EXata 扩展（六）：添加 PRODUCER

1. EXata 支持的传输层协议

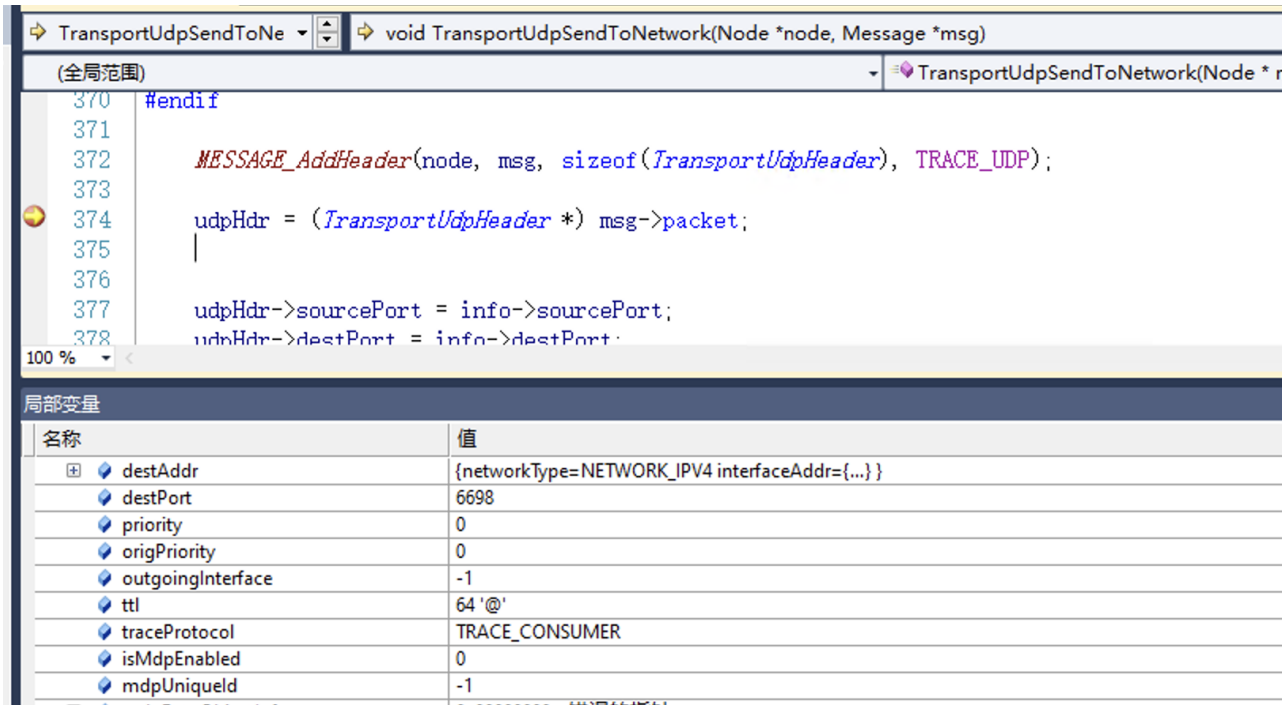
EXata 5.1 目前支持的传输层协议有 UDP、TCP 和 RSVP-TE。RSTP-VE 不是用于数据传输，而是一种信令控制协议，用于在 MPLS 网络进行资源预留和流量工程。

此外，借助于 UDP/IP，还有一个特殊的传输层协议，MDP（Multicast Dissemination Protocol），它利用UDP/IP组播技术实现一个源到一组目的地的可靠传输。它特别适合异构网络间大数据量的可靠组播。内部，MDP 基于NAck 实现可靠组播，并可选支持端到端自适应的速率调节。

2. UDP 传输过程

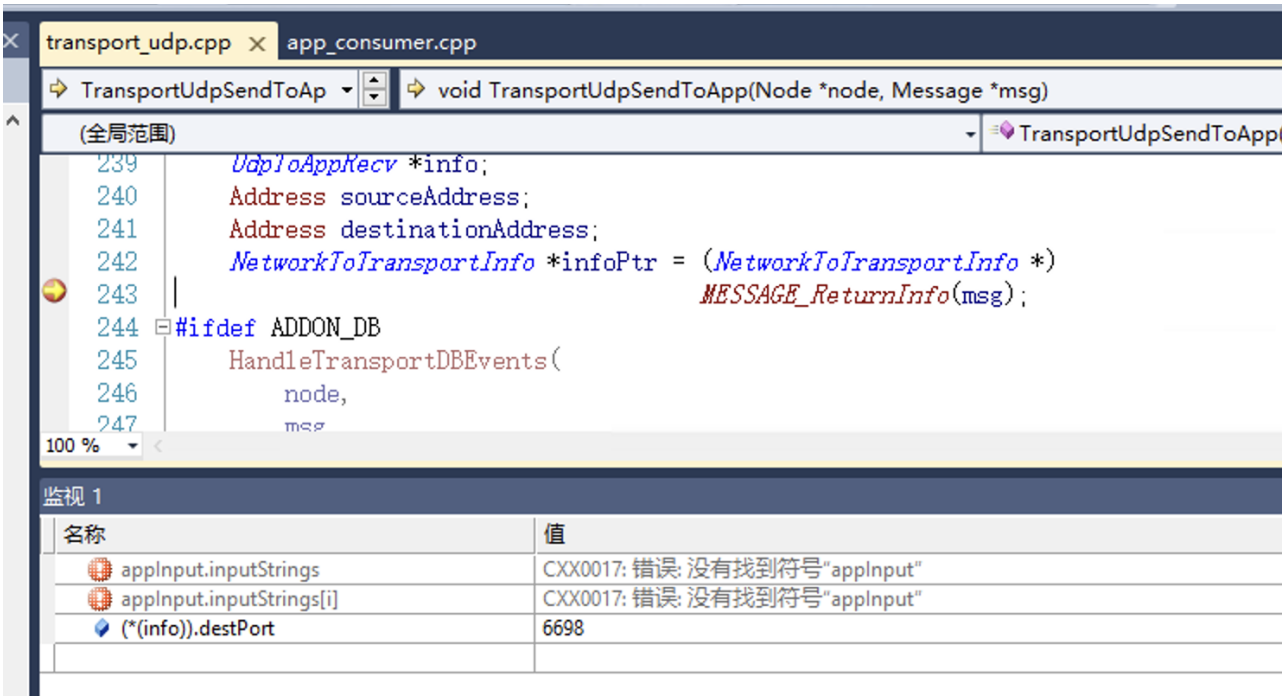
2.1 UDP down to Network

调试进入 TransportUdpSendToNetwork 方法，在 CONSUMER 层发送完后，可以发现 UDP 已经将 Message 发送给网络层：从目的端口 6698 和跟踪协议 TRACE_CONSUMER可以判别：



2.2 UDP 到 APP

很快，在 TransportUdpToApp 中也追踪到该消息：从 destPort = 6698 判断



UDP 会设定一个消息（事件）：*MSG_APP_FromTransport，亦即未来 Producer 要处理的事件。*

至此，由于采用广播地址（ANY_ADDR or ANY_DEST），应该多个节点都能收到，只要安装 PRODUCER 实例（监听 #6698 端口）应该就能收到。因此，可以考虑实现 PRODCUER。

3. 添加 PRODUCER

3.1 添加文件

1. 在application.h中添加协议类型：APP_PRODUCER【Done】
2. 在trace.h中添加追踪协议类型：TRACE_PRODUCER【Done】
3. applicaiton.cpp 中 APP_InitializeApplications方法中添加应用层协议初始化入口：

```
// LuoJT: initialize PRODUCER
                        else
                        if (strcmp(appStr, "PRODUCER") == 0)
                        {
                                // add here
                        }
```

4. 首先要确定 PRODUCER 的配置接口
   a. 生产者名字ProducerName：必选，不可见
   b. 资源名字（rscName）：必选，不可见，默认为 hero.mp4
   c. 资源版本（rscVersion）：必选，不可见，默认 1.0
   d. 资源类型（rscType）：：必选，不可见

