EXata 扩展（一）

目标：了解 EXata 扩展的基础知识

参考：Chapter 1–3 in EXata 5.1 Programmer‘s Guide

1. Introduction

EXata 由一些核心构建和众多插件构成。

a. 核心构件

i. EXata Simulator

主要特征：1）强大的GUI能力；2）快速仿真能力；3）扩展性，可支持上万个节点的仿真；4）用户或硬件接入等实时仿真（半实物）；5）多平台支持。

ii. EXata Architect

即图形化工具，提供启发式的建模和执行的能力，包括两个模式：Design 和 Visualization。

Design 模式下，Architect 用来创建和设计实验。Visualization 模式下，用于执行和动画演示仿真的过程和结果。

iii. EXata Analyzer

这是一个统计图工具，用于显示仿真过程产生的统计量。

b. EXata Protocol Stack

类似与 TCP/IP网络，EXata 协议栈包括 5 层：Application，Transport， Network， Lin（MAC）和 Physical Layers。通常只允许各层根据定义好的 API 进行通信，但有绕开的方法（4.1 1 节）

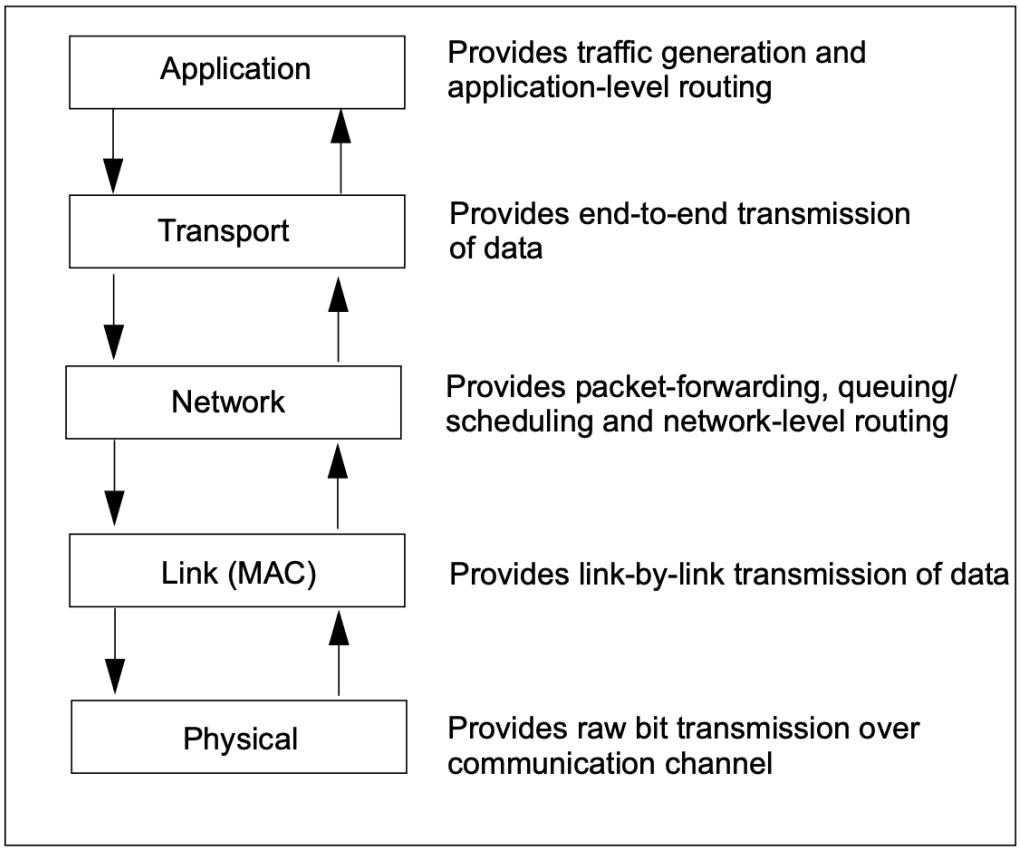


FIGURE 1-1. EXata Protocol Stack

- **Application Layer**

  主要负责产生和消费流量、执行应用层多路由。前者比如CBR、FTP和 Telnet等，后者如 RIP、Bellman-Ford和 BGP等。

  4.2 节描述应用层协议实现细节，以及定制的方法。

- **Transport Layer**

  比如 TCP 和 UDP，4.3 节描述传输层协议实现细节，以及如何扩展。

- **Network Layer**

  1）分组转发、排队和调度，如 IP、FIFO、RED等；2）网络层的路由，如 AODV、OSPF等。

  4.4 节详细描述网络层协议实现细节，以及定制方法。

- **Link（MAC） Layer**

  逐个链路的收发：point-to-point、802.3、802.11，CSMA等。

  4.5 节详细描述 Link（MAC）层协议实现，以及定制方法。

- **Physical Layer**

  从无线或有线信道收发原始比特。注意：有线物理信道与 Link（MAC）协议集成在一起。

  4.6 节详细描述物理层协议实现，及定制方法。【encoding/decoding】

- **Communication Medium**

通信媒介在节点间收发信号，与物理层接口。通信媒介模型包括：Pahtloss model （比如free space， two ray），fading model （Rayleigh fading，Ricean Fading model），and a shadowing model （constant，lognormal）。

4.7 节相信描述该层的实现与定制方式。

- **Node Mobility**

  节点的 Mobility 模型与节点的 Placement 模型以及地形（terrain）模型一起描述节点的一定行为。

  支持的移动模型包括：waypoint，group mobility，pedestrian mobility 和file-based mobility （低轨卫星星座的主要方式）。

