****

计算机网络小组虚拟实验

基于NS-3的TCP拥塞控制算法探究

课 程 老 师 ： 金伟祖

小组成员 一 ： 2052030 韩淑榕

小组成员 二 ： 2050255 何懿励

小组成员 三 ： 2051840 梁 厚

小组成员 四 ： 2050235 倪韵旖

软件学院 院（系）软件工程 专业 大三 年级

同济大学

Tongji University

目 录

[1 实验目的 1](#_Toc122279153)

[2 实验原理 1](#_Toc122279154)

[2.1 TCP拥塞 1](#_Toc122279155)

[2.2 拥塞控制算法 1](#_Toc122279156)

[2.3 TCP-Vegas 2](#_Toc122279157)

[2.4 TCP-NewVegas 3](#_Toc122279158)

[2.5 TCP-Westwood 4](#_Toc122279159)

[3 实验环境 4](#_Toc122279160)

[4 实验内容 4](#_Toc122279161)

[5 实验步骤 5](#_Toc122279162)

[5.1 配置实验环境 5](#_Toc122279163)

[5.2 实现TCP-NewVegas 算法 5](#_Toc122279164)

[5.3 搭建实验拓扑结构 7](#_Toc122279165)

[5.4 可视化处理 11](#_Toc122279166)

[5.4.1 Netanim 11](#_Toc122279167)

[5.4.2 Python 14](#_Toc122279168)

[6 实验结论 17](#_Toc122279169)

[6.1 单流情况 17](#_Toc122279170)

[6.1.1 拥塞窗口大小对比 17](#_Toc122279171)

[6.1.2 吞吐量大小对比 18](#_Toc122279172)

[6.1.3 丢包率对比 19](#_Toc122279173)

[6.1.4 结论 19](#_Toc122279174)

[6.2 多流情况 20](#_Toc122279175)

[6.1.1 拥塞窗口大小对比 20](#_Toc122279176)

[6.2.2 吞吐量大小对比 21](#_Toc122279177)

[6.2.3 丢包率对比 22](#_Toc122279178)

[6.2.4 结论 22](#_Toc122279179)

# 实验目的

使用离散事件驱动的网络仿真器NS-3，实现TCP拥塞控制算法TCP-NewVegas， 并与NS-3库中的TCP-Vegas、TCP-Westwood进行对比探究，从而：

1. 学习网络仿真器NS-3的使用。
2. 掌握TCP拥塞及拥塞控制算法原理
3. 了解常用的TCP拥塞控制算法
4. 加深对TCP拥塞控制算法的具体实现步骤的理解。
5. 比较TCP-NewVegas、TCP-Vegas、TCP-Westwood三种TCP拥塞控制算法在单流和多流启动的情况下各自的优缺点。

# 实验原理

## TCP拥塞

TCP/IP协议栈开始广泛运行后，网络开始遭受拥塞崩溃；即数据发送主机会以建议允许的速度将其数据包发送到互联网，当某些路由器发生拥塞，导致数据包被丢弃；对于TCP这种有重传机制的传输协议，当发生数据丢失时，重传数据将延长数据到达的时间；同时，高频率的重传，也将导致网络的拥塞得不到缓解，从而引发更多的拥塞。为了避免这类问题，1980年代后期，TCP拥塞控制被引入到网络协议中。

## 拥塞控制算法

TCP拥塞控制算法发展的过程当中出现了以下几种不一样的思路：

基于丢包的拥塞控制：将丢包视为出现拥塞，采取缓慢探测的方式，逐渐增大拥塞窗口，当出现丢包时，将拥塞窗口减少，如Tahoe、Reno、BIC-TCP、Cubic等；

基于时延的拥塞控制：将时延增长视为出现拥塞，延时增长时增大拥塞窗口，延时减少时减少拥塞窗口，如Vegas、Westwood等；

基于链路容量的拥塞控制：实时测量网络带宽和时延，认为网络上报文总量大于带宽时延乘积时出现了拥塞，如BBR；

基于学习的拥塞控制：没有特定的拥塞信号，而是借助评价函数，基于训练数据，使用机器学习的方法造成一个控制策略，如Remy。

## TCP-Vegas

TCP-Vegas通过观察以前的TCP连接中RTT值的改变情况来测量网络状况，并以此来控制

TCP-Vegas通过观察以前的TCP连接中RTT值的改变情况来测量网络状况，并以此来控制拥塞窗口cwnd变化，所以能够在较好地预测网络带宽的同时，减少丢包事件的发生。有关TCP-Vegas的拥塞控制算法如下：

计算期望的吞吐量:

Expected\_rate = cwnd(t)/Base\_rtt

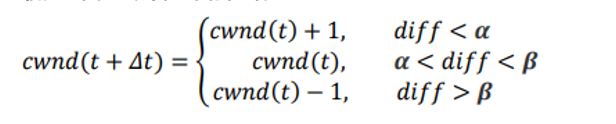
计算实际的吞吐量:

Actual\_rate = cwnd(t)/rtt

计算差值:

diff = (Expected\_rate - Actual\_rate) × Base\_rtt

拥塞窗口的控制策略:



图表 1 TCP-Vegas 拥塞控制算法

## TCP-NewVegas

TCP-NewVegas是TCP-Vegas的继任者，New-Vegas对Vegas在执行过程中存在提前进入拥塞避免、拥塞窗口未在短时间内增大到可用带宽致使网络带宽没用得到充分利用等问题进行了相应改善：

慢启动阶段：

在建立连接初始或是超时重传的第一个RTT内，由于对当前连接状况较为模糊，NewVegas不增加拥塞窗口。在第二个RTT之后，用介于线性式和指数级增长方式之间的某一个速率来增大cwnd。发送端在第n个RTT内减少或增加拥塞窗口，只有等到第n+2个RTT方能探测到其对网络的影响。



图表 2 TCP-NewVegas慢启动阶段算法

拥塞避免阶段：

cwnd与实际吞吐量成正比，diff与实际吞吐量成反比。期望吞吐量会随cwnd的减小而减小，产生的结果便是diff值不在α与β之间。

在任意RTT内，当前网络吞吐量发生轻微变化时，改进算法会增大α、β和cwnd值，cwnd的增加带动吞吐量上升，循环反复，这使得α、β和cwnd值不断增大。但是当α>diff或者β<diff时，拥塞窗口独自增加，进而使可用带宽得到了充分利用。

## TCP-Westwood

TCP-Westwood改良自New Reno，不同于基于丢包的拥塞控制， TCP-Westwood发送端监控ACK报文的接收速率，进而估算当前连接可达到的数据发送速率（可用带宽）。当发送端检测到丢包时（超时或者3个重复ACK），发送端根据估算的发送速率设置拥塞窗口大小（ CWND ）和慢启动阈值（ SSTHRESH ）。其通过对确认包测量来确定一个“合适的发送速度”，并以此调整拥塞窗口和慢启动阈值。

