OMNeT++中文手册

1.1 OMNeT++为何物？
OMNeT++是一款面向对象的离散事件网络模拟器，可以实现的功能如下：

.无线电通信网络信道模拟

- 协议模拟
- 模拟队列网络
- 模拟多处理器和其他分布式硬件系统
- 确认硬件结构
- 测定复杂软件系统多方面的性能
- 模拟其他的任何一种合适的离散事件系统

一款 OMNeT++模拟器包括一些分层次的嵌入式模型,嵌入式模型的深度是无限的,即允许用户在模拟环境中绘制实际系统的逻辑结构.各模块通过信息的传输进 行通信,其信息可以包括任意复杂的数据结构,各模块均可以通过门或者线路直接发送信息给目标点或者也可以通过预先的路径进行传输.

各个模块可以有自己的参数集,参数集可以被用于定制模块行为,或者可以用于确定模拟拓扑图的参数.

模拟网络最底层的模块可以嵌入行为,这些模块被称为基本模块,它可以利用模拟器的库函数在 C++进行编程.

OMNeT ++模拟器可以在根据不同的目的来改变用户接口： 调试、实例和批量执行。高级用户的接口可以把模块透明的交给用户,即允许控制模拟器执行以及可以通过改变模块中的变量/对象来干涉模拟器的执行， 这在开发 /调试模拟器工程师非常有用的， 用户接口也促进了模块工作的实现。

模拟器的接口和工具都非常轻便： 目前得知它可以在 Windows 和各种 UNIX 操作系统下利用 c++进行编译。

OMNeT ++还支持分布式并行仿真， OMNeT++可以利用多种机制来进行用于几个并联的分布式模拟器之间的通信仿真， 比如 MPI 和指定的通道。这种并行仿真算法 可以很容易的进行扩展，也很容易加入新的模块。各个模块不必须要特定的结构来并行运行，这只是一个配置的问题。OMNeT++甚至还可以被用于并行模拟仿 真算法的多层次描述，因为模拟器可以在 GUI 下并行运行， 这种 GUI 为运行过程提供了详细的反馈。

OMNEST 是 OMNeT++的一个商业版本， OMNeT++只在学术和非盈利性活动免费， 在进行商业性研究时需要从 Global 公司获得 OMNEST 许可证。

1.2 本手册的组织结构

本手册的组织结构如下:

第[1],[2]章包括介绍性的资料

第二组章节,[3],[4],和[6]是编程向导.他们提出了 NED 语言,仿真的概念和他们在 OMNet++中的执行,解释了如何写一个简单的模块并描述了类库.

第[9],[11]进一步阐述了主题,通过解释如何定制网络图,从产生的方件中,如何写 NED 源代码注释.

[7],[8],[10]处理了实际的问题,比如建立,运行仿真器,分析结果,提出了 OMNet++工具提供的所支持的任务.

[12]章支持分布式执行

最后[13]解释了 OMNet++内部结构

附录[14]提供了参考的 NED 语言

第二章 概述

2.1 建模的概念

OMNeT++为用户提供了有效的用于描述实际系统结构的工具。一些主要的特征表现如下：

(1) 分层次嵌入式模块

(2) 各模块以模块类型分类

(3) 模块之间通过信号在通道上 的传输进行通信

(4) 灵活的模块参数

(5)拓扑描述语言

2.1.1 分层次的各模块

OMNeT ++模块包括分层次的嵌入式模块，这些模块通过彼此之间传输消息来进行通信。OMNeT++经常被描述成网络结构，最顶层的模块称为系统模块，系统模块包括子模块，其子模块还可以包括本身的子模块，模块嵌入的深度是没有限制的，它允许用户在模块结构中根据实际系统来绘制逻辑结构图。

模块结构利用 OMNeT++ 的 NED 语言进行描述。

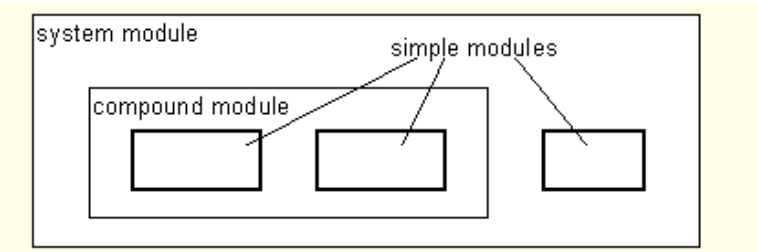


Figure: Simple and compound modules

包含子模块的模块称为混合模块,与在层次模块最底层的简单模块相反.在模型中简单模块包括算法.使用 OMNet++的仿真类库, 用户通过 C++执行简单模块.

2.1.2 模块类型

基本模块和复合模块都是模块类型的实例。在描述模块时,用户定义了模块类型;这些模块类型的实例用于组成更复杂的模块类型.最终,用户创建系统模块为前面所定义的模块类型的实例;所有的网络模块都被实例为系统模块的子模块和子子模块.

当一种模块类型被用作一个建立块，则不管是基本模块和复合模块都没有区别。,这使用户在不影响现有的模块类型用户的条件下,可以将一个基本模块分割成多个基本模块嵌入至一个复合模块,或者相反,集成一个复合模块的功能为单个基本模块.

模块类型可以存储于文件中，并且可以保证与它实际的用法分别开来，这就意味着用户可以通过存在的模块类型进行分组，也可以创造组成库，这一特征在后面第[8]章将会给出详细的介绍。

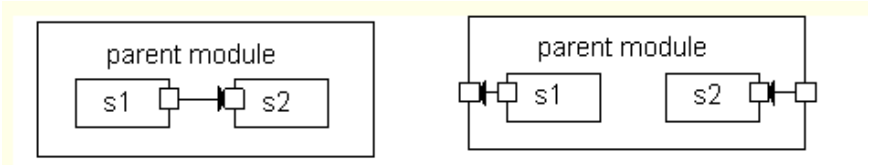
2.1.3 消息、门、链路

模块之间通过交换消息进行通信，在一个实际的模拟器中，可以使用计算机网络中的帧和包来替代消息,在队列网络中可以用作业或消费者来替代消息,或者其他的移动实体类型。消息可以包括任意复杂的数据结构.基本模块可以通过门或连接,直接发消息至目的地,也可以通过预先确定的路径发送消息.

当模块接收一个消息时,模块的”本地仿真时间”前进.消息能够从其他的模块或从相同的模块抵达(自身的消息用于执行定时器).

门是模块的输入/输出接口，消息通过输出门发送出去，通过输入门进行接收。

每个连接(也称之为链接)被创建成一个单一层次的模块层次:在一个复合模块中,可以连接相应的两个子模块的门,或一个子模块的门和一个复合模块的门.



子模块彼此连接　　子模块连接父模块

由于模块的层次结构,典型的消息传输是通过一系列的连接,开始和到达都在简单模块中.这些连接系列从简单模块到简单模块,被称之为路由.在模块中的复合模块可以看成”纸盒”,在其内部和外部世界之间透明地转播消息.

2.1.4 包输出的建模

连接被分配三个参数,用于方便通信网络的建模,但是在其他的建模中也是有用的:传播延迟,比特错误率和数据率,所有三个都是可选的.对每个连接都可以分别指定链接参数,或者定义链接类型,在整个网络中使用.

传播延迟是指由于通过通道传输,消息抵达的延迟的时间数.

位错误率指一比特数据被错误传输的概率,允许简单的噪音通过建模.

数据率 bit/second,用于计算传输一个包的时间.

当数据率在使用的时候,模块中发送的消息对就于传输的第一个比特,消息接收对应于接收的最后一个比特.这个模块不是总是可用的,例如,类似于 Token 环和 FDDI 协议,不等待结构到达其实体,而是开始重复它的第一个比特,然后他们到达—换句话说,”流量通过”结构,仅存在很少的延迟.如果你想模块化这些网络,OMNet++的数据率建模特征将不能使用.

2.1.5 参数表

模块可以有参数表，参数表可以在 NED 文件中指定，也可以在 omnetpp.ini 中进行配置。

参数可以用于定制简单模块行为,也可以参数化模型拓扑.

参数可以是 string, numeric 或 boolean 值,或者也可以包括 XML 数据等.numeric 值包含使用其他参数的表达式以及调用 C 函数,不同分类的随机变量,和由用户交互输入的值.

Numeric 值的参数可以以灵活的方式构成拓扑结构.在一个复合模块中,其参数定义子模块数,门数,和形成内部连接的方法.

2.1.6 拓扑描述方法

用户使用 NED 描述语言定义了模型的结构.NED 语言将在第[3]章讨论.

2.2　设计算法

一个模型的简单模块包括像 C++函数的算法.使用设计语言的灵活性和能力,支持 OMNet++的仿真类库.仿真程序员可以选择事件驱动或进程式的描述,可以自由使用面向对象概念(继承,多态等)和设计模式来扩展仿真功能.

仿真对象(消息,模块,队列等)由 C++类表示.他们被设计成有效地共同工作,创建一个有力的仿真设计结构.以下的类是仿真类库中的一部分:

modules, gates, connections etc.

parameters

messages

container classes (e.g. queue, array)

data collection classes

statistic and distribution estimation classes (histograms, P^2 algorithm for calculating quantiles etc.)

transient detection and result accuracy detection classes

这些类是一个特殊的工具,允许运行的仿真对象的移动,显示他们的信息如,名称,类名,状态变量或内容.这个特点使他可能创建一个仿真 GUI,其中所有的仿真内在都是可见的.

2.3 使用 OMNeT++

2.3.1 新建运行模拟器

这节提供了在实践中观察 OMNet++的工作:比如讨论了模型文件,编译,运行仿真器等问题.

一个 OMNet++模型包括以下几部分:

NED 语言拓扑描述(.ned 文件),其使用参数,门等描述了模块结构.NED 文件可以使用任何文本编辑器或 GNED 图形化编辑器来编写.

消息定义(.msg 文件).可以定义变量消息类型,以及在其上添加数据文件.OMNet++将消息定义转化成完全的 C++类.

简单模块源.他们是 C++文件,.h 或.cc 后缀.

仿真系统提供了以下的组件:

仿真内核.这包含用 C++编写的管理仿真和仿真类库的代码,编译使其形成一个库文件(扩展名为.a 或.lib).

用户接口.OMNet++用户接口在仿真执行的时候使用,用于方便调试,演示或者批处理仿真的执行.有许多用 C++编写的用户接口, 编译使其形成一个库文件(扩展名为.a 或.lib).

从以上的组件中创建仿真程序.首先,使用 opp_msgc.程序将.msg 文件转化成 C++代码.然后编译所有的 C++源文件,链接仿真内核和用户的接口库,形成一个仿真可执行文件.NED 文件可以转化成 C++文件(使用 nedtool)进行链接,当仿真程序开始执行时,也可在他们原始的文本里动态加载.

仿真器的运行和结果分析

仿真执行文件是一个单独的程序,因此它可以运行在没有 OMNet++,或正在显示模型文件的其他机器上.当程序开始执行,它读一个配置文件(通常为 omnetpp.ini)这个文件包括设置,它控制了仿真如何被执行,模型参数的值,等.配置文件也指定了许多仿真运行;在最简单的情况下,他们将被仿真程序接连地执行.

仿真的输出写入一个数据文件:输出向量文件,输出标量文件,以及用户自己的输出文件.OMNet++提供一个GUI工具Plove来查看,制出输出向量文件的内容图.它不希望仅仅使用OMNet++来处理结果文件:输出文件的格式是一个文本文件,可以读进数学包像 Matlab 或 Octave,或导入电子数据表像 OpenOffice Calc, Gnumeric 或MS Excel(许多预处理将需要sed,awk,perl,这将在后面讨论).所有这些外部的程序提供了丰富的功能用于统计分析和可视化,OMNet++范围之外的程序使他们的成就加倍.本手册简单描述了许多数据测绘程序,以及如何使用 OMNet++.

输出标量文件使用标量工具可视化.它可以画出柱形统计图表,x-y图表(比如吞吐量VS提供的负载),或导出数据通过剪贴板至电子数据表和其他的程序进行更详细的分析.

用户接口

用户接口的基本目的是使模型的内部对用户可视,控制仿真执行,通过改变模型内部的变量/对象允许用户干涉.这在项目仿真的开发/调试阶段非常重要的.一个传递下去的经验允许用户得到一个模型行为的”感觉”也同样重要.图形用户接口可以用于证明一个模型的操作.

相同的仿真模型在模型文件本身不做任何改变的情况下被不同的用户执行.用户可以使用一个有力的图形化用户接口进行测试调试仿真,最终使用一个简单快速的支持批处理执行的用户接口运行.

组件库

存储在文件的模块类型从他们实际使用的地方分离出来.这个使用用户组合现有的模块类型,创建组件库.

通用的单独仿真程序

仿真执行文件可以存储许多独立的模型,使用用相同的简单模块集.用户可以在配置文件中指定运行哪个模型.允许创建一个包括许多仿真模型的大的可执行文件,发布为一个单独的传感器工具.拓扑描述语言的灵活性也支持这种方法.

2.3.2 各分类的内容

如果安装了发布的源程序,你系统上的 omnetpp 目录将包括以下的子目录.(如果你安装了一个预处理发布,将缺少一些目录,或者会有额外的目录,比如 包括OMNet++绑定的软件).

仿真系统本身:

omnetpp/		OMNeT++ 根目录
	bin/	OMNeT++ 可执行文件目录(GNED, nedtool 等)
	include/	仿真模块的头文件
	lib/	库文件
	bitmaps/	图形会网络中使用的图标
	doc/	手册(PDF),readme, license 等
	manual/	HTML 帮助文件
	tictoc-tutorial/	介绍使用 OMNeT++
	api/	参考的 HTML API
	nedxml-api/	API 参考 NEDXML 库
	src/	文件源

src/		OMNeT++ 源
	nedc/	nedtool,消息编译器
	sim/	仿真内核
	parsim/	发布执行的文件
	netbuilder/	动态读取 NED 文件的文件
	envir/	用户接口的公共代码
	cmdenv/	用户接口命令行
	tkenv/	基于 Tcl/Tk 的用户接口
	gned/	图形化 NED 编辑器
	plove/	输出向量分析器和制图工具
	scalars	输出标量分析器和制图工具
	nedxml/	NEDXML 库
	utils/	makefile 创建器,文档工具等
test/		回归测试
	core/	仿真库的回归测试
	distrib/	创建发布的回归测试

...

在 samples 目录中的是仿真例子

samples/		仿真例子的目录
	aloha/	Aloha 协议模型
	cqn/	关闭的队列网络

...

contrib 目录包括 OMNeT++的贡献内容.

contrib/		贡献内容目录
	octave/	用于结果处理的 Octave 脚本
	emacs/	Emacs 高亮显示 NED 语法

你也会发现一些附加的目录,像 msvc/, 其包括 VC++的综合组件等.

3 NED 语言

3.1 NED 概述

模型的拓扑结构可以使用 NED 语言详细描述.NED 语言方便了一个网络的模块描述.这意味着一个网络描述包括许多组件描述(通道,简单/复合模块类型).网络描述的通道,简单模块和复合模块可以在另一个网络描述中重复使用.

包含网络拓扑模型描述的文件通常以.ned 为后缀名,它可以动态地载入仿真程序或由 NED 编译器翻译为 C++代码,并链接到可执行文件中.

EBNF 语言描述见附录[14].

3.1.1 一个 NED 描述组件

一个 NED 描述包括以下组件,以任意数据或顺序:

导入命令

定义信道

简单和复合模块定义

网络定义

3.1.2 保留字

网络描述必须注意不能使用保留字命名.NED 语言的保留字有:

Import, channel, endchannel, simple, endsimple, module, endmodule, error, delay, datarate, const, parameters, gates, submodules, connections, gatesizes, if, for, do, endfor, network, endnetwork, nocheck, ref, ancestor, true, false, like, input, numeric, string, bool, char, xml, xmldoc.

3.1.3 标识符

标识符是模块名,信道,网络,子模块,参数,网关,信道属性和函数.

标识符必须由英文字母表(a-z,A-Z),数字(0-9)和下划线”_”.可以由字母或下划线开始.如果你想以数据开头的话,在前面加个下划线,例如_3Com.

如果标识符由几个单词组成时,按惯例大写每个单词的首字母,建议大写模块,信道,网络等标识符的首字母,小写参数,门,子模块等标识符的首字母.下划线很少使用.

3.1.4 大小写敏感

网络描述和所有的标识符是大小写敏感的.例如,TCP 和 Tcp 是两个不同的命名.

3.1.5 注释

注释可以放在 NED 文件的任何地方,跟 C++语法类似:由双斜线开始”//”,一直延续到这行的结尾,注释被 NED 编译器忽略.

NED 注释可以用于产生文档,像 JavaDoc 和 Doxygen.此特性在第[11]章描述.

3.2 导入命令

导入命令是用于从其他网络描述文件中导入描述.在导入一个网络描述后,可以使用组件(信道,简单/复合模块类型定义.

一个文件被导入,仅仅是其声明被使用,当父文件被编译时,被导入的文件并不会被编译,即必须编译和链接每个网络描述文件,而不仅仅是最上一层的文件.

可以指定有无.ned 扩展名的文件名.也能在文件中中包括路程,或使用 NED 编译器的命令行选项-I <path>指定被导入的文件.

例如: import "ethernet"; // imports ethernet.ned

3.3 信道定义

信道定义是详细说明链接类型的特征(属性). 信道名可用于后面的 NED 描述来创建这些参数的连接.语法:

```
channel ChannelName
    //...
endchannel
```

在信道描述中,三个可选属性可以被赋值:延迟,比特错误率,和数据速率.延迟是传播延迟,以仿真秒为单位.比特错误率是比特数据传输时发生错误的概率;数据速率是指信息的带宽(比特/秒),用于计算数据包的传输时间,三个属性可以以任何顺序出现,所赋值应为常数.

例如:

```
channel LeasedLine
    delay 0.0018  // sec
    error 1e-8
    datarate 128000  // bit/sec
endchannel
```

3.4 简单模块定义

简单模块是其他(复合)模块的基本构建块.简单模块类型由名称标识.惯例是,模块名以大定字母开头.

简单模块通过声明参数和门来定义.简单模块由发下语法来声明:

```
simple SimpleModuleName
    parameters:
        //...
    gates:
        //...
endsimple
```

3.4.1 简单模块参数

参数是属于模块的变量.简单模块参数可以被简单模块算法使用.例如,TrafficGen 模块可能有参数 numOfMessages,该参数决定多少消息将被产生.

参数由名称标识.按惯例,参数名以小写字母开头.

Parameters 域列出其名字即可声明参数.参数类型可以被指定为: numeric, numeric const (或 simply const), bool, string, 或 xml, numeric 为缺省类型.

例如:

```
simple TrafficGen
    parameters:
        interarrivalTime,
        numOfMessages : const,
        address : string;
```

```
    gates: //...
endsimple
```

参数可以由 NED 指定(当模块用于大的复合模块的构建块时)或从配置文件 omnetpp.ini,配置文件在第[8]章详细描述.

随机参数和常数 truncnormal(2,0.8),当每次从简单模块(C++代码)读参数时,剪切 2.0 正态分布和标准差为 0.8, 返回一个新随机数.例如,这用于指定产生包或作业的间隔时间.

Numeric 参数可以被设置从统一的分布或各种不同的分布返回随机数.例如,设置一个参数为

如果想初始化参数值来被随机选择,但在后面并不改变.可以通过声明其为常量.常量参数仅在仿真开始时计算,然后设置为一个常数值.

推荐标记每一个参数为常量,除非想要使用随机数的特性.

XML 参数

许多模块需要描述比简单模块参数更复杂的输入.然后输入这些参数至一个外部配置文件,然后让模块读文件,处理文件.在一串参数中通过文件到达模块.

最近 XML 越来越成为一种配置文件的标准格式,可以在 XML 中描述配置.从 3.0 版的 OMNet++就包含了支持 XML 配置文件.

OMNet++包括了 XML 解释器(LibXML, Expat,等),读取和验证文件类型定义的文件(如果 XML 文档包括一个 DOCTYPE),缓存文件(如果多个模块引用,只加载一次),通过一个 XPath 子集符号来选择文档的一部分,在类 DOM 的对象树中显示内容.

这种机制可以通过 NED 参数类型 xml 和 xmldoc()操作访问.可以通过 xmldoc()操作指定 xml 类型模块参数至一个具体的 XML 文件(或 XML 内的一个元素).可以从 NED 和 omnetpp.ini 指定 xml 参数.

3.4.2 简单模块门

门是模块的链接点.模块间起点和终点就是门.OMNet++支持单向链接,因此有输入和输出门.消息通过输出门发送,输入门接收.

门由名称标识,按惯例,门名以小写字母开头.

支持门向量:一个门向量包含多个单一门.

模块描述的 gates 域:列出其名字即可声明门.空的方括号对[]表示门向量.向量的元素从 0 开始编号.

例:

```
simple NetworkInterface
    parameters: //...
    gates:
        in:  fromPort, fromHigherLayer;
        out: toPort, toHigherLayer;
endsimple
```

```
simple RoutingUnit
    parameters: //...
    gates:
        in:  output[];
        out: input[];
endsimple
```

门向量的大小在被用作复合模块的部件时给定.因此,每个模块实例的门向量大小不同.

3.5 复合模块定义

复合模块由一个或多个子模块组成.任何模块类型(简单或复合模块)都可被用作子模块.复合模块的定义类似简单模块,也有 gates 和 parameters 域,可用于任何使用简单模块的地方.

可以把复合模块看成是”纸板盒”,帮助你组织仿真模型,产生结构.没有活动行为与复合模块相联—他们简单地组合模块成更大的部件,可用于作为一个模型或其他复合模块的部件.

按惯例,模块类型名(也包括复合模块类型名)以大写字母开头.

子模块可以使用复合模块的参数.他们彼此相连或与复合模块本身相联.

复合模块的定义类似于简单模块的定义:有 gates 域和 parameters 域,它还有两附加的域 submodules 和 connections.

复合模块的语法如下:

```
module CompoundModule
    parameters:
        //...
    gates:
        //...
    submodules:
        //...
    connections:
        //...
endmodule
```

所有的域(parameters, gates, submodules, connections) 都是可选的.

3.5.1 复合模块和门

复合模块的参数和门与 3.4.1 与 3.4.2 中所描述的简单模块的参数和门一样定义使用.

通常复合模块参数是用于传递给子模块,对子模块的参数初始化的.

参数也可以用于描述复合模块的内部结构,子模块的数目,门向量的林小可借助于复合模块的参数来指定,参数也用于定义复合模块的内部连接.例如,Router 复合模块有若干端口,端口数由参数 numOfPorts 指定.

影响复合模块内部结构的参数必须声明为 const,保证每次访问该参数都返回相同值.否则,如果参数分配随机值,在构建复合模块的内部时每次访问参数都是不同的值,这绝对不是所表达的意思.

例:

```
module Router
    parameters:
        packetsPerSecond : numeric,
        bufferSize : numeric,
        numOfPorts : const;
    gates:
        in: inputPort[];
        out: outputPort[];
    submodules: //...
    connections: //...
endmodule
```

3.5.2 子模块

在复合模块声明的 submodules:域定义子模块. 标识子模块的名称通常以小写字母开头.

子模块是模块类型(简单/复合,之间没有区别)的实体.子模块类型对 NED 编译器必须是可知的,即模块类型必须在该 NED 文件中定义过或者从其他文件中导入.

可以定义子模块向量,其大小可由某个参数值决定.

当定义子模块时,可以为其参数赋值,如果相应的模块类型有门向量,则必须指定其大小.

例:

```
module CompoundModule
    //...
    submodules:
        submodule1: ModuleType1
            parameters:
                //...
            gatesizes:
                //...
        submodule2: ModuleType2
            parameters:
                //...
            gatesizes:
```

```
            //...
endmodule
```

模块向量

可以创建一个子模块(一个模块向量)数组.这是通过模块类型名后面的中括号这间的表达式完成的.这个表达式可以引用模块参数.允许模块数为 0.

例

```
module CompoundModule
    parameters:
        size: const;
    submodules:
        submod1: Node[3]
            //...
        submod2: Node[size]
            //...
        submod3: Node[2*size+1]
            //...
endmodule
```

3.5.3 作为参数的子模块类型

有时将子模块类型作为参数非常方便,因此可以很方便地插入任何模块.

例如,假定仿真学习的目的是比较不同的路由算法,将参与比较的路由算法设计为简单模块:DistVecRoutingNode, AntNetRouting1Node, AntNetRouting2Node 等,同时还创建了为复合模块的网络拓扑 RoutingTestNetwork,这些将用来测试路由算法.目前 RoutingTestNetwork 使用 DistVecRoutingNode 进行的硬编码(所有的子模块都是这类型),但是如果切换到其他路由算法则比较麻烦.

NED 可以添加一个字符串参数,比如 routingNodeType 到复合模块 RoutingTestNetwork,子模块不再是某个固定的类型,其类型包含在 routingNodeType 参数中.现在用户可以自由地从字符串常量"DistVecRoutingNode", "AntNetRouting1Node"或"AntNetRouting2Node"中选择,网络将使用选择的路由算法.

如果指定一个错误的值,比如"FooBarRoutingNode",但用户没有实现"FooBarRoutingNode"模块,那么仿真开始时得到一个运行时间错误: module type definition not found.

在 RoutingTestNetwork 模块内部指定参数值和连接路由模块的门.为了提供一些类型安全等级 NED 要保证没有拼错的参数或门名称,并正确地使用.为了完成这些检验,NED 需要一些帮助:必须命名一个现存的模块类型(比如 RoutingNode),并保证运行的所有模块中指定的 routingNodeTyp 参数至少与 RoutingNode 模块的参数和门是相同的.

[以上的方法,与面向对象语言中的多态性类似— RoutingNode 类似于基类, DistVecRoutingNode 和 AntNetRouting1Node 类似于派生类, routingNodeType 参数类似指向基类的指针,可以向下指定类型.]

以上的都通过 like 关键字完成.语法如下:

```
module RoutingTestNetwork
    parameters:
        routingNodeType: string; // should hold the name of an existing module type
    gates: //...
    submodules:
        node1: routingNodeType like RoutingNode;
        node2: routingNodeType like RoutingNode;
        //...
    connections nocheck:
        node1.out0 --> node2.in0;
        //...
endmodule
```

RoutingNode 模块类型不必用 C++语言实现,因为并不创建其实体,仅用于检查 NED 文件的正确性.

一方面,实际模块类型将被替代(如 DistVecRoutingNode, AntNetRoutingInode 等),不需要在 NED 文件中声明.

like 短语可以用于创建模块族,服务类似的目的,实现相同的接口(相同的门和参数),在 NED 文件中交替使用.

3.5.4 指定子模块参数的值

如果子模块的模块类型声明中含有参数,则可以在子模块声明的 parameters 域给其赋值,所赋值可以使用一个常量(比如 42 或"www.foo.org"),各种参数(常见的为复合模块参数),或任意表达式.

parameters 域并不强制要求给每个参数赋值.未赋值的参数可以在运行时得到值,仿真器会交互地提示输入.当然为了灵活性, 通常不在 NED 文件中为参数赋值,而是留给配置文件 omnetpp.ini,这样改变起来更方便.例如:

```
module CompoundModule
    parameters:
        param1: numeric,
        param2: numeric,
        useParam1: bool;
    submodules:
        submodule1: Node
            parameters:
                p1 = 10,
                p2 = param1+param2,
```

```
            p3 = useParam1==true ? param1 : param2;
        //...
endmodule
```

表达式语法非常类似于 C.表达式可以包含常量和定义的复合模块.参数通过值或引用传递.后者的意思为每次访问表达式值时,是在运行时计算(比如从简单模块代码),为仿真打开可能感兴趣的.可以用语法 submodule.parametername(或 submodule[index].Parametername)来引用已经定义的子模块参数.在[3.7]详细描述了表达式.

关键字 input

当一个参数并没有从 NED 文件和配置文件(omnetpp.ini)内获得值,在仿真开始时,将提示用户输入其值.如果计划使用交互式提示,可以指定提示文字和缺省值.

语法如下:

```
   parameters:
      numCPUs = input(10, "Number of processors?"), // default value, prompt
      processingTime = input(10ms), // prompt text
      cacheSize = input;
```

第三个形式实际上是省略参数,但是可以使用其明确你不想从 NED 文件中获得值.

3.5.5 定义子模块门向量的大小

使用 gatesizes 关键字来定义门向量的大小.门向量大小可以是常量,参数或表达式.

例如:

```
simple Node
    gates:
        in: inputs[];
        out: outputs[];
endsimple

module CompoundModule
    parameters:
        numPorts: const;
    submodules:
        node1: Node
            gatesizes:
        //在这个地方指定了该节点的门大小只能是 2
                inputs[2], outputs[2];
        node2: Node
```

```
            gatesizes:
                inputs[numPorts], outputs[numPorts];
        //...
endmodule
```

gatesizes 并不是强制性的.如果一个门向量缺省 gatesizes 那么其大小为 0.

省略 gatesizes 的一个原因是后面在连接域将使用 gate++符号(新的门扩展的门向量).

批注 **[z1]:** gateSizes 的作用

3.5.6 条件参数和 gatesizes 域

在一个子模块定义中可以存在多个参数和 gatesizes 域,它们中每个都有状态标志.

例:

```
module Chain
    parameters: count: const;
    submodules:
        node : Node [count]
            parameters:
                position = "middle";
            parameters if index==0:
                position = "beginning";
            parameters if index==count-1:
                position = "end";
            gatesizes:
                in[2], out[2];
            gatesizes if index==0 || index==count-1:
                in[1], in[1];
    connections:
        //...
endmodule
```

如果状态是不相交的,且参数值或门大小被定义两次,那么最后一次定义的有效,覆盖前面定义的.因此,缺省值将第一个在域中出现.

3.5.7 连接

复合模块定义指定了复合模块的门如何与其直接子模块相连.

可以连接两个子模块或一个子模块与其上层的复合模块.(也可以在内部连接两个复合模块的门,但是很少用).这表示 NED 不允许跨越多个层次级别进行连接—这限制了复合模型实现自我包含,因此促进了可用性.必须遵守门方向,即不能连接两个输出门或两个输入门.

仅支持一对一的连接,于是门只能被用于某一个方向的链接.一对多或者多对一的链接可以利用复制消息或者合并消息流的简单模块实现,其基本原理是这种扇入或者扇出无论在模型的什么地方发生总是与一些处理过程相联系的.

连接在复合模块定义的 onnections:域指定,所列出的所有链接用分号隔开.

例:

```
module CompoundModule
    parameters: //...
    gates: //...
    submodules: //...
    connections:
        node1.output --> node2.input;
        node1.input <-- node2.output;
        //...
endmodule
```

批注 **[z2]:** 指定模块联接

源门是一子模块的输出门或复合模块的输入门,目的门可以是一子模块的输入门或复合模块的输出门.箭头可以是左-右或右-左指向.

符号 gate++可以扩展一个新门的门向量,在前面的 gatesizes 不需要预先声明其大小.这个特性非常有利于连接一个网络的节点.

批注 **[z3]:** 有点不明白,要再想一想,gate++

```
simple Node
    gates:
        in: in[];
        out: out[];
endsimple


module SmallNet
    submodules:
        node: Node[6];
    connections:
        node[0].out++ --> node[1].in++;
        node[0].in++ <-- node[1].out++;

        node[1].out++ --> node[2].in++;
        node[1].in++ <-- node[2].out++;
```

```
        node[1].out++ --> node[4].in++;
        node[1].in++ <-- node[4].out++;

        node[3].out++ --> node[4].in++;
        node[3].in++ <-- node[4].out++;

        node[4].out++ --> node[5].in++;
        node[4].in++ <-- node[5].out++;
endmodule
```

一个连接:

- 可能有属性 (延迟,比特错误率或数据速率)或使用一个命名信道;
- 内部会出现一个 for-loop 循环(来创建多个连接);
- 可能会有条件.

这些连接类型在以下部分描述.

单连接和信道

如果不指定一个信道,连接将没有传播延迟,没有传输延迟,也没有比特错误率:

```
node1.outGate --> node2.inGate;
```

可以由名称指定一个信道:

```
node1.outGate --> Fiber --> node2.inGate;
```

在这种情况下 NED 源必须包含信道定义.

也可以直接指定信道参数:

```
node1.outGate --> error 1e-9 delay 0.001 --> node2.inGate;
```

参数可以被缺省,也可以以任何顺序出现.

循环连接

如果使用子模块或门向量,可以用一条语句声明多个连接.它们被称为 multiple 或 loop connection.

循环链接由 for 语句创建.

```
for i=0..4 do
    node1.outGate[i] --> node2[i].inGate
endfor;
```

批注 **[z4]:** 循环链接的写法

以上的循环链接结果可以用下图描述.

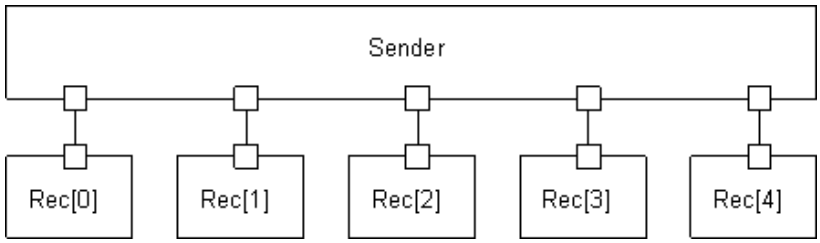


Figure: Loop connection

在一个子 for 语句主体中可以有多个链接,用分号隔开.

在 for 语句中通过指定多个索引可以创建嵌套循环,第一变量形成最外层的循环.

```
for i=0..4, j=0..4 do
    //...
endfor;
```

也可以在并发索引的上下界表达式里使用一个索引:

```
for i=0..3, j=i+1..4 do
    //...
endfor;
```

条件连接

创建有条件的链接可以使用 if 关键字:

```
for i=0..n do
    node1.outGate[i] --> node2[i].inGate if i%2==0;
endfor;
```

批注 **[z5]:** 插入有条件链接

if 条件计算每个链接(在上面的例子中,计算每个 i 的值),每次单独决定是否创建一个链接.
在上面的例子中,我们链接每个偶数门.在下章显示的,条件也使用随机变量.

nocheck 修改器

缺省情况下,NED 要求所有的门都被链接.由于这个检验有时不方便,就会被关闭,使用
nocheck 修改器.

下面的例子产生完整图的一个随机子图.

```
module RandomConnections
    parameters: //..
    gates: //..
    submodules: //..
    connections nocheck:
        for i=0..n-1, j=0..n-1 do
            node[i].out[j] --> node[j].in[i] if uniform(0,1)<0.3;
```

```
        endfor;
endmodule
```

当使用 nocheck 时,简单模块不发送消息至没有被链接的门.

3.6 定义网络

模块的模块声明(复合和简单模块声明)仅仅是定义模块类型.为了真正产生能够运行的模型,需要定义网络.

网络定义即以一个已有模块类型的实体作为仿真模型.通常这里使用复合模型,尽管也有可能设计一个自我包含的简单模块,实体化为网络.

在一个或多个 NED 文件中可以有多个网络定义,用这些 NED 文件仿真程序可以运行其中的任何一个,用户在配置文件(omnetpp.ini)中选择所需要的.

定义网络的语法类似于子模块的声明:

```
network wirelessLAN: WirelessLAN
    parameters:
        numUsers=10,
        httpTraffic=true,
        ftpTraffic=true,
        distanceFromHub=truncnormal(100,60);
endnetwork
```

这里 WirelessLAN 是前面定义的复合模块类型,大概包含 WirelessHost, WirelessHub 等类型的复合模块.

自然地,只有那些没有定义门的模块类型才可用于定义网络.

与子模块中一样,不需要指定所有参数值.未指定的参数可以从配置方件或交互提示获得值.

3.7 表达式

在 NED 语言中,有许多地方允许表达式出现.

表达式具有 C 语言风格的语法.由常用的数学操作符构成.可以值传递或引用传递参数,调用 C 函数,包含随机和输入值等.

当表达式被用来给参数赋值时,每次访问该参数都将执行一次表达式(除非表达式声明来常量).这就意味着在仿真期间一个简单模块查询一个非常量参数每次都会得到不同的值(比如,如果值包含一个随机变量,或其他引用传递的参数).其他的表达式(包括常量参数值)仅执行一次.

XML 类型参数用于方便地访问外部的 XML 文件,XML 类型参数可以由 xmldoc()操作赋值.

3.7.1 常量

Numeric 和 字符串常量

Numeric 常量为一般十进制数或科学记数法表示的数.

字符串常量

字符串常量用双引号引出.

Time 常量

任何地方都可以用 Numeric 常量(整形或实数型)来表示以秒为单位的时间,也可以用毫秒,分,或小时为单位.

```
...
parameters:
    propagationDelay = 560ms, // 0.560s
    connectionTimeout = 6m 30s 500ms, // 390.5s
    recoveryIntvl = 0.5h; // 30 min
```

可以使用以下单位:

Unit	Meaning
ns	nanoseconds
us	microseconds
ms	milliseconds
s	seconds
m	minutes (60s)
h	hours (3600s)
d	days (86400s)

3.7.2 引用参数

表达式可以使用包含它的复合模块的参数(被定义的),或者在 NED 文件中子模块已经定义的参数.后者的语法为 submod.param 或 submod[index].param.

参数名可以使用两个关键字: ancestor 和 ref. ancestor 表示如果复合模块没有该参数,则更高层次的模块将被搜索. ancestor 被认为是不好的习惯,因为它破坏了封闭原则而且只能在运行时间检查其使用是否正确.它存在仅仅是因为在很少的情况下确实需要它来解决.

ref 参数通过引用获得传递值,意味着运行时参数的改变将影响到所有按引用获取参数值的模块.跟 ancestor 一样,ref 也很少被使用.一种可能的使用情况就是:在运行期间调整整个模型,搜索最适合的值:在模块的最高层次定义了一个参数,并使其他的所有模块引用访问该参数—这样如果在运行时改变了参数值(手动或从一个简单模块),将影响整个模块.另一种情形就是,使用参数引用向相邻模块传递状态信息.

3.7.3 操作符

NED 中支持的运算符跟 C/C++中类似,不过也有以下的不同之处:

- ^ 表示幂运算(不是 C 中的按位异或)
- ##用于逻辑异域或 (与值之间的!=相同), # is 用于按位异或
- 按位操作的优先权 (&, |, #) 比关系操作更强的约束,这种优先通常比 C/C++更方便.

所有的值都按双精度型表示.按位运算时,双精度型被转换为无符号长整型.

[会担心长整型值不能精确表示双精度值,这不是问题,IEEE-754 的双精度有 52 位尾数,在这个范围表示的整型数不会有误差.]

执行运算时,使用 C/C++的内部转换原则,再将结果转换回双精度.

类似的,逻辑运算符&&,||和##,运算时,按 C/C++的内部转换原则转换成布尔型,运算完成后,再转换回双精度.对于模数运算,操作数被转换成长整型.

下面是完整的运算符列表,优先权从高到低:

Operator	Meaning
-, !, ~	unary minus, negation, bitwise complement
^	power-of
*, /, %	multiply, divide, modulus
+, -	add, subtract
<<, >>	bitwise shift
&, |, #	bitwise and, or, xor
==	equal
!=	not equal
>, >=	greater, greater or equal
<, <=	less, less or equal
&&, ||, ##	logical operators and, or, xor
?:	the C/C++ ``inline if''

3.7.4 sizeof() 和 index 运算符

批注 **[z6]:** Sizeof()和 index 运算符

sizeof()是一个有用的运算符,可以获得向量门的大小.index 运算符返回当前子模块在模块向量中的索引(以零开始).

以下的例子描述了有多个端口的路由器和一个路径单位.假定门向量 in[]和 out[],大小相同.

```
module Router

    gates:

        in: in[];

        out: out[];

    submodules:

        port: PPPInterface[sizeof(in)]; // one PPP for each input gate

            parameters: interfaceId = 1+index; // 1,2,3...

        routing: RoutingUnit;

            gatesizes:

                in[sizeof(in)];  // one gate pair for each port
```

```
                out[sizeof(in)];
    connections:
        for i = 0..sizeof(in)-1 do
            in[i] --> port[i].in;
            out[i] <-- port[i].out;
            port[i].out --> routing.in[i];
            port[i].in <-- routing.out[i];
        endfor;
endmodule
```

3.7.5 xmldoc()运算符

运算符 xmldoc()可以为 XML 类型参数赋值,即可以指定至 XML 文件或 XML 文件中具有的元素.

xmldoc()有两种看法:一种认为是一个文件,另一种认为是选择 XML 文件内的元素,插入类 XPath 表达式的文件名.例如:

```
xmlparam = xmldoc("someconfig.xml");
xmlparam = xmldoc("someconfig.xml", "/config/profile[@id='2']");
```

OMNeT++支持 XPath 1.0 规范的子集;以下有详细说明.

从 C++代码中访问 XML 元素,像:

```
cXMLElement *rootelement = par("xmlparam").xmlValue();
```

cXMLElement 类提供了一个类 DOM 的访问 XML 文件.可以通过文档树,提取你需要的信息,然后存储至变量或内部文件结构中. cXMLElement 在第[6]章详细说明.

也可以从 omnetpp.ini 中读取 XML 参数:

```
[Parameters]
**.interface[*].config = xmldoc("conf.xml")
```

或

```
[Parameters]
**.interface[*].config=xmldoc("all-in-one.xml","/config/interfaces/interface[2]")
```

3.7.6 支持 XML 文档和 XPath 子集

有两个参数的 xmldoc()中,用一个路径表达式来选择文档中的元素.表达式语法类似 XPath.

如果表达式匹配许多元素,那么将选择第一个元素.这与 XPath 不同,其选择所有匹配节点.

表达式语法如下:

- 表达式由 path components (或"steps")组成,之间用"/"或"//"分隔.
- 路径组件是一个标签名的元素, "*", "." 或"..".
- "/"表式子元素(比如, /usr/bin/gcc), "//"表示在当前元素下的任何层次的元素.

- "." , ".." 和"*"分别表示当前元素,父元素和任何标签名的元素.
- 标签名元素和"*"可以在"[position]"或"[@attribute='value']"形式中有一个可选谓词,从 0 开始.
- "[@attribute=$param]"形式的谓词中, $param 可以是以下的任意一个: $MODULE_FULLPATH, $MODULE_FULLNAME, $MODULE_NAME, $MODULE_INDEX, $MODULE_ID, $PARENTMODULE_FULLPATH, $PARENTMODULE_FULLNAME, $PARENTMODULE_NAME, $PARENTMODULE_INDEX, $PARENTMODULE_ID, $GRANDPARENTMODULE_FULLPATH, $GRANDPARENTMODULE_FULLNAME, $GRANDPARENTMODULE_NAME, $GRANDPARENTMODULE_INDEX, $GRANDPARENTMODULE_ID. [New!]

例:

- /foo - 根元素,必须称为<foo>
- /foo/bar - 根元素<foo>的第一个子元素<bar>
- //bar - 任何地方的第一个<bar> (深度优先搜索)
- /*/bar - 任何标签名的根元素的第一个子元素<bar>
- /*/*/bar - 根元素下的两层子元素的第一个<bar>
- /*/foo[0] - 根元素的第一个子元素<foo>
- /*/foo[1] - 根元素的第二个子元素<foo>
- /*/foo[@color='green'] - 属性"color"值为"green"的第一个子元素<foo>
- //bar[1] - 在任何地方的第二个<bar>元素
- //*[@color='yellow'] - 任何地方的属性"color"值为"yellow"的元素.
- //*[@color='yellow']/foo/bar - 任何地方的属性"color"值为"yellow"第一个子元素为<foo>的第一个子元素<bar>.

当要终止很多小的文件时,路径支持允许输出所有的 XML 配置文件至单个 XML 文档.例如,以下的 sample.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<root>

    <traffic-profile id="low">

        ...

    </traffic-profile>

    <traffic-profile id="medium">

        ...

    </traffic-profile>

    <traffic-profile id="high">

        ...

    </traffic-profile>
```

</root>

在仿真时可以配置模块参数如 xmldoc("sample.xml", "//traffic-profile[@id='low']"); 另一种运行方式如 xmldoc("sample.xml", "//traffic-profile[@id='medium']"),等.

3.7.7 函数

在 NED 表达式中,可以使用以下的算术函数:

- C 语言的<math.h>类的许多函数: exp(), log(), sin(), cos(), floor(), ceil(), 等.
- 产生随机变量的函数: uniform, exponential, normal 和其他已经讨论过的.

此外还可以添加用户自定义的,见[3.7.9].

3.7.8 随机值

表达式可能从不同的分布中包含随机变量.比如参数,除非声明为常量,否则每次运算都返回不同的值.

如果参数被声明为常量,那仅在仿真开始的时候运算,后来对参数的访问都返回相同的值.

随机变量函数使用由 OMNeT++提供的一个随机数产生器(RNGs).缺省情况下,产生 0,但是可以指定使用哪一个.

OMNeT++ 有以下的预定义分布:

批注 **[z7]:** OMNeT++ 有以下的预定义分布:

函数	分布
连续分布	
uniform(a, b, rng=0)	在[a,b)范围内平均分布
exponential(mean, rng=0)	给定的平均数 mean 指数分布
normal(mean, stddev, rng=0)	给定平均数 mean 和标准差的正态分布
truncnormal(mean, stddev, rng=0)	正态分布裁剪非负值
gamma_d(alpha, beta, rng=0)	参数为 alpha>0, beta>0 的 gamma 分布
beta(alpha1, alpha2, rng=0)	参数为 alpha1>0, alpha2>0 的 beta 分布
erlang_k(k, mean, rng=0)	k>0 阶段并且给定的平均数 mean 的 Erlang 分布
chi_square(k, rng=0)	自由度 k>0 的 Chi 平方分布
student_t(i, rng=0)	student-t 分布,自由度 i>0
cauchy(a, b, rng=0)	Cauchy 分布,参数 a,b 其中 b>0
triang(a, b, c, rng=0)	三角分布,其中 a<=b<=c, a!=c
lognormal(m, s, rng=0)	对数正态分布,平均值为 m 且变化 s>0
weibull(a, b, rng=0)	Weibull 分布,参数 a>0, b>0
pareto_shifted(a, b, c, rng=0)	普通的 Pareto 分布,参数 a, b 和 变化 c

离散分布	
intuniform(a, b, rng=0)	从 a 到 b 的平均整数
bernoulli(p, rng=0)	Bernoulli 试验结果,概率 0<=p<=1 (1 表示概率 p,0 表示概率(1-p)
binomial(n, p, rng=0)	二项式分布,参数 n>=0 且 0<=p<=1
geometric(p, rng=0)	几何分布,参数 0<=p<=1
negbinomial(n, p, rng=0)	二项式分布 n>0 且 0<=p<=1
poisson(lambda, rng=0)	Poisson 分布,参数 lambda

如果不指定可选的 rng 参数,函数将使用产生的随机数 0.

例:

```
intuniform(0,10)/10  // one of: 0, 0.1, 0.2, ..., 0.9, 1

exponential(5)       // exponential with mean=5 (thus parameter=0.2)

2+truncnormal(5,3)   // normal distr with mean 7 truncated to >=2 values
```

以上的分布由 C 函数实现,可以很方便地添加自定的函数.自定义的函数与库中的函数一样处理.

3.7.9 自定义函数

用户自定义的函数必须用 C++编码.C++函数必须要有双精度的 0,1,2,3 或 4 个参数,返回一个双精度值.函数必须在 C++文件中用 Define_Function()宏登记过.

函数的例子(以下代码必须出现在一个 C++源文件中):

```
#include <omnetpp.h>

double average(double a, double b)

{

  return (a+b)/2;

}
```

```
Define_Function(average, 2);
```

数字 2 表示 average()有两个参数.这样,以后 average()函数就可以在 NED 文件中使用了.

批注 **[z8]:** 定义一个函数,Define_Function(average,2) .average 表示函数名,2 表示这个函数有几个参数

```
module Compound

    parameter: a,b;

    submodules:

        proc: Processor

            parameters: av = average(a,b);

endmodule
```

批注 **[z9]:** 表明了怎么在 ned 文件中使用这个函数

如果函数的参数是 int 或 long 或其他不是双精度的类型,可以创建一个外围函数通过转换,使其全为双精度.在这个例子中必须使用 Define_Function2()宏来登记外围函数,允许函数的登记名与函数的实现名不一样.如果返回值不是双精度也可以同样做.

```
#include <omnetpp.h>

long factorial(int k)

{

  ...

}

static double _wrap_factorial(double k)

{

  return factorial((int)k);

}

Define_Function2(factorial, _wrap_factorial, 1);
```

3.8 参数化复合模块

通过条件参数,门大小块和条件连接,我们可以创建复杂的拓扑.

3.8.1 例子

Example 1: 路由器

下面的例子包括一个有多个端口路由器,其中端口作为参数.复合模块使用三个模块类型:Application, RoutingModule, DataLink.假定它们是在将导入的单独的文件中定义的.

```
import "modules";

module Router

    parameters:

        rteProcessingDelay, rteBuffersize,

        numOfPorts: const;

    gates:

        in: inputPorts[];

        out: outputPorts[];

    submodules:

        localUser: Application;

        routing: RoutingUnit

            parameters:

                processingDelay = rteProcessingDelay,

                buffersize = rteBuffersize;

            gatesizes:
```

```
            input[numOfPorts+1],
            output[numOfPorts+1];
        portIf: PPPNetworkInterface[numOfPorts]
            parameters:
            retryCount = 5,
            windowSize = 2;
    connections:
        for i=0..numOfPorts-1 do
            routing.output[i] --> portIf[i].fromHigherLayer;
            routing.input[i] <-- portIf[i].toHigherLayer;
            portIf[i].toPort --> outputPorts[i];
            portIf[i].fromPort <-- inputPorts[i];
        endfor;
        routing.output[numOfPorts] --> localUser.input;
        routing.input[numOfPorts] <-- localUser.output;
endmodule
```

Example 2: 信道

比如,可以创建一个类似的模块信道:

```
module Chain
    parameters: count: const;
    submodules:
        node : Node [count]
            gatesizes:
            in[2], out[2];
            gatesizes if index==0 || index==count-1:
            in[1], out[1];
    connections:
        for i = 0..count-2 do
            node[i].out[i!=0 ? 1 : 0] --> node[i+1].in[0];
            node[i].in[i!=0 ? 1 : 0] <-- node[i+1].out[0];
        endfor;
endmodule
```

Example 3: 二进制树形网络

可以使用条件链接构成一个二进制树形网络.下面的 NED 代码循环通过所有可能的节点对,创建二进制树形网络所需要的链接.

```
simple BinaryTreeNode

    gates:

        in: fromupper;

        out: downleft;

        out: downright;

endsimple

module BinaryTree

    parameters:

        height: const;

    submodules:

        node: BinaryTreeNode [ 2^height-1 ];

    connections nocheck:

        for i = 0..2^height-2, j = 0..2^height-2 do

            node[i].downleft --> node[j].fromupper if j==2*i+1;

            node[i].downright --> node[j].fromupper if j==2*i+2;

        endfor;

endmodule
```

注意,不是每个模块门都会被链接.缺省情况下,当仿真开始的时候,不链接的门产生一个运行时错误信息,但是这个错误信息在这里通过 nocheck 修改器关闭.因此,简单模块不负责发送消息到任何地方都不是第一个的门.

提醒读者注意,以上代码要更好的形式.除了在树最底层的节点,每个节点必须正确地链接两个节点,因此,我们可以使用单循环来创建链接.

