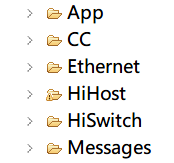
# HiNA-DCN INET项目

本文档记录了我们组使用的适用于omnetpp6.0的INET项目，基于inet4.4构建，只需要使用omnetpp6.0自动安装一个INET框架即可移植进去运行。

项目位置为INET4.4/src/inet，其中**examples**文件夹为拓扑及配置文件，而其中**HiNA**文件夹的架构为：



App为应用层模块，CC为传输层模块，Ethernet为链路层模块，HiHost与HiSwitch为主机节点与交换机节点的ned文件，Messages文件夹存放我们自己定义的.msg文件，它编译后可以直接生成对应的.cc和.h文件。

本文档主要描述各模块的实现思路与使用配置，给出的配置需要根据不同情况自行调整以达到最优性能。

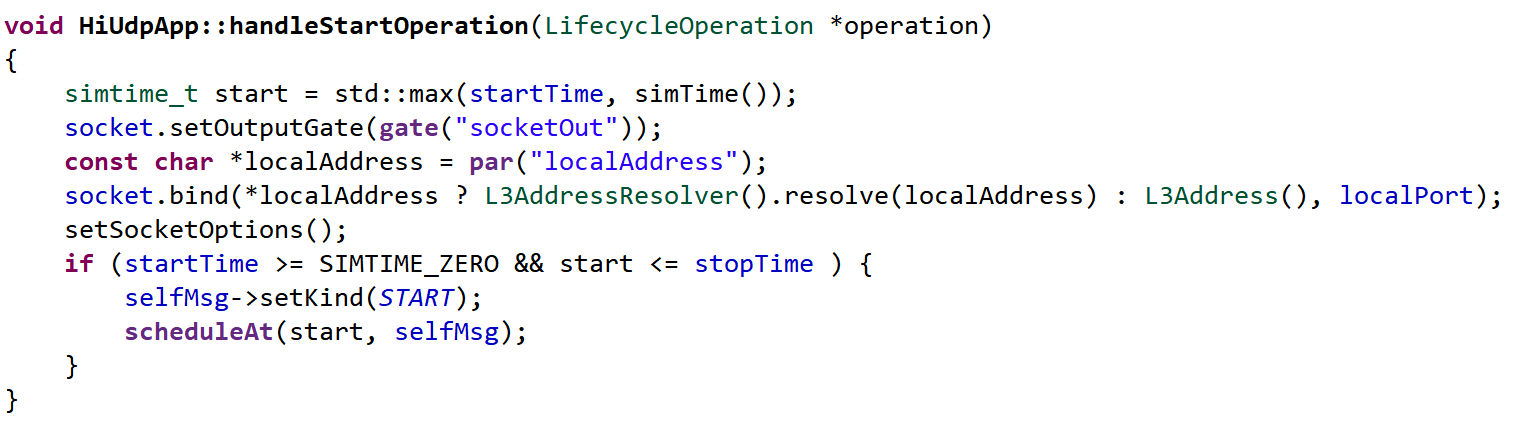
# 一、应用层

应用层我们实现的功能有收发一体、配置workload比例和类型、标记tag信息、全局流id和测量goodput/FCT等。应用层APP分为UdpApp和TcpApp，其中UdpApp因为重名所以命名为HiUdpApp，数据中心中TCP主要是指DCTCP为主的通信流，UDP则是指RDMA中的存储流。

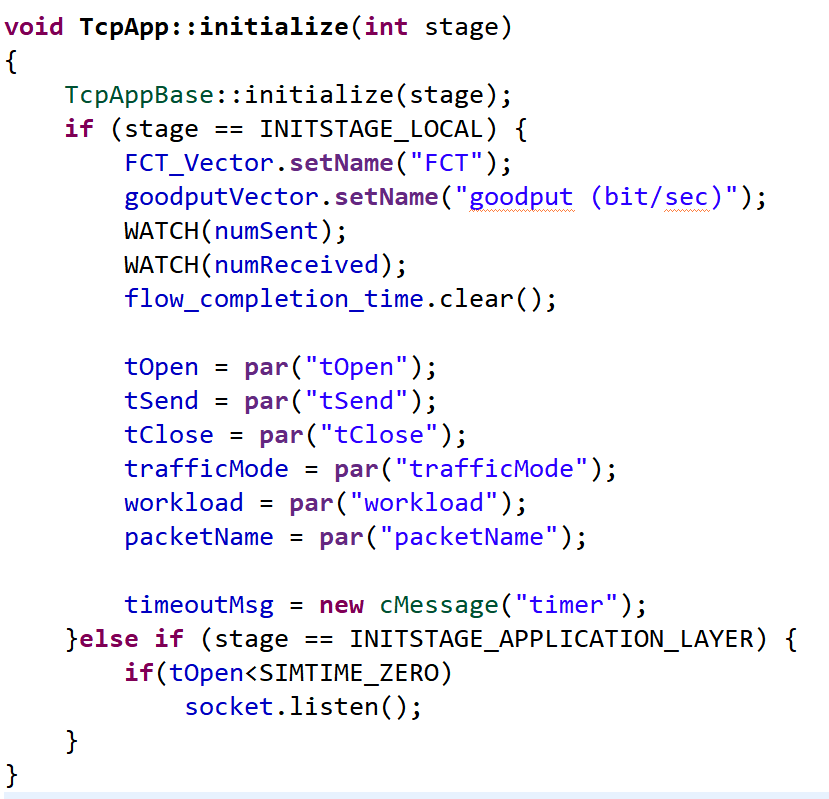
## 1.收发一体

收发一体即两端使用同一个APP模块就能完成接收与发送。

因此对于HiUdpApp，我们将绑定socket的操作放在发送端接收端都会经历的handleStartOperation()中，startTime为默认值-1时即视为接收端。



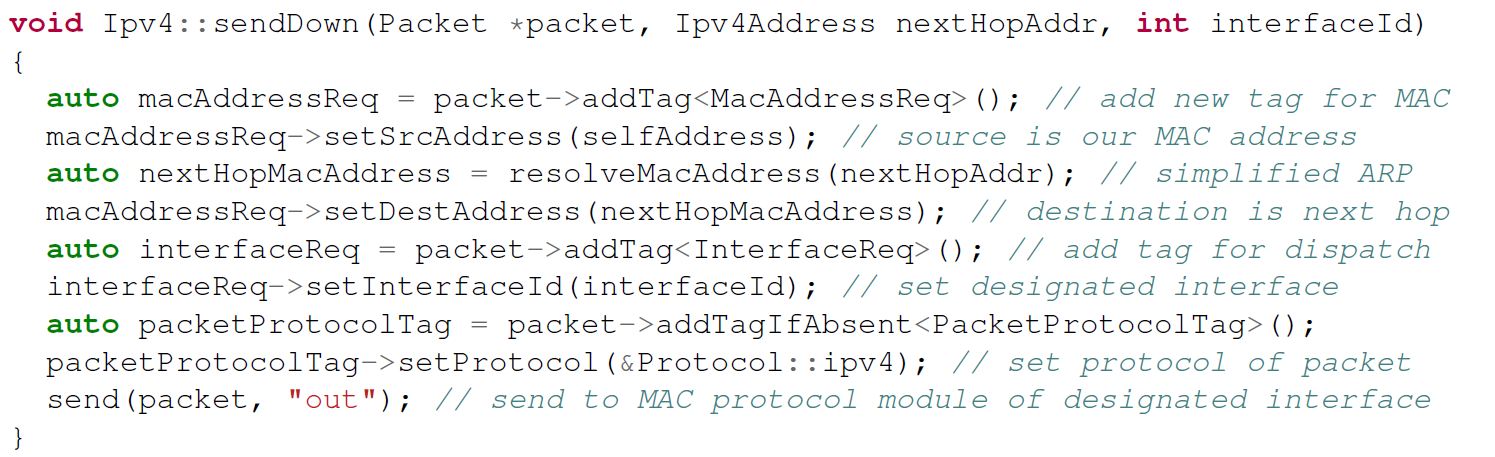
而对于TcpApp，接收端需要保持一个socket监听，再新建一个socket接收数据。所以我们结合了TcpSessionApp的自定义发送流量功能与TcpSinkApp的多socket接收功能（暂未使用其多线程功能）。由于接收端需要用TcpSocket.listen()开启监听才能正常进行三次握手，而父类TcpAppBase已经在initialize()的INISTAGE\_APPLICATION\_LAYER阶段就完成了socket绑定，所以在TcpApp相同阶段的initialize()添加。tOpen为默认值-1时即视为接收端，绑定本地端口后开启监听。



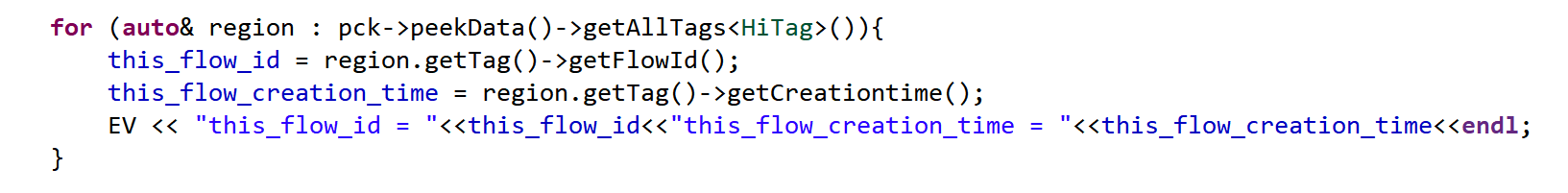
也尝试过在handleStartOperation()中添加但会报listen()之前必须bind()的错，但实际上handleStartOperation()确实是在所有initialize()之后触发的。

## 2.标记信息

由于INET4.4增加了函数使用的规范性，对于附在包后面的payload不能再像以前用peekAtBack<类型>()函数直接提取，而是需要指定具体偏移量与长度，而这对网络传输中的其它层而言是未知的。所以我们改用打上Tag的方法来传递信息，tag又分为packet tag和retion tag，packet tag是形如

的标记，直接添加在包上，打上后只能在同样的封装状态下读取，因此一般用于主机内部相邻层之间传递信息，从上往下传的称为**request**，后缀**Req**，从下往上传的称为**indication**，后缀**Ind**。而region tag是形如

的标记，需要依附于某一个段，可以被跨层读取。具体读取方法如下：



我们通过message机制定义了自己的region tag——HiTag，在HiTag.msg中写入需要的参数直接编译即可生成对应的\_m.cc和\_m.h文件。

## 3.其他

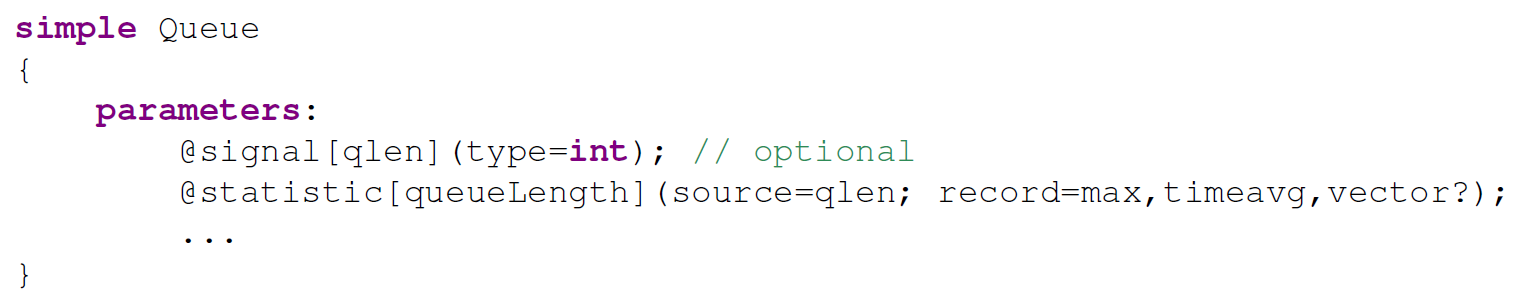
全局流id：由于TcpApp发出的报文在TCP模块中只被存储为字节，其中写入的所有信息（HiTag等）都会丢失，所以我们目前只在HiUdpApp中使用全局流id，这个功能可以用来配合实现流级别的ECMP 负载均衡。全局流id通过定义APP类的静态变量来实现，**static** uint32\_t *flowid*; ，然后在.cc文件中完成初始化uint32\_t *HiUdpApp::flowid*=0;。

工作负载（workload）：工作负载的配置分为两种方式，一种是规定比例和类型，另一种是通过sendScript手动配置报文长度和发送时间。工作负载的比例指实际发送占满链路负载的百分比，通过参数workload配置。为此，我们需要配置linkSpeed参数使APP模块知晓出口链路速率，然后用帧长度除以链路速率得到发送间隔，通过除以不同workload比例就可以得到不同比例的负载。负载类型有预设的几类包括长流（LongFlow）类型等，通过参数trafficMode配置，UDP侧的真实负载类型来自于XPASS——**[ns2-xpass/workloads at master · kaist-ina/ns2-xpass · GitHub](https://github.com/kaist-ina/ns2-xpass/tree/master/workloads)**。而使用sendScript则可以通过直接输入字符串以及python脚本生成txt文件，然后让app模块读取两种方式。目前读取txt文件的代码在Windows的debug模式下会导致乱码，只建议在release模式或Linux中直接使用！

流完成时间（FCT）：对于HiUdpApp，FCT的测量于接收时完成，接收到每一个包会查看HiTag中的islastPck标识，如果是一条流的最后一个包则计算FCT。我们使用名为FCT\_Vector的cOutVector向量来存储FCT信息，最终会生成名为“FCT”的数据。而对于TcpApp，由于前述的Tcp数据包不方便附带信息的原因，我们决定借用INET的Tcp模块在发送端自带的测量RTT的功能，来体现FCT，具体是在TcpBaseAlg:: rttMeasurementComplete ()中测量，最终存储为”rtt:vector”的数据。（目前这个数据画出来的图似乎有点问题，但导出成表格得到的数据是对的）

Goodput：goodput指去除包头、重传等开销后实际传输的吞吐量，因此我们统一在接收时测量，使用goodputVector来存储goodput信息，数据属性名为“goodput(bit/sec)”。

在使用vector统计数据时可能需要在ini文件中配置\*\*.\*.**vector-recording** = true。另外，通过信号机制来统计数据的方法也可以尝试。



## 4.使用

HiUdpAPP模块直接使用参考配置如下，发送端必配参数有connectAddresses、connectPort（默认100）、startTime≥0（默认为-1）、stopTime≥startTime（默认为-1）。可选参数有localAddress 、localPort（默认100）、workload（默认为1）、trafficMode（默认为LongFlow）、packetName（默认为”HiUdpData”）、linkSpeed（默认为1e9bps）、AppPriority（默认为1）、sendScript。接收端必配参数有localPort（默认100）。可选参数有localAddress。

TcpApp模块直接使用参考配置如下，发送端必配参数有connectAddresses、connectPort（默认100）、tOpen≥0（默认为-1）、tSend>tOpen（默认为-1）、tClose>tOpen（默认为2）。可选参数有localAddress、localPort（默认100）、packetName（默认为”TcpData”）、AppPriority（默认为1）、sendScript。接收端必配参数有localPort（默认100）。可选参数有localAddress。

# 二、传输层

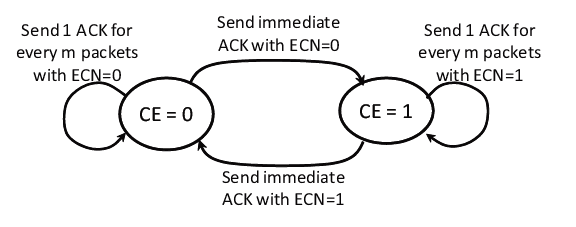
传输层实现的功能为各种拥塞控制和流量整形机制，对于UDP侧，我们往往会在UDP模块下方新增一个UDPcc来执行我们设计的算法，继承自cSimpleModule，而对于TCP侧，由于不能脱离TCP自身的框架，所以更多的把算法设计成TCP的一种算法类。

## DCTCP

DCTCP是数据中心最基础的通信协议，依赖于传统TCP框架，新增了使用ECN信号来处理拥塞的机制。INET框架对其进行了官方实现，位于transportlayer/tcp/flavours文件夹。

传统的ECN的开启需要在IP头标记ECT位（ECN Capable Transport），这是TOS段最低的两个有效位，00为非ECT，10或01为支持ECT，这样交换机在遇到拥塞的时候就可以将其改为11即CE码点。接收端网络层收到CE码点后，通过名为EcnInd的packet tag将信息传到传输层，然后接收端TCP会在接下来每一个ACK的TCP头标记ECE位（ECN Echo），发送端收到带有ECE位的ACK就会进行拥塞控制并在下一个数据包中标记CWR位（Congestion Window Reduced），通知接收端已处理拥塞，这样的操作每个RTT只会执行一次。接收端收到CWR位才会停止对ACK的标记。

而DCTCP则是按照自己的交换机进行，如下图所示。



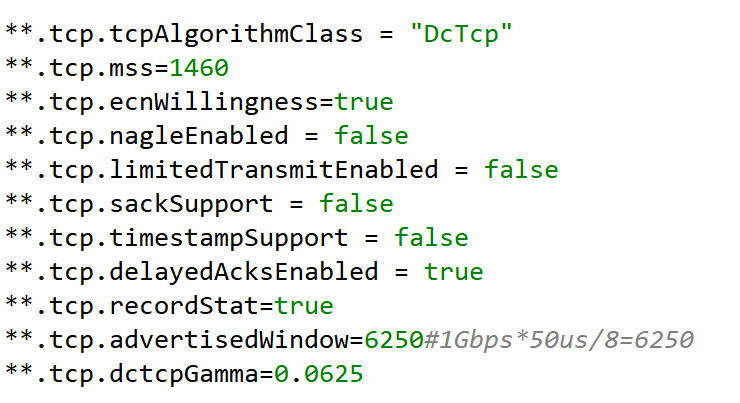
### 实现

所以INET对三个函数进行了override，分别是receivedDataAck()，shouldMarkAck()，processEcnInEstablished()。

### 使用

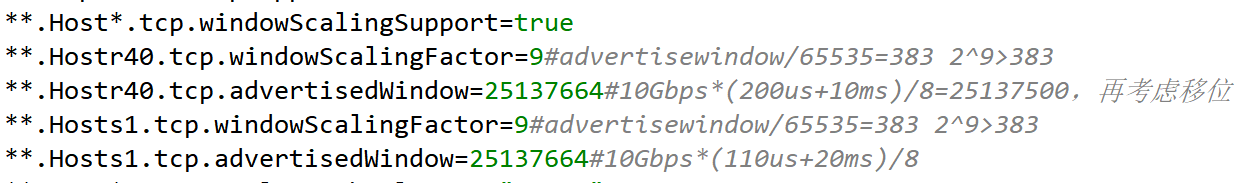
INET中ECN的使用需要先从发送端传输层通过名为EcnReq的packet tag将想要使用ECN的信息传递到网络层。