# 实验环境

本实验涉及的软件环境如下：

• VMWare Workstation Player

• Ubuntu 16.04

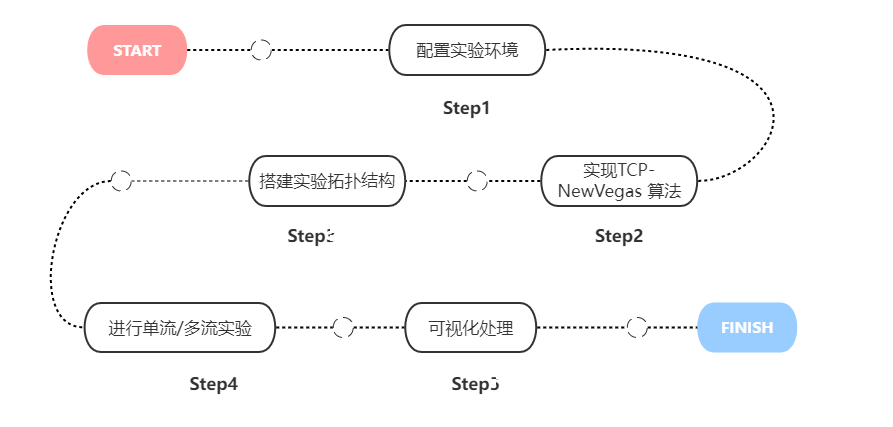
• NS3 version 3.30

• NetAnim

• Gnuplot

# 实验内容

本次实验用C++实现TCP-NewVegas 算法，并将其加入NS-3中TCP拥塞算法库中。在比较TCP拥塞控制算法时，由于进行的均为单条流或混合流仿真实验，为保证实验链路收发两端的一致性，采用了左右对称的单瓶颈链路哑铃拓扑结构，节点两两配对，共三对，绑定不同的TCP拥塞算法。实验主要分为两部分，单流情况下，即一次运行一对节点的数据收发，和多流情况下，即一次运行三对节点的数据收发。最后进行可视化处理，使用NetAnim进行动画模拟和导出原始实验数据，使用Python绘图库进行绘制。



图表 3 实验内容

# 实验步骤

## 配置实验环境

* 安装NS-3依赖库，安装NS-3 3.30版本，手动安装NetAnim软件和gnuplot软件
* 编译NS-3等软件，设置编译参数，避免net模块异常报错

./build.py --disable-python

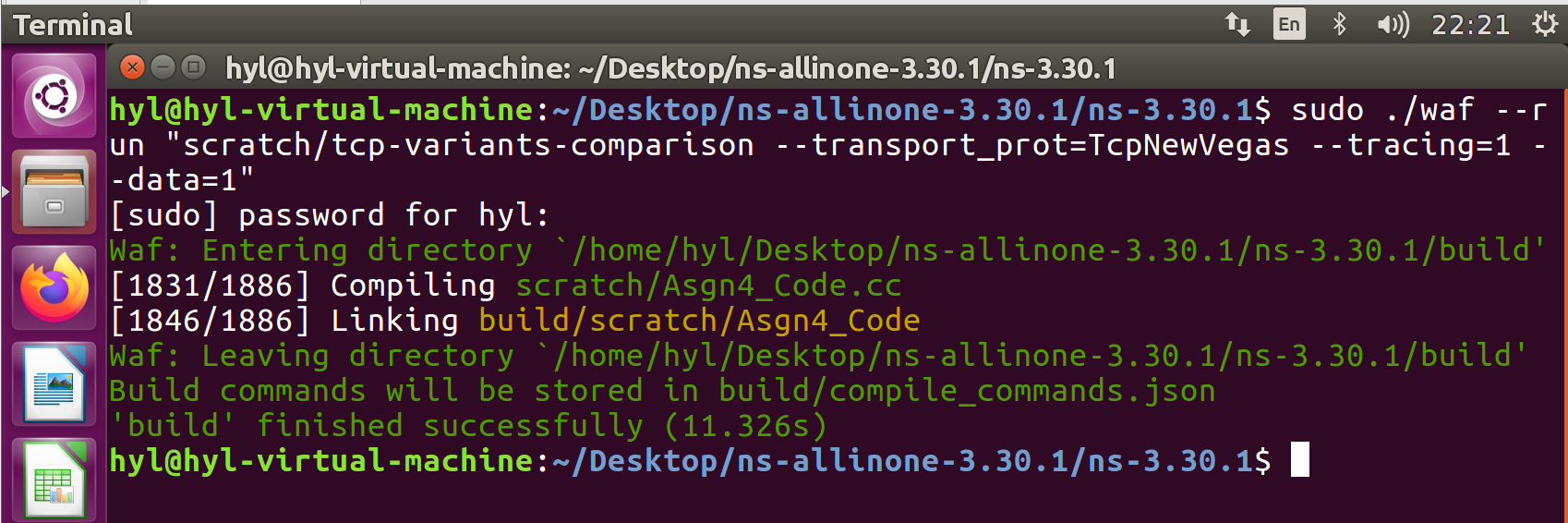
* 成功运行example目录下的tcp-variants-comparison.cc（NS-3官方的tcp拥塞控制算法对比程序）
* 增加cmd参数，设置data为1，tcp算法为vegas，tracing为1，使example程序得到vegas拥塞控制算法的point-to-point网络的tracing数据。
* 使用NetAnim软件绘制tracing数据的动画
* 使用gnuplot软件绘制该次模拟的cwnd拥塞窗口随时间变化趋势

## 实现TCP-NewVegas 算法

* 在ns3/internet/model目录下完成tcp-nv.cc和tcp-nv.h文件的开发

1. class TcpNewVegas : public TcpNewReno
2. {
3. public:
4. static TypeId GetTypeId (void);
5. TcpNewVegas (void);
6. *// Init (tcpnv\_init)*
7. TcpNewVegas (const TcpNewVegas& sock);
9. virtual ~TcpNewVegas (void);
10. virtual std::string GetName () const;
11. virtual void TcpNewVegasReset(Ptr<TcpSocketState> tcb);
12. *// Hook for packet ack accounting (tcpnv\_acked)*
13. virtual void PktsAcked (Ptr<TcpSocketState> tcb, uint32\_t segmentsAcked, const Time& rtt);
14. *// Call before changing ca\_state (tcpnv\_state)*
15. virtual void CongestionStateSet (Ptr<TcpSocketState> tcb, const TcpSocketState::TcpCongState\_t newState);
16. *// New cwnd calculation (tcpnv\_cong\_avoid)*
17. virtual void IncreaseWindow (Ptr<TcpSocketState> tcb, uint32\_t segmentsAcked);
18. *// Returns the slow start threshold (tcpnv\_recalc\_ssthresh)*
19. virtual uint32\_t GetSsThresh (Ptr<const TcpSocketState> tcb, uint32\_t bytesInFlight);
20. virtual Ptr<TcpCongestionOps> Fork ();