```
module BinaryTree2

    parameters:

        height: const;

    submodules:

        node: BinaryTreeNode [ 2^height-1 ];

    connections nocheck:

        for i=0..2^(height-1)-2 do

            node[i].downleft --> node[2*i+1].fromupper;

            node[i].downright --> node[2*i+2].fromupper;
```

```
        endfor;
endmodule
```

Example 4: 任意图表

条件链接也用于产生随机的拓扑.以下代码产生一个整图的任意子图:

```
module RandomGraph
    parameters:
        count: const,
        connectedness; // 0.0<x<1.0
    submodules:
        node: Node [count];
            gatesizes: in[count], out[count];
    connections nocheck:
        for i=0..count-1, j=0..count-1 do
            node[i].out[j] --> node[j].in[i]
                if i!=j && uniform(0,1)<connectedness;
        endfor;
endmodule
```

注意这里也使用 nocheck 修改器,为了关闭由未链接门在网络安装代码时产生的错误消息.

3.8.2 复合模块的设计模式

创建有规则结构的复杂的拓扑结构可以使用多个方法,下面介绍其中三种:

全图的子图

这个模式用一个全图的子链接.使用条件来从全图”切割”必须的内部链接:

```
for i=0..N-1, j=0..N-1 do
    node[i].out[...] --> node[j].in[...] if condition(i,j);
endfor;
```

RandomGraph 复合模块(前面描述的)是这个模式的一个例子,但是适当的 condition(i,j) 形成的公式可以产生任意图表.例如,当产生一个树结构时,条件会返回是否节点 j 是节点 j 的子节点,或者相反.

尽管这个模式非常普通,当 N 节点数在高处,且图是稀疏的,那么这种模式就会受到限制(有更少的 N^2 链接).以下两个模式没有这个缺点.

每个节点的链接

模式循环通过所有节点并且创建每个必需的链接.可以类似产生:

```
for i=0..Nnodes, j=0..Nconns(i)-1 do
    node[i].out[j] --> node[rightNodeIndex(i,j)].in[j];
```

endfor;

Hypercube 复合模块(后面所描述的)是这种方法的很简单的例子. BinaryTree 也是这种模式的例子,其中没有内循环 j.

这个模式的适应用依赖于用于公式表示 rightNodeIndex(i,j) 有多简单.

枚举所有的链接

第三个模式是在一个循环内列出所有链接:

```
for i=0..Nconnections-1 do
    node[leftNodeIndex(i)].out[...] --> node[rightNodeIndex(i)].in[...];
endfor;
```

如果 leftNodeIndex(i) and rightNodeIndex(i) 映射函数可以用公式充分地表述,那么可以使用这个模式.

系列模块是这种方法的一个例子,其中映射函数非常简单: leftNodeIndex(i)=i 且 rightNodeIndex(i)=i+1.这个模式也可以用于创建一个有固定链接数量的全图的任意子图.

在不规则的结构中,其中没有布署以上的任何模式,可以使用指定固定的子模块/门向量大小和精确的列出所有链接,就如在大多数现存的仿真器中所做的一样.

3.8.3 拓扑模板

概述

拓扑模板不比复合模块多什么,其中一个或多子模块类型被作为参数(使用 NED 语言的 like 短语).可以写这些实现了 mesh, hypercube, butterfly, perfect shuffle 或其它拓扑的模块,也可以在仿真的任何地方使用它们.有了拓扑模板,就可以重复使用 interconnection structure.

例: hypercube

这个概念在超立方体互连网络中论证过.当构建一个 N 维超立方体时,我们可以充分利用事实,其每个节点都链接 N 个其它的节点,不同的仅仅是节点索引表示的二进制位.

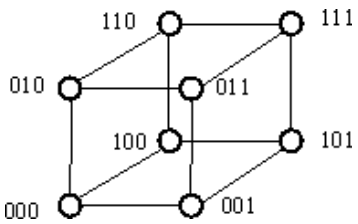


Figure: Hypercube topology

超立方体拓扑模板如下:(可以放入单独的文件,比如 hypercube.ned)

```
simple Node
    gates:
        out: out[];
        in: in[];
```

```
endsimple

module Hypercube

    parameters:

        dim, nodetype;

    submodules:

        node: nodetype[2^dim] like Node

        gatesizes:

            out[dim], in[dim];

    connections:

        for i=0..2^dim-1, j=0..dim-1 do

            node[i].out[j] --> node[i # 2^j].in[j]; // # is bitwise XOR

        endfor;

endmodule
```

当创建一个实际的超立方体时,替换现有的模块类型名(比如"Hypercube_PE")为 nodetype 参数.模块类型实现了用户仿真的算法,必须有与 Node 类型相同的门.通过导入文件使用拓扑模板代码:

```
import "hypercube.ned";

simple Hypercube_PE

    gates: out: out[]; in: in[];

endsimple

network hypercube: Hypercube

    parameters:

        dim = 4,

        nodetype = "Hypercube_PE";

endnetwork
```

如果将 nodetype 参数放入 ini 文件,就可以使用相同的仿真模型来测试,又如,在一个超立方体内有许多路由算法,每个算法由不同的简单模块类型实现—必须为 nodetype 提供不同的值,比如"WormholeRoutingNode", "DeflectionRoutingNode"等.

3.9 大型网络

There are situations when using hand-written NED files to describe network topology is inconvenient, for example when the topology informa tion comes from an external source like a network management program.

在使用手写 NED 文件的情况下描述网络拓扑是非常方便的,例如:当从外部源得到的拓扑信息类似一个网络管理程序.

在这种情况下,有两种可能性:

1. 从数据文件产生 NED 文件
2. 从 C++代码构建网络

这两种方法有不同的优缺点.第一个模型开发阶段更有用,而后一个更有利于写一个比较大的规模的,更多的按产品分类的仿真程序.在下面会测试这两种方法.

3.9.1 产生 NED 文件

像 awk 或 perl 文本处理程序是读文本数据文件的非常杰出的工具,在这里创建 NED 文件.Perl 也可以扩展来访问 SQL 数据库,因此如果拓扑存储在数据库中也能使用.

优点是必需的 awk 或 perl 程序可以在相对短的时间内写出,它后期的维护并不昂贵:如果数据文件的结构改变,NED 创建程序可以简单地修改.NED 文件可以由 nedtool 转换成 C++并且编译,或动态加载.

3.9.2 从 C++代码来构建网络

另一种方法就是写 C++代码,成为仿真可执行文件的一部分.代码会从数据文件或数据库来读拓扑数据,并且使用动态模块创建来直接构建网 络(后面[4.11]详细描述).需要写的代码类似于由 nedtool 输出的*_n.cc 文件.

由于写代码比用 perl 产生 NED 文件复杂,当仿真程序必须更多地按产品分类的时候推荐使用这种方法,例如当 OMNet++和仿真模型被嵌入一个更大的程序时,比如一个网络设计工具.

3.10 XML 绑定 NED 文件

为了增加互用性,NED 文件(还有消息定义文件)有 XML 表示文件.任何 NED 文件都可以转换成 XML,并且任何与 NED DTD 相应的 XML 文件都可以转换成 NED 文件

[DTD 表示文件类型定义,它定义了 XML 文件的"grammar",更多信息见 www.w3.org]

XML 很适合机器处理.例如, stylesheet 转换(XSLT)可以用于从 NED 文件提取信息,或其他的方法从外部的 XML 形式表示的信息创建 NED 文件.一个 XML 的实际应用是 opp_neddoc 文档产生工具,在第[11]章描述.

nedtool 程序可以用于在 NED 和 XML 之间转换.

将 NED 文件转换为 XML 文件:

```
nedtool -x wireless.ned
```

它产生 wireless_n.xml.许多转换控制提取 XML 结果的内容和详细资料,以及在输入时进行许多检验.

将 XML 文件转换加 NED 文件:

```
nedtool -n wireless.xml
```

结果是 wireless_n.ned.

使用 nedtool 作为 NED 编译器产生 C++代码:

```
nedtool wireless.ned
```

结果代码比 nedtool 处理器 nedc 创建的更加简洁.最后,nedtool 创建的_n.cc C++文件编译更快.

也可从 XML 格式产生 C++代码:

```
nedtool wireless.xml
```

4 简单模块

Simple modules 是模型中的活动模块.简单模块用 C++设计,使用 OMNeT++类库.以下简短地介绍普通的离散事件仿真,解释了在 OMNeT++概念和实现中要使用的术语,给出了概述以及如何设计编码简单模块的实践建议.

4.1 仿真概念

为了说明一些在解释 OMNeT++的概念和实现中要使用的术语,这个部分先简短的介绍离散事件仿真(DES)如何工作.

4.1.1 离散事件仿真

“离散事件”系统,即状态改变(事件)发生点在时间域上是离散的,并且事件发生并不需要时间.它假设在两个连续事件之间没有任何事件(如没有任何感觉兴趣的事件)发生,即系统在两个事件之间没有发生状态改变(与连续系统相反,其中状态改变是连续的).这些被看成是离散事件系统的系统可以使用离散事件仿真模块化.(其他系统也可以被模块化为连续仿真模型).

例如,计算机网络通常被视为离散事件系统.有许多事件:

- 包传输开始
- 包传输结束
- 重发超时

表示在两个事件之间,比如包传输开始和包传输结束,没有感兴趣的事件发生.即包的其余状态在传输.所谓”感兴趣的事件和状态”总取决于进行建模的目的和意图.如果我们对单个比特传输感兴趣,那么在我们的事件之间就要包括一些诸如比特传输开始和 结束之类事件.

事件出现的时间通常称为事件时间戳; 在 OMNeT++之间我们通常称为到达时间(由于在类库,”timestamp”为保留字,用于设置事件类中的用户属性).模块中的时间通常称为仿真时间,模块时间或虚拟时间,与实时时间或 CPU 时间不同,其指仿真程序运行的时候和其消耗的 CPU 时间.

4.1.2 事件循环

维护数据结构的事件特征集的离散事件仿真通常称为 FES(特征事件集)或 FEL(特征事件列表).这些仿真通常根据以下的伪代码工作:

```
initialize -- this includes building the model and inserting initial events to FES

while (FES not empty and simulation not yet complete)

{

    retrieve first event from FES

    t:= timestamp of this event

    process event

    (processing may insert new events in FES or delete existing ones )

}

finish simulation (write statistical results, etc.)
```

首先,初始化时通常建立表示仿真模型的数据结构,调用任何用户自定义的初始化代码,插入初始事件到 FES 来确保仿真能够开始.仿真之间的初始化策略各不相同.

随后循环从 FES 消耗事件,并且处理事件.为了保持前后一致关系,事件以严格的时间戳顺序处理,即没有事件会影响前面的事件.

处理一个事件包括调用用户提供的代码.例如,使用计算机网络仿真例子,处理一个”超时”事件可能要包括重新发送网络的包副本,更新重试计数器,调度其他”超时”事件,等.比如当取消超时时,用户代码需从 FES 中移除事件.

当没有事件时(在实践中很少发生),或由于模型时间或 CPU 时间达到了给定的限制或者由于达到了统计所要求的精确性,仿真不需要再进行仿真时,仿真停止.在程序退出之间,用户通常要记录统计的信息至输出文件.

4.1.3 OMNeT++中的简单模块

在 OMNeT++,事件在简单模块内部发生.简单模块封装了 C++代码,产生事件,与事件相互作用,换句话说,实现了模块的行为.

用户使用 cSimpleModule 类的子类创建简单模块类型,其是 OMNeT++类库的一部分. 正如 cCompoundModule,cSimpleModule 都是从公共基类 cModule 派生而来的.

cSimpleModule,尽管打包了仿真相关的函数,但其本身并不做任何事—必须重新定义一些虚拟的成员函数来使其工作.

成员函数如下:

- void initialize()
- void handleMessage(cMessage *msg)
- void activity()
- void finish()

在初始化阶段, OMNeT++构建了网络:创建了需要的简单和复合模块,并根据 NED 定义链接这些模块. OMNeT++也调用所有模块的 initialize()函数.

在事件处理期间调用 handleMessage()和 activity()函数.表示用户在这些函数中实现模型的行为. handleMessage()和 activity()实现不同的事件处理策略:对每个简单模块,用户必须重新正确定义这些函数的中一个.

当模块接收到消息时,仿真内核调用 handleMessage()方法. activity()是基于协同操作的解决方法,其实现了交互处理的方法(协同操作是非抢先线程,如合作).通常,更推荐使用 handleMessage()而不是 activity()—主要是由于 activity()不能很好地度量.在这章的后面将讨论两个方法,包括他们的优缺点.

写有 activity()和 handleMessage()的模块,可以自由地混合一个仿真模型.

当仿真成功终止时调用 finish()函数. finish()最典型的使用是在仿真期间收集统计记录.

4.1.4 OMNeT++中的事件

OMNeT++使用消息来表示事件.每个事件都是 cMessage 或其子类的实例表示;没有隔离事件类.消息从一个模块发送至另一个模块—这表示,”事件的发生地”就是消息的目的模块,事件发生时的模型时间就是消息到达时间.像”超时”事件通过模块发送消息给自己来实现.

OMNeT++中的仿真时间存储在 C++的 simtime_t 类型中,其类型定义为双精度的.

以到达时间顺序从 FES 消耗事件,来保持前后关系.更精确地,给定两个消息,应用以下规则:

1. 先到达的消息首先执行.如果到达时间相同

2. 优级先权小的先执行.如果优级先权相同,

3. 预定义的或发送早的先执行.

优先权是用户指定的整型消息属性.

存储仿真为双精度的时会带来麻烦.由于有限的机器精度,用不同的方法进行两个双精度计算并不能总是相同的结果,即使他们算术上应该是这样的.例如,加号不是一个结合操作,当其进行浮点计算: (x+y)+z != x+(y+z)!这表示,依赖于两个事件到达时间并不是一个好的方法,除非可以精确地计算.

有人建议在仿真内核引入一个小的 simtime_precision 参数, t_1 和 t_2 如果他们”很接近”(之差小于 simtime_precision)则强制认为他们是相等的.然而,这种方法比实际处理更有可能引起混淆.

4.1.5 FES 实现

在离散事件仿真的实现中,FES 的实现是非常关键的. 在 OMNeT++中,FES 实现了 binary heap,因此大范围地使用数据结构.尽管在一些情况下,奇异的数据结构 skiplist 可能比堆操作更好,但堆也是我们知道的最好的算法.在感兴趣的情况下,FES 在 cMessageHeap 类中实现,但是作为仿真程序员,不需要考虑这个.

4.2 包传输模型

4.2.1 延迟,比特错误率,数据速率

链接有三个参数:用于方便通信网络建模,但对其他模型也很有用:

- 传输延迟(sec)
- 比特错误率(errors/bit)
- 数据速率(bits/sec)

这些参数每个都是可选的.可以为每个链接单独地指定链接参数,或定义链接类型(也称为信道类型)在整个模型中使用.

传输延迟是,当消息通过信道传输时,消息到达时延迟的时间.传输延迟单位为秒.

比特错误率影响通过信道传输的消息.比特错误率(ber)是一比特数据被错误传输的概率.因此,n 比特长度的消息被无错传输的概率为:

$$P_{\text{no bit error}} = (1 - \text{ber})^{\text{length}}$$

在消息传输错误的情况下,设置消息错误标志.

数据速率单位为比特/秒,它用于计算传输延迟.通常消息的发送时间对应传传输的第一个比特,到达时间相对接收的最后一比特. (如下图)

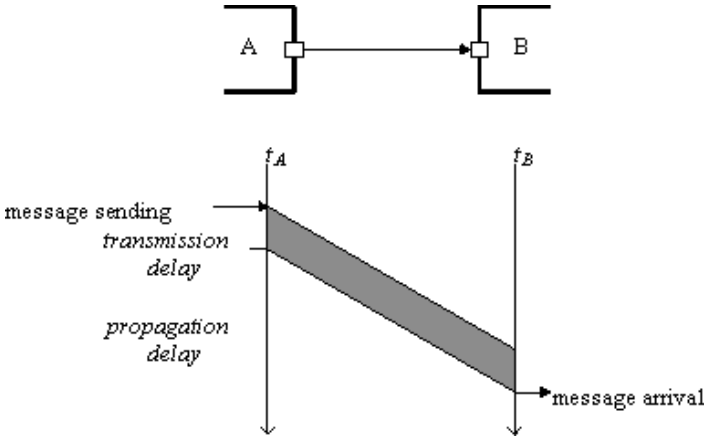


Figure: Message transmission

以上的模型不适合所有协议的建模.在令牌环网和 FDDI 中,在整个帧到达站点之前就开始转发比特数据;换句话说.帧”流过”站点,仅仅被延迟了几个比特.在这种情况下, OMNeT++网络建模时就不使用数据速率.

如果消息沿一条路径传输,通过连续的链接和复合模块,模型的行为仿佛每个模块在等待,直到消息的最后一个比特到达,才开始下一传输.(如下图)

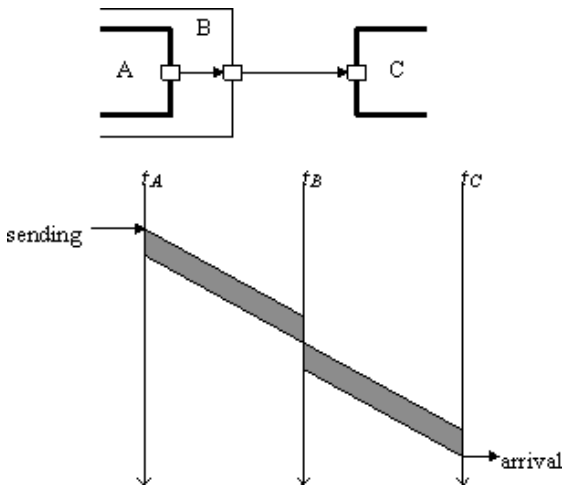


Figure: Message sending over multiple channels

由于以上并不是需要的,通常在路只有一个链接的情况不指定数据速率.

4.2.2 多传输链接

如果为一个链接指定数据速率,那么依赖于链接的长度,消息就有一个非 0 的传输时间.这表示, 通过一输出门的消息,”存储”了门的给定时间(被传输的时间).

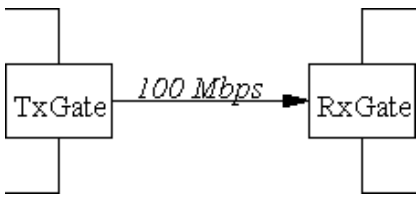


Figure: Connection with a data rate

当一个消息在传输,那么其他消息必须等待直到为个传输完成.当门忙时可以一直发送消息报,但是当模块消息传输开始时将被延迟,就像门一个消息等待传输的内部队列.

OMNeT++类库提供了函数来检查是否一个确定的门正在传输或当传输完成时通知.

如果一个有数据速率的外国投资并不直接链接到简单模块的输出门,但是是路由中的第二个门的话,必须检查门的忙闲状态.

实现消息发送

消息发送的实现如下:在 send()(或仿真)函数激活之后,立即计算消息的到达时间和比特错误率.即如果消息在到达目的地之前要经过多个链接,在 send()调用的计算并不为每个链接单独预定,只执行一次.由于运行时的效率,选择这种实现.

在门忙且建模传输延迟的实际消息队列实现中,在门上消息实际上并不排队;门没有内部队列.每个门将完成传输时通告消息发送,到达的时间可以预先计算出来.然后信息将存储在事件队列(FES)只到仿真时间它的到达时间,被目的模块提取.

结果

实现有以下结果.如果在仿真期间改变链接的延迟(或位错误率,数据速率),在参数改变之前的发送的消息建模将不精确.即如果链接参数改变而在模块中的消息是正在发送,那么消息不会受参数改变的影响,尽管它应该受到影响.然而,后面所有的消息将被正确地模块化.数据速率类似:如果在仿真期间数据速率改变,改变将仅影响改变之后发送的消息.

如果模型门和信道改变重要的参数,可以使用以下两种方法:

- 写一个发送模块,当门完成当前的传输并且发送后调度事件.
- 另外,可以用简单模块(“活动模块”)实现信道.

其他的一些仿真方法

注意一些仿真(比如 OPNET)分配包队列至输入门(端口),发送的消息缓存在目的模块(或链接的远程终点)直到时他们被目的模块接收.在这种方法中,事件和消息是单独的实体,即一个发送操作包括将消息放至包队列以入高度一个事件,用信号通知包到达.在一些实现中,输出门也有包队列,其中包被缓存直到信道准备好(可以传输).

OMNeT++门没有相关的队列.发送但不接收消息的地方缓存在 FES 中. OMNeT++的方法潜在地比上面所说的快,因为它没有入列/出列,也节省事件的创建.缺点是,信道参数的改变不会立即有效.

在 OMNeT++中如果需要可以实现包队列的点-点传输模块.例如,INET 框架就使用这种方法.

4.3 定义简单模块类型

4.3.1 Overview

如上面[4.1.3]所提及的,一个简单模块只是一个必须为 cSimpleModule 子类的 C++类,有一个或多个虚拟成员函数重新定义其行为.

类必须通过 OMNeT++的 Define_Module()宏注册. Define_Module()应该被翻译成.cc 或.cpp 文件,而不是头文件(.h),因为编译器从这里面产生代码.

[为了完整,还有一个 Define_Module_Like()宏,但不鼓励使用,在后面的版本中可能不用]

下面的 HelloModule 是写的最简单的简单模块.(我们不考虑 initialize()使其更小,但如何输出 Hello?)注意 cSimpleModule 是基类,以及 Define_Module()行.

```
// file: HelloModule.cc
```

```
#include <omnetpp.h>

class HelloModule : public cSimpleModule
{
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

// register module class with OMNeT++
Define_Module(HelloModule);

void HelloModule::initialize()
{
    ev << "Hello World!\n";
}

void HelloModule::handleMessage(cMessage *msg)
{
    delete msg; // just discard everything we receive
}
```

为了能够在 NED 文件中引用这个简单模块,我们也需要一个相关的 NED 声明,如下:

```
// file: HelloModule.ned

simple HelloModule
    gates:
        in: in;
endsimple
```

4.3.2 构造器

简单模块不能被用户直接实例化,而由仿真内核实例化.这表示,我们不能任意地构造:署名必须是仿真内核预期的.幸运的是,这个约定很简单:构造器必须为 public,且不能有参数:

```
  public:
    HelloModule();  // constructor takes no arguments
```

cSimpleModule 本身有两个构造器:

1. cSimpleModule() -- 一个没有参数
2. cSimpleModule(size_t stacksize) - 通过协同程序获得堆栈大小

第一个应用于 handleMessage()简单模块,第二个用于 activity()模块.(后面的 activity()模块类方法支行为一个协同程序,其需要单独的 CPU 堆栈,通常 16—32K.这将在后面详细讨论).如果模块构造使用 handleMessage(),那么堆栈大小为 0.

因此,以下定义的构造器都是没问题的,并且使用 handleMessage()用于模块中:

HelloModule::HelloModule() {...}

HelloModule::HelloModule() : cSimpleModule() {...}

也可以缺省构造器,因为也可以编译器产生.

以下的构造器定义选择 activity()用于模块中,协同程序堆栈为 16K:

HelloModule::HelloModule() : cSimpleModule(16384) {...}

4.3.3 构造器和析构器 VS initialize()和 finish()

initialize()和 finish()方法将在后面详细讨论,但由于它们外观类似于构造器和析构器会引起一些混淆,我们在这里简单概括一下.

构造器做为模块安装处理的一部分,在模块创建时调用.在那时,每个都是正在创建,因此不能从构造器做大量的事情.相反, initialize()在仿真开始执行的时候调用,这时每个都已经设置好了.

finish()用于记录统计信息,仅当仿真正常结束时调用.当仿真由于错误消息终止时不调用.析构器也在终点时调用,不管仿真是如何结束的,但公平地假定仿真模型已经中途出错.

基于以上,习惯上存在以下四种方法:

构造器惯例:

设置模块类成员的指针为 NULL;延迟所有的初始化任务至 initialize().

initialize() 惯例:

执行所有的初始化任务:读模块参数,初始化类变量,分配动态数据结构;如果需要也分配且初始化自身消息(定时器).

finish()惯例:

记录统计信息.不删除任何东西,也不取消定时器—所以的清除操作必须在析榴弹内完成.

析构惯例:

删除新建分配的模块类中仍然存在的所有东西.若有自身消息,使用 cancelAndDelete(msg)函数.大多数会错误地从析构器删除一自身消息,由于这它可能在调度事件列表.cancelAndDelete(msg)函数首先检查,然后如果需要,则在删除之间取消消息.

4.3.4 与早期版本的兼容性

OMNeT++3.2 之前的版本,期望一个不同的模块类构造器,如下:

MyModule(const char *name, cModule *parentModule, size_t stack=<stacksize>);

为了方便,也提供了一个 Module_Class_Members()宏,可以缺省的构造器执行.

在 OMNeT++ 3.2 中, Module_Class_Members()宏被保留,但是扩展了新的构造器定义.因此使用 Module_Class_Members()的模块在 OMNeT++ 3.2 或以后的版本中都不需要改变.当不再需要兼容老版本时,只需要简单地删除宏.

一些(很少)模块有手工编写构造器取代使用 Module_Class_Members().这些模块在 OMNeT++ 3.2 或以后版本中将产生一个编译错误,说”no appropriate constructor available”.使其工作的最简单的方法就是对名称和 parentModule 的参数都添加为 NULL 缺省值.

MyModule(const char *name=NULL, cModule *parentModule=NULL, size_t stack=<stacksize>);

另外,当不再需要兼容以老的 OMNeT++版本时,可以删除冗余的构造器参数.

4.3.5 "垃圾收集"和兼容性

在 OMNeT++3.2 之前发布的版本中通常有一个特性,非正式地且稍微不正确地称为”垃圾收集”(GC).这个特点的目的是减少写析构器需要的内容,通常构造器也这样,在仿真终点自动清除.(在仿真期间不做任何事)

OMNeT++(所有版本)保持用户收集仿真对象(通常:消息)的痕迹和其它所有权.”垃圾收集”的特性是在模块清除期间,在每个模块的析构器都完成任务以后,它检查是否还有由模块所有的,但没有被析构器回收的仿真对象— 如果发现这些对象,它调用删除.

它精细地处理 90%的情况,但是偶尔也会导致仿造损毁,不熟悉 OMNeT++内部或缺少高级 C++技术的用户很难调试.

[由于缺少可用的 GC 机制信息,会出现损毁,比如 C++没有提供方法来探测指针是否对象是数组的一部分,或在一个结构体或类中.这个解决方法是使用指针:指针数组,类成员指针等.]

从 OMNeT++ 3.2 开始,这个清除时间 GC 机制缺省情况下是不可用的(perform-gc=配置选项,见[8.2.6]),一般不建议把那设回.不用 GC 的仿真模型运行没有任何损害(除了内存泄漏).

希望通过添加适当的构造器和析构器,现存的模型迟早更新至 OMNeT++ 3.2.为了促成这些处理, OMNeT++在仿真终点丢弃未发布的对象列表.这个丢弃也可以在配置的时候关闭(print-undisposed=配置选项).

4.3.6 例子

以下的代码有一点长,但实际有用的简单模块实现.显示了以上的许多概念,插入将在后面解释的一些概念:

1. 构造器,初始化和析构器惯例
2. 使定时器消息
3. 访问模块参数
4. 在仿真终点记录统计信息
5. 使用 ASSERT()证明程序员的假定

```
// file: FFGenerator.h

#include <omnetpp.h>

/**
 * Generates messages or jobs; see NED file for more info.
 */
class FFGenerator : public cSimpleModule
{
  private:
    cMessage *sendMessageEvent;
```

```
    long numSent;
  public:
    FFGenerator();
    virtual ~FFGenerator();
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};
// file: FFGenerator.cc
#include "FFGenerator.cc"
// register module class with OMNeT++
Define_Module(FFGenerator);
FFGenerator::FFGenerator()
{
    sendMessageEvent = NULL;
}
void FFGenerator::initialize()
{
    numSent = 0;
    sendMessageEvent = new cMessage("sendMessageEvent");
    scheduleAt(0.0, sendMessageEvent);
}
void FFGenerator::handleMessage(cMessage *msg)
{
    ASSERT(msg==sendMessageEvent);
    cMessage *m = new cMessage("packet");
    m->setLength(par("msgLength"));
    send(m, "out");
    numSent++;
    double deltaT = (double)par("sendIaTime");
    scheduleAt(simTime()+deltaT, sendMessageEvent);
```

```
}

void FFGenerator::finish()

{

    recordScalar("packets sent", numSent);

}

FFGenerator::~FFGenerator()

{

    cancelAndDelete(sendMessageEvent);

}
```

如果要能够在 NED 文件内引用,也需要进行 NED 声明:

```
// file: FFGenerator.ned

simple FFGenerator

    parameters:

        sendIaTime: numeric;

    gates:

        out: out;

endsimple
```

4.3.7 使用全局变量

如果可能,避免使用全局变量,包括静态类成员.他们容易引起许多问题.首先,在 Tkenv 中重新构建仿真或在 Cmdenv 中开始运行另一个时它们不会重置初始值 (0).这会产生意想不到的值.其次,其禁止并行运行仿真.当使用并行仿真时,分别处理模块每部分的运行,每部分都有全局变量的副本.这通常不是我们想要的.

解决方法是,将这些变量封装入简单模块中,作为 private 或 protected 的成员数据,通过 public 方法使用.那么其它模块就可以调用这些 public 方法来得到或设置值.其它模块调用方法将在[4.10]中讨论.这些模块的例子是移动框架中的 Blackboard 和 INET 框架中的 InterfaceTable 和 RoutingTable.

4.4 添加函数至 cSimpleModule

这部分讨论之前提及的 cSimpleModule 的成员函数,方便用户重新定义: initialize(), handleMessage(),activity()和 finish(),再加上不经常使用的 handleParameterChange.

4.4.1 handleMessage()

批注 **[z10]:** 每一个消息或者叫事件到达时调用

每个事件调用的函数

意思是在每个事件(消息到达)中,我们简单调用一个用户定义的函数. 这个函数, handleMessage(cMessage *msg)是 cSimpleModule 虚拟的成员函数,缺省情况下不做任何动作—用户必须在子类中重新定义,添加消息处理代码.

handleMessage()函数在模块中每个消息到达的时候调用.函数应用处理消息并且处理后立即返回.每个调用的仿真时间不同.在调用 handleMessage()时不消耗仿真时间.

仿真器内部事件循环处理 activity()和 handleMessage()简单模块,有以下相应的伪代码:

```
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    m:= module containing this event
    if (m works with handleMessage())
        m->handleMessage( event )
    else // m works with activity()
        transferTo( m )
}
```

具有 handleMessage()的模块不会自动开始:仿真内核仅为具有 activity()模块创建开始消息.这表示,如果想要一个没有从其它模块接收消息的 handleMessage()模块通过”自身”开始工作,就需要从 initialize()调度自身消息.

handleMessage()设计

在简单模块中使用 handleMessage(),指定堆栈的大小为 0.这非常重要,因为这用于告知 OMNeT++你要使用的是 handleMessage()而不是 activity().

你可以在 handleMessage()中使用相关的消息/事件函数:

- send()函数簇 - 发送消息至其它模块
- scheduleAt() - 调度一个事件(模块发送消息给自己)
- cancelEvent() -- scheduleAt()删除一个事件调度

不能在 handleMessage()中使用 receive()和 wait()函数,由于他们本质是基于协同工作的,在关于 activity()的章节中介绍.

对每个要存储的信息片面,必须添加成员数据至模块类.这信息不能存储在 handleMessage()的局部变量中,因为它们在函数返回时销毁.他们也不存储为函数的静态变量,因为它们被所有的类实例共享.

成员数据添加至模块类,通常要包括类似以下的工作:

- 状态 (比如 IDLE/BUSY, CONN_DOWN/CONN_ALIVE/...)
- 发球模块状态的其它变量:重试次数,包队列等
- 接收/计算值,然后存储:模块参数值,门索引,路由信息等.
- 消息对象指针创建,然后为定时器,超时器等重复使用.
- 统计/收集的变量/对象

可以通过 initialize()函数初始化这些变量.由于构造器在网络安装阶段调用,模型仍然在构建,因此要使用的大量信息还不可用,所以构造器不是做这些操作的最佳处.

其它在 initialize()中做的任务是调度初始化事件,触发时首先调用 handleMessage().在调用之后, handleMessage()本身必须关注调用的下一个事件,使得”信道”不被破坏. 如果模块仅处理从其它模块到来的信息时,调度事件不是必须的.

finish()通常在仿真终点,用于记录统计类成员数据中的信息堆积.

应用领域:

handleMessage()在大多数情况下比选择 activity()好

1. 当你希望模块用于大的仿真中,包括数几千个模块.在这种情况下, activity()需要的模块堆栈数会消耗大量的内存
2. 对维护很小或没有状态信息的模块,比如包汇聚, handleMessage()更便于设计.
3. 其它的好处是有模块的大的状态空间和许多任意状态传输概率 (比如对任何状态可能有许多并发的状态).这些算法在 activity()中设计非常复杂,但其结果代码比 handleMessage()好.大多数的通信协议都像这样.

Example 1: 协议模型

在通信网络中的协议层模型往往在高层次中有一公共结构,因为基本上他们作用于三类事件:从高层次到达消息的协议(或 apps),从低层次到达消息的协议(从网络),和各种定时器和超时器(即自身消息)和协议.

这通常形成以下的代码形式:

```
class FooProtocol : public cSimpleModule
{
  protected:
    // state variables
    // ...
    virtual void processMsgFromHigherLayer(cMessage *packet);
    virtual void processMsgFromLowerLayer(FooPacket *packet);
    virtual void processTimer(cMessage *timer);
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
// ...
void FooProtocol::handleMessage(cMessage *msg)
{
    if (msg->isSelfMessage())
        processTimer(msg);
    else if (msg->arrivedOn("fromNetw"))
        processMsgFromLowerLayer(check_and_cast<FooPacket *>(msg));
```

```
    else
        processMsgFromHigherLayer(msg);
}
```

函数processMsgFromHigherLayer(), processMsgFromLowerLayer()和processTimer()通常分离成更多的:它们隔离方法来处理分离的包类型和分离的定时器.

Example 2: 简单通讯产生器和汇聚

handleMessage()的简单包产生器和汇聚设计可以像以下的伪代码一样简单:

```
PacketGenerator::handleMessage(msg)
{
    create and send out a new packet;
    schedule msg again to trigger next call to handleMessage;
}
PacketSink::handleMessage(msg)
{
    delete msg;
}
```

注意 PacketGenerator 需要重新定义 initialize()来创建 m 和调度第一个事件.

以下的简单模块产生指数级到达时间的包(代码中一些细节还未讨论,但是代码可以理解)

```
class Generator : public cSimpleModule
{
  public:
    Generator() : cSimpleModule()
  protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
Define_Module(Generator);
void Generator::initialize()
{
    // schedule first sending
    scheduleAt(simTime(), new cMessage);
}
void Generator::handleMessage(cMessage *msg)
```

```
{
    // generate & send packet
    cMessage *pkt = new cMessage;
    send(pkt, "out");
    // schedule next call
    scheduleAt(simTime()+exponential(1.0), msg);
}
```

Example 3: 脉冲通信产生器

更现实的例子就是重写产生器来产生脉冲包,每个都包括包的脉冲长度.

我们在类中添加两个成员数据:

- burstLength 将存储在参数中,指定一脉冲必须包含多少包.
- burstCounter 计算在当前脉冲中还剩多少包要发送.

代码:

```
class BurstyGenerator : public cSimpleModule
{
  protected:
    int burstLength;
    int burstCounter;
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
Define_Module(BurstyGenerator);
void BurstyGenerator::initialize()
{
    // init parameters and state variables
    burstLength = par("burstLength");
    burstCounter = burstLength;
    // schedule first packet of first burst
    scheduleAt(simTime(), new cMessage);
}
void BurstyGenerator::handleMessage(cMessage *msg)
{
```

```
    // generate & send packet
    cMessage *pkt = new cMessage;
    send(pkt, "out");
    // if this was the last packet of the burst
    if (--burstCounter == 0)
    {
        // schedule next burst
        burstCounter = burstLength;
        scheduleAt(simTime()+exponential(5.0), msg);
    }
    else
    {
        // schedule next sending within burst
        scheduleAt(simTime()+exponential(1.0), msg);
    }
}
```

使用 handleMessage()的优缺点:

优点:

- 消耗很少的内存:简单模块不需要单独的栈
- 快速:函数调用比在协同程序间切换要快

缺点:

- 局部变量不能用于存储状态信息
- 需要重新定义 initialize()

通常宁愿选择 handleMessage()而不是 activity().

其它仿真器

许多使用类似方法的仿真包通常都超越隐藏了底层的函数调用如状态机 (FMS)之类,比如系统:

- 使用图形化编辑进行 FSM 设计 OPNET™
- NetSim++与 OPNET 的方法一样
- SMURPH(艾伯特大学)定义语言来描述 FSM,并使用预编译器将其转换成 C++代码.
- Ptolemy(加州大学伯克利分校)使用类似的方法

OMNeT++的 FSM 支持的描述在下面描述.

4.4.2 activity()

进程式描述

通过 activity()编写简单模块可以像对操作系统进程/线程编程一样.可以在代码的任何点等待到来的消息(事件),在后面的某个时间(模型时间)可以挂起执行等.当 activity()函数结束时,模块被终止.(如果还有其它模块在运行,那么仿真可以继续.)

在 activity()中可以使用的最重要的函数是(在后面详细讨论):

- receive() - 接收消息(事件)
- wait() - 用于后面某个时间(模型时间)挂起执行
- send()函数簇 - 发送消息给其它模块
- scheduleAt() - 调度一个事件 (模块发送消息给本身)
- cancelEvent() -- scheduleAt()删除一个事件调度
- end() - 结束模块的执行 (如同退出 activity() 函数)

activity() 函数通常包括一个有限循环,在函数体内至少调用 wait()或 receive()函数.

应用领域

通常宁愿选择 handleMessage()而不是 activity().activity()主要的问题是由于每个模块都需要单独的协同工作栈,不具有可扩展性.同时 activity()也不提倡良好的编程风格.

有一种情况下使用 activity()的进程式描述比较方便:当进程有很多的状态,但状态转换很少,比如,从一个进程状态仅能转换为其他少数几个状态.例如,使用单个网络链接的网络应用程序的设计就是这种情况.应用程序对传输层协议的伪代码类似如下:

```
activity()
{
    while(true)
    {
        open connection by sending OPEN command to transport layer
        receive reply from transport layer
        if (open not successful)
        {
            wait(some time)
            continue // loop back to while()
        }
        while(there's more to do)
        {
            send data on network connection
            if (connection broken)
            {
```

```
                continue outer loop // loop back to outer while()
            }
            wait(some time)
            receive data on network connection
            if (connection broken)
            {
                continue outer loop // loop back to outer while()
            }
            wait(some time)
        }
        close connection by sending CLOSE command to transport layer
        if (close not successful)
        {
            // handle error
        }
        wait(some time)
    }
}
```

如果需要同时处理多个链接,可以动态创建它们为以上简单模块的实例.动态模块创建将在后面讨论.

有一些不想使用 activity()有情况.如果在 activity()函数中不包括 wait(),仅在有限循环的顶部有一个 receive()调用,那么就没必要使用 activity(),用 handleMessage()编写代码.

例如:

```
void Sink::activity()
{
    while(true)
    {
        msg = receive();
        delete msg;
    }
}
```

should rather be programmed as:

```
void Sink::handleMessage(cMessage *msg)
```

```
{
    delete msg;
}
```

Activity() 作为协同程序运行

activity()作为协同程序运行.协同程序是一些非优先抢占调度的线程(也称为协同多任务处理).通过调用 transferTo(otherCoroutine)可以从一个协同程序切换到另一个协同程序.那么这个协同程序将被挂起, otherCoroutine 运行.然后,当 otherCoroutine 调用 transferTo(firstCoroutine)那么第一个协同程序将从 transferTo(firstCoroutine)调用点继续执行.当其它协同程序在执行时,该协同程序全部的状态包括局部变量都被存储.这表示每个协同程序必须有其自己的处理栈,且 transferTo()包括切换一个处理栈至另一个栈.

协同程序在 OMNeT++非常重要,在协同程序类库中仿真程序员不需要调用 transferTo()或其它函数,也不需要关心协同程序类库的实现.然而,重要的是理解,在进行协同程序的离散事件仿真中如何发现事件循环.

当使用协同程序时,事件循环类似如下:

```
while (FES not empty and simulation not yet complete)
{
    retrieve first event from FES
    t:= timestamp of this event
    transferTo(module containing the event)
}
```

也就是说,当模块有一事件时,仿真内核传输控制至模块的协同程序.要求当模块决定完成事件处理时,它将调用 transferTo(main)传输控制回仿真内核.最初, 在仿真开始的时候,activity()的简单模块通过仿真内核的事件(''starter messages'')插入至 FES.

协同程序如何知道其”事件处理完成”?答复是:当其请求另一个事件时.从仿真内核请求事件的函数是 receive()和 wait(),因此,他们的实现在某个地方包含调用 transferTo(main).

他们在 OMNeT++中的伪代码实现:

```
receive()
{
    transferTo(main)
    retrieve current event
    return the event // remember: events = messages
}
wait()
{
    create event e
    schedule it at (current sim. time + wait interval)
```

```
    transferTo(main)
    retrieve current event
    if (current event is not e) {
        error
    }
    delete e  // note: actual impl. reuses events
    return
}
```

因此, receive()和 wait()的调用在 activity()函数的特殊点,因它们:

- 仿真时候封装在模块中,且
- 其它模块有执行信道

始发消息

activity()写的模块需要始发消息来导入.这些始发消息在仿真开始时,甚至在调用 initialize()函数之前,由 OMNeT++自动插入至 FES.

协同程序栈大小

仿真程序员需要定义协同程序处理堆栈的大小.这个不是自动的.

16 或 32KB 通常较好,但是如果模块使用递归函数或有局部变量,需要大量的栈空间,那么栈的大小应该再大点. OMNeT++有一个内置机制用于检测是否模块的栈太小而溢出.OMNeT++也能返回一个模块实际使用了多少个栈空间,因此可以发现是否过高在估计了栈需求.

使用 activity()的 initialize()和 finish()

由于 activity()的局部变量保存在事件中,在里面可以存储每个变量(状态信息,包缓存等).局部变量在 activity()函数的顶部初始化,因此不需要使用 initialize().

然而,如果你在仿真终端要写统计写,那么就不需要 finish().因为 finish()不能访问 activity()的局部变量,必须将变量和对象统计信息包括在模块类中.因为类成员在 activity()顶部被初始化,也不需要 initialize().

因此,一个典型的安装伪代码如下:

```
class MySimpleModule...
{
    ...
    variables for statistics collection
    activity();
    finish();
};
MySimpleModule::activity()
{
```

```
    declare local vars and initialize them
    initialize statistics collection variables
    while(true)
    {
        ...
    }
}
MySimpleModule::finish()
{
    record statistics into file
}
```

使用 activity()的优缺点:

优点:

- 不需要 initialize(),状态可以存储在 activity()
- 在许多情况下进程式描述是一种自然程序模型

缺点:

- 有限的扩展性:如果有成千上万个简单模块,协同程序栈不能接受仿真程序增长内存的需求.
- 运行时负担:协同程序之间的状态转换比简单的函数调用慢.
- 不提倡好的编程风格:使用 activity()容易导致不可靠的 spaghetti 代码.

在大多数情况下,缺点大过优点,且可以使用 handleMessage()替代.

其他的仿真器

大量的仿真包使用协同程序:

- 所有从 SIMULA 继承的仿真软件(如 C++SIM)都是基于协同程序的,尽管程序模型是完全不同的.
- 仿真/并行设计语言 Maisie 和其后面的 PARSEC 也使用协同程序(尽管是由正常的抢占式线程实现).其基本原理与 OMNeT++非常类似. PARSEC 仅是一个设计语言,它有比较优美的语法但是远远不及 OMNeT++.
- 许多基于 JAVA 的仿真库都是基于 JAVA 线程的.

4.4.3 initialize() and finish()

目的

initialize() - 为许多用户提供安装代码.

finish() -- 在仿真完成时,使用户可以记录统计信息.

它们何时调用,如何调用?

模块的 initialize()函数在第一个事件被处理之前,初始事件被仿真内核置入 FES 之后调用.

简单模块和复合模块都有 initialize()函数.一个复合模块的 initialize()函数在它的子模块之前运行.

当事件循环结束的时候, 仅当其正常结束时(比如非运行时错误)调用 finish()函数.调用顺序与 initialize()相反:第一个子模块包含复合模块.

批注 **[z11]:** Initialize()和finish()的使用说明

这些概括为如下伪代码:

```
perform simulation run:
    build network
      (i.e. the system module and its submodules recursively)
    insert starter messages for all submodules using activity()
    do callInitialize() on system module
        enter event loop // (described earlier)
    if (event loop terminated normally) // i.e. no errors
        do callFinish() on system module
    clean up
callInitialize()
{
    call to user-defined initialize() function
    if (module is compound)
        for (each submodule)
            do callInitialize() on submodule
}
callFinish()
{
    if (module is compound)
        for (each submodule)
            do callFinish() on submodule
    call to user-defined finish() function
}
```

initialize() vs. 构造器

通常不应该将仿真相关的代码写入简单模块构造器.这是因为在仿真开始时模块通常需要研究他们的环境,并且将收集的信息保存至内部表.因为当构造器被调用时,网络开始安装,所以像那样的代码并不能放入构造器中.

finish() vs. 析构器

记住并不总是调用 finish(),因此这里并不是每次删除模块时清除代码的最好地方. finish()仅适合写统计信息,加工处理的结果和其他操作,其仅在成功完成时运行.清除代码应该放入析构器.

多级初始化

在仿真模型中,当一级初始化由 initialize()提供并不足够,可以使用多级初始化.模块有两个函数可以用户重新定义:

```
void initialize(int stage);

int numInitStages() const;
```

仿真开始时,所有的模块都调用 initialize(0),然后是 initialize(1), initialize(2)等.可以想像,像初始化可以在许多情况下发生.对每个模块, numInitStages()必须重新定义返回需要的初始级数,例如对两级初始化, numInitStages()应该返回 2, 且 initialize(int stage)必须实现处理 stage=0 和 stage=1 的情况.

[注意 numInitStages()声明为常量,如果你忘记了,通过 C++规则,在基类中创建一个不同的函数来取代重新定义的现存的函数,因此现存的保持有效并返回 1.]

callInitialize()函数为模块和所有子模块执行全部的多级初始化.

如果不定义多级初始化函数,缺省的是单级初始化:缺省 numInitStages()返回 1,缺省的 initialize(int stage)仅调用 initialize().

仿真终点事件

在许多仿真器中通过引入一个特殊的仿真终止事件解决了 finish()的任务.因为仿真程序员必须编写模块(通常表示为 FSM)并不是一个非常好的方法,因此无论哪个状态,他们总能合适地响应仿真终止事件.

这可以在 PARSEC 仿真语言的设计中被见证.其前面的 Maisie 使用仿真终止事件,但是— 像 PARSEC 手册所描述的—这会导致在许多情况下编程困难,因此对于 PARSEC 仿真终止事件在 finish()中被放弃.

4.4.4 handleParameterChange() [New!]

在 OMNeT++ 3.2 中添加了 handleParameterChange()方法,当模块参数改变时由仿真内核调用.这个方法表示如下:

```
void handleParameterChange(const char *parname);
```

用户可以重新定义这个方法使模块响应运行时参数的改变.典型的使用是重读改变的参数,如果需要则更新模块状态.例如,如果超时值改变,那么可以重启或修改运行的定时器.

这个函数主要的目的是为了方便实现 scenario manager 模块的实现,该模块在一定的仿真中,可以编程来改变参数.这些模块非常方便于研究包括瞬时行为.

以下的例子显示了一个队列模块,支持参数 serviceTime 运行时改变:

```
void Queue::handleParameterChange(const char *parname )

{

    if (strcmp(parname, "serviceTime")==0)

    {
```

```
        // queue service time parameter changed, re-read it
        serviceTime = par("serviceTime");
        // if there any job being serviced, modify its service time
        if (endServiceMsg->isScheduled())
        {
            cancelEvent(endServiceMsg);
            scheduleAt(simTime()+serviceTime, endServiceMsg);
        }
    }
}
```

4.4.5 通过子类复用模块代码

一个简单模块通常需要许多变量.一个好的设计策略是创建一个有公共特性的简单模块类,然后从其创建具体的简单模块类型子类.

例如:

```
class ModifiedTransportProtocol : public TransportProtocol
{
  protected:
    virtual void recalculateTimeout();
};
Define_Module(ModifiedTransportProtocol);
void ModifiedTransportProtocol::recalculateTimeout()
{
    //...
}
```

4.5 OMNeT++中的有限状态机制

概述

有限状态机制(FSM)可以使用 handleMessage()更容易. OMNeT++提供一个类和宏集来构建 FSM. OMNeT++的 FSM 的工作非常类似 OPNET 或 SDL.

关键点是:

- 有两类状态:瞬时的和稳定的.在每个事件中(即在每次调用 handleMessage()),FSM 传输当前(稳定)的状态,经历一系列的状态改变(通过许多的瞬时状态运行),最终到达另一个稳定状态.因此在两事件之间,系统总是处于稳定状态.瞬时状态并不真正必须—他们存在仅仅是在传输过程中以一个便利的方法来组合行动.
- 可以设计程序代码来处理进入和离开一个状态.停留不动的相同的状态被处理为离开和重进入状态.

- 进入代码不应该修改状态(由 OMNeT++检验).状态改变(传输)必须被写入退出代码.

OMNeT++的 FSM 可以被嵌套.这表示任何状态(不管进入或退出代码)可以包括深一层的 FSM_Switch().这允许引入子状态,如果太大的话,可以引入一结构至状态空间.

FSM API

FSM 状态被存储在一个 cFSM 类型对象中.通过枚举定义可能的状态;枚举也是确切的定义,哪个状态是瞬时和那个状态是稳定的.在下面的例子中 SLEEP 和 ACTIVE 是稳定状态类型,SEND 是瞬时状态(在状态类型中,圆括号内的数字必须惟一,用于构造状态的数字标识):

```
enum {
  INIT = 0,
  SLEEP = FSM_Steady(1),
  ACTIVE = FSM_Steady(2),
  SEND = FSM_Transient(1),
};
```

实际的 FSM 嵌套在一个类似转换的声明 FSM_Switch()中,其中有进入,离开每个状态的情况:

```
FSM_Switch(fsm)
{
  case FSM_Exit(state1):
    //...
    break;
  case FSM_Enter(state1):
    //...
    break;
  case FSM_Exit(state2):
    //...
    break;
  case FSM_Enter(state2):
    //...
    break;
  //...
};
```

状态转换通过调用 FSM_Goto()来完成,在 cFSM 对象中简单地存储了新的状态:

```
FSM_Goto(fsm,newState);
```

FSM 从数字代码 0 开始;这个状态通常命名为 INIT.

调试 FSMFSM 可以记录他们的状态传输 ev,其输出类似:

```
...
FSM GenState: leaving state SLEEP
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SEND
FSM GenState: leaving state SEND
FSM GenState: entering state ACTIVE
...
FSM GenState: leaving state ACTIVE
FSM GenState: entering state SLEEP
...
```

为了能够实现以上的输出,必须在包含 omnetpp.h 之前#define FSM_DEBUG.

```
#define FSM_DEBUG    // enables debug output from FSMs
#include <omnetpp.h>
```

实际的日志输出是通过 FSM_Print()宏完成的.它通常定义如下,但可以在包含 omnetpp.ini 之后未定义 FSM_Print()来改变输出格式,提供一个新的定义.

```
#define FSM_Print(fsm,exiting)
  (ev << "FSM " << (fsm).name()
      << ((exiting) ? ": leaving state " : ": entering state ")
      << (fsm).stateName() << endl)
```

执行

FSM_Switch()是一个宏.它扩展了嵌入在重复直至 FSM 到达稳定状态的 for()循环中的 switch()声明.(代码在 cfsm.h 中)

无限的循环通过计算状态传输的次数来避免:如果一个 FSM 经过 64 次传输还未到达一稳定状态的话,仿真将通过一个错误消息终止.

例子

我们写另一个脉冲产生器.其有两个状态,SLEEP 和 ACTIVE.在 SLEEP 状态,模块什么都不做.在 ACTIVE 状态,它发送一给定的内部到达时间的消息.代码是来自 Fifo2 仿真例子.

```
#define FSM_DEBUG
#include <omnetpp.h>
class BurstyGenerator : public cSimpleModule
{
  protected:
```

```
    // parameters
    double sleepTimeMean;
    double burstTimeMean;
    double sendIATime;
    cPar *msgLength;
    // FSM and its states
    cFSM fsm;
    enum {
      INIT = 0,
      SLEEP = FSM_Steady(1),
      ACTIVE = FSM_Steady(2),
      SEND = FSM_Transient(1),
    };
    // variables used
    int i;
    cMessage *startStopBurst;
    cMessage *sendMessage;
    // the virtual functions
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
Define_Module(BurstyGenerator);
void BurstyGenerator::initialize()
{
    fsm.setName("fsm");
    sleepTimeMean = par("sleepTimeMean");
    burstTimeMean = par("burstTimeMean");
    sendIATime = par("sendIATime");
    msgLength = &par("msgLength");
    i = 0;
    WATCH(i); // always put watches in initialize()
    startStopBurst = new cMessage("startStopBurst");
```

```
    sendMessage = new cMessage("sendMessage");
    scheduleAt(0.0,startStopBurst);
}
void BurstyGenerator::handleMessage(cMessage *msg)
{
  FSM_Switch(fsm)
 {
    case FSM_Exit(INIT):
      // transition to SLEEP state
      FSM_Goto(fsm,SLEEP);
      break;
    case FSM_Enter(SLEEP):
      // schedule end of sleep period (start of next burst)
      scheduleAt(simTime()+exponential(sleepTimeMean),
                 startStopBurst);
    break;
    case FSM_Exit(SLEEP):
      // schedule end of this burst
      scheduleAt(simTime()+exponential(burstTimeMean),
                 startStopBurst);
      // transition to ACTIVE state:
      if (msg!=startStopBurst) {
        error("invalid event in state ACTIVE");
      }
      FSM_Goto(fsm,ACTIVE);
      break;
    case FSM_Enter(ACTIVE):
      // schedule next sending
      scheduleAt(simTime()+exponential(sendIATime), sendMessage);
    break;
    case FSM_Exit(ACTIVE):
      // transition to either SEND or SLEEP
```

```
      if (msg==sendMessage) {
        FSM_Goto(fsm,SEND);
      } else if (msg==startStopBurst) {
        cancelEvent(sendMessage);
        FSM_Goto(fsm,SLEEP);
      } else {
        error("invalid event in state ACTIVE");
      }
      break;
    case FSM_Exit(SEND):
    {
      // generate and send out job
      char msgname[32];
      sprintf( msgname, "job-%d", ++i);
      ev << "Generating " << msgname << endl;
      cMessage *job = new cMessage(msgname);
      job->setLength( (long) *msgLength );
      job->setTimestamp();
      send( job, "out" );
      // return to ACTIVE
      FSM_Goto(fsm,ACTIVE);
      break;
    }
  }
}
```

4.6 发送和接收消息

在一个抽象层次, OMNeT++仿真模型是一个简单模块的集合,其中每个模块通过消息传递彼此通信.简单模块的本质就是他们创建,发送,接收,存储,修改,调度和销毁消息——一切都是为了促进这项工作,并且收集关于收到的统计信息.

在OMNeT++中的消息是cMessage类的实例,或其的一个子类.消息对象使用C++新建的操作符创建,当它们不再需要时使用删除操作符销毁.在它们的生命期中,模块之间通过门和链接进行消息传播(或通过链接直接发送),或者被调度,传递至模块,表示模块的内部事件.

在第[5]章详细描述消息.在这一点上,我们所需要知道的是他们被称为 cMessage *指针.消息对象也给定了描述名称(常量 char *字符串),通常帮助仿真调试.消息名字符串可以在构造器里指定,所以如果在下的例子中看到 cMessage("token")不必惊奇.

4.6.1 发送 messages

批注 **[z12]:** 发送 messages

消息对象一旦创建,就可以使用以下任一种函数通过一个输出门发送:

```
send(cMessage *msg, const char *gateName, int index=0);

send(cMessage *msg, int gateId);

send(cMessage *msg, cGate *gate);
```

在第一个函数中,参数 gateName 是发送消息的门名称.如果这个门是一个向量门,index 决定了必须通过哪一个门;反之,则不需要 index.

在第二个和第三个函数中使用的 gateId 和门对象指针.它们比第一个更快,因为它们不需要通过门数组搜索.

例:

```
send(msg, "outGate");

send(msg, "outGates", i); // send via outGates[i]
```

以下的代码例子每 5 个仿真秒创建和发送消息:

```
int outGateId = findGate("outGate");

while(true)

{

  send(new cMessage("packet"), outGateId);

  wait(5);

}
```

包传输建模

如果在一链接上发送的消息有数据速率,那么它如本手册前面所描述的建模,[4.2]节.

如果想完全控制传输过程,有可能需要 cGate 的成员函数 isBusy()和 transmissionFinishes().在[4.8.3]节描述.

4.6.2 广播和中继

当执行仿真协议中频繁出现的两个任务广播和中继时,在多个 send()操作中可能会愿意使用相同的消息.不要这样做—不能多次发送相同的消息对象.在这种情况,可以使用消息副本.

广播消息

在模型中,需要广播消息给多个目的端.广播可以在一个简单模块中通过相同消息的副本来实现,例如,在一门向量的每个门上.如上所描述,在所有的 send()调用中不能使用相同的消息指针—必须创建消息对象的副本,然后再发送.

例如:

```
for (int i=0; i<n; i++)

{

    cMessage *copy = (cMessage *) msg->dup();
```

```
    send(copy, "out", i);
}
delete msg;
```

需要注意最后一个门的消息副本是多余的(我们可以发送原始的消息),所以可以优化为:

```
for (int i=0; i<n-1; i++)   // note n-1 instead of n
{
    cMessage *copy = (cMessage *) msg->dup();
    send(copy, "out", i);
}
send(msg, "out", n-1);  // send original on last gate
```

中继

关于包(帧)中继有许多的通信协议.当执行中继时,不能使用一个相同消息对象的指针,然后一次又一次地发送—这样在第一次重复发送的时候就会得到 not owner of message 错误.

而是,当消息传输时,应该创建并发送消息的副本,而保留其原始的.当确定没有任何中继时,可以删除原始消息.

创建发送一个副本:

```
// (re)transmit packet:
cMessage *copy = (cMessage *) packet->dup();
send(copy, "out");
and finally (when no more retransmissions will occur):
delete packet;
```

为什么?

消息就像现实世界中的对象— 相同时间不能在两个地方.一旦发送了,消息对象就不再属于模块:它被仿真内核接收,最终传递给目的模块.发送模块甚至不能引用它的任何指针.一旦消息到达目的模块,那么该模块就具有所有的权限—可以发送,也能立即销毁,或存储用于以后的处理.被调度的消息应用相同—它们属于仿真内核直至它们被传递回模块.

为了实现以上的规则,所有的消息函数检验发送的消息自身是否实际拥有.如果是其它模块的消息,被调度或在队列中等,就会得到一个运行时错误: not owner of message.

[这个特点不会显著增长运行时的负担,因为它使用对象拥有管理;它仅检验消息的拥有者是否是发送的模块.]

4.6.3 延迟发送

通常在消息发送后立即需要模拟延迟.在 OMNeT++中,可以类似如下实现:

```
wait( someDelay );
send( msg, "outgate" );
```

在延迟期间如果模块需要处理到达的消息,不能使用 wait(),定时器机制在[4.6.7]描述.需要布署”自身消息”.

没有比上面提及的方法更简单的方法了:延迟发送.延迟发送可以通过使用这些函数中的任一个来完成:

```
sendDelayed(cMessage *msg, double delay, const char *gateName, int index);

sendDelayed(cMessage *msg, double delay, int gateId);

sendDelayed(cMessage *msg, double delay, cGate *gate);
```

参数与 send()相同,除了 delay 参数.函数的效果就是为了传输间隔,模块保持消息然后再发送.即消息的发送时间将是当前仿真时间(调用 sendDelayed()的时间)加上延迟.延迟值必须是非负的.

例如:

```
sendDelayed(msg, 0.005, "outGate");
```

4.6.4 直接发送消息

有时需要或者为了便利要忽略门/链接,直接发送消息至远程目的模块.用 sendDirect()函数完成:

```
sendDirect(cMessage *msg, double delay, cModule *mod, int gateId)

sendDirect(cMessage *msg, double delay, cModule *mod, const char *gateName, int index=-1)

sendDirect(cMessage *msg, double delay, cGate *gate)
```

另外消息和延迟,也获得目的模块和门.门应该是输入门,并且不能被链接.换句话说, 通过 sendDirect()函数,模块需要有专门的接收门.(注意,在一个复合模块中离开一个未链接的门,需要指定链接是不检验的:代替在 NED 文件中的普通链接域.)

例如:

```
cModule *destinationModule = parentModule()->submodule("node2");

double delay = truncnormal(0.005, 0.0001);

sendDirect(new cMessage("packet"), delay, destinationModule, "inputGate");
```

在目的模块,直接接收和消息和通过链接接收的消息之间没有区别.

4.6.5 接收消息

仅与 activity()相关!消息接收函数仅在 activity()函数中使用, handleMessage()在其参数列表接收消息.

使用 receive()函数接收消息. receive()是 cSimpleModule 的成员函数.

