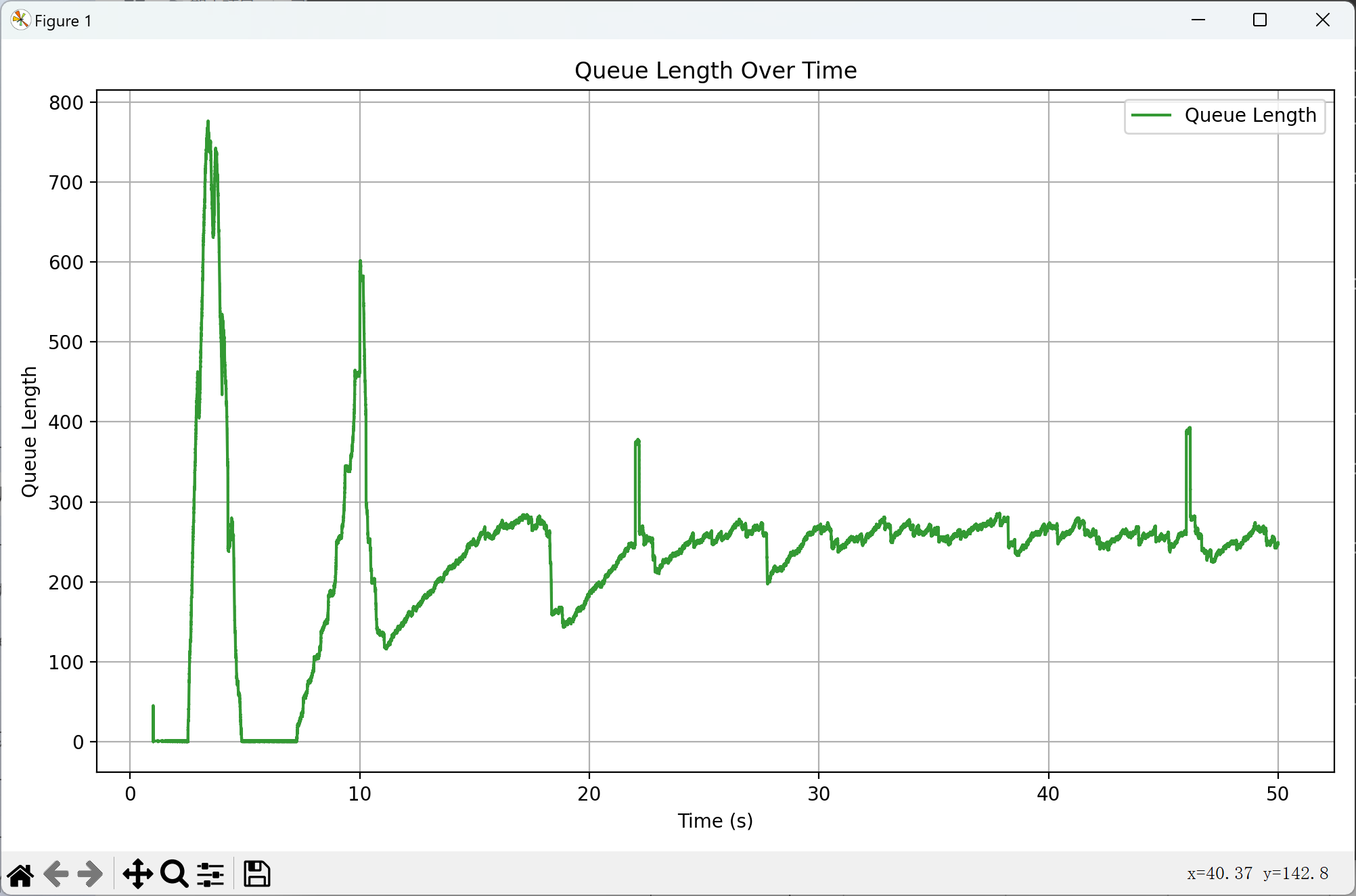


queue\_length.tr文件记录了队列长度随时间的变化，以下是部分数据展示：



1. 运行queue\_length.py文件，得到可视化队列随时间的变化界面：

|  |
| --- |
| Python python queue\_length.py |