DCTCP参考使用配置如图所示：

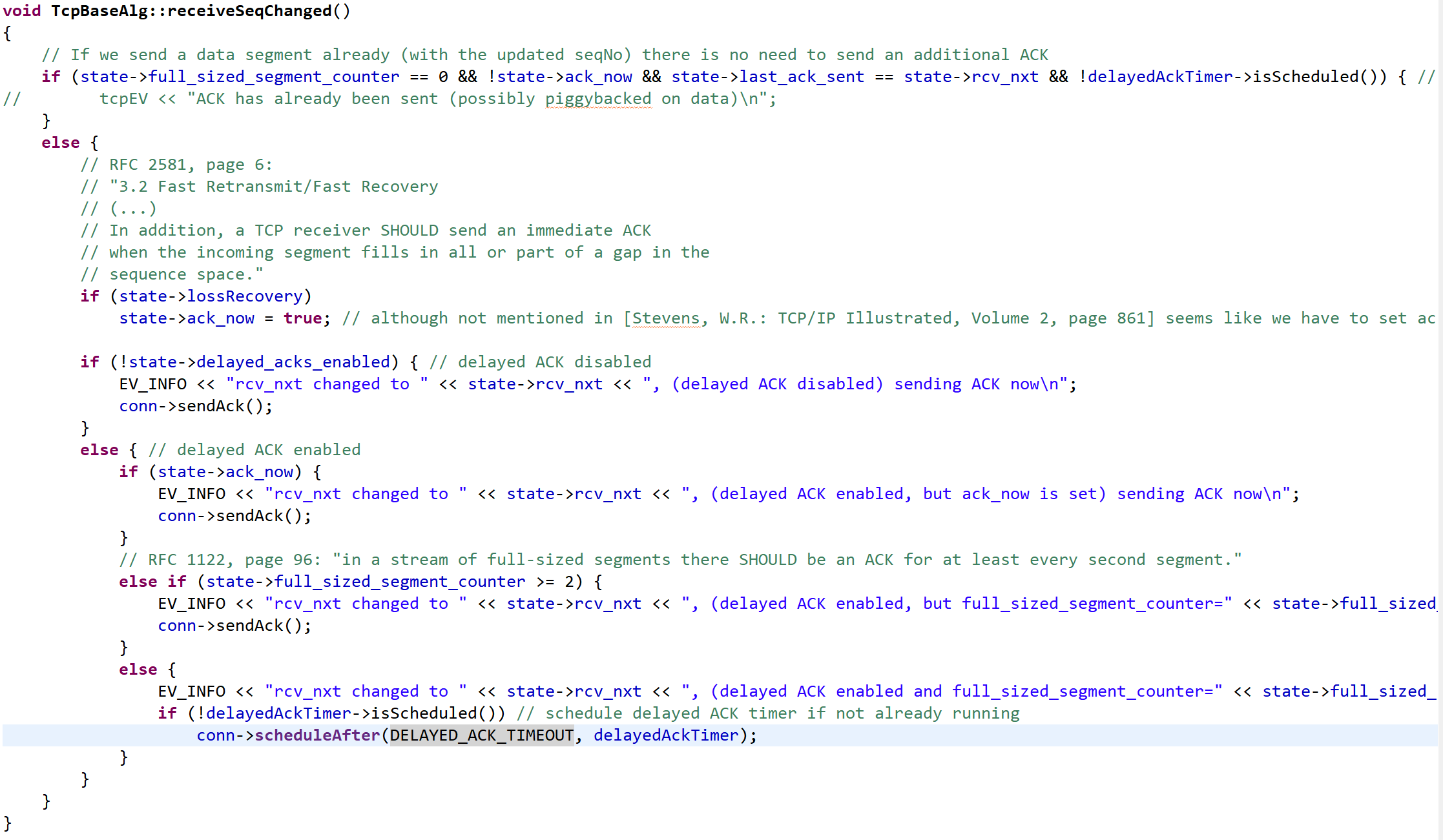


因为TCP的窗口是在拥塞窗口与接收窗口advertisedWindow中取较小值，想要跑满链路速率的话，接收窗口至少应大于等于BDP=链路速率×RTT，RTT为各跳链路延迟＋数据包（1538B）与ACK（84B）的上传延迟。而事实上这个值应该尽量大一点，因为当产生拥塞时，窗口的触发间隔可能变长，此时一个BDP的窗口大小表现出来的速率可能就小于链路速率了。同理即使窗口远大于BDP，速率也不一定能达到线速率，所以窗口大小不能作为速率的直接衡量对象。这也是为什么我们不能用接收窗口大小来控制TCP侧的workload比例。

如果接收窗口大于了65535的最大值，那么就需要进行窗口扩展（windowScaling），首先将windowScalingSupport置为true，然后去计算windowScalingFactor，此参数的意义是将65535左移几位，即乘以2的几次方，才能大于需要的接收窗口，因此可能无法传递一个精确的值。参考使用如下：



开启ecnWillingness选项即让双方在三次握手阶段开始协商将ECT位设置为支持。原文中需要开启delayedAck来实现两个包发送一次ack，减少数据中心负载，但INET示例并没有使用这一点。开启INET的delayedACK会导致在第一个窗口（默认为一个MSS大小）发送的时候，由于达不到两个包的条件所以必须等到超时（DELAYED\_ACK\_TIMEOUT=200ms）才能触发ack继续后面的传输。



使用中可以等待200ms再发送数据，也可以选择自行修改。

## XPASS

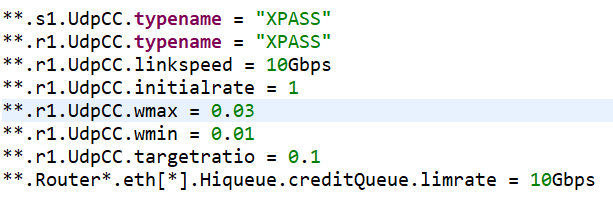
### 实现

XPASS是接收端驱动的信用协议，我们使用map<L3Address, SenderState> sender\_StateMap和map<L3Address, ReceiverState> receiver\_StateMap来维护发送端与接收端对不同地址的状态信息，以及map<uint32\_t, sender\_flowinfo> sender\_flowMap和map<L3Address, receiver\_flowinfo> receiver\_flowMap来分别根据流id和地址在发送端和接收端维护流状态信息。

它的流程是：发送端收到应用层传来的包后将其记录到sender\_flowMap、删除，并开始更新对应目的地址的sender\_StateMap发送信用请求包、等待信用包。接收端收到信用请求包后初始化对于源地址的receiver\_flowMap和receiver\_StateMap，发送信用包。发送的信用包会携带包id和创建时间，用于计算RTT、丢包率和反向路由。在XPASS的计算中，信用帧大小由包括了帧间距的最大以太网帧1538B计算得来，再加上最多8B的抖动，最终为84+[0，8]B，所以信用包负载大小为84 -12(空闲线路状态码)-14(EthernetMacHeader)-4(EthernetFcs)-8(EthernetPhyHeader)-20(IP Header) =26B。其触发的每个数据包都会携带自己的流id、信用包id、反向路由标记和创建时间返回接收端。当发送端记录的流全部传完且在一个小的超时阈值内没有新流到达时则视为发送结束，向接收端发送信用停止包。

### 使用

XPASS的使用需要开启ECMP以及RateLimit队列，队列长默认为8个信用包大小，限速固定为5%链路速率，初始速率默认为1/16线速率，丢包阈值默认为0.1，使用参考配置如图所示：



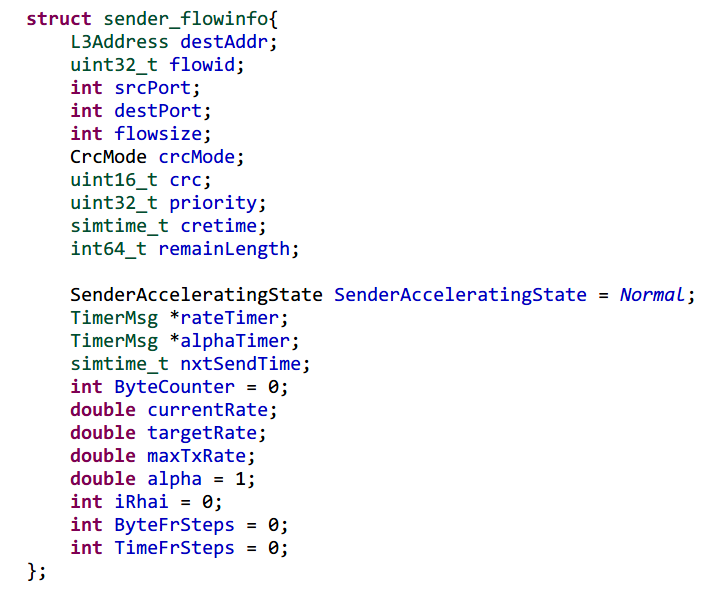
注意需要配置的是接收端的参数以及开启交换机的限速队列。最简情况只需要在接收端配置两个激进度参数wmin、wmax，w一般在0~0.5之间。而我们设计的让XPASS利用ECN和RTT信号来处理拥塞的机制可以通过参数useECN和useRTT来开启。

## DCQCN

由于PFC的使用（因为PFC是粗粒度的，VC级别的，不能分辨每条流）会损害无辜流的性能（尤其是在大范围的部署中），而流级别的QCN（quantized congestion notification）又是基于链路层实现，并且想将其扩展到网络层也很困难，所以设计了DCQCN，一种流级别的传输层CC。

### 实现

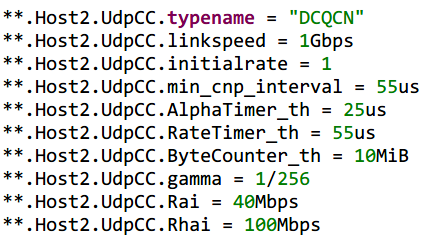
DCQCN解决PFC弊端的手段是使用流级别的拥塞控制，所以在发送端维护一个结构sender\_flowinfo，其中包括这条流的基本信息、加速状态SenderAcceleratingState和它的计时计数器。



它的流程是：首先发送端收到应用层传来的包后会将其记录到sender\_flowMap，这是一个流id与sender\_flowinfo配对的map，然后删除这个包。如果此时发送端处于停止状态就开始发送，发送端按照配置的max\_pck\_size和linkspeed对sender\_flowinfo中的流按顺序进行发送。接收端收到带有ECN标记的数据包，如果此时间隔大于这条流的最小CNP间隔就会发送CNP包。发送端收到CNP包后如果对应的流还未发完就会减速并清零计时计数器，如果超过计时计数器没收到CNP则会开始多阶段加速过程。我们定义了一个TimerMsg信息来定时触发这些事件。

### 使用

DCQCN需要配置的参数主要有链路速率linkspeed、每条流的初始速率比例initialrate、最小CNP间隔min\_cnp\_interval、计时计数器阈值和更新系数gamma等。文章给出的默认参数为计时器55μs，计数器10MB，gamma为1/256，RED-ECN标记Kmax=200KB，Kmin=5KB，Pmax=1%。

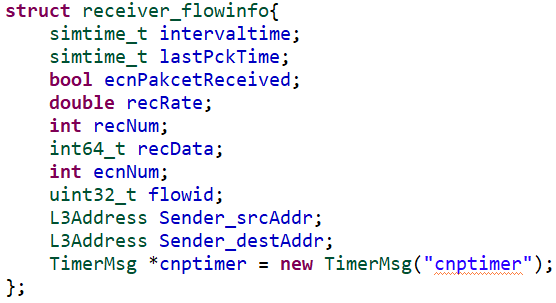


## PCN

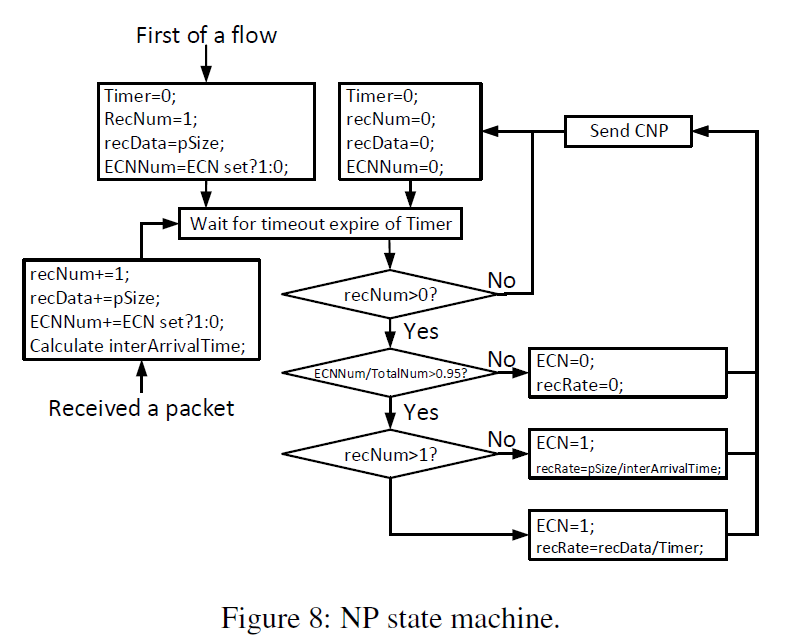
PCN主打的是一种可与PFC等逐跳流控和平共存的基于速率的端到端CC，因为要解决PFC伤害无辜流性能的问题，所以它与DCQCN一样也是流级别的CC，同时还伴有交换机上的算法设计。为了避免pause帧带来的误导性拥塞，PCN会在交换机上记录pause帧到达时队列中剩余的包，并在这些包排完之前不去标记ECN。那么到了接收端生成CNP时，就只会把一个周期内95%以上的包都有ECN标记的流视为拥塞流。同时PCN在增速和减速时都会生成CNP，与DCQCN的只在减速时生成不同。发送端收到CNP时，就可以按下示算法进行加减速而不需要用到计时器与计数器。

### 实现

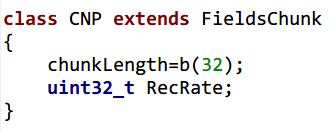
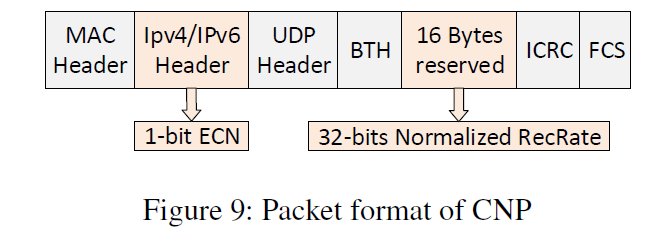
PCN与DCQCN大致上相似，所以可以在其上修改。区别主要是在于接收端收到数据包时，增加一个结构receiver\_flowinfo来维护它的状态变量。

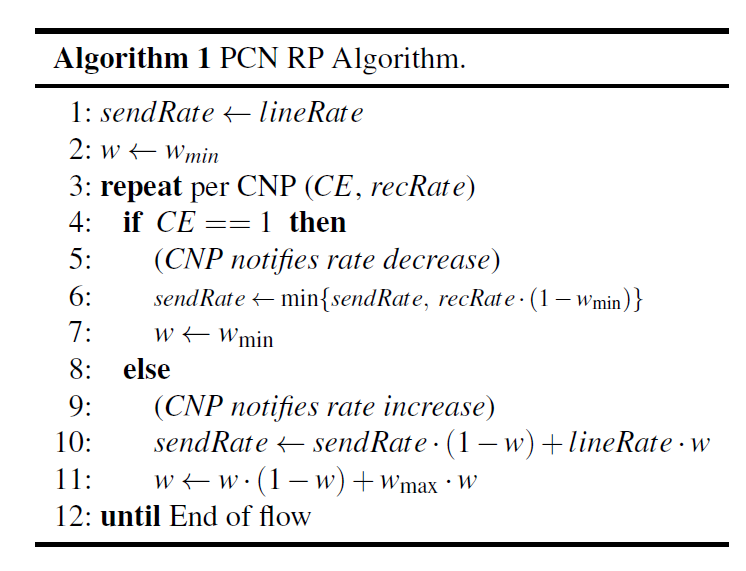


然后根据CNP周期内是否有收到包以及被ECN标记的比例来决定是否发送CNP。

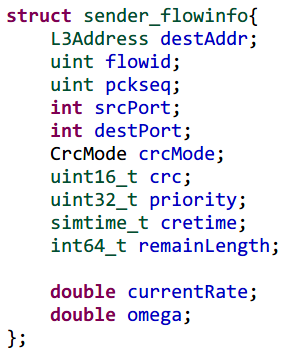


在发送CNP的时候，PCN设计了一个32位的字段来存储接受速率信息，我们将其定义为CNP字段。还利用了IP头中的ECT位来标识是否收到带有ECN标的包。



最后发送端收到CNP时，会根据下列算法决定加速与减速。 

所以sender\_flowinfo结构与DCQCN的相比主要是去除了计时计数器的部分。



### 使用

PCN需要配置的参数主要有链路速率、最小CNP间隔、更新系数omega等，其中omega\_max默认为0.5，omega\_min为0.1/N，N为共享瓶颈链路的发送端数量。每条流初始为链路速率。同时还需要开启PCNQueue。

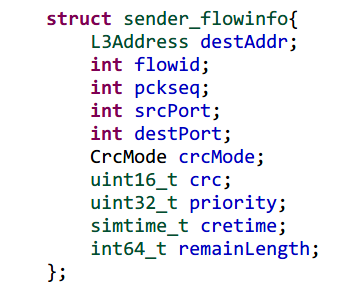


## TIMELY

TIMELY使用在链路层测量的RTT作为拥塞信号，这样可以排除主机网络栈的误差。但由于OMNET++模拟器链路层和传输层之间没有时间差，所以在哪测都一样。接收端的链路层收到数据包后会直接发回一个ACK，发送端的链路层收到ACK会计算出RTT并传往上层，传输层再根据RTT计算速率。它不是流级别的CC。

### 实现

整体架构依然与DCQCN、PCN类似，但因为不是流级别的CC，所以在sender\_flowinfo中不需要维护速率变量，仅维护报文的基本信息即可。



但TIMELY还要求在传输层以自定义段长度的方式传输，所以我们定义了TIMELYseg和accumlength参数，使其可以在一个段内连续发包。

在链路层，我们在HiEthernetMac模块的processMsgFromNetwork(EthernetSignalBase \*signal)和startFrameTransmission()中增加了TIMELY的设计，前者主要是收到数据包时判断来自其源端的数据量是否达到发送ACK的阈值，如果达到我们将构造一个ACK包发回，然后在收到TIMELYACK时计算RTT；后者则是在数据包发出时标记排除了发送端网络栈误差的真实发送时间tsend。在此需要注意的是ACK包的构造，与PFC的pause帧不同，这个ACK包虽然同样在NIC中构造，但是需要能够一直传回源端而不只是上一跳，所以还需要构造它的IP头，相当于在NIC中实现了部分网络栈功能。由于全部自行填写包头选项过于复杂，所以我们选择复制数据包包头再进行修改的方式。新建一个TIMELYACK后，UDP头直接插入，IP头我们交换了数据包的源地址与目的地址并重新填写TotalLengthField字段（因为新的ACK与数据包的长度会不同），mac头由于在NIC中不能通过IP层路由获得下一跳地址，所以我们填写了广播地址作为目的地址。



### 使用

TIMELY使用参考配置如下，注意要在使用HiEthernetMac的时候打开链路层的配置。 原文实验配置为TIMELYseg=16KB，Tlow=50μs，Thigh=500μs，Rai=10Mbps，beta=0.8。原文链路速率为10Gbps。还需要注意的是minRTT参数的配置，虽然文中说这个值的大小无所谓只是用来标准化gradient，但如果过小会导致速率被降为负，因为TIMELY并没有设置一个最小值，如果过大则会导致减速效果很弱，是影响较大的一个参数。



