# 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。

项目分为src/inet和example两个文件夹下的HiNA文件夹，分别对应源文件和拓扑与数据文件 。

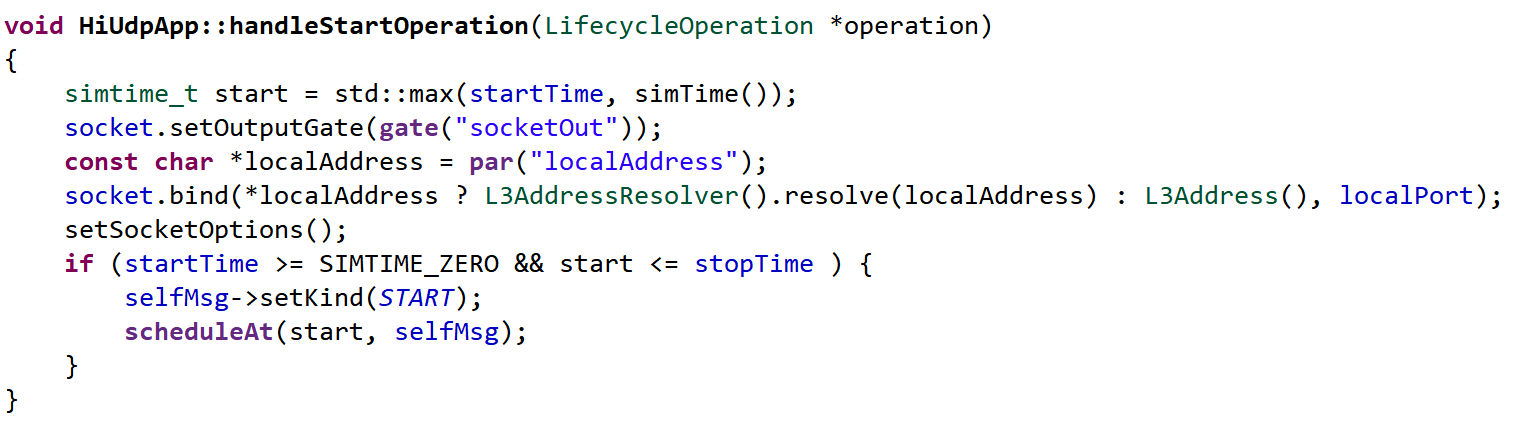
这里主要描述实现思路，给出的配置需要根据不同情况自行调整以达到最优性能。

## 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

收发一体即使用同一个APP模块就能完成接收与发送，实现这点的关键在于两种情况下都要用同一个socket完成对本地端口和地址的绑定。用同一个socket是因为发送端接收端使用同样的代码，如果在接收端新建一个socket B绑定用于接收，那么在发送端就也会只能使用B进行接收，而B对于发送端来说是未绑定的。因此对于HiUdpApp，将其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。



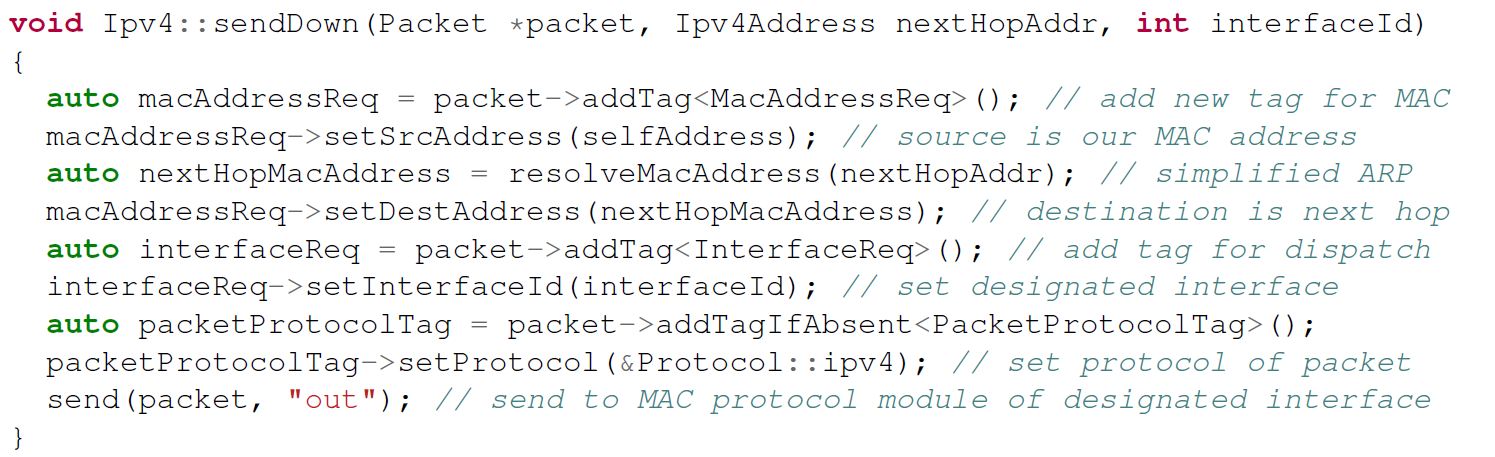
而对于TcpApp，由于需要用TcpSocket.listen()开启监听才能正常进行三次握手，而父类TcpAppBase已经在INISTAGE\_APPLICATION\_LAYER阶段的initialize()就完成了绑定，所以在TcpApp相同阶段的initialize()添加。tOpen为默认值-1时即视为接收端，绑定本地端口后开启监听。