  4.8 节详细描述如何添加移动性模型。

2. Exata 的文件组织、编译与调试

a. EXata 的目录结构

TABLE 2-1. Default EXata Subdirectories

Subdirectory	Description
EXATA_HOME/addons	Components developed as custom add-on modules
EXATA_HOME/bin	Executable and other runtime files, such as DLLs
EXATA_HOME/contributed	Files related to models contributed by third parties
EXATA_HOME/data	Data files for the Wireless Model Library, including antenna configurations, modulation schemes, and sample terrain files.
EXATA_HOME/documentation	Documentation (User's Guide, Release Notes, etc.)
EXATA_HOME/gui	Graphical components, including icons, and GUI configuration files
EXATA_HOME/include	EXata kernel header files
EXATA_HOME/installers	Installers for supplemental third party software
EXATA_HOME/interfaces	Code to interface EXata with third party tools or external networks, such as HLA and DIS
EXATA_HOME/kernel	EXata kernel objects used in the build process
EXATA_HOME/lib	Third party software libraries used in the build process
EXATA_HOME/libraries	Source code for models in EXata model libraries, such as Developer, Wireless, and Multimedia & Enterprise.
EXATA_HOME/license_dir	License files and license libraries required for the build process
EXATA_HOME/main	Kernel source files and Makefiles
EXATA_HOME/scenarios	Sample scenarios

b. 如何编译

以Windows操作系统、VS 2022 为例

i. 可执行文件

1. exata-precompiled-32bit.exe: 32位命令行执行文件
2. exata.exe: 是 exata-precompiled-32bit.exe 的副本，不过，每编译一次，exata.exe 将被覆盖一次。
3. EXataGUI.exe：带 GUI 的可执行文件。

4. 如果源代码修改了，就需要重新编译，

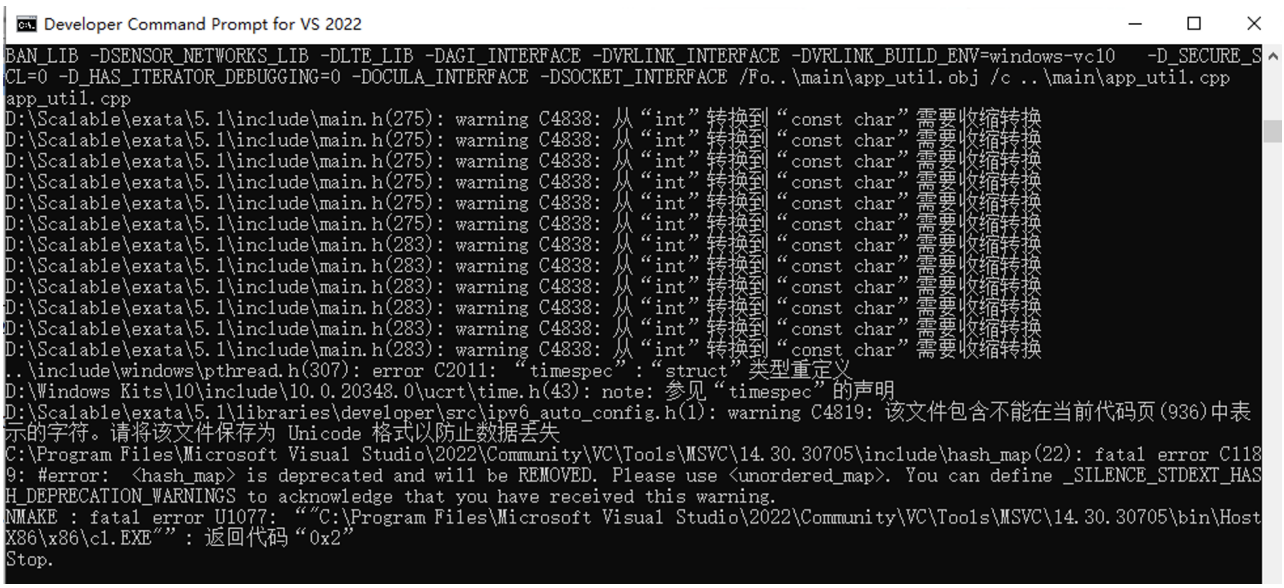
ii. 命令行编译

1. 打开 Developer Command Prompt for VS 2022;
2. 切换到 exata 的main 目录：D:\Scalable\exata\5.1\main>，这个目录下面有多个版本的makefile，包括用于编译的 Makefile，还有适用于不同编译环境多多个版本的Makefile；首先将 Makefile 进行备份，选择合适的makefile，比如 Makefile-windows-vc10 Makefile

TABLE 2-3. Windows Makefiles

Compiler	Makefile for 32-bit Executable (for 32-bit and 64-bit Platforms)
VC9	Makefile-windows-vc9
VC9 Express	Makefile-windows-vc9
VC10	Makefile-windows-vc10
VC10 Express	Makefile-windows-vc10

3. copy Makefile-windows-vc10 Makefile：复制合适版本的makefile作为新的 Makefile
4. 用 nmake 进行编译。
5. 正常情况，应该在bin目录下生成新的 exata.exe.
6. 首次编译出现错误，没有生成新的 exe 文件



7. 换用 vs 2019 运行 nmake 出现同样的错误
8. 切换到 VS2010 命令行，运行nmake，可以正常编译；如果需要完全重新编译，则 nmake clean; nmake.
9. 运行WiFiDemo场景，检验新生成的 exata.exe运行正常。

iii. VS2010 IDE编译

1. 在 ~/5.1/ 目录下，为 VS IDE工程文件写一个makefile：Makefile，主要是重定向到5.1/main/下合适版本的makefile内容如下（注意：缩进采用 TAB，而不是空格）：

Makefile

```
%% Created by Jiangtao Luo
%% for compiling EXata using VS 2010

all:
    cd main
    nmake -f Makefile-windows-vc10

rebuild:
    cd main
    nmake -f Makefile-windows-vc10

clean:
    cd main
    nmake -f Makefile-windows-vc10 clean
```