## HPCC

HPCC是一个sender-driven、window-based的CC，通过在数据包上打INT头，记录沿途的INT信息，然后让接收端将这些信息使用ACK传回。

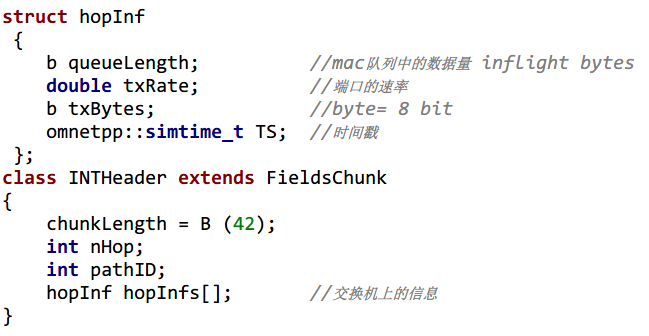
它引入了TCP的窗口和go-back-N重传机制，但需要指出的是，HPCC表示在RDMA硬件上部署滑动窗口是很麻烦的，因为滑动窗口需要支持任意包重传，从而需要对内存进行随机访问，这样仅仅是一条流的管理都是很困难的，所以HPCC采用的是发送窗口而不是滑动窗口。两者区别在于HPCC每次收到ACK就会直接计算出下一次的窗口值，而TCP的滑动窗口每收到一个ACK确认后窗口值增加一个最大段长度。

而由于使用了重传机制，所以也不需要PFC做到完全无损，HPCC将交换机队列的共享缓冲区动态阈值（dynamic threshold，DT）算法的α设为1，即单个拥塞端口可以占用一半的缓冲区。

此外它的发送端会将计算出的窗口除以baseRTT，按得到的速率进行packet pace以避免burst。

### 实现

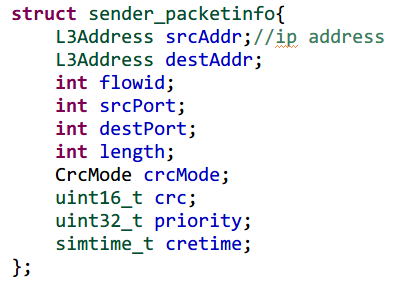
我们在INTHeader.msg中定义了HPCC需要的INT头以及一个hopInf结构体用来存储每一跳信息。再用hopInf的数组将一条路径上所有跳的信息组织起来，数组大小默认为10。



传输层方面，我们首先实现了UDP侧的窗口机制，在SenderState中加入了窗口不足时的PAUSING状态。在实现窗口机制时，需要注意的是正在使用的发送窗口与参与拥塞控制算法的拥塞窗口之间的关系，何时更新发送窗口以及不能将会实时变化的发送窗口的值当成拥塞窗口参与计算。HPCC由于采用了一个临时的参考窗口Wc，我们使用变量csend\_window表示，所以不存在此问题。针对HPCC算法的特点，我们还补充了算法中没有的最大窗口机制，因为通过实践发现如果没有这点HPCC在链路容量充裕时很容易超过链路速率导致hostqueue产生排队，进而导致在需要减速时hostqueue仍未排完而体现不出减速效果。



此外，由于HPCC采用了GBN重传机制，所以我们将sender\_flowinfo改为sender\_packetinfo，然后与连续的packetid在sender\_packetMap中配对方便重传，使用变量nxtSendPacketid作为snd\_nxt指针，sender\_packetinfo只记录包的基本状态信息。



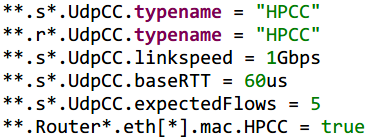
链路层设计与TIMELY一样放在HiEthernetMac中的startFrameTransmission()，以开关形式控制。



此处用到的模块名为HiEthernetInterface内部使用HiQueue队列模块的情况，如果队列模块或者端口模块发生更改，那么模块名也有可能随之变化。

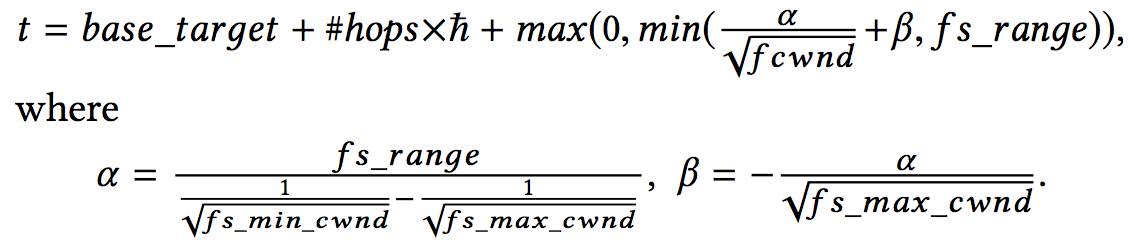
### 使用

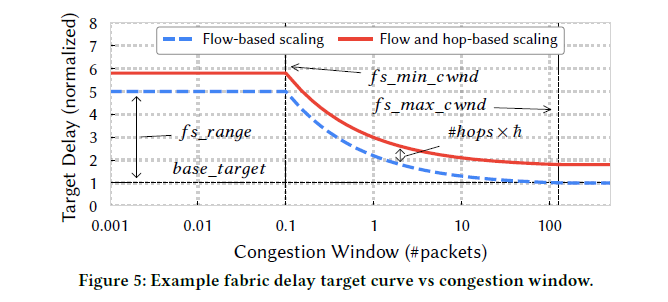
HPCC使用参考配置如下：baseRTT为链路基本RTT，expectedFlows为瓶颈链路最多并发流条数。需要打开交换机上的HPCC配置开关。



## 7.SWIFT

SWIFT是谷歌在TIMELY的基础上改进的基于delay的window-based CC。它在窗口大于1时使用基于窗口的速率控制，在窗口小于1时使用基于速率的速率控制，这样允许窗口大小最低降到0.001个包。它同样将delay分为主机拥塞和网络拥塞两部分，然后从两者计算出的窗口中选较小值，但由于OMNET++模拟器没有模拟主机网络栈时延，所以两者相同。它的delay是一个target delay而不是使用梯度，target delay的大小是基于baseRTT然后根据拓扑和流的规模进行扩展，具体来说跳数越多越大，竞争流数量越多越大，但由于发送端不知道竞争流的具体数量，所以改为与成正比，因为cwnd与竞争流数量成反比，具体由下图公式决定：



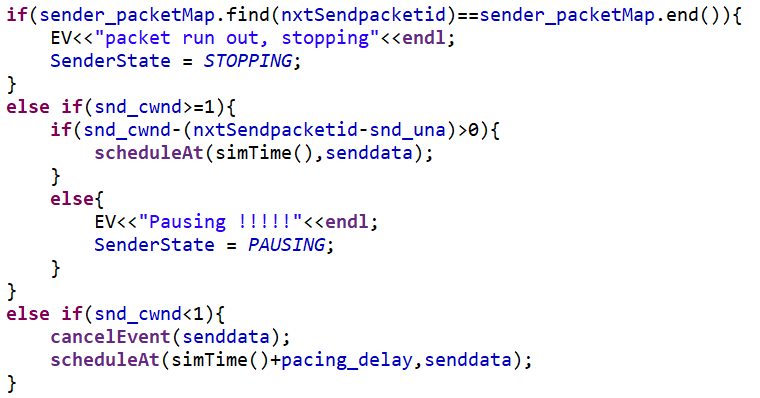


其中参数的值由上图决定，fs\_range为流扩展范围，一般为base\_target的倍数，默认最优配置为4，fs\_min\_cwnd和fs\_max\_cwnd可设为0.1和100（个包），而h可以设为1μs或一跳链路延迟。

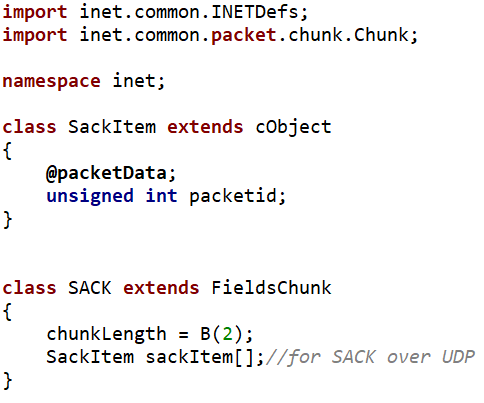
同时SWIFT的环境不是无损的，它还启用了丢包恢复机制，SACK快恢复和超时重传。

### 实现

首先我们还是在UDP侧使用窗口机制，不过这次改为了滑动窗口，其次还需要为其中加入SACK快速恢复与超时重传。



SACK基本的逻辑是，接收端收到乱序数据包后就将其插入sacks\_array中，然后清理重复和已被接收的包，然后发送SACK。SACK包含实际收到还未连续的所有包sacks\_array，最多填写最新的4个，而接收端当前期望收到的包rcv\_nxt以packetid的形式返回。发送端收到SACK会记录未重复的缺失的包到sacks\_array\_snd，累计三个SACK就会开启快速恢复重传，同时用变量pre\_snd记录当前的发送点，将缺失的包重传一遍后继续从pre\_snd开始发送。为了在UDP侧传递这个信息，我们在Messages/SWIFT里定义了新的字段SACK。

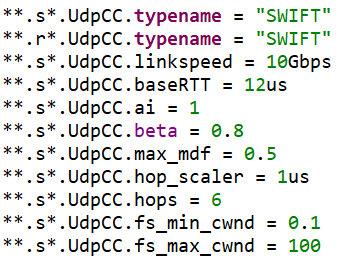


超时重传在send\_data()发送一个包后设置，初始值设了0.1s，可以覆盖大多数情况下的RTT。而收到一个ack则会取消超时重传的RTO计算则放在receive\_ack()中，计算方法如下网址所示，INET中也是用的这个计算方法，<https://blog.csdn.net/jxh_123/article/details/27345151>。此外send\_data()函数改为了不带packetid参数的形式，因为在设置超时时不知道超时后应该传哪个包。

然后是SWIFT自己的CC算法，需要注意的是SWIFT的窗口是以包个数为单位，与HPCC可以使用长度作单位不同。

### 使用

SWIFT使用参考配置如下，SWIFT的参数配置灵活性很高，因为大多数参数没有给出明确要求。首先是计算target delay部分的参数，baseRTT、每跳修正时间hop\_scaler、跳数hops、流扩展最小拥塞窗口fs\_min\_cwnd和流扩展最大拥塞窗口fs\_max\_cwnd。然后是速率计算算法的参数每RTT增速步长ai、计算乘性减系数的系数beta和最大乘性减系数max\_mdf。

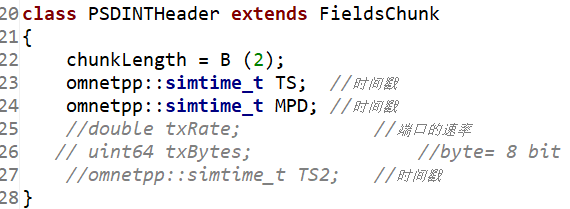


## 8.POSEIDON

Poseidon是NSDI’23的一篇基于INT的工作。它提出INT应该只对真正的瓶颈链路起反应，以降低开销、提升公平性。大致设计是把SWIFT的算法与INT携带的最大每跳延迟（maximum per-hop delay，MPD）相结合。

### 实现

Poseidon的INT头只有2B，我们用来存储MPD和用来计算MPD的上一跳时间戳。但是发现2B的INT头无法通过打开enableImplicitChunkSerialization的方式在仿真中点开，只有长度达到8B以上才可以，或者数据包IP头的morefragments标识由于IP层进行了切分而置为真也可以。两种情况都不符合Poseidon的实用状况，所以暂时保留Poseidon在仿真中点开的功能。



算法上大部分是与SWIFT相似的设计，配上他自己的计算target的函数。Poseidon对于速率控制算法抛弃了传统的AIMD，提出了一种更加高效的单调速率控制，是基于对数函数来实现的。这个函数计算target时需要用到实时速率currentRate，



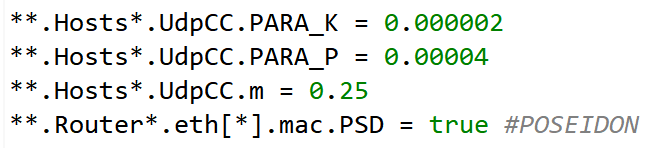
对于currentRate的计算我们发现不能直接使用拥塞窗口snd\_cwnd去除以baseRTT或者currentRTT，因为这两者都不能准确反映实际运行时的发包间隔，还是得实际测量，因此我们在Poseidon中加入了测速功能。在以时间和以包个数衡量测量间隔之间，我们选择了以时间maxInterval衡量，因为基于窗口的CC可能多个包同时发送。重传机制方面，由于它没有特别提到使用哪种特殊的ACK，所以我们只保留了超时重传及收到重复ACK的快恢复。

在交换机上 ，每一跳会根据上一跳时间戳来计算per-hop delay，如果是目前最大的则写入MPD。

### 使用

Poseidon的主要参数有最小目标延迟PARA\_K（默认为2），避免网络不充分利用，K越高RTT越高网络利用率也越高；目标延迟scaling系数PARA\_P（默认为40），P越高RTT越高速率抖动越小；计算利用率的系数m（默认为0.25），m越高收敛时间越长速率抖动越小；和最小乘性减系数min\_md（默认为0.5）。

参考配置如下：原文中PARA\_K和PARA\_P单位为微秒，因此是小数，除此之外基本没有需要灵活配置的参数。（记得配置baseRTT）

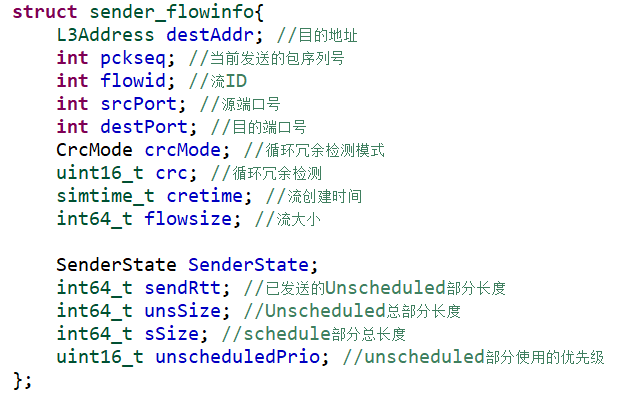


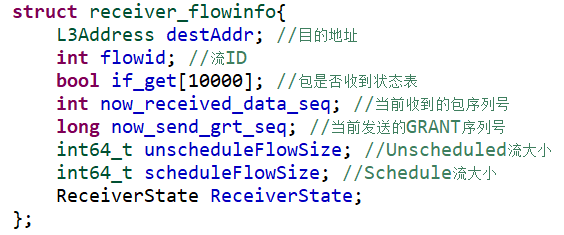
## 9.HOMA

HOMA是SIGCOMM’18提出的一篇信用预约式的CC，与XPASS类型相同。它提出无脑传输短消息：任何一条流到达时都会直接发送其unscheduled包（长度为RTTbytes即一个BDP）。所以可能出现同时发送多条流的情况，sender\_StateMap需要改为流id索引，因此可以直接写到sender\_flowinfo中。

### 实现

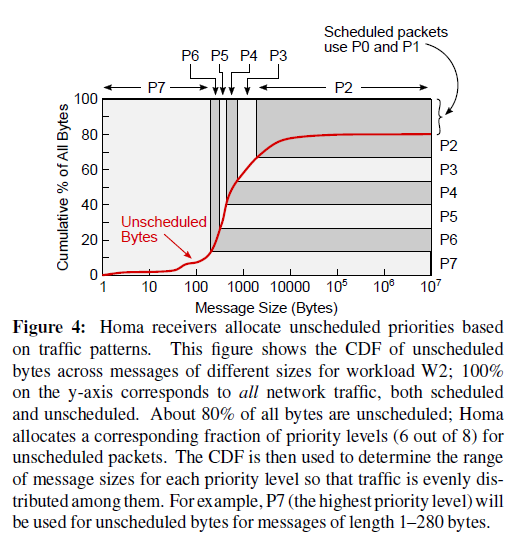
主要数据结构如下：





HOMA的流程是：发送端收到一条流先发送其一个BDP的长度，然后剩下部分等待GRANT包的确认再发送。接收端收到一个包就发送一个GRANT包，以此保证链路上总有一个BDP的包。直到收到流的最后一个包就停止发送。对于scheduled包，还使用了超时重传机制，超时未收到会发送resend包，而发送端收到resend包会重传对应包，如果繁忙则发送busy包。

此外，HOMA在最后一跳使用优先级，将不同的流隔离开来，以保证短流优先。对于发送端，unscheduled包使用高优先级，接收端根据unscheduled包携带的消息长度，通过GRANT包给scheduled包指定优先级，同样是短流优先，且保证每条消息使用不同的优先级。使每个优先级分配到的数据量大致相等。



### 使用

由于HOMA本身没有自带拥塞控制算法，所以除了baseRTT没有参数需要配置。

## 10.BFC

BFC（NSDI 2022）是一种不同于端到端拥塞控制的协议，采用一种每跳每流的流量控制，结合交换机内有限的状态信息、恒定的交换机处理时间，以及虚拟化设计交换机内存空间和队列结构（小拓扑下设计32条虚拟队列，大拓扑下设计128条虚拟队列）。BFC实现接近最佳吞吐量和极低的队尾延迟，即使在高网络负载和INCAST交叉流量场景下，依然效果良好。BFC的主要设计是在链路层的虚拟队列改造，以及链路层和网络层之间的一个以太网封装模块（EthernetEncapsulation模块），细节将在链路层队列和HiEthernetEncapsulation模块中介绍。

### 实现

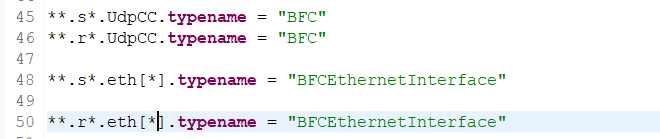
BFC在传输层的操作是简单的发送数据包和接收数据包处理。在传输层简单处理来自上层APP传来的数据流，为数据流添加流号flowid、当前速率currentRate、目标速率targetRate、包序号pckseq等等信息用于确定某条流和属于该流的数据包。

### 使用

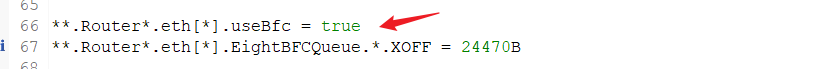
BFC在传输层模块发送数据包时，为数据包添加必要的tag信息，保证数据包在交换机内能够被准确识别是某条流的报文。

BFC在接收数据包时，正常接收，没有其他控制模块，或者触发操作。

在.ini文件中，将主机的CC配置为BFC，eth端口配置为BFCEthernetInterface.