```
cMessage *msg = receive();
```

cSimpleModule 函数有一个可选的 timeout 参数.(这是一个 delta,不是绝对的仿真时间.)如果相应的消息在 timeout 周期内没有到达,那个函数将返回一个 NULL 指针.

[Putaside 队列和 receiveOn(), receiveNew(),以及 receiveNewOn()函数在 OMNeT++ 2.3 中不推荐使用,并在 OMNeT++ 3.0 中删除.]

```
simtime_t timeout = 3.0;

cMessage *msg = receive( timeout );


if (msg==NULL)

{

    ...   // handle timeout

}

else

{

    ...  // process message

}
```

4.6.6 wait() 函数

仅与 activity()有关! wait()的实现包括调用不能在 handleMessage()函数中使用的 receive()函数.

wait()为一个给定的仿真时间数量(delta)挂起了模块的执行.

```
wait( delay );
```

在其它的仿真软件中, wait()通常称为延迟(hold). wait()函数是通过在 receive()之后的 scheduleAt()函数实现的. wait()在模块中非常方便,不需要为消息到达做准备,比如消息产生器.例:

```
for(;;)

{

  // wait for a (potentially random amount of) time, specified

  // in the interArrivalTime module parameter

  wait( par("interArrivalTime") );

  // generate and send message

  ...

}
```

如果消息在等待间隔到达那么是运行时错误.如果希望消息在等待周期到达的话,可以使用 waitAndEnqueue()函数.它除了等待间隔外,还有一队列对象(cQueue 类,第[6]章描述)指针.在等待间隔期间到达的消息会被放入队列,因此可以调用 waitAndEnqueue()返回后处理这些消息.

```
cQueue queue("queue");

...

waitAndEnqueue(waitTime, &queue);

if (!queue.empty())
```

```
{
  // process messages arrived during wait interval
  ...
}
```

4.6.7 使用自身消息模型化事件

在大多数的仿真模型中,需要实现定时器,或调度未来可能发生的事件.例如,当通信协议模型发送一个包时,当超过超时器时需要调度一个事件,因为它将在后面重新发送.另一个例子,假设不想写处理队列作业的服务器模块.只要开始处理一个作业,当作业处理完成后,那么服务模型将要调度一个事件发生,使其可以开始处理下一个作业.

OMNeT++通过让简单模块发送消息给自身来解决这个问题;消息在时间点之后被传送回简单模块.这种方式使用的消息称为自身消息.自身消息用于在模拟模块中发生的事件.

事件调度

模块可以使用 scheduleAt()函数发送消息给自身. scheduleAt()获得一个绝对仿真时间,通常计算为 simTime()+delta:

```
scheduleAt(absoluteTime, msg);
scheduleAt(simtime()+delta, msg);
```

自身消息传递至模块与其它消息的方法相同(通过调用 receive 或 handleMessage());模块可以通过任何接收消息的 isSelfMessage()成员函数来确定是否是自身消息.

例如,这里有一个如果仿真内核没有提供 activity()简单模块中自身的 wait()函数,那么如何实现?

```
cMessage *msg = new cMessage();
scheduleAt(simtime()+waitTime, msg);
cMessage *recvd = receive();
if (recvd!=msg)
   // hmm, some other event occurred meanwhile: error!
...
```

通过调用其成员函数 isScheduled()来确定消息当前是否在 FES 中:

```
if (msg->isScheduled())
  // currently scheduled
else
  // not scheduled
```

Re-scheduling an event

如果想重新调度当前所调度的事件至一个不同的仿真时间,首先需要使用 cancelEvent()取消.

取消事件

自身消息调度可以被取消(从 FES 移除).由于自身消息通常用于模拟定时器,所以这是相当有用的.

```
cancelEvent( msg );
```

cancelEvent()函数取消指向消息的指针,并返回相同的指针.在其被取消后,可以删除消息或在下一个 scheduleAt()调用中重新使用.如果消息不在 FES 中 cancelEvent()会出现错误.

实现定时器

下面的例子显示了如何实现一个定时器:

```
cMessage *timeoutEvent = new cMessage("timeout");

scheduleAt(simTime()+10.0, timeoutEvent);

//...

cMessage *msg = receive();

if (msg == timeoutEvent)

{

  // timeout expired

}

else

{

  // other message has arrived, timer can be cancelled now:

  delete cancelEvent(timeoutEvent);

}
```

4.6.8 终止仿真

正常终止

可以使用 endSimulation()函数来终止仿真:

```
endSimulation();
```

通常不需要 endSimulation(),因为可以在 ini 文件中指定仿真时间和 CPU 时间.

错误终止

如果检测到错误状态,想要终止仿真器,可以调用 cModule 的成员函数 error().其使用方法类似 printf():

```
if (windowSize<1)

  error("Invalid window size %d; must be >=1", windowSize);
```

在错误文本中不需要包括换行('\n')或标点符号,这会由 OMNet++添加.

4.7 访问模块参数

模块参数可以通过调用 cModule 的成员函数 par()来访问:

```
cPar& delayPar = par("delay");
```

cPar 类是一个普通的值存储对象.支持将类型转换成数值类型,因此参数可以如下读取:

```
int numTasks = par("numTasks");

double processingDelay = par("processingDelay");
```

如果参数是一个随机变量或其值在运行期间会改变,那么最好将其存储为一个引用,要需要的时候每次都重读值:

```
cPar& waitTime = par("waitTime");

for(;;)

{

  //...

  wait( (simtime_t)waitTime );

}
```

如果 wait_time 参数是在 NED 源文件或 ini 文件中给定的一个随机值(如 exponential(1.0)),那么以上的代码每次都会导致不同的延迟.

参数也可以在执行期间通过程序改变.如果参数是引用(在 NED 文件中的引用修改器),其它模块也会看到改变.因此,参数为引用可用作模块通信的方法.

例如:

```
par("waitTime") = 0.12;
```

Or:

```
cPar& waitTime = par("waitTime");

waitTime = 0.12;
```

cPar 类在[6.6]节详细讨论.

4.7.1 仿真参数数组

OMNet++ 3.2 不支持参数数组,但是实践中可以使用字符串参数仿真.可以分配参数一个字符串,其包括了所有的文本形式的值(例如, "0 1.234 3.95 5.467"),然后在简单模块中解析字符串.

cStringTokenizer 类对这种应用相当有用.构造器接收一个字符串,被认为是通过字符串分隔符(缺省是空格),将单词分隔记号的序列.然后多次调用 nextToken()方法将一个接一个地返回记号.在最后一个记号之后,将返回 NULL.

例如可以使用以下的代码解析一个包括整数序列的字符串至一个向量 :

```
const char *str = "34 42 13 46 72 41"; // input

std::vector<int> numbers;  // array to hold the result

cStringTokenizer tokenizer(str);

const char *token;

while ((token = tokenizer.nextToken())!=NULL)

    numbers.push_back(atoi(token));   // convert and store
```

该类也有一个 hasMoreTokens()方法,所以以上的代码也能写为:

```
...
cStringTokenizer tokenizer(str);
while (tokenizer.hasMoreTokens())
    numbers.push_back(atoi(tokenizer.nextToken()));
```

对于转换长整型和双精度型的数据,分别使用 atol()和 atof()来取代 atoi().

为了在字符串向量中存储记号, cStringTokenizer 类有一个方便的函数 asVector(),因此转换可以通过一行代码来完成:

```
const char *str = "34 42 13 46 72 41";
std::vector<std::string> strVec = cStringTokenizer(str).asVector();
```

4.8 访问门和链接

4.8.1 门对象

模块门是 cGate 对象.门对象知道它们与哪个门链接.他们也在链接参数(延迟,数据速率等)上被查询.

cModule 的成员函数 gate()返回一个 cGate 对象的指针,使用函数的超负载形式(overloaded form)访问门向量元素:

```
cGate *outgate = gate("out");
cGate *outvec5gate = gate("outvec",5);
```

对于门向量,第一种形式返回向量中的第一个门(index 为 0).

isVector()成员函数用于确定门是否属于门向量.

给定一个门指针,可以使用 cGate 的成员函数 size()和 index()在确定门向量的大小和向量中门的索引.

```
int size2 = outvec5gate->size(); // --> size of outvec[]
int index = outvec5gate->index(); // --> 5 (it is gate 5 in the vector)
```

也可以调用 cModule 的 gateSize()方法来取代 gate->size(),其使用方法相同:

```
int size2 = gateSize("out");
```

对非向量门, size()返回 1, index()返回 0.

大小为 0 的向量被表示为一个占位符的门,其 size()方法返回 0,不能被链接.

type()成员函数返回一个字符, 'I'为输入门, 'O'为输出门:

```
char type = outgate->type() // --> 'O'
```

门 ID

模块门(输入和输出,单个和向量)存储在他们模块中的一个数组内.门在数组中的位置被称为 gate ID.通过成员函数 id()返回门 ID:

```
int id = outgate->id();
```

对于输入门 fromApp 和 in[3]以及输出门 toApp 和 status,数组类似如下:

ID	dir	name[index]
0	input	fromApp
1	output	toApp
2	empty	
3	input	in[0]
4	input	in[1]
5	input	in[2]
6	output	status

数组会有空的位置.位向量确保占用邻近的 ID,因此其合理地计算 gate[k]的 ID 为 gate("gate",0).id()+k.

消息的发送和接收函数都接受门名称和门 ID;使用门 ID 的函数速度更快点.门 ID 在执行期间不改变,因此通常可以预先获得,使用门 ID 取代门名称.

也可以通过 cModule 的成员函数 findGate()获得门 ID:

```
int id1 = findGate("out");

int id2 = findGate("outvect",5);
```

4.8.2 链接参数

链接属性(传播延迟,传输数据速率,比特错误速率)表示为信道对象,其通过源链接门是可用的.

```
cChannel *chan = outgate->channel();
```

cChannel 是一个小的基类.所有关心的属性都是其子类 cBasicChannel 的一部分,因此必须在得到延迟,错误和数据速率值之前转换为指针.

```
cBasicChannel *chan = check_and_cast<cBasicChannel *>(outgate->channel());

double d = chan->delay();

double e = chan->error();

double r = chan->datarate();
```

也可以通过相应的 setXXX()函数来改变信道属性.不过要注意,以改变不会改变已经发送的消息,即使其并未开始传输([4.2]节解释过).

4.8.3 传输状态

isBusy()成员函数返回是否当前的门正在传输,若然, transmissionFinishes()成员函数当门结束传输时返回仿真时间.(如果门当前不在传输, transmissionFinishes()返回其最后一次传输完成的时间.)

语义在[4.2]节描述.

例如:

```
cMessage *packet = new cMessage("DATA");
packet->setByteLength(1024);  // 1K
if (gate("TxGate")->isBusy()) // if gate is busy, wait until it
{                             // becomes free
  wait( gate("TxGate")->transmissionFinishes() - simTime());
}
send( packet, "TxGate");
```

如果一个数据速率的链接并不直接与简单模块的输出门相链接,但是是路由中的第二个,那么必须检验门的忙闲状态.可以使用以下的代码:

```
if (gate("mygate")->toGate()->isBusy())
  //...
```

注意,如果数据速率在仿真期间改变,那么改变将仅影响到改变发送以后的消息.

4.8.4 连通性

isConnected()成员函数返回是否门被链接.如果是一个输出门,那么与其链接的门通过成员函数 toGate()获得.对于输出门,是 fromGate()函数.

```
cGate *gate = gate("somegate");
if (gate->isConnected())
{
  cGate *othergate = (gate->type()=='O') ? gate->toGate() : gate->fromGate();
  ev << "gate is connected to: " << othergate->fullPath() << endl;
}
else
{
  ev << "gate not connected" << endl;
}
```

可选择的 isConnected()是用于检验 toGate()或 fromGate()的返回值.下面的代码与上面的完全等同:

```
cGate *gate = gate("somegate");
cGate *othergate = (gate->type()=='O') ? gate->toGate() : gate->fromGate();
if (othergate)
  ev << "gate is connected to: " << othergate->fullPath() << endl;
else
  ev << "gate not connected" << endl;
```

为了找出给定的输出门最后平均属于哪个简单模块,可以像这样遍历整个路径(ownerModule()成员函数返回门所属于的模块):

```
cGate *gate = gate("out");
while (gate->toGate()!=NULL)
{
  gate = gate->toGate();
}
cModule *destmod = gate->ownerModule();
```

但是幸运的是有两个方便的函数来完成这个工作： sourceGate() 和 destinationGate()

4.9 遍历模块层次

模块向量

如果模块是模块向量的一部分， index() 和 size()成员函数就可用于查询其索引和向量大小:

```
ev << "This is module [" << module->index() <<
      "] in a vector of size [" << module->size() << "].\n";
```

模块 ID

在网络中的每个模块都有一个通过成员函数 id()返回的惟一的 ID.仿真内核在内部使用模块 ID 来标识模块.

```
int myModuleId = id();
```

如果知道模块 ID,那么就可以从仿真对象(一全局变量)模块指针:

```
int id = 100;
cModule *mod = simulation.module( id );
```

模块 ID 保证是惟一的,即使当模块被动态的创建和销毁.即一个属于被删除模块的 ID 在后面就不能用于其它的模块.

遍历模块层次

周围的复合模块可以通过 parentModule()成员函数访问:

```
cModule *parent = parentModule();
```

例如,父模块的参数像这样被访问:

```
double timeout = parentModule()->par( "timeout" );
```

cModule 的 findSubmodule()和 submodule()成员函数使通过 name(如果子模块在模块向量中,则通过 name+index)来查看模块的子模块成为可能.首先返回子模块的模块 ID 数值,然后返回模块指针.如果没有发现子模块,则分别返回-1 和 NULL.

```
int submodID = compoundmod->findSubmodule("child",5);
cModule *submod = compoundmod->submodule("child",5);
```

moduleByRelativePath()成员函数可用于寻找一层以下的嵌套的子模块.例如:

compoundmod->moduleByRelativePath("child[5].grandchild");

下面代码会得到相同的结果:

compoundmod->submodule("child",5)->submodule("grandchild");

(假定 child[5]存在,因为如果不存在的话,第二个版本中由于 NULL 指针解除引用会访问失败.)

cSimulation::moduleByPath()函数类似于 cModule 的成员函数 moduleByRelativePath(),并且在最顶层开始搜索.

迭代子模块

为了使用一个复合模块中的所有模块,使用 cSubModIterator.

例如:

```
for (cSubModIterator iter(*parentModule()); !iter.end(); iter++)
{
  ev << iter()->fullName();
}
```

(iter()是指向当前迭代的模块的指针.)

以上的方法也可以用于迭代一个模块向量,由于 name()函数所有模块返回的都是相同的:

```
for (cSubModIterator iter(*parentModule()); !iter.end(); iter++)
{
  if (iter()->isName(name())) // if iter() is in the same
                              // vector as this module
  {
    int itsIndex = iter()->index();
    // do something to it
  }
}
```

遍历链接

为了确定模块其它链接的终点,使用 cGate 的 fromGate(),toGate()和 ownerModule()函数,例如:

```
cModule *neighbour = gate( "outputgate" )->toGate()->ownerModule();
```

对于输入门,则用 fromGate()代替 toGate().

4.10 模块之间的直接调用方法

在一些仿真模型中,有可能有一些模块紧密联系,进行有效的基于消息的通信.在这种情况下,需要从另一个模块调用一个简单模块的公共的 C++方法.

简单模块是 C++类,因此可以进行正常的 C++方法调用.然而,需要提及两个问题:

- 如何得到表示模块的对象指针
- 如果让仿真内核知道在模块中发生的一个方法调用.

通常,在相同的复合模块中,调用模块称为调用者,因此 cModule 的 parentModule()和 submodule()可用于得到被调用模块的指针 cModule*. cModule*指针然后必须转换成模块的 C++类,因此,其方法变得可见.

这导致了以下的代码:

```
cModule *calleeModule = parentModule()->submodule("callee");

Callee *callee = check_and_cast<Callee *>(calleeModule);

callee->doSomething();
```

在第二行的 check_and_cast<>()模板函数是 OMNet++的一部分.是一个标准的 C++ dynamic_cast,检验结果:如果是 NULL, check_and_cast 会报一个 OMNet++错误.使用 check_and_cast 节省了你写错误检查代码:如果由于没有发现”calleeModule”子模块,第一行的 calleeModule 是 NULL,或者模块实际上不是 callee 类型,那么 check_and_cast 就会抛出错误.

第二问题是如何让仿真内核知道模块内发生了一个方法调用.为什么有必要放在首位?首先,为了许多内部机制正确地执行,仿真内核总是需要知道当前执行哪个模块代码.(此类机制之一是所有权处理.)其次, Tkenv 仿真 GUI 鼓励使用方法调用,但是由于必须知道他们,所以不能实现.

解决的办法是在方法的顶部添加可以从其它模块调用的 Enter_Method() 或 Enter_Method_Silent()宏. 这些调用实现的背景切换,在使用 Enter_Method()的情况下,通知仿真 GUI,因此可以实现驱动方法的调用. Enter_Method_Silent()不驱动调用. Enter_Method()就像的一个类 printf()的参数列表—结果字符串在其驱动期间显示.

```
void Callee::doSomething()

{

    Enter_Method("doSomething()");

    ...

}
```

4.11 动态模块创建

4.11.1 什么时候需要动态创建模块

在一些情况下需要动态创建和销毁模块.例如,当仿真一个移动网络时,当一个新用户进入仿真领域量就需要创建一个新的模块,当用户离开领域时丢弃它们.

另一个例子是,当实现一个服务器或一个传输协议时,它需要很方便地动态创建模块来服务新的链接,当链接关闭时丢弃它们.(需要写一个管理模块接收链接请求,并且为每个链接创建一个模块. Dyna 仿真实例就是做类似事的.)

简单模块和复合模块都可以被动态创建.如果创建一个复合模块,其所有子模块都需要被递归创建.

它通常可以很方便地直接发送动态创建模块消息.

一旦创建,并且开始,动态模块就”静态模块”就没有任何的区别;例如,我们可以在仿真期间删除静态模块(尽管很少这样使用).

4.11.2 概述

为了理解如何动态创建模块,需要知道一些关于 OMNeT++通常如何实例化模块.每个模块类型(类)都有一个相应的 cModuleType 类对象.这个对象通过 Define_Module()宏声明后被创建,有一个函数族成以实例化模块类(这个函数基本上只包括一个返回新建的 module-class(...)声明).

cModuleType 对象可以通过名称字符串(与模块类名相同)查看.一旦有了其指针,就可以调用其族方法并且创建相应模块类的实例—不需要在源文件中包括有模块类声明的 C++头文件.

cModuleType 对象也知道给定的模块类型的哪些门和参数(该信息来自编译的 NED 代码.)

简单模块可以一步创建.对一个复合模块来说,情况复杂一些,由于其内部的结构(子模块,链接)可能依赖于参数值和门向量大小.因此,对于复合模块来说,通常需要首先创建模块自身,然后设置参数值和门向量大小,然后调用方法创建其子模块和内部链接.

正如已经知道的, activity()的简单模块需要一个始发消息.为了静态创建模块,这个消息由 OMNeT++自动创建,但是对于动态创建模块,需要明确地调用相应的函数来完成.

调用 initialize()需要在插入始发消息以后发生,由于初始化代码需要插入新的消息至 FES,并且这些消息应用在始发消息之后被处理.

4.11.3 创建模块

第一步,找出对象族:

```
cModuleType *moduleType = findModuleType("WirelessNode");
```

简单形式

cModuleType 有一个 createScheduleInit(const char *name, cModule *parentmod)的便利函数来一步完成创建一个模块并且运行.

```
mod = modtype->createScheduleInit("node",this);
```

它完成了 create() + buildInside() + scheduleStart(now) + callInitialize().

这个方法可以用于简单和复合模块.然后其应用稍微有限:因为其在一步内完成,就不需要改变来设置参数或门大小,在调用 initialize()之前链接门.(initialize()希望在其调用时所有的参数和门都已置入并且网络已经完全构建.)由于以上的限制,这个函数主要用于创建基本的简单模块.

扩展形式

如果不能使用以上的简单模式.有五步:

1. 寻找对象族
2. 创建模块
3. (如果需要)设置参数和门大小
4. 调用函数构建子模块并且完成模块的构 建
5. 调用函数为新建的简单模块创建激活消息

每一步(除了第 3 步)可以使用一行代码完成.

见以下的例子,其中忽略第 3 步:

```
// find factory object
cModuleType *moduleType = findModuleType("WirelessNode");
// create (possibly compound) module and build its submodules (if any)
cModule *module = moduleType->create("node", this);
module->buildInside();
// create activation message
module->scheduleStart( simTime() );
```

如果需要设置参数值或门向量大小(第 3 步), 那么在 create() 和 buildInside()之间调用代码:

```
// create
cModuleType *moduleType = findModuleType("WirelessNode");
cModule *module = moduleType->create("node", this);
// set up parameters and gate sizes before we set up its submodules
module->par("address") = ++lastAddress;
module->setGateSize("in", 3);
module->setGateSize("out", 3);
// create internals, and schedule it
module->buildInside();
module->scheduleStart(simTime());
```

4.11.4 删除模块

用于动态删除一个模块:

```
module->deleteModule();
```

如果模块是一个复合模块,就包括递归删除其所有的子模块.一个简单模块也能删除自身;在这种情况下, deleteModule()的调用不会返回给调用者.

目前,不能安全地在简单模块里删除一个复合模块,必须委托复合模块外的模块来完成.

4.11.5 模块删除和 finish()

当在仿真期间要删除一个模块时,其不会自动调用 finish()函数(deleteModule()不会这样做.)如何使被创建的模块在里面不起任何作用: finish()调用所有的子模块 — 在仿真的终站.如果一个模块不再存活,则 finish()不调用,但是仍然可以手动调用.

可以使用 callFinish()来安排调用 finish().这通常不是直接调用 finish()的好方法.如果要删除一个复合模块 callFinish()会递归地为所有子模块调用 finish(),如果要从其它的模块删除一个简单模块, callFinish()会在调用期间切换背景.

[finish()在 cSimpleModule 类中是 protected,为了防止从其它模块调用.]

例如:

```
mod->callFinish();
```

mod->deleteModule();

4.11.6 创建链接

链接可以使用 cGate 的 connectTo()方法创建.

[前面的 connect()全局函数有两个门不推荐使用,可能从 OMNet++以后发布的版本中移除.]

connect()应该在链接的源门上被调用,将目标门指针作为一个参数:

srcGate->connectTo(destGate);

source 和 destination 单词对应于 NED 文件中的箭头方向.

例如,我们创建两个模块,并且链接它们:

cModuleType *moduleType = findModuleType("TicToc");

cModule *a = modtype->createScheduleInit("a",this);

cModule *b = modtype->createScheduleInit("b",this);

a->gate("out")->connectTo(b->gate("in"));

b->gate("out")->connectTo(a->gate("in"));

connectTo()也有一个可选参数,一个附加的信道对象.信道是 cChannel 的子类.大多数情况下都要使用 cBasicChannel 的实例作为信道 — 这支持延迟,比特错误率和数据速率.信道对象由链接的源门所拥有,在多个链接中不能使用相同的信道对象.

cBasicChannel 有 setDelay(), setError()和 setDatarate()方法来设置信道属性.

一个设置信道延迟的例子:

cBasicChannel *channel = new cBasicChannel("channel");

channel->setDelay(0.001);

a->gate("out")->connectTo(b->gate("in"), channel); // a,b are modules

4.11.7 删除链接

cGate 的 disconnect()方法可以用于删除链接.这个方法在链接的源端调用.如果设置的话,也可以删除链接的信道对象.

srcGate->disconnect();

5 消息

5.1 消息和包

5.1.1 cMessage 类

cMessage 是 OMNet++中的核心类. cMessage 对象和其子类可以模拟许多东西:事件;消息;网络中的包,帧,单元,比特或单个传输;系统中的实体传输等.

属性

cMessage 有许多属性.一些是由仿真内核使用的,其它的为仿真提供方便.一个较全的列表:

- name 属性是一个字符串(const char *),其可由仿真程序员自由使用.在 Tkenv 中消息名出现在许多地方,并且通常对于选择描述名非常有用.这个属性是从 cObject 继承而来.

- message kind 属性支持传送一些消息类型信息.0 和正数可以自由使用.负数由 OMNeT++类库保留使用.
- length 属性(bit) 当消息通过已分配数据速率的链接传输时,用于计算传输的延迟.
- bit error flag 当消息通过分配了比特错误速率(ber)的信道发送消息时,由概率为 $1-(1-ber)^{length}$ 的仿真内核设为 true.
- priority 属性由仿真内核使用,为相同的到达时间值的消息队列(FES)中的消息排序.
- time stamp 属性不由仿真内核使用.可以用于当消息进入队列或重新发送时通知.
- 其它的属性和数据成员使得仿真程序员工作更简单,将在后面讨论: parameter list, encapsulated message, control info 和 context pointer.
- 一些只读的属性存储关于消息的(最后一条)发送/调度的信息: source/destination module and gate, sending (scheduling) 和 arrival time. 当消息在 FES 中时,他们大多数由仿真内核使用,但是当模块接收消息时,信息仍然在消息对象中.

基本用途

cMessage 构造器有许多对数.最常用的是,使用一个 object name (a const char * string) 和 message kind (int)来创建一条消息:

```
cMessage *msg = new cMessage("MessageName", msgKind);
```

两个参数都是可选的,初始化为 null 字符串("")和 0,因此以下的声明都是合法的:

```
cMessage *msg = new cMessage();
cMessage *msg = new cMessage("MessageName");
```

使用消息名是一个好的方法 — 当调试或演示仿真中时特别有用.

消息种类通常初始化为一个符号常量(比如,枚举值)用以表示仿真中的消息对象(比如,数据包,堵塞信号,一个作业等).请使用正数或 0 为表示消息种类 - 负数由仿真内核保留使用.

cMessage 的构造器也可有多个参数(length, priority, bit error flag),但是为了代码的可读性,其最好通过下面描述的 set...()方法来明确设置. Length 和 priority 是整数, bit error flag 是布尔型的.

一旦消息被创建,其成员数据就通过以下的函数来改变:

```
msg->setKind( kind );
msg->setLength( length );
msg->setByteLength( lengthInBytes );
msg->setPriority( priority );
msg->setBitError( err );
msg->setTimestamp();
msg->setTimestamp( simtime );
```

通过这些函数用户可以设置消息种类,消息长度,优先权,错误标志和时间戳. 如果函数setTimeStamp()没有任何参数的话就设置时间戳为当前仿真时间. setByteLength()设置与setLength()相同的长度域,为其参数值的 8 倍.

其值可以通过以下的函数获得:

```
int k       = msg->kind();

int p       = msg->priority();

int l       = msg->length();

int lb      = msg->byteLength();

bool b      = msg->hasBitError();

simtime_t t = msg->timestamp();
```

byteLength()也可读作 length()的长度,但结果除 8 取整.

复制消息

通常需要复制消息(例如,发送或保留一个副本).对其它的 OMNet++对象可以使用相同的方法完成:

```
cMessage *copy = (cMessage *) msg->dup();
```

或

```
cMessage *copy = new cMessage( *msg );
```

这两个是等同的.结果消息是原始消息的精确拷贝,包括消息参数(cPar 或其它对象类型)和封装的消息.

5.1.2 自消息

使用一个消息为自消息

消息通常用于表示模块的内部事件,比如一个周期触发的超时定时器.当在这种情况下使用时,该消息称为自消息—否则自消息就是从 cMessage 或其子类派生出的正常消息.

当通过仿真内核派生出模块的消息时,可以调用 isSelfMessage()来决定是否是自消息,也就是说是否可被 scheduleAt()调度或 send...()方法发送. 如果消息当前被调度,isScheduled()返回 true.一条调度消息也可以被取消(cancelEvent()).

```
bool isSelfMessage();

bool isScheduled();
```

以下的方法返回创建和调度消息的时间以及其到达时间.当消息被调度,到达时间是其将要被传送至模块的时间.

```
simtime_t creationTime()

simtime_t sendingTime();

simtime_t arrivalTime();
```

背景指针

cMessage 是一个 void*指针,由 setContextPointer()和 contextPointer()设置/返回:

```
void *context =...;

msg->setContextPointer( context );

void *context2 = msg->contextPointer();
```

可以由仿真程序员使用.它不由仿真内核使用,它仅作为一个指针来对象(在其上没有内存管理).

意图目的:一个调度许多自消息的模块,当自消息传回模块时需要标识,比如,模块必须决定哪个定时器离开,然后再做什么.背景指针可以指向一个模块的数据结构,可以获得关于事件的足够的背景信息.

5.1.3 模拟包

到达门和时间

以下的方法可以告诉我们消息从哪来,发向哪(如果消息当前被调度或正在进行中,将到到达哪.)

```
int senderModuleId();

int senderGateId();

int arrivalModuleId();

int arrivalGateId();
```

以下的方法是以上基础上的便利函数.

```
cModule *senderModule();

cGate *senderGate();

cGate *arrivalGate();
```

另外的便利函数告诉我们是否消息到达了指定 id 或 name+index 的门.

```
bool arrivedOn(int id);

bool arrivedOn(const char *gname, int gindex=0);
```

以下的方法返回了消息创建时间和最后一次发送,到达的时间.

```
simtime_t creationTime()

simtime_t sendingTime();

simtime_t arrivalTime();
```

控制信息

OMNeT++的主要应用领域是通信网络的仿真.这里,协议层通常实现为交换包的模块.包本身由 cMessage 子类的消息表示.

然而,协议层之间的通信需要发送附在包上的附加信息.例如,TCP 实现向下发送一个 TCP 包至具体的目的 IP 地址和可能的其它参数的 IP.当从 IP 头解封后,IP 通过一个包至 TCP 时,至少要让 TCP 知道源 IP 地址.

这个附加的信息在 OMNet++中表示为 control info 对象.控制信息对象必须是 cPolymorphic 的子类(一个小覆盖区域的没有成员数据的基类),并且附加消息来表示包.为此,cMessage 有以下的方法:

```
void setControlInfo(cPolymorphic *controlInfo);

cPolymorphic *controlInfo();

cPolymorphic *removeControlInfo();
```

当一个”命令”与发送消息相连(诸如 TCP OPEN,SEND,CLOSE 等),消息类别域(cMessage 的 kind(), setKind()方法)需要传送命令代码.当命令不包括数据包时(比如 TCP CLOSE 命令),发送一个假包(空 cMessage).

协议标识

在 OMNeT++模型中,协议类型通常表示为消息子类.例如, 在消息类别值中,IPv6Datagram 类实例表示 IPv6 数据包,EthernetFrame 表示以太网帧. PDU 类型值通常表示为消息类内部的一个域.

C++ dynamic_cast 操作符可以用于决定消息对象是否是一个具体的协议.

```
cMessage *msg = receive();

if (dynamic_cast<IPv6Datagram *>(msg) != NULL)

{

    IPv6Datagram *datagram = (IPv6Datagram *)msg;

    ...

}
```

5.1.4 封装

封装包

当模拟计算机网络的层次协议时通常需要封装消息至另一个中.尽管可以通过添加一些参数列表来封装消息,但有更好的方法.

encapsulate()函数封装消息至另一个中.消息的长度将随着封装的消息的长度而增长.有一例外:当 encapsulating (outer)的消息长度为 0,OMNet++假定其不是一个真的包而是一些波段信号,所以其长度仍为 0.

```
cMessage *userdata = new cMessage("userdata");

userdata->setByteLength(2048);  // 2K

cMessage *tcpseg = new cMessage("tcp");

tcpseg->setByteLength(24);

tcpseg->encapsulate(userdata);

ev << tcpseg->byteLength() << endl; // --> 2048+24 = 2072
```

一条消息一次仅能封装一条消息.第二次 encapsulate()调用将会报错.如果封装的消息不是模块所拥有的也会出错.

可以由 decapsulate()得到封装的消息:

```
cMessage *userdata = tcpseg->decapsulate();
```

decapsulate()会相应地减少消息的长度,除非其为 0.如果长度出现负数,则发生错误.

encapsulatedMsg() 返回一个指向解封消息的指针,或者如果没有消息被解封则为 NULL.

引用计数[New!]

从 3.2 版本以后,OMNet++实现了封装消息的引用计数,即如果 dup()一条包含封装的消息,那么封装消息将不能被复制,仅仅引用计数增加.封装消息的复制被延迟至实际调用 decapsulate().如果外部的消息在还没有调用 decapsulate()方法时被删除,那么封装消息的引用计数可以很方便在减少.当其引用计数为 0 时,封装消息被删除.

引用计数可以显著地提高性能,特别在 LAN 和无线网络中.例如,在广播 LAN 或 WLAN 的仿真中,如果 MAC 地址与最初的不符,IP,TCP 以及高层的包不能被复制(然后丢弃没有使用).

引用计数透明地工作.然而,有一层意思:对封装入另一条消息的消息,必须不能进行任何改变!即 encapsulatedMsg()应该视为返回一个只读对象的指针(事实上返回一个常量指针),那是因为:封装消息可能在多条消息之间共享,改变会影响其它消息.

封装多条消息

cMessage 并不支持直接添加多条消息至一个消息对象,但可以通过 cMessage 的子类添加需要的函数(推荐使用消息定义语法[5.2],消息定制在本章后面描述—可以节省一些工作.)

可以将消息存储至一固定大小或动态分配的数组,或者可以使用 STL 类像 std::vector 或 std::list.有一个不期望的额外的特点:消息类需要有插入消息的所有权,当它们从消息中移除时释放.这些通过 take() 和 drop()方法完成.让我们看一例子,假定添加到类中一个 std::list 成员中的消息存储在消息指针中:

```
void MessageBundleMessage::insertMessage(cMessage *msg)
{
    take(msg);  // take ownership
    messages.push_back(msg);  // store pointer
}
void MessageBundleMessage::removeMessage(cMessage *msg)
{
    messages.remove(msg);  // remove pointer
    drop(msg);  // release ownership
}
```

也需要提供一个=()操作符方法来确定消息对象可以被完全地拷贝和复制— 这通常在仿真中是需要的(广播和中继!),[6.11]节包括派生新类时需要考虑的更多的信息.

5.1.5 附加参数和对象

如果要添加参数或对象至消息,首选的方法就是通过消息定义,在[5.2]节描述.

附加对象

cMessage 类有一个内部 cArray 对象,可以传送对象.仅有从 cObject 派送的对象(大多数 OMNet++类都是)可以被附加. addObject(), getObject(), hasObject(), removeObject()方法使用对象为关键字传送.例:

```
cLongHistogram *pklenDistr = new cLongHistogram("pklenDistr");
```

```
msg->addObject( pklenDistr );
...
if (msg->hasObject("pklenDistr"))
{
   cLongHistogram *pklenDistr =
       (cLongHistogram *) msg->getObject("pklenDistr");
   ...
}
```

应该考虑附加对象的名称彼此之间或 cPar 参数名之间不冲突(见下一节).如果不附加任何东西至消息且不调用 parList()函数,那么内部的 cArray 将不会被创建.这样节省了存储和执行时间.

可以附加非对象类型(或非 cObject 对象)至消息,通过使用(cPar 的 void* pointer 'P')类型(见后面的 cPar 描述).例如:

```
struct conn_t *conn = new conn_t; // conn_t is a C struct
msg->addPar("conn") = (void *) conn;
msg->par("conn").configPointer(NULL,NULL,sizeof(struct conn_t));
```

附加参数

具有新数据域的消息扩展的首选方法是使用消息定义(见[5.2]).

旧的,不推荐使用的添加新域至消息的方法是通过附加一个 cPar 对象.这种方法有许多的缺点,最大的缺点就是消耗大量的内在和执行时间.cPar 本身是重型且相当复杂的对象.据报告,使用 cPar 消息参数会占用大部分的执行时间,有时差不多达到 80%.使用 cPar 也易于出错因为 cPar 对象必须动态添加并且分别添加至每个对象.相反,子类的好处是静态类型检验:如果在 C++代码中误输了一个域名,编译器可以检测出错误.

然而,如果仍然需要使用 cPar,这里简短地概述一个如何完成.使用 addPar()成员函数添加一个新的参数至消息,然后使用 par()成员函数返回一个参数对象的引用. hasPar()得知消息是否有一个给定的参数.消息参数通过索引在参数数组中访问. findPar()函数返回参数索引,如果参数没有发现刚返回-1.参数然后可以通过重载的 par()函数访问.

例:

```
msg->addPar("destAddr");
msg->par("destAddr") = 168;
...
long destAddr = msg->par("destAddr");
```

5.2 消息定义

5.2.1 介绍

实际上,需要添加变量域至 cMessage 使其可用.例如,如果要模拟通信网络中的包,需要在消息对象中存储协议头文件.由于仿真库用 C++编写, 扩展 cMessage 的基本方法就是通过子类.

然而,由于对每个域都至少要写三样事(私有数据成员,getter,setter 方法),最终的类必须整合仿真框架,写所需的 C++代码是乏味,耗时的任务.

OMNeT++提供了一个更方便的途径称为消息定义.消息定义提供了非常紧凑的语法来描述消息内容.C++代码从消息定义中自动产生,节省了大量的输入.

关于代码产生器的公共的抱怨通常是丢失了灵活性:如果对于类似如何产生代码有不同的看法,那么你所能做就很少.然而在 OMNet++中,不需要有任何担心:你可以定制类产生至你所喜欢的任何程度.即使你决定定制重型的产生类,消息定义仍然节省了你大量的手动工作.

在 OMNet++中消息子类的特征仍然有些实验性,意为:

- 消息描述语法和特性在未来会基于回馈信息有一点改变.
- 将消息描述转换成 C++的编译器是一个 perl 脚本 opp_msgc.这是一个暂时的解决方案,直至基于 C++的 nedtool 完成.

用于添加消息参数的子类方法,原先是由 Nimrod Mesika 建议的.

第一个消息类

让我们开始一个简单的例子.假定,需要消息对象来传送源和目的地址以及跳数.可以写一个有以下内容的 mypacket.msg 文件:

```
message MyPacket

{

    fields:

       int srcAddress;

       int destAddress;

       int hops = 32;

};
```

子类消息编译器的任务是产生可以从模型中使用的 C++类以及一个”映射”类允许 Tkenv 检查这些数据结构.

如果用消息子类编译器处理 mypacket.msg,那么将创建以下的文件: mypacket_m.h 和 mypacket_m.cc. mypacket_m.h 包括 MyPacket C++类的声明,应该被包括入需要处理 MyPacket 对象的 C++源码中.

产生的 mypacket_m.h 将包含以下的类声明:

```
class MyPacket : public cMessage {

    ...

    virtual int getSrcAddress() const;

    virtual void setSrcAddress(int srcAddress);

    ...

};
```

所以在 C++文件中,可以类似地使用 MyPacket 类:

```
#include "mypacket_m.h"
```

...

```
MyPacket *pkt = new MyPacket("pkt");

pkt->setSrcAddress( localAddr );
```

...

mypacket_m.cc 文件包含产生 MyPacket 类的实现以及允许在 Tkenv GUI 中检查这些数据结构的”映射”代码. mypacket_m.cc 应该编译链接至仿真中.(如果使用 opp_makemake 工具来产生 makefiles,后者会自动获得.)

消息子类不是什么?

对于消息定义的目的和概念会有一些混淆,因此在这里我们正确地理解它们.

它不是:

- ...一个企图复制具有另外的语法的 C++函数.不寻找复杂的 C++类型,模板,条件编辑等.它仅定义了数据(或一个访问数据的接口)—没有任何的活动行为.
- ... 一个普通的类产生器.即定义输入消息内的消息内容和数据结构.方法定义不支持这个目的.也可能使用语法来产生简单模块内使用的类和结构 体,但这不是一个好方法.

这个目的是定义消息的接口(getter/setter 方法)而不是它们在 C++中实现.提供了一个简单易懂的域实现—如果要在内部的多个域用不同的表示,那么可以通过定制类.

可能会问一些问题:

- 为什么不支持 std::vector 和其它的 STL 类?消息定义关注于类接口(getter/setter 方法),将其实现留给用户 — 所以我们可以使用 std::vector 实现域(动态数组域).(这完全符合 STL 理念 — 其为 C++程序设计了接口).
- 为什么其支持 C++数据类型,但并不支持 octets, bytes, bits,等?这限制网络中消息定义的范围, OMNeT++还支持其它的应用领域.而且,所支持的必需的概念集可能没有限制,其总是技术服务新的数据类型.
- 为什么没有嵌入类?因为其并不与上述原则冲突,它可能在以后被补充.

下面的部分详细地描述了消息语法和特征.

5.2.2 枚举声明

enum {..}产生一个 C++枚举,另外创建一个存储常量的文本表示的对象.后者使得可以在 Tkenv 中显示符号名.例如:

```
enum ProtocolTypes

{

   IP = 1;

   TCP = 2;

};
```

枚举值必须惟一.

5.2.3 消息声明

基本使用

可以使用以下语法描述消息：

```
message FooPacket

{

    fields:

        int sourceAddress;

        int destAddress;

        bool hasPayload;

};
```

处理具有消息编译器的声明会产生一个 FooPacket 类的 C++头文件. FooPacket 是 cMessage 的子类.

对以上描述的每个子域,产生的类都有一 protected 数据成员,一个 getter 和一个 setter 方法.方法名以 get 和 set 开头,跟着域名,其首字母大写.因此 FooPacket 将包含以下的方法:

```
virtual int getSourceAddress() const;

virtual void setSourceAddress(int sourceAddress);

virtual int getDestAddress() const;

virtual void setDestAddress(int destAddress);

virtual bool getHasPayload() const;

virtual void setHasPayload(bool hasPayload);
```

注意方法都声明为 virtual,在子类中有很重要的好处.

将产生两个构造器:一个是任意的对象名和(对台戏 cMessage 子类)消息种类和一个复制构造器:

```
FooPacket(const char *name=NULL, int kind=0);

FooPacket(const FooPacket& other);
```

相应在也产生分配操作符(operator=())和 dup()方法.

域的数据类型不限制为 int 和 bool 型.可以使用以下的基本类型(如在 C++语言中定义的基本类型):

- bool
- char, unsigned char
- short, unsigned short
- int, unsigned int
- long, unsigned long
- double

域值初始化为 0.

初始值

可以使用以下的语法来初始化域值：

```
message FooPacket
{
   fields:
        int sourceAddress = 0;
        int destAddress = 0;
        bool hasPayload = false;
};
```

初始化代码将放在产生类的构造中.

枚举声明

可以声明为一个从枚举中获得值的 int(或其它整数类型)域.消息编译器然后产生代码允许 Tkenv 显示域的符号值.

例如：

```
message FooPacket
{
  fields:
      int payloadType enum(PayloadTypes);
};
```

枚举必须在消息文件中单个声明.

固定大小数组

可以指定固定大小的数组：

```
message FooPacket
{
    fields:
        long route[4];
};
```

产生的 getter 和 aetter 方法将一个额外的 k 参数,表示数组的索引：

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, l ong route);
```

如果调用一个带越界索引的方法,将会抛出异常.

动态数组

如果数组大小事先不知道,可以声明域为动态数组:

```
message FooPacket
{
   fields:
       long route[];
};
```

在这种情况下,产生的类除了 getter 和 setter 方法外,将有两个额外的方法:一个设置数组大小,另一个返回当前数组的大小.

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, long route);
virtual unsigned getRouteArraySize() const;
virtual void setRouteArraySize(unsigned n);
```

set...ArraySize()方法在内部分配一个新的数组.数组现有的值将被存储(拷贝至新数组.)

缺省的数组大小是 0.这表示需要在开始填充数组元素之前调用 set...ArraySize().

字符串成员

可以用以下语法声明字符串值的域:

```
message FooPacket
{
   fields:
       string hostName;
};
```

产生的 getter 和 setter 方法将返回和传递 const char* 指针:

```
virtual const char *getHostName() const;
virtual void setHostName(const char *hostName);
```

产生的对象将有其自身的字符串拷贝.

注意:一个字符串成员与字符数组不同,其作为另一种类型的数组.例如,

```
message FooPacket
{
   fields:
       char chars[10];
};
```

将会产生以下的方法:

```
virtual char getChars(unsigned k);
```

```
virtual void setChars(unsigned k, char a);
```

5.2.4 继承,合成

正如我们讨论的如何添加基本类型(int, double, char, ...)的域至 cMessage.这对于简单模块已经足够,但是如果有更多复杂的模块,就可能需要:

- 设置一个消息(包)类继承,即不仅是 cMessage 的子类还有从自身拥有的消息类中.
- 域不仅使用基本类型,也有结构体,类或类型定义.有时可以使用 C++类型表示在已存在的头文件中,还有时候可以使用由消息编译器产生的结构 体或类,从而可以方便 Tkenv 的检查.

以下部分描述了如何完成这些工作

消息类之间的继承

缺省情况下，消息是 cMessage 的子类。然而，可以使用扩展的关键字来明确指定基类：

```
message FooPacket extends FooBase
{
    fields:
        ...
};
```

以上的例子中，产生的 C++代码类似:

```
class FooPacket : public FooBase { ... };
```

也可以继承结构体和类(详见下一节)

类定义

到现在为止,我们使用了消息关键字来定义类,也就是说无论是直接还是间接的,基类都是 cMessage.

但是作为复杂消息的一部分,需要一些结构体和其它的类(cObject 中的根或非根)为构建块.类可以使用 class 关键字来创建;结构体将在下一节描述.

定义类的语法与消息定义类似,仅仅是用关键字取代了消息.

略有不同的是代码产生的类是 cObject 中的根,而其它则不是.如果不扩展的话,产生的代码将不能从 cObject 派生,因而就没有 name(), className()等方法.为了创建一个有这些方法的类,必须明确在写上 extends cObject.

```
class MyClass extends cObject
{
    fields:
        ...
};
```

定义简单的 C 结构体

可以在消息类中定义一个 C 式的结构体用作一个域,C 式的就是仅包括数据没有方法.(实际上,在 C++结构体中可以有方法,通常其可以完成任何类能完成的事情.)

语法类似于定义消息:

```
struct MyStruct

{

    fields:

        char array[10];

        short version;

};
```

然而产生的代码是不同的.产生的结构体没有 getter 或 setter 方法,取代的域由公共数据成品表示.对于以上的定义,可以产生以下的代码:

```
// generated C++

struct MyStruct

{

    char array[10];

    short version;

};
```

一个结构体可以有基本类型或其它的结构体作为其域.它不能有 string 或类作为其域.

结构体支持继承:

```
struct Base

{

    ...

};

struct MyStruct extends Base

{

    ...

};
```

但是由于一个结构体没有成员函数,其有一些限制:

- 不支持动态数组(不放置任何数组分配代码)
- 代沟或抽象域(见后面)不能使用,由于它们不能构建虚拟函数.

使用结构体和类作为域

除了基本类型也可以使用其它的结构体或对象作为一个域.例如,如果有一个名为 IPAddress 的结构体,可以如下写:

```
message FooPacket
```

```
{
   fields:
       IPAddress src;
};
```

消息编译器发表事先知道 IPAddress 结构;即在消息描述文件中其必须被在之前被定义为一个结构体或类,或者它必须是在其头文件中通过 cplusplus {{...}}和它的类型声明过的 C++类型(见声明 C++类型).

产生的类将包含一个 IPAddress 成员数据(即不是一个指向 IPAddress 的指针).会产生下面的 getter 和 setter 方法:

```
virtual const IPAddress& getSrc() const;
virtual void setSrc(const IPAddress& src);
```

指针

还未支持

5.2.5 使用现有的 C++类型

声明 C++类型

如果想要在消息定义中使用自己的类型(一个类,结构体或类型定义,在 C++头文件中声明),需要声明这些类型至消息编译器.也必须确定头文件包括进 generated _m.h 文件,从而 C++编译器可以编译.

假定你有一个 IPAddress 结构,在一个 ipaddress.h 中定义的:

```
// ipaddress.h
struct IPAddress {
    int byte0, byte1, byte2, byte3;
};
```

为了能够在消息定义中使用 IPAddress,消息文件(比如 foopacket.msg)应该包含以下的行:

```
cplusplus {{
#include "ipaddress.h"
}};
struct IPAddress;
```

前三行的作用是简单的#include 声明将被拷贝入产生的 foopacket_m.h 文件,让 C++编译器知道 IPAddress 类.消息编译器本身不会去试着弄明白 cplusplus {{ ... }}中主体所表示的文本.

下面一行 struct IPAddress,告诉消息编译器 IPAddress 是一个 C++结构体.这条消息将影响代码的产生.

类可以使用 class 关键字声明:

```
class cSubQueue;
```

以上的语法假定类是从 cObject 直接或间接派生而来.如果不是的话,需要使用 noncobject 关键字:

```
class noncobject IPAddress;
```

从 cObject 派生的类与不是从 cObject 派生的类之间的区别很重要,因为不同的地方生成的代码不同.如果偶然忘记了 noncobject 关键字(从而误导消息编译器认为该类是 cObject 的根,然而事实并不是),那么将会在生成头文件时得到一个 C++编译错误.

5.2.6 定制生成的类

代沟模式

有时需要生成代码来完成更多的事情或不同于消息编译器生成的版本的事实 .例如,当设计一个名为 payloadLength 的整型域名,也可能需要调整包的长度.也就是说,以下缺省的 setPayloadLength()方法是不合适的:

```
void FooPacket::setPayloadLength(int payloadLength)
{
    this->payloadLength = payloadLength;
}
```

它需要类似如下:

```
void FooPacket::setPayloadLength(int payloadLength)
{
    int diff = payloadLength - this->payloadLength;
    this->payloadLength = payloadLength;
    setLength(length() + diff);
}
```

共同认为生成的代码其最大的缺点就是很难或根本不可能实现这样的愿望.手工编辑生成的文件，是没意义的，因为在代码生成周期里,它们将被覆盖，改变会丢失。

然而,面向对象的设计提供一个解决方法.一个生成的类可以通过其简单地定制子类,和重新定义两者的方法要与其产生的版本不同。这种做法被称为代沟的设计模式。它符合以下语法：

```
message FooPacket
{
   properties:
       customize = true;
   fields:
       int payloadLength;
};
```

消息声明的属性部分包括影响如何生成类似代码的 Meta 信息.定制属性可以使用代沟模式.

如果处理以上的消息编译器代码,生成的代码将包含 FooPacket_Base 类代替 FooPacket.这个思想是你必须从 FooPacket_Base 的子类来产生 FooPacket,重新定义需要的方法来完成定制.

```
class FooPacket_Base : public cMessage
{
  protected:
    int src;
    // make constructors protected to avoid instantiation
    FooPacket_Base(const char *name=NULL);
    FooPacket_Base(const FooPacket_Base& other);
  public:
    ...
    virtual int getSrc() const;
    virtual void setSrc(int src);
};
```

由于不是FooPacket_Base 的每个部分都可以预先生成,对于FooPacket 必须写最小数量的代码,比如:构造器不能继承.这个最小代码如下(也可以发现其产生的 C++头文件,作为注释):

```
class FooPacket : public FooPacket_Base
{
  public:
    FooPacket(const char *name=NULL) : FooPacket_Base(name) {}
    FooPacket(const FooPacket& other) : FooPacket_Base(other) {}
    FooPacket& operator=(const FooPacket& other)
        {FooPacket_Base::operator=(other); return *this;}
    virtual cPolymorphic *dup() {return new FooPacket(*this);}
};
Register_Class(FooPacket);
```

注意重新定义 dup()并且分配操作符(operator=())很重要.

因此,返回至我们原始例子的关于负载的长度会影响包的长度,代码将写成如下:

```
class FooPacket : public FooPacket_Base
{
    // here come the mandatory methods: constructor,
    // copy contructor, operator=(), dup()
    // ...
```

```
    virtual void setPayloadLength(int newlength);
}
void FooPacket::setPayloadLength(int newlength)
{
    // adjust message length
    setLength(length()-getPayloadLength()+newlength);
    // set the new length
    FooPacket_Base::setPayloadLength(newlength);
}
```

抽象域

抽象域的目的是不考虑存储在类中的值,在 Tkenv 中仍然有利于查看.

例如,当你想存储一个 bit 域,并且你想显示 bit 为单个包域的情况下.其对于计算域的实现很有用.

可以使用以下语法声明任何域为抽象的:

```
message FooPacket
{
   properties:
       customize = true;
   fields:
       abstract bool urgentBit;
};
```

对于抽象域,消息编译器不产生数据成员,产生的 getter 和 setter 方法将是纯虚函数.