i. Object：目标类型（default），比如文件；内容类型（content type）放到Parameters 列表中

ii. service：服务类型，比如算力

e. Chunk 数（chunkDum）：必选，不可见，默认 100

f. Chnk大小（chunkSize）（Bytes）：必选，不可见，默认 2048

g. 发布时间：多久之后发布，必选，不可见，默认10s

h. 有效时长：多久之后失效【以后考虑】

i. 发布范围：必选，不可见

i. Global （default）

ii. Domain

iii. Local

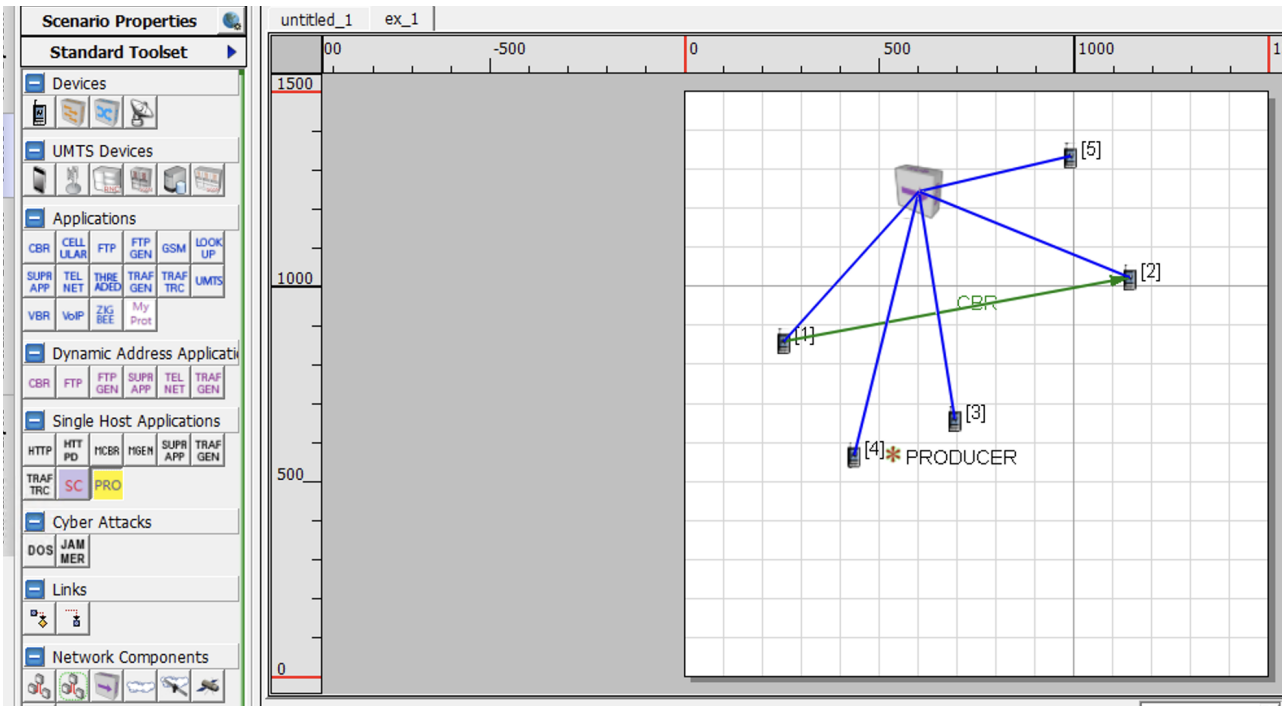
5. 因此，PRODCUER 的配置字符串如：【最后面有修改】

a. PRODUCER CQUPT hero.mp4 1.0 Object 100 2048 10S GLOBAL

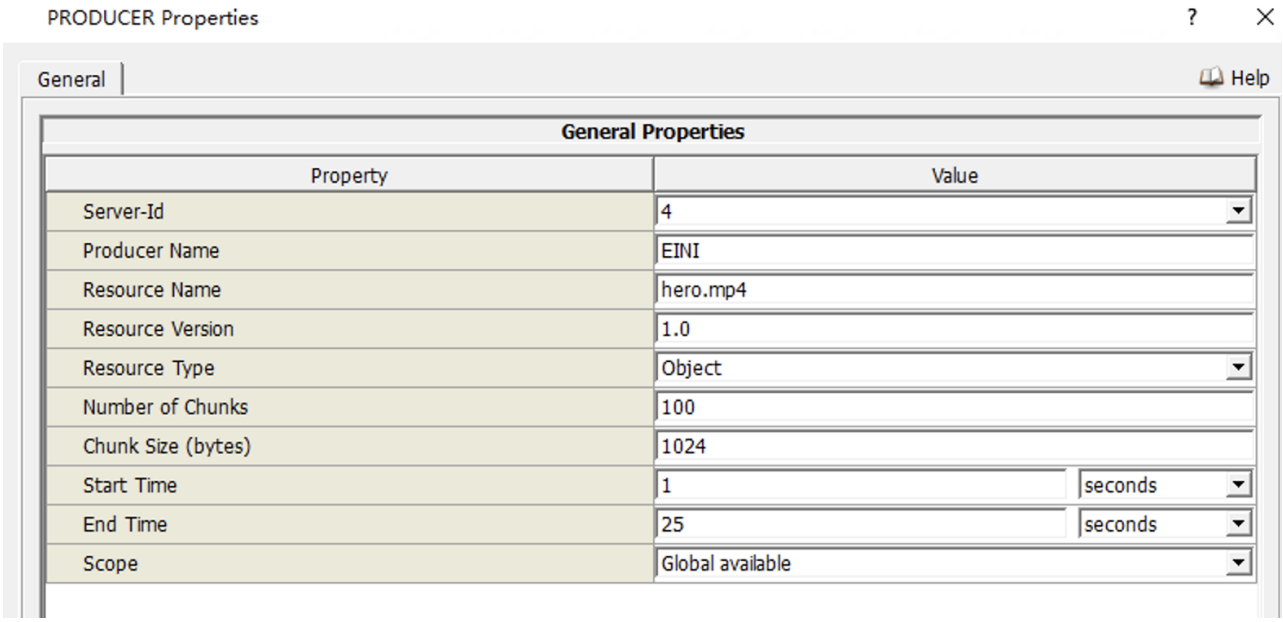
6. EXata GUI 添加 Prod 单机应用：

a. 新建 producer.cmp文件：gui/settings/components/

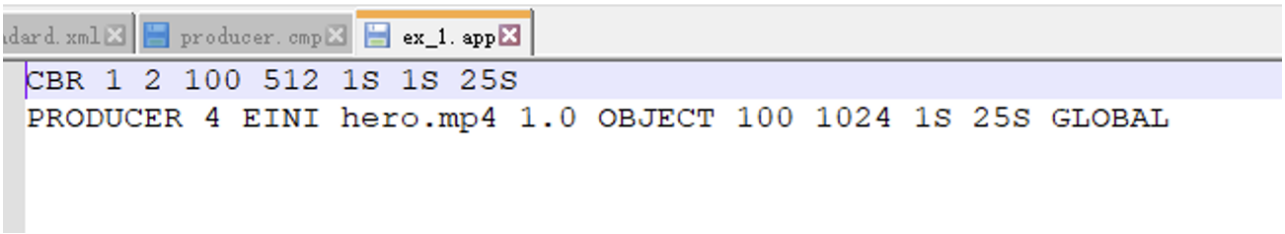
b. 在工具箱中添加新按钮：修改Standard.xml文件，在category “Single Host Applications”下添加一个新的subcategory “PRODUCER”



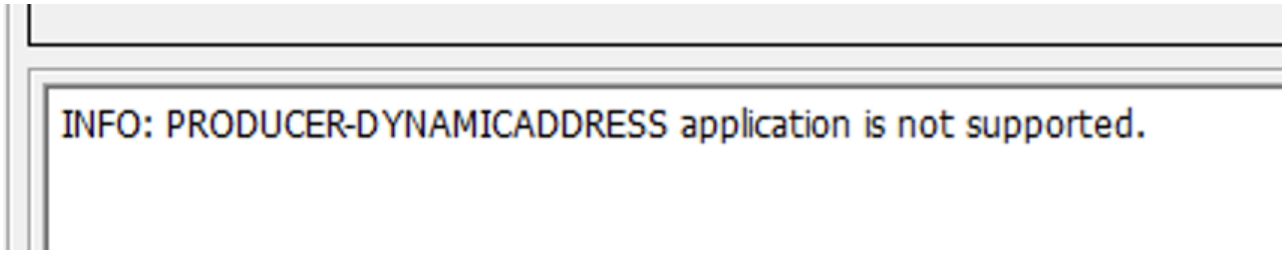
c. 设置 PRODUCER 属性如下【有更新】：



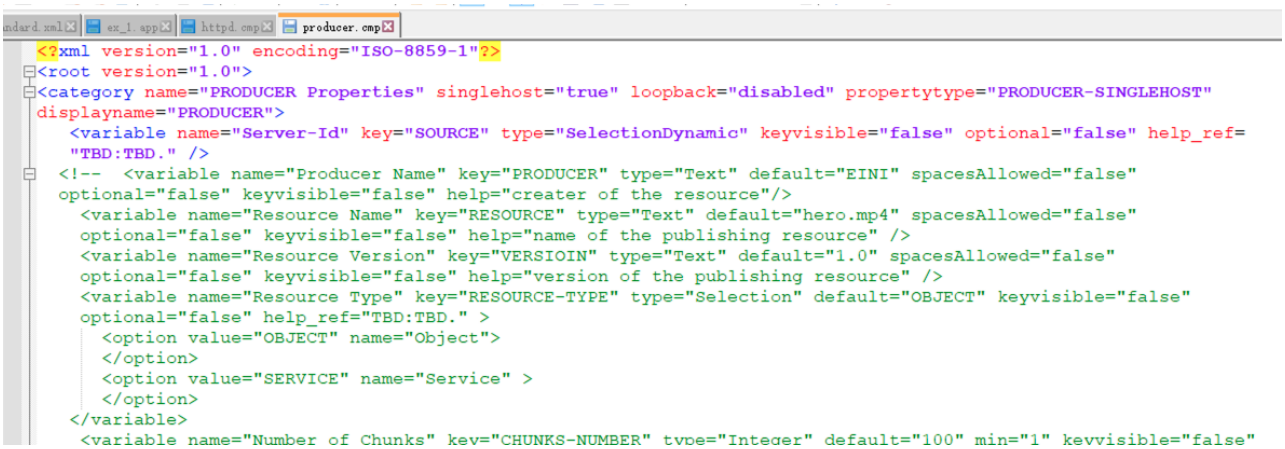
d. 生成的参数行如下：【GUI 能否认可呢？？？】【有更新；更新后认可】



e. 保存后，重启gui，重新打开ex_1.config，同样有警告信息！！！【后面有更新，问题已解决！】



f. producer.cmp中只保留“Server-Id”一行，就像 httpd.cmp 那样，是可以记住的。但更多时候程序直接退出！！！



g. 下周反馈问题给厂家，请求协助解决。【问题终于解决！附后】

7. 添加 app_producer.cpp

a. 在 user_models/ 下添加 app_producer.cpp；

b. 本地 Makefile-common 中添加新建等源文件一起编译

Makefile-common | app_consumer.h | app_consumer.cpp | hr

```
USER_MODELS_OPTIONS =


USER_MODELS_DIR = ../libraries/user_models
USER_MODELS_SRCDIR = ../libraries/user_models/sr

#
# common sources
#
USER_MODELS_SRCS = \
$(USER_MODELS_SRCDIR)/app_myprotocol.cpp \
$(USER_MODELS_SRCDIR)/app_consumer.cpp \
$(USER_MODELS_SRCDIR)/app_producer.cpp


USER_MODELS_INCLUDES = \
-I$(USER_MODELS_SRCDIR)
```

c. 在 app_producer.cpp中添加 AppProducerInit、AppProducerNewProducer 【参数赋值待补充】方法完成初始化。

