NS 手册（中文版）

NS 手册翻译小组作品

统筹：ReTurner.D

译制人员（按所翻译章节排序）：

ReTurner.D swarthur inforworm cloudwoods

天竹笑 hritian dekst zxf583

dongchongcao bushuang gaogilbert 小青 lylnuaa

译者序

第一次接触 NS 的时候是做本科毕设，那已经是两年前的事了。这两年，多少 NSer 匆匆而来，又匆匆而去。但他们留下的却是一笔弥足珍贵的经验财富，为后来人指明了方向。两年前的网络论坛上，经常有新手抱怨 NS 的安装很困难，安装就用一个星期那算少的；两年后，一名新手只用几个小时就可以搞定，出了问题，google 一下吧，几乎所有的出错方式都可以找到解决方法。

很久以前就有了想翻译 NS 手册的冲动，一直没有机会与精力。但百思论坛的逐步壮大让我的想法成了现实。在此要特别感谢参与翻译的这 12 位网友，他们大多来自计算机和通信系，对专业术语的把握和理解相当深刻，极大地提高了翻译的质量，而且所翻译的章节都对应了自己的研究方向，使得翻译的结构性和逻辑性有了极为准确的保证。下面是所有参与译制人员的具体分工：

ReTurner.D	1~3, 10~15,35~37,47
swarthur	4~6
inforworm	7~9
cloudwoods	16~17
天竹笑	18~20
hritian	21, 33~34
dekst	22~23
zxf583	24~28
dongchongcao	29~30
bushuang	31~32
gaogilbert	38~41
小青	42~44
lylnuaa	45~46

由于参与译制的好多同学都有毕业设计和找工作的巨大压力，还有本人中途换了个项目，回了趟西工大，折腾耗费了不少精力，所以手册的发布一拖再拖，请大家谅解。

我们在翻译的过程中尽量保持了文章的原汁原味，对名词术语都有专门的中英文对照，并且建了一个临时群以协调好翻译的一致性，但错误还是在所难免，请大家多多批评指正，不吝赐教。请将好的建议与意见发往 ReTurner.D@163.com。我们会在下个版本中及时修正。

如有疑惑，请上百思论坛 www.baisi.net 的NS版块发贴提问。最好先搜寻一下以前发过的贴子，就

不用再麻烦别人了。要注意提问的艺术，提供必要的模拟环境，代码和出错提示，别人才会知道你到底在问什么，并给出你建议或答案。

最后，感谢广大 NSer 的支持，你们的支持是我们坚持下去的最大动力，在此对提供临时群的 LaMer 一并致谢。

ReTurner.D

2007 年 12 月

ns 使用手册
(原ns注解与文档)[1]

VINT 项目
由 UC Berkeley，LBL，USC/ISI 以及 Xerox PARC 的研究人员共同开发
Kevin Fall(kfall@ee.lbl.gov)，编辑
Kannan Varadhan(kannan@catarina.usc.edu)，编辑

2007 年 10 月 31 日

ns©是 LBNL(译者注：Lawrence Berkeley National Laboratory 美国劳伦斯伯克利国家实验室)的网络模拟器[24]。该模拟器用 C++编写；使用 OTcl 作为命令与配置的接口。ns 从第 1 版发展到第 2 版有三个重大的改变：(1)为了有更好的灵活性和组织能力，在第 1 版中太过复杂的对象已经被分解成几个比较简单的组件；(2)现在的配置接口是 OTcl，一种面向对象版本的 Tcl 语言；(3)OTcl 解释器的接口代码从模拟器的主程序中被分离出来。
ns 文件有 html, Postscript 和 PDF 等格式。请参考
http://www.isi.edu/nsnam/ns/ns-documentation网页来获取这些文档。

[1] VINT 项目结合了 UC Berkeley、USC/ISI、LBL 和 Xerox PARC 的人力。该项目由(美国)国防部高级研究计划局(DARPA)在 LBL 的 DABT63-96-C-0105 授权，在 USC/ISI 的 ABT63-96-C-0054 授权和在 Xerox PARC 的 DABT63-96-C-0105 授权所支撑。此文件中任何观点、研究成果、结论或是建议都属于作者，并不代表 DARPA 的看法。

目录

第 1 章

简介

让我们从最初的阶段开始，
一个非常美好的地方开始，
当你歌唱时，你从 A，B，C 唱起，
当你仿真时，你从拓扑[2]的搭建开始，
… …

本文档（ns 的注解与文件）为 ns 的参考文档。虽然我们是从一个非常简单的模拟程序开始，但对 ns 的初学者而言，许多 ns 指导，如 Marc Greis 的指导网页（最初在他的个人网站，现在已经移到 http://www.isi.edu/nsnam/ns/tutorial/）或幻灯片可能会是更好的地方。

首先我们从一个简单的模拟程序开始。这个程序可以在~ns/tcl/ex/simple.tcl 找到。该程序定义了一个包含四个节点（node）的简单的网络拓扑（topology）和两个代理（agent），一个是有 CBR 流量发生器的 UDP agent，另一个是 TCP agent。模拟时间为 3 秒。输出为两个跟踪（trace）文件，out.tr 和 out.nam。当模拟在 3 秒结束时，它将试图执行 nam 程序，并将模拟结果以可视化的方式显示在屏幕上。

```
# 前置动作
set ns [new Simulator]                                    ;# 初始化模拟器
# 预先定义跟踪文件（tracing）
set f [open out.tr w]
$ns trace-all $f
set nf [open out.nam w]
$ns namtrace-all $nf
#那么，现在让我们来定义网络拓扑（topology）
#
#    n0
#          \
#        5Mb \
#            2ms \
#                  \
#                     n2 --------- n3
#                   / 1.5Mb
#         5Mb / 10ms
#       2ms /
#        /
#     n1
```

[2] 在此对 Rodgers 和 Hammerstein 说声抱歉

```
#
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
$ns duplex-link $n0 $n2 5Mb 2ms DropTail
$ns duplex-link $n1 $n2 5Mb 2ms DropTail
$ns duplex-link $n2 $n3 1.5Mb 10ms DropTail
# 一些代理（agent）.
set udp0 [new Agent/UDP]                                    ;# UDP agent
$ns attach-agent $n0 $udp0                                  ;# 在节点n0上
set cbr0 [new Application/Traffic/CBR]                      ;# CBR流量产生器agent
$cbr0 attach-agent $udp0                                    ;# 附加在UDP agent上
$udp0 set class_ 0                                          ;# 实际上是预设，但...
set null0 [new Agent/Null]                                  ;# 这是接收器（sink）
$ns attach-agent $n3 $null0                                 ;# 在节点n3上
$ns connect $udp0 $null0
$ns at 1.0 "$cbr0 start"
puts [$cbr0 set packetSize_]
puts [$cbr0 set interval_]
# TCP/Tahoe之上的FTP，从节点n1到节点n3, flowid为2
set tcp [new Agent/TCP]
$tcp set class_ 1
$ns attach-agent $n1 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink
set ftp [new Application/FTP]                               ;# TCP不会产生自己的流量
$ftp attach-agent $tcp
$ns at 1.2 "$ftp start"
$ns connect $tcp $sink
$ns at 1.35 "$ns detach-agent $n0 $tcp ; $ns detach-agent $n3 $sink"
# 这个模拟程序会执行3秒。
# 模拟会在调度器（scheduler）调用下面的finish{}程序后结束。
# 这个程序会关闭所有trace文件，并调用nam将其中一个trace文件以可视化方式呈现。
$ns at 3.0 "finish"
proc finish {} {
global ns f nf
$ns flush-trace
close $f
close $nf
puts "running nam..."
exec nam out.nam &
exit 0
}
```

```
# 最后，开始模拟。
$ns run
```

第 2 章

未文档化的工具(Undocumented Facilities)

ns 的发展经常伴随着新协议的增加。可惜的是，相关文档却不能经常更新。本节列出哪些部分需要整理成文档，哪些部分需要改进。
(如果你想添加它的话，本文档位于 ns 源代码的 doc 子目录下。☺)

解释器的接口(Interface to the Interpreter)
- ●目前没有

模拟器基础(Simulator Basics)
- ●局域网部分需要更新以支持新的有线/无线标准(Yuri 更新它了？)
- ●需要增加对无线网络的支持
- ●在队列管理那一章应明确列出队列选项

支持(Support)
- ●应将数据包管理单独列出，加入文档
- ●应将 bin 目录下的迹后处理(trace-post-processing)作用加入文档

路由(Routing)
- ●链路状态的使用与设计以及 MPLS 路由模块毫无记录可言。(注：自 2000 年 9 月 14 日后，链路状态与 MPLS 只出现在每日快照与发布中)
- ●需要增加层次(hierarchical)路由/寻址部分(Padma 已经完成)
- ●需要增加一个章节以支持 ad-hoc 路由协议

排队(Queueing)
- ●CBQ需要建立文档(或许可以建立ftp://ftp.ee.lbl.gov/papers/cbqsims.ps.Z?)

传输(Transport)
- ●需要添加 MFTP 文档
- ●需要添加 RTP 文档(session-rtp.cc 等)
- ●需要添加组播模块文档
- ●应修改并将snoop与tcp-int[3]加入文档

流量与场景(Traffic and scenarios)(新增章节)
- ●应增加如何从 trace 驱动模拟器的描述
- ●应增加对场景发生器的讨论
- ●应增加对 http 流量源的讨论

应用(Application)
- ●是 non-Haobo http 资料记载的么？不是。

规模(Scale)
- ●应增加对混合模式的讨论(待定)

仿真(Emulation)
- ●目前没有

[3] 译者注：tcp-int(integrated TCP congestion control)整合性 TCP 拥塞控制。

其它(Other)

- 应增加准入控制策略么？
- 应增加一个审定章节并照搬 ns-test.html 的内容
- 应照搬 Marc Greis’ s 的指导而非仅仅参考它么？

第一篇

解释器的接口

第 3 章

OTcl联接(Linkage)

ns 是一个用 C++语言编写，以 OTcl 解释器为前台，面向对象的模拟器。该模拟器支持 C++中类的层次结构(本文档中也称作编译层次) 和 OTcl 解释器中的相似层次结构(本文档中也称作解释层次)。这两种层次结构紧密相关 ;从用户的角度看，解释层次中的类和编译层次中的类是一一对应的。这种层次中的基类是 TclObject 类。用户通过解释器创建新的模拟对象；这些对象先是在解释器中被实例化，然后由编译层次中相应的对象来产生映射。解释类层次通过 TclClass 类中定义的方法自动创建，用户则是通过 TclObject 类中定义的方法映射这些实例化的对象。C++代码和 OTcl 脚本里面还有其它的层次，但这些层次并不会以 TclObject 的方式被映射。

3.1 概念综述

为什么要使用两种语言？因为模拟器需要有两种不同类型的事情要做。一方面，协议的细节性模拟需要一种系统的编程语言，这种语言能够有效地控制字节，分组头(packet header),实现使用大规模数据的算法。对于这些任务，执行时的速度是相当重要的，而回转(turn-around)时间(执行模拟，找出错误，修正错误，重新编译，重新运行)相对而言不是那么重要了。

但另一方面，相当一部分的网络研究对可变参数，配置或是快速搜索大量场景都只有微小的变化。在这些网络研究里，迭代时间(改变模式并重新运行)变得更加重要。既然配置只设置一次(在模拟开始的时候)，这部分工作的执行时间就不再那么重要了。

ns 使用 C++和 OTcl 两种语言同时满足了这两种需求。C++能够快速执行便变化得慢，使其适于具体协议的实现。OTcl 执行得比较慢却可以迅速地变化(交互式的)，使其成为模拟配置的理想对象。ns(通过 tclcl)提供了使两种语言的对象和变量同时出现的纽带。

欲了解更多关于脚本语言(scripting language)和分裂语言(split-language)编程思想的信息，请参考 IEEE Computer[26]上 Ousterhout 的文章。关于网络模拟分裂层次编程的更多信息，请参考 ns paper[2]。

两种语言分别何时使用？使用两种语言，我们将面对何种语言应用于何种场合的问题。
我们的基本建议是：
OTcl 适用于

- ●配置、安装以及一次性的工作
- ●如果你能使用已有的 C++对象做你所想做的

C++适用于：

- ●如果你做的需要处理数据流里的每个分组
- ●如果你必须以不可预测的方式改变已有的 C++类的行为

例如，链接(links)是 OTcl 对象，它集合了延迟(delay)，排队(queueing)，和可能的丢失模型(loss modules)。如果你的实验只有这些就可以完成，那当然最好。但是如果你想做得更实际点(一种特殊的 queueing 策略或 loss 模型)，你就需要一个新的 C++对象了。

确实有一些灰色地带：大多数路由在 OTcl 中实现(虽然其核心算法 Dijkstra 算法是在 C++中实现的)。我们模拟 HTTP 时，让每条数据流在 OTcl 中启动，而每个 packet 的处理却在 C++中进行。这种方法是可行的，直到模拟时间中每秒钟的数据流达到 100 条。总之，如果我们需要在每秒钟很多次地调用 Tcl，那最好还是将这些代码移植到 C++代码中。

3.2 代码综述

本文档中，我们用术语解释器(interpreter)来表示 OTcl 中的解释器。解释器的接口代码在单独的 tclcl 目录下。其它的模拟器代码在目录 ns-2 下。我们将用符号 ~tclcl/<file>来表示在 Tcl 目录下的一个特定的<file>。同样的，我们将用符号~ns/<file>来表示在 ns-2 目录下一个特定的<file>。

在~tclcl 里面定义了很多类。我们只关注在 ns 中使用的 6 个类：Tcl 类(第 3.3 节)包含 C++代码将要用来访问解释器的方法；TclObject 类(第 3.4 节)是所有模拟器对象及其在编译层次映射的基类；TclClass 类(第 3.5 节)定义了被解释类的层次，以及允许用户实例化 TclObject 的方法；TclCommand 类(第 3.6 节)用来定义简单的全局解释器的命令。EmbeddedTcl 类(第 3.7 节)包含装载高层内建命令的方法，这些命令使得模拟的配置更加容易。最后，InstVar 类(第 3.8 节)含有访问 C++成员变量作为 OTcl 瞬时变量的方法。

本章介绍的过程与函数可以在~tclcl/tcl.{cc,h}，~tclcl/tcl2.cc，~tclcl/tcl-object.tcl 和~tclcl/tracedvar.{cc,h}中找到。文件~tclcl/tcl2c++.c 用来建立 ns，而且在本章有简略的介绍。

3.3 Tcl类

Tcl 类封装(encapsulates)的是 OTcl 解释器真正的实例，并提供与解释器访问及通信的方法。本节描述的方法和编写 C++代码的程序员相关，该类提供了以下操作方法：

- 获得 Tcl 实例的一个指针
- 通过解释器调用 OTcl 过程
- 取出或将结果返回解释器
- 报告错误状态并以统一的方法退出
- 存储并查找“TclObjects“
- 获取对解释器的直接访问

在下面各小节里，我们分别介绍各种方法。

3.3.1 获取Tcl类实例的指针(reference)

一个简单的类的实例在~tclcl/Tcl.cc 中被声明为一个静态成员变量；程序员必须获取一个这个实例的指针，以访问这个部分描述的其它方法。对该实例的访问需要的命令表述为：

```
Tcl& tcl = Tcl::instance();
```

3.3.2 调用OTcl过程

通过Tcl实例，一共有四种不同的方法来调用一个OTcl命令。他们在调用参数方面有本质的区别。每个函数都传递一个字符串（string）给解释器，然后解释器通过一个全局文本来识别这个字符串。如果解释器返回TCL _ OK，则这些函数将会返回一个相应的OTcl过程。反过来，如果解释器返回TCL _ ERROR，则这些函数将调用tkerror{}。用户可以重载（overload）这个过程，以便有选择地忽略某些类型的错误。OTcl编程的这些复杂过程不在本文档讨论范围之列。下一小节（3.3.3）介绍

了获取解释器返回结果的方法。

●tcl.eval(char* s)调用Tcl_GlobalEval()，通过解释器执行s。

●tcl.evalc(const char* s)保存字符串参数s。它将字符串s复制到中间缓冲区；然后再在中间缓冲区里面调用之前的eval(char* s)。

●tcl.eval()假设命令已经存储在internal bp_类中；它直接调用tcl.eval(char* bp_)。缓冲区自己的指针可以通过tcl.buffer(void)方法获得。

●tcl.evalf(const char* s, . . .)类似于一个Printf(3)。它在内部使用vsprintf(3)来创建输入的字符串。

作为一个例子，这里有一些使用上面所讲函数的使用方法：

```
Tcl& tcl = Tcl::instance();
char wrk[128];
strcpy(wrk, "Simulator set NumberInterfaces_ 1");
tcl.eval(wrk);

sprintf(tcl.buffer(), "Agent/SRM set requestFunction_ %s", "Fixed");
tcl.eval();

tcl.evalc("puts stdout hello world");
tcl.evalf("%s request %d %d", name_, sender, msgid);
```

3.3.3 从解释器传出或传入结果

当解释器调用一个C＋＋方法时，它所期望的是结果能够返回私有成员变量tcl_->result，有两种方法可以设置这个变量。

- tcl.result(const char* s)

将结果字符串s传回解释器。

- tcl.resultf(const char* fmt, . . .)

上面的varargs(3)用vsprintf(3)来格式化结果，再将结果字符串返回给解释器。

```
if (strcmp(argv[1], "now") == 0) {
tcl.resultf("%.17g", clock());
return TCL_OK;
}
tcl.result("Invalid operation specified");
return TCL_ERROR;
```

同样地，当一个C＋＋方法调用一个OTcl命令时，解释器返回结果到tcl_->result。

●tcl.result(void)必须用于取回结果。注意这里的结果是一个字符串，它必须被转化成一个适合结果类型的内部格式。

```
tcl.evalc("Simulator set NumberInterfaces_");
char* ni = tcl.result();
if (atoi(ni) != 1)
tcl.evalc("Simulator set NumberInterfaces_ 1");
```

3.3.4 错误报告与退出

编译代码中提供了一种统一的报告错误的方法。

●tcl.error(const char* s)执行以下功能 :将s写入stdout ;将tcl_->result写入stdout ;退出 ,并将错误代码(error code ）置1。

```
tcl.resultf("cmd = %s", cmd);
tcl.error("invalid command specified");
/*不可达*/
```

需要注意的是，这里所讲的返回原TCL _ ERROR和我们前面一小节（3.3.3）讲的调用Tcl::error()有些微小的差别。前者在解释器内产生一个异常（exception）；用户可以捕获到这个exception从而可能从错误中恢复。如果用户没有设定陷阱（trap），解释器将打印出一个追踪堆栈（stack trace），然后退出。然而，如果代码调用error()，那么模拟的用户将不能捕捉这个错误；同时，ns也将不会打印出任何stack trace。

3.3.5 解释器内的哈希（Hash）函数

ns将每个TclObject在编译层次的一个指针（reference）存在一个hash表中；这样就能快速地访问该对象了。该hash表在解释器的内部。Ns用户以TclObject的名字为关键字（key）在hash表中进行插入、查找或者删除TclObject的操作。

●tcl.enter(TclObject* o)将在hash表插入一个指向TclObject o的指针（pointer）。它被TclClass::create_shadow()用来在对象建立时，将其插入表中。

●tcl.lookup(char* s)将取回名为s的TclObject。可以这样被使用：TclObject::lookup()。

●tcl.remove(TclObject* o)将删除hash表中TclObject o的指针。可以用TclClass::delete_shadow()来移出hash表中存在的入口，此时该对象已经被删除。

这些函数是在TclObject类和TclClass类内部被使用的。

3.3.6 解释器上的其它操作

如果以上的方法依然不够，那么我们必须获得解释器的句柄（handle），用来编写我们自己的函数。

●tcl.interp(void)返回一个解释器的handle，并存储在Tcl类中。

3.4 TclObject类

TclObject类是解释和编译层次大多数其它类的基类（base class）。TclObject类中的每个对象都由用户从解释器中创建。编译层次中同时有一个与之对应的影子对象（shadow object）被创建。这两个对象相互紧密联系。下一小节描述的TclClass类，包含了执行这种投射（shadowing）的机制。

在本文档的剩余部分，我们所说的对象通常指的是TclObject[4]。据此，我们所涉及到的对象要么在TclObject类里，要么

[4] 在ns和ns/tcl最新的发行版里，这个对象被重新命名为SplitObject，更为精确地反映了这个对象存在的本质。然而，现在我们将继续使用TclObject来这些对象和这个类。

在TclObject类的派生类里。如果有必要的话，我们将明确地标出对象到底是在解释器中的，还是在编译器中的。因此，我们将用缩写“解释对象”和“编译对象”来区别两者。这些将在编译代码中分别标志出来。

与版本1的差别 :与版本1不同的是 ,TclObject类包含了早期的NsObject类的函数。因此 ,它储存了变量绑定(bindings ）的接口（ 3.4.2 ）这些变量绑定绑定了解释对象中的实例变量（ instance variables ）和相应的编译对象中的C + + 成员变量（ member variables ）。这种绑定比ns版本1要强，因为OTcl变量的任何变化都是被跟踪（ trapped ）的，而且每次当前的C + + 和OTcl的值被解释器访问后都要保持一致。这种一致性是由InstVar类（ 3.8 ）来完成的。同样，和ns版本1不同的是，TclObject类的对象不再存储在一个全局链表（ link list ）中。而是存储在Tcl类（ 3.3.5 ）的一个hash表中。

TclObject配置的例子：下面的例子是一个SRM agent的配置（ Agent/SRM/Adaptive类 ）。

```
set srm [new Agent/SRM/Adaptive]
$srm set packetSize_ 1024
$srm traffic-source $s0
```

根据ns里的协定（ convention ），Agent/SRM/Adaptive类是Agent/SRM的子类（ subclass ），而Agent/SRM又是Agent的子类 ,Agent类又是TclObject的子类。相应的编译类层次是ASRMAgent ,由SRMAgent派生 ,SRMAgent由Agent派生，Agent由TclObject派生。上面例子中的第一行表明一个TclObject如何被创建（ 或撤销 ）（ 3.4.1 ）；下一行配置一个约束变量（ bound variable ）（ 3.4.2 ）；最后一行解释对象要调用一个C + + 方法充当一个实例过程（ 3.4.4 ）。

3.4.1 创建（ creating ）和撤销（ Destroying ）TclObjects

当用户创建一个新的TclObject时 ,通常调用new{}和delete{}过程(procedures ）;这些过程定义在~tclcl/tcl-object.tcl中。它们可以用于创建和撤销所有类的对象 ,包括TclObjects[5]。在这一节 ,我们将描述在创建TclObject时内部动作(internal actions ）是如何执行的。

创建TclObjects 用户可以调用new{}来创建一个解释类的TclObject。这时解释器将执行这个对象的构造函数（ constructor ）init{} ,同时给它传递用户提供的任何参数。ns自动创建相应的编译对象。shadow对象是通过基类TclObject的构造函数被创建的。因此，新的TclObject的构造函数必须首先调用父类的构造函数。new{}方法返回一个对象的handle，用户可以通过这个handle对对象进行进一步的操作。

下面是Agent/SRM/Adaptive的构造函数：

```
Agent/SRM/Adaptive instproc init args {
eval $self next $args
$self array set closest_ "requestor 0 repairor 0"
$self set eps_ [$class set eps_]
}
```

解释器执行下面一系列动作，作为实例化一个新的TclObject的一部分。为了方便说明，我们就描述创建一个Agent/SRM/Adaptive对象时执行的步骤。这些步骤如下：

1. 从TclObject的名字空间（ name space ）获取一个惟一的新的对象的handle。这个handle将返回给用户。ns中大多数handle都是以 _ o<NNN>的形式出现的，这里的<NNN>是一个整数。这个handle由getid{}创建。它可以从C + + 中的name(){}的方法获得。

[5] 作为例子 ,Simulator 类、Node 类、Link 类以及 rtObject 类不是由 TclObject 类派生而来。这些类中的对象因为不是 TclObjects。但是一个 Simulator、Node、Link 或是 route 对象也是调用 ns 中的 new 过程实例化的。

2. 执行新对象的构造函数。任何的特定的用户（user-specified）输入参数都将作为构造函数的参数传入。这个构造函数必须调用其父类的构造函数。

在上面的例子里，Agent/SRM/Adaptive在第一行就调用了其父类。

需要注意的是，每个构造函数，依次调用其父类的构造函数[6]。那么ns中的最后一个构造函数就是TclObject的构造函数。这个构造函数用来创建对应的shadow对象，并执行其它的初始化工作与绑定，我们下面将予以说明。我们最好在构造函数中先调用父类的构造函数，然后再初始化。这样可以使shadow对象先被建立，从而有变量可以绑定。

3. TclObject的构造函数为Agent/SRM/Adaptive类调用create-shadow{}实例过程。

4. 当shadow对象建立以后，ns为编译对象调用所有的构造函数，它们每个都有可能为类中的对象建立相应的变量绑定，同时执行其它的必要的初始化工作。因此我们最好把调用父类的构造函数的语句放在类初始化语句之前。

5. 在shadow对象成功创建后，create_shadow(void)

(a)如前(3.5.5)所述，将新的对象加入到TclObjects的hash表中。

(b)使cmd{}成为一个新创建的解释对象的实例化过程。这个实例化过程会调用编译对象中的command()方法。在接下来的一小节里(3.4.4)，我们将介绍command方法是如何定义并被调用的。

需要注意的是，所有上述的映射机制（shadowing mechanisms）只有当用户通过解释器创建新的TclObject的时候才有效。但如果程序员只是单向创建编译TclObject，这个机制将会失效。因此，程序员不要直接用C++的新方法来创建编译对象。

TclObject的撤销 delete操作将同时撤销解释对象和相当的shadow对象，例如，use-scheduler{ < scheduler > }用delete过程来移除默认的链表计划（list scheduler），同时在原处实例化一个替代的计划。

```
Simulator instproc use-scheduler type {
$self instvar scheduler_
delete scheduler_                    ;# 首先删除已有的list scheduler
set scheduler_ [new Scheduler/$type]
}
```

同构造函数一样，对象的析构函数必须明确地调用其父类的析构函数，作为该析构函数的声明的最后部分。TclObject的析构函数将调用delete-shadow实例过程，从而依次调用对应的编译方法来撤销shadow对象。解释对象将由解释器自身撤销。

3.4.2 变量绑定（Variable Bindings）

大多数情况下，访问编译成员变量只能通过编译代码，同样地，访问解释成员变量只能通过解释代码；但是，在它们之间建立一个双向绑定是有可能的，这就使得解释成员变量和编译成员变量都可以访问同样的数据，而且它们中的任何一个变量值的变化均可以使另一个相应地变为同样的值。

这种绑定是对象在实例化的时候，由编译对象的构造函数建立起来的；它也作为一个实例变量被解释对象自动访问。ns支持五种不同的数据类型：实型（reals），带宽变量（bandwidth valued variables），时间变量（time valued variables），整型（integers），布尔型（booleans）。对各种不同的变量类型来说，OTcl中的值的语法是不一样的，如下说明：

[6] 译者注：原文中的 ad nauseum 是拉丁词汇，意即令人厌烦，枯燥无味的。

- 实型和整型变量以“一般形式”指定（specified），例如：

```
$object set realvar 1.2e3
$object set intvar 12
```

- 带宽变量像实型变量一样指定，而且后缀可以随意地加‘k’或者‘K’表示kilo-quantities，后缀‘m’或者‘M’表示mega-quantities，最后可以加一个后缀‘B’表示是字节/秒。默认的带宽单位表示为比特/秒。例如，下面所有的表述是等价的：

```
$object set bwvar 1.5m
$object set bwvar 1.5mb
$object set bwvar 1500k
$object set bwvar 1500kb
$object set bwvar .1875MB
$object set bwvar 187.5kB
$object set bwvar 1.5e6
```

- 时间型可以像实型一样指定，可以选择加上后缀‘m’表示单位为毫秒（milli-seconds），或是‘n’表示纳秒（nano-seconds），或是‘p’表示皮秒（pico-seconds）。默认的时间单位为秒。例如，下面的表述是等价的：

```
$object set timevar 1500m
$object set timevar 1.5
$object set timevar 1.5e9n
$object set timevar 1500e9p
```

注意，我们也可以在最后加上一个s表示时间的单位为秒。ns将忽略除了有效的实数表示和后缀为‘m’，‘n’，及‘p’外的任何其它符号。

- 布尔型既可以像整型一样表示，也可以用‘T’ or ‘t’来表示真。第一个字母外的后续符号将被忽略。如果这个值既不是整型，也不是真值，那么默认为假。例如：

```
$object set boolvar t                    ;# 置为真
$object set boolvar true
$object set boolvar 1                    ;# 或其它非零值（真）
$object set boolvar false                ;# 置为假
$object set boolvar junk
$object set boolvar 0
```

下面是ASRMAgent[7]的构造函数：

```
ASRMAgent::ASRMAgent() {
bind("pdistance_", &pdistance_);              /* 实变量 */
bind("requestor_", &requestor_);              /* 整型变量 */
bind_time("lastSent_", &lastSessSent_);       /* 时间型变量 */
bind_bw("ctrlLimit_", &ctrlBWLimit_);         /* 带宽型变量 */
bind_bool("running_", &running_);             /* 布尔型变量 */
}
```

需要注意的是，上述所有的函数都需要两个参数：OTcl变量的名字和绑定的相应的编译成员变量的地址。通常来说，这些绑定是在由构造函数建立的，但是也可以由其它方式完成。我们将在接下来详细介绍InstVar类（3.8）时，讨论这些其它

[7] 注意这个函数被修改过，以用来突出变量的绑定机制。

的替代方法。

每个被绑定的变量在对象创建的初始化时候，自动地被赋予默认值。这些默认值被指定为解释类变量。这个初始化过程是在init-instvar{}中被执行的，而这又是被Instvar类中的方法调用的（3.8中介绍）。init-instvar{}检查（check）解释对象的类和该对象所有的父类，从而找到定义变量的第一个类。它利用那个类中的变量值来初始化这个对象。大部分绑定的初始化值定义在~*ns*/tcl/lib/ns-default.tcl中。

例如，如果下面的类变量是为ASRMAgent定义的：

```
Agent/SRM/Adaptive set pdistance_ 15.0
Agent/SRM set pdistance_ 10.0
Agent/SRM set lastSent_ 8.345m
Agent set ctrlLimit_ 1.44M
Agent/SRM/Adaptive set running_ f
```

因此，对于每个新的Agent/SRM/Adaptive对象，将变量pdistance_的值设置为15.0，变量lastSent_的值设置为8.345m，这个是在其父类中的变量设置的；变量ctrlLimit_的值设置为，这个是在它的上两层父类变量中设置的；变量running的值设置为false。实例变量pdistance_没有被初始化，因为它不存在于解释对象层次的任何现存的类中。在这个例子中，init-instvar{}将调用warn-instvar{}来打印出关于该变量的警告。用户可以在自己的模拟脚本里有选择的覆盖这些过程，以用来避免这一警告[8]。

需要注意的是，实际的绑定过程是在InstVar类的对象实例化过程中完成的。InstVar类中的每个对象绑定一个编译成员变量和一个解释成员变量。一个TclObject存储一个InstVar对象及相应的成员变量的链表。这个链表头被存储在TclObject的成员变量instvar_里。

最后一点需要注意的是，ns将确保解释对象和编译对象中的变量的实际值在任何时候都要保持一致。然而，如果编译对象里有一些方法和其它变量跟踪这个变量的值，只要当这个变量的值发生改变，它们就必须被明确地调用或是改变。这就通常要求用户调用一些附加原语（primitives）。在ns中提供这种原语的一种方式就是通过下一节介绍的command()方法。

3.4.3 变量跟踪（Variable Tracing）

除变量绑定以外，TclObject同时也支持对C++和Tcl实例变量的跟踪。一个被跟踪的变量既可以在C++又可以在Tcl中创建和配置。如果在Tcl层建立变量跟踪，变量必须在Tcl下是可视的，这也就意味着它必须是一个绑定的C++/Tcl，或者是一个纯Tcl实例变量。此外，拥有跟踪变量的对象同样要用TclObject中的Tcl跟踪方法来建立跟踪。trace方法的第一个参数必须是变量的名字，第二个可选的参数指定了负责跟踪那个变量的trace对象。如果trace对象没有被指定，那么拥有这个变量的对象将负责跟踪它。

如果一个TclObject去跟踪变量，它必须扩展C++中的trace方法，那本来是定义在TclObject中的一个虚函数。Trace类只实现一个简单的trace方法，因此，它可以作为一个一般的变量跟踪函数。

```
class Trace : public Connector {
...
virtual void trace(TracedVar*);
};
```

下面是一个在Tcl中设置跟踪变量的简单的例子：

[8] 这里的警告指的是未对上文中的变量 pdistance_进行初始化。

```
# \$tcp跟踪它自己的变量cwnd_
\$tcp trace cwnd_

# \$tcp中的变量ssthresh_ 被一个一般的\$tracer跟踪
set tracer [new Trace/Var]
\$tcp trace ssthresh_ \$tracer
```

如果一个C + +变量是可跟踪的，那么它必定属于TracedVar类的派生类。虚基类TracedVar拥有跟踪变量的名字，所有者以及跟踪对象。从TracedVar类派生的类都必须实现虚函数的值，它是把一个字符缓冲区当作一个参数，并把变量的值放入这个缓冲区。

```
class TracedVar {
...
virtual char* value(char* buf) = 0;
protected:
TracedVar(const char* name);
const char* name_;                    // 变量的名字
TclObject* owner_;                    // 拥有这个变量的对象
TclObject* tracer_;                   // 当变量改变时的反馈
...
};
```

TclCL库里输出两种TracedVar类：TracedInt类和TracedDouble类。这两种类可以分别用于基本类型int和double。TracedInt和TracedDouble都重载（overload）了像赋值、增加和减少之类可以改变变量值的运算符。这些重载运算符用assign方法将新值赋给变量，并当新值不同于旧值时调用tracer。TracedInt和TracedDouble同样也实现了它们的value方法，用于将变量的值以字符串输出。 输出串的宽度和精度可以事先指定。

3.4.4 command方法: 定义与调用（Invocation）

对于每个建立的TclObject，ns都将建立cmd{}实例过程，作为一个挂钩（hook）通过编译的shadow对象来执行一些方法。cmd{}过程自动调用shadow对象的command()方法，并将command()方法的参数以矢量的形式传递给cmd{}过程。用户可以通过下面两种方式中的一种来调用cmd{}：通过显式地调用过程，指定所需的操作作为第一个参数，或者隐式地调用，就好象有一个同名的实例过程作为所需的操作。模拟脚本大多都采用后者，因此，我们先介绍后面那种调用方式。

我们都知道SRM中的距离计算是在编译对象中完成的；然后，它却通常是被解释对象使用。它通过以如下方式被调用：

```
$srmObject distance? (agentAddress)
```

如果没有实例过程叫做distance?，那么解释器将调用实例过程unknown{}，这个过程在TclObject基类中定义。然后，unknown过程将调用

```
$srmObject cmd distance? <agentAddress>
```

通过编译对象的command()过程来执行这些操作。

当然，用户也可以显式地直接调用操作。其中一个原因可能是通过一个同名的实例过程来重载这个操作。例如：

```
Agent/SRM/Adaptive instproc distance? addr {
$self instvar distanceCache_
```

```
if ![info exists distanceCache_($addr)] {
set distanceCache_($addr) [$self cmd distance? $addr]
}
set distanceCache_($addr)
}
```

我们现在就以ASRMAgent::command()为例，来说明command()方法是如何定义的。

```
int ASRMAgent::command(int argc, const char*const*argv) {
Tcl& tcl = Tcl::instance();
if (argc == 3) {
if (strcmp(argv[1], "distance?") == 0) {
int sender = atoi(argv[2]);
SRMinfo* sp = get_state(sender);
tcl.tesultf("%f", sp->distance_);
return TCL_OK;
}
}
return (SRMAgent::command(argc, argv));
}
```

我们可以从上述代码中得出以下几点：

- 函数调用时需要两个参数：

第一个参数(argc)代表解释器中该行命令说明的参数个数。

命令行参数向量(argv)包括：

— argv[0]包含方法的名字，“cmd”。

— argv[1]说明所需的操作 specifies the desired operation.

