CS339 Project 2

姓名：刘雨巷　　学号：516021910395

致谢

本次完成第二次网络课作业基于对以下实例代码的参考，特向它们的作者致谢!

examples/matrix-topology/matrix-topology.cc

examples/routing/simple-global-routing.cc

examples/traffic-control/queue-discs-benchmark.cc

examples/traffic-control/traffic-control.cc

程序说明

该 cc 程序首先读入一个邻接矩阵文件（adjacency_matrix.txt）。

```
// ---------- Read Adjacency Matrix ------------------------------------

vector<vector<bool> > Adj_Matrix;
Adj_Matrix = readNxNMatrix(adj_mat_file_name);
int n_nodes = Adj_Matrix.size();
//printMatrix (adj_mat_file_name.c_str (),Adj_Matrix);

// ---------- End of Read Adjacency Matrix -----------------------------
```

基于自定义的读取函数 readNxNMatrix()，其输入为 txt 文件，文件的第一行分别是结点数和链路数，以下每行分别是主机和路由器以及路由器之间链路的端点，其中 0~49 对应 50 个主机，50~79 对应 30 个路由器。输出为一个标准的邻接矩阵，bool 变量 0 和 1 分别表示对应的结点是否直接邻接。通过 printMatrix（）函数可以进行输出验证。

之后基于拓扑结构相对应的建立一个点对点链路的有线网络拓扑。设置链路的传输速率，延迟，以及排队队列。在每个结点上安装栈协议，分配 IP 地址，最后在结点之间建立链路，因篇幅原因以下展示部分。

```
for (size_t i = 0; i < Adj_Matrix.size(); i++)
{
    for (size_t j = 0; j < Adj_Matrix[i].size(); j++)
    {
        if (Adj_Matrix[i][j] == 1)
        {
            NodeContainer n_links = NodeContainer(nodes.Get(i), nodes.Get(j));
            NetDeviceContainer n_devs = p2p.Install(n_links);

            QueueDiscContainer qdiscs = tch.Install(n_devs);
            q[i][j] = qdiscs.Get(1);
            q[i][j]->TraceConnectWithoutContext("PacketsInQueue", MakeCallback(&TcPacketsInQueueTrace));

            ipv4_n.Assign(n_devs);
            ipv4_n.NewNetwork();
            linkCount++;
            NS_LOG_INFO("matrix element [" << i << "][" << j << "] is 1");
        }
        else
        {
            NS_LOG_INFO("matrix element [" << i << "][" << j << "] is 0");
        }
    }
}
```

然后创建网络流，每 0.1s 随机生成一次数量为 1~25 的随机主机对群组，进行点对点通信。需要注意，在一对主机对中，两个主机是不能相同的，但不同对的主机对之间，是可以重复的。举例来说，主机 A 向主机 B 发送消息，A 和 B 是不能相同的；但可以存在两对主机对，分别是 A 向 B 发送信息，A 向 C 发送信息。所以只需要保证在生成一对主机的时候两者是不同的即可。注意生成的随机主机对数量为[1,25]的整数，而结点的编号为[0,49]的整数。在程序中，最外层的循环对应着 1200 个时间片（120s 总测试时间，每个时间片为 0.1s）。在安装应用程序时，采取 onoff，设置对应的数据包为 UDP 报文，并设置每个包的大小，数据包的发送速率。

```
for (int t = 0; t < 1200; t++) {
    //cout << endl << " *** *** *** StartTime:  " << StartTime + t * 0.1 << "  StopTime:  " << StopTime +

    // Create random number of random couples of hosts
    //cout << endl << "*** Hosts Couples Info ***" << endl;
    n_couple[t] = (rand() % 25) + 1;  //n_couple[t] in [1,25]
    //cout << "  Number of hosts couples  " << n_couple[t] << endl;

    for (int k = 0; k < n_couple[t]; k++) {
        //i, j in [0,50)
        int i = (rand() % 50), j;
        do {
            j = (rand() % 50);
        } while (j == i);
        //cout << "  Client:  " << j << "  Server:  " << i << endl;

        // traffic flows from node[i] to node[j]
        apps_sink[k][t] = sink.Install(nodes.Get(i));
        apps_sink[k][t].Start(Seconds(StartTime + 0.1 * t));
        apps_sink[k][t].Stop(Seconds(StopTime + 0.1 * t));

        NS_LOG_INFO("Setup Traffic Sources.");

        // Create the OnOff application to send UDP datagrams of size 210 bytes at a rate of 448 Kb/s
        Ptr<Node> n = nodes.Get(j);
        Ptr<Ipv4> ipv4 = n->GetObject<Ipv4>();
        Ipv4InterfaceAddress ipv4_int_addr = ipv4->GetAddress(1, 0);
        Ipv4Address ip_addr = ipv4_int_addr.GetLocal();
        OnOffHelper onoff("ns3::UdpSocketFactory", InetSocketAddress(ip_addr, port));
        onoff.SetAttribute("OnTime", StringValue("ns3::ConstantRandomVariable[Constant=0.0001602173]"));
        onoff.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0.00375]"));
        onoff.SetAttribute("PacketSize", UintegerValue(PacketSize));  // int PacketSize = 210
        onoff.SetConstantRate(DataRate(AppPacketRate));  // string AppPacketRate("448Kbps")
        apps[k][t] = onoff.Install(nodes.Get(i));
        apps[k][t].Start(Seconds(StartTime + 0.1 * t));
        apps[k][t].Stop(Seconds(StopTime + 0.1 * t));
    }
}
```