* 修改ns3/internet的wscript文件，增加两个tcp-nv文件的条目
* 使用tcp-variants-comparison.cc文件测试nv，设置cmd参数，调用tcp算法为NewVegas



图表 4 设置cmd参数

## 搭建实验拓扑结构

本实验采用左右对称的单瓶颈链路哑铃拓扑结构，其中，S1、S2、S3为源端，R1、R2为主干道上的路由器，D1、D2、D3为接收端。

在此拓扑中，建立3条链路，由S（1，2，3）通过主干道路由器转发至D（4，5，6），各结点的传输速率均为100Mb/s，传输延迟为20ms，路由器的传输速率是10Mb/s，传输延迟是50ms。

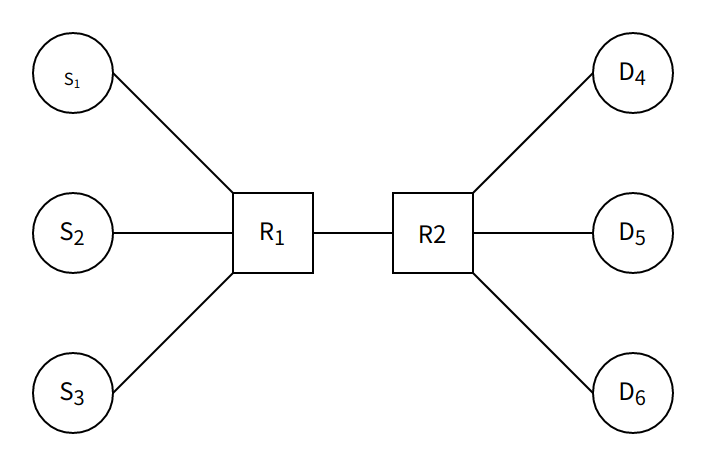
• 链路BDP[S-R/D-R]：100Mbps\*20ms=2000000bits

