

EXata扩展（七）：添加一个传输层协议

目标：为实现传输层的融合协议，先练习实现传输层新协议

参考：Programmer Guide 4.3 Transport Layer Protocol

1. 传输层的数据结构

传输层的主要数据结构定义在 transport.h (include 目录下)。

- TransportProtocol：枚举所支持的传输层协议类型

```
// /**
// ENUM      :: TransportProtocol
// DESCRIPTION :: Enlisting different transport layer protocol
// **/
enum TransportProtocol
{
    TransportProtocol_UDP,
    TransportProtocol_TCP,
    //InsertPatch TRANSPORT_ENUMERATION
    TransportProtocol_RSVP
};
```

- TransportData：主数据结构

```
// /**
// STRUCT      :: TransportData
// DESCRIPTION :: Main data structure of transport layer
// **/
struct TransportData {
    TransportDataUdp* udp;
    // TransportDataTcp* tcp;
    int tcpType; // It may be TCP_REGULAR or TCP_ABSTRACT
    void* tcp;
    BOOL rsvpProtocol;
    void *rsvpVariable;
#ifdef ADDON_NGCNMS
    TransportResetFunctionList* resetFunctionList;
#endif
};
```

2. 传输层 API 与层间通信

2.1 应用层到传输层

应用层协议采用 API 与传输层协议通信，这些API定义在/main/api_util.cpp 中。比如

APP_UdpSend，就是应用层用来发送 UDP 消息；这些 API 主要通过 Message 来实现。常用的消息有：

- **MSG_TRANSPORT_FromAppSend**: used by Application Layer to **send data to Transport Layer**
- **MSG_TRANSPORT_FromAppListen**: used by the Application Layer to **direct TCP to listen on a port**.
- **MSG_TRANSPORT_FromAppOpen**: used by the Application Layer to request TCP to open a TCP connection.
- **MSG_TRANSPORT_FromAppClose**: used by the Application Layer to request TCP to close a TCP connection.

注意：EXata 在实际实现时，往往又把这些 API 出现进行封装，比如封装到 AppTrafficSender 类（in app_trafficSender.cpp）。

2.2 传输层到应用层

传输层协议通过**消息**与应用层协议通信。这些消息类型定义在 api.h文件中，常用的有：

- **MSG_APP_FromTransport**：UDP用来传送输入的包给上面的应用层协议。
- **MSG_APP_FromTransOpenResult**：TCP 用来通知应用层新建连接请求的结果：接受或拒绝
- **MSG_APP_FromTransDataSent**：TCP 用来给应用层协议指示数据已发出
- **MSG_APP_FromTransDataReceived**：TCP用来给应用层协议指示收到数据包
- **MSG_APP_FromTransListenResult**：TCP用来通知应用层服务器收到打开 TCP 连接的请求
- **MSG_APP_FromTransCloseResult**：TCP 用来通知应用层客户端或服务端TCP连接已关闭

应用层在ProcessEvent 中根据消息类型进行分别处理。

2.3 传输层跟网络层通信（ ↓ ）

传输层调用 API：**NetworkIpReceivePacketFromTransport**，与网络层 IP 协议通信，其实现在 network_ip.cpp中。

2.4 网络层跟传输层通信(↑)

IP 协议通过几个 API 跟传输层协议通信：

- SendToUdp
- SendToTcp
- SendToRsvp
- SendToTransport

这几个函数原型声明在 network_ip.h，实现在network_ip.cpp。这些 API 通过 MSG_TRANSPORT_FromNetwork实现。

3. 添加一个新的传输层协议：MYTRANS

3.1 创建文件

在libraries/user_models /src 中添加 transport_mytrans.h和 transport_mytrans.cpp两个文件。

3.2 添加新传输层协议类型

- 添加传输层协议枚举类型

在transport.h 中的 TransportProtocol 枚举类型中添加新的协议，以便在节点配置时选用。

```
4 // /**
5 // ENUM      :: TransportProtocol
6 // DESCRIPTION :: Enlisting different transport layer protocol
7 // **/
8 enum TransportProtocol
9 {
10     TransportProtocol_UDP,
11     TransportProtocol_TCP,
12     //InsertPatch TRANSPORT ENUMERATION
13     TransportProtocol_RSVP,
14     // LuoJT: exercise
15     TransportProtocol_MYTRANS
16 };
```