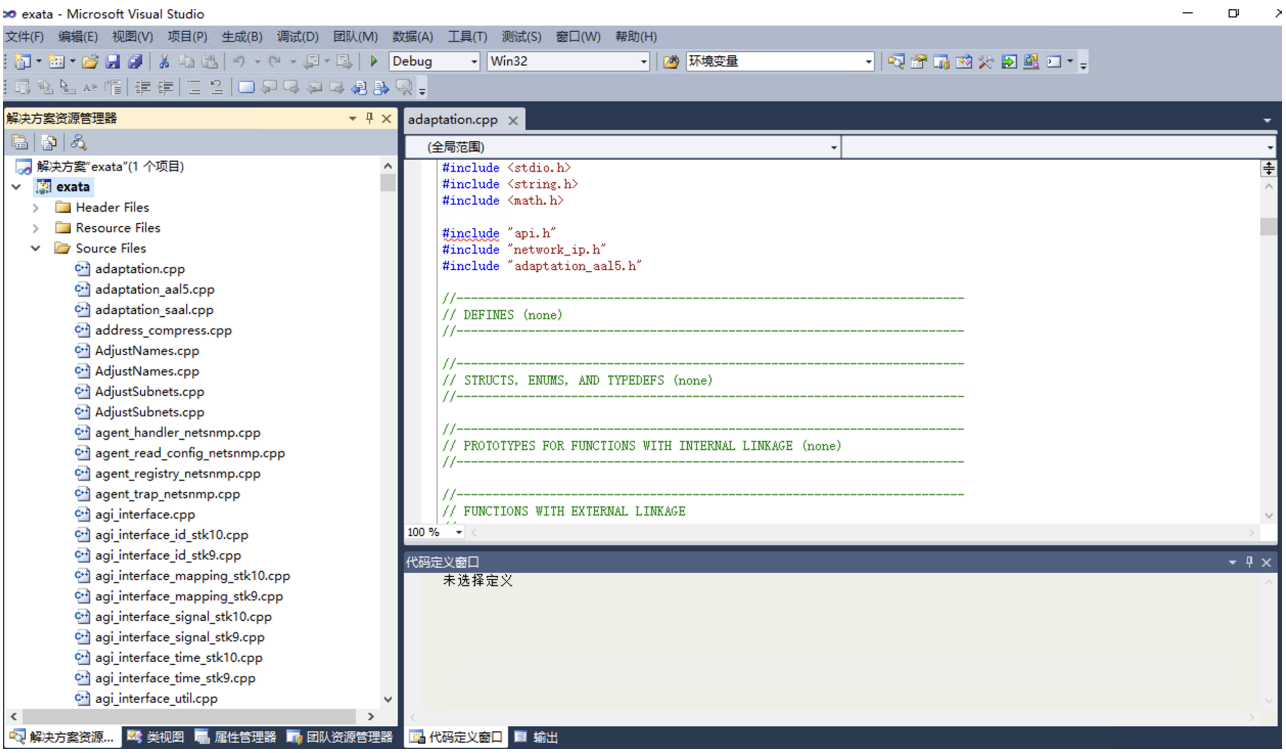
2. 打开 VS 2010，选择基于现有代码新建工程，按本机目录结构配置如下

3. 下一步选择“使用外部生成系统”

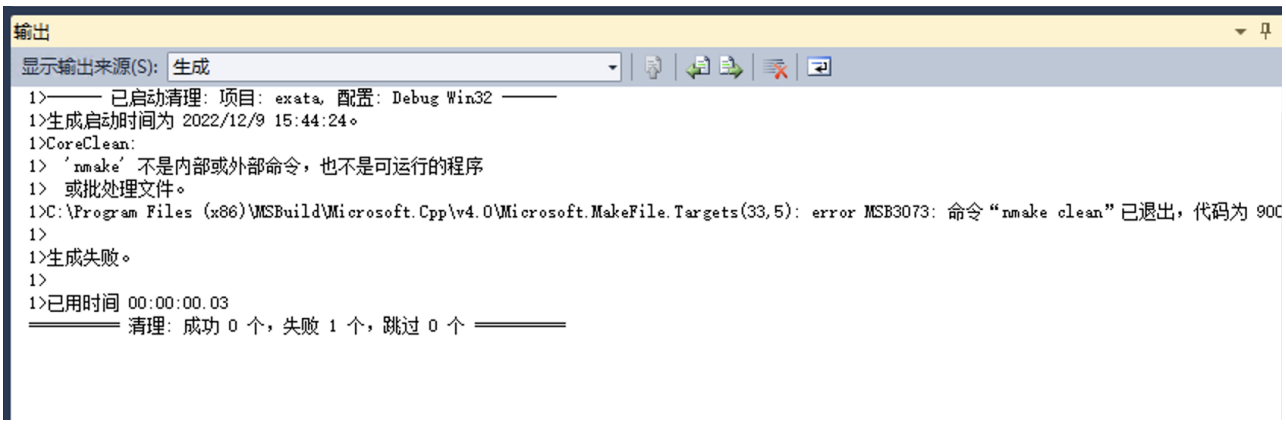

从现有代码文件创建新项目
?
指定项目设置
这些详细信息决定了将如何生成项目以及所创建的项目类型。
您希望如何生成项目?
使用 Visual Studio(V)
项目类型(P):
Windows 应用程序项目
添加对 ATL 的支持(A)
添加对 MFC 的支持(M)
添加对公共语言运行时的支持(C)
公共语言运行时支持:
公共语言运行时支持
使用外部生成系统(E)
要指定生成命令行，请单击"下一步"在"指定调试
配置设置"页和"指定发布配置设置"页上进行设置。
< 上一步(P)
下一步(N) >
完成(F)
取消