• 链路BDP[R1-R2]：10Mbps\*50ms=500000bits

• 发送方（即 S1、S2 和 S3）分别与 TCP-NewVegas、TCP-Vegas 和TCP-Westwood连接。

• 数据包大小为 1.5KB。

• 数据包数量=BDP/数据包大小。



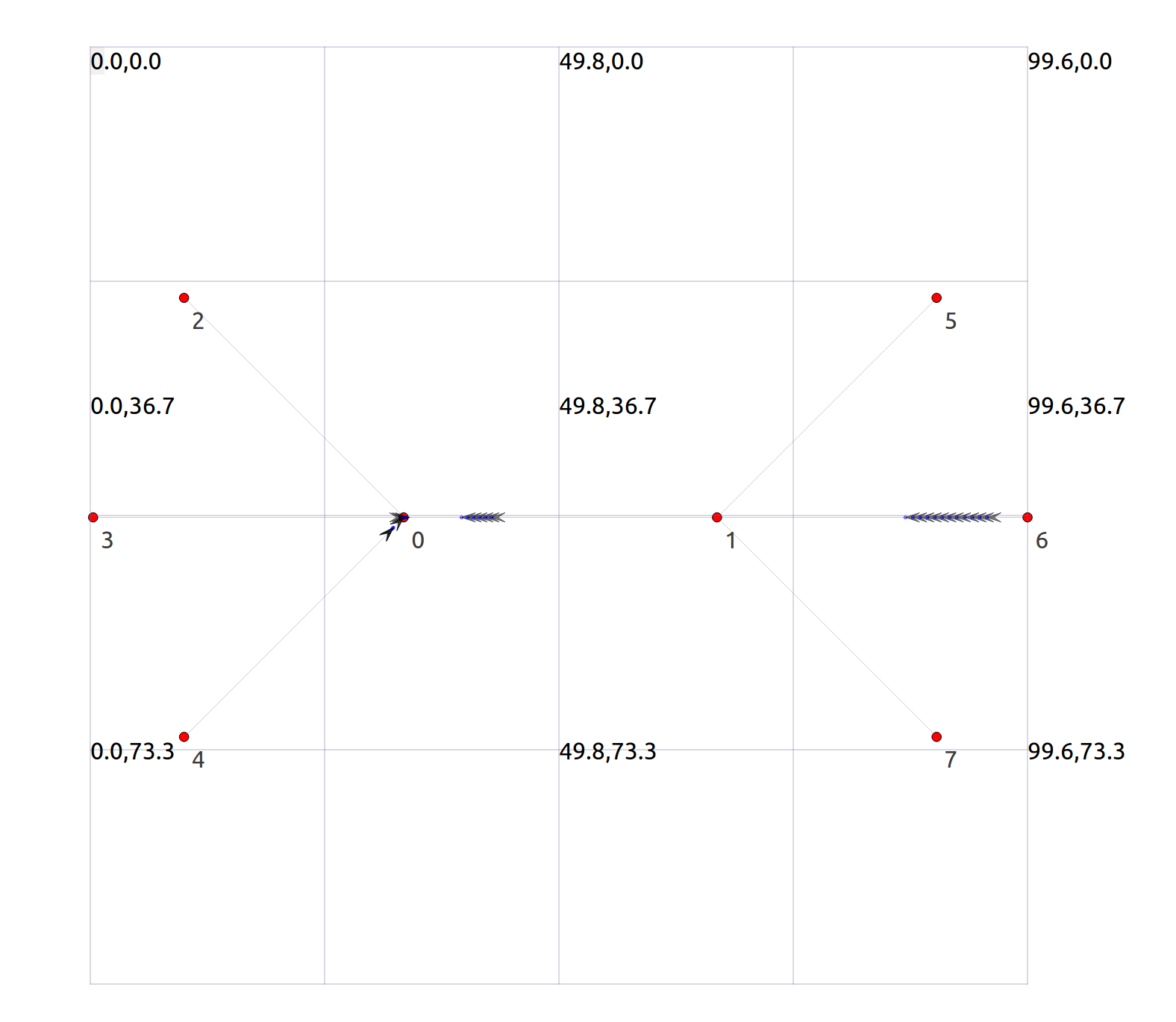
图表 5 实验理论拓扑结构

搭建流程如下：

* 使用PointToPointHelper帮助构建点到点的连接。
* //Configuring the network
* PointToPointHelper p2pHR = help\_configP2P(HR\_Bandwidth, HR\_Delay, std::to\_string(HR\_QueueSize)+"p" );
* PointToPointHelper p2pRR = help\_configP2P(RR\_Bandwidth, RR\_Delay, std::to\_string(RR\_QueueSize)+"p" );
* Ptr<RateErrorModel> Error\_Model = CreateObjectWithAttributes<RateErrorModel> ("ErrorRate", DoubleValue (error));
* 使用NodeContainer创建节点对象。
* NodeContainer Routers, Senders, Receivers;
* //Create n nodes and append pointers to them to the end of this NodeContainer.
* Routers.Create(2);
* Senders.Create(num\_sender);
* Receivers.Create(num\_sender);
* 使用 NetDeviceContainer 存放被创建的 NetDevice 对象列表，设置error rate。
* NetDeviceContainer routerDevices = p2pRR.Install(Routers);
* NetDeviceContainer leftRouterDevices, rightRouterDevices, senderDevices, ReceiverDevices;
* //Adding links
* std::cout << "Adding and configuring links between nodes..."<< std::endl;
* for(int i=0;i<num\_sender;i++)
* {
* //Configuring all the HiR1 links i=1,2,3 (senders to R1 links)
* NetDeviceContainer Config\_Left = p2pHR.Install(Routers.Get(0), Senders.Get(i));
* leftRouterDevices.Add(Config\_Left.Get(0)); // left router
* senderDevices.Add(Config\_Left.Get(1)); // all devices on left of dumbbell
* Config\_Left.Get(0)->SetAttribute("ReceiveErrorModel", PointerValue(Error\_Model));
* }
* for(int i=0;i<num\_sender;i++)
* {
* //Configuring all the HiR2 links i=4,5,6 (receivers to R2 links)
* NetDeviceContainer Config\_Right = p2pHR.Install(Routers.Get(1), Receivers.Get(i));
* rightRouterDevices.Add(Config\_Right.Get(0)); // right router
* ReceiverDevices.Add(Config\_Right.Get(1)); // all devices on right of dumbbell
* Config\_Right.Get(0)->SetAttribute("ReceiveErrorModel", PointerValue(Error\_Model));
* }
* 使用 InternetStackHelper为每一个节点容器中的节点安装一个网络协议栈。
* //Installing Internet Stack
* std::cout << "Installing internet stack on the nodes..."<< std::endl;
* InternetStackHelper stack;
* stack.Install(Routers);
* stack.Install(Senders);
* stack.Install(Receivers);
* 使用 Ipv4AddressHelper 类和Ipv4InterfaceContainer类为节点上的设备设置 IP 地址。
* std::cout << "Assigning IP addresses to the nodes and initialising network interfaces..."<< std::endl;
* Ipv4AddressHelper routerIP = Ipv4AddressHelper("15.3.0.0", "255.255.255.0");    //(network IP, mask)
* Ipv4AddressHelper senderIP = Ipv4AddressHelper("15.1.0.0", "255.255.255.0");
* Ipv4AddressHelper ReceiverIP = Ipv4AddressHelper("15.2.0.0", "255.255.255.0");
* Ipv4InterfaceContainer router\_Interface, sender\_Interfaces, Receiver\_Interfaces, leftRouter\_Interfaces, rightRouter\_Interfaces;
* router\_Interface = routerIP.Assign(routerDevices);
* for(int i=0;i<num\_sender;i++)
* {
* NetDeviceContainer senderDevice;
* senderDevice.Add(senderDevices.Get(i));
* senderDevice.Add(leftRouterDevices.Get(i));
* Ipv4InterfaceContainer sender\_Interface = senderIP.Assign(senderDevice);
* sender\_Interfaces.Add(sender\_Interface.Get(0));
* leftRouter\_Interfaces.Add(sender\_Interface.Get(1));
* senderIP.NewNetwork();
* NetDeviceContainer ReceiverDevice;
* ReceiverDevice.Add(ReceiverDevices.Get(i));
* ReceiverDevice.Add(rightRouterDevices.Get(i));
* Ipv4InterfaceContainer Receiver\_Interface = ReceiverIP.Assign(ReceiverDevice);
* Receiver\_Interfaces.Add(Receiver\_Interface.Get(0));
* rightRouter\_Interfaces.Add(Receiver\_Interface.Get(1));
* ReceiverIP.NewNetwork();
* }
* 设置端口等TCP sockets参数，以多流情况为例。
* for(int i=0;i<num\_sender;i++){
* std::cout<<"From H"<<i<<" to H"<<i+4<<" : ";
* std::cout<<"Connection type: "<<types[i]<<std::endl;
* Create\_files(i,tput,cwnd,closs,gput,2); //create files to store data
* // assigning attributes to ns3 TCP sockets
* if(i==0) //First flow starts at FirstFlowStart=0
* ns3TCPSocket[i] = Create\_Socket(InetSocketAddress(Receiver\_Interfaces.GetAddress(i), port), port,types[i], Senders.Get(i), Receivers.Get(i), FirstFlowStart, FirstFlowStart+Duration\_Gap, Packet\_Size, Num\_Packets, Transfer\_Speed , FirstFlowStart, FirstFlowStart+Duration\_Gap);
* else     //Other flows start at FirstFlowStart+OtherFlowStart = 20, while first one in progress
* ns3TCPSocket[i] = Create\_Socket(InetSocketAddress(Receiver\_Interfaces.GetAddress(i), port), port,types[i], Senders.Get(i), Receivers.Get(i), OtherFlowStart, OtherFlowStart+Duration\_Gap, Packet\_Size, Num\_Packets, Transfer\_Speed , OtherFlowStart, OtherFlowStart+Duration\_Gap);
* ns3TCPSocket[i]->TraceConnectWithoutContext("CongestionWindow", MakeBoundCallback (&Store\_cwnd\_size, cwnd[i], 0));
* ns3TCPSocket[i]->TraceConnectWithoutContext("Drop", MakeBoundCallback (&packet\_dropped, closs[i], 0, i+1));
* // Measuring packet sinks
* Config::Connect(sink1[i], MakeBoundCallback(&Calc\_gput, gput[i], 0));
* Config::Connect(sink2[i], MakeBoundCallback(&Calc\_tput, tput[i], 0));
* }