- 添加Trace 协议类型

在 include/trace.h 中添加trace 的协议类型，TRACE_MYTRANS.

```
425
426 // LuoJT: CONSUMER and PRODUCER
427 TRACE_CONSUMER,
428 TRACE_PRODUCER,
429
430 // LuoJT: MYTRANS
431 TRACE_MYTRANS,
432
```

- 添加传输协议状态指针

每个传输层协议的状态保存在TransportData 中，新协议需要在其中添加新的协议状态指针，和使能指示。

```

83 // /**
84 // STRUCT      :: TransportData
85 // DESCRIPTION :: Main data structure of transport layer
86 // **/
87 struct TransportData {
88     TransportDataUdp* udp;
89     // TransportDataTcp* tcp;
90     int tcpType; // It may be TCP_REGULAR or TCP_ABSTRACT
91     void* tcp;
92     BOOL rsvpProtocol;
93     void *rsvpVariable;
94
95     // LuoJT: state
96     BOOL mytransEnabled; // Flag to indicate if MYTRANS is enabled
97     void *mytransVariable; // Pointer to MYTRANS's state
98
99 #ifdef ADDON_NGCNMS
100     TransportResetFunctionList* resetFunctionList;
101 #endif
102 };

```

3.3 初始化

3.3.1 确定协议配置格式

一个协议可以通过特有的配置参数进行配置，这些配置参数通过场景配置文件进行设置。具体格式如下：

[<Identifier>] <Parameter-name> [<Index>] <Parameter-value>

对于非必需的协议可以设定其激活状态。对于本例子有两个配置参数：

- TRANSPORT-PROTOCOL-MYTRANS YES/NO 【是否激活】
- MYTRANS-STATISTICS YES/NO 【是否进行统计】

3.3.2 读取配置参数进行初始化

EXata中，协议栈初始化自下而上（bottom up）按顺序完成。节点初始化方法

PARTITION_InitializeNodes 调用TRANSPORT_Initialize方法（in transport.cpp）进行传输层的初始化。TRANSPORT_Initialize读取场景配置文件，并调用各协议的初始化方法，把这些配置参数保存在各协议的数据结构中。具体方法如下：

- 如果该协议是必需的，则直接调用它的初始化方法。比如 UDP，在TRANSPORT_Initialize 中则直接调用 TransportUdpInit 完成 UDP 协议的初始化。
- 如果该协议是非必需的，则首先通过搜索 TRANSPORT-PROTOCOL-<Protocol_name> 来确定该协议是否激活。该参数的布尔值对应它是否激活。如果该协议激活，则调用其初始化方法进行初始化。搜索通过调用 IO_ReadString 函数实现字符串参数读入。

本例子属于非必需协议，因此首先判断“TRANSPORT-PROTOCOL-MYTRANS”是否为 TRUE；该参数为 TRUE 时，调用MYTRANS 的初始化方法（待实现）。

代码修改在 transport.cpp （main 目录下）中的 TRANSPORT_Initialize方法中。代码插入如下：

```
94 // InsertPatch TRANSPORT_INIT_CODE
95
96 // LuoJT: for MYTRANS
97 node->transportData.mytransVariable = NULL; // specified before initialization
98
99 // read enabled parameter
100 IO_ReadString(node->nodeId, ANY_ADDRESS, nodeInput,
101 "TRANSPORT-PROTOCOL-MYTRANS", &wasFound, buf);
102
103 if (wasFound)
104 {
105     if (0 == strcmp(buf, "YES"))
106     {
107         - node->transportData.mytransEnabled = TRUE; // enabled
108           TransportMytransInit(node, nodeInput); // Initializing
109     }
110     else
111     if (0 == strcmp(buf, "NO"))
112     {
113         node->transportData.mytransEnabled = FALSE;
114     }
115     else
116     {
117         ERROR_ReportError("Expeting YES or NO for "
118 "TRANSPORT-PROTOCOL-MYTRANS parameter\n");
119     }
120 }
121 else // not found
122 {
123     node->transportData.mytransEnabled = FALSE;
124 }
125 }
```