4. 配置编译命令行

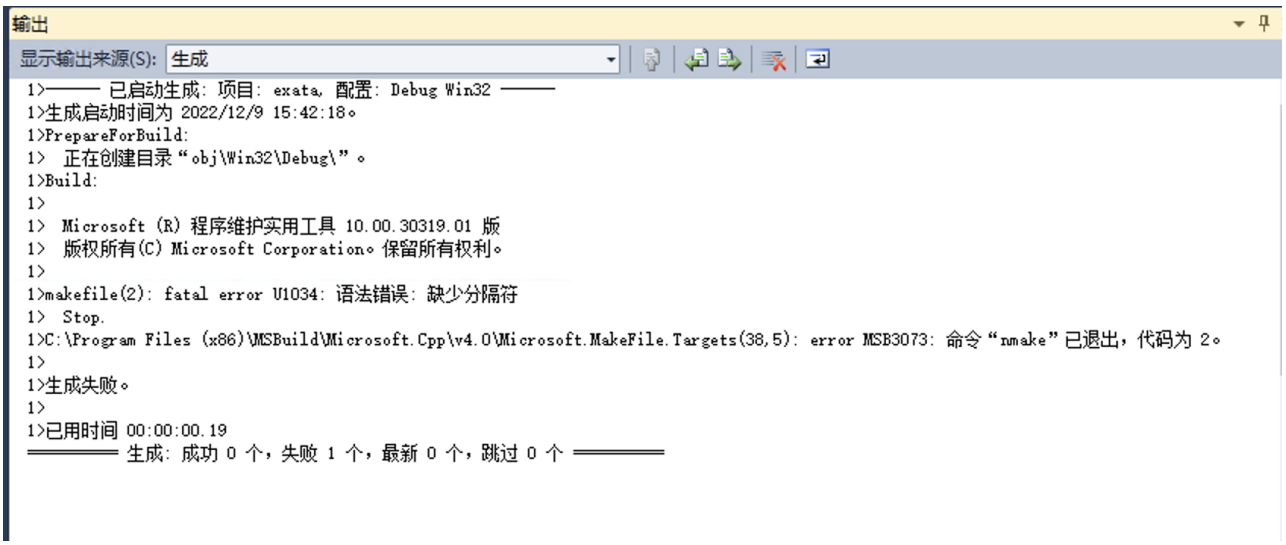
5. 点击“完成”
6. 查看exata 工程



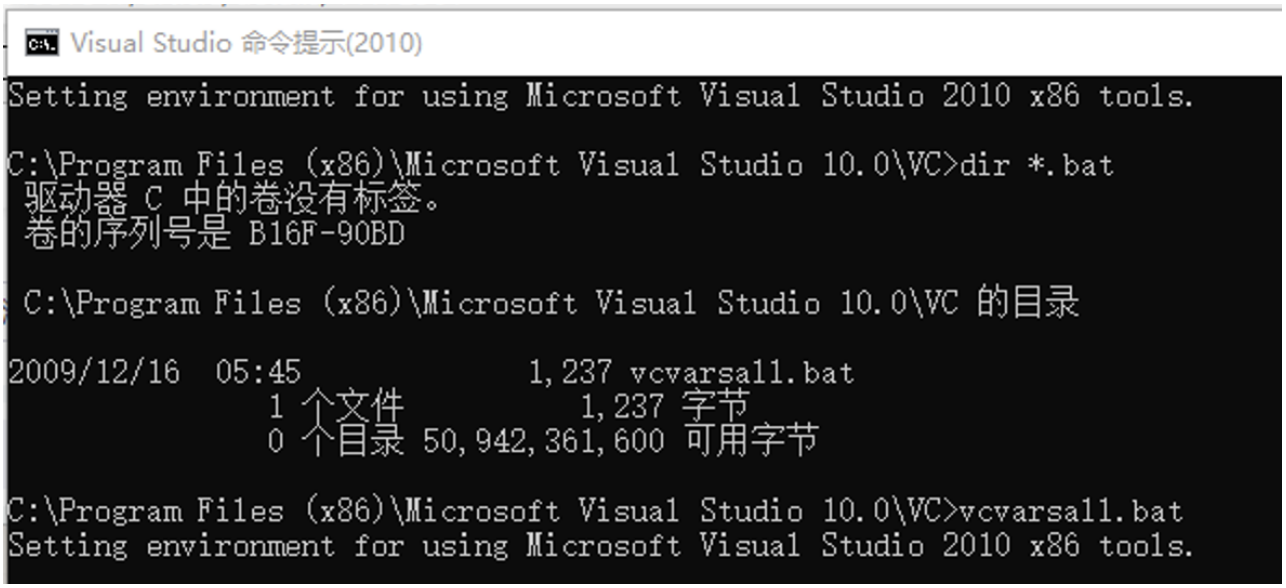
7. Build–》清除或rebuild 出错



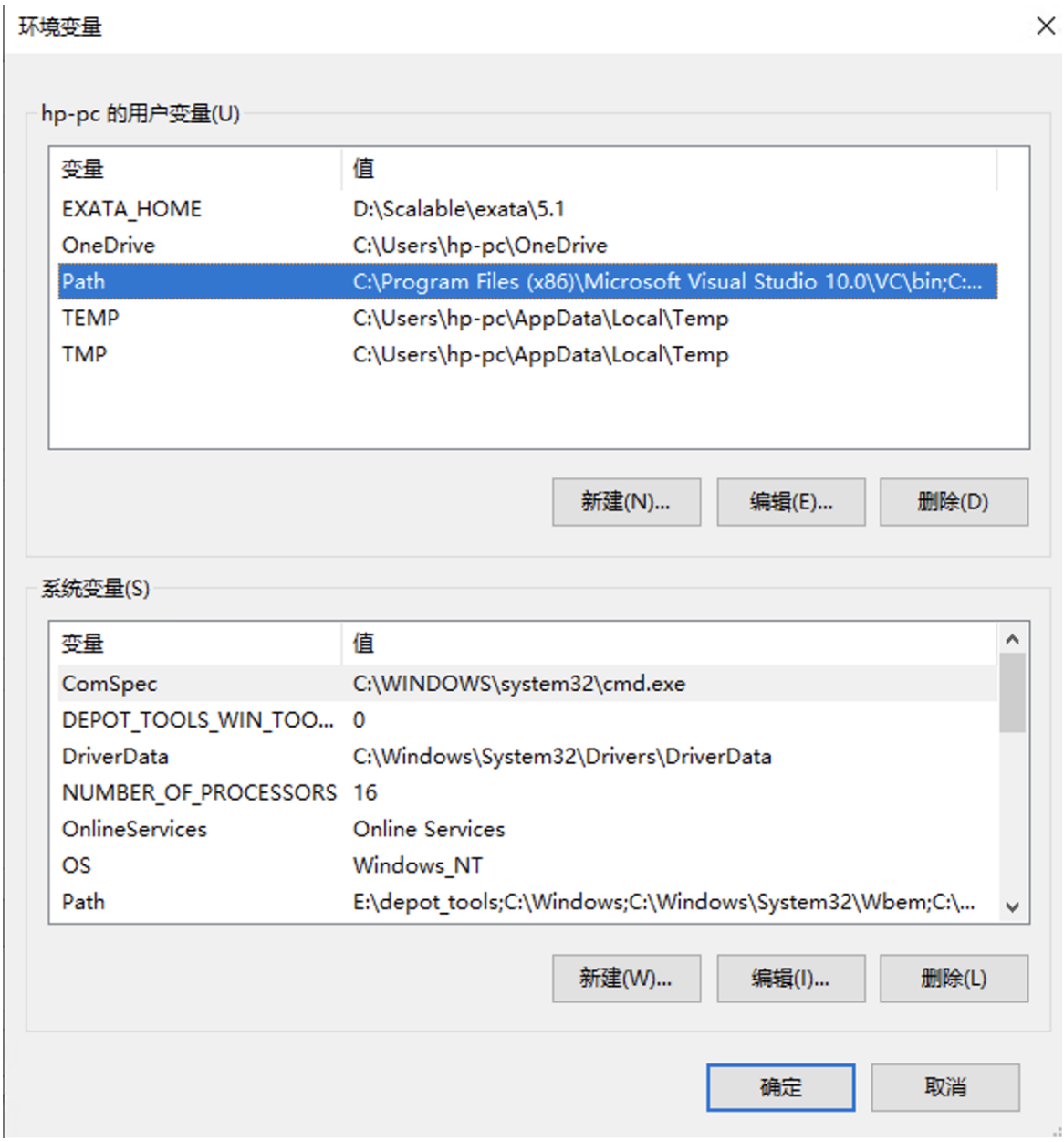
提示无法识别 nmake，或者 Makefile 存在“语法错误，缺少分隔符”！！！ 【？？？】