最后是仿真阶段，为了获取每个流的信息以便于后期相关数据（除了队列长，队列长下文另外说）的获取，利用 flow monitor 模块中的 minitor 生成相应的文件，然后再从中获取并处理流信息以得到时延，吞吐量和丢包率。

```
Ptr<FlowMonitor> flowmon;
FlowMonitorHelper flowmonHelper;
flowmon = flowmonHelper.InstallAll();

NS_LOG_INFO("Run Simulation.");

for (double s = 0.1000; s <= 120.00; s += 0.1000) { ... }

Simulator::Stop(Seconds(SimTime));
Simulator::Run();
flowmon->SerializeToXmlFile(flow_name.c_str(), true, true);

Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowmonHelper.GetClassifier());
map<FlowId, FlowMonitor::FlowStats> stats = flowmon->GetFlowStats();
```

参数的设置

首先是时间方面的设置，其中 SimTime 仿真时间总共为 120s，StartTime 是第一个 0.1s 的开始时间，StopTime 是第一个 0.1s 的结束时间，以后每个时间片递增 0.1s 即可。此处 StartTime 我认为不能设置成 0s，因为每个时间片的开始时间不能和上个时间片的结束时间一样，否则在仿真时可能会出现问题。

```
double SimTime = 120.00;
double StartTime = 0.00001;
double StopTime = 0.10000;
```

对于链路的传输速率 10Mbps 和延迟 2ms，我们利用点对点通信时，进行如下属性的设置。

```
NS_LOG_INFO("Create P2P Link Attributes.");
PointToPointHelper p2p;
p2p.SetDeviceAttribute("DataRate", StringValue(LinkRate));  // string LinkRate("10Mbps");
p2p.SetChannelAttribute("Delay", StringValue(LinkDelay));  // string LinkDelay("2ms");
```

对于路由器的缓存长度，即我们所考虑的队列长，因为缓存长度是 2M，而每个包的大小为 210bytes，近似将队列中的最大包数设置为 10000，设置如下。

```
NS_LOG_INFO("Set Queue.");
TrafficControlHelper tch;
uint16_t handle = tch.SetRootQueueDisc("ns3::RedQueueDisc");
// Add the internal queue used by Red
// 2M buffer for 210 bytes per packet, approximately 10000
tch.AddInternalQueues(handle, 1, "ns3::DropTailQueue", "MaxPackets", UintegerValue(10000));
```

对于报文的大小 210bytes，包的发送间隔 3.75ms，我们采取 onoff，依次进行如下设置。

```
onoff.SetAttribute("OffTime", StringValue("ns3::ConstantRandomVariable[Constant=0.00375]"));
onoff.SetAttribute("PacketSize", UintegerValue(PacketSize));  // int PacketSize = 210
onoff.SetConstantRate(DataRate(AppPacketRate));  // string AppPacketRate("448Kbps")
```

数据的获取

由于使用的 flow monitor 模块主要负责流信息的记录，所以时延，吞吐量，丢包率等都可以从相应的流信息中直接或间接地计算出来，而排队队列这一实时信息则需要通过另一种方式——Simulator 的 Schedule 来收集。