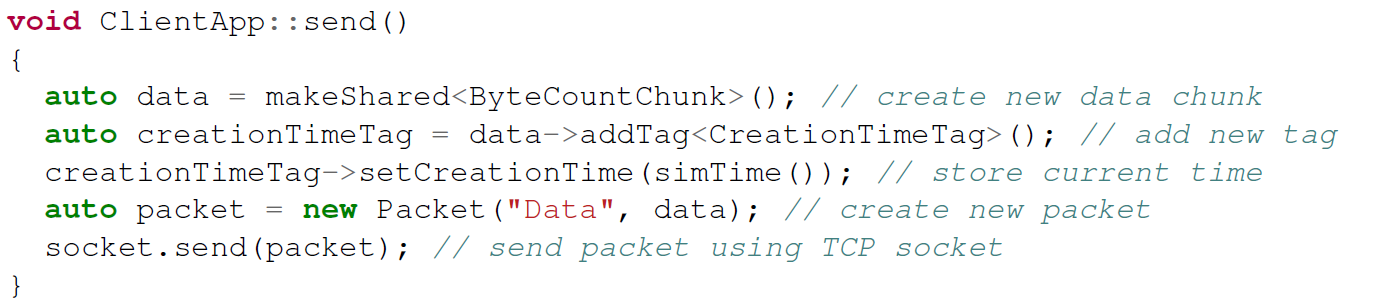


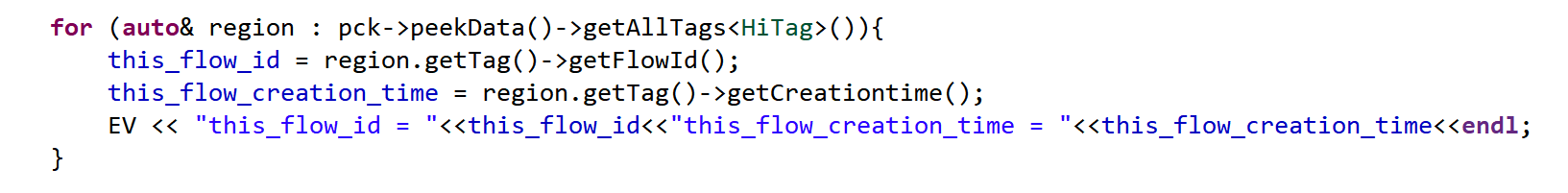
也尝试过在handleStartOperation()中添加但会报listen()之前必须bind()的错，但实际上handleStartOperation()确实是在所有initialize()之后触发的。

## 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

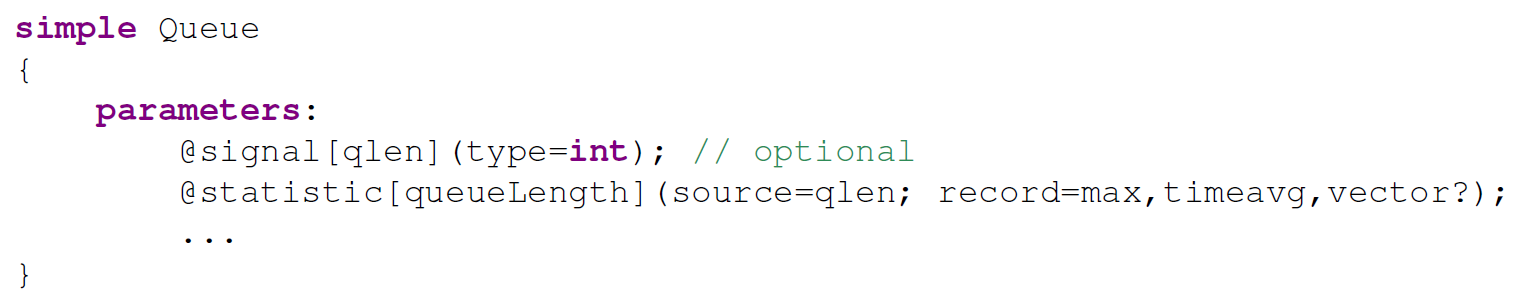
全局流id通过定义APP类的静态变量来实现，**static** uint32\_t *flowid*;。