注意：其中的 TransportMytransInit 待实现。

3.3.3 实现协议初始化

一个协议的初始化通常完成以下功能：

- 初始化协议状态，保存用户配置参数；
- 创建协议实例；
- 初始化数据结构和必须的变量，比如内存表、默认值等；
- 给自己设定一个timer，以启动协议。

具体如下：

1. 编译新协议源码

- a. 在 user_models/src/下的Makefile-common中添加 transport_mytrans.cpp

```
Makefile-common x node.h transport.cpp transport.h trace.h tr
USER_MODELS_OPTIONS =

USER_MODELS_DIR = ../libraries/user_models
USER_MODELS_SRCDIR = ../libraries/user_models/src

#
# 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_SRCDIR)/hnp_list.cpp \
$(USER_MODELS_SRCDIR)/transport_mytrans.cpp

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

2. 定义协议头部

这里暂时模仿 UDP 头部

```
13 typedef struct { /* MYTRANS header */
14     unsigned short sourcePort; /* source port */
15     unsigned short destPort; /* destination port */
16     unsigned short length; /* length of the packet */
17     unsigned short checksum; /* checksum */
18 } TransportMytransHeader;
```

3. 定义状态数据结构

在transport_mytrans.h中，声明如下状态数据结构

```
2
3 struct TransportDataMytransStruct {
4     BOOL mytransStatsEnabled; /* whether to collect stats */
5     BOOL traceEnabled;
6
7     STAT_TransportStatistics* newStats;
8
9 };
n
```

4. 实现 TransportMytransInit 方法

涉及两个辅助方法：TransportMytransPrintTrace 和 一个静态函数

TransportMytransInitTrace。

```
1 ...
2 // create an instance
```

```

3      TransportDataMytrans* mytrans =
4          (TransportDataMytrans*)MEM_malloc(sizeof(TransportDataMytrans));
5
6  // install to the node
7      node->transportData.mytransVariable = mytrans;
8
9  // Initialize trace
10     TransportMytransInitTrace(node, nodeInput);
11
12 // set statistics state
13     IO_ReadBool(node->nodeId,
14                 ANY_ADDRESS,
15                 nodeInput,
16                 "MYTRANS-STATISTICS",
17                 &retVal,
18                 &readVal);
19
20     if (!retVal || !readVal)
21     {
22         mytrans->mytransStatsEnabled = FALSE;
23     }
24     else if (readVal)
25     {
26         mytrans->mytransStatsEnabled = TRUE;
27         mytrans->newStats = new STAT_TransportStatistics(node,
28 "mytrans");
29     }
30     else
31     {
32         printf("TransportMytrans unknown setting (%s) for "
33             "MYTRANS-STATISTICS. \n", buf);
34         mytrans->mytransStatsEnabled = FALSE;
35     }
36 }
37 ... ..

```

5. 初始化定时器（可选）

根据协议需要，初始化方法中可能对自己需要设定一个定时器。

3.4 实现事件触发器 (Event Dispatcher)

MYTRANS 的事件处理，由Transport_ProcessEvent中派发，定义在transport.cpp。

1. 修改Transport 层的事件派发

在 Transport_ProcessEvent中增加 TransportProtocol_MYTRANS 的 case，调用 TransportMytransLayer方法。

对于非必需协议，在事件派发中首先确认该协议是否激活。

```
4 //InsertPatch LAYER_FUNCTION
5 // LuoJT: MYTRANS
6 case TransportProtocol_MYTRANS:
7 {
8     if (FALSE == node->transportData.mytransEnabled)
9     {
10         ERROR_ReportError("MYTRANS is not not enabled\n");
11     }
12     // call mytrans event dispatcher
13     TransportMytransLayer(node, msg);
14     break;
15 }
16 default:
17     assert(FALSE); abort();
18     break;
```

2. 实现 TransportMytransLayer 方法

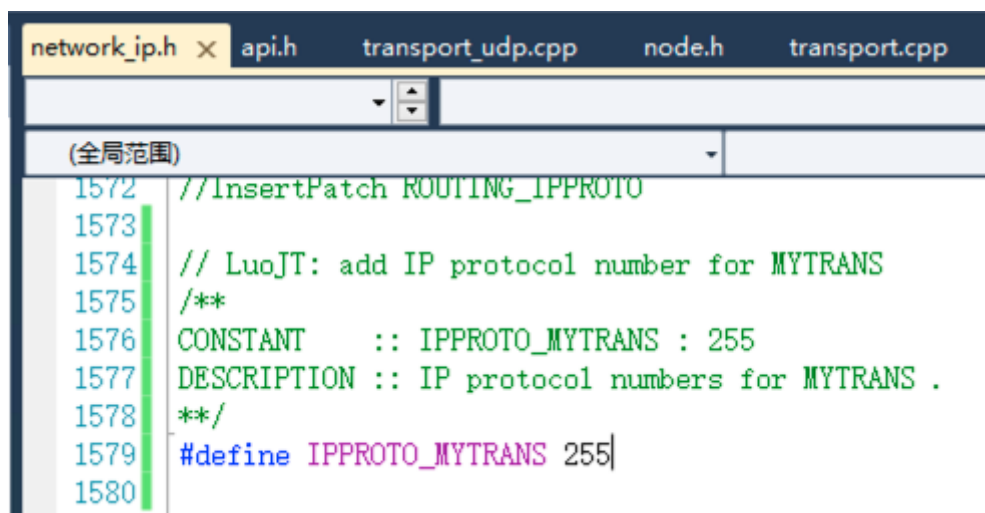
a. 针对不同消息，调用不同的方法：