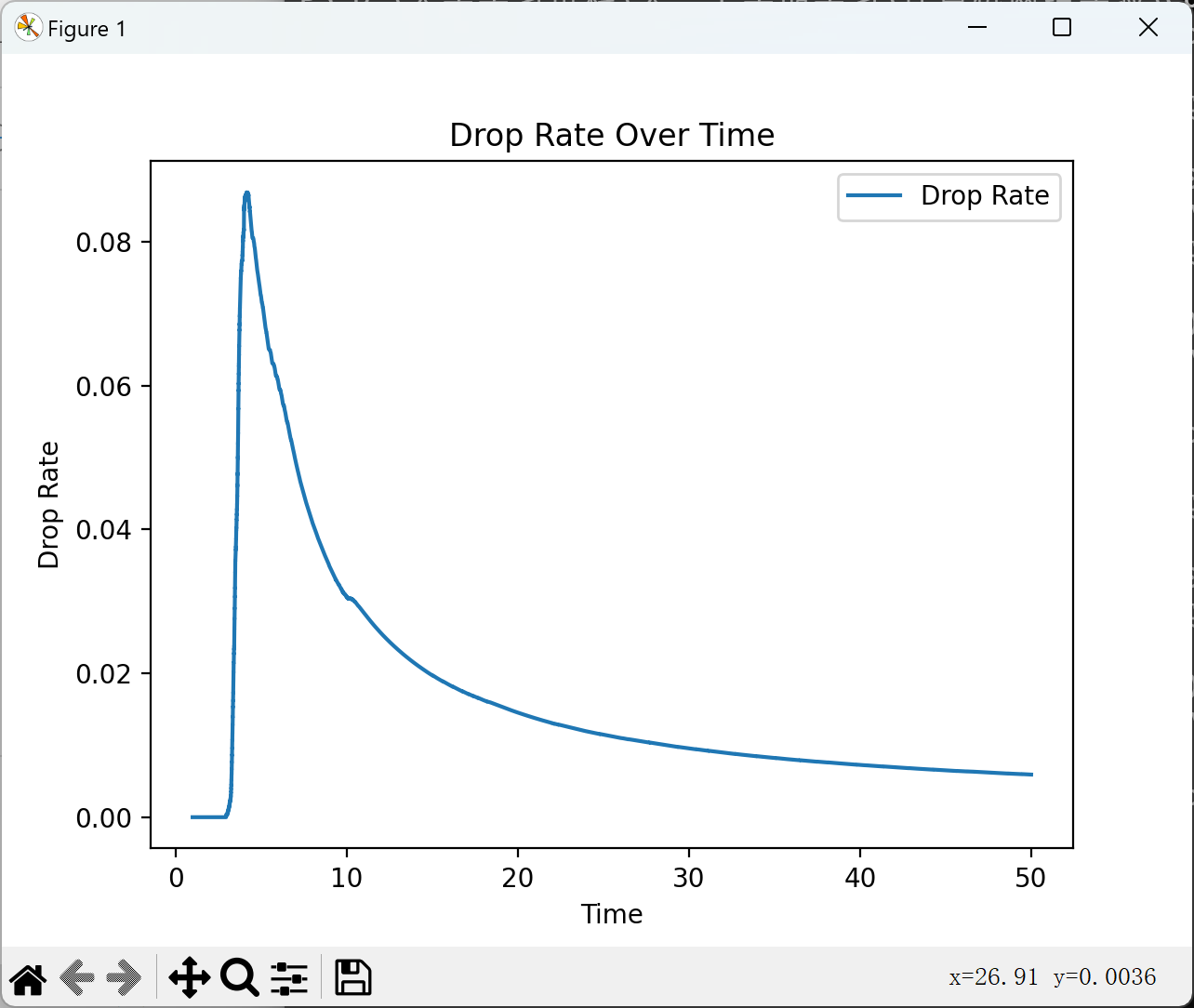
1. 运行packet\_loss.py文件，得到丢包时机的界面：

|  |
| --- |
| Python python packet\_loss.py |



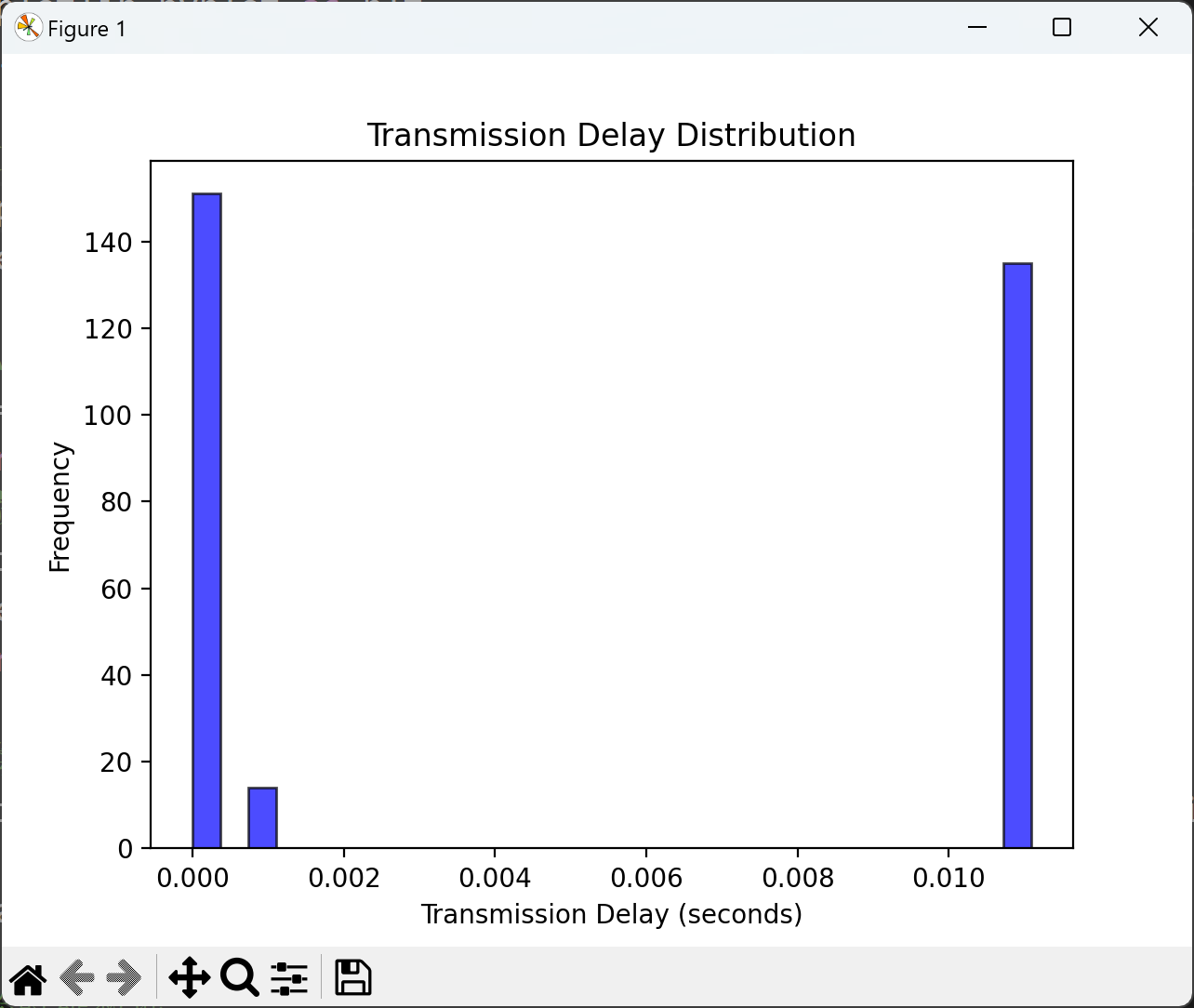
1. 运行packet\_loss\_rate.py文件，得到丢包率随时间变化的可视化界面：