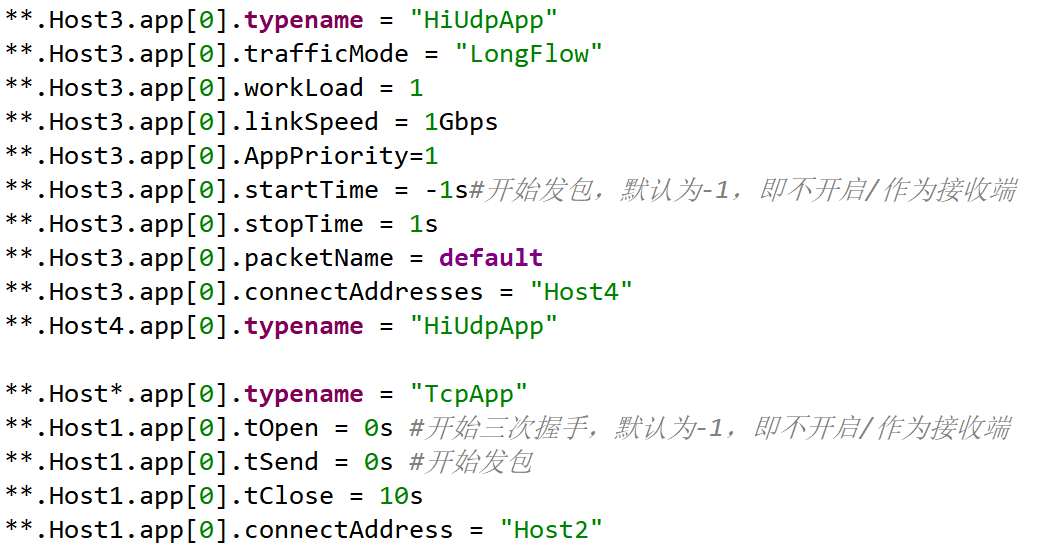
工作负载的配置分为比例和类型，比例指发送满负载的百分比，通过参数workload配置。对于HiUdpApp，我们配置linkSpeed参数使APP模块知晓链路速率，然后用包长度除以链路速率得到发送间隔，通过除以不同workload比例的链路速率就可以得到不同比例的负载。负载类型有预设的几类以及长流（LongFlow）类型，通过参数trafficMode配置，UDP侧的真实负载类型来自于XPASS——**[ns2-xpass/workloads at master · kaist-ina/ns2-xpass · GitHub](https://github.com/kaist-ina/ns2-xpass/tree/master/workloads)**。

而对于基于窗口的TcpApp，无法直接输入workload来控制比例，所以我们保留了通过sendBytes和sendScript手动配置包长度和发送时间的方式。而工作负载的比例则放到传输层配置。

FCT/goodput的测量都于接收时完成，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，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，从INET4.1开始还新增了通过信号机制来统计数据的方法也可以尝试。





APP模块直接使用参考配置如上，可以见得，UDP侧由于是基于速率的发送方式以及RoCEv2中PFC的存在，多了linkSpeed和AppPriority的选项。此外，HiUdpApp还支持从多个选项中选择目的地址可自行寻找参数配置。

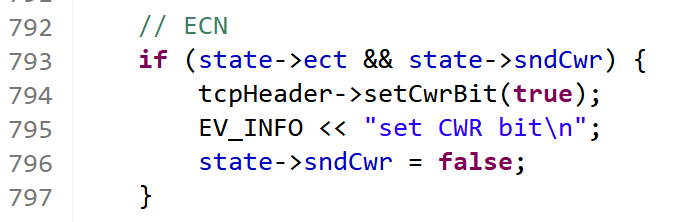
## 传输层

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，则是更多的把算法设计成TCP的一种算法类。

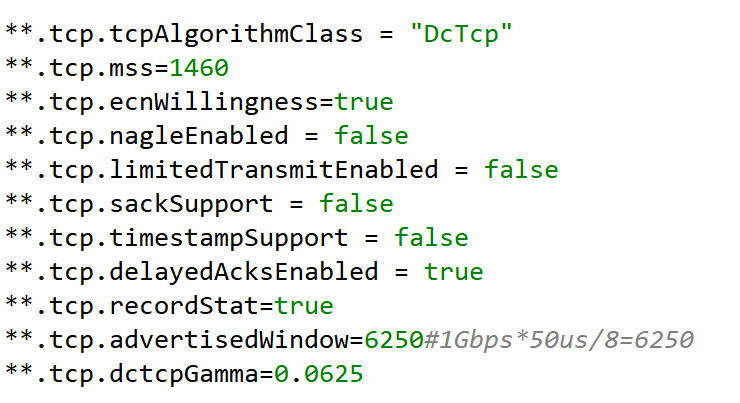
## 1.DCTCP

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。因为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。

而当前INET4.4版本下DCTCP的实现由于TcpConnectionUtil.cc::SendSemgnt()中新增的一段代码而不能正确工作，需要注释掉。这段代码的意思是发送端每次发送标记了CWR位的数据包后会继续允许标记CWR位。这样会导致接收端连续收到CWR位，确认掉后面的ACK，从而造成错误。