```
virtual bool getUrgentBit()  const = 0;
virtual void setUrgentBit(bool urgentBit) = 0;
```

通常想结合抽象域和代沟模式一起使用,因此可以直接重新定义抽象(纯虚)方法并且提供实现.

5.2.7 在消息类中使用 STL

在消息类中,可以要使用 STL 向量或栈类.这可能要使用抽象域.毕竟,向量和栈都是一个序列的表示—同一抽象为动态大小向量.即,可以声明一个域为 abstract T fld[],使用 vector<T>提供一个基本的实现.也可以添加方法至消息类中,调用基本的 STL 对象上的 push_back(), push(), pop()等

看以下的声明:

```
struct Item
{
```

```
    fields:
        int a;
        double b;
}
message STLMessage
{
   properties:
       customize=true;
   fields:
       abstract Item foo[]; // will use vector<Item>
       abstract Item bar[]; // will use stack<Item>
}
```

如果编译以上的代码,在生成的代码中仅会发现一对 foo 和 bar 的抽象方法,没有成员数据或其它信息.你也可以随意实现.你可以写以下的 C++ 文件,然后使用 std::vector 和 std::stack:实现 foo 和 bar.

```
#include <vector>
#include <stack>
#include "stlmessage_m.h"
class STLMessage : public STLMessage_Base
{
  protected:
    std::vector<Item> foo;
    std::stack<Item> bar;
  public:
    STLMessage(const char *name=NULL, int kind=0) : STLMessage_Base(name,kind) {}
    STLMessage(const STLMessage& other) : STLMessage_Base(other.name())
{operator=(other);}
    STLMessage& operator=(const STLMessage& other) {
        if (&other==this) return *this;
        STLMessage_Base::operator=(other);
        foo = other.foo;
        bar = other.bar;
        return *this;
    }
```

```
    virtual cPolymorphic *dup() {return new STLMessage(*this);}
    // foo methods
    virtual void setFooArraySize(unsigned int size) {}
    virtual unsigned int getFooArraySize() const {return foo.size();}
    virtual Item& getFoo(unsigned int k) {return foo[k];}
    virtual void setFoo(unsigned int k, const Item& afoo) {foo[k]=afoo;}
    virtual void addToFoo(const Item& afoo) {foo.push_back(afoo);}
    // bar methods
    virtual void setBarArraySize(unsigned int size) {}
    virtual unsigned int getBarArraySize() const {return bar.size();}
    virtual Item& getBar(unsigned int k) {throw new cRuntimeException("sorry");}
    virtual void setBar(unsigned int k, const Item& bar) {throw new
cRuntimeException("sorry");}
    virtual void barPush(const Item& abar) {bar.push(abar);}
    virtual void barPop() {bar.pop();}
    virtual Item& barTop() {return bar.top();}
};
Register_Class(STLMessage);
```

一些附加的注意事项:

1. setFooArraySize(), setBarArraySize() 是多余的.
2. getBar(int k)不能直接实现(std::stack 不支持通过索引访问成员).其仍然通过重复使用 STL 这种低效的方法实现,因为仅有 Tkenv 会调用这个函数,因为效率不见得是主要问题.
3. setBar(int k, const Item&)不能被实现,但是这不是显著的问题.当尝试改变域值时, Tkenv 错误诊断器将实现异常.

对 STL 向量/栈不能直接查看会觉得遗憾.你可以查看它们(通过在类中添加 vector<Item>& getFoo() {return foo;}方法),但是这并不是一个好的方法.STL 本身的目的是设计一个低层方法,来提供 C++设计的接口,STL 更好地用于表示其它类的内部数据.

5.2.8 概述

这节我们试着概述可能的事.

可以生成:

- 来自 cObject 的类
- 消息(缺省基类为 cMessage)
- 非来自 cObject 的类
- 普通 C 结构体

The following data types are supported for fields:

域支持以下的数据类型:

- 基本类型: bool, char, short, int, long, unsigned short, unsigned int, unsigned long, double
- string,动态分配字符串,表示为 const char *
- 以上类型的固定大小的数组
- 结构体,类(来自 cObject 和非来自 cObject),消息语法或在 C++代码外部声明.
- 以上类型大小变化的数组(存储为动态分配数组及整型表示的数组大小).

其它特性:

- 域初始化为 0 (除了结构体成员)
- 域可以指定初始化(除了结构体成员)
- 指定枚举变量的完整类型
- 继承
- 通过子类定制生成类(代沟模式)
- 抽象域(用于非标准存储和计算域)

生成代码 (所有生成的方法都是虚方法.尽管在下表并未写出):

Field declaration	Generated code
primitive types double field;	double getField(); void setField(double d);
string type string field;	const char *getField(); void setField(const char *);

fixed-size arrays
double field[4];
	double getField(unsigned k);
	void setField(unsigned k, double d);
	unsigned getFieldArraySize();

dynamic arrays
double field[];
	void setFieldArraySize(unsigned n);
	unsigned getFieldArraySize();
	double getField(unsigned k);
	void setField(unsigned k, double d);

customized class
class Foo {
properties:
customize=true;
	class Foo_Base { ... };
	必须写成
	class Foo : public Foo_Base {
	...
	};

abstract fields
 	
abstract double field;
	double getField() = 0;
	
	void setField(double d) = 0;

仿真示例

许多仿真实例(令牌环网, Dyna,超立方体)使用消息定义.例如在 Dyna 中将会发现:

- dynapacket.msg 定义了 DynaPacket 和 DynaDataPacket;
- dynapacket_m.h 和 dynapacket_m.cc 是由子类消息编译生成的,他们包括生成 DynaPacket 和 DynaDataPacket C++类(加上 Tkenv 检验代码);
- 其它模拟文件(client.cc, server.cc, ...)使用生成的信息类.

5.2.9 还有什么可以生成代码?

除了消息类和它的实现,消息编译器也生成映射代码,使其可以在 Tkenv 中检查消息内容.为了阐述其必需性,假定你手动从 cMessage 子类得到一个新的消息类.可以写成如下:

[注意代码仅用于例证.在实际代码中, freq 和 power 应为私有成员,getter/setter 方法应该存在来访问他们.以上的类定义也忽略了许多应该写的成员函数(构造器,分配操作符等).]

```
class RadioMsg : public cMessage

{

  public:

    int freq;

    double power;

    ...

};
```

现在可以简单模块中使用 RadioMsg:

```
RadioMsg *msg = new RadioMsg();

msg->freq = 1;

msg->power = 10.0;

...
```

需要注意这种解决方法,在使用 Tkenv 调试时有一个缺点.当基于 cPar 的消息参数在消息查看窗口查看时,通过子类添加的域并不出现.因为 Tkenv 仅是仿真程序中的另一个 C++库,并不知道 C++实例变量.这个问题在 Tkenv 中不能完全解决,因为 C++并不像 JAVA 一样支持”映射”(在运行时提取类信息).

然而有一个解决方法:可以提供 Tkenv 忽略关于新类的”映射”信息.映射信息可能采取的形式是一个单独的 C++类,其方法返回关于 RadioMsg 域的信息.该描述类可以如下:

```
class RadioMsgDescriptor : public Descriptor

{

  public:
```

```
    virtual int getFieldCount() {return 2;}
    virtual const char *getFieldName(int k) {
        const char *fieldname[] = {"freq", "power";}
        if (k<0 || k>=2) return NULL;
        return fieldname[k];
    }
    virtual double getFieldAsDouble(RadioMsg *msg, int k) {
        if (k==0) return msg->freq;
        if (k==1) return msg->power;
        return 0.0; // not found
    }
    //...
};
```

然后必须通知 Tkenv 存在 RadioMsgDescriptor,当 Tkenv 寻找 RadioMsg 类型时应该使用(根据目前的实现,对象的 className()方法返回"RadioMsg").因此当要在仿真中检查 RadioMsg 时, Tkenv 可以使用 RadioMsgDescriptor 来提取和显示 freq 和 power 的变量值.

实际的实现比这个复杂,但不是太复杂.

6 仿真库

OMNeT++有大量的 C++类库,可以使用它们来实现简单模块.类库一部分已经在前面的章节提及过:

- 消息类 cMessage (第[5]章)
- 发送和接收消息,调用和取消事件,终止模块或住址([4.6]节)
- 通过 cModule 成员函数访问模块门和参数([4.7]和[4.8]节)
- 在网络中访问其它模块([4.9]节)
- 动态模块创建([4.11]节)

本章讨论其余的仿真库:

- 生成随机数: normal(), exponential()等.
- 模块参数: cPar 类
- 在容器中存储数据: cArray 和 cQueue 类
- 路由支持和网络拓扑的发现: cTopology 类
- 记录统计信息文件: cOutVector 类
- 收集简单统计信息: cStdDev 和 cWeightedStddev 类
- 分布估计: cLongHistogram, cDoubleHistogram, cVarHistogram, cPSquare, cKSplit 类

- 使变量可以图形化用户界面中检查(Tkenv): WATCH() 宏
- 发送调度输出和提示用户输入的图形化用户界面(Tkenv):ev object (cEnvir class)

6.1 类库公约

6.1.1 基类

OMNeT++仿真库中的类都是从 cObject 派生而来. cObject 的功能和公约:

- name 属性
- className() 成员和其它成员函数给出了关于对象的信息文本信息
- 对于 assignment, copying, duplicating 对象的公约
- 从 cObject 派生的对容器的所有权控制.
- 支持检查(traversing)对象树
- 支持在用户的图形化界面(Tkenv)中检查对象
- 类继承并且重新定义了 cObject 成员函数,在下面我们将讨论一些实践中重要的类.

6.1.2 Setting 和 getting 属性

成员函数设置和查询对象属性遵循一贯的命名. setter 成员函数有 setFoo(…)形式,相应的 getter 命名为 foo().(get 动词是 JAVA 中发明的,为了简洁忽略了一些其它的库.)例如, cMessage 类的 length 属性可以进行类似的设置和读取:

```
msg->setLength(1024);

length = msg->length();
```

6.1.3 className()

对每个类,className()成员函数都返回一个字符串式的类名:

```
const char *classname = msg->className(); // returns "cMessage"
```

6.1.4 Name 属性

一个对象可以分配一个 name(字符串).name 字符串是每个类构造器的第一个参数,缺省情况是 NULL(没有 name 字符串).例如:

```
cMessage *timeoutMsg = new cMessage("timeout");
```

也可以在对象创建之后设置 name:

```
timeoutMsg->setName("timeout");
```

可以类似得到一个内部存储的 name 字符串的指针:

```
const char *name = timeoutMsg->name(); // --> "timeout"
```

由于方便和效率的原因,库对象认为空字符串"" 和 NULL 是等同的.即"" 被存储为 NULL, 但是返回为"".如果创建一个 name 字符串为 NULL 或 ""的消息对象,将存储为 NULL,name() 将返回一个指向静态""的指针.

```
cMessage *msg = new cMessage(NULL, <additional args>);

const char *str = msg->name(); // --> returns ""
```

6.1.5 fullName()和 fullPath()

对象有两个以上成员函数,返回基于对象名的字符串: fullName() 和 fullPath().对门和模块,其中为门或模块向量的一部分, fullName()返回中括号内索引的名称.即对子模块向量 node[10]中一个模块向量 node[3], name()返回"node", fullName()返回"node[3]".对其它对象 fullName()与 name()相同.

fullPath()返回 fullName(),预先考虑父对象和自身对象的 fullPath(),用点隔开.即,如果上面的 node[3]模块是在复合模块"net.subnet1"里,其 fullPath()方法将返回 "net.subnet1.node[3]".

```
ev << this->name();     // --> "node"

ev << this->fullName(); // --> "node[3]"

ev << this->fullPath(); // --> "net.subnet1.node[3]"
```

className(), fullName() 和 fullPath() 大量地用于图形化运行时环境 Tkenv 中,也会出现错误消息.

name()和 fullName()返回 const char * pointers, fullPath()返回 std::string.使用 ev<< 时就没有区别,但当 fullPath()在 sprintf()中使用"%s"参数时,须写成 fullPath().c_str()

```
char buf[100];

sprintf("msg is '%80s'", msg->fullPath().c_str()); // note c_str()
```

6.1.6 拷贝和复制对象

dup()成员函数创建对象的原样副本,如果需要复制也包含对象.这在消息对象的情况下,特别有用. dup()返回一个 cObject*类型指针,所以需要转换相应的类型:

```
cMessage *copyMsg = (cMessage *) msg->dup();
```

dup()通过调用复制构造器工作,反过来又依赖于对象之间的分配操作符. operator=()可以用于拷贝对象的内容至另一个相同类型的对象.这是一个深度拷贝:包含在对象中的对象,如果需要的话,也会复制. operator=()不复制名称字符串—这个任务由复制构造器完成.

6.1.7 Iterators

库中有许多的容器类(cQueue, cArray 等).对于许多的类,都有一个相应的 iterator 类,可以使用循环遍历容器中存储的对象.

例如:

```
cQueue queue;

//..

for (cQueue::Iterator queueIter(queue); !queueIter.end(); queueIter++)

{

    cObject *containedObject = queueIter();

}
```

6.1.8 错误处理

当库对象检测到一个错误状态时,会抛一个 C++异常.这个异常然后被仿真环境捕捉,提出一个错误对话框或显示错误消息.

有时它可以用于在出错的地方停止仿真(在异常抛出之前),使用一个 C++调试器来查看堆栈跟踪和变量查看.可以进入调试上的错误 INI 文件, 使你可以核查, 见[8.2.6]节.

如果在代码中检测到一个错误状态,可以使用 opp_error()函数弹出错误消息,停止仿真.opp_error()的参数列表类似 printf():第一个参数是一个格式字符串,可以包含"%s", "%d"等,使用后面的参数填充.

例如:

```
if (msg->controlInfo()==NULL)
    opp_error("message (%s)%s has no control info attached",
              msg->className(), msg->name());
```

6.2 模块日志

这里我们介绍的例子代码中,广泛在使用了日志的特性.

ev 对象表示用户的仿真界面.可以发送具有 C++-式的输出操作符的调试输出信息至 ev:

```
ev << "packet received, sequence number is " << seqNum << endl;
ev << "queue full, discarding packet\n";
```

另一种方法是 ev.printf():

```
ev.printf("packet received, sequence number is %d\n", seqNum);
```

消息显示给用户的具体方法是依赖于用户界面.在命令行用户界面 (Cmdenv) 中,其简单地显示标准的输出.(这个输出可以从 omnetpp.ini 禁止,因此当没有必要的时候,它不会减慢仿真.)在运行时 GUI Tkenv 中,可以为每个模块打开一个文本输出窗口.不推荐使用 printf()或 cout 来输出消息—ev 输出可以从 omnetpp.ini 中更好的控制,并且更方便使用 Tkenv 查看.

通过在任何地方日志,声明状态是否实际输出显示或记录,来节省 CPU 循环.当 ev<< 输出禁止时,ev.disabled()调用返回 true,诸如在 Tkenv 或 Cmdenv 中的表示模式.因此,可以写成如下的代码:

```
if (!ev.disabled())
    ev << "Packet " << msg->name() << " received\n";
```

同样问题的更复杂的实现是定义一个 EV 宏,可以用于取代 ev 用于日志声明.定义:

```
#define EV  ev.disabled()?std::cout:ev
```

在这之后,可以简单地使用 EV<<来代替 ev<<.

```
EV << "Packet " << msg->name() << " received\n";
```

稍微有些技巧的 EV 定义使得使用<<操作符的绑定比?:更宽松.

6.3 仿真时间转换

仿真时间由 simtime_t 类型表示,其类型定义为双精度.OMNet++提供了有用的函数,转换 simtime_t 为一个可打印的字符串("3s 130ms 230us"),反之亦然.

simtimeToStr()函数将 simtime_t(第一个接收的参数)转换为一个文本形式.结果放入给定的第二个参数,字符数组中.如果第二个参数缺省或为 NULL, simtimeToStr()将结果放入每个调用都覆盖的静态缓存中.例如:

批注 **[z13]:** 将时间转换为字符串

```
char buf[32];

ev.printf("t1=%s, t2=%s\n", simtimeToStr(t1), simTimeToStr(t2,buf));
```

simtimeToStrShort()类似于 simtimeToStr(),但是其输出更简明.

simtimeToStr()函数接收一个字符串,解析为一个指定的时间,返回一个 simtime_t.如果字符串不能完全解释,则返回-1.

```
simtime_t t = strToSimtime("30s 152ms");
```

另一种不同的情况,如果时间字符串是一较大字符串的子串,可以使用 strToSimtime0().它使用一个 char* (char*&)的引用来代替 char*作为第一个参数.函数设置指针至时间字符串中不能被解释的那部分的第一个字符,并且返回值.它不会返回-1;如果在字符串开始时没有任何类似仿真时间的字符,则返回 0.

```
const char *s = "30s 152ms and something extra";

simtime_t t = strToSimtime0(s); // now s points to "and something extra"
```

6.4 产生随机数

随机数在仿真中从来都不是随意的.相反,他们使用 deteministic 算法产生。算法有一个 seed（种子）值，在其上进行 deterministic 计算，产生一个“随机”值和下一个种子。这些算法和它们的实现称为随机数产生器或 RNG，或者有时是伪随机数产生品或 PRNG 来显示其 deterministic 特性。

[也有真实的随机数,见 http://www.random.org/, http://www.comscire.com,或 Linux /dev/random 驱动.对于一个非随机数,尝试 www.noentropy.net.]

从相同的种子开始,RNG 总产生一些相同序列的随机数.这是一个很有用的属性,并具有重要意义,因为其重复性好.

RNG 在一定范围内,产生均匀分布的整数,通常是介于 0,1,2^{32}之间.数学变换从相应的具体分布,来产生随机变量.

6.4.1 随机数产生器

Mersenne Twister

缺省情况下,由 M. Matsumoto 和 T. Nishimura [Matsumoto98], OMNeT++使用 Mersenne Twister RNG (MT).MT 的周期为 2^{19937}-1,和确定的 623 维分配属性.MT 也非常快速:跟 ANSIC 的 rand()一样快或比其更快.

RNG 的”最小标准”

OMNet++ 3.0 之前版本使用一个在[Jain91], pp. 441-444,455 中描述的,循环长度为 2^{31}-2,线性同余产生器(LCG).RNG 仍然有用,可以从 omnetpp.ini 选择.这个 RNG 仅适合小范围的仿真研究. Karl Entacher 等人在[Entacher02]中所示的, 2^{31}的循环长度大小了(现今快速计算机很容易枚举所有的随机数),产生的”随机”数的结构太规则. [Hellekalek98]论文提出仿真中使用与 RNG 相关的主要问题概述,它也非常值得学习.它也包括关于主题的有用的链接和引用.

Akaroa RNG

当在 Akaroa 控制下执行仿真时,由于采用 OMNet++随机数功能的 RNG 底层也可以选择 Akaroa 的 RNG. Akaroa RNG 也几乎全部选自 omnetpp.ini.

其它 RNG

OMNeT++也允许添加你自己的 RNG.这个机制基于 cRNG 界面,在[13.5.3]节描述.例如, 可以是包括周期为 2^{191} 的 L'Ecuyer 的 CMRG[LEcuyer02],提供大量保证的独立流.

6.4.2 随机数流,RNG 映射

仿真程序会从多个流消耗随机数,即从多个独立的 RNG 实例.例如,如果一个网络仿真使用随机数来产生包和在传输中的仿真比特错误,对这两种情况,使用不同的数据流是个好的方法.由于每个流的种子可以被独立配置,这个安排会允许你执行发生在不同地方的相同负荷但有比特错误的多条仿真.仿真技术称为减少变化,也关系到用不同的随机数流.

但不同的流和不同仿真运行时使用非重叠的随机数序列也非常重要.在产生随机数序列中的重叠可以引入想要的相关结果.

随机数流的数量和单个流的种子可以在 omnetpp.ini 中被配置([8.6]节).对于"最小标准 RNG", seedtool 程序可用于选择好的种子([8.6.6]节).

在 OMNet++中,流是由 RNG 数标识.在简单模块中使用的 RNG 数可以是从 omnetpp.ini 中对实际随机数流的任意映射([8.6]节). 在 RNG 使用和随机数流配置中,映射允许极大的灵活性 — 甚至不需要写 RNG 的仿真模型.

6.4.3 访问 RNG

intrand(n)函数产生[0,n-1]范围内的随机整数, dblrand()产生[0,1)范围内的一个随机双精度.这些函数简单地约束了底层的 RNG 对象.例如:

```
int dice = 1 + intrand(6); // intrand(6) 返回结果在 0 和 5 范围之内.

double p = dblrand();      // dblrand() 产生 [0,1)之间的数
```

也可以使用产生器 k 产生一个副本:

```
int dice = 1 + genk_intrand(k,6); // 使用一个产生器 k

double prob = genk_dblrand(k);    // ""
```

底层的 RNG 对象是 cRNG 的子类,他们可以通过 cModule 的 rng()方法来访问. rng()的参数是经过 RNG 映射的本地的 RNG 数.

```
cRNG *rng1 = rng(1);
```

cRNG 包含实现以上 intrand()和 dblrand()函数的实现. cRNG 界面也允许你访问由 RNG 产生的"原始"的 32bit 随机数,并且获悉他们的范围(intRand(), intRandMax())以及查询产生的随机数(numbersDrawn()).

6.4.4 随机变量

以下的函数是基于 dblrand(),返回不同分布的随机变量:

随机变量函数使用由 OMNet++提供的一个随机数产生器(RNG).缺少情况下产生 0,但是可以指定使用哪一个.

OMNet++有以下预定义分布:

函数	描述
连续分布	

uniform(a, b, rng=0)	在[a,b)范围内的均匀分布
exponential(mean, rng=0)	给定均值的指数分布
normal(mean, stddev, rng=0)	给定均值和标准差的正态分布
truncnormal(mean, stddev, rng=0)	截去非负值的正态分布
gamma_d(alpha, beta, rng=0)	参数 alpha>0, beta>0 的 gamma 分布
beta(alpha1, alpha2, rng=0)	参数 alpha1>0, alpha2>0 的 beta 分布
erlang_k(k, mean, rng=0)	k>0 状态和给定均值的 Erlang 分布
chi_square(k, rng=0)	k>0 自由度的 chi 平方分布
student_t(i, rng=0)	自由度为 i>0 的 student-t 分布
cauchy(a, b, rng=0)	参数 a,b,其中 b>0 的柯西分布
triang(a, b, c, rng=0)	参数 a<=b<=c, a!=c 的三角分布
lognormal(m, s, rng=0)	均值 m 和方差 s>0 的对数正态分布
weibull(a, b, rng=0)	参数 a>0, b>0 的 Weibull 分布
pareto_shifted(a, b, c, rng=0)	参数 a,b 和移动 c 产生的 Pareto 分布
离散分布	
intuniform(a, b, rng=0)	a..b 的均匀分布
bernoulli(p, rng=0)	布尔试验的结果,概率 0<=p<=1((1 表示概率 p,0 表示概率(1-p))
binomial(n, p, rng=0)	参数 n>=0 且 0<=p<=1 的二项式分布
geometric(p, rng=0)	参数 0<=p<=1 的几何分布
negbinomial(n, p, rng=0)	参数 n>0 和 0<=p<=1 的二项式分布
poisson(lambda, rng=0)	参数为 lambda 的 Poisson 分布

在NED文件中,它们是可以使用的相同的函数. intuniform()产生整型数,包括上,下界,所以举例来说,抛硬币可以写为 intuniform(1,2). truncnormal()是截去非负值的正态分布;会执行产生一个正态分布的数, 如果结果是负数,它会产生其它数直至结果是非负的.

如果上述分布不是足够,可以写自己的函数.如果 Register_Function()使用宏注册了函数,可以在 NED 文件和 ini 文件中使用.

6.4.5 直方图得到的随机数

可以指定分布为一个直方图. cDoubleHistogram, cVarHistogram, cKSplit 或 cPSquare 类从等距或等概率的直方图中产生随机数.这个统计类的特性在后面描述.

6.5 容器类

6.5.1 队列类: cQueue

批注 **[z14]:** 队列类: cQueue

基本用途

cQueue 是一个容器类作为一个队列. cQueue 支持从 cObject 派生的类型对象(几乎所有的类都是来自 OMNet++库),比如 cMessage, cPar 等.在内部, cQueue 使用一个双精度链接列表来存储元素.

一个队列对象有头和尾.通常,新的元素从头部插入,元素从尾部移出.

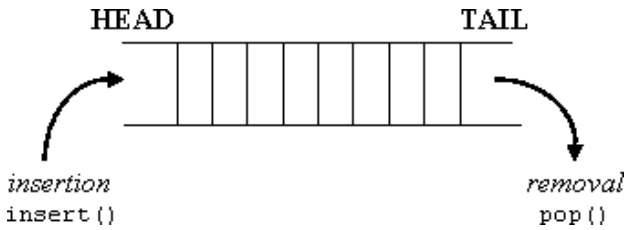


Figure: cQueue: insertion and removal

基本的 cQueue 成员函数处理插入和移出是 insert()和 pop().他们像这样使用:

```
cQueue queue("my-queue");

cMessage *msg;

// insert messages

for (int i=0; i<10; i++)

{

  msg = new cMessage;

  queue.insert( msg );

}

// remove messages

while( ! queue.empty() )

{

  msg = (cMessage *)queue.pop();

  delete msg;

}
```

length()成员函数返回队列中的项目成员, empty()告知队列中是否有什么数据.

有其它的函数处理插入和移除. insertBefore()和 insertAfter()指定在之前或之后插入一个新项,不管排序函数.

tail()和 head()函数返回指向队列对象头部和尾部的指针,不影响队列内容.

pop()函数可以用于从队列尾部移除项目, remove()函数可以用于从队列中移除指针指向的任何项目:

```
queue.remove( msg );
```

优先队列

缺省情况下, cQueue 实现了 FIFO,但是它也能作为一个优先队列,即可以保持插入的对象序列.如果你想使用这个特性,必须提供一个有两个 cObject 指针参数的函数,比较两个对象,且结果返回-1,0 或 1(详见参考文献).设置一个有序的 cQueue 的例子:

```
cQueue sortedqueue("sortedqueue", cObject::cmpbyname, true );
                                // sorted by object name, ascending
```

如果队列对象被设置为一个有序的队列, insert()函数使用一个有序的函数:它从头部开始寻找队列内容,直到搜索到新项需要插入的位置,然后插入.

Iterators

通常,你可以在队列的头部或尾部访问对象.然而,如果你使用一个 iterator 类, cQueue::Iterator,可以检查队列中的每个对象.

cQueue::Iterator 构造器有两个参数,第一个是队列对象,第二个是指定的 iterator 的初始位置: 0=tail, 1=head.否则它可以作为任何其它的 OMNeT++ iterator 类:你可以预先使用++和--操作符,()操作符得到指向当前项的指针,end()成员函数检查是否在队列的终点.

例如:

```
for( cQueue::Iterator iter(queue,1); !iter.end(), iter++)
{
  cMessage *msg = (cMessage *) iter();
  //...
}
```

6.5.2 扩展的数组: cArray

基本使用

cArray 是一个持有从 cObject 派生的对象的容器类. cArray 存储对象指针的插入而不是复制. cArray 作为一个数组,但它数组满时自动增长.在内部, cArray 实现了一个指针数组;当数组填满时,会重新分配.

在 OMNet++中使用的 cArray 对象存储参数附加至消息,并在内部,用于存储模块参数和门.

创建一个数组:

```
cArray array("array");
```

在第一个空索引处添加一个对象:

```
cPar *p = new cPar("par");
int index = array.add( p );
```

在给定的索引处添加一个对象(如果索引被占用,将得到一个错误消息):

```
cPar *p = new cPar("par");
int index = array.addAt(5,p);
```

在数组中寻找一个对象:

```
int index = array.find(p);
```

得到指向给定索引对象的指针：

```
cPar *p = (cPar *) array[index];
```

可以搜索数组或通过对象名得到一个指向对象的指针：

```
int index = array.find("par");

Par *p = (cPar *) array["par"];
```

通过对象名,索引位置或对象指针调用 remove(),可以从数组中移除一个对象：

```
array.remove("par");

array.remove(index);

array.remove( p );
```

remove()函数不能分配存储对象,但是它返回对象指针.如果也想分配存储对象,可以写：

```
delete array.remove( index );
```

Iteration

cArray 没有 iterator,但是容易循环所有一个整型变量的所有索引. items()成员函数返回最大的指数加 1.

```
for (int i=0; i<array.items(); i++)

{

  if (array[i]) // is this position used?

  {

    cObject *obj = array[i];

    ev << obj->name() << endl;

  }

}
```

6.6 参数类：cPar

模块参数([4.7]节所讨论)表示为 cPar 对象.模块参数名是 cPar 对象名,对象可以存储任何 NED 语言支持的任何参数类型,即,数值(长整型或双精度),布尔,字符串和 XML 配置文件引用.

[cPar 对象用于被部署,对添加参数至 cMessage.而从技术上来说吧,仍然是可行的,消息定义在各文献都远优于上面的解决方案.]

通过 cModule 的 par()方法访问模块参数：

```
cPar& par(const char *parameterName);
```

6.6.1 读取值

cPar 有大量的方法得到参数值：

```
bool boolValue();

long longValue();
```

```
const char *stringValue();

double doubleValue();

cXMLElement *xmlValue();
```

也有 C/C++基本类型的重载类型转换操作符,包括:bool,int,long,double,const char *, 也有 cXMLElement *.

[cPar 也支持 void *和 cObject *类型,但是这些类型基本用于支持消息定义前的消息参数,不能从 NED 文件中创建这些模块参数.]

因此,以下的任何一方法都会在变量中存储一个参数值:

```
double foo = par("foo").doubleValue();

double foo = (double) par("foo");

double foo = par("foo");
```

如果在表达式中使用 par("foo")参数(比如 4*par("foo")+2),C++编译器参于重载的操作符不能决定,报告不明确.在那种情况下,需要添加一个明确的转换((double)par("foo")或(long)par("foo"))或使用 doubleValue()或 longValue()来阐明.

isConstant()可以用于确定是否 cPar 存储了一个常量,或一个表达式可能每次读对象的时候产生不同的值,比如 1+exponential(0.5).

6.6.2 值改变

有许多方法来设置一个 cPar 值.其中一个是 set...Value()成员函数:

```
cPar& foo = par("foo");

foo.setLongValue(12);

foo.setDoubleValue(2.7371);

foo.setStringValue("one two three");
```

C++基本类型也有重载分配的操作符 const char *,和 cXMLElement *.

```
cPar pp("pp");

pp = 12;

pp = 2.7371;

pp = "one two three";
```

cPar 对象对其自身的字符串拷贝,因此原始的并不需要存储.短字符串(小于 20 字符)处理更有效,因为他们都存储在对象的内存空间(不是动态分配的).

cPar 也存储其它类型,那些产生数值结果的,比如常量参数的函数;将在下一节所及.

对于数值和字符串类型,可以设置一个输入模块.在这种情况下,当对象的值第一次使用,参数值将在配置文件中搜索;如果没配置文件中没有发现,用户将需要交互地提供输入值.

例如:

```
cPar foo("foo");

foo.setPrompt("Enter foo value:");
```

```
foo.setInput(true);   // make it an input parameter

double d = (double)foo; // the user will be prompted HERE
```

此外, set..()函数也分配其它的存储类型,比如,有常量参数(MathFuncNoArgs, MathFunc1Args 等)的 double 函数,反向 Polish 表达式,基于 cDoubleExpression 编译表达式,基于 cStatistic 的 random()方法的随机分布,指向 cObject 的指针等将在下一节列出.然而,他们在设计仿真模型中很少使用.

6.6.3 cPar 存储类型

cPar 通过存储类型支持基本数据类型(long, double, bool, string, XML).存储类型由类型字符串在内部标识.类型字符串由 type()方法返回.

例如:

```
cPar par = 10L;

char typechar = par.type(); // 返回存储类型'L'
```

所有的 cPar 数据类型在下表中概述. isNumeric()函数告诉我们对象存储一个数据类型是否允许被 doubleValue()方法调用.

类型字符串	存储类型	成员函数	描述

S	string	setStringValue(const char *);
		const char *stringValue();
		op const char *();
		op=(const char *);
			string 值.短字符串(len<=27)存储于 cPar 对象内部,不使用堆分配.

B	boolean	setBoolValue(bool);
		bool boolValue();
		op bool();
		op=(bool);
			Boolean 值.也可从对象如 long 中提取(0 或 1)

L	long int	setLongValue(long);
		long longValue();
		op long();
		op=(long);
			分配长整型值.也可从对象如 double 中提取

D	double	setDoubleValue(double);
		double doubleValue();
		op double();
		op=(double);
			双精度浮点值

F	function	setDoubleValue(MathFunc,
		[double],
		[double],
		[double]);
		double doubleValue();
		op double();
			具有常量参数的数学函数.函数由其指针给定;它必须是 0,1,2 或 3 个双精度,并且返回一个双精度.这个类型主要用于产生随机数:比如有均值和标准差的函数,返回一个一定分布的随机变量.

X	expr.	setDoubleValue(	运行时评价反向 Polish 表达式,可以含常

		cPar::ExprElem*,int);
		double doubleValue();
		op double();
			量, cPar 对象,引用其它 cPar(比如模块参数),可以使用数学操作符(+-*/^%等),函数调用(函数必须有0,1,2或3个双精度并返回一个双精度).表达式必须在一个给定的cPar::ExprElem 结构体数组中

.C	compiled expr.	setDoubleValue(
		cDoubleExpression *expr);
		double doubleValue();
		op double();
			运行时评价编译的表达式.表达式应用在cDoubleExpression 的一个对象子类的方法中提供.

T	distrib.	setDoubleValue(
		cStatistic*);
		double doubleValue();
		op double();
			通过统计数据收集对象(cStatistic 派生),从分布收集中产生的随机变量.

M	XML	setXMLValue(cXMLElement *node); cXMLElement *xmlValue(); op cXMLElement*();	引用在 XML 配置文件中的,XML 元素

P	void* pointer	setPointerValue(void*);
		void *pointerValue();
		op void *();
		op=(void *);
			指向一个非 cObject 项的指针(C 结构体,非cObject 对象等).内存管理可以通过成员函数 configPointer()来控制.

O	object pointer	setObjectValue(cObject*);
		cObject *objectValue();
		op cObject *();
		op=(cObject *);
			指向一个从 cObject 派生的对象的指针.所有权管理是通过 takeOwnership()来完成的.

I	indirect value	setRedirection(cPar*);
		bool isRedirected();
		cPar *redirection();
		cancelRedirection();
			值是非直接的其它 cPar 对象.所有在其它cPar 上的值设置和读取操作;甚至 type()函数都将返回其它 cPar 的类型.这种非直接仅被成员函数 cancelRedirection()破坏.使用这个机制引用模块参数.

6.7 路由支持: cTopology

6.7.1 概述

cTopology 类设计的基本目的是支持在通信或多处理器网络中支持路由.

cTopology 对象以图形形式存储一个抽象网络表示:

- 每个 cTopology 节点对应于一个模块(简单或复合)
- 每个 cTopology 边对应于一个链接或链接序列.

可以指定图形中需要包括哪个模块(简单或复合).图形将包括所有选中模块的链接.在图形中,所有节点都在同一层次,没有嵌套的子模块.跨越算命模块边界的链接也表示为一个图形边.图形边是有向的,正如模块门.

如果写一个路由或转换模型, cTopology 图可以帮助决一哪些节点通过门是可用的,并找出最优的路由. cTopology 对象可以在节点之间为你计算出最短路径.

图形(节点,边)之间的映射和网络模型(模块,门,链接)被存储: 可以简单地发现,cTopology 节点相应的模块,否则相反.

6.7.2 基本用途

通过调用单个函数,可以提取网络拓扑到一个 cTopology 对象.也可许多方法来选择哪个模块需要包括在拓扑中:

- 通过模块类型
- 通过一个参数和其值
- 有一个用户提供了布尔函数

首先,指定需要包括哪些节点类型.以下的代码提取了所有路由器或主机的类型模块.(路由器和主机可以是简单或复合模块类型.)

```
cTopology topo;

topo.extractByModuleType("Router", "Host", NULL);
```

可以提供任何数量的模块类型.列表以 NULL 终止.

一个动态模块类型组合列表可以通过一个以 NULL 结尾的 const char*指针数组传递,或在一个 STL 字符串向量 std::vector<std::string>中传递.前者的例子:

```
cTopology topo;

const char *typeNames[3];

typeNames[0] = "Router";

typeNames[1] = "Host";

typeNames[2] = NULL;

topo.extractByModuleType(typeNames);
```

其次可以提取所有具有一定参数的模块

```
topo.extractByParameter( "ipAddress" );
```

也可以指定图形中需要包括的模块,其参数必须有一定的值:

```
cPar yes = "yes";

topo.extractByParameter( "includeInTopo", &yes );
```

第三种形式允许传递一个函数,其可以决定每个模块是否应该或不应该被包括.可以通过 void*指针,使 cTopology 传递追加的数据至函数.选择所有顶层模块的例子(不使用 void*指针):

```
int selectFunction(cModule *mod, void *)

{

  return mod->parentModule() == simulation.systemModule();

}

topo.extractFromNetwork( selectFunction, NULL );
```

使用两个类型的 cTopology 对象: 节点的 cTopology::Node 和边的 cTopology::Link.通过 (sTopoLinkIn 和 cTopology::LinkOut 是 cTopology::Link 的别名,我们将在稍后讨论.)

一旦提取了拓扑,可以开始查看.考虑以下的代码(我们简短解释):

```
for (int i=0; i<topo.nodes(); i++)
{
  cTopology::Node *node = topo.node(i);
  ev << "Node i=" << i << " is " << node->module()->fullPath() << endl;
  ev << " It has " << node->outLinks() << " conns to other nodes\n";
  ev << " and " << node->inLinks() << " conns from other nodes\n";
  ev << " Connections to other modules are:\n";
  for (int j=0; j<node->outLinks(); j++)
  {
    cTopology::Node *neighbour = node->out(j)->remoteNode();
    cGate *gate = node->out(j)->localGate();
    ev << " " << neighbour->module()->fullPath()
       << " through gate " << gate->fullName() << endl;
  }
}
```

nodes()成员函数(第一行)返回图形中节点数量,

cTopology::Node *node = topo.nodeFor(module); node(i)返回指向第 i 个节点为指针,一个 cTopology::Node 结构.

图形节点和模块之间对应关系的获得通过:

```
cModule *module = node->module();
```

nodeFor()成员函数返回一个指向给定模块的图形节点的指针.(如果模块不在图中,则返回 NULL). nodeFor()使用 cTopology 内的二元搜索,因此其足够快速.

cTopology::Node 的其它成员函数,可以决定该节点的链接: inLinks(), outLinks()返回链接数, in(i)和 out(i)返回指向图形边对象的指针.

通过调用图形边对象的成员函数,可以决定包括的模块和门.remoteNode()函数返回链接的其它终点, localGate(), remoteGate(), localGateId()和 remoteGateId()返回所包括的门指针和门 id.(实际上,在这里的实现有一个技巧:相同的图形边对象 cTopology::Link 被返回的是 cTopology::LinkIn 或 cTopology::LinkOut,因此 remote 和 local 可以被正确地解释为双向边.)

6.7.3 最短路径

cTopology 的现实作用是在网络中寻找最短路径来支持最优路由. cTopology 从所有的节点至目标节点寻找最短路径.算法的计算是廉价的.最简单的情况是, 假定所有的边有相同的权值.

一个现实生活中的例子,当我们有了目标模块的指针,寻找最短路径类似这样:

```
cModule *targetmodulep =...;

cTopology::Node *targetnode = topo.nodeFor( targetmodulep );

topo.unweightedSingleShortestPathsTo( targetnode );
```

这个 Dijkstra 算法并在 cTopology 对象中存储结果.结果可以使用 cTopology 和 cTopology::Node 方法来提取.很自然地,每次调用 unweightedSingleShortestPathsTo() 都重写了之前的调用结果.

从我们的模块遍历至目的节点:

```
cTopology::Node *node = topo.nodeFor( this );

if (node == NULL)

{

  ev < "We (" << fullPath() << ") are not included in the topology. \n";

}

else if (node->paths()==0)

{

  ev << "No path to destination.\n";

}

else

{

  while (node != topo.targetNode())

  {

    ev << "We are in " << node->module()->fullPath() << endl;

    ev << node->distanceToTarget() << " hops to go\n";

    ev << "There are " << node->paths()

       << " equally good directions, taking the first one\n";

    cTopology::LinkOut *path = node->path(0);

    ev << "Taking gate " << path->localGate()->fullName()

       << " we arrive in " << path->remoteNode()->module()->fullPath()

       << " on its gate " << path->remoteGate()->fullName() << endl;

    node = path->remoteNode();

  }

}
```

节点 distanceToTarget() 成员函数的目的是不需要加以说明的.在无权重的情况下,它返回跳数. paths()成员函数返回最短路径的边的数量, path(i)返回 cTopology::LinkOut 形式

的第 i 条边.如果最短路径通过...SingleShortestPaths()函数创建, paths()总是返回 1 (或如果目标不可达,返回 0),即仅是寻找到的许多最短路径中的一条. 函数 ...MultiShortestPathsTo()寻找所有路径,增加运行的代价. cTopology 的 targetNode()函数返回搜索的最短路径的目的节点.

可以允许/禁止图形中的节点或边.这是通过调用它们的 enable()或 disable()成员函数.禁止节点或边在最短路径算法中被忽略. enabled()成员函数返回拓扑图中节点或边的状态.

disable()的一个用途是,决定从我们节点的一个特定的输出门到达目的节点有多少跳的情况.为了计算这个,你计算邻居节点至目的节点的最短路径,但是要禁止当前的节点参与:

```
cTopology::Node *thisnode = topo.nodeFor( this );

thisnode->disable();

topo.unweightedSingleShortestPathsTo( targetnode );

thisnode->enable();

for (int j=0; j<thisnode->outLinks(); j++)

{

  cTopology::LinkOut *link = thisnode->out(i);

  ev << "Through gate " << link->localGate()->fullName() << " : "

     << 1 + link->remoteNode()->distanceToTarget() << " hops" << endl;

}
```

以后,其它最短路径算法也会实现:

```
unweightedMultiShortestPathsTo(cTopology::Node *target);

weightedSingleShortestPathsTo(cTopology::Node *target);

weightedMultiShortestPathsTo(cTopology::Node *target);
```

6.8 统计和分布估计

6.8.1 cStatistic 和派生

有许多统计和结果收集类: cStdDev, cWeightedStdDev, LongHistogram, cDoubleHistogram, cVarHistogram, cPSquare 和 cKSplit.他们都是从抽象基础 cStatistic 派生而来.

- cStdDev 保存样本数目, 平均值, 标准差, 最高和最低值等.
- cWeightedStdDev 类似于 cStdDev,但是接受权值查看. 例如 cWeightedStdDev 可以用于计算时间平均值.它仅是一个权值统计类.
- cLongHistogram 和 cDoubleHistogram 是 cStdDev 的派生,也保存使用等距(相等大小)的单元柱状图查看的分布近似值.
- cVarHistogram 实现了一个柱状图,其中单元不需要相同的大小.可以手动添加单元边界,或自动创建一个分割,其中每个单元有相同(或尽可能近似)的观测数.
- cPSquare 是在[Jch85]中描述的 P^2 使用算法的类.该算法计算没有存放观测数据的分位数,可以把它作为一种具有等概率的柱状图单元.
- cKSplit 使用一个新的,实验的方法,基于一个适应类柱状图的算法.

基本用途

可以使用 collect()函数或+=操作符(他们等同)插入一个观测数据至统计对象. cStdDev 有以下的方法得到统计出的对象: 有明显含义的 samples(), min(), max(), mean(), stddev(), variance(), sum(), sqrSum().cStdDev 的一个用例:

```
cStdDev stat("stat");

for (int i=0; i<10; i++)

  stat.collect( normal(0,1) );

long numSamples = stat.samples();

double smallest = stat.min(),

       largest = stat.max();

double mean = stat.mean(),

       standardDeviation = stat.stddev(),

       variance = stat.variance();
```

6.8.2 分布估计

初始化和用法

分布估计类(cLongHistogram, cDoubleHistogram, cVarHistogram, cPSquare and cKSplit)是从 cDensityEstBase 中派生而来.分布估计类(除了 cPSquare)假定在一个范围内观察.可以明确指定范围(基于先前关于分布的信息)或可以让对象首先收集一些观察数据,然后从中决定范围.指定范围设置的方法是 cDensityEstBase 的一部分.

以下的的成员函数存在用于设置范围,指定多少观察数据应该用于自动决定范围.

```
setRange(lower,upper);

setRangeAuto(numFirstvals, rangeExtFactor);

setRangeAutoLower(upper, numFirstvals, rangeExtFactor);

setRangeAutoUpper(lower, numFirstvals, rangeExtFactor);

setNumFirstVals(numFirstvals);
```

以下的例子创建了一个有 20 个单元的柱状图,和自动范围评价:

```
cDoubleHistogram histogram("histogram", 20);

histogram.setRangeAuto(100,1.5);
```

这里 20 是一个单元数量(不包括下溢/上溢单元),100 是在设置单元之前收集的观察数据数量.1.5 是范围扩展因子.表示实际的初始观察范围要扩展 1.5 倍,其扩展范围用于布置单元.这个方法增加了机会来再一步观察在一个单元内的数据,不是在柱状图范围外的.

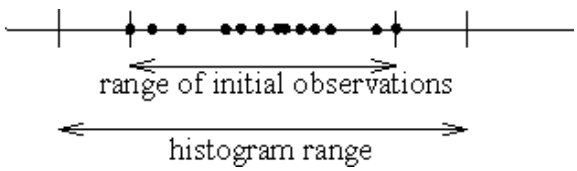


Figure: Setting up a histogram's range

在单元设置之后,继续收集.

当单元已经设置好时,transformed()函数返回 true.通过 transform()函数调用,可以加强范围估计并且设置单元.

观察数据在柱状图之外被计数为下溢和上溢. underflowCell()和 overflowCell()成员函数返回下溢和下溢的数量.

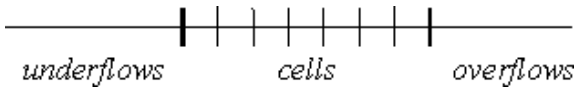


Figure: Histogram structure after setting up the cells

通过指定的单元数量创建一个 P^2 对象:

```
cPSquare psquare("interarrival-times", 20);
```

然后, cPSquare 使用相同成员函数作为一个柱状图.

得到柱状图数据

有三个成员函数来明确返回单元边界和每个单元观察的数量.cells()返回单元数量, basepoint(int k)返回第 k 个基点, cell(int k)返回在单元 k 中观察的数量, cellPDF(int k)返回在单元内的 PDF 值(比如,在 basepoint(k)和 basepoint(k+1)之间).这些函数为所有的柱状图类型,以及 cPSquare 和 cKSplit 工作.

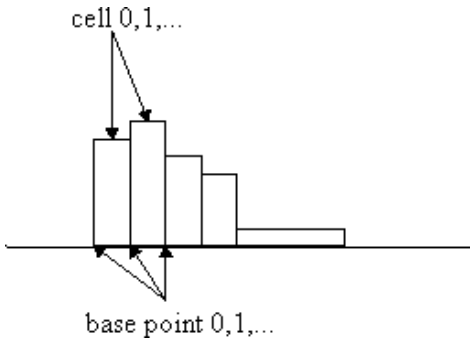


Figure: base points and cells

例如:

```
long n = histogram.samples();

for (int i=0; i<histogram.cells(); i++)

{

  double cellWidth = histogram.basepoint(i+1) -histogram.basepoint(i);

  int count = histogram.cell(i);

  double pdf = histogram.cellPDF(i);

  //...

}
```

pdf(x)和 cdf(x)成员函数分别返回给定 x 的概率密度函数值和累积密度函数值.

从分布中产生随机数

random()成员函数从对象存储的分布中产生随机数：

```
double rnd = histogram.random();
```

cStdDev assumes normal distribution.

也可以使用在 cPar 中的分布对象：

```
cPar rndPar("rndPar");

rndPar.setDoubleValue(&histogram);
```

cPar 对象存储指向柱状图(或 P^2 对象)的指针，当要求值时，调用柱状图对象的 random()函数：

```
double rnd = (double)rndPar; // random number from the cPSquare
```

存储/下载分布

统计类有 loadFromFile()成员函数从一个文本文件读取柱状图数据．如果需要一个定制分布不能(或效率低)像 C 函数一样被写，可以通过存储在一个文本文件中描述柱状图，使用一个 loadFromFile()柱状图对象．

可以使用 saveToFile()，通过柱状图对象写出分布收集：

```
FILE *f = fopen("histogram.dat","w");

histogram.saveToFile(f); // save the distribution

fclose(f);

cDoubleHistogram hist2("Hist-from-file");

FILE *f2 = fopen("histogram.dat","r");

hist2.loadFromFile(f2); // load stored distribution

fclose(f2);
```

单元定制柱状图

cVarHistogram 类可以用于创建任意(非等距的)单元柱状图．可以在两种模式下操作：

- 手动，其中在开始收集之前，明确指定单元边界
- 自动，其中 transform()将在收集一定的初始观察数据之后设置单元．单元将被设置，使相等数量的观察数据属于同一个单元(等距单元)．

选择 transform-type 参数的模型：

- HIST_TR_NO_TRANSFORM：不转换；使用先前由 addBinBound()定义的 bin 界限；
- HIST_TR_AUTO_EPC_DBL：自动创建等概率单元
- HIST_TR_AUTO_EPC_INT：类似上面，但是对于整型数据的

创建一个对象：

```
cVarHistogram(const char *s=NULL,
              int numcells=11,
              int transformtype=HIST_TR_AUTO_EPC_DBL);
```

手动添加单元界限：

void addBinBound(double x);

在收集 numFirstVals 初始观察数据之后，选择 rangemin 和 rangemax. 当柱状图已经转换后，不能添加单元界限

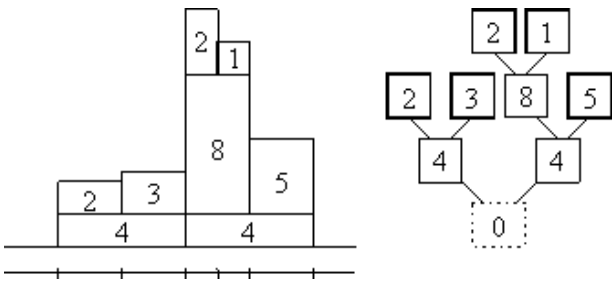
6.8.3 k 分裂算法

目的

k 分裂算法是一个即时分布估计方法. 它设计用于在仿真程序中即时结果收集. 这方法是由 Varga 和 Fakhamzadeh 在 1997 年提出的. k 分裂的主要优点是不存储观察数据，在不需要先前分布信息包括采样大小的情况下，给出一个好的估计. k 分裂算法可以扩展为多维分布，但是这里仅解决一维.

算法

k 分裂算法适应柱状图类型估计，通过完成单元分裂，维护一个良好的分割. 柱状图范围 $[x_{lo}, x_{hi})$ 有 k 个相同大小的柱状图单元，其观察数据计数为 $n_1, n_2, \ldots n_k$. 每个收集的观察数据增加相应的观察计数. 当一个观察数据计数为 n_i 达到了分裂极限，单元就分裂为 k 个更小的，相同大小的单元，观察数据计数为 $n_{i,1}, n_{i,2}, \ldots n_{i,k}$ 初始化为 0. n_i 观察数据被记住，并称之为新建单元的母观察计数. 进一步观察全引起起单元的进一步分裂（如 $n_{i,1,1}, \ldots n_{i,1,k}$ 等），因而创建观察数据计数的 k 序列树，其中叶子包括活动计数器，实际上由新的观察数据自动增长. 如果一个观察数据落在柱状图范围之外，以普通的方法通过在树的顶部插入一个新的层次来扩充范围. 算法的基本的参数是分裂因子 k. 经验显示 k=2 工作最佳.



图：阐述了 k 分裂算法，k=2. 格子里的数字表示观察计数值

为了密度估计，分割的每个单元的总的观察数量必须确定. 这样，树中每个内部节点的母观察数据必须被分布在它的子单元中，并传至叶子节点.

假设 $n_{\ldots,i}$ 是一个单元的（母）观察计数，$s_{\ldots,i}$ 是一个单元 $n_{\ldots,i}$ 内的观察计数加上其所有 sub-, sub-sub-等观察计数的总和，$m_{\ldots,i}$ 母观察数据按比例至单元. 我们关注于 $\tilde{n}_{\ldots,i} = n_{\ldots,i} + m_{\ldots,i}$ 估计树中节点，尤其是叶子节点的观察数量. 即，如果我们在一个单元内有 $\tilde{n}_{\ldots,i}$ 估计观察数量，如果划分为可以按比例至子单元的 $m_{\ldots,i,1}, m_{\ldots,i,2} \ldots m_{\ldots,i,k}$. 通常 $m_{\ldots,i,1} + m_{\ldots,i,2} + \ldots + m_{\ldots,i,k} = \tilde{n}_{\ldots,i}$.

两种普通的分布方法是平均分布 $(m_{\ldots,i,1} = m_{\ldots,i,2} = \ldots = m_{\ldots,i,k})$ 和按比例分布 $(m_{\ldots,i,1} : m_{\ldots,i,2} : \ldots : m_{\ldots,i,k} = s_{\ldots,i,1} : s_{\ldots,i,2} : \ldots : s_{\ldots,i,k})$. 当 $s_{\ldots,i,j}$ 值非常小时，平均分布是最优的，当 $s_{\ldots,i,j}$ 值大于 $m_{\ldots,i,j}$ 时，按比例分布是好的. 实际应用中，它们的线性组合更适合，其中 λ=0 表示平均，λ=1 表示按比例分布：

$m_{\ldots,i,j} = (1-\lambda)\tilde{n}_{\ldots,i}/k + \lambda\ \tilde{n}_{\ldots,i}\ s_{\ldots,i,j} / s_{\ldots,i}$ where λ is in $[0,1]$

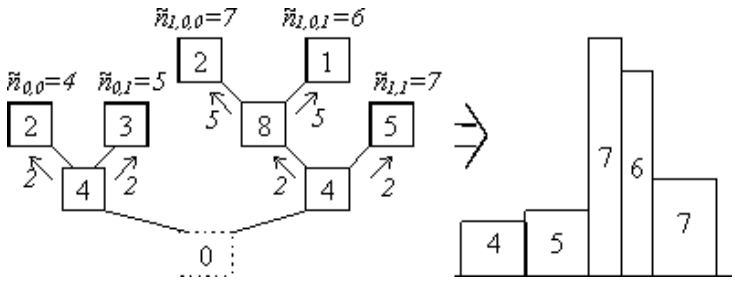


Figure: Density estimation from the k-split cell tree. We assume λ=0, i.e. we distribute mother observations evenly.

注意, $n_{...,i}$ 是整型, $m_{...,i}$ 和 $\tilde{n}_{...,i}$ 通常是实数.从 k 分裂的柱状图分析计算是不精确的,由于以上方法的频繁计算包括估计本身的度.这里引入一定单元分割错误;应该选择 λ 参数,因而最小化错误.其显示了单元分裂错误可以减少到更可以接受的小的值.

严格来说,k 分裂算法是半即时的,由于它需要一些观察来设置初始的柱状图范围.由于范围扩充和单元分裂能力,算法对于初始范围的选择并不非常敏感,因此较少的观察数据对于范围估计来说是足够的(比如 N_{pre}=10).因此,我们可以考虑 k 分裂算法是一个即时方法.

k 分裂也手于半即时模式,当算法仅用于从一个较大的 N_{pre} 个观察数据中,创建一个最优的分割.当分割创建时,观察计数被清除, N_{pre} 观察数据被再次反馈给 k 分裂.这种方法所有的母(非叶)观察计数将为 0,单元分割错误被消除.显示了 k 分裂创建的分割较等距和等频率分割更好.

OMNet++包括 k 分割算法的实验实现, cKSplit. k 分裂的研究仍在进行中.

cKSplit 类

cKSplit 类是 k 分裂方法的实现.成员函数:

```
void setCritFunc(KSplitCritFunc _critfunc, double *_critdata);

void setDivFunc(KSplitDivFunc \_divfunc, double *\_divdata);

void rangeExtension( bool enabled );

int treeDepth();

int treeDepth(sGrid& grid);

double realCellValue(sGrid& grid, int cell);

void printGrids();

sGrid& grid(int k);

sGrid& rootGrid();

struct sGrid

{

  int parent;   // index of parent grid

  int reldepth; // depth = (reldepth - rootgrid's reldepth)

  long total;   // sum of cells & all subgrids (includes `mother')

  int mother;   // observations `inherited' from mother cell
```

```
  int cells[K]; // cell values
};
```

6.8.4 瞬时检测和结果精确性

在许多的仿真中,仅关注稳定的状态实现(比如,系统到达稳定状态后的操作).系统初始阶段称为瞬时周期.在模型进入稳定状态后,仿真必须继续直至收集了足够的统计数据来计算出所需的精确性的结果.

OMNet++支持瞬时周期终点的探测和一定结果精确性.用户可以附加瞬时探测和结果精确性对象至一个结果对象(cStatistic 的派生).瞬时检测和结果精确性对象将在数据反馈至结果对象上完成指定的算法,并且告知是否瞬时周期已经结束或已经达到结果的精确性.

执行指定瞬时检测和结果精确性的检测算法类的基类是:

- cTransientDetection: 瞬时检测基类
- cAccuracyDetection: 结果精确性检测基类

基本用途

附加检测对象至 cStatistic,并且得到指向附加对象的指针:

addTransientDetection(cTransientDetection *object);

addAccuracyDetection(cAccuracyDetection *object);

cTransientDetection *transientDetectionObject();

cAccuracyDetection *accuracyDetectionObject();

检测周期结束:

- 轮流检测对象的 detect()函数
- 安装一个检测之后的函数

瞬时检测

目前实现一个瞬时检测的算法,如从 cTransientDetection 派生的类. cTDExpandingWindows 类使用有两个窗口的滑动窗口,检测两个平均之间的区别,来查看瞬时检测周期是否结束.