## 可视化处理

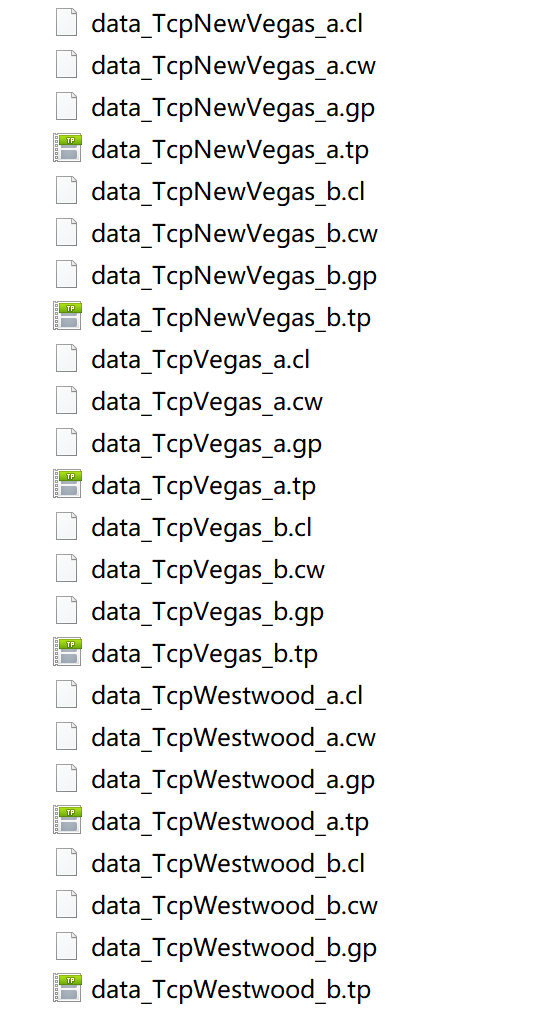
### 5.4.1 Netanim

* 为使用NetAnim记录数据收发过程，在Header.h中加入相关头文件，#include "ns3/flow-monitor-module.h"、#include "ns3/ipv4-global-routing-helper.h"。
* 实验中声明动画xml文件的存储名称，std::string animFile = "tcppart13-animation.xml"; std::string animFile = "tcppart23-animation.xml";
* 参考ns3源码point-to-point-dumbbell，设置节点位置，使其更易于观察。
* // set anim nodes position
* double xAdder = 99.0 / 3.0;
* double thetaL = M\_PI / (num\_sender + 1.0);
* double thetaR = M\_PI / (num\_sender + 1.0);
* // Place the left router
* Ptr<Node> lr = Routers.Get(0);
* Ptr<ConstantPositionMobilityModel> loc = lr->GetObject<ConstantPositionMobilityModel>();
* if (!loc)
* {
* loc = CreateObject<ConstantPositionMobilityModel>();
* lr->AggregateObject(loc);
* }
* Vector lrl(Vector(33.3, 50.0, 0));
* loc->SetPosition(lrl);
* // Place the right router
* Ptr<Node> rr = Routers.Get(1);
* loc = rr->GetObject<ConstantPositionMobilityModel>();
* if (!loc)
* {
* loc = CreateObject<ConstantPositionMobilityModel>();
* rr->AggregateObject(loc);
* }
* Vector rrl(Vector(66.6, 50.0, 0));
* loc->SetPosition(rrl);
* double theta = -M\_PI\_2 + thetaL;
* for (uint32\_t l = 0; l < num\_sender; ++l)
* {
* Ptr<Node> ln = Senders.Get(l);
* loc = ln->GetObject<ConstantPositionMobilityModel>();
* if (l % 2 == 1)
* {
* if (l == num\_sender / 2)
* {
* theta = 0.0;
* }
* }
* if (!loc)
* {
* loc = CreateObject<ConstantPositionMobilityModel>();
* ln->AggregateObject(loc);
* }
* Vector lnl(lrl.x - std::cos(theta) \* xAdder,
* lrl.y + std::sin(theta) \* xAdder,
* 0);
* loc->SetPosition(lnl);
* theta += thetaL;
* }
* theta = -M\_PI\_2 + thetaR;
* for (uint32\_t r = 0; r < num\_sender; ++r)
* {
* if (r % 2 == 1)
* {
* if (r == num\_sender / 2)
* {
* theta = 0.0;
* }
* }
* Ptr<Node> rn = Receivers.Get(r);
* loc = rn->GetObject<ConstantPositionMobilityModel>();
* if (!loc)
* {
* loc = CreateObject<ConstantPositionMobilityModel>();
* rn->AggregateObject(loc);
* }
* Vector rnl(rrl.x + std::cos(theta) \* xAdder, // Right node location
* rrl.y + std::sin(theta) \* xAdder,
* 0);
* loc->SetPosition(rnl);
* theta += thetaR;
* }
* 使用Ipv4GlobalRoutingHelper::PopulateRoutingTables()帮助节点模拟。
* 使用 AnimationInterface anim(animFile); 将动画数据存入xml文件中
* 

图表 6 NetAnim中拓扑结构展示

### 5.4.2 Python

* 将虚拟机中的数据文件（单流、多流拥塞窗口、吞吐量）拷贝到本机



图表 7 原始数据文件

* 使用Python的绘图库，将三种拥塞控制算法的曲线合成到一个图像中，绘制单流拥塞窗口随时间变化曲线、多流拥塞窗口随时间变化曲线、单流吞吐量随时间变化曲线、多流吞吐量随时间变化曲线图

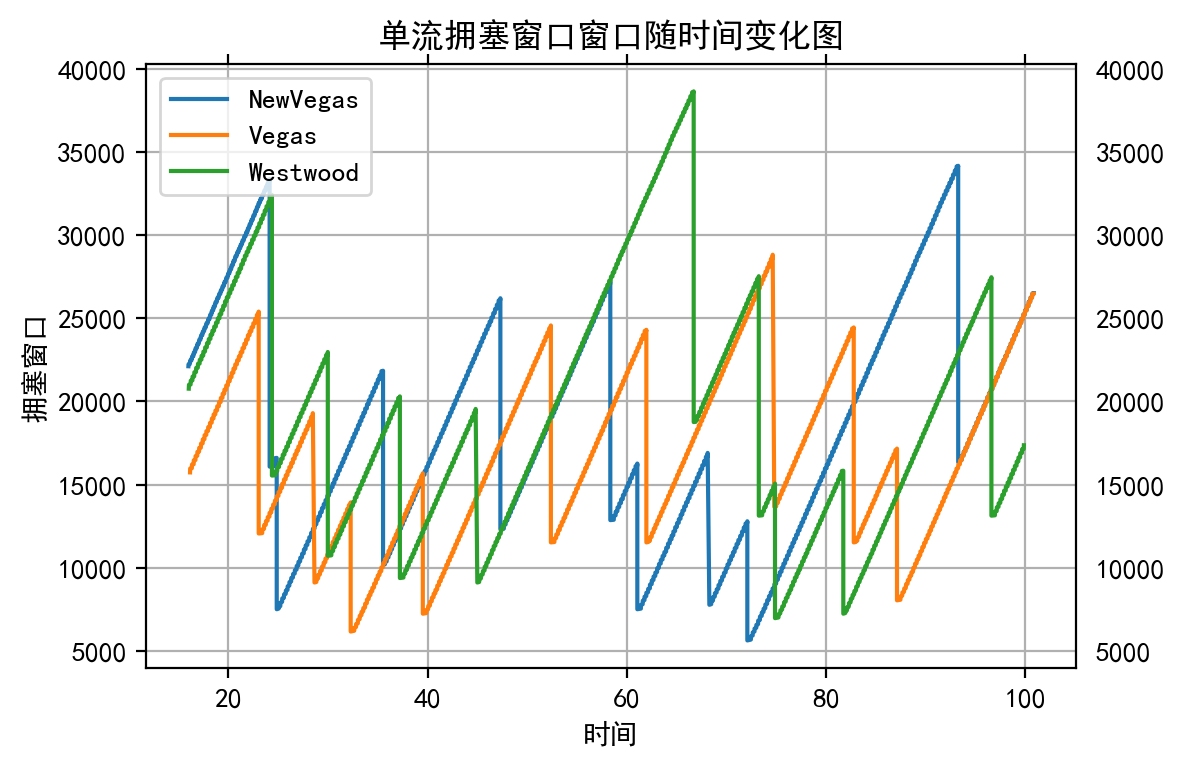
1. if \_\_name\_\_=="\_\_main\_\_":
2. method1="data\_TcpNewVegas\_"
3. method2="data\_TcpVegas\_"
4. method3="data\_TcpWestwood\_"
5. typename="b"
6. houzhui=".tp"
8. data1=np.loadtxt('graph/'+method1+typename+houzhui)
9. *#print(data1)*
10. data2=np.loadtxt('graph/'+method2+typename+houzhui)
11. data3=np.loadtxt('graph/'+method3+typename+houzhui)
12. x1=data1[:,0]
13. c=x1>16
14. x1=x1[c]
15. y1=data1[:,1]
16. y1=y1[c]
18. x2=data2[:,0]
19. c=x2>16
20. x2=x2[c]
21. y2=data2[:,1]
22. y2=y2[c]
24. x3=data3[:,0]
25. c=x3>16
26. x3=x3[c]
27. y3=data3[:,1]
28. y3=y3[c]
30. plt.rcParams['font.sans-serif']=['SimHei']
31. plt.plot(x1,y1,label="NewVegas")
32. plt.plot(x2,y2,label="Vegas")
33. plt.plot(x3,y3,label="Westwood")
34. plt.legend()
35. *# 右侧刻度*
36. ax = plt.gca()
37. ax.set\_xlabel('时间')   # 设置 x 轴名字
38. ax.set\_ylabel('吞吐率')
39. ax.tick\_params(top=True,labeltop=False,labelright = True)
40. ax.grid()
41. plt.title('多流吞吐率窗口随时间变化图')
42. plt.show()

