EXata 扩展（二）

目标：在 EXata 添加新的协议模型

参考：Chapter 4 in EXata 5.1 Programmer‘s Guide

4. Developing Protocol Models in EXata

EXata 协议栈如下图：

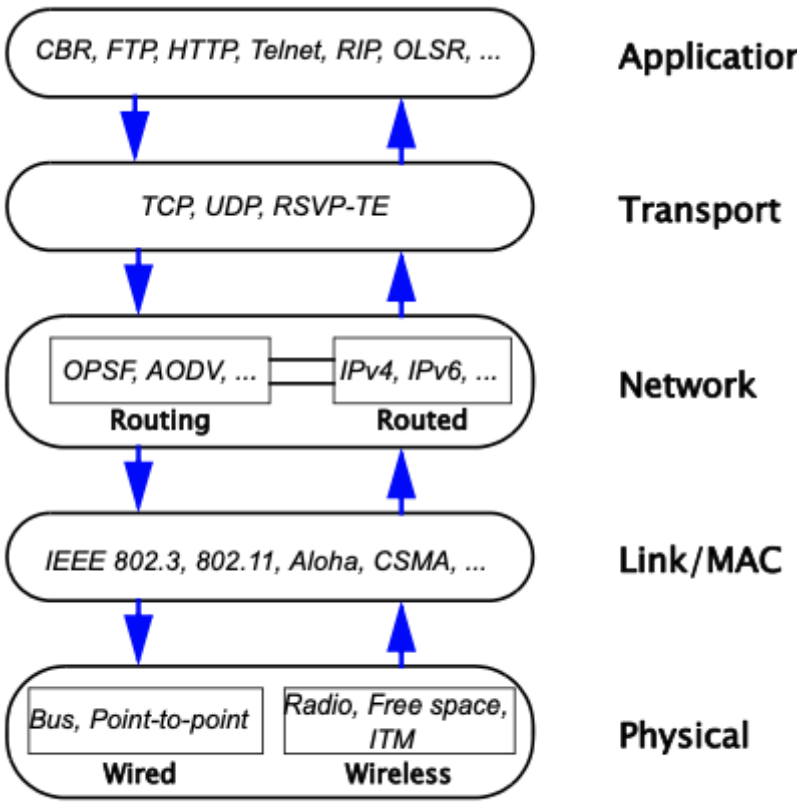


FIGURE 4-1. EXata Protocol Stack

4.1 General Programming Utility Functions 通用的编程工具函数

4.1.1 Reading Input from a Confuguration File 配置文件读入

全局默认配置文件：EXATA_HOME/scenarios/default/default.config

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

where:

`<Identifier>`	: Node identifier, subnet identifier, or IP address to which this parameter declaration is applicable, enclosed in square brackets. This specification is optional, and if it is not included, the parameter declaration applies to all nodes.
`<Parameter-name>`	: Name of the parameter.
`<Index>`	: Instance to which this parameter declaration is applicable, enclosed in square brackets. This is used when there are multiple instances of the parameter. This specification is optional, and if it is not included, the parameter declaration applies to all instances.
`<Parameter-value>`	: Value to be used for the parameter.

配置文件格式如上。

常用的读入配置文件的API有：

1. IO_ReadString：主要用于读取一个变量的值。
2. IO_ReadStringInstance：在标识符和实例都确定时读取参数的值。
3. IO_ReadCachedFile：读取并保存一个文件的内容。【问题：能否读取二进制数据？】

4.1.2 Programming with Message Info Fields 消息 Info 域编程

消息中的 Info 域用于保存一些附加信息，用于：1）处理消息；2）层间传递信息。因此，不对应真实网络中的信息，往往用于辅助完成仿真任务，比如收集统计量，传递跨层信息等。

- Info 类型：由 MessageInfoType 枚举类型定义
- 通过 API 进行访问：MESSAGE_AddInfo、MESSAGE_InfoAlloc、MESSAGE_RemoveInfo、MESSAGE_ReturnInfo、MESSAGE_CopyInfo、MESSAGE_ReturnInfoSize
- **声明用户自定义 Info 类型**，在MessageInfoType **最后部**追加新的类型，务必在最后面追加，不要在中间添加，否则，可能引起 Exata 崩溃！

```
typedef enum message_info_type_str
{
    INFO_TYPE_UNDEFINED = 0,   // an empty info field.
    INFO_TYPE_DEFAULT = 1,     // default info type used in situations where
                               // specific type is given to the info field.
    INFO_TYPE_AbstractCFPropagation, // type for abstract contention free
                                     // propagation info field.
    INFO_TYPE_AppName,       // Pass the App name down to IP layer
    INFO_TYPE_StatCategoryName,
    INFO_TYPE_DscpName,
    ...
    INFO_TYPE_ForwardTcpHeader,
    INFO_TYPE_MYINFOTYPE     // Type for Myinfo field
} MessageInfoType;
```

FIGURE 4-3. Declaring User-defined Info Field Type

- 添加一个 Info 域

  假定已经声明一个用户定义的 Info 类型：INFO_TYPE_MYINFO，而且已在头文件中定义了 MyInfoField 结构类型，可以通过 MESSAGE_AddInfo 添加 Info 字段

```
...
Message* msg;
struct MyInfoField* infoPtr;
...
msg = MESSAGE_Alloc(node, layer, protocol, eventType);

infoPtr = MESSAGE_AddInfo(node,
msg,
sizeof(MyInfoField),
INFO_TYPE_MYINFO);
...
// fill data in the info field using infoPtr now.
...
```

- 访问 Info 字段

  通过 MESSAGE_ReturnInfo 来访问特点字段

```
...
struct MyInfoField* infoPtr;
...
infoPtr = MESSAGE_ReturnInfo(msg, INFO_TYPE_MYINFO);
...
// Access fields of MyInfoField using pointer infoPtr.
...
```

- 擦除一个 Info 字段

  使用 MESSAGE_RemoveInfo 来擦除该字段内容，注意该字段被分配的空间仍然可以继续覆盖使用，除非采用 MESSAGE_Free。

```
...
struct MyInfoField* infoPtr;
...
infoPtr = MESSAGE_ReturnInfo(msg, INFO_TYPE_MYINFO);
if (infoPtr != NULL)
{
    ...
    // Access fields of MyInfoField using pointer infoPtr.
    ...
    MESSAGE_RemoveInfo(node, msg, INFO_TYPE_MYINFO);
}
```

- Info 字段的 持久性Persistence

  基本原则：1）每个用户创建的 Model 不要修改其他 Model 的Info 字段；2）

4.1.3 随机数生成

- EXata 采用伪随机数序列模拟真实系统的随机数；
- 一个随机数序列称之为“随机流”（random stream）
- 确保随机流的**独立性**与**可重复性**
- 三个基本函数：
  - RANDOM_erand: 返回 0.0 – 1.0 之间的一个实数；
  - RANDOM_jrand：返回 – 2^(31) -- 2^(31) 之间的一个整数
  - RANDOM_nrand：返回 0 – 2^(31)之间的一个整数。
- 随机数种子：24 bits，决定随机流是否独立

- 类型：typedef unsinged short RandomSeed[3]

- 设定初始种子的方法：RANDOM_SetSeed，它利用 4个输入参数生成确定的，但唯一的初始种子（RandomSeed 类型）
  - 参数1 – globalSeed: 即 场景 configuration file 中的 SEED，引入它用户可以为不同的实验选择不同的随机流；
  - 参数2 – nodeId：引入节点标识，确保不同的节点使用不同的随机流。
  - 参数3–protocolId：引入协议标识符确保同一节点的不同协议采用不同的随机流。
  - 参数4–instanceId：引入实例标识符，确保同一协议的不同实例采用不同的随机流。
- 设定随机数种子的例子：MACA MAC 协议，该协议需要两个独立的随机流，用于指示 backoff 时间和 channel yield 时间，两个都是均匀分布，需要不同的初始种子。在设定种子时，RANDOM_SetSeed的最后一个参数不同，保证不同的随机流分别用于产生 backoff 和 yield 时间，通过。

```
void MacMacaInit(
   Node *node, int interfaceIndex, const NodeInput *nodeInput)
{
    MacDataMaca *maca = (MacDataMaca *) MEM_malloc(sizeof(MacDataMaca));

    assert(maca != NULL);

    memset(maca, 0, sizeof(MacDataMaca));
    maca->myMacData = node->macData[interfaceIndex];
    maca->myMacData->macVar = (void *)maca;
    ...
    maca->currentNextHopAddress = ANY_DEST;

    RANDOM_SetSeed(maca->backoffSeed,
                   node->globalSeed,
                   node->nodeId,
                   MAC_PROTOCOL_MACA,
                   interfaceIndex);
    RANDOM_SetSeed(maca->yieldSeed,
                   node->globalSeed,
                   node->nodeId,
                   MAC_PROTOCOL_MACA,
                   interfaceIndex + 1);
    ...
}
```

FIGURE 4-4. Setting Random Number Seeds

- 内置的随机数分布
  - EXata 采用模板类描述随机数分布 RandomDistribution

```
template <class T>
class RandomDistribution
{
public:
    ...
    void setDistributionUniform(T min, T max);
    void setDistributionUniformInteger(T min, T max);
    void setDistributionExponential(T mean);
    void setDistributionGaussian(double sigma);
    void setDistributionGaussianInt(double sigma);
    void setDistributionPareto(T val1, T val2, double alpha);
    void setDistributionPareto4(T val1, T val2, T val3, double alpha);
    void setDistributionGeneralPareto(T val1, double alpha);
    void setDistributionParetoUntruncated(T val1, double alpha);
    void setDistributionDeterministic(T val);
    void setDistributionNull();
    int setDistribution(char* inputString,
                        char* printStr,
                        RandomDataType dataType);
    T getRandomNumber();
    T getRandomNumber(RandomSeed seed);
    void setSeed(UInt32 globalSeed,
                 UInt32 nodeId = 0,
                 UInt32 protocolId = 0,
                 UInt32 instanceId = 0);
 ...
};
```

FIGURE 4-7. Class `RandomDistribution`

- 如何使用 RandomDistribution 类
  - 首先声明一个随机分布变量，明确随机数类型 T。以 EXata 的 Shadowing 模型为例，

```
struct PropData {
    int   numPhysListenable;
    int   numPhysListening;
    ...
    RandomDistribution<double> shadowingDistribution;
    int  nodeListId;
    int  numSignals;
    ...
};
```

FIGURE 4-8. Declaring a Random Distribution Variable

  - 然后设定初始种子和分布类型

```
void PROP_Init(Node *node, int channelIndex, NodeInput *nodeInput) {
    PropData* propData = &(node->propData[channelIndex]);
    ...
    propData->shadowingDistribution.setSeed(
        node->globalSeed,
        node->nodeId,
        channelIndex);
    if (propProfile->shadowingModel == CONSTANT) {
        propData->shadowingDistribution.setDistributionDeterministic(
            propProfile->shadowingMean_dB);
    }
    else { // propProfile->shadowingModel == LOGNORMAL
        propData->shadowingDistribution.setDistributionGaussian(
            propProfile->shadowingMean_dB);
    }
```

FIGURE 4-9. Initializing a Random Distribution

- 使用 getRandomNumber 得到随机数

```
...
switch (propProfile->pathlossModel) {
    case FREE_SPACE:
    case TWO_RAY:
    {
        double shadowing_dB = 0.0;
        if (propProfile->shadowingMean_dB != 0.0) {
            shadowing_dB =
              propData->shadowingDistribution.getRandomNumber();
        }
        ...
        return;
```

- 文件解析功能
  - RandomDistributionClass 类有从文件中读取分布类型的功能，通常从 .app 文件中设定分布类型。
  - 格式：<Distribution Identifier> <Parameter List>，如
    - UNI 10 30: 代表一个 Uniform 分布，10 和 30
    - DET 20MS：代表一个确定性分布，值为 20ms
  - 分布标识和参数如表4–1

TABLE 4-1. Distribution Identifiers and Parameters

Distribution Name	Distribution Identifier	Parameters
Uniform	UNI	• Lower end of the range • Upper end of the range
Exponential	EXP	• Mean value
Pareto	TPD	• Lower end of the range (= lower limit of the truncation) • Upper limit of the truncation • Shape parameter
Pareto4	TPD4	• Lower end of the range • Lower limit of the truncation • Upper limit of the truncation • Shape parameter
Deterministic	DET	• Value

4.2 Application Layer 应用层

详细讲解如何添加一个应用层协议。

4.2.1 应用层协议

应用层协议可以分为两大类：

- 流量生成协议（Traffic-generating Protocols）
- 路由协议（Routing Protocols）

4.2.1.1 流量生成协议

这类协议有些直接就是网络应用，比如 FTP和 Telnet，还有些是来模拟真实网络应用，比如 CBR，它可以模拟很多真实的网络应用，比如音频流或老的视频编码流都可以用 CBR 来模拟。EXata 中的流量发生器见表4-2

TABLE 4-2. Traffic Generators in EXata

Traffic Generator	Description	Model Library
CBR	Constant Bit Rate (CBR) traffic generator. This UDP-based client-server application sends data from a client to a server at a constant bit rate.	Developer
CELLULAR-ABSTRACT-APP	Abstract cellular application. This is an application to generate traffic for networks running abstract cellular models.	Cellular
FTP	File Transfer Protocol (FTP). This tcplib application generates TCP traffic based on historical trace data.	Developer
FTP/GENERIC	Generic FTP. This model is similar to the FTP model but allows the user to have more control over the traffic properties. It uses FTP to transfer a user-specified amount of data.	Developer
GSM	Global System for Mobile communications (GSM). This is an application for generating traffic for GSM networks.	Cellular

TABLE 4-2. Traffic Generators in EXata (Continued)

Traffic Generator	Description	Model Library
HTTP	HyperText Transfer Protocol (HTTP). The HTTP application generates realistic web traffic between a client and one or more servers. The traffic is randomly generated based on historical data.	Developer
LOOKUP	Look-up traffic generator. This is an abstract model of unreliable query/response traffic, such as DNS look-up, or pinging.	Developer
MCBR	Multicast Constant Bit rate (MCBR). This model is similar to CBR and generates multicast constant bit rate traffic.	Developer
PHONE-CALL	Phone call traffic generator. This model simulates phone calls between two end users in a UMTS network.	UMTS
SUPER-APPLICATION	Super application. This model can simulate both TCP and UDP flows as well as two-way (request-response type) UDP sessions.	Developer
TELNET	Telnet application. This model generates realistic Tenet-style TCP traffic between a client and a server based on historical data. It is part of the tcplib suit of applications.	Developer
TRAFFIC-GEN	Random distribution-based traffic generator. This is a flexible UDP traffic generator that supports a variety of data size and interval distributions and QoS parameters.	Developer
TRAFFIC-TRACE	Trace file-based traffic generator. This model generates traffic according to a user-specified file, and like Traffic-Gen, it supports QoS parameters.	Developer
VBR	Variable Bit Rate (VBR) traffic generator. This model generates fixed-size data packets transmitted using UDP at exponentially distributed time intervals.	Developer
VOIP	Voice over IP traffic generator. This model simulates IP telephony sessions.	Multimedia and Enterprise
ZIGBEEAPP	ZigBee Application. This is similar to the CBR application but is used only in sensor networks.	Sensor Networks

4.2.1.2 路由协议

除了流量发生器，有些**服务提供协议**（service–providingprotocols）也位于应用层，比如路由协议。这些路由协议就属于公共服务提供的应用层协议，它们利用 UDP 或 TCP 服务。

EXata 支持的应用层路由协议见表4–3，还有一些路由协议在网络层实现，见表4–8、4–9.

TABLE 4-3. Application Layer Routing Protocols in EXata

Routing Protocol	Description	Model Library
BELLMANFORD	Bellman-Ford routing protocol.	Developer
BGPv4	Border Gateway Protocol version 4 (BGPv4). This protocol can be used for IPv4 and IPv6 networks.	Multimedia and Enterprise
EIGRP	Enhanced Interior Gateway Routing Protocol (EIGRP). This is a distance vector routing protocol designed for fast convergence.	Multimedia and Enterprise
FISHEYE	Fisheye Routing Protocol. This is a link state-based routing protocol.	Wireless
IGRP	Interior Gateway Routing Protocol (IGRP). This is a distance vector Interior Gateway protocol (IGP).	Multimedia and Enterprise
OLSR-INRIA	Optimized Link State Routing (OLSR) protocol. This is a link state-based routing protocol.	Wireless
OLSRv2-NIIGATA	Optimized Link State Routing, version 2 (OLSRv2) protocol. This is a successor of the OLSR protocol.	Wireless
RIP	Routing Information Protocol (RIP) routing protocol.	Developer
RIPng	Routing Information Protocol, next generation (RIPng) routing protocol. This protocol can be used for IPv6 networks.	Developer

其他路由协议可以直接在网络层发送消息，而不使用 TCP 或 UDP 服务，比如 AODV 和 DSR 路由协议。应用层路由协议和网络层路由协议的主要区别见表4–4：

TABLE 4-4. Application Layer versus Network Layer Routing Protocols

Application Layer Routing Protocols	Network Layer Routing Protocols
Use UDP or TCP to transmit their route discovery and control packets.	Use IP directly to transmit their route discovery and control packets.
Use an IP kernel function to update the IP forwarding table.	Use IP kernel functions to register itself as the packet routing function.
Do not receive data packets to forward, IP handles those itself.	Receive data packets and decide outgoing interface to forward packets.

