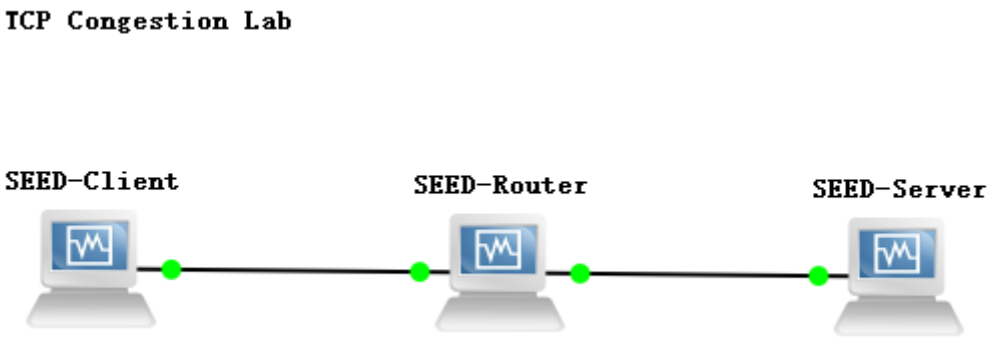
实验3 TCP Congestion实验

1. GNS3下的TCP Congestion测试

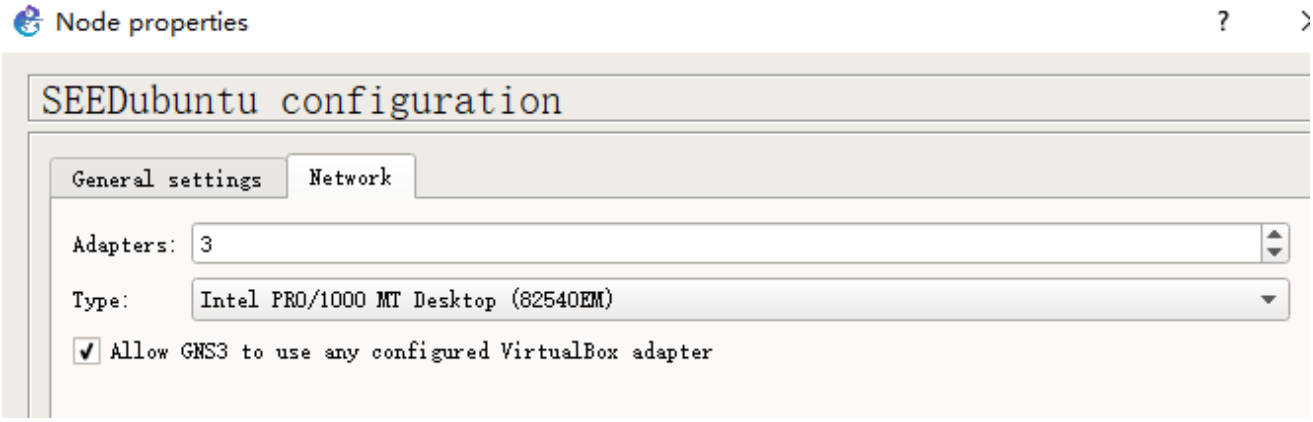
在GNS3中创建一个新项目，用三台SEED虚拟机按照如图方式连接起来(参考实验1，三台虚拟机分别为SEED-Client，SEED-Router，SEED-Server)，配置IP地址，让他们可以互相通信

如果物理机器性能一般，请用slitaz代替SEED，不过必须自己安装一些软件，安装软件参考： slitaz文档：http://doc.slitaz.org



注意的地方：

在GNS3中先给每台虚拟机配置三个网卡：



eth0网卡不要在GNS3中使用，而是用VirtualBox如下配置：

可以发现虚拟机可以同时访问外部网络

在SEED-Client和SEED-Server上安装iperf软件

```
sudo apt-get install iperf
```

2. Linux下的TCP拥塞控制算法

- 列出所有直接编译到内核的拥塞控制算法：

```
sysctl net.ipv4.tcp_available_congestion_control
cubic reno
```