— 如果用户要指定其它的参数，它们就被放在argv[2...(argc - 1)]。参数是以字符串的形式被传递的，它们必须被成合适的数据类型。

- 如果操作匹配成功，将通过前面的方法（3.3.3）返回操作的结果。

- command()本身必须以TCL_OK或TCL_ERROR作为它的返回代码，以表明成功或是失败。

- 如果操作在这个方法中未能匹配，它必须调用父类的command方法，并返回相应的结果。这就允许用户创建和实例过程或编译方法一样层次特性的操作。

当command方法是为多继承（multiple inheritance）类定义时，程序员可以自由地选择其中一个实现方式：

1)可以调用其中一个父command方法，然后返回该调用相应的结果，或

2)可以以某种顺序依次调用每一个的父command方法，然后返回第一次成功调用的结果。如果没有一个调用成功，将返回错误。

在本文档里，我们把通过command()执行的操作叫做准成员函数（instproc-likes）。这个名字反映了这些操作作为一个对象的OTcl实例过程的用途，但是在实际和用途中还有一些微小的差别。

3.5 TclClass类

这个编译类(class TclClass)是一个纯虚类。从这个基类派生的类提供两种功能：构建与编译类层次镜像的解释类层次；和提供实例化新TclObjects的方法。每一个这样的派生类与编译类层次中的特定的编译类相互关联，同时可以实例化这个关联类的新对象。

例如，拿RenoTcpClass类来说，它是由TclClass类派生，并和RenoTcpAgent类相关联。它将实例化RenoTcpAgent类中的新对象。RenoTcpAgent的编译类层次结构是：RenoTcpAgent由TcpAgent派生，TcpAgent由Agent派生，而Agent（粗略地说）由TclObject派生。RenoTcpClass定义如下：

```
static class RenoTcpClass: public TclClass {
public:
RenoTcpClass() : TclClass("Agent/TCP/Reno") {}
TclObject* create(int argc, const char*const* argv) {
return (new RenoTcpAgent());
}
} class_reno;
```

我们可以从这个定义里面得出如下几点：

1. 类只定义了构造函数和另外一个方法，以用来创建关联的TclObject实例。

2. 当静态变量class_reno第一次创建时，ns将执行RenoTcpClass的构造函数。这就建立了适当的方法和解释类层次。

3. 构造函数明确地说明了构造函数为Agent/TCP/Reno。这也同时隐式地说明了解释类的层次结构。
回想ns中的约定，斜杠（‘/’）用来作为分隔符。对给出的任意A/B/C/D类，A/B/C/D类是A/B/C的子类，而A/B/C类本身又是A/B的子类，同理，A/B是A的子类，而A本身则是TclObject的子类。
在上面的例子里，TclClass的构造函数创建了三个类，Agent/TCP/Reno子类，Agent/TCP子类 以及Agent子类。

4. 这个类与RenoTcpAgent类相互关联：它在这个关联类中创建新对象。

5. RenoTcpClass::create方法返回RenoTcpAgent类中的TclObjects。

6. 当用户定义一个新的Agent/TCP/Reno，常规的RenoTcpClass::create将被调用。

7. 参数向量vector (argv)包含：
— argv[0]包含对象的名字
— argv[1...3]包含$self, $class，以及$proc。由于create通过create-shadow实例过程调用，argv[3]中包含create-shadow。
— argv[4]包含用户提供的任何附加参数（以字符串形式传递）。

Trace类表明参数由TclClass方法控制。

```
class TraceClass : public TclClass {
public:
TraceClass() : TclClass("Trace") {}
```

```
TclObject* create(int args, const char*const* argv) {
if (args >= 5)
return (new Trace(*argv[4]));
else
return NULL;
}
} trace_class;
```

新的Trace对象用如下方式创建：

```
new Trace "X"
```

最后，详细地说明一下解释类层次是如何构建的：

1. 当ns初次启动时，执行对象的构造函数。
2. 这个构造函数以解释类名为其参数，调用TclClass的构造函数。

3．TclClass的构造函数存储这个类的名字，并把这个对象插入到TclClass对象的链表中。
4．在模拟器的初始化过程中，Tcl_AppInit(void)调用TclClass::bind(void)。
5．对TclClass对象链表中的每一个对象，bind()调用register{}，以解释器名作为其参数。
6．register{}建立层次类，建立那些必需，却还创建的类。
7．最后，bind()为新类定义create-shadow和delete-shadow两个实例过程。

3.5.1 如何绑定静态C＋＋类成员变量

在3.4节中，我们已经知道如何将C＋＋对象中成员变量导入到OTcl空间中。但这并不适用于C＋＋类的静态成员变量。当然，你可以给每一个C＋＋类的静态成员变量创建一个对应的OTcl变量；显然这违背了静态成员的本义。

我们不能用与绑定TclObject（基于InstVar）的相似的解决方法来解决这种绑定问题（静态类成员变量的绑定），因为TclCL中的InstVar需要一个TclObject的出现。但是我们可以为相应的TclClass创建一个方法，然后通过这个方法访问C＋＋类中的静态成员变量。其过程如下：

1． 如上述创建一个你自己的TclClass的派生类；

2． 在你创建的派生类中，声明bind()方法和method()方法；

3． 用add_method("your_method")实现自己的bind()时创建自己的绑定方法，然后用与TclObject::command()相似的方法实现method()中的handler。注意传递给TclClass::method()和TclObject::command()的参数个数是不一样的，前者要在前面多两个参数。

我们将以*~ns*/packet.cc中的PacketHeaderClass的简单版本为例。假充我们有下面的Packet类，它里面有一个静态变量hdrlen_，现在我们想从OTcl中访问它：

```
class Packet {
……
static int hdrlen_;
};
```

接着我们通过以下为该变量构建一个访问途径（accessor）：

```
class PacketHeaderClass : public TclClass {
protected:
PacketHeaderClass(const char* classname, int hdrsize);
TclObject* create(int argc, const char*const* argv);
/* 这是两个OTcl类的访问方法的实现 */
virtual void bind();
virtual int method(int argc, const char*const* argv);
};
void PacketHeaderClass::bind()
{
/* 调用基类的bind()必须在add_method()之前 */
TclClass::bind();
add_method("hdrlen");
}
int PacketHeaderClass::method(int ac, const char*const* av)
{
Tcl& tcl = Tcl::instance();
/* 注意这个参数的转移；然后我们就可以像在TclObject::command()里那样控制它们 */
int argc = ac - 2;
const char*const* argv = av + 2;
if (argc == 2) {
if (strcmp(argv[1], "hdrlen") == 0) {
tcl.resultf("%d", Packet::hdrlen_);
return (TCL_OK);
}
} else if (argc == 3) {
if (strcmp(argv[1], "hdrlen") == 0) {
Packet::hdrlen_ = atoi(argv[2]);
return (TCL_OK);
}
}
return TclClass::method(ac, av);
}
```

在此之后，我们可以使用下面的OTcl命令来访问或者改变Packet::hdrlen_的值：

```
PacketHeader hdrlen 120
set i [PacketHeader hdrlen]
```

3.6 TclCommand类

该类（TclCommand类）仅仅为ns提供向解释器输出简单命令的机制，然后由解释器在全局范围内执行。这里有两个函数：ns-random和ns-version，定义在~*ns*/misc.cc里。这两个函数通过定义在~*ns*/misc.cc中的init_misc(void)初始化；init_misc则是在启动时由Tcl_AppInit(void)调用。

- VersionCommand类中定义了ns-version命令。它不包含任何参数，同时返回ns当前版本的字符串。

```
% ns-version                                    ;#  获取当前版本号
2.0a12
```

- RandomCommand类定义了ns-random命令，也不包括任何参数，同时返回一个整数，且这个整数在区间$[0, 2^{31} - 1]$上服从均匀分布。

如果指定一个参数，它会将该参数视为种子。如果该种子的值为0，命令将使用一个自定义的种子值；否则，它将给随机数字产生器设置特定的种子，从而获得特定的值。

```
% ns-random                                      ;#  返回一个随机数
2078917053
% ns-random 0                                    ;#自定义地设置种子
858190129
% ns-random 23786                                ;#设置特定的种子得到特定的值
23786
```

需要注意的是，我们通常不建议构建用户可以调用的顶层（top-level）命令。我们现在描述如何定义一个新的命令，以say_hello类为例。这个例子定义了hi命令，然后打印出字符串“hello world”，加上用户指定的任意命令行参数。例如：

```
% hi this is ns [ns-version]
hello world, this is ns 2.0a12
```

1. 该命令必须定义在TclCommand类的派生类中。类的定义如下：

```
class say_hello : public TclCommand {
public:
say_hello();
int command(int argc, const char*const* argv);
};
```

2．该类的构造函数必须以命令作为参数调用TclCommand的构造函数。例如：

```
say_hello() : TclCommand("hi") {}
```

TclCommand的构造函数将"hi"设置为全局过程，用来调用TclCommand::dispatch_cmd()。

3．command()方法必须执行其所需的操作。

该方法传递了两个参数。第一个参数是argc，是用户实际传递参数的个数。用户实际传递的参数将以矢量的argv被传递。它包含下面几部分：

— argv[0]为命令名(hi)。

— argv[1...(argc - 1)]为用户有命令行附加的参数。

command()是被dispatch_cmd()调用的。

```
#include <streams.h>                        /*  因为我们要用到I/O流 */
int say_hello::command(int argc, const char*const* argv) {
cout << "hello world:";
for (int i = 1; i < argc; i++)
cout << ' ' << argv[i];
```

```
cout << ' \n' ;
return TCL_OK;
}
```

4．最后，我们需要该类（TclCommand）的一个实例，它可以在init_misc(void)常规方法中创建。

```
new say_hello;
```

需要注意的是，通过还有一些函数如ns-at和ns-now可以通过这种方法访问。大多数这些函数包含在已有的类中。特别地，ns-at和ns-now可以通过TclObject调度访问。这些函数定义在~ns/tcl/lib/ns-lib.tcl中。

```
% set ns [new Simulator]          ;#获取新的模拟器实例
_o1
% $ns now                         ;#向模拟器查询当前时间
0
% $ns at ...                      ;#  为模拟器定义特定的操作
...
```

3.7 EmbeddedTcl类

ns允许对编译代码或者解释代码的功能扩展，这个扩展代码将在初始化的时候被执行。例如，在~tclcl/tcl-object.tcl脚本和~*ns*/tcl/lib的脚本里。脚本的这种装载和赋值是通过EmbeddedTcl类中的对象来完成的。

最简单的扩展ns的方法就是向~tclcl/tcl-object.tcl里面添加OTcl代码，或者在~ns/tcl/lib目录下加入脚本。注意，对于后一种情形，由于ns是自动读取~ns/tcl/lib/ns-lib.tcl源脚本，因此程序员必须向该脚本文件添加几行代码，以便使新添加的脚本在ns启动时自动添加到源文件库中。举个例子，~ns/tcl/mcast/srm.tcl文件定义了几个执行SRM的实例过程。在~ns/tcl/lib/ns-lib.tcl里，我们有下面的语句：

```
source tcl/mcast/srm.tcl
```

使得srm.tcl在ns启动时自动被加入到源文件库中。

EmbeddedTcl代码有三点需要注意的地方：第一，如果赋值的时候代码遇到错误，ns将自动停止运行。第二，用户可以显式地重载脚本中的任何代码。特别地，他们可以做了自己的修正后重新定义整个源文件库脚本。最后，在添加了~ns/tcl/lib/ns-lib.tcl脚本之后，并且每次改变他们的脚本，用户都必须为这些变动重新编译ns才能生效。当然，对于大多数情况来说[9]，用户可以加入他们自己的源文件来重载embedded代码。

这一小节的剩余部分 我们将介绍如何把单个的脚本直接整合到ns中去。第一步 将脚本转化为一个EmbeddedTcl对象。下面的代码行扩展了ns-lib.tcl，并创建了EmbeddedTcl对象实例，名为et_ns_lib：

```
tclsh bin/tcl-expand.tcl tcl/lib/ns-lib.tcl | \
../Tcl/tcl2c++ et_ns_lib > gen/ns_tcl.cc
```

~*ns*/bin/tcl-expand.tcl脚本通过将ns-lib.tcl中所有的source行替换成相应的source文件来扩展了ns-lib.tcl。

~tclcl/tcl2cc.c程序将OTcl代码转化成等价的EmbeddedTcl对象et_ns_lib。

在初始化时，调用EmbeddedTcl::load方法显式地给数组赋值。

— ~tclcl/tcl-object.tcl由Tcl::init(void)方法载入；Tcl_AppInit()调用了Tcl::Init()，这一加载命令的准备地语法为：

```
et_tclobject.load();
```

[9] 少数不行的情况是当一定的变量可能被或不被定义，或者相反的脚本包含代码而不是过程和变量定义并且直接执行且不可以逆转的行为。

— 与之类似的，~ns/tcl/lib/ns-lib.tcl直接被~ns/ns_tclsh.cc中的Tcl_AppInit加载。
et_ns_lib.load();

3.8 InstVar类

这一小节描述InstVar类的内部。该类定义了将编译的shadow对象中的C＋＋成员变量绑定到与之对应的解释对象中的OTcl实例变量上的方法与机制。这种绑定的建立可以使变量的值在任何时候都能从解释器或编译代码中设置和访问。

有五种实例变量类：InstVarReal类，InstVarTime类，InstVarBandwidth类，InstVarInt类以及InstVarBool类，分别对应实型，时间型，带宽型，整型以及布尔型变量的绑定。

我们现在来介绍实例变量建立的机制。我们用InstVarReal类来解释这个概念。然而，这种机制对五种类型的实例变量都可行。

当建立一个解释变量来访问一个成员变量，InstVar类的成员函数假定它们是在以一个适当的方法执行脚本；因此，它们不要求解释器确定文本内中这个变量是必须存在的。

为了保证脚本按正确的方法执行，如果一个变量的类已经在解释器中被创建，并且解释器当前正在对该类的某个对象进行操作，那么这个变量必须被唯一的绑定。注意，前者要求当一个给定的类中的方法通过解释器去使其变量可被访问时，必须定义一个相关联的类（3.5）来识别解释器正确的类层次。因此，执行脚本的合适的方法可以通过下述两种途径中的一种创建。

一种隐式的解决方案是当一个新的TclObject在解释器内被创建时。这将在解释器内建立一个方法执行脚本。当一个解释的TclObject的编译的shadow对象创建时，那个编译对象的构造函数可以用新创建解释对象文本的方式，绑定它对象的成员变量到解释的实例变量。

一种显式的解决方案是在一个command函数中定义一个bind-variables操作，这个操作可以通过cmd方法调用。为了执行cmd方法，必须建立正确的文本执行方法。同样的，编译代码也在此时对shadow对象进行操作，因此可以安全地绑定所需的成员变量。

一个实例变量是通过说明解释变量的名字和编译对象中的成员变量的地址来创建的。基类InstVar的构造函数在解释器里创建一个变量的实例，然后设置一个常规陷阱（trap routine）来捕捉通过解释器来访问变量的所有通道。

任何时候通过解释器来读取变量，常规陷阱就会在读取之前被调用。该routine再去调用相应的get函数用于返回变量的当前值。这个返回的值常常用来设置解释变量的值，然后由解释器来读取这个解释变量的值。

同样的，任何时候通过解释器设置变量，常规陷阱则会在写操作之后被调用。该routine取得由解释器设置的当前值，接着调用相应的set函数，将编译成员的值置为解释器中的当前值。

第二篇

模拟器基本知识

第 4 章

Simulator类

模拟器全都是在 Tcl 的 Simulator 类中定义的。它提供了一套用做模拟配置和选择用于驱动模拟的事件调度器类型的接口。一个模拟脚本通常是以创建一个类的实例开始的，然后调用多种方法来创建节点、拓扑结构以及配置模拟的其他方面。Simulator 类中有一个 OldSim 子类，它是用来支持 NSv1 版本的，以此实现向后兼容。

本章内容涉及到的过程和函数可以在~ns/tcl/lib/ns-lib.tcl，~ns/scheduler.{cc,h}和~ns/heap.h 文件中找到。

4.1 模拟器初始化

当在 Tcl 中创建一个新的模拟对象时，初始化过程将执行以下操作：

- 初始化 packet 格式（调用 create_packetformat）
- 创建一个调度器（缺省情况为时间调度器（calendar scheduler））
- 创建一个空代理（null agent）(一个用于在各种场景下丢弃接收到的 packet 的接收器）

packet 格式初始化在一次模拟中用于设置 packet 内的偏移域。它将在后续的章节（第 12 章　packet 头部以及格式）中得到更详细的讲解。调度器是以事件驱动的方式来执行模拟过程的，它也可以被其他提供不同语义的调度器替换（在下面的部分中有详细讲述）。空代理则是通过下面的调用创建的：

```
set nullAgent_ [new Agent/Null]
```

空代理通常是用作丢弃接收到的packet的接收器，或者用作那些在模拟中不被统计和记录的packet的目的地。

4.2 调度器和事件

Simulator是一个由事件驱动的模拟器。当前模拟器中有四种可用的调度器，其中每一种都是由不同的数据结构实现的：一个简单的链表（linked-list）、堆（heap）、缺省的时间队列（calendar queue）和一种特殊的类型—实时（real-time）。每种数据结构都将在下面进行讲解。调度器是这样运行的：先选取下一个最早的事件执行，执行完毕后再返回执行下一个事件。调度器所使用的时间单位是秒。当前的模拟器是单线程的，因此在给定的任意时刻都只有一个事件能得到执行。如果在同一时刻有多个事件被调度执行，那么它们将以先调度先分配（first scheduled -first depatched）的方式得到执行。调度

器不再对同一时刻的事件进行重排序 （和较早的版本一样），并且所有的调度器都能根据输入得到同样的分配顺序。

调度器不支持事件的部分执行和预占用。

一个事件通常包含一个起始时间和一个句柄函数。事件的完整定义可以在~ns/scheduler.h文件中找到：

```
class Event {
public:
Event* next_;          /*事件链表*/
Handler* handler_;        /*事件就绪时调用的句柄*/
double time_;          /*事件就绪的时间*/
int uid_;                  /*唯一的事件ID */
Event() : time_(0), uid_(0) {}
};
/*
* 所有事件句柄的基类。当一个事件的调度时间结束时，
* 它就被传递给需要消耗时间的句柄。
* 例如，如果它要被释放，那一定要被句柄释放。
*/
class Handler {
public:
virtual void handle(Event* event);
};
```

从基类Event中派生出两类对象 :packet和at-event。Packet将在以后的章节(章节 12.2.1)中进行详细的讲解。At-event对象是一个Tcl过程，它在调度给定的时刻执行。它在模拟脚本中会频繁用到。下面是一个如何使用它的简单例子：

```
...
set ns_ [new Simulator]
$ns_ use-scheduler Heap
$ns_ at 300.5  "$self complete_sim"
...
```

这段Tcl代码先创建了一个模拟对象，然后将其缺省的调度实现替换为基于堆（Heap-based,见下文）的调度，最后调度函数$self complete_sim，使之在300.5秒时执行（注意：这段特殊的代码应该是被封装在一个对象实例过程中，其中已经正确的定义了$self所指的对象）。At-event是作为事件实现的，其中句柄在Tcl解释器中得到了高效执行。

4.2.1 链表调度器

链表调度器（Scheduler/List 类）是用一个简单的链表结构实现的。链表是按照时间顺序（从最早到最晚）组织的，所

以事件的插入和删除都需要扫描链表以发现合适的入口。选择下一个执行的事件需要将链表的第一个入口从表头处删除。这种实现方式保证了在事件执行时对同时发生的事件按照先进先出（FIFO）的原则进行。

4.2.2 堆调度器

堆调度器（Scheduler/Heap 类）是用一个堆结构实现的。这种结构在处理大量事件时优于链表结构，因为 n 个事件的插入和删除的时间复杂度为 O（log *n*）。在 NSv2 中的这种实现是由 MaRS-2.0 模拟器[1]借鉴而来的；而大家公认的 MaRS 本身又是借鉴了 NetSim[2]的代码，尽管这种线型的借鉴关系还没有得到确切的证实。

4.2.3 时间队列调度器

时间队列调度器（Scheduler/Calendar 类）是用这样的一种数据结构实现的：类似于桌子上的年历，同月同日但不同年发生的事件可以被记录在同一天。它的正式描述在[6]中，非正式描述则可以在 Jain（第 400 页）[15]中找到。NSv2 中时间队列的实现要归功于 David Wetherall（现就职于麻省理工的计算机科学实验室，MIT/LCS）。

4.2.4 实时调度器

实时调度器（Scheduler/RealTime 类）试图将事件的执行和实时同步。它现在是作为链表调度器的一个子类实现的。NS 中的实时功能仍然在开发之中，但实时功能可用作将 NS 的模拟网络导入到现实的网络拓扑中进行实验，只需要简单的配置网络拓扑和交错通信等。但这只能在相对比较慢的网络数据传输速率下实现，因为模拟器必须能够跟上现实网络中的 packet 到达速率，但这种同步在当前来说并不是必须的。

4.2.5 ns中调试器时钟的精度

调度器时钟精度可以这样定义：模拟器所能正确表现出来的最小的时间刻度。NS的时钟变量是用双精度实数来表示的。双精度实数如同IEEE标准中规定的浮点数一样，包含有必须分配给符号、指数和尾数域的64个bit。

符号 指数 尾数

1bit 11bit 52bit

任一浮点数都可以表示为（X*2n）的形式，其中X是尾数，n是指数。所以NS的时钟精度可以表示为（1/2（52））。随着模拟过程的运行，剩下的用来表示时间的bit数也在减少，由此可以推断出精度也随之降低。在给定52bit的情况下，2（40））左右的时间可以被认为是准确的。任何大于它的值都不是十分准确的，因为你已经预留了12bit用来表示时间的变化。然而（2（40））是一个十分巨大的数值，我们在NS中已经不会考虑时间精度的问题了。

4.3 其他方法

Simulator 类提供了许多用于建立模拟的方法。它们大致可以分为三类：创建和管理网络拓扑的方法（其中依次包括节点管理（第五章）和链路管理（第六章））；用作跟踪的方法（第二十六章）；与调度器相关的辅助函数。下面的清单中列举了与拓扑无关的模拟器方法：

```
Simulator instproc now ;                          #返回调度器指示的当前时间
Simulator instproc at args ;                      #在指定时刻调度执行代码
Simulator instproc cancel args ;                  #取消事件
Simulator instproc run args ;                     #启动调度器
Simulator instproc halt ;                         #终止（暂停）调度器
Simulator instproc flush-trace ;                  #消除写缓冲区内的所有跟踪对象
Simulator instproc create-trace type files src dst ;      #创建跟踪对象
Simulator instproc create_packetformat ;          #设置模拟器的packet格式
```

4.4 命令一览

摘要：

ns <OTcl 文件> <参数> <参数>..

描述：

在 NS 中执行一个模拟脚本的基本命令。

模拟器（NS）是由 NS 解释器激活的，NS 解释器是 vanilla otclsh 命令行 shell 的一种扩展。模拟过程是在 OTcl 脚本（文件）中定义的。可以在 ns/tcl/ex 目录下找到几个 OTcl 脚本的例子。

下面的清单中列出了在模拟脚本中常用的模拟器命令：

set ns_ [new Simulator]

这个命令创建了一个模拟器对象的实例。

set now [$ns_ now]

在模拟过程中，调度器对时间进行跟踪。这个命令用作返回调度器执行的当前时间。

$ns_ halt

这个命令用作暂停或终止调度器。

$ns_ run

这个命令用作启动调度器。

$ns_ at <time> <event>

这个命令用作在指定的<time>执行<event>（event通常是一段代码）。

例如

$ns_ at $opt(stop) "puts ¨NS EXITING..¨ ; $ns_ halt"或$ns_ at 10.0 "$ftp start"。

$ns_ cancel <event>

取消某个事件<event>。实际上，事件被移除出调度器的就绪执行事件链表。

$ns_ create-trace <type> <file> <src> <dst> <optional arg: op>

这个命令用作在<src>对象和<dst>对象之间创建一个<type>类的trace-object，并将这个trace-object对象与<file>文件关联起来，并将跟踪输出写入<file>。如果”op”被定义为”nam” ，这个命令将创建一个nam跟踪文件；如果”op”没有定义，则以NS的缺省方式创建跟踪文件。

$ns_ flush-trace

这个命令用作清除写缓冲区中的所有跟踪对象。

$ns_ gen-map

这个命令会删除为给定的模拟过程创建的信息：节点、节点组件、链路等。这个命令可能会破坏某些模拟场景（例如无线场景）。

$ns_ at-now(arg)

这个命令在效果上是和“$ns_ at $now $args”命令一样的。有一点需要注意：这个函数可能会因为Tcl的字符串长度的处理方法而不能正常工作。

这些是模拟器（内部的）附加辅助函数（通常用坐扩展/修改NS的核心代码）：

$ns_ use-scheduler <type>

这个命令用作为模拟过程指定某种类型的调度器。调度器的不同可用类型为：链表（list）、时间（calendar）、堆（heap）

和实时（real-time）。当前的缺省类型为时间。

$ns_ after <delay> <event>

在<delay>长度的时间延迟后调度执行<event>。

$ns_ clearMemTrace

这个命令用作内存调试。

$ns_ is-started

这个命令在模拟器启动的情况下返回值为真；否则返回值为假。

$ns_ dumpq

这个命令用作在调度器终止时将调度器事件队列中的信息清除。

$ns_ create_packetformat

这个命令用作设置模拟器中的packet格式。

第 5 章

节点和Packet转发

这一章讲述了在NS中创建网络拓扑的一个方面，例如：创建节点。在下一章（第六章）中，我们将讲述创建网络拓扑的下一个部分的内容，例如：将节点互连起来形成链路。

回忆一下前面的内容：每个模拟过程都需要创建一个独立的Simulator类的实例来控制和对本次模拟进行操作。这个类提供实例过程来创建和管理网络拓扑，并在内部存储每个网络拓扑元素的相关信息。我们从讲述Simulator类的过程（章节5.1）开始这一章的内容。然后我们将讲述Node类中实现对独立节点访问和操作的实例过程。这一章将以对分类器（Classfier）的详细讲解结束，由分类器组成了更复杂的节点对象。

本章中所讲述的过程和函数可以在以下文件中找到：

~ns/tcl/lib/ns-lib.tcl, *~ns*/tcl/lib/ns-node.tcl,*~ns*/tcl/lib/ns-rtmodule.tcl, *~ns*/rtmodule.{cc,h}, *~ns*/classifier.{cc, h}, *~ns*/classifier-addr.cc, *~ns*/classifier-mcast.cc, *~ns*/classifiermpath.cc,*~ns*/replicator.cc.