# 实验结论

## 单流情况

### 拥塞窗口大小对比

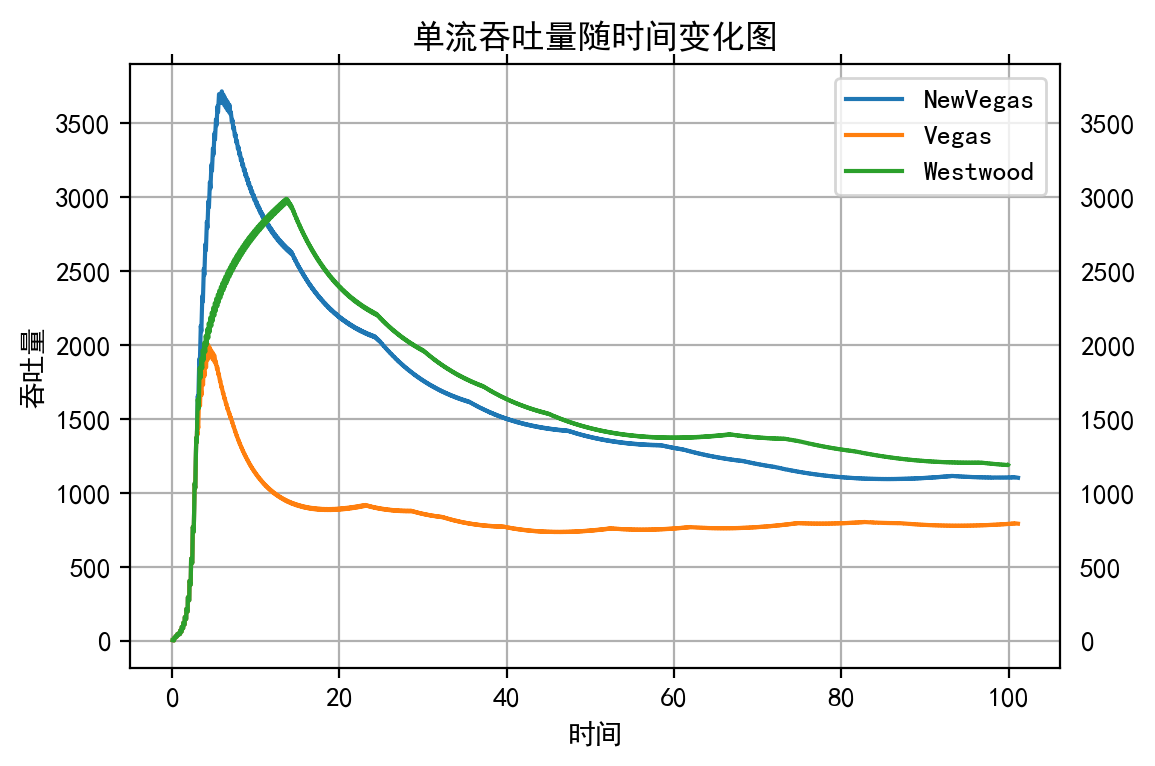
剔除不稳定数据后，单流拥塞窗口时间变化图如下图所示：



图表 8 单流情况拥塞窗口大小对比

1. 从图中可以看到在20到100时段内，Westwood和Vegas算法的拥塞窗口大小在大多数情况下相对于Vegas较大，体现了两种算法较Vegas算法的优越性。
2. Vegas 在大约time=20时达到峰值，较Westwood与Vegas算法都更提前，由此可以看出NewVegas 的拥塞窗口大小在初始时上升的较快，即在慢启动阶段表现出了算法的优势。
3. 改进后的NewVegas算法在time=90左右结束了数据包的发送，是最早结束发送数据包的算法，体现了算法的优越性。