并开启useBfc功能，用于触发BFC机制的暂停帧功能。



## 11.BCN

BCN在BFC基础上改进，主要功能是在交换机队列中添加了触发快速发送CNP速率控制报文传回发送方的操作。同时，相对于BFC的32或128条虚拟队列设计，BCN采用类似PFC的8条虚拟队列，并且在Classifier分类器中，根据虚拟队列的排队长度，优先选择短排队长度的队列，为新到达的数据包随机分配新的队列号，通过该操作保证每条队列长度均匀。此外，在Scheduler调度器中，按照轮询的方式公平地调度每条队列，进一步保证每条队列得到公平的机会被及时调度，避免队列长时间排队（相对于PFC按照严格优先级机制调度，低优先级队列可能长时间得不到调度，导致队列饿死，BCN轮询调度更公平有效）。

### 实现

BCN在传输层的设计除了简单的收发报文，主要添加的功能是对于接收到CNP控制报文的处理。

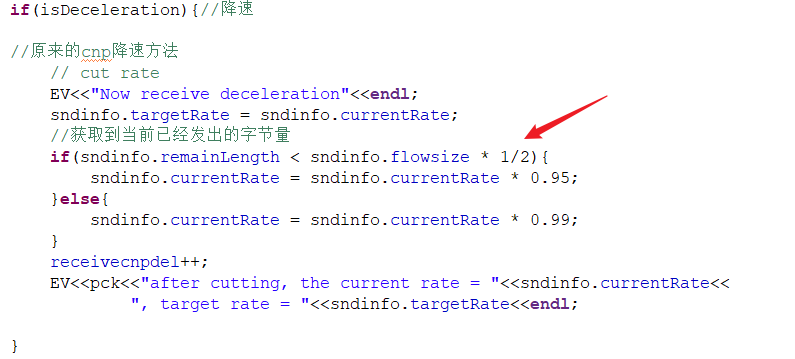
BCN摒弃端到端拥塞通知，采用在交换机处发送网络状态通知的方式。当队列中某条流的第一个数据包导致队列长度大于阈值时，将该数据包标记为isPause数据包，并发送CNP降速通知包；当isPause数据包排出队列时，将发送CNP升速通知包。

发送端收到网络状态通知包后，精准调节指定数据流的发送速率，具体过程如下图所示。其中，降速参数de\_1,de\_2和升速参数sp\_1,sp\_2的设计在BCN原文的第V章节的A部分进行参数有效性分析。

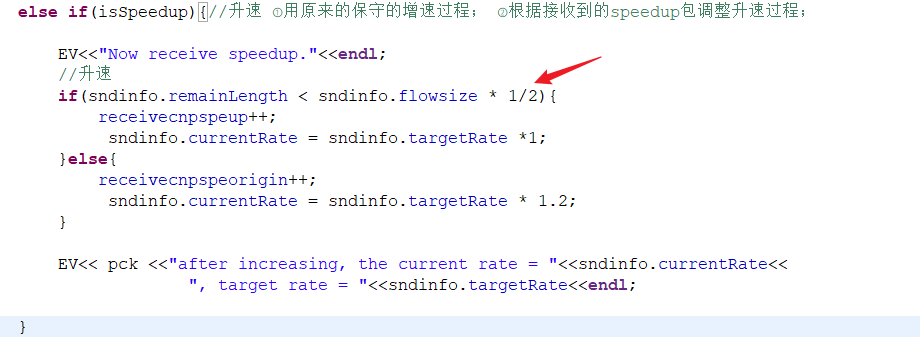


### 使用

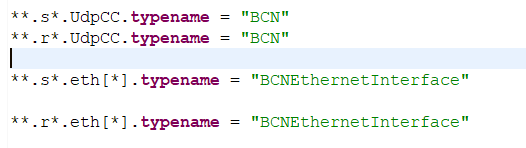
BCN在传输层的主要参数有降速参数de\_1,de\_2和升速参数sp\_1,sp\_2。当某条拥塞流的剩余字节数小于总字节数1/2时，将该流的下一次发送速率降为当前速率的de\_2倍；当某条拥塞流的剩余字节数大于总字节数1/2时，将该流的下一次发送速率降为当前速率的de\_1倍。



当某条拥塞流的isPause数据包排出队列时，说明该拥塞流拥塞情况已经得到缓解，因此触发发送升速CNP控制报文。当升速CNP报文达到发送方时，识别该流的剩余字节数小于总字节数1/2时，将该流的下一次发送速率升为当前速率的sp\_2倍；当某条拥塞流的剩余字节数大于总字节数1/2时，将该流的下一次发送速率降为当前速率的sp\_1倍。



在.ini文件中，将主机的CC配置为BCN，eth端口配置为BCNEthernetInterface.

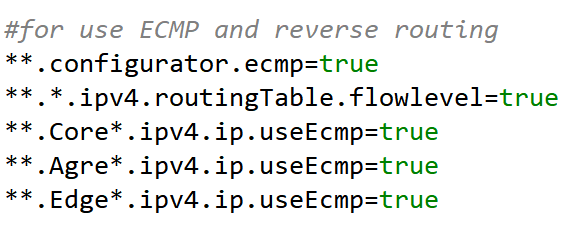


同样的，BCN的暂停机制与BFC一致，开启useBfc功能，用于触发BFC机制的暂停帧功能。



# 三、网络层

网络层实现的功能为ECMP与反向路由，ECMP是数据中心重要的负载均衡机制，因为数据中心常为对称的规则拓扑，会有多条等价的路径；而反向路由则是信用协议所必需的条件。ECMP目前多以流级别进行多路径传输，因为包级别的多路径与TCP等协议兼容不好，所以我们实现了两种方式可供选择。所有这些功能的使用不需要详细配置，只需要打开相应的开关，如下图所示。



本来打算将这些功能与其它功能一样作为单独模块移植出来实现，但由于当前版本INET网络层框架的基类不具备单独实现的基础（即默认查找路由的函数findBestMatchingRoute(Ipv4Address &)只能传目的地址这一个参数，无法传入包等信息），所以会需要将基类复制出来，进而会牵涉到整个框架导致改动小复制量大的情况，所以最终决定采取修改原生代码并在移植的时候覆盖的形式。

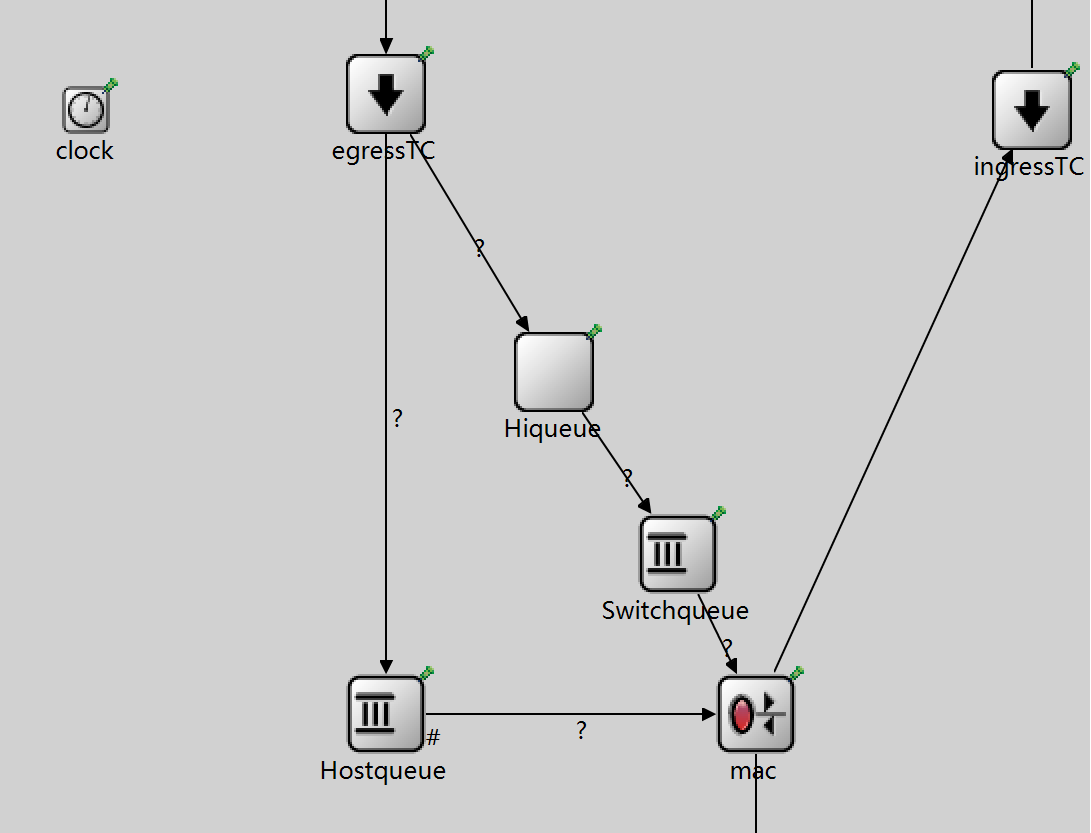
实现ECMP的思路是：在Ipv4NetworkConfigurator的第12初始化阶段，INISTAGE\_NETWORK\_CONFIGURATION，添加静态路由的函数Ipv4NetworkConfigurator::addStaticRoutes()中，将INET框架计算拓扑内到某一结点的最短路径的Topology::calculateWeightedSingleShortestPathsTo()函数，修改为计算所有的最短路径而不是只算一条。然后在遍历任意一对节点之间的路径时，将所有这些路径都进行递归，就可以得到所有的等价最短路径。然后Ipv4NetworkConfigurator模块的ecmp选项开启时，会将符合ECMP的路由信息代替正常路由信息存入每个节点的路由表结构中。到了Ipv4NodeNetworkConfigurator的第15初始化阶段，INISTAGE\_STATIC\_ROUTING，再在Ipv4RoutingTable::internalAddRoute()写入每个节点的路由表routes的同时，使用map<Ipv4Address, vector<Ipv4Route \*>> ecmpIFs记录去往相同目的地的等价路由。

我们在Ipv4::routeUnicastPacket()中添加了查找ECMP路由的函数findecmpRoute()，相对于默认的findBestMatchingRoute()函数新增了源地址、目的地址、包和端口号等参数，声明于IIpv4RoutingTable.h中，定义于Ipv4RoutingTable.cc中。在选择等价路由时，我们选择模N哈希的方法，即哈希完包头信息模等价接口数取余。而由于反向路由是信用协议独有的需求，所以我们不再设计一个查找反向路由的函数，而是在信用协议的数据包上打上反向路由标记来触发。我们在查找ECMP路由之后就会将包id和对应源端口号存入map<long, int> reverseIFCache，以供查找。

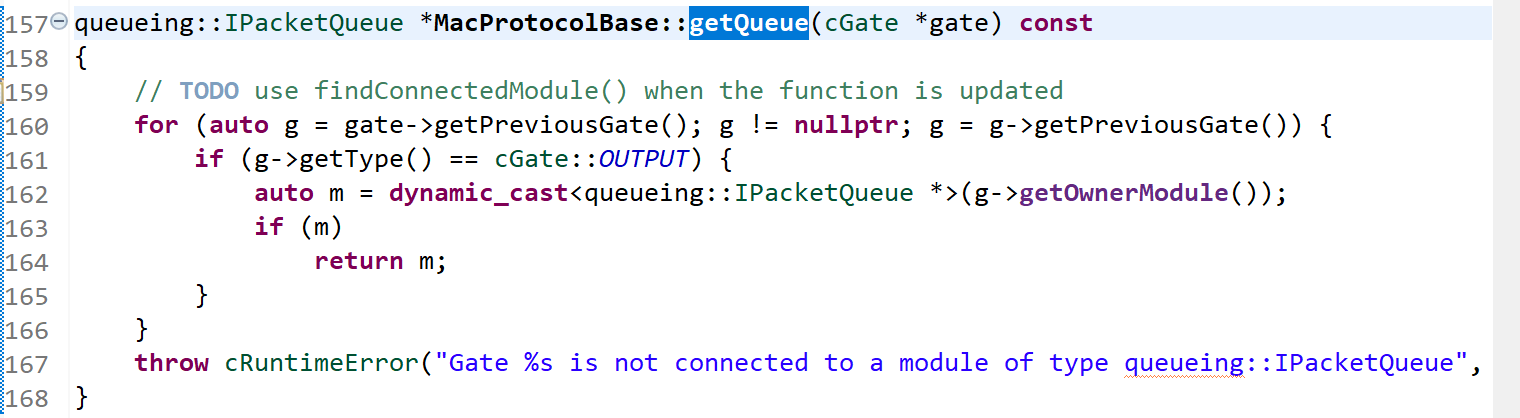
最终需要覆盖的文件有inet/networklayer/ipv4：Ipv4.cc/h/ned，Ipv4RoutingTable.cc/h/ned， IIpv4RoutingTable.h；inet/networklayer/configurator/ipv4：Ipv4NetworkConfigurator.cc/h/ned；common：Topology.cc。

# 四、链路层

链路层实现的功能有数据测量与各种功能队列，在链路层的eth模块处使用我们定义的HiEthernetInterface模块即可，架构如下图所示：



INET4.4版本对于队列模型相关的模块的输入输出门赋予了主动收(ActiveSink)主动发(ActiveSource)被动收(PassiveSink)被动发(PassiveSource)的属性，并定义了四种属性的基础模块接口（IActiveSink、IActiveSource、IPassiveSink、IPassiveSource），相互连接的两个输入输出门必须为互补的属性（比如主动发->被动收）才能进行传输，同一个门可以具备多种属性。由于INET4.4版本的mac模块将buffer外置，并要求mac的上一个模块必须为IPacketQueue类型的队列，而继承自IPacketQueue的模块又都不具备主动收属性，无法在其之前加入scheduler等模块 ，所以需要作出修改。

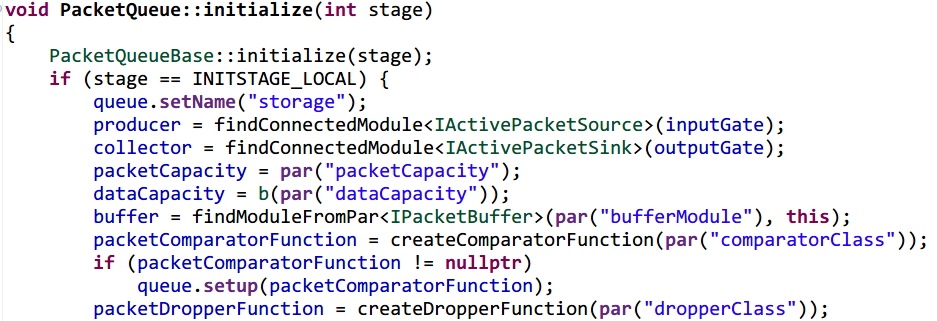


HiQueue模块中部署了我们的功能队列，也是我们实际的缓冲区.

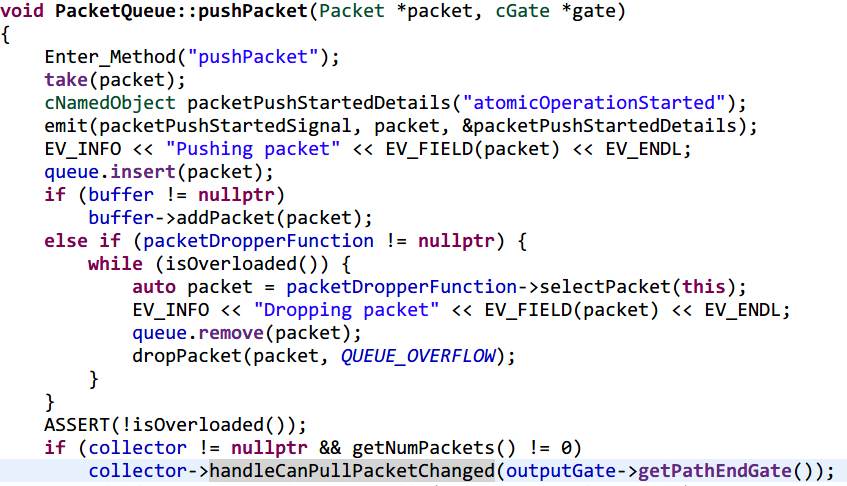
## 1.功能队列的构建

一个功能队列往往基于INET中自带的简单队列构建而成，就像在INET4.0中基于common/queue/PassiveQueue。由于INET从4.2版本开始删除了common/queue文件夹将其内容移至新增的queueing文件夹，并且进行了大量修改与补充，所以现在我们的功能队列往往基于queueing/queue/PacketQueue构建。

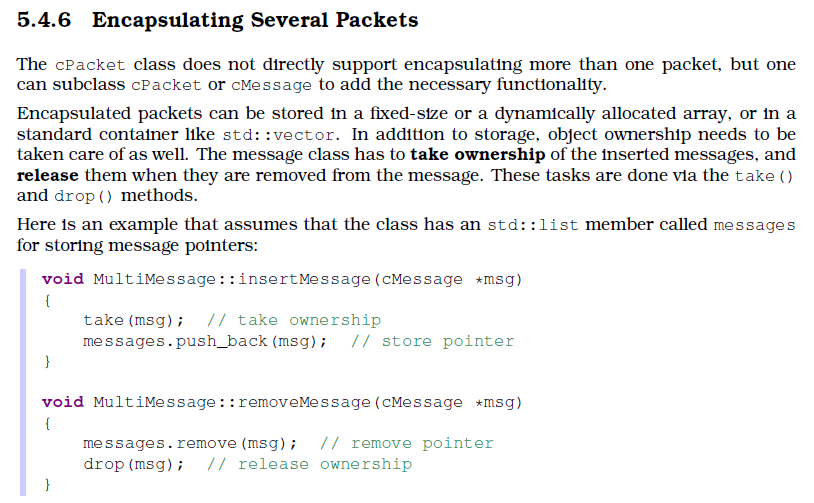
PacketQueue是一个被动收被动发属性的队列，需要上下游分别是主动发和主动收的门，主动发名为producer，主动收名为collector，被动发名为provider，被动收名为consumer。此外还需要配置队列容量、丢包算法、队列内排序算法、外置缓冲区等。



主要的功能函数包括pushPacket()、pullPacket()、canPushSomePacket()、用来创建丢包和排序算法的createDropperFunction()、createComparatorFunction()等。每当有数据包从上游进入时就会触发handleMessage()，然后调用到pushPacket()，然后触发下一个主动收模块的handleCanPullPacketChanged()通知其上游有数据包可以被拉取了。



出队列时则是调用pullPacket()。在包进出队列的时候分别有一个take(packet)和drop(packet)需要声明，以确保包的所有权不会出现重复。

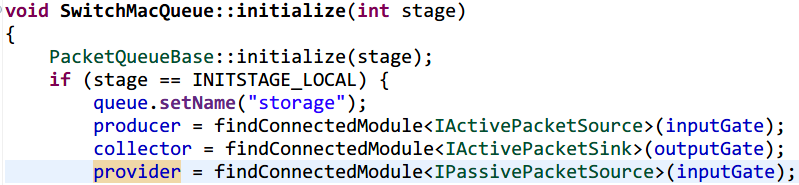


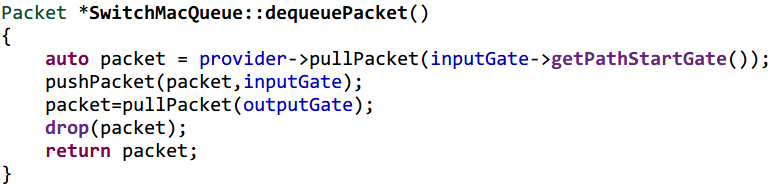
在修改的一开始，首先要注意queueing文件夹下的package.ned文件定义了一个新的命名空间inet::queueing用于防止重名，即在这个文件夹内的文件才属于这个命名空间，在之外的模块可以与之内的重名。我们的代码一般会需要用到这个命名空间内的模块但又不会定义到这个命名空间内而是定义在自己的文件夹，所以需要使用**using** **namespace** inet::queueing;再去继承基类，而不能直接在namespace queueing{}中定义新的队列类，或直接继承queueing::基类名。