- 修改当前的拥塞控制算法：

```
sysctl -w net.ipv4.tcp_congestion_control=cubic
```

- 增加新的拥塞控制算法：
  linux支持的拥塞控制算法：

```
18:05] seed@ubuntu:~$ ls /lib/modules/3.5.0-37-generic/kernel/net/ipv4/
  ip_gre.ko     tcp_highspeed.ko  tcp_probe.ko      tcp_yeah.ko
  ipip.ko       tcp_htcp.ko       tcp_scalable.ko   tunnel4.ko
  netfilter     tcp_hybla.ko      tcp_vegas.ko      udp_diag.ko
  tcp_bic.ko    tcp_illinois.ko   tcp_veno.ko       xfrm4_mode_beet.ko
  tcp_diag.ko   tcp_lp.ko         tcp_westwood.ko   xfrm4_mode_transport.ko
```

怎么加载这些内核模块，来运行新的拥塞控制算法：

```
#echo "westwood" > /proc/sys/net/ipv4/tcp_congestion_control
#cat /proc/sys/net/ipv4/tcp_congestion_control
westwood
```

或者：

```
#echo "westwood" > /proc/sys/net/ipv4/tcp_congestion_control
#lsmod | grep westwood
tcp_westwood 12675 0
```

各种拥塞控制算法简介：

```
* High Speed TCP
The algorithm is described in RFC 3649. The main use is for
connections with large bandwidth and large RTT (such as
Gbit/s and 100 ms RTT).
* H-TCP
H-TCP was proposed by the Hamilton Institute for
transmissions that recover more quickly after a congestion
event. It is also designed for links with high bandwidth and
RTT.
* Scalable TCP
This is another algorithm for WAN links with high bandwidth
and RTT. One of its design goals is a quick recovery of the
window size after a congestion event. It achieves this goal
by resetting the window to a higher value than standard TCP.
* TCP BIC
BIC is the abbreviation for Binary Increase Congestion
control. BIC uses a unique window growth function. In case
of packet loss, the window is reduced by a multiplicative
factor. The window size just before and after the reduction
is then used as parameters for a binary search for the new
window size. BIC was used as standard algorithm in the Linux
kernel.
* TCP CUBIC
```

CUBIC is a less aggressive variant of BIC (meaning, it doesn't steal as much throughput from competing TCP flows as does BIC).

* TCP Hybla

TCP Hybla was proposed in order to transmit data efficiently over satellite links and "defend" the transmission against TCP flows from other origins.

* TCP Low Priority

This is an approach to develop an algorithm that uses excess bandwidth for TCP flows. It can be used for low priority data transfers without "disturbing" other TCP transmissions (which probably don't use TCP Low Priority).

* TCP Tahoe/Reno

These are the classical models used for congestion control. They exhibit the typical slow start of transmissions. The throughput increases gradually until it stays stable. It is decreased as soon as the transfer encounters congestion, then the rate rises again slowly. The window is increased by adding fixed values. TCP Reno uses a multiplicative decrease algorithm for the reduction of window size. TCP Reno is the most widely deployed algorithm.

* TCP Vegas

TCP Vegas introduces the measurement of RTT for evaluating the link quality. It uses additive increases and additive decreases for the congestion window.

* TCP Veno

This variant is optimised for wireless networks, since it was designed to handle random packet loss better. It tries to keep track of the transfer, and guesses if the quality decreases due to congestion or random packet errors.

* TCP Westwood+

Westwood+ addresses both large bandwidth/RTT values and random packet loss together with dynamically changing network loads. It analyses the state of the transfer by looking at the acknowledgement packets. Westwood+ is a modification of the TCP Reno algorithm.

