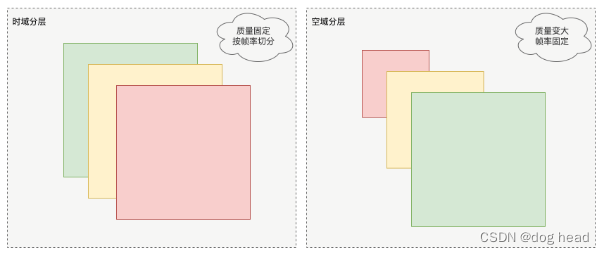
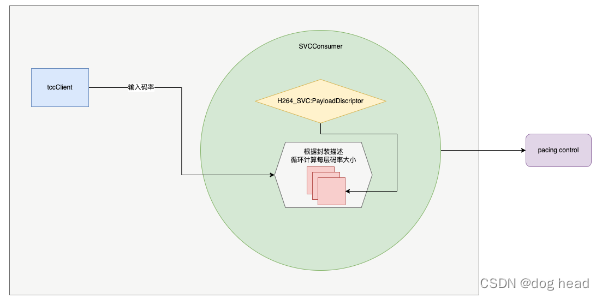
### **细节3：分层切换逻辑示意**

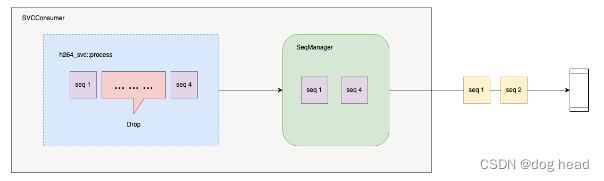
  目前仅支持时域分层、空域分层。  


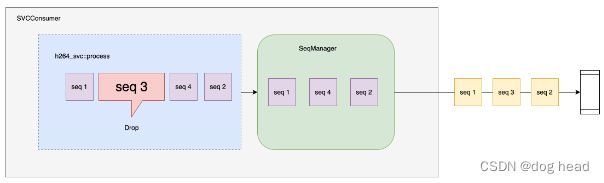
  在上述分层中：  
    时域分层—按标记丢弃帧实现帧率降低达到降低码率的目的；  
    空域分层—保留基础编码帧丢弃高质量帧实现降低码率的目的；  
  具体的码率控制流程如下：  


  1. 当gcc算法收到feedback计算出码率值后会通过回调函数调用SVCConsumer的层变换函数；  
  2. 层变换函数for循环计算producer发送来的流每个层的码率大小；  
  3. 同时通过根据丢包率与当前估计的码率进行换算。  
    丢包率<2：码率*1.08  
    丢包率>10：码率*(1-0.5 \* (丢包率) )  
    丢包率<10 && 率>2 维持不变 ；  
  4. 通过计算出的码率大小对比每层码率计算的大小将需要变化的层级插入变化队列；  
  5. 码率不足时循环层级变化队列放入SVCConsumer进行层级切换，最终实现码率。