```
void setParameters(int reps=3,
                   int minw=4,
                   double wind=1.3,
                   double acc=0.3);
```

精确性检测

目前实现一个精确性检测算法,如,从 cAccuracyDetection 派生的类.算法在 cADByStddev 类中实现:通过值数量的平方来分割标准差,检测是否足够小.

```
void setParameters(double acc=0.1, int reps=3);
```

6.9 记录仿真结果

6.9.1 输出向量: cOutVector

cOutVector 类型的对象负责写时间序列数据(称为输出向量)至一个文件. record()方法用于输出一个带时间戳的值(或值对).对象名用于作为输出向量的名称.

向量名可以在构造器中传递,需要使用 cOutVector 为模块类的一个成员,在 initialize()时设置名称.需要从 handleMessage()或从 handleMessage()调用的函数中记录值.

下面的例子是一个子 Sink 模块,记录了每个消息到达的生命时间.

```
cOutVector responseTimeVec("response time");
```

但是通常

```
class Sink : public cSimpleModule
{
  protected:
    cOutVector endToEndDelayVec;
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};
Define_Module(Sink);
void Sink::initialize()
{
    endToEndDelayVec.setName("End-to-End Delay");
}
void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->creationTime();
    endToEndDelayVec.record(eed);
    delete msg;
}
```

也有一个 recordWithTimestamp()方法[New!],使得可能记录值至带时间戳的输出向量,而不是 simTime().仍然需要加强时间戳序列.

所有的 cOutVector 对象都写单个输出向量文件,缺省命名为 omnetpp.vec.可以从 omnetpp.ini 文件中配置输出向量:可以禁止写文件或记录时限制一定的仿真时间间隔([8.5]节).

输出向量文件的格式和处理在[10.1]节描述.

如果输出向量对象被禁止或仿真时间在指定的间隔之外, record()不写任何数据至输出文件.然而,如果为输出向量对象打开 Tkenv 查看窗口,那么将在窗口显示,不管输出向量对象的状态.

6.9.2 输出标量

由于输出向量存储时间序列数据,因而在仿真运行期间,他们通常记录大量的数据列,输出标题用于次仿真运行时记录单个值.可以使用输出标量

- 在仿真运行结束处,来记录概要数据
- 完成多个不同 settings/random 参数的运行,决定一些参数设置方法的依赖.例如,多运行和输出标题是产生吞吐量与负载的方法.

输出标量使用 cSimpleModule 的 recordScalar() 方法记录,通常要插入这段代码至 finish() 函数.例如:

```
void Transmitter::finish()
{
    double avgThroughput = totalBits / simTime();
    recordScalar("Average throughput", avgThroughput);
}
```

可以通过调用 recordScalar() 方法,声明为 cStatistic 的一部分,记录整统计对象.在下面的例子中,我们创建了一个 Sink 模块,用于计算均值,标准差,变量的最小和最大值,并且在仿真结束的时候记录.

```
class Sink : public cSimpleModule
{
  protected:
    cStdDev eedStats;
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();
};
Define_Module(Sink);
void Sink::initialize()
{
    eedStats.setName("End-to-End Delay");
}
void Sink::handleMessage(cMessage *msg)
{
    simtime_t eed = simTime() - msg->creationTime();
    eedStats.collect(eed);
    delete msg;
}
```

```
void Sink::finish()
{
    recordScalar("Simulation duration", simTime());
    eedStats.recordScalar();
}
```

以上调用缺省情况下写入名为 omnetpp.sca 的输出标量文件.输出标量文件通过仿真运行保存(不像输出向量文件,在每课外作业仿真运行开始的时候删除).数据在文件结尾处追回,从由特殊线分隔的不同的仿真运行输出.输出微量文件的格式和处理在[10.2]节描述.

6.9.3 精度[New!]

输出标量和输出向量文件都是文本文件,浮点值(双精度)使用 fprintf()的"%g"记录.一些重大的数字可以使用 output-scalar-precision=和 output-vector-precision=配置条目进行配置(见[8.2.6]).缺省精度是 12 位.当改变缺省值时,以下必须考虑:

IEEE-754 双精度是 64 位.该尾数是 52 位,大约相当于 16 个小数(52*log(2)/log(10)).然而,由于四舍五入的误差,通常仅有 12..14 位是正确的,其余是应该忽略的随机无用数据.然而,当要将小数转换表示为 IEEE-754 双精度(比如在 Plove 和 Scalars), 由于 0.1,0.01 等不能在二进位中精确表示,这将引起一个额外的小错误.这种转换错误通常小于记录至文件之前的双精度变量,但如果其很重要,可以通过设置文件>16 位数字精度来消除(但同样,得到的最后的数据位是无用的).实际的上限是 17 位,设置更高的话,在 fprintf()的输出中没有任何区别.

转换 to/from 小数表示的错误可以通过选择存储原始二进制形式的输出向量 /输出标量的管理类来消除.每当的配置条目为 outputvectormanager-class= 和 outputvectormanager-class=;见[8.2.6].例如,在 samples/database 中提出的 cMySQLOutputScalarManager 和 cMySQLOutputScalarManager 完成这个需求.

然而,在担心四舍五入和转换的错误之前,值得考虑什么是结果真实的精确性.需要考虑:

- 在现实生活中,很难衡量数字的精确度(重量,距离,甚至时间).哪些精度是输入的数据?例如,如果近似交互到达时间为 exponential(0.153),均值实际是 0.152601...,并且甚至分布不是确切的指数,已经开始比四舍五入引起的误差要大.
- 仿真模型本身是现实生活中的近似.在结果中,有多少误差(已知或未知)做了简化?

6.10 查看和快照

6.10.1 基本查看

其非常好,但是类型变量 int,long,double 在 Tkenv 缺省情况下不显示;既不是 STL 类(std::string, std::vector 等)也不是自己定义的构体.这是由于仿真内核是一个库,并不知道在源代码中的类型和变量.

OMNeT++提供了 WATCH()和其它宏集,使变量中以在 Tkenv 中查看,并输出至快照文件.WATCH()宏通常放在 initialize()中(查看实例变量)或在 activity()函数的顶部(查看局部变量),有一种观点认为,它们只应执行一次.

```
long packetsSent;
double idleTime;
WATCH(packetsSent);
```

```
WATCH(idleTime);
```

当然,类成员和结构体也可以查看:

```
WATCH(config.maxRetries);
```

当在 Tkenv 开始检查简单模块时,点击 Objects/Watches 标签,将看到要查看的变量和它们的值. Tkenv 也可以让你改变这些观察到的值.

WATCH()宏可以用于任何类型,有一个数据流输出操作(operator<<)定义.缺省情况下,包括所有的基本类型和 std::string,但是既然可以为类/结构体和基本类型写 operator<<,那么 WATCH()可以用于任何情况.唯一的限制是由于输出需要或多或少适合于单行,信息量显示的便利性受限.

一个流输出操作例子:

```
std::ostream& operator<<(std::ostream& os, const ClientInfo& cli)
{
    os << "addr=" << cli.clientAddr << "  port=" << cli.clientPort; // no endl!
    return os;
}
```

WATCH()行:

```
WATCH(currentClientInfo);
```

6.10.2 读写查看[New!]

查看基本类型和 std::string,也允许从 GUI 改变值,但对其它类型需要明确添加支持.需要做的是定义一个流输入操作符(operator>>),并且使用 WATCH_RW()宏取代 WATCH().

流输入操作:

```
std::ostream& operator>>(std::istream& is, ClientInfo& cli)
{
    // read a line from "is" and parse its contents into "cli"
    return is;
}
```

和 WATCH_RW()行:

```
WATCH_RW(currentClientInfo);
```

6.10.3 结构体查看[New!]

WATCH()和 WATCH_RW()是基本查看:他们允许显示一行文本(非结构体).然而,如果从消息定义产生一个数据结构(见第[5]章),那么有一个可以完成的更好.消息编译器自动产生元信息来描述单个类域或结构体,使用其可能在域层次显示内容.

用于这个目的的 WATCH 宏是 WATCH_OBJ()和 WATCH_PTR().两者都需要对象是 cPolymorphic 的子类; WATCH_OBJ()希望引用这样的类, WATCH_PTR()需要一指针变量.

```
ExtensionHeader hdr;
```

```
ExtensionHeader *hdrPtr;

...

WATCH_OBJ(hdr);

WATCH_PTR(hdrPtr);
```

注意:对于 WATCH_PTR(),指针变量每次都必须指向一个有效的对象或为 NULL,否则 GUI 在尝试显示对象时,会被破坏.这实际上表示,指针即使不使用也应该被初始化为 NULL,当对象指针被删除时应该设置为 NULL.

```
delete watchedPtr;

watchedPtr = NULL;  // 当对象被删除时设置为 NULL
```

6.10.4 STL 查看 [New!]

标准 C++容器类(vector, map, set 等)也有结构查看,通过以下的宏使其可用:

WATCH_VECTOR(), WATCH_PTRVECTOR(), WATCH_LIST(), WATCH_PTRLIST(), WATCH_SET(), WATCH_PTRSET(), WATCH_MAP(), WATCH_PTRMAP().

由于这是如何显示它们,所以该 PTR 较少的版本希望数据项(“T”)有流输出操作(operator <<).PTR 版本假定数据项是指向一些有 operator <<的类型的指针. WATCH_PTRMAP()假定仅仅值类型(“第二”)是指针,关键类型(“第一”)不是.(如果恰巧要使用指针作为关键安,那么为指针类型本身定义 operator <<.)

例如:

```
std::vector<int> intvec;

WATCH_VECTOR(intvec);

std::map<std::string,Command*> commandMap;

WATCH_PTRMAP(commandMap);
```

6.10.5 快照

snapshot()函数输出关于仿真的所有或选中对象的文本信息(包括通过用户在模块函数内的对象创建)至快照文件.

```
bool snapshot(cObject *obj = &simulation, const char *label = NULL);
```

函数可以从模块函数中类似的调用:

```
snapshot();      //转储整个网络

snapshot(this); //转储该模块及其所有的对象

snapshot(&simulation.msgQueue); //转储未来的事件
```

这将追回快照信息至快照文件的结尾.(快照文件名的扩展名为.sna,缺省是 omnetpp.sna.实际的文件名可以在配置文件中设置.)

快照文件输出足够详细,用于调试仿真:通过常规调用 snapshot(),我们可以追踪变量值和对象在仿真期间如何改变.参数: label 是一个字符串,在输出文件中显示; obj 是对象,其内部是我们关注的.缺省情况下,将写出整个仿真(所有模块等).

如果运行带 Tkenv 的仿真,也可以从菜单创建一个快照.

一个快照文件的例子:

```
[...]
(cSimulation) 'simulation' begin
  Modules in the network:
     'token' #1 (TokenRing)
       'comp[0]' #2 (Computer)
         'mac' #3 (TokenRingMAC)
         'gen' #4 (Generator)
         'sink' #5 (Sink)
       'comp[1]' #6 (Computer)
         'mac' #7 (TokenRingMAC)
         'gen' #8 (Generator)
         'sink' #9 (Sink)
       'comp[2]' #10 (Computer)
         'mac' #11 (TokenRingMAC)
         'gen' #12 (Generator)
         'sink' #13 (Sink)
end
(TokenRing) 'token' begin
  #1 params     (cArray) (n=6)
  #1 gates      (cArray) (empty)
  comp[0]          (cCompoundModule,#2)
  comp[1]          (cCompoundModule,#6)
  comp[2]          (cCompoundModule,#10)
end
(cArray) 'token.parameters' begin
  num_stations (cModulePar) 3 (L)
  num_messages (cModulePar) 10000 (L)
  ia_time      (cModulePar) truncnormal(0.005,0.003) (F)
  THT          (cModulePar) 0.01 (D)
  data_rate    (cModulePar) 4000000 (L)
  cable_delay  (cModulePar) 1e-06 (D)
```

```
end
[...]
(cQueue) `token.comp[0].mac.local-objects.send-queue' begin
  0-->1          (cMessage) Tarr=0.0158105774 ( 15ms) Src=#4 Dest=#3
  0-->2          (cMessage) Tarr=0.0163553310 ( 16ms) Src=#4 Dest=#3
  0-->1          (cMessage) Tarr=0.0205628236 ( 20ms) Src=#4 Dest=#3
  0-->2          (cMessage) Tarr=0.0242203591 ( 24ms) Src=#4 Dest=#3
  0-->2          (cMessage) Tarr=0.0300994268 ( 30ms) Src=#4 Dest=#3
  0-->1          (cMessage) Tarr=0.0364005251 ( 36ms) Src=#4 Dest=#3
  0-->1          (cMessage) Tarr=0.0370745702 ( 37ms) Src=#4 Dest=#3
  0-->2          (cMessage) Tarr=0.0387984129 ( 38ms) Src=#4 Dest=#3
  0-->1          (cMessage) Tarr=0.0457462493 ( 45ms) Src=#4 Dest=#3
  0-->2          (cMessage) Tarr=0.0487308918 ( 48ms) Src=#4 Dest=#3
  0-->2          (cMessage) Tarr=0.0514466766 ( 51ms) Src=#4 Dest=#3
end
(cMessage)  'token.comp[0].mac.local-objects.send-queue.0-->1' begin
  #4 --> #3
  sent:          0.0158105774 ( 15ms)
  arrived:       0.0158105774 ( 15ms)
  length:        33536
  kind:          0
  priority:      0
  error:         FALSE
  time stamp:    0.0000000 ( 0.00s)
  parameter list:
    dest         (cPar) 1 (L)
    source       (cPar) 0 (L)
    gentime      (cPar) 0.0158106 (D)
end
[...]
```

很有可能快照格式的文件在 OMNet++未来发布的版本转换为 XML 格式.

6.10.6 断点

仅用于 sctivity()!在那些用户界面中支持调试,断点停止执行以及可以检查的仿真状态.

在源代码中插入调用 breakpoint()来设置一个断点:

```
for(;;)
{
    cMessage *msg = receive();
    breakpoint("before-processing");
    breakpoint("before-send");
    send( reply_msg, "out" );
    //..
}
```

在用户界面不支持调试的情况下, breakpoint()调用可以简单忽略.

6.10.7 获得协同程序栈的用途

正确选择模块的栈大小非常重要.如果栈太大,会不必要地消耗内存;如果太小,会有栈破坏行为发生.

从 Feb99 发布, OMNeT++包括检测栈溢出的机制.它在栈边界检查字节模式的完好性,如果它被重写的话,就报告”栈破坏”.这个机制通常工作的很好,但是偶尔也会被欺骗—不是完全用于—局部变量(比如, char buffer[256]):如果字节模式落入这些局部变量的中间,它会被完整存储, OMNeT++不检测栈破坏.

为了更好地猜测堆栈大小,可以调用 stackUsage(),会告诉你模块实际使用了多少栈.最方便的是从 finish()调用:

```
void FooModule::finish()
{
  ev << stackUsage() <<  "bytes of stack used\n";
}
```

值包括由用户界面库添加的额外的栈(见 envir/omnetapp.h 中的 extraStackforEnvir),目前 Cmdenv 为 8K, Tkenv 至少 16K.

[实际值是依赖于平台的.]

在堆栈区域,stackUsage()检查存在的预定义字节模式,所以其也受上述局部变量影响

6.11 派生新类

6.11.1 cObject 或非 cObject?

如果计划实现一个完全的新类(相对于已经在 OMNeT++描述的子类),必须询问是否新建的类是基于 cObject.注意,我们不是说要始终是 cObject 的子类.这两个解决方案都各有优缺点,必须每个类单独考虑.

cObject 已经支持(或提供了一个框架)的重要功能是,是否相关于特定的目的. cObject 子类通常表示写更多的代码(因为你必须重新定义一定的虚拟函数,并遵守约定),并且你的类将有一点超重.然而,如果需要地 OMNet++对象中存储对象,类似 cQueue,或者想在对象中存储

OMNet++类,那么必须是 cObject 的子类.

[为了简便,在这部分里的”OMNet++对象”应该被理解为” cObject 子类的对象”.]

cObject 最重要的特性是名称字符串(必须存储在某处,因而有其自身的代价)和所有权管理(见[6.12]节),有优点但也需要一些代价.

作为一个普通的规则,小的类结构体类,比如 IPAddress, MACAddress, RoutingTableEntry, TCPConnectionDescriptor 等非 cObject 子类也比较好.如果类至少有一个虚拟成员函数,考虑 cPolymorphic 子类,由于它根本没有成员数据,仅有虚拟函数,所以没有任何额外的代价.

6.11.2 cObject 虚拟方法

在仿真类库中的大多数类都是继承 cObject.如果想从 cObject 或 cObject 派生类派生一个新类,必须重新定义一些成员函数,因而新建的类型对象可以与仿真系统的其它部分合作.在这里表示这些函数的完整列表.不需要担心列表的长度:大多数函数不是绝对需要实现的.例如,不需要重新定义 forEachChild(),除非是一个容器类.

以下的方法必须实现:

- 构造器:至少应该提供两个构造器:一个参数为对象名字符串 const char *(推荐使用),另一个是无参数的(必须出现).这两个通常实现为一个单一方法,其名称字符串缺省为 NULL.
- 拷贝构造器:类必须有以下的形式 X: X(const X&).拷贝构造器是用于复制一个对象.通常实现拷贝构造器是初始化所接收到的其它对象的具有 name (name())基类.
- 析构器.
- 复制函数, cPolymorphic *dup() const.需要创建并返回对象的副本.通常是单行函数,协助实现新的操作符和拷贝构造器.
- 分配操作符:即类 X 的 X& operator=(const X&).它应该复制其它对象的内容至一个这个类中,除了名称字符串.如果对象包含指向其它对象的指针应该如何做见后面.

如果类包括其它 cObject 子类的对象,通过指针或成员数据,必须实现以下的函数:

- Iteration 函数, void forEachChild(cVisitor * v).这个实现应该为每个所包含的对象调用函数通过指针或成员数据传递,见如果实现 forEachChild()的 cObject API 参考. forEachChild()使得可以在 Tkenv 中显示对象树,在其上执行搜索,等.它也被 snapshot()和其它的库函数使用.

以下方法是推荐实现的:

- 对象信息, std::string info().info()函数应该返回单行字符串描述对象的内容或状态. info()在 Tkenv 的许多地方显示.
- 详细对象信息, std::string detailedInfo().除了 info() 外,这个方法可能会被执行;它可以返回多行描述. detailedInfo()也可以在 Tkenv 的对象检查中显示.
- 序列化, netPack()和 netUnpack()方法.如果想要这种类型的对象通过分割被传输,这些方法需要并行仿真.

6.11.3 类注册

也需要使用 Register_Class()宏来注册新类.由 createOne()族函数使用,其可以创建给定名称字符串的任何对象. createOne() 由 Envir 库实现 omnetpp.ini 选项时使用,如 rng-class="..." 或 scheduler-class="...".(见[13]章)

例如,一个 omnetpp.ini 条目如:

rng-class="cMersenneTwister"

会导致一些类似以下的代码执行用于创建 RNG 对象:

cRNG *rng = check_and_cast<cRNG*>(createOne("cMersenneTwister"));

但是对这点工作,我们需要在代码中的某处有以下一行:

Register_Class(cMersenneTwister);

createOne()也需要由并行分布仿真特性([12]章),在接收方创建空白对象排列.

6.11.4 详细信息

我们将通过一例子来描述详细信息.我们创建一个新类 NewClass,重新定义以上所提及的 cObject 成员函数,解释规定,规则,以及与其相联的.尽可能多的显示,类将包括一个 int 成员数据,动态分配非 cObject 数据(双精度数组),作为成员数据的 OMNeT++对象(cQueue),和动态分配的 OMNet++对象(cMessage).

以下是类描述.它包含前面所讨论的所有方法.

```
//
// file: NewClass.h
//
#include <omnetpp.h>


class NewClass : public cObject
{
  protected:
    int data;
    double *array;
    cQueue queue;
    cMessage *msg;
    ...
  public:
    NewClass(const char *name=NULL, int d=0);
    NewClass(const NewClass& other);
    virtual ~NewClass();
    virtual cPolymorphic *dup() const;
    NewClass& operator=(const NewClass& other);
    virtual void forEachChild(cVisitor *v);
    virtual std::string info();
```

```
};
```

我们将讨论方法的实现.这是.cc 文件的顶部:

```
//
// file: NewClass.cc
//
#include <stdio.h>
#include <string.h>
#include <iostream.h>
#include "newclass.h"
Register_Class( NewClass );
NewClass::NewClass(const char *name, int d) : cObject(name)
{
    data = d;
    array = new double[10];
    take(&queue);
    msg = NULL;
}
```

构造器(上面的)使用对象名调用基类构造器,然后初始化其自身的成员数据.对于基于 cObject 的成员数据需要调用 take().

```
NewClass::NewClass(const NewClass& other) : cObject(other.name())
{
    array = new double[10];
    msg = NULL;
    take(&queue);
    operator=(other);
}
```

复制构造器依赖于分配操作符.由于规定中分配操作符不复制 name 成员,这里传递至基类构造器.(或者,我们可以在函数主体内写 setName(other.name()).)

注意,指向成员的指针在调用分配操作符之前必须被初始化 (为 NULL 或一个分配的对象/内在),避免破坏.

对于基于 cObject 的成员数据需要调用 take().

```
NewClass::~NewClass()
{
    delete [] array;
```

```
    if (msg->owner()==this)
        delete msg;
}
```

析构器应该删除所有对象分配的数据结构.基于 cObject 的对象如果是对象所拥有的,那么应该被删除.详细讨论在[6.12]节.

```
cPolymorphic *NewClass::dup() const
{
    return new NewClass(*this);
}
```

dup()函数通常只有一行,类似以上.

```
NewClass& NewClass::operator=(const NewClass& other)
{
    if (&other==this)
        return *this;
    cObject::operator=(other);

    data = other.data;

    for (int i=0; i<10; i++)
        array[i] = other.array[i];

    queue = other.queue;
    queue.setName(other.queue.name());

    if (msg && msg->owner()==this)
        delete msg;
    if (other.msg && other.msg->owner()==const_cast<cMessage*>(&other))
        take(msg = (cMessage *)other.msg->dup());
    else
        msg = other.msg;
    return *this;
}
```

拷贝和复制对象复杂相联的是分配操作符的浓缩,因此它通常需要 cObject 要求的所有方法中的大部分.

如果不想实现对象拷贝和复制,应该实现分配操作符来调用 copyNotSupported() -- 如果函数被调用,那么它抛出一个异常来终止带有错误消息的仿真.

分配操作符拷贝了其它对象的内容,除了名称字符串.应该总是返回*this.

首先,我们应该确定我们不尝试拷贝对象本身,因为它可能具有破坏性.如果这样(即&other==this),我们将立即返回,不做任何操作.

基类部分是通过调用基类的分配操作符实现拷贝的.

新的成员数据以普通的 C++方法拷贝.如果类包括指针,将最大可能的包括他们所指向的数据的深度拷贝,而不是仅仅拷贝指针值.

如果类包括指向 OMNet++对象的指针,需要考虑所有权.如果对于包括的对象没有所有权,那么我们假定它是指向”外部”对象的指针,然后我们仅拷贝指针.如果有其所有权,那么我们复制,并且拥有新建的对象.所有权分配将在[6.12]节详细介绍.

```
void NewClass::forEachChild(cVisitor *v)
{
    v->visit(queue);
    if (msg)
        v->visit(msg);
}
```

forEachChild()函数应该为类中每个 obj 成员调用 v->visit(obj).更多 forEachChild()信息见 API 参考.

```
std::string NewClass::info()
{
    std::stringstream out;
    out << "data=" << data << ", array[0]=" << array[0];
    return out.str();
}
```

info()方法应该产生的一个关于对象的精确的,单行字符串.应该不超过 40-80 个字符,由于字符串应该在工具提示和列表框中显示.

cPolymorphic 和 cObject 的虚拟函数的更多信息见类库参考.Sim 库的源(include/, src/sim/)可以进行进一步的例子.

6.12 对象所有权管理

6.12.1 所有权树

OMNeT++有一个内置的所有权管理机制,用于有为的判断检测,作为支持 Tkenv 观察的基础部分.

容器类比如 cQueue 拥有插入其内的对象所有权.但是这不局限于插入容器的对象:每个基于 cObject 的对象始终都有一个拥有者.从用户的观点看,所有权是透明的.例如,当创建一个新 cMessage 类时,它将由简单模块拥有.当发送它里,它将首先被移交至 FES,然后到达时,移交至目的简单模块.当你封装消息至另一个时,封装消息将成为所有者.当你再次解封时,当前活动的简单模块为其拥有者.

在 cObject 中定义的 owner()方法,返回对象的拥有者:

```
cObject *o = msg->owner();

ev << "Owner of " << msg->name() << " is: " <<

   << "(" << o->className() << ") " << o->fullPath() << endl;
```

另一个方向,列举对象所有者可以由 forEachChild()方法实现,遍历所有包括的对象并且检查其每个对象的所有者.

我们为什么需要这样?

对象所有权的传统概念是与”正确删除”对象相联系的.此外,追踪所有者及对象拥有的列表也可以用 OMNet++中其它目的:

- 使得方法,如 fullPath()可以被实现.
- 防止一定的设计错误类型,即那些与错误所有权处理相关的.
- 使得 Tkenv 显示当前在简单模块内的仿真对象列表.这对于寻找内存泄漏引起的忘记删除不再需要的消息非常有用

通过所有权管理,一些设计错误的例子可能被捕捉:

- 试着发出一条仍在队列中的消息,封装在另一条消息内,等等.
- 试着发送/调度一条仍由仿真内核所有的消息(如调度一个未来发生的事件)
- 在相同时间试着发送非常相同的消息对象至多个目的端(如所有链接模块)

例如, send()和 scheduleAt()函数检查消息发送/调度必须由模块所拥有.如果不是的话,那么发送设计错误信号:消息可能由其它模块所拥有(之前已经发送?)或当前调度,或在队列内部,一条消息或一些其它对象 — 在这些情况下,模块没有任何权力.当得到一条错误消息("not owner of object"),需要仔细检查错误消息:哪个对象有消息的所有权,为什么是这样,然后可能需要在设计的某地方修改逻辑.

以上的错误在代码中很容易出现, 其通常很难追查,如果不自动检测,会引起随机的灾难.当然,也有一些同类的错误仍然不能自动检测,像调用已经发送(目前保存的)至其它模块的消息的成员函数.

6.12.2 管理所有权

所有权管理对用户透明,但是这个机制必须由分割类本身支持.它将有助于寻找内部的 cQueue 和 cArray,由于他们会给出一个提示当你使用非 OMNet++容器类来存储消息或其它基于 cObject 的对象时,哪些动作需要你去实现.

插入

cArray 和 cQueue 有内部数据结构(数组和链接列表)来存储将要插入其中的对象.然而,他们不需要拥有所有这些对象(是否他们拥有一个对象可以从对象的 owner()指针来确定.)

cQueue 和 cArray 缺省的动作是获得对象的所有权插入.这个动作可以通过 takeOwnership 标志来改变.

这里是 cQueue (或 cArray)插入操作所要做的:

- 插入对象至内部数组/列表数据结构
- 如果 takeOwnership 为 true,那么获得对象的所有权,否则与其原始的所有者分离.

相应的源代码:

```
void cQueue::insert(cObject *obj)
{
    // insert into queue data structure
    ...
    // take ownership if needed
    if (takeOwnership())
        take(obj);
}
```

移除

这里是 cQueue (或 cArray)中移除操作所要做的:

- 从内部的数组/列表的数据结构中移除对象.
- 如果对象实际是由 cQueue/cArray 所拥有的,那么释放对象的所有权,否则仅离开当前的拥有者.

在对象从 cQueue/cArray 移除后,可以再使用它,或如果它不再需要的话,可以删除.
释放所有权短语需要进一步解释.当你从一队列或一个数组中移除一个对象,所有权需要移至简单模块的局部对象列表中.这是由 drop()函数完成的,其传输所有权至对象的缺省拥有者. defaultOwner()是一个虚拟函数,返回在 cObject 中定义的 cObject*,其实现返回当前执行的简单模块的局部对象列表.
例如, cQueue 的 remove()方法的实现类似如下:
[在 src/sim 中实际的代码的结构略有不同,但是意思是一样的]

```
cObject *cQueue::remove(cObject *obj)
{
    // remove object from queue data structure
    ...
    // release ownership if needed
    if (obj->owner()==this)
        drop(obj);
    return obj;
}
```

析构器

所有权的概念是用户有唯一的权利,负责删除其所拥有的对象.例如,如果删除一个包括 cMessage 的 cQueue,其包括的所有的消息和所有权都将被删除.

析构器也删除所有对象分配的数据结构.从包括的对象中,仅有拥有的对象被删除 - 即,其中 obj->owner()==this.

对象拷贝

所有权机制也必须考虑当 cArray 或 cQueue 对象被复制的情况.复制假定有与原始对象相同的内容,然而问题是,是否包括的对象也应该被复制或仅有他们的指针传递给复制的 cArray 或 cQueue.

cArray/cQueue 的规定是仅有拥有的对象被复制,包括但不拥有的对象将复制他们的指针,和其原来的拥有者关系不改变.

事实上,相同的问题出现在三个地方:分配操作符 operator=(),拷贝结构体和 dup() 方法.在 OMNet++中,规定是拷贝在分配操作符中实现,其它两个仅也依赖于此.(拷贝构造器仅构造一个空的对象并调用分配,而 dup() 仅实现为一个新的 cArray(*this)).

7 构建仿真程序

7.1 概述

正如已经提及的,一个 OMNet++模型一般由以下几部分组成:

- NED 语言拓扑描述.这些文件后缀为.ned.
- 消息定义,在后缀为.msg 的文件中.
- 简单模块实现和其它 C++代码,在.cc 文件(或 Windows 中的.cpp 文件)

建立一个可执行的仿真程序,首先需要使用 NED 编译器(nedtool)和消息编译器(opp_msgc)将 NED 文件和消息文件翻译为 C++.这步之后,处理就类似于从源文件构建任何 C/C++程序:所有 C++源文件需要被编译为目标文件(Unix/Linux 中的.o 文件, Windows 中的.obj 文件),所有的目标文件需要链接所需的库来得到一个可执行的文件. Unix/Linux 和 Windows 库文件名不同,静态库和共享库也不同.我们假定有一个称为 Tkenv 的库.在 Unix/Linux 系统中,静态库的文件名应该类似于 libtkenv.a(或 libtkenv.a.<version>),共享库应该被称之为 libtkenv.so (或 libtkenv.so.<version>).Windows 版的静态库应该是 tkenv.lib, DLL 文件(这相当于 Windows 共享库)命名为 tkenv.dll.

需要链接以下的库:

- 仿真内核和类库,称为 sim_std (file libsim_std.a, sim_std.lib, etc).
- 用户界面.用户界面的公共部分是 envi 库(libenvir.a 文件等),具体的用户界面是 tkenv 和 cmdenv (libtkenv.a, libcmdenv.a 等).必须与 envir 链接,外加 tkenv 或 cmdenv.

幸运的是,不需要担心以上的细节,由于自动工具如 opp_makemake 将为你考虑这部分困难.

下图给出了一个构建,运行仿真程序处理的概述.

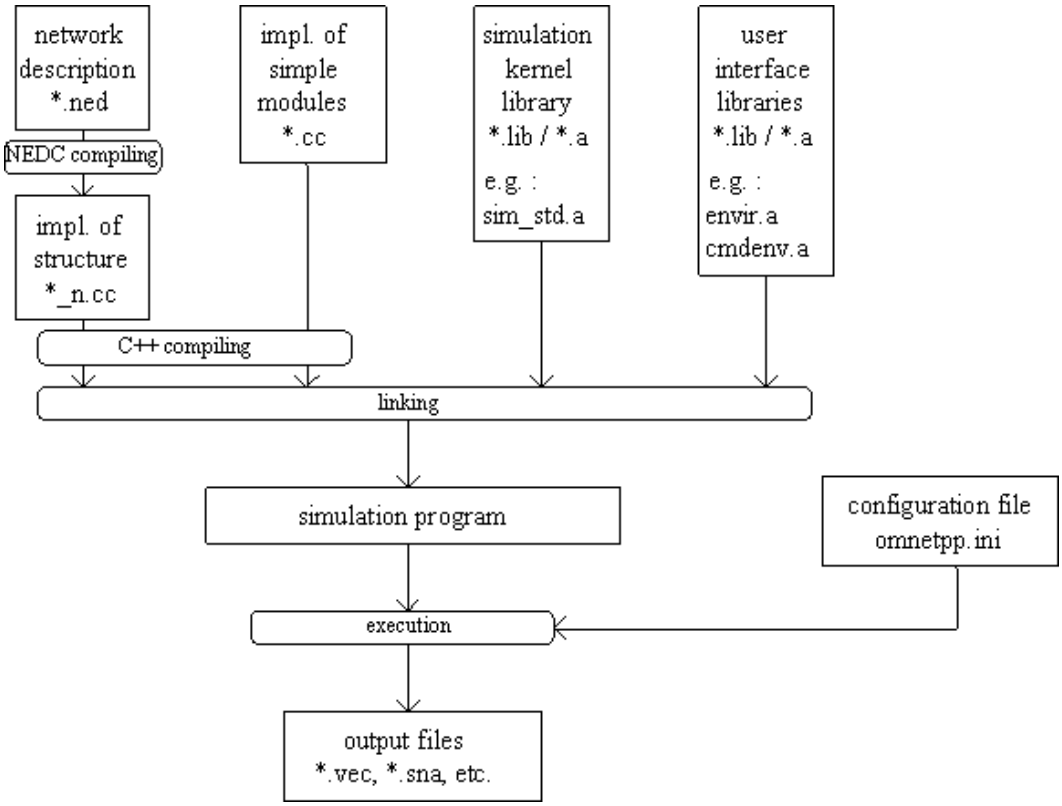


图: 构建和运行仿真程序

这部分讨论了如何在以下的平台上使用仿真系统:

- 具有 gcc 的 Unix (或 Windows 中的 Cygwin 或 MinGW)
- Windows 上的 MSVC 6.0

7.2 使用 Unix 和 gcc

这部分在 Linux, Solaris, FreeBSD 和 Unix 系列,以及 Windows 中的 Cygwin 和 MinGW 上应用 OMNeT++.

在手册里,我们仅能给出一个粗略的概述.OMNet++安装的 doc/目录包含自述.<platform>文件提供了最新的,更详细更精确的用法说明.

7.2.1 安装

安装过程依赖于所采用的分布状态(源文件,预编译的 RPM 等),从一个版本至另一个版本会改变,因此参考自述文件比较好.如果编译了源代码,可以作为普通的 GNU 程序:make ./configure.

7.2.2 构建仿真模型

opp_makemake 脚本可以基于当前目录的源文件,自动产生仿真程序的 Makefile.(它也可以处理大的分散在多个目录中的模型,这稍后在本节描述.)

opp_makemake 有许多选项,-h 选项显示摘要.

% opp_makemake -h

一旦在目录中有源文件(*.ned, *.msg, *.cc, *.h),进入那个目录然后输入:

% opp_makemake

这将创建一个 Makefile 文件.因此,如果简单的输入 make, 那么会构建仿真程序.可执行的名称与包括文件的目录名相同.

刚创建的Makefile不包括依赖,较可取的做法是通过输入 make 依赖添加他们.在依赖产生期间的警告处理可以很安全的忽略.

除了仿真可执行文件外, Makefile 也包括其它的对象(target).这里是较重要的对象列表:

Target	Action
	缺省的 target 是构建仿真可执行文件
depend	添加或更新 Makefile 中的依赖
clean	删除所有由 make 处理产生的文件
makefiles	使用 opp_makemake 重新生成 Makefile(这很有用,如在 OMNeT++更新后,或 opp_makemake 改变之后)
makefile-ins	类似于 make makefiles, 但是它重新产生的 Makefile.in 来取代

如果在目录中已经有了一个 Makefile, opp_makemake 将拒绝覆盖.可以使用-f 选项强制覆盖旧的 Makefile.

% opp_makemake -f

如果有问题,可以在配置脚本中检查路径定义(include 文件和库文件的位置等),如果需要则改正它们.然后重新运行,使改变生效.

可以使用-u 选项指定用户界面(Cmdenv/Tkenv),如果没有-u 的话,缺省是 Tkenv.

% opp_makemake -u Tkenv

或:

% opp_makemake -u Cmdenv

输出文件名的设置,使用-o 选项(缺省是目录名):

% opp_makemake -o fddi-net

如果你的一些源文件会是从其它文件(例如,可以使用产生的 NED 文件)产生的,那么写你的 make 规则至 makefrag 文件.当你运行 opp_makemake,它将自动将 makefrag 插入至最终的 makefile.使用-i 选项,也可以指定其它文件包括进 Makefile.

如果想使模型更轻便,可以产生 Makefile.in 取代 Makefile,使用 pp_makemake 的-m 选项.然后可以使用类 autoconf 的配置脚本来产生 Makefile.

7.2.3 多目录模型

在大的项目情况下,源文件会分散在多个目录中.必须决定是否要使用静态链接,共享或运行时加载(共享)库.这里我们讨论静态链接.

在每个包括源文件的子目录中(如 app/ 和 routing/)运行

opp_makemake -n

-n 选项表示不需要任何链接,仅完成编译.

在非叶节点的目录的,运行

opp_makemake -r -n

-r 选项允许递归 make:当你构建仿真时,make 会进入子目录,并且也在其内运行 make.缺省情况下,-r 进入所有子目录;-X 目录选项可以使 make 忽略一定的子目录.

如果包括其它目录的文件,需要使用-I 选项. -I 选项可以用于 C++和 NED 文件.在我们的例子中,可以运行

opp_makemake -n -I../routing

在 app/ 目录中, 反之亦然.

构建一个可执行文件,可以使用-w 选项;它导致使用包括(-I)目录中的所有对象来构建仿真可执行文件:

opp_makemake -w -I../routing -I../app

可以在 makefrag 文件(makefrag.vc)中添加子目录之间的依赖来影响构建的顺序.

例如一个使用 opp_makemake 的复杂例子,检查 INET 框架的 Makefiles,或者更确切地说,包含产生 makefile 命令的 makemake 脚本(和 makemake.bat 文件).

7.2.4 静态与共享 OMNet++系统库

缺省的链接使用共享库.使用静态链接的一个原因是调试具有共享库的 OMNet++类库较困难.另一个原因可能是在其它机制上运行可执行文件,不需要担心设置 LD_LIBRARY_PATH 变量(其中应该包括 OMNet++共享库的目录名).

如果要使用静态链接,在 configure.user 脚本中查找

build_shared_libs=yes

这一行,并且改变为

build_shared_libs=no

然后必须重新运行配置脚本,重建仿真:

./configure

make clean

make

7.3 使用 Windows 和 Microsoft Visual C++

这里仅是一个粗略的概述.最新的,更详细更精确的用法说明可以在你的 OMNet++安装的 doc/目录中找到,在 Readme.MSVC 文件中.

7.3.1 安装

最简单的是二元开始的安装版本.它包含所有需要的软件,除了 MSVC,你可以非常快速地使工作系统的启动和运行.

稍后,你可能也想下载和构建源分布.原因是你可能编译不同标志的类库,调试它们,或重新编译所支持的额外包(如 Akaroa, MPI).在你从 configuser.vc 文件中得到不同组件的正确目录后,汇编应该是不困难的(仅需要单个 nmake -f Makefile.vc 命令).编译需要的额外的软件也在 doc/下描述了.

7.3.2 在命令行构建仿真模块

OMNeT++有一个名为 makefile 的自动 MSVC makefile 创建器,这很容易完成.其使用非常类似于 Unix 的命名工具.

如果在模块的源文件目录运行 opp_nmakemake,它将收集目录中所有的源文件名,并且为它们创建一个 makefile.最终的 makefile 称为 Makefile.vc.

为了使用 opp_nmakemake,打开一个命令窗口(Start menu -> Run... --> type cmd),然后 cd 进入模型的目录,并且输入:

opp_nmakemake

opp_nmakemake 有多个命令行选项,与 Unix 版本极类似.

然后可以构建程序,通过输入:

nmake -f Makefile.vc

最公共的问题是 nmake(MSVC 的一部分)不能被找到,因为其不在目录中.可以通过运行 vcvars32.bat 来固定这个,这个可以在 MSVC 的 bin 目录下找到(通常是 C:\Program Files\Microsoft Visual Studio\VC98\Bin).

7.3.3 从 MSVC IDE 构建仿真模型

也可以使用 MSVC IDE 开发. 通常最好是拷贝一个简单仿真开始.

如果想使用编译的 NED 文件(如反对动态 NED 加载,在[8.3]描述),你需要添加 NED 文件至项目中,使用 Custom Build Step 命令来为其调用 NED 编译器(nedtool).你也需要添加由 nedtool 产生的_n.cc 至项目中.有一个 AddNEDFileToProject 宏,可以明确地执行这个任务:添加一个 NED 文件和相应的_n.cc 文件,配置 Custom Build Step.

一些变量(更多请阅读 doc/Readme.MSVC!):

- 如何获得图形化环境.缺省情况,如果从 IDE 重建的话,该样本仿真与 Cmdenv 链接.要改变至 Tkenv,从菜单选择 Build|Set active 配置,选择``Debug-Tkenv''或``Release-Tkenv'',然后重新链接可执行文件.
- 无法找到可用的 init.tcl.如果收到这个消息,Tcl/Tk 漏了 TCL_LIBRARY 变量,这通常是由安装程序设置的.如果见到这消息,需要自己在 Tcl lib/目录下设置这个变量.
- 改变编译器设置.改变 OMNeT++ 2.2:需要异常处理,打开 RTTI,栈大小设置为低于 64K.原理和更多信息见自述文件.
- 添加 NED 文件.在添加了一个.nde 文件至项目中后,也必须添加一个_n.cpp 文件,并且为其设置一个 Custom Build Step.
- 描述: NED Compiling $(InputPath)
- 命令: nedtool -s _n.cpp $(InputPath)
- 输出: $(InputName)_n.cpp

对于 msg 文件也需要一个类似的步骤.

- 文件扩展名:MSVC6.0 不能识别.cc 文件为 C++源程序.可以选择.cpp 为扩展名,可以通过改变相应的注册项来明确地改变 MSVC.进行 WEB 搜索明确找出需要改变的.

8 配置运行仿真

8.1 用户界面

OMNeT++仿真可以在不同的用户界面运行.目前,支持两个用户界面:

- Tkenv: 基于 Tcl/Tk 的图形化窗口用户界面.
- Cmdenv: 用于批处理的的命令行用户界面.

通常在 Tkenv 下测试和调试仿真,使用 Cmdenv 从命令行或 shell 脚本运行实际的仿真实验. Tkenv 更适合于教育或示范.

Tkenv 和 Cmdenv 都是以库的形式提供,在仿真可执行文件中在两者之间选择链接.(创建可执行文件在第[7]章描述.)两个用户界面都支持 Unix 和 Windows 平台.

Tkenv 和 Cmdenv 公共的性能被收集并放置在 Envir 库中,对于两个用户界面可以被认为是”公共基类”.

用户界面与仿真内核分离,并为两部分交互提供了一个定义好的接口.这也表示,如果需要可以写自己的用户接口或在自己的应用程序中不改变任何模型或仿真库,嵌入 OMNeT++仿真.

仿真的配置和输入数据通常在 omnetpp.ini 配置文件中描述.这文件中的一些项仅应用于 Tkenv 或 Cmdenv,其它有效的设置项与用户界面无关.两个用户接口也有命令行参数.

以下各节解释了 omnetpp.ini 和用户接口的公共部分,详细描述 Cmdenv 和 Tkenv,然后再描述具体的问题.

8.2 配置文件: omnetpp.ini

8.2.1 举例

一开始让我先看一个简单的 omnetpp.ini 文件,可以用于在 Cmdenv 下运行 Fifo1 样本仿真.

```
[General]

network = fifonet1

sim-time-limit = 500000s

output-vector-file = fifo1.vec

[Cmdenv]

express-mode = yes

[Parameters]

# generate a large number of jobs of length 5..10 according to Poisson (泊松)

fifonet1.gen.num_messages = 10000000

fifonet1.gen.ia_time = exponential(1)

fifonet1.gen.msg_length = intuniform(5,10)

# processing speeed of queue server

fifonet1.fifo.bits_per_sec = 10
```

这个文件组合了[General], [Cmdenv] 和[Parameters],每个都包括了一些条目. [General]部分应用了 Tkenv 和 Cmdenv,在这种情况下的条目指定了被仿真的网络名为 fifonet1, 仿真运行 500,000 秒,并且向量结果应该写入 fifo1.vec 文件.[Cmdenv]条目告诉 Cmdenv 全速运行仿真,并且打印关于仿真进展更新的周期.[Parameters]条目分配了不能在 NED 文件里得到的(或输入值)参数值.

以”#”或”;”开头的行是注释.

当使用 Cmdenv 构建样本时,通过在命令提示行输入 fifo1(或在 Unix 上,./ fifo1),就可以看到类似以下的界面.

```
OMNeT++ Discrete Event Simulation  (C) 1992-2003 Andras Varga

See the license for distribution terms and warranty disclaimer

Setting up Cmdenv (command-line user interface)...


Preparing for Run #1...

Setting up network `fifonet1'...

Running simulation...

** Event #0         T=0.0000000   ( 0.00s)   Elapsed: 0m  0s   ev/sec=0

** Event #100000    T=25321.99 ( 7h  2m)    Elapsed: 0m  1s   ev/sec=0

** Event #200000    T=50275.694 (13h 57m)   Elapsed: 0m  3s   ev/sec=60168.5

** Event #300000    T=75217.597 (20h 53m)   Elapsed: 0m  5s   ev/sec=59808.6

** Event #400000    T=100125.76 ( 1d  3h)   Elapsed: 0m  6s   ev/sec=59772.9

** Event #500000    T=125239.67 ( 1d 10h)   Elapsed: 0m  8s   ev/sec=60168.5

...

** Event #1700000   T=424529.21 ( 4d 21h)   Elapsed: 0m 28s   ev/sec=58754.4

** Event #1800000   T=449573.47 ( 5d  4h)   Elapsed: 0m 30s   ev/sec=59066.7

** Event #1900000   T=474429.06 ( 5d 11h)   Elapsed: 0m 32s   ev/sec=59453

** Event #2000000   T=499417.66 ( 5d 18h)   Elapsed: 0m 34s   ev/sec=58719.9

<!> Simulation time limit reached -- simulation stopped.


Calling finish() at end of Run #1...