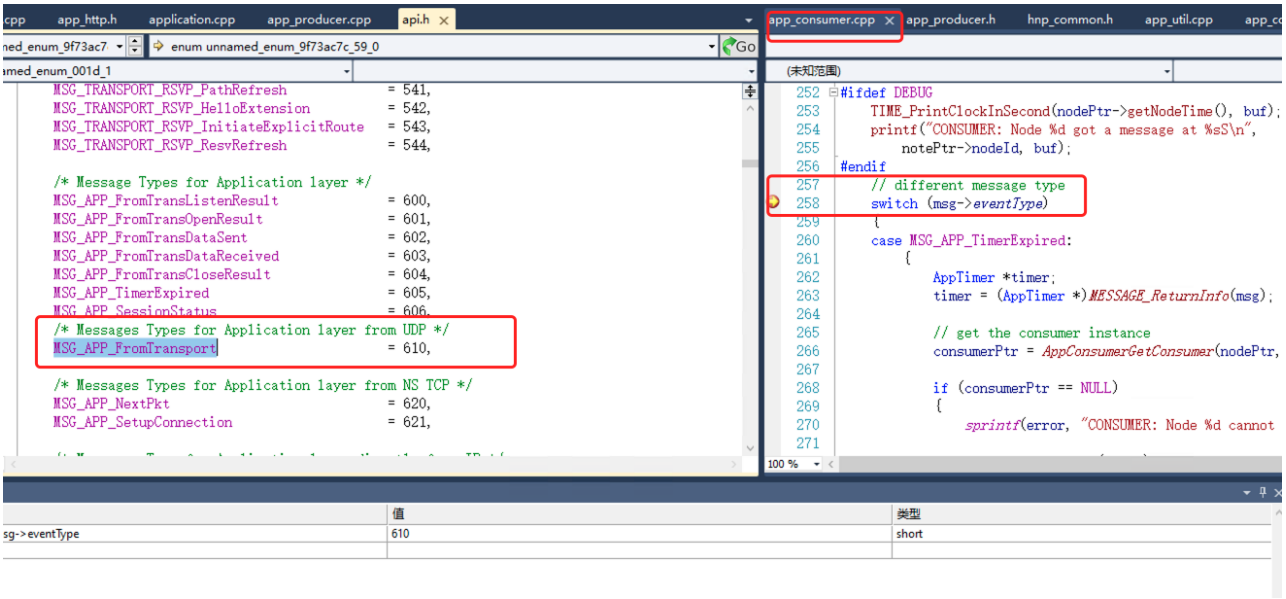
d. 下步完成事件处理！特别是包括发布消息，以及收到 UDP 传上来的来自 Consumer 的 Request 消息！

```
F:\ex\ex_1>exata ex_1.config
EXata Developer Version 5.1
Kernel Version: 12.10
Build Number: 201310091
Build Date: Oct  9 2013, 18:55:48
EXATA_HOME = D:\Scalable\exata\5.1

Attempting license checkout (should take less than 2 seconds) ...Loading scenario ex_1.config
Partition 0, Node 1 (256.01, 859.08, 0.00).
Partition 0, Node 2 (1143.30, 1022.69, 0.00).
Partition 0, Node 3 (694.41, 659.80, 0.00).
Partition 0, Node 4 (436.87, 568.29, 0.00).
Partition 0, Node 5 (991.74, 1334.08, 0.00).
Warning in file ..\interfaces\pas\src\packet_analyzer.cpp:1560
EXata interface not found. Packet Sniffer disabled

Initialization completed in 0.187 sec at 2023-01-09 22:00:33.158
Current Sim Time[s] =    0.324682905  Real Time[s] =    0  Completed  1%
Current Sim Time[s] =    0.613889680  Real Time[s] =    0  Completed  2%
Current Sim Time[s] =    0.997545544  Real Time[s] =    0  Completed  3%
Error in file ..\main\application.cpp:7792
Application layer receives a message with unidentified Protocol = 6698
```

4. Producer 的事件处理：AppLayerProducer 发送 PUBLISH

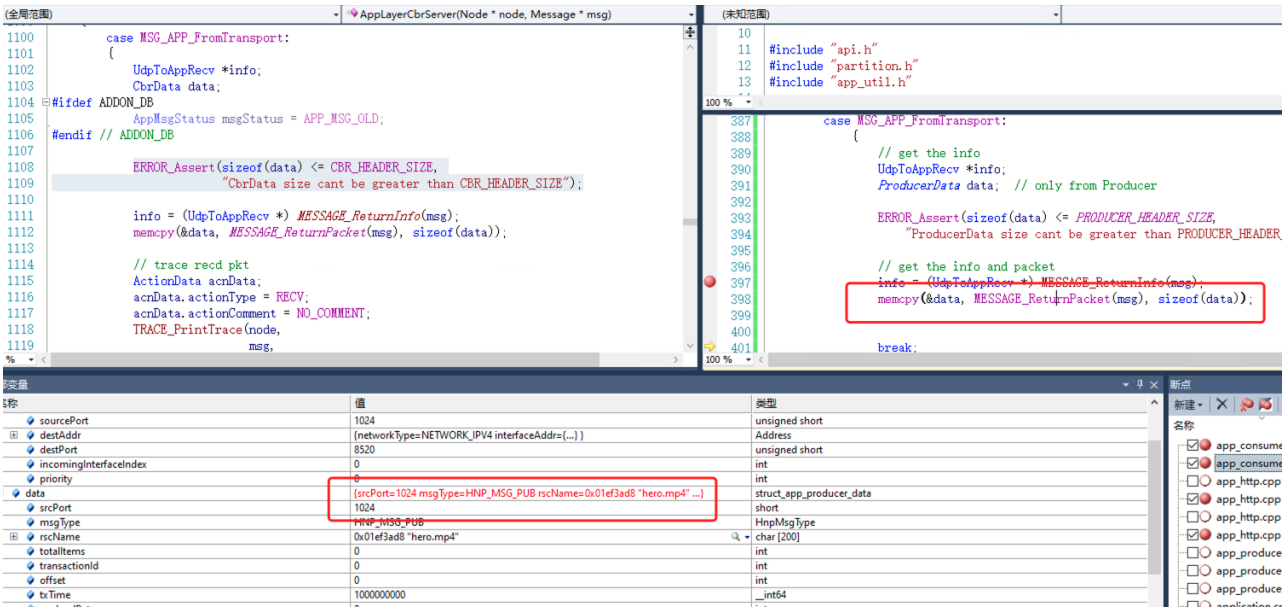
1. 在 APP_ProcessEvent （application.cpp）中添加 Producer的事件处理入口：在 AppType 的switch 判断中添加 APP_PRODUCER （#6698）的 case. 在处理中调用AppLayerProducer（node， msg）方法。【为了可以添加不同的producer，还是引入 本地 source port，用以区分不同的 Producer instance】
2. 添加 AppLayerProducer、AppProducerGetProducer 方法，在 app_producer.h 和 app_producer.h中文件中。
3. 在 AppLayerProducer 中向下交付 PUBLISH 消息，全网广播，对端接口协议为 CONSUMER （APP_CONSUMER） 【只向 Consumer 发消息】。

4. 调试在 AppLayerConsumer 收到传输从送上来的消息：消息类型为 610. 【下面进一步检查消息 Info】



5. CONSUMER 事件处理：收到 PUBLISH

1. 在 AppLayerConsumer 方法中：添加一个MSG_APP_FromTransport 消息类型处理的case。通过 MESSAGE_ReturnPacket方法可以获取传递上来的对方数据：调试发现，已收到ProducerData，即收到对端的 PUBLISH 消息。接下来，CONSUMER 应根据消息类型做不同的处理，结合 Consumer 的不同状态。



2. 定义一个 *HnpObject* 结构体，代表一个内容对象，便于Producer 或 Consumer 进行发布、保存、请求或响应【未来增加版本信息】

```
typedef
struct HNP_Object_Entry
{
    char objectName[MAX_STRING_LENGTH]; // name of object
    char producerName[MAX_STRING_LENGTH];   // name of the producer
    HnpObjectContentType contentType;   // content type of the objec
    Int32 size; // object size in bytes
    Int32 chunksNum;    // number of chunks
    Int32 chunkSize;    // chunk size in

}HnpObject;
```

3. Consumer 收到 *MSG_APP_FromTransport* 后的处理流程：
   a. 检查消息类型（默认全部来自 Producer）；
   b. 如果是来自 Producer 的 PUBLISH 消息，则调用 *HandlePublish* 方法。
   c. 如果是 REQUEST asp 消息，则调用 *HandleReqRsp* 方法。
4. *HandlePublish* 操作流程设计：
   a. 取资源类型（OBJECT vs SERVICE）；
   b. 对于 OBJECT 类型：创建 Object，保存在 Consumer 的 *ObjectList* 中；通知所有本地 Consumers 有新 Object 发布（由 Consumer 决定后续行为）【1.16】。
   c. 对于 SERVICE 类型：创建 Service 对象，保存在 Consumer 的*ServcieList* 中。
   d. Consumer 状态不变。
5. 在 AppDataConsumer 中增加一个 List 成员：List<HnpObject> objectList，用于保存获取到网络中发布的内容。
   a. 添加 hnp_list.h 和 hnp_list.cpp两个文件，参考 list.h和 list.cpp中指针链表和 整数链表的实现，实现一个 HNP Object 对象的链表：ObjInfoList。
   b. 在hnp_common.h文件中，原先定义的 HnpObject ==> ObjectInfo 类型。
   c. 在 user_models下的 Makefile_common中 添加新增的 hnp_list.cpp 文件，以便其参与编译。
   d. 在 AppDataConsumer 结构体中增加一个 ObjInfoList 参数，保存收到 Published 的对象信息，并且在 AppConsumerNewConsumer 中对该 List 进行初始化，调用 ObjInfoListInit。
   e. 每个 Consumer 保存 ObjectInfo 时可能存在的问题：当本地 Consumer 尚未创建时，系统收到了 Producer 的 Publish 消息，此时没有 ObjInfoList 可用。但如何向节点保存数据结构呢？我不愿意修改 Node 属性，改动过大。【待续！】
   f. 能不能将这个 List 设计为 static 类型，供所有的 Consumer 使用？【需要研究一下 Static 类型】
   g. 还是决定向 Node 添加一个 ObjectInfoList 成员。【再斟酌】：==》暂时把 ObjectInfoList 作为 COnsuymer 的静态成员，供所有 Consumer 访问；后面将其作为融合层（Converged Layer）的成员。
      i. 对于收到 PUBLISH 消息的AppLayer，应该首先判断本节点是否注册有 AppConsumer，如果没有，立即退出。
      ii. 添加一个方法：BOOL AppConsumerWithConsumer（Node *nodePtr），检查该节点是否安装有 Consumer 协议。

iii. Consumer 收到远端 Publish 消息后，1） 讲 Object 保存在 list 中；2）通知所有的 Local consumers， 列表有更新。