5.1 节点基础知识

创建节点的基本命令是：

```
set ns [new Simulator]
$ns node
```

这个实例过程node创建了一个包含多个简单分类器对象的节点（章节5.4）。Node本身也是OTcl中的一个独立的类。然而构成节点的大部分组件本身也是TclObject。一个节点（单播）的典型结构如图5.1所示。这种简单的结构包含两个TclObject：一个地址分类器（classifer_）和一个端口分类器（dmux_）。这些分类器的功能就是将接收到的包分发到正确的代理或者输出链路。

所有的节点都至少包含了以下的组件：

- 一个地址或者id_，模拟名字空间随着节点的创建而单调逐次加1（初始值为0）。
- 一个邻居节点链表（neighbor_）。

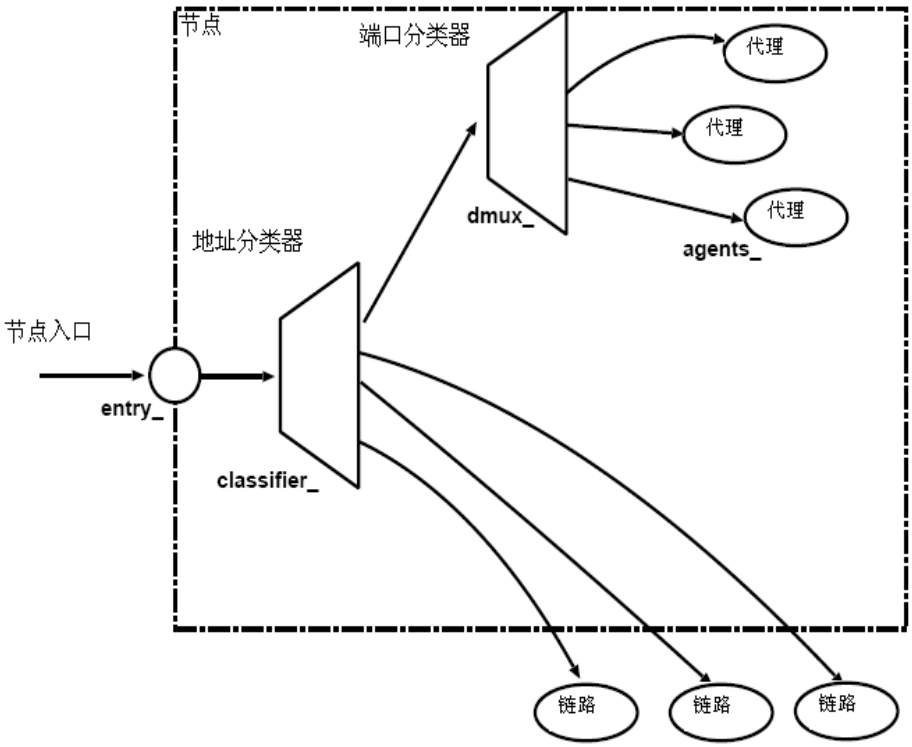


图5.1 单播节点的结构。

注意：entry_只是一个简单的标签变量而并非一个像classier_一样的真实对象。

- 一个代理链表（agent_）。
- 一个节点类型识别器（nodetype）。
- 一个路由模块（在下面的5.5章节将进行讲解）。

缺省情况下，在NS中创建的节点是单播模拟的。要想启用组播模拟，在模拟创建的时候要使用”-multicast on”选项。例如：set ns [new Simulator -multicast on]

一个典型的组播模拟节点的内部结构如图5.2所示。

当一次模拟过程使用组播路由时，地址的最高bit位就用来指示给定的地址是组播地址还是单播地址。如果最高的bit位为0，则此地址代表了一个单播地址；否则这个地址就是组播地址。

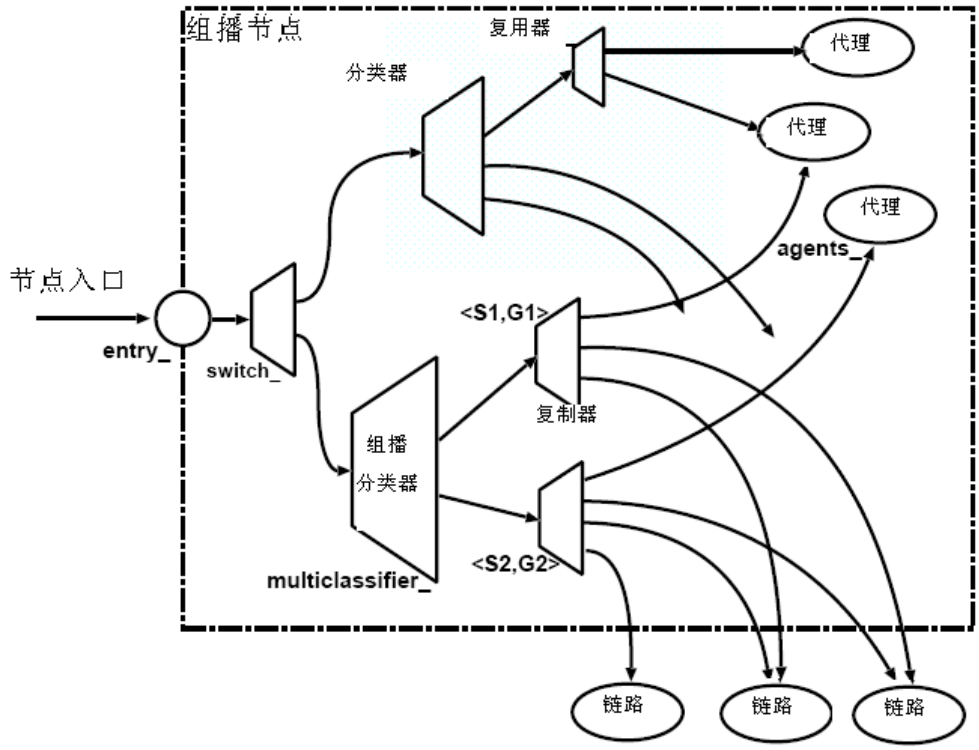


图5.2组播节点的内部结构

5.2 节点方法：配置节点

配置独立节点的方法可以分为以下几类：

——控制函数

——地址和端口号管理，单播路由函数

——代理管理

——添加邻居

我们将在下面的篇幅中对每一个函数进行讲述。

控制函数

1. $node entry 返回节点的入口指针。这是第一个处理packet到达节点的组件。entry_是一个Node实例变量，它用来存储这个元素的指针。对于单播节点而言，这个变量是用作检查目的地址的高位字段的地址分类器。classifier_实例变量包含了这个地址分类器的指针。而对于组播节点而言，switch_才是入口指针，它通过查看第一位信息来决定这个packet应该被转发到单播分类器还是应该被转发到组播分类器。
2. $node reset命令会把该节点的所有代理复位。

地址和端口号管理

$node id过程返回一个节点的节点号。这个节点号是Simulator类用$node reset方法在创建节点时自动递增并分配给节点的。Simulator类同时也存储了一个实例变量数组[10]—Node_，这个数组以节点ID为下标，其中包含了一个指向与ID相应的节点的指针。

$node agent <port>这个过程返回的是在给定端口的代理的句柄。如果在给定的端口号上没有可用的代理，这个过程就返回一个空串。

alloc-port过程返回下一个可用的端口号。它使用了一个过程变量np_来跟踪下一个没有被分配的端口号。

add-route和add-routes过程被用作单播路由（第26章）中添加路由以生成classifier_。使用语法为：$node add-route <destination id> <TclObject>。TclObject是dmux_的入口，是所在节点的端口多路复用器，如果destination id和这个节点的ID一致，那么它通常是一个发送到那个节点的packet的链路的头部，但也有可能是其他分类器或者不同种类的分类器的入口。

$node add-routes <destination id> <TclObjects>通常用作为对同一个目的地添加多重路由，所有到达这个目的地的链路必须同时以轮换的方式平均分配带宽。这个语句只用在实例变量multiPath_设置为1的时候，此时复杂动态路由策略也要启用，而且使用一个多路分类器。我们将在本章的后续部分（章节5.4）对多路分类器的实现进行讲解，但是我们将把多路路由的讲述（第29章）推迟到单播路由的章节之后。

与add-routes{}相对应的是delete-routes{}。它有id，一个TclObjects的链表和一个指向模拟器空代理的指针。在多路分类器它将链表中TclObjects从安装的路由路径中移除。如果分类器中的路由入口并没有指向一个多路分类器，则按照惯例简单的将入口从classifier_中移除，并安装空代理作为代替。

复杂动态路由也使用两种附加的方法：init-routing{}这个实例过程设置实例变量multiPath_，使它的值与和它同名的类变量相等。它同样附加了一个指向在同一个节点上的路由控制器对象（实例变量rtObject_）的指针。rtObject?{}过程返回了在节点中路由对象的句柄。

最后，当节点关联的链路事件发生变动时，intf-changed{}过程将被网络动态代码激活。关于如何使用这个过程的详细情况将在后续的动态网络的章节（第31章）中进行讨论。

代理管理

attach{}过程将把给定的一个代理<agent>添加到它的agents_链表中，为之分配一个端口号，并设置好它的源地址，将指向代理的指针设置为它的（例如，这个节点）entry{}，同时在节点上附加一个端口复用器的指针(dmux_)，它指向代理相应的dmux_分类器的插槽上。

[10]例如，一个类的实例变量同样也是一个数组变量。

相反的，detach{}则是将代理从agents_上移除，并将代理的目标和节点的入口dmux_指向空代理。

跟踪邻居

每个节点都维护了一个在它的实例变量中的与之临近的邻居节点的链表—neighbor_。add-neighbor{}这个过程将一个邻居节点添加到链表中。neighbors{}这个过程返回的就是这个链表。

5.3 节点配置接口

注意：这个 API，尤其是它的内部实现至今仍是比较混乱的，它仍然在变化中。它或许在不久的将会发生影响深远的变化。然而，我们仍将力来维持其接口与当前章节所讲述的接口的一致。另外，这个 API 当前并没有涵盖所有的已在旧格式中存在的所有节点，名义上讲，节点的建立是采用了继承，还有一部分关于移动 IP 的。它主要是面向无线和卫星模拟的。[2000 年 9 月 15 日，在 2001 年 6 月更新]

Simulator::node-config{}采用可变的模块化结构，以此实现不同类型的节点在相同的Node基类下的定义。例如：要创建一个具备无线通信能力的移动节点，你不必再使用给定的节点创建命令，例如：dsdv-create-mobile-node{}；而是改变缺省的配置参数，例如：$ns node-config –adhocRouting dsdv，把它放在实际创建节点的命令：$ns node之前就可以了。和路由模块一起，这个命令允许我们将“任意”路由函数合并到一个单一节点上，而不必求助于多重继承和其他高级对象的技巧。我们将在章节5.5中对此进行详细讲述。和新的节点API相关的函数和过程可以在目录~ns/tcl/lib/ns-node.tcl中找到。

节点配置接口包含两方面的内容。第一部分负责处理节点的配置，而第二部分则根据给定的类型实际创建节点。我们已经在章节5.1中讲过了后者，在这部分中，我们将讲述配置部分。

节点配置本质上包括在节点创建前定义不同的节点特征。它们可能包括在模拟中要使用到的地址结构类型，定义移动节点的网络组件，在Agent/Router/MAC层开启或者关闭跟踪选项，为无线节点选择adhoc路由协议的类型或者定义它们的能量模型。

例如，一个在层次拓扑结构中运行AODV作为其adhoc路由协议的无线移动节点的配置如下所示。我们假定只在代理层和路由层开启跟踪。同时我们假定已经用命令：set topo [new Topography]实例化了一个拓扑结构。节点配置命令如下所示：

```
$ns_ node-config   -addressType hierarchical \
                  -adhocRouting AODV \
                   -llType LL \
                   -macType Mac/802_11 \
                   -ifqType Queue/DropTail/PriQueue \
                   -ifqLen 50 \
                   -antType Antenna/OmniAntenna \
                   -propType Propagation/TwoRayGround \
```

```
-phyType Phy/WirelessPhy \
-topologyInstance $topo \
-channel Channel/WirelessChannel \
-agentTrace ON \
-routerTrace ON \
-macTrace OFF \
-movementTrace OFF
```

除了-addressingType的缺省值为flat之外，上面所给出所有选项的缺省值均为NULL。-rest选项可以将所有的节点配置参数复位到缺省值。

有一点需要注意：配置命令可以被分成如下所示的独立两行：

```
$ns_ node-config -addressingType hier
$ns_ node-config -macTrace ON
```

那些需要改变的选项将只是被调用。例如，如上所示，在配置完AODV移动节点之后（并且在创建AODV移动节点之后），我们可以用下面的方式来配置AODV基站节点：

```
$ns_ node-config -wiredRouting ON
```

因为除此之外，所有基站节点的特征都和移动节点一样，所以基站节点同样也适用于有线路由，但移动节点就不行。用这种方法我们可以只在需要的时候更改节点配置。

所有在给定的节点配置命令之后创建的节点实例都将拥有相同的性质，除非一部分或者所有的节点配置命令都用不同的参数值执行。并且所有参数值将保持不变，除非它被显式的改变了。所以在AODV基站和移动节点创建后，如果我们想创建简单的节点，我们将用到以下的节点配置命令：

```
$ns_ node-config –reset
```

这个命令将把所有的参数值设置为它们的缺省值，这将基础的定义一个简单节点的配置。

当前，这种类型的节点配置是面向无线和卫星节点的。表5.1给出了这些类型的节点的可用的选项参数。实例脚本~ns/tcl/ex/simple-wireless.tcl和~ns/tcl/ex/sat-mixed.tcl给出了其使用例子。

5.4 分类器

当一个节点接收到一个 packet 时，它的功能就是去检查 packet 的域，通常是它的目的地址，有时也检查源地址。然后就是将值标示给一个出口接口对象，也就是该 packet 的下游接收者。

在 NS 中，这个任务是由一个简单的 classifier 对象完成的。多路分类器对象，每一个都负责 packet 的一个特定部分，然后再通过这个节点转发 packet。在 NS 中，一个节点出于不同的目的会使用多种类型的分类器。这一部分将讲解 NS 中比较常用，比较简单的分类器对象。

我们以对基类的讲解作为这一章节的开始。下一个子章节是地址分类器（章节 5.4.1），组播分类器（章节 5.4.2），多路径分类器（章节 5.4.3），哈希分类器（章节 5.4.4）最后讲解的是复制器（章节 5.4.5）。

分类器提供一种将 packet 与逻辑准则进行匹配的方法，并返回一个基于匹配结果的指向另一个模拟对象的指针。每一个分类器包含一个以 slot number 为索引的模拟对象表。分类器的工作就是决定某个 slot number 所关联的接收到的 packet，并将这个 packet 转发给由给定的 slot 所指向的对象。C++类 Classifier（在~ns/classifier.h 中定义）提供了一个衍生出其他分类器的基类。

选项	可用值	缺省
普通		
addressType	flat, hierarchical	flat
MPLS	ON, OFF	OFF
面向卫星通信和无线通信		
wiredRouting	ON, OFF	OFF
llType	LL, LL/Sat	""
macType	Mac/802_11, Mac/Csma/Ca, Mac/Sat, Mac/Sat/UnslottedAloha, Mac/Tdma	""
ifqType	Queue/DropTail, Queue/DropTail/PriQueue	""
phyType	Phy/WirelessPhy, Phy/Sat	""
面向无线通信		
adhocRouting	DIFFUSION/RATE, DIFFUSION/PROB, DSDV, DSR, FLOODING, OMNIMCAST, AODV, TORA	""
propType	Propagation/TwoRayGround, Propagation/Shadowing	""
propInstance	Propagation/TwoRayGround, Propagation/Shadowing	""
antType	Antenna/OmniAntenna	""
channel	Channel/WirelessChannel, Channel/Sat	""
topoInstance	<topology file>	""
mobileIP	ON, OFF	OFF
energyModel	EnergyModel	""
initialEnergy	<value in Joules>	""
rxPower	<value in W>	""
txPower	<value in W>	""
idlePower	<value in W>	""
agentTrace	ON, OFF	OFF
routerTrace	ON, OFF	OFF
macTrace	ON, OFF	OFF
movementTrace	ON, OFF	OFF
errProc	UniformErrorProc	""
FECProc	?	?
toraDebug	ON, OFF	OFF
面向卫星通信		
satNodeType	polar, geo, terminal, geo-repeater	""
downlinkBW	<bandwidth value, e.g. "2Mb">	""

表 5.1：节点配置中可用的选项变量（参考 tcl/lib/ns-lib.tcl 文件）

```
class Classifier : public NsObject {
public:
~Classifier();
void recv(Packet*, Handler* h = 0);
protected:
Classifier();
void install(int slot, NsObject*);
void clear(int slot);
virtual int command(int argc, const char*const* argv);
virtual int classify(Packet *const) = 0;
void alloc(int);
NsObject** slot_; /* 将slot number标记到一个NsObject的表 */
int nslot_;
int maxslot_;
};
```

方法 classify()是一个单纯的虚函数，用来表示 Classifier 类只是被用做基类。方法 alloc()动态的在表中分配足够大的空间以维持特殊的 slot number。方法 install()和 clear()用作对表中的对象进行添加或者删除。方法 recv()和 OTcl 接口是以如下所示方法在~ns/classifier.cc 文件中实现的：

```
/*
* 对象只对从入口链路或者本地代理进来的packet事件可见(例如，packet源头)。
*/
void Classifier::recv(Packet* p, Handler*)
{
NsObject* node;
int cl = classify(p);
if (cl < 0 || cl >= nslot_ || (node = slot_[cl]) == 0) {
Tcl::instance().evalf("%s no-slot %d", name(), cl);
Packet::free(p);
return;
}
node->recv(p);
}
int Classifier::command(int argc, const char*const* argv)
{
Tcl& tcl = Tcl::instance();
if (argc == 3) {
/*
* $classifier clear $slot
*/
if (strcmp(argv[1], "clear") == 0) {
int slot = atoi(argv[2]);
clear(slot);
return (TCL_OK);
```

```
}
/*
* $classifier installNext $node
*/
if (strcmp(argv[1], "installNext") == 0) {
int slot = maxslot_ + 1;
NsObject* node = (NsObject*)TclObject::lookup(argv[2]);
install(slot, node);
tcl.resultf("%u", slot);
return TCL_OK;
}
if (strcmp(argv[1], "slot") == 0) {
int slot = atoi(argv[2]);
if ((slot >= 0) || (slot < nslot_)) {
tcl.resultf("%s", slot_[slot]->name());
return TCL_OK;
}
tcl.resultf("Classifier: no object at slot %d", slot);
return (TCL_ERROR);
}
} else if (argc == 4) {
/*
* $classifier install $slot $node
*/
if (strcmp(argv[1], "install") == 0) {
int slot = atoi(argv[2]);
NsObject* node = (NsObject*)TclObject::lookup(argv[3]);
install(slot, node);
return (TCL_OK);
}
}
return (NsObject::command(argc, argv));
}
```

当一个分类器recv()一个packet的时候，它将packet提交给classify()方法。由不同的基类衍生出来的每个种类的分类器的定义都是不同的。classify()方法常用的格式是决定并返回一个slot表中的索引。如果索引是有效的，并且指向了一个有效的TclObject，分类器将把packet提交给使用recv()方法的对象。如果索引无效，分类器将激活实例过程no-slot{}来尝试正确的生成表。然而在基类Classifier::no-slot{}中，打印输出错误消息并终止执行。

command()方法为解释器提供以下的类成员函数：

- clear{<slot>}清空特定slot的入口。
- installNext{<object>}在下一个可用的slot安装对象，并返回slot号。

注意一点：这个类成员函数是被一个存储了指向对象存储的指针的同名实例过程重载的。这将有助于从OTcl中的分类器中快速查询安装的对象。

- slot{<index>}返回在指定的slot中存储的对象。
- install{<index>,<object>}在<index>slot中安装指定的<object>。

注意：这个类成员函数也是被一个存储了指向对象存储的指针的同名实例过程重载的。这也是为了可以在OTcl中的分类器中快速查询安装的对象。

5.4.1 地址分类器

地址分类器是用于支持单播packet的转发。它对packet的目的地址应用一个逐位位移和掩码操作来生成一个slot号。这个slot号是由classify()方法返回的。 AddressClassifier类（在~ns/classifier-addr.cc文件中定义）的定义如下所示：

```
class AddressClassifier : public Classifier {
public:
AddressClassifier() : mask_(~0), shift_(0) {
bind("mask_", (int*)&mask_);
bind("shift_", &shift_);
}
protected:
int classify(Packet *const p) {
IPHeader *h = IPHeader::access(p->bits());
return ((h->dst() >> shift_) & mask_);
}
nsaddr_t mask_;
int shift_;
};
```

这个类并没有在packet目的地址域强加直接的语意。它只是从packet的dst_域返回某些比特位，作为slot号以供Classifier::recv()方法使用。mask_和shift_的值是通过OTcl设置的。

5.4.2 组播分类器

组播分类器根据源地址和目的（组）地址将packet进行分类。它维持了一个（链式哈希）表，这个表将源/组对标记到slot号。当一个packet到达时，它包含的源/组对分类器是未知的时候，它将激活一个OTcl过程Node::new-group{}来为它的表添加一个入口。这个OTcl过程可能使用到set-hash方法来向分类器表中添加新的（源、组、slot）的三元数组。这个组播分类器在~ns/classifier-mcast.cc文件中有如下所示定义：

```
static class MCastClassifierClass : public TclClass {
public:
```

```
MCastClassifierClass() : TclClass("Classifier/Multicast") {}
TclObject* create(int argc, const char*const* argv) {
return (new MCastClassifier());
}
} class_mcast_classifier;
class MCastClassifier : public Classifier {
public:
MCastClassifier();
~MCastClassifier();
protected:
int command(int argc, const char*const* argv);
int classify(Packet *const p);
int findslot();
void set_hash(nsaddr_t src, nsaddr_t dst, int slot);
int hash(nsaddr_t src, nsaddr_t dst) const {
u_int32_t s = src ^ dst;
s ^= s >> 16;
s ^= s >> 8;
return (s & 0xff);
}
struct hashnode {
int slot;
nsaddr_t src;
nsaddr_t dst;
hashnode* next;
};
hashnode* ht_[256];
const hashnode* lookup(nsaddr_t src, nsaddr_t dst) const;
};
int MCastClassifier::classify(Packet *const pkt)
{
IPHeader *h = IPHeader::access(pkt->bits());
nsaddr_t src = h->src() >> 8; /*XXX*/
nsaddr_t dst = h->dst();
const hashnode* p = lookup(src, dst);
if (p == 0) {
/*
* 未发现入口
* 调用一次tcl来安装一个
* 如果tcl没有通过则失败
*/
Tcl::instance().evalf("%s new-group %u %u", name(), src, dst);
p = lookup(src, dst);
if (p == 0)
```

```
return (-1);
}
return (p->slot);
}
```

MCastClassifier类实现了一个链式哈希表，并在packet的源地址和目的地址应用了一个哈希函数。哈希函数返回slot号，以作为隐式对象slot_表的索引。哈希中空缺的位置表示packet是被发送到一个之前未知的组中；为了处理这种情况，可以调用OTcl。OTcl的代码可以在哈希表恰当的位置插入一个入口。

5.4.3 多路分类器

这个对象是为了支持以相同的代价来进行多路转发而设计的，多路转发就是某个节点有多个相同代价的路径来到达同一个目的地，并且希望可以同时全部使用它们。这个对象并不关注packet的任何一个域。对每个连续到达的packet，它只是简单的以轮换的方式返回下一个slot。这个分类器的定义在~ns/classifier-mpath.cc文件中，如下所示：

```
class MultiPathForwarder : public Classifier {
public:
MultiPathForwarder() : ns_(0), Classifier() {}
virtual int classify(Packet* const) {
int cl;
int fail = ns_;
do {
cl = ns_++;
ns_ %= (maxslot_ + 1);
} while (slot_[cl] == 0 && ns_ != fail);
return cl;
}
private:
int ns_; /*下一个要使用的slot。可能是一个误称？*/
};
```

5.4.4 哈希分类器

这个对象被用作将packet以特殊“流”的成员进行分类的。就像其名字所指的一样，哈希分类器在其内部使用了一个哈希表来将packet指派到流。这些对象是在需要水平流信息的地方使用的（例如，在指定流队列规则和统计收集中）。有几个“流粒度”是可用的。在某些特殊情况下，packet可以根据流ID、目的地址、源/目的地址或者源/目的地址和流ID的组合的不同而被分配到不同的流中。哈希分类器所能访问的域仅限于packet的ip头部：src(),dst(),flowid()（参见ip.h文件）。

哈希分类器和一个指定哈希表初始大小的整数参量一起创建。当前哈希表的大小可以用resize方法逐渐修改（见下）。当创建表的时候，实例变量shift_和mask_被模拟器当前的NodeShift和NodeMask值独立初始化。这些值在哈希分类器实例

化的时候被从AddrParams对象收回。如果AddrParams结构体没有初始化将导致哈希分类器正常操作的失败。下面的构造体在不同的哈希分类器汇总得到使用：

Classifier/Hash/SrcDest
Classifier/Hash/Dest
Classifier/Hash/Fid
Classifier/Hash/SrcDestFid

哈希分类器接收到packet，根据它们的流标准分类，并取回用作指示下一个接收该packet的节点的分类器slot。在使用哈希分类器的某些情况下，大部分packet应该只被分配一个slot，并且有一些流应该被指向别处。哈希分类器包含一个default_实例变量，用于那些不满足之前的流标准的packet。default_可以被有选择的设置。

哈希分类器的方法如下：

$hashcl set-hash buck src dst fid slot
$hashcl lookup buck src dst fid
$hashcl del-hash src dst fid
$hashcl resize nbuck

set-hash()方法在哈希分类器的范围内向哈希表插入了一个新的入口。buck参数用来指定插入这个入口所用的哈希表的桶号。当这个桶号是未知的时候，buck将被指为auto。src、dst和fid参数指明了用来匹配流分类的IP源、目的和流ID。没有被特定的分类器使用的域（例如，一个流ID分类器中给定的src）将被忽略。slot参数指示的是衍生出哈希分类器的Classifier基类对象中隐式slot表的索引。lookup函数返回的是与给定的buck/src/dst/fid数组相关联的对象的名字。buck参数可以像set-hash一样被设置为auto。del-hash函数从哈希表中移除指定的入口。当前，这种操作只是由简单的将给定的入口标记为非活动状态来实现的，所以可能通过未使用的入口来扩展哈希表。resize函数通过nbuck参数来重新设置哈希表中桶的数量以词改变哈希表的大小。

因为在定义时没有提供缺省设置，一个哈希分类器在接收到一个不能匹配任何流标准的packet时会调用OTcl。调用格式如下：

$obj unknown-flow src dst flowid buck

因此，当一个接收到的packet不匹配任何流标准时，实例化的哈希分类器对象中的unknown-flow方法将被packet的源、目的和流ID域激活。另外，如果哈希表使用了set-hash，则buck域就指明了包含这个流的哈希桶。这种安排防止了在执行插入到分类器操作时，且桶已知的情况下，其他哈希表的查询。

5.4.5 复制器

复制器和其他我们之前已经讲述过的分类器不同，它内部没有使用分类函数。而是简单的把分类器作为一个含有 n 个 slot 的表；它重载了 recv()方法来产生 n 份 packet 的拷贝，这样就可以把它们分发给所有的与这个表关联的 n 个对象。

为了支持 packet 的组播转发，一个分类器接从源 S 收到一个目的为组 G 的组播 packet，这个分类器计算一个哈希函数

h（S，G），从而在分类器对象表中给出一个 slot 号。在组播的传输中，packet 必须被一次一次的复制，因为每一个通向 G 子集节点的链路都会把 packet 减去一份。生产额外的 packet 拷贝是由一个 Replicator 类完成的，它在 replicator.cc 中是这样定义的：

```
/*
* 复制器并非一个真正的packet分类器，
* 但是我们只是想方便的影响其slot表。
* （这个对象用来实现输出组播路由和LAN广播）
*/
class Replicator : public Classifier {
public:
Replicator();
void recv(Packet*, Handler* h = 0);
virtual int classify(Packet* const) {};
protected:
int ignore_;
};
void Replicator::recv(Packet* p, Handler*)
{
IPHeader *iph = IPHeader::access(p->bits());
if (maxslot_ < 0) {
if (!ignore_)
Tcl::instance().evalf("%s drop %u %u", name(),
iph->src(), iph->dst());
Packet::free(p);
return;
}
for (int i = 0; i < maxslot_; ++i) {
NsObject* o = slot_[i];
if (o != 0)
o->recv(p->copy());
}
/* 我们知道maxslot是非空的 */
slot_[maxslot_]->recv(p);
}
```

我们从代码中可以看到，这个类并不是真正的对 packet 进行分类。它只是复制一个 packet，为表中的每一个入口复制一份，并将拷贝分发给表中列出的每一个节点。表中的最后一个入口将得到“最初”的 packet。因为 classify()方法在基类中是一个单纯的虚函数，所以复制器定义了一个空的 classify()方法。

5.5 路由模块和分类器组织

我们可以看到，一个 NS 节点本质上就是一个分类器的集合。最简单的节点（单播）只包含一个地址分类器和一个端口分类器，如图 5.1 所示。当你扩展节点的功能时，更多的分类器被添加到基础节点中。例如，图 5.2 所示的组播节点。由于更多的功能模块被添加进来，且每个模块都需要它们自己的分类器，节点为组织这些分类器而提供一个统一的接口并把这些分类器桥接到路由计算模块就显的重要起来。

处理这个问题的经典方法是通过类的继承。例如，如果你想要一个节点支持层次路由，可以只从基础节点派生出一个 Node/Hier，然后重写分类器的设置方法来插入层次分类器。当这些新的功能模块是独立的并且不能被“随意”混合的时候，这种方法可以有效工作。例如，层次路由和 ad hoc 路由使用它们各自的分类器集。继承要求我们有 Node/Hier 来支持前者，而要求 Node/Mobile 来支持后者。当我们想要一个 ad hoc 路由节点支持层次路由的时候，就有些问题了。在这个简单的例子中，你可以用多重继承的方法来解决这个问题，但当这种功能模块的数量增加的时候，这种方法就不可行了。

解决这个问题唯一的方法就是对象合成。基础节点需要为分类器的访问和组织定义一套接口。这些接口应该

- 允许实现了它们自己的分类器的独立路由模块将它们的分类器插入到节点中
- 允许路由计算模块在所有需要这个信息的路由模块中建立到分类器的路由
- 为已经存在的路由模块提供一个独立的管理点

另外，我们应该为路由模块定义一个统一的接口以连接到节点接口，也就是为扩展节点功能提供一个系统的途径。在这一部分中，我们将讲述路由模块的设计和相应的节点接口。

5.5.1 路由模块

一般来说，在 NS 中每个路由的实现都包含三个功能模块：

- 路由代理和邻居进行路由 packet 的交换
- 路由逻辑用路由代理（或者是在静态路由下的全局拓扑数据库）收集到的信息进行实际的路由计算
- 分类器位于节点的内部。它们使用计算过的路由表来进行 packet 的转发

注意：当实现一个新的路由协议的时候，不必必须实现所有这三个模块。例如，当实现一个链路状态路由协议时，只是简单的实现以链路状态方式交换信息的路由代理和一个使用 Dijkstra 算法计算拓扑数据库的路由逻辑。它可以和其他单播路由协议一样使用相同的分类器。

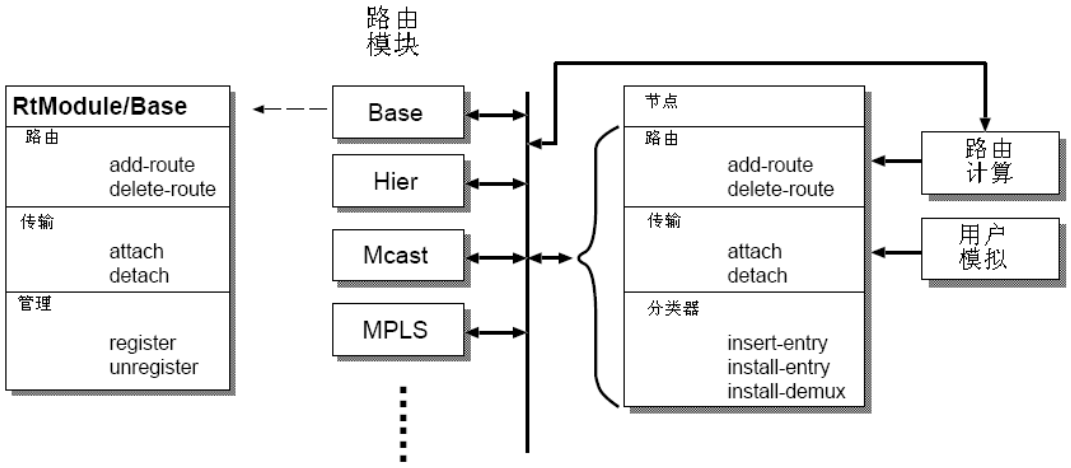


图 5.3 节点、路由模块和路由之间的关系 虚线表示路由模块的细节

当一个新的路由协议的实现包含了超过一个功能模块的时候，尤其是当它包含它自身的分类器的时候，它就需要建立另一个对象，我们称之为路由模块。它管理所有这些功能模块并与节点接口联系以管理分类器。图 5.3 给出了这些对象间的功能联系。注意一点，路由模块可能和路由计算模块有直接联系，例如，路由逻辑和（或）路由代理。然而，路由计算可能不直接通过路由模块安装它们的路由，因为可能存在其他的模块对学习新的路由感兴趣。这不是必须条件，因为有可能某些路由计算是为某个特定的路由模块指定的，例如，在 MPLS 模块中标签的安装。

一个路由模块包含三个主要功能：

1. 一个路由模块通过register{}来初始化它到节点的连接，通过unregister{}来断开连接。通常情况下，在register{}中一个路由模块（1）告知节点它是否对得知路由更新和传输代理的连接感兴趣；（2）创建它的分类器并将其安装到节点中（详细讲解见下一子章节）。在unregister{}中，一个路由模块做的工作则恰好相反：它移除掉其分类器，并将其与节点的路由更新相关联的钩子函数摘除。
2. 如果一个路由模块对得知路由更新感兴趣，节点将通过RtModule::add-route{dst,target}和RtModule::delete-route{dst,nullagent}来告知这个模块。
3. 如果一个路由模块对得知节点上传输代理的连接和解除连接的情况感兴趣，节点将通过RtModule::attach{agent,port}和RtModule::detach{agent,nullagent}来告知这个模块。

要编写你自己的路由模块有两个步骤：

1. 你需要声明你的路由模块中的C++部分（参见~ns/rtmodule.{cc,h}文件）。对很多模块来说，这只是意味着要声明一个虚方法name()，它返回模块的描述符的串。然而你可以自由的用C++实现你希望的功能，如果需要，你可以在以后将OTcl的功能移植到C++中以得到更好的表现。
2. 你需要查阅上面给出的在基础路由模块中的接口实现（参见~ns/tcl/lib/ns-rtmodule.tcl文件），以此来决定你要继承

哪一个，重载哪一个，并将其放在你自己的模块中的OTcl接口。

在~ns/tcl/lib/ns-rtmodule.tcl文件中有几个派生出来的路由模块实例，它们可以作为你的模块的模板。

当前，在NS中有六种路由模块实现：

模块名	功能
RtModule/Base	单播路由协议接口。提供添加/删除路由和连接/解除连接代理的基本功能。
RtModule/Mcast	组播路由协议接口。它存在的意义在于建立组播分类器。所有其他的组播功能都是由Node的成员函数实现的。这在将来会有所改变。
RtModule/Hier	层次路由。它为管理层次分类器和路由安装服务。可以和其他的路由协议合并，例如：ad hoc路由。
RtModule/Manual	手动路由。
RtModule/VC	使用虚分类器，而不是vanilla分类器。
RtModule/MPLS	实现MPLS功能。这是存在的唯一一个自主模块，它不会增加Node名字空间。