### **细节4：seq转换逻辑**

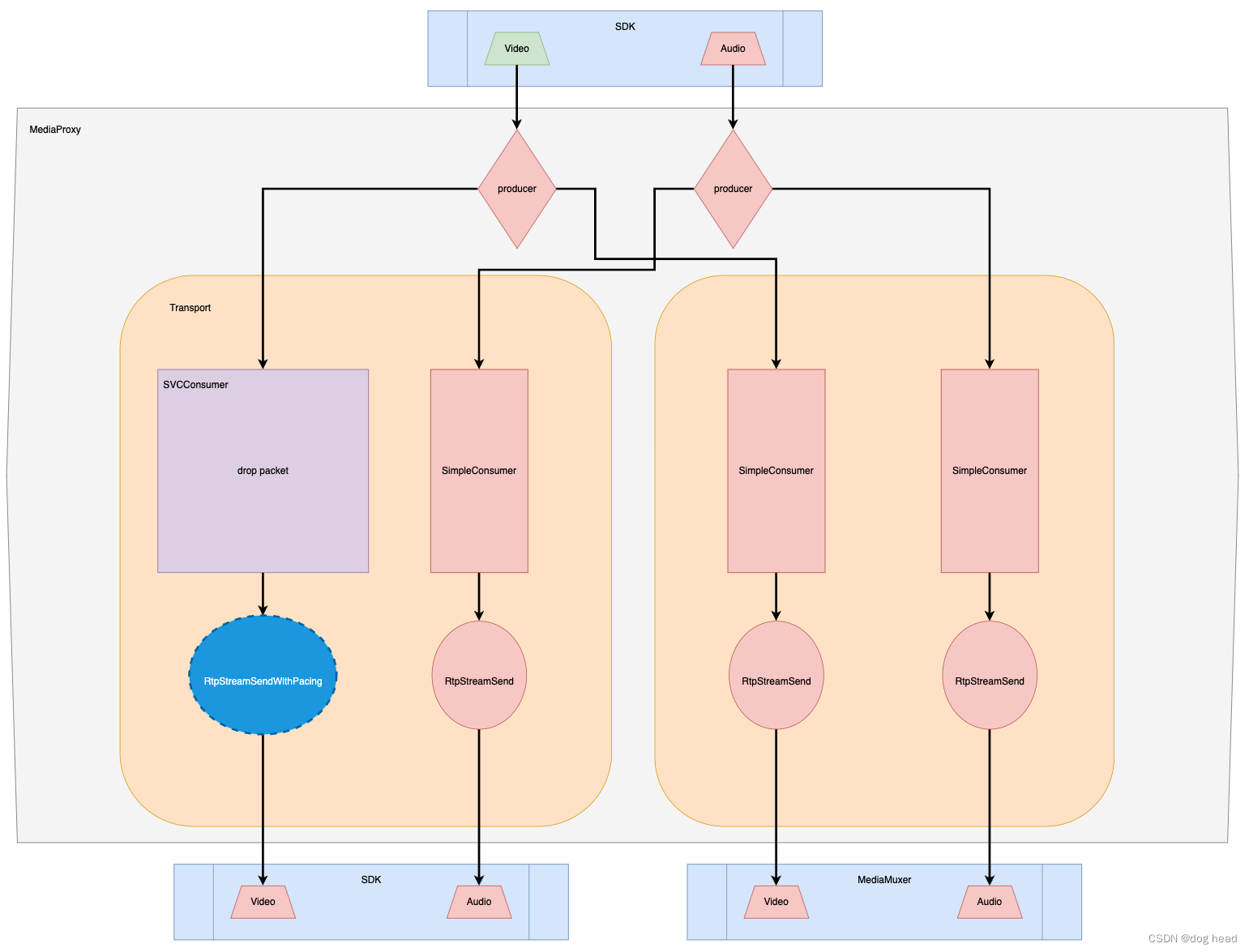
  下行seq切换逻辑为：  
  1. 记录初始基准为0，当有同步包时会切换为最大输出-基准seq；  
  2. 当rtp包进入到SVCConsumer的发送逻辑时，H264\_SVC解析器会对根据encoding上下文内容决定是否丢弃这个包（当该包属于要切换的层并且可丢弃时就会被丢弃）；  
  3. 被丢弃的包就会进入drop队列，并且不被发送；  
  4. 当来新包的时候会对drop队列中进行异常值二分查找（查找内容为 当前序号-最大值/2）;  
  5. 随后统计出查找到的内容与begin的距离作为计数，同时移除drop队列到当前异常值位置，并使用基准减去该计数作为新的基准值；  
  6. 验证完异常值后再次对输入的seq进行二分查找：  
    a. 如果存在并且不等于当前rtp包，就计算出它与drop队列最后一个包的距离作为计数。再次使用基准减去该计数。  
    b. 如果不存在则，不处理；  
  7. 最后把当前输入的序号+基准值作为新的序号。

  例如下图：  


  无乱序的情况：  
  1. 进入数据包1，并且没有同步包，base为0；  
  2. 当2、3两个个rtp包进入到process后，假设2、3因带宽不足被丢弃，此时drop队列中存在2、3；  
  3. 当4完成校验进入到序号转换逻辑，此时4不在drop队列里，异值者（4-65535/2）二分查找到drop队列的尾—移除所有内容，base = base - drop.size —> base = 0-2 = -2;  
  4. 那么新的seq就为 4 - 2 = 2；  
  5. 最后输出的包序号为 1、2。  