6. 封装 Consumer 发送 请求 Object 的消息：AppConsumerRequestObject（Node*， ）

6. Producer事件处理：FromTransport

在 AppLayerProducer 中添加消息事件类型：MSG_APP_FromTranspor，检查是否收到 Consumer 发来的请求消息。

基本思路：根据请求的 Object 信息，反馈带 VirtualPayload 的Chunks （数目和大小）即可，目前缺一个 Rate 的属性。

6.1 添加已发布内容列表

类似Consumer 中的ObjectInfoList，只是这里指某个 Producer 发布大内容列表，用于收到 GET 消息时核对和回应。

6.2 添加Object 请求响应 GetRsp

7. Consumer 中通过Message Info 记录 Producer 地址与端口

1. Message 中能否添加多个 Info？如何提取?
2. 增加一种INFO_TYPE_OBJECT_NAME在 meesage.h，用于传递 name of Object。
3. CONSUMER 中没有收到 GET_Rsp 消息！【待解决】
    a. APP_CONSUMER： 8520
    b. APP_PRODUCER： 6698
4. 问题的根源找到了！将 Producer 响应中的原端口号改为 APP_CONSUMER协议CONSUMER 就能接收到。因此，是Application 层利用 Consumer 的端口号无法识别协议。
5. 端口 Info 传递过程中发生变化：从 1024 ---》39000！【待找原因】
    a. 原因找到：是取 Info 参数的方法有误！取 unsigned short 类型的指针参数，应采用 *（short *）方法，而不能直接采用 （short）：

```
// Handle GET_Rsp
void
AppConsumerHandleGetResponse(Node *nodePtr, Message *msg)
{
    ERROR_Assert(nodePtr != NULL,
        "CONSUMER: Invalid Node pointer");
    ProducerData data;
    // get data from the message
    memcpy(&data, MESSAGE_ReturnPacket(msg), sizeof(data));
    ERROR_Assert(data.msgType == HNP_MSG_GET_RSP,
        "Message type is not GET Response");

    char error[MAX_STRING_LENGTH];

    // get the local port
    int payloadSize = *(int*)MESSAGE_ReturnInfo(msg, INFO_TYPE_DataSize);
    unsigned short localPort = *(short*) MESSAGE_ReturnInfo(msg, INFO_TYPE_DestPort);


    // Get the requesting consumer
    AppDataConsumer * pConsumer = AppConsumerGetConsumer(nodePtr,localPort);
```

8. 观察调用栈：Producer --》Consumer实例

1. PARTITION_ProcessPartition
2. PARTITION——RunPartition
3. NODE_ProcessEvent: node.cpp ==>调用各层的 XXX_ProcessEvents
4. APP_ProcessEvent
5. AppLayerProducer
6. AppLayerHandleGet
7. AppLayerHandleGetObject
    a. APP_UdpCreateMessage：

```
Message*
APP_UdpCreateMessage(
    Node *node,
    NodeAddress sourceAddr,
    short sourcePort,
    NodeAddress destAddr,
    short destinationPort,
    TraceProtocolType traceProtocol,
    TosType priority)
```

        i. 其中，调用 APP_UdpCreateMessage，函数名称与上面那个一样，第2个参数不同。

```
Message* APP_UdpCreateMessage(
    Node *node,
    const Address& sourceAddr,
    short sourcePort,
    const Address& destAddr,
    short destinationPort,
    TraceProtocolType traceProtocol,
    TosType priority)
{
```

ii. 在第2个函数内真正生成 MSG_TRANSPORT_FromAppSend （503）消息！

b. AppTrafficSender::appUdpSend: app_trafficSender.cpp

c. sendUdpPackets

d. sendUdpApplicationPackets

8. (1–3)
9. TRANSPORT_ProcessEvent
10. TRANSPORT_UdpLayer【transport_udp.cpp】
11. 处理 MSG_TRANSPORT_FromAppSend 消息，调用TransportUdpSendToNetwork: 添加 UDP 头部
12. NetworkIpReceivePacketFromTransportLayer：network_ip.cpp
13. NetworkIpSendRawMessage：添加 IP 头，AddIpHeader
14. RoutePacketAndSendToMac: 还在 network_ip.cpp
15. *FixedComms_DelayDrop：in fixed_comms.cpp*
16. 。。。
17. static void Mac802_3NetworkLayerHasPacketToSend 【mac.cpp】
18. *(1–3)*
19. *MAC_ProcessEvent 【】*
20. *MAC802_3Layer*
21. *MAC802_3HandleReceivedFrame*
22. MAC_HandOffSuccessfullyReceivedPacket: 传递给 网络层
23. NETWORK_ReceivePacketFromMacLayer： network.cpp
24. NetworkIpReceivePacketFromMacLayer: network_ip.cpp
25. NetworkIpReceivePacket: network_ip.cpp
26. NetworkIpSendOnBackplane
27. DeliverPaket: network_ip.cpp
28. SendToUdp: network_ip.cpp ， 这里应该是通过事件消息传递给 UDP 的

```
void
SendToUdp(
    Node *node,
    Message *msg,
    TosType priority,
    NodeAddress sourceAddress,
    NodeAddress destinationAddress,
    int incomingInterfaceIndex)
{
    NetworkToTransportInfo *infoPtr;

    MESSAGE_SetEvent(msg, MSG_TRANSPORT_FromNetwork);
    MESSAGE_SetLayer(msg, TRANSPORT_LAYER, TransportProtocol_UDP);
    MESSAGE_SetInstanceId(msg, 0);
    MESSAGE_InfoAlloc(node, msg, sizeof(NetworkToTransportInfo));

    infoPtr = (NetworkToTransportInfo *) MESSAGE_ReturnInfo(msg);

    SetIPv4AddressInfo(&infoPtr->sourceAddr, sourceAddress);
    SetIPv4AddressInfo(&infoPtr->destinationAddr, destinationAddress);

    infoPtr->priority = priority;
    infoPtr->incomingInterfaceIndex = incomingInterfaceIndex;

    MESSAGE_Send(node, msg, PROCESS_IMMEDIATELY);
}
```

29. 顺便发现了这个：

```
// /**
// CONSTANT    :: PROCESS_IMMEDIATELY  : 0
// DESCRIPTION :: Used to prioratize a process
// **/
#define PROCESS_IMMEDIATELY ((clocktype) 0)
```

30. (1-3)

31. Transport_ProcessEvent: transport.cpp

32. TransportUdpLayer：这里对 MSG_TRANSPORT_FromNetwork事件进行处理

```
197  void
198  TransportUdpLayer(Node *node,  Message *msg)
199  {
200      /* Retrieve the pointer to the data portion which relates
201         to the TCP protocol. */
202
203      switch (msg->eventType)
204      {
205          case MSG_TRANSPORT_FromNetwork:
206          {
207              TransportUdpSendToApp(node, msg);
208              break;
209          }
210          case MSG_TRANSPORT_FromAppSend:
211          {
212              TransportUdpSendToNetwork(node, msg);
213              break;
214          }
215          default:
```

33. *TransportUdpSendToApp*(Node *node, Message *msg)：把数据上传给 Application
    a. 设定　MESSAGE_SetEvent(msg, MSG_APP_FromTransport);
    b. 通过消息机制，通知 Application层
34. （1-3）略
35. APP_ProcessEvent：application.cpp
    a. 提取应用层协议：　protocolType = *APP_GetProtocolType*(node,msg);
    b. 通过switch 转到不同应用层协议进行处理：

```
    switch(protocolType)
    {
        case APP_TIMER:
        {
            AppLayerTimerTest(node, msg);
            break;
        }
        case APP_ROUTING_BELLMANFORD:
        {
            RoutingBellmanfordLayer(node, msg);
            break;
        }
```

    c. APP_CONSUMER

```
68          }
69          // LuoJT: CONSUMER
70          case APP_CONSUMER:
71          {
72              AppLayerConsumer(node, msg);
73              break;
74          }
```

36. AppLayerConsumer：app_consumer.cpp

a. 提取消息的事件类型：可能是MSG_APP_TimerExpired，也可能是 MSG_APP_FromTransport

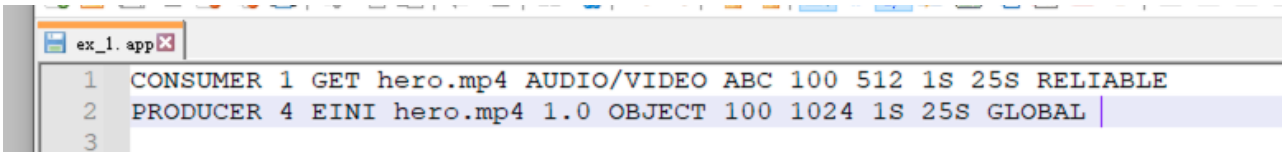
b. 处理 MSG_APP_FromTransport。

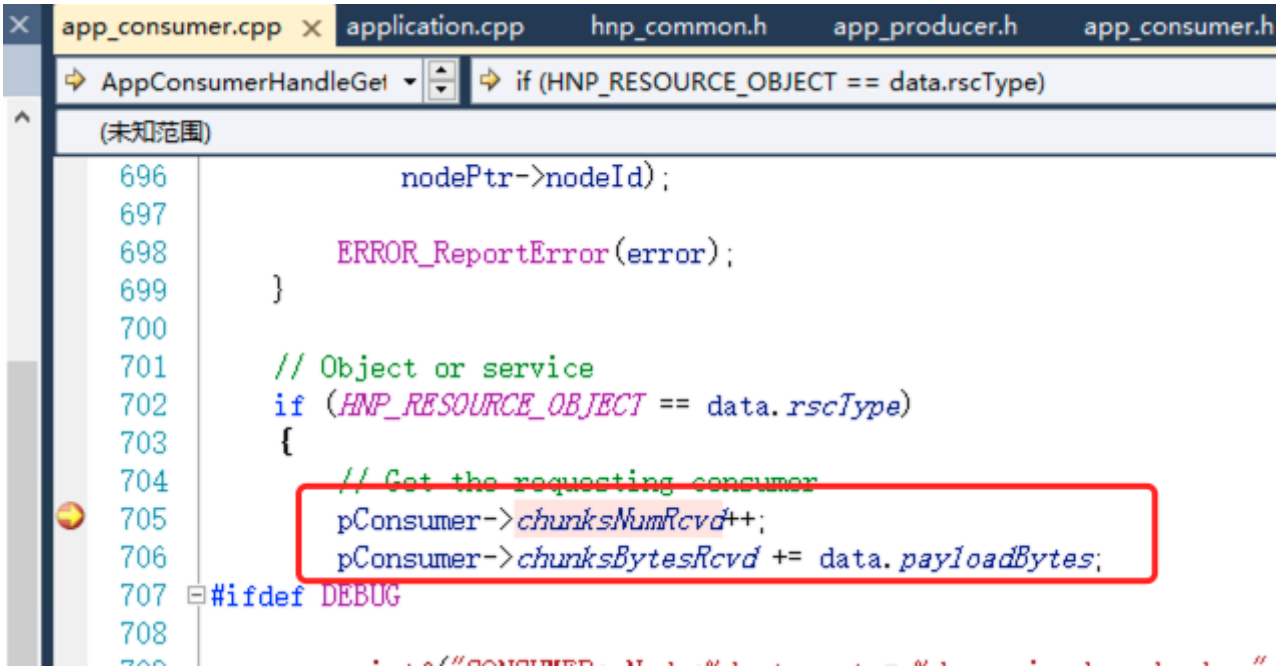
37. 至此，完成了数据从一个节点的应用层（Prodcuer）传输至了另一个节点的应用层（Consumer）！