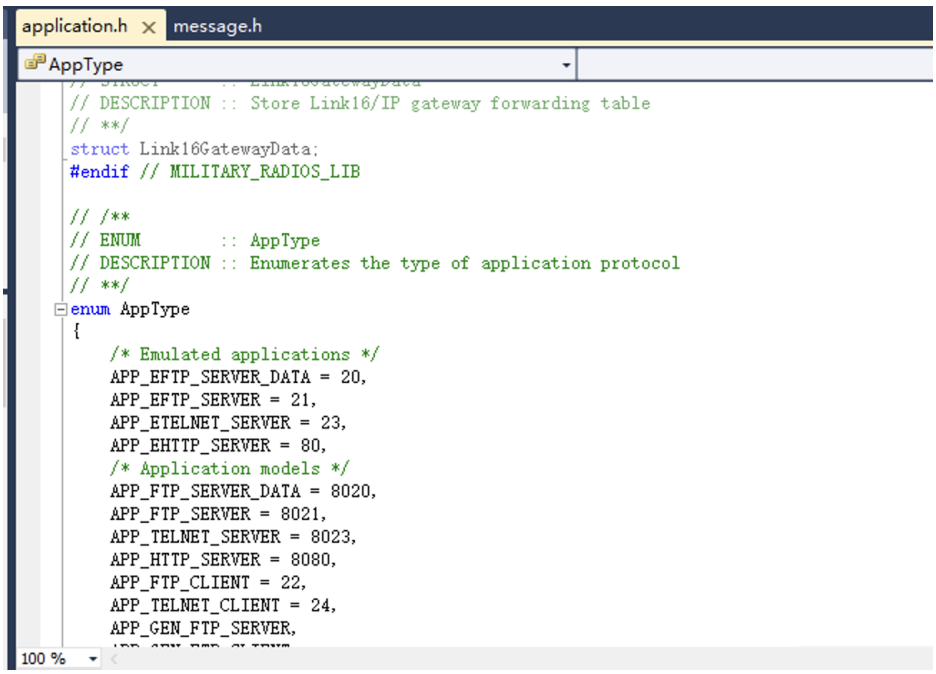
4.2.2 应用层的组织：文件和文件夹

1. 应用层的主要头文件
   - EXATA_HOME/include/api.h
   - EXATA_HOME/include/application.h
   - EXATA_HOME/include/app_util.h
2. 应用层相关头文件
   - EXATA_HOME/include/fileio.h
   - EXATA_HOME/include/mapping.h
3. 应用层相关的文件夹和源文件
   - EXATA_HOME/libraries/developer/src
   - EXATA_HOME/main/application.cpp

- EXATA_HOME/main/app_util.cpp

4.2.3 应用层数据结构

EXata 的应用层数据结构定义在 application.h 中。比如 AppType



1. AppType：应用层协议枚举类型
2. AppInfo：包含一个应用层协议实例的相关信息。
3. AppData：主要的数据结构， 包含一个节点所有协议的相关信息；appPtr：指向流量生成协议列表的指针；
4. AppTimer：实现应用层等一些Timers。

4.2.4 应用层 API 和层间通信

4.2.4.1 应用层到传输层通信

应用层API 定义在app_util.h，使用 TCP 或 UDP 的服务。

- APP_UdpSendNewDataWithPriority
- APP_TcpOpenConnection
- APP_TcpServerListen
- APP_TcpSendData
- APP_TcpCloseConnection

4.2.4.2 传输层到应用层通信

传输层到应用层的通信利用消息。 定义在 api.h文件中。主要消息包括：

- MSG_APP_FromTransport
- MSG_APP_FromTransOpenResult
- MSG_APP_FromTransDataSent
- MSG_APP_FromTransDataReceived

- MSG_APP_FromTransListenResult
- MSG_APP_FromTransCloseResults

4.2.4.3 应用层工具API

有些常用的内部函数，定义在 app_util.h中。

- APP_IsFreePort
- APP_GetProtocolType
- APP_RegisterNewApp
- APP_SetTimers

4.2.5 添加一个生成流量的应用协议

以 CBR 为例，详细介绍如何实现一个生成流量大应用协议。

4.2.5.1 命名规则

假设我们要添加一个协议“MYPROTOCOL”，那么我们使用“Myprotocol”作为协议命令的部分。比如用 Cbr 代表 CBR 用在各种命名中。

4.2.5.2 创建文件

在EXATA_HOME/libraries/ 下添加一个新文件夹：**user_models**，在其下面再创建一个子目录：src

- **激活 user-models ：**
  - **1）创立 用户库的 Makefile：**置于user_models/ 目录下，与 src/ 并列；可以拷贝其他路径下（比如developer）下的两个 Makefile：Makefile-common、Makefile-windows，参照修改。注意：确保符号名称和路径正确无误！！！目前在Makefile-common中只包含了一个 cpp 文件，可以后续继续添加。非最后一行的结尾换行使用“\”

Makefile-common

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

Makefile-windows

```
#
# Define User library models.
#

include ../libraries/user_models/Makefile-common

ADDON_OPTIONS    = $(ADDON_OPTIONS) $(USER_MODELS_OPTIONS)
ADDON_INCLUDES   = $(ADDON_INCLUDES) $(USER_MODELS_INCLUDES)

USER_MODELS_OBJS = $(USER_MODELS_SRCS:.cpp=.obj)

LIBRARIES_OBJ = $(LIBRARIES_OBJ) $(USER_MODELS_OBJS)
```

- 2）参照 EXata 扩展（一）：Simulator Basics中激活 Addons 的办法修改 main/Makefile-addons-windows，进行插件激活。

Makefile-addons-windows

```
#include ../addons/db/Makefile-windows
#INSERT Contributed Models HERE
#include ../contributed/maodv/Makefile-windows

# LUOJT: INSERT User-defined addons
include ../libraries/user_models/Makefile-windows
```

- **创建新协议的头文件和源文件**：
  - 通过 VS 新建一个头文件：app_myprotocol.h 和 一个源文件：app_myprotocol.cpp，保存在刚创建的 src/ 文件夹下。**【注意：要将两个文件添加到工程中】**
  - 注意头文件，要加上#ifndef宏，避免重复包含；
  - 仿照 app_cbr.h/.cpp，暂时只在头文件中定义一个数据结构：MyProtocolData；
- **在Applicaiton层协议列表中包含新协议**
  - 在 applicaiton.h 中的枚举类型 AppType 中添加 APP_MYPROTOCOL_CLIENT（客户端应用）和 APP_MYPROTOCOL_SERVER（服务器段）应用两个新的应用类型；
  - 注意：务必添加在最后面，刚好在APP+PLACEHOLDER之前。

```
// /**
// ENUM        :: AppType
// DESCRIPTION :: Enumerates the type of application protocol
// **/
enum AppType
{
    /* Emulated applications */
    APP_EFTP_SERVER_DATA = 20,
    APP_EFTP_SERVER = 21,
    APP_ETELNET_SERVER = 23,
    APP_EHTTP_SERVER = 80,
    /* Application models */

    APP_FTP_SERVER_DATA = 8020,
    APP_FTP_SERVER = 8021,
```

```
    // LuoJT: add MYPROTOCOL
    APP_MYPROTOCOL_CLIENT,
    APP_MYPROTOCOL_SERVER,

    APP_PLACEHOLDER
};
```

4.2.5.3 在协议追踪列表中添加新协议

- 在 trace.h 文件中，TraceProtocolType 枚举类型中添加新协议。
- 同样，注意务必加在最后面，刚好在 TRACE_ANY_PROTOCOL 之前。

```
TraceProtocolType
// /**
// ENUM        :: TraceProtocolType
// DESCRIPTION :: Enlisting all the possible traces
// **/
enum TraceProtocolType
{
    TRACE_UNDEFINED = 0,
    TRACE_TCP,                  // 1
    TRACE_UDP,                  // 2
    TRACE_IP,                   // 3
    TRACE_CBR,                  // 4
    TRACE_FTP,                  // 5
    TRACE_GEN_FTP,              // 6
    TRACE_BELLMANFORD,          // 7
    TRACE_BGP,                  // 8
    TRACE_FISHEYE,              // 9
```

```
    // DHCP
    TRACE_DHCP,

    // Luojt: MYPROTOCOL
    TRACE_MYPROTOCOL,

    // Must be last one!!!
    TRACE_ANY_PROTOCOL
};
```

4.2.5.4 定义数据结构：

- 在新协议的头文件中定义该协议数据单元的结构。通常包括：
  - 应用参数；
  - 应用实例标识，比如端口号；
  - 应用状态；
  - 统计变量等。
- 参考 CBR 为例，
- **MyProtocolData：相当于协议数据单元，应用层PDU**

```
/*
 * Data item structure used by myprotocol.
 */
typedef
struct struct_app_myprotocol_data
{
    short sourcePort;
    char type;
    Int32 seqNo;
    clocktype txTime;

}MyprotocolData;
```

- **AppDataMyprotocolClient**: 包含 Myprotocol 客户端信息的数据结构

```
/* Structure containing myprotocol client information. */

typedef
struct struct_app_myprotocol_client_str
{
    Address localAddr;
    Address remoteAddr;
    D_Clocktype interval;
    clocktype sessionStart;
    clocktype sessionFinish;
    clocktype sessionLastSent;
    clocktype endTime;
    BOOL sessionIsClosed;
    UInt32 itemsToSend;
    UInt32 itemSize;
    short sourcePort;
    Int32 seqNo;
    D_UInt32 tos;
    int  uniqueId;
    Int32 mdpUniqueId;
    BOOL isMdpEnabled;
    std::string* applicationName;
}
AppDataMyprotocolClient;
```

- AppDataMyprotocolServer: 新协议服务器端信息的数据结构：

```
/* Structure containing myprotocol server related information. */

typedef
struct struct_app_myprotocol_server_str
{
    Address localAddr;
    Address remoteAddr;
    short sourcePort;
    clocktype sessionStart;
    clocktype sessionFinish;
    clocktype sessionLastReceived;
    BOOL sessionIsClosed;
    Int32 seqNo;
    int uniqueId ;

}
AppDataMyprotocolServer;
```

4.2.5.5 初始化

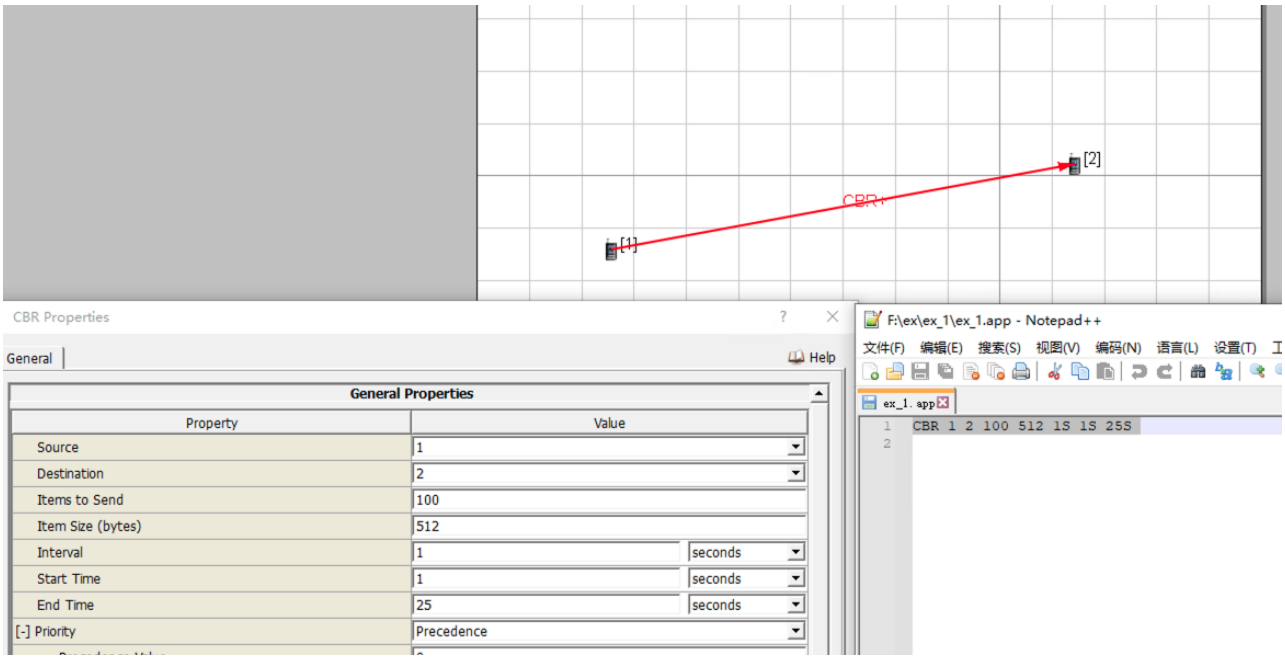
- 首先**确定协议配置格式（命令行和APP配置文件接口）**：

  `<Protocol-name> <param1> <param2> ... <paramN>,`

  比如 `CBR` 的配置文件格式：

`CBR <src> <dest> <items_to_send> <item_size> <interval> <start time><end time>`

  - 如果我们通过 EXataGUI 创建一个最简单的场景（ex_1.config），在两个节点（1和2）间添加一个 CBR 业务，全部采用默认设置，其应用配置文件（ex_1.app）中只有一行：CBR 1 2 100 512 1S 1S 25S。所以，这里确定的格式，将决定应用配置文件，新协议的配置格式。下面所述的初始化过程，也将按此格式读入参数，进行新协议的初始化。



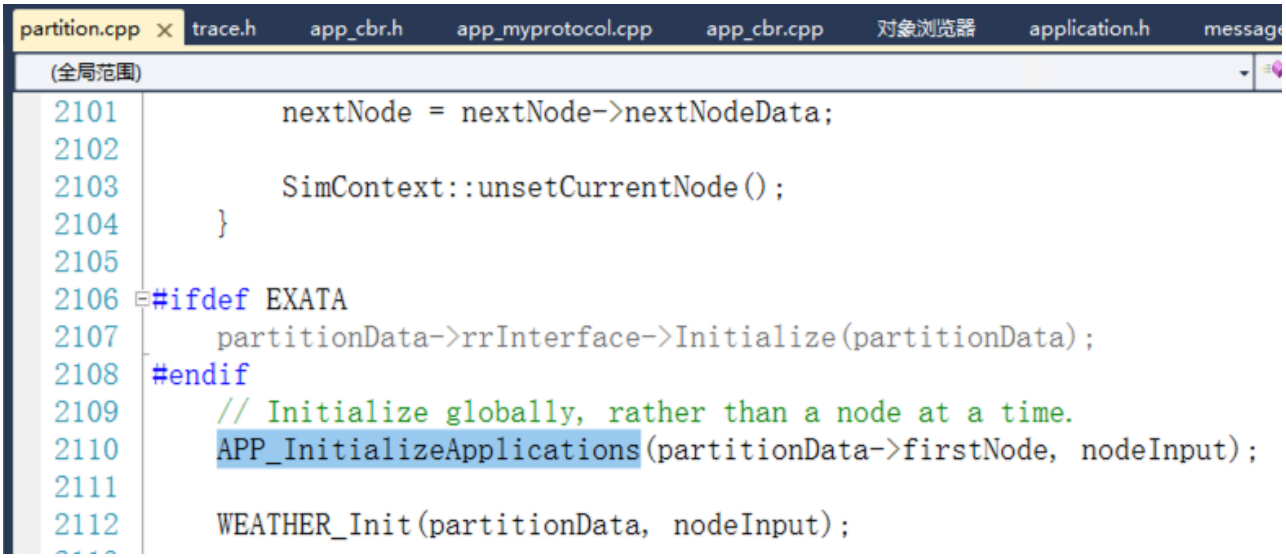
- **读应用配置参数，调用协议初始化函数**
  - 协议初始化时要读取配置文件获取配置参数，然后调用协议初始化函数。
  - 初始化过程按照节点、层和协议分级进行（见 EXata 扩展（一）：Simulator Basics）。
  - 每个节点的协议栈初始化自下而上进行，因此，**应用层协议的初始化最后进行。**
  - 节点初始化过程在 **节点初始化函数 PARTITion.InitializeNodes 中执行**，该函数定义在main/partition.cpp 中。
  - 在PARTITion.InitializeNodes 中调用应用层初始化函数：**APP_InitializeApplications** 和 **APP_Initialize**, 分别进行应用协议初始化和应用层路由协议初始化；这两个应用层初始化函数定义在 application.cpp 中，

partition.cpp | trace.h | app_cbr.h | app_myprotocol.cpp | app_cbr.cpp | 对象浏览器 | applicatio

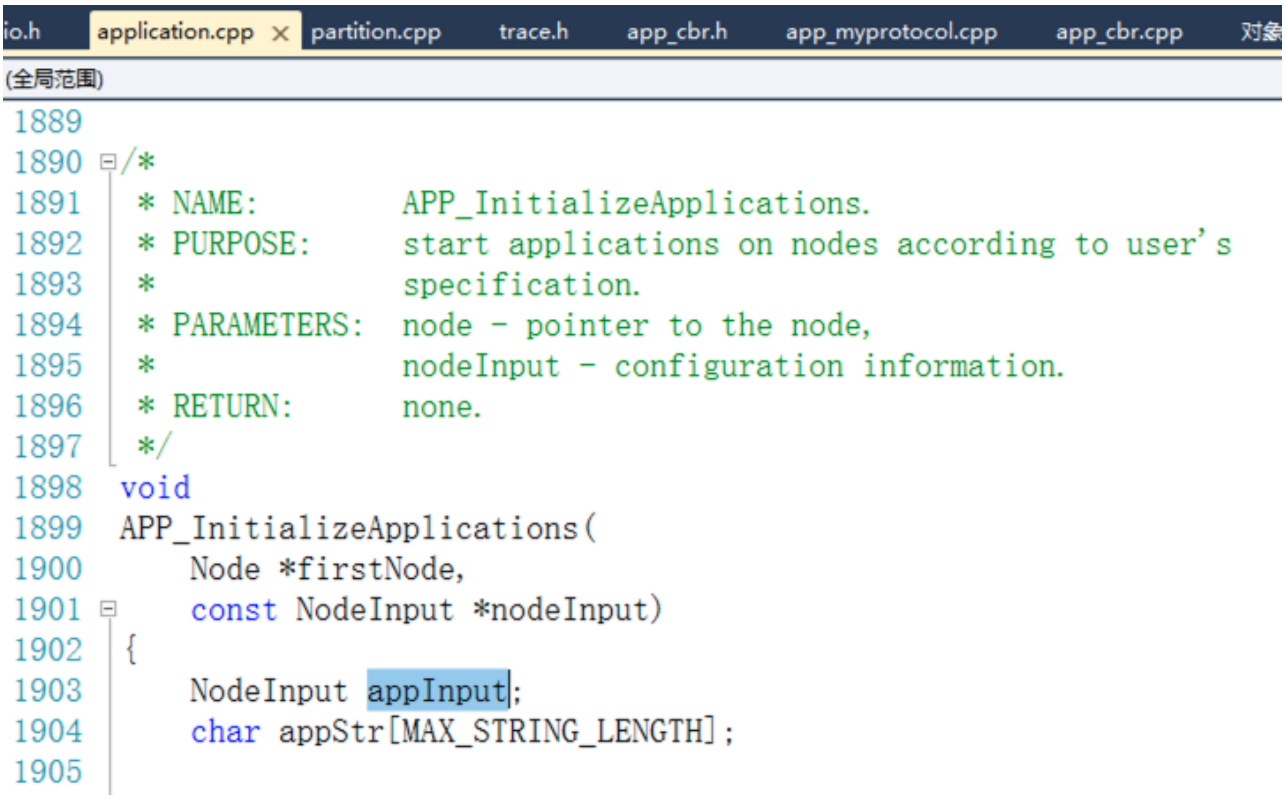
(全局范围)

```
2071            SimContext::setCurrentNode(nextNode);
2072        #ifdef JNE_LIB
2073            JNE_Initialize(nextNode, nodeInput);
2074        #endif /* JNE_LIB */
2075
2076            ADAPTATION_Initialize(nextNode, nodeInput);
2077
2078            // If ther is any adaptation protocol initialize
2079            // the end systems & switch stack differently
2080
2081            if ((nextNode->adaptationData.adaptationProtocol
2082                == ADAPTATION_PROTOCOL_NONE)
2083                || (nextNode->adaptationData.endSystem))
2084            {
2085            NETWORK_Initialize(nextNode, nodeInput);
2086            TRANSPORT_Initialize(nextNode, nodeInput);
2087        #ifdef EXATA
2088            SocketLayerInit(nextNode, nodeInput);
2089        #endif
2090            APP_Initialize(nextNode, nodeInput);
2091            USER_Initialize(nextNode, nodeInput);
2092            }
```

  -



- APP_InitializeApplications **函数读取应用配置文件**，然后**调取各协议的初始化函数**进行协议的初始化。
  - 应用配置文件（*.app）的输入保存在变量 appInput （NodeInput 类型，定义在 fileio.h）；appInput.inputStrings 代表配置文件读入的全部内容，通常包括多行文本，用字符串数组表示。



  - 使用 C 语言的 sscanf 方法对配置文件的一行进行读入，包括第一个字的协议类型，以及后面的参数。
  - 比如用 sscanf(appInput.inputStrings[i], "%s", appStr); 读入第 i 行的第一个字，即 **appStr 保存协议类型**。而采用下面的方式读入其后的配置参数：注意第一个“ %*s”格式符用来跳过第一个字符串（即已经读入的协议类型），numValues 为正确读取的参数个数。

```
numValues = sscanf(appInput.inputStrings[i],
                   "%*s %s %s %d %s",
                   sourceString,
                   destString,
                   &itemsToSend,
                   startTimeStr);
```

- 然后通过比较 appStr 与协议标识符（比如“CBR”，“MYPROTOCOL”）判断应用协议类型；利用 sscanf 格式化输入读入该协议各配置参数。

eio.h | application.cpp | partition.cpp | trace.h | app_cbr.h | app_myprotocol.cpp | app_cbr.cpp