|  |
| --- |
| Python python packet\_loss\_rate.py |



1. 运行transmission\_delay.py文件，得到不同传输时延值的频率分布可视化界面：

|  |
| --- |
| Python python transmission\_delay.py |

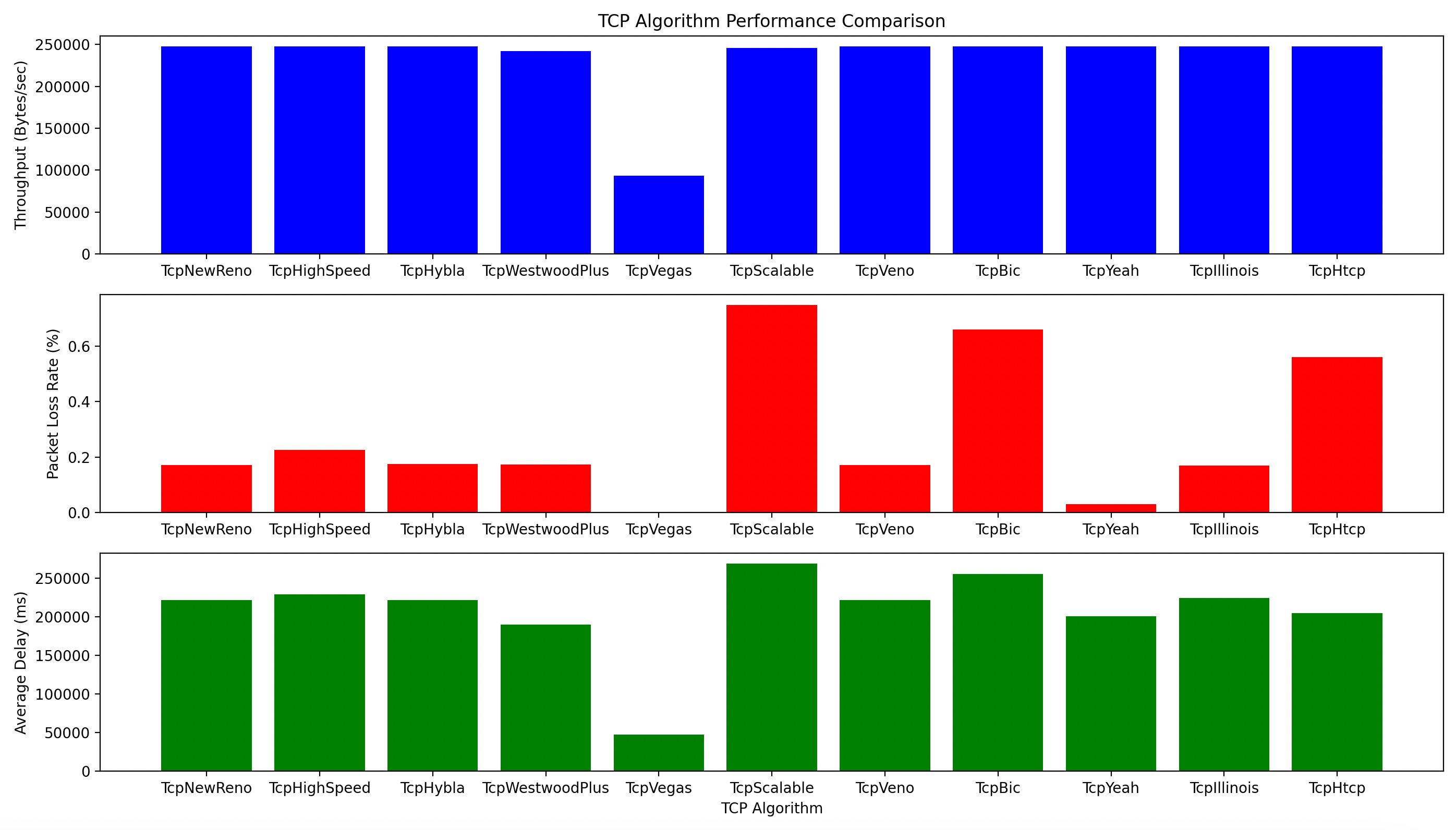


1. 用wireshark打开pcap文件，观察吞吐量和往返时间

|  |  |
| --- | --- |
|  |  |

1. 重复步骤4-9，获取多种组合的性能指标结果，分析实验数据，得出结论。

**八、实验小结**



基于实验数据，我们观察到以下几个关键点：

1. 吞吐量:

* 除了TcpVegas外，大多数TCP算法的吞吐量相近，都在247851 Bytes/sec左右。这表明在这些条件下，除了TcpVegas外，其他算法的吞吐量表现相对一致。
* TcpVegas的吞吐量显著低于其他算法，仅为93424.99 Bytes/sec。