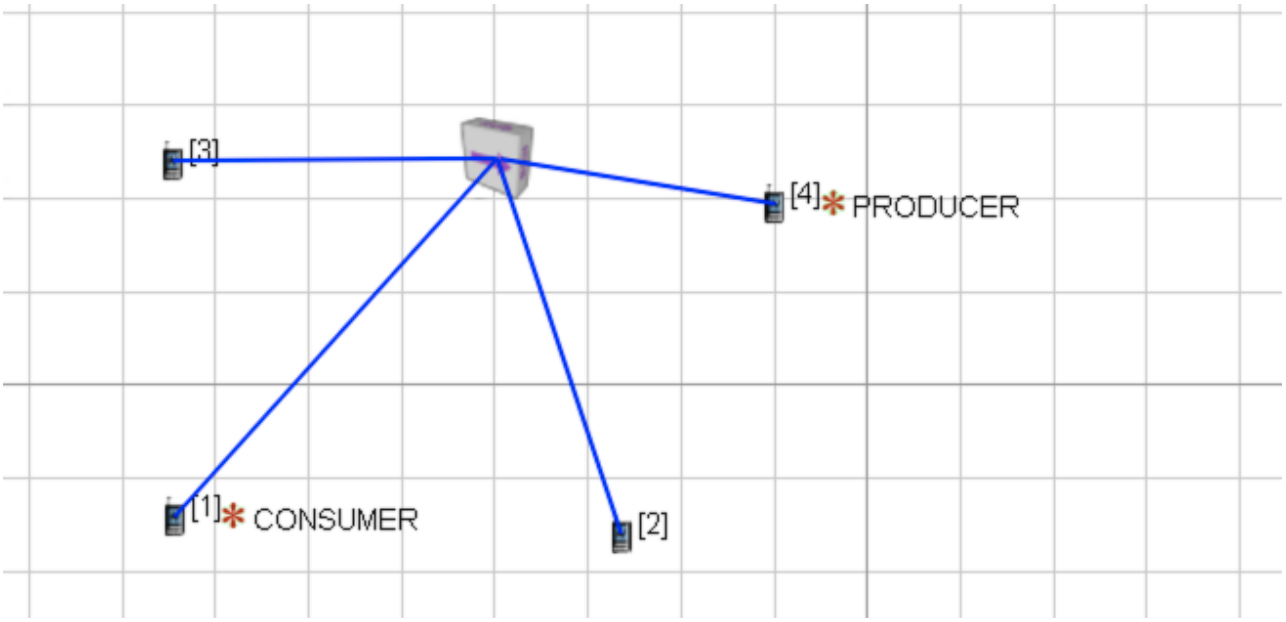
9. 跨域通信

之前广播模式下（比如，PUBLISH 消息），如果不是一个局域网，发现Consumer 和 Producer 之间无法通信，进行下一步工作之前，首先确认并解决这一问题。

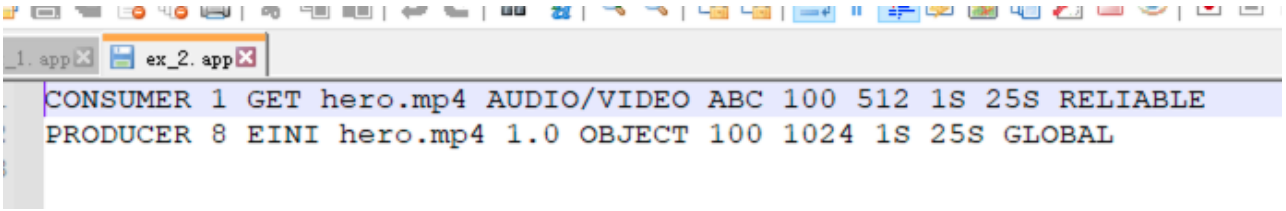
1. **验证目前状态**：Producer 向全网发布一个Object，Consumer 请求该 Object，包含多个 Chunks，Consumer 可以向该 Producer 发布请求，并可以连续收到所有Chunks。

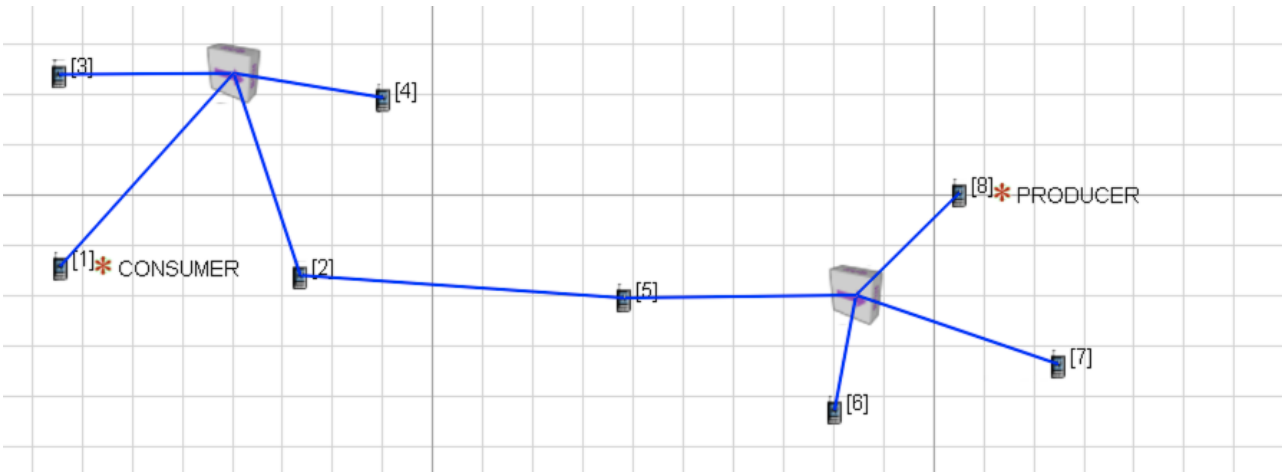






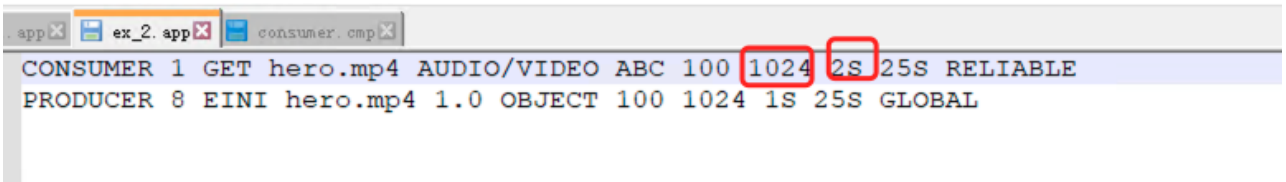
2. 调整网络拓扑，不再是一个LAN：修改为以下拓扑，另存为ex_2.config





3. 目前Consumer 和 Producer 默认chunk大小不同（分别为512和1024），先修改为一致

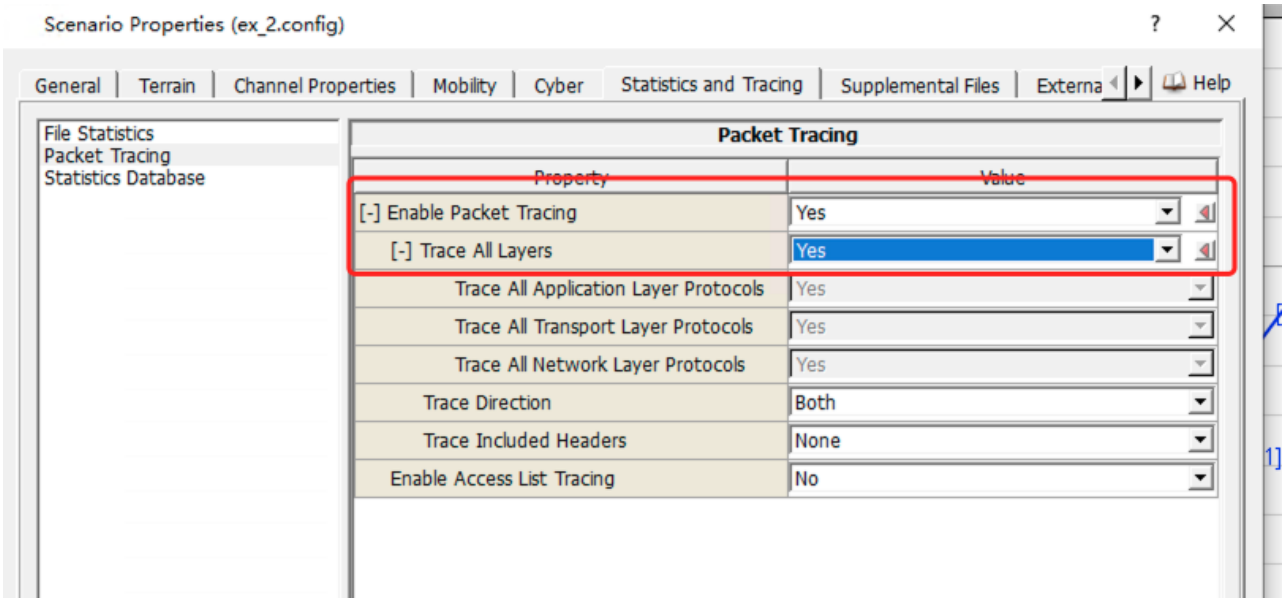
将Consumer 请求的 Chunk Size，默认值修改为 1024，与 Producer 默认一致【接口的进一步细化，留后】，并修改Consumer 的 Start Time 为 2sec，此时， ex_2.app 如下：