```
void TransportMytransLayer(Node *node, Message *msg)
{
    // depend the event type of message

    switch (msg->eventType)
    {
        case MSG_TRANSPORT_FromNetwork:
        {
            TransportMytransSendToApp(node, msg);
            break;
        }
        case MSG_TRANSPORT_FromAppSend:
        {
            TransportMytransSendToNetwork(node, msg);
            break;
        }
        default:
            assert(FALSE);
            abort();
    }
}
```

b. 为 MYTRANS 添加 IP Protocol Number，以便 IP 协议能识别该新协议。注意：协议号不得与已有的声明冲

突！在 network_ip.h 中添加常数声明如下：



```
1572 //InsertPatch ROUTING_IPPROTO
1573
1574 // LuoJT: add IP protocol number for MYTRANS
1575 /**
1576 CONSTANT    :: IPPROTO_MYTRANS : 255
1577 DESCRIPTION :: IP protocol numbers for MYTRANS .
1578 **/
1579 #define IPPROTO_MYTRANS 255
1580
```

c. 实现两个方法：

i. **TransportMytransSendToApp**: 响应 FromNetwork 消息，上送给 App。主要完成以下工作：

1. 从消息中读取：源地址、目的地址及输入接口Id；
2. 读取传输层的端口号：利用 MESSAGE_ReturnPacket；
3. 给该消息添加新的 Info 字段送给 APP：利用 MESSAGE_IMESSAGE_InfoAlloc；
4. 复制源地址、目的地址、输入接口Id以及目的端口号给新的 Info；
5. 去除传输层头部：MESSAGE_RemoveHeader；
6. 设定一个包接收时间 (MSG_APP_FromTransport) 给应用层的目的协议：MESSAGE_SetLayer, MESSAGE_SetEvent、MESSAGE_Send.
7. 其他协议追踪和统计量更新。

ii. **TransportMytransSendToNetwork**: 相应FromAppSend 消息，下发给 Network 层或 IP协议。主要完成以下工作：

1. 创建一个本层协议头部，并添加到消息：MESSAGE_AddHeader
2. 更新头部字段信息，从应用层下传的 Info 中复制；
3. 设定头部长度和整个包的长度：MESSAGE_ReturnPacketSize；
4. 发送包到网络层，传递参数中包括一个整数，代表本层协议的IP协议号 (IP Protocol Number) ，以指示对端的传输层协议：NetworkIpReceivePacketFromTransport.

3.5 与应用层集成

要使新的传输层协议能够被特定的应用层协议使用，在应用层协议的实现中必须能够调用传输层的方法，包括上面的两个方法；根据应用层协议的需要可能需要特定的方法。统一的方法如下：

1. 实现所需的 API 函数在 transport_mytrans.cpp 中；
2. 声明该函数原型在 transport_mytrans.h 头文件中；
3. 在应用层协议源文件 (.cpp) 中包含 transport_mytrans.h 文件，并调用该 API。

3.6 与网络层集成

对于网络层协议（IP）而言，传输层协议实例凭借 **IP协议号（IP Protocol Number）** 进行标识。因此，作为传输层的一个新协议，必须首先注册新的 IP 协议号，才能与IP集成；然后在 IP 上送处理部分，调用新协议的 API。大致如下：

1. 为新的传输层协议定义新的 IP 协议号。在network_ip.h文件中，添加新的常量。**注意不能与已有的定义重复。**
2. 在 IP 的 **DeliverPacket** 方法中增加一个新协议的switch case，调用一个新的方法，比如，SendToMytrans，将包送个新的协议。
3. 实现这个新方法，比如SendToMytrans。可以直接在network_ip.cpp中实现，也可以在transport_mytrans.cpp中实现。建议后者。

3.7 搜集和报告统计量

在传输层新协议的主数据结构中有一个专门负责收集统计量的指针，类型为 STAT_TransportStatistics，由它负责在适当的触发点完成各统计量更新工作。该类型是一个 C++ 类，STAT_TransportStatistics，定义在 stats_transport.h/cpp 文件中。