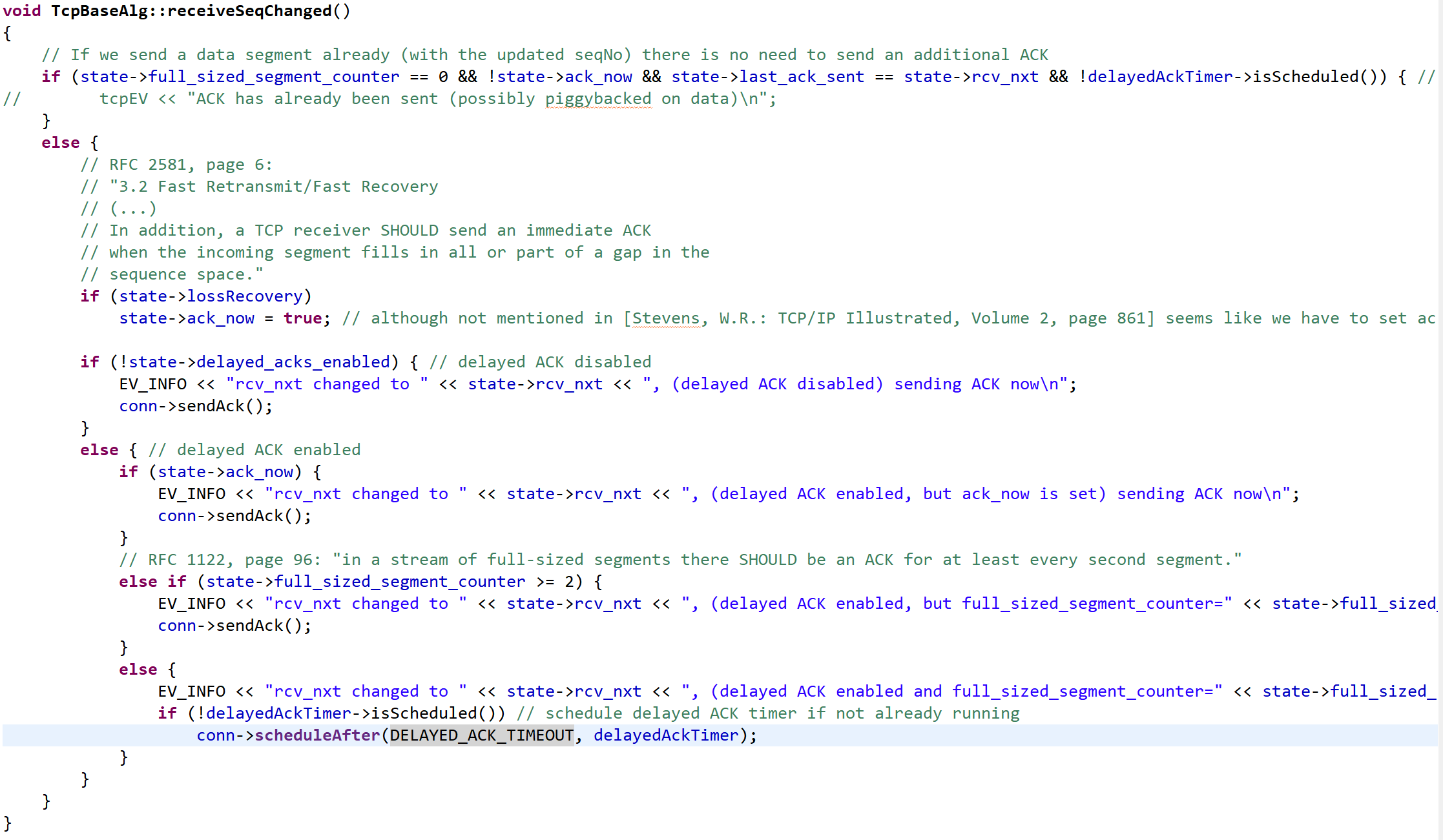


DCTCP使用配置如图所示：



通过配置接收窗口advertisedWindow即可调整负载比例，因为TCP的窗口是在拥塞窗口与接收窗口中取较小值。想要跑满链路速率的话，接收窗口应大于等于链路速率×RTT，在此基础上乘以想要配置的比例即可。

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。



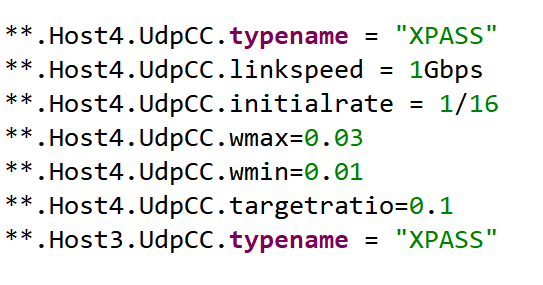
使用中可以等待200ms再发送数据，也可以选择自行修改。

## 2.XPASS

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，信用帧大小为84+[0，8]B，所以信用包大小为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(1480B)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。

XPASS的使用需要开启ECMP和反向路由以及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：



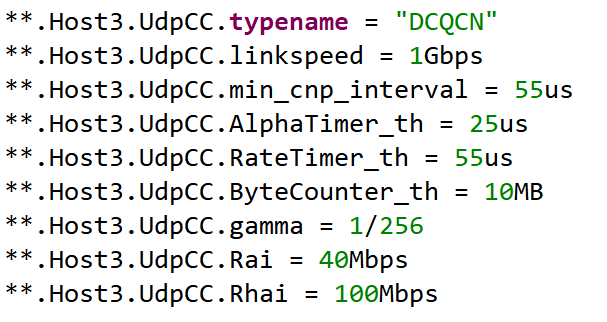
最简情况只需要在接收端配置两个激进度参数wmin、wmax，w一般在0~0.5之间。而我们设计的让XPASS利用ECN和RTT信号来处理拥塞的机制可以通过参数useECN和useRTT来开启。

## 3.DCQCN

DCQCN作为UDP侧的基于发送端的拥塞控制协议，因为是点对点通信，所以不会有一对多的情况只可能会有多对一，所以只在发送端维护一个加速状态SenderAcceleratingState和状态SenderState。