丢包算法如果不配置会在队列满时报错，排序算法如果不配置那么默认为FIFO。所以使用中一般默认配置丢包算法为队尾丢包，**string** dropperClass = **default**("inet::queueing::PacketAtCollectionEndDropper");。

## 2.SwitchMacQueue

这种功能队列是为了满足mac模块外置缓冲区的要求而设计的。我们分为两条情况，主机和交换机，可以通过节点ned中的isSwitch选项来区分。对于主机一般没有复杂的队列设计，所以直接使用最基础的PacketQueue即可满足要求；而对于交换机往往会需要队列设计，所以我们基于PacketQueue修改出一个可以传递拉取包信号的SwitchMacQueue用于满足mac模块的要求。

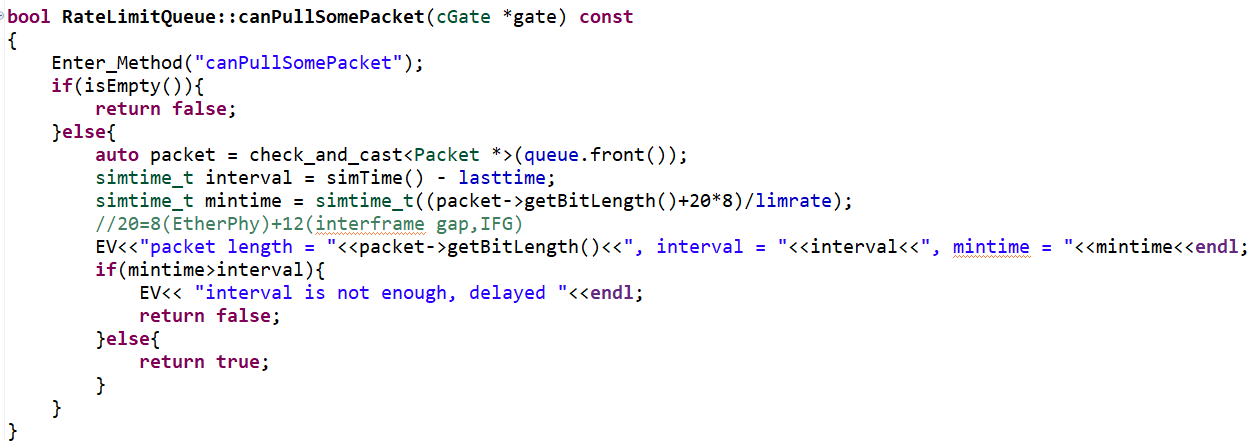
具体来说，我们为SwitchMacQueue的输入门添加了主动收的属性，

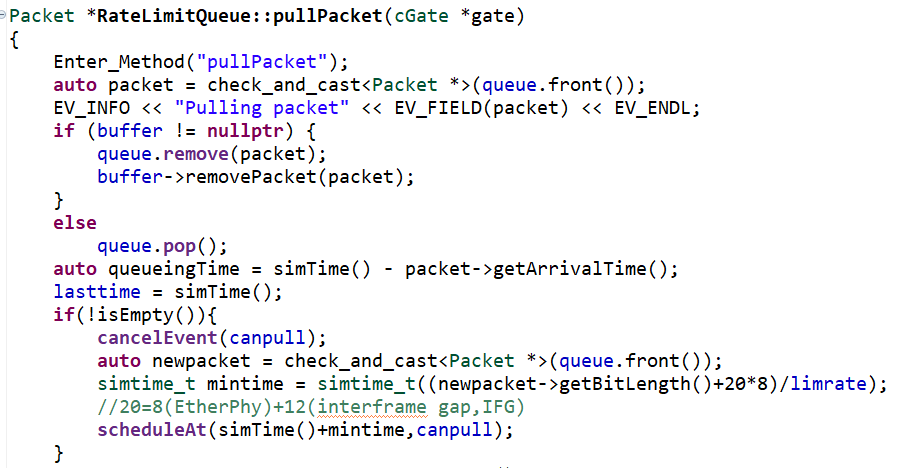
使队列组可以将出包的信息通过handleCanPullPacketChanged()传递到mac模块，再将mac模块的拉取信息通过SwitchMacQueue中override的dequeuePacket()传回scheduler，同时将SwitchMacQueue的pushPacket()和pullPacket()都放入dequeuePacket()，

删除SwitchMacQueue的pushPacket()触发handleCanPullPacketChanged()功能，mac模块本身的根据链路传输状态连续拉取包的信息则通过SwitchMacQueue中override的canPullsomePacket()传回scheduler，达到将SwitchMacQueue架空只作为一个通道的目的。

## 3.RateLimitQueue

RateLimit功能通过复写canPullsomePacket()函数和pullPacket()函数实现，为此需要为此队列配置限制速率limrate。

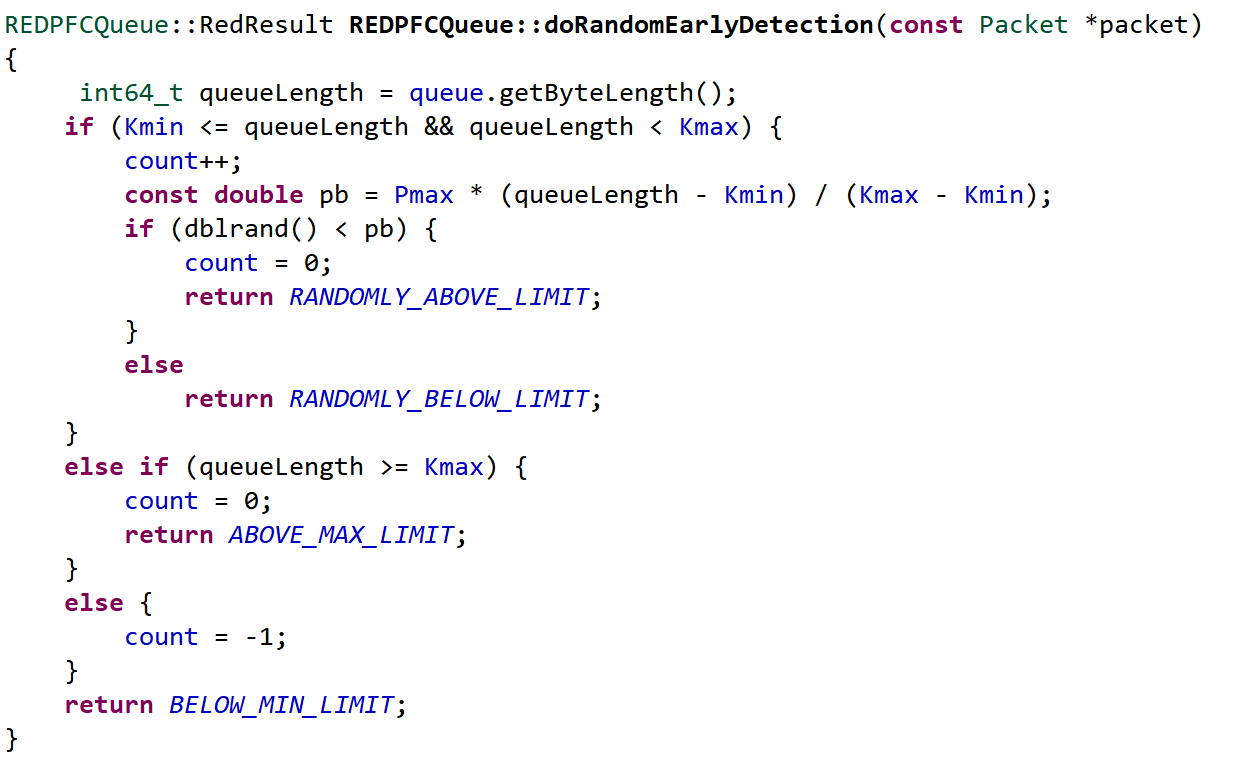






## 4.ECN打标

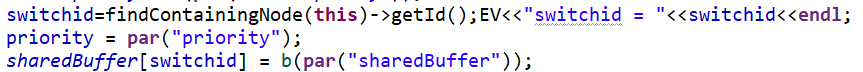
ECN功能在出队时pullPacket()测量，普通ECN队列使用单阈值打标，RED-ECN使用如下RED机制打标。



## 5.共享缓冲区

现实中交换机通常采用共享缓存的方式部署，每条队列有一部分自己的保留缓冲区，以及和其它端口共享的缓冲区。然后使用DT（dynamic threshold，动态阈值）的方法来分配每条队列的总容量，即：每条队列的maxSize为剩余共享缓存的α倍，α一般为2的整次方（可正可负）。

我们采取全局变量的方法，对于每个队列类，在原本的dataCapacity之外定义了一个静态变量sharedBuffer数组，与全局流ID的实现方法一致。这里由于共享缓冲区需要以交换机为单位配置，所以我们采用了数组来隔离，下标即每个交换机的编号，在配置时将交换机内同类队列的这个数组共同设置为交换机总共享缓冲区的值就行，即sharedBuffer[switch id]=xMB。



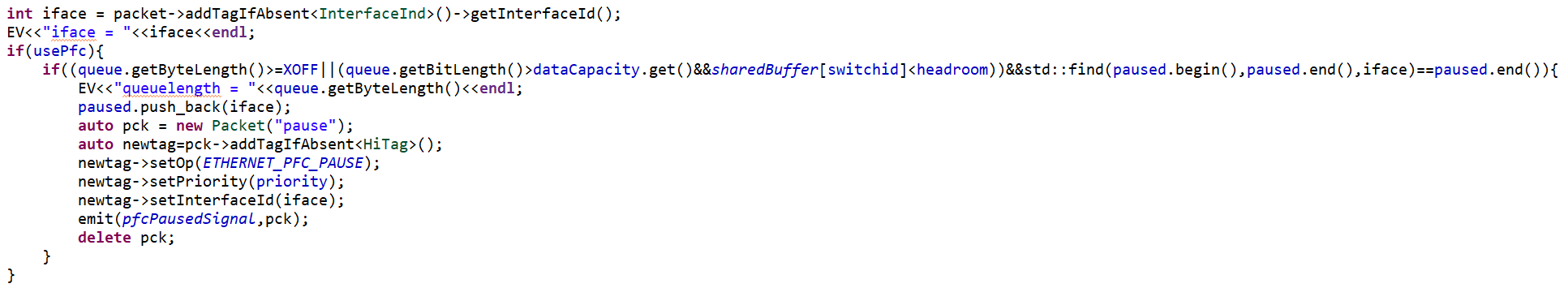
原本的dataCapacity即队列的保留缓冲区，在其满了之后，便会从共享缓冲区中继续拿取缓存空间，直到达到最大值maxSize。

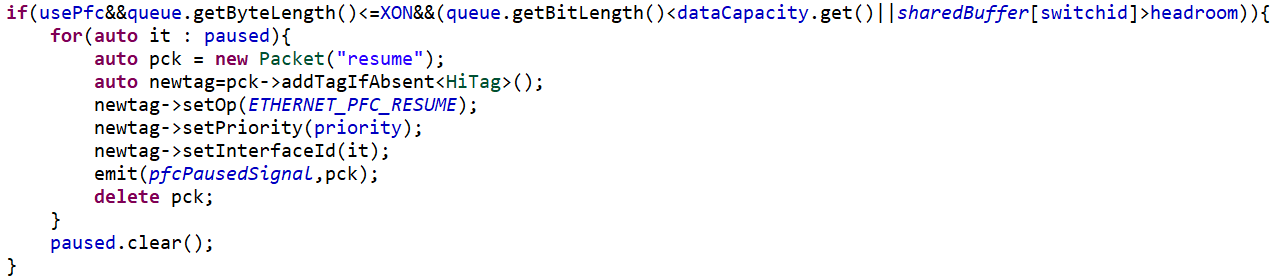
目前共享缓冲区功能主要部署于REDPFCQueue和ABMQueue，我们通过在handleMessage()时进入BufferManagement()函数来实现这个功能。需要注意的是，实现了共享缓冲区的队列不能再使用原生定义的statistic信号queuelength和queuebitlength来测量队列长度，否则在使用共享的缓存时会出现队列长度为负的情况，因此我们新定义了两个cOutVector来存储队列长度相关数据，名为“queuelength (bit)”和“sharedbuffer (bit)”。

## 6.PFC与信号机制的使用

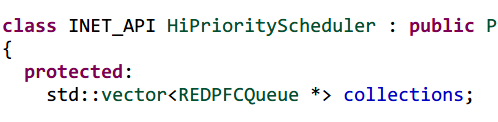
PFC功能的实现思路是 ：队列达到阈值后发送信号给对应端口mac模块，信号携带包的入接口号、优先级、操作码信息，mac模块收到后遵循物理层发送间隔往上游发送对应的操作帧，上一跳的mac模块收到后发送信号给自己接口中的HiPriorityScheduler模块，对对应优先级队列进行操作。

在队列内，我们主要是在pushPacket()中添加了发送pause帧的功能，根据是否达到单队列的阈值XOFF或在使用共享缓冲区的情况下是否还有一个headroom（大小为上一跳的所有BDP），以及此端口是否已经被暂停来决定是否发送；以及在pullPacket()中添加了发送resume帧的功能，在队列长度小于XON时，向所有已暂停端口发送resume帧。



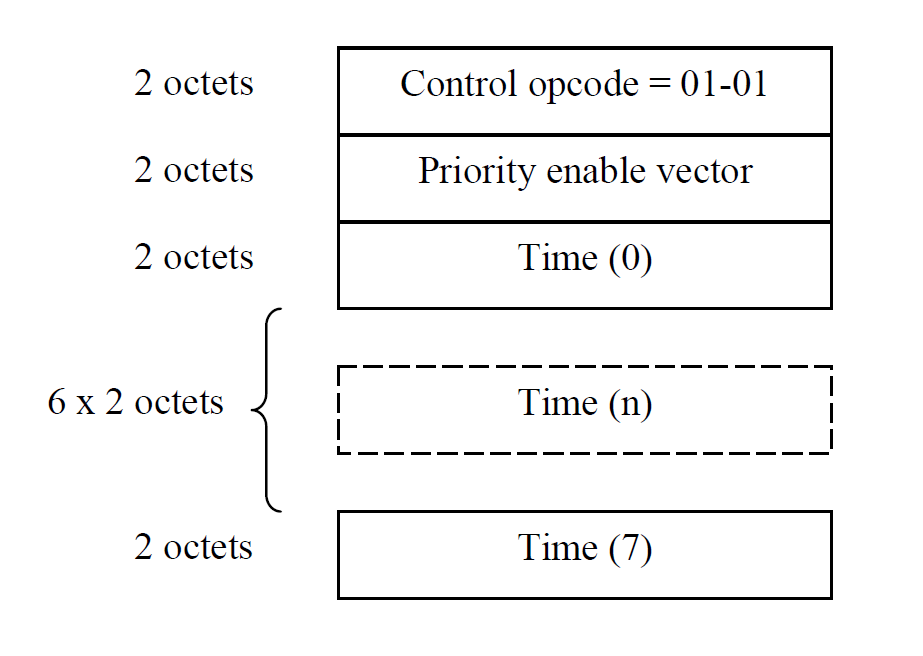


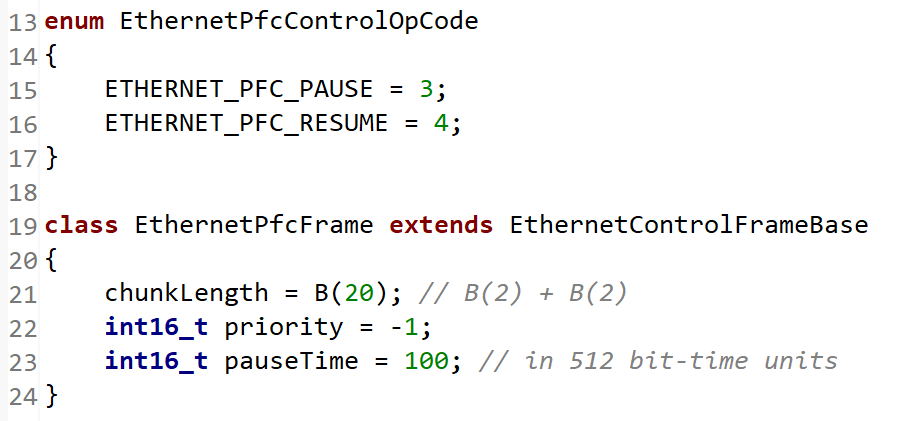
而HiPriorityScheduler模块是专门为PFC设计的调度器，并且与REFPFCQueue直接绑定使用。

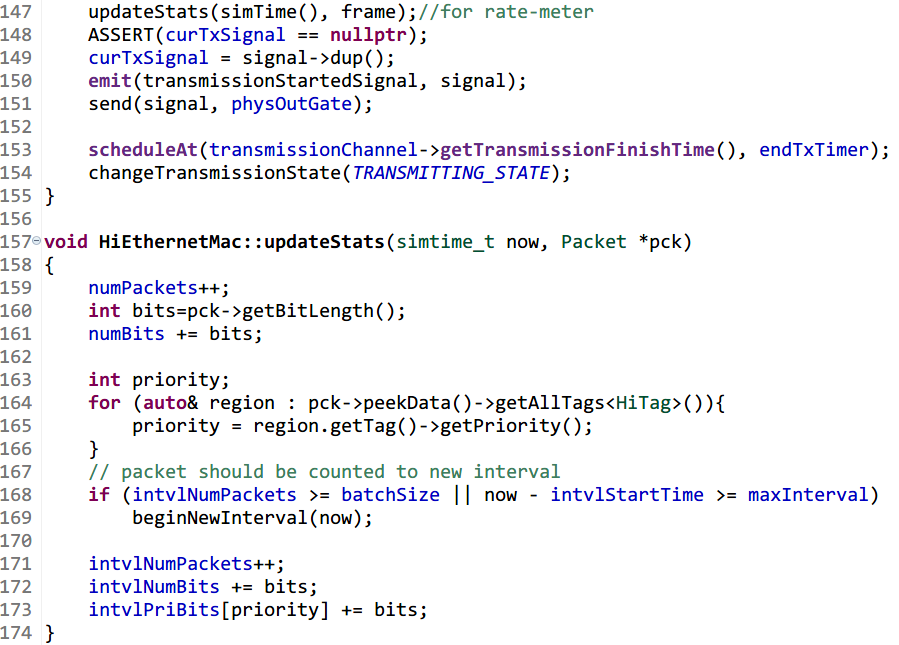


信号(signal)机制的使用：信号需要有发送模块，并由接收模块接收。信号拥有信号NAME和信号ID，并且可以携带一个对象用于传递。首先在发送模块ned文件中声明其NAME以及携带对象的类型，**@signal**[pfcPaused](type=inet::Packet);，然后在.h文件中声明其ID，**static** simsignal\_t *pfcPausedSignal*;，再在.cc文件中定义，simsignal\_t *PFCQueue::pfcPausedSignal* = cComponent::**registerSignal**("pfcPaused");，将两者绑定起来。最后在需要使用的地方将信号发射出去emit(*pfcPausedSignal*,pck);。接收模块需要是cListener的子类，接收时，在initialize()中订阅信号radioModule->**subscribe**(PFCQueue::*pfcPausedSignal*,**this**);，subscribe()的作用是为一个信号登记一个listener，listener一般为模块本身，即this，而第一个参数可以使用信号ID也可以使用信号NAME。而调用subscribe()的模块则决定了接收信号的“高度”，因为信号是随着模块层级往上传，比如我们在mac模块中接收主机中所有PFCQueue传来的信号，意味着需要把高度往上调到host级别，可以使用getParentModule()，需要两次。最后就可以使用receiveSignal()处理接收到的信号。