表5.2 可用路由模块

5.5.2 路由模块

为了连接到以上的路由模块接口，一个节点要提供一套类似的接口集：

- 为了知道哪个模块在创建的时候进行了注册，Node类维护了一个模块的链表作为一个类变量。这个链表的缺省值仅包含了基础路由模块。Node类提供以下两个过程来控制这个模块链表：

Node::enable-module{[name]} 如果模块RtModule/[name]存在，这个过程将把[name]加入到模块链表。

Node::disable-module{[name]} 如果[name]在模块链表中，就把它从链表中移除。

当创建一个节点的时候，它会查看Node类的模块链表，创建包含在链表中的所有模块，并将这些模块在节点进行注册。

在节点创建完毕后，你可能用到以下的成员函数来列出在节点注册的模块，或者用一个指定的名字或许模块的句柄：

Node::list-modules{} 返回一个所有注册过的模块的句柄（影子对象）链表。

Node::get-module{[name]} 返回一个名字与给定名字匹配的注册过的模块的句柄。注意：任何路由模块在任一节点只能注册一个实例。

- 为了允许路由模块注册其对路由更新感兴趣，一个节点对象提供以下的成员函数：

Node::route-notify{module} 将module添加到路由更新通知链表中。

Node::unreg-route-notify{module} 将module从路由更新通知链表中移除。

类似的，下面的成员函数提供关于传输代理连接状况的钩子功能：

Node::port-notify{module} 将module添加到代理连接状况通知链表中。

Node::unreg-port-notify{module} 将module从代理连接状况通知链表中移除。注意：在所有这些成员函数中，module应该是一个模块句柄而不是一个模块名。

- 节点提供以下的成员函数来控制其地址分类器和端口分类器：

--Node::insert-entry{module,clsfr,hook}将clsfr分类器插入到节点的入口点。它也新的分类器与模块进行关联，这样的话，如果这个分类器在稍后被移除掉，module将撤消注册。如果hook是以数字方式给出的，已存在的分类器将被插入新分类器的钩子slot。用这种方法，你就可以建立一个分类器链，如图5.2所给出的例子。注意：clsfr不必是一个分类器。在某些情况下你可能想要把一个代理或者任意从Connector派生出来的类放到节点的入口处。在这种情况下，你只需要简单的为参数hook提供target即可。

--Node::install-entry{module,clsfr,hook}与Node::insert-entry不同，它在节点入口处删除已经存在的分类器，撤消与路由模块关联的任何注册，并在入口点安装新的分类器。如果给定了钩子hook，并且原来的分类器被连接到一个分类器链上，它将把分类器链连接到新分类器的slot钩子上，就像上面讲的那样，如果钩子和目标一直，clsfr将被当作一个从Connector派生出来的对象，而不是一个分类器。

--Node::install-demux{demux,port}将给定的分类器demux作为缺省的多路复用器。如果port是给定的，它将已经存在的多路复用器插入到新的分类器的以port为标志的slot中。注意：无论是在哪一种情况下，它都不会删除掉已经存在的多路复用器。

5.6 命令一览

下面是在模拟脚本中用到的一般节点命令清单：

$ns_ node[<hier_addr>]

这个命令用于创建并返回一个节点实例。如果<hier_addr>是给定的，节点的地址将被分配为<hier_addr>。注意：后者只有在层次地址被通过set-address-format hierarchical{}或者node-config-addressType hierarchical{}启用的时候才能使用。

$ns_ node-config -<config-parameter> <optional-val>

这个命令用于配置节点。不同的配置参数有：地址类型，不同类型的网络堆栈组件，跟踪是否将被开启，移动IP标志位是否被启用，有没有用到能量模型等。选项-reset可被用来将节点配置设置为缺省状态。缺省的节点配置，例如，如果没有给定值，将创建一个使用平面地址/路由简单的节点（基础Node类）。详细的语法参见章节5.3。

$node id

返回节点的ID号。

$node node-addr

返回节点的地址。在平面地址的情况下，节点地址和它的节点ID是一致的。在层次地址的情况下，节点地址是以串的格式返回的（即”1.4.3”）。

$node reset

将所有连接到节点的代理复位。

$node agent <port_num>

返回位于指定端口的代理的句柄。如果在给定端口没有发现代理，返回一个空串。

$node entry

返回节点的入口点。这是在节点中第一个处理接收到的packet的对象。

$node attach <agent> <optional:port_num>

将<agent>连接到这个节点。如果没有传递一个特定的端口号，节点将为这个代理分配一个端口号，并将其与这个端口绑定。另外，一旦这个代理连接到节点了，它就接收目的为这个主机（节点）和端口的packet。

$node detach <agent> <null_agent>

这个命令与上一个刚好相反。它将与节点相连的代理移除并在这个代理所在的端口安装一个空代理。这样就可以处理那些传输目的为被移除的代理的packet了。这些失去目的代理的packet就可以被空代理回收处理了。

$node neighbors

这个命令返回节点的邻居节点的链表。

$node add-neighbor <neighbor_node>

这是一个向由节点维持的邻居节点链表中添加<neighbor_node>的命令。

下面的是一个节点内部方法的清单：

$node add-route <destination_id> <target>

这是在单播路由中用来生成分类器的。目标是一个Tcl对象，如果<destination_id>和这个节点ID相同，那它就是dmux_（节点中的端口多路复用器）的入口。否则，它通常会是到达某个目的地的链路的头部。它也可能是其他分类器的入口。

$node alloc-port <null_agent>

这个命令返回下一个可用的端口号。

$node incr-rtgtable-size

实例变量rtsize_用来保持每个节点的路由表的大小。这个命令用于当有路由入口被加入到分类器中的时候，增加路由表的大小。

还有其他支持层次路由、详细的动态路由、等代价多路径路由、手动路由和移动节点的能量模块的节点命令。这些方法和其他的一些方法及其早期的讲解可以在~ns/tcl/lib/ns-node.tcl和~ns/tcl/lib/ns-mobilenode.tcl文件中找到。

第 6 章

链路：简单链路

这是定义拓扑的第二个方面。在前一章中（第五章），我们已经讲述了在NS中如何在拓扑中创建节点。现在我们讲解一下如何创建链路以将节点连接起来完成拓扑。在这一章中，我们将讲解范围限制在简单的点对点链路。NS支持多种其他的介质，包括使用简单链路网格的多路访问LAN模拟，以及其他的有线和广播介质的仿真模拟。它们将在独立的章节中进行讲述。CBQ链路是从简单链路派生出来的，也是一个相当复杂的链路结构，所以在本章中也不作讨论。

我们将以讲解创建链路的命令作为本章的开始。和由分类器组成节点一样，一个简单的链路也是由一系列的连接器建成的。我们简要的讲述一下在一个简单链路中的某些连接器。然后我们将讲解被这些连接器定义的用于操作不同组件的实例过程（章节6.1）。我们将以对连接器对象的讲解作为本章的结束（章节6.2），其中包括对一般连接器的简要讲解。

Link类在OTcl中是一个独立的类，它提供一些简单的原语。SimpleLink类提供了用一个点对点链路将两个节点连接到一起的能力。NS提供实例过程simplex-link{}来构造从一个节点到另一个节点的单工链路。这个链路在SimpleLink类中。下面给出了单工链路的语法：

```
set ns [new Simulator]
$ns simplex –link <node0> <node1> <bandwidth> <delay> <queue_type>
```

这个命令创建了一个从<node0>到<node1>的链路，并指定了带宽<bandwidth>和时延<delay>特征值。链路应用了<queue_type>队列类型。这个过程还向链路中添加了一个TTL检验。在链路中定义了五个实例变量：

head_ 链路的入口点，它指向链路中的第一个对象。

queue_ 指向链路主队列组件的指针。简单链路通常每个链路只有一个队列。其他复杂的链路类型可能在一个链路中有多个队列组件。

link_ 一个指向链路实际模型组件的指针，这个组件负责处理链路的时延和带宽特征值。

ttl_ 指向操作packet的ttl域的组件的指针。

drophead_ 指向负责处理链路丢包的组件队列的头部所在的对象。

另外，如果模拟器实例变量$traceAllFile_被定义过，这个过程将添加跟踪组件，这个组件负责跟踪packet从queue_的入队和出队情况。进一步说，在drophead_后，跟踪将引入一个丢包跟踪组件。下面的实例变量记录跟踪组件：

enqT_ 指向跟踪进入队列queue_的packet的组件。

deqT_ 指向跟踪离开队列queue_的packet的组件。

drpT_ 指向跟踪从队列queue_中丢弃的packet的组件。

rcvT_ 指向跟踪从下一个节点中接收到的packet的组件。

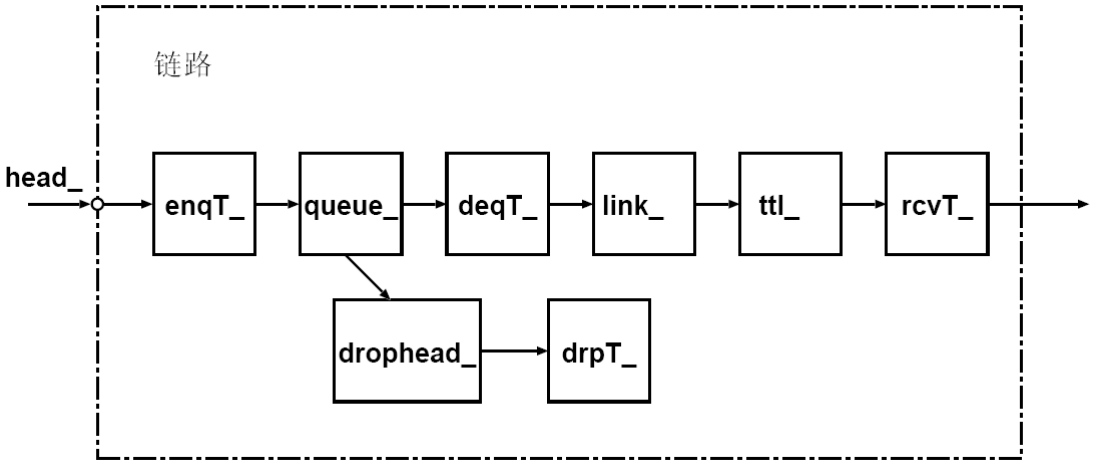


图6.1 一个单工链路的复合结构

然而要注意一点：如果用户在链路中多次启用跟踪，这些实例变量将只保存最后一个插入的组件的指针。

其他向简单链路中添加组件的配置机制有：网络接口（在组播路由中使用），链路动态模型，跟踪和监视。我们将在本章的结尾对相关的对象做一个简要的讲解，将在其他章节对其功能及实现做进一步的讨论。

实例过程duplex-link{}用两个单工链路构件成一个双工链路。

6.1 链路和简单链路的实例过程

链路过程 Link类完全都是用OTcl实现的。OTcl的SimpleLink类使用了C++的LinkDelay类来模拟packet的传输时延。Link类中的实例过程有：

head{} 返回head_的句柄。

queue{} 返回queue_的句柄。

link{} 返回时延组件link_的句柄。

up{} 将dynamics_组件中的链路状态设置为“up”。向每个文件中写入一条跟踪记录，这些文件是通过trace-dynamics{}过程指定的。

down{} 和up{}一样，将dynamics_组件中的链路状态设置为“down”。也向每个文件中写入一条跟踪记录，这些文件也是通过trace-dynamics{}过程指定的。

up?{} 返回链路状态。状态可以是“up”或者“down”，如果动态链路没有启用，则状态为“up”。

all-connectors{} 所有链路上的连接器都应用指定的操作。这种应用的一个例子：链路的所有连接器都进行复位操作。

cost{} 将链路权限设置为指定值。

cost?{} 返回链路的权限。缺省的链路权限为1，前提如果之前没有指定权限。

简单链路过程 OTcl类SimpleLink实现了一个简单的点对点链路，其中关联了一个队列和时延[11]。它是从OTcl基类Link中派生出来的，如下：

```
Class SimpleLink -superclass Link
SimpleLink instproc init { src dst bw delay q { lltype "DelayLink" } } {
$self next $src $dst
$self instvar link_ queue_ head_ toNode_ ttl_
...
set queue_ $q
set link_ [new Delay/Link]
$link_ set bandwidth_ $bw
$link_ set delay_ $delay
$queue_ target $link_
$link_ target [$toNode_ entry]
...
# XXX
# 将ttl检查放在时延之后
# 所以我们不必担心在跟踪/监视中的ttl-drop计数
#
set ttl_ [new TTLChecker]
$ttl_ target [$link_ target]
$link_ target $ttl_
}
```

注意：当创建一个SimpleLink对象时，同时创建的还有新的Delay/Link和TTLChecker对象。同时还要注意，Queue对象必须已经创建完毕。

还有两个附加的方法实现（用 OTcl 实现）是 SimpleLink 类的组成部分：trace 和 init-monitor。这些功能将在跟踪的章节（第 26 章）进行深入的讲解。

6.2 连接器

连接器与分类器不同，只为一个接收者产生数据，无论 packet 将被传输到 target_邻居还是被传输到 drop-target_。一个连接器将接收到一个 packet，执行一些功能，然后将 packe 传输到它的邻居，或者将 packet 丢弃。在 NS 中有许多不同的连接器种类。每种连接器都完成不同的功能。

networkinterface 用输入接口识别器来标记 packet，这经常在组播路由协议中使用。类变量 Simulator NumberInterfaces_1 告知 NS 添加这些接口，然后它就被添加到单工链路的任意一端。组播路由协议将在独立的一章中进

[11] 当前版本也包括了一个检查网络层ttl域的对象，在这个域的值变为0的时候将packet丢弃。

行讨论（第 30 章）。

DynaLink 根据链路状态是 up 还是 down 来控制传输的对象。在链路的头部，在模拟开始之前插入链路。它的 status_ 变量控制链路状态是 up 还是 down。DynaLink 对象使用的详细讲解单独在一个章节（第 31 章）中进行。

DelayLink 模拟链路时延和带宽特征值的对象。如果链路不是动态的，那这个对象就简单的在相应的时间为下游对象的每个 packet 调度接收到的事件。然而，如果链路是动态的话，它就会在内部将 packet 进行排列，并为其自身调度一个接收事件，以实现下一个必须被传输的 packet。另外，如果链路在某一时刻断开，这个对象的 reset()将被激活，这个对象将会因为链路失败而丢弃所有在此刻传输的 packet。我们将在另一章中讨论这个类的使用（第八章）。

Queues 模拟一个连接在链路上的网络中“真实”路由器的输出缓冲区。在NS中，它们附加在链路上并被当作链路的一部分。我们将在另一章中单独讨论队列的不同类型和NS中队列的应用（第七章）。

TTLChecker 在packet到达时减少其ttl的值。如果ttl是一个正值，这个packet将被转发到链路的下一个组件。在简单链路中，TTLCheckers是自动添加的，并在链路中作为最后的一个组件，在时延组件和下一个节点的入口之间。

6.3 对象继承

用来代表链路的基类叫做Link。这个类的方法在下面的部分中将列举出来。其他从基类派生出来的链路对象在如下所示：

- 简单链路对象 一个简单链路对象被用来代表一个简单的单工链路。这个对象没有状态变量或者配置参数。这个类的方法是：

$simplelink enable-mcast<src> <dst>

这个方法为链路的目的创建一个输入网络接口并为源创建一个输出接口来开启组播。

$simplelink trace <ns> <file> <optional:op>

为这个链路创建跟踪对象并更新对象链接。如果op被指定为nam则创建nam格式的跟踪文件。

$simplelink nam-trace <ns> <file>

在链路中创建nam跟踪。

$simplelink trace-dynamics <ns> <file> <optional:op>

这个命令为动态链路设置特殊的跟踪。<op>也允许设置为nam跟踪。

$simplelink init-monitor <ns> <gtrace> <sampleInterval>

插入对象以允许我们监视链路中队列的大小。返回可以查询决定队列平均长度的对象名。

$simplelink attach-monitors <insnoop> <outsnoop> <dropsnoop> <qmon>

这个和init-monitor类似，但是允许使用更多目标的规范。

$simplelink dynamic

为链路设置动态标志位。

$simplelink errormodule <args>

在队列之前插入一个报告错误的模块。

$simplelink insert-linkloss <args>

在队列之后插入一个报告错误的模块。

//其他从SimpleLink类派生出来的链路对象有：FQLink，CBQLink和IntServLink。

FQLink的配置参数有：

queueManagement_在链路中队列管理的类型。缺省情况下为尾部丢弃DropTail。

CBQLink和IntServLink对象没有指定配置参数。

- 链路时延对象　链路时延对象决定了packet在链路中传输所需要的时间量。这个是以大小/带宽+时延的方式定义的。大小是指packet的大小，带宽是指链路的带宽而时延则是链路传播常数的时延。没有方法或者状态变量与这个对象相关联。

配置参数有：

bandwidth_　链路的带宽，单位是比特每秒。

delay_　链路传输常数时延，单位是秒。

6.4 命令一览

下面是在模拟脚本中常用的链路命令：

$ns_ simplex-link <node1> <node2> <bw> <delay> <qtype> <args>

这个命令用于在节点1和节点2之间创建一个单工链路，它指定了带宽<BW>和时延特征值。链路使用的队列类型为<qtype>，依赖于不同队列类型的参数由<args>传递。

$ns_ duplex-link <node1> <node2> <bw> <delay> <qtype> <args>

这个命令在节点1和节点2之间创建了一条双工链路。这个过程本质上是由两条单工链路创建的双工链路，一条是从节点1到节点2的单工链路，另一条则是从节点2到节点1的。双工链路的语法和上面给出的单工链路的语法一致。

$ns_ duplex-intserv-link <n1> <n2> <bw> <dly> <sched> <signal> <adc> <args>

这个命令在节点1和节点之间创建了一条使用intserv类型队列的双工链路，指定了带宽和时延。这种队列类型实现了一个两

级服务优先级的调度器。这种intserv队列类型是由<sched>给定的，接收控制单元的类型为<adc>而信号模型类型为<signal>。

$ns_ simplex-link-op <n1> <n2> <op> <args>

这个命令用于为单工链路设置属性。属性可以有：方向、颜色、标签或者队列位置。

$ns_ duplex-link-op <n1> <n2> <op> <args>

这个命令用于为双工链路设置链路属性（例如链路通信方向、颜色、标签或者队列位置）。

$ns_ link-lossmodel <lossobj> <from> <to>

这个函数用于在链路中生成在nam中可见的丢包（在链路的<from>节点和<to>节点间使用了丢包模型<lossobj>）。

$ns_ lossmodel <lossobj> <from> <to>

这个命令用于在常规链路中插入一个丢包模型。

下面是内部与链路相关的过程清单：

$ns_ register-nam-linkconfig <link>

这是一个被”$link orient”用来注册/更新在nam中链路被创建顺序的内部过程。

$ns_ remove-nam-linkconfig <idl> <id2>

这个过程是用来删除重复的链路的（重复的链路可能是由GT-ITM拓扑生成器产生的）。

$link head

为链路返回实例变量head_。head_是链路的入口点，并且它指向链路的第一个对象。

$link add-to-head <connector>

这个命令允许链路中的head_指向<connector>对象，例如：<connector>现在变成了链路中的第一个对象。

$link link

返回实例变量link_。link_是链路中的一个组件，它是链路中实际上模拟链路时延和带宽特征值的组件。

$link queue

返回实例变量queue_。queue_是链路中的队列组件。在一个特殊的链路中可以有一个或多个队列组件。

$link cost <c>

这个命令将链路优先权设置为<c>。

$link cost ?

返回链路的优先权值。缺省的链路优先权被设置为1。

$link if-label?

返回与链路相关的网络接口（用与组播路由）。

$link up

这个命令将链路状态设置为一“up”。这个命令在NS中是支持动态网络的一部分。

$link down

和$link up命令相似，这个命令将链路状态标记为“down”。

$link up?

返回链路状态。缺省情况下，状态值永远为“up”，如果动态链路没有被启用。

$link all-connectors op

这个命令将给定的操作<op>应用到链路上所有的连接器。例如：$link all-connectors reset或者$link all-connectors isDynamic。

$link install-error <errmodel>

这个命令将在link_组件之后安装一个报告错误的模型。

除了上面列出的链路及链路相关的命令之外，还有其他的一些支持不同类型链路的特殊需求的过程，这些链路都是从基类Link中派生出来的：简单链路（SimpleLink）、综合服务链路（IntServLink）、基类队列链路（CBQLink）、公平队列链路（FQLink）以及支持组播路由、sessionsim和nam的过程等。这些以及上面的过程可以在ns/tcl/lib文件（ns-lib.tcl,ns-link.tcl,ns-intserv.tcl,ns-namsupp.tcl,ns-queue.tcl）、ns/tcl/mcast文件（McastMonitor.tcl,ns-mcast.tcl）和ns/tcl/session/session.tcl文件中找到。

第 7 章

队列管理（ Queue Management ）和包调度 (Packet Scheduling)

队列代表的是存放（或者丢弃）包的地方。包调度是指用于选择哪个包应该被服务或者丢弃的抉择过程。缓冲区管理是指规范一个特定队列的占用率的特殊规则。当前所提供的支持包括：丢弃队尾(drop-tail)（先进先出(FIFO)）排队，RED缓冲区管理，CBQ（包括优先级和轮换调用），和各种公平排队(Fair Queueing)包括： 公平排队(Fair Queueing(FQ)) ， 随即公平排队(Stochastic Fair Queueing(SFQ))，和赤字轮换(Deficit Round-Robin(DRR))。通常情况下，一个延时元素是一个队列的下游，这时如果队列被阻塞，他下游的邻居可以使之重新活动。这就是传输延时模拟的机制。另外，有时候队列可能被他们的邻居强制阻塞或者畅通（这是用队内流控制来实现多队列集合队列）。丢包由队列所包含的“丢弃目标”("drop destination")来实现的；丢弃目标是用来接收所有被队列丢弃的包的对象。这（例如）对统计丢弃包非常有帮助。

7.1 C++中的队列类

队列类是由一个连接器基类派生的。它提供一个基类。这个基类可以派生出各种队列类，同样也可以调用函数实现阻塞（见下一节）。下面的定义由queue.h 提供：

```
class Queue : public Connector {
public:
virtual void enque(Packet*) = 0;
virtual Packet* deque() = 0;
void recv(Packet*, Handler*);
void resume();
int block();
void unblock();
void block();
protected:
Queue();
int command(int argc, const chat*const* argv);
int qlim_; /*队列所允许的最大包数* /
int blocked_;
int unblock_on_resume_; /*畅通的处于空闲状态的队列* /
QueueHandler qh_;
 }
```

enque和deque函数是纯虚函数，表示队列类用做基类；任何队列都由队列类派生，并且必须实现这两个函数。通常，特殊的队列不重载recv函数，因为它调用了这个队列的enque和deque。

队列类没有包含太多的内部状态。这些通常是些特殊的监视对象（23章）。qlim_成员是用来表示队列的最大占有率的。

但是没有被队列类强化；如果一些特殊的队列子类需要这个值，他们将会使用它。blocked_成员是一个布尔值，表示队列是否可以马上发报到它的下游邻居。当一个队列被阻塞时，它可以接收包，而不可以发送他们。

7.1.1 队列阻塞（Queue blocking）

一个队列在任何时候都既可以是阻塞有可以畅通。通常，当一个包在队列和它的下游邻居传输时（如果大多数时间队列非空闲）那么队列就是阻塞的。一个队列将保持阻塞，只要它的下游链路繁忙并且队列里面至少有一个包需要发送。一个队列只有当它的resume 函数被调用（这是由它的下游邻居通过回调调度）时才能变成畅通，这个通常是因为队列中没有包时发生。这个回调是通过使用下面的类和方法实现的：

```
class QueueHandler : public Handler {
public:
inline QueueHandler(Queue& q) : queue_(q) {}
void handle(Event*); /*调用 queue_.resume() */
private:
Queue& queue_;
};
void QueueHandler::handle(Event*)
{
queue_.resume();
}
Queue::Queue() : drop_(0), blocked_(0), qh_(*this)
{
Tcl& tcl = Tcl::instance();
bind("limit_", &qlim_);
}
void Queue::recv(Packet* p, Handler*)
{
enque(p);
if (!blocked_) {
/*
* 我们没有阻塞。获得一个包，然后发送。我们执行一个额外的检测，因为队列可能丢弃包，甚至它以
* 是空的！ （例如，RED可能是这样。）
* /
p = deque();
if (p != 0) {
blocked_ = 1;
target_->recv(p, &qh_);
}
}
}
void Queue::;resume()
{
Packet* p = deque();
if (p != 0)
```

```
target_->recv(p, &qh_);
else {
if (unblock_on_resume_)
blocked_ = 0;
else
blocked_ = 1;
}
}
```

这里的句柄管理从某种程度来讲有一些微妙。当一个新的队列对象创建的时候，它包含了一个队列句柄(QueueHandler)对象(qh_)，这个对象初始化时就指向这个新的队列对象(Queue& QueueHandler::queue_)。这是由队列的构造函数用表达式qh_(*this)执行的。当一个队列接收到一个包时，它调用子类（例如，特殊的排队规则）中的enque 函数来处理包。如果队列没有被阻塞，它允许发送包并且调用特定的deque 方法来决定哪一个包应该被发送，然后阻塞队列（因为现在一个包正在传输中），最后发送这个包到队列的下游邻居。注意：以后任何从上游邻居接收到的包将会到达一个阻塞的队列。当一个下游邻居希望是队列变为畅通的，它通过传递&qh_到模拟调度器来调度队列句柄的Handle 函数。Handle 函数调用resume，它将发送下一个该调用的包到下游（使队列阻塞），或者如果没有包可以被发送就使队列畅通。这个过程可以通过参考LinkDelay::recv()方法(8.1)来进一步了解。

7.1.2 PacketQueue类

队列类可以实现缓冲区管理和调度，但是不能在特定的队列上实现低层次的操作。包队列类就是用于这个目的的，它是如下定义的（见queue.h）：

```
class PacketQueue {
public:
PacketQueue();
int length; /*队列的长度，以包来计算*/
void enque(Packet* p);
Packet* deque();
Packet* lookup(int n);
/*删除特定的队列中包*/
void remove(Packet*);
protected:
Packet* head_;
Packet** tail_;
int len_; //包数
};
```

这个类包含一个包的连接链表，通常被特殊的调度和缓冲管理规则用来保存一种包的顺序。特别的调度或者缓冲管理方案可能利用几种包队列对象。包队列类包含当前的队列中的包数，这个包数是由length()方法返回的。函数enque 将特定的包放在队列尾部并且修改成员变量len_。函数deque 返回队列头部的包，然后将其从队列中删除（然后修改计数器），或者如果队列为空则返回空。函数lookup返回从队列头数器的第n 个包，否则则返回空。函数remove 从队列中删除给定地址的包（然后修改计数器）。如果队列不存在，它将导致程序异常而终止。

7.2 示例：丢尾(Drop Tail)

下面的例子说明的是Queue/DropTail对象的实现，这个对象实现了在当前大多数因特网路由器中比较典型的先进先出(FIFO)调度和丢弃多余的缓冲区管理。下面定义声明了类和它的OTcl连接：

```
/*
* 一个有限制的丢尾队列
*/
class DropTail : public Queue {
protected:
void enque(Packet*);
Packet* deque();
PacketQueue q_;
};
```

派生出DropTail 类的队列基类提供大多数所需的功能。丢尾队列包含有且仅有一个先进先出(FIFO)队列，它是由包括包队列类的对象实现。丢尾队列实现自己的enque和deque，如下：

```
/*
* 丢尾
*/
void DropTail:;enque(Packet* p)
{
q_.enque(p);
if(q.length() >= qlim_) {
q_.remove(p);
drop(p);
}
}
Packet* DropTail::deque()
{
return (q_.deque());
}
```

因此，函数enque首先在队列内部（这是没有大小限制的）存储包通过qlim_来检测队列的大小。丢弃多余就是当极限达到或超过的时候从队列中丢弃最近添加的包。注意：在上面实现enque 的时候，设置qlim_为n，实际上意味着队列的大小为n-1。简单的先进先出(FIFO)调度就是通过函数enque 一直返回队列中的第一个包来实现的。

7.3 不同类型的队列对象

一个队列对象是一个通常意义上的类对象，它可以保持、可能的标记或丢弃包，当这些包通过模拟拓扑的时候。队列对象的设置参数是：

limit_ 队列的大小，以包为单位。

blocked_ 默认为假，如果队列阻塞（不能发送包到它的下游邻居）则为真。

unblock_on_resume_ 默认为真，表示一个队列在最后一个包发送出去（不一定非要被接收到）的时候畅通。

其他的由基类Queue 派生队列对象是drop-tail、FQ、SFQ、DRR 和CBQ 队列对象。每一个描述如下：

- Drop-tail 对象：它是队列对象的一个实现了先进先出队列的子类。drop-tail对象没有特别的方法、设置参数、或者状态变量。

- FQ 对象：它是队列对象的一个实现了公平排队的子类。FQ 对象没有特别的方法。设置参数为：

secsPerByte_ 这个对象没有相关的状态变量。

- SFQ 对象：SFQ 对象是队列对象的一个实现了随机公平排队的子类。SFQ对象没有特别的方法，设置参数为：

maxqueue_

buckets_

这个对象没有相关的状态变量。

- DRR 对象：DRR对象是队列对象的一个实现了赤字轮换调度的子类。这些对象在不同的流（一种特殊的流就是它的包拥有相同的节点和端口id或者只拥有相同的节点id）内实现赤字轮换调度。同样和其他的多队列对象不同的是，这个队列对象实现了一个唯一的对所有流开放的共享缓冲区空间。设置参数是：

buckets_ 表示用于哈希每个流的桶总数。

blimit_ 表示共享缓冲区的大小，以字节计算。

quantum_ 表示轮到的时候每个流可以发送多少字节。

mask_ 当设置为 1 的时候，意味着一个含有相同节点id（不一定有相同的端口id）包的流，否则，表示一个含有相同节点和端口id 的包的流。

- RED 对象：RED 对象是队列对象的一个实现了随机先查网关的子类。这个对象可以被配置为丢弃或者“标记”包。RED对象没有特殊的方法。设置参数为：

bytes_ 当到达的包会影响可能的标记（丢弃）包时，就设为“真”来开启“字节模式(byte-mode)”RED 。

queue-in-bytes_ 设置为“真”来用字节衡量队列的大小，而不是包数。启用这个选项也同时让mean_pktsize_（见下）自动扫描thresh_和maxthresh_。

thresh_ 队列平均包数的最小值。

maxthresh_ 队列平均包数的最大值。

mean_pktsize_ 一个对包的字节大小的粗略估计。用在一段空闲时期后更新计算出的队列的平均大小。

q_weight_ 队列的权重，用于指数权重的动态平均值，这个平均值是用来计算队列大小的均值。

wait_ 设置为真则使丢包之间保持间隔。

linterm_ 由于队列的平均大小在"thresh_"和"maxthresh_"之间变化，丢包的概率可能就在0 和"1/lintern"之间变化。

setbit_ 设置为“真”，通过设置包头中的拥塞显示位标记包，而不是丢弃包。

drop-tail_ 设置为真，当队列溢出或者平均大小超过"maxthresh_"时，使用丢尾而不是随机丢弃。这个变量更深入的解释在[2]中。

RED实现中的状态变量不可显示。

- CBQ 对象：CBQ 对象是队列对象的一个实现了class-based 排队的子类。

$cbq insert <class>

插入交通类到与链路对象cbq 关联的共享链路结构。

$cbq bind <cbqclass> <id1> [$id2]