(全局范围)

```
2583            else
2584            if (strcmp(appStr, "CBR") == 0)
2585            {
2586                char profileName[MAX_STRING_LENGTH] = "/0";
2587                BOOL isProfileNameSet = FALSE;
2588                BOOL isMdpEnabled = FALSE;
2589                char sourceString[MAX_STRING_LENGTH];
2590                char destString[BIG_STRING_LENGTH];
2591                char intervalStr[MAX_STRING_LENGTH];
2592                char startTimeStr[MAX_STRING_LENGTH];
2593                char endTimeStr[MAX_STRING_LENGTH];
2594                int itemsToSend;
2595                int itemSize;
2596                NodeAddress sourceNodeId;
2597                Address sourceAddr;
2598                NodeAddress destNodeId;
2599                Address destAddr;
2600                unsigned tos = APP_DEFAULT_TOS;
2601                BOOL isRsvpTeEnabled = FALSE;
2602                char optionToken1[MAX_STRING_LENGTH];
2603                char optionToken2[MAX_STRING_LENGTH];
2604                char optionToken3[MAX_STRING_LENGTH];
2605                char optionToken4[MAX_STRING_LENGTH];
2606                char optionToken5[MAX_STRING_LENGTH];
2607                char optionToken6[MAX_STRING_LENGTH];
2608
2609                numValues = sscanf(appInput.inputStrings[i],
2610                                   "%*s %s %s %d %d %s %s %s %s %s %s %s %s %s",
2611                                   sourceString,
2612                                   destString,
2613                                   &itemsToSend,
2614                                   &itemSize,
2615                                   intervalStr,
2616                                   startTimeStr,
```

- **获取源和目的节点标识符**：利用EXata 库函数 IO_AppParseSourceAndDestStrings （定义在fileio.h），从该应用协议的输入配置行中提取源和目的节点的Id （和地址），比如 1 和 2，保存在 sourceNodeId、destNodeId。
- **获取源和目的节点指针**：利用 EXata 的 Mapping 函数：MAPPING_GetNodePtrFromHash（定义在 mapping.h）利用节点 Id 获取节点指针。
- **时间参数格式转换**：将时间相关的参数的数据类型由 string 转换为 clocktype，利用

TIME_ConvertToClock （定义在 clock.h）；

- 至此，APP层某个协议的初始化就准备完毕，可以分别调用协议初始化函数了，对于 CBR 来讲，分客户端/服务端两个协议初始化函数了。
- **CBR 客户端初始化**：调用 AppCbrClientInit 函数，输入源节点指针和读入解析、转换好的其他参数。

```
2843                    else
2844                    {
2845                        AppCbrClientInit(
2846                            node,
2847                            sourceAddr,
2848                            destAddr,
2849                            itemsToSend,
2850                            itemSize,
2851                            interval,
2852                            startTime,
2853                            endTime,
2854                            tos,
2855                            sourceString,
2856                            destString,
2857                            isRsvpTeEnabled,
2858                            appNamePtr);
2859                    }
```

- **检查该协议是否为环回协议**（loop-back）：利用 APP_SuccessfullyHandledLoopback 函数（application.cpp）来检查，返回值为 FALSE，则为非环回。
- **提取目的节点指针，进行服务器初始化**：同样使用 MAPPING_GetNodePtrFromHash来提取，不同的是输入 destNodeId。调用 AppCbrServerInit (node)进行 服务器初始化。
- **添加新协议处理代码**：仿造并简化 CBR，在 **APP_InitializeApplicatioins**中“CBR”后面添加 ”MYPROTOCOL“ 协议处理部分代码

```
/* Luojt: exercise new app protocol*/
else
    if (strcmp(appStr, "MYPROTOCOL") == 0) {
        // Intitialize variables fir reading user input
        char sourceString[MAX_STRING_LENGTH];
        char destString[BIG_STRING_LENGTH];
        char interval1Str[MAX_STRING_LENGTH];
        char startTimeStr[MAX_STRING_LENGTH];
        char endTimeStr[MAX_STRING_LENGTH];
        int itemsToSend;
        int itemSize;
        NodeAddress sourceNodeId;
        Address sourceAddr;
        NodeAddress destNodeId;
        Address destAddr;

        // Read user input into approprate variables
        numValues = sscanf(appInput.inputStrings[i],
            "%*s %s %s %s %s %s %s %s",
            sourceString,
            destString,
            &itemsToSend,
            &itemSize,
            interval1Str,
            startTimeStr,
            endTimeStr);
        if (numValues != 7) {
            char errorString[MAX_STRING_LENGTH + 100];
            sprintf(errorString,
                "Wrong MYPROTOCOL configuration format!\n"
                "MYPROTOCOL <src> <dest> <items to send> "
                "<item size> <interval> <start time> "
                "<end time>\n");
            ERROR_ReportError(errorString);
        }
```

```
// Ger source and destination nodeId and address
IO_AppParseSourceAndDestStrings(
    firstNode,
    appInput.inputStrings[i],
    sourceString,
    &sourceNodeId,
    &sourceAddr,
    destString,
    &destNodeId,
    &destAddr);
// Get source node ptr
node = MAPPING_GetNodePtrFromHash(nodeHash, sourceNodeId);
if (node != NULL)
{
    clocktype startTime = TIME_ConvertToClock(startTimeStr);
    clocktype endTime = TIME_ConvertToClock(endTimeStr);
    clocktype interval = TIME_ConvertToClock(intervalStr);

    // Call MYPROTOCOL client initialization function
    AppMyprotocolClientInit(
        node,
        sourceAddr,
        destAddr,
        itemsToSend,
        itemSize,
        interval,
        startTime,
        endTime
        );

}

    // Check if loop-back
    // Get dest node pointer
    if (node == NULL || !APP_SuccessfullyHandledLoopback(
        node,
        appInput.inputStrings[i],
        destAddr,
        destNodeId,
        sourceAddr,
        sourceNodeId))
    {
        node = MAPPING_GetNodePtrFromHash(nodeHash, destNodeId);
    }

    if (node != NULL)
    {
        // Call MYPROTOCOL server initialization funciton
        AppMyprotocolServerInit(node);

    }

}
```

- 注意到新协议的初始化函数尚未定义。
- 注意：
  - 必须在applicatioin.cpp开头包含新协议的头文件；
  - 添加该头文件路径到工程的包含路径中。项目--》属性--》配置属性--》VC++目录--》包含目录

- **实现新协议的客户端初始化**
  - 创建一个应用实例，并初始化其状态
  - 实现新协议的 ClientInit 函数：**AppMyprotocolClientInit**，注意需要在头文件中进行声明。在其中，首先要检查输入参数有效，然后调用一个～NewMyprotocolClient 函数来生成客户端实例。

```
// MYPROTOCOL Client initialization
void
    AppMyprotocolClientInit(
    Node *node,
    Address clientAddr,
    Address serverAddr,
    Int32 itemToSend,
    Int32 itemSize,
    clocktype interval,
    clocktype startTime,
    clocktype endTime,
    char* appName,
    const NodeInput* nodeInput)
{
    char eror[MAX_STRING_LENGTH];
    AppTimer *timer;
    AppDataMyprotocolClient *clientPtr;
    Message *timerMsg;
    int minSize;

    startTime -= getSimStartTime(node);
    endTime -= getSimStatyTime(ndoe);

    ERROR_Assert(sizeof(MyprotocolData) < MYPROTOCOL_HEADER_SIZE,
        "MyprotocolData size should be greater than MYPROTOCOL_HEADER_SIZER");
    minSize = MYPROTOCOL_HEADER_SIZE;

    // Check to make sure the items to send correct
    if (itemsTosend < 0)
    {
        sprintf(error, "MYPROTOCOL Client: Node %d item to sends needs"
            " to be >= 0\n", node->nodeId);

        ERROR_ReportError(error);
    }
```

```
// Make sure the end time correct
if (!((endTime > startTime) || (endTime == 0)))
{
    sprintf(error, "MYPROTOCOL Client: Node %d end time needs to be > "
        "start time or equal to 0.\n", node->nodeId);
    ERROR_ReportError(error);
}

// Create an instance of the application client
clientPtr = AppMyprotocolClientNewMyprotocolClient(node,
    clientAddr,
    serverAddr,
    itemsToSend,
    itemSize,
    interval,
    startTime,
    endTime);
```

- 在这个～NewMyprotocolClient 函数中，使用 MEM_malloc 动态内存分配方法，分配一个状态结构体大小的内存，保存用户设定的状态参数，然后返回状态结构体的指针。

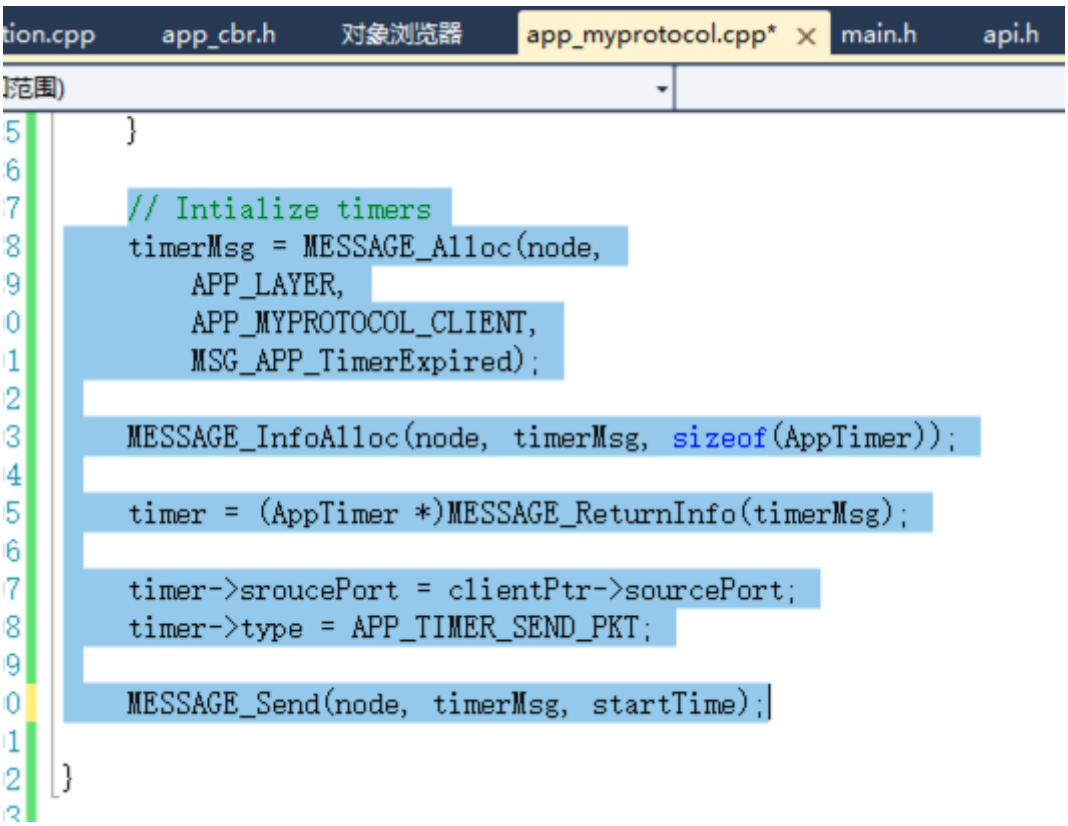
```
// Create New Client instance
AppDataMyprotocolClient *
AppMyprotocolClientNewMyprotocolClient(Node *node,
                            Address localAddr,
                            Address remoteAddr,
                            Int32 itemsToSend,
                            Int32 itemSize,
                            clocktype interval,
                            clocktype startTime,
                            clocktype endTime,
                            TosType tos,
                            char* appName)
{
    AppDataMyprotocolClient *myprotocolClient;

    // Allocate a new client instance
    myprotocolClient = (AppDataMyprotocolClient *)
        MEM_malloc(sizeof(AppDataMyprotocolClient));
    memset(myprotocolClient, 0, sizeof(AppDataMyprotocolClient));

    //Fill in the client info

    memcpy(&(myprotocolClient->localAddr), &localAddr, sizeof(Address));
    memcpy(&(myprotocolClient->remoteAddr), &remoteAddr, sizeof(Address));
    myprotocolClient->interval = interval;

    myprotocolClient->endTime = endTime;
    myprotocolClient->itemsToSend = itemsToSend;
    myprotocolClient->itemSize = itemSize;

    // Set the source port, identifer of this instance
    myprotocolClient->sourcePort = node->appData.nextPortNum++;

    myprotocolClient->seqNo = 0;

    if (appName)
    {
        myprotocolClient->applicationName = bew std::string(appName);
    }
    else
    {
        myprotocolClient->applicationName = new std::string();
    }

    // Register the new app instance
    APP_RegisterNewApp(node, APP_MYPROTOCOL_CLIENT, myprotocolClient);

    return myprotocolClient;
}
```

- 这个新的客户端实例由 sourcePort 来进行唯一标识，在创建一个新的实例时，该源端口自动加一。注意：新实例生成后，必须向系统注册。
- **注册新的应用实例**：将新生成的应用添加到该节点的一个应用列表中，当需要访问该应用的状态信息时，首先从

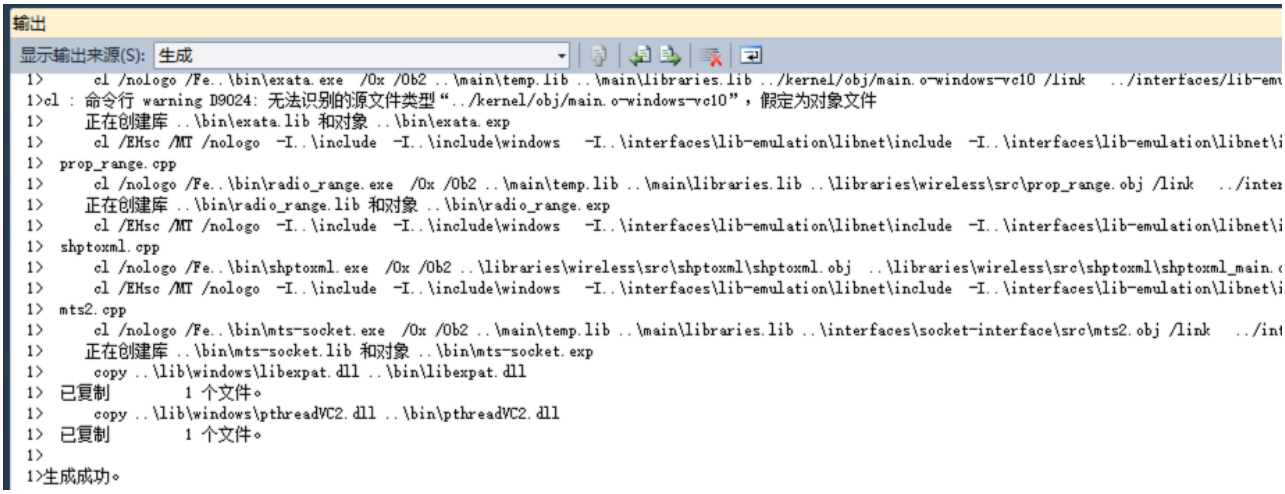
该应用列表中取出该应用的状态结构体。该链表的类型为 AppInfo （见 4.2.3）。

- 利用 APP_RegisterNewApp方法，定义在 app_util.cpp中。
- 三个参数：**Node 指针、应用类型、状态结构体指针。**

- **初始化新协议的 Timers**：除了初始化状态结构体，还需要初始化Timer。由于节点上可能有多个同一类型的应用实例，因此，应用层 timer 会经常访问**消息的 info 字段**以确定所属的应用实例。应用层Timer 由专门的结构体类型 AppTimer 来描述（4.2.3 节）。
  - **AppTimer 包含的信息**：
    - Timer 类型：
    - Connectioin Id：所属的连接
    - SourcePort：所属的session
  - **Timer 的类型**有三种：
    - APP_TIMER_SEND_PKT：发包用定时器，用于模拟发包速率。
    - APP_TIMER_UPDATE_TABLE：更新本地 Table； 或删除超时表项等；
    - APP_TIMER_CLOSE_SESS：用于关闭session。
  - 在新协议的ClientInit最后要进行Timer 的初始化，对 Client 来讲，主要是设定开始发包的时间。另外，这里也展示了如何在Timer 中保存额外的信息，比如用 sourcePort 来区分不同的Client 实例。



- **至此，重新编译 exata**，检验系统完整性。重要的是要补充user-models 的 Makefile，并激活新插件，然后，编译成功。



- **实现服务端初始化**
  - 对于基于 UDP 的应用，比如 CBR，服务端开始只是监听服务端口，收到客户端的第一个数据包时，再进行服务端的初始化，在事件处理部分完成代码。

4.2.5.6 实现事件派发器 Implementation of Event Dispatcher

- **EXata 的事件处理顺序**：节点事件处理==》层事件处理==》协议事件处理。NODE_ProcessEvent----> Layer: **APP_ProcessEvent** （定义在 application.cpp）有两个参数： node 和 msg，msg 中携带有协议类型信息，可以通过APP_GetProtocolType 获取，然后通过 switch 语句调用每个协议的事件处理函数。
- 在 APP_ProcessEvent 中添加新协议的事件处理函数：确保应用协议类型 APP_MYPROTOCOL_CLIENT和 APP_MYPROTOCOL_SERVER 已在 AppType 中添加完毕（application.h）：

```
6797 /*
6798  * NAME:        APP_ProcessEvent.
6799  * PURPOSE:     call proper protocol to process messages.
6800  * PARAMETERS:  node - pointer to the node,
6801  *              msg - pointer to the received message.
6802  * RETURN:      none.
6803  */
6804 void APP_ProcessEvent(Node *node, Message *msg)
6805 {
6806     unsigned short protocolType;
6807     protocolType = APP_GetProtocolType(node,msg);
6808     NetworkDataIp *ip = (NetworkDataIp *) node->networkData.networkVar;
6809     NetworkDataIcmp *icmp = (NetworkDataIcmp*) ip->icmpStruct;
6810
```

… ….

```
6843        switch(protocolType)
6844        {
6845            case APP_TIMER:
6846            {
6847                AppLayerTimerTest(node, msg);
6848                break;
6849            }
6850            case APP_ROUTING_BELLMANFORD:
6851            {
6852                RoutingBellmanfordLayer(node, msg);
6853                break;
6854            }
```

… …

```
6948            case APP_CBR_SERVER:
6949            {
6950                AppLayerCbrServer(node, msg);
6951                break;
6952            }
6953            // LuoJT: exercise
6954            case APP_MYPROTOCOL_CLIENT:
6955            {
6956                AppLayerMyprotocolClient(node, msg);
6957                break;
6958            }
6959            case APP_MYPROTOCOL_SERVER:
6960            {
6961                AppLayerMyprotocolServer(node, msg);
6962            }
```

- 注意到：新协议的两个事件处理函数尚未完成，下面逐个实现。
- 新协议客户端事件处理 AppLayerMyprotocolClient
  - **按消息（事件）类型分别处理**：EXata 中的所有事件类型定义在 api.h中，首先，在 api.h 中添加新协议的事件类型。注意添加在 MSG_DEFAULT之前。

```
1137    /*
1138     * Any other message types which have to be added should be added before
1139     * MSG_DEFAULT. Otherwise the program will not work correctly.
1140     */
1141     // LuoJT: exercise, 2022-12-15
1142     MSG_APP_MYPROTOCOL_NewEvent1,
1143     MSG_APP_MYPROTOCOL_NewEvent2,
1144
1145    MSG_DEFAULT                                 = 10000
1146 };
```

  - 在app_myprotocol.h/.cpp中添加 AppLayerMyprotocolClient 和 AppLayerMyprotocolServer 方法，模仿 CBR 的实现。
  - **AppLayerMyprotocolClient(node, msg)** 方法：
    - 根据 msg 中的 eventType，在 MSG_APP_TimerExpired 类型中继续进行下面处理：
    - 通过 **MESSAGE_ReturnInfo**(msg) 提取该 消息的 Info，获取timer 的指针，然后进行 info 填充；
    - 进一步根据 timer->sourcePort 取协议实例的指针：clientPtr：