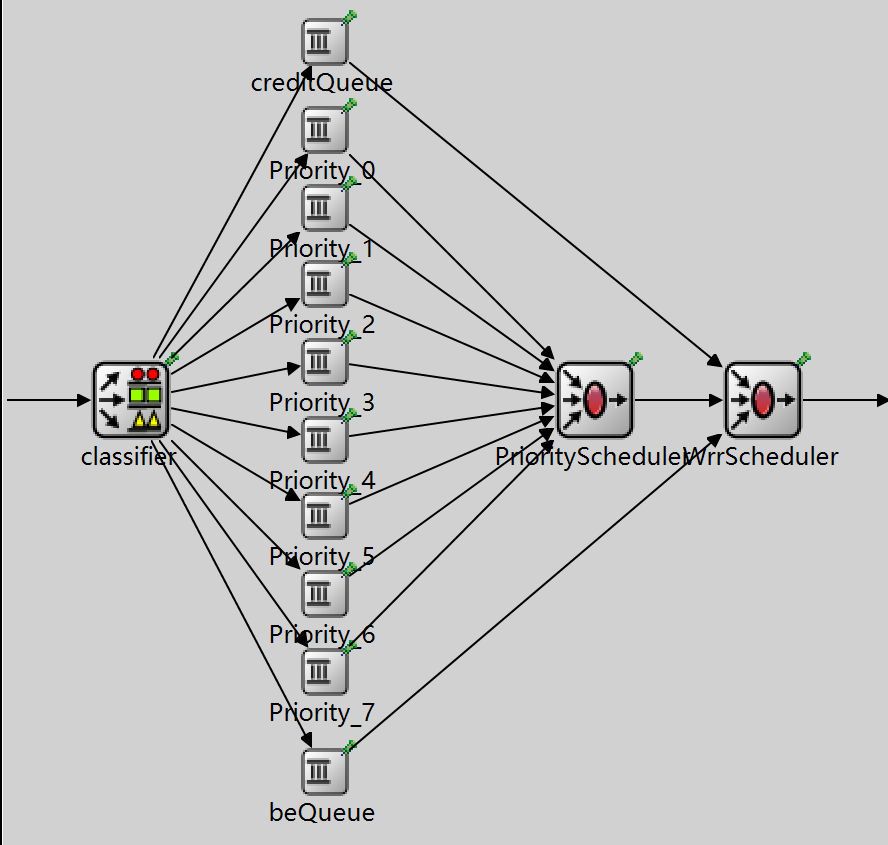
我们使用msg文件构造PFC帧，PFC帧结构如下图所示，pausetime字段我们一般不使用，因为会有RESUME帧，但还是按照规定的20B的长度来构造。





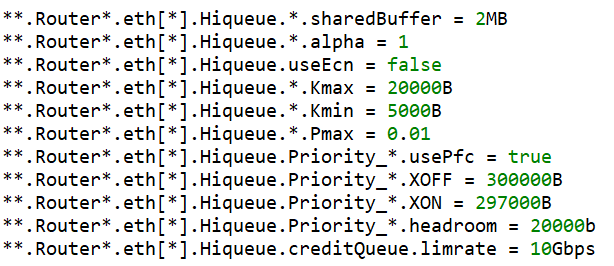


## 7.HiQueue构造与默认配置



默认状态下(HiQueue.ned)部署PFC的八个优先级队列REDPFCQueue，一个普通队列以及一个信用队列，其中信用队列使用RateLimitQueue，基于PacketQueue做了一个简单的限速功能；普通队列也使用REDPFCQueue，只是配置Pmax=0或Kmin=Kmax；而每一个优先级队列都具有PFC和RED-ECN功能。首先我们定义了一个classifer将包分类，名称为credit进入信用队列，tag中设置了进入优先级队列，否则进入默认队列。然后我们使用一个优先级调度器HiPriorityScheduler来调度优先级队列，并使用一个权重轮询调度器WrrScheduler来调度整体队列，并在WrrScheduler中override了pullPacket()使用bufferOccupancyVector测量缓冲区占用，输出数据“bufferOccupancy(byte)”。

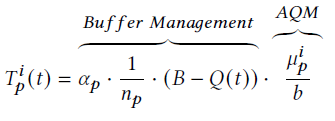
默认配置： 单交换机共享缓冲区大小16MB，每条队列保留缓冲区大小20KB，信用队列672B（八个信用包大小，for XPASS）。在使用时，PFC的阈值配置要注意大于ECN完全触发阈值，XOFF与XON之间一般为两个MTU，headroom为上一跳所有BDP之和。



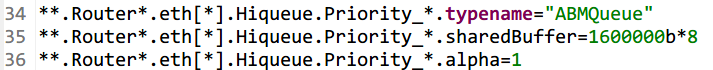
## 8.ABMQueue

ABM是一种结合了动态阈值DT的缓冲区管理策略，要实现ABM首先需要实现共享缓冲区。为此我们继承REDPFCQueue来构建ABMQueue，覆写BufferManagement()函数。

ABM将DT计算当前队列的最大容量的方法替换为如下公式，所以我们还需要拿到当前优先级的排出速率。我们通过在HiEthernetMac中新设置的数组intvlPriBits来记录不同优先级的排出比特数，计算出的排除速率存放在数组pribitpersec。

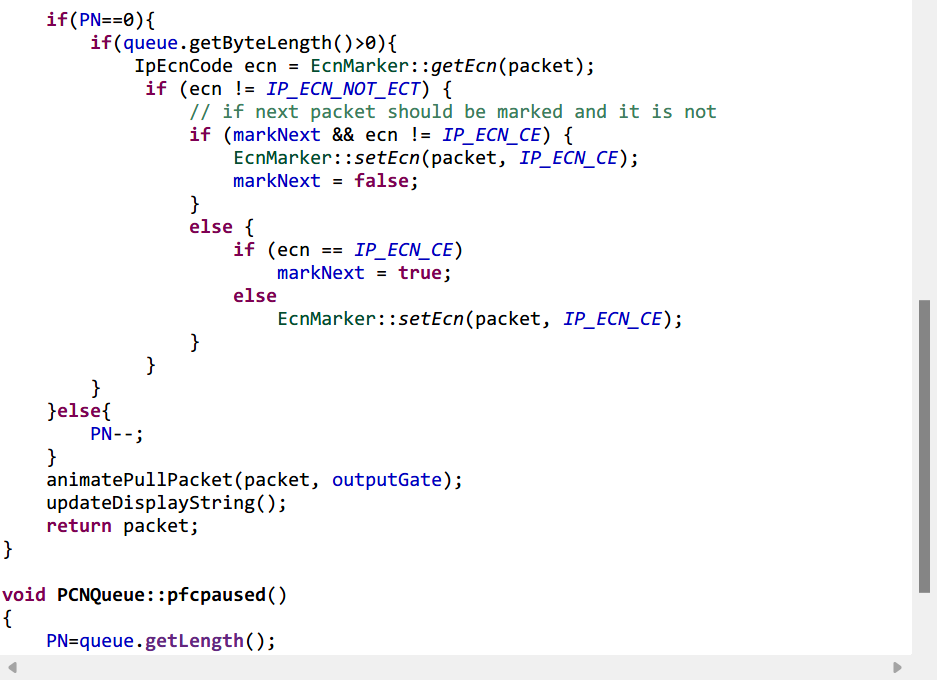
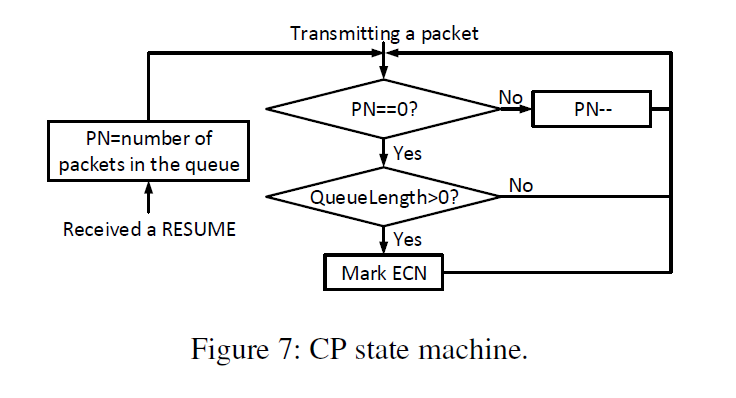


参考使用配置如下：

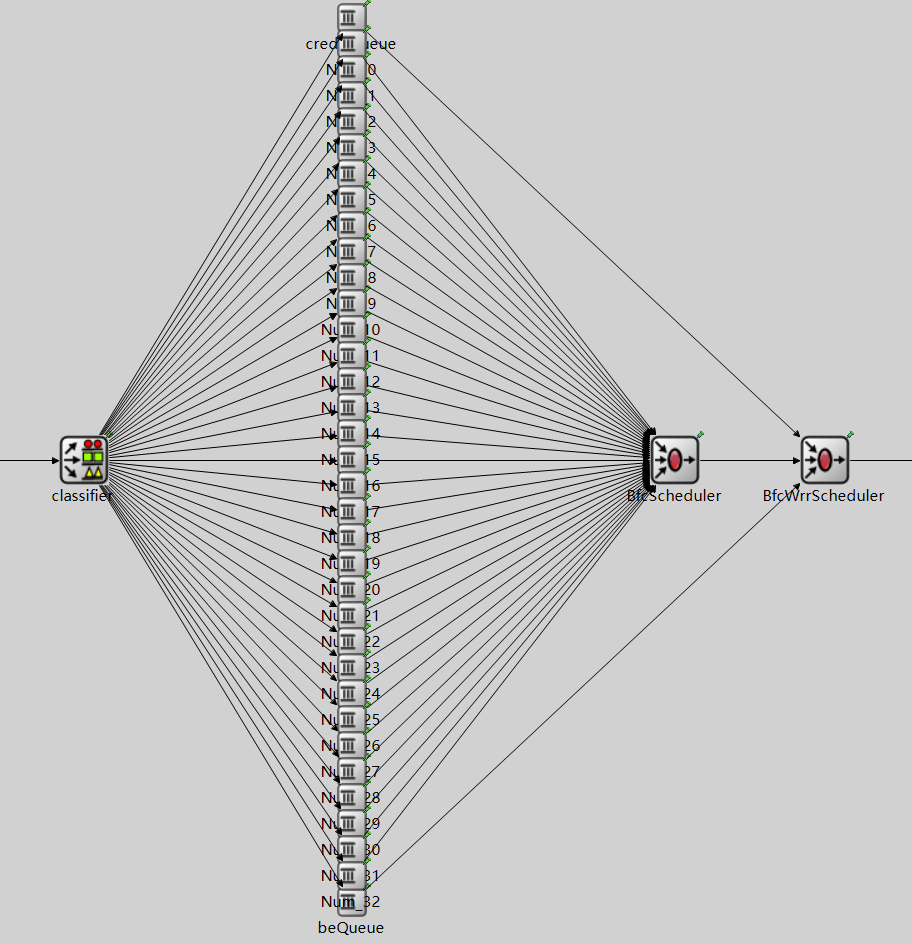


## 9.PCNQueue

PCN的交换机设计需要在被暂停时记录队列中剩余包数量，然后在这些包排除之前停止标记ECN。因为背景有对PFC的需求，所以我们同样继承REDPFCQueue进行修改。在这里需要修改的是RED的ECN标记方式。



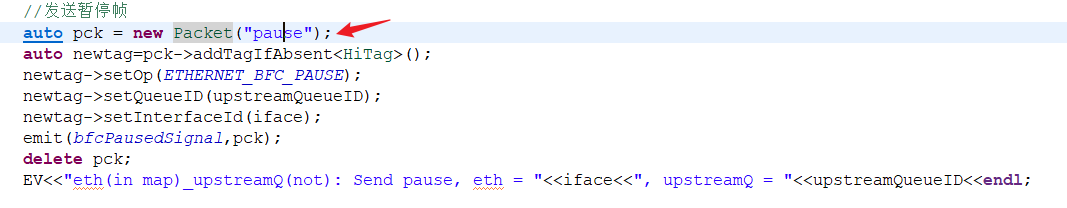
## 10.BFCQueue构造与队列设计



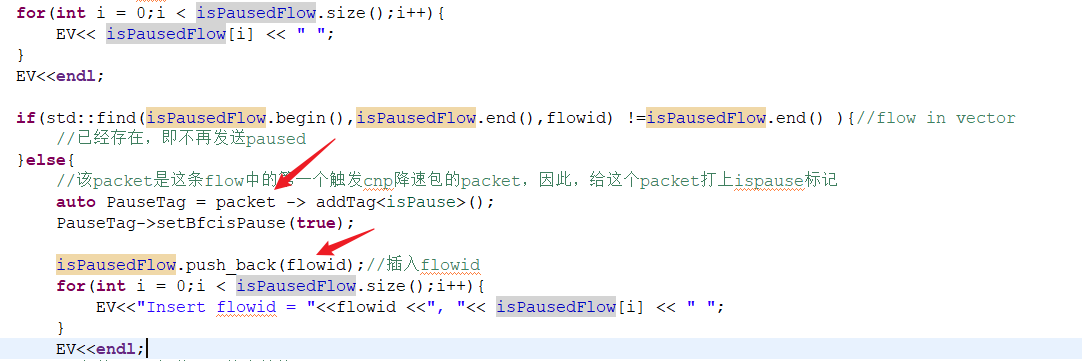
在小拓扑结构中，BFCQueue采用了32个虚拟BFCqueue队列设计。数据包入队列后，检查当前队列长度，如果队列长度>XOFF，或者满足数据包字节量>队列字节容量和共享缓存区间，即会导致触发PAUSE帧。



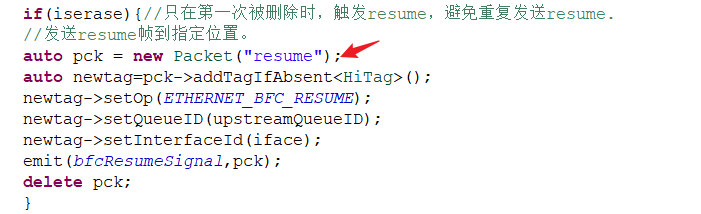
对于队列长度大于XOFF位置的数据包，在触发发送PAUSE帧时，需要查询Map<iface,upstreamQueueID>，检查该数据包对应的上一跳交换机的eth和queueid是否已经被暂停。如果已经存在于Map中，说明上游的端口队列已经暂停，因此不再触发PAUSE帧。



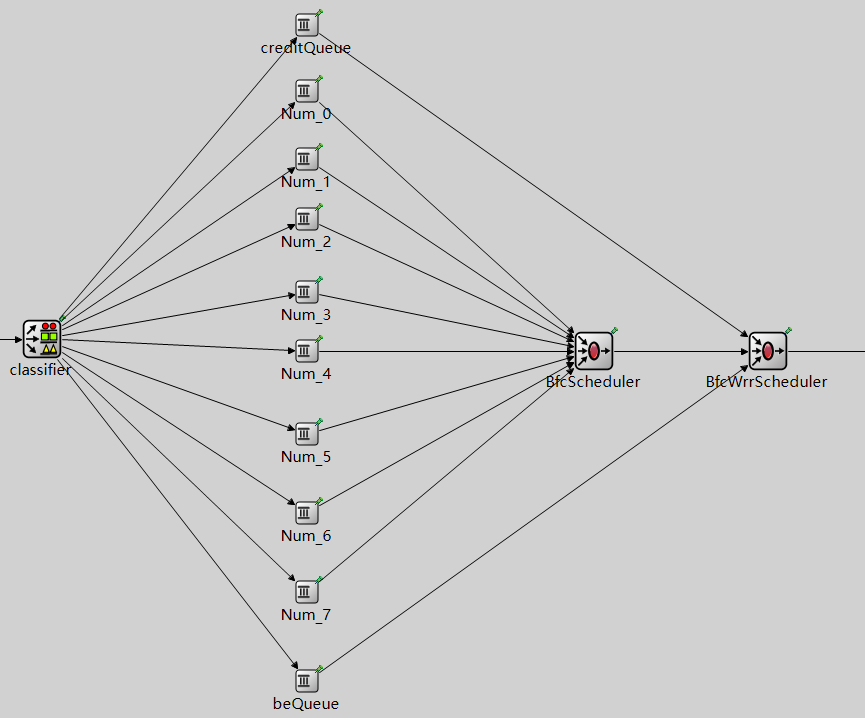
同时，如果该数据包是该条流中第一个触发暂停的数据包，则将该数据包标记为isPaused数据包，用于在排出队列时，触发发送RESUME帧。并且将该流号存入到isPausedFlow vector中。



在出队列时，如果是isPause数据包正在排出队列，检查该数据包的上游端口eth和队列号upstreamQid是否处于暂停状态。若<eth,upstreamQid>存在Map中，说明依然处于暂停状态，因此触发发送RESUME帧，恢复上游队列。