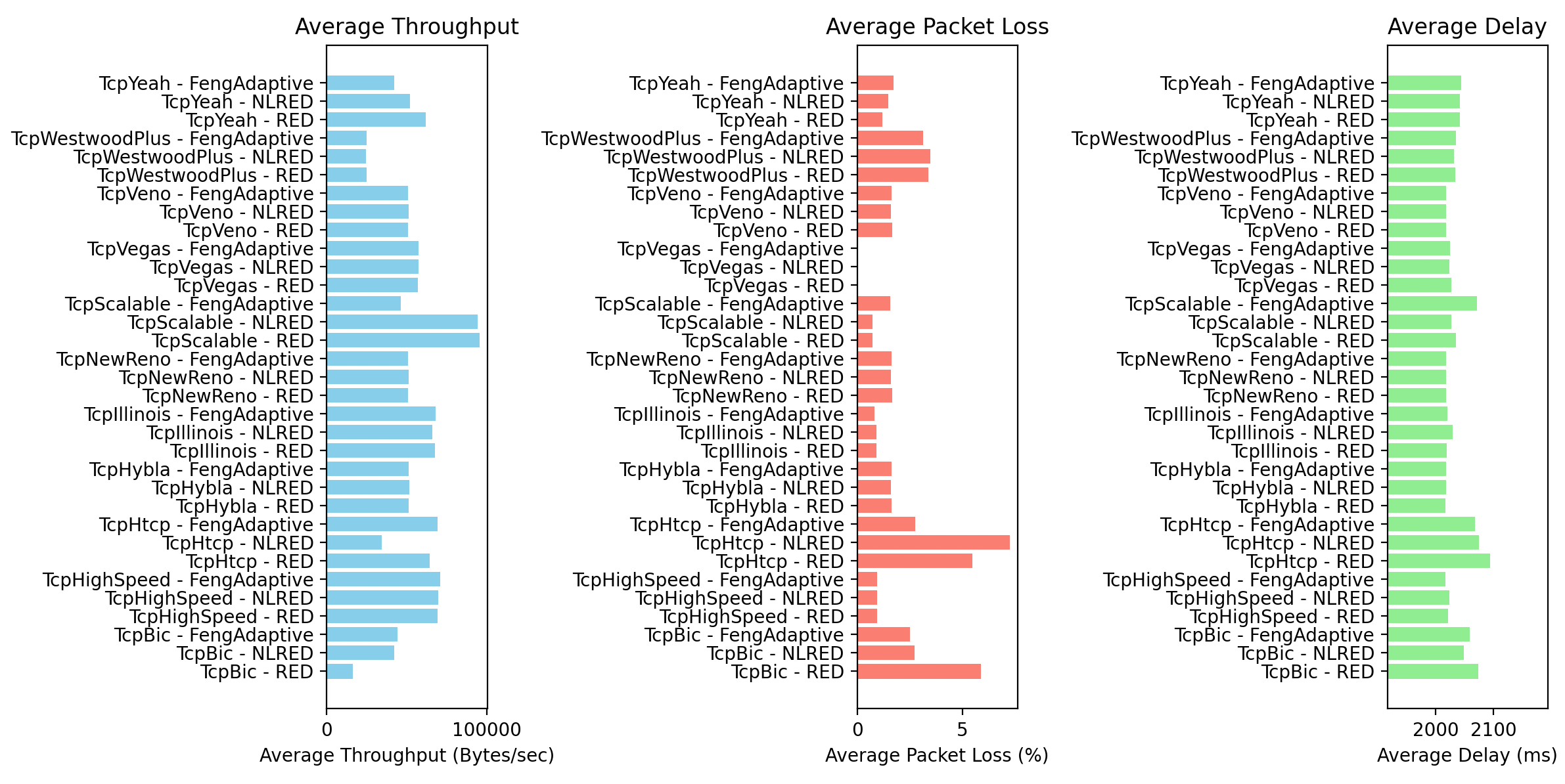
1. 丢包率:

* TcpYeah展现出最低的丢包率，仅为0.02998%。这表明TcpYeah在这些条件下非常可靠。
* TcpScalable和TcpBic的丢包率相对较高，分别为0.74883%和0.65955%。

1. 平均时延:

* TcpVegas的平均时延最低，为47699.40 ms，这可能与其较低的吞吐量有关。
* TcpScalable的平均时延最高，为269391.85 ms，这表明在高吞吐量条件下，时延可能增加。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TCP算法 | 队列管理算法 | 吞吐量 (Bytes/sec) | 丢包率 (%) | 平均时延 (ms) |
| TcpBic | RED | 16338.33 | 5.8731% | 2073.09 |
| TcpBic | NLRED | 42084.14 | 2.7323% | 2048.30 |
| TcpBic | FengAdaptive | 44231.83 | 2.5064% | 2058.41 |
| TcpHighSpeed | RED | 69344.02 | 0.9569% | 2021.10 |
| TcpHighSpeed | NLRED | 69664.02 | 0.9583% | 2023.76 |
| TcpHighSpeed | FengAdaptive | 70699.55 | 0.9591% | 2016.20 |
| TcpHtcp | RED | 64142.70 | 5.4753% | 2094.32 |
| TcpHtcp | NLRED | 34258.49 | 7.2608% | 2075.10 |
| TcpHtcp | FengAdaptive | 69267.12 | 2.7721% | 2068.40 |
| TcpHybla | RED | 51255.22 | 1.6367% | 2017.22 |
| TcpHybla | NLRED | 51668.40 | 1.5984% | 2017.65 |
| TcpHybla | FengAdaptive | 51364.42 | 1.6253% | 2018.29 |
| TcpIllinois | RED | 67687.55 | 0.9084% | 2019.17 |
| TcpIllinois | NLRED | 65923.55 | 0.9324% | 2029.30 |
| TcpIllinois | FengAdaptive | 68135.55 | 0.8145% | 2020.64 |
| TcpNewReno | RED | 50664.28 | 1.6571% | 2017.31 |
| TcpNewReno | NLRED | 51125.79 | 1.6169% | 2017.75 |
| TcpNewReno | FengAdaptive | 50757.40 | 1.6456% | 2018.36 |
| TcpScalable | RED | 95451.57 | 0.7219% | 2034.64 |
| TcpScalable | NLRED | 94264.86 | 0.7456% | 2027.40 |
| TcpScalable | FengAdaptive | 46428.12 | 1.5764% | 2071.50 |
| TcpVegas | RED | 56792.31 | 0.0000% | 2027.42 |
| TcpVegas | NLRED | 57359.93 | 0.0000% | 2023.91 |
| TcpVegas | FengAdaptive | 57227.93 | 0.0000% | 2025.17 |
| TcpVeno | RED | 50664.28 | 1.6571% | 2017.31 |
| TcpVeno | NLRED | 51125.79 | 1.6169% | 2017.75 |
| TcpVeno | FengAdaptive | 50757.40 | 1.6456% | 2018.36 |
| TcpWestwoodPlus | RED | 24823.07 | 3.3813% | 2034.06 |
| TcpWestwoodPlus | NLRED | 24669.77 | 3.4800% | 2031.80 |
| TcpWestwoodPlus | FengAdaptive | 24862.35 | 3.1406% | 2034.40 |
| TcpYeah | RED | 61890.65 | 1.2154% | 2041.95 |
| TcpYeah | NLRED | 52101.25 | 1.4743% | 2041.40 |
| TcpYeah | FengAdaptive | 42250.85 | 1.7413% | 2044.16 |