AppMyprotocolClientGetMyprotocolClient （这是一个新方法，需要添加）

- 接下来，根据 timer 的类型进行不同的处理。如果是 APP_TIMER_SEND_PKT，则根据startTime、interVal 进行发包（包括构造包头、添加virtualPayload等，并利用 appData::appTrafficSender::appUdpSend 发出），并设定下一个发包的新 timer （利用 **AppMyprotocolClientScheduleNextPkt**，需要添加）。
- 添加从消息中搜寻特定源端口的实例信息新方法：**AppMyprotocolClientGetMyprotocolClient：**

```
// Search for a myprotocol client data structure
AppDataMyprotocolClient *
AppMyprotocolClientGetMyprotocolClient(Node *node, short sourcePort)
{
    AppInfo *appList = node->appData.appPtr;
    AppDataMyprotocolClient *myprotocolClient;

    for (; appList != NULL; appList = appList->appNext)
    {
        if (appList->appType == APP_MYPROTOCOL_CLIENT)
        {
            myprotocolClient = (AppDataMyprotocolClient *) appList->appDetail;

            if (myprotocolClient->sourcePort == sourcePort)
            {
                return myprotocolClient;
            }
        }
    }

    return NULL;
}
```

- **添加设定下一个发送包定时器的新方法：AppMyprotocolClientScheduleNextPkt.**主要是创建一个 Timer Message，并分配 Message Info，携带 source port、timer type 等身份和类型信息，最后，利用 MESSAGE_Send方法发出。【定时器的设定也是靠发出消息来实现，这是与其他仿真工具最大的不同！】

```
// Schedule the next packet the client will send.
void
AppMyprotocolClientScheduleNextPkt(Node *node, AppDataMyprotocolClient *clientPtr)
{
    AppTimer *timer;
    Message *timerMsg;

    // Create a timer message
    timerMsg = MESSAGE_Alloc(node,
        APP_LAYER,
        APP_MYPROTOCOL_CLIENT,
        MSG_APP_TimerExpired);

    // Add timer info
    MESSAGE_InfoAlloc(node, timerMsg, sizeof(AppTimer));

    timer = (AppTimer *)MESSAGE_ReturnInfo(timerMsg);
    timer->sourcePort = clientPtr->sourcePort;
    timer->type = APP_TIMER_SEND_PKT;

#ifdef DEBUG
    {
        char clockStr[24];
        printf("MYPROTOCOL Client: Node %u scheduling next data packet\n",
            node->nodeId);
        printf("    timer type is %d\n", timer->type);
        TIME_PrintClockInSecond(clientPtr->interval, clockStr);
        printf("    interval is %sS\n", clockStr);
    }
#endif

    MESSAGE_Send(node, timerMsg, clientPtr->interval);

}
```

- **至此，客户端的事件处理已完成！！！【编译正常，当然拼写出了不少错误！】==》服务端**

- AppLayerMyprotocolServer 方法
  - 与 Client 不同，Server 端主要是从下层**接收 packet**。
  - 所处理的事件：`MSG_APP_FromTransport` 来自传输层（UDP）
  - 本方法在收到该事件时被调用，然后处理收到的packet。
  - 处理流程：

```
// Server event dispatcher
void
AppLayerMyprotocolServer(Node *node, Message *msg)
{
    char error[MAX_STRING_LENGTH];
    AppDataMyprotocolServer *serverPtr;

    switch (msg->eventType)
    {
        case MSG_APP_FromTransport:
            {
                UdpToAppRecv *info;
                MyprotocolData data;

                ERROR_Assert(sizeof(data) <= MYPROTOCOL_HEADER_SIZE,
                    "Myprotocol Data size can be greater than MYPROTOCOL_HEADER_SIZE

                // Get info and packet
                info = (UdpToAppRecv *) MESSAGE_ReturnInfo(msg);
                memcpy(&data, MESSAGE_ReturnPacket(msg), sizeof(data));
```

- **确定协议实体**：要利用 source address 和 source port 信息，使用 App*Server Get*Server 方法来搜索（本节点可能有多个协议实例）。 **【“Get 方法”】**
- 如果搜索返回 NULL，说明是一个新连接（该会话或连接的第一个包），则需要创建一个新的服务端实例。要调用 App*ServerNew*Server 方法来实现。**【“New” 方法】**

```
                // Search the server instance
                serverPtr = AppMyprotocolServerGetMyprotocolServer(node,
                    info->sourceAddr,
                    data.sourcePort);

                // New connection, so create a new server instance
                if (serverPtr == NULL)
                {
                    serverPtr = AppMyProtocolServerNewMyprotocolServer(node,
                        info->destAddr,
                        info->sourceAddr, data.sourcePort);
                }
```

- **从消息中提取 packet 或净荷进行处理**：MESSAGE_ReturnPacket；使用 MESSAGE_ReturnPacketSize 获取包的大小；
- 记得最后 MESSAGE_Free，释放处理过消息的内存空间。
- 查询新协议 Server 的方法：App*Server Get*Server。每个节点会维护一个 AppInfo 类型的应用列表，里面包含每个应用的重要标识信息，比如远端地址、源端口号、会话时间信息等。应用节点在创建后即在节点注册，保存相关信息，参加“New 方法”。

```
// Search for server instance
AppDataMyprotocolServer *
    AppMyprotocolServerGetMyprotocolServer(
    Node *node, Address remoteAddr, short sourcePort)
{
    AppInfo *appList = node->appData.appPtr;
    AppDataMyprotocolServer * myprotocolServer;

    for (; appList != NULL; appList = appList->appNext)
    {
        myprotocolServer = (AppDataMyprotocolServer *) appList->appDetail;

        if ((myprotocolServer->sourcePort == sourcePort) &&
            IO_CheckIsSameAddress(
            myprotocolServer->remoteAddr, remoteAddr))
        {
            return myprotocolServer;
        }

    }

    return NULL;

}
```

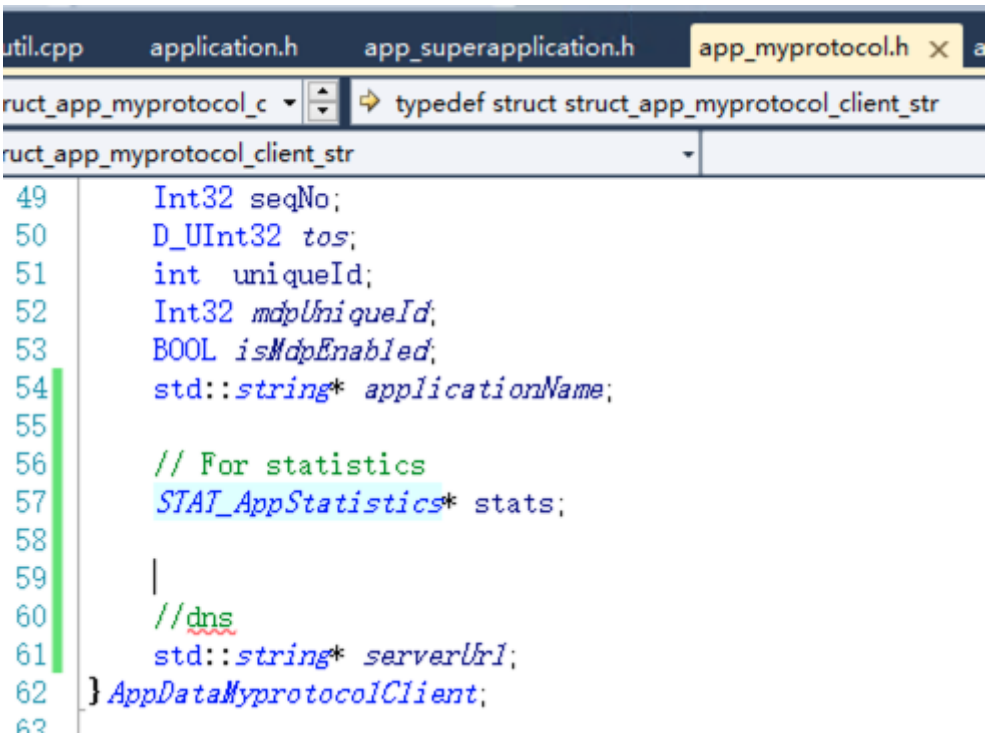
- **创建新 Server 的方法**：创新新的 Server 实例及 相应的重要 Info，并完成注册（将 Info 加入 appList）

```
// Create a new server data structure
AppDataMyprotocolServer *
AppMyprotocolServerNewMyprotocolServer(
    Node *node, Address localAddr, Address remoteAddr, short sourcePort)
{
    AppDataMyprotocolServer *myprotocolServer;

    // Allocate a new server
    myprotocolServer = (AppDataMyprotocolServer *)
        MEM_malloc(sizeof(AppDataMyprotocolServer));
    memset(myprotocolServer, 0, sizeof(AppDataMyprotocolServer));

    // Fill in info
    memcpy(&(myprotocolServer->localAddr), &localAddr, sizeof(Address));
    memcpy(&(myprotocolServer->remoteAddr), &remoteAddr, sizeof(Address));
    myprotocolServer->sourcePort = sourcePort;
    myprotocolServer->sessionStart = node->getNodeTime();
    myprotocolServer->sessionFinish = node->getNodeTime();
    myprotocolServer->sessionIsClosed = FALSE;
    myprotocolServer->seqNo = 0;
    myprotocolServer->uniqueId = node->appData.uniqueId++;

    APP_RegisterNewApp(node, APP_MYPROTOCOL_SERVER, myprotocolServer);

    return myprotocolServer;

}
```

- 重新编译，成功！！！

4.2.5.7 收集和报告统计量 Collecting and Reporting Statistics

1. **EXata 的统计方法**：
   a. 从代码上看，无论 EXata 5.1 或 EXata 7.0，统计量的收集方法跟 Programmer‘s Guide 上描述的方法都一样。代码实现的方式并不是直接把统计量（比如 numBytesRecvd）直接定义为 AppDataCbrServer 的成员，而是用一个 统计类的成员来实现的。这个类是 STAT_AppStatistics，即在 AppDataCbrServer 结构体中包含一个 **STAT_AppStatistics 类型的指针** *stats。由它在客户端、服务端状态变化时进行计数，最后统一打印或报告。
   b. 类 STAT_AppStatistics 从 STAT_ModelStatistics 派生而来，
2. 声明统计量：
   a. CBR 是在客户端和服务端数据结构体中都增加了专用的统计成员：STAT_AppStatistics* stats；我们也在新协议的数据结构体中添加统计成员。



```
typedef
struct struct_app_myprotocol_server_str
{
    Address localAddr;
    Address remoteAddr;
    short sourcePort;
    clocktype sessionStart;
    clocktype sessionFinish;
    clocktype sessionLastReceived;
    BOOL sessionIsClosed;
    Int32 seqNo;
    int uniqueId;

    // For statistics
    STAT_AppStatistics* stats;

}AppDataMyprotocolServer;
```

b. 在新协议实例在创建时（**New 方法**），注册之前将统计量指针置为 NULL。

```
156
157     // Initialize stats NULL
158     myprotocolClient->stats = NULL;
159
160     // Register the new app instance
161     APP_RegisterNewApp(node, APP_MYPROTOCOL_CLIENT, myprotocolClient);
162
163     return myprotocolClient;
164 }
```

```
    // initialize the stats
    myprotocolServer->stats = NULL;

    APP_RegisterNewApp(node, APP_MYPROTOCOL_SERVER, myprotocolServer);
```

c. **Server 在创建实例（创建新连接时）创建统计量指针**，并进行会话开始时的初始化：

```
454                     // Create statistics
455                     if (node->appData.appStats) //?
456                     {
457                         serverPtr->stats = new STAT_AppStatistics(
458                             node,
459                             "myprotocolServer",
460                             STAT_Unicast,
461                             STAT_AppReceiver,
462                             "MYPROTOCOL Server");
463                         serverPtr->stats->Initialize(
464                             node,
465                             info->sourceAddr,
466                             info->destAddr,
467                             STAT_AppStatistics::GetSessionId(msg),
468                             serverPtr->uniqueId);
469                         serverPtr->stats->SessionStart(node);
470                     }
```

3. **在协议服务端事件dispatcher 中进行统计量更新**（in AppLayer*Server 中）。通过调用统计类 API 完成统计。

   在收到packet 的统计中，要求先调用 FragmentReceivedDataPoints 然后调用 MessageReceivedDataPoints。

```
479
480                 // Update statistics
481                 if (data.seqNo >= serverPtr->seqNo)
482                 {
483                     if (node->appData.appStats)
484                     {
485                         //
486                         serverPtr->stats->AddFragmentReceivedDataPoints(
487                             node,
488                             msg,
489                             MESSAGE_ReturnPacketSize(msg),
490                             STAT_Unicast);
491                         serverPtr->stats->AddMessageReceivedDataPoints(
492                             node,
493                             msg,
494                             0,
495                             MESSAGE_ReturnPacketSize(msg),
496                             0,
497                             STAT_Unicast);
498                     }
499
500                 }
```

a. 客户端在创建实例时创建统计量指针，并进行初始化

```
    // Set applicatio name, and Create statistics
    if (serverAddr.networkType != NETWORK_INVALID && node->appData.appStats)
    {
        std::string customName;
        if (clientPtr->applicationName->empty())
        {
            customName = "MYPROTOCOL Client";
        }
        else
        {
            customName = *clientPtr->applicationName;
        }

        clientPtr->stats = new STAT_AppStatistics(
            node,
            "myprotocol",
            STAT_Unicast,
            STAT_AppSender,
            customName.c_str());
        clientPtr->stats->Initialize(node,
            clientAddr,
            serverAddr,
            (STAT_SessionIdType)clientPtr->uniqueId,
            clientPtr->uniqueId);
        //clientPtr->stats->setTos(tos);

    }
```

b. **客户端 Event Dispatcher 中更新统计量。**【这里有个疑问：客户端没找到在哪里更新发送的数据和字节？这里只看到 session closed 的更新】

```
        {
            data.type = 'd'; // regular data? Why not update statistics?
        }
        else
        {
            data.type = 'c';  // The last one?, session closed
            clientPtr->sessionIsClosed = TRUE;
            clientPtr->sessionFinish = node->getNodeTime();

            // Update statistics: session closed
            if (node->appData.appStats && clientPtr->stats)
            {
                clientPtr->stats->SessionFinish(node);
            }

        }
```

只是在 info 更新时，进行了统计量多保存，但没注意到统计值本身的更新，比如调用 FragmentSent 或 MessageSent。这里是个疑问！

```
321         // dns ??
322         AppUdpPacketSendingInfo pktSendInfo;
323
324         pktSendInfo.itemSize = clientPtr->itemSize;
325         pktSendInfo.stats = clientPtr->stats;    // update stats in info
326
327         pktSendInfo.fragNo = NO_UDP_FRAG;
328
```

4. **打印统计量**
    a. 打印统计量函数用于在仿真结束时，将协议统计量打印到文件中，用于后期数据分析。
    b. 客户端和服务端添加两个**统计量打印方法**，在协议的**终止化函数（Finalizatioin）**中被调用。
        i. App***ClientPrintStats：
        ii. App***ServerPrintStats
5. 添加动态统计量
    a. 就是在仿真过程中能观察到的统计量，不是待仿真结束时分析的，可以通过 GUI 添加和观察的。参见 5.2.3 节.

4.2.5.8 终止化（Finalization）

- **主要功能**：打印统计量到统计文件。
- **调用**：分级进行：分区终止PARTITION_Finalize -->分层终止APP_Finalize-->协议终止***_Finalize
- **要做两件事**：
    - 在应用层终止 APP_Finalize 方法中调用新协议的终止操作；

```
(全局范围)                                  APP_Finalize(Node * node)
7809            }
7810            // LuoJT: for exercise
7811            case APP_MYPROTOCOL_CLIENT:
7812            {
7813                AppMyprotocolClientFinalize(node, appList);
7814                break;
7815            }
7816            case APP_MYPROTOCOL_SERVER:
7817            {
7818                AppMyprotocolServerFinalize(ndoe, appList);
7819                break;
7820            }
7821
7822            case APP_FORWARD:
7823            {
7824                AppForwardFinalize(node, appList);
7825                break;
7826            }
```

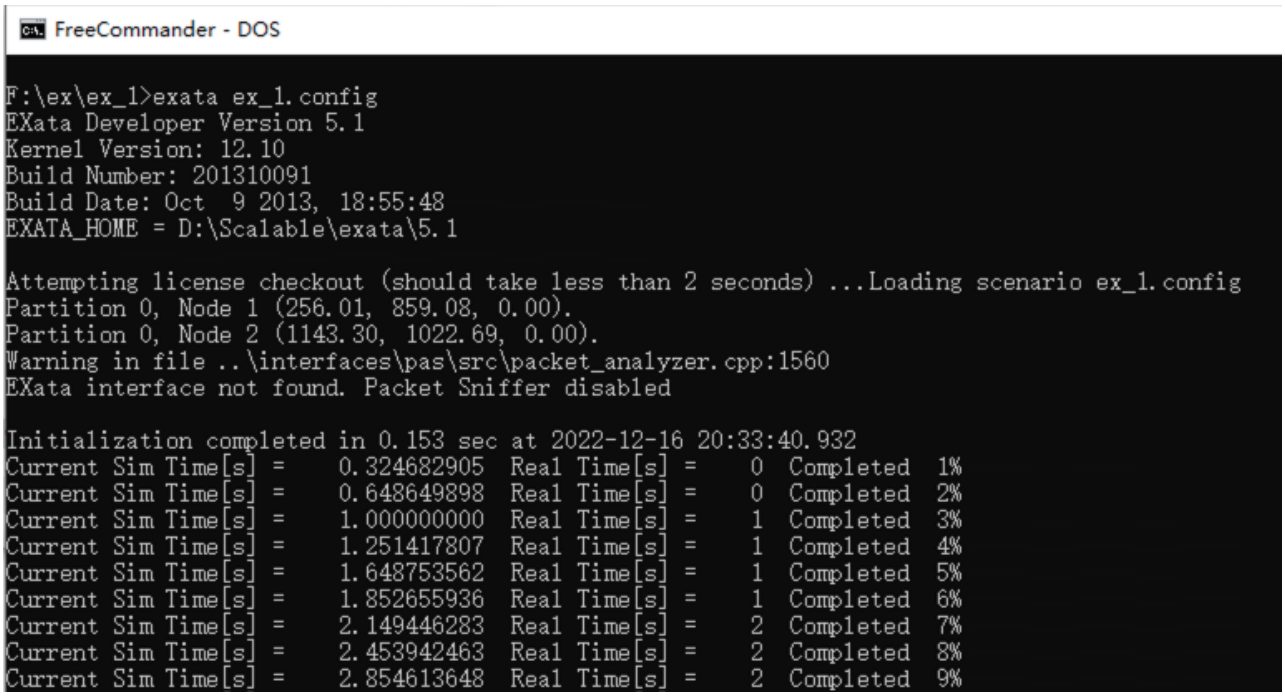
    - 实现新协议的终止操作：客户端终止App***ClientFinalize 和 服务端终止App***ServerFinalize。

4.2.5.10 GUI中集成新协议

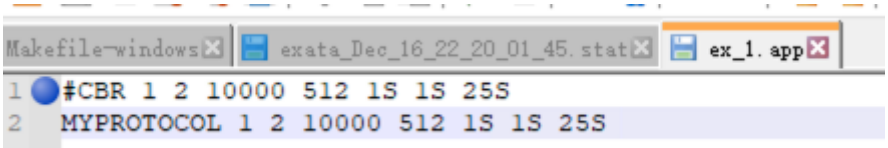
参考 5.1.4 节

补充：（1）命令行检验新协议

- 命令行运行 EXata：参见《User Guide》 2.1 Running EXata from the Command Line Interface
- 以运行 F:\ex\ex_1/ex_1.config 为例
- 在 DOS 窗口下，切换路径至 F:\ex\ex_1\，输入：exata ex_1.config，则可以启动运行。