*** Module: fifonet1.sink***

Total jobs processed: 9818

Avg queueing time:    1.8523

Max queueing time:    10.5473

Standard deviation:   1.3826


End run of OMNeT++
```

使用 Cmdenv 运行仿真,周期性地打印当前事件的序列号,仿真时间,消逝的(真实)时间,以及仿真结果(每秒处理多少事件;开始的两个值为 0,由于还没有足够的数据让其计算).在仿真结束时,运行简单模块的 finish()方法,然后从其输出显示.在我的机制上运行花了 34 秒.这

个 Cmdenv 可以通过 omnetpp.ini 的条目进行定制.输出文件 fifo1.vec 包括在仿真期间记录的向量数据(这里,是排队时间),可以使用 Plove 或其它工具处理.

8.2.2 仿真运行的概念

OMNeT++可以一个接一个的自动运行多个仿真.如果选择了多个运行,选项设置和参数值可以由每个运行单独给出,或为所有运行一起给出,这依赖于要出现哪部分的选项或参数.

8.2.3 文件语法

ini 文件是一个文本文件,包括组合不同部件的项.不用关心各部件之间的顺序.如果有两个相同名称的部件(比如在文件中[General]出现两次)将会合并.

以”#”或”;”开头的行是注释,在处理期间被忽略.

长的行可以打断使用反斜线符号:如果最后一个字符是”\”,它将合并下一行.

ini 文件的大小(部件和条目的数量)是不受限的.当前限制行的长度为 1024 字符,其不会因为在行中使用反斜线打断而增长.这种限制在以后发布的版本中可能会升高.

例如:

```
[General]

# this is a comment

foo="this is a single value \

for the foo parameter"

[General]  # duplicate sections are merged

bar="belongs to the same section as foo"
```

8.2.4 文件包含的内容

OMNeT++支持一个 ini 文件包含另一个 ini 文件,通过 include 关键字.这个特性允许你分割大小 ini 文件为一些逻辑单元,固定的和变化的部分等.

例如:

```
# omnetpp.ini

...

include parameters.ini

include per-run-pars.ini

...
```

也可包含其它目录的文件.如果包含的 ini 文件里也包含其它的文件,它们的路径名将被认为相对于包含引用的文件目录,而不是仿真当前的工作目录.这个规则也适用于在 ini 文件中出现的其它文件名(比如 preload-ned-files=, load-libs=, bitmap-path=, output-vector-file=, output-scalar-file= 等条目,和 xmldoc()模块参数值.)

8.2.5 部件

可以存在以下部件:

部件	描述

[General]	包含一般的设置,应用于所有仿真运行和所有用户接口.细节见[8.2.6].
[Run 1], [Run 2], ...	包含每次运行的设置.这里面可能包含任何在其它部件中的条目.
[Cmdenv]	包含指定 Cmdenv 的设置.细节见[8.7.2]
[Tkenv]	包含指定 Tkenv 的设置. 细节见[8.8.2]
[Parameters]	包含不能从 NED 文件内获得值的模块参数值.细节见[8.4]
[OutVectors]	配置记录的输出向量.可以通过向量名和仿真时间进行过滤(开始/停止记录).细节见[8.5].

8.2.6 [General] 部件

[General]部件最重要的选项是以下的:

- network 选项选择安装运行的模型
- 仿真的长度可以由 sim-time-limit 和 cpu-time-limit 选项进行设置(可以使用的一般的时间单位 ms, s, m, h 等).
- 输出文件名可以通过以下选项设置:output-vector-file, output-scalar-file 和 snapshot-file.

支持选项完全列表如下.几乎这些选项中的每个也都可以放入[Run n]部件.每个运行的设置优先全局设置.

名称和缺省值	描述
[General]	
ini-warnings = yes	当允许时, OMNeT++列出 ini 文件的条目名使用的缺省值.这个有时可以有益于调试 ini 文件.
preload-ned-files =	列出动态加载的 NED 文件 (见[8.3]).
network =	用于仿真的网络名
snapshot-file = omnetpp.sna	快照文件名.每次调用 snapshot()的结果将被追加到该文件.
output-vector-file= omnetpp.vec	输出向量的文件名.
output-scalar-file= omnetpp.sca	输出标量的文件名.
pause-in-sendmsg = no	仅进行按步班的执行.如果允许的话, OMNeT++将 send()调用分为两步.
sim-time-limit =	仿真期间的仿真时间.
cpu-time-limit =	仿真期间的现实时间.
num-rngs = 1	产生随机数的数量.
rng-class= "cMersenneTwister"	使用的 RNG 类.可以是"cMersenneTwister", "cLCG32", 或"cAkaroaRNG",或者使用自己的 RNG 类(必须是 cRNG 的子类)

seed-N-mt =, seed-N-lcg32 =	为 cMersenneTwister 和 LCG32 RNGs 指定种子(代替 RNG 中的 N 为数字:0,1,2…);缺省是自动选择种子. 过时的 random-seed= 和 gen0-seed=, gen1-seed=等已经不再使用.
total-stack-kb =	在千字节指定总的堆栈大小(所有协同堆栈的总和).如果等到``Cannot allocate coroutine stack...'' 的错误,需要增长这个值.
debug-on-errors = false	当设置为 true,运行时错误将引起仿真程序错误,断点进入 C++调试器(如果仿真运行下一个,或只启动及时调试).一旦进入调试器,就可以查看堆栈跟踪或审查变量.
load-libs =	共享库列表(由空格分隔)加载进初始化阶段. OMNeT++追加一个指定的平台扩展至库名:Windows 系统的.dll 和 Unix 系统的.so.这个特性可以用于动态加载 Envir 扩展(RNGs, 输出向量管理器等)或简单模块.例如: load-libs = "../lib/rng2 ../lib/ospfrouting"
perform-gc = false	如果为 true, 仿真内核通过简单模块析构器,在网络清除阶段删除,仿真对象不删除.不推荐,由于这某些情况下会引起破坏.见[4.3.5]
print-undisposed = true	当 perform-gc 为 false 时(缺省设置).它选择是否对象由简单模块析构器删除,应该由仿真内核报告.
output-scalar-precision=12	调整记录进输出标量文件的有意义的位数,见[6.9.3]的讨论.
output-vector-precision=12	调整记录进输出标量文件的有意义的位数,见[6.9.3]的讨论.
fname-append-host = false	打开这个,会导致主机名和处理 ID 追加至输出文件名(比如,omnetpp.vec, omnetpp.sca).这对于并行分布式的仿真特别有用([12]章).
parallel-simulation = false	允许并行分布式仿真 (见 [12]章).
scheduler-class = cSequentialScheduler	部分 Envir 插件机制:选择调度类.这个插件接口允许实现实时,硬件循环,分布式和分布式并行仿真.类必须实现在 envirext.h 中定义的 cScheduler 接口.更详细见[13.5.3]节.
configuration-class = cInifile	部分Envir插件机制:从获得的所有配置中选择类.即,这个选项可以取代一些其它实现的 omnetpp.ini,比如数据库输入.仿真程序必须从 omnetpp.ini 中引导程序(其中包括配置类设置).该类必须实现在 envirext.h 中定义的 cConfiguration 接口.更多细节见[13.5.3]
outputvectormanager-class = cFileOutputVectorManager	部分Envir插件机制:选择使用的输出向量管理类来从输出向量记录数据.该类必须实现在 envirext.h 中定义的接口 cOutputVectorManager。更多细节见[13.5.3]
outputscalarmanager-class = cFileOutputScalarManager	部分Envir插件机制:选择使用的输出标量管理类来记录数据传递给 recordScalar()。该类必须实现在 envirext.h 中定义的 cOutputScalarManager 接口。更多细节见[13.5.3]

snapshotmanager-class = cFileSnapshotManager	部分 Envir 插件机制:选择类来处理流，snapshot()写其输出文件。该类必须实现 envirext.h 中定义的 cSnapshotManager 接口。更多细节见[13.5.3]

8.3 动态加载 NED

OMNeT++ 3.0 之前，NED 文件必须通过 NED 编译器转换为 C++，编译和链接至仿真程序。从 OMNeT++ 3.0 之后，可以使用动态 NED 加载，表示一个仿真程序当其开始运行时，可以在运行时加载 NED 文件 — 不再需要编译 NED 文件至仿真程序。这导致更多的灵活性，也节省了模型开发时间。

关键是 omnetpp.ini 配置文件的[General]部件中的 preload-ned-files=配置项。这个选项需要列出当仿真程序开始时需要加载的 NED 文件名。

例如：

```
[General]

preload-ned-files = host.ned router.ned networks/testnetwork1.ned
```

也可以使用通配符：

```
[General]

preload-ned-files = *.ned networks/*.ned
```

也可以使用带@符号的文件列表：

```
[General]

preload-ned-files = *.ned @../nedfiles.lst
```

其中 nedfiles.lst 文件包含 NED 文件列表，每个一行，类似：

```
transport/tcp/tcp.ned

transport/udp/udp.ned

network/ip/ip.ned
```

Unix 寻找命令通常是非常方便的方法来创建列表文件（尝试 find . -name '*.ned' > listfile.lst）。

而且，列表文件也可以包括通配符，参考其它的列表文件：

```
transport/tcp/*.ned

transport/udp/*.ned

@moreprotocols.lst
```

文件给出的相对路径是相对于文件列表位置的（不是当前工作目录）。即，以上例子中 transport 目录和 moreprotocols.lst，认为与 nedfiles.lst 在相同的目录，不管当前的工作目录。

但必须要注意的是，加载的 NED 文件可能包括任何数量的模块，信道，以及任何数量的网络。不管你是否使用全部或其中一部分仿真。你可以使用 network= omnetpp.ini 条目，选择任何时候在加载文件中的网络，以及其加载的每个模块，信道等。网络将被成功设置。

8.4 在 omnetpp.ini 中设置模块参数

仿真通过模块参数得到输入，可以在 NED 文件或 omnetpp.ini 中分配一个值 - 以这个顺序（NED 再 omnetpp.in）。由于在 NED 文件中分配的操作符不能在 omnetpp.ini 中覆盖，可以将其考虑为“硬编码”。相反，在 omnetpp.ini 中可以更加容易，更加灵活在维护模块参数。

在 omnetpp.in 中，模块参数通过它们的完全路径或层次名引用。这个名称包括以点分隔的模块名列表（从顶层模块往下至模块包含的参数），加上模块名（见[6.1.5]）。

omnetpp.in 的一个例子，其中设置 toplevel 模块的 numHosts 参数和 server 模块的 transactionsPerSecond 参数：

```
[Parameters]

net.numHosts = 15

net.server.transactionsPerSecond = 100
```

8.4.1 指定的 Run 和 general 部件

模块参数的值可以放入[Parameters]或 [Run 1], [Run 2]等 ini 文件的部件。指定的 run 设置优先于整体设置。

虽然 run 由数字标识，也可以为它们分配短的说明标签，这些将在 Tkenv 的 run 选择对话框显示。仅需在[Run x]头部放置一行 description="some text"。

omnetpp.ini 的一个例子（#之后的为注释）：

```
[Parameters]

net.numHosts = 15

net.server.transactionsPerSecond = 100


[Run 1]

description="general settings"

# uses settings from the [Parameters] section


[Run 2]

description="higher transaction rate"

net.server.transactionsPerSecond = 150  # overrides the value in [Parameters]

# net.numHosts comes from the [Parameters] section


[Run 3]

description="more hosts and higher transaction rat e"

# override both setting in [Parameters]

net.numHosts = 20

net.server.transactionsPerSecond = 150
```

8.4.2 使用通配符模式

模型可以有大量的配置参数，很难在 omnetpp.ini 中一个一个的设置。OMNeT++支持通配符模式，允许一次设置多个参数。

符号与通常的整体式模式有些变化。与通常使用的规则最大的区别是*和**，而且字符的范围应该写在大括号内而不是方括号（即，任何字母是{a-zA-Z}而不是 [a-zA-Z],因为方括号已经是模块向量索引预留的符号）。

模式语法

- ？:匹配任何字符，除了点（.）
- * : 匹配 0 个或多个字符，除了点（.）
- ** : 匹配 0 个或多个字符(任何字符)
- {a-f} : 设置：匹配一个在 a-f 范围内的字符
- {^a-f}: 否定设置：匹配一个不在 a-f 范围内的字符
- {38..150} : 数字范围：在 38..150 范围的任何数字（比如 99）
- [38..150] : 索引范围： 任何在方括号中 38..150 范围内的数字（比如[99]）
- 反斜线 (\) : takes away the special meaning of the subsequent character

优先权

如果使用通配符，条目的顺序是重要的：如果一个参数名匹配多个通配模式，使用第一次发生的匹配。这表示，需要使用列出的具体设置的第一个，更综合的在稍后。应该在最后覆盖所有的设置。

一个 ini 文件的例子：

```
[Parameters]

*.host[0].waitTime = 5ms   # specifics come first

*.host[3].waitTime = 6ms

*.host[*].waitTime = 10ms  # catch-all comes last
```

*与**

通配符*是匹配在路径名中单个模块或参数名，而**可以用于匹配路径中的多个部件。例如，**.queue*.bufSize 匹配在模型中，名称是以 queue 开始的任何模块的 bufSize 参数，而 *.queue*.bufSize 或 net.queue*.bufSize 仅直接选择在网络层的 queue。注意 **.queue**.bufSize 也匹配 net.queue1.foo.bar.bufSize。

设置与否定设置（Sets, negated sets）

设置与非设置可以包括多个字符范围和字符枚举。例如{_a-zA-Z0-9}匹配任何字母和数字，加上下划线；{xyzc-f}仅匹配字符 x, y, z, c, d, e, f.为了在集合中包括'-'，将其放入，但不能解释为范围。例如{a-z-}或{-a-z}.如果要在集合中包括'}'，必须是第一个字符：{}a-z}或一个否定设置：{^}a-z}.一个反斜线也作为设置定义中的反斜线符号（不是作为转义字符）。

数字范围和索引范围

仅可匹配非负的整数。范围的开始或结束（或两者）可以省略：{10..}, {..99}或{..}都是有效的范围（最后一个匹配任何数字）。规范中必须明确使用两个点(.)。注意*{17..19}将匹配 a17, 117 和 963217，因为*也可以匹配数字！

一个数值范围的例子：

```
[Parameters]

*.*.queue[3..5].bufSize = 10

*.*.queue[12..].bufSize = 18

*.*.queue[*].bufSize = 6  # this will only affect queues 0,1,2 and 6..11
```

兼容性

在 OMNeT++3.0 版本之前,不存在**通配符,*也匹配点.这表示早期 OMNet++版本的 ini 文件在 OMNet++3.0 中的意义不同 — 因此,ini 文件必须更新.实际上,每个以*开头的行,都应改变为以**开始 — 大多数的时候就做这个,很少需要做更一步的调整.

如果仍然想运行旧版本的 omnetpp.ini(例如,阻止新的版本),可以添加行:

```
#% old-wildcards
```

在每个旧的 ini 文件的顶部.这将转换回旧版本的执行行为.由于#% old-wildcards 仅提供从 OMNeT++ 2.3 转换为 3.0,将在以后的版本中删除.

8.4.3 缺省应用

也可以在 NED 文件中利用缺省值指定输入(缺省值). <parameter-name>.use-default=yes 设置分配缺省值给参数,如果在 NED 文件中没有缺省值,则为 0,false 或空字符串.

以下的例子设置 hostA 的 ip 模块 ttl(生存时间)为 5,而网络中所有其它的节点将通过 input()得到 NED 文件中的缺省值.

```
[Parameters]

**.hostA.ip.ttl = 5

**.ip.ttl.use-default = yes
```

To make use of all defaults in NED files, you'd add the following to omnetpp.ini:

```
[Parameters]

**.use-default = yes
```

8.5 配置输出向量

做为一个仿真程序是不断变化的,它有能力收集越来越多的统计数据.输出向量的大小可以很容易在达到数十或上百兆字节大小,但很多时间,仅有部分记录的数据用于分析.

在 OMNeT++中,可以控制如果将 cOutVector 对象记录数据至磁盘.你可以打开或关闭输出向量或者分配结果收集间隔.输出向量配置在 ini 文件的[OutVectors]部件中给出.缺省情况下,所有输出向量是打开的.

输出微量可以使用如下的语法进行配置：

```
module-pathname.objectname.enabled = yes/no

module-pathname.objectname.interval = start..stop

module-pathname.objectname.interval = ..stop

module-pathname.objectname.interval = start..
```

对象是字符串通过其构造器中的 cOutVector 或 setName()成员函数传递.

```
cOutVector eed("End-to-End Delay");
```

启动或停止值可以是在 NED 或配置文件中指定的时间(比如, 10h 30m 45.2s).

同参数名,可以在对象名和模块路径名中使用通配符.

例如:

```
#
# omnetpp.ini
#


[OutVectors]
**.interval = 1s..60s
**.End-to-End Delay.enabled = yes
**.Router2.**.enabled = yes
**.enabled = no
```

以上的配置限制所有输出向量的收集间隔为 1s..60s,禁止输出向量的收集,除了端到端延迟和在 Router2 模块中.

8.6 配置随机数生成器

在[6.4]节已经给出 OMNeT++随机数结构的概述.这里,我们将讨论 omnetpp.ini 中 RNG 配置.

8.6.1 RNG 数量

num-rngs=配置项设置仿真模型可用的随机数生成实例(如随机数流)的数量(见[6.4]).引用一个 RNG 数大于或等于这个数(来自简单模块或 NED 文件)将引起运行时错误.

8.6.2 RNG 选择

rng-class=配置项设置使用的随机数生成器类.缺省为"cMersenneTwister", Mersenne Twister RNG.其它可用的类为"cLCG32"(OMNet++2.3 和之前版本留下的 RNG,循环长度为 2^{31}-2)和"cAkaroaRNG"(Akaroa 的随机数生成器,见[8.10]).

8.6.3 RNG 映射

在简单模块中使用的 RNG 数可以从 omnetpp.ini 任意映射实际的随机数流(实际的 RNG 实例).映射允许 RNG 使用和随机数流的配置—甚至对于没有写 RNG 的仿真模型有更大的灵活性.

RNG 映射可以在 omnetpp.ini 中指定.配置项的语法如下:

```
 [General]
<modulepath>.rng-N=M  (where N,M are numeric, M<num-rngs)
```

这个映射本地模块 RNG N 为物理的 RNG M.以下的例子映射了所有产生的模块的缺省(N=0)RNG 为物理的 RNG 1,所有噪音模块的缺省(N=0)RNG 为物理 RNG 2.

```
[General]
num-rngs = 3
```

```
**.gen[*].rng-0 = 1

**.noisychannel[*].rng-0 = 2
```

这个映射允许变量约简技术应用于 OMNet++模块,不需要任何模块改变或重编辑.

8.6.4 自动种子选择

如果不在 omnetpp.ini 中明确指定种子, RNG 将使用自动种子选择.自动和手动的种子选择可以一起存在:对于特殊的仿真,一些 RNG 可以手动配置,一些也可以自动配置.

自动种子选择机制使用两个输入:运行数量(比如[Run 1], [Run 2]等的部件名的数量),和 RNG 数量.对于相同的运行数量和 RNG 数量, OMNeT++对于任何仿真模型选择相同的种子值.如果运行数量或 RNG 数量不同, OMNeT++最好是选择不同的种子,充分区别 RNG 的序列从而生成不重叠的序列.

运行数量可以在 omnetpp.ini(通过[Cmdenv]/runs-to-execute=项)或命令行指定:

```
./mysim -r 1

./mysim -r 2

./mysim -r 3
```

对于 cMersenneTwister 随机数生成器,选择种子, 由于 RNG 极长的序列,因而可以很容易地生成不重叠的序列.RNG 使用 32 位的种子值初始化 seed = runNumber*numRngs + rngNumber.(这表示参与研究的仿真运行应该有相同的 RNG 数量设置).

[而就我们所知,没有人证明种子 0,1,2,...排除在序列之外的, 由于 MT 极长序列,这可能是真的.然而,笔者在论文中关注于发布的 MT 种子选择.]

对于 cLCG32 随机数生成器,情况更加复杂,由于 RNG 的范围是相当短的(2^{31}-1, 约 20 亿).对这个 RNG,OMNet++使用一个 256 预生成种子的表格,在 RNG 的序列中等间隔,表中索引使用 runNumber*numRngs + rngNumber 公式计算.应该注意索引不应该超过 256 个,或它将覆盖,再次使用相同的种子.最好不使用 cLCG32—cMersenneTwister 各方面都优于它.

8.6.5 手动种子配置

在一些情况下,可能想手动配置种子值.这样做的原因是可能要使用变量约简技术,或可能多个仿真运行使用相同的种子.

对于 cLCG32 RNG, OMNeT++提供了一个单独的程序来产生种子值(seedtool 在[8.6.6]节讨论),可以在 ini 文件中指定这些种子值.

以下的 ini 文件明确初始化了两个随机数生成器,并且每个运行使用不同的种子值:

```
[General]

rng-class=cLCG32  # needed because the default is cMersenne Twister

num-rngs = 2


[Run 1]

seed-0-lcg32 = 1768507984

seed-1-lcg32 = 33648008
```

```
[Run 2]
seed-0-lcg32 = 1082809519
seed-1-lcg32 = 703931312
...
```

为了手动设置 Mersenne Twister RNG 种子(这很少需要这样做),使用 seed-0-mt=, seed-1-mt=等设置:

```
[General]
num-rngs = 2

[Run 1]
seed-0-mt = 1317366363
seed-1-mt = 1453732904

[Run 2]
...
```

为所有运行设置种子值,在[General]中添加一些需要的种子项.

8.6.6 选择好的种子值: 利用 seedtool

seedtool 程序可以为 cLCG32 RNG 选择好的种子.当不使用命令行参数启动时,程序输出如下帮助:

```
seedtool - part of OMNeT++/OMNEST, (C) 1992-2004 Andras Varga
See the license for distribution terms and warranty disclaimer.

Generates seeds for the LCG32 random number generator. This RNG has a
period length of 2^31-2, which makes about 2,147 million random numbers.
Note that Mersenne Twister is also available in OMNeT++, which has a
practically infinite period length (2^19937).

Usage:
  seedtool i seed          - index of 'seed' in cycle
  seedtool s index         - seed at index 'index' in cycle
  seedtool d seed1 seed2   - distance of 'seed1' and 'seed2' in cycle
  seedtool g seed0 dist    - generate seed 'dist' away from 'seed0'
  seedtool g seed0 dist n - generate 'n' seeds 'dist' apart, starting at 'seed0'
```

```
  seedtool t              - generate hashtable

  seedtool p              - print hashtable
```

最后两个选项 p 和 t 在内部用于产生一个预计算种子的哈希表,大大加快速度的工具.在实际使用中,g 选项最重要.假定有四个仿真运行每个都需要两个独立的随机数生成器,启动它们的种子值至少是 10,000,000.第一个种子可以简单的为 1.可以输入以下的命令:

```
C:\OMNETPP\UTILS> seedtool g 1 10000000 8

The program outputs 8 numbers that can be used as random number seeds:

1768507984

33648008

1082809519

703931312

1856610745

784675296

426676692

1100642647
```

可以在 ini 文件中指定这些种子值.

8.7 Cmdenv: 命令行接口

命令行用户接口是小的,可移植的,快速用户接口,可以在所有平如上编译运行. Cmdenv 设计的基本目的是用于批处理.

Cmdenv 简单运行在配置文件中描述的一些或所有仿真.如果一个运行由于错误消息停止,后面的将仍然执行.运行可以通过命令行参数或在 ini 文件里执行传递.

8.7.1 命令行参数

Cmdenv 构建的仿真程序接受以下的命令行参数:

-h	程序输出可执行文件的一个简短帮助消息和包含的网络,然后退出.
-f <fileName>	指定配置文件的名称.缺省是 omnetpp.ini.可以给出多个-f 参数;可以允许分割配置文件.例如,一个文件可以包括你的普通设置,另一个可能包括大多数的模块参数,另一个为经常改变的模块参数.
-l <fileName>_n.o	加载一个共享对象(Unix 中的.so 文件).可以使用多个-l 参数.你的.so 文件可以包括模块代码等.通过动态加载所有简单模块代码和编译的网络描述(Unix 上的文件),可以在源文件中,可以在每次改变之后的不再需要重新链接仿真程序.(共享库的创建为 gcc -shared...)
-r <runs>	它指定了哪些 run 应该执行(比如-r 2,4,6-8).这个选项重载了 ini 文件中的[Cmdenv] 的 runs-to-execute=选项.

所有其它的选项都是从配置文件中读取的.

使用-h 标志运行 OMNeT++可执行文件的例子:

```
% ./fddi -h
```

```
OMNeT++/OMNEST Discrete Event Simulation  (C) 1992-2005 Andras Varga
See the license for distribution terms and warranty disclaimer
Setting up Tkenv...

Command line options:
  -h            Print this help and exit.
  -f <inifile>  Use the given ini file instead of omnetpp.ini. Multiple
                -f options are accepted to load several ini files.
  -u <ui>       Selects the user interface. Standard choices are Cmdenv
                and Tkenv. To make a user interface available, you need
                to link the simulation executable with the cmdenv/tkenv
                library, or load it as shared library via the -l option.
  -l <library>  Load the specified shared library (.so or .dll) on startup.
                The file name should be given without the .so or .dll suffix
                (it will be appended automatically.) The loaded module may
                contain simple modules, plugins, etc. Multiple -l options
                can be present.

Tkenv-specific options:
  -r <run>      Set up the given run, specified in a [Run n] section of
                the ini file.

The following components are available:
  module types:
    FDDI_Monitor
    FDDI_Generator4Sniffer
    FDDI_Generator4Ring
    ...

End run of OMNeT++
```

8.7.2 Cmdenv ini 文件选项

Cmdenv 可以在两种模式下执行,通过 express-mode ini 文件项来选择:

- Normal (非 express)模式用于调试:详细信息将写入标准输出文件(事件标志,模块输出等).
- Express 模式可以用于长期仿真运行:仅仅显示关于仿真进度的周期状态更新.

Cmdenv 所接受的 ini 文件选项完整列表:

选项和缺省值	描述
[Cmdenv]	
runs-to-execute =	指定哪个仿真 run 应该执行.它接受 run 成员或 run 成员范围逗号分隔的列表.比如 1,3-4,7-9.如果忽略值, Cmdenv 执行 ini 文件中所有的 run:如果在 ini 文件中没有指定的 run, Cmdenv 执行一个 run. -r 命令行选项覆盖这个 ini 文件设置.
express-mode=yes/no (default: no)	选择 ``normal'' (调试/跟踪) 或 ``express'' 模式.
module-messages=yes/no (default: yes)	仅在 normal 模式: 打印模块 ev<< 输出 on/off
event-banners=yes/no (default: yes)	仅在 normal 模式: 打印事件标志 on/off
message-trace=yes/no (default: no)	仅在 normal 模式: 打印一行关于每条发送的消息(通过 send(),scheduleAt()等)和标准输出上的传输.
autoflush=yes/no (default: no)	在每个事件标志或状态更新后调用 fflush(stdout);影响 express 和 normal 模式.打开 autoflush 可以用于打印式的调试,来追踪下面程序的失败原因.
status-frequency=<integer> (default: 50000)	仅在 express 模式:打印每 n 个事件的状态更新(就今天的计算机,针对一个典型的模式,将每隔几秒产生一个更新,也许每秒几次).
performance-display=yes/no (default: yes)	仅在 express 模:打印详细的实现信息.打开这个的话,在每次更新的时候会导致三行的项,包括 ev/sec, simsec/sec, ev/simsec,FES 中消息创建/仍然显示/当前调度的数量.
extra-stack-kb = 8	指定额外的栈数量,用于保存在 Cmdenv 下运行仿真时的每个 activity()简单模块

8.7.3 解释 Cmdenv 输出

当仿真在" express"模式下允许显示详细显示的运行时, Cmdenv 周期性地输出三行关于仿真处理的状态信息.输出类似:

```
...
** Event #250000   T=123.74354 ( 2m  3s)    Elapsed: 0m 12s
     Speed:     ev/sec=19731.6   simsec/sec=9.80713   ev/simsec=2011.97
     Messages:  created: 55532   present: 6553   in FES: 8
```

```
** Event #300000   T=148.55496 ( 2m 28s)    Elapsed: 0m 15s
     Speed:     ev/sec=19584.8   simsec/sec=9.64698   ev/simsec=2030.15
     Messages:  created: 66605   present: 7815   in FES: 7
...
```

状态显示的第一行(以**开头)包括:

- 到目前为止处理了多少事件
- 当前的仿真时间(T)和
- 从仿真启动开始到目前消逝的时间(墙上挂钟的时间).

第二行显示了关于仿真实现的信息:

- ev/sec 表示了性能:在一个实时秒内处理多少个事件.一方面它依赖于硬件(较快的 CPU 每秒处理更多的事件),另一方面,它依赖于事件相联的复杂性.例如,协议仿真中的每个事件往往需要更多的处理,例如比队列网络,因此后者产生更高的 ev/sec 值.在任何情况下,这个值依赖于模型中模块数量的大小.
- simsec/sec 显示了相对的仿真速度,即相对于实时时间如何进行快速的仿真,每个实时秒可以完成多少个仿真秒.这个虚拟值依赖于所有的:硬件,仿真模型的大小,事件的复杂性,和事件之间的平均仿真时间.
- ev/simsec 是事件密度:每个仿真秒有多少个事件.事件密度仅依赖于仿真模型,不管用于仿真的硬件:在一个 ATM 单元级仿真,可以有非常高的值(10^9),而在银行柜台仿真中,这个值可能低于 1.它也依赖模块的大小:如果在模型中模块的数量增加一倍,可以预计事件的密度也增加一倍.

第三行显示了消息数量,由于它表示了仿真的”健康”状况,所以很重要.

- Created:从仿真开始运行至目前,创建的消息对象的总和.这并不表示,这么多消息对象实际存在,由于他们中有些可能被删除了.这也不表示你创建了所有这些消息 — 仿真内核也创建其所使用的消息(比如在 activity() 简单模块中实现 wait()).
- Present:在仿真模块中当前显示的消息对象的数量,即消息创建的数量减去已经删除的消息数量.这个数字包括在 FES 中的消息.
- In FES:在未来事件中的当前调度的消息数量.

第二个值,消息显示的数量可能比开始想像的要重要.它可以表示仿真的”健康”状态:如果其稳定增长,那么会有内存泄漏和消息丢失(表示为一个程序错误),或仿真的网络超负荷,队列不断地填满(可能会显示为错误输入参数).

当然,如果消息数量不增长,不表示没有内存泄漏(其它的内存泄漏也有可能).然而值是有用的,由于目前为止在仿真中大多数公共的内存泄漏方法是不删除消息的.

8.8 Tkenv:图形化用户接口

特点

Tkenv 是一个可移植的图形化用户接口. Tkenv 支持仿真交互执行,追踪和调试. Tkenv 在仿真的开发阶段,或用于演示或研究是推荐使用的,由于它可以得到仿真执行期间任何点的详细仿真状态图,可以了解网络内部发生了什么.最重要的特点是:

- 消息流动画

- 仿真执行期间的统计(柱状图等)和输出向量的图形化显示
- 为每个模块的文本输出分割窗口
- 调度可以窗口查看的消息来进一步仿真
- 一个事件接一个事件,正常快递执行
- 标记断点
- 检查窗口以审视和改变模型的对象和变量
- 仿真可以重启
- 快照(详细报告模型:对象,变量等)

Tkenv 使得在执行期间可能查看仿真的结果(输出向量等).结果可以显示为柱状图和时序图.这可以加速检验仿真程序操作的正确性并提供一个好的环境在执行期间进行模型实验 .当与 gdb 或 xxgdb 一起使用的话, Tkenv 可以加速大量的调试.

Tkenv 使用 Tcl/Tk,可以工作在所有的平台上,其中 Tcl/Tk 已被移植到: Unix/X, Windows, Macintosh.在参考文献中列出的网页上可以得到更多关于 Tcl/Tk 的信息.

8.8.1 命令行参数

使用 Tkenv 构建的仿真程序可以接受以下的命令行参数:

-h 这个程序打印一个简短的帮助信息和可执行文件包括的网络,然后退出.

-f <fileName> 指定配置文件的名称.缺省是 omnetpp.ini.可以给出多个-f 参数;可以允许分割配置文件.例如,一个文件可以包括你的普通设置,另一个可能包括大多数的模块参数,另一个为经常改变的模块参数.

-l <fileName> 加载一个共享对象(Unix 中的.so 文件).可以使用多个-l 参数.你的.so 文件可以包括模块代码等.通过动态加载所有简单模块代码和编译的网络描述(Unix 上的_n.o 文件),可以在源文件中,可以在每次改变之后的不再需要重新链接仿真程序.(共享库的创建为 gcc -shared...)

-r <run-number> 它与[Tkenv]/default-run= ini 项相同的效果(但优先于其)

8.8.2 Tkenv ini 文件设置

Tkenv accepts the following settings in the [Tkenv] section of the ini file.

Tkenv 接受 ini 文件中的[Tkenv]部件以下的设置

项和缺省值	描述
[Tkenv]	
extra-stack-kb = 48	指定额外的栈数量(千字节),用于保存当仿真在 Tkenv 下运行时的每个 activity()简单模块.这个值显著地高于 Cmdenv 中类似的—处理 GUI 事件需要大量的栈空间.
default-run = 1	指定哪个 run Tkenv 应该在启动后自动安装.如果没有 default-run=项或值为 0,那么 Tkenv 将询问要安装哪一个.

以下的配置项是无用的,因为相应的设置也可以在 Tkenv GUI 的仿真选项对话框中获得,并且 GEI 设置优先. Tkenv 在当前目录的.tkenvrc 文件中存储设置 — ini 文件设置仅当没有.tkenvrc 文件的情况下使用.

和贵设定优先。 tkenv 商店设置在。 tkenvrc 文件在当前目录下-I NI 文件设置是否只用了，如果是否定的。 t kenvrc 档案。

项和缺省值	选项
[Tkenv]	
use-mainwindow = yes	允许/禁止写 ev 输出至 Tkenv 主窗口.
print-banners = yes	允许/禁止打印每个事件的标志.
breakpoints-enabled = yes	指定仿真是否应该在简单模块中每个 breakpoint()调用应该停止.
update-freq-fast = 10	在 Fast 执行模式中,两个显示更新之间的事件执行数量
update-freq-express = 500	在 Express 执行模式中,两个显示更新之间的事件执行数量
animation-delay = 0.3s	当减慢执行仿真时,两步骤之间的延迟.
animation-enabled = yes	允许/禁止消息流动画
animation-msgnames = yes	在消息流动画期间,允许/禁止显示消息名
animation-msgcolors = yes	在消息流动画期间,允许/禁止每种消息使用不同的颜色.
animation-speed = 1.0	指定消息流动画的速度.
methodcalls-delay =	调用动画方法后设置延迟.
show-layouting = true	查看网络图形处理的布局

8.8.3 使用图形化环境

Tkenv 中的仿真运行模式

Tkenv 中的仿真运行模式有以下几种:

- Step
- Run
- Fast run
- Express run

在 Tkenv 的工具条上有运行模式相应的按钮.

在 Step 模式中, 可以一个事件接一个事件的仿真执行.

在 Run 模式中,仿真进行所有追踪帮助运行.消息动画是活动的,检查窗口在每个事件之后被更新.输出消息在主窗口和模块输出窗口中显示.工具条上的 Stop 按钮可以停止仿真.当仿真运行的时候,可以完全与用户接口交互:可以打开检查器等.

在 Fast 模式中,仿真动画是关掉的.检查器和消息输出窗口在每 10 个事件之后更新(实际的数量设置可以在 Options|Simulation 选项中,也可以 ini 文件中).Fast 模式比 Run 模式快了好几倍.可以加速接近于 10(或配置事件数量).

在Express模式中,仿真运行与Cmdenv速度相同,禁止所有的追踪.模块输出不再记录在输出窗口.在一段时间内只能与仿真交互一次(缺省是 1000 个事件),因而用户界面的运行时花销是微乎其微的.如果想更新的话,可以直接按 Update 检查按钮.

Tkenv 有一个状态栏,当仿真运行是定期更新.测试显示类似于 Cmdenv 中,[8.7.3]详细描述.

检查器

在 Tkenv 中,对象可以通过检查器查看.首先,从菜单选择 Inspect|Network.用法应该是很明显的;仅需要双击,通过右击弹出菜单.在第一步, Run 和 Fast Run 模式,随机仿真的运行检查器自动更新.为了在 Tkenv 中显示普通的变量(int, double, char 等),在 C++代码中使用 WATCH()宏.

Tkenv 检查器也显示对象指针,也复制指针值至剪贴板.这对于调试是很重要的:当仿真在调试的情况下运行,如 gdb 或 MSVC IDE,可以粘贴对象指针至调试器,并仔细查看数据结构.

配置 Tkenv

在非标准安装的情况下,可能需要设置 OMNETPP_TKENV_DIR 环境变量,因而 Tkenv 可以找到 Tcl 脚本中的部件.

图标被加载的缺省路径可以使用 OMNETPP_BITMAP_PATH 改变,是一个以分号分隔的目录列表,缺省为 omnetpp-dir/bitmaps;./bitmaps.

在可执行文件中嵌入 Tcl 代码

Tkenv 的重要部分写入了 Tcl,在多个.Tcl 脚本文件中.缺省的脚本路径通过编译传递给 tkapp.cc,它可在运行时通过 OMNETPP_TKENV_DIR 环境变量覆盖.现存的单独的脚本库可以很方便地实现,将单独的可执行文件运用于不同的机制.为了解决这个问题,可以编译脚本部分至 Tkenv.

详细: tcl2c 程序(它的 C 源代码在 Tkenv 目录)用于将.tcl 文件翻译为 C 代码,其包括在 tkapp.cc 中.这是建立在 makefiles 中的,可以允许选择.

8.8.4 In Memoriam...

这用于其它可以从分布中移除的用户界面窗口:

- TVEnv. 一个基于 Turbo Vision 的用户界面.第一个与 OMNet++交互的 UI. Turbo Vision 的一个显著的特性是图形化窗口环境,通常在 Borland C++ 3.1 上运行.
- XEnv. 用纯 X/Motif 写的 GUI. 这是一个实验,写之前我们不管 Tcl/Tk 以及发现在 GUI 构建时巨大的生产力. XEnv 从没有更多的发展,因为在 Motif 中编程是非常非常慢的.

8.9 反复或循环仿真 run

一旦你们模型可靠地工作,通常要运行多个仿真.你可能要使用各种参数设置来启动模型,或要(应该要?) 运行相同参数设置但不同的随机数产生种子的相同模型,来统计更可靠的结果.

通过手动地很容易的多次运行仿真变得很单调,一个很好的方法就是写一个控制脚本,实现任务自动化. Unix shell 是一个自然语言选择写控制脚本,但是其它的语言像 Perl, Matlab/Octave, Tcl, Ruby 也可以达到这个目的.

下一章节仅对于 Unix 用户.我们将使用 Unix 的”Bourne” shell (sh, bash)来写控制脚本.如果你选择 Matlab/Octave, contrib/octave/目录下包括脚本样例(由 Richard Lyon 贡献).

8.9.1 执行多个 run

在简单有情况下,可以定义所有需要的仿真 run,在 omnetpp.ini 中的[Run 1], [Run 2]等部件中,每次使用-r 标志调用你的仿真. -f 标志可以使用与 omnetpp.ini 不同的文件名.

以下的脚本可以多次执行名为 wireless 的仿真,不同 run 的参数在 runs.ini 给定.

```
#! /bin/sh

./wireless -f runs.ini -r 1

./wireless -f runs.ini -r 2

./wireless -f runs.ini -r 3

./wireless -f runs.ini -r 4

...

./wireless -f runs.ini -r 10
```

为了运行以上的脚本,将其输入在文本文件中如 run,使用 chmod 给其 x 权限(可执行),然后可以通过输入./run 来执行:

```
% chmod +x run

% ./run
```

可以通过使用一个子 for 循环来简化以上的脚本.在下面的例子中,变量 i 通过在 in 关键字之后给出的列表值循环.这非常实用,由于可以离开或添加 run,或通过简单编辑列表改变 run 的序列—来说明这点,我们跳过 run 6,而包括 run 15 来取代.

```
#! /bin/sh

for i in 3 2 1 4 5 7 15 8 9 10; do

   ./wireless -f runs.ini -r $i

done
```

如果有许多的 run,可以使用 C 风格的循环:

```
#! /bin/sh

for ((i=1; $i<50; i++)); do

   ./wireless -f runs.ini -r $i

done
```

8.9.2 参数值的变化

在 ini 文件中手写所有 run 的描述可能不切实际,尤其是要设置许多的参数,或想尝试所有可能的两个或多个参数组合.解决方法可能是仅产生带可变参数的 ini 文件的一小部分,通过 ini 文件的包含来使用它.例如,可以写类似的 omnetpp.ini:

```
[General]

network = Wireless


[Parameters]

Wireless.n = 10
```

```
...   # other fixed parameters
include params.ini  # include variable part
```

和以下的控制脚本.它使用两个嵌套的循环来显示所有可能的 alpha 和 beta 参数的组合.注意 params.ini 有创建是通过使用>和>>重定向操作符,输出至文件中.

```
#! /bin/sh
for alpha in 1 2 5 10 20 50; do
   for beta in 0.1 0.2 0.3 0.4 0.5; do
       echo "Wireless.alpha=$alpha" > params.ini
       echo "Wireless.beta=$beta" >> params.ini
       ./wireless
   done
done
```

作为一个重型的例子,有一个 Joel Sherrill 文件系统仿真的”runall”脚本.也证明了循环可以遍历所有的字符串值,不仅仅是数字.(omnetpp.ini 包括产生的 algorithms.ini.)

注意取代了重定向每个 echo 命令至文件,他们使用圆括号组合重定向.效果是一样的,但这种方式可以节省一些输入.

```
#! /bin/bash
#
# This script runs multiple variations of the file system simulator.
#
all_cache_managers="NoCache FIFOCache LRUCache PriorityLRUCa che..."
all_schedulers="FIFOScheduler SSTFScheduler CScanScheduler..."

for c in ${all_cache_managers}; do
  for s in ${all_schedulers}; do
  (
    echo "[Parameters]"
    echo "filesystem.generator_type = \"GenerateFromFile\""
    echo "filesystem.iolibrary_type = \"PassThroughIOLibrary\""
    echo "filesystem.syscalliface_type = \"PassThroughSysCallIface\""
    echo "filesystem.filesystem_type = \"PassThroughFileSystem\""
    echo "filesystem.cache_type = \"${c}\""
    echo "filesystem.blocktranslator_typ e = \"NoTranslation\""
    echo "filesystem.diskscheduler_type = \"${s}\""
```

```
    echo "filesystem.accessmanager_type = \"MutexAccessManager\""
    echo "filesystem.physicaldisk_type = \"HP97560Disk\""
  ) >algorithms.ini


  ./filesystem
  done
done
```

8.9.3 种子值的变化(多个独立的 run)

如果想执行多个不同随机种子的 run,可以使用相同的控制脚本.以下的代码完成 500 个独立种子的 run(omnetpp.ini 应该包括 parameters.ini.)

种子是除序列里的(seedtool 参数)之外的 1000 万个数字,因此一个 run 不应该使用比这更多的种子,否则将在序列中重叠,将不能独立运行.

```
#! /bin/sh
seedtool g 1 10000000 500 > seeds.txt
for seed in `cat seeds.txt`; do
   (
     echo "[General]"
     echo "random-seed = ${seed}"
     echo "output-vector-file = xcube-${seed}.vec"
   ) > parameters.ini
   ./xcube
done
```

8.10 支持 Akaroa: 在并行中的多复制

8.10.1 介绍

典型的仿真是 Monte-Carlo:他们使用(伪)随机数来驱动仿真模型.为了使仿真能产生可靠的统计结果,必须谨慎地考虑以下几点:

- 什么时候初始的短暂状态结束,什么时候我们开始收集数据?当仿真仍然在准备阶段的时候,我们通常不想包括初始的短暂状态.
- 什么时候终止仿真?我们要等待足够长的时候,以便我们收集的统计信息是稳定的,可以达到可靠统计所需要的样本大小.

两个问题都没有得到充分的答案.可能仅建议等待”非常长”或”足够长”.然而,这既不简单(如果知道什么是”足够长”)也不实际(即使当今适度复杂的高速处理器仿真的也需要数小时,无法供应多个运行时间,比如说 10,”仅仅可以确保安全.”)如果需要进一步的证明,请阅读[Pawlikowsky02],将更加吃惊.

一个可能的解决方法是在仿真运行时查看统计信息,在运行时决定何时收集了足够的数据,达到了需要的结果精确性.其中一个可行的标准是由信任层次提供,更确切地说,由其相对平

均的宽度提供.但是事先并不知道要收集多少的观测数据才能达到这个层次 —它必须在运行时决定.

8.10.2 什么是 Akaroa

Akaroa [Akaroa99]解决了以上问题.根据其作者, Akaroa (Akaroa2)在集群环境下是一个完全自动的仿真工具,在 MRIP 环境下,设计运行分布式随机仿真.

MRIP 支持并行里的多复制.在 MRIP 中,集群中的计算机中独立运行整个仿真处理中的复本(比如对于相同参数不同种子的 RNG(随机数产生器)),产生统计相等的仿真输出数据流.这些数据流输出至一个负责最终结果分析的全局数据分析器,当结果达到满意的精确度时终止仿真.

独立仿真运行过程中相互独立,不断地发送他们的观察数据至集中的分析器和控制处理器.这个处理组合了独立数据流并从这些观测数据中计算每个参数的平均值估计 . Akaroa2 由一个给定的信任层次和精度决定是平否有了足够的观测数据 .当判断有了足够的观测数据则停止仿真.

akaroa2 决定由某一置信水平和精度是否有足够的意见或不是 。当法官表示，是有足够的观测，它停止了模拟。

如果使用 n 个处理器,需要执行的仿真时间通常比单处理器仿真小 n 倍(需要的观测数据的数量稍后产生).因此,仿真会按比例大约地加速至处理器的数量,有时甚至更多.

Akaroa 是由坎特伯雷大学在新西兰克赖斯特彻奇设计的,可以用来免费为教学和非营利性的研究活动。

8.10.3 OMNeT++使用 Akaroa

Akaroa

在仿真可以在 Akaroa 下并行运行之前,必须启动系统:

- 在一些主机上在后台运行 akmaster
- 想运行仿真引擎的每个主机,在后台运行 akslave

每个 akslave 与 akmaster 建立一个链接.

然后可以使用 akrun 来启动一个仿真， akrun 等待仿真完成,然后写结果报告至标准输出文件. akrun 的基本使用命令是:

```
akrun -n num_hosts command [argument..]
```

其中 command 是想要启动的仿真的名称. Akaroa 的参数是从工作目录下名为 Akaroa 的文件中读取的.从进程发送至 akmaster 的过程中收集数据,当达到需要的精度时 akmaster 通知仿真处理终止.结果被写入标准输出文件.

以上的描述并没有足够详细地帮助你安装并成功使用 Akaroa — 你需要阅读 Akaroa 手册.

为 Akaroa 配置 OMNeT++

首先,必须编译 OMNeT++允许支持 Akaroa.

OMNeT++仿真必须配置在 omnetpp.ini 中,从而它传递观测数据至 Akaroa.仿真模型本身并不需要改变—它继续写观测数据至输出向量(cOutVector 对象,见第[6]章).可以在 Akaroa 控制下放置一些输出向量.

需要添加以下的代码至 omnetpp.ini:

```
[General]

rng-class="cAkaroaRNG"

outputvectormanager-class="cAkOutputVectorManager"
```

这些行会使得仿真从 Akaroa 获得随机数，允许数据写至选择的输出向量，传递至 Akaroa 的全局数据分析器.

[更详细的插件机制使用，见[13.5.3].]

Akaroa 的 RNG 是组合多个递归的，周期大约为 2^{191} 的伪随机数产生器(CMRG)的随机数，并为每个仿真引擎提供唯一的随机数数流.从 Akaroa 获得随机数流是非常重要的，所有的仿真处理需要运行相同的 RNG 种子，并产生完全相同的结果！

那么需要指定哪个输出向量要在 Akaroa 的控制之下，默认是，所有的输出向量都在 Akaroa 控制之下；

```
<modulename>.<vectorname>.akaroa=false
```

这个设置会使 Akaroa 忽略指定的向量.这里可以使用*,**通配符(见[8.4.2]).例如，如果想要很少的向量在 Akaroa 之下，可以使用以下的技巧：

```
<modulename>.<vectorname1>.akaroa=true

<modulename>.<vectorname2>.akaroa=true

...

**.*.akaroa=false  # catches everything not mat ched above
```

使用共享文件系统

实际上通常在集群中的所有计算机上挂载相同的物理磁盘(比如通过 NFS 或 Samba).然而，由于所有的 OMNeT++仿真处理运行相同的设置，他们会覆盖彼此的输出文件(比如 omnetpp.vec，omnetpp.sca).可以通过使用 ini 文件中的 fname-append-host 项来防止发生覆盖：

```
[General]

fname-append-host=yes
```

当打开这个，将追加主机名至输出文件名(输出向量，输出标量，快照文件).

8.11 典型问题

8.11.1 堆栈问题

“Stack violation (FooModule stack too small?) in module bar.foo”

OMNeT++检测到模块使用大于其所分配的栈空间.解决方法是为模块类型增加栈.可以从 finish()调用 stackUsage()找出模块实际使用了多少栈.

“Error: Cannot allocate nn bytes stack for module foo.bar”

这个解决方法依赖于是在 Unix 还是在 Windows 上使用 OMNeT++.

Unix. 如果获得以上的消息，必须增加总的栈大小(所有协同工作的栈的总和).可以在 omnetpp.ini 中完成：

```
[General]

total-stack-kb = 2048 # 2MB
```

如果设置 total-stack-kb 太高的话,不会出错.建议设置为低于操作系统所允许的最大处理栈大小(ulimit -s;见下一节).

Windows.需要在链接器选项设置一个低的”预留栈大小”,例如将设置为 64K(/stack:65536 链接器标志).”预留的栈大小”是 Windows exe 文件的内部报头的一个属性.可以从链接器或利用微软的 editbin 设置.可以使用 opp_stacktool 程序(依赖于利用另一个微软的 dumpbin)来显示可执行文件的预留的栈大小.

需要使用一个低的预留栈大小,由于 Win32 Fiber API 是 activity()底层的机制,使用这个数字为协同工作的栈大小,默认是 1MB,可能很容易就用完 2GB 的地址空间(2GB/1MB=2048).

下面解释更多的细节.每个 fiber 有其自己的堆栈,默认是 1MB(即”预留栈空间” — 比如,预留在地址空间中,但未满 1MB 的实际上”表明”,比如有物理内存分配给它).这表示,2GB 地址空间将在 2048 个 fibers 之后用完,其太少了.(实际上,甚至不能创建如果多的 fiber,由于物理内存也是一个限制因素).因此, 1MB 预留栈大小(RSS)必须设置一个较小的值:模块需要的协同栈大小,加上 Cmdenv/Tkenv 的 extra-stack-kb 数量—这使得 Cmdenv 约有 16K,当使用 Tkenv 时大约 48K.不幸的是, CreateFiber() Win32 API 不允许指定 RSS.更高级的接受 RSS 为一个参数的 CreateFiberEx() API 不幸的是仅对 Windows XP.

可选的栈大小参数存储在 EXE 报头中,可以通过 STACKSIZE .def 文件参数,通过/stack 链接器选项,或在一个现有的可执行文件上使用 editbin /stack 设置.这个参数为主程序堆栈 fiber 和线程堆栈指定一个公共的 RSS.64K 应该足够了.这是一个方法应该创建仿真可执行文件:链接/stack:65536 选项,或/stack:65536 参数适用于使用 editbin 其后.例如,在应用 editbin /stacksize:65536 命令至 dyna.exe,可以在有 256M RAM 的 Win2K PC 机上成功地运行有 8000 个 Client 模块的 Dyna 样本(这表示运行时大约有 12000 个模块,包括大约 4000 个动态创建的模块.)

“Segmentation fault(分割故障)”

在 Unix 上,如果设置较高的总的栈大小,由于会超过操作系统限制的最大堆栈大小,会在网络安装期间(如果使用动态创建模块的话,在执行期间)产生分割故障.例如,在 Linux 2.4.x 里,默认的堆栈大小是 8192K (即 8MB). ulimit shell 命令可以用于修改资源限制,可以增加允许的最大栈大小至 64M.

```
$ ulimit -s 65500

$ ulimit -s

65500
```

仅由 root 用户可以进一步增加.资源限制是由子处理继承的.以下的序列可以用于 Linux 下得到一个有 256M 栈限制的 shell:

```
$ su root

Password:

# ulimit -s 262144

# su andras

$ ulimit -s

262144
```

如果在每次登录的时候通过以上的处理,可以在 PAM 配置文件中修改限制.在 Redhat Linux(或其它系统)添加以下的行至/etc/pam.d/login:

```
session    required    /lib/security/pam_limits.so
```

添加以下的行至/etc/security/limits.conf:

```
*    hard    stack    65536
```

更极端的解决方法是重编译较大栈限制的内核.编辑 /usr/src/linux/include/linux/sched.h,并增加_STK_LIM 从(8*1024*1024)至(64*1024*1024).

最终,与内存紧密相联,可以切换至 Cmdenv. Tkenv 的每个模块的堆栈大小大约增加了 32K,因而从简单模块的背景下调用的用户接口代码可以安全地执行. Cmdenv 不需要额外的堆栈.

最终...

一旦得到指向必须调整总堆栈大小的指针至你运行的程序,应该考虑转换(一些) activity()简单模块至 handleMessage().activity()不能用于测量大的仿真.

8.11.2 内存泄漏和破坏

C++中最公共的问题是与内存分配有关的(创建和删除的用法):

- 内存泄漏,即忘记删除不再使用的对象或内存块;
- 崩溃,通常由于引用一个已经删除的对象或内存块,或试着两次删除对象.
- 堆破坏(最终导致崩溃),由于超出分配块的限制,如在已经越过分配的数组的尾部写.

目前在仿真程序中最常见的内存泄漏方法是由于没有删除消息 (cMessage 对象或其子类). Tkenv 和 Cmdenv 都能显示当前仿真中的消息数量,如见[8.7.3].如果发现消息数量稳定增长,就需要寻找消息对象在哪.可以从 Tkenv 菜单选择 Inspect|From list of all objects... 来完成,审查弹出的对话框中的列表.(如果模型较大,可能需要过一段时间才弹出对话框.)

如果消息数量是固定的,仍然可能有其它基于 cObject 的对象的泄漏.可以通过使用 Tkenv 的 Inspect|From list of all objects... 函数.

如果泄漏非基于 cObject 的对象或仅仅内存块(结构体, int/double/struct 数组,等通过新建分配的),不能通过 Tkenv 找到.可能需要一个特殊的内存调试工具,比如下面所描述的一个.

内存调试工具

如果检查到有内存分配问题(与二次删除或访问已经删除的与会或内存泄漏相关的崩溃),可以使用特殊的工具来跟踪.

目前最有效,最健壮,并且最通用的工具是 Valgrind,原本是为调试 KDE 开发的.

其它的内存调试器是 NJAMD, MemProf, MPatrol, dmalloc 和 ElectricFence.以上的工具大多都支持跟踪内存泄漏以扩检测二次删除,超过分配的块的尾部写,等/

被证实的商业工具是 Rational Purify.它有一个好名声,并多次证明其有效性.

8.11.3 仿真迟缓执行

如果仿真执行比预期的要慢,该怎么做?这里指出的最好建议是,应该使用一个好的剖析器来找出程序中每部分花费的时间.不要错误地忽略这一步,认为你知道”哪部分是慢的”!即使对有经验的程序员来说,剖析对象经常充满惊奇.它通常证实,大量的 CPU 时间花在完全没必要有状态查看,而大且复杂的自满不没花引见预期有时间.不假定任何事情—在优化之前剖析!

[在抱怨仿真内核表现欠佳之前...]

在 Linux 上真正让人印象深刻的剖析器是基于 Valgrind 的 callgrind,其可视化观察器是 KCachegrind.不幸的是,它不能很快移植到 Windows 中.在 Windows 中,运气不大好—可以借助商业产品,或移植仿真至 Linux.后者通常比人们期望的更加顺利.

9 网络图形和动画

9.1 显示字符串

9.1.1 显示字符串语法

显示字符串在图形化用户界面中指定模块的排列与显示(目前仅 Tkenv):他们控制对象(复合模块,它们的子模块和链接)如何显示.显示字符串出现在 NED 描述的显示:短语.

显示字符串的格式是以分号相隔的标签列表.每个标签包括一个关键字(通常是一个字母),一个等号和一个逗号相隔的参数列表,如:

```
"p=100,100;b=60,10,rect;o=blue,black,2"
```

参数可以在尾部,也可以参数列表里忽略,如:

```
"p=100,100;b=,,rect;o=blue,black"
```

模块/子模块参数可以包括$name 符号:

```
"p=$xpos,$ypos;b=rect,60,10;o=$fillcolor,black,2"
```

可能显示字符串的对象是:

- 子模块—显示的字符串包括位置,排列(对于模块向量),图标,图标颜色,辅助图标,状态文本,通信范围(如圆或填充圆)等.
- 链接—显示的字符串可以指定位置,箭头颜色,箭头粗细.
- 复合模块—显示的字符串可以指定背景颜色,边框颜色,边框厚度
- 消息—显示字符串可以指定图标,图标颜色等.

以下的 NED 例子显示了代码中哪里可以放置显示字符串.

```
module ClientServer
    submodules:
        pc: Host;
            display: "p=66,55;i=comp"; // position and icon
        server: Server;
            display: "p=135,73;i=server1";
    connections:
        pc.out --> server.in
            display "m=m,61,40,41,28"; // note missing ":"
        server.out --> pc.in
            display "m=m,15,57,35,69";
```

```
    display: "o=#ffffff"; // affects background
endmodule
```

9.1.2 子模块显示字符串

下表列出了在子模块显示字符串中的标签：

标签	意义
p=xpos,ypos	在(xpos,ypos)像素位置放置子模块,原先是在封装模块的左上角. 默认:适当的,子模块不重叠的自动布局. 如果应用于子模块向量,自动选择圆环或行布局.
p=xpos,ypos,row,deltax	用于模块向量.在一行排列子模块,以(xpos,ypos)开始,保持 deltax 距离. 默认： 选择 deltax,因而子模块不重叠. row 简写为 r.
p=xpos,ypos,column,deltay	用于模块向量.在一列排列子模块,以(xpos,ypos)开始,保持 deltax 距离. 默认： 选择 deltax,因而子模块不重叠. column 简写为 col 或 c.
p=xpos,ypos,matrix,itemsperrow,deltax,deltay	用于模块向量.在一个矩阵中排列子模块,以(xpos,ypos)开始,最大的 itemsperrow 子模块在一行,保持 deltax 和 deltay 距离. 默认：itemsperrow=5,选择 deltax 和 deltay,因而子模块不重叠. matrix 简写为 s.
p=xpos,ypos,ring,width,height	用于模块向量.在一个椭圆中排列子模块,椭圆的左上角限定在(xpos,ypos),具有 width 和 height. 默认:选择 width 和 height,因而子模块不重叠. ring 简写为 ri.
p=xpos,ypos,exact,deltax,deltay	用于模块向量.每个子模块被放置在(xpos+deltax,ypos+deltay).如果 deltax 和 deltay 参数(''p=100,100,exact,$x,$y'')在向量中的每个模块里都有不同的值,那么是有用的. 默认:无 exact 简写为 e 或 x.
b=width,height,rect	给定 height 和 width 的矩形. 默认： width=40， height=24
b=width,height,oval	给定 height 和 width 的椭圆.

	默认: width=40, height=24
o=fillcolor,outlinecolor,borderwidth	为矩形或椭圆指定选项.对于颜色符号,见[9.2]. 默认: fillcolor=#8080ff(淡蓝色), outlinecolor=black, borderwidth=2
i=iconname,color,percentage	使用指定的图标.它可以着色,并且使用百分比来指定着色的大小. 默认: iconname:无缺省值—如果没有给出图标名,使用box; color:没有颜色; percentage: 30%.
is=size	指定图标的大小 size 可以是 l, vl, s 和 vs(large, very large, small, very small)中的一个.如果给出了这个选项,size 不能使用"i=<iconname>_<size>"符号包括在图标名("i=" tag)中
i2=iconname,color,percentage	在原先图标的右上角显示一个小的" modifier"图标.建议图标是 status/busy, status/down, status/up, status/asleep 等. 参数与那些"i="是类似的.
r=radius,fillcolor,color,width	围绕子模块给定的半径画一个圆(或一个填充圆).可以用于可视化无线节点的传输范围. 默认: adius=100, fillcolor=none, color=black, width=1(未填充的黑圆)
q=queue-object-name	显示子模块图标的下一个的队列长度.要求 cQueue 对象的名称 (通过 setName()方法设置,见[6.1.4]). Tkenv 将通过深度优先搜索来寻找对象,将在子模块里发现队列对象.
t=text,pos,color	在图标之上或之下显示一个短文本.文本是传递状态信息("up", "down", "5Kb in buffer")或统计信息("4 pks received").pos 可以是"l", "r"或"t"表示 left, right 和 top. 默认: pos="t", color=blue
tt=tooltip-text	当用户移动鼠标至图标时,在工具提示中显示给定的文本.这个补充 t=标签,使你可以显示更多的信息,否则这些不会出现在屏幕上.

Examples:

```
"p=100,60;i=workstation"

"p=100,60;b=30,30,rect;o=4"
```

9.1.3 显示字符串的背景

复合模块显示字符串指定背景.他们可以包括以下的标签:

标签	意义
p=xpos,ypos	在(xpos,ypos)像素位置放置封装的模块,窗口的

	左上角为(0,0).
b=width,height,rect	显示封装模块为给定 height 和 width 的矩形. 默认: 自动选择 width,height
b=width,height,oval	显示封装模块为给定 height 和 width 的椭圆. 默认: 自动选择 width,height
o=fillcolor,outlinecolor,borderwidth	指定矩形或椭圆的选项.对于颜色符号,见[9.2]. 默认:fillcolor=#8080ff(浅蓝), outlinecolor=black, borderwidth=2
tt=tooltip-text	当用户移动鼠标至左上角的模块名时,在工具提示中显示给定的文本.

9.1.4 链接显示字符串

链接显示字符串可用的标签:

标签	意义
m=auto m=north m=west m=east m=south	绘制模式.精确指定链接箭头的位置.参数可以简写为 a,e,w,n,s.
m=manual,srcpx,srcpy,destpx,destpy	手动模式有四个参数,精确指定了箭头锚定的终点:srcpx, srcpy, destpx, destpy.每个值是源/目的模块封装的矩形的 width/height 的百分比,左上角是原点.因而 m=m,50,50,50,50 会链接两个模块矩形的中心.
o=color,width	指定箭头的细节.对于颜色符号,见[9.2]. 默认: color=black, width=2
t=text,color	在链接箭头附近显示一个短文本.文本可能传递状态信息或链接属性("down", "100Mb")或统计信息. 默认: color=#005030
tt=tooltip-text	当用户移动鼠标至链接箭头时,在工具提示中显示给定的文本. 这个补充 t=标签,使你可以显示更多的信息,否则这些不会出现在屏幕上.

例如:

```
  "m=a;o=blue,3"
```

9.1.5 消息显示字符串

消息对象默认情况下不存储显示的字符串,但是可以重定义 cMessage 的 displayString()方法,使其可以返回显示的字符串.