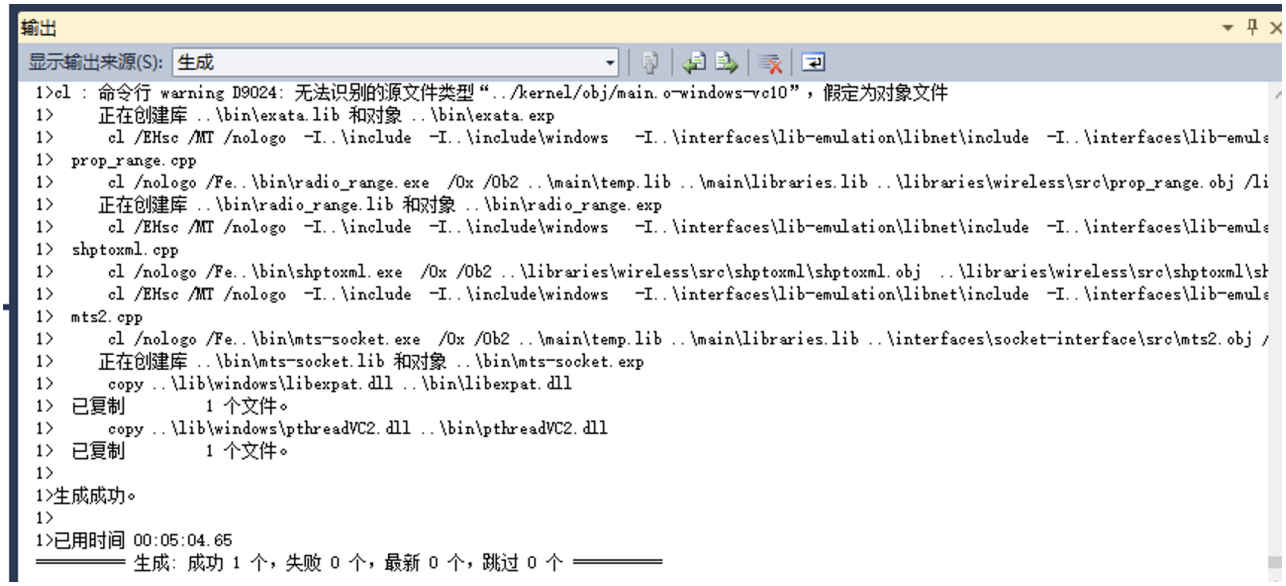
8. 判断应该是未安装 VS 的环境变量，对于 VS2010，可以到 VS 2010 命令行环境下运行批处理文件 vcvarsall.bat，安装环境变量；有时会失效，务必到Windows系统环境变量中检查 nmake 路径是否已添加到位：～VC/bin.



9. 检查 ～VC/bin路径是否已添加



10. nmake路径配置成功，则 exata 即可正常编译成功：



c. 如何激活插件

EXata 利用插（Addons）件来提供增强的特色和功能。插件可能是多种类型：

- **Libraries**：预编译好的库，如和 EXata一起出售的协议模块；
- **Interfaces**：第三方外部接口，STK 等，
- **自定义插件**：特定的插件或用户自己开发的插件；
- **贡献的插件**：用户贡献出来的插件，愿意一同发行的。

总体分为两大类：

i. 发行版中包含的预编译模型库

EXatat基础版包括的预编译库有：

- Developer (including STK interface)
- Multimedia and Enterprise
- Network Management
- Wireless

插件库包括：

- Advanced Wireless
- Cellular
- Cyber
- Federation Interfaces
- LTE
- Sensor Networks
- UMTS
- Urban Propagation

ii. 发行版中未包含的预编译模型库

这些需要下载，如何重新编译和激活。

i. Windows 下如何激活插件

1. 打开插件的配置文件：EXATA_HOME/main/Makefile-addons-windows
2. 确定要激活或去激活的插件的 makefile 文件所在的行，比如要激活 Cellular Model 库，则去掉该行最前面的“#”；反之，则加上“#”。

```
#include ../libraries/cellular/Makefile-windows
```

to

```
include ../libraries/cellular/Makefile-windows
```

3. nmake clean；删除所有的 obj 文件；
4. nmake

ii. Linux 下如何激活

1. 略

d. 编译高级选项

略

e. 如何调试

i. 首先要编译debug 版本

在 EXATA_HOME/main/ 下找到 Makefile，保留 DEBUG，注释掉 OPT，则编译出 Debug 版本；反之，则编译的为 Optimized 版本。记得修改后，务必 首先 nmake clean，然后 nmake。