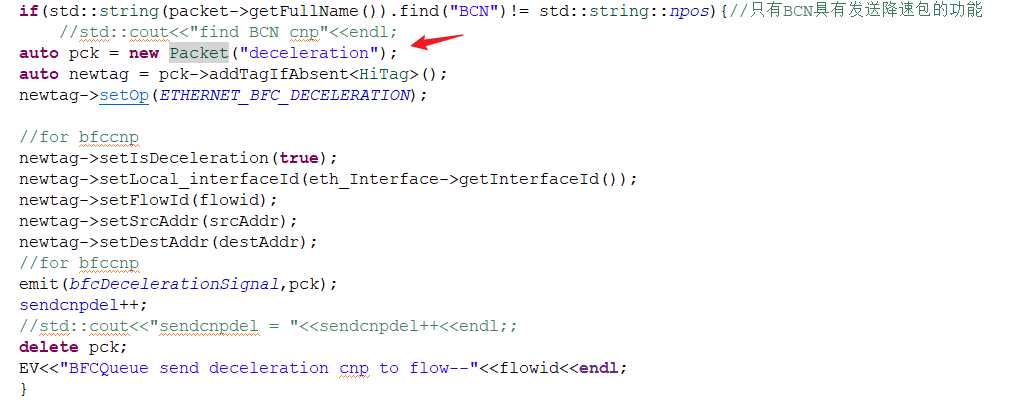
## 11.BCNQueue构造与队列设计



BCNQueue采用与PFC数量一致的8条虚拟BCNqueue队列设计，相对于BFCQueue降低了队列分配的难度，简化交换机Classifier分类操作。

BCNqueue队列延续BFCqueue数据包入队列和出队列的触发条件，同时增加了触发发送CNP降速和升速数据包的操作。具体如下。

在入队列时，当某条流的首个数据包的队列长度大于XOFF，除了会触发发送PAUSE帧，还会触发向发送方发送CNP降速数据包。

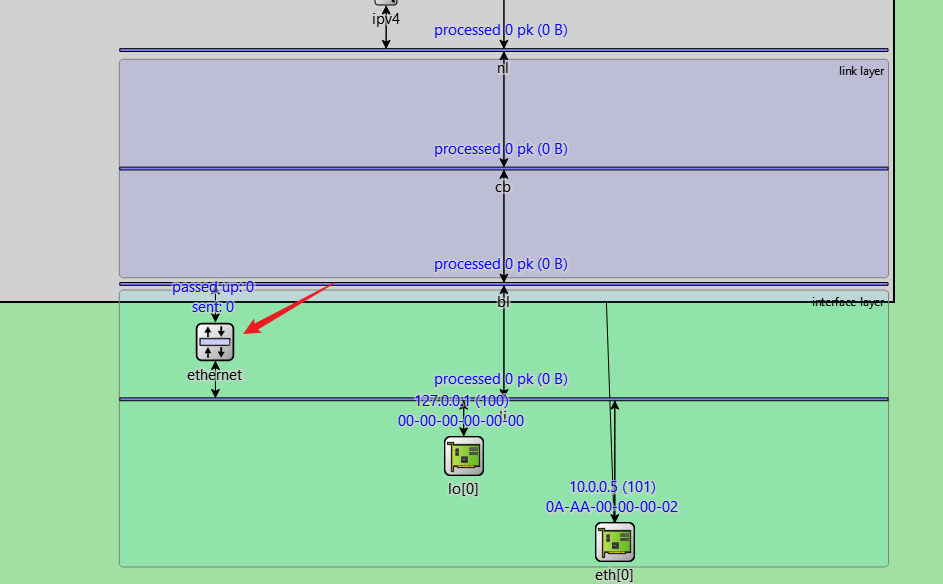


在出队列时，当某条流的isPause数据包出队列，除了会触发发送RESUME帧，还会触发向发送方发送CNP升速数据包。



## 12.HiEthernetEncapsulation模块

Ethernet封装层是处于IP层和链路层之间的模块，在inet4.4框架中的位置如下图所示。



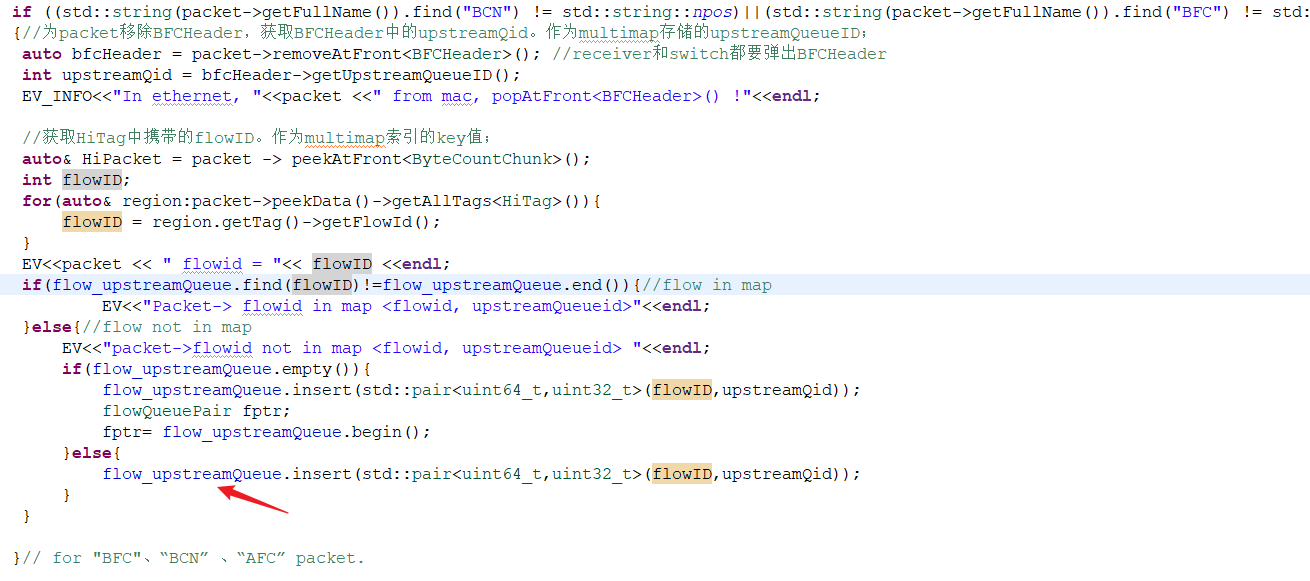
由于BFC、BCN的精准背压机制，需要根据当前排队的数据包中携带的信息，准确找到其携带的上一跳队列号，因此，需要在数据包头部添加BFCHeader/BCNHeader字段，用于保存该数据包的上一跳队列号和当前队列号。并且每经过一跳交换机都需要重新更新。

BFCHeader/BCNHeader字段，字段大小为8字节；该字段中包含两段信息，一段是当前的队列号QueueID，另一段是上游的队列号UpstreamQueueID。整个数据包各段的设计如下图所示。

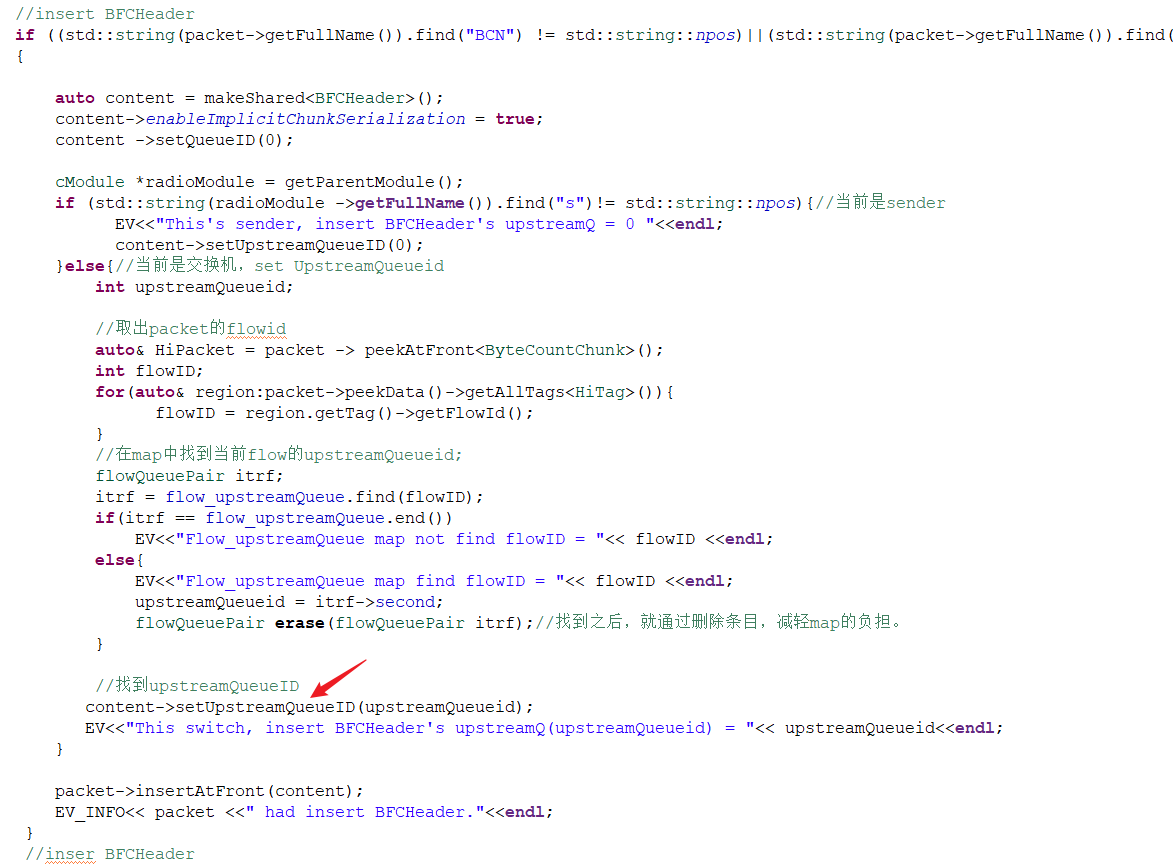


因此，在主机上的Ethernet模块，当上层IP层传来报文后，进行封装，BFCHeader/BCNHeader中QueueID=0，UpstreamQueueID=0。

该报文到达第一跳交换机时，继续向该交换机的上层传输，此时经过Ethernet模块，将报文中的flowid和UpstreamQueueID记录在Map<flowID,upstreamQid>中。然后，剥离该报文的BFCHeader/BCNHeader字段，之后继续向上层传输，传到IP层，再根据IPHeader携带的目的地址信息，进行IP路由到相应的端口。



报文经过IP层转发到对应端口后，再经过Ethernet模块继续向下层传输。此时，Ethernet再对报文添加BFCHeader/BCNHeader字段进行封装。在Map<flowID,upstreamQid>中查询报文携带的flowid号，在BFCHeader/BCNHeader字段中填写该流的upstreamQid。然后继续向下链路层传输。



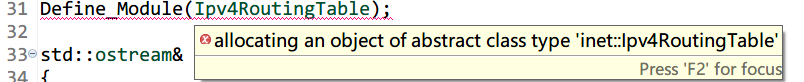
以上报文封装过程，对于IP层和链路层而言，都是透明传输过程，并不会影响IP路由和链路层端口的报文传输。

## 13.测速功能

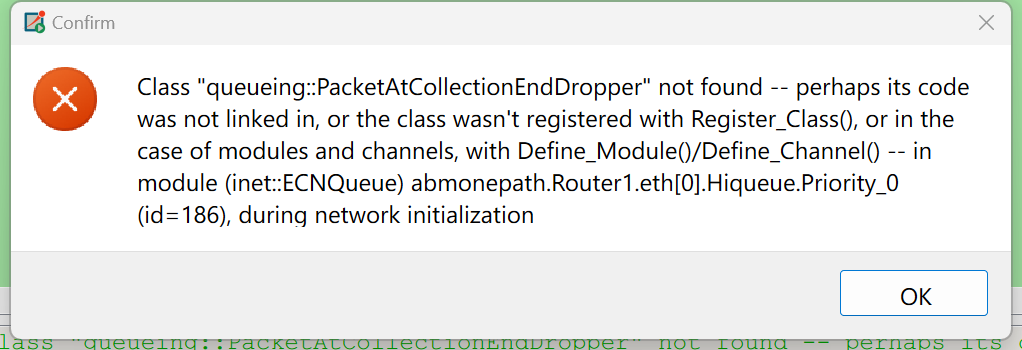
我们还在HiEthernetmac模块中实现测速功能，因为它位于数据包通路的出口，不会受buffer排队的影响。测速的原理是**速率=时间间隔通过的比特数/时间间隔**，我们使用变量intvlNumPackets和intvlNumBits记录通过的包个数与比特数，然后用名为bitpersecVector的向量来存储实时速率数据，数据属性名为“thruput (bit/sec)”。同样还有“packet/sec”、“total packets”、“total bits”、“avg throughput (bit/s)”、“avg packets/s”等数据输出。

# 五、常见错误与解决方法

1、纯虚函数未定义。纯虚函数是形如“virtual 函数类型 函数名 （参数列表）= 0”的函数，需要在后续派生类中赋予定义，不然派生类将因成为抽象类而无法被实例化。常见于：

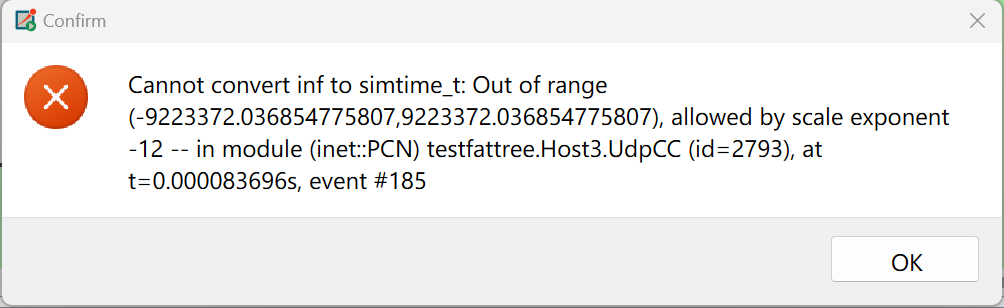


2、算法类未登记。常见于：



在确定类已存在且没有拼写错误的情况下考虑是否因为命名空间不同导致无法找到。

3、计算时缓冲区溢出。常见于发送间隔的计算：



由于发送间隔=包长度/速率，当速率足够小时发送间隔可能会上溢，对此可以用simtime\_t对结果进行类型转换。如果不是这个原因，则检查其它参数是否正确。

4.仿真完成后控制台报错：

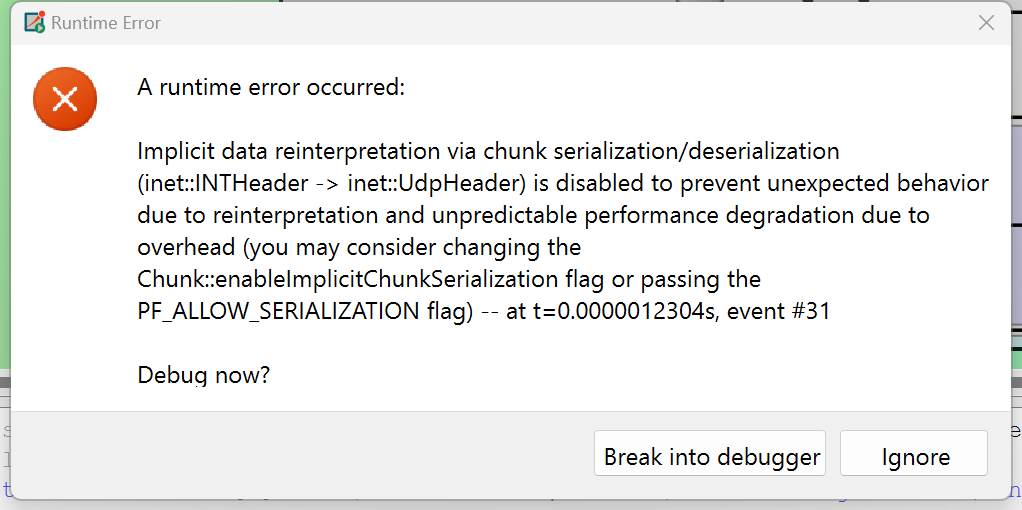


像这种没有说明具体地方的报错，考虑使用的类的析构函数是否有问题。或者使用Debug模式定位到精确的出错行。

5.分屏使用omnet时可能出现的仿真界面不弹出问题。

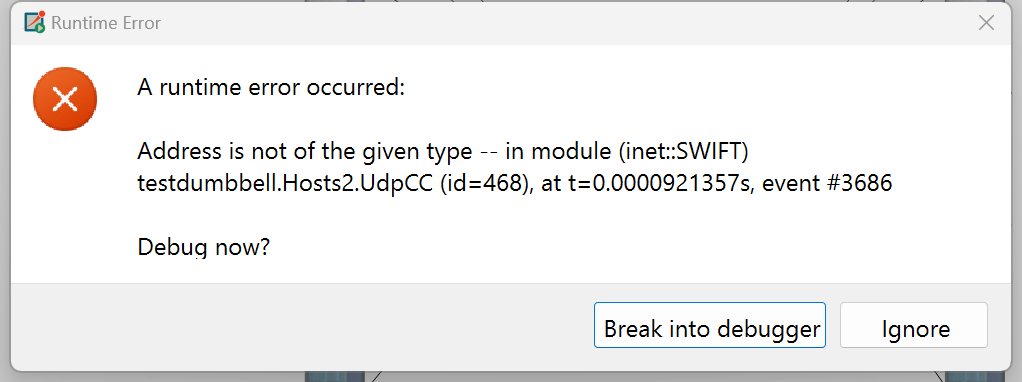
<https://www.cnblogs.com/gslgb/p/14262543.html>

1. 读取字段错误：



常见于拆包的时候没有按顺序读取，意思是此字段无法正常序列化。另一种情况是对我们自己写的FieldsChunk类型的字段如INT头、CNP包，在点击仿真界面中代表包的红点打开包的内容时，需要与此字段配套的序列化文件（serializer）和拆分文件（dissector），此外还需要将这个字段作为正式的协议注册到INET框架的Protocol.cc等文件中，比较繁杂一般不建议尝试。

1. Ipv4地址错误：



一般我们没有对Ipv4地址进行修改，所以常见原因是从Map中提取了一个不存在的包信息。

# 六、数据处理

这一章总结OMNeT++的数据导出以及用OriginPro等软件处理数据并画图的方法。

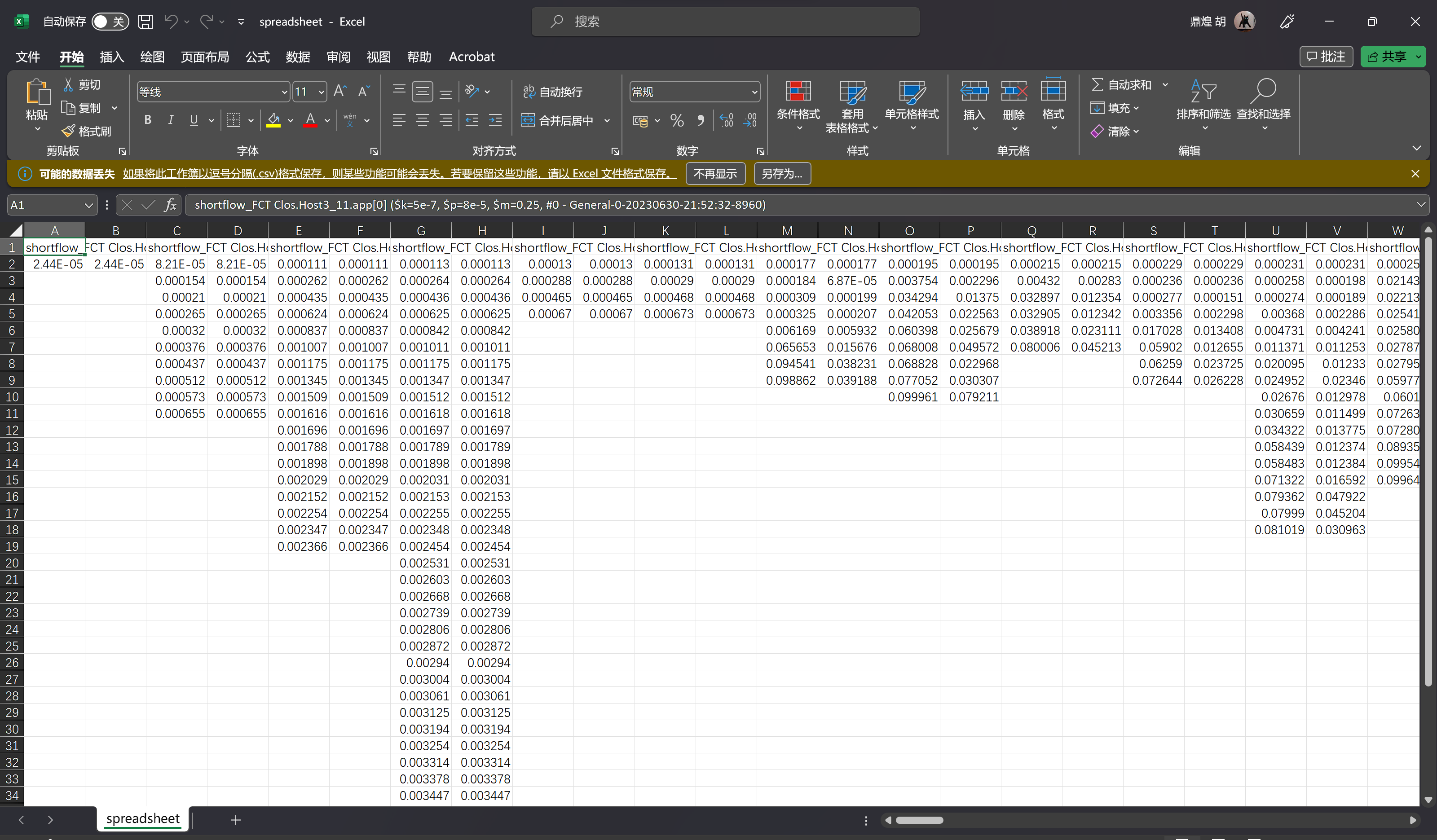
## 导出数据

OMNeT++的数据文件有三种类型，分别是vec、sca和vci，其中vec文件存储向量数据，即会随时间变化的数据；sca文件存储标量数据，即不随时间变化的数据；vci存放参数数据（在OMNeT++不能直接打开，需要用另外的文本编辑器打开）。在每次仿真开始时会重置此次要生成的对应文件名的数据文件，vec文件会随着仿真的进行写入数据（但不能在仿真时读取），而sca文件大部分要在仿真结束时调用finish函数再计算写入。保存了这三个文件即可保存这次实验的完整记录。