  有乱序的情况：  
  1. 进入数据包1，并且没有同步包，base为0；  
  2. 数据包2丢失，3数据包进入了丢弃队列；  
  3. 4完成校验进入到转换逻辑，此时计算出来base为-1-1 = -2；  
  4. 新的seq = 5-2 = 3；  
  意味着，一旦出现丢包的情况，往出转发的内容将会过丢失的序号并且如果2是需要丢弃的也不会补上。如果2不是需要丢弃的则会补上2发出去变为 1、3、2的发送顺序。

# **二、新增能力**

  在原版的mediasoup中，只有SvcConsumer和SimulcastConsumer这两个可以提供码率控制的能力，而且nack的数据是不受码率控制的。这会引入一个比较大的问题，就是对比较复杂的弱网环境（带宽限制 + 丢包混合）传输控制能力变得非常差。我们需要结合发送控制来提供更可靠的下行传输能力。  
  因此我们考虑基于RtpStreamSend类提供一个带有pacing功能的新类：  
  
  根据上图，可得以下流程：  
    a. 数据包从router进入SVCConsumer后会根据GCC带宽进行丢帧处理；  
    b. 随后会进入到RTPStreamSendWithPacing进行Nack、FEC、统计等，流相关的处理：  
      ⅰ. 这里还会缓存下行的Nack和识别seq的正确性；  
      ⅱ. 验证完成后，将seq放入pacing部分等待SVCConsumer定时取数据上抛；  
      ⅲ. pacing为优先级队列，优先级最高的（Nack包、FEC包、RTP包）。  
     c. 上抛到transport做传输相关的处理（例如：给数据包打下行发送的GCC序号、信令处理等）；

## **2.1 优先队列**

### **2.1.1 存储内容**

namespace RTC {

class RtpPriorityQueue {

public:

typedef enum {

NONE = 0,

NACK = 1,

RTP = 2,

PADDING = 3

} RtpPacketType;

// 父元素结构体

struct Element {

Element(bool init = false, int prior = 0, uint64\_t now = 0,

uint16\_t seq = 0, uint32\_t size = 0, RtpPacketType type = NONE)

: isInit(init),

priority(prior),

timeStamp(now),

sequence(seq),

packetSize(size),

packetType(type) {}

// 重载判断符，根据 优先级 > 入队时间 > 序号排列

bool operator<(const Element& e) const {

if (priority < e.priority)

return true;

else if (priority == e.priority){

if (timeStamp < e.timeStamp) {

return true;

} else if (timeStamp == e.timeStamp) {

if (sequence < e.sequence) {

if (e.sequence - sequence > 65536/2)

return false;

return true;

} else if (sequence - e.sequence > 65536/2)

return true;

}

return false;

} else {

return false;

}

}

bool operator<=(const Element& e) const {

return priority <= e.priority || timeStamp <= e.timeStamp;

}

bool operator>=(const Element& e) const {

return priority >= e.priority && timeStamp >= e.timeStamp;

}

bool isInit;

int priority;

uint64\_t timeStamp;

uint16\_t sequence;

uint32\_t packetSize;

RtpPacketType packetType;

/\* -------- 数据结构关联 -------- \*/

// list to queue

int index { 0u };

// queue to list

std::shared\_ptr<Element> pre;

std::shared\_ptr<Element> next;

};

typedef std::shared\_ptr<Element> ElementPtr;

public:

RtpPriorityQueue();

~RtpPriorityQueue();

public:

// 队列操作

void Push(Element element);

Element Pop();

bool IsEmpty() { return element\_queue\_.empty(); }

size\_t GetQueueSize() { return element\_queue\_.size(); }

void ClearQueue() {

element\_queue\_.clear();

oldest\_element\_ptr\_ = nullptr;

last\_element\_ptr\_ = nullptr;

}

private:

void DelOldElement();

ElementPtr QueuePushHeap(Element element);

ElementPtr QueuePopHeap();

void QueueDelNode(Element element);

void PushBackToList(ElementPtr elementPtr);

void DelListNode(ElementPtr elementPtr);

Element DelListHead();

private:

// 优先级队列索引，用于删除最旧元素

ElementPtr oldest\_element\_ptr\_{ nullptr };

ElementPtr last\_element\_ptr\_{ nullptr };