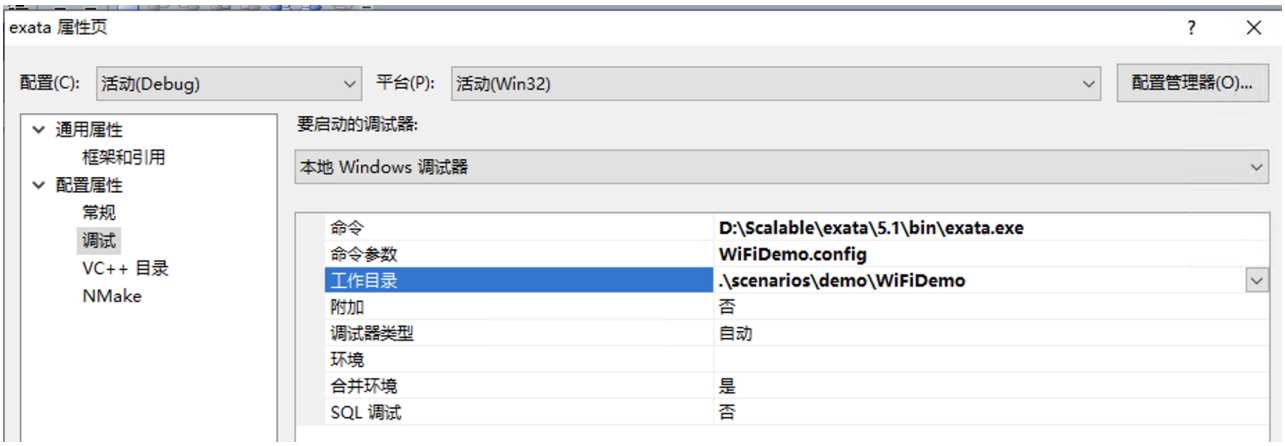
```
##DEBUG = /Zi
OPT = /Ox /Ob2
```

ii. Visual Studio 调试

与其他Windows 程序调试类似，需要在工程属性（Project Properties）中配置 Debug 时的命令行程序和运行参数。这里以调试 WiFiDemo 场景为例，配置如下：

- 命令（Command）：exata.exe
- 命令参数（Command Arguments）：这里填写要调试的场景文件，比如WiFiDemo.config；
- 工作目录（Worling Directory）：这里填写场景文件主目录

工程属性设置完， 即可添加断点运行调试，与其他程序完全相同。



3. Simulator Basics（仿真器基础）

a. Overiew of Discrete-Event Simulation

本节介绍 EXata 仿真器的基础知识，包括离散事件仿真，以及协议如何建模、EXata 中的离散事件仿真如何实现，最后介绍EXata 仿真器的架构。

其核心要领：网络中行为或状态的变化用事件（event）来描述，比如包到达（packet arrival）。仿真器要维护两个重要东西：事件队列（event queue）和模拟时钟（simulation clock）。事件队列中的每个事件设定在特定时间发生，仿真器根据时间顺序执行特点事件来推进模拟时钟，发生过的事件从事件队列中清除。 事件的发生会引起系统状态的变化，设定新的事件加入队列等，模拟时钟以离散的方式逐步推进。

b. Modeling Protocols in EXata

EXata 中的协议作为有限状态机（Finite State Machine）来运行。事件的发生对应协议状态机的状态跃迁。相邻协议之间的接口也基于事件完成。

EXata 中每个协议对应如下图一个 FSM。其核心是一个事件调度器（Event Dispatcher）。它包含一个Wait For Event（事件等待）状态和一个或多个处理事件（Event Handler）状态。每个事件触发进入相应的处理事件状态。