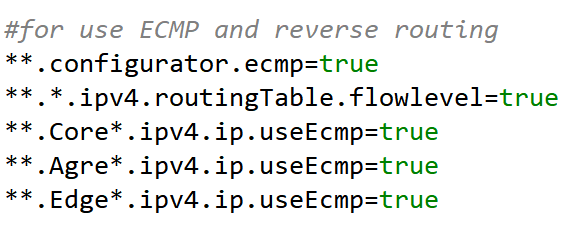
它的流程是：发送端收到应用层传来的包同样是将其记录到sender\_flowMap、删除，如果发送端处于停止状态就开始发送。每次发送会直接发送记录到的整条流，由IP层进行切分。接收端收到带有ECN标记的数据包，如果此时间隔大于最小CNP间隔就会发送CNP包。发送端收到CNP包后减速并清零计时计数器，如果超过计时计数器没收到CNP则会开始多阶段加速过程。我们定义了一个TimerMsg信息来传递这些参数。

DCQCN需要配置的参数主要有链路速率、最小CNP间隔、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%。



## 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。

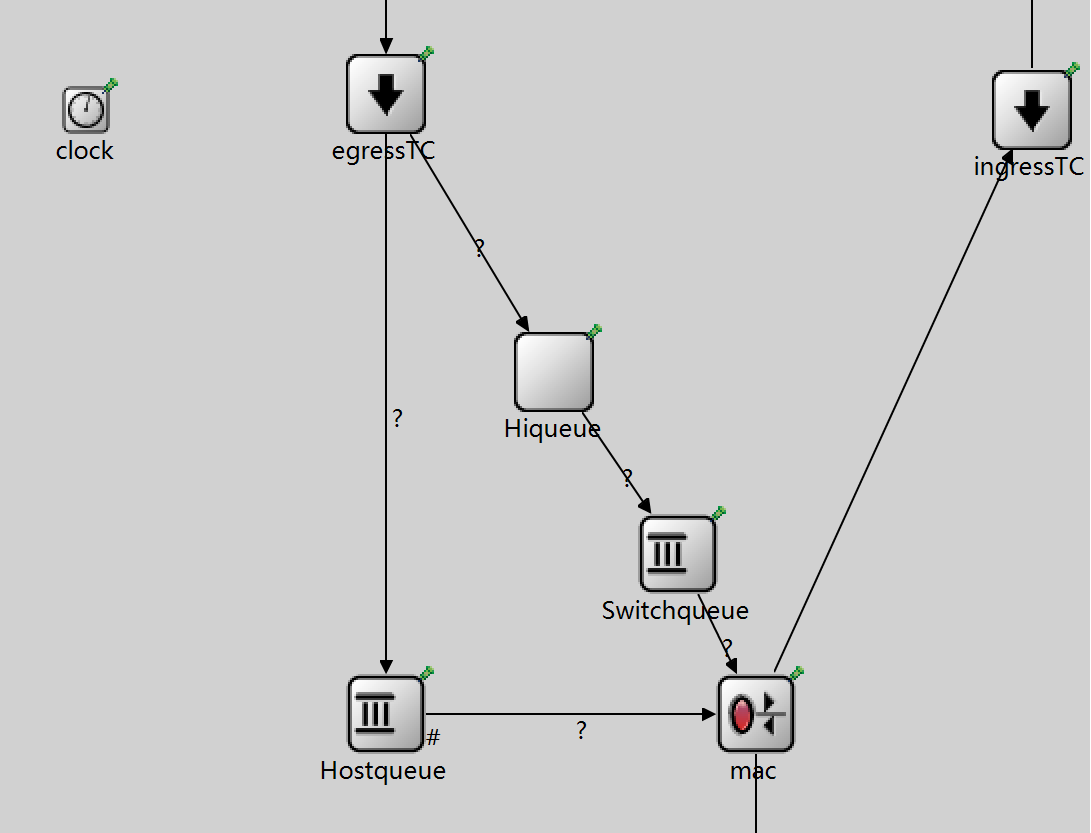
实现ECMP的思路是：在Ipv4NetworkConfigurator的第12初始化阶段，INISTAGE\_NETWORK\_CONFIGURATION，添加静态路由的函数Ipv4NetworkConfigurator::addStaticRoutes()中，将INET框架计算拓扑内到某一结点的最短路径的Topology::calculateWeightedSingleShortestPathsTo()函数，修改为计算所有的最短路径而不是只算一条。然后在遍历任意一对节点之间的路径时，将所有这些路径都进行递归，就可以得到所有的等价最短路径。然后Ipv4NetworkConfigurator模块的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，再写入每个节点的Ipv4RoutingTable的路由表routes，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。