3.从SEED-Client虚拟机发包测试

下面以SEED-Client和SEED-Server都设置拥塞算法为cubic

先在SEED-Server端运行下面的命令，让iperf的服务器端运行守护在5001端口：

```
# iperf -s
```

在SEED-Client端运行下面的命令，让iperf的客户端运行，同时利用Linux内核的tcp probe模块监控特定连接中参数变化：

```
# modprobe tcp_probe port=5001 //对端口为5001（源或目的）的tcp连接进行监控
# cat /proc/net/tcpprobe > data.txt &    // tcpprobe捕捉的信息是持续性的，可以放到后台读
# pid=$!   // 保存上一个读命令的pid，用于结束读tcpprobe接口
# iperf -c otherhost    // 使用iperf建立一个TCP流
# kill $pid
```

在文件data.txt中保存了tcp的拥塞窗口变化情况，可以通过虚拟机共享目录或者网络导出，作为实验数据，iperf运行结果：

```
[12/20/2017 00:09] seed@ubuntu:~$ iperf -c 192.168.2.1
------------------------------------------------------------
Client connecting to 192.168.2.1, TCP port 5001
TCP window size: 21.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.1.1 port 58400 connected with 192.168.2.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec   458 MBytes   384 Mbits/sec
```

记录在data.out中的数据大致是这个样子：

```
11.172120204 193.168.0.2:8089 193.168.0.25:54320 32
0x842ccad 0x84299c5 10 168 14608 143
```

每行的各列分别为：

```
timestamp    //时间戳
saddr:port   // 源IP及端口，数据是在发送端捕捉的，所以port是固定的
daddr:port   // 目的IP及端口
skb->len     // 收到的数据包skb大小，如果收到的都是ACK包，那么len都比较小。
snd_nxt      // 下一个待发送数据的序列号
snd_una      // 待确认数据的序列号
snd_cwnd     // 拥塞窗口大小
ssthresh     // 慢启动阈值
snd_wnd      // 接收窗口大小
srtt         // smoothed RTT
```

实际的例子，看到拥塞窗口线性增加：

```
46.587495893 192.168.1.1:58401 192.168.2.1:5001 32 0x498cef1f 0x498c7dff 23 16 118784 1
46.588639927 192.168.1.1:58401 192.168.2.1:5001 32 0x498d4f47 0x498cde27 24 18 133248 1
46.589521999 192.168.1.1:58401 192.168.2.1:5001 32 0x498da9c7 0x498d1c5f 25 18 165120 1
46.590444279 192.168.1.1:58401 192.168.2.1:5001 32 0x498ddcaf 0x498d5a97 26 19 196992 1
46.592416488 192.168.1.1:58401 192.168.2.1:5001 32 0x498e4dcf 0x498dd15f 27 19 199936 1
46.593640396 192.168.1.1:58401 192.168.2.1:5001 32 0x498eb947 0x498e4dcf 28 21 199936 1
46.595295931 192.168.1.1:58401 192.168.2.1:5001 32 0x498f51ff 0x498eca3f 29 21 199936 1
46.596185745 192.168.1.1:58401 192.168.2.1:5001 32 0x498ff05f 0x498f46af 30 22 199936 1
46.597034003 192.168.1.1:58401 192.168.2.1:5001 32 0x4990617f 0x498fc31f 31 22 199936 1
46.598337979 192.168.1.1:58401 192.168.2.1:5001 32 0x4990f48f 0x49903f8f 32 24 199936 1
46.599525889 192.168.1.1:58401 192.168.2.1:5001 32 0x49914f0f 0x4990bbff 33 24 199936 1
46.599697273 192.168.1.1:58401 192.168.2.1:5001 32 0x4991cb7f 0x49911c27 34 25 214400 1
46.600584263 192.168.1.1:58401 192.168.2.1:5001 32 0x49920f5f 0x49915a5f 35 25 246272 1
46.602239674 192.168.1.1:58401 192.168.2.1:5001 32 0x4992971f 0x4991cb7f 36 27 252032 1
46.603035489 192.168.1.1:58401 192.168.2.1:5001 32 0x4992e64f 0x499247ef 37 27 252032 1
46.603535137 192.168.1.1:58401 192.168.2.1:5001 32 0x49937f07 0x4992c45f 38 28 252032 1
```