除了Event Dispatcher，还有两个状态：**初始化（Initialization）**和**终结（Finalization）**状态。

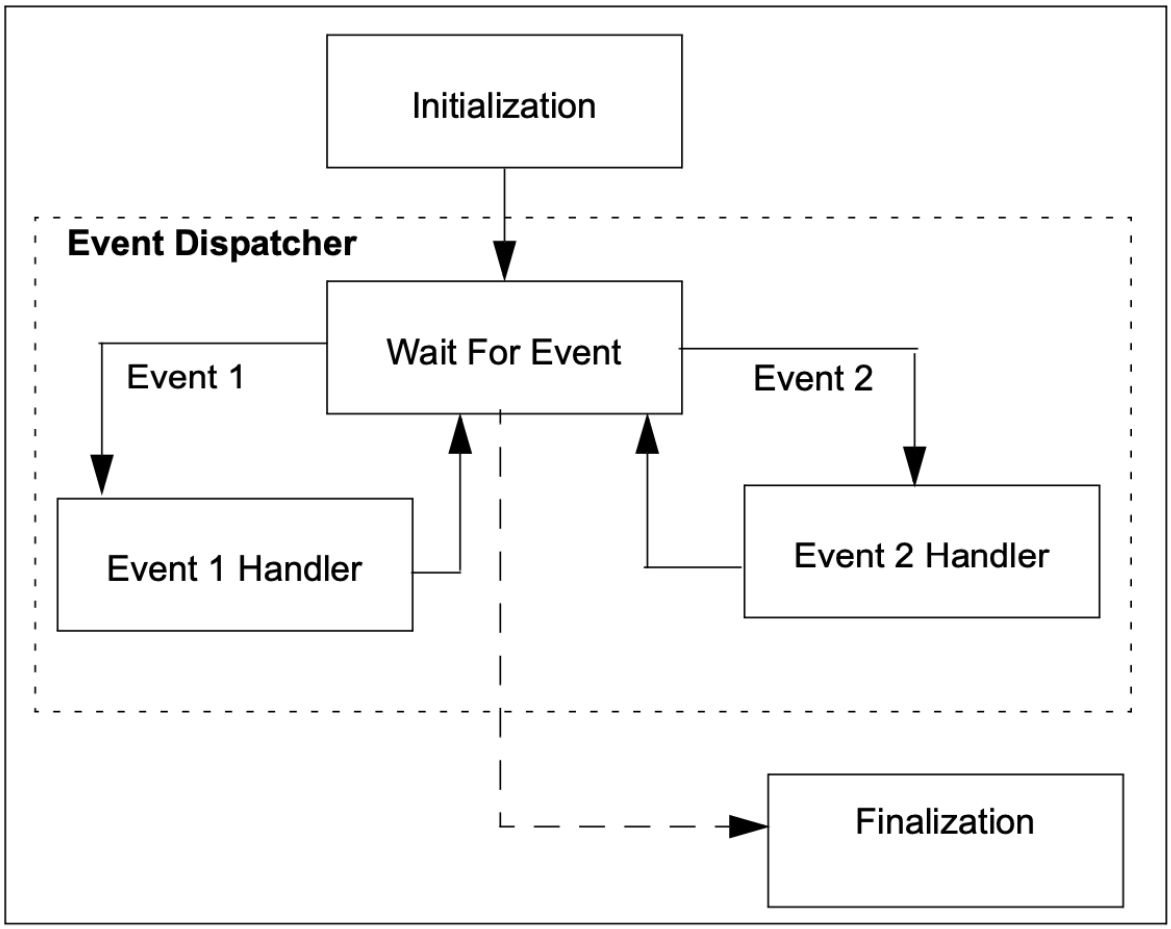


FIGURE 3-1. Protocol Model in EXata

c. Discrete-Event Simulation in EXata

如何实现离散事件仿真。

i. Events and Messages 事件和消息

1. Message Class 消息类

Message 类定义在文件 message.h，用来实现 events。

message.h

(全局范围)

```
// /**
// STRUCT      :: Message
// DESCRIPTION :: This is the main data strucure that represents a discrete
//                event in qualnet. This is used to represent timer as well
//                as to simulate actual sending of packets across the network.
// **/
class Message
{
private:
    static const UInt8 SENT = 0x01; // Message is being sent
    static const UInt8 FREED = 0x02; // MESSAGE_Free has been called
    static const UInt8 DELETED = 0x04; // Deleted using "delete"
    UInt8 m_flags;
public:
    // The default constructor should not be used unless under specific
    // circumstances.  The message is not initialized here.
    Message();

    Message(const Message& m);
    Message(PartitionData *partition,
            int  layerType,
            int  protocol,
            int  eventType,
            bool isMT = false);
    virtual ~Message();
```

2. 比较重要的成员包括：

- **layerType**：layer which will receive this message；
- **protocolType**：Protocol which will receive this message in this layer;
- **instanceId**：which instance will give message to
- **eventType**: Message's event type
- **packetSize** (仅用于真实的包): which indicate the simulated packet size.
- **virtualPayloadSize**: 用户数据部分大大小，不在内存中真实分配，但影响到缓存大小和transmission time的计算
- **packetCreationTime**:仿真真实包时，代表包的生成时间
- **infoArray**: 用于协议层次之间、节点之间传输的信息，用于处理 events，不代表网络中实际数据的传输，不影响 transmission delay 的计算。

3. infoArray 信息数组

- 它是一个MessageInfoHeader 类型的结构体，同样定义在 message.h中

```
typedef struct message_info_header_str
{
    unsigned short infoType; // type of the info field
    unsigned short infoSize; // size of buffer pointed to by "info" variable
    char* info;              // pointer to buffer for holding info
} MessageInfoHeader;
```

FIGURE 3-3. `MessageInfoHeader` Data Structure

- **infoType**：indicates the type of the information contained in this struct. It is an enumeration type. Users can add addtional members to this enumeration, as explained in Section 4.1.2.
- infoSize
- char* info.

4. **char *packet**: 用来放置真实的 packets
5. **Message APIs**
   - defined in message.cpp; refer to *API Reference Guide*
   - MESSAGE_Alloc
   - MESSAGE_Free
   - MESSAGE_AddInfo
   - MESSAGE_ReturnInfo
   - MESSAGE_ReturnInfoSize
   - MESSAGE_PacketAlloc
   - MESSAGE_ReturnPacket
   - MESSAGE_ReturnPacketSize
   - MESSAGE_CancelSelfMsg: Do not to use a cancelled message!
   - MESSAGE_AddHeader: adds a header to the simulated packet.
   - MESSAGE_RemoveHeader
   - MESSAGE_GetLayer
   - MESSAGE_GetProtocol
   - MESSAGE_GetEvent
   -

ii. Types of Events 事件类型

两类事件：packet 和 timer

1. Packet 事件

   Packet events are used to simulate transmission of packets across the network. A packet is defined as a unit of virtual or real data at any layer of the protocol stack.

   An example of a protocol sending a packet to its peer on the other node is shown in Fig. 3–5.

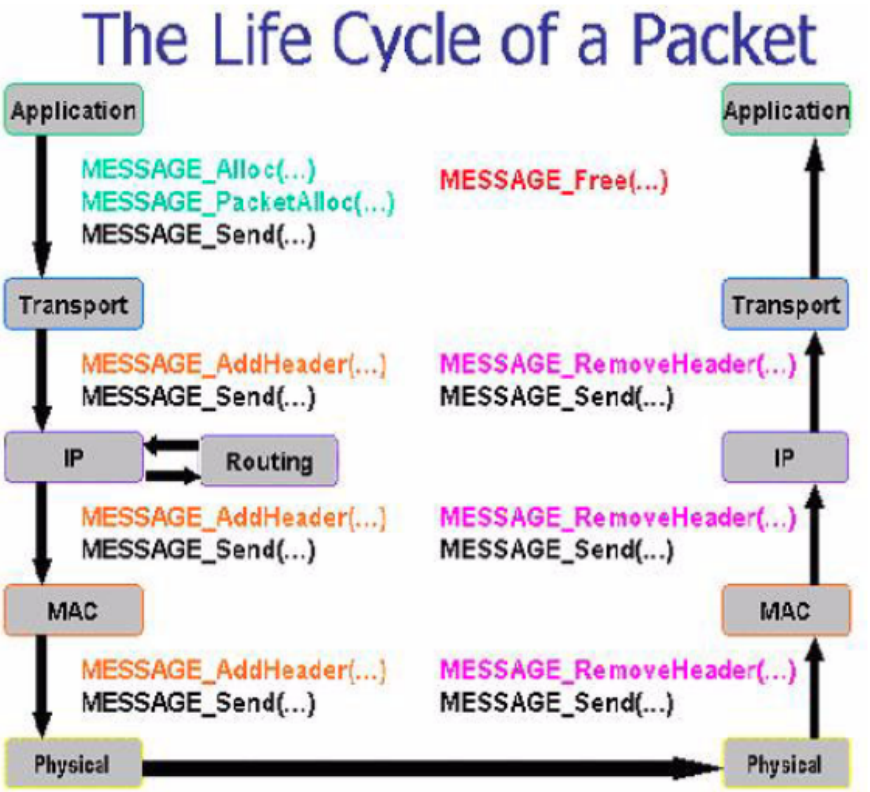


FIGURE 3-5. The Life Cycle of a Packet

- 两类 APIs：message APIs 和 层 API （Layer-specific API）：前者可以在任意协议层次调用；后者只能在特定层调用。
- Sending Packets Using Layer-specific APIs