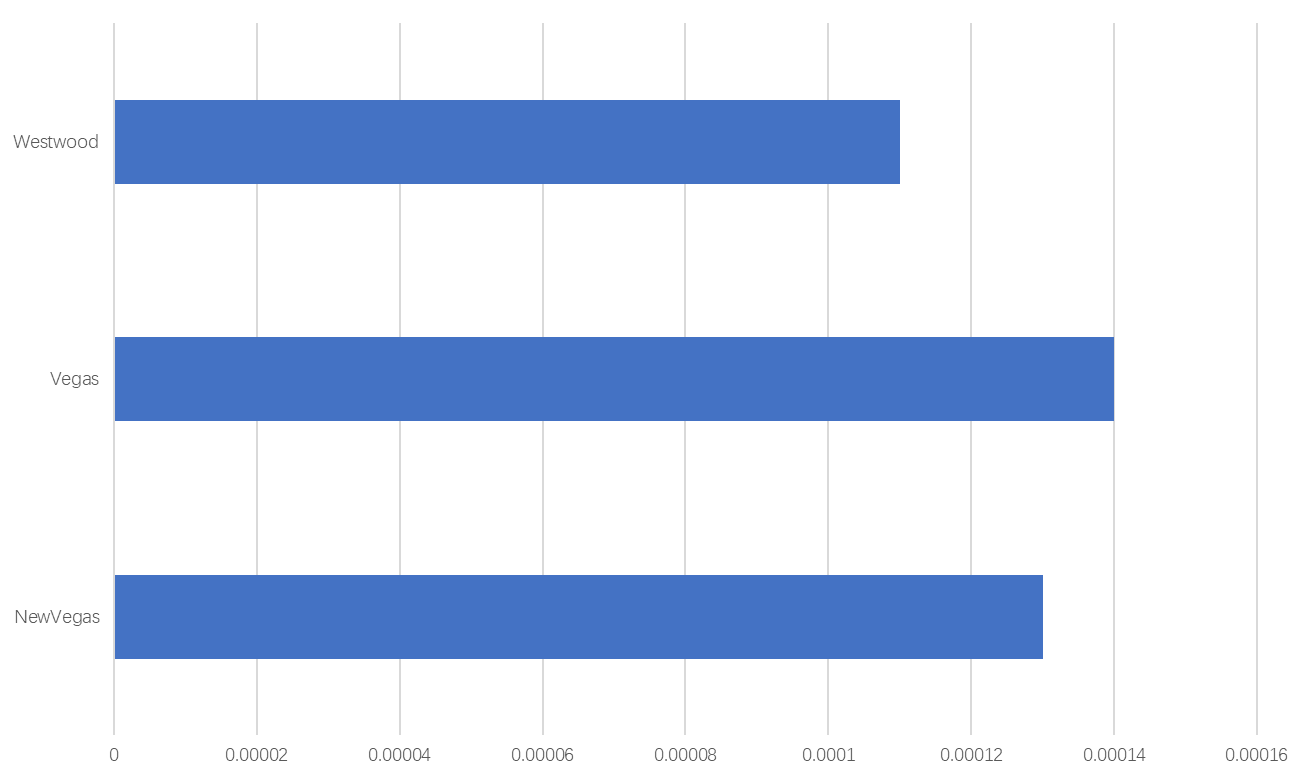
### 6.1.2 吞吐量大小对比



图表 9 单流情况吞吐量对比

上图对比了Vegas 与NewVegas 以及Westwood三个算法的吞吐量，从图中可以看出，NewVegas在慢启动阶段拥塞窗口增长算法使得其相较于Vegas与Westwood保持了拥塞窗口处于更大的值，由此产生了上升快且峰值更好的性能，体现了其算法具有优越性。在吞吐量下降阶段，NewVegas与Westwood相近，Westwood略高于NewVegas，而两者都远大于Vegas，故综合比较而言，在单流启动的情况下，NewVegas的性能高于Westwood高于Vegas。

### 6.1.3 丢包率对比



图表 10 单流情况丢包率对比

根据图表显示的丢包率的数据来看，三个算法的丢包率相差在0.00001之间，所以可以认为3中算法在丢包率的表现上相差不大。但是，就数值而言，NewVegas还是略胜于另外两种算法。

### 6.1.4 结论

在单流实验中，我们根据3种算法（改进算法NewVegas，Vegas和Westwood）在拥塞窗口大小，吞吐量和丢包率三个指标上的表现初步评判了算法的性能。我们得出初步的结论是：

a) NewVegas在慢启动阶段执行的时间较长，拥塞窗口一直维持在一个较大的值，有效解决了Vegas 算法在慢启动阶段结束过早，发送速率过慢导致的花费较长时间达到可用带宽的问题，对比改进前的Vegas算法有更加优良的性能；

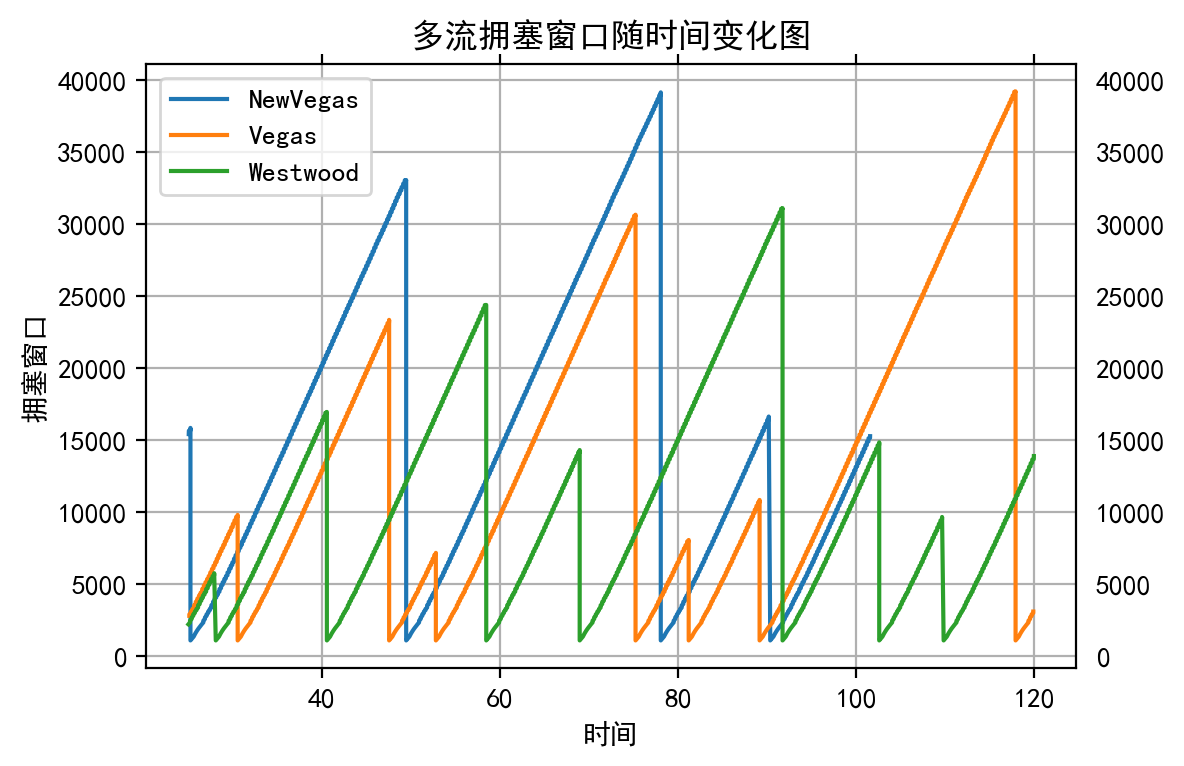
b) NewVegas的拥塞窗口大小下降速度与Westwood接近，在此段时间内吞吐量没有明显的优势，但NewVegas与Westwood的整体吞吐量都高于Vegas，体现了二者具有更优良的性能。