看到拥塞窗口线性减少，ssthresh折半回退（跟特定的拥塞控制算法相关例如Vegas）：

```
46.802903932 192.168.1.1:58401 192.168.2.1:5001 60 0x4a1c0aa7 0x4a0ee52f 259 254 874624 3
46.802962135 192.168.1.1:58401 192.168.2.1:5001 60 0x4a1c15f7 0x4a0ee52f 258 254 874624 3
46.803020867 192.168.1.1:58401 192.168.2.1:5001 60 0x4a1c2147 0x4a0ee52f 257 254 874624 3
46.803095039 192.168.1.1:58401 192.168.2.1:5001 60 0x4a1c323f 0x4a0ee52f 256 254 874624 3
46.803144033 192.168.1.1:58401 192.168.2.1:5001 60 0x4a1c3d8f 0x4a0ee52f 255 254 874624 3
46.803145283 192.168.1.1:58401 192.168.2.1:5001 60 0x4a1c3d8f 0x4a0ee52f 254 254 874624 3
46.813263356 192.168.1.1:58401 192.168.2.1:5001 52 0x4a213637 0x4a1711ff 253 177 711040 2
46.813313576 192.168.1.1:58401 192.168.2.1:5001 52 0x4a214187 0x4a1711ff 252 177 711040 2
46.813367241 192.168.1.1:58401 192.168.2.1:5001 52 0x4a214cd7 0x4a1711ff 251 177 711040 2
46.813452744 192.168.1.1:58401 192.168.2.1:5001 52 0x4a215dcf 0x4a1711ff 250 177 711040 2
46.813510809 192.168.1.1:58401 192.168.2.1:5001 52 0x4a21691f 0x4a1711ff 249 177 711040 2
46.813556590 192.168.1.1:58401 192.168.2.1:5001 52 0x4a21746f 0x4a1711ff 248 177 711040 2
46.813628023 192.168.1.1:58401 192.168.2.1:5001 52 0x4a218567 0x4a1711ff 247 177 711040 2
46.813675527 192.168.1.1:58401 192.168.2.1:5001 52 0x4a2190b7 0x4a1711ff 246 177 711040 2
```

4.增加延迟和丢包

可以在SEED-Router这台虚拟机连接SEED-Server网卡上增加延迟和丢包来改变网络环境，利用的是Linux中的 traffic control 工具，以及内核中的Network Simulator特性：

```
# tc qdisc change dev eth0 root netem loss 2.5%
设置丢包
# tc qdisc add dev eth0 root netem delay 30ms 10ms
设置延迟
# tc qdisc add dev eth0 root netem duplicate 1%
该命令将 eth0 网卡的传输设置为随机产生 1% 的重复数据包
# tc qdisc add dev eth0 root netem corrupt 0.2%
该命令将 eth0 网卡的传输设置为随机产生 0.2% 的损坏的数据包

删除网卡上面的相关配置，将之前命令中的 add 改为 del 即可删除配置
```

qdisc表示在网卡上的排队策略，可以通过ifconfig命令查看：

```
root@dl:~# ip link show
2: eth0: <BROADCAST,MULTICAST,PROMISC,UP> mtu 1500 qdisc
pfifo_fast qlen 100
link/ether 52:54:00:de:bf:19 brd ff:ff:ff:ff:ff:ff
```

qdisc支持的排队策略包括：

```
FIFO - simple FIFO (packet (p-FIFO) or byte (b-FIFO) )
PRIO - n-band strict priority scheduler
TBF - token bucket filter
CBQ - class based queue
CSZ - Clark-Scott-Zhang
SFQ - stochastic fair queue
RED - random early detection
GRED - generalized random early detection
TEQL - traffic equalizer
ATM - asynchronous transfer mode
DSMARK - DSCP (Diff-Serv Code Point)marker/remarker
```

如果条件所限，还可以利用Linux防火墙来进行丢包，在Router上运行以下命令，实现0.5%的丢包（**丢包策略不固定在某个网卡**）

```
iptables -I INPUT -s 121.14.48.1 -m statistic --mode random
--probability 0.005 -j DROP
```

5.实验要求

根据以上网络配置和TCP性能测试方案，改变Router的延迟和丢包来测试TCP拥塞控制算法的性能：

1. 选择Linux内核支持的三种拥塞控制算法
2. 针对每种拥塞控制算法进行性能测试，在性能测试中改变SEED-Router的丢包率和延迟，得到不同的性能测试结果，记录测试结果，比较三种拥塞算法的优缺点
3. 如果性能不行，换成slitaz虚拟机测试（需要编译内核）
4. 对实验结果进行分析和比较（最好有图表说明）