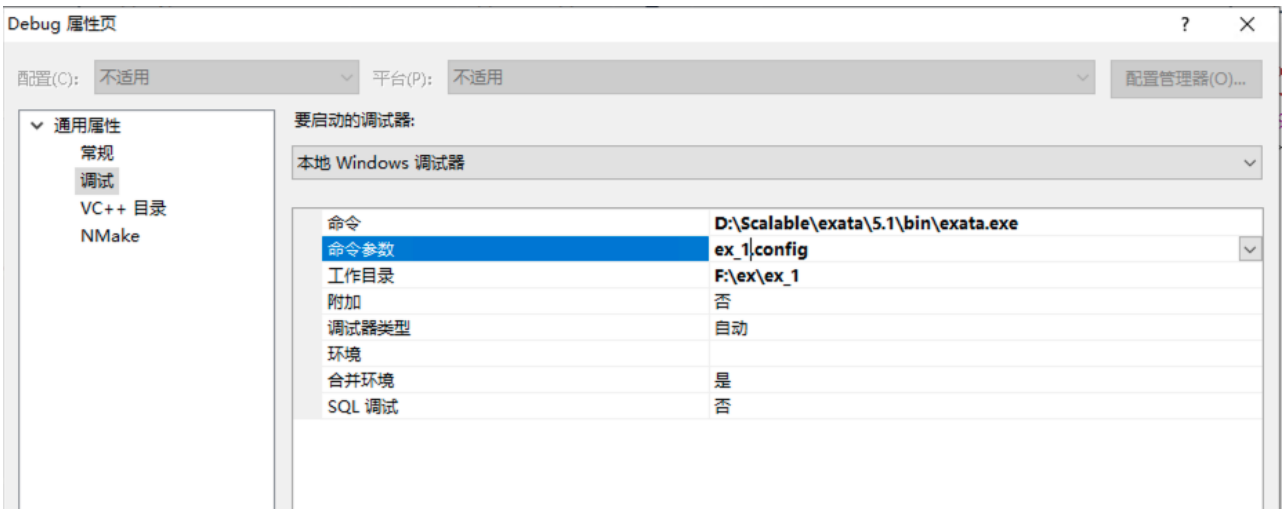
- 正常运行后生成的统计量将保存在“*.stat”文件中。
- EXata 运行有三个主要的输入文件：
  - *.config: 主输入文件，确定网络场景和参数。
  - *.nodes: 确立节点的初始位置，注意这里定义的只是“初始位置”，后续位置的变化则取决于 mobility model。
  - *.app: 确立节点上运行的应用程序
- 修改ex_1.app文件，在节点 1 和 2 之间添加一个 MYPROTOCOL 业务，测试新协议：



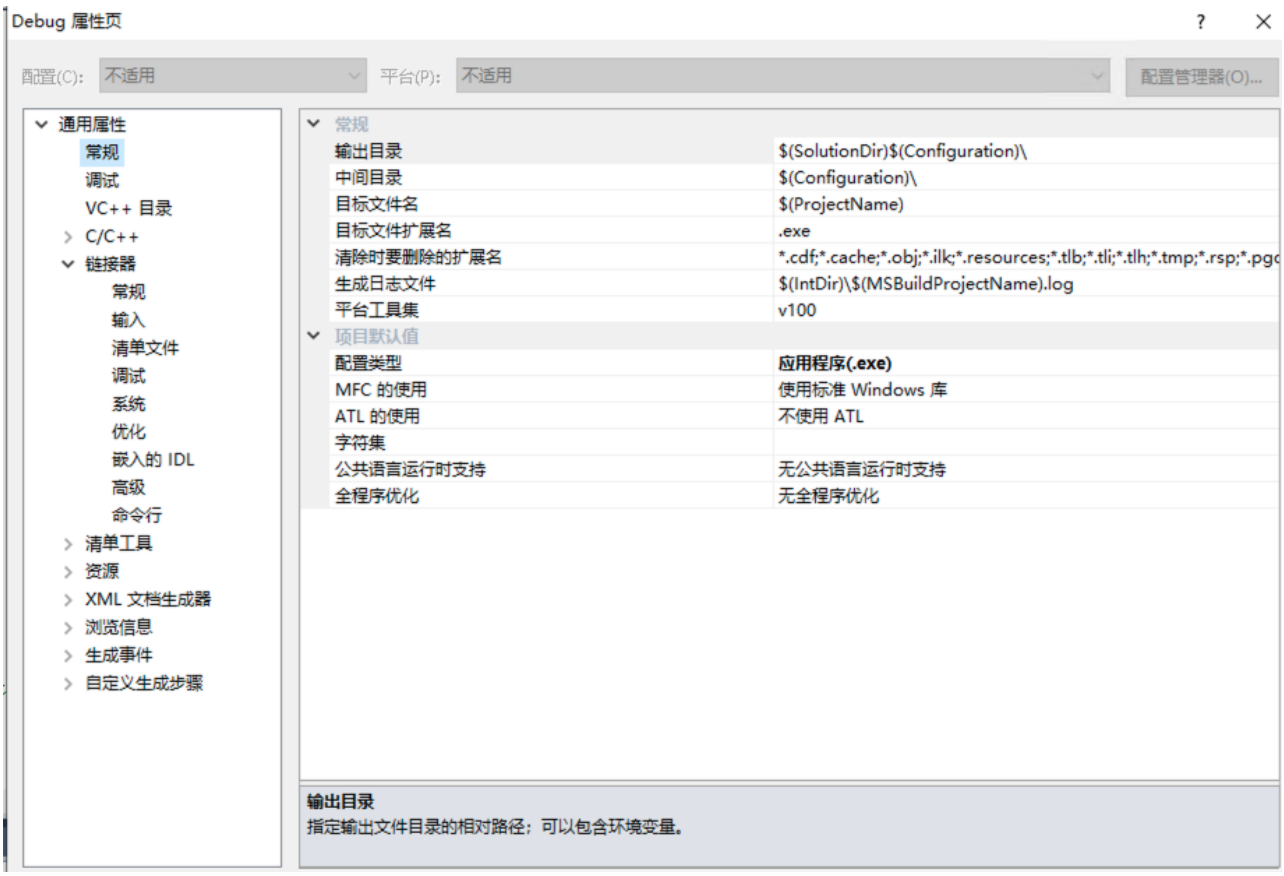
- 运行 ex_1.app，到 %2 非正常退出【**出错！**】需要进行 debug!

```
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).
Warning in file ..\interfaces\pas\src\packet_analyzer.cpp:1560
EXata interface not found. Packet Sniffer disabled

Initialization completed in 0.172 sec at 2022-12-17 15:08:08.995
Current Sim Time[s] =    0.324682905  Real Time[s] =    0  Completed  1%
Current Sim Time[s] =    0.648649898  Real Time[s] =    0  Completed  2%

F:\ex\ex_1>
```

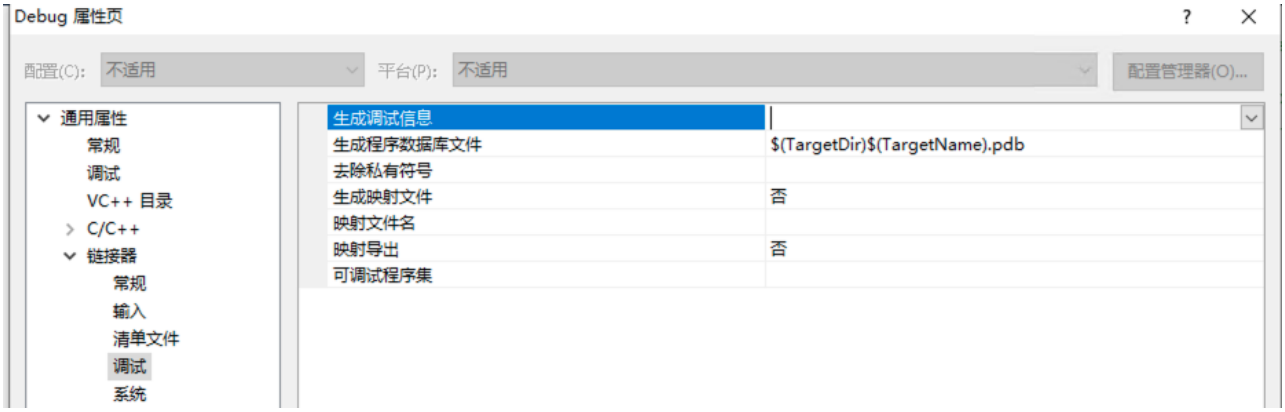
- 调试 EXata，设置 Debug 属性页，命令参数选择 ex_1.config，工作目录：F:\ex\ex_1\



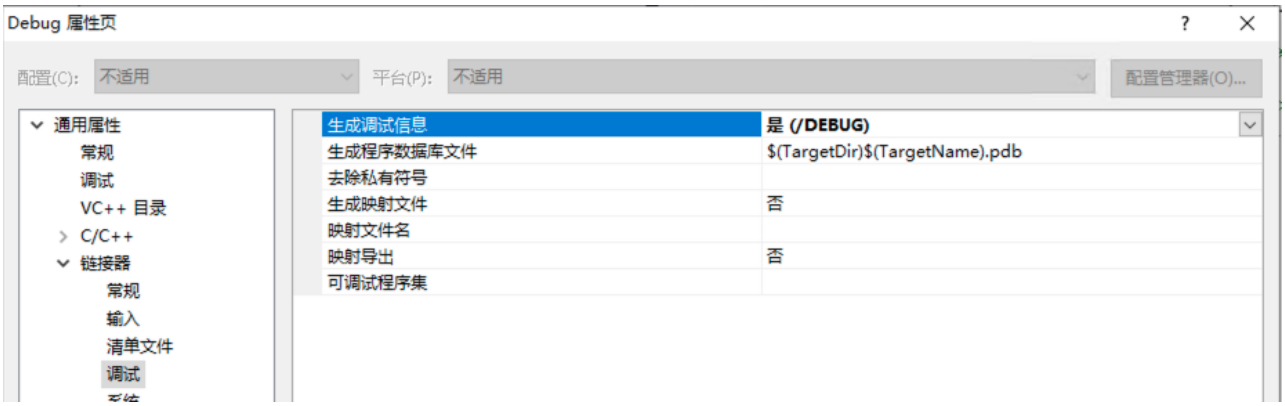
- 【忽略】发现“通用属性”中缺乏“链接器”，将“常规”--》“配置类型”==》“应用程序（.exe）”.（原默认为“生成文件”）则 Debug 属性页多了很多选项



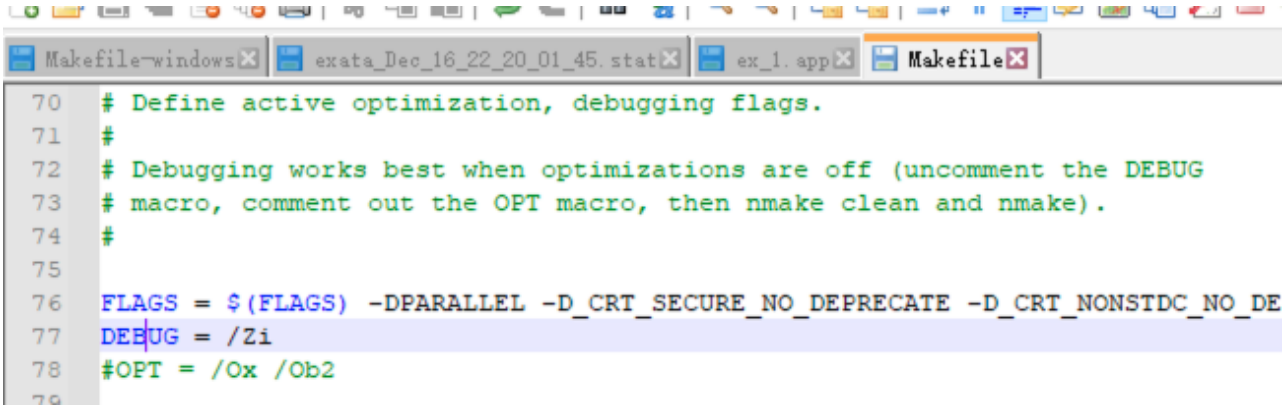
检查“调试”页面，发现无配置：==? 是/DEBUG



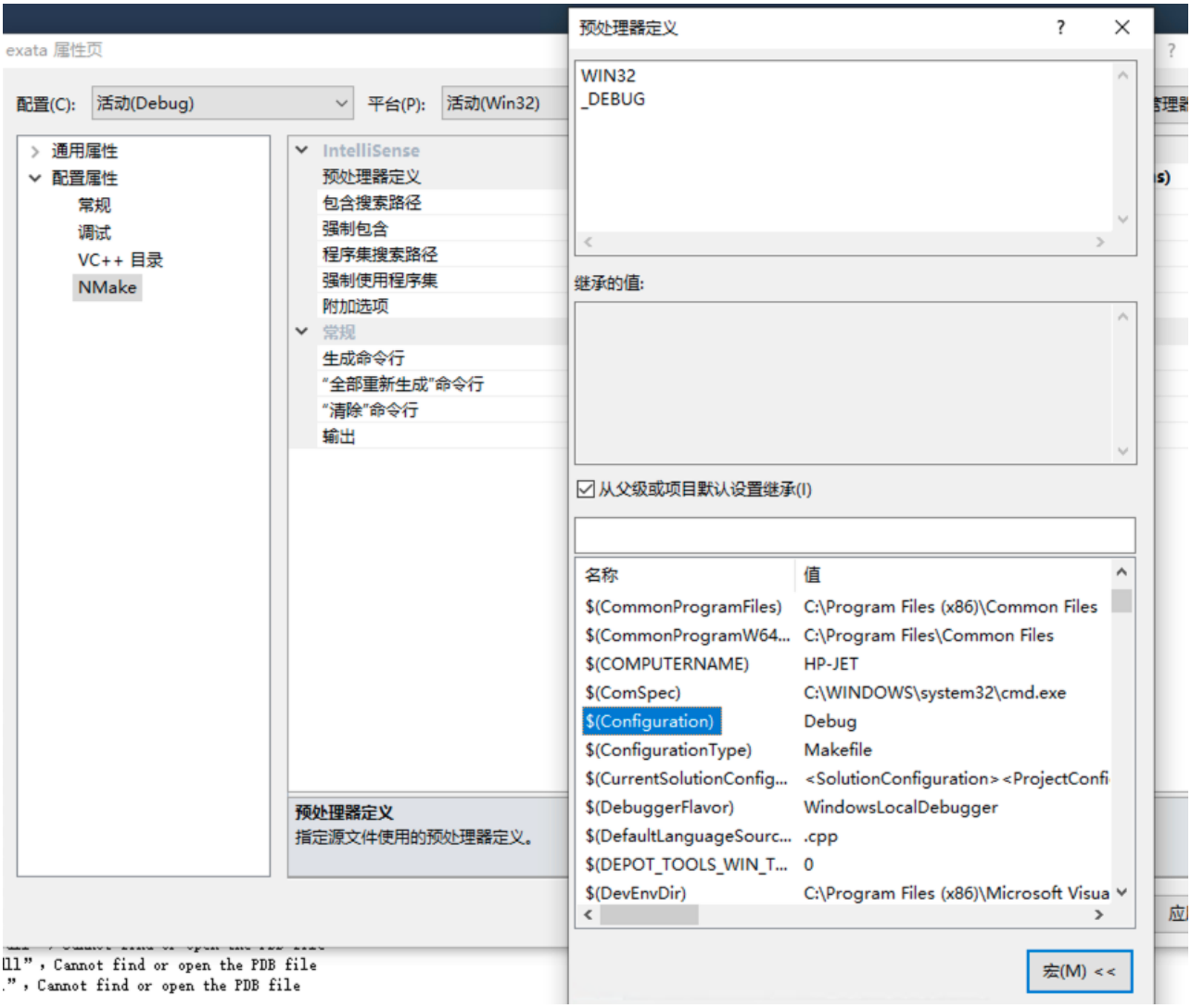
- 【忽略】改为：



- 【忽略】但改为“生成应用程序”后，编译通不过！！！配置类型设置为“生成文件”，能正常编译，同样能生成新的 exata.exe。
- 【忽略】判断应该是 Makefiel 设置的问题，检查发现EXATA_HOME/main/下 Makefile 原设置为优化编译，而非Debug，去掉DEBUG 注释，而将 OPT 注释掉【待验证】



- 调试问题：目前修改 main/Makefile，在 CMD 下运行 nmake rebuild，可以在 main/ 下找到 *.pdb 文件，在 VS 2010 下启动调试，不再提示“找不到调试信息”，但在 app_cbr.cpp 中设置断点，调试运行听不下来！
- 调试问题：如果在 VS 2010 IDE 环境下，重新编译 DEBUG 版本，启动调试会提示“无法找到调试信息 或者调试信息不匹配”，即找不到 pdb 文件。
- 了解已定义的预处理宏：
  - 在Project 属性页：NMake ---预处理器定义---编辑---宏---

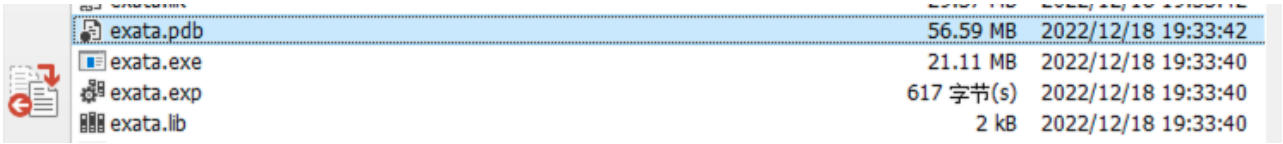


  - **几个常用宏**：$SolutionDir = ~/exata/5.1/; $Configuration = Debug

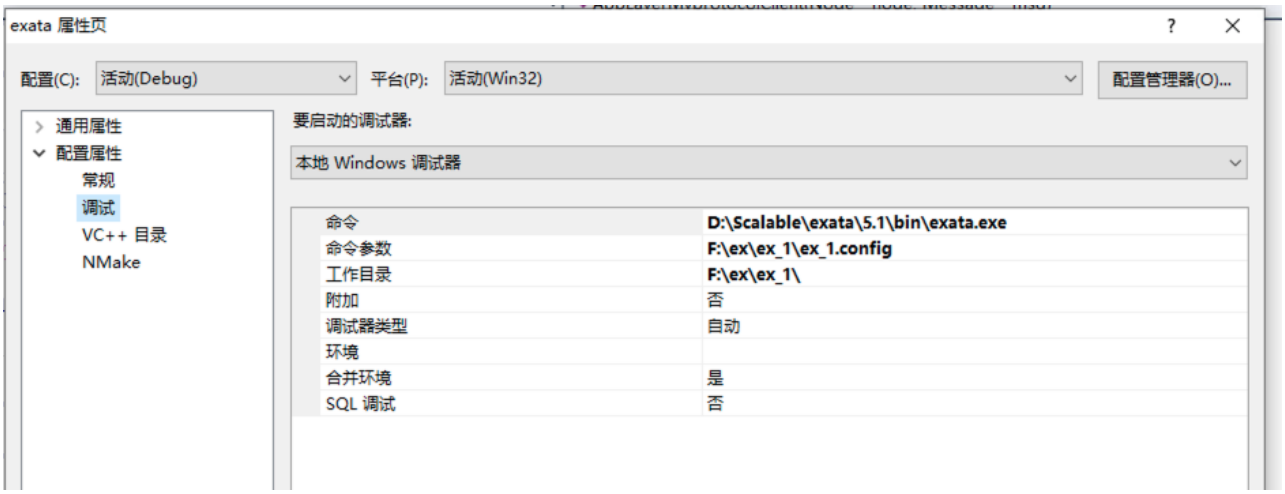
- 解决调试启动的问题，首先要确定**配置类型选择的问题**：（1）生成文件；（2）生成应用程序（exe）。前者能顺利编译，但（IDE启动生成时）没有生成调试信息（*.pdb），而且是 Guide 文档指明的方式; 后者无法正常编译，但只有选这种类型，才会有更多的配置属性，包括生成调试信息等链接器的相关配置。【答案：肯定是第（1），“生成文件”，即 Makefile，意味着读取 Makefile，一切配置依赖 Makefile】【debug 版本运行时，为什么没有输出_DEBUG 宏下的打印信息呢？这还是个问题？】
- **关于调试信息*.pdb**：（https://blog.51cto.com/u_14349334/3476888）
  - 程序数据库（PDB）文件保存着调试和项目状态信息，使用这些信息可以对程序的调试配置进行增量链接。当以 /ZI 或 / Zi （用于C/C++）生成时，将创建一个 PDB 文件。
  - 在 Visual C++ 中，/Fd 选项用于命名由编译器创建的pdb 文件。当使用向导在 Visual Studio 中创建项目时，/Fd 选项被设置为创建一个名为 project.PDB 的 PDB 文件。
- 如果使用“生成文件”创建 C/C++ 应用程序，并指定 /ZI 或 /Zi 而不指定 /Fd 时 【如何指定？in main/Makefile】，则最终生成两个 PDB 文件：
  - VC100.PDB：更笼统地说是 VCx0.PDB，其中 x 代表 VC 的版本。该文件存储各个OBJ文件的所有调试信息并与项目生成文件驻留在同一个目录中。VCx0.PDB 文件由编译器生成，插入的信息包括类型信息，但不包括函数定义等符号信息。因此，即使每个源文件都包含公共头文件（比如windows.h），这些头文件中的 typedef 也只存储一次，而不是每个 OBJ 文件中都存在。