```
10 |
17 | // /**
18 | // CLASS      :: STAT_TransportStatistics
19 | // DESCRIPTION :: Implements statistics for UDP or TCP.
20 | // **/
21 | class STAT_TransportStatistics : public STAT_ModelStatistics
22 | {
23 | protected:
24 |     // Statistics for unicast, multicast and broadcast
25 |     STAT_TransportAddressStatistics m_AddrStats[STAT_NUM_ADDRESS_TYPES];
26 |
27 |     // UDP or TCP
28 |     std::string m_Protocol;
29 |
30 |     // Contains all active sessions
31 |     std::map<STAT_TransportSummarySessionKey, STAT_TransportSessionStatistics*
32 |
33 | // Return the session for the (sourceAddr, sourcePort) and (destAddr, dest
34 | // If session statistics do not exist for this session then new ones are c
35 | STAT_TransportSessionStatistics* GetSession(
```

在收到或发送包时分别调用相应的方法，完成统计：

```

// Transport protocols should call this function when a segment is received
// from the network layer
void AddSegmentReceivedDataPoints(
    Node* node,
    Message* msg,
    STAT_DestAddressType type,
    UInt64 controlSize,
    UInt64 dataSize,
    UInt64 overheadSize,
    const Address& sourceAddr,
    const Address& destAddr,
    int sourcePort,
    int destPort);

// Transport protocols should call this function when sending a packet to the upper layer
void AddSentToUpperLayerDataPoints(
    Node* node,
    Message* msg,
    const Address& sourceAddr,
    const Address& destAddr,
    int sourcePort,
    int destPort);

```

统计量的打印通过调用传输层统计类的打印方法Print () 完成，可以在终止化方法Finalize () 中进行。

3.8 终止化

TransportMytransFinalize()在仿真结束时由TRANSPORT_Finalize调用，这里主要完成统计量打印的工作。

- 实现本协议的终止化方法

```

4 void
5 TransportMytransFinalize(Node *node)
6 {
7     char buf[MAX_STRING_LENGTH];
8     TransportDataMytrans* mytrans = node->transportData.mytransVariable;
9
10    if (mytrans->mytransStatsEnabled)
11    {
12        // Print the statistics
13        mytrans->newStats->Print(
14            node,
15            "Transport",
16            "MYTRANS",
17            ANY_ADDRESS,
18            -1/* instance id*/);
19    }
20 }

```

- 修改传输层的终止化操作，其中，调用MYTRANS 的终止操作。