基于实验数据，我们可以得出以下几点结论：

1. **吞吐量分析细化：**

* **高性能分析：** TcpScalable 在 RED 和 NLRED 队列管理算法下表现最佳，其吞吐量分别高达 95451.57 Bytes/sec 和 94264.86 Bytes/sec。这一表现突出，显示 TcpScalable 在处理大量数据时的优越性，特别适合于数据中心和大规模分布式应用。
* **性能对比：** 相对地，TcpWestwoodPlus 的吞吐量最低，这可能是由于其算法设计更侧重于其他性能指标，如更平滑的速率调整，而不是最大化吞吐量。这提示我们在网络环境和应用需求多样的情况下，需要更加细致地选择TCP算法。

1. **丢包率分析细化：**

* **稳定性突出：** TcpVegas 在所有环境下的丢包率均为0%，这不仅显示了其卓越的稳定性，还可能表明其拥塞控制机制在预防丢包方面非常有效。这使得TcpVegas成为对网络质量要求极高的应用，如视频会议和实时游戏的理想选择。
* **问题识别：** TcpHtcp 在 NLRED 下的高丢包率（7.2608%）可能指出了该算法在特定网络条件下的局限性，需要进一步研究其在不同网络拓扑和流量条件下的表现，以优化其性能。

1. **平均时延分析细化：**

* **时延稳定性：** 大多数算法在不同队列管理下的时延相对稳定，这表明这些TCP算法能够在不同的网络负载条件下保持一致的延迟表现，这对于需要低延迟的应用至关重要。
* **算法敏感性：** TcpBic 和 TcpHtcp 显示出对队列管理算法的高度敏感性，这提示我们在选择TCP算法时，需要考虑网络的队列管理策略，以确保最优的时延表现。

1. **综合比较与应用场景建议：**

* **高效与可靠性：** TcpScalable 和 TcpHighSpeed 在不同队列管理下均显示出良好的吞吐量和低丢包率，使它们成为要求高效率和可靠性的网络环境（如企业级应用和云计算环境）的理想选择。
* **稳定性与应用适配：** TcpVegas 的0%丢包率和适中吞吐量使其成为对网络稳定性有高要求的应用的理想选择，例如VoIP和在线游戏。

**总结：**

这项研究揭示了不同TCP算法在各种队列管理策略下的表现差异，强调了根据具体网络环境和应用需求选择合适TCP算法的重要性。例如，数据密集型和高可靠性需求的场景更适合TcpScalable和TcpHighSpeed，而对于需要极低丢包率的实时应用，则TcpVegas是更佳选择。未来的研究可以进一步探索这些算法在不同网络拓扑和流量模式下的表现，以及如何通过调整算法参数来优化它们在特定场景下的性能。

1. **TcpNewReno**

* TCPNewReno+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TCPNewReno+RED

|  |  |
| --- | --- |
|  |  |

* TCPNewReno+NLRED

|  |  |
| --- | --- |
|  |  |

2. **TcpHybla**

* TcpHybla+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpHybla+RED

|  |  |
| --- | --- |
|  |  |

* TcpHybla+NLRED

|  |  |
| --- | --- |
|  |  |

3. **TcpHighSpeed**

* TcpHighSpeed+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpHighSpeed+RED