4. 打开场景的 Packet TRACE功能

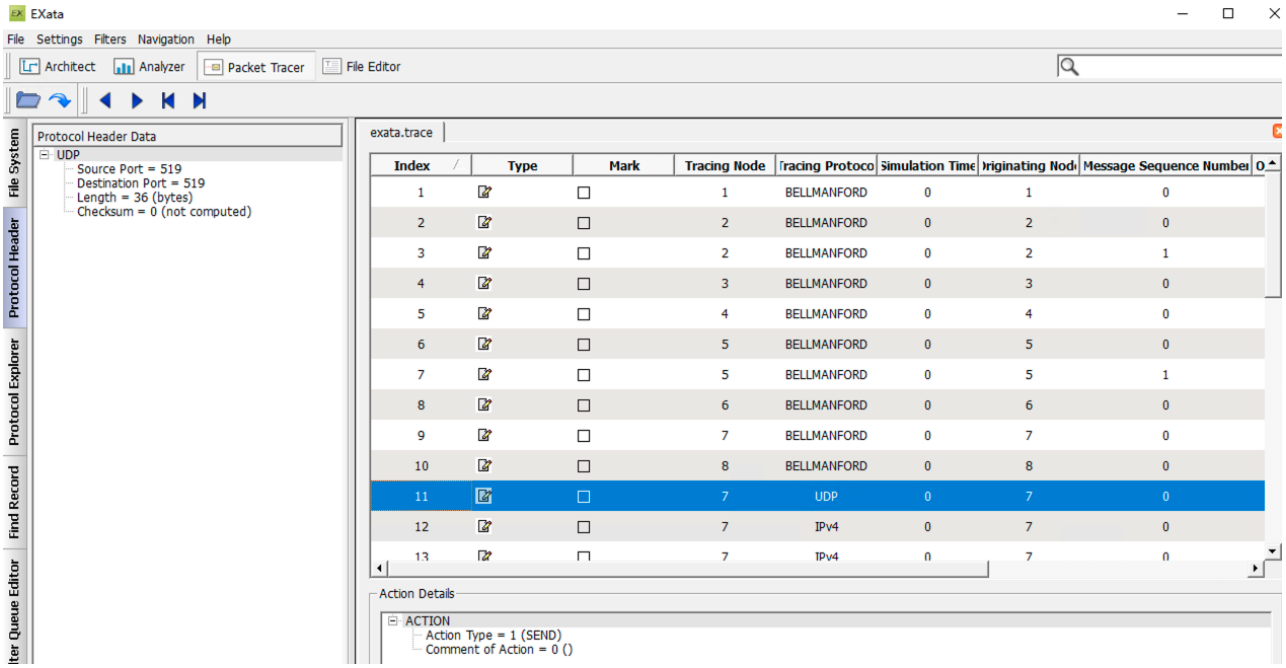
a. 打开场景的config 文件，在“PACKET TRACING”部分，将“PACKET-TRACE”参数改为“YES”，并添加“TRACE-ALL YES”，这样打开 TRACE 开关，并TRACE 所有协议。【暂时还是看不到应用层协议】。 或

者通过GUI 设置，在场景全局属性“Statistics and Tracing”页，打开 开关

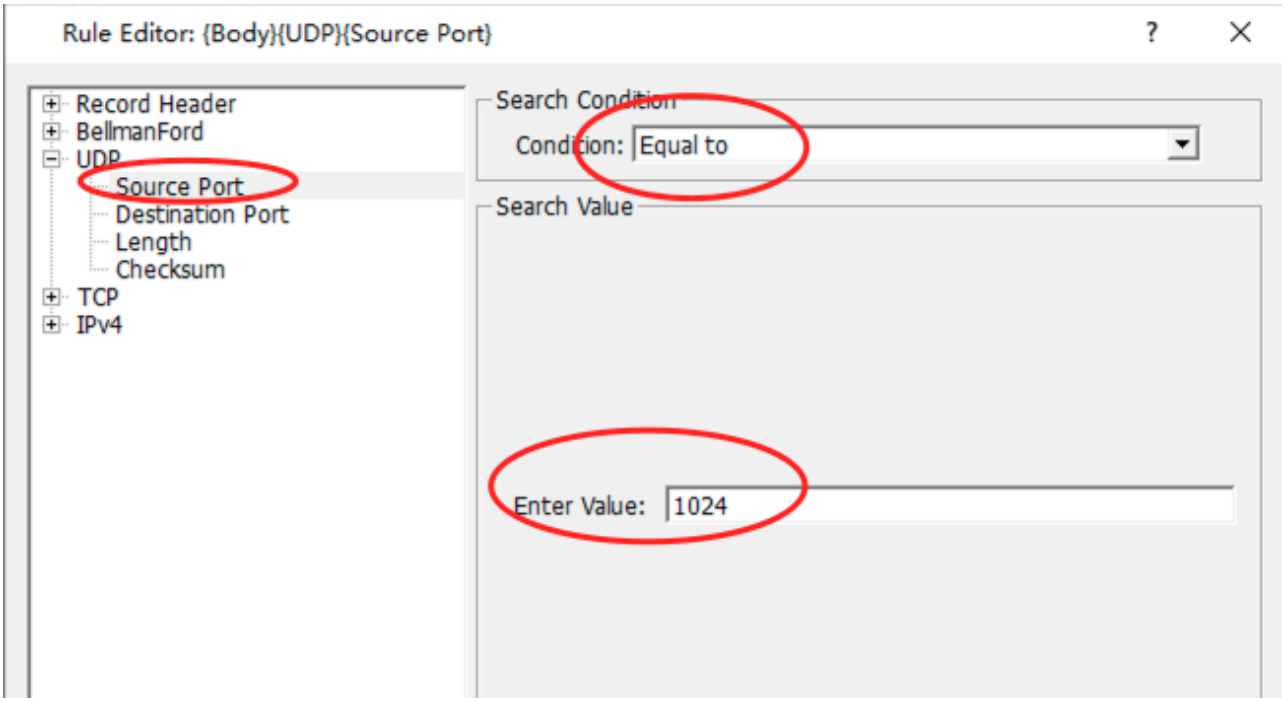


b. 命令行运行 ex_2，“exata ex_2.config –simulation”，此时工作目录下将会生成：exata.trace 文件。

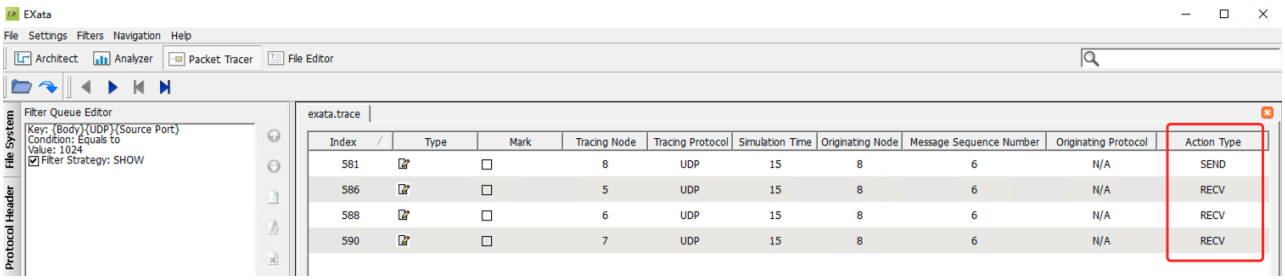
c. 用 EXata 的 Packet Tracer 打开改文件



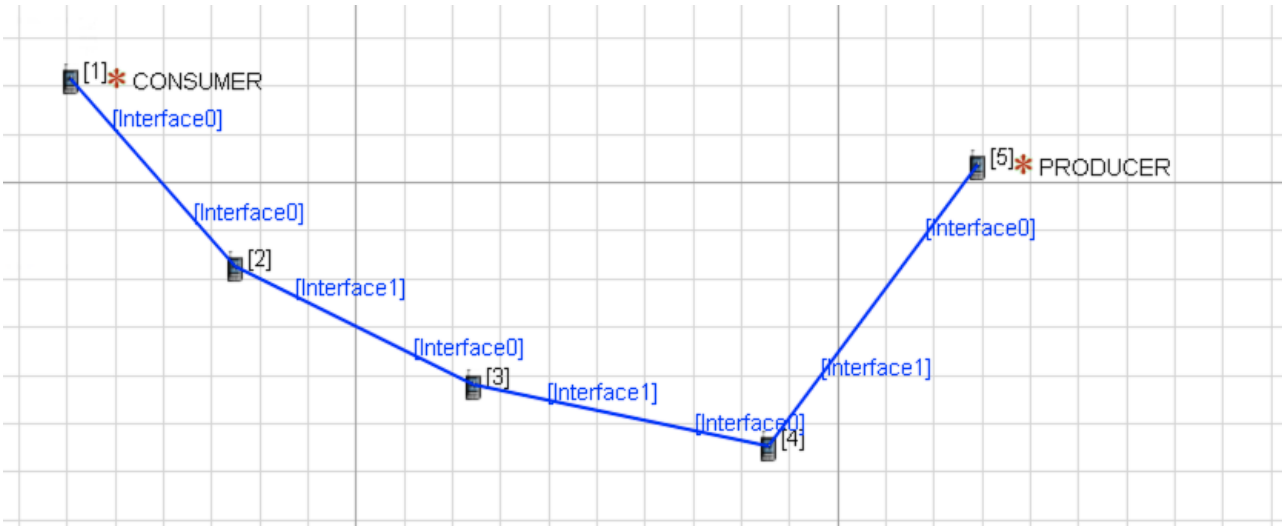
d. 可以添加过滤器查看想要观察的包：比如Producer PUBLISH 的消息的源端口为 1024，可以过滤UDP source port = 1024 的所有包，过滤器在“Filter Queue Editor”中编辑过滤器，结果如下：