使含有id 为<id1>（或者从id1 到id2，包含边界）的流的包与交通类cbqclass关联。

$cbq algorithm <alg>

选择CBQ 的内部算法。<alg>可能是"ancestor-only","top-level"或者"formal"中的一种。

- CBQ/WRR 对象：CBQ/WRR 对象是队列对象的一个实现了在同一个优先级水平上的类之间的权重轮换调度的子类。与此对照的是，CBQ 对象实现的是同一优先级层次的类之间包与包的轮换调度。设置参数是：

maxpkt_ 包的最大字节数。这个仅仅在CBQ/WRR 对象为权重的轮换调度器计算最大带宽分配时用到。

CBQCLASS OBJECTS

CBQClass 对象实现与CBQ 对象相关联的交通类。

$cbqclass setparams <parent> <okborrow> <allot> <maxidle> <prio> <level>

为CBQ 交通类设置一些配置参数（见下）。

$cbqclass parent <cbqcl|none>

指定在共享链路树中这个类的父亲。当父亲指定为"none"时，表示这个类是根。

$cbqclass newallot <a>

改变这个类的链路分配到特定的量（从0.0 到1.0）。注意：只有指定的类有效。

$cbqclass install-queue <q>

安装一个队列对象到复合的CBQ 或者CBQ/WRR 链路结构里。当一个CBQ 对象最初被创建时，它没有包含内部队列（只有一个包分类器和调度器）。

配置参数是：

okborrow_ 是一个布尔值，表示类可以从它的父亲处借到带宽。

allot_ 是分配给这个类的链路带宽的最大值，数值在0.0 到1.0 之间。

maxidle_ 是一个类可能需要的在传送包之前将包放入队列的时间的最大量。

priority_ 是类的相对于其他类的优先级水平。这个值可能从0 到10，而且同一个优先级可以存在多个类。优先级0 最高。

level_ 是在共享链路树中这个类的层次。树中的叶子节点层次为1；他们的父亲为2，等等。

extradelay_ 特定时间通过一个延时类增加延时。

QUEUE-MONITOR OBJECTS

QueueMonitor 对象用来监视一套包和字节的到达、离开和丢弃的计数器。它也包含对集合统计的支持，例如，队列大小的平均值等等。

$queuemonitor

重新设置下述所有的累积计数器（到达、离开和丢弃）为0。同样，如果定义了积分器和延时采样，则重新设置。

$queuemonitor set-delay-samples <delaySamp_>

设置采样对象delaySamp_来记录关于队列延时的统计数据。delaySamp_是一个采样对象的句柄。也就是采样对象应该已经创建。

$queuemonitor get-bytes-integrator

返回一个积分器对象，这个对象可以用来查找以字节计算的队列大小的总数。

$queuemonitor get-pkts-integrator

返回一个积分器对象，这个对象可以用来查找以包数计算的队列大小的总数。

$queuemonitor get-delay-samples

返回一个采样对象delaySamp_来记录队列延时的统计数据。

这个对象没有特殊的设置参数。

状态变量为：

size_ 队列的瞬时大小，以字节计。

pkts_ 队列的瞬时大小，以包数计。

parrivals_ 计算到达包的总数。

barrivals_ 计算到达包中所含的字节的总数。

pdepartures_ 计算离开（或者丢弃）包的总数。

bdepartures_ 计算离开（或者丢弃）包所含字节的总数。

pdrops_ 丢弃的包的总数。

bdrops_ 丢弃的字节的总数。

bytesInt_ 以字节计算队列大小总数的积分器对象。这个对象的变量sum_有以字节计算的队列大小的总数。

pktsInt_ 以包数计算队列大小总数的积分器对象。这个对象的变量sum_有以包数计算的队列大小的总数。

QUEUEMONITOR/ED OBJECTS

这个派生对象可以从早期的丢包中区分出有规律的丢包来。一些队列是从其他丢包（如，RED 队列中的随机丢包）区分出特殊的丢包（如，由于缓冲区透支而丢包）。在一些环境中。区分这两种丢包是非常有用的。

状态变量是：

epdrop_ 早期被丢弃的包的数量。

ebdrops_ 组成早期被丢弃的包的字节总数。

注意：因为这个类是Queuemonitor 类的子类，这种类型的对象也有一些域如：pdrops_和bdrops_。这些域描述了丢的包和字节的总数，包括早期和非早期的丢包。

QUEUEMONITOR/ED/FLOWMON OBJECTS

这些子类可能用在传统的QueueMonitor 对象的地方，而且是希望收集包括基本的QueueMonitor 提供的集合数量和统计数据在内的每个流的数量和统计的时候。

$fmon classifier <cl>

这是插入（读取）特定的分类器到（从）流监视器对象。这是用来制定输入包到与之关联的流上。

$fmon flows

返回一个字符串，包含所有这个监视器知道的流对象的名称。每个这些对象都是QueueMonitor/ED/Flow 类型的。

$fmon attach <chan>

附加一个tclI/O 通道到流监视器。当执行转存操作时流统计数据就写到通道上去。

配置参数是：

enable_in_ 默_____认设置为真，表示每个流到达状态应该被这个流的监视器保存。如果设置为假，仅仅只有集合的到达信息被保存。

enable_out_ 默认设置为真，表示每个流离开状态应该被这个流的监视器保存。如果设置为假，仅仅只有集合的离开信息被

保存。

enable_drop_ 默认设置为真，表示每个流丢包状态应该被这个流的监视器保存。如果设置为假，仅仅只有集合的丢包信息被保存。

enable_edrop_ 默认设置为真，表示每个流早期丢包状态应该被这个流的监视器保存。如果设置为假，仅仅只有集合的早期丢包信息被保存。

QUEUEMONITOR/ED/FLOW OBJECTS

这个对象包含由一个QueueMonitor/ED/Flowmon 对象管理的每个流的计数和统计。当一个流监视器监视到一个它未知流的包时，他们通常被一个OTcl 回调过程创建。注意：流监视器的分类器负责用一些强制的方法让包到特定的流。因此，根据使用的分类器种类，不是所有的状态变量都相关（例如，可以仅仅根据流id分类，如果这样的话，源和目的地址就不是很重要了）。状态变量是：

src_ 属于这个流的包的源地址。

dst_ 属于这个流的包的目的地址。

flowid_ 属于这个流的包的流id。

7.4 命令一览

下面是用在模拟脚本里的队列命令的清单：

$ns_ queue-limit <n1> <n2> <limit>

这个设置了n1 与n2 之间链路上的队列缓冲区的最大值。

$ns_ trace-queue <n1> <n2> <optional;file>

这是设置纪录队列中事件的跟踪对象。如果tracefile 没有设置，它将使用traceAllFile_来纪录事件。

$ns_ namtrace-queue <n1> <n2> <optional:file>

和上面的trace-queue 相似，这个是设置队列中的nam-tracing 的。

$ns_ monitor-queue <n1> <n2> <optional:flie> <optional:sampleinterval>

这个命令插入一个允许我们监视队列大小的对象。这个返回这个对象的句柄，它可能用来查询并决定队列的平均大小。sampleinterval 的默认值为0.1。

7.5 Queue/JoBS

JoBS 由Nicolas Christin<nicolas@cs.virginia.edu>发展并贡献的。

这章介绍的是关联缓冲区的管理和调度(Joint Buffer Management and Scheduling(JoBS))算法在ns 中的实现。这章有三个部分。第一部分总结了JoBS算法的目标。第二部分解释了怎样在ns 中配置一个JoBS 队列。第三部分主要讲JoBS 中实现的跟踪机制。

这章里面所讲的过程和函数可以在ns/jobs.{cc,h} ，ns/marker.{cc,h} ，ns/demarker.{cc,h}中找到。可以在ns/tcl/ex/jobs-{lossdel,cn2002}.tcl 中找到脚本例子。

其他的信息，可以在http://qosbox.cs.virginia.edu 找到。

7.5.1 JoBS 算法

这个部分给出了JoBS算法所要达到的目标，和到达这些目标所使用的机制的概述。像文献[20]中所描述的一样，原始的JoBS算法采用了非线性优化问题的解决方法。这个ns-2的实现采用了基于文献[8]中所描述的试探法的反馈控制(feedback-control)。

注意：这个ns-2的实现源于ns-2.1b5的代码和从BSD内核层次的JoBS算法的实现派生的代码综合而成。这个仍然处于试验阶段。由于在tclcl中当前缺乏Tcl和C++之间的数组的绑定方法 ,交通类的数量只有静态的4 种 ,而且如果不修改C++代码，这个就不会变化。

目标(Objective)

JoBS 算法的目的是提供绝对的和相对的丢包率和每个节点的通信类的独立延时区别的。JoBS 因此提供一个每跳的确保服务。该系列的表现需要服务质量(QoS)约束的特殊算法。作为例子，对于三种类，服务质量约束可以使下列形式：

- Class-1 延时 ≈ 2• Class-2 延时，
- Class-2 延时 ≈ 10(−1) • Class-3 丢包率，或者
- Class-3 延时 ≤ 5 ms。

这里，前两个是相对约束，最后一个是绝对约束。这一系列约束可以是任何相对和绝对约束的混合。进一步说，JoBS 支持下面五种类型的约束：

- 相对延时约束(RDC)表示类与类之间一个按比例的延时差别。作为一个例子，对于类-1 和2，RDC 划定的一个关系：
- 绝对延时约束(ADC)：类i 上的一个ADC 要求类i 的延时必须满足一个有最差为的延时。Di
- 相对丢包率约束(RLC)表示类与类之间一个按比例的丢包率差别。
- 绝对丢包率约束(ALC)：类i 上的一个ALC 要求这个类的延时必须有个上限。Li
- 绝对速率约束(ARC)：类i 上的一个ARC 意味着类i 的输出有个底线iμ 。

JoBS 不依靠管理控制或交通规则，也不对交通到达做任何假设。因此，一个约束系统可能变得不现实，同时一些约束可能需要放松。QoS约束的优先级顺序如下：

ALC > ADC,ARC >Relative Constraints.

也就是，如果JoBS不能同时满足绝对和相对约束，它将优先考虑绝对约束。

机制(Mechanisims)

JoBS 通过同一个通道进行调度和缓冲区管理。为了满足延时约束JoBS 动态的分配服务速率到各个类。用来满足绝对延时约束的这些服务速率必须在包到达的时候被分配到，然而由相对延时约束产生的服务只有当每N 个包到达的时候才能被计算。如果没有可行的服务速率分配存在1，或者如果包缓冲区溢出，包将根据丢包率约束被丢弃。

服务速率通过一个和赤字轮换一样的算法被改为包调度的决定。也就是说，调度器尝试着通过跟踪每个类的实际传输速率和他们需要的服务速率的差别来达到期望的服务速率。JoBS中的调度是节省型工作(work-conserving)。

7.5.2 配置

执行一个JoBS 模拟需要创建和配置JoBS “链路” ,创建和配置标志器(Markers)和去标志器(Demarkers)来管理通信的标志/去标志；附加一个应用层数据源（流量产生器），和启动流量产生器。

初始化步骤

```
set ns [new Simulator]  ;# 最初的初始化
Queue/JoBS set drop_front_ false  ;# 使用丢尾
Queue/JoBS set trace_hop_ true  ;# 开启统计跟踪
Queue/JoBS set adc_resolution_type_ 0  ;# 见“命令一览表”
```

Queue/JoBS set shared_buffer_ 1 ；# 所有类共享一个缓冲区
Queue/JoBS set mean_pkt_size_ 4000 ；# 我们期望受到500-字节的包
Queue/Demarker set demarker_arrvs1_ 0 ；#重置每个地方的到达数
Queue/Demarker set demarker_arrvs2_ 0
Queue/Demarker set demarker_arrvs3_ 0
Queue/Demarker set demarker_arrvs4_ 0
Queue/Marker set demarker_arrvs1_ 0
Queue/Marker set demarker_arrvs2_ 0
Queue/Marker set demarker_arrvs3_ 0
Queue/Marker set demarker_arrvs4_ 0
set router(1) [$ns node] ；# 设置第一个路由器
set router(2) [$ns node] ；# 设置第二个路由器
set source [$ns node] ；# 设置源
set sink [$ns node] ；# 设置交通接收器
set bw 10000000 ；# 10Mbps
set delay 0.001 ；# 1ms
set buff 500 ；# 500 个包
创建JoBS链路
$ns duplex-link $router(1) $router(2) $bw $delay JoBS ；#创建一个JoBS 链路
$ns_ queue-limit $router(1) $router(2) $buff
set l [$ns_ get-link $router(1) $router(2)]
set q [$l queue]
$q init-rdcs -1 2 2 2 ；# 类-2、3 和4 被按比例因子为2 的延时差别约束
$q init-rlcs -1 2 2 2 ；# 类-2、3 和4 被按比例因子为2 的丢包率差别约束
$q init-alcs 0.01 -1 -1 -1 ；# 类-1 提供了一个1%的丢包率限制
$q init-adcs 0.005 -1 -1 -1 ；# 类-1 提供了一个5ms 的延时限制
$q init-arcs -1 -1 -1 500000 ；# 类-4 提供了一个最小输出为500Kbps 的限制
$q link [$l link] ；# 这个链路附加在队列上（必须的）
$q trace-file jobstrace ；# 每跳每类的跟踪， 写入j o b s t r a c e
$q sampling-period 1 ；# 每个到达的相关速率分配
$q id 1 ；# 分配给JoBS 队列一个ID，为1
$q initialize ；# 进行初始化

标记流量

标记交通是通过标记器对象控制的。标记器是一个先进先出(FIFO)队列，这个队列设置了每个包的类下标。为了保证模拟的精确性，最好给这些队列配置非常大的缓冲区，可以使包丢弃到标记器里。去标记器是用来收集端对端的延时统计数据的。

$ns_ simplex-link $source $router(1) $bw $delay Marker ；# 设置标记器
$ns_ queue-limit $source $router(1) [expr $buff*10] ；# 为标记器选取大量的缓冲区
$ns_ queue-limit $router(1) $source [expr $buff*10] ；# 为了避免交通丢包
set q [$ns_ get-queue $source $router(1)] ；# 在标记器内部
$q marker_type 2 ；# 统计的标记器
$q marker_frc 0.1 0.2 0.3 0.4 ；# 10%类-1，20%类-2，30%类-3，40%类-4。
$ns_ simplex-link $router(2) $sink $bw $delay Demark ；# 设置去标志器

$ns_ queue-limit $router(2) $sink [expr $buff*10]
$q trace-file e2e ；# 跟踪端对端的延时，并记录到文件e2e 中

剩余的步骤（附加代理和交通产生器或者应用到节点）将在第10和38章介绍，也像通常一样控制。这些章节和例子脚本供你们的ns使用。

7.5.3 跟踪

JoBS中的跟踪是通过调度器内部控制的。每个JoBS队列可产生一个包含下列信息的跟踪文件。跟踪文件的每行包括17列。第一列示模拟时间，第2到5列表示类-1到类-4在当前繁忙时刻的丢包率，第6到9列表示每一类的延时（在一个0.5秒时间片的均值），第10到13列表示过去0.5秒内分配给每个类的平均服务率，第14 到17 列表示瞬间的队列包数。另外，去标志器可以用来跟踪端对端的延时。

7.5.4 变量

这部分总结被JoBS、标记器和去标记器对象使用的变量。

JoBS 对象

trace_hop_ 可以为真或者假。如果为真，每一跳每一个类的数据将被记录。（跟踪文件然后被用<JoBS object> trace-file <filename>说明），默认值为假。

drop_front_ 可以为真或者假，如果为真，交通将从队列的前面开始丢弃。默认为假（丢尾）。

adc_resolution_type_ 可以是0或者1 ,如果是0 ,且如果这个ADC不能通过调节服务率得到满足 ,交通将丢弃有ADC 的类。如果是1，交通将丢弃所有类。一个为1的解决模式因此更公平，只是从痛苦是被所有类分担的的感觉上说，但是这样可能会导致更多的强制限期。默认为0。

shared_buffer_ 可以是0 或者1。如果是0，所有类使用各自的缓冲区（如果只有速率保证要提供，这就是必须的）。所有每个类缓冲区一样大。如果是1，所有类共享同一个缓冲区（如果要提供丢包率差别，则这是必须的）。默认值为1。

mean_pkt_size_ 用来设置JoBS 链路中期望的到达包的大小的中值。为了保证适当的延时差别 ,这个变量的设置是必须的。

标记器对象

marker_arrvs1_ 类-1 进入标记器链路的包数。

marker_arrvs2_ 类-2 进入标记器链路的包数。

marker_arrvs3_ 类-3 进入标记器链路的包数。

marker_arrvs4_ 类-4 进入标记器链路的包数。

去标记器对象

demarker_arrvs1_ 类-1 进入去标记器链路的包数。

demarker_arrvs2_ 类-2 进入去标记器链路的包数。

demarker_arrvs3_ 类-3 进入去标记器链路的包数。

demarker_arrvs4_ 类-4 进入去标记器链路的包数。

7.5.5 命令一览

下面是用来配置JoBS、标记器和去标记器对象的命令列表

JoBS

set q [new Queue/JoBS]

这个创建一个JoBS 队列实例。

$q init-rdcs <k1> <k2> <k3> <k4>

这是设置RDC 给四个JoBS 类。例如，用值4给k2表示类-3 延时将大约为类-2延时的4倍。值为-1表示这个类不考虑RDC。注意：由于RDC 要限制两个类，所以，只有三个参数可以传递（k1，k2，k3，由于k4 理论上限制类-4 和类5，但是类5 不存在）。但是，在原始的实现中，如果类-4 要考虑RDC，则必须设置不同于0 和-1 的值。

例如：$q init-rdcs -1 2 1 -1 说明类-2 和类-3 被一个延时差别因子2 限制，$qinit-rdcs 4 4 4 4 说明所有类都被延时差别因子4 限制，和$q init-rdcs 4 4 4 1 等效，由于最后一个仅仅说明类-4 要受按比例的差别限制。

$q init-rlcs <k'1> <k'2> <k'3> <k'4>

这是给四个JoBS类设置RLC的。例如，用值3给k1表示类-2丢包率将大约是类-1的3倍。值为-1表示这个类不考虑RLC。和RDC一样，每个RLC限制两个类，因此，只有三个参数可以传递（k'1,k'2,k'3，由于k'4 理论上限制类-4 和类5，但是类5不存在）。像上面解释的一样，如果类-4要考虑RLC，则必须设置不同于0 或-1 的值。

$q init-alcs <L1> <L2> <L3> <L4>

这是设置绝对丢包率保证给所有四个类。L1 到L4 给定了一个小于1 的数。例如，设置L1 为0.05 表示类-1 的丢包率将确保比5%少。值为-1 表示相应的类不受ALC限制。

$q init-adcs <D1> <D2> <D3> <D4>

这是设置绝对延时保证给所有四个类。D1 到D4 给定的单位为微秒。值为-1 表示相应的类不受ADC 限制。

$q trace-file <filename>

这是说明每跳数据的跟踪文件。JoBS采用一个内置的模块来跟踪丢包率和延时，服务率，和每个类队列中的包数。如果filename 设置为空，将不会跟踪。

$q link [<link-object> link]

这个命令是要绑定一个链路到一个JoBS 队列。注意JoBS 需要知道这个链路的容量。因此，这个命令应该在模拟开始前给出。

$q sampling-period <sampling-interval>

这个命令表示抽样的间隔（以包为单位），以此来执行按比例差别的服务率的调整。这个的默认值为一个包的抽样间隔。这意味着当每个包到达时都要重新评估速率分配。抽样间隔越大将加速模拟。但是通常造成更差的按比例差别。

$q id <num_id>

这个命令影响JoBS队列的数字ID。

$q initialize

这个命令是必须的，并且应该在所有配置操作执行之后执行。这个命令将执行JoBS队列的最后校正和配置。

$q copyright-info

显示作者和版权信息。

一个简单的完全注释和解释了的例子脚本（ 有nam 输出） 可以在ns/tcl/ex/jobs-lossdel.tcl中找到。一个用JoBS队列模拟

的更加现实的例子可以在ns/tcl/ex/jobs-cn2002.tcl 中找到。这个脚本和[21]中表述的一个模拟非常相似。相关的跟踪文件和产生视觉效果的gnuplot 脚本（如果你比xgraph 更喜欢gnuplot的话）可以在ns/tcl/ex/jobs-lossdel 和 ns/tcl/ex/jobs-cn2002 中可以找到。

标记器对象

$q marker_type <1|2>

选择标记器的类型。1是DETERMINISTIC，2是STATISTICAL。

$q marker_class <1|2|3|4>

对一个deterministic 的标记器，选择要标记的包的类。

$q marker_frc <f1> <f2> <f3> <f4>

对一个statistic 的标记器，给一个标记各个类中包的比例。例如，用0.1 给f1 表示10%的到标记器的交通将被标记为类-1。

去标记器对象

$q trace-file <filename>

这个命令表示用于去标记对象的跟踪文件。filename.1 包含每个类-1 的到达去标志器链路的包端对端的延时。filename.2 包含每个类-2 的到达去标记器链路的包端对端的延时，以此类推。（当然对4 个类应该有4 个跟踪文件）。

第 8 章

延时和链路

延时是指一个包穿过链路所需要的时间。这种对象（“动态链路”）的一种特殊形式还可以模拟链路失败的概率。一个包穿过链路所需要的时间量定义为s/b+d，这里s是包的大小（如IP 头记录的那样），b是链路的网速，以位/秒为单位，d是链路延时，以秒为单位。链路延时的实现和7.1.1 中队列描述的阻塞过程密切相关。

8.1 LinkDelay类

类LinkDelay是由基类Connector 派生而来。它的定义在~ns/delay.cc里，下面将做简单的摘要：

```
class LinkDelay : public Connector {
public:
LinkDelay();
void recv(Packet* p, Handler*);
void send(Packet* p, Handler*);
void handle(Event* e);
double delay(); /*这个链路的线性延时*/
double bandwidth(); /* 这个链路的带宽* /
inline double txtime(Packet* p) { / *在这个链路上发送包p的时间* /
hdr_cmn* hdr = (hdr_cmn*) p->access(off_cmn_);
return (hdr->size() * 8. / bandwidth_);
}
protected:
double bandwidth_; /*基层链路的带宽（位/秒）*/
double delay_; /*线性延时* /
int dynamic_; /*表示链路是否~ * /
Event inTransit_;
PacketQueue* itq_; /* 动态链路的内部包队列* /
Packet* nextPacket_; /* 动态链路所要投递的下一个包* /
Event intr_;
};
```

recv()方法重载了基类Connector 版本的方法。定义如下：

```
void LinkDelay::recv(Packet* p, Handler* h)
{
double txt =txtime(p);
Scheduler& s = Scheduler::instance();
if (dynamic_) {
Event* e = (Event*)p;
e->time_ = s.clock() + txt + delay_;
```

```
itq_->enque(p);
schedule_next();
} else {
s.schedule(target_, p, txt + delay_);
}
/ * XXX 只需要一个intr_，因为上游对象在句柄被调用之前会处于阻塞状态。
*
* 这个只是在链路不为动态的时候成立。如果为动态，那么
* 链路自己将保存这个包，然后在适当的时候调用上游对象。这第二个间断叫做
* inTransit_，它是通过schedule_next()调用
*/
s.schedule(h, &intr_, txt);
}
```

这个对象支持一个类成员函数，$object dynamic，来设置它的变量，dynamic_。这个变量决定链路是否为动态（例如，在某个时候倾向失效/恢复）。在这两种情况下，链路的内部行为是不同的。

对于“非动态”链路，这个方法是被包p的接收和调度两个事件操控。假设这两个事件为E1和E2，E1在E2前调度。E1是在依附在这个延时元素上的上游节点已经完全发送了当前的包（这个时间应该等于包的大小除以链路带宽）时被调度发生。通常与一个队列对象相关，也会激发它使之（可能）变为畅通（见7.1.1）。表示包的到达这个有延时元素的下游邻居的事件。事件E2发生在E1之后的时间大小与链路延时应该相等。

另外一种，当链路是动态的，接收到p，然后它将调度E1去在某一个时刻使队列畅通。但是， E2只有当p是当前传输的唯一包时才能被调度。否则，传送中至少有一个链路上的包必须在E2也就是在p之前被投递。因此，包p被保存在对象的传输队列itq_中。当这个包在p 在链路上传输之前被投递到邻居节点，DelayLink 对象将为自己调用一个事件在E2时发生。在那个时候，它的handle()方法将直接发送p 到它的目的地。对象的内部方法schedule_next()将调度在适当的时候这些事件让包传输。

8.2 命令一览

对象LinkDelay 表示一个包穿过链路，并且在链路里需要的时间。因此这里我们没有列出任何与链路延时相关而且适合于模拟脚本的命令。

第 9 章

ns中的差异服务模块（Differentiated Services Module）

注：本章中讲解的差异服务模型已被集成到**ns-2.1b8** 中。

差异服务（简称为DiffServ）是一种IP QoS 体系，它基于包标记以根据用户要求使得不同的包有不同的优先级。发生拥塞的时候，优先级较低的包丢失的可能性比优先级高的更大一点。本章中讲述的DiffServ模型起初应用于Nortel网中的高级IP网络。

9.1 概述

DiffServ体系通过把业务划分为不同的类别以保证QoS，每个包都用一个代码点加以标记来指示它的类别，并相应地进行排序。Ns中的DiffServ模型能够支持四类业务，而每一类又包含三种丢包优先级，以区别对待同一类中的不同的包。同一类业务中的包被排列成一个RED 队列，该队含有三种虚拟队列（每一个对应一个丢包序列）。

不同的RED参数被配置成虚拟队列，这使得来自一个虚拟队列的包比来自另外一个的包更易被丢失。在发生拥塞的时候，具有优先级较低的丢包序列的包被优先考虑，因为它被指定了一个代码点，该代码点对应具有RED参数的虚拟队列。

Ns中的DiffServ模型有主要有三个组成部分：

策略： 策略是由网络管理员声明的关于网络中每类业务应享有的服务级别。

边际路由器：边际路由根据所声明的策略用一个代码点来标记包。

核心路由器：核心路由器检查包的被标记的代码点并相应地改进他们。

DiffServ试图限制到仅相当于边际路由器的复杂程度。

9.2 实现

本部分中讲述的程序和函数能够在以下部分找到：~ns/diffserv/dsred,dsredq,dsEdge,dsCore,dsPolicy.{cc,h}.

9.2.1 DiffServ模型中的RED队列

一个派生于基类Queue 的DiffServ 队列(在类dsREDQueue 中)可以应用于DiffServ模型中，用以提供基本的DiffServ路由器功能。参考Dsred.{h,cc}。dsREDQueue 有以下功能：

- 在一个单一链路上应用多种物理的RED 队列；
- 在一个物理队列中应用多种虚拟队列，每一个虚拟队列有一套不同的参数；
- 根据包的密码点决定它应该在哪一个物理和虚拟的队列中排队；
- 根据选择的时序决定包应该从哪一个物理和虚拟的队列中排队；

dsREDQueue类包含有四种物理的RED队列，每一个队列又包含三种虚拟队列。物理和虚拟队列的数目在numPrec和numQueues_中定义。每一个物理和虚拟队列数目的组合都和一个代码点相关联，这个点唯一的标明了服务的级别。

物理队列在类redQueue中定义，这个类通过定义具有独立结构和状态参数的虚拟队列而可以区分不同业务，参考dsredq.{h,cc}。举例来说，每个虚拟队列的长度仅仅根据映射到那条队列的包来计算。因此，可以根据那条虚拟队列的状态和结构参数来决定这个包是否应该被丢弃。类redQueue并不等同于类REDQueue，后者已然存在于ns中了。然而，它是RED应用的一个改进的版本，并且包含虚拟队列的概念；它仅应用于类redQueue中以识别物理队列。用户的所有和redQueue之间的相互作用都通过类dsREDQueue 的命令界面来控制。

类dsREDQueue包含一个数据结构，在DiffServ中被称为Per Hop Behavior(PHB) Table，边际路由器用代码点来标识包，而核心路由器仅仅是响应存在的代码点；二者都用PHB表将代码点绘制成一个特有的物理和虚拟队列。PHB 表被定义为一个有三个区的指针：

```
struct phbParam {
int codePt_; // corresponding code point
int queue_; // physical queue
int prec_; // virtual queue (drop precedence)
};
```

9.2.2 边际和核心路由器（Edge and core routers）

DiffServ 的边际和核心路由器在类edgeQueue 和coreQueue 中定义，这两个类都派生于类dsREDQueue，参考dsEdge, dsCore. {h,cc}。包标记使用于类edgeQueue 中。在把包放到相应的物理和虚拟队列之前，根据所说的策略把包标记上代码点。类edgeQueue 有关于PolicyClassifier 类的实例的参考，后者包含包标记的策略。

9.2.3 策略

Policy类和它的子类（参考 dsPolicy.{cc,h}）定义了边际路由器标记到达包的策略，一个策略建立在源结点和目的结点之间。所有和某一对源结点、目的结点匹配的数据流都可以被看做是一个单一的业务整体。用于每一个不同业务整体的策略有一个相关联的policer类型、meter 类型和内部代码点。meter类型说明了policer 所需的状态变量的测量方法，比如，TSW Tagger 是用来测量平均传输速率的meter，用的是一种专门的时间窗口。

当一个包到达一个边际路由器的时候，首先检查它属于哪一个业务整体。 由相应的策略所描述的meter被激活以更新所有的状态变量。policer类型也被激活以根据业务整体的状态变量决定如何标记包：专有的内部代码点或者一个低级的代码点。然后包就相应的被排队。

目前，共定义了六种不同的策略模型：

- 带有2 种颜色标记的时间滑动窗口（TSW2CMPolicy）：应用一个CIR和两种丢包优先级。当CIR数量过多时，有可能使用较低的优先级。
- 带有3 种颜色标记的时间滑动窗口（TSW2CMPolicy）：应用一个CIR、一个PIR和三种丢包优先级。当CIR数量过多时，中间的丢包优先级可能先被使用，当PIR数量过多时，可能使用最低的优先级。
- 特征桶（tokenBucketpolicer）：应用一个CIR、一个CBS和两个丢包优先级。当且仅当到达的包比特征桶大时，它就会被标记上较低的优先级。
- 单一速率3 色标记（srTCMPolicer）：应用一个CIR、CBS和一个EBS从三种丢包优先级中选择。

- 双速率3 色标记（trTCMPolicer）：应用一个CIR、CBS、PIR和一个PBS从三种丢包优先级中选择。

上述策略被定义为一个dsPolicy 的子类。特定的meter 和policer 应用在函数applyMeter和applyPolicer中，这些函数在类dsPolicy中定义为虚拟函数。用户声明的策略可以通过简单的方式加载上去。请参考DumbPolicy中最简单的例子。

所有的policer都存贮在PolicyClassifier类的一个策略表中，这个表是一个指针类型的，它包括源结点和目的结点、一个policer 类型、一个meter 类型、一个内部代码点和各种状态信息，如下所示：
CIR和PIR的速率以bit/s计：
CIR：委托信息速率
PIR：最高信息速率
桶CBS、EBS和PBS以字节计：
CBS：委托脉冲大小
EBS：过量脉冲大小
PBS：高峰脉冲大小
C bucket：CBS的当前大小
E bucket：EBS的当前大小
P bucket：PBS的当前大小
最后一个包的到达时间
平均发送速率
TSW 窗口长度
在类PolicyClassiffier 中同样也包含一个策略表格，它存贮着从一个policer 类型和内部代码点到它的相关联的低级代码点。

9.3 配置

物理和虚拟队列的数目可以如下配置：

```
$dsredq set numQueues_ 1
$dsredq setNumPrec 2
```

在类dsREDQueue 中的变量numQueues_表明了物理队列的数目，它有一个默认值为4，在~ns/tc/lib/ns-default.tcl中定义并且可以象上面那样被改变。变量setNumPrec 设置了在一个物理队列中的虚拟队列的数目。

RED 参数可以象下面那样为每一个虚拟队列进行配置：

```
$dsredq configQ 0 1 10 20 0.10
```

平均包大小（以字节计）对平均RED 队列长度的计算来说也是所需的。

```
$dsredq meanPktSize 1500
```

可以配置用来计算包大小的不同MRED。

```
$dsredq setMREDMode RIO-C 0
```

上述命令设定了物理队列从0 到RIO-C 的MRED 模式，如果没有其他情况的话，所有的队列都将设成默认的RIO-C 模式。

在DiffServ 模型中所支持的各种MRED模式有：

- **RIO-C**（**RIO Coupled**）：丢弃一个out-of-profile包的可能性取决于所有虚拟队列的加权的平均长度；然而丢失一个in-profile 包的可能性仅仅取决于它自己的虚拟队列的加权平均长度。
- **RIO-D**（**RIO De-coupled**）:和RIO-C类似；除了丢失一个out-of-profile包的可能性取决于他的虚拟队列的大小
- **WRED**（**Weghted RED**）：所有的可能性都取决于一个单一队列的长度。

● **DROP**：和具有由RED最小门限规定的队列限度的丢尾队列一样，当队列大小达到了最小门限的时候，所有的包都不管它有什么样的标记，统统被丢弃。

下面的命令在PHB 表中增加了一条入口，并把代码点11连接到物理队列0和虚拟队列1。

$dsredq addPHBEntry 11 0 1

在ns 中，包被默认成一个零值的代码点，因此，用户为了执行最尽力的负荷，必须为零代码点增加一个PHB 入口。

此外，要有现成的命令允许用户在俩个物理队列之间选择时序模式。例如

$dsredq setSchedularMode WRR

$dsredq addQueueWeights 1 5

上面两行命令给加权的Round Robin 和1 到5 队列的权值设置了时序模式，其他可以支持的时序模式有Weighted Interleaved ound Robin(WIRR)、Round Robin(RR)和Priority(PRI)。默认的时序模式是Round Robin。

对于优先权时序，按优先权排成一列，其中队列0 有最高的优先权，同样，可以在限制一个特殊队列所能获得的最大带宽，如下：

$dsredq setSchedularMode PRI

$dsredq addQueueRate 0 5000000

这些命令表明队列0 所能获得的最大带宽是5Mb。

命令addPolicyentry 用来在策略表上增加一个入口，它根据所用的policer 策类型来采用不同的参数。命令名字后面的前两个参数往往是源结点和目的结点的ID，下一个参数是policer 类型，紧随其后的是以下policer 类型所需要的参数：