|  |  |
| --- | --- |
|  |  |

* TcpHighSpeed+NLRED

|  |  |
| --- | --- |
|  |  |

4. **TcpHtcp**

* TcpHtcp+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpHtcp+RED

|  |  |
| --- | --- |
|  |  |

* TcpHtcp+NLRED

|  |  |
| --- | --- |
|  |  |

5. **TcpVegas**

* TcpVegas +Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpVegas +RED

|  |  |
| --- | --- |
|  |  |

* TcpVegas +NLRED

|  |  |
| --- | --- |
|  |  |

6. **TcpScalable**

* TcpScalable+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpScalable+RED

|  |  |
| --- | --- |
|  |  |

* TcpScalable+NLRED

|  |  |
| --- | --- |
|  |  |

7. **TcpVeno**

* TcpVeno+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpVeno+RED

|  |  |
| --- | --- |
|  |  |

* TcpVeno+NLRED

|  |  |
| --- | --- |
|  |  |

8. **TcpBic**

* TcpBic+Fengadaptive

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

* TcpBic+RED

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

* TcpBic+NLRED

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

9. **TcpYeah**

* TcpYeah+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpYeah+RED

|  |  |
| --- | --- |
|  |  |

* TcpYeah+NLRED

|  |  |
| --- | --- |
|  |  |

10. **TcpIllinois**

* TcpIllinois+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpIllinois+RED

|  |  |
| --- | --- |
|  |  |

* TcpIllinois+NLRED

|  |  |
| --- | --- |
|  |  |

11. **TcpWestwoodPlus**

* TcpWestwoodPlus+Fengadaptive

|  |  |
| --- | --- |
|  |  |

* TcpWestwoodPlus+RED

|  |  |
| --- | --- |
|  |  |

* TcpWestwoodPlus+NLRED

|  |  |
| --- | --- |
|  |  |

**九、附录**

**tscp1.cc**

|  |
| --- |
| C++ #include "ns3/applications-module.h" #include "ns3/core-module.h" #include "ns3/internet-module.h" #include "ns3/network-module.h" #include "ns3/point-to-point-layout-module.h" #include "ns3/point-to-point-module.h" #include "ns3/traffic-control-module.h" #include "ns3/netanim-module.h" #include "ns3/mobility-module.h" #include "ns3/flow-monitor-module.h"  #include <iomanip> #include <iostream> #include <map>  using namespace ns3;  // --定义全局的 AsciiTraceHelper 对象 AsciiTraceHelper ascii;  // --定义丢包回调函数 void YourDropTraceFunction(std::string context, Ptr<const QueueDiscItem> item) {  static Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream("queue\_drop.tr");  }  // --队列长度回调函数 void QueueLengthTrace(std::string context, uint32\_t oldValue, uint32\_t newValue) {  static Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream("queue\_length.tr");  }  // --监控 Dequeue 和 Enqueue 事件 void EnqueueTrace(std::string context, Ptr<const QueueDiscItem> item) {  static Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream("enqueue.tr");  }  void DequeueTrace(std::string context, Ptr<const QueueDiscItem> item) {  static Ptr<OutputStreamWrapper> stream = ascii.CreateFileStream("dequeue.tr");  }  int main(int argc, char\* argv[]) {     // 网络拓扑和通用参数  uint32\_t nLeaf = 20; // 叶节点数量  uint32\_t maxPackets = 100; // 最大数据包数量  bool modeBytes = false; // 模式为字节（false 表示模式为数据包）  uint32\_t queueDiscLimitPackets = 1000; // 队列限制（数据包数量）  uint32\_t pktSize = 512; // 数据包大小（字节）  uint16\_t port = 1024; // 使用的端口  std::string pcapPath = "pcap\_output/"; // pcap文件路径  bool tracing = false; // 是否启用跟踪  uint64\_t data\_mbytes = 0; // 数据量（兆字节）  uint32\_t mtu\_bytes = 400; // 最大传输单元  uint16\_t num\_flows = 6; // 流数量  double duration = 20.0; // 持续时间  uint32\_t run = 0; // 运行次数  bool flow\_monitor = true; // 是否监控流量  bool pcap = false; // 是否生成pcap文件  bool sack = true; // 是否启用选择性确认   // RED 队列参数  double minTh = 200; // RED算法的最小阈值  double maxTh = 500; // RED算法的最大阈值  std::string queueDiscType = "RED"; // 队列类型   // TCP和链路参数  std::string transport\_prot = "TcpWestwoodPlus"; // TCP拥塞控制算法  double error\_p = 0.01; // 错误概率  std::string appDataRate = "100Mbps"; // 应用数据速率  std::string bottleNeckLinkBw = "10Mbps"; // 瓶颈链路带宽  std::string bottleNeckLinkDelay = "100ms"; // 瓶颈链路延迟  std::string queue\_disc\_type = "ns3::PfifoFastQueueDisc"; // 队列算法类型  std::string recovery = "ns3::TcpClassicRecovery"; // 恢复算法类型   // 命令行参数设置  CommandLine cmd(\_\_FILE\_\_);  cmd.AddValue("nLeaf", "Number of leaf nodes on each side of the bottleneck link", nLeaf);  cmd.AddValue("maxPackets", "Maximum number of packets allowed in the device queue", maxPackets);  cmd.AddValue("queueDiscLimitPackets", "Maximum number of packets allowed in the queue discipline", queueDiscLimitPackets);  cmd.AddValue("queueDiscType", "Type of queue discipline: RED, NLRED, or FengAdaptive", queueDiscType);  cmd.AddValue("appPktSize", "Packet size for the OnOff application", pktSize);  cmd.AddValue("appDataRate", "Data rate for the OnOff application", appDataRate);  cmd.AddValue("modeBytes", "Queue discipline mode: Packets (false) or Bytes (true)", modeBytes);  cmd.AddValue("redMinTh", "Minimum threshold for RED queue", minTh);  cmd.AddValue("redMaxTh", "Maximum threshold for RED queue", maxTh);  cmd.AddValue("transportProt",   "Transport protocol to use: TcpNewReno, TcpLinuxReno, "  "TcpHybla, TcpHighSpeed, TcpHtcp, TcpVegas, TcpScalable, TcpVeno, "  "TcpBic, TcpYeah, TcpIllinois, TcpWestwoodPlus, TcpLedbat, "  "TcpLp, TcpDctcp, TcpCubic, TcpBbr",  transport\_prot);  cmd.AddValue("errorP", "Packet error probability", error\_p);  cmd.AddValue("bottleneckLinkBw", "Bandwidth of