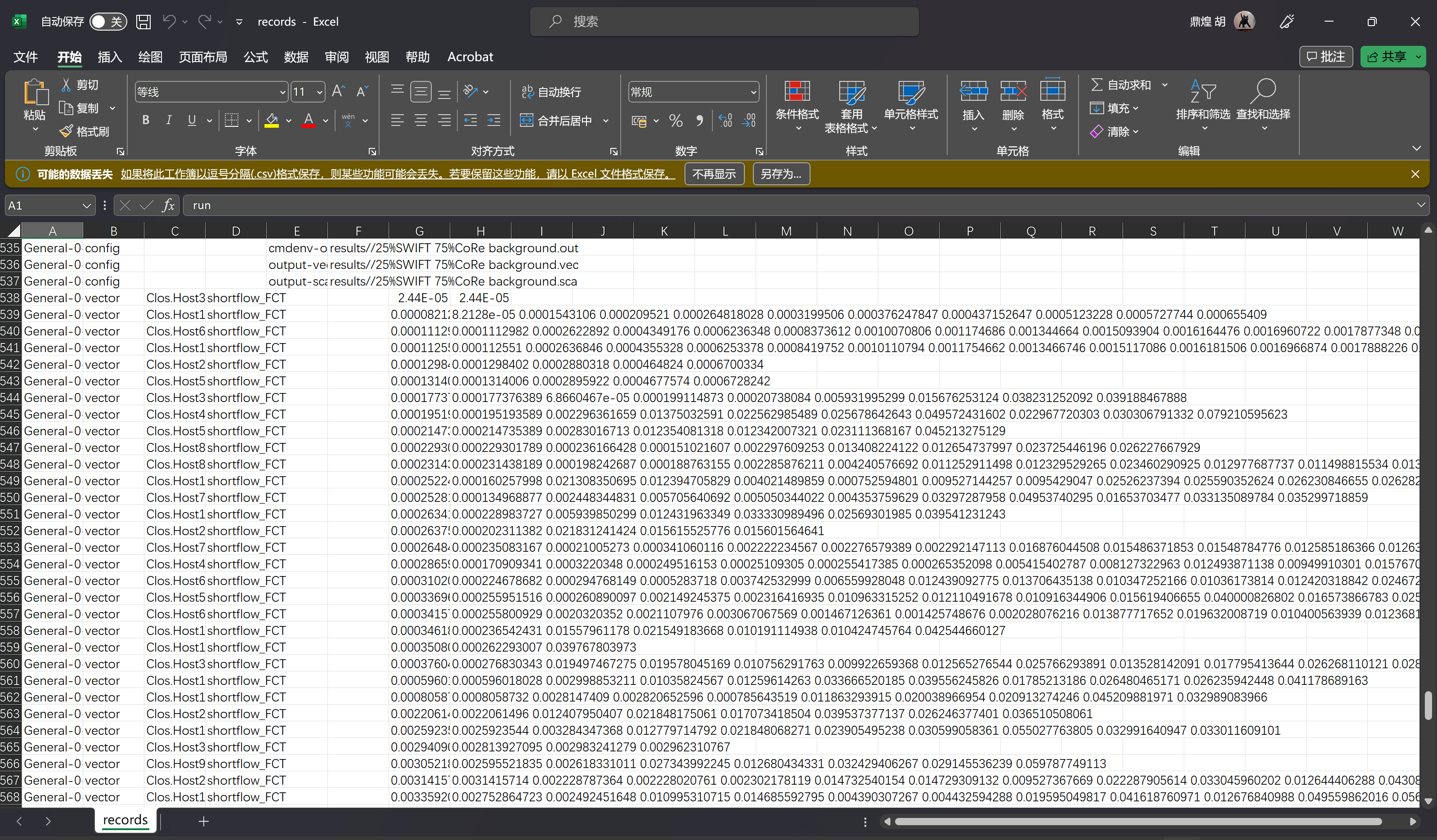
OMNeT++利用数据文件生成anf文件，可以在其中查看所包含数据文件的所有数据，并可以画出预览图。一般我们会需要导出向量数据，可以右键选中想要导出的部分，在Export Data选项下有**CSV for Records**以及**CSV for Spreadsheets**两种选择，分别适合不同的情况。



**Spreadsheets：**

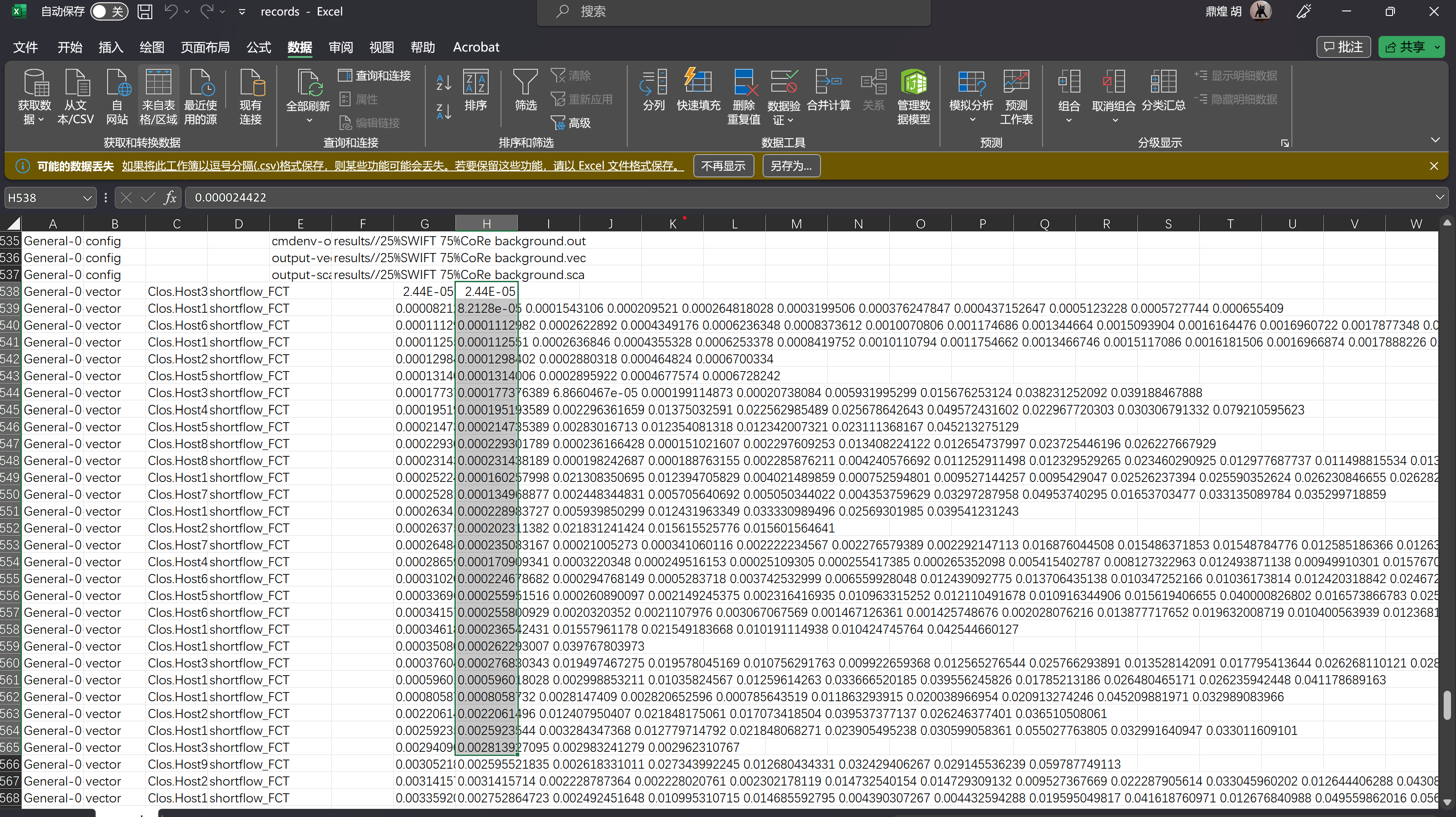


**Records：**

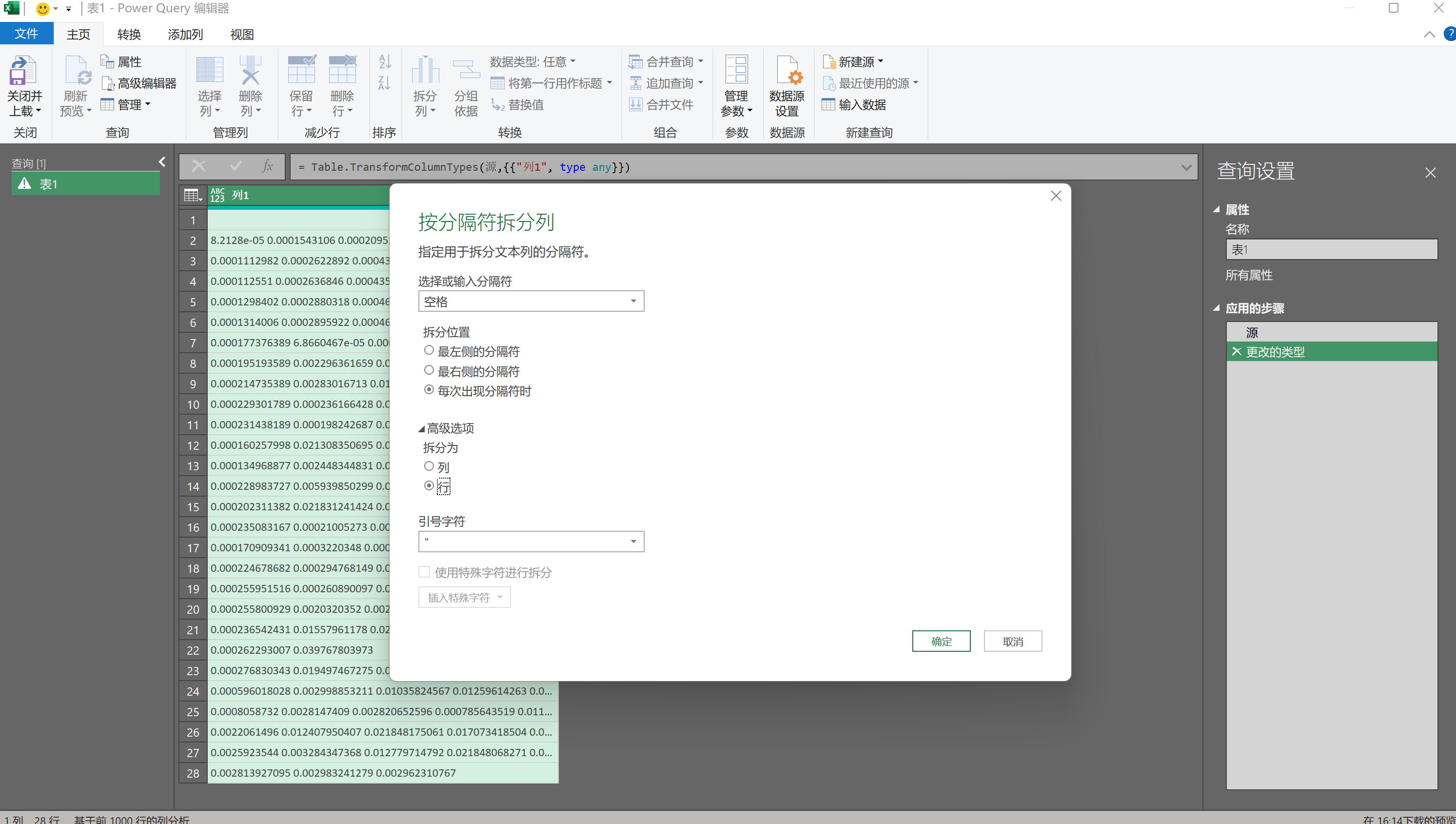


由图可见，Spreadsheets只包含数据，而Records除了数据之外还会包含参数配置等内容。此外，两者的数据排列也有区别，对于向量数据来说，一组数据应该具备时间坐标与具体数据两个维度。Spreadsheets将每一组数据放到一个不同的两列，时间占一列，数据占一列，一组数据一共两列，在每一列中一个数据占据一行；而Records则是将每一组数据放到一个行，在每一行中所有时间和向量数据分别在两个单元格。

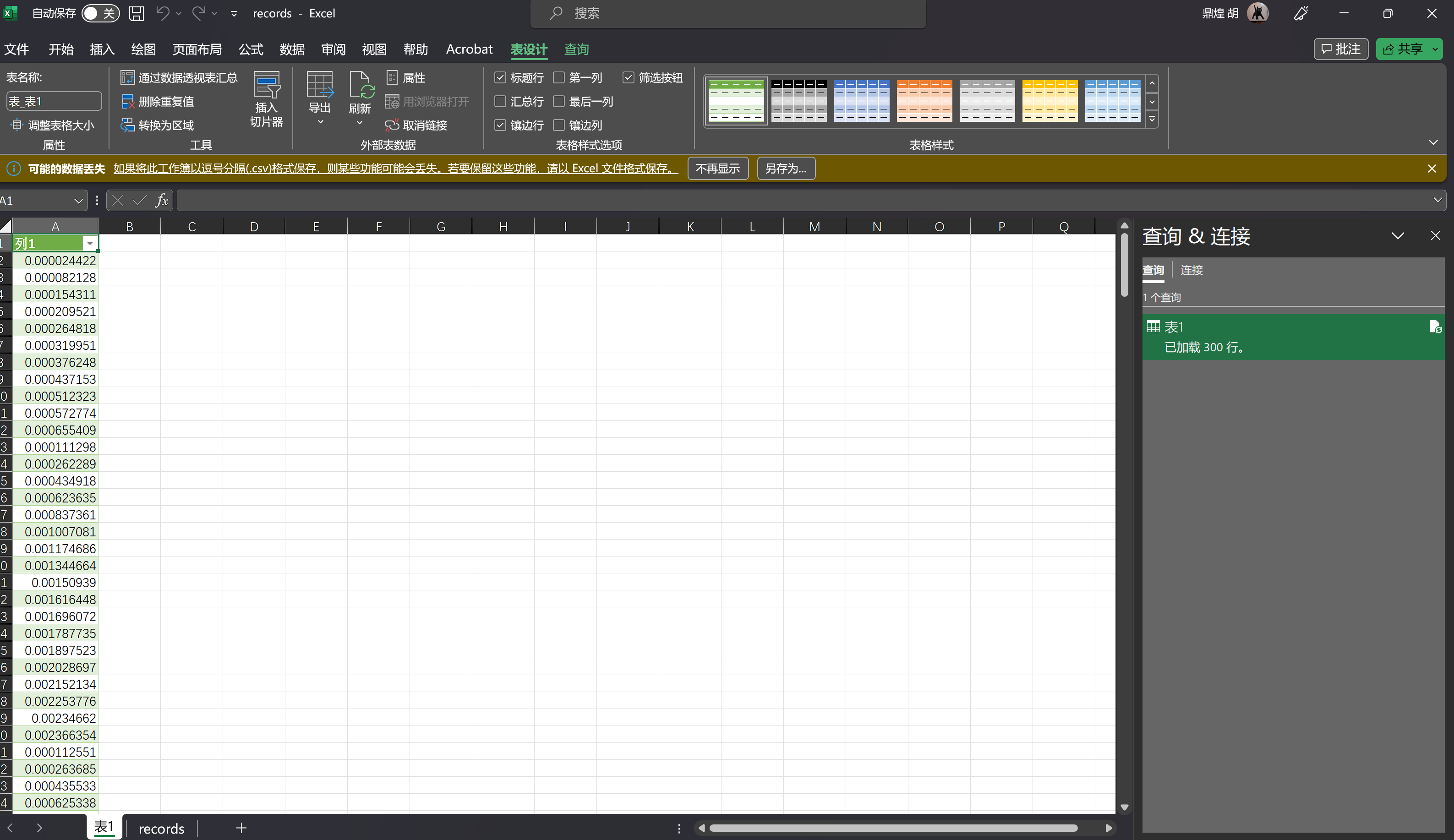
在将数据导入到OriginPro时，可以直接复制某一块数据，也可以通过“数据-连接到文件”来将整个文件内容导入。因此，Spreadsheets的形式更适合每组数据的每个值都需要体现在图上的情况，比如“time-throughput”图，这样在进行数据对齐后可以直接导入OriginPro；而Records的形式更适合需要对所有数据进行处理再得出有效数据的情况。在Records格式下将所有数据整合到一列，可以通过Excel的“数据-来自表格/区域”来根据指定区域的内容创建成一个新的表。



在Power Query编辑器中选择按分隔符拆分，一般为空格。



然后即可保存为一张新的表。



这一步有时会出现因为有某一个数据转换错误而无法生成一张新的表的情况，这是可以考虑直接从Power Query将一列数据复制到OriginPro里进行处理。

## 画图

Origin可以在网上搜到密钥，但是密钥需要隔段时间再更新一次。（如何判断密钥是否失效？当画好图，“导出图片”时，预览界面出现水印）采用学校的邮箱给官网发邮件，可以一直免费使用（学生特权）

举例1：画FCT的CDF图

1. 把所有的流完成时间整合成一列，选中该列，右键-频数分布-确定。这里需要注意的是，为了使最后的CDF图线条更顺滑，建议至少使用7000+条数据，并且扩大区间个数，区间个数一般设置1w+，频率选择“百分比”。

2. 这时出现了“FreqCounts1”，选择“区间中心”和“累积频率”这两列，右键>绘图>折线图。此时的图就是流完成时间的cdf图。

3. 图片优化完毕无误后，在图片空白处右键，选择导出图片，导出成pdf即可。

举例2：数据图中含有多个数据时

如果数据图中有需要分组展示的数据时（一般是柱状图），双击数据。在分组>启用子组>按大小选择要每组的数量，然后改变填充颜色和填充图案的“子组”可以实现区分。