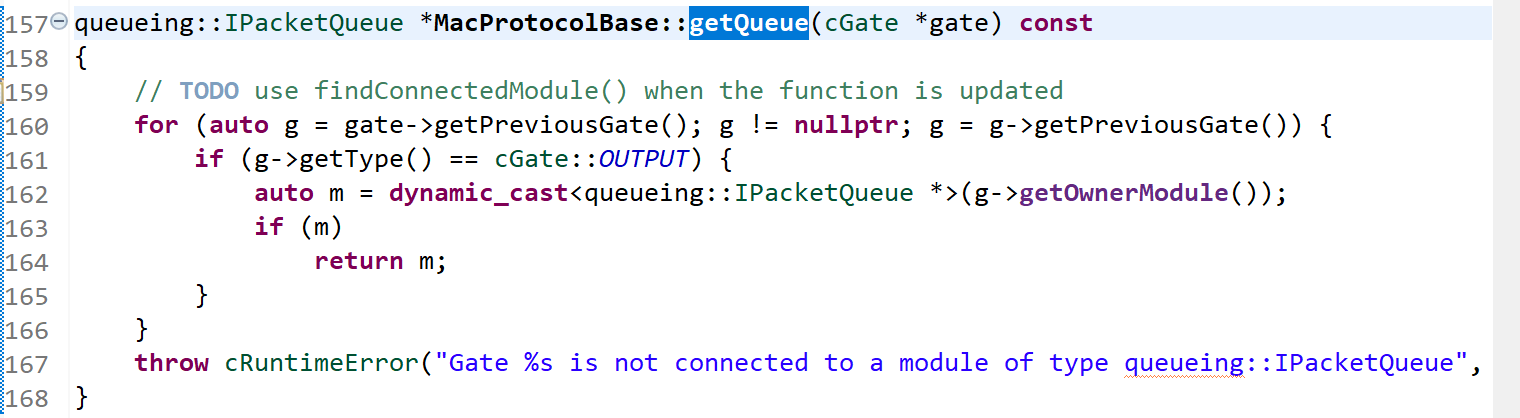
最终需要覆盖的文件有inet/networklayer/ipv4：Ipv4.cc/h/ned，Ipv4RoutingTable.cc/h/ned， IIpv4RoutingTable.h；inet/networklayer/configurator/ipv4：Ipv4NetworkConfigurator.cc/h/ned；common：Topology.cc。

## 链路层

链路层实现的功能有RateLimit、ECN、测速与PFC，在链路层的eth接口处使用我们定义的HiEthernetInterface模块即可，架构如下图所示：

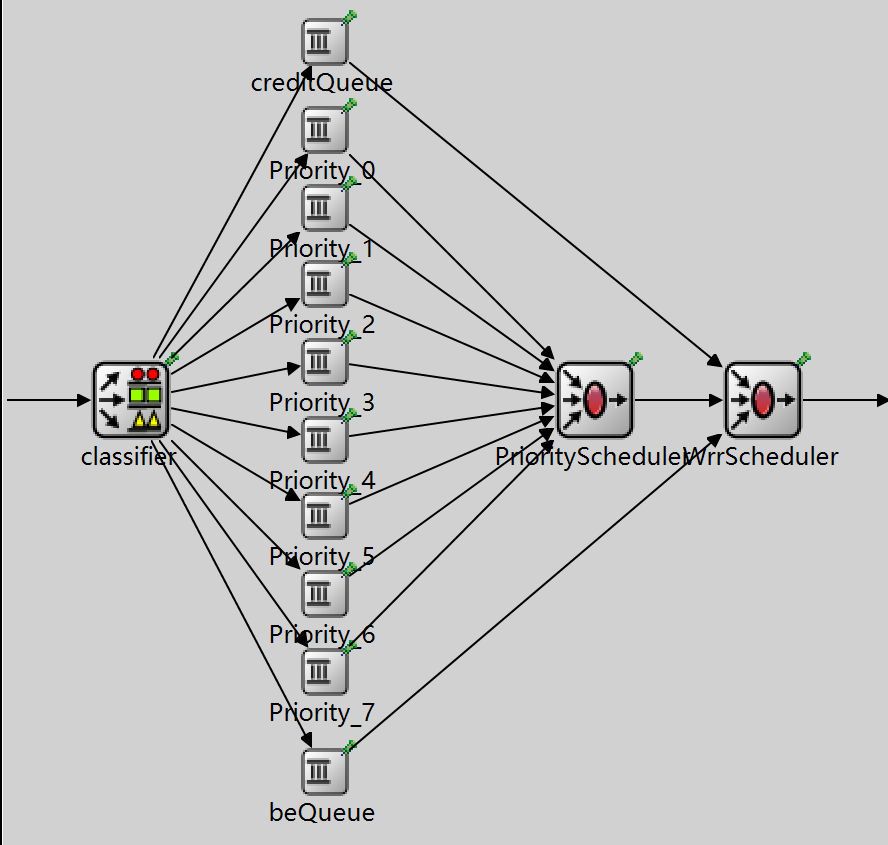


INET4.4版本对于队列模型相关的模块赋予了主动收(ActiveSink)主动发(ActiveSource)被动收(PassiveSink)被动发(PassiveSource)的属性，并定义了四种基础模块接口（IActiveSink、IActiveSource、IPassiveSink、IPassiveSource）。由于INET4.4版本的mac模块将buffer外置，并要求mac的上一个模块必须为IPacketQueue类型的队列，而继承自IPacketQueue的模块又都不具备主动收ActiveSink功能，无法在其之前加入scheduler等模块 ，所以需要作出修改。