TSW2CM Initial code point CIR

TSW3CM Initial code point CIR PIR

TokenBucket Initial code point CIR CBS

srTCM Initial code point CIR CBS EBS

trTCM Initial code point CIR CBS PIR PBS

考虑一个Tcl 的脚本，其中$q 是一个边际路由器的变量，$s 和$d 分别表示源结点和目的结点。下面的命令为从源到目的的业务增加了一个TSW2CM 的策略。

$q addPolicyEntry [$s id] [$d id] TSW2CM 10 2000000

其他的参数可以代替“TSM2CM”而用于不同的策略：

TSW3CM 10 2000000 3000000

TokenBucket 10 2000000 10000

srTCM 10 2000000 10000 20000

trTCM 10 2000000 10000 3000000 10000

注意，仅有一个策略可以应用于一对源和目的之间。

下面的命令在策略表中增加了一个入口，表明trTCM 有内部的代码点10，低级的（黄色）代码点11 和更低级的（红色）的代码点12：

$dsredq addPolicerEntry trTCM 10 11 12

在每一个policer 类型和内部代码点的地方一定有一个策略表。

所支持的疑问：

策略表中的输出口，一次执行一行：
$dsredq printPolicyTable
策略表中的输出口，一次执行一行：
$dsredq printPolicerTable
PHB表中的输出口，一次执行一行：
$dsredq printPHBTable
包的统计结果：
$dsredq printStats
样本输出：
包的统计情况：
CP TotPkts TxPkts ldrops edrops
All 249126 249090 21 15
10 150305 150300 0 5
20 98821 98790 21 10
CP: 代码点
TotPkts: 接收到的包
TxPkts: 发送的包
ldrops: 因为链路拥塞而丢失的包
edrops: RED以前的丢失

返回RED的特定物理队列的加权平均大小：
$dsredq getAverage 0

返回当前C漏桶大小(字节)：
$dsredq getCBucket

9.4 命令一览

下面是一个用于模拟脚本的相关命令的列表：
$ns simplex-link $edge $core 10Mb 5ms dsRED/edge
$ns simplex-link $core $edge 10Mb 5ms dsRED/core

下面两个命令在边际路由器和核心路由器之前沿链路创建了一些队列：
set qEC [[$ns link $edge $core] queue]
Set DS RED parameters from Edge to Core:

$qEC meanPktSize $packetSize
$qEC set numQueues_ 1
$qEC setNumPrec 2
$qEC addPolicyEntry [$s1 id] [$dest id] TokenBucket 10 $cir0 $cbs0
$qEC addPolicyEntry [$s2 id] [$dest id] TokenBucket 10 $cir1 $cbs1
$qEC addPolicerEntry TokenBucket 10 11
$qEC addPHBEntry 10 0 0
$qEC addPHBEntry 11 0 1

```
$qEC configQ 0 0 20 40 0.02
$qEC configQ 0 1 10 20 0.10
```

这段代码得到了从边际路由器到核心路由器的DiffServ 队列的句柄，并为它配置了所有的参数。

为了精确计算RED 状态变量需要用到平均的Pkt 大小。物理队列和优先级的数目的设置是可选择的 ,但它可以提高效率。因为时序或者MRED 模式都没有设置，故他们默认成为Round Robin 和RIO-C 活动队列管理。

命令addPolicyEntry 在边际队列里建立两个策略：一个届于结点S1 和目的结点之间；另一个届于结点S2 和目的结点之间。注意到[$s1 id]这条命令返回的是addPolicyEntry 所需要的ID 值，在这些策略中所需要的CIR 和CBS 值是在脚本开始时设置的那些。

AddPolicyEntry 这一行是必需的，因为每一对策略类型和内部代码点都需要策略表中的一个入口。每一个策略用的是同一个策略和内部代码点，所有仅需要一个入口。

AddPHBEntry 这条命令将任一个代码点映射到一个物理和虚拟队列的组合上去。尽管这个例子中的每个代码点被映射到了一个独特的物理和虚拟队列的组合上，但多个代码点可以受到同样的待遇。

最后，configQ 命令为每个虚拟队列设置了RED 参数。注意到随着优先级数值的增加，RED 参数也会变的不精确。

```
set qCE [[$ns link $core $e1] queue]
# Set DS RED parameters from Core to Edge:
$qCE meanPktSize $packetSize
$qCE set numQueues_ 1
$qCE setNumPrec 2
$qCE addPHBEntry 10 0 0
$qCE addPHBEntry 11 0 1
$qCE configQ 0 0 20 40 0.02
$qCE configQ 0 1 10 20 0.10
```

注意到一个核心队列的配置匹配一个边际队列的配置，除非核心路由器中没有策略表和策略类型表加以配置。一个核心路由器的主要功能是它可以为它看见的所有代码点提供一个入口。

```
$qE1C printPolicyTable
$qCE2 printCoreStats
```

这些方法输出了链路上的策略表和策略类型表以及不同的统计数字。

为了得到进一步的信息，请参考~ns/tcl/ex/diffserv 下的脚本示例。

第 10 章

代理（Agents）

Agent代表了网络层的分组的起点和终点，并被用于各层协议的实现。Agent 类是由OTcl 和C++共同实现的。C++部分的代码包含在~ns/agent.cc和~ns/agent.h中，而OTcl 部分的代码则位于~ns/tcl/lib/ns-agent.tcl。

10.1 代理声明(state)

对于一个被模拟的分组，在分组发送前，C++的Agent类包含足够多的内部状态变量来表示分组的各个字段。这些状态变量包括：

addr_ 本节点的地址（分组的本地节点地址）
dst_ 分组的目标节点地址
size_ 分组大小，单位为字节（放入公共分组头中）
type_ 分组类型（在通用分组头中，参考packet.h）
fid_ IP流标志（以前在ns-1中分类）
prio_ IP优先级
flags_ 分组的标记（与ns-1类似）
defttl _ IP缺省的ttl值
这些变量可由任意Agent类的派生类来修改，但是一个特定的Agent 可能不会用到所有的状态变量。

10.2 代理函数

Agent类支持分组的产生和接收。下面这些成员函数是在C++的Agent类中实现的，并且通常不会被派生类重载：
Packet* allocpt () 生成一个新的分组并给其分配字段
Packet* allocpkt (int) 生成一个新的分组，该分组的数据部分长度为n字节，并给其分配字段

下面的成员函数也是由Agent类定义的，但它们需要被派生类重载：
void timeout (timeout number) 特定子类（subclass-specific）超时方法
void recv (Packet*,Handler*) 接收代理主接收路径

派生类用allocpkt()方法来产生要发送的分组。这个函数填写分组的通用头(第12小节)中的以下几个字段 :uid ,ptype，size，以及IP 报头中的以下几个字段：src，dst，flowid，prio，ttl。它还会把Flags 头中的以下几个字段置零：ecn，pri，usr1，usr2。任何未包含在这里面的分组头信息必须由Agent类的派生类自己来处理。

recv()方法是Agent在接收分组时的主入口，且上游节点（upstream nodes）在发送一个分组时会调用目标节点的相应的Agent的recv()函数。在大多数情况下，Agent并不使用recv()函数的第2个参数（由上游节点所定义的处理）。

10.3 协议代理

模拟器支持许多协议代理。以下是它们在OTcl 中的名字：

TCP “Tahoe” TCP 发送端（任何损耗下cwnd = 1）
TCP/Reno “Reno” TCP 发送端（快速修复）
TCP/Newreno 改进的Reno TCP 发送端（改进了快速修复）
TCP/Sack1 SACK TCP 发送端
TCP/Fack “转发“SACK发送端TCP
TCP/FullTcp 功能更全的双工TCP
TCP/Vegas “Vegas” TCP发送端
TCP/Vegas/RBP 具有“步进速率（rate based pacing）” 的Vegas TCP
TCP/Reno/RBP 具有“步进速率” 的Reno TCP
TCP/Asym 非对称链路的实验的Tahoe TCP
TCP/Reno/Asym 非对称链路的实验的Reno TCP
TCP/Newreno/Asym 非对称链路的实验的Newreno TCP
TCPSink Reno或Tahoe TCP接收器（不用于FullTcp）
TCPSink/DelAck 延时ACK的TCP接收器
TCPSink/Asym 非对称链路的实验的TCP接收端
TCPSink/Sack1 SACK TCP接收器
TCPSink/Sack1/DelAck 延时ACK的SACK TCP接收器
UDP 基本的UDP代理
RTP RTP发送端和接收端
RTCP RTCP发送端和接收端
LossMonitor 检测丢失的分组接收器
IVS/Source IVS源
IVS/Receiver IVS接收器
CtrMcast/Encap “集中式多播” 封装器
CtrMcast/Decap “集中式多播” 解封装器
Message 携带文本消息的协议
Message/Prune 处理多播路由剪除消息
SRM 有非适应性定时器SRM代理
SRM/Adaptive 有适应性定时器的SRM代理
Tap 模拟器到真实网络的接口
Null 丢弃分组的衰弱代理
rtProto/DV 距离矢量路由协议代理

代理用于各层协议的实现。因此，对于一些传输层的协议（如UDP），分组的大小和发送时间通常是由代表应用层的独立的对象来控制的，这些控制是通过Agent扩展应用层程序开发的应用程序接口（API）来进行的。对于在低层使用的代理（比如路由代理），分组的大小和发送时间通常由代理自己的协议消息过程控制。

10.4 OTcl 链接

代理可以在OTcl中创建，并通过Tcl的set函数或者修改代理自己（或其基类）的Tcl函数来设定代理的内部状态变量。注：有些Agent 的内部状态变量可能只存在于OTcl中，因此也就不能从C++中直接访问它们。

10.4.1 创建并操作代理

下面这个例子说明如何创建一个Agent 并改变其内部状态变量：

```
set newtcp [new Agent/TCP]          ;# 创建新的对象(还有C++的影子对象)
$newtcp set window_ 20              ;#设定tcp代理的窗口值为20
$newtcp target $dest                ;#target在Connect类中实现
$newtcp set portID_ 1               ;# 只在OTcl中出现，C++中没有
```

10.4.2 缺省值

那些在OTcl中可见的变量（包括仅存在于OTcl中的变量和通过bind命令在OTcl和C++间绑定的变量）是在~ns/tcl/lib/ns-default.tcl中初始化的。代理的初始化是如下：

```
Agent set fid_ 0
Agent set prio_ 0
Agent set addr_ 0
Agent set dst_ 0
Agent set flags_ 0
```

通常，在任何一个Agent类的对象创建前，这些初始化就存在于OTcl的名字空间中了。因此，当一个Agent对象创建时，对象的构造函数中的bind 命令会把C++中相应的成员变量设定为指定的缺省值。

10.4.3 OTcl 函数

Agent类在OTcl中的实例过程当前可以在源文件~ns/tcl/lib/ns-agent.tcl中找到。它们是：

```
Port        agent的端口标志符
Dst-port    目标节点的端口标志符
Attach-source <stype>       创建并为agent绑定一个源对象
```

10.5 TCP，TCP接收代理的例子

TCP类描绘了一个简化的TCP发送端。它将数据发给一个TCPSink代理并处理返回的确认消息。它有一个与之关联的独立的对象，代表了应用层的需求。通过观察TCPAgent类和TCPSinkAgent类，我们可以了解如何构造相对复杂的agent。为阐述定时器的使用方法，文中同时也给出了一个来自Tahoe TCP代理TCPAgent 的例子。

10.5.1 创建Agent

以下的OTcl 代码片断创建并设置了一个TCP代理：

```
set tcp [new Agent/TCP]                 ;# 创建发送端代理
$tcp set fid_ 2                         ;# 设定IP层的流标识
set sink [new Agent/TCPSink]            ;# 创建接收端代理
```

```
$ns attach-agent $n0 $tcp                    ;# 将发送端放在节点$n0上
$ns attach-agent $n3 $sink                   ;# 将接收端放在节点$n3上
$ns connect $tcp $sink                       ;# 建立TCP连接
set ftp [new Application/FTP]                ;# 创建一个FTP源"应用"
$ftp attach-agent $tcp                       ;# 将FTP与TCP发送端关联起来
$ns at 1.2 "$ftp start"                      ;# 安排FTP在1.2s时启动
```

OTcl的new Agent/TCP命令创建了一个C++的TcpAgent对象。TcpAgent对象的构造函数首先调用了Agent类的构造函数，然后执行自己的构造函数来完成变量的绑定工作。下面是TcpAgent类构造函数的代码：

TcpSimpleAgent构造函数（~ns/tcp.cc）：

```
TcpAgent::TcpAgent() : Agent(PT_TCP), rtt_active_(0), rtt_seq_(-1),
rtx_timer_(this), delsnd_timer_(this)
{
bind("window_", &wnd_);
bind("windowInit_", &wnd_init_);
bind("windowOption_", &wnd_option_);
bind("windowConstant_", &wnd_const_);
...
bind("off_ip_", &off_ip_);
bind("off_tcp_", &off_tcp_);
...
}
```

Agent 构造函数（~ns/agent.cc）：

```
Agent::Agent(int pkttype) :
addr_(-1), dst_(-1), size_(0), type_(pkttype), fid_(-1),
prio_(-1), flags_(0)
{
memset(pending_, 0, sizeof(pending_));                    /* 定时器 */
// 这是一个真实的IP代理，被创建用来产生合适的IP字段... ...
bind("addr_", (int*)&addr_);
bind("dst_", (int*)&dst_);
bind("fid_", (int*)&fid_);
bind("prio_", (int*)&prio_);
bind("flags_", (int*)&flags_);
...
}
```

这些代码片断描述了通用的情形，即TcpAgent的构造函数会把分组类型标志符传递给Agent 类的构造函数。各种不同的分组类型会在使用trace功能（第26.5小节）时被用到，且是在~ns/trace.h文件中定义的。这些绑定在TcpAgent构造函数中的变量都是普通的类的实例/成员变量，除了特殊的整型值off_tcp_和off_ip_。为了能分别访问TCP报头和IP报头，这些变量都是必须的。更多的细节见关于分组头的章节（第12.1节）。

注：TcpAgent构造函数包括了对两个定时器rtx_timer_和delsnd_timer_的初始化。TimerHandler对象是通过给相关的agent 提供一个指针（this指针）来初始化的。

10.5.2 启动代理

在1.2秒时FTP数据源启动，TcpAgent也同时启动。Start操作是Applicaion/FTP 类的一个实例过程（请参考~ns/tcl/lib/ns-source.tcl）：

```
Application/FTP instproc start {} {
[$self agent] send -1
}
```

在这里，“agent”指向我们简易的TCP代理，而send -1表示发送一个任意大的文件。

Send的调用最终将在简易的TCP发送端产生分组。下面的output函数完成发送分组的工作：

```
void TcpAgent::output(int seqno, int reason)
{
Packet* p = allocpkt();
hdr_tcp *tcph = (hdr_tcp*)p->access(off_tcp_);
double now = Scheduler::instance().clock();
tcph->seqno() = seqno;
tcph->ts() = now;
tcph->reason() = reason;
Connector::send(p, 0);
...
if (!(rtx_timer_.status() == TIMER_PENDING))
/*没有定时器被挂起。调度一个计时器*/
set_rtx_timer();
}
```

这里我们看到了对Agent::allocpkt()函数使用的一个说明。Allocpkt()首先生成一个新的分组（分组的公共报头和IP报头已经填好），但TCP层报头字段的信息还需要填写。为了发现分组中的TCP头（假设分组已经激活（12.2.4 节）），必须合理初始化off_tcp_，如构造函数中所描述的那样。分组的access()函数返回一个指向TCP头的指针，填入序列号和时间戳，并调用Connector类的send()函数以发送分组到下一跳。注：这里使用C++::范围运算以避免调用TcpSimpleAgent::send()（亦已定义）。检查挂起的定时器，可以使用TimerHandler基类定义的status()函数。这里用status()来设置一个重传定时器，如果该定时器尚未被设置（每个连接的分组的每个窗口，TCP头只设置一个定时器）。

10.5.3 在接收端处理输入

许多TCP代理可以和TCPSink类成对使用。该类定义了如下的recv()和ack()函数：

```
void TcpSink::recv(Packet* pkt, Handler*)
{
hdr_tcp *th = (hdr_tcp*)pkt->access(off_tcp_);
87
acker_->update(th->seqno());
ack(pkt);
Packet::free(pkt);
}
void TcpSink::ack(Packet* opkt)
{
```

```
Packet* npkt = allocpkt();
hdr_tcp *otcp = (hdr_tcp*)opkt->access(off_tcp_);
hdr_tcp *ntcp = (hdr_tcp*)npkt->access(off_tcp_);
ntcp->seqno() = acker_->Seqno();
ntcp->ts() = otcp->ts();
hdr_ip* oip = (hdr_ip*)opkt->access(off_ip_);
hdr_ip* nip = (hdr_ip*)npkt->access(off_ip_);
nip->flowid() = oip->flowid();
hdr_flags* of = (hdr_flags*)opkt->access(off_flags_);
hdr_flags* nf = (hdr_flags*)npkt->access(off_flags_);
nf->ecn_ = of->ecn_;
acker_->append_ack((hdr_cmn*)npkt->acc ess(o ff_cmn _),
ntcp, otcp->seqno());
send(npkt, 0);
}
```

recv()函数重载了Agent::recv()函数（只不过丢弃了接收到的分组）。它用接收分组的序列号更新了一些内部声明，因此需要合理初始化off_tcp_变量。接着生成该接收分组的确认信息。ack()函数中大量访问了分组头的各个字段，包括TCP头、IP 头、Flags头及common头。对send()的调用相当于调用了Connector::send()函数。

10.5.4 在发送端处理响应（Responses）

一旦简易的TCP对等端接收到数据并返回ACK，发送者（通常）必须处理ACK。在TcpAgent代理中是这样处理的：

```
/*
* 主接收路径 - 应该只可以看到acks，否则说明网络连接配置错误。
*/
void TcpAgent::recv(Packet *pkt, Handler*)
{
hdr_tcp *tcph = (hdr_tcp*)pkt->access(off_tcp_);
hdr_ip* iph = (hdr_ip*)pkt->access(off_ip_);
...
if (((hdr_flags*)pkt->access(off_flags _))-> ecn_)
quench(1);
if (tcph->seqno() > last_ack_) {
newack(pkt);
opencwnd();
} else if (tcph->seqno() == last_ack_) {
if (++dupacks_ == NUMDUPACKS) {
...
}
}
Packet::free(pkt);
send(0, 0, maxburst_);
}
```

当ACK分组到达后，recv()函数被调用。这样情况下，一旦ACK中的信息被处理完（通过newack），分组就不再需要并将被返回到分组内存分配器。此外，收到ACK意味着可以发送更多的数据，因此调用TcpSimpleAgent::send()函数，在TCP窗口允许的范围内发送更多的数据。

10.5.5 定时器的实现

如下一章（第11章）所述，特定的定时器类必须从一个抽象的基类TimerHandler类继承而来，定义见于~ns/timer-handler.h。于是这些子类的实例可用于各种代理定时器。一个代理可能希望重载Agent::timeout()方法（该方法什么也不做）。在Tahoe TCP代理的情况下，采用两种定时器：延时发送定时器delsnd_timer_和重传定时器rtx_timer_。我们将在TCP（第11.1.2 节）中以一个定时器应用的实例来描述重传定时器。

10.6 创建一个新的代理

创建一个新的代理，需完成以下操作：
1、确定其继承结构（第10.6.1节），并建立合适的类定义，
2、定义recv()和timeout()方法(第10.6.2节)，
3、定义任何必须的定时器类，
4、定义OTcl链接函数（第10.6.3节），
5、编写必要的OTcl代码以访问代理（第10.6.4节）。

创建代理所需的操作可由一个非常简单的例子来说明。假设我们希望构造一个执行ICMP ECHO REQUEST/REPLY（或“ping”）操作的代理。

10.6.1 示例：一个“ping”请求器（继承结构）

确定继承的结构是一个个人选择的问题，但它可能与代理操作所在的层及其在低层函数的假设有关。最简单的Agent类型（面向数据的无连接传输）是Agent/UDP基类。流量发生器可方便地连接到UDP Agent上。对于希望采用面向连接的流传输（如TCP）的协议，可以使用不同的TCP Agent。最后，如果要开发一个新的传输或“子传输”协议，采用Agent作为基类可能是最佳的选择。在我们的例子中，我们将采用Agent类作为基类，假设我们正构造一个逻辑上属于IP层（或其上层）的代理。

我们可以采用如下的类定义：

```
class ECHO_Timer;
class ECHO_Agent : public Agent {
public:
ECHO_Agent();
int command(int argc, const char*const* argv);
protected:
void timeout(int);
void sendit();
double interval_;
ECHO_Timer echo_timer_;
```

```
};
class ECHO_Timer : public TimerHandler {
public:
ECHO_Timer(ECHO_Agent *a) : TimerHandler() { a_ = a; }
protected:
virtual void expire(Event *e);
ECHO_Agent *a_;
};
```

10.6.2 recv()和timeout()方法

这里不定义recv()方法，因为这个agent代表一个请求函数，并且通常不接收事件或分组[12]。由于不定义recv()方法，因此采用recv()的基类(例如Connector::recv())。timeout()函数被用于周期地发送请求分组。所用的timeout()函数如下所示，它具有一个辅助函数sendit()：

```
void ECHO_Agent::timeout(int)
{
sendit();
echo_timer_.resched(interval_);
}
void ECHO_Agent::sendit()
{
Packet* p = allocpkt();
ECHOHeader *eh = ECHOHeader::access(p->bits());
eh->timestamp() = Scheduler::instance().clock();
send(p, 0); // Connector::send()
}
void ECHO_Timer::expire(Event *e)
{
a_->timeout(0);
}
```

timeout()方法仅安排sendit()在每一个interval_时间段间执行。sendit()方法产生一个新的分组，分组头的大多数字段已经由allocpkt()设置好。只不过分组没有当前的时间戳。对access()的调用提供了一个到分组头的结构接口，并可用来设置时间戳。注：该代理使用本身特定的头（ “ECHOHeader” ）。分组头的创建和使用在后续章节（第12章）介绍；为发送分组给下游节点（downstream node），调用了无句柄的Connector::send()函数。

10.6.3 “ping” 代理与OTcl的链接

前面（第3章）我们有建立OTcl链接的函数和机制。本节是前面章节的基本性质的一个简短回顾，并描述创建ping代理所需的最少函数。

[12] 这可能过于简单，不切实际。我们希望一个 ICMP ECHO REQUEST 可以处理 ECHO REPLY 消息。

将我们的agent链接到OTcl上，我们必须合理处理3个问题。首先，当OTcl中需要类的实例时，我们需要在该类的OTcl和所创建的真实对象之间建立一个映射。完成方法如下所示：

```
static class ECHOClass : public TclClass {
public:
ECHOClass() : TclClass("Agent/ECHO") {}
TclObject* create(int argc, const char*const* argv) {
return (new ECHO_Agent());
}
} class_echo;
```

这里创建了一个static 对象“class_echo”。是构造器（当模拟器执行时，它也立即执行）将类名“Agent/ECHO”放如OTcl名字空间的。这种混合的情况是按照惯例的；回忆以下前面的3.5 节：“/”符号是一个用于解释层的层分隔符。create()方法定义了当OTcl解释器接收指令以创建一个“Agent/ECHO”类的对象时，如何创建一个C++影子对象。在这种情况下，将返回一个动态分配的对象。这是创建新的C++影子对象的通常办法。

一旦我们对象创建了，我们就想把C++成员变量链接到OTcl名字空间相应的变量上，这样对于OTcl变量的访问实际上返回的是C++的成员变量。假设我们希望OTcl可以调整发送间隔和分组大小。在类构造器中是这样完成的：

```
ECHO_Agent::ECHO_Agent() : Agent(PT_ECHO)
{
bind_time("interval_", &interval_);
bind("packetSize_", &size_);
}
```

这里C++变量interval_和size_分别被链接到OTcl实例变量interval_和pachetSize_上。任何对于Otcl变量的读入或修改将导致对于其下的C++变量的相应的访问。bind()函数的细节参考别处（第3.4.2节）。已定义的PT_ECHO常量被传递到Agent()构造器，这样Agent::allocpkt()方法可以设置用于支持trace的（第26.5节）分组类型。在这种情况下，PT_ECHO表示一个新的分组类型且必须在~ns/trach.h（第26.4节）中定义。

一旦对象创建和变量绑定设置完成，我们可能希望创建在C++中实现但可从OTcl调用（第3.4.4小节）的函数。这些函数常常是些控制函数，具有初始化、终止或修改的行为。在我们给出的例子中，我们希望通过一个“start”命令，可以从OTcl开始ping查询代理。这可由以下代码实现：

```
int ECHO_Agent::command(int argc, const char*const* argv)
{
if (argc == 2) {
if (strcmp(argv[1], "start") == 0) {
timeout(0);
return (TCL_OK);
}
}
return (Agent::command(argc, argv));
}
```

这里，用于OTcl的start()函数只是调用了C++的成员函数timeout()，timeout()函数初始化开始产生的分组并调度后面产生的分组。注：该类过于简单，甚至没有包括停止这样的操作。

10.6.4 通过OTcl使用代理

我们已经创建的agent 需要示例并绑定到一个节点上。注：假设节点或模拟器对象已经创建。以下OTcl 代码实现这些功能：

```
set echoagent [new Agent/ECHO]
$simulator attach-agent $node $echoagent
```

为了设置发送间隔和分组大小并开始产生分组，可执行下面的OTcl代码：

```
$echoagent set dst_ $dest
$echoagent set fid_ 0
$echoagent set prio_ 0
$echoagent set flags_ 0
$echoagent set interval_ 1.5
$echoagent set packetSize_ 1024
$echoagent start
```

每隔1.5 秒，这些代码将产生一个去往节点$dest的1024比特的分组。

10.7 代理应用程序接口（API）

模拟应用可以在协议代理的顶端来实现。第38章描述了用于访问协议代理所提供的服务所用到的API。

10.8 各种代理对象

Agent类形成了基类，各种对象类型如Nullobject、TCP等均继承了Agent类。用于Agent 类的方法将在下节描述。Agent类的配置参数如下：

fid_ Flowid
prio_ Priority
agent_addr_ 代理的地址
agent_port_ 代理的端口地址
dst_addr_ 代理的目标地址
dst_port_ 代理的目标端口地址
flags_
ttl_ TTL的缺省值32

对于一般的agent类，不定义状态变量。其它继承了Agent类的对象有：

Null Objects Null对象是agent对象的一个子类，用于实现流量的接收。它们继承了所有的通用对象函数。该对象中不定义任何方法。其状态变量有：

- sport_
- dport_

LossMonitor Objects LossMonitor对象是agent对象的一个子类，用于实现流量的接收，同时也对接收数据的进行一些统计，如接收的比特数、丢失的分组数等。它们继承了所有的通用agent对象函数。

$lossmonitor clear
重置期望的序列号为-1。
其状态变量有：
nlost_ 丢失分组数。
npkts_ 接收分组数。
bytes_ 接受比特数。
lastPktTime_ 接收最后一个分组的时间。
expected_ 下一个分组的期望序列号。
TCP objects TCP对象是agent对象的一个子类，用于实现BSD Tahoe TCP 传输协议，该协议可参考文献：“Fall, K., and Floyd, S. Comparisons of Tahoe, Reno, and Sack TCP.December 1995.”URL ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z.
它们继承了通用agent类的所有函数。配置参数有：
window_ TCP连接的传输窗上限。
maxcwnd_ TCP链接拥塞窗上限。忽略可设置为零（缺省值）。
windowInit_ slow-start上拥塞窗的初始尺寸。
windowOption_ 用于管理拥塞窗的算法。
windowThresh_ 对于各种window-increase 算法的研究，用于计算awnd（见下面）的指数均衡滤波器增益常量。
overhead_ 用于每个输出分组延时的均匀分布随机变量的取值范围。在源节点插入随机延时是为了防止相位的影响，当期望得到更好时[参考 Floyd,S., and Jacobson, V. On Traf_c Phase Effects in Packet-Switched Gateways.Internetworking: Research and Experience, V.3 N.3, September 1992.pp.115-156]。这里仅实现了tcp的Tahoe（“tcp”），而不是tcp-reno。这并不意味着它是一个CPU处理开销的实际模型。
ecn_ 除了分组丢失外，还正确设置为使用清楚的拥塞提示。在quench()后，允许由ECN比特引起的快速重传。
packetSize_ 用于所有来自源的分组的尺寸，以字节计。
tcpTick_ 用于测量roundtrip时间的TCP时钟间隔。注意缺省设置为非标准值100ms。
bugFix_ 对于在单一数据窗中的分组丢失，当允许多个快速重发，设置为排除故障。
maxburst_ 忽略该项可设为零。此外，源可发送的响应单一ACK的最大分组数。
slow_start_restart_ 对于slow-start，在连接空闲后可设为1。缺省值为On。
定义的常量有：
MWS TCP连接的分组最大窗尺寸。MWS决定了tcpsink.cc中排列的尺寸。缺省的MWS值为1024个分组。对于Tahoe TCP，“window”参数（表示接收端传输窗）可以小于MWS-1。对于Reno TCP，“window”参数可以小于(MWS-1)/2。
状态变量为：
dupacks_ 从任何新的数据被确认接收后可见的acks副本的数目。
seqno_ 从数据源到TCP的数据最大序列号。
t_seqno_ 当前传送序列号。
ack_ 接受端可见的最高确认消息。cwnd_拥塞窗的当前值。
awnd_ 拥塞窗的低通滤波器当前值。用于研究不同的窗口递增算法。
ssthresh_ slow-start门限的当前值
rrt_ round-trip时间估计
srtt_ 平滑round-trip时间估计
rttvar_ round-trip时间均方误差估计
backoff_ round-trip指数退避时间常数

TCP/Reno Objects TCP/Reno对象是TCP对象的子类，用于实现Reno TCP传输协议，协议参考：“Fall, K., and Floyd, S. Comparisons of Tahoe, Reno, and Sack TCP. December1995.” URL ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z 没有为

该对象定义函数、配置参数或状态变量。

TCP/Newreno Objects TCP/Newreno对象是TCP对象的一个子类，实现了对BSD Reno TCP传输协议的一个改进。没有为该对象定义函数或状态变量。
配置参数为：
Newreno_changes_ 对于“Fall, K., and Floyd, S. Comparisons of Tahoe,Reno,and Sack TCP. December 1995”中所描述的New Reno，缺省设为零。对于附加的New Reno算法，设为1。[参考Hoe, J., Improving the Start-up Behaviorof a Congestion Control Scheme for TCP. in SIGCOMM 96, August 1996, pp. 270-280.URLhttp://www.acm.org/sigcomm/sigcomm96/papers/hoe.html.]；该参数包括了slow-start期间对ssthresh参数的估计。

TCP/Vegas Objects 该对象没有定义函数或配置参数。状态变量为：

- v_alpha_
- v_beta_
- v_gamma_
- v_rtt_

TCP/Sack1 Objects TCP/Sack1对象是TCP对象的一个子类，用于实现“Fall, K., and Floyd, S.Comparisons of Tahoe, Reno, and Sack TCP. December 1995”所描述的带选择确认的扩展BSD Reno TCP传输协议。
URLftp://ftp.ee.lbl.gov/papers/sacks.ps.Z. 它们继承了TCP对象的功能。该对象不定义函数、配置参数或状态变量。

TCP/FACK Objects 该对象是TCP对象的一个子类，用于实现带转发确认拥塞控制的BSD Reno TCP传输协议。它们继承了TCP对象的功能。该类不定义函数或状态变量。
配置参数为：
ss-div4 强衰减算法。如果在slow-start的1/2 RTT内检测到拥塞，则将ssthresh一分为四（而不是二）。（1=Enable,0=Disable）
rampdown Rampdown数据平滑算法。慢慢减少拥塞窗而不是快速地将它一分为二。（1=Enable,0=Disable）

TCP/FULLTCP Objects 此处尚未添加该节。实现和配置参数参考“Fall, K., Floyd,S., and Henderson, T., Ns Simulator Tests for Reno FullTCP. July, 1997.” URL ftp://ftp.ee.lbl.gov/papers/fulltcp.ps.