- Project.PDB：该文件存储 .exe 文件的所有的调试信息。对于本机C/C++代码，它驻留在 /Debug 目录中。对于托管代码，它驻留在 /WINDEBUG 子目录中。此PDB 文件由链接器生成，包含项目的 EXE 文件的调试信息，即完整的调试信息（函数原型），而不仅仅是在 VCx0.PDB文件中找到的类型信息。这两个 PDB 文件都允许增量更新。链接器还在其创建的 .exe 或 .dll 文件中嵌入. pdb 文件的路径。

- 调试器使用 EXE 或 DLL 文件中的 PDB 路径查找 project.PDB 文件。如果调试器在该位置无法找到 PDB 文件或者如果路径无效，调试器将搜索包含 EXE 的路径，即在“选项”对话框（“调试”文件夹，“符号”节点）中指定的符号路径。调试器不会加载与所调试的二进制不匹配的 PDB。
- 【推断】感觉这是 VS 2010的 一个 Bug：在 IDE 下 Build 的无法按照 Makefile 的指示生成 PDB 文件。；必须在命令行下用 nmake 来运行生成！很奇怪！
- 【非常重要！！！】正确的 DEBUG 启动过程【就像 Guide 手册描述的一样，但是很奇怪！因为 IDE 下的生成同样配置的使用 nmake】
  - 在命令行下，在 Makefile 中打开 Debug 开关后，**运行 nmake clean； nmake**，此时在 main/ 下生成 vc100.pdb，在 bin/ 下生成 exata.pdb；



  - 然后打开 VS 2010，“打开项目/解决方案”，选择刚生成的 exata.exe，创建工程。
  - 右键点击生成的 exata.exe【注意，这里与第一次 compile 时“基于已有的代码生成方案”不同】
  - 在生成的工程视图中，右键点击“exata” --- 属性--debugging，配置如下：命令参数=选择场景文件X.config；工作目录是场景文件所在目录。
  -



  - （修改ex_1.app，将 CBR 改为新增第应用协议 MYPROTOCOL，保存。）在 app_myprotoco.cpp 中设置断点，启动调试运行，即可在断点出终止。
  - 修改代码后，可以用 IDE 的 Build，但如果要 Debug，必须用 命令行下的 nmake clean；nmake才能正常生成调试信息文件（可能是个 Bug！）
  - 因此，**命令行下 nmake 生成后，在 IDE 启动 Debug** 时，提示 EXE 文件已过时，是否重新生成时，一定要选 否（N）！！！

- CBR 应用在UDP发送时参数

名称	值	类型
sentMsg	0x04465f38 {m_flags=0 next=0x00000000 m_partitionData=0x0438acf8 ...}	Message *
payloadSize	472	int
packetSendingInfo	{itemSize=512 stats=0x044669b0 appParam={...} ...}	AppUdpPacketSendingInfo
data	{sourcePort=1024 type='d' isMdpEnabled=0 ...}	struct_app_cbr_data
cbrData	0x01b93820 ""	char [40]
appParam	{m_NodeId=0 m_SessionInitiator=1 m_ReceiverId=2 ...}	StatsDBAppEventParam
timer	0x0446e058 {type=1 connectionId=0 sourcePort=1024 ...}	AppTimer *
node	0x04400058 {prevNodeData=0x00000000 nextNodeData=0x044045d8 nodeIndex	Node *
msg	0x0446e010 {m_flags=0 next=0x00000000 m_partitionData=0x0438acf8 ...}	Message *
buf	0x01b93ce0 ""	char [200]
error	0x01b93da8 "??O□["	char [200]
clientPtr	0x044668f0 {localAddr={...} remoteAddr={...} interval={...} ...}	struct_app_cbr_client_str *

局部变量

名称	值	类型
this	0x04457d58 {udpPacketBuffer=[0]() tcpOpenRequestBufferMap=[0]() urlToAddr	AppTrafficSender * const
packetSendingInfo	{itemSize=512 stats=0x044669b0 appParam={...} ...}	AppUdpPacketSendingInfo
node	0x04400058 {prevNodeData=0x00000000 nextNodeData=0x044045d8 nodeIndex	Node *
url	""	std::basic_string<char,std::char_traits<char>,st
clientAddr	{networkType=NETWORK_IPV4 interfaceAddr={...} }	Address
appType	APP_CBR_CLIENT	AppType
sourcePort	1024	unsigned short
msg	0x04465f38 {m_flags=0 next=0x00000000 m_partitionData=0x0438acf8 ...}	Message *
info	0x01b92e3c {sourceAddr={...} sourcePort=31684 destAddr={...} ...}	AppToUdpSend *
msgBuffered	false	bool

- MYPROTOCOL 在 UDP 发送时的参数

局部变量

名称	值	类型
sentMsg	0x04465f38 {m_flags=0 next=0x00000000 m_partitionData=0x0438acf8 ...}	Message *
data	{sourcePort=1025 type='d' seqNo=0 ...}	struct_app_myprotocol_data
pktSendInfo	{itemSize=1024 stats=0x0446e378 appParam={...} ...}	AppUdpPacketSendingInfo
payloadSize	984	int
myprotocolData	0x01b93cb4 "□□d"	char [40]
timer	0x0446fb20 {type=1 connectionId=0 sourcePort=1025 ...}	AppTimer *
node	0x04400058 {prevNodeData=0x00000000 nextNodeData=0x044045d8 nodeIndex	Node *
msg	0x0446fad8 {m_flags=0 next=0x00000000 m_partitionData=0x0438acf8 ...}	Message *
buf	0x01b93ce0 "1852655936"	char [200]
error	0x01b93da8 "?????? lj?"	char [200]
clientPtr	0x0446e268 {localAddr={...} remoteAddr={...} interval={...} ...}	struct_app_myprotocol_client_str *

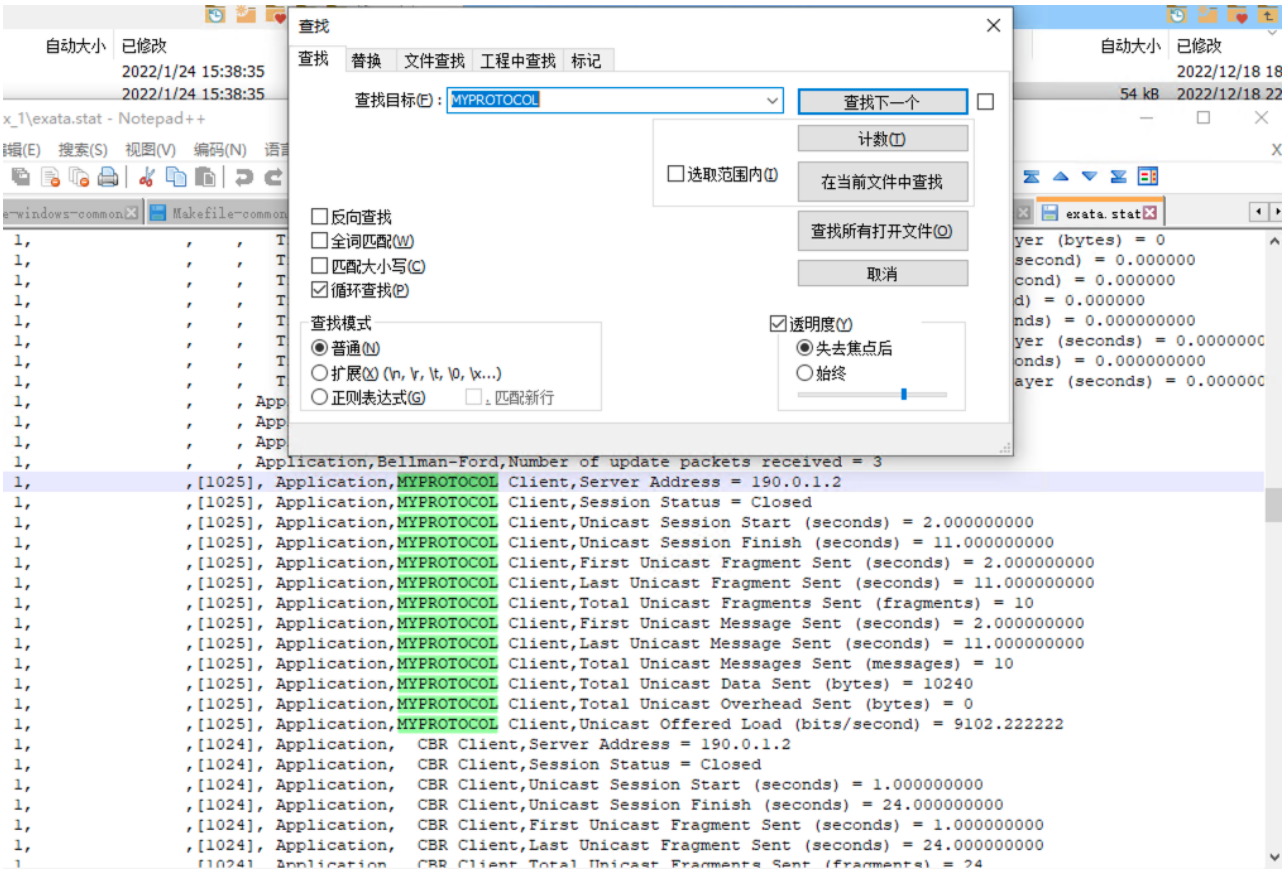
-

名称	值	类型
endTime	25000000000	__int64
sessionIsClosed	0	int
itemsToSend	10	unsigned
itemSize	1024	unsigned
sourcePort	1025	short
seqNo	1	int
tos	{value=0 }	D_UInt32
uniqueId	0	int
mdpUniqueId	0	int
isMdpEnabled	0	int
applicationName	0x0446dac0 ""	std::basic
stats	0x0446e378 {m_Type=0x0446fa28 "myprotocol" m_CustomName="MYPROTOC	STAT_App
serverUrl	0x00000000	std::basic

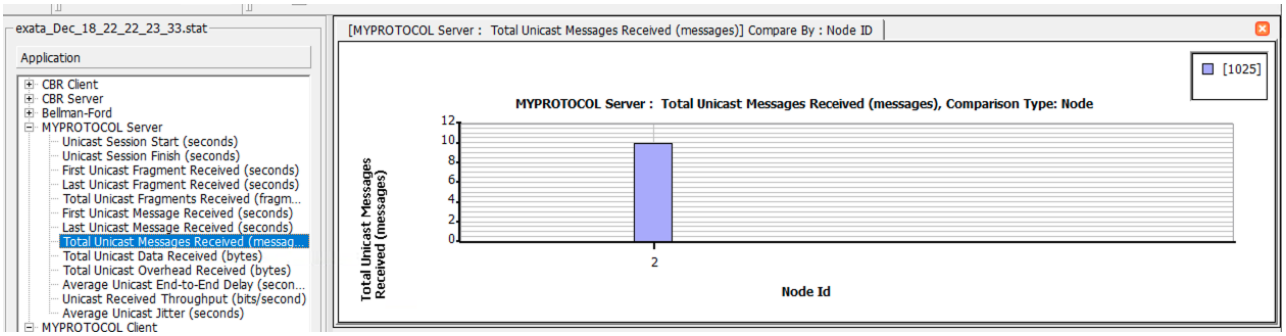
  - 对比 CBR 发现： serverUrl 地址指针为 Null，造成*serverUrl 出错！对比检查发现是在 Client 创建时该指针没有初始化，导致在调用时仍为“飞指针”。补充以下初始化语句：

```
190     myprotocolClient->tos = tos;
191
192     //dns
193     myprotocolClient->serverUrl = new std::string();
194     myprotocolClient->serverUrl->clear();
195
196     if (appName)
```

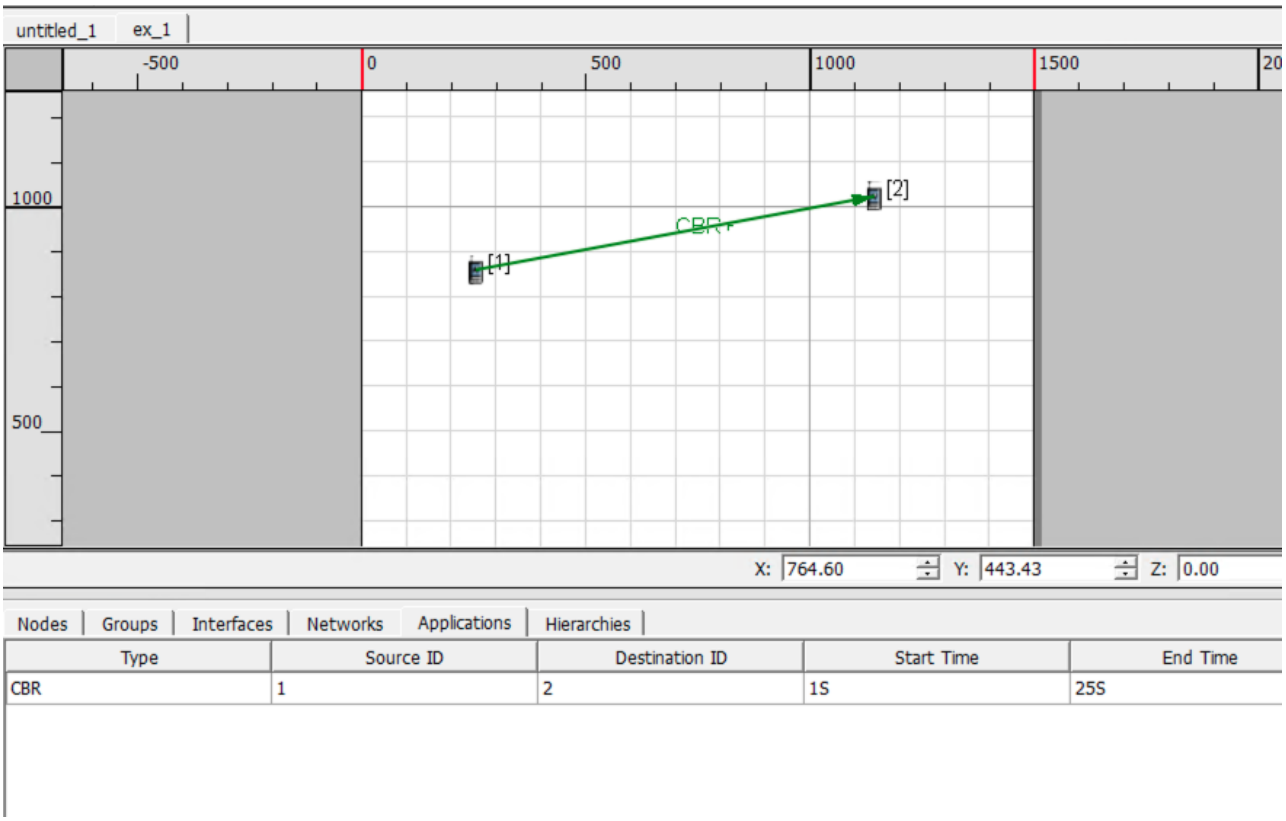
  - 【牢记】代码修改保存后，仍由命令行 nmake 进行编译。
  - 搞定！！！新协议运行成功！查看统计量输出



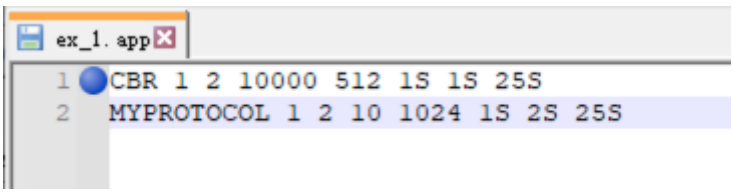
  - 此时，在 EXata Analyzer 中统计量已有 MYPROTOCOL 的统计输出，但在 Architect 中还无法显示，或进行应用配置【就在下一步】