最大队列长度的获取：最大队列的长度包括每个时间片（0.1s）内的最大长度和总测试时间内的最大队列长度。对于每个时间片内，我们通过 Simulator 的 Schedule 来定时执行获取当前时间片内的最大队列长度，此处设置为每 0.1s 获取一次。

```
for (double s = 0.1000; s <= 120.00; s += 0.1000) {
    Simulator::Schedule(Seconds(s), &getInfo);
}
```

其中 Schedule 的参数为当前时间和调用的函数 getInfo()，调用的函数输出当前时间片内的最大队列，并在输出之后把 current_max_q 清零以重新计算下一时间片内的最大队列长。其中 current_max_q 是由回调函数（稍后介绍）进行不断更新的，如下所示：

```
void getInfo()
{
    //output the max queue for current time slot
    cout << "  Max Queue at ";
    cout << Simulator::Now().GetSeconds() << "\t" << current_max_q << endl;
    //reset for the next time slot
    current_max_q = 0;
}
```

关于回调函数在此处作一个说明，在建立网络拓扑时，对于相连的结点 i 和 j，我们安装了 device，之后用 q[i][j]表示其队列长，每次队列长发生变化时，就执行一次回调函数，由于对所有队列长执行相同的回调函数，所以最后获取的就是与结点无关（所有结点）的最大队列长。回调函数使用如下：

```
QueueDiscContainer qdiscs = tch.Install(n_devs);
q[i][j] = qdiscs.Get(1);
q[i][j]->TraceConnectWithoutContext("PacketsInQueue", MakeCallback(&TcPacketsInQueueTrace));
```

而在回调函数内部，每当存在一个 q[i][j]发生变化时，就更新 max_q（所有时间内的最大）和 current_max_q（每个时间片的最大），如下：

```
void TcPacketsInQueueTrace(uint32_t oldValue, uint32_t newValue)
{
    if (max_q <= int(newValue)) max_q = int(newValue);
    if (current_max_q <= int(newValue)) current_max_q = int(newValue);
}
```

时延，吞吐量和丢包率的获取：如前所述，由于采取了 FlowMonitorHelper，所以可以直接从流记录的信息中获取，首先用 GetFlowStas()获取流统计信息记录至 stas，如下所示：

```
Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowmonHelper.GetClassifier());
map<FlowId, FlowMonitor::FlowStats> stats = flowmon->GetFlowStats();
```

之后便可访问 stas，其中每个时间片内流的数量与主机对数一致，每个流有多个包，由于流的信息按照时间顺序排列记录下来，所以访问完每个时间片内主机对数目的流后再访问便是下一个时间片内的流的信息。如此便可计算各种数据在不同时间片内的均值。其中 t 表示 time slot 的编号，st 表示流的编号（注意在 stas 中流的 ID 是从 1 开始的），如下：

```
int st = 1;
for (int t = 0; t < 1200; t++) {
    double throughput = 0.0, current_throughput;
    double delay = 0.0, current_delay;
    double loss = 0.0, current_loss;
    double jitter = 0.0, current_jitter;
    for (int num = st; num < st + n_couple[t]; num++) {
        current_loss = 1.0 - double(stats[num].rxPackets) / stats[num].txPackets;
        loss += current_loss;

        current_throughput = stats[num].rxBytes * 8.0 / (stats[num].timeLastRxPacket.GetSeconds() - stats[num].timeFirstRxPacket.GetSeconds()) / 1000000;
        throughput += current_throughput;

        current_delay = stats[num].delaySum.GetSeconds() / stats[num].rxPackets;
        delay += current_delay;

        current_jitter = stats[num].jitterSum.GetSeconds() / (stats[num].rxPackets - 1);
        jitter += current_jitter;
    }
    //cout << "  Mean Delay:  ";
    //cout << StartTime + 0.1*t << '\t';
    //cout << delay / n_couple[t] << endl;

    //cout << "  Mean Throughput:  ";
    //cout << StartTime + 0.1*t << '\t';
    //cout << throughput / n_couple[t] << endl;

    //cout << "  Mean Loss Rate:  ";
    //cout << StartTime + 0.1*t << '\t';
    //cout << loss / n_couple[t] << endl;

    //cout << "  Mean Jitter:  ";
    //cout << StartTime + 0.1*t << '\t';
    //cout << jitter / n_couple[t] << endl;

    st += n_couple[t];
}
```