```

1 //-----
8 void TRANSPORT_Finalize(Node * node)
9 {
10     TransportUdpFinalize(node);
11     TransportTcpFinalize(node);
12
13 #ifdef ENTERPRISE_LIB
14     if (node->transportData.rsvpProtocol == TRUE)
15     {
16         RsvpFinalize(node);
17     }
18 #endif // ENTERPRISE_LIB
19
20 //InsertPatch FINALIZE_FUNCTION
21 // LuoJT: MYTRANS
22 if (node->transportData.mytransEnabled)
23 {
24     TransportMytransFinalize(node);
25 }
26
27

```

3.9 如何设置广播地址

设定广播地址可以有两种方法：

- 把目的地址设为 ANY_DEST：由main.h 定义的常量 0xFFFFFFFF，代表本地的广播地址。【注意：只是本子网内的广播，而非全网广播！ just local broadcast, not global】
- 通过 NetworkIpGetInterfaceBroadcastAddress 方法：由network_ip.cpp 实现，用于获取特定接口的广播地址。比如，

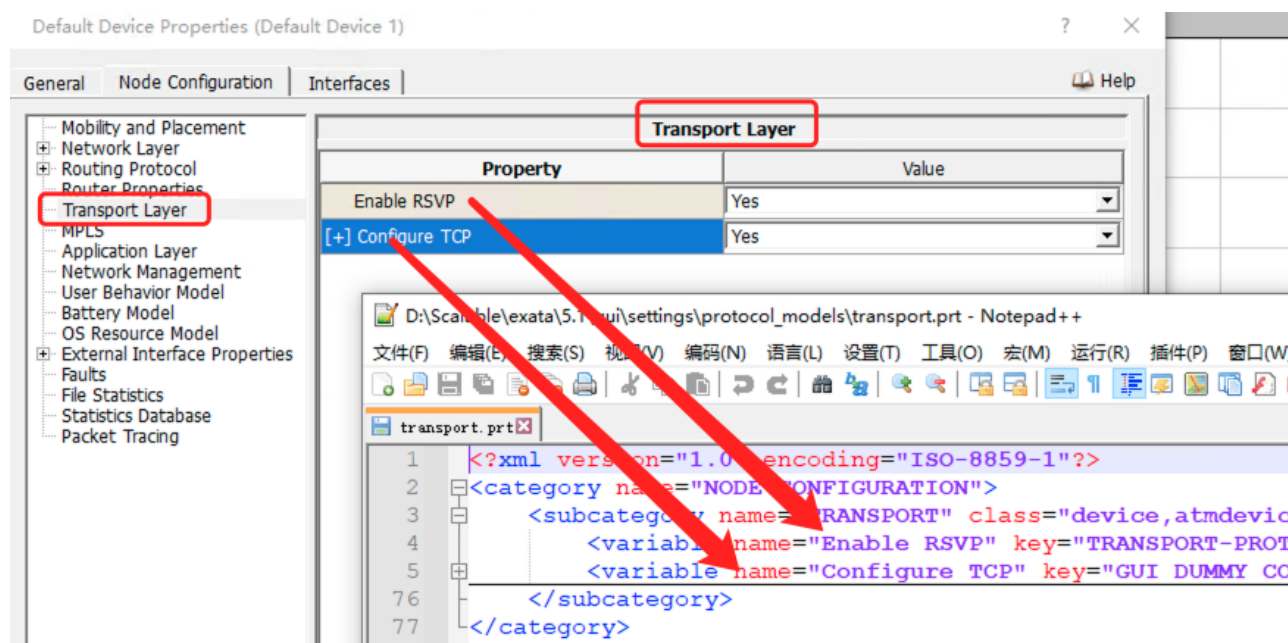
```

static void SendRouteAdvertisement(
    Node *node, RouteAdvertisementType type)
{
    ...
    int i;
    for (i = 0; i < node->numberInterfaces; i++)
    {
        NodeAddress destAddress;
        ...
        if (NetworkIpIsWiredNetwork (node, i))
        {
            destAddress =
                NetworkIpGetInterfaceBroadcastAddress(node, i);
        }
        else
        {
            destAddress = ANY_DEST;
        }
        ...
    }
    ...
}