the bottleneck link", bottleNeckLinkBw);  cmd.AddValue("bottleneckLinkDelay", "Delay of the bottleneck link", bottleNeckLinkDelay);  cmd.AddValue("queueDiscType", "Type of queue discipline for the bottleneck link", queue\_disc\_type);  cmd.AddValue("recovery", "TCP recovery algorithm to use", recovery);  cmd.AddValue("tracing", "Enable or disable pcap tracing", tracing);  cmd.AddValue("dataMBytes", "Data to transfer in megabytes", data\_mbytes);  cmd.AddValue("mtuBytes", "Maximum Transmission Unit size in bytes", mtu\_bytes);  cmd.AddValue("numFlows", "Number of flows", num\_flows);  cmd.AddValue("duration", "Duration of the simulation in seconds", duration);  cmd.AddValue("run", "Run number for random variable generation", run);  cmd.AddValue("flowMonitor", "Enable or disable flow monitoring", flow\_monitor);  cmd.AddValue("pcap", "Enable or disable pcap file generation", pcap);  cmd.AddValue("sack", "Enable or disable TCP SACK", sack);  cmd.AddValue("pcapPath", "Path to save pcap files", pcapPath);  // 可以继续添加更多的参数  cmd.Parse(argc, argv);   // 检查队列规则类型是否有效  if ((queueDiscType != "RED") && (queueDiscType != "FengAdaptive") && (queueDiscType != "NLRED"))  {  std::cout << "Invalid queue disc type: Use --queueDiscType=RED, --queueDiscType=NLRED, or --queueDiscType=FengAdaptive" << std::endl;  exit(1);  }   // 设置应用层数据包的大小和数据速率  Config::SetDefault("ns3::OnOffApplication::PacketSize", UintegerValue(pktSize));  Config::SetDefault("ns3::OnOffApplication::DataRate", StringValue(appDataRate));   // 配置设备队列的最大容量  Config::SetDefault("ns3::DropTailQueue<Packet>::MaxSize",  QueueSizeValue(QueueSize(QueueSizeUnit::PACKETS, maxPackets)));   // 根据模式设置RED队列的最大容量  if (!modeBytes)  {  // 按数据包数量配置队列大小  Config::SetDefault("ns3::RedQueueDisc::MaxSize",  QueueSizeValue(QueueSize(QueueSizeUnit::PACKETS, queueDiscLimitPackets)));  }  else  {  // 按字节大小配置队列大小，并调整阈值  Config::SetDefault("ns3::RedQueueDisc::MaxSize",  QueueSizeValue(QueueSize(QueueSizeUnit::BYTES, queueDiscLimitPackets \* pktSize)));  minTh \*= pktSize;  maxTh \*= pktSize;  }   // 设置RED队列的其他参数  Config::SetDefault("ns3::RedQueueDisc::MinTh", DoubleValue(minTh));  Config::SetDefault("ns3::RedQueueDisc::MaxTh", DoubleValue(maxTh));  Config::SetDefault("ns3::RedQueueDisc::LinkBandwidth", StringValue(bottleNeckLinkBw));  Config::SetDefault("ns3::RedQueueDisc::LinkDelay", StringValue(bottleNeckLinkDelay));  Config::SetDefault("ns3::RedQueueDisc::MeanPktSize", UintegerValue(pktSize));   // 根据选定的队列规则类型配置额外的特性  if (queueDiscType == "FengAdaptive")  {  // 设置FengAdaptive，一种RED的变体  Config::SetDefault("ns3::RedQueueDisc::FengAdaptive", BooleanValue(true));  }  else if (queueDiscType == "NLRED")  {  // 设置NLRED，另一种RED的变体  Config::SetDefault("ns3::RedQueueDisc::NLRED", BooleanValue(true));  }  // 配置 TCP 参数  transport\_prot = std::string("ns3::") + transport\_prot;  // 设置 TCP 恢复类型  Config::SetDefault("ns3::TcpL4Protocol::RecoveryType", TypeIdValue(TypeId::LookupByName(recovery)));  // 配置 TCP 套接字的接收和发送缓冲区大小  Config::SetDefault("ns3::TcpSocket::RcvBufSize", UintegerValue(1 << 21)); // 2 MB  Config::SetDefault("ns3::TcpSocket::SndBufSize", UintegerValue(1 << 21)); // 2 MB  // 设置 TCP SACK 选项  Config::SetDefault("ns3::TcpSocketBase::Sack", BooleanValue(sack));  // 确保 TCP 变体类型有效  TypeId tcpTid;  NS\_ABORT\_MSG\_UNLESS(TypeId::LookupByNameFailSafe(transport\_prot, &tcpTid), "TypeId " << transport\_prot << " not found");  Config::SetDefault("ns3::TcpL4Protocol::SocketType", TypeIdValue(tcpTid));   // 创建瓶颈链路属性  PointToPointHelper bottleNeckLink;  bottleNeckLink.SetDeviceAttribute("DataRate", StringValue(bottleNeckLinkBw));  bottleNeckLink.SetChannelAttribute("Delay", StringValue(bottleNeckLinkDelay));   // 创建点对点叶节点链路属性  PointToPointHelper pointToPointLeaf;  pointToPointLeaf.SetDeviceAttribute("DataRate", StringValue("10Mbps"));  pointToPointLeaf.SetChannelAttribute("Delay", StringValue("1ms"));   // 构建哑铃拓扑  PointToPointDumbbellHelper d(nLeaf, pointToPointLeaf, nLeaf, pointToPointLeaf, bottleNeckLink);   // 添加错误模型到链路上  Ptr<RateErrorModel> em = CreateObject<RateErrorModel>();  em->SetAttribute("ErrorRate", DoubleValue(error\_p));  bottleNeckLink.SetDeviceAttribute("ReceiveErrorModel", PointerValue(em));    // 安装网络堆栈  InternetStackHelper stack;  for (uint32\_t i = 0; i < d.LeftCount(); ++i)  {  stack.Install(d.GetLeft(i));  }  for (uint32\_t i = 0; i < d.RightCount(); ++i)  {  stack.Install(d.GetRight(i));  }   stack.Install(d.GetLeft());  stack.Install(d.GetRight());    // 安装流量控制  TrafficControlHelper tchBottleneck;  tchBottleneck.SetRootQueueDisc("ns3::RedQueueDisc");  // 为瓶颈链路两端安装 RED 队列  tchBottleneck.Install(d.GetLeft()->GetDevice(0));  QueueDiscContainer queueDiscs = tchBottleneck.Install(d.GetRight()->GetDevice(0));   //监控Enqueue和Dequeue  queueDiscs.Get(0)->TraceConnect("Enqueue", "bottleneck-enqueue", MakeCallback(&EnqueueTrace));  queueDiscs.Get(0)->TraceConnect("Dequeue", "bottleneck-dequeue", MakeCallback(&DequeueTrace));   // 为每个网络分配IPv4地址  d.AssignIpv4Addresses(Ipv4AddressHelper("10.1.1.0", "255.255.255.0"),  Ipv4AddressHelper("10.2.1.0", "255.255.255.0"),  Ipv4AddressHelper("10.3.1.0", "255.255.255.0"));   // 在所有右侧节点上安装 On/Off 应用  OnOffHelper clientHelper("ns3::TcpSocketFactory", Address());  clientHelper.SetAttribute("OnTime", StringValue("ns3::ExponentialRandomVariable[Mean=1.0]"));  clientHelper.SetAttribute("OffTime", StringValue("ns3::ExponentialRandomVariable[Mean=1.0]"));  Address sinkLocalAddress(InetSocketAddress(Ipv4Address::GetAny(), port));  PacketSinkHelper packetSinkHelper("ns3::TcpSocketFactory", sinkLocalAddress);   // --创建和配置OnOffHelper  // OnOffHelper clientHelper("ns3::TcpSocketFactory", Address());  // clientHelper.