```
const char *CustomPacket::displayString() const
{
    return "i=msg/packet_vs";
```

```
}
```

这个显示字符串影响了如何在动画期间显示消息.默认情况下,他们显示为一个小的填充的圆,8 个基本颜色(颜色通过消息种类模除 8 来决定),并与消息类和/或名称在其下显示,后者在 Tkenv 选项对话框中可以配置,消息种类依赖于颜色,也可以在那关闭掉.

以下的标签是可以在消息显示字符串中使用的:

标签	意义
b=width,height,oval	给定 heigh 和 width 的椭圆 默认: width=10, height=10
b=width,height,rect	给定 heigh 和 width 的矩形 默认: width=10, height=10
o=fillcolor,outlinecolor,borderwidth	指定矩形或椭圆的选项.对于颜色符号,见[9.2]. 默认: fillcolor=red, outlinecolor=black, borderwidth=1
i=iconname,color,percentage	使用指定的图标.它可以着色,并且使用百分比来指定着色的大小.如果颜色名为"kind",那么使用消息种类依赖的颜色(类似默认行为). 默认: iconname:无缺省值—如果没有给出图标名,使用一小的红实心圆; color:没有颜色; percentage: 30%.
tt=tooltip-text	当用户移动鼠标至消息图标时,在工具提示中显示给定的文本.

例如:

```
   "i=penguin"
   "b=15,15,rect;o=white,kind,5"
```

9.2 颜色

9.2.1 颜色名

可以接受任何有效的 Tk 颜色:英文名(blue, lightgray, wheat)或#rgb, #rrggbb 格式(其中 r,g,b 是十六进制数).

也可以 HSB((hue-saturation-brightness 色调-饱合-亮度)中指定颜色为@hhssbb(h,s,b 中十六进制数).HSB 可以更简单地测量颜色,比如从白至光亮红.

通过指定颜色为连字号("-")来产生一个透明的背景.

9.2.2 彩色图标

"i="显示字符串标签允许彩色化图标.它接受一个目的颜色的一个彩色度的百分比为参数.如果忽略目的颜色,百分比无效.也影响图标的亮度—为了保持原先的亮度,指定颜色约 50%亮度(比如, #808080 中灰色, #008000 中绿色)

例如:

- "i=device/server,gold" 创建一个金色的服务器图标

- "i=misc/globe,#808080,100"使图标为灰度色标
- "i=block/queue,white,100" 产生一个”燃烧”的黑白图标

着色工作可以作用于子模块和消息图标.

9.3 图标

9.3.1 位图路径

在目前的 OMNeT++版本中,模块图标是 GIF 文件. OMNeT++的图标在 GNED 编辑器和 Tkenv 都需要确切的目录位置来加载图标.

图标在 bitmap path 的所有目录中加载,一个分号分隔的列表.默认的位图路径被编译至 GNED 和 Tkenv 中,值为"omnetpp-dir/bitmaps;./bitmaps"—如果不移动目录的话,这可以很好地工作,也可以从当前目录的 bitmaps/子目录中加载更多的图标.正如人们通常从模型的目录运行仿真模型,这实际上表示模型目录的 bitmaps/子目录中的定制图标会被自动加载.

编译的位图路径可以覆盖 OMNETPP_BITMAP_PATH 环境变量.设置环境变量的方法是系统指定的:在 Unix 中,如果使用 bash shell,添加一行:

export OMNETPP_BITMAP_PATH="/home/you/bitmaps;./bitmaps"

至 ~/.bashrc 或~/.bash_profile 就可以完成; 在 Windows 中, 环境变量可以通过我的电脑→属性对话框来设置.

也可以从 omnetpp.ini 添加位图路径,进行 bitmap-path 设置:

```
[Tkenv]
bitmap-path = "/home/you/model-framework/bitmaps;/home/you/extra-bitmaps"
```

值需要引号,否则第一个分号分隔符将被视为注释符,将导致忽略其余的目录.

9.3.2 分类图标

由于 OMNeT++ 3.0 中,图标被组织为多个类别,表示为文件夹.这些类别包括:

- block/ - 子部件的图标(队列,协议等).
- device/ - 网络驱动:服务器,主机,路由器等.
- abstract/ - 各种驱动的符号图标.
- misc/ - 节点,子网,污点,建筑,城镇,城市等.
- msg/ - 可以用于消息的图标.

旧的(3.0 之前) 图标在 old/ 文件夹中.

Tkenv 和 GNED 现在从位图路径的所有目录的子目录中加载图标,这些图标也可以通过命令子目录,从显示字符串中引用: "subdir/icon", "subdir/subdir2/icon"等.

对于兼容性,如果显示字符串包括一个没有类别(如子目录)的图标名, OMNeT++将视为在"old/icon"中.

9.3.3 图标大小

图标可以是各种大小: normal, large, small, very small.大小可以编码进图标名的后缀: _l, _s, _vs.在显示字符串,可以使用后缀("i=device/router_l"),也可以使用"is" (图标大小)字符串标签("i=device/router;is=l").

9.4 布局

OMNeT++实现了自动布局的特点,使用 SpringEmbedder 算法变化.模块没有通过"p="标签明确分配位置的将通过算法自动地放置.

SpringEmbedder 是基于物理模型的图形布局算法.图形节点(模块)彼此相斥,就像相同符号的电极,链接是 spring 分类,尝试缩短或弹出被附加的节点. 为了防止节点振动,也可摩擦构建.布局算法仿真这个物理系统直到它达到了平衡(或超时).微调以上的物理规则,可以得到更好的结果.

算法不移动任何固定坐标的模块.预定义行,矩阵,圆环或其它排列(通过第三个和进一步的"p="标签参数定义)将被预留—可以想像他们,仿佛这些模块被附加至一个木制的框架,因而可以像一个单元一样移动他们.

注意事项:

- 如果全图在布局后太大,则进行缩减以便适合在屏幕上显示,除非它包括任何固定位置的模块.(原因很明显:如果有一个手动指定位置的模块,我们不想移动它).为了防止重新调节,可以在背景中指定一个充分大的范围盒显示字符串,比如"b=2000,3000".
- 大小通过当前的布局器被忽略,因此稍长的模块(比如以太网段)可能产生奇异的结果.
- 算法容易产生奇怪的结果,特别是当子模块的数量小的时候,可当使用预定义(矩阵,行,圆环等)的布局时. "Re-layout"工具条按钮也非常有用.较大的网络通常产生令人满意的结果.

布局器算法的参数(排斥/吸引力, 循环的数量,随机数种子) 可以通过"l="后台显示字符串标签.它当前的参数是(缺省值):

"l=<repulsion>=10,<attraction>=0.3, <edgelen>=40,<maxiter>=500,<rng-seed>". "l="标签有些根据经验,它的参数可能在以后发布的版本中改变.

9.5 GNED - 图形化 NED 编辑器

GNED 编辑器允许图形化设计复合模块. GNED 直接使用 NED 工作—没有任何的内部文件格式.可以加载任何现有的 NED 文件,在其中图形化编辑复合模块,然后保存文件.NED 文件中其它的组件(简单模块,信道,网络等.)将免于操作.GNED 将所有与图形相关的数据放入显示字符串.

GNED 工作通过解析 NED 文件至一个内部数据结构,当保存文件时重新生成 NED 文本.这样的一个结果是压痕将被”推荐”.在原始 NED 中的注释被保留—解析器将与他们所属于 NED 元素相联,因此即使你通过移除/添加子模块,门,参数,链接等编辑图形化的表述,注释不会被搞得一塌糊涂.

GNED 是一个完全双向的可视化工具.而编辑图形,可以随时切换至 NED 源文件视图,在其中编辑,然后再切换至图形.在 NED 源文件中的改变将立即解析至图形;事实上,图形将从 NED 源文件中完全正确生构,在其中显示字符串.

9.5.1 绑定键盘和鼠标

在图形视图中,有两个编辑模式: draw 和 select/move.鼠标如下绑定:

鼠标	作用
在 draw 模式中:	

在空区域拖出一个矩形：	创建一个新的子模块
将一个子模块拖至另一个中：	创建一个新的链接
在空区域点击：	切换至 select/move 模式
在 select/move 模式中：	
点击子模块/链接：	选择它
Ctrl+点击子模块/链接	添加至选择
在空区域点击：	清除选择
拖一个选择的对象：	移动选择的对象
拖一个子模块或链接	移动它
拖任一链接的终点：	移动终点
拖(子)模块的角落：	调整模块大小
在空区域开始拖：	选择封装的子模块/链接
Del 关键字	删除选择的对象
两个编辑模式：	
在模块/子模块/链接上右击：	弹出菜单
在子模块上双击	进入子模块
点击 name 标签	编辑 name
拖拽模块类型从树视图至画布	创建该类型的子模块

9.6 增强动画

9.6.1 在运行时改变显示字符串

通常在运行时处理显示字符串有用.当仿真移动网络时,改变颜色,图标,或可能传递状态改变的文本,以及改变一个模块的位置是有益的.

显示字符串存储在 cDisplayString 对象里,在模块和门的内部. cDisplayString 也可以操作字符串.

为了得到指向 cDisplayString 对象的指针,可以调用模块的 displayString()方法:

```
cDisplayString *dispStr = displayString();

cDisplayString *bgDispStr = parentModule()->backgroundDisplayString();

cDisplayString *gateDispStr = gate("out")->displayString();
```

据所关注的 cDisplayString 来说,一个显示字符串(比如"p=100,125;i=cloud")是一个字符串,由多个分号分隔的标签组成,每个标签在等号之后有一个 name,逗号分隔 0 或多个参数.

类有利于分派任务,比如找出显示字符串有哪些标签,添加新标签,添加参数至现有的标签,清除标签或更换参数.内部存储方法允许快速操作;它一般比直接的字符串操作快.类不尝试以任何方式来解释字符串,也不知道不同标签的意思;它仅解析字符串为由分号相隔的数据元素,等号和逗号.

例如:

```
dispStr->parse("a=1,2;p=alpha,,3");

dispStr->insertTag("x");

dispStr->setTagArg("x",0,"joe");

dispStr->setTagArg("x",2,"jim");

dispStr->setTagArg("p",0,"beta");

ev << dispStr->getString();  // result: "x=joe,,jim;a=1,2;p=beta,,3"
```

9.6.2 Bubbles

模块通过显示的一个弹出的短消息("Going down", "Coming up"等),让用户了解重要的事件(比如,节点下降或提升).这是由 cModule 的 bubble()方法完成的.该方法使字符串被显示为一个 const char *指针.

例如:

```
bubble("Going down!");
```

如果模块包括大量的代码,修改显示字符串或显示 bubble,那么建议在 ev.isGUI()之上获得这些条件调用. 当仿真在 Cmdenv 下运行时,ev.isGUI()调用返回 false,因此可以使代码跳过潜在的代价高的显示字符串操作.

```
if (ev.isGUI())

    bubble("Going down!");
```

10 分析仿真结果

10.1 输出向量

输出向量是时间序列数据:带时间戳的值.可以使用输出向量来记录端到端的延迟或包的往返时间,队列长度,排队时间,链路利用率,丢弃的包的数量,等.一任何有用的数据,在仿真运行期间得到模型中发生的完全图.

输出向量通过 cOutVector 对象(网民[6.9.1])从简单模块记录.由于输出向量通常记录大量的数据,在 omnetpp.ini 中可以禁止向量或指定一个记录的仿真时间间隔(见[8.5]).

所有 cOutVector 对象写至相同的公共的文件.以下部分描述了文件的格式,及如何处理它.

10.1.1 使用 Plove 绘制输出向量

Plove 特点

通常,会得到输出向量作为仿真的结果.数据从简单模块写至 cOutVector 对象被写入输出向量文件.可以使用 Plove 来查看输出向量文件,并从中绘制向量.

Plove 是用于绘制 OMNet++输出微量的便利的工具.可以设置每个向量的线类型(线,点等),以及最频繁绘制的选项,如轴范围,缩放比例,标题,标签.可以点击保存图形至文件(就像压缩的 Postscript 或光栅格式,比如 GIF).在 Windows 中,也可复制向量格式的图形至剪贴板中(Windows 的元文件),并粘贴至其它的应用中.

[注意: OMNeT++ 3.0 之前, Plove 是 gnuplot 的前期.这个 Plove 的旧版本不再支持,但是在 OMNeT++的源文件发行中仍然有用.]

在可能的绘制之前过滤结果.过滤可以完成平均计算,截断极端值,平滑,他们通过计算柱状图等完成密度估计.一些过滤器是内置的,可以通过参数以及集合现有的来创建一个新的过滤器.一个向量可以应用多个过滤器.

启动时, Plove 自动读 home 目录下的.ploverc 文件.文件包含一般的应用设置,其中包括你创建的定制过滤器.

用法

首先需要加载一个输出向量文件(.vec)至左边的窗格中.可以从左窗格中拷贝向量至右窗格,通过点击中间的向右的箭头. PLOT 按钮将在右窗格中的初始化绘制选中的向量.在 Windows 中选择工作:按 shift 加左击拖动选定一个矩形, ctrl 加左击选定/取消选定单个的项目.为了调整绘制的形式,改变向量的标题,或添加一个过滤器,点击选项...按钮.这也可以用于多个选定的向量. Plove 接受类 nc/mc 的按键: F3, F4, F5, F6, F8, grey '+' 和 grey '*'.

左窗格的工作为一般所工作的向量存储.可以加载多个向量文件,删除不想处理的向量,重命名等.这些改变将不影响磁盘上的向量文件.(Plove 从不修改输出向量文件本身.)在右窗格中,如果想过滤向量,并且又保持原始的,可能复制向量.如果设置向量的合适的选项,但暂时不想留在右窗格附近,可以将其放回左窗格中存储.

10.1.2 输出向量文件的格式

一个输出向量文件可以包括仿真期间产生的多个时间序列.文件是文本形式的,类似如下:

```
mysim.vec:

vector 1   "subnet[4].term[12]"  "response time"  1

1  12.895  2355.66666666

1  14.126  4577.66664666

vector 2   "subnet[4].srvr"  "queue length"  1

2  16.960  2.00000000000.63663666

1  23.086  2355.66666666

2  24.026  8.00000000000.44766536
```

有两个行类型:向量声明行(以 vector 开头)和数据行.一个向量声明行引入的一个新输出向量,其列是向量 Id,创建的模块, cOutVector 对象名,和多样性(通常是 1).实际的数据记录在这个向量中的是数据行,以向量 Id 开头.在数据行中进一步的列是仿真时间和记录值.

10.1.3 无 Plove 的工作

在有大量重复实验的情况下,可能想自动处理输出向量文件. OMNeT++可以让你使用任何工具,来达到这个目的,由于输出向量文件是文本文件,他们的格式很简单,可以使用公共的工具处理,如 perl, awk, octave 等.

从文件中提取向量

可以使用 Unix 的 grep 工具从文件中提取一个特定的向量.首先,你必须找到向量的 Id.可以使用文本编辑器找到相应的向量行,或可以使用 grep 来完成:

```
% grep "queue length" vector.vec
```

或者,可以通过输入得到文件中的所有向量列表:

```
% grep ^vector vector.vec
```

这将输出相应的向量行:

```
vector 6  "subnet[4].srvr"  "queue length"  1
```

选择向量 Id,在这里是 6,grep 文件的向量的数据行:

```
grep ^6 vector.vec > vector6.vec
```

这里 vector6.vec 包括相应的向量.唯一潜在的问题是,向量 Id 在每行的开始之处,这会很难使用后处理和/或可视化来对一些程序进行分类.这个问题通过利用 OMNeT++ 的 splitvec 消除(写进 awk),在下一节讨论.

使用 splitvec

splitvec 脚本(OMNeT++的一部分)自动处理之前的描述:将向量文件打断至多个文件,每个文件包括一个向量.命令为:

```
% splitvec mysim.vec
```

将创建文件 mysim1.vec, mysim2.vec 等,其内容类似如下:

```
mysim1.vec:

# vector 1  "subnet[4].term[12]"  "response time"  1

12.895  2355.66666666

14.126  4577.66664666

23.086  2355.66666666


mysim2.vec:

# vector 2  "subnet[4].srvr"  "queue length"  1

16.960  2.00000000000.63663666

24.026  8.00000000000.44766536
```

正如所看到的,向量 Id 列从文件中分解.最终文件可以直接加载入电子数据表或其它程序[10.3].

10.2 标量统计数据

输出向量捕捉仿真运行的短暂行为.然而,为了在各种参数设置下比较模型行为,输出标量更加有用.

10.2.1 输出标量文件的格式

标量结果通常在模块的 finish()方法中调用 recordScalar()记录, 代码类似如下:

```
void EtherMAC::finish()

{

    double t = simTime();

    if (t==0) return;
```

```
    recordScalar("simulated time", t);
    recordScalar("rx channel idle (%)", 100*totalChannelIdleTime/t);
    recordScalar("rx channel utilization (%)", 100*totalSuccessfulRxTxTime/t);
    recordScalar("rx channel collision (%)", 100*totalCollisionTime);

    recordScalar("frames sent",     numFramesSent);
    recordScalar("frames rcvd",     numFramesReceivedOK);
    recordScalar("bytes sent",      numBytesSent);
    recordScalar("bytes rcvd",      numBytesReceivedOK);
    recordScalar("collisions",      numCollisions);

    recordScalar("frames/sec sent", numFramesSent/t);
    recordScalar("frames/sec rcvd", numFramesReceivedOK/t);
    recordScalar("bits/sec sent",    8*numBytesSent/t);
    recordScalar("bits/sec rcvd",    8*numBytesReceivedOK/t);
}
```

相应的输出标量文件(默认 omnetpp.sca) 将类似:

```
run 1 "lan"
scalar "lan.hostA.mac" "simulated time"             120.249243
scalar "lan.hostA.mac" "rx channel idle (%)"        97.5916992
scalar "lan.hostA.mac" "rx channel utilization (%)" 2.40820676
scalar "lan.hostA.mac" "rx channel collision (%)"   0.011312
scalar "lan.hostA.mac" "frames sent"                99
scalar "lan.hostA.mac" "frames rcvd"                3088
scalar "lan.hostA.mac" "bytes sent"                 64869
scalar "lan.hostA.mac" "bytes rcvd"                 3529448
scalar "lan.hostA.mac" "frames/sec sent"            0.823290006
scalar "lan.hostA.mac" "frames/sec rcvd"            25.6799953
scalar "lan.hostA.mac" "bits/sec sent"              4315.63632
scalar "lan.hostA.mac" "bits/sec rcvd"              234808.83
scalar "lan.hostB.mac" "simulated time"             120.249243
scalar "lan.hostB.mac" "rx channel idle (%)"        97.5916992
```

```
scalar "lan.hostB.mac" "rx channel utilization (%)" 2.40820676
scalar "lan.hostB.mac" "rx channel collision (%)"   0.011312
[...]
scalar "lan.hostC.mac" "simulated time"             120.249243
scalar "lan.hostC.mac" "rx channel idle (%)"        97.5916992
scalar "lan.hostC.mac" "rx channel utilization (%)" 2.40820676
scalar "lan.hostC.mac" "rx channel collision (%)"   0.011312
[...]

run 2 "lan"
scalar "lan.hostA.mac" "simulated time"             235.678665
[...]
```

每个 recordScalar()调用都会在文件中产生一行.(如果通过 recordScalar()方法记录统计对象(cStatictic 子类,如 cStdDev),会产生多行:均值,标准差等).另外,多个仿真运行可以记录它们的结果至单个文件—容易对其进行比较,创建 x-y 图(提供负载与类型吞吐量的图),等.

10.2.2 标量工具

标量工具可以用于可视化 omnetpp.sca 文件的内容.它可以画柱形统计图, x-y 图(如吞吐量与负载),或通过剪贴板输出数据在电子统计表或其它程序中进行更详细的分析.

可以从标量程序的菜单或指定一个命令行参数至 Scalars 打开一个标量文件.

该程序在表中显示数据,表中的列有文件名,run 数量,记录的模块名,和值.通常行太多而不能得到一个概况,因而可以通常选择(或编辑)顶部的三个组合框进行过滤.(过滤器也接受*,**通配符.)

实际上,可以进一步加载标量文件至窗口中,因而可以一起进行分析.

通过 Edit|Copy 或相应的工具条按钮,可以复制选中的行至剪贴板,然后粘贴它们,比如至 OpenOffice Calc, MS Excel 或 Gnumeric.

柱形统计图工具条按钮在一个新的窗口创建一个较好的柱形统计图.可以在其上右击定制图表,或从背景菜单选择.可以被导出为 EPS, GIF,或通过 Windows 剪贴板的元文件(当然后者在 Unix 上是不可用的).

10.3 分析和可视化工具

输出向量文件(或由 splitvec 产生的文件)和输出标量文件,除了使用 Plove 和 Scalars 之外还可以通过大量的应用程序分析和/或绘制.这些程序可以产生各种形式的输出(在屏幕上,如 PostScript,各种图象格式等).

一个直接的解决方法是导入或粘贴它们至电子表格程序,如 OpenOffice Calc, Microsoft Excel 或 GNOME Gnumeric.这些程序具有很好的绘图与统计的特点,但是行的数量通常限制在大约 32,000..64,000. 电子表格的一个有用的功能是,提供对标量文件的分析,称为 Excel 中的 PivotTable,和 OpenOffice DataPilot.最简单的方法是通过复制/粘贴标量,将标量文件导入至它们中.

或者可以使用数字包,如 Octave, Matlab 或统计包 R.除了支持统计计算外,它们也可以创建各种图.

有一些开源的程序可以直接绘图. Gnuplot 仍然是最常使用的.另外,可能有更强大的,包括 Grace, ROOT 和 PlotMTV.

10.3.1 Grace

Grace(也称为 xmgrace, ACE/gr 的后继或 Xmgr)是一个基于 GPL 平台的, 通过点击图形用户界面的,强有力的 WYSIWIG 数据可视化开发程序.这是为 Unix 开发的,但也有 Windows 版本.通过在对话框中选中,加载相应的文件.图标栏和菜单命令可以用于定制图形.

至 2003 年 6 月, Grace 1.5.12 可以导出图形至(E)PS, PDF, MIF, SVG, PNM, JPEG 和 PNG 格式.它有许多有用的特点,如内置式统计和分析功能(如,相关性,柱状图), fitting, splines 等,也有其本身内置的编程语言.

10.3.2 ROOT

ROOT 是一个强有力的面向对象数据分析框架,强力支持普通的绘图与图形. ROOT 在 CERN 开发,并发布类 BSD 的许可.

ROOT 是基于 CINT,一个”C/C++解释器”旨在处理 C/C++脚本.开始使用 ROOT 可能要比 Gnuplot 或 Grace 较难,但如果要认真地分析仿真结果,会发现 ROOT 提供了其它两个程序很难达到的,更强大更灵活的特性.

Curt Brune 在 Stanford 的页面(http://www.slac.stanford.edu/~curt/omnet++/)显示了一个在 OMNeT++中使用 ROOT 的例子.

10.3.3 Gnuplot

Gnuplot 有一个交互的命令接口.为了在 mysim1.vec 和 mysim4.vec()中绘制数据为相同的图形,可以输入:

```
plot "mysim1.vec" with lines, "mysim4.vec" with lines
```

为了调整 y 的范围,可以输入:

```
set yrange [0:1.2]

replot
```

许多的命令可以用于调整范围,绘制风格,标签,缩放比例等. Gnuplot 可以绘制 3D 图. Gnuplot 可用于 Windows 和其它平台.在 Windows 中,可以从 Gnuplot 窗口的系统菜单,拷贝结果图至剪贴板,然后插入至工作的应用程序中.

11 NED 文档和消息

11.1 概述

OMNeT++提供了一个工具,可以从 NED 文件和消息定义中产生 HTML 文档.像 Javadoc 和 Doxygen, opp_neddoc 利用源代码注释. opp_neddoc 生成的文档列出了简单和复合模块,以及显示了他们的细节包括描述,门,参数,非指派的子模块参数和高亮的语法源代码.文档也包括可以点击的网络图(通过 GNED 图形编辑器导出)和模块使用图以及消息的继承图.

opp_neddoc 与 Doxygen 取得很好的效果,即在 Doxygen 文档中,可以超链接简单模块和消息至它们的 C++实现类.如果与 Doxygen 产生 C++文档的特性打开 (如 inline-sources 和 referenced-by-relation, 与 extract-all, extract-private 和 extract-static 的组合),结果很容易浏览和非常有益于源代码的显示.当然,在代码中必须写文档注释.

11.2 授权文档

11.2.1 文档注释

嵌入一般注释的文档.所有在//注释”右方”的(从文档工具的观点)将包括进产生的文档.

[相反, avadoc 和 Doxygen 使用源代码中特殊的注释(他们是以/**, ///, //<或类似的标记开始)来区别文档中一般的注释.在 OMNet++中,不需要这样:NED 和消息语法非常简洁,因而实际上所有的注释都要写入文档中,从而达到文档的目的.不过,有一种方法写注释,但并不生成至文档中—以//#开头.]

例如:

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//
simple Gen
    parameters:
        destAddress: string,  // destination MAC address
        protocolId: numeric,  // value for SSAP/DSAP in Ethernet frame
        waitMean: numeric;    // mean for exponential interarrival times
    gates:
        out: out;             // to Ethernet LLC
endsimple
```

也可以在 parameters 和 gates 之上放置注释.如果需要较长的解释的话,这较有用.例如:

```
//
// Deletes packets and optionally keeps statistics.
//
simple Sink
    parameters:
        // You can turn statistics generation on and off. This is
        // a very long comment because it has to be described what
        // statistics are collected (or not).
        statistics: bool;
    gates:
        in: in;
endsimple
```

如果想要一个注释行不出现在文档中,以//#开头.这些行将被文档生成器忽略,可以用于注释非 NED 代码或”私有”的注释,像 FIXME 或 TBD.

```
//
// An ad-hoc traffic generator to test the Ethernet models.
//# FIXME above description needs to be refined
//
simple Gen
    parameters:
        destAddress: string,  // destination MAC address
        protocolId: numeric,  // value for SSAP/DSAP in Ethernet frame
        //# burstiness: numeric;  -- not yet supported
        waitMean: numeric;    // mean for exponential interarrival times
    gates:
        out: out;             // to Ethernet LLC
endsimple
```

11.2.2 文本布局与格式化

如果想写较长的描述,需要文本格式化能力.文档的格式化工作像在 Javadoc 或 Doxygen 中一—可以将文本分解为段落,创建没有特殊命令的 bulleted/numbered 列表,使用 HTML 有更多奇异的格式.

段落由空行分隔,像在 LaTeX 和 Doxygen 中一样.以‘-’开始的行将被分解为 bulleted 列表,以’-#’开头的,将被分解至 numbered 列表.

例如:

```
//
// Ethernet MAC layer. MAC performs transmission and reception of frames.
//
// Processing of frames received from higher layers:
// - sends out frame to the network
// - no encapsulation of frames -- this is done by higher layers.
// - can send PAUSE message if requested by higher layers (PAUSE protocol,
//   used in switches). PAUSE is not implemented yet.
//
// Supported frame types:
// -# IEEE 802.3
// -# Ethernet-II
//
```

11.2.3 特殊标签

OMNeT++_neddoc 理解以下的标签,并相应地表示它们: @author, @date, @todo, @bug, @see, @since, @warning, @version.用例:

```
//
// @author Jack Foo
// @date 2005-02-11
//
```

11.2.4 使用 HTML 添加文本格式

普通的 HTML 标签被理解为格式化命令.这些标签中最有用的是: .. (斜体), .. (粗体), <tt>..</tt> (typewriter 字体), _{..} (下标), ^{..} (上标),
 (换行), <h3> (标题), <pre>..</pre> (预先格式文本)和 .. (链接),以及其它一些用创建表格的标签(见下方).例如, Hello将被处理为”Hello”(使用斜体).

opp_neddoc 解释的 HTML 标签完整列表为: <a>, , <body>,
, , <caption>, <code>, <dd>, <dfn>, <dl>, <dt>, , <form>, <font>, <hr>, <h1>, <h2>, <h3>, , <input>, <img>, <li>, <meta>, <multicol>, <ol>, <p>, <small>, , , <sub>, <sup>, <table>, <td>, <th>, <tr>, <tt>, <kbd>, <ul>, <var>.

任何不在上面的标签则不会被解释为格式化命令,但会被逐字打印—例如<what>bar</what>将会被逐字处理为”<what>bar</what>”(不像 HTML,我们不认识的标签被简单地忽略,如果 HTML 中会显示为”bar”).

如果插入链接至外部页面(web 站点),添加 target="_blank"属性将很有用,保证页面出现在新的浏览窗口而不是在当前的框架中,很难使用.(或者,可以使用 target="_top"属性,可以在当前的浏览器中替换所有的框架).

例如:

```
//
// For more info on Ethernet and other LAN standards, see the
// <a href="http://www.ieee802.org/" target="_blank">IEEE 802
// Committee's site</a>.
//
```

也可以使用 <a href=..>标签在页面内创建链接:

```
//
// See the <a href="#resources">resources</a> in this page.
// ...
// <a name="resources"><b>Resources</b></a>
// ...
//
```

可以使用<pre>..</pre>标签为插入源代码的例子至文档中.行分解和缩排将被保留,但是 HTML 标签继续解释处理(或者可以使用<nohtml>关闭,见后面).

例如:

```
// <pre>
// // my preferred way of indentation in C/C++ is this:
// <b>for</b> (<b>int</b> i=0; i<10; i++)
// {
//     printf(<i>"%d\n"</i>, i);
// }
// </pre>
```

将被处理为

```
// my preferred way of indentation in C/C++ is this:
for (int i=0; i<10; i++)
{
    printf("%d\n", i);
}
```

HTML 也可以创建表格.例子如下:

```
//
// <table border="1">
//   <tr>  <th>#</th> <th>number</th> </tr>
//   <tr>  <td>1</td> <td>one</td>    </tr>
//   <tr>  <td>2</td> <td>two</td>    </tr>
//   <tr>  <td>3</td> <td>three</td>  </tr>
// </table>
//
```

将被大约处理为:

#	number
1	one
2	two
3	three

11.2.5 回避 HTML 标签

有时可能不需要解释处理 HTML 标签(<i>, <b>等)为格式化指令,要在文档中显示<i>, <b>文本.可以通过在标签外加上<nohtml>...</nohtml>为完成.例如,

```
// Use the <nohtml><i></nohtml> tag (like <tt><nohtml><i>this</i></nohtml><tt>)
```

```
// to write in <i>italic</i>.
```

将被处理为

```
”Use the <i> tag (like <i>this</i>) to write in italic.”
```

<nohtml>...</nohtml>也防止 opp_neddoc 超链接与现有模块或消息名偶然相同的关键词. 单词的前缀为一反斜号,也是相同的效果.即以下的两个都可以完成:

```
// In <nohtml>IP</nohtml> networks, routing is...

// In \IP networks, routing is...
```

如果在 NED 文件中有 IP 模块,这将防止超链接单词 IP.

11.2.6 哪里输出注释

必须在 nedtool 可以找到的地方放置注释.即 a) 在文档项之上,或 b)在文档项之后,相同的行.

如果放置在之上的话.确定在注释和文本项之间没有空行.空行从文档项中分离注释.

例如:

```
// This is wrong! Because of the blank line, this comment is not

// associated with the following simple module!


simple Gen

    parameters:

    ...

endsimple
```

不要试着将参数组合注释,这将很难使用.

11.2.7 定制标题页面

在浏览器中打开文档之后,主框架中显示标题页面.默认情况下,它包括普通标题的样板文件” OMNeT++ Model Documentation”.可能想要定制它,至少改变标题至文档仿真模型的名称.

可以提供自己版本的标题页面,通过添加一个@titlepage 指令至文件级的注释(出现在 NED 文件顶部的注释,但通过至少一个空行定义,与第一个 import,信道,模块等是分开的).在理论上,可以旋转你的标题页面定义至任何 NED 或 MSG 文件,但是为其创建一个单独的 index.ned 文件可能不是一个好的方法.

在@titlepage 行之后直至下一个@page 行(见后面)或注释的结尾将用于标题页面. 由于注释工具不增加标题(它可以使你完全控制页面内容),想以一个标题开始.可以使用 <h1>..</h1>HTML 标签来定义标题.

例如:

```
//

// @titlepage

// <h1>Ethernet Model Documentation</h1>
```

```
//
// This documents the Ethernet model created by David Wu and refined by Andras
// Varga at CTIE, Monash University, Melbourne, Australia.
//
```

11.2.8 添加额外的页面

可以通过类似于定制标题页面的方法来添加一个新的页面至文档.使用@page 指令,可以在任何文件级注释中出现(见上面).

@page 指令的语法如下:

```
// @page filename.html, Title of the Page
```

请选择一个不与文档工具产生的文件冲突的文件名(比如 index.html).提供的页面标题将出现在页面以及页面索引的顶部.

@page 行之后直至下一个@page 行或注释的结尾将用于页面的主体.由于文档工具自动添加,不需要添加标题.

例如:

```
//
// @page structure.html, Directory Structure
//
// The model core model files and the examples have been placed
// into different directories. The <tt>examples/</tt> directory...
//
//
// @page examples.html, Examples
// ...
//
```

可以使用标准 HTML 的...标签来创建链接至产生的页面.所有的 HTML 文件都放置在单个目录中,因此不需要担心详细的目录.

例如:

```
//
// @titlepage
// ...
// The structure of the model is described <a href="structure.html">here</a>.
//
```

11.2.9 合并外部创建的页面

可能想在文档工具的外部创建页面(使用 HTML 编辑器),并将它们包括至文档中.这是可行的,你所必须做的事是在 NED 文件的任何地方,使用@externalpage 指令,声明页面,它们将被添加至页面索引.页面然后从其它的页面使用 HTML...标签链接.

@externalpage 指令在语法上类似于@page:

```
// @externalpage filename.html, Title of the Page
```

文档工具不检查页面是否存在.你负责手动拷贝它们至产生文档的目录,然后确定超链接可以工作.

11.3 调用 opp_neddoc

opp_neddoc 工具接受以下的命令行选项:

```
opp_neddoc - NED and MSG documentation tool, part of OMNeT++

(c) 2003-2004 Andras Varga


Generates HTML model documentation from .ned and .msg files.


Usage: opp_neddoc options files-or-directories ...

 -a, --all    process all *.ned and *.msg files recursively

              ('opp_neddoc -a' is equivalent to 'opp_neddoc .')

 -o <dir>     output directory, defaults to ./html

 -t <filename>, --doxytagfile <filename>

              turn on generating hyperlinks to Doxygen documentation;

              <filename> specifies name of XML tag file g enerated by Doxygen

 -d <dir>, --doxyhtmldir <dir>

              directory of Doxygen-generated HTML files, relative to the

              opp_neddoc output directory (-o option). -t option must also be

              present to turn on linking to Doxygen. D efault: ../api-doc/html

 -n, --no-figures

              do not generate diagrams

 -p, --no-unassigned-pars

              do not document unassigned parameters

 -x, --no-diagrams

              do not generate usage and inheritance diagrams

 -z, --no-source

              do not generate source code listing

 -s, --silent suppress informational messages
```

```
-g, --debug  print invocations of external programs and other info

-h, --help   displays this help text
```

文件指定为被解析的参数并记录在案.由于目录为参数,在其下的所有.ned 和 .msg 文件(在目录的子树中)都被记录在案.接受通配符,但不递归,比如 foo/*.ned 不处理在 foo/bar/或其它任何子目录中的文件.

程序缺点(1)仅处理.ned 和 .msg 为扩展名的文件,其它的文件都被忽略;(2)不能过滤出文件副本(它们将在文档中显示多次);(3)在 Windows 中,文件名大小写敏感.

11.3.1 多工程

产生的 tags.xml 可以用于生成其它的文档,通过 HTML 链接引用这个文档中页面.

11.4 opp_neddoc 如何工作?

收集*.ned 和 *.msg 文件(如果在 Unix 上使用 find 命令的-a 选项),并使用 nedtool 处理.Nedtool 处理它们,并输出结果语法树至 XML — 独立的大的包括所有文件的 XML 文件.

*.ned 使用 gned 的-c 选项处理(导出图形并退出).这会使得 gned 在 Postscript 中导出复合模块的图形. Postscript 文件然后使用转换工具(ImageMagick 包的一部分)转换为 GIF.gned 也导出一个 images.xml 文件,描述图象是从哪个复合模块生成的,也包括创建可点击图象的附加信息(子模块的矩形坐标和图象中的图标).

包括解析的 NED 和消息文件的 XML 文件然后使用 XSLT 样式表处理生成 HTML.XSLT 是转换一个 XML 文档至另一个 XML(或 HTML,或文本)文档的强有力的方法.另外,样式表读 images.xml,并使用其内容生成可点击的复合模块图象.样式表也输出一个 tags.xml 文件,描述在哪个.html 文件里,记录了什么,因而外部的文件可以与其链接.

最后一步,在生成的 HTML 文件中的注释使用 perl 脚本处理. perl 脚本也实现 HTML 中源代码语法高亮显示的列表,在模块,信道,消息等名称上超链接.(对于以后的任务,它使用 tags.xml 中的信息.)这最后一步,注释格式和源代码的颜色将很难从 XSLT 中获得,(至少在标准 1.0 版本中),完全缺乏强大的字符串操作能力.(在字符串中连简单的查找/替换都不支持,更别说正则表达式.可能 XSLT 的 2.0 版本会改善这个问题.)

使用 opp_neddoc 脚本控制整个过程.

12 并行分布式仿真

12.1 介绍并行离散事件仿真

OMNeT++支持并行执行大量的仿真.下面的段落提供了一简短的问题图,和并行离散事件仿真(PDES)的方法.有兴趣的读者推荐研究文献.

对于并行执行,模型被分割为多个 LP(逻辑处理)在不同的主机或处理上独立地仿真.每个 LP 将有其自身的本地未来事件集,因而,他们将保持他们自身的本地仿真时间.并行仿真的主要问题是保持 LP 同步,从而避免违反事件之前的关系.如果没有同步,由一个 LP 发送的消息到达另一个 LP,接收 LP 中的仿真时间已经传递了消息的时间戳(到达时间).在接收 LP 中,这将违反其事件的因果关系.

有两个主要的并行仿真算法的类别,区别在于处理上述问题的方法:

1. 保守算法可防止非因果关系发生.NULL 消息算法利用时间的知识,当 LP 发送消息至其它的 LP 时,使用’null’消息来传播这个信息至另一个 LP.如是一个 LP 知道它不从其它 LP 接收消息,直至 t+Δ t 仿真,它可能不需要外部的同步,前进至 t+Δ t.如果模型中没有足够的平行关系,或平行关系不通过发送足够的 null 消息来加以利用的话, 保守仿真趋于收敛至连续的仿真(通过 LP 之间的通信减慢).

2. 乐观同步允许非因果关系发生,但是检测,并修复它们.修复包括回滚先前的状态,在消息发送期间通过发送反消息来取消发送达到回滚等.乐观同步很难实现,由于它需要定期保存状态,和以及恢复先前状态的能力.在任何情况下,在 OMNeT++中实现乐观仿真同步需要—除了更复杂的仿真内核—用户写明显更复杂的简单模块,乐观同步可能由于过多的回滚会很缓慢.

12.2 访问仿真模型中可用的平行关系

OMNeT++目前通过 Chandy-Misra-Bryant(或 null 消息)算法,支持保守同步[chandymisra79].为了评定仿真如何有效地使用这个算法并行化,我们需要以下的变量:

- P performance 表示每秒处理的事件的数量(ec/sec).

  [注意:ev:事件,sec”真实的秒,simsec:仿真秒]

  P 依赖于硬件的性能和处理一个事件的计算强度.P 独立于模型的大小.依赖于仿真模型的种类和计算机的性能,P 通过在 20,000..500,000 ev/sec 范围内.

- E event density 是每个仿真秒发生的事件的数量(ev/simsec).E 仅依赖于模型,而不是模型在哪执行.E 通过大小,详细的层次,以及仿真系统的类别决定(比如:单元层次 ATM 模型产生比调用集中仿真更高的 E 值.)

- R relative speed 测量每秒仿真时间前进多少(simsec/sec).R 强烈依赖于模型和模型执行的软/硬件环境.注意 R = P/E.

- L lookahead 在仿真秒测量(simsec).当仿真通信网络,并使用链接延迟预测未来时,L 通常在 msimsec-μ simsec 范围内.

- τ latency (sec)表现的并行仿真硬件的特点. τ 是从一个 LP 发送消息至另一个 LP 的时间.τ 可以使用简单基准程序决定.作者在 Linux 互联的集群上测量,通过一个 100M 以太网,使用 MPI 获得 τ=22μs,与[ongfarrell2000]是一致的测量.特殊的硬件,如 Quadrics Interconnect [quadrics]可以提供 τ=5μs 或更好.

在大的仿真模型中,P,E,R 通常相对固定(即在时间上显示很少的波动).他们也很直观,易于测量. 当仿真运行时,OMNeT++在 GUI 显示它们的值,见下图. Cmdenv 也可以配置显示这些值.

Figure: Performance bar in OMNeT++ showing P, R and E

在有了 P, E, L 和 τ 的近似值之后,计算 λ 耦合因子为 LE 和 τ P 的比率:

```
λ = (LE) / (τ P)
```

不进入细节:如果产生的最小 λ 值大于 1,而是在 10..100 范围内,这是一个好的改变,当并行运行时,仿真可以很好地工作.如果 λ < 1,性能欠佳.细节见论文[ParsimCrit03].

12.3 OMNeT++支持的并行分布仿真

12.3.1 概述

本章提出了 OMNeT++并行仿真的结构.这种设计允许代码不做任何改变,使得仿真模型并行运行—它仅需要配置.实现依赖于占位符模块和代理网关的方法,在不同的 LP 上示例模型—占位符方法允许仿真技术不修改即可以 PDES 工作,如拓扑发现和定向消息发送.这个结构是模块化和可扩展的,因而可以作为一个框架,用于研究并行仿真.

OMNeT++设计强调模型从实验中分离.主要的原理,是通常大量的仿真实验,在得到现实系统的结论之前,需要在单个模型上完成.实验往往是 ad-hoc,改变比仿真模型快速,因而,通常

要求可以进行实验,不考虑仿真模型本身.不需要特殊的源代码工具或拓扑描述,如分割或其它 PDES 配置完全是在配置文件中描述.

OMNeT++支持静态拓扑的 Null 消息算法,使用链接延迟为预测未来.Null 消息发送的缓慢可以调整.也支持 Ideal Simulation Protocol (ISP),在 2000 年由 Bagrodia 提出的. ISP 是一个有力的研究工具来测量 PDES 算法的有效性,包括乐观的和保守的;更确切地说,它有助于通过特定模型和仿真环境的 PDES 算法,确定可达到的最高的加速.在 MNeT++中, ISP 可以用于 Null 消息算法实现的基准.另个,模型执行可以不需要任何的同步,这对于教研目的(演示需要同步)或简单的测试,是很有用的.

对于 LP(逻辑处理)之间的通信, OMNeT++通常使用 MPI,消息传递接口标准[mpiforum94].另一个通信机制是基于命名管道,用于共享存储的多处理器,不需要安装 MPI.另外,基于通信机制的文件系统也是可用的.它通过创建在共享目录中的文本文件通信,可以用于教研的目的(在 PDES 算法中,分析或演示消息)或调试 PDES 算法.基于共享内存通信机制的实现,是今后的计划,完全利用多处理的能力,不需要安装 MPI 的花销.

几乎每个模型都可以并行化运行. 包含以下:

- 模块仅通过发送消息通信(没直接的方法调用或成员访问),除非映射至相同的处理器.
- 没有全局变量
- 在直接发送的时候有一些限制(不能发送至其它模块的子模块,除非映射至相同的处理器)
- 预测未来必须以链接延迟的形式表示.
- 目前支持静态拓扑(我们研究工程的目的是消除这个限制)

在 OMNeT++中支持的 PDES 具有模块化和可扩展的结构.新的通信机制可以通过实现一个补充的 API(特殊的 C++类)来添加和注册实现—在这之后,新的通信机制可以在配置中选择使用.

新的 PDES 同步算法可以用类似的方法添加.PDES 算法也由 C++类表示,必须实现一个非常小的 API 与仿真内核整合.在各个 LP 上设置模型,以及关注通过 LP 的中继模型消息,而不是该同步算法实现需要担心的.(虽然如果需要的话,可以介入,由于提供必须的 hook).

Null 消息算法的实现其本身也是模块化的,因而 lookahead 发现可以通过定义的 API 插入,当前使用链接延迟来实现 lookahead 发现,但是也有可能实现更复杂的,并在配置中选中它们.

新 的偏微分方程同步算法可以被添加在一个类似的方法 。偏微分方程算法， 也派代表参加了由 C + +类已实施一个很小的空气污染指数， 以便与仿真内核。设立该模型对各种毒素以及继电保护模型讯息跨越脂多糖已经是照顾而不是执行该同步算法需要操心 （虽然 它可以介入， 如果有需要， 因为必要的鱼钩提供） 。

实施零讯息算法也是模块化本身就是前进的发现可以插入， 在通过一个定义的 API 。目前已实施前进发现使用环节的延误， 但也有可能推行更多的先进， 并选择他们的配置。

12.3.2 并行仿真例子

我们将使用并行 CQN 例子仿真来演示 OMNeT++的 PDES 性能.模块包括 N 个一前一后的队列,每个队列都由一个开关,k 个指数服务时间的单服务器队列(见下图).最后一个队列循环至本身的开关.每个开关使用均匀分布,随机选择作为目的队列中的一个作为第一个队列.队列和开关与非 0 传播延迟的链接相连接. 我们关于 CQN 的 OMNeT++模型,将队列封装至复合模块.

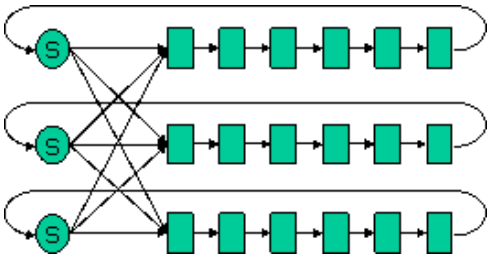


Figure: The Closed Queueing Network (CQN) model

为了并行运行模型,我们分配队列不同的 LP(见下图),在标记的链接上,通过延迟提供 Lookahead.

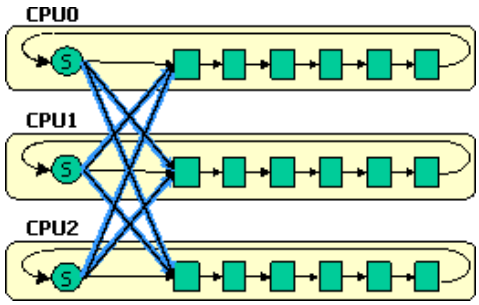


Figure: Partitioning the CQN model

为了并行运行 CQN 模型,我们必须配置它并行执行.在 OMNeT++中,配置是在 omnetpp.ini 文本文件中.对于配置,我们首先必须指定分割,即,分配模块至处理器.这由以下行完成:

```
[Partitioning]

*.tandemQueue[0]**.partition-id = 0

*.tandemQueue[1]**.partition-id = 1

*.tandemQueue[2]**.partition-id = 2
```

等号之后的数字表示 LP.

然后我们必须选择通信库和并行仿真算法,并且允许并行仿真:

```
[General]

parallel-simulation=true

parsim-communications-class = "cMPICommunications"

parsim-synchronization-class = "cNullMessageProtocol"
```

当我们运行并行仿真时,LP 由相同程序的多运行实例来表示.当使用 LAM-MPI [lammpi]时,mpirun 程序(LAM-MPI 的一部分)用于在需要的处理器上启动程序.当选择命名管道或文件通信的话, OMNeT++利用 opp_prun 可以用于启动进程.或者,可以通过手动运行进程(-p 标签告诉 OMNeT++给定 LP 的索引和 LP 的总数):

```
./cqn -p0,3 &

./cqn -p1,3 &

./cqn -p2,3 &
```

对于 PDES,我们通常需要选择命令行用户界面,重定向输出文件.(OMNeT++提供必需的配置选项.)

也可以使用 OMNeT++的图形化用户界面(由下图所显示的),独立于选中的通信机制.GUI 界面可以用于教研或演示的目的. OMNeT++显示关于 Null 消息算法, 可被查看的 EIT 和 EOT 等的调试输出,

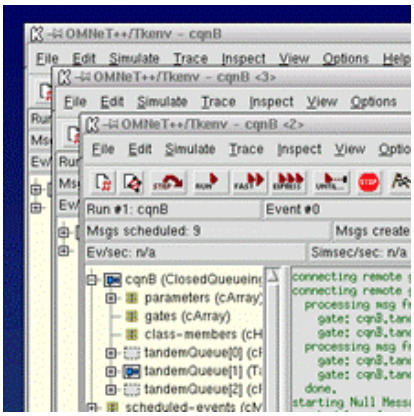


Figure: Screenshot of CQN running in three LPs

12.3.3 占位符模块和代理网关

当建立一个模型分割为多个 LP 时, OMNeT++使用占位符模块和代理网关.在本地的 LP 中,占位符表示同属的子模块在其它 LP 上的示例.由于占位符模块,每个模块在本地 LP 中有其所有的同属—为占位符或”真实的事情”.代理网关关注于转发消息至 LP,其中模块被实例化(见下图).

使用占位符的主要优点是算法比如嵌入至模块的拓扑发现可以使用未更改的 PDES.模块也使用直接消息发送至任何同属模块,包括占位符.这是由于直接消息发送的目的端是目的模块的输入门—如果目的模块是一个占位符,输入门将是一个代理网关,透明地转发消息至 LP,其中”真实”的模块被实例化.有一限制,即直接消息发送的目的端不能是一个同属的子模块(这是一个不好的做法,由于其破坏了封装性),简单的原因是占位符是空的,因而其子模块不能显示在本地的 LP 中.

在复合模块的实例中,可能会更加复杂.由于子模块可以在不同的 LP 上,复合模块不能在任何给定的 LP 上”完全显示”,它必须显示在多个 LP 上(那些有子模块实例的地方).因而,复合模块被实例化,无论在什么情况下它们至少有一个子模块实例,通过占位符表示(见下图).

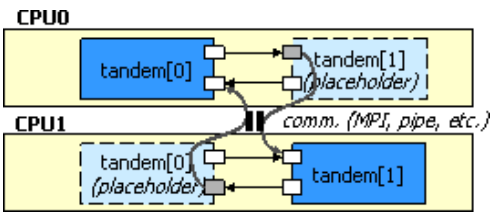


Figure: Placeholder modules and proxy gates

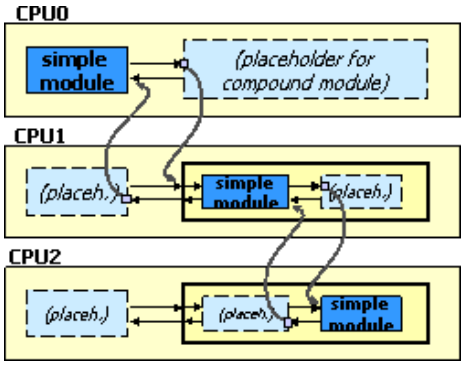


Figure: Instantiating compound modules

12.3.4 配置

并行仿真配置是在 omnetpp.ini 中的[General]部分.

项和缺省值	描述
[General]	
parallel-simulation = <true/false> default: false	允许并行分布仿真. parallel-simulation=true 时,才会检查以下的配置项
parsim-debug = <true/false> default: true	允许调试输出
parsim-mpicommunications-mpibuffer = <bytes> default: 256K * (numPartitions-1) + 16K	分配 MPI 发送缓冲区的大小,见 MPI_Buffer_attach()MPI 调用.如果缓冲区太小,会出现死锁.
parsim-namedpipecommunications-prefix = <string> default: "omnetpp" or "comm/"	控制命名管道的命名. Windows:默认是"omnetpp", 表示管道名将是"\\.\pipe\omnetpp-xx-yy"的形式(其中 xx 和 yy 是数字). Unix:默认值是"comm/",表示创建的命名管道的名称为"comm/pipe-xx-yy". 当仿真启动时,"comm/"子目录必须已经存在
parsim-filecommunications-prefix = <string> default: "comm/"	(见以下)
parsim-filecommunications-preserve-read = <true/false> default: false	(见以下)
parsim-filecommunications-read-prefix = <string> default: "comm/read/"	以上三个选项控制 cFileCommunications 类.默认情况下,删除被读取的文件.通过允许"preserve-read"设置,可以移动读取的文件至另一个目录来代替(默认是"comm/read/").注意:有一些难以理解的原因,在目录中不能超过 19800 文件.当达到这个点里,在标准 C 库内的某个地方会

	抛出异常,在 OMNeT++具体为一个"Error: (null)"消息…奇怪的是,在 Unix 和 Windows 中都可以复制.
parsim-nullmessageprotocol-lookahead-class = <class name string> default: "cLinkDelayLookahead"	为 Null 消息算法选择 lookahead,该类必须是 cNMPLookahead 的子类.
parsim-nullmessageprotocol-laziness = <0..1> default: 0.5	控制 Null 消息算法发送 null 消息的频率;值被理解为与 lookahead 的比例,比如 0,5 表示,每 lookahead/2 仿真秒.
parsim-idealsimulationprotocol-tablesize = <int> default: 100,000	(在表项中)块的大小,其中应该加载外部事件文件(由 cISPEventLogger 记录).(一个项是 8 字节,因此, 100,000 相应的是大约 800K 分配的内存)

当使用 cross-mounted 主目录时(在磁盘上的仿真目录挂载了所有的集群节点),一个有用的配置设置为:

```
[General]

fname-append-host=yes
```

这将导致主机名被追加至所有输出向量文件名.因此分割并不覆盖彼此的输出文件(见[8.10.3]).