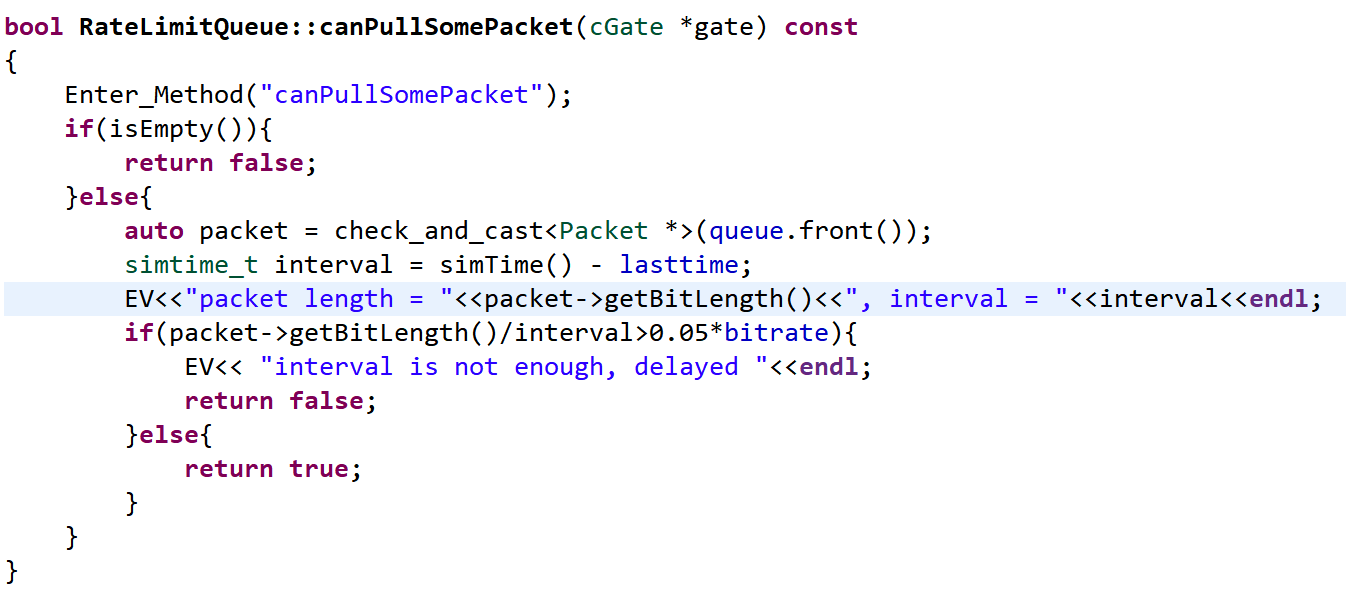


我们在输出端分为两条情况，主机和交换机，可以通过节点ned中的isSwitch选项来区分。对于主机一般没有复杂的队列设计，所以直接使用最基础的PacketQueue即可满足要求；而对于交换机往往会需要队列设计，所以我们基于PacketQueue修改出一个可以传递拉取包信号的SwitchMacQueue用于满足mac模块的要求。具体来说，队列组可以出包的信息通过handleCanPullPacketChanged()传递到mac模块，再将mac模块的拉取信息通过SwitchMacQueue中override的dequeuePacket()传回scheduler，同时将SwitchMacQueue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。然后在它前面添加了我们自己的HiQueue模块作为真正的buffer。

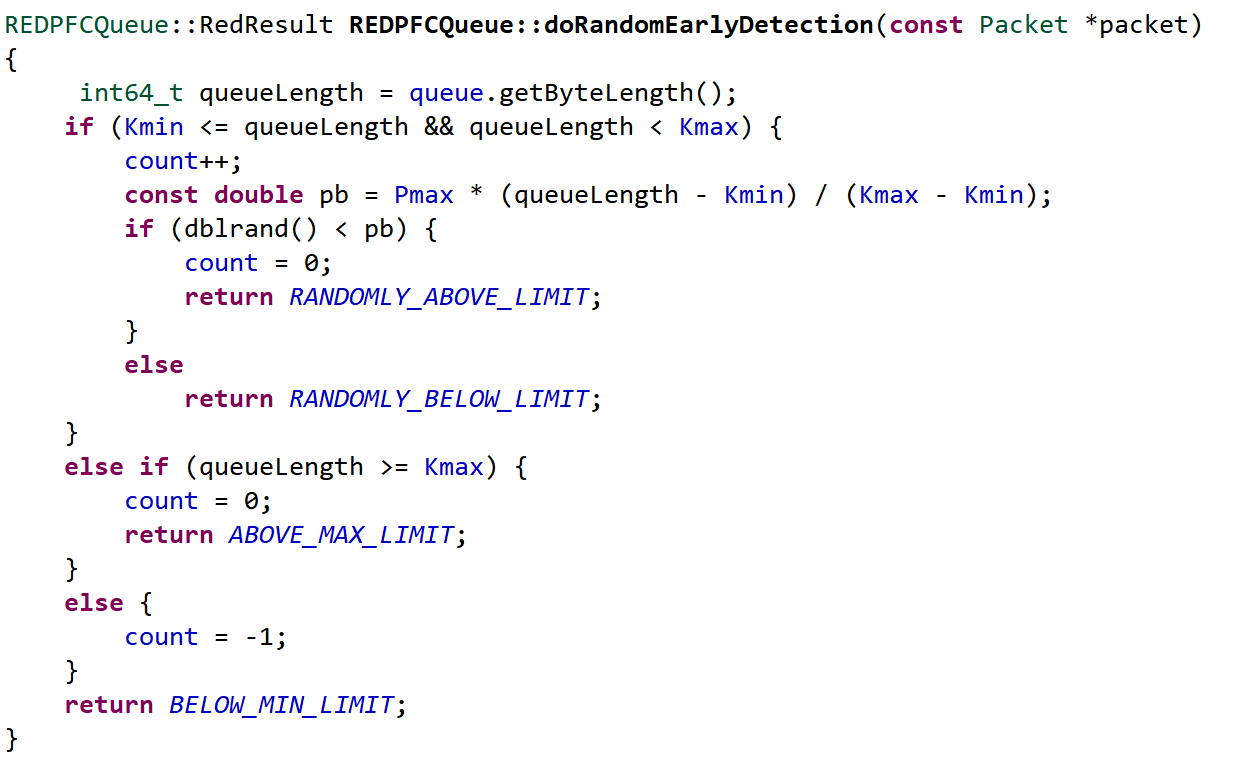
HiQueue默认部署PFC的八个优先级队列，一个普通队列以及一个信用队列，其中信用队列使用RateLimitQueue，基于PacketQueue做了一个简单的限速功能；普通队列使用ECNQueue；而每一个优先级队列都具有PFC和RED-ECN功能。首先我们定义了一个classifer将包分类，名称为credit进入信用队列，tag中设置了进入优先级队列，否则进入默认队列。然后我们使用一个优先级调度器HiPriorityScheduler来调度优先级队列，并使用一个权重轮询调度器WrrScheduler来调度整体队列，并在WrrScheduler中override了pullPacket()使用bufferOccupancyVector测量缓冲区占用，输出数据“bufferOccupancy(byte)”。缓冲区大小PFC优先级队列0.4MB，普通队列0.8MB，信用队列672B（八个信用包大小，for XPASS），总计一个接口约4MB。HiQueue结构如图所示：