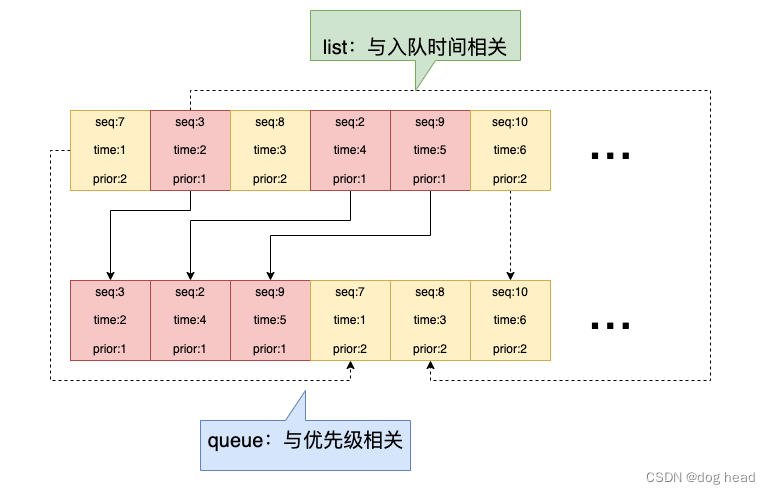
// 优先级队列存储容器，使用heap计算堆

std::vector<ElementPtr> element\_queue\_;

};

} // namespace RTC

### **2.1.2 数据结构示意**

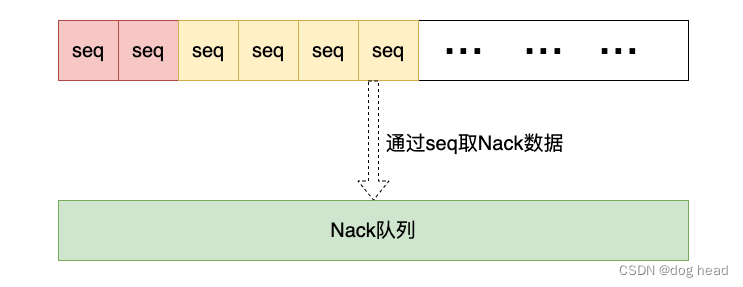


   上图展示了数据结构的示意图，分别由一个小顶堆和一个链表组成。  
   ○ 链表：入队顺序为时间强相关，用于淘汰过时的数据；  
   ○ 堆：队列顺序以优先级强相关，用于优先发送Nack数据包。

   每个节点存储数组下标、链表前后指针，用于快速插入弹出已经清除数据。

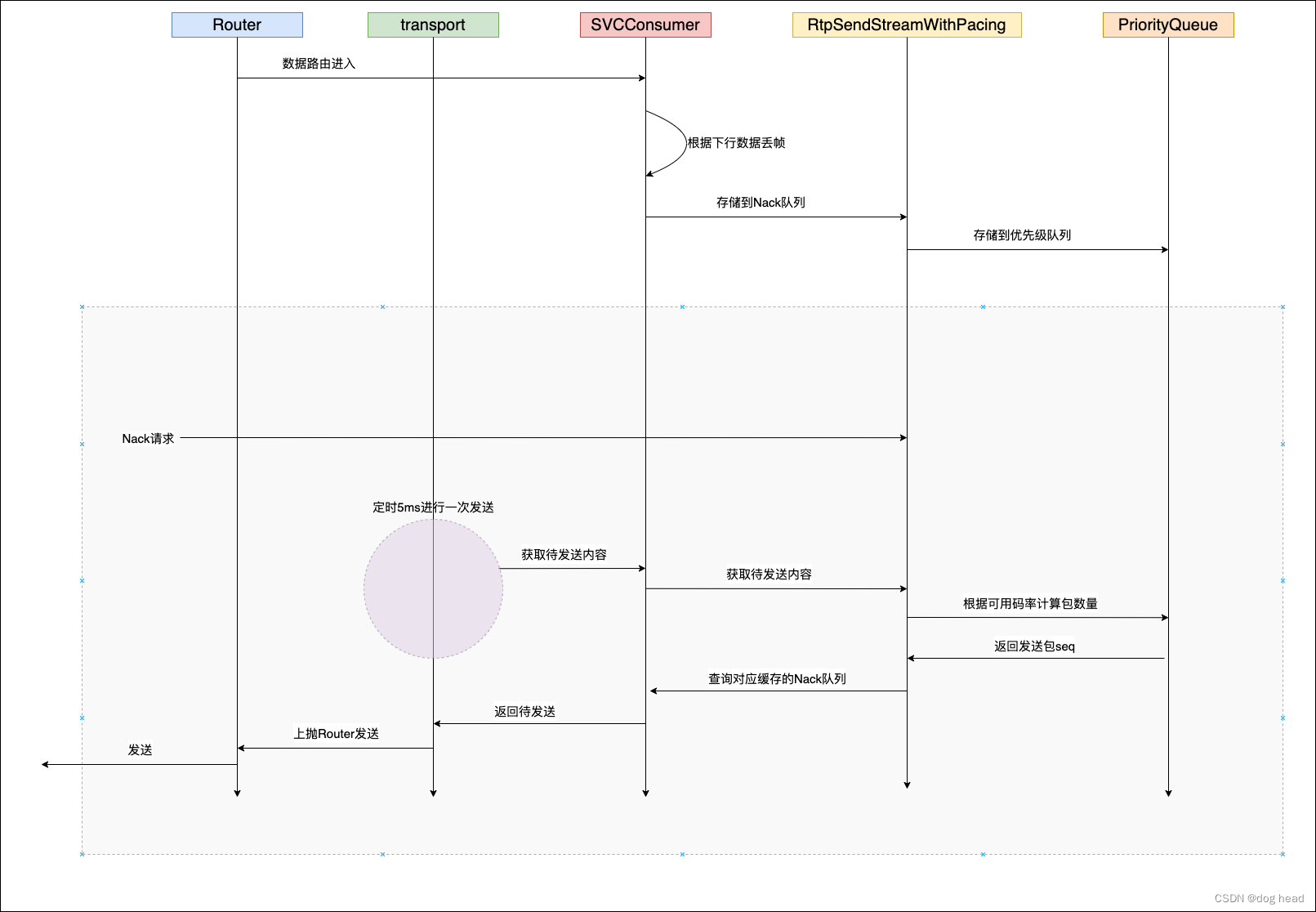
   淘汰机制：  
   ○ 超过最大数组限制（8192），则淘汰时间最早的数据包；  
   ○ 正常弹出。