12.3.5 OMNeT++支持的 PDES 设计

在 OMNeT++中支持 PDES 设计遵循层次方法,模块化和可扩展的结构.整体结构在下图描述.

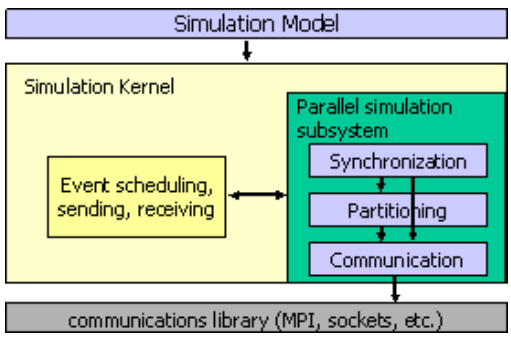


Figure: Architecture of OMNeT++ PDES implementation

并行仿真内核本身是一个可选的组件,如果不需要的话,可以从仿真内核中移除.它包括三个层次:从底至上,通信层,分割层和同步层.

通信层的目的是为上层分割提供基本的消息服务.服务包括发送,阻塞接收,非阻塞接收和广播.发送/接收操作与缓冲器一块工作,其封装了基本 C++类型的打包和解包操作.在仿真库中消息类和其它类可以在这些缓冲中打包,解包.通信层 API 定义在 cFileCommunications 接口中(抽象类);具体的实现类似于 MPI(cMPICommunications)的一个子类,封装 MPI send/receive 调用.匹配缓冲类 cMPICommBuffer 封装 MPI pack/unpack 操作.

为了配置代理门,分隔层负责根据配置中指定的分割,在不同的 LP 上实例化模块.在仿真期间,该层也保证 cross-partition 仿真消息到达其目的地.它解释到达代理门的消息,并使用

通信层服务传输它们至目的 LP.接收 LP 解开消息包,并注入代理门所指向的门.这个实现基本上封装了 cParsimSegment, cPlaceHolderModule, cProxyGate 类.

同步层封装了并行仿真算法.并行仿真算法也表示为类, cParsimSynchronizer 抽象类的子类.并行仿真算法在以下的 hook 上调用:事件调度,处理由 LP 产生的模块信息,和从其它 LP 到达的消息(模块消息或内部消息).第一个 hook,事件调度是一个函数由仿真内核调用,决定下一个仿真事件;它也可以完全访问未来事件集(FES),可以添加/删除其自身所拥有的事件.保守并行仿真算法中,如果下一个事件是不安全的话,将使用这个 hook 来阻塞仿真,比如,如果到达了 EIT 直至 null 消息的到达,null 消息算法实现(cNullMessageProtocol)阻塞了仿真(见 [bagrodia00]术语);也使用这个 hook 来定期发送 null 消息.当发送任何模型消息至另一个 LP 时,调用第二个 hook;对于即将离去的模型消息,null 消息算法使用这个 hook 来传输 null 消息.当从其它 LP 到达任何消息时,调用第三个 hook,允许并行仿真算法从其它部分来处理其自身的内部消息;null 消息算法在这里处理到来的 null 消息.

Null 消息协议实现本身是模块化的,部署了一个单独的,可配置的 lookahead 发现对象.目前仅实现了基于链接延迟的 lookahead 发现,但也可能实现更加复杂的.

Ideal Simulation Protocol (ISP; 见[bagrodia00])实现事实上包括两个并行仿真协议的实现:第一个是基于 null 消息算法加上记录外部事件(从其它 LP 接收的事件)至跟踪文件;第二个是使用跟踪文件执行仿真,找出哪个事件是安全的,哪个是不安全的.

注意,尽管我们实现了一个保守协议,提供的 API 本身也允许实现乐观协议.并行仿真算法可以访问执行仿真模型,因此, 如果模块支持的话,它可以实现保存/恢复模型状态.

[不幸的是,支持状态保存/恢复需要单独地手动添加至仿真中的每个类,包括用户设计的简单模块.]

我们也认为,由于模块性,可扩展性和清除并行仿真子系统的内部结构, OMNeT++框架将潜在地成为 PDES 研究的首选平台.

13 定制和嵌套

13.1 结构

OMNeT++有一个模块化的结构,以下的图显示了 OMNeT++仿真的高级结构:

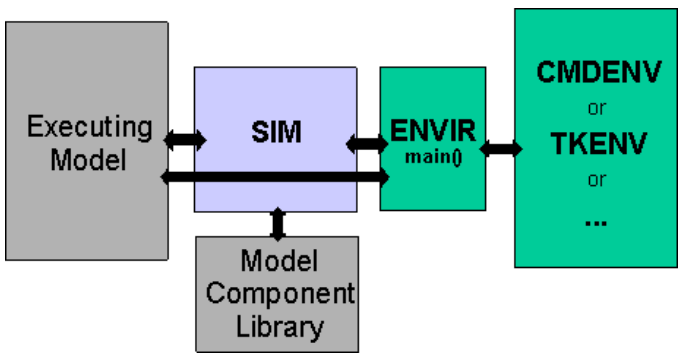


Figure: Architecture of OMNeT++ simulation programs

图中的矩形表示组件:

- Sim 是仿真内核和类库.Sim 存在为链接仿真程序的库.

  [也可能使用动态(共享)库,但为了简洁,我们这里使用链接.]

- Envir 是另一个类库,包括所有用户界面公共的代码.在 Envir 中也有 main().Envir 提供的服务类似于 ini 文件处理特定的用户界面实现. Envir 向 Sim 显示其内容,通

过 ev 门面对象执行模型,隐藏其它的内部用户界面. Envir 的一些方面可以通过插件接口定制.嵌入 OMNeT++应用程序可以达到实现除了 Cmdenv 和 Tkev 之个的,新的用户界面,或以 Envir 取代其它的 ev 实现(见[13.5.3]和[13.2])

- Cmdenv and Tkenv 特殊的用户界面实现.仿真与 Cmdenv 或 Tkenv 链接.
- Model Component Library(模型组件库)包括简单模块定义和它们的 C++实现,复合模块类型,信道,网络,消息类型和一般的属于模块,链接至仿真的内容.一个仿真程序可以运行具有所有必要链接组件的任何模块.
- Executing Model(执行的模块)是为仿真设置的模块.包括在模型组件库中,所有组件的实例对象(模块,信道等).

图中箭头显示了组件如何彼此交互:

- Executing Model vs Sim.仿真内核管理未来的事件,并且当事件发生时,在执行的模型中调用模块.执行模型的模块存储在 Sim 的主对象中, simulation(cSimulation 类).依次,执行模型在仿真内核中调用函数,在 Sim 库中使用类.
- Sim vs Model Component Library.当仿真模型在仿真运行开始的时候被设置时,仿真内核实例化简单模块与其它模块.当使用动态模块创建时,也引用组件库.作为 Sim 的一部分,实现了模型组件库中的注册器与查寻组件.
- Executing Model vs Envir.对象 ev, Envir 的逻辑部分,是用户界面到执行模型的门面.模型使用 ev 写调试日志(ev<<, ev.printf()).
- Sim vs Envir. Envir 完全支配仿真程序中发生的事情. Envir 包括 main()函数,在这里开始执行. Envir 决定了哪个模型应该设置仿真,并指示 Sim 完成. Envir 包括主要的仿真循环(determine-next-event, execute-event 序列),为了达到必需的功能,调用仿真内核(在 Sim 中事件调度和事件执行的实现). 在执行期间,Envir 捕捉处理在仿真内核或类库所抛出的错误和异常. Envir 表现为单个的对象(ev),表示 Sim 的环境(用户界面)— Envir 的内部对于 Sim 或执行的模型是不可见的.在仿真模型的设置期间, 当 sim 要求参数值时,Envir 为 Sim 提供参数值.Sim 通过 Envir 写输出文件,因而可以通过改变 Envir 来重新定义输出向量存储机制.Sim 和其类使用 Envir 打印调试信息.
- Envir vs Tkenv and Cmdenv. Envir 定义 TOmnetApp 为用户界面的基类, Tkenv 和 Cmdenv 都是 TOmnetApp 的子类. main()函数作为 Envir 的一部分,决定了适当的用户界面类(TOmnetApp 的子类),创建一个实例,并运行它 — 下面发生的任何事情(打开一个 GUI 窗口,或运行一个命令行程序)都是由相应 TOmnetApp 子类的 run()方法决定的.Sim 和模型在 ev 对象上的调用都简单地实现为 TOmnetApp 实例.通过 TOmnetApp 和其它的类, Envir 为 Tkenv 和 Cmdenv 提供一个框架和基类.

13.2 OMNeT++嵌套

这部分我们讨论将仿真内核或仿真模型嵌入至一个更大的应用程序的问题.

仿真运行绝对需要的是 Sim 库.如果不需要保留仿真程序的显示,因而不需要 Cmdenv 和 Tkenv.会不会保留 Envir.如果提供了适合设计的理念和基础设施(omnetpp.ini,一定的命令行选项等)可以保留 Envir.那么在应用程序中,嵌入设计将取代 Cmdenv 和 Tkenv.

如果 Envir 不适合你的需求(例如,想从数据库中获得模型参数,而不是 omnetpp.ini),那么必须替换它.你替换的 Envir(尤其是嵌套的应用)必须从 envir/cenvir.h 中实现 cEnvir 成员函数,但可以完全控制仿真.

通常,设置网络或构建复合模块内部的代码是来自编译的 NED 源文件.可能不喜欢限制仿真程序仅应用仿真设置代码链接的网络.没有问题;程序可以包括类似于目前由 nedtool 产生

的代码片,然后可以构建链接组件(主要是简单模块)的任何网络.而且,也可以写一个整合的环境,其中可以使用图形化编辑器,在可以运行之后加上一个网络,不需要介入 NED 编译和链接.

13.3 Sim: 仿真内核和类库

这里较少地介绍 Sim,由于第[4],[6]章,和第[5]章的一部分已经介绍了.这些章节中的所说的类在由 Doxygen 产生的 API 参考文献中有更详细地介绍.我们这里详细阐述一些没有在一般章节中包括的内部资料.

仿真内核和类库的源代码存储在 src/sim/子目录下.

13.3.1 全局仿真对象

全局仿真对象是 cSimulation 的实例.它存储模型,封装了设和运行仿真模型功能的大多数功能.

仿真有两个基本任务:

- 它存储了执行模型的模块.
- 保存了未来事件模块(FES)对象

13.3.2 协同程序包

协同程序包实际上是由两个包组成:

- 可移植的协同包创建主堆栈内的所有协同堆栈.是基于 Kofoed 的解决方法[Kofoed95].通过层层递归分配堆栈,然后使用 setjmp()和 longjmp()切换至另一个.
- 在 Windows 上,使用 Fiber 函数 (CreateFiber(), SwitchToFiber()等),是标准 Win32 API 的一部分.

协同程序由 cCoroutine 类表示. cSimpleModule 接受 cCoroutine 作为基类.

13.4 模型组件库

所有编译链接至仿真程序的模型组件(简单模块定义和它们的 C++实现,复合模块类型,信道,网络,消息类型等)都在模型组件库中注册登记.任何具有仿真程序组件库中必需组件的模型都可以由仿真程序运行.

如果你的仿真程序链接至 Cmdenv 或 Tkenv,可以使用-h 选项,打印其组件库的内容.

```
% ./fddi -h


OMNeT++ Discrete Event Simulation  (C) 1992-2004 Andras Varga

...

Available networks:

  FDDI1

  NRing

  TUBw

  TUBs
```

```
Available modules:
  FDDI_MAC
  FDDI_MAC4Ring
  ...

Available channels:
  ...
End run of OMNeT++
```

组件信息保留在注册登记列表上.有注册组件的宏(即,将其添加至注册列表): Define_Module(), Define_Module_Like(), Define_Network(), Define_Function(), Register_Class(),还有一些其它的.对于在 NED 文件中的定义的组件,宏调用是由 NED 编译器产生的;在其它的情况下必须将其写入 C++源文件中.

让我们看一下模块注册的例子.

```
Define_Module(FIFO);
```

宏展开为以下的代码:

```
static cModule *FIFO__create(const char *name, cModule *parentmod)
{
    return new FIFO(name, parentmod);
}

EXECUTE_ON_STARTUP( FIFO__mod,
    modtypes.instance()->add(
       new cModuleType("FIFO","FIFO",(ModuleCreateFunc)FIFO__create)
    );
)
```

当仿真程序启动时,一个新的 cModuleType 对象将添加至 modtypes 对象,其保存了可用模块类型的列表. cModuleType 将作为一个簇:当调用其 create()方法时,将通过以上的静态函数 FIFO__create 产生一个新的 FIFO 模块对象.

cModuleType 对象也存储了相应 NED 模块声明的名称.这使得当模块创建时,可以添加 NED 文件中声明的门和参数至模块.

注册列表的管理器是 Sim 库的一部分.注册列表实现为一个全局对象.

注册列表是:

列表变量	列表上的宏/对象	函数

networks	Define_Network() cNetworkType	可用网络列表.每个 cNetworkType 是指定网络类型的簇.即, cNetworkType 对象有设置指定网络的方法.在由 NED 编译器产生的代码中,出现 Define_Network()宏.
modtypes	Define_Module(), Define_Module_Like() cModuleType	可用模块类型的列表.每个 cModuleType 是指定模块类型的簇.通常复合模块的 Define_Module()宏是在由 NED 编译器产生的代码中出现;简单模块的 Define_Module()行是由用户添加.
channeltypes	Define_Channel() cChannelType	信道类型列表.每个 cChannelType 作为信道类型的簇,是从 cChannel 派生的类.
classes	Register_Class() cClassRegister	可创建实例的可用的类列表.每个 cClassRegister 对象是指定类对象的簇.该列表由 createOne()函数使用:它可以创建任何类的对象,给定类名为一字符串.(比如声明 ptr = createOne("cArray")创建一个 cArray 对象.)为了允许一个使用 createOne()工作,必须使用 Register_Class(classname)宏注册.
functions	Define_Function() cFunctionType	双精度参数的函数列表,返回一个双精度(见 MathFuncNoArg...MathFunc3Args 类). 一个 cFunctionType 保存指向函数的指针,知道有多少个参数.

13.5 Envir, Tkenv 和 Cmdenv

OMNeT++用户界面的源代码保存在 src/envir/目录(公共部分)和 src/cmdenv/, src/tkenv/目录.

用户界面的类不是从 cObject 派生,与仿真内核完全分离.

13.5.1 main() 函数

OMNeT++的 main()函数简单设置用户界面,并运行它.实现的仿真是由 cEnvir::run()完成.

13.5.2 cEnvir 界面

cEnvir 类仅有一个实例,一个称为 ev 的全局对象:

cEnvir ev;

cEnvir 基本上是一个门面,它的成员函数包括很少的代码. cEnvir 维护一个指向完成所有实际工作的动态分配仿真应用程序对象(从 TOmnetApp 派生)的指针,

cEnvir 成员函数实现以下的任务组:

- 模块活动的 I/O;每个用户界面的实际实现是不同的(比如 Cmdenv 为 stdin/stdout, Tkenv 为 windowing).
- cEnvir 为仿真内核提供方法来访问配置信息(如,模块参数设置)
- cEnvir 也提供仿真内核调用的方法,通知部分用户事件(删除一个对象;模块创建或删除;消息被发送或传递等)

13.5.3 定制 Envir

可以通过插件接口定制 Envir 的一些方面.支持以下的插件接口:

- cRNG. 随机数产生器接口.
- cScheduler. 调度类.这个插件接口允许实现实时,硬件循环,分布式并行仿真.
- cConfiguration. 它定义获得所有配置的类.即,它选择让你用一些其它的实现来取代 omnetpp.ini,比如数据库输入.
- cOutputScalarManager. 它处理记录标量输出数据,通过 cModule::recordScalar() 函数族输出.默认的输出标量管理器是 Envir 库定义的 cFileOutputScalarManager .
- cOutputVectorManager.它处理记录 cOutVector 对象的输出.默认输出向量管理器是 cFileOutputVectorManager,在 Envir 库中定义.
- cSnapshotManager. 它提供了一个输出流至写快照管理器(见[6.10.5]).默认快照管理器是 cFileSnapshotManager,在 Envir 库中定义.

类(cRNG, cScheduler 等)在 API 参考文献中记载.

为了实际地实现,选择使用的插件:

1. 使用给定界面类(比如对一个定制的 RNG, cRNG)的子类来创建自己的版本.
2. 通过将 Register_Class(MyRNGClass)行加入 C++源文件中注册一个类.
3. 编译链接你的接口类至 OMNeT++仿真可执行文件.重要的是:确定你的可执行文件实际上包括了你的类代码.在优化链接器时(特别是在 Unix 上)趋向于省略那些非外部引用的代码.
4. 在 omnetpp.ini 中添加一项来告诉 Envir 使用你的类来取代默认的.对于 RNG 来说,这个设置是在[General]中的 rng-class.

Ini 文件项允许选择你的插件类为 configuration-class, scheduler-class, rng-class, outputvectormanager-class, outputscalarmanager-class 和 snapshotmanager-class, 在[8.3.6]中记载.

插件类如何能够访问配置

通过 cEnvir 的 config()方法,配置对于插件类百可用的,其返回一个配置对象(cConfiguration).这使得插件类有它们自己的配置项.

从[General]部分读取 parsim-debug 布尔项的例子,默认为 true:

```
bool debug = ev.config()->getAsBool("General", "parsim-debug", true);
```

配置插件启动顺序

配置插件的启动顺序如下(见源代码中的 cEnvir::setup()):

1. 首先, 读取 omnetpp.ini(或通过命令行选项"-f"指定 ini 文件).
2. 加载在[General]/load-libs 中的共享库.(也可以是命令行选项"-l"指定的)
3. 检查[General]/configuration-class,如果有,实例化给定的配置对象.配置对象会进一步从 ini 文件(比如数据库链接参数,蔌 XML 文件名)读取项.
4. 删除原始的 omnetpp.ini cInifile 配置对象.其它设置不从其获得.
5. 处理新配置对象的[General]/load-libs.
6. 然后一切正常,使用新建的配置文件.

13.5.4 用户界面的实现：仿真应用程序

仿真应用的基类是 TOmnetApp.指定从 TOmnetApp 派生的用户界面如 TCmdenv, TOmnetTkApp.

TOmnetApp 的成员函数大多是虚拟的.

- 它们中的一部分实现了 cEnvir 函数(在前部分描述)
- 其它实现了所有用户界面的公共部分(例如,从配置文件读取选项;使选项在仿真内核中有效)
- run()是纯虚函数(每个用户界面不同)

TOmnetApp 的数据成员:

- 一个指针指向保存配置文件内容的对象(cInifile 类型)
- 可以从配置文件设置的选项和参数(这些成员以 opt_开头)

仿真应用:

- 添加新的仿真选项
- 提供一个 run() 函数
- 实现函数,离开 TOmnetApp(像 breakpointHit(), objectDeleted())

14 NED 语言文法

NED 语言, OMNeT++的网络拓扑描述语言使用扩展的 BNF 符号给定.

空格,水平制表符和新字符计数行作为界定,因而在两个描述元素之间需要它们中的一个或多个,否则不可分离. '//'(两个斜线)可以用于在行最后写注释.语言仅名称的大小写,而不区分关键字的大小写.

在这个描述中,对于一个或多个使用空格,制表符或新行字符分隔的 xxx,支持{xxx...}符号,{xxx...}支持使用一个逗号和(可选)空格,制表符或新行字符分隔的一个或多个 xxx.

为了方便读取,在一些情况下我们使用文本定义. networkdescription 符号是语法句子符号.

notation	meaning
[a]	0 or 1 time a
{a}	a
{a,,,}	1 or more times a, separated by commas
{a...}	1 or more times a, separated by spaces
a|b	a or b
`a'	the character a
bold	keyword
italic	identifier

```
networkdescription ::=
    { definition... }

definition    ::=
      include
    | channeldefinition
    | simpledefinition
    | moduledefinition
    | networkdefinition

include ::=
    include { fileName ,,, } ;

channeldefinition ::=
    channel channeltype
     [ delay numericvalue ]
     [ error numericvalue ]
     [ datarate numericvalue ]
    endchannel

simpledefinition ::=
    simple simplemoduletype
     [ paramblock ]
     [ gateblock ]
    endsimple [ simplemoduletype ]

moduledefinition ::=
    module compoundmoduletype
     [ paramblock ]
     [ gateblock ]
     [ submodblock ]
     [ connblock ]
```

```
    endmodule [ compoundmoduletype ]

moduletype ::=
    simplemoduletype | compoundmoduletype

paramblock ::=
    parameters: { parameter ,,, } ;

parameter ::=
    parametername
    | parametername : const [ numeric ]
    | parametername : string
    | parametername : bool
    | parametername : char
    | parametername : anytype

gateblock ::=
    gates:
     [ in: { gate ,,, } ; ]
     [ out: { gate ,,, } ; ]
gate ::=
    gatename [ '[]' ]

submodblock ::=
    submodules: { submodule... }

submodule ::=
    { submodulename : moduletype [ vector ]
     [ substparamblock... ]
     [ gatesizeblock... ] }
  | { submodulename : parametername [ vector ] like moduletype
     [ substparamblock... ]
```

```
     [ gatesizeblock... ] }

substparamblock    ::=
    parameters [ if expression ]:
      { substparamname = substparamvalue,,, } ;

substparamvalue ::=
    ( [ ancestor ] [ ref ] name )
    | parexpression

gatesizeblock ::=
    gatesizes [ if expression ]:
      { gatename vector ,,, } ;

connblock ::=
    connections [ nocheck ]: { connection ,,, } ;

connection ::=
     normalconnection | loopconnection

loopconnection ::=
    for { index... } do
      { normalconnection ,,, } ;
    endfor

index ::=
    indexvariable '=' expression ``...'' expression

normalconnection ::=
     { gate { --> | <-- } gate [ if expression ]}
   | {gate --> channel --> gate [ if expression ]}
   | {gate <-- channel <-- gate [ if expression ]}
```

```
channel ::=
     channeltype
    | [ delay expression ] [ error expression ] [ datarate expression ]


gate ::=
    [ modulename [vector]. ] gatename [vector]

networkdefinition ::=
    network networkname : moduletype
     [ substparamblock ]
    endnetwork

vector ::=    '[' expression ']'

parexpression ::=
    expression | otherconstvalue

expression    ::=
      expression + expression
    | expression - expression
    | expression * expression
    | expression / expression
    | expression % expression
    | expression ^ expression
    | expression == expression
    | expression != expression
    | expression < expression
    | expression <= expression
    | expression > expression
    | expression >= expression
```

| expression ? expression : expression

| expression and expression

| expression or expression

| not expression

| '(' expression ')'

| functionname '(' [expression ,,,] ')'

| - expression

| numconstvalue

| inputvalue

| [ancestor] [ref] parametername

| sizeof '(' gatename ')'

| index

numconstvalue ::=

integerconstant | realconstant | timeconstant

otherconstvalue ::=

'characterconstant'

| "stringconstant"

| true

| false

inputvalue ::=

input '(' default , "prompt-string" ')'

default ::=

expression | otherconstvalue

15 References

[BT00] R. L. Bagrodia and M. Takai. Performance Evaluation of Conservative Algorithms in Parallel Simulation Languages. 11(4):395--414, 2000.

[CM79] M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. IEEE Transactions on Software Engineering, (5):440--452, 1979.

[EHW02] K. Entacher, B. Hechenleitner, and S. Wegenkittl. A Simple OMNeT++ Queuing Experiment Using Parallel Streams. PARALLEL NUMERICS'02 - Theory and Applications, pages 89--105, 2002. Editors: R. Trobec, P. Zinterhof, M. Vajtersic and A. Uhl.

[EPM99] G. Ewing, K. Pawlikowski, and D. McNickle. Akaroa2: Exploiting Network Computing by Distributing Stochastic Simulation. In Proceedings of the European Simulation Multiconference ESM'99, Warsaw, June 1999, pages 175--181. International Society for Computer Simulation, 1999.

[For94] Message Passing~Interface Forum. MPI: A Message-Passing Interface Standard. 8(3/4):165--414, 1994.

[Gol91] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Computing Surveys, 23(1):5--48, 1991.

[Hel98] P. Hellekalek. Don't Trust Parallel Monte Carlo. ACM SIGSIM Simulation Digest, 28(1):82--89, jul 1998. Author's page is a great source of information, see http://random.mat.sbg.ac.at/.

[HPvdL95] Jan Heijmans, Alex Paalvast, and Robert van~der Leij. Network Simulation Using the JAR Compiler for the OMNeT++ Simulation System. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[Jai91] Raj Jain. The Art of Computer Systems Performance Analysis. Wiley, New York, 1991.

[JC85] Raj Jain and Imrich Chlamtac. The P^2 Algorithm for Dynamic Calculation of Quantiles and Histograms without Storing Observations. Communications of the ACM, 28(10):1076--1085, 1985.

[Kof95] Stig Kofoed. Portable Multitasking in C++. Dr. Dobb's Journal, November 1995. Download source from http://www.ddj.com/ftp/1995/1995.11/mtask.zip/.

[LAM] LAM-MPI home page. http://www.lam-mpi.org/.

[Len94] Gábor Lencse. Graphical Network Editor for OMNeT++. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[LSCK02] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An Objected-Oriented Random-Number Package with Many Long Streams and Substreams. Operations Research, 50(6):1073--1075, 2002. Source code can be downloaded from http://www.iro.umontreal.ca/~lecuyer/papers.html.

[MN98] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudorandom Number Generator. ACM Trans. on Modeling and Computer Simulation, 8(1):3--30, 1998. Source code can be downloaded from http://www.math.keio.ac.jp/~matumoto/emt.html.

[MvMvdW95] André Maurits, George van Montfort, and Gerard van~de Weerd. OMNeT++ Extensions and Examples. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1995.

[OF00] Hong Ong and Paul~A. Farrell. Performance Comparison of LAM/MPI, MPICH and MVICH on a Linux Cluster Connected by a Gigabit Ethernet Network. In Proceedings of the 4th Annual Linux Showcase & Conference, Atlanta, October 10-14, 2000. The USENIX Association, 2000.

[PFS86] Bratley P., B. L. Fox, and L. E. Schrage. A Guide to Simulation. Springer-Verlag, New York, 1986.

[PJL02] K. Pawlikowski, H. Jeong, and J. Lee. On Credibility of Simulation Studies of Telecommunication Networks. IEEE Communications Magazine, pages 132 --139, jan 2002.

[Pon91] György Pongor. OMNET: An Object-Oriented Network Simulator. Technical report, Technical University of Budapest, Dept. of Telecommunications, 1991.

[Pon92] György Pongor. Statistical Synchronization: A Different Approach of Parallel Discrete Event Simulation. Technical report, University of Technology, Data Communications Laboratory, Lappeenranta, Finland, 1992.

[Pon93] György Pongor. On the Efficiency of the Statistical Synchronization Method. In Proceedings of the European Simulation Symposium (ESS'93), Delft, The Netherlands, Oct. 25-28, 1993. International Society for Computer Simulation, 1993.

[Qua] Quadrics home page. http://www.quadrics.com/.

[SVE03] Y. Ahmet Sekercioglu, András Varga, and Gregory~K. Egan. Parallel Simulation Made Easy with OMNeT++. In Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands. International Society for Computer Simulation, 2003.

[Var92] András Varga. OMNeT++ - Portable Simulation Environment in C++. In Proceedings of the Annual Students' Scientific Conference (TDK), 1992. Technical University of Budapest, 1992. In Hungarian.

[Var94] András Varga. Portable User Interface for the OMNeT++ Simulation System. Master's thesis, Technical University of Budapest, 1994. In Hungarian.

[Var98a] András Varga. K-split -- On-Line Density Estimation for Simulation Result Collection. In Proceedings of the European Simulation Symposium (ESS'98), Nottingham, UK, October 26-28. International Society for Computer Simulation, 1998.

[Var98b] András Varga. Parameterized Topologies for Simulation Programs. In Proceedings of the Western Multiconference on Simulation (WMC'98) Communication Networks and Distributed Systems (CNDS'98), San Diego, CA, January 11-14. International Society for Computer Simulation, 1998.

[Var99] András Varga. Using the OMNeT++ Discrete Event Simulation System in Education. IEEE Transactions on Education, 42(4):372, November 1999. (on CD-ROM issue; journal contains abstract).

[Vas96] Zoltán Vass. PVM Extension of OMNeT++ to Support Statistical Synchronization. Master's thesis, Technical University of Budapest, 1996. In Hungarian.

[VF97] András Varga and Babak Fakhamzadeh. The K-Split Algorithm for the PDF Approximation of Multi-Dimensional Empirical Distributions without Storing Observations. In Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997, pages 94--98. International Society for Computer Simulation, 1997.

[VP97] András Varga and György Pongor. Flexible Topology Descripti on Language for Simulation Programs. In Proceedings of the 9th European Simulation Symposium (ESS'97), Passau, Germany, October 19-22, 1997, pages 225--229, 1997.

[VSE03] András Varga, Y. Ahmet Sekercioglu, and Gregory~K. Egan. A practical efficiency criterion for the null message algorithm. In Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands. International Society for Computer Simulation, 2003.

OMNeT++学习笔记（1）

OMNeT++是一个事件驱动的仿真器，适合做离散事件网络系统仿真。通常可进行通信系统通信模型仿真、协议仿真、硬件体系结构验证、复杂软件系统性能评估、任何其他离散事件驱动应用的建模与仿真。

OMNeT++在原理和结构上与 OPNET（大名鼎鼎的商用仿真器）相似。而它是免费的，且上手很快。

一、安装

首先确保系统中已安装一个 c++编译器，推荐 MSVC6+sp6，并确保设置了环境变量。

二、OMNeT++结构

Simple(基本块，由 c++编写的.cc 文件定义)，module(包含若干子模块（submodules），以及它们之间的连接(connections)说明)，network（有模块组成）

文件：.ned（网络拓扑定义），.cc(基本块定义)，omnetpp.ini（配置文件，可以设置参数）。

仿真内核库和用户接口库（*.lib/*.a）

三、OMNeT++命令行编译

1、 opp_nmakemake -f （创建 Makefile.vc，将包含当前目录中所有.ned/.cc/omnetpp.ini)

2、 nmake -f Makefile.vc depend（增加 depend)

3、 nmake -f Malefile.vc （编译）

每当源文件内容作了修改，直接进行第 3 步；如果源文件名有修改（增删），则必须重新执行第 1 步来创建新的 Makefile.vc 文件。

四、建模步骤

1、 确定系统中的实体(entities)，在网络中通常是节点

2、 设置网络拓扑。创建一个.ned 文件来描述网络

3、 定义事件模式。OMNeT++中用消息表示事件。消息从一个 module 发送到另一个 module，对于接收 module，消息的到达就是一个事件。也可以向自己发送消息来安排未来事件（达到延迟效果）。

4、 事件处理顺序：先发生的先处理，同时发生的按优先级来处理，优先级也相同的按先调度（发送，安排）先处理。

五、定义模块动作（行为）

1、 事件处理

2、 事件调度

六、收集统计数据

OMNet++仿真基础

OMNet++是面向对象的离散事件模拟工具，为基于进程式和事件驱动两种方式的仿真提供了支持。

OMNet++采用了混合式的建模方式，同时使用了 OMNet++特有的 ned（NEtwork Discription）语言和 C++进行建模。

在 ned 中，主要的实体是模块（module）。模块分为两种，一种是普通模块（simple）；一种为复合模块（compound）。模块有 gates，模块之间通过门（gates）进行消息（message）传输。

下面是一个 simple 模块的实例：

```
simple Acceptor
    gates:
        in: in;
        out: out;
endsimple
```

Acceptor 包括了一个输出 gate 和一个输入 gate。

而复合模块有一组模块将相互之间的 gates 连接而成，比如：

```
import "client", "server";

module MyModel
    submodules:
        client1: Client;
        server1: Server;

    connections:
        client1.out --> delay 10ms --> server1.in;
endmodule
```

对于 simple 模块来说，其行为还需要使用 c++进行定义，而对于复合模块就不需要了。

ned 语言还可以定义自己的 message 格式。在完成消息格式、ned 和 c++代码以后，使用 opp_nmakemake，就可以直接生成 VC 的 makefile。如果是类 unix 平台，用 opp_makemake 即可。

编译完成，得到一个可执行文件。为这个可执行文件添加一个配置文件交 omnetpp.ini，就可以在任意机器上执行仿真过程，完全脱离仿真平台了。

Linux 下安装 OMNeT++

出处：http://xlq0372.blog.sohu.com/

我安装的是 OMNeT++3.3 的版本，首先你可以去 OMNeTpp 的官方网站 http://www.omnetpp.org/下载最新版本。以 3.3 为例：下载的是 omnetpp-3.3-src.tgz

1)解压缩： ~#tar zxvf omnetpp-3.3-src.tgz

2)由于后边安装的时候,omnet++会提示需要 tcl 和 tk 两个安装文件,所以在这里先安装了，以后就省事了，tcl 和 tk 可以去官方网站下载的。http://www.tcl.tk/,跟 omnetpp-3.3 匹配的是 tcl/tk 8.4.+版本，不要下错了。下面以 tcl/tk8.4.16 为例。

3）tcl/tk 下载后，分别解压：#tar zxvf tcl(tk)8.4.16.tgz

之后安装 tcl: ~# cd tcl8.4.16/unix

~# ./configure

~#make

~#make test

~#make install

tk 的安装跟 tcl 类似。

4）在/root/.bashrc 中添加 omnetpp 环境变量

~# vi /root/.bashrc

添加： export PATH=$PATH:/root/omnetpp-3.3/bin

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/root/omnetpp-3.3/lib

export TCL_LIBRARY=/usr/share/tcl8.4

5）如果还有错误出现，就在 omnetpp-3.3 中找到 configure.user，在里边添加：

TK_CFLAGS="-I/usr/include/tcl8.4-fwritable-strings"

TK_LIBS="-L/usr/lib -ltk8.4-ltcl8.4 -I/usr/lib"

6）好了，作完以上准备工作，我们就可以开始安装了

~#cd omnetpp-3.3

~#./configure

如果此时没有 error 出现，你就会在输出的最后看见

YourPATH contains /root/omnetpp/bin. Good!

YourLD_LIBRARY_PATH is set. Good!

TCL_LIBRARYis set. Good!

那么，接下来就可以 make 了

7）~#make

至此，安装成功。

8）PS：你或许还要去/etc/selinux 下做一点小小的改动，不然你后边 simulation 时会失败滴！在该目录下，找到 configure，在 SELINUX 这一行，把值改为 Permissive

9）OK，此时，你可以在 ~#cd omnetpp-3.3/samples 中找一个 sample 试一下，比如 dyna,

~#cd dyna

~#./dyna

成功了，一切搞定！

MNet++安装步骤

OMNeT++是一个事件驱动的仿真器，适合做离散事件网络系统仿真。通常可进行通信系统通信模型仿真、协议仿真、硬件体系结构验证、复杂软件系统性能评估、任何其他离散事件驱动应用的建模与仿真。

OMNeT++在原理和结构上与 OPNET（大名鼎鼎的商用仿真器）相似。而它是免费的，且上手很快。

因为某种需要，所以在 FC5 下安装了一个 OMNeT++，并整理了相关步骤。

1.先从网上下载 omnetpp 包，我的是 omnetpp-3.3-src.gz

然后解压到/root 下，我的是解压到/root 下，其实路径可以自己定。

$tarzxvf omnetpp-3.3-src.gz
$mv omnetpp-3.3 /root/

2.设置环境变量

$vi /root/.bashrc
-->加入如下设置
export PATH=$PATH:/root/omnetpp－3.3/bin
export LD_LIBRARY_PATH=/root/omnetpp-3.3/lib

exportTCL_LIBRARY=/usr/lib/tcl8.4

需要设置 PATH,LD_LIBRARY_PATH 还需要设置 TCL_LIBRARY,让 OMNEST/OMNeT++ GUI 程序能够找到 BLT 库.实际上,./configure 结束后,会检测环境变量是否设置.若未设置(或设置后未 logout, log in), 它会提示你设置.所以在这里先设置了。

3../configure 观察 warning 和 error

下面列出一些 error 的排除方法

(1)configure:error: Cannot build Tcl/Tk apps, probably due to misconfigured ormissing X11 headers or libs. Check config.log for more info!

安装 doxygenimagemagick

如果有 yum 并且能用的话，可以执行

$yuminstall doxygen

类似的命令把以上的安装包装了，如果有安装碟则可以在安装碟中将以下几个 rpm 拷贝到电脑中安装，或者到清华大学的 ftp 中找，里面有,地址是 ftp3.tsinghua.edu.cn。

doxygen-1.4.6-3.i386.rpm
doxygen-doxywizard-1.4.6-3.i386.rpm
ImageMagick-6.2.5.4-4.2.1.i386.rpm
ImageMagick-c++-6.2.5.4-4.2.1.i386.rpm
ImageMagick-c++-devel-6.2.5.4-4.2.1.i386.rpm
ImageMagick-devel-6.2.5.4-4.2.1.i386.rpm
ImageMagick-perl-6.2.5.4-4.2.1.i386.rpm

(2)configure:error: Cannot build Tcl/Tk apps, probably due to misconfigured ormissing X11 headers or libs. Check config.log for more info!

未安装 tcl 或者 tk,可以用 yum 安装也可以上清华找,也可以在安装盘里面找下面几个 rpm。
tcl-8.4.12-4.i386.rpm
tcl-devel-8.4.12-4.i386.rpm
tcl-html-8.4.12-4.i386.rpm
tk-8.4.12-1.2.i386.rpm
tk-devel-8.4.12-1.2.i386.rpm
安装。
(3)configure:error: Tcl/Tk not found, needed for all GUI parts. Version 8.4.0+ anddevel package required. Check config.log for more info!
在 configure.user 中加入
TK_CFLAGS="-I/usr/include/tcl8.4-fwritable-strings"
TK_LIBS="-L/usr/lib -ltk8.4-ltcl8.4 -I/usr/lib"
(4)或许你还需要以下程序，不过很大部分在安装系统时已经带了，以防万一，下来装了吧。
bison-2.1-1.2.1.i386.rpm
bison-devel-2.1-1.2.1.i386.rpm
bison-runtime-2.1-1.2.1.i386.rpm
boost-1.33.1-5.i386.rpm
boost-devel-1.33.1-5.i386.rpm
boost-doc-1.33.1-5.i386.rpm
byacc-1.9-29.2.1.i386.rpm
flex-2.5.4a-37.4.i386.rpm
giftrans-1.12.2-19.i386.rpm
libxml2-2.6.23-1.2.i386.rpm
libxml2-devel-2.6.23-1.2.i386.rpm
libxml2-python-2.6.23-1.2.i386.rpm
4.至此，如果你在 XWindows 下边 configure 那么就可以通过了!下面是我的结果：
[root@localhostomnetpp]# ./configure
checkingbuild system type... i686-pc-linux-gnu
checkinghost system type... i686-pc-linux-gnu
configure:--
configure:reading configure.user for your custom settings
configure:--
checkingfor icc... no
checkingfor gcc... gcc
.....(省略了很多).....
configure:WARNING: Cannot compile or link program with BLT: BLT or BLT -develmay not be installed -- check config.log for details. Plove andScalars require BLT to build.
.....(省略了很多).....
config.status:creating samples/queues/Makefile
config.status:creating samples/tictoc/Makefile
config.status:creating samples/tokenring/Makefile
config.status:creating samples/sockets/Makefile
config.status:creating test/Makefile
WARNING:The configuration script could not detect the following packages:
Graphviz BLT MPI (optional) Akaroa (optional)
Scrollup to see the warning messages (use shift+PgUp key), and seeconfig.log formore details. While you can use OMNeT++/OMNEST in the current configuration,please be aware that some functionality may be unavailable orincomplete.

YourPATH contains /root/omnetpp/bin. Good!
YourLD_LIBRARY_PATH is set. Good!
TCL_LIBRARYis set. Good!
[root@localhostomnetpp]#

之所以会出现几个 warning，是因为 MPI 和 Akaroa 没有安装，不过没有什么大碍，因为是可选项目。你可以下载并安装消除 warning。

6.好了，最后一步

[root@localhostomnetpp]#make
.....(一大堆信息).....

至此安装成功了。

7。运行

[root@localhostomnetpp]#cd /root/omnetpp-3.3/samples/dyna
[root@localhostomnetpp]#./dyna

或许你的会出现这个提示： errorwhile loading shared libraries /root/omnetpp/lib/libnedxml.so cannotrestore segment prot after reloc:

关闭 SELinux 就好了，把 selinux 的值改成 Permissive

再次运行。结果如下。。。。。

1.设置 msvc。
打开选单 -> 工具(Tools) -> 自订(Customize)
点选 Add-ins and Macro files 分页标签(Tab) , 看到 omnetpp 选项, 将之勾选
回到命令(Commands) 分页标签(Tab), 分页左上角的分类(Category) 选单会多出宏(Macros) 选项.
选择宏(Macros) 选项, 看到 addNEDfileToProject 宏,
点选 addNEDfileToProject 宏, 使用鼠标将巨集拖曳到工具列上,
选择合适的位置, 图示后按确定.

在完成上两步之后, 已经将 VC 环境设定完成。

2.从头开始。
1), 建立工程文件夹 project
2), 其次, 定义*.ned, 用 文本工具或者在 omnet++Gend 中图形建立然后设置也可以。放在 project 文件夹下。
3), 开始建立工程。
omnet 安装目录/sample 中复制 .dsw, .dsp workspace 到工程的文件夹中.

复制 work space 的目的是因为范例 work space 已经将 omnet 编译所需的编译器, 连结旗标,Tkenv/Cmdenv 参数等都设定好了, 不需再重新设定.
点击 .dsw 进入 VC 编辑画面. 将 (File View) 中的文件名称清空后, 即可开始编辑新的工程。

4), 开始工程
首先用设置的 addNEDfileToProject 宏添加*.ned 文件到工程中，会自动形成*.cpp 文件，打开编辑它，将程序写入。

5), 编译工程
点击“开始”中的“打开”--》cmd
在 dos 环境下切换到工程目录，键入 opp_nmakemake -e cpp

出现如下提示：
```
opp_nmakemake: you have both .cc and .cpp files  -- use -e cc or -e cpp option to
 select which set of files to use
```

```
E:\work\programs\omnet\tictor>opp_nmakemake -e cpp
Creating Makefile.vc in E:/work/programs/omnet/tictor...
Makefile.vc created.
Please type `nmake -f Makefile.vc depend' NOW to add dependencies!
```

说明成功形成 Makefile.vc
再键入 nmake -f Makefile.vc，输出如下：
```
Microsoft (R) Program Maintenance Utility    Version 6.00.8168.0
Copyright (C) Microsoft Corp 1988-1998. All rights reserved.

        cl.exe /nologo -c /EHsc /GR /FD /Zm250 /O2 /DNDEBUG /D_CRT_SECURE_NO_DEP
RECATE   -IC:/OMNeT++/include  /Tp titctor_n.cpp
titctor_n.cpp
        link.exe /nologo /subsystem:console /opt:noref titctor_n.obj     /libpath
:C:/OMNeT++/lib envir.lib tkenv.lib tcl84.lib tk84.lib /libpath:"C:/OMNeT++/lib"
 sim_std.lib nedxml.lib libxml2.lib iconv.lib wsock32.lib    /out:tictor.exe
```

Creating library tictor.lib and object tictor.exp
成功，你会发现工程文件夹下有了*.exe 文件了。

不过还没有完。

6)，设置参数
用记事本写一个 omnetpp.ini,将需要的参数传入，具体操作代码参考 omnet 文档。

7)，运行。窗口出来了，选定参数，运行，如果需要采集仿真数据，则可以在 omnetpp.ini 中设置可以自动生成*.vec 纪录文件用于统计。

8）重新编译工程
之前已编译过时资料夹中会存有旧的 Makefile.vc 档案, 可键入 opp_nmakemake -f 来强制覆写 Makefile.vc .

在编译前要清除过时连结资料, 可以在命令列中输入
nmake Makefile.vc clean 也可以在 VC 中使用 选单/Build/Clean 选项将过时连结资料清除。

over。

注：从 msvc 中编译的方法：为了更容易看到仿真结果, 可以透过以下方法将仿真切换到在 TKenv 环境下执行:

选单/Build/Set Active Project Configuration
从四个选项中选取 xx- win32 Release Tkenv

（但是还没有成功，我编译后的运行是在 dos 下执行的）

还有一种方法，成功了，
选单/组建/配制/将 xx- win32 Release Cmd 那两个删去，build 就可以在图形截面运行了。

Example Application: SeMA

传感器查询机制(SQS)提供了本地查询优化和网络内关于消息传递能量效率的消息串联.该协议的目的是作为层次结构的最底层,通过一个服务感知 adhoc,获得监测应用中的目标域的微型传感器.该项目归类于相对于数据产生特性的 SQS 查询,解释了该查询系统中的组件.

在这项研究中,传感器节点假定是布署在一个 adhoc 的目的域中,节点间构成一个反树状.汇聚节点散布一个查询,并通过反广播树收集返回的信息.

考虑传感器网络的特性,在该项目中提出了发送者初始路切换算法.算法允许即时发送者动态改变包的路由,当在反广播树中的父节点出现故障. 在网络的生存能力上的整体路径切换效率分析来衡量可靠事件传递的方法。

http://cse.yeditepe.edu.tr/tnl/sqs.php?lang=en

http://cse.yeditepe.edu.tr/tnl/rewise.php?lang=en

http://cse.yeditepe.edu.tr/tnl/sema.php?lang=en

节点配置

```
#include "gen/csmRouterTest_TinyOSModule.h"
#include "gen/csmRouterTest_Main.h"

   csmRouterTest_Main *m = new csmRouterTest_Main(this,main,"Main",index());
   m->StdControl_init();
   m->StdControl_start();
```

代码生成

NesCT 用于在 gen 目录构建所有的类.在 windows 操作系统中,将下面的 cp 取代为 copy,斜线(/)取代为反斜线(\).

```
[root@sinan tictoc]# cd components/
[root@sinan components]# cp smRouterTest.nc ../Application.nc
[root@sinan components]# cd ..
[root@sinan tictoc]# ./nesct.exe Application.nc
```

done.

更新 Makefile

输入 'opp_makemake -c config -f'来为当前的环境创建一个新的 Makefile.如果你构建的环境是 windows 输入'opp_nmakemake -c config.win32 -f' 来为当前环境创建 Makefile.另外,windows 需要运行'nmake -f Makefile.vc depend' .

注意Nesct 假定 OMNeT++在linux 中的目录是/opt/omnetpp-3.2, windows 中是 c:\omnet++.如果安装在其它目录的话,需要编辑 config 或 config.win32 文件来改变 OMNet++的 nedtool,lib 和 include 目录的路径.

```
[root@sinan tictoc]# opp_makemake -c config -f
Makefile created, adding dependencies...
Done.
```

构建

linux 中输入"make" , windows 中输入"nmake -f makefile.vc" 来构建二进制文件.

```
[root@sinan tictoc]# make
g++ -c -g -fpermissive -fPIC -DWITH_NETBUILDER -w -DTOSNODES=1000 -DLINUX
-DPLATFORM_OMNETPP -I./include -I/root/projects/tinyos-1.x/tos/interfaces
-I./include_tos -I/opt/omnetpp-3.2/include simstart.cc
g++ tictoc1_n.o debug.o simstart.o tinyos.o tinyosmain.o t ossim.o txc1.o -g
-L/opt/omnetpp-3.2/lib -lenvir -lcmdenv -lsim_std -lnedxml -lxml2 -ldl -lstdc++
-lpthread -o tictoc
echo>.tstamp
```

运行

编辑 omnetpp.ini 文件,改变 wait-for-sf 为 true.在允许这个选项之后,在开始 sf 链接之前仿真将等 10 秒.

通过运行./bin/sf_sim 将启动 SerialForwarder.对于 windows,可能需要输入在控制窗口(cygwin)里的文件本身的内容.一旦启动了的仿真,异常将丢失.

```
[root@sinan tictoc]# ./bin/sf_sim
Listening to tossim-serial
SF enabled, 0 clients, 0 packets read, 0 packets written getenv JNI library not found.
Env.getenv will not work
(please consult installation directions in
tinyos-1.x/tools/java/net/tinyos/util/Env.INSTALL)
Platform avrmote
Opening tossim-serial source
Connecting to Tossim event port at localhost:10585
Listening for client connections on port 9001
SF enabled, 0 clients, 0 packets read, 0 packets written java.net.ConnectException:
Connection refused
```

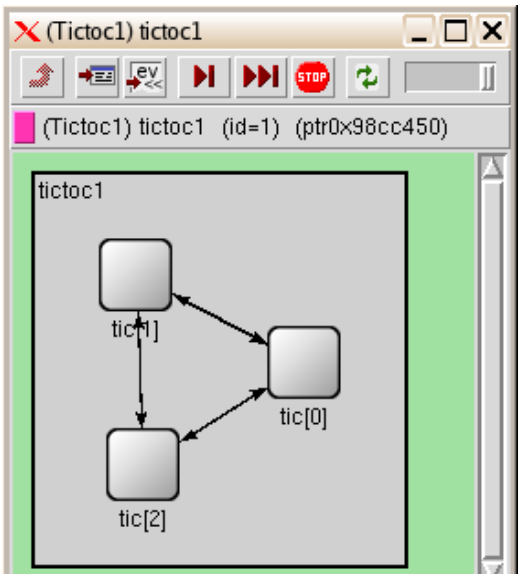
启动另一个控制窗口,输入 java 目录,运行驱动.

```
[root@sinan tictoc]#cd java
[root@sinan tictoc]#java driver.Driver
Created server socket

Thread starting to wait for mote to pc connection

Thread starting to wait for pc to mote connection
```

为了进行演示,我们允许 USR1,USR2 和 USR3 选项. Linux 使用"export"关键字,windows 使用"set"来改变环境变量.应该看到一个类似于 TK 窗口的输出.



仿真屏幕应该像这个.

```
[root@sinan bin]# export DBG=usr1,usr2,usr3
[root@sinan bin]# ./tictoc
```

SeMA 应用默认是设置所有的传感器节点为,启用无线接口的睡眠模式.每当传感器节点接收一个无线消息时,节点唤醒,开始处理消息.因而,节点不做任何事,直至从 PC 注入了消息.SeMA 协议由一个 pc 方的软件 inject_listener",注入消息至网络,转储控制窗口的接收消息.

为了进行演示,打开另一个控制窗口(对于 windows 必须是一个 cygwin 窗口,我们将使用 make 文件系统),进入项目目录, tictoc.

```
$ pwd
/z/projects/src/tictoc
```

进入 msg_inject 目录

```
$ cd 3rdparty/sema/msg_inject
```

输入 make listener 来构建工具

```
$ make listener
gcc main.c -c -g -DLISTENER
gcc main.o fileio.o -g -lm -DLISTENER -o inject_listener.exe
```

Inject_listener 从"params.in"文件中读取查询参数.可以根据仿真需要改变这些值.

这些都是默认值,可以使用"params.in"文件中的参数覆盖.总的来说,一个 setup 消息发送至网络.这个消息告诉网络, SeMA 查询网络必须被构建持续 60 秒,setup 消息需要传播至最多 20 跳.我们不想传感器读数以这种方式串联或获得.因而这是一个最有效的传递.传感器节点每五秒选举获得传感器接口,并且每个定时器返回结果.因此,我们在 pc 端将从每个节点获得 12 消息.就是这么简单.可以要求一传感器的节点 ID 或邻居数同时处理传感器要注.通常,为了适应需求,需要这类信息.对于其它的查询类型,检测 smTypes.h.

```
u16 local_node_id = 0xfffa;
u16 destionation_id = 0;
u16 message_type = SETUP_MESSAGE;
```

```
u16 region_id = 1;
u16 hop_count = 0;
u16 flooding_degree = 20;
u16 query_timeout_type = SEMA_SECOND;
u16 query_timeout_value = 60;
u16 concatenate_flag = 0;
u16 acknowledge_flag = 0;
u16 continuous_flag = 1;
u16 query_type = SENSING_RELATED;
u16 query_frequency_type = SEMA_SECOND;
u16 query_frequency_value = 5;
u16 subquery_count = 1;

u16 subquery_type = SENSING_RELATED;
u16 subquery_sensor_type = TEMPERATURE_SENSOR;
u16 subquery_sample_count = 1;
u16 subquery_compilation_function = LOWER;
u16 subquery_compilation_data = 65535;
```

一旦运行 inject_listener,应该看到像这样的输出:

```
$ ./inject_listener.exe
src_id 65533
dest_id 65535
message_type 0
region_id 5
hop_count 0
flooding_degree 20
query_timeout_type 2
query_timeout_value 60
concatenate_flag 0
acknowledge_flag 0
continuous_flag 1
query_frequency_type 2
query_frequency_value 5
subquery_count 1
subquery_type 0
subquery_sensor_type 2
subquery_sample_count 1
subquery_compilation_function 7
subquery_compilation_data 65535
subquery_type_2 0
subquery_sensor_type_2 0
subquery_sample_count_2 0
subquery_compilation_function_2 0
subquery_compilation_data_2 0
subquery_type_3 0
subquery_sensor_type_3 0
subquery_sample_count_3 0
subquery_compilation_function_3 0
```

subquery_compilation_data_3 0
subquery_type_4 0
subquery_sensor_type_4 0
subquery_sample_count_4 0
subquery_compilation_function_4 0
subquery_compilation_data_4 0
ff ff 14 7d 0f 00 01 7f ff ff ff 45 00 10 78 50 0a 88 05 ff ff c0 00 00 00 00 0
00 00 00 00 00 00 00 00 00
connected to server , socket 3
0 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:1 datacnt:1 tm:10949 data[0]:41
1 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:10960 data[0]:18467
2 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:10960 data[0]:6334
3 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:1 datacnt:1 tm:5005 data[0]:26500
4 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:19169
5 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:15724
6 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:1 datacnt:1 tm:5005 data[0]:11478
7 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:29358
8 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:26962
9 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:1 datacnt:1 tm:5005 data[0]:24464
10 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:5705
11 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:28145
12 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:1 datac nt:1 tm:5005 data[0]:23281
13 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:16827
14 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:9961
15 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:1 datacnt:1 tm:5005 data[0]:49 1
16 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:2995
17 m.type:1 reg_id:5 msg_id:0 resp_cnt:1 hc:2 datacnt:1 tm:5016 data[0]:11942