RateLimit功能通过复写canPullsomePacket()函数实现，固定限速为XPASS的5%，为此需要配置端口的链路速率bitrate参数。



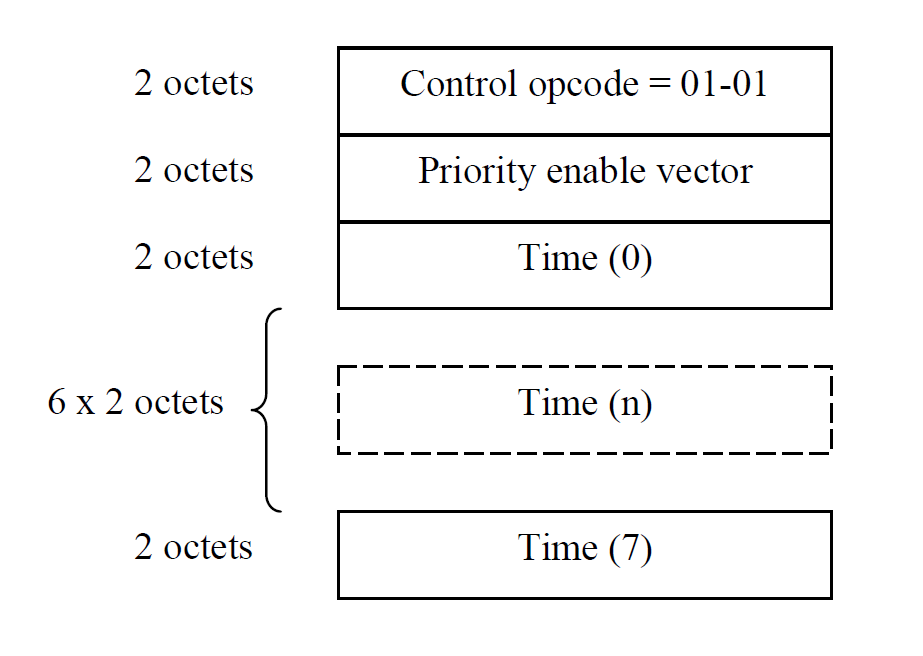
ECN功能在出队时pullPacket()测量，普通ECN队列使用单阈值打标，RED-ECN使用如下RED机制打标。

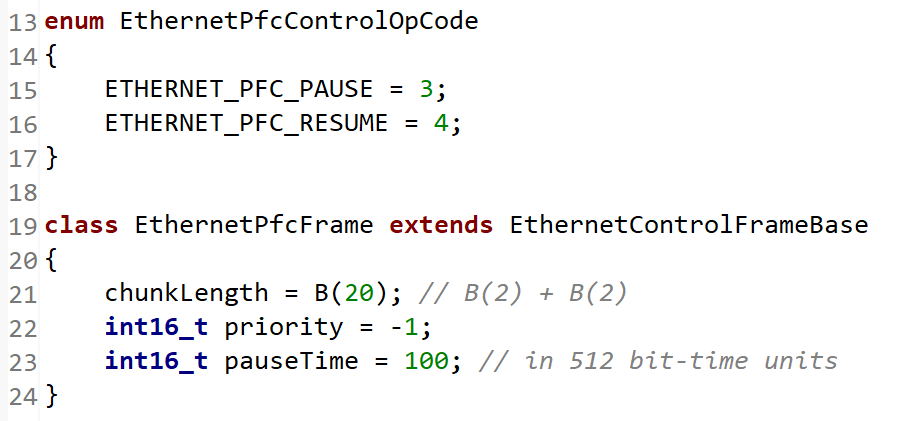


PFC功能的实现思路是 ：队列达到阈值后发送信号给主机里所有mac模块 ，信号携带自己队列所处接口号、优先级、操作码信息，mac模块收到后判断是不是自己接口传来的，如果不是则遵循物理层发送间隔往上游发送对应的操作帧，上一跳的mac模块收到后发送信号给自己接口中的HiScheduler模块，对对应优先级队列进行操作 。PFC的阈值配置要注意大于ECN完全触发阈值，同时剩余容量要能够容纳往上一跳的所有BDP，XOFF与XON之间一般为两个MTU。

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。





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。