TCPSINK Objects 该对象是agent对象的一个子类，用于实现对TCP分组的接收。模拟器只实现“单径”TCP连接，这里TCP源发送数据分组而TCP接收端发送ACK分组。TCPSink对象继承了一般代理的所有功能。该对象不定义函数或状态变量。
配置参数为：
packetSize_ 所有ack分组使用的尺寸，以字节计。
maxSackBlocks_ 在SACK选项中被确认的数据最大阻塞数。对于一个也使用时间戳选项[RFC 1323]的接受端，RFC 2018所定义SACK选项可容纳3个SACK阻塞。它仅用于TCPSink/Sack1子类。在对象分配完毕后，该值在任何特殊的TCPSink对象内可能无法增加。（一旦TCPSink对象配置好，该参数的值可能只会下降不会增加）。

TCPSINK/DELACK Objects DelAck对象是TCPSink对象的一个子类，用于一个TCP分组的延时ACK接收器。它们继承了所有TCPSink对象的功能。DelAck对象不定义函数或状态变量。配置参数为：
interval_ 对于单一分组产生一个ack之前的总延时。如果另外一个分组在延时期满前到达，则立刻生成一个ack。

TCPSINK/SACK1 Objects 该对象是TCPSink的一个子类，用于实现一个TCP分组的SACK接收器。它们继承了所有TCPSink对象的功能。该对象不定义函数、配置参数或状态变量。

TCPSINK/SACK1/DELACK Objects 该对象是TCPSink/Sack1的子类，用于实现一个TCP分组的延时SACK接收器。它们继承了TCPSink/Sack1对象的所有功能。该对象不定义函数或状态变量。配置参数为：
interval_ 产生一个单一分组的ack之前的总延时。如果在延时期满之前，有另外一个分组到达，则立刻产生一个ack。

10.9 命令一览

以下是在模拟脚本中使用的与agent相关的命令：
ns_ attach-agent <node> <agent>
该命令在<node>上捆绑了一个<agnet>。这里我们假设<agent>已经创建。创建agent可用set agent [new Agent/AgentType],这里Agent/AgentType定义了agent的类型。

$agent port
该命令将返回被捆绑的agent的端口号。

$agent dst-port
该命令返回目标端口号。在两个节点间的任何连接设置时，每一个agent在其dst_port_实例变量中存储一个目标端口号。

$agent attach-app <s_type>
该命令在agent上捆绑一个<s_type>类型的应用。返回一个访问该应用对象的句柄。注意：必须在pachet.h中定义该应用类型为一个分组类型。

$agent attach-source <s_type>
该命令用于在agent上捆绑<s_type>类型源的过程。但是现在这已经是个陈旧的办法了。用attach-app(上面有描述)方法来替代。

$agent attach-tbf <tbf>
在agent上捆绑一个token bucket 滤波器（tbf）

$ns_ connect <src> <dst>
在src和dst代理间建立一条连接。

$ns_ create-connection <srctype> <src> <dsttype> <dst> <pktclass>
在两个agent间建立一条完整的连接。首先创建一个<srctype>类源并将它捆绑到<scr>上。然后创建一个<dsttype>类型的目标并将它捆绑到<dst>上。最后连接scr和dst代理并返回一个句柄给源agent。

$ns_ create-connection-list <srctype> <src> <dsttype> <dst> <pktclass>
该命令与上面所述的闯将连接非常相似。但不同与仅返回给源agent，它返回一个列表给源和目标agent。

内部过程：
$ns_ simplex-connect <src> <dst>

这是一个内部函数，它争取设置了一个<src>agent和<dst>agent间的单向连接。它仅设置了<src>的目标地址和目标端口及<dst>的agent-address和agent-port。上面所述的“connect”调用了该函数两次并设置了一个src和dst之间的双向连接。

$agent set <args>

这是一个用于通知用户由于升级到32比特地址空间后所造成的向后兼容问题的内部函数。

$agent attach-trace <file>

该过程将<file>捆绑到agent上并允许agent的nam-tracing时间。除了以上所述的与agent相关的过程外，还有一些附加的函数用来支持各种agent，如Agent/Null、Agent/TCP、Agent/CBR、Agent/TORA、Agent/mcast等。这里所述的这些附加的函数及过程可在ns/tcl/lib/(ns-agent.tcl、ns-lib.tcl、ns-mip.tcl、ns-mobilenode.tcl、ns-namsupp.tcl、ns-queue.tcl、ns-route.tcl、ns-sat.tcl、ns-source.tcl)中找到。它们也已在前一章节描述过了。

第 11 章

定时器（Timers）

定时器既可以在C++中实现，也可以在OTcl中实现。在C＋＋中，定时器是基于定义在~ns/timer-handler.h中的一个抽类基类。它们大多用于代理对象，但是其结构的一般性使其可以用于其它的对象。下面的讨论即源于定时器在代理中的应用。

本章中所描述的过程和函数均可在~ns/tcl/ex/timer.tcl和~ns/timer-handler.{cc,h}中找到。

在OTcl中，~ns/tcl/timer.tcl中定义了一个简易的定时器。其继承而来的子类提供了在OTcl层调度事件的简单机制。

11.1 C＋＋抽象基类TimerHandler

抽象基类TimerHandler包含下面的公有成员函数：

void sched(double delay)　调度定时器用来终止将来的延迟

void resched(double delay)　重新调度定时器（类似与sched()），但定时器可能被挂起

void cancel()　取消一个挂起的定时器

int status()　返回定时器状态（TIMER_IDLE, TIMER_PENDING, 或者TIMER_HANDLING)

抽象基类TimerHandler包含下面的私有成员：

virtual void expire(Event* e) =0　该方法必须填入定时器客户端

virtual void handle(Event* e)　消耗一个事件; 调用expire()函数并适当设定定时器的status_值

int status_　跟踪定时器的当前状态

Event event_　在定时器终止后用于消耗的事件

纯虚函数expire()必须由这个抽象基类的派生定时器类来定义。

最后，定义了两个私有的函数：

```
inline void _sched(double delay) {
(void)Scheduler::instance().schedule(this, &event_, delay);
}
inline void _cancel() {
(void)Scheduler::instance().cancel(&event_);
}
```

从这段代码中，我们可以看出定时器使用了Scheduler类中的方法。

11.1.1 定义一个新的定时器(timer)

要定义一个新的timer,subclass 该函数并在必要时定义handle()(handle()并不总是需要)。

```
class MyTimer : public TimerHandler {
public:
MyTimer(MyAgentClass *a) : TimerHandler() { a_ = a; }
virtual double expire(Event *e);
protected:
MyAgentClass *a_;
};
```

然后定义expire:

```
double
MyTimer::expire(Event *e)
{
// do the work
// return TIMER_HANDLED; // => do not reschedule timer
// return delay; // => reschedule timer after delay
}
```

注：expire()既可以返回标志(flag)TIMER_HANDLED 又可返回时延值,这取决于对这个timer的要求。

通常MyTimer将是MyAgentClass的友元,或expire()将只调用MyAgentClass的一个公共函数。用户不能直接从OTcl层上访问Timers，尽管用户可以根据自己的意愿建立方法绑定。

11.1.2 示例：Tcp重传定时器

TCP是一个需要timers的Agent的例子。在tcp.cc中定义的基本Tahoe Tcp agent中定义了三个定时器：

```
rtx_timer_;                                   /* Retransmission timer */
delsnd_timer_;          /* Delays sending of packets by a small random amount of time, */
                                              /* to avoid phase effects */
burstsnd_timer_;        /* Helps TCP to stagger the transmission of a large window */
                                  /* into several smaller bursts */
```

在~ns/tcp.h中，三个类都是由基类TimerHandler派生出来的:

```
class RtxTimer : public TimerHandler {
public:
RtxTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
virtual void expire(Event *e);
TcpAgent *a_;
};
class DelSndTimer : public TimerHandler {
public:
DelSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
virtual void expire(Event *e);
TcpAgent *a_;
```

```
};
class BurstSndTimer : public TimerHandler {
public:
BurstSndTimer(TcpAgent *a) : TimerHandler() { a_ = a; }
protected:
virtual void expire(Event *e);
TcpAgent *a_;
};
```

在tcp.cc 的TcpAgent构造函数中,每个定时器都用指针this来初始化。指针this被分配给指针a_。

```
TcpAgent::TcpAgent() : Agent(PT_TCP), rtt_active_(0), rtt_seq_(-1),
...
rtx_timer_(this), delsnd_timer_(this), burstsnd_timer_(this)
{
...
}
```

下面，我们将集中讨论重传定时器。可定义多种帮助方法(helper method)来规划timer事件。例如:

```
/*
* Set retransmit timer using current rtt estimate. By calling resched()
* it does not matter whether the timer was already running.
*/
void TcpAgent::set_rtx_timer()
{
rtx_timer_.resched(rtt_timeout());
}
/*
* Set new retransmission timer if not all outstanding
* data has been acked. Otherwise, if a timer is still
* outstanding, cancel it.
*/
void TcpAgent::newtimer(Packet* pkt)
{
hdr_tcp *tcph = (hdr_tcp*)pkt->access(off_tcp_);
if (t_seqno_ > tcph->seqno())
set_rtx_timer();
else if (rtx_timer_.status() == TIMER_PENDING)
rtx_timer_.cancel();
}
```

上述代码中，方法set-rtx-timer()通过调用rtx-timer-resched()重新调度了重传计时器。注：如果不清楚timer是否已经在运行，则调用resched()除去该需要以明确删除timer。第二个函数里，给出了关于status()和cancel(void)方法的用处(use)的例子。

最后，RtxTimer类的expire(void)方法必须被定义。此时expire(void)调用TcpAgent的timerout(void)方法。这是可能的，因为timeout()是一个公共成员函数;如果它不是,则RtxTimer将不得不被声明TcpAgent的友元类。

```
void TcpAgent::timeout(int tno)
{
/* retransmit timer */
if (tno == TCP_TIMER_RTX) {
if (highest_ack_ == maxseq_ && !slow_start_restart_) {
/*
* TCP option:
* If no outstanding data, then don’ t do anything.
*/
return;
};
recover_ = maxseq_;
recover_cause_ = 2;
closecwnd(0);
reset_rtx_timer(0,1);
send_much(0, TCP_REASON_TIMEOUT, maxburst_);
} else {
/*
* delayed-send timer, with random overhead
* to avoid phase effects
*/
send_much(1, TCP_REASON_TIMEOUT, maxburst_);
}
}
void RtxTimer::expire(Event *e) {
a_->timeout(TCP_TIMER_RTX);
}
```

不同 TCP 代理含有附加的 timer 例子__

11.2 OTcl timer 类

一个简单的timer 类定义在~ns/tcl/timer.tcl里。Timer子类也可按需要定义。与C++定时器的API不同，如果timer已被设置，则sched()会异常中断；而这里，sched()和resched()是一样的：例如,没有状态为OTcl定时器保持。下面方法定义在Timer基类里：

```
$self sched $delay ;# causes "$self timeout" to be called $delay seconds in the future
$self resched $delay ;# same as "$self sched $delay"
$self cancel ;# cancels any pending scheduled callback
$self destroy ;# same as "$self cancel"
$self expire ;# calls "$self timeout" immediately
```

11.3 命令一览

下面是timer类的一系列方法.。注：从基类中派生出许多不同的Timer类型
(viz.:ogTimer,timer/Iface,Timer/Iface/Prune,CacheTimer,Timer/Scuba etc)。

$timer sched<delay>
该命令在<delay>时间后删除所有其它事件,这些事件可能已被执行，并要重新执行其它事件。

$timer resched<delay>
与”sched”类似，添加它以使它有与c++ timer相似的APIs。

$timer cancel
删除所有已被执行的事件。

$timer destroy
与cancel类似，取消所有调度的事件。

$timer expire
该命令访问一个time-out,而time-out过程需要在子类中定义。

所有这些过程都在~ns/tcl/mcast/timer.tcl 里。

第 12 章

分组头及其格式

本章所描述的过程与函数可以在~*ns*/tcl/lib/ns-lib.tcl，~*ns*/tcl/lib/ns-packet.tcl以及~*ns*/packet.{cc, h}中找到。

Packet类中的对象是模拟中各种对象之间的基本交换单元。Packet类提供足够的信息将分组链接到一个列表当中（例如，在一个PacketQueue中或者一个分组的free list之上），它既可以代表包含分组头（定义在任一种协议基础上）的一段缓冲区，也可以代表分组数据的一段缓冲区。新协议可以定义它们自己的分组头或者通过添加域（field）来扩展已有的分组头。新的分组头通过以下的方法引入模拟器中。首先，定义一个C++的结构体（structure）来携带所需的域，还要定一个静态类来提供OTcl联接，然后修改模拟器的一些初始化代码，在每个分组中分配一个字节偏移量（byte offset）来定位新的分组的相对位置。

当模拟器通过OTcl初始化的时候，用户可以选择只激活（enable）已编译的分组格式的一个子集，这样可以在模拟执行过程中最大限度的节约内存。当前，大多数配置的分组格式是默认激活的。可以通过下面描述的一个特殊的分组头管理对象来管理当前模拟中激活的分组格式。这个对象提供一个OTcl方法来指定哪种分组头将用于模拟。如果模拟器中有个对象使用了还没有被激活的域，将会出现一个run-time fatal program异常中断。

12.1 协议特定的分组头

协议的开发者们通常希望能够在分组中提供一个特定的header类型。这样可以使一个新的协议实现的时候避免重载已经存在的header域。以一个RTP协议的简化版本为例。RTP协议的header需要一个序列号域和一个源标识（source identifier）域。下面的类创建了所需的header（见~*ns*/rtp.h和~*ns*/rtp.cc）

在rtp.h中：

```
/* rtp协议分组，对于当前，只有srcid和seqno. */
struct hdr_rtp {
u_int32_t srcid_;
int seqno_;
/* 各个域成员函数 */
u_int32_t& srcid() { return (srcid_); }
int& seqno() { return (seqno_); }
/* 分组头访问函数 */
static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_rtp* access(const Packet* p) {
return (hdr_rtp*) p->access(offset_);
}
};
```

在rtp.cc中:

```
class RTPHeaderClass : public PacketHeaderClass {
public:
RTPHeaderClass() : PacketHeaderClass("PacketHeader/RTP",
sizeof(hdr_rtp)) {
bind_offset(&hdr_rtp::offset_);
}
} class_rtphdr;
void RTPAgent::sendpkt()
{
Packet* p = allocpkt();
hdr_rtp *rh = hdr_rtp::access(p);
lastpkttime_ = Scheduler::instance().clock();
/* 给srcid_和seqno赋值 */
rh->seqno() = seqno_++;
rh->srcid() = session_->srcid();
target_->recv(p, 0);
}
RTPAgent::RTPAgent()
: session_(0), lastpkttime_(-1e6)
{
type_ = PT_RTP;
bind("seqno_", &seqno_);
}
```

第一个结构体，hdr_rtp，定义了RTP协议分组头的设计（layout）（根据它们的名字和它们所处的位置）：需要哪些域以及这些域占用多大空间。这一结构体的定义只用于编译器计算域的字节偏移量；这个结构体类型中的所有对象都不能直接分配。这个结构体还提供了成员函数，这些成员函数为需要读取和修改分组的header域的对象提供数据隐藏的层次。注意，静态类变量offset_用于在任意一个ns分组中定位的rtp协议header部分的字节偏移量。有两种方法可以利用这个变量在任意分组中访问这个header：offset()和access()。大多数用户应该选择后一种方法用来访问一个分组中这个特殊的header；前一种方法被分组头management类调用，很少被用户使用。例如，要访问一个分组中被指针p指向的RTP分组头，简单地说，就是hdr_rtp::access(p)。实际一个分组中这个header的位置绑定offset_的代码在文件~*ns*/tcl/lib/ns-packet.tcl以及*ns*/packet.cc中。access()中的const参数提供（假设）对一个const分组的只读访问，由于返回的指针不是const类型，所以只能是只读的。一种正确的做法是同时提供两个方法，一个用于只写访问，另一个用于只读访问。然而，这部分目前还没有实现。

重要提示：注意，这与用原始的（或陈旧的）的方法来访问分组头完全不同，原始的方法需要为访问任意分组头定义一个整型变量，off_<hdrname>_。这个方法目前已经过时了；必须谨慎地使用它，且很难察觉出误用。

当RTP协议的分组头在配置时被激活，RTPHeaderClass类的静态对象class_rtphdr用于提供OTcl联接。当模拟执行时，这个静态对象以PacketHeader/RTP和sizeof(hdr_rtp)为参数调用构造函数PacketHeaderClass。这为RTP协议分组头的大小（size）分配了空间，并在配置时刻由分组头管理器（manager）可用。注意到bind_offset()必须在这个类构造函数中被调用，因此，分组头管理器知道在哪里为这个特殊的分组头存储偏移量。

RTPAgent的样本（sample）成员函数sendpkt()方法通过调用来创建并发送一个新的分组，这个allocpkt()函数负责所有网络层分组的header域的分组(此处为IP)。除IP以外的header是单独处理的。在这里,代理使用上面定义的RTPHeader。成员函数Packet::access(void)返回用于保留分组头信息的缓冲区中的第一个字节的地址(见下文)。其返回值是指向interest的header的一个指针，此后RTPHeader对象的成员函数用于访问单独的域。

12.1.1 添加新的分组头类型

如果要创建一个叫newhdr的新分组头，需要执行以下步骤：

1． 创建一个新的结构体定义原始域（名为hdr_newhdr），定义offset_和访问方法。

2． 为所需的域定义成员函数。

3． 创建一个静态类来执行OTcl联接（定义了PacketHeader/Newhdr），在它的构造函数中执行bind_offset()。

4． 编辑~*ns*/tcl/lib/ns-packet.tcl来激活新的分组类型（见12.2.2及12.2.4）。

推荐使用这个方式添加你的分组头。如果你不依照这个方法，模拟也许仍然能够工作，但当更多的协议添加进模拟的时候，它的行为是不可预知的。原因是ns分组中的BOB(比特包,见12.2.1)是一个很大的稀疏的空间,分配一个错误的分组头offset可能不会立即触发错误。

12.1.2 在模拟中选择包含的分组头

默认情况下，模拟中ns的每个分组中都会包含所有协议的所有分组头。这是很多的头（overhead），并且随着更多的协议添加进ns,还会继续增加。对于“分组密集(packet-intensive)”型的模拟， 这将是一笔巨大的开销。例如,现在(2000年8月30日），ns中所有协议的分组头的大小大约是1.9KB；然而，如果只关心公共（common）header，IP header和TCP header，加起来大约才100个字节。如果用一些长肥管道做大规模的web流量模拟，去掉那些不需要使用的分组头可以节省大部分的内存。

为了在你指定的模拟中只包含你所感兴趣的那些分组头，可以使用以下这种模式（譬如，你想要在模拟中移除AODV和ARP头）：

```
remove-packet-header AODV ARP
.....
set ns [new Simulator]
```

注意， remove-packet-header必须在模拟器对象创建之前使用。所有分组头的名字都使用PacketHeader/[hdr]的形式。用户只需要提供[hdr]部分，不需要前缀。分组头名字，可以在~*ns*/tcl/lib/ns-packet.tcl中查找，也可以在ns中运行下面的简单的命令：

```
foreach cl [PacketHeader info subclass] {
puts $cl
}
```

如果在模拟中只想包含特定集合的分组头，例如IP和TCP，可以使用如下的模式：

```
remove-all-packet-headers
add-packet-header IP TCP
.....
set ns [new Simulator]
```

重要提示：模拟中绝对不能移除common头。正如~*ns*/tcl/lib/ns-packet.tcl所示，它影响着这些分组头的处理过程。

注意，在默认情况下，所有的分组头都是包含的。

12.2 Packet类

通常有四个C++类来对分组和分组头进行相应的处理：Packet类，p_info类，PacketHeader类以及PacketHeaderManager类。在模拟中，Packet类定义了所有分组的类型；它是Event的子类，所以可以被调度（例如，在一些队列中可以后到达）。packet_info类为所有的分组名字保存文本。PacketHeader类为所有模拟的分组头配置提供基类。实际上它提供了足够的内部声明，用来定位任意给定分组中分组头集合中的特定的分组头。PacketHeaderManager类定义了一个用于收集和管理已经配置的分组头的类。在模拟配置的时候，它被OTcl提供的一个方法调用来激活已编译的分组头的一些子集。

12.2.1 Packet类

Packet类定义了一个分组的结构体，并为这种类型的对象提供成员函数用以处理一个空闲链表（free list）。图12.1描述了这种结构，并在packet.h中有如下定义：

```
class Packet : public Event {
private:
friend class PacketQueue;
u_char* bits_;
u_char* data_;                                  /* data的变量大小缓存 */
u_int datalen_;                                 /* 变量大小缓存的长度 */
protected:
static Packet* free_;
public:
Packet* next_;                                  /* 为了排队和空闲链表 */
static int hdrlen_;
Packet() : bits_(0), datalen_(0), next_(0) {}
u_char* const bits() { return (bits_); }
Packet* copy() const;
static Packet* alloc();
static Packet* alloc(int);
inline void allocdata(int);
static void free(Packet*);
inline u_char* access(int off) {
if (off < 0)
abort();
return (&bits_[off]);
}
inline u_char* accessdata() { return data_; }
};
```

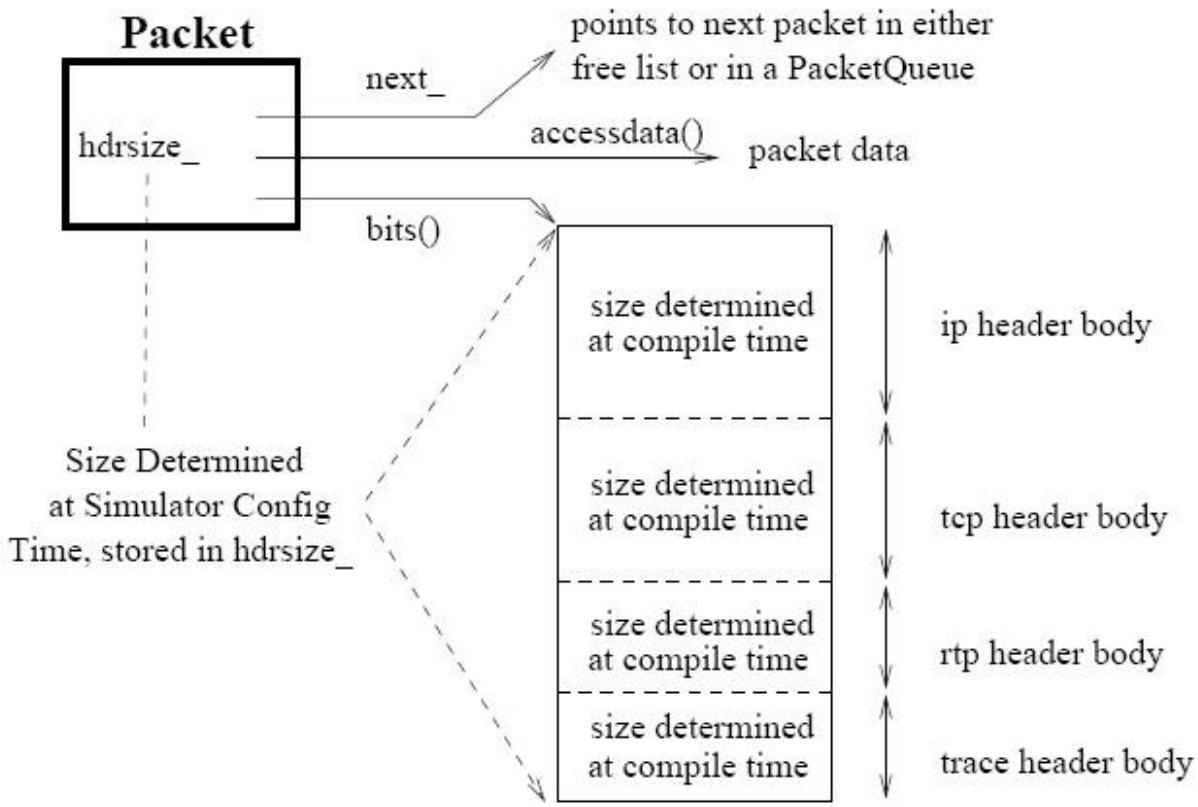


图12.1 Packet对象

这个类保留了一个指向无符号字符组成的普通数组的指针（通常叫做“比特包 ”，或简称BOB），这个数组存储了分组头域。它同时还保留了一个指向分组“数据”的指针（模拟中通常用不到）。变量bits_存储BOB第一个字节的地址信息。（目前实现的）有效的BOB是配置的每个分组头中定义的所有结构体（按照惯例，这些结构体的名字以hdr_hsomething开头）的一个串联（concatenation）。BOB在一个模拟中通常保持一个固定的大小，这个大小记录在Packet::hdrlen_的成员变量中。用OTcl[13]进行模拟配置的过程中，这个大小被更新。

Packet类的其它方法用于在一个私有空闲链表中创建新的分组并存储旧的（未使用过的）分组。这种分配和存储（在*~ns*/packet.h中）用以下代码实现：

```
inline Packet* Packet::alloc()
{
Packet* p = free_;
if (p != 0)
free_ = p->next_;
else {
p = new Packet;
p->bits_ = new u_char[hdrsize_];
if (p == 0 || p->bits_ == 0)
abort();
}
return (p);
```

[13] 这里没有规定在配置时间之后更新。在配置之后更新应该也是可行的，但是目前还未测试。

```
}
/* 分配给一个分组n个字节的数据缓存 */
inline Packet* Packet::alloc(int n)
{
Packet* p = alloc();
if (n > 0)
p->allocdata(n);
return (p);
}
/* 给一个已有的分组分配一个n个字节的数据缓存 */
inline void Packet::allocdata(int n)
{
datalen_ = n;
data_ = new u_char[n];
if (data_ == 0)
abort();
}
inline void Packet::free(Packet* p)
{
p->next_ = free_;
free_ = p;
if (p->datalen_) {
delete p->data_;
p->datalen_ = 0;
}
}
inline Packet* Packet::copy() const
{
Packet* p = alloc();
memcpy(p->bits(), bits_, hdrlen_);
if (datalen_) {
p->datalen_ = datalen_;
p->data_ = new u_char[datalen_];
memcpy(p->data_, data_, datalen_);
}
return (p);
}
```

alloc()方法通常是创建新的分组时作为一个辅助函数（support function）。它被代表agent Agent::allocpkt()调用，而不是通常的被大多数对象直接调用。首先，它试图在使用C++的new操作分配失败时在空闲链表中定位一个旧的分组。注意，Packet类的对象和BOB是分别分配的。free()方法通过将一个分组返回到空表中来释放这个分组。注意，分组决不会返回系统的内存分配器中。取而代之的是，在Packet::free()被调用的时候，它们存储在一个空表中。成员copy()创建一个分组的新的，同样的拷贝，除了唯一的uid_域不同。这个函数被Replicator对象使用，用以支持多点传输分布以及局域网。

12.2.2 p_info类

该类是绑定数值分组类型值和它们符号名字的“胶水”。当定义一个新的分组类型的时候，它的数值代码要添加到packet_t的枚举结构中（见~*ns*/packet.h）[14]。它的符号名字也应该添加到p_info的构造函数中：

```
enum packet_t {
PT_TCP,
...
PT_NTYPE                    // 这必须是最后一项
};
class p_info {
public:
p_info() {
name_[PT_TCP]= "tcp";
...
}
}
```

12.2.3 hdr_cmn类

在模拟器中，每一个分组都有一个”common”头，定义在~*ns*/packet.h中，如下所示：

```
struct hdr_cmn {
double ts_;                 /* 时间戳：用于测量q-delay */
packet_t ptype_;            /* 分组类型（见上文） */
int uid_;                   /* 唯一的id */
int size_;                  /* 模拟的分组大小 */
int iface_;                 /* 接收接口（标签） */
/* 分组头访问函数 */
static int offset_;
inline static int& offset() { return offset_; }
inline static hdr_cmn* access(Packet* p) {
return (hdr_cmn*) p->access(offset_);
}
/* 每个域的成员函数 */
int& ptype() { return (ptype_); }
int& uid() { return (uid_); }
int& size() { return (size_); }
int& iface() { return (iface_); }
double& timestamp() { return (ts_); }
};
```

这个结构体主要定义了一些域，用于跟踪分组流或者测量其它量。时间戳（timestamp）域用来测量交换节点（switch

[14] 注解：PT _ NTYPE 保留枚举变量的最后一个元素。

nodes）上的排队时延。ptype_域用来识别分组的类型，可以使跟踪文件的读取更为简便。uid_域被调度器用于调度分组的到达。size_域经常使用，它以字节为单位虚拟分组的大小。注意，模拟中消耗的字节的实际数量可能与这个域的值没有关系（例如，size_与sizeof(struct hdr_cmn)或者ns中其它的结构体没有关系）。更为恰当的说法是，大多数情况下它用于计算网络链路上一个分组传输所需的时间。因此它应被设置为模拟的分组的应用数据大小，IP层，传输层和应用层header大小的和。iface_域用于多播分布树的计算。它是一个标识有分组到达的链路的标签。

12.2.4 PacketHeaderManager类

PacketHeaderManager类的一个对象用于管理目前激活状态的分组头类型，并且在BOB中为它们各自分配一个唯一的offsets。这在C＋＋和OTcl中代码中都有定义：

在tcl/lib/ns-packet.tcl中:

```
PacketHeaderManager set hdrlen_ 0
......
foreach prot {
AODV
ARP
aSRM
Common
CtrMcast
Diffusion
......
TORA
UMP
} {
add-packet-header $prot
}
Simulator instproc create_packetformat {} {
PacketHeaderManager instvar tab_
set pm [new PacketHeaderManager]
foreach cl [PacketHeader info subclass] {
if [info exists tab_($cl)] {
set off [$pm allochdr $cl]
$cl offset $off
}
}
$self set packetManager_ $pm
}
PacketHeaderManager instproc allochdr cl {
set size [$cl set hdrlen_]
$self instvar hdrlen_
set NS_ALIGN 8 ;# round up to nearest NS_ALIGN bytes, (needed on sparc/solaris)
set incr [expr ($size + ($NS_ALIGN-1)) & ~($NS_ALIGN-1)]
set base $hdrlen_
```

```
incr hdrlen_ $incr
return $base
}
From packet.cc:
/* 管理激活状态的分组头类型 */
class PacketHeaderManager : public TclObject {
public:
PacketHeaderManager() {
bind("hdrlen_", &Packet::hdrlen_);
}
};
```

模拟器在初始化的时候，执行~*ns*/tcl/lib/ns-packet.tcl中的代码。因此，在模拟开始时就执行foreach语句，并且初始化OTcl类和tab_数组来包含类的名字和当前激活状态的分组头类的名字之间的映射。如上（12.1）所讨论的，分组头应该用hdr_hhdrnamei::access()来访问。

create_packetformat{}实例过程是基础的Simulator类的一部分，并且在模拟配置的时候就被调用。它首先创建一个单独的PacketHeaderManager对象。接着C++构造函数将OTcl实例变量hdrlen_（PacketHeaderManager类中的）和C++变量Packet::hdrlen_（Packet类的一个静态成员）链接起来。这就起到了将Packet::hdrlen_置0的效果。注意，以这种方法通过类的类型的绑定是不常见的。

在创建分组管理器之后，foreach循环激活我们感兴趣的每个分组头。这个循环遍历具有（h_i, o_i）格式定义的分组头列表。

这里的h_i表示第i个分组头的名字，o_i是包含分组头在BOB中的位置的变量的名字。Header的放置是通过PacketHeaderManager这个OTcl类的allochdr instproc来执行的。当新的分组头被激活时，此过程维护一个当前BOB长度的hdrlen_变量。它还为所有的新激活的分组头分配一个8字节的线性结构。这是为了保证在需要使用double-world的计算机器上分组头使用这个长度量时，不会出现访问错误[15]。

12.3 命令一览

下面是packet-header相关过程的列表：

Simulator::create_packetformat

这是一个模拟器内部的过程，在模拟器配置开始的时候调用来创建一个packetHeaderManager对象。

PacketHeaderManager::allochdr

这是PacketHeaderManager类的另一个内部过程，当新的分组头被激活时，它被用来跟踪变量hdrlen_。它还为任意新激活的分组头分配一个8字节的线性结构。

add-packet-header

维护一个参数列表，每个参数是一个分组头的名字（不包括PacketHeader/前缀）。这个全局过程会通知模拟器在模拟中包含特定的分组头。

[15] 在包括 Sparc 和 HP - PA 在内的一些处理器架构中，double-world 的访问必须在 double-world 的边界上进行（例如，地址在 0 模 8 结束）。试图进行非线性访问将会导致程序异常终止。

remove-packet-header

使用同一种语法执行，但它是从模拟中移除特定的分组头。注意，即使它被指定来移除common头，也不会将common头移除。

remove-all-packet-headers

是一个全局Tcl过程。它不包括任何参数，并移除除common头以外所有的分组头。

add-all-packet-headers

与它的作用相反。

第 13 章

错误模型（Error Model）

本章阐述了在模拟中引入分组丢失概念的错误模型的实现与配置。

下面除详细描述的基本的ErrorModel类外，还描述了许多其它尚未完全载入文档的error模型，包括：

- SRMErrorModel，PGMErrorModel：SRM和PGM的错误模型
- ErrorModel/Trace：能读取一个丢失trace的错误模型（而不是一个数学/计算模型）。
- MrouteErrorModel：多播路由的错误模型，现在由trace继承而来。
- ErrorModel/Periodic：周性性的丢包模型（我们看到每次的第n个包丢失）。该模型可以很方便地与一个基于流量的分类器（classifier）相结合，用来实现在特定数据流中的丢包。
- SelectErrorModel：用于选择性丢包。
- ErrorModel/TwoState：两种状态：error-free和error。
- ErrorModel/TwoStateMarkov, ErrorModel/Expo, ErrorModel/Empirical：均由ErrorModel/TwoState继承而来。
- ErrorModel/List：定义了一系列丢失的分组/字节数，可以组合为任意的顺序。