  相当于将 Message APIs 封装在特定协议层次使用，是一种简化开发的手段。每层协议有哪些 API 可用，在第 4 章相应章节有描述。比如，应用层的包交换有两类 API可以调用，分别对应 Transport Layer 采用 UDP 和 TCP

TABLE 3-1. API Functions for Sending Packets via UDP

API Function	Description
APP_UdpSendNewData	Sends user data via UDP to a destination after a user-specified delay.
APP_UdpSendNewDataWithPriority	Sends user data via UDP to a destination after a user-specified delay with a user-specified priority value.
APP_UdpSendNewHeaderData	Appends an application header to user data and sends it via UDP to a destination after a user-specified delay.
APP_UdpSendNewHeaderDataWithPriority	Appends an application header to user data and sends it via UDP to a destination after a user-specified delay with a user-specified priority
APP_UdpSendNewHeaderVirtualDataWithPriority	Sends a packet composed of an application header containing useful information and user data whose contents are unimportant and serve only to add to resource consumption (queue capacity, transmission delays, etc.), via UDP to a destination after a user-specified delay with a user-specified priority value.

TABLE 3-2. API Functions for Sending Packets via TCP

API Function	Description
APP_TcpSendData	Sends user data via TCP to a destination.
APP_TcpSendNewHeaderVirtualData	Sends a packet composed of an application header containing useful information and user data whose contents are unimportant and serve only to add to resource consumption (queue capacity, transmission delays, etc.), via TCP to a destination.

- Sending Packets Using Message APIs

  参加Fig. 3-5，以及层 API 的实现，直接利用Message API 发送包，主要调用三个 API 函数：MESSAGE_Alloc、MESSAGE_PacketAlloc、MESSAGE_Send.

2. Timer 事件

   a. Setting Timers

   与其他仿真工具不同，EXatat 设定Timer是通过 Timer 类型的消息完成的。比如，RIP 通过下面的代码设定一个 5 秒钟的常规更新 Timer。如果一个 timer 需要额外的信息，则可以通过该消息的 infoArray 携带。比如对于一个 time-out 定时器，它等待一个发出去的包的确认响应，则可以在该 timer message 的 infoArray 中包含序列号和目的 IP 地址。

```
Message *newMsg;
clocktype delay;
newMsg = MESSAGE_Alloc(node,
                       APP_LAYER,
                       APP_ROUTING_RIP,
                       MSG_APP_RIP_RegularUpdateAlarm);
delay = 5* SECOND;
MESSAGE_Send(node, newMsg, delay);
```

   b. Cancelling Timers

   利用 MESSAGE_CancelSelfMsg 来终止一个已启动的 Timer，参数为节点和待终止的消息指针。

   c.

d. EXata Simulator Architecture

EXata 主要由三部分构成：初始化（Initialization）、事件处理(Event Handling)和完结(Finalization)。每部分的功能都以**分级**（hierarchically）的方式执行：首先是**节点级**，然后是**层级**，最后是**协议级**。

i. Initialization Hierarchy 初始化的分级

1. **节点初始化**（对每一个节点执行）：PARTITATION_InitialzeNodes
2. **层初始化**（对每个节点自底而上逐层执行，在节点初始化中调用层初始化函数）
   a. MAC_Initialize
   b. TRANSPORT_Initialize
   c. NETWORK_PreInit、NETWORK_Initialize
   d. APP_Initialize（初始化应用层路由协议）、APP_InitializeApplications（初始化流量生成协议）

3. **协议初始化**（在每个层初始化中调用）
    a. 比如，在TRANSPORT_Initilize中调用 TransportTcpInit 和 TransportUdpInit 分别完成 TCP 和 UDP 协议初始化。

ii. Event Handling hierarchy 事件处理的分级

1. 节点事件处理（EXata Kernel 获取节点 Id）：
    a. NODE_ProcessEvent：首先获取 Layer 信息，然后调用层事件处理函数
2. 层事件处理 ： 进一步根据协议类型，调用相应的协议事件处理函数
    a. APP_ProcessEvent
    b. TRANSPORT_ProcessEvent
    c. NETWORK_ProcessEvent
    d. MAC_ProcessEvent
    e. PHY_ProcessEvent
    f. PROP_ProcessEvent 【? 通用的? 】
3. 协议事件处理：不同层会有多个不同协议的事件处理函数，比如应用层，可能有应用层路由或者应用的事件处理

```
void APP_ProcessEvent(Node *node, Message *msg)
{
    short protocolType;
    protocolType = APP_GetProtocolType(node,msg);

    switch(protocolType)
    {
        case APP_ROUTING_BELLMANFORD:
        {
            RoutingBellmanfordLayer(node, msg);
            break;
        }
        case APP_ROUTING_FISHEYE:
        {
            RoutingFisheyeLayer(node,msg);
            break;
        }
        ..
        case APP_FTP_CLIENT:
        {
            AppLayerFtpClient(node, msg);
            break;
        }
        case APP_FTP_SERVER:
        {
            AppLayerFtpServer(node, msg);
            break;
        }
       ...
    }//switch//
}
```

FIGURE 3-13. Layer Event Dispatcher Function

4. 事件处理（协议事件处理内部）：根据消息的 eventType 调用不同的event handler 函数进行处理

iii. Finalization Hierarchy 终止化的分级

仿真结束时，每个协议需要打印各自的统计量，也需要分级进行终止操作。

1. 节点终止化：PARTITION_Finalize，其中逐个节点进行层终止化；

2. 层终止化：在节点终止化中按节点逐个调用，可以根据需要定义终止化操作，其中调用相应协议终止化操作
    a. PHY_Finalize
    b. MAC_Finalize
    c. NETWORK_Finalize
    d. TRANSPORT_Finalize
    e. APP_Finalize
    f. USER_Finalize
    g. MOBILITY_Finalize
    h. ... ...
3. 协议终止化：层终止化函数提取协议类型，调用各协议终止化函数

【本章完】