实验仿真结果

为了便于观察，我输出了一些**中间结果**：

首先是在一个时间片中随机生成的主机对，以下展示在 110~110.1s 内随机生成的 19 对主机对。

```
*** *** *** StartTime:  110  StopTime:  110.1 *** *** ***

*** Hosts Couples Info ***
  Number of hosts couples  19
  Client:  44  Server:  39
  Client:  15  Server:  14
  Client:  38  Server:  3
  Client:  48  Server:  31
  Client:  3  Server:  38
  Client:  1  Server:  9
  Client:  46  Server:  28
  Client:  25  Server:  30
  Client:  2  Server:  4
  Client:  27  Server:  44
  Client:  34  Server:  6
  Client:  10  Server:  19
  Client:  34  Server:  11
  Client:  2  Server:  49
  Client:  22  Server:  3
  Client:  44  Server:  22
  Client:  39  Server:  18
  Client:  22  Server:  9
  Client:  43  Server:  27
```

如下展示随机生成的主机对只有一组的所有输出，包括时间片序号（总共 1200 个），流 ID，表示到目前为止产生了 14775 个流（或者说 14775 个随机主机对，因为一个主机对在 0.1s 内只有一个流），但在该 0.1s 的时间片中，每个流只有 10 个包。

```
*** *** *** *** *** *** *** *** *** ***
  Time Slot:   1123
*** *** *** *** ***
  Flow ID:   14775
  Tx Packets:   10
  Rx Packets:   10
  Loss Rate:  0
  Throughput: 0.525 Mbps
  Mean delay:   0.012168
  Mean jitter:   0
*** *** *** *** ***
  Mean Delay:  0.012168
  Mean Throughput:  0.525
  Mean Loss Rate:  0
  Mean Jitter:  0
```

这样的实验结果并不如预期所想，目前我认为：尽管设置了相关参数，但在路由协议中或者其他网络条件会有自己的一套参数存在影响，所以实际每个 0.1s 的时间片中，每个流才会只有 10 个包。

接下来是对所有队列的输出情况，以下展示在连续多个时间片上的最大队列。

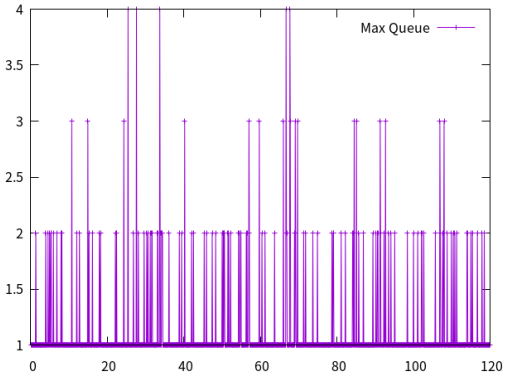
```
Max Queue at 65.7     1
Max Queue at 65.8     1
Max Queue at 65.9     1
Max Queue at 66       1
Max Queue at 66.1     3
Max Queue at 66.2     4
Max Queue at 66.3     1
Max Queue at 66.4     1
Max Queue at 66.5     2
Max Queue at 66.6     1
Max Queue at 66.7     2
Max Queue at 66.8     2
Max Queue at 66.9     1
Max Queue at 67       5
Max Queue at 67.1     2
Max Queue at 67.2     1
```

最后展示在结束时的一些数据输出，可以看到最后的流 ID 是 15708，表示整个仿真过程中所有流的数量；Max Queue 是整个仿真中出现的最大队列长（即所有时间片上最长队列长度的最大值）。

```
*** *** *** *** ***
 Flow ID:   15708
 Tx Packets:   10
 Rx Packets:   9
 Loss Rate:  0.1
 Throughput: 0.531563 Mbps
 Mean delay:   0.012168
 Mean jitter:   0
*** *** *** *** ***
 Mean Delay:  0.012168
 Mean Throughput:  0.531563
 Mean Loss Rate:  0.1
 Mean Jitter:  0
*** *** *** *** *** *** *** *** *** ***
 Max Queue:   5
```