### **2.1.3 数据操作复杂度**

   我们音视频数据传输中，最大缓存2s数据，pacing队列也遵循以上数据原则。  
  
   计算：  
   目前最大数据量 500包/s，可知Nack队列大概在1000 ~ 1500个包左右；  
   pacing队列为了保证Nack和RTP同时发送，我们可将上限值限制为 8192 或者更小。

   队列：  
   ○ 正常操作，根据堆操作的复杂度来说，添加、删除都为O(logN)。在常数范围内，最大耗时在数据量极低的情况下，引入的操作复杂度为O(1)级别；  
   ○ 超时删除，删除指定节点调节的时间复杂也是O(logN)，同上。  
   链表：  
   ○ 超时删除，对于链表我们只需要做头删除，同时删除堆。链表头删除O(1)；  
   ○ 正常弹出，对于链表已经记录了前后指针，同样是O(1)。

## **2.2 pacing逻辑**

   我们展示整个pacing的时序图可知，在Transport级别进行5ms定时器（具体性能消耗不确定、同时会引入峰值发送的问题），随后在通过我们新增的类进行发送控制：  


## **2.3 \* padding逻辑**

   由于我这边做的比较简单，只使用3层时域分层（码率跨度很大）。因此整个传输过程中提供给GCC使用的码率变化严重，因此可以考虑引入padding逻辑。  
   开启padding条件：  
   ○ 不是最大层；（SVCConsumer感知，每次切换到最大层则关闭padding。）  
   ○ 带宽有剩余；（SVCConsumer感知，每次发送剩余的带宽统计。）  
   ○ padding 进入发送队列，优先级最低；  
   ○ 不存在NACK等待发送；  
   ○ 整个优先级队列数量少于正常发送RTP时一个周期的最大包数；

   padding 包内容：  
   ○ 最新的T0层的包（T0层占最大码率约 40%）；  
   ○ 后期可拓展（T0层的FEC包）；

# **三、总结**

   本文简单介绍了mediasoup中的SVC，同时给了一个使用H264-SVC的案例——代码就不展示了。整个测试的效果是：  
    1.带宽场景，最低抵抗600kbps的带宽限制（跟T0层、分辨率有关，做的更小可以抵抗更低）；  
    2.混合场景，可抵抗10%随机丢包+700kbps带宽限制。  
  大家可以一起讨论一下方案的问题，我们共同进步。