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[Constant=1]"));  // clientHelper.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0]"));  clientHelper.SetAttribute("DataRate", StringValue(appDataRate));  clientHelper.SetAttribute("PacketSize", UintegerValue(pktSize));  clientHelper.SetAttribute("StartTime", TimeValue(Seconds(1.0)));   ApplicationContainer sinkApps;  for (uint32\_t i = 0; i < d.LeftCount(); ++i)  {  // 在左侧节点安装 PacketSink 应用(jieshou shujuduan)  sinkApps.Add(packetSinkHelper.Install(d.GetLeft(i)));  }   ApplicationContainer clientApps;  // 在每个右侧节点上安装多个 OnOff 应用（模拟客户端）  for (uint32\_t i = 0; i < d.RightCount(); ++i) {  // 每个客户端选择一组服务器来建立连接  std::set<uint32\_t> selectedServers;  while (selectedServers.size() < num\_flows) {  uint32\_t serverIndex = rand() % d.LeftCount(); // 随机选择一个左侧节点（服务器）  selectedServers.insert(serverIndex);  }   // 为每个选中的服务器创建一个OnOff应用  uint16\_t currentPort = port; // 开始的端口号  for (auto serverIndex : selectedServers) {  // 设置远程服务器的地址和端口  AddressValue remoteAddress(InetSocketAddress(d.GetLeftIpv4Address(serverIndex), currentPort++));  InetSocketAddress addr = InetSocketAddress::ConvertFrom(remoteAddress.Get());    clientHelper.SetAttribute("Remote", remoteAddress);  clientApps.Add(clientHelper.Install(d.GetRight(i)));  }  }   clientApps.Start(Seconds(1.0)); // 在 Sink 开始后 1 秒启动  clientApps.Stop(Seconds(duration)); // 在 Sink 停止前结束   // 生成全局路由表  Ipv4GlobalRoutingHelper::PopulateRoutingTables();   // 配置节点的移动性模型（固定位置）  MobilityHelper mobility;  Ptr<ListPositionAllocator> positionAlloc = CreateObject<ListPositionAllocator>();   // 设置左侧节点的位置  for (uint32\_t i = 0; i < d.LeftCount(); ++i)  {  positionAlloc->Add(Vector(10 \* i, 100, 0)); // 水平排列  }   // 设置右侧节点的位置  for (uint32\_t i = 0; i < d.RightCount(); ++i)  {  positionAlloc->Add(Vector(10 \* i, 0, 0)); // 水平排列  }   // 设置瓶颈节点（路由器）的位置  positionAlloc->Add(Vector(10 \* nLeaf / 2, 30, 0)); // 左侧路由器  positionAlloc->Add(Vector(10 \* nLeaf / 2 + 10, 60, 0)); // 右侧路由器   mobility.SetPositionAllocator(positionAlloc);  mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");   // 安装节点移动性模型  for (uint32\_t i = 0; i < d.LeftCount(); ++i)  {  mobility.Install(d.GetLeft(i));  }  for (uint32\_t i = 0; i < d.RightCount(); ++i)  {  mobility.Install(d.GetRight(i));  }  mobility.Install(d.GetLeft());  mobility.Install(d.GetRight());  // 生成动画文件  AnimationInterface anim("tcps.xml");    if(pcap){  // 为点对点设备启用 pcap 输出  bottleNeckLink.EnablePcapAll(pcapPath + "bottleneck", true);  pointToPointLeaf.EnablePcapAll(pcapPath + "leaf", true);  }  // --获取瓶颈链路的设备  NetDeviceContainer bottleneckDevices;  bottleneckDevices.Add(d.GetLeft()->GetDevice(0));  bottleneckDevices.Add(d.GetRight()->GetDevice(0));   // // --在瓶颈链路上监控丢包事件  // Ptr<NetDevice> device = bottleneckDevices.Get(0);  // Ptr<PointToPointNetDevice> ptpDevice = device->GetObject<PointToPointNetDevice>();  // Ptr<Channel> channel = ptpDevice->GetChannel();  // Ptr<PointToPointChannel> ptpChannel = channel->GetObject<PointToPointChannel>();  // ptpChannel->TraceConnectWithoutContext("Drop", MakeCallback(&YourDropTraceFunction));   // // --监控队列长度  // queueDiscs.Get(0)->TraceConnectWithoutContext("PacketsInQueue", MakeCallback(&QueueLengthTrace));   // --在瓶颈链路上监控丢包事件  // 使用 TraceConnect 方法并传递上下文信息  Ptr<NetDevice> device = bottleneckDevices.Get(0);  Ptr<PointToPointNetDevice> ptpDevice = device->GetObject<PointToPointNetDevice>();  Ptr<Channel> channel = ptpDevice->GetChannel();  Ptr<PointToPointChannel> ptpChannel = channel->GetObject<PointToPointChannel>();      queueDiscs.Get(0)->TraceConnect("Drop", "bottleneck-queue-drop", MakeCallback(&YourDropTraceFunction));   // --监控队列长度  // 使用 TraceConnect 方法并传递上下文信息  queueDiscs.Get(0)->TraceConnect("PacketsInQueue", "bottleneck-queue", MakeCallback(&QueueLengthTrace));   // Flow monitor  Ptr<FlowMonitor> flowMonitor;  FlowMonitorHelper flowHelper;  flowMonitor = flowHelper.InstallAll();    Simulator::Stop(Seconds(duration));   std::cout << "Running the simulation" << std::endl;  Simulator::Run();   // --在实验结束时计算吞吐量  uint64\_t totalBytesReceived = DynamicCast<PacketSink>(sinkApps.Get(0))->GetTotalRx();  double throughput = (totalBytesReceived \* 8.0) / (duration \* 1000000.0); // 吞吐量（Mbps）    QueueDisc::Stats st = queueDiscs.Get(0)->GetStats();   if (st.GetNDroppedPackets(RedQueueDisc::UNFORCED\_DROP) != 0)  {  std::cout << "There should be some unforced drops" << std::endl;  }   if (st.GetNDroppedPackets(QueueDisc::INTERNAL\_QUEUE\_DROP) != 0)  {  std::cout << "There should be zero drops due to queue full" << std::endl;  }   std::cout << "\*\*\* Stats from the bottleneck queue disc \*\*\*" << std::endl;  std::cout << st << std::endl;  std::cout << "Destroying the simulation" << std::endl;  if(flow\_monitor)  flowMonitor->SerializeToXmlFile("tcps.xml", true, true);   Simulator::Destroy();   return 0; } |