- 此时，在 Architect 中能看到有两个业务，因为链路上显示“CBR+”，但在下面 Application 列表中只有一个 CBR



- 目前的 ex_1.app文件如下：



-

5 自定已GUI 显示新协议

参考 5.1.4

目标：将用户自己开发的新协议或模型与 Architect 集成在一起，能够通过 Architect 进行设置、运行和分析。

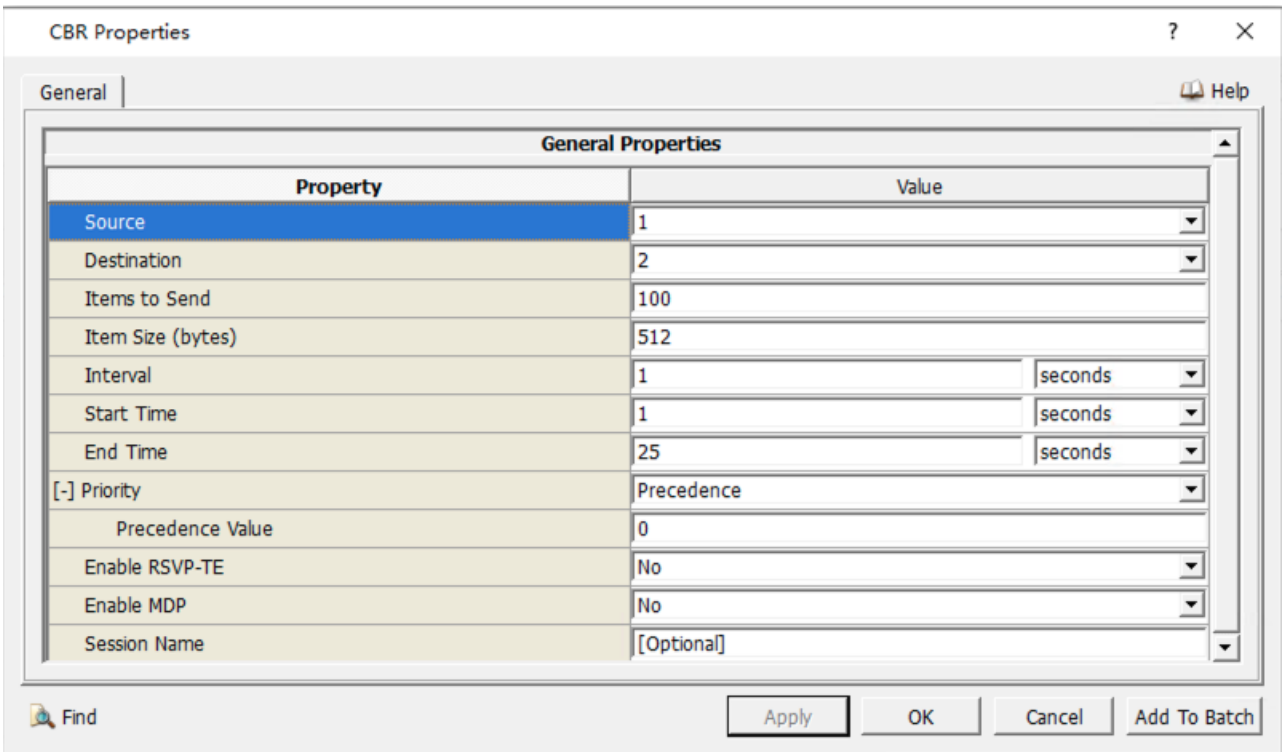
5.1 定制Architect 的 design mode

5.1.1 GUI 描述文件

- **5.1.1.0一些基本概念**
  - **属性编辑器 Property Editor**

    所谓属性编辑器即设置对象属性的对话框，用于输入对象的参数，背后对应创建或修改配置文件，可能是场景文件*.config、*.app等。

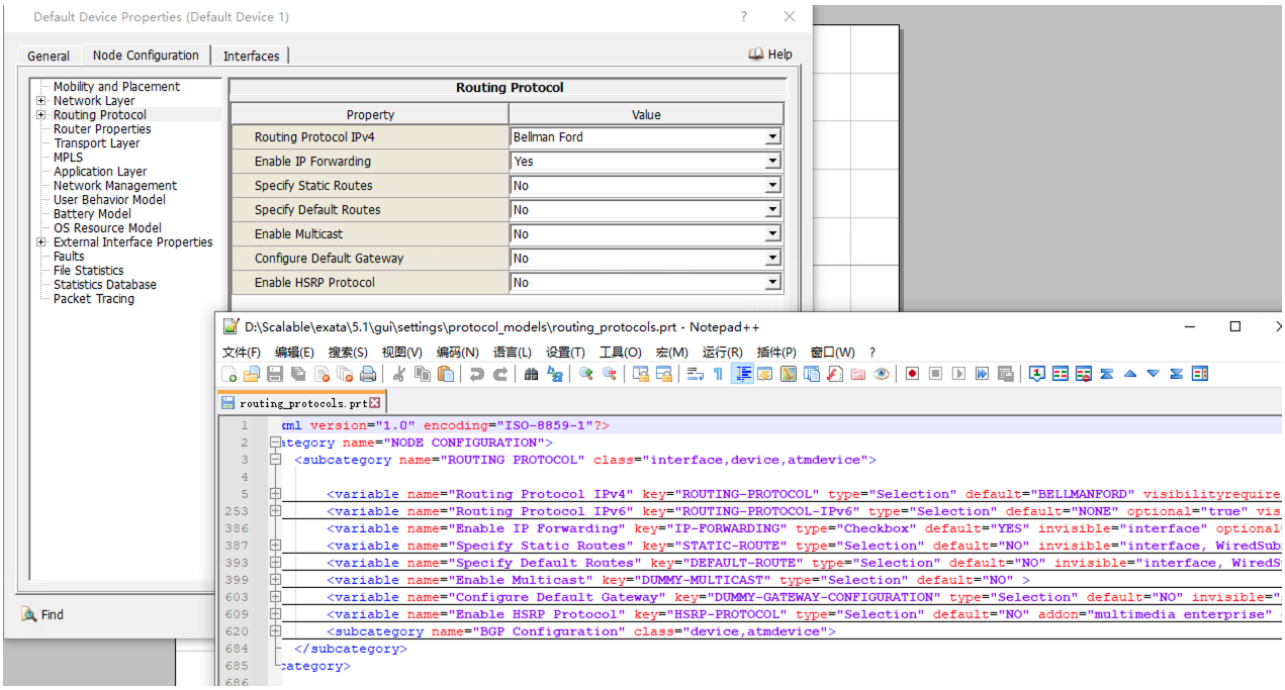
    总的来说有两类属性编辑器，我喜欢称之为“属性框”，因为GUI上它就表现为一个对话框，这个对话框可能有多个 TAB 或列表。
    - **组件特性**，比如device、link、application 等；
    - **全局和接口特性**。
  - **Segment of a property editor**
    - 对象的一组部分特性，作为一组出现在一个编辑器或多个编辑器。
    - 通常作为属性框的一个页面。
  - 三种 setting files：Architect 使用 setting files 构建特性编辑框。
    - 组件文件（Component Files）：
      - *. cmp;
      - 对应每个编辑框，不含 Segment 组合的结构信息；
      - EXATA_HOME/gui/setting/components/ 下
      - XML 格式，下面为 cbr.cmp 例子，对应 CBR 属性编辑



cbr. cmp

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<root version="1.0">
<category name="CBR Properties" singlehost="false" loopback="enabled" propertytype="CBR" displayname="CBR">
    <variable name="Source" key="SOURCE" type="SelectionDynamic" keyvisible="false" optional="false" help_ref="Constant Bit Ra
    <variable name="Destination" key="DESTINATION" type="SelectionDynamic" keyvisible="false" optional="false" help_ref="Const
    <variable name="Items to Send" key="ITEM-TO-SEND" type="Integer" default="100" min="0" keyvisible="false" help="Number of
    <variable key="ITEM-SIZE" type="Integer" name="Item Size (bytes)" default="512" min="24" max="65023" keyvisible="false" he
    <variable name="Interval" key="INTERVAL" type="Time" default="1S" keyvisible="false" optional="false" help_ref="Constant B
    <variable name="Start Time" key="START-TIME" type="Time" default="1S" keyvisible="false" optional="false" help_ref="Consta
    <variable name="End Time" key="END-TIME" type="Time" default="25S" keyvisible="false" optional="false" help_ref="Constant
    <variable name="Priority" key="PRIORITY" type="Selection" default="PRECEDENCE" keyvisible="false" optional="true" help_ref
      <option value="TOS" name="TOS" help="value (0-255) of the TOS bits in the IP header">
        <variable name="TOS Value" key="TOS-BITS" type="Integer" default="0" min="0" max="255" keyvisible="false" optional="
      </option>
      <option value="DSCP" name="DSCP" help="value (0-63) of the DSCP bits in the IP header">
        <variable name="DSCP Value" key="DSCP-BITS" type="Integer" default="0" min="0" max="63" keyvisible="false" optional=
      </option>
      <option value="PRECEDENCE" name="Precedence" help="value (0-7) of the Precedence bits in the IP header">
        <variable name="Precedence Value" key="PRECEDENCE-BITS" type="Integer" default="0" min="0" max="7" keyvisible="false
      </option>
    </variable>
    <variable name="Enable RSVP-TE" key="ENABLE-RSVP-TE" type="Selection" default="N/A" keyvisible="false" optional="true" hel
      <option value="N/A" name="No" />
      <option value="RSVP-TE" name="Yes" />
    </variable>
<variable name="Enable MDP" key="MDP-ENABLED" type="Selection" default="N/A" keyvisible="false" optional="false" help_ref="Co
     <option value="N/A" name="No" />
     <option value="MDP-ENABLED" name="Yes">
       <variable name="Specify MDP Profile" key="MDP-PROFILE" type="Selection" default="N/A" keyvisible="false" optional="fals
          <option value="N/A" name="No" />
          <option value="MDP-PROFILE" name="Yes">
             <variable name="MDP Profile Name" key="MDP-PROFILE-NAME" type="Text" default="[Required]" keyvisible="false" op
          </option>
       </variable>
     </option>
   </variable>
   <variable name="Session Name" key="APPLICATION-NAME" type="Text" default="[Optional]" spacesAllowed="false" optional="true"
</category>

</root>
```

- 共享描述文件（ Shared Description Files）：
  - 对应每个 **segment**，segment 可以在多个多个编辑框中出现，对 Segment 的描述放在共享描述文件中。
  - EXATA_HOME/gui/settings/protocol–models/
  - 扩展名 prt
  - 这是routing_protocols.prt 例子



- 工具集描述文件（Toolset Description Files）：设定 Architect 中 Toolset 的描述，比如 device、application、link 等的 icons。
  - EXATA_HOME/gui/settings/EXATA_HOME/gui/settings/Toolsets/standard.xml

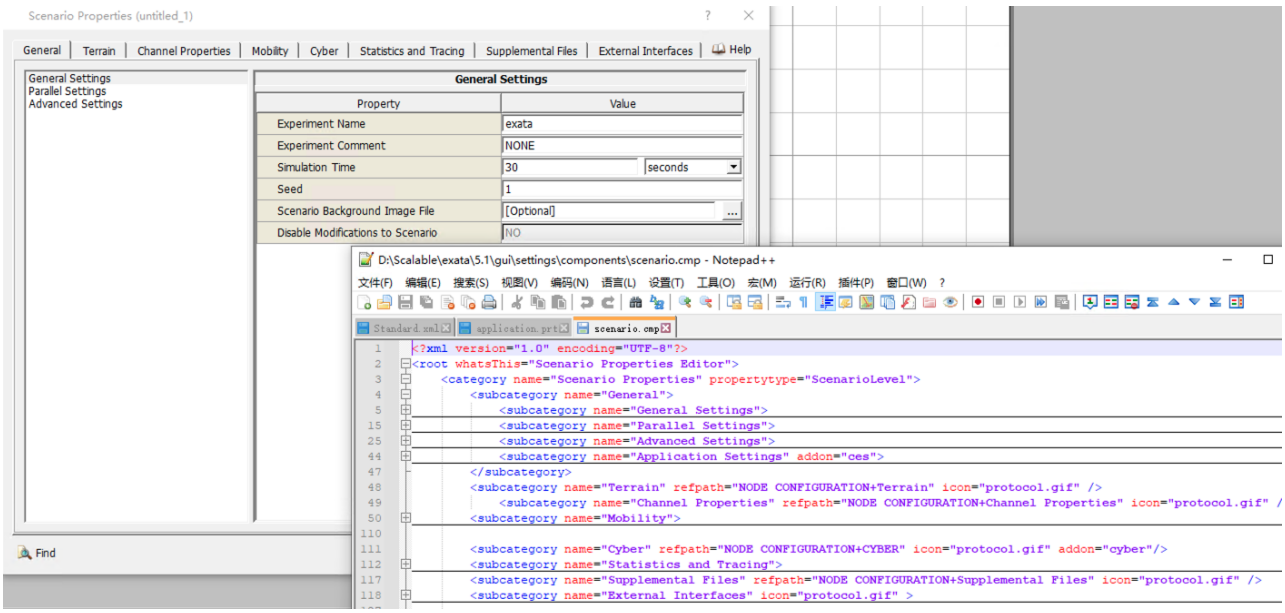
5.1.1.1 GUI 设置文件的结构

GUI 设置文件是标准的 XML 格式，采用共同的结构。每类设置文件有四个可以嵌套的层次：

- **类category**：category 是最高层级的元素。在组件文件中，category 对应 属性框，在共享描述文件中，cetegory 对应共享 segment 的组。其下面可以有多个subcategory 和 variable 元素。
- **子类subcategory**：下面可以有 subcategory 和 option。Subcategory 对应属性框一个 segment。
- **变量variable**：下面可以继续有 varible 和 option。variable 对应属性框的一个参数。
- **选项option**：其下面可以继续有 variable 。如果一个参数是 list 类型，option 对应一个枚举列表。

5.1.1.2 组件文件（Component Files）

全局属性框：scenario.cmp；下面是 cmp 文件和属性框的对照。



可见：“Scenario Properties”为 Category，一级Subcategory 为“General”、“Terrain”、“Channel Properties”、“Mobility”等；“General” 还有下一级的 Subcategory：“Genral Settings”、“Parallel Settings”、“Advanced Settings”等。

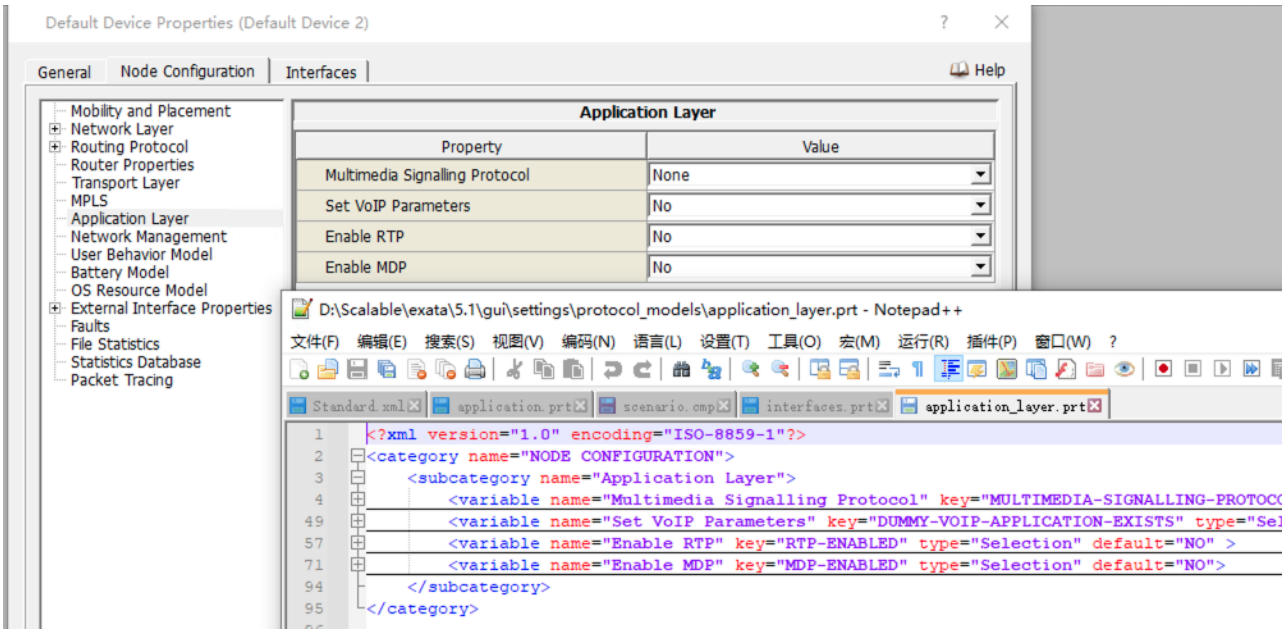
工具箱中的每个对象：比如 defaultnode.cmp 对应 的default device, switch.cmp, CBR.cmp 对应 CBR applcation。

5.1.1.3 Shared Description Files

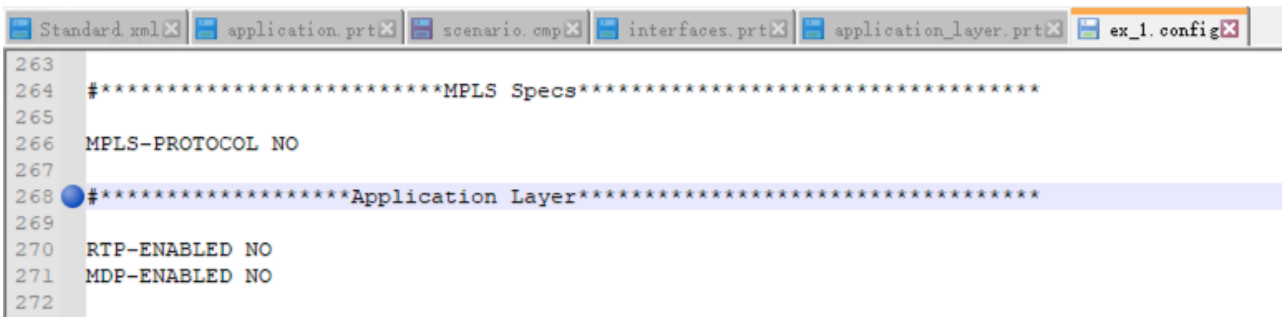
SDF 对应各个属性框共享的 **段 Segments**，Table 5-2 列出来不同的 SDF 以及对应生成的配置文件。第二列列出来该 SDF 包含的 category 和 subcategories 。比如 application.prt 包括一个 category “NODE CONFIGURATION” 和一个 subcategory：“APPLICATION”，其他 subcatories 是 “APPLICATION” 的子 subcategory。

- 例子一：

下图是一个 default device 属性框 Application Layer 段的实例：它对应的 SDF 魏 application_layer.prt，它包含一个 category：NODE CONFIGURATION，一个 subcategory：“Application Layer”，下面又包含 4 个 variables：“Multimedia Signaling Protocol”、“Set VoIP Parameter”、“Enable RTP”、“Enable MDP”，对应对话框中四个选项。



对应场景配置文件中的一个部分 “Application Layer”



5.1.2 设置文件元素Setting File Elements

5.1.2.1 元素 category

category 是 GUI setting 文件的最高级别。

- 对于compoent files：category 代表该属性框；
- 对于 shared Descriptioin File：category 代表 一个段描述的组 group of segment description。

表 5-3 列举了 category 元素的主要属性，特别是对于 Component 来讲，其 propertytype 对应组件的标识符，是必选项。

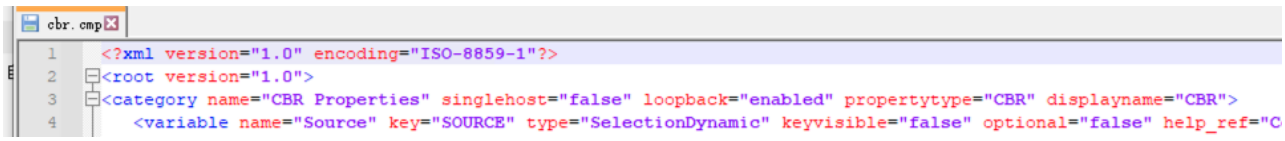
TABLE 5-3. Attributes of the `category` Element

Attribute Name	Attribute Values or Type	Description
`name` *Required*	String	Name of the `category`. For a component file, this is the name displayed in the associated property editor's title bar.
`addon` *Optional*	Comma-separated list of strings	Name(s) of the addon module(s) in which this `category` is available. At least one of the listed addon modules should be installed for this `category` to be available. **Note: For Scalable Network Technologies use only.**
`propertytype` *Required for component files* *Optional for shared description files*	String	Component identifier. The `propertytype` should be unique across all component files.
`displaynme` *Required for component files*	String	This attribute is used only in component files representing applications and specifies the name of the application that is displayed on the canvas.

TABLE 5-3. Attributes of the `category` Element

Attribute Name	Attribute Values or Type	Description
`singlehost` *Optional*	List: • `true` • `false`	This attribute is used only in component files representing applications and specifies whether the application is a single-host application. `true` : Application is a single-host application. `false` : Application is a client-server application.
`Loopback` *Optional*	List: • `true` • `false`	This attribute is used only in component files representing applications and specifies whether the application is loop-back enabled. `true` : Application is a loop-back enabled. `false` : Application is not loop-back enabled.

比如 cbr.cmp 文件：对于“CBR Properties”category 来讲，其“propertytype”=“CBR”，即其组件标识符。



5.1.2.2 subcategory 元素

它对应属性框中显示一个 segment 的属性，用 Tab 或 List 形式。

它的属性见 表 5-4.

5.1.2.3 variable 元素

variable 对应属性框中的一个parameter。其属性见表5-5. 其中，除 name 以为，key、type和default 也都是必选项。

- **name**：对应属性框中参数名字
- **key**：对应confugration 文件中打印出来的标识符（参数名称）
- **type**：参数的数据类型，如果是 selection 类型，对应 list 或 combo-box；支持的类型及相应的默认值见表 5-6.
- **default**：参数的默认值，与参数类型有关。

TABLE 5-5. Attributes of the `variable` Element

Attribute Name	Attribute Values or Type	Description
`name` *Required*	String	Name of the `variable`. When the `variable` is included in a component file (either directly or through a reference), this name is displayed as a parameter name in a property editor.
`key` *Required*	String	Identifier (parameter name) printed to the configuration file, for example, `SEED` or `SIMULATION-TIME`.
`type` *Required*	List See Table 5-6.	Type of the `variable`. This attribute determines the type of the associated parameter and the specialized component used to accept the value of the parameter. For example, if `type` is `Selection`, then the parameter can take a value from a list, and a combo-box with possible values is displayed.
`default` *Required*	See Table 5-6.	Default value of the parameter represented by the `variable`. The default value depends upon the `type`. See Table 5-6.
`help` *Optional*	String	Help text that explains the purpose of the parameter associated with the `variable` and is typically displayed as a tool-tip when the mouse is placed over the parameter name in the property editor.

注意：Java表达式或宏可以作为 variable 各属性的值！

5.1.2.4 option 元素

用于描述某参数的可选项。

5.1.3 使用 shared descriptions

- 一个component 文件描述一个属性框（property editor）的结构信息；
- 一个属性框由一个或多个 段（segment）构成。段表现为 tabs 或 list 项。
- 因此，一个 component 文件可以包含：
  - 段的详细描述
  - 一个描述段的shared description 文件（SDF）。通常用于可能经常出现在多个属性框的segment。
- 这种描述的共享仅**仅限于 segment 的层面**。
- 由于 segment 的描述对应与 subcategory 元素，因此，Component 文件中的 subcategory 元素可以指向 SDF

中的 subcategory。

- 这种指向通过 subcategory 的 refpath 属性实现（5.1.2.2）
- refpath 通过category和 多个 subcategory 之间用“+“ 级联实现，例如