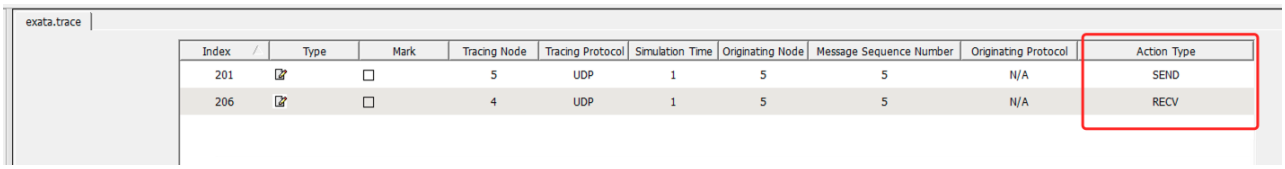
e. 过滤结果如下：可以发现只有 4 个包，而且是节点8发出，节点 5，6，7接收，节点1–2–3–4并没有收到。对照前面的网络拓扑，可知CONSUMER不可能接收到。【为什么节点 5 收到后没有继续前传？路由为什么没建立起来？】



f. 为简化起见，去掉子网，线形拓扑



g. TRACE 结果发现，仍然只传递了一跳：节点5--》节点4



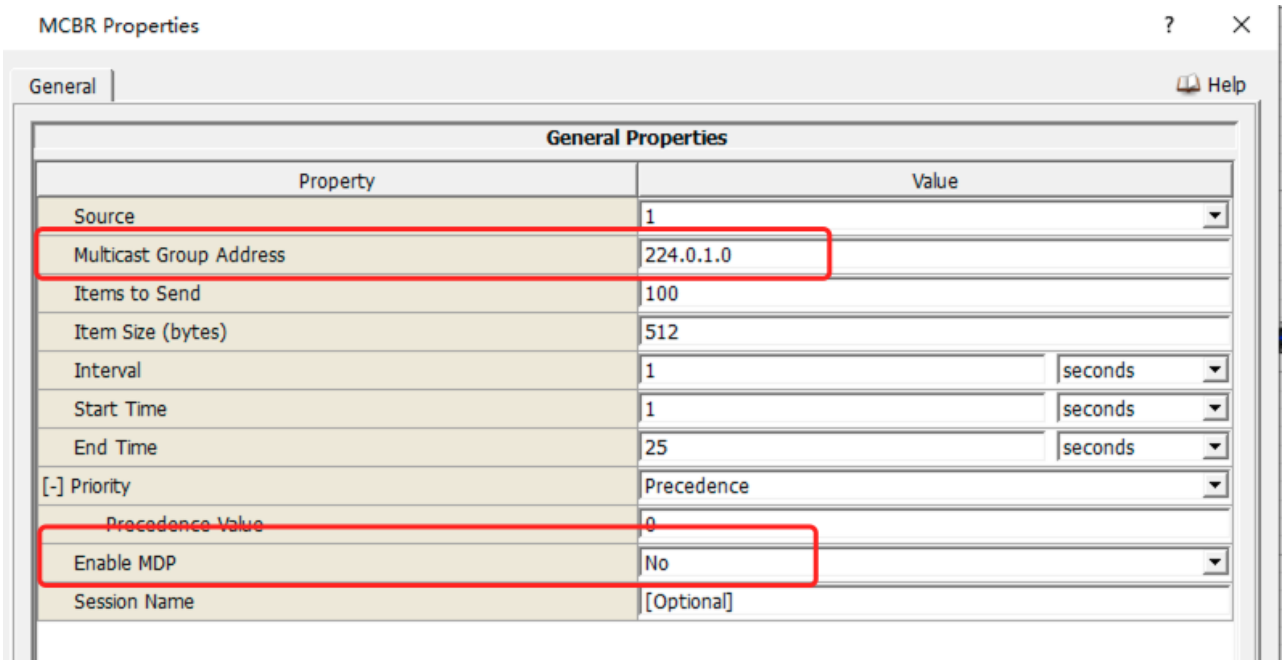
h. 推断如下：在 Publish 一个 Object 时，目的地址采用：ANY_ADDRESS，而destPort 采用

APP_CONSUMER，本意是发给任意节点的 CONSUMER协议，而实际上，Producer 所在节点只能发给所有的相邻节点，而相邻节点无法继续前传。后面需要研究下传输层协议的实现方式，或者研究如何让 IP 层继续前传（无论本地是否有 CONSUMER）。【ANY_ADDRESS ： = 0xFFFFFFFF，也是 IPv4 最大的组播地址，按道理应该继续广播的】

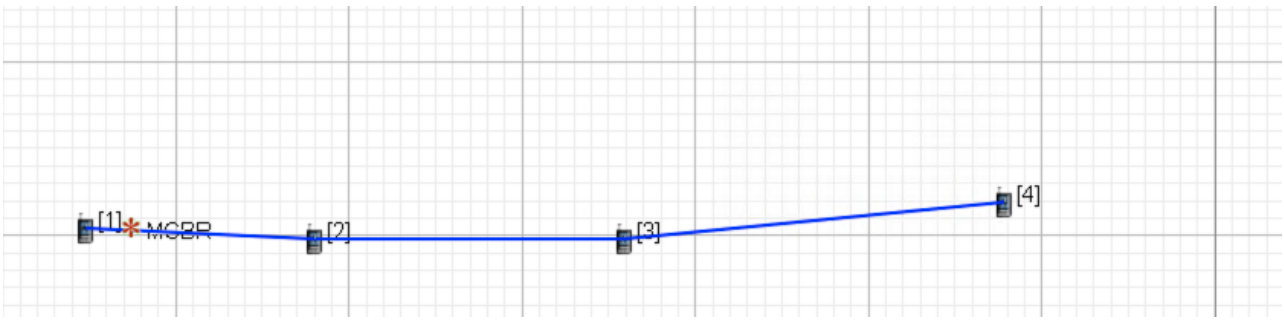
i. 查看 MDP 协议

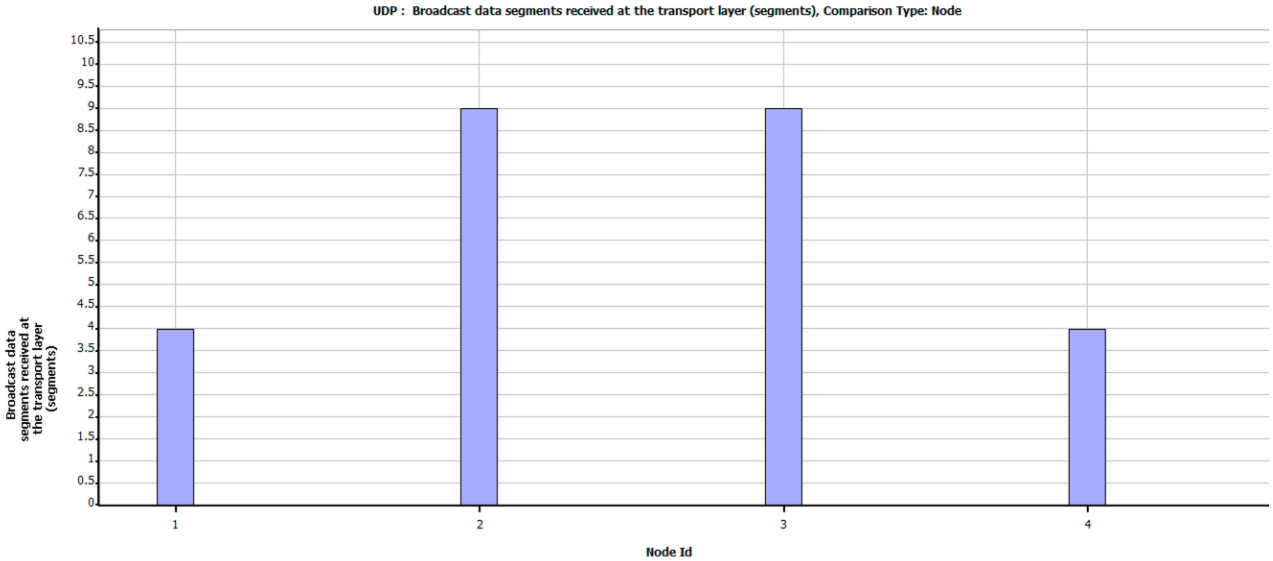
j. 查看 MCBR，建立一个MCBR 场景：线形拓扑，它能够实现多跳自动广播，考虑能否借鉴其实现方式

k. MCBR 的属性中有一个组播地址（Multicast Group Address），注意：IPv4 组播地址范围是224.0.0.0 ～ 255.255.255.255 ； 但默认不需要激活 MDP：