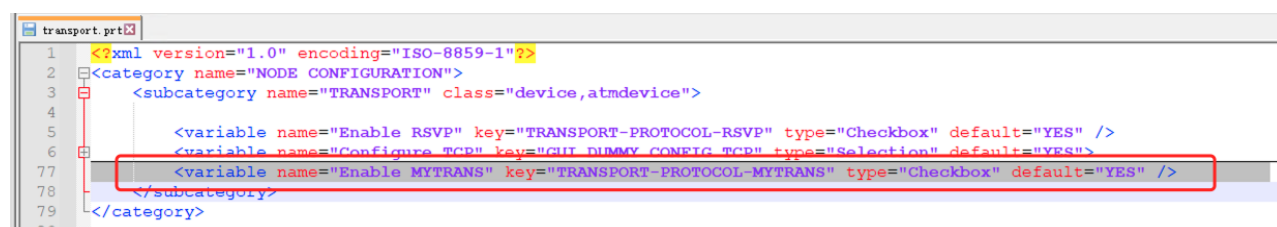
```

3.10 与 GUI 集成

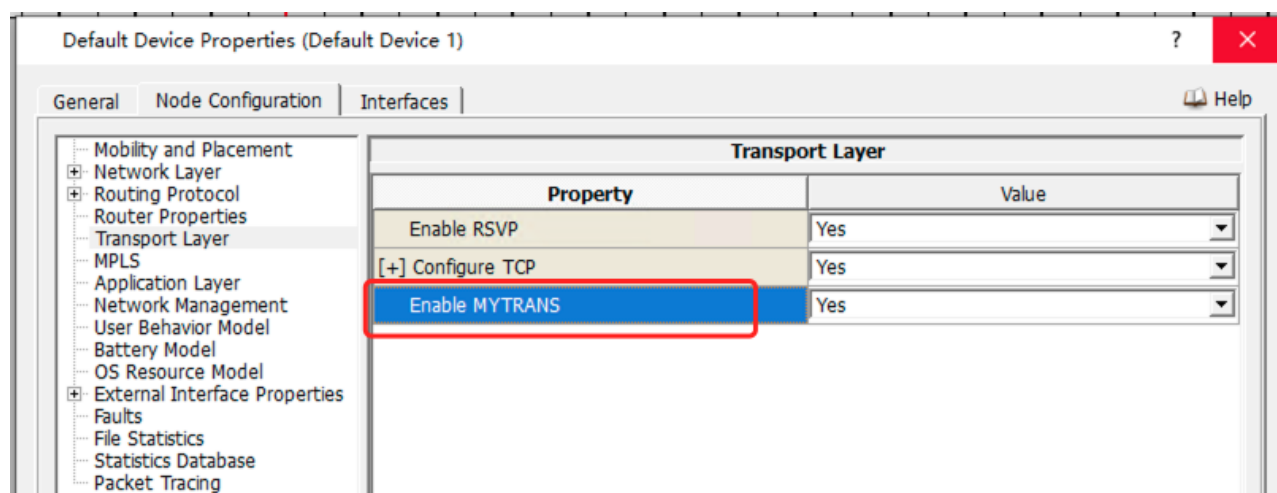
不是应用层协议，只需要修改 SDF 文件（shared description files）即可，不需要添加 CMP 文件。传输层对应的SDF为 transport.prt。对应节点配置的 Property Editor。如下图，transport.prt 控制节点配置中“Transport Layer”的属性框。



参考 Enable RSVP，在transport.prt 中添加一个 variable 行，“TRANSPORT-PROTOCOL-MYTRANS”，name = “Enable MYTRANS”，



重启 GUI，查看 Node Configuration 中的 Transport Layer，已添加 Enable MYTRANS 参数



在场景 ex_2.config中修改 node[1] 为 NO。查看场景配置文件，发现有两个地方出现 MYTRANS：一个是在“TRANSPORT”块，还有一个是在“Node Configuration”

```

*****TRANSPORT*****
TRANSPORT-PROTOCOL-RSVP YES
GUI_DUMMY_CONFIG_TCP YES
TCP LITE
TCP-USE-RFC1323 NO
TCP-DELAY-SHORT-PACKETS-ACKS NO
TCP-USE-NAGLE-ALGORITHM YES
TCP-USE-KEEPALIVE-PROBES YES
TCP-USE-OPTIONS YES
TCP-DELAY-ACKS YES
TCP-MSS 512
TCP-SEND-BUFFER 16384
TCP-RECEIVE-BUFFER 16384
TRANSPORT-PROTOCOL-MYTRANS YES

```

上面对应所有传输层的配置，代表默认值；下面是对所有节点的配置，节点[1] 采用的非默认值，没有激活 MYTRANS 协议。这个参数将在该节点初始化时发挥作用，在节点1将不激活该协议。参见 3.3.2。

```

transport.prt ex_2.config
9
0
1 *****Node Configuration*****
2
3
4 [2 thru 4] DUMMY-MULTICAST YES
5 [2 thru 4] MULTICAST-PROTOCOL PIM
6 [2 thru 4] ROUTING-PIMDM-BROADCAST-MODE NO
7 [2 thru 4] ROUTING-PIMDM-JOIN-PRUNE-SUPPRESSION NO
8 [2 thru 4] ROUTING-PIMDM-ASSERT-OPTIMIZATION NO
9 [1] TRANSPORT-PROTOCOL-MYTRANS NO
0 [2 thru 4] ROUTING-PIMDM-HELLO-SUPPRESSION NO
1 [2 thru 4] ROUTING-PIM-HELLO-PERIOD 30S
2 [2 thru 4] ENABLE-SSM-ROUTING NO
3 [1] HOSTNAME host1
4 [2] HOSTNAME host2
5 [3] HOSTNAME host3

```

至此，一个类似 UDP 的新的传输层协议添加完毕。