b) NewVegas、Vegas、Westwood的丢包率相近，Vegas的丢包率稍高于其他二者，进一步验证了NewVegas与Westwood算法的优越性。

## 多流情况

### 6.1.1 拥塞窗口大小对比

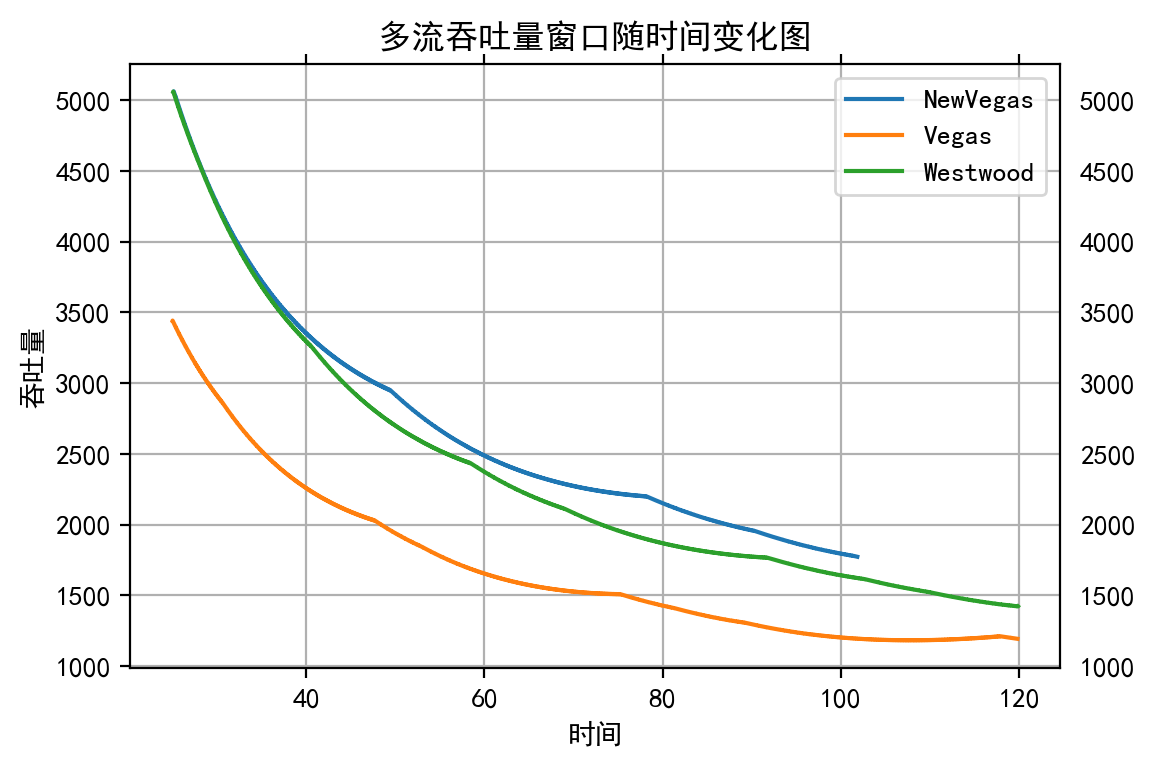
剔除不稳定数据后，多流拥塞窗口时间变化图如下图所示：



图表 11 多流情况拥塞窗口大小对比

1. 从图中可以看到在0到100时段内，NewVegas算法的拥塞窗口大小在大多数情况下相对于Vegas和Westwood较大；
2. Vegas 在大约time=40时达到峰值，且Vegas的拥塞窗口的峰值在15000左右；而改进算法NewVegas 在50s 左右才达到峰值，峰值在32500左右，远大于Vegas算法。此次仿真实验说明了改进算法有效解决了Vegas 算法在慢启动阶段结束过早，发送速率过慢导致的花费较长时间达到可用带宽的问题。从而提升了网络的带宽利用率，由此可以看出NewVegas 的性能高于Vegas；
3. 此外改进后的NewVegas算法在time=100左右结束了数据包的发送，是最早结束发送数据包的算法，说明在我们的实验环境下多流竞争的情况中，改进算法Newvegas表现良好。

### 6.2.2 吞吐量大小对比



图表 12 多流情况吞吐量对比

上图对比了Vegas 与NewVegas 以及Westwood三个算法的吞吐量，从图中可以看出，3个算法达到可用宽带的时间基本一致，但是NewVegas达到的吞吐量的峰值大于Vegas，并且在下降阶段表现优于Westwood。推测NewVegas在慢启动阶段执行的时间较长，拥塞窗口一直维持在一个较大的值，所以改进算法NewVegas 的吞吐量远大于Vegas 的吞吐量。并且在3个算法中，NewVegas最早完成数据包的传输，突出了改进算法的优越性。

### 6.2.3 丢包率对比

图表 13 多流情况丢包率对比

根据图表显示的丢包率的数据来看，三个算法的丢包率相差在0.00001之间，所以可以认为3中算法在丢包率的表现上相差不大。但是，就数值而言，NewVegas还是略胜于另外两种算法。

### 6.2.4 结论

在多流实验中，我们根据3种算法（改进算法NewVegas，Vegas和Westwood）在拥塞窗口大小，吞吐量和丢包率三个指标上的表现初步评判了算法的性能。我们得出初步的结论是：

* 1. NewVegas在慢启动阶段执行的时间较长，拥塞窗口一直维持在一个较大的值，有效解决了Vegas 算法在慢启动阶段结束过早，发送速率过慢导致的花费较长时间达到可用带宽的问题，对比改进前的Vegas算法有更加优良的性能；
  2. 在与其他两个算法竞争宽带资源的情况下，NewVegas表现良好。

图表索引：

[图表 1 TCP-Vegas 拥塞控制算法 2](#_Toc122279180)

[图表 2 TCP-NewVegas慢启动阶段算法 3](#_Toc122279181)

[图表 3 实验内容 5](#_Toc122279182)

[图表 4 设置cmd参数 7](#_Toc122279183)

[图表 5 实验理论拓扑结构 7](#_Toc122279184)

[图表 6 NetAnim中拓扑结构展示 14](#_Toc122279185)

[图表 7 原始数据文件 15](#_Toc122279186)

[图表 8 单流情况拥塞窗口大小对比 17](#_Toc122279187)

[图表 9 单流情况吞吐量对比 18](#_Toc122279188)

[图表 10 单流情况丢包率对比 19](#_Toc122279189)

[图表 11 多流情况拥塞窗口大小对比 20](#_Toc122279190)

[图表 12 多流情况吞吐量对比 21](#_Toc122279191)

[图表 13 多流情况丢包率对比 22](#_Toc122279192)