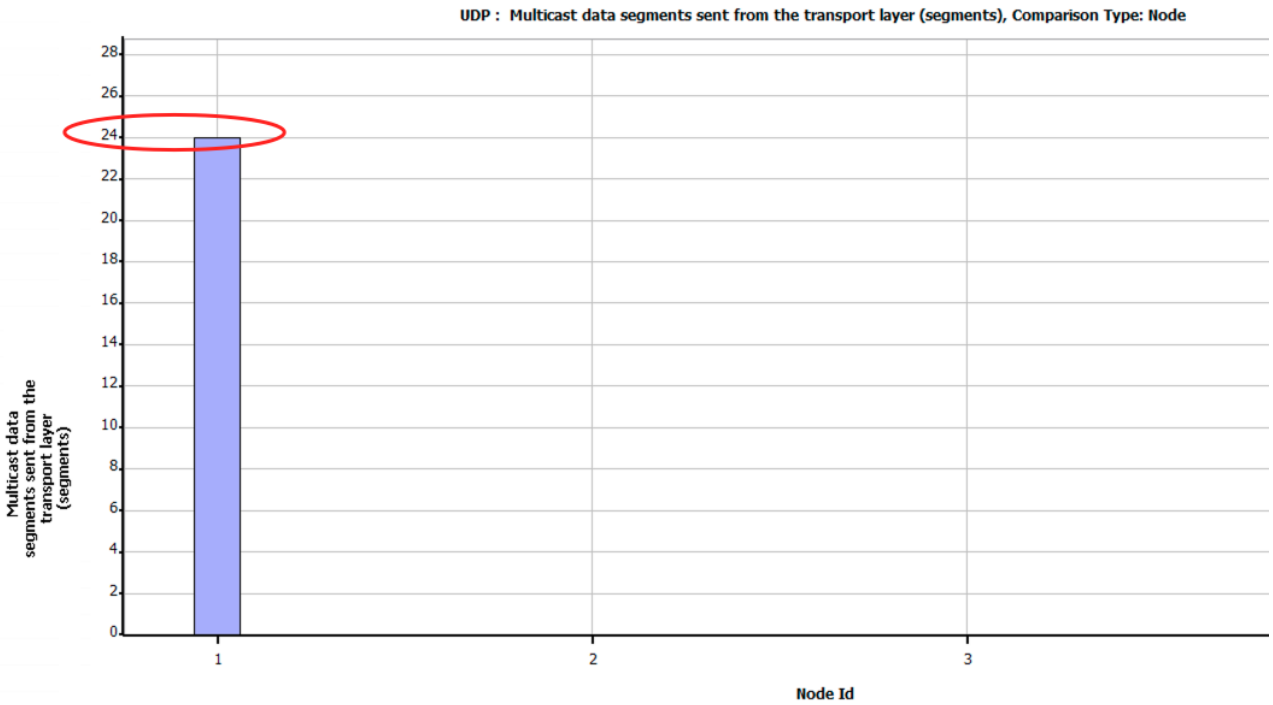


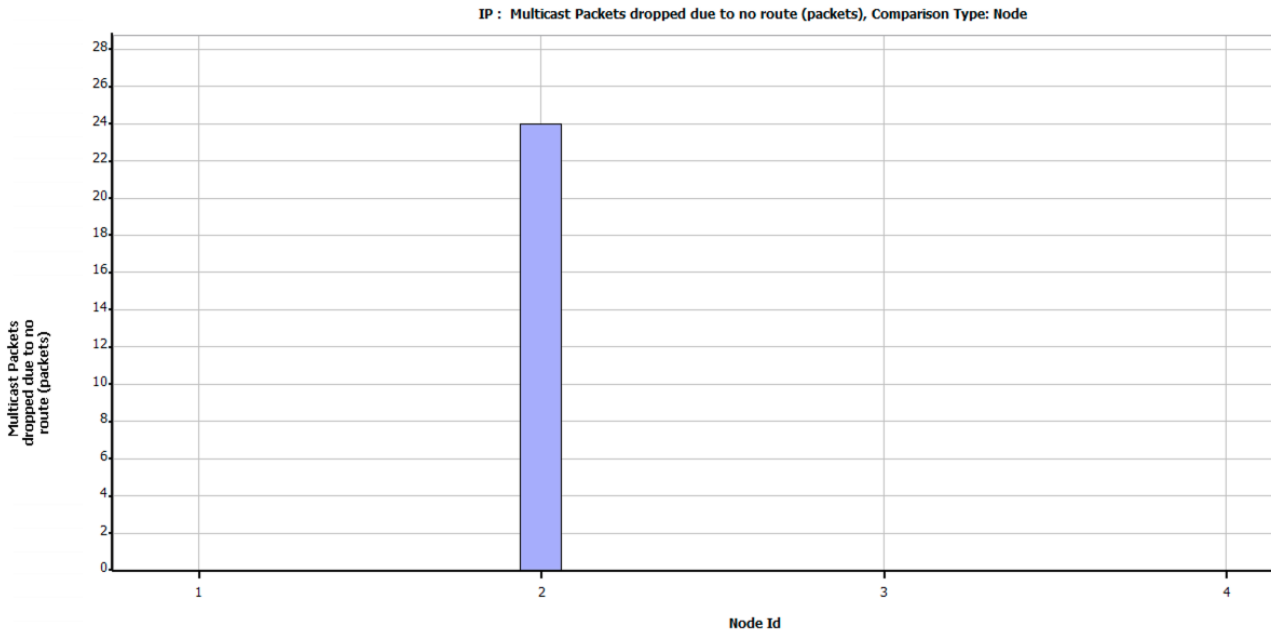
l.





i. 上图有误，Broadcast 消息主要是Bellman–Ford 路由包，并非 MCBR 数据包，事实上，节点1共发出 MCBR 数据组播包 24 个，但在节点 2 的网络层都被丢弃了





ii. 上图表明24个组播包由于no route 而在节点2被丢弃！

10. MDP协议

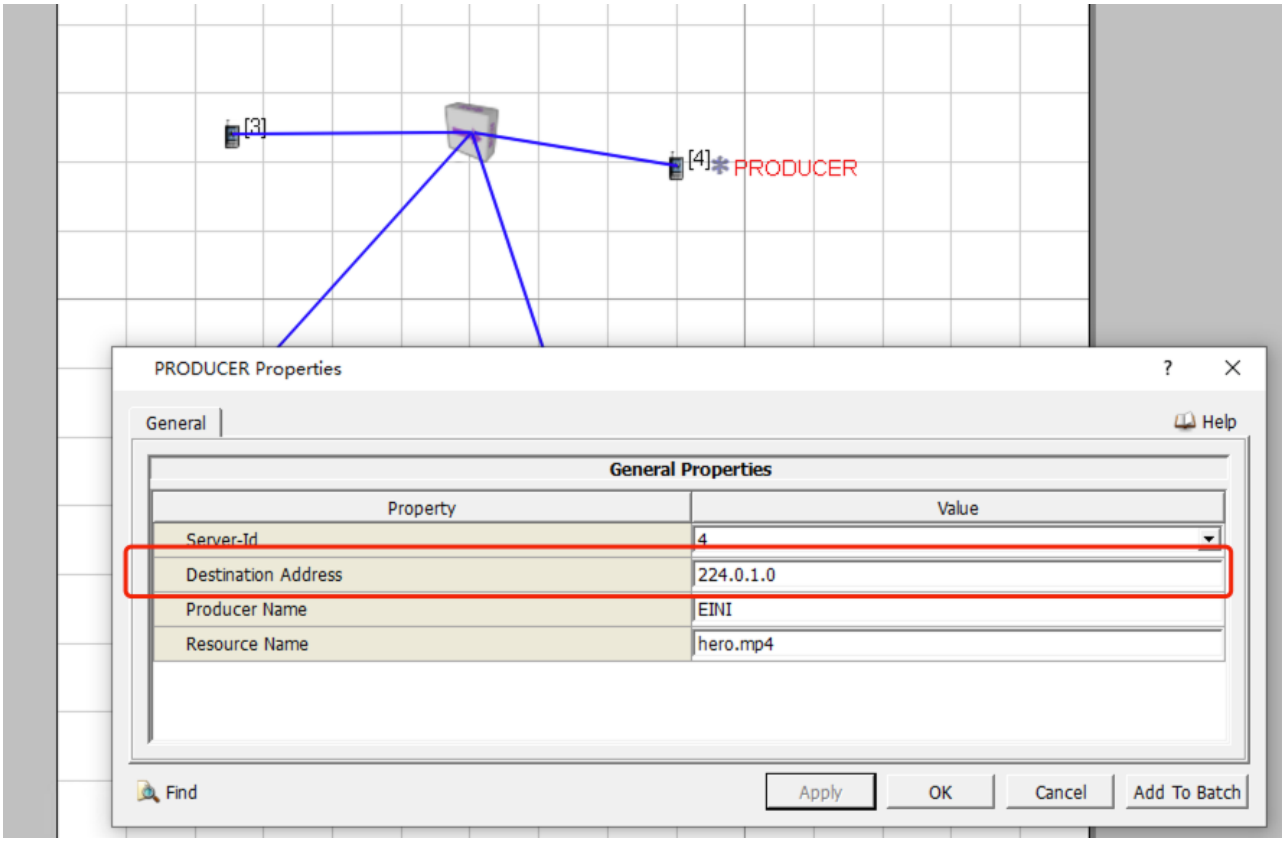
参考《Developer Model Library》之 Multicast Dissemination protocol。

11. GUI 显示问题解决

a. 【GUI 添加协议显示问题终于解决！】分析原因：自编协议的目的地址不能为空，添加一个广播地址，即可解决。在 producer.cmp 的第二行添加目的地址参数【注意：必须是第二行】，如下：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root version="1.0">
<category name="PRODUCER Properties" singlehost="true" loopback="disabled" propertytype="PRODUCER-SINGLEHOST"
displayname="PRODUCER">
   <variable name="Server-Id" key="SOURCE" type="SelectionDynamic" keyvisible="false" optional="false" help_ref=
   "TBD:TBD." />
  <!--  <variable name="Producer Name" key="PRODUCER" type="Text" default="EINI" spacesAllowed="false"
  optional="false" keyvisible="false" help="creater of the resource"/>
    <variable name="Resource Name" key="RESOURCE" type="Text" default="hero.mp4" spacesAllowed="false"
    optional="false" keyvisible="false" help="name of the publishing resource" />
    <variable name="Resource Version" key="VERSIOIN" type="Text" default="1.0" spacesAllowed="false"
    optional="false" keyvisible="false" help="version of the publishing resource" />
    <variable name="Resource Type" key="RESOURCE-TYPE" type="Selection" default="OBJECT" keyvisible="false"
    optional="false" help_ref="TBD:TBD." >
      <option value="OBJECT" name="Object">
      </option>
      <option value="SERVICE" name="Service" >
      </option>
  </variable>
    <variable name="Number of Chunks" key="CHUNKS-NUMBER" type="Integer" default="100" min="1" keyvisible="false"
```

b. 此时，属性对话框显示如下（其余熟悉暂时未恢复）： 此时，保存场景并关闭，应用程序没有问题都能保存。



c. 修改 app_producer.cpp 参数读取部分

i. 在App_InitializeApplications中读取配置参数时，添加一个参数：Remote Address；

ii. In app_producer.h，在 AppDataProducer 中添加新成员： remoteAddr ；

iii. In app_producer.h：AppProducerInit 和 AppProducerNewProducer 接口中添加 remoteAddr 参数。

12. 添加统计量

问题，如何确定 Session Id? 在统计量初始化时需要此值，以便进行 Session 层面的统计。