```
NODE CONFIGUARTION+INTERFACES FAULTS
NODE CONFIGURATION+NETWORK LAYER+QoS
COMPONENTS+LINKS+Background Traffic
```

5.1.4 集成新模型

集成不同协议需要修改的文件不完全一样：

- **集成应用协议**（5.1.4.2节）：需要
  - 创建一个新的 component 文件
  - （可能）修改一个或多个 SDF
- **集成其他协议**（5.1.4.1节），只需要
  - 修改 SDF。

修改 GUI 的基本原则：

1. 不要删除或移动任何已有元素。
2. 只在相应位置添加集成新协议需要的元素。
3. 保持 XML 文件架构不变。
4. 遵守每个元素的规则。
5. 【自己添加】**要改动的文件首先做好备份**。【注意扩展名不要用 cmp 或 prt ，否则容易属性框出现重复元素，造成混乱！！！】

5.1.4.1 集成一个新协议：非应用

通过以下几步集成一个应用以为的新协议，比如添加一个新的 IPv4 路由协议：

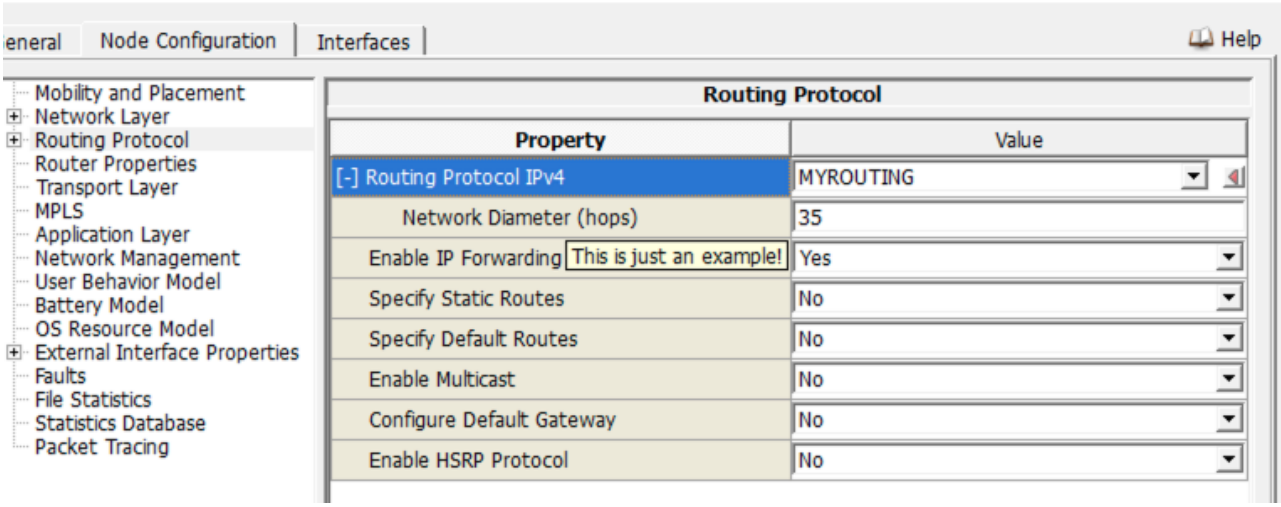
1. 在 subcategory 中找一个最接近的协议，表 5-2 指示所有由 SDF 文件描述的协议。
2. 在 SDF 文件中找到类似协议的部分作为模板；
3. 基于该模板添加新协议的部分。

下面添加一个新的 IPv4 路由协议：MYROUTING，作为例子：

1. 分析 routing_protocol.prt 文件结构，IPv4 路由协议，比如 AODV， 是作为 subcategory “ROUTING PROTOCOL” 下面 变量“Routing Protocol IPv4”的一个 option；
2. 因此，添加一个新的 IPv4 新的路由协议，就是在变量“Routing Protocol IPv4”下面添加一个新的 option。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<category name="NODE CONFIGURATION">
    <subcategory name="ROUTING PROTOCOL" class="interface,device,atmdevice">

        <variable name="Routing Protocol IPv4" key="ROUTING-PROTOCOL" type="Selection" default="BELLMANFORD" visibilityrequires="[NETWORK-PROTOCOL] != 'IPv6'">
            <option value="NONE" name="None"/>
            <option value="MYROUTING" name="MYROUTING">
                <variable name="Network Diameter (hops)" key="ANODR-NET-DIAMETER" type="Integer" default="35" help="This is just an example!" optional="true" />
            </option>
            <option value="ANODR" name="ANODR" addon="cyber">
                <variable name="Network Diameter (hops)" key="ANODR-NET-DIAMETER" type="Integer" default="35" optional="true" />
                <variable name="Node Traversal Time" key="ANODR-NODE-TRAVERSAL-TIME" type="Time" default="40MS" optional="true" />
                <variable name="Active Route Timeout Interval" key="ANODR-ACTIVE-ROUTE-TIMEOUT" type="Time" default="5000MS" optional="true" />
                <variable name="Maximum RREQ Retries" key="ANODR-RREQ-RETRIES" type="Integer" default="2" min="0" optional="true" />
                <variable name="Maximum Number of Buffered Packets" key="ANODR-BUFFER-MAX-PACKET" type="Integer" default="100" optional="true" />
                <variable name="Maximum Buffer Size (bytes)" key="ANODR-BUFFER-MAX-BYTE" optional="true" type="Integer" default="0" />
            </option>
```

3. 备份 routing_protocol.prt 为routing_protocol.prt .backup@20221220
4. 在routing_protocol.prt 添加一个如上图的 option “MYROUTING”，其下只有一个 variable “Network Diameter（hops）”，默认值 35；并特别添加了备注 help=“This is just an example!”【删除了 addon 字段，对于真正的协议实现，这个是重要的。这里只是为了显示 GUI 修改效果，不会实际运行。】
5. 重启 EXata GUI；
6. 打开一个节点属性框，查看routing protocol



7. 在 Routing Protocol v4 中可以选择新增的 “MYROUTING”协议，并可以看到默认参数及协议提示“This is just an example！”
8. GUI 保存场景后，查看场景文件：ex_1.config，可以发现在 Node Configuration 部分，节点 [1] 的路由协议已改为新增的路由协议： MYROUTING

```
cbr.cmp | routing_protocols.prt | ex_1.config

#**************Hierarchy Configuration************************************


#******************Node Configuration************************************


[1]                 HOSTNAME host1
[2]                 HOSTNAME host2
[1]                 ROUTING-PROTOCOL MYROUTING
[1 2]               NODE-PLACEMENT FILE
NODE-POSITION-FILE ex_1.nodes
```

9. 当然这个新增的路由协议发挥不了任何作用，EXata 执行时输入此场景问题，通常会出错。真正的新增协议必须对应正确的组件，以进行正常的参数提取。

总结 EXata 执行的流程：

1. EXata GUI 启动：读取 Component、SDF等文件，初始化、配置 GUI；

2. EXata GUI 创建、配置场景，保存**生成场景文件**：XXX.config, XXX.app, XXX.nodes, ...
3. exata 运行时：exata 读取输入文件（**各种场景文件**），初始化，事件处理，。。。。。
4. exata 运行结束：生成统计分析文件：XXX.stats；
5. EXata Analyzer：读取 stats 文件，显示数据。

5.1.4.2 集成一个新的流量发生器 traffic generator

集成一个新的流量发生器或应用协议，需要经过以下两步：

1. 创建一个新的 CMP 文件，位于 EXATA_HOME/gui/EXATA_HOME/gui/settings/components，对应一个新的属性框；
2. 在工具箱中添加一个按钮：编辑 Standard.xml，位于～/gui/settings/toolsets/，将新按钮对应新的应用。

下面举例，把前面新增的仿造 CBR 的新协议 MYPROTOCOL 添加到 GUI 中，能够在节点上添加该应用。

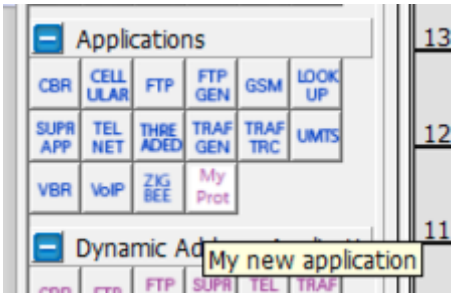
1. 创建 CMP 文件：在～/gui/settins/components/ 下，复制 cbr.cmp 生成 myprotocol.cmp；
2. 将 CBR 部分全部修改为 MYPROTOCOL；删除代码开发时，省掉的属性，比如 precedence。
3. 在工具箱中添加新按钮：修改Standard.xml文件，在category “Applications”下添加一个新的 subcategory “MYPROTOCOL”

```
<category name="Applications">
    <subcategory name="CBR" icon="cbr_a.png" tooltip="Constant Bit Rate" categorytype="Applications" type="App" propertytype="CBR" />
    <subcategory name="CELLULAR-ABSTRACT-APP" addon="cellular" icon="cellular_a.png" tooltip="Abstract Cellular Call" categorytype="Applications" type="App" pr
    <subcategory name="Ftp" icon="ftp_a.png" tooltip="FTP" categorytype="Applications" type="App" propertytype="FTP" />
    <subcategory name="Ftp-Generic" icon="ftp_gen_a.png" tooltip="Generic FTP" categorytype="Applications" type="App" propertytype="FTP/GENERIC" />
    <subcategory name="GSM" addon="cellular" icon="gsm_a.png" tooltip="GSM Call" categorytype="Applications" type="App" propertytype="GSM" />
    <subcategory name="Lookup" icon="lookup_a.png" tooltip="Lookup Traffic Generator" categorytype="Applications" type="App" propertytype="LOOKUP" />
    <subcategory name="Super-App" icon="supr_app_a.png" tooltip="Super Application Traffic Generator" categorytype="Applications" type="App" propertytype="SUPE
    <subcategory name="Telnet" icon="telnet_a.png" tooltip="TELNET" categorytype="Applications" type="App" propertytype="TELNET" />
    <subcategory name="THREADED-APP" icon="threaded_app_a.png" addon="military" tooltip="Threaded Application" categorytype="Applications" type="App" propertyt
    <subcategory name="Traffic-Gen" icon="traf_gen_a.png" tooltip="Traffic Generator" categorytype="Applications" type="App" propertytype="TRAFFIC-GEN" />
    <subcategory name="Traffic-Trace" icon="traf_trc_a.png" tooltip="Trace-based Traffic Generator" categorytype="Applications" type="App" propertytype="TRAFFI
    <subcategory name="UMTS-CALL" icon="umts_a.png" addon="umts" tooltip="UMTS Call" categorytype="Applications" type="App" propertytype="UMTS-CALL" />
    <subcategory name="VBR" icon="vbr_a.png" tooltip="Variable Bit Rate" categorytype="Applications" type="App" propertytype="VBR" />
    <subcategory name="VoIP" icon="voip_a.png" tooltip="Voice over IP" categorytype="Applications" type="App" propertytype="VOIP" />
    <subcategory name="ZIGBEEAPP" icon="zigbee_a.png" tooltip="Application for Zigbee GTS service" categorytype="Applications" type="App" propertytype="ZIGBEEA
    <subcategory name="MYPROTOCOL" icon="myproto.png" tooltip="My new application" categorytype="Applications" type="App" propertytype="MYPROTOCOL" />
</category>
```

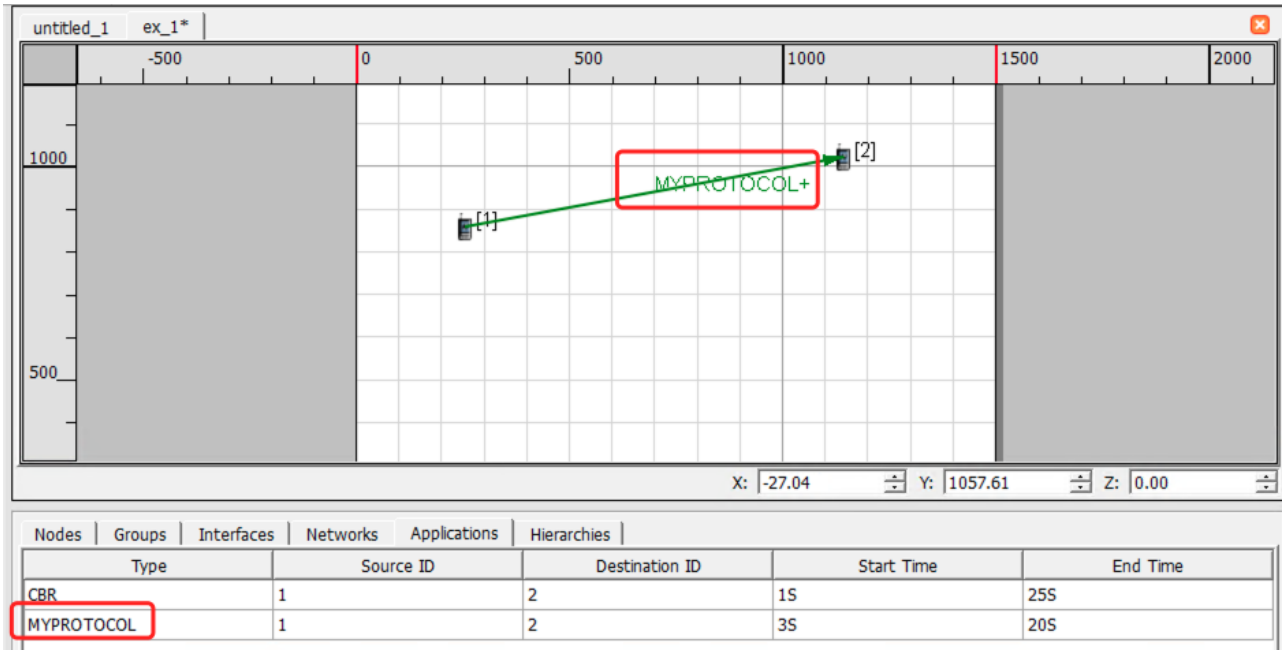
4. 其各个属性：在画图下创建一个 24✖24像素的 icon：myproto.png，修改 tooltip 为“My new application”。

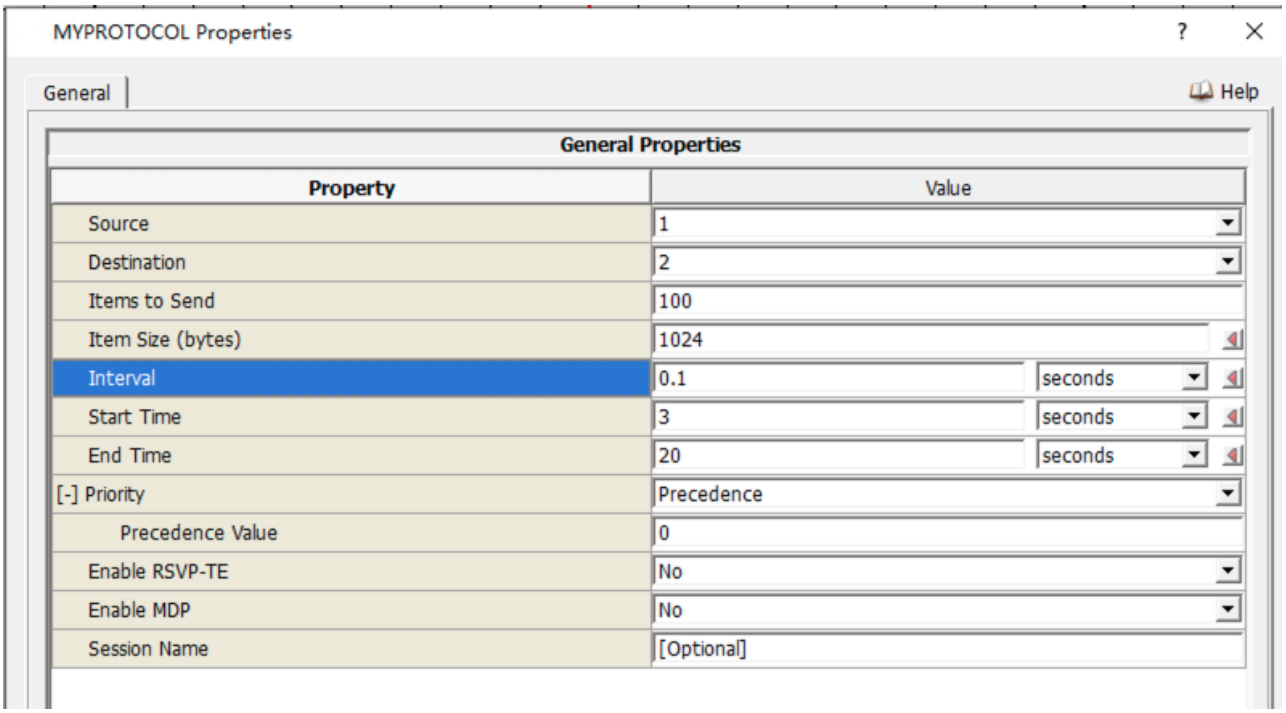
- `name`: Name of the application.
- `icon`: Name of the image file for the button. This image file should be placed in the folder EXATA_HOME/gui/icons/3DVisualizer/icons. The image file should be in PNG format.
- `tooltip`: Text that is displayed when the mouse is placed over the button for the application.
- `type`: This should be “App” for all applications.
- `propertytype`: This should be the same as the `propertytype` of the `category` in the component file for the application. 这里和 CMP 文件建立关联

5. 重启 GUI，检查工具箱中的 Appliucations 中最后多了一个 “My Prot”的按钮，这就是我们新添加的协议。

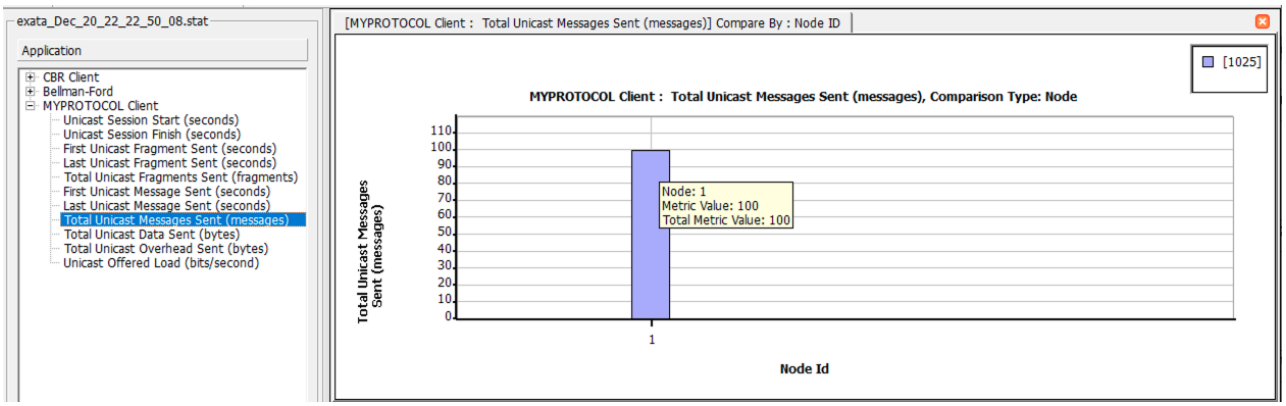


可以在节点上像添加 CBR 一样添加业务，并进行属性设置





- 运行正常，观察统计结果：MYPROTOCOL 发送数据报文 100个。



上述结果表明：新协议 MYPROTOCOL 基本实现与 CBR 类似的功能。

Done。