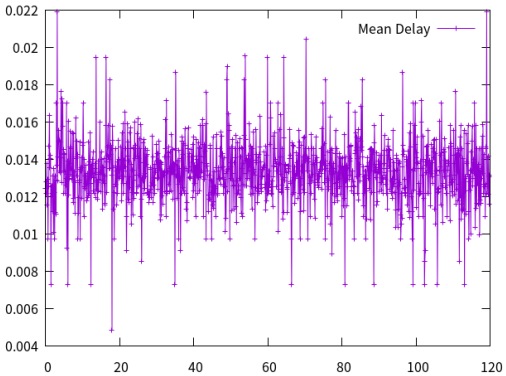
以下是数据输出的**可视化**部分：

1、最大队列长度：队列内包的数量（p）~时间（s）



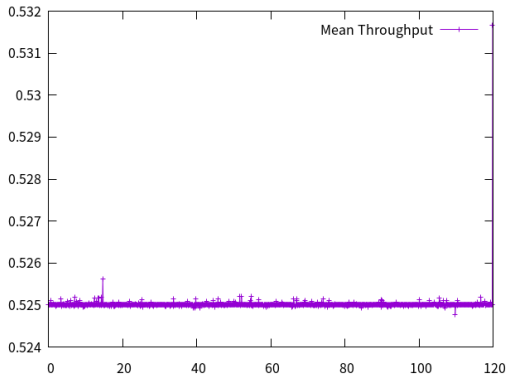
可以看出，每个时间片的最大队列长度都比较小（因为在每个 0.1s 内一对主机对的 flow 只有 10 个 packets），但滞留在队列中的 packets 数量达到了 4 成。由于是每 0.1s 进行统计最大队列，而在 0.1s 内肯定存在通信，当由于包的发送与接收导致队列中的 packet 数由 1 变为 0 或者由 0 变为 1 时，那么就会得到 1，所以队列中至少有一个包。

2、平均时延：时延（s）~时间（s）



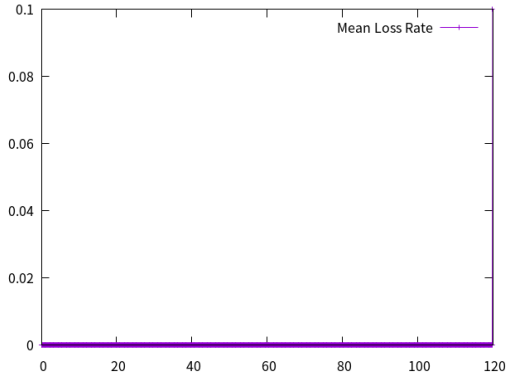
平均时延波动比较大，但基本在[0.004,0.022]范围内变化，变化的范围较小。

3、平均吞吐量：吞吐量（Mbps）~时间（s）



平均吞吐量的变化也相对较小，基本上只是在 0.525 附近波动。

4、平均丢包率：丢包的比例~时间（s）



平均丢包率除了最后一个时间片时为 0.1（表示平均丢了一个包，因为每个流有 10 个 packets），其余均为 0，但经过多次实验发现，始终是 0.1 的丢包率，我认为，这是因为最后设置的接收端的结束时间和发送端的结束时间都在 120.000s，发送端最后发送的包接收端来不及接收就关闭了，所以始终存在 1 个 packet loss。

实验结果显示，每个时间片的：最大队列长度始终在比较小的范围内变化；平均时延变化稍大，但是仍在较小的范围内变化；平均吞吐量也保持稳定的状态，在较小的范围内波动；平均丢包率则除了最后一个时间片存在 0.1 的平均丢包率，始终为 0。

实验总结

本次实验要求学习 ns3，自定义路由协议，优化网络以实现拥塞控制，输出相应的参数。实验结果表明，最后的各方面指标均较好。在完成作业过程中，ns3 的 examples 给我很大帮助，通过学习和模仿 examples 中和作业要求有关的示例，使得最后的代码比较的清晰和规范。通过这次实验，我对 ns3 有了比较深的了解，同时也巩固了课堂所学的基础网络知识。

注：以上列出的代码只是主要程序块，具体可参考相应的 cc 文件。另外，所提供的两个 cc 文件中，hw_all.cc 是输出完成仿真状态（很多中间结果）的代码，hw_clear.cc 是输出实验要求的四个指标（最大队列长度，平均时延，平均吞吐量，平均丢包率）到指定的四个 dat 文件中，不需要命令行设置并进行重定向。

所以运行时只需要：./waf 和./waf –run scratch/hw_clear。