上述这些错误模型的定义都可以在~*ns*/queue/errmodel.{cc, h}，~*ns*/tcl/lib/ns-errmodel.tcl,以及ns-default.tcl中找到。

13.1 实现

本小节所描述的过程和函数可在~*ns*/errmodel.{cc, h}中找到。

错误模型通过标记分组的error标志位（flag）或将分组放入（dump）drop对象来模拟链路层的错误或者丢失（loss）。在模拟中，errors可由一个简单的模型（如分组错误率）或一些更加复杂的统计和经验模型来产生。为了支持更为宽泛多样的模型，error的单位可以按分组，比特或是基于时间来定义。

ErrorModel类是由Connector基类派生的。因此，它继承了某些方法以钩住（hook up）target和drop-target等对象。如果drop target存在，它将从ErrorModel接收中断的分组。否则，ErrorModel仅在common头标记error_标志位，从而可允许代理来处理这种丢失。ErrorModel同时也增加了Tcl方法unit来定义error的单位，以及ranvar来定义产生errors的随机变量。如果没有定义，error的将以分组作为单位，而随机变量将均匀分布在0到1的区间上。下面是一个创建一个error模型的简单的例子，其分组错误率为1%（0.01）：

```
# 创建一个loss_module，并将其分组错误率置为1%
set loss_module [new ErrorModel]
$loss_module set rate_ 0.01
# 可选项：设置单位和随机变量
$loss_module unit pkt                    ;# error单位：分组（默认）
$loss_module ranvar [new RandomVariable/Uniform]
# 为丢失的分组设置target
$loss_module drop-target [new Agent/Null]
```

在C++中，ErrorModel包括分组丢失的机制和策略。分组丢失机制由recv方法来处理，而分组中断策略则由corrupt

方法来处理。

```
enum ErrorUnit { EU_PKT=0, EU_BIT, EU_TIME };
class ErrorModel : public Connector {
public:
ErrorModel();
void recv(Packet*, Handler*);
virtual int corrupt(Packet*);
inline double rate() { return rate_; }
protected:
int command(int argc, const char*const* argv);
ErrorUnit eu_;                          /* error以分组，比特或时间为单位 */
RandomVariable* ranvar_;
double rate_;
};
```

ErrorModel只是实现了一个基于单一的错误率（或在比特分组中）的简单策略。更为复杂的丢失策略可以通过在C++中对ErrorModel派生，并重新定义其corrupt方法实现。

13.2 配置

前一小节讨论了错误模型，在这一小节中我们讨论如何在ns中使用有线网络或无线网络的错误模型。

为了在有线网络上使用错误模型，首先必须把它插入到一个SimpleLink对象中。由于SimpleLink是一个复合的对象（第6章），所以可以在许多地方插入错误模型。现在我们提供如下的方法用来在3个不同的地方插入一个error模块：

- 在SimpleLink中，将一个error模块插入到队列前。由下述两种OTcl方法实现：

SimpleLink::errormodule args 当一个错误模型的参数给定时，它将其插入到简单链路中，正好在队列模型之后，并将错误模型的drop-target设置为简单链路的丢失trace对象。注意需要以下的配置次序：插入错误模型后，接着配置链路配置，最后ns namtrace-all。未给定参数的情况下，它将返回当前的错误模型（如果有的话）。该方法定义在*ns*/tcl/lib/ns-link.tcl中。

Simulator::lossmodel <src> <dst> 调用SimpleLink::errormodule来将给定的错误模型插入到简单链路(src, dst)中。

- 在SimpleLink中，将error模块插入到队列之后延时链路之前。由如下两种方法提供：

SimpleLink::insert-linkloss args 除了直接在队列对象之后插入一个错误模型，这个方法的行为（behavior）同SimpleLink::errormodule是一样的。它定义在*ns*/tcl/lib/ns-link.tcl中。

Simulator::link-lossmodel <src> <dst> 这是SimpleLink::insert-linkloss的一个封装。它定义在*ns*/tcl/lib/ns-lib.tcl中。

通过这两种方法插入错误模型来产生的nam trace不需要特殊的处理，并可以用旧版本的nam来显示。

- 在链路中，将error模块插入到延时模块之后。可由Link::install-error来完成。目前这种API不产生任何trace。它仅为未来可能的扩展提供占位符（placeholder）。

为了在无线网络上添加一个错误模型，每个节点都可在输出或输入的无线信道上插入一个给定的统计的错误模型。更为确切地说，实例化的错误模型刚好介于图16.2中的mac和netif模块之间。对于输出链路，error模块由上层的mac模块的

downtarget_指针指向，而对于输入链路，它将由下层的netif模块的uptarget_指针来链接。且在每种情况下，error模块的target_各自指向netif和mac。这两种不同放置位置的区别在于：由于error在无线信道模块packet拷贝之前就已经确定了，所以输出将导致所有接收器接收到同样程度error的分组。另一方面，由于error是在各个error模块中独立计算的，所以输入error模块让每个接收器接收被不同程度error污染的分组。

无线协议栈的插入可以通过调用node-config命令来完成，它有两个选项：IncomingErrrProc和OutgoingErrProc。我们可以同时使用这两个选项或分开使用其中任何一个。这两个选项的自变量（argument）是可创建错误模型对象的全局过程名，给新建的error模块赋适合的必要的初始值，最终会返回该对象的指针。以下展示了在无线协议栈中添加error模块的简单的Tcl示例脚本。

```
$ns node-config -IncomingErrProc UniformErr -OutgoingErrProc UniformErr
proc UniformErr
set err [new ErrorModel]
$err unit packet
return $err
```

13.3 多态错误模型

由Jianping Pan (jpan@bbcr.uwaterloo.ca)投稿。

多态错误模型实现了基于时间的error状态转换。转换到下一个error态发生在当前状态持续时间的结尾。然后采用状态转换矩阵选择下一个error态。

为了生成多态错误模型，需要提供以下参数（如*ns*/tcl/lib/ns-errmodel.tcl所定义的）：

●states: 状态序列（错误模型）
●periods: 状态持续时间序列
●trans: 状态转换模型矩阵
●transunit: [pkt|byte|time]之一
●sttype: 用于状态转换的类型：time或者pkt
●nstates: 状态数
●start: 起始态

这里有一个生成多态错误模型的简单的示例脚本：

```
set tmp [new ErrorModel/Uniform 0 pkt]
set tmp1 [new ErrorModel/Uniform .9 pkt]
set tmp2 [new ErrorModel/Uniform .5 pkt]
# 状态序列（错误模型）
set m_states [list $tmp $tmp1 $tmp2]
# tmp, tmp1以及tmp2状态各自的持续时间
set m_periods [list 0 .0075 .00375]
# 状态转换模型矩阵
set m_transmx { {0.95 0.05 0}
{0 0 1}
{1 0 0} }
set m_trunit pkt
# 使用基于时间的转换
set m_sttype time
```

```
set m_nstates 3
set m_nstart [lindex $m_states 0]
set em [new ErrorModel/MultiState $m_states $m_periods $m_transmx
$m_trunit $m_sttype $m_nstates $m_nstart]
```

13.4 命令一览

以下是一系列在模拟脚本中常用的与错误模型有关的命令：

```
set em [new ErrorModel]
$em unit pkt
$em set rate_ 0.02
$em ranvar [new RandomVariable/Uniform]
$em drop-target [new Agent/Null]
```

这是一个关于如何生成和配置一个错误模型的简单例子。在简单链路中放置错误模型的命令如下述所示。

```
$simplelink errormodule <args>
```

该命令将错误模型插入到简单链路中的队列对象之前。然而，在这种情况下，错误模型的drop-target指针指向链路的drophead_元素。

```
$ns_ lossmodel <em> <src> <dst>
```

该命令将错误模型插入到简单链路的队列之前，该简单链路由<src>和<dst>节点定义。这基本是对上面的方法的一个封装。

```
$simplelink insert-linkloss <args>
```

该命令将丢失模块（loss-module）插入到简单链路中的队列之后，但正好在延迟link_元素之前。这是因为只有当分组在链路上或者在队列中时，nam才会显示出分组的丢失。该模型的drop-target指针指向链路的drophead_元素。

```
$ns_ link-lossmodel <em> <src> <dst>
```

该命令也是上面所述的插入丢失链路方法的一个封装方法。只不过该命令将error模块刚好插入到简单链路（源 - 目的）的队列元素之后。

第 14 章

局域网

无线网和局域网（LAN）的特征从本质上与那些点到点的链路不同。由多条点对点链路组成的网络不能获得局域网的共享和竞争特性。为了模拟这些特性，我们创建了一个新的类型的节点，称为LanNode。LanNode的关于OTcl的配置和接口在ns主目录的下面三个文件中：

tcl/lan/vlan.tcl

tcl/lan/ns-ll.tcl

tcl/lan/ns-mac.tcl

14.1 Tcl配置

创建和配置局域网的接口与那些点对点的链路有轻微的差别。在顶层，OTcl类Simulator输出了一个新的方法：make-lan。这个方法的参数与duplex-link方法相似，除了make-lan只接收节点的一个列表作为一个参数，而duplex-link中作为两个参数。

```
Simulator instproc make-lan {nodes bw delay lltype ifqtype mactype chantype}
```

make-lan的可选参数规定了所要创建的对象的类型，如：链路层（LL），接口队列，MAC层（Mac），物理层（Channel）。

下面是怎样创建一个新的CSMA/CD（以太网）局域网的例子。

例如：

```
$ns make-lan "$n1 $n2" $bw $delay LL Queue/DropTail Mac/Csma/Cd
```

创建了有基本的链路层，drop-tail队列，CSMA/CD机制的MAC层的局域网。

14.2 局域网的组成

局域网链路具有在网络栈中三个最底层的功能：

1.链路层（LL)

2.媒介访问控制层（MAC)

3.物理层(PHY)

图14.1解释了这种扩展的网络栈结构，该结构使得在ns中模拟局域网成为可能。沿着这种栈结构向下发送的数据包流经链路层（Queue和LL），MAC层（Mac），以及物理层（Classifier/Mac的Channel）。然后分组继续向前通过Mac和LL向上回到栈结构。

在栈的底部，物理层有两个模拟对象组成：Channel和Classifier/Mac。Channel对象模拟共享媒介，并且支持在传输的发送端的MAC对象的访问机制。在接收端，Classifier/Mac负责分发和随意分组的拷贝给接收的MAC对象。

依赖于物理层的类型，MAC层必须包含某些功能的集合，比如 载波侦听（carrier sense），冲突检测（collision detection），冲突避免（collision avoidance）等等。由于这些功能会同时影响发送和接收端，因此它们在单一的Mac对象里面实现。对

于发送端，在向channel上传分组之前，Mac对象必须遵循某种媒介访问协议。对于接收端，MAC层负责把分组递交给链路层。

在MAC层上面，链路层能潜在地拥有许多功能，比如排队和链路层重传。这种有多种链路层设计的需求导致功能划分为两个组件：队列和链路层。模拟接口队列的Queue对象属于在第七章中描述的相同的Queue类。LL对象实现了一种特别的数据链路协议，比如ARQ。通过把发送和接收功能结合成一个模块，LL对象也能支持其它的机制，比如捎带（piggybacking）。

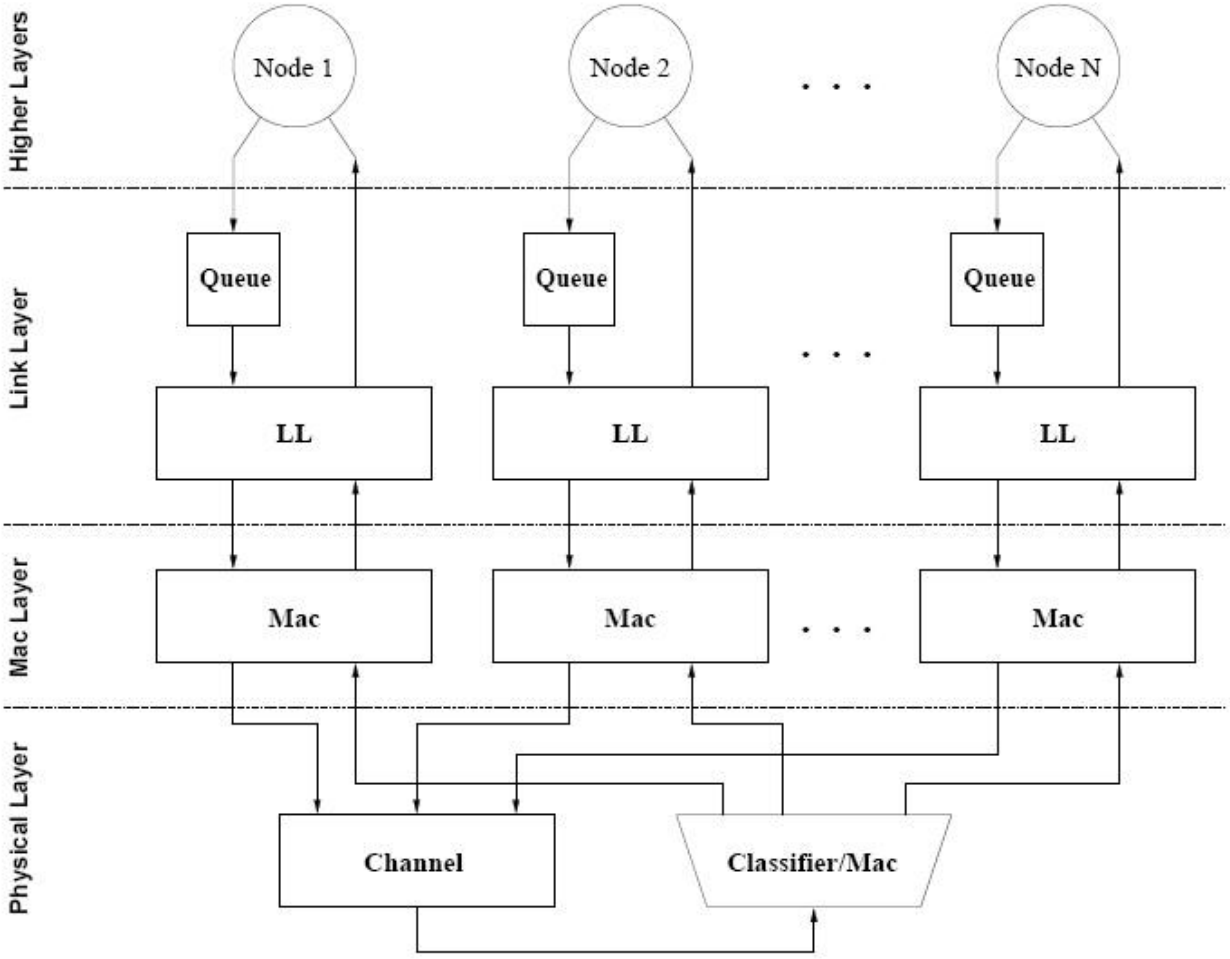


图16.1 局域网中的连接性

14.3 Channel类

Channel类模拟分组在物理层上的实际传输。基本的Channel类实现了支持竞争机制的共享媒介。它允许MAC实现载波侦听，竞争以及冲突检测。如果同时不止一条传输相重叠，通道就不会引起冲突标记。通过检查这个标记，MAC对象能够实现对冲突的检测与处理。

由于传输时间是分组里面的比特数和每一个单独接口（MAC）的调制速度的函数，因此Channel对象对于MAC对象请求的持续时间仅仅设置它的忙信号。在传输时间加上传播时延后，它也要调度发送到MAC对象的分组。

14.3.1 Channel状态

C++的Channel类包括足够多的用于调试分组的发送以及冲突检测的内部状态。它输出下面的OTcl配置参数：

delay_ 通道上的传播时延

14.3.2 例子：物理层的Channel和分类器

```
set channel_ [new Channel]
$channel_ set delay_ 4us # propagation delay
set mcl_ [new Classifier/Mac]
$channel_ target $mcl_
$mcl_ install $mac_DA $recv_iface
. . .
```

14.3.3 C++中的Channel类

在C++中，Channel类由Connector对象扩展而来，并且带有几个新的方法用于支持一系列的MAC协议。该类在~*ns*/channel.h中定义如下：

```
class Channel : public Connector {
public:
Channel();
void recv(Packet* p, Handler*);
virtual int send(Packet* p, double txtime);
virtual void contention(Packet*, Handler*);
int hold(double txtime);
virtual int collision() { return numtx_ > 1; }
virtual double txstop() { return txstop_; }
. . .
};
```

Channel类的重要方法有：

- txstop()：当通道变成空闲的时候，该方法返回时间，这个时间被MAC用来实现载波侦听。
- contention()：该方法允许MAC在发送分组前竞争通道。然后通道用这个分组在每个竞争周期结束时通知相应的MAC对象。
- collision()：该方法指示在竞争周期内是否有冲突发生。当通道在竞争周期结束发信号时，MAC能用collision()方法检测出冲突。
- send()：该方法允许MAC对象在特定的持续时间在通道上传输一个分组。
- hold()：该方法允许MAC对象在特定的持续时间不需要实际传输任何分组来占用通道。这个方法在模拟某些MAC协议的拥塞机制是很有用的。

14.4 MacClassifier类

MacClassifier类扩展了Classifier类，实现了一个简单的广播机制（broadcasting mechanism）。 它以下面的方式修改了recv()方法：由于分组的复制是花销大，通常情况下，单播分组会通过MAC的目的地地址macDA_进行分类，并且直接分发给这样一个地址的MAC对象。然而，如果发现不了目的地地址对象，或者MAC目的地地址被明确设置为广播地址

BCAST_ADDR ,分组就会被复制 ,并且发送给局域网上分组源地址以外的所有MAC地址。最后 ,通过将变量(bound variable) MacClassifier::bcast_设置为非零值，促使MacClassifier不断地复制分组。

```
class MacClassifier : public Classifier {
public:
void recv(Packet*, Handler*);
};
void MacClassifier::recv(Packet* p, Handler*)
{
Mac* mac;
hdr_mac* mh = hdr_mac::access(p);
if (bcast_ || mh->macDA() == BCAST_ADDR || (mac = (Mac *)find(p)) == 0) {
// 复制分组给所有的时隙（广播）
. . .
return;
}
mac->recv(p);
}
```

14.5 MAC类

Mac对象模拟媒介访问协议，这些协议在无线和局域网等等这种共享媒介环境下是必须的。由于发送和接收机制与大多数类型的MAC层是紧密耦合的，因此，MAC对象必须是双工的。

在发送端，MAC对象负责增加MAC头以及把分组传送到通道上。在接收端，MAC对象从物理层的分器异步接收分组。在MAC协议处理之后，它把数据分组传递给链路层。

14.5.1 Mac的状态

C + + 中的Mac类包含了足够多的内部状态来模拟特定的MAC协议。它也输出了下面的OTcl配置参数：

bandwidth_ MAC的调制速率
hlen_ 增加到分组的MAC头的额外字节
label_ MAC地址

14.5.2 Mac的方法

Mac类增加了几个Tcl方法用于配置，特别地是在与其它模拟对象进行链接时：

channel 指定传输的通道
classifier 发送分组给接收MAC的分类器
maclist 在同一个节点上的MAC接口的链路列表

14.5.3 C++中的Mac类

在C++中，Mac类从Connector类继承而来。当recv()方法获取一个分组的时候，根据回调（callback）处理器的存在情况，它识别分组的传输方向。如果callback处理器存在的话，分组就是被发送出去的，否则，就是接收进来的。

```
class Mac : public Connector {
public:
Mac();
virtual void recv(Packet* p, Handler* h);
virtual void send(Packet* p);
virtual void resume(Packet* p = 0);
. . .
};
```

当一个Mac对象通过recv()方法接收一个分组时，它检查这个分组是出去的还是进来的。对于一个出去的分组，它假设发送器的链路层已经获取目的地MAC地址，并且填充了MAC头的macDA域，hdr_mac。MAC对象将源MAC地址和帧类型填充至MAC头剩余的部分。然后它把分组传递给它的send()方法，这个方法实现了媒介访问协议。对于基本的Mac对象，send方法调用txtime()来计算传输时间，接着调用Channel::send方法来传送分组。最后， 在传输时间结束之后，它调度自己重新开始。

14.5.4 基于CSMA的MAC

CsmaMac类用一些新方法扩展了Mac类，这些方法实现了载波侦听和铲背机制（backoff mechanisms）。CsmaMac::send()方法利用Channel::txtime()来检测通道何时变成空闲状态。如果通道忙，MAC调度下一个载波侦听直到通道变成空闲状态。一旦通道空闲，CsmaMac对象通过Channel::contention()来初始化竞争周期。在竞争周期结束时，endofContention()方法被调用。在这个时候，基本的CsmaMac正好用Channel::send来传输分组。

```
class CsmaMac : public Mac {
public:
CsmaMac();
void send(Packet* p);
void resume(Packet* p = 0);
virtual void endofContention(Packet* p);
virtual void backoff(Handler* h, Packet* p, double delay=0);
. . .
};
class CsmaCdMac : public CsmaMac {
public:
CsmaCdMac();
void endofContention(Packet*);
};
class CsmaCaMac : public CsmaMac {
public:
CsmaCaMac();
virtual void send(Packet*);
```

```
};
```

CsmaCdMac扩展了CsmaMac，用于实现CSMA/CD（以太网）协议的冲突检测过程。当通道在竞争周期结束后发出信号时，endofContention方法使用Channel::collision()检测冲突。如果存在冲突，MAC调用它的backoff方法调度下一个载波侦听来重传分组。

CsmaCaMac扩展了CsmaMac的send方法，用于实现CSMA/CA的冲突避免过程。当信道处于空闲状态时，CsmaCaMac对象不是立即传输分组，而是返回随机数量的时隙，如果信道仍然空闲，那么就进行传输直到铲背周期结束。

14.6 LL(链路层)类

链路层对象负责模拟数据链路协议。许多协议能在这层得以实现，比如packet的分组和重组，可靠的链路协议。

链路层的另一个重要功能是设置分组的MAC头中的MAC目的地地址。在当前的实现情况下，这个任务包括了两个单独的方面：找到下一跳节点的IP地址（路由）和把这个IP地址解析成正确的MAC地址（ARP）。为了简化起见，在MAC和IP地址之间的默认映射是一对一的，这就意味着IP地址在MAC层被重用。

14.6.1 C++中的LL类

C++中的LL类从LinkDelay类继承而来。由于它是一个复合对象，它为发送目标和接收目标保持单独的指针，分别为sendtarget和recvtarget。它同时也定义了方法recvfrom()和sendto()，分别用来处理进出的数据包。

```
class LL : public LinkDelay {
public:
LL();
virtual void recv(Packet* p, Handler* h);
virtual Packet* sendto(Packet* p, Handler* h = 0);
virtual Packet* recvfrom(Packet* p);
inline int seqno() return seqno_;
inline int ackno() return ackno_;
inline int macDA() return macDA_;
inline Queue *ifq() return ifq_;
inline NsObject* sendtarget() return sendtarget_;
inline NsObject* recvtarget() return recvtarget_;
protected:
int command(int argc, const char*const* argv);
void handle(Event* e) recv((Packet*)e, 0);
inline virtual int arp (int ip_addr) return ip_addr;
int seqno_;                    // 链路层序列号
int ackno_;                    // 目前为止接收的ACK编号
int macDA_;                    // 目的MAC地址
Queue* ifq_;                   // 接口队列
NsObject* sendtarget_;         // 对外出分组
NsObject* recvtarget_;         // 对进入分组
LanRouter* lanrouter_;         // 下一跳的路由
```

};

14.6.2 例子：链路层的配置

```
set ll_ [new LL]
set ifq_ [new Queue/DropTail]
$ll_ lanrouter [new LanRouter $ns $lan]          # LanRouter是一个对象
# 对每个局域网
$ll_ set delay_ $delay                  #链路级的负载
$ll_ set bandwidth_ $bw                 # 带宽
$ll_ sendtarget $mac                    # 发送端的队列
$ll_ recvtarget $iif                    # 接收端的输入接口
. . .
```

14.7 LanRouter类

缺省情况下，每个局域网恰好只有一个LanRouter对象，该对象是由新的局域网节点在初始化时被创建。对局域网中的每个节点而言，链路层对象（LL）有一个指向LanRouter的指针，因此它能为在局域网上发送的分组找到下一跳：

```
Packet* LL::sendto(Packet* p, Handler* h)
{
int nh = (lanrouter_) ? lanrouter_->next_hop(p) : -1;
. . .
}
```

LanRouter能通过查询当前的RouteLogic找到下一跳：

```
int LanRouter::next_hop(Packet *p) {
int next_hopIP;
if (enableHrouting_) {
routelogic_->lookup_hier(lanaddr_, adst, next_hopIP);
} else {
routelogic_->lookup_flat(lanaddr_, adst, next_hopIP);
}
```

这种方法的一个局限性在于RouteLogic可能无法感知路由的动态改变。但是从LanRouter继承定义一个新的类是可行的，该类重新定义它的next-hop方法来适当地处理动态路由的改变。

14.8 其它的组件

除了上面描述的C＋＋组件外，模拟局域网同时也需要许多在ns中已有的组件，比如Classifier，Queue，以及Trace，networkinterface等等。配置这些对象需要知道用户想模拟什么。缺省配置以在本章开头部分所提到的两个Tcl文件实现。为了获得更多无线网络的实际的模拟，我们可以使用13章描述的ErrorModel。

14.9 局域网和ns路由

当通过make-lan或者newLan新建一个局域网时，一个“虚拟局域网节点”LanNode被创建了。LanNode把所有的局域网上的Channel，Classifer/Mac，LanRouter等共享对象保存在一起。然后对于局域网中的每一个节点，创建一个LanIface对象。LanIface包含了在每一个节点上都需要的所有其它的对象：如Queue，LL，Mac等等。需要强调的是，LanNode是一个仅仅用于路由算法的节点：Node和LanNode几乎没有共同点。他们共享的部分很少，其中一个便是从NodeID空间取得的标识符。如果使用了层次路由，LanNode必须像其它节点一样被分配一个层次地址。从ns（静态）路由这点来看，LanNode正好是局域网中与每个节点相连的另外的节点。连接LanNode和局域网中的节点的链路也就是“虚拟链路”（Vlink）。这样一条链路的默认路由代价是1/2，因此穿越两个Vlinks（例如：n1->Lan->n2）被算作是一跳。

Vlink最重要的方法是一个给出了链路head的方法：

```
Vlink instproc head {} {
$self instvar lan_ dst_ src_
if {$src_ == [$lan_ set id_]} {
#如果这是从局域网节点出来的一条链路，可以返回任意，因为它仅仅被$lan add-route使用
return ""
} else {
#  如果这是到局域网节点的一条链路，返回入口（entry）给LanIface对象
set src_lif [$lan_ set lanIface_($src_)]
return [$src_lif entry]
}
}
```

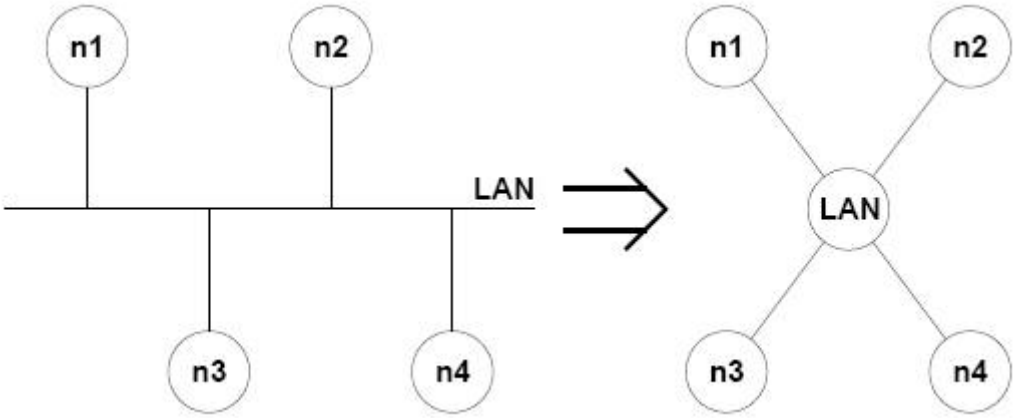


图16.2　真实局域网配置（左）和NS中路由配置（右）

这个方法被静态（缺省）路由用来在节点上安装正确的路由（参见在tcl/lib/ns-route.tcl里面的Simulator方法compute-flat-routes和compute-hier-routes，以及在tcl/lib/ns-node.tcl里面的Node方法add-route和add-hroute）。

从上面的代码段可以看出，它返回节点的LAN接口作为链路的头来安装在合适的分类器上。

因此，Vlink并没有在分组上强加任何时延，而只是为了实现其唯一的目的，这就是安装LAN接口，而不是节点分类器的普通链路。

注意，这种设计使得节点能够通过并行LAN相连，然而在当前的实现情况下，用并行简单链路将节点连接越来，并同时使用两者（数组Simulator instvar link_为每对连接节点的源和目的保存link对象，并且对于每个源/目的对只有一个对象）是不可能的。

14.10 命令一览

下面是在模拟脚本中通常用到的与局域网相关的命令的列表：

$ns_ make-lan <nodelist> <bw> <delay> <LL> <ifq> <MAC> <channel> <phy>

从一个由<nodelist>给定的节点集创建一个局域网。其带宽、时延特征连同链路层、接口队列、Mac层以及通道类型也需要被定义。使用的缺省值如下：

<LL> .. LL

<ifq>.. Queue/DropTail

<MAC>.. Mac

<channel>.. Channel and

<phy>.. Phy/WiredPhy

$ns_ newLan <nodelist> <BW> <delay> <args>

这个命令与上面描述的make-lan创建局域网相似。虽然make-lan是一个更方便、更简单的命令，但这个命令能用来作更为精确地控制。譬如，newLan可以用来创建一个层次地址的局域网。newLan的用法参见ns/tcl/ex/vlantest-hier.tcl, vlantest-mcst.tcl, lantest.tcl, mac-test.tcl。可以传递的可能的参数类型包括LL，ifq，MAC，channel，phy以及address。

$lannode cost <c>

这个命令把局域网中每条（单向）链路的代价设置为c/2。

$lannode cost?

返回局域网中（双向）链路的代价，例如c。

内部过程：

$lannode addNode <nodes> <bw> <delay> <LL> <ifq> <MAC> <phy>

Lan被作为一个虚拟节点来实现。LanNode模拟一个真实的节点，并且使用从节点的地址空间来的地址（id）。这个命令将<nodes>的列表增加到由lannode代表的局域网中去。带宽，时延以及节点的网络特征均将由上述参数给出。这是一个被make-lan和newLan使用的内部命令。

$lannode id

返回虚拟节点的id。

$lannode node-addr

返回虚拟节点的地址。

$lannode dump-namconfig

这个命令用于在nam中创建一个给定的局域网的布局（layout）。这个功能可以改变来以不同的方式重定义局域网的布局。

$lannode is-lan?

这个命令总是返回1，由于这儿的节点是代表局域网的虚拟节点。对于基类Node的相应命令是$node is-lan?，总是返回0。

第 15 章

NS中的寻址结构（修正版）

本章描述了在ns中修正过的可以实现的寻址格式的特性。本章包括五小节，我们描述了各种API，这些API能给ns的寻址结构分配比特。正如第3章介绍过的地址空间，它能被看作n个比特的邻近范围，这里的n可以随着模拟的地址需求的变化而变化。n的缺省值为16（正如由MAXADDRSIZE_定义的一样）。n的最大值设为32（定义为MAXADDRSIZE_）。地址的缺省值和最大值定义在~ns//tcl/lib/ns-default.tcl里。

地址空间由两部分组成：节点标志（node-id）和端口标志（port-id）。高位被分配为node-id或id_，剩余的低位则分配来组成port-id或者附在节点上的代理的标识。在高位中，多播被分配1比特。地址空间由32位比特组成，端口空间也由32位比特组成。高32位为node-id，MSB为多播，低32位为port-id。另外，地址空间也可以设置为层次格式，构成多级的寻址层次结构。我们将会像上面讲的一样来描述API设置不同格式的地址结构，以及扩展地址空间。本章描述的过程和函数可以在~ns/tcl/lib/ns-address.tcl, address.cc以及address.h中找到。

15.1 缺省的地址格式

缺省设置分配低32位为port-id，1个高位作为广播用，剩余的32高位用于node-id。在模拟器初始化过程中，设置缺省模式的地址格式的过程以如下代码被调用：

```
# 前置动作
set ns [new Simulator] ;                    # 初始化模拟
```

同时也可以显式地调用：

```
$ns set-address-format def
```

15.2 层次式的地址格式

有两个选项用来将一个地址设置为层次格式：缺省的和特定的。

15.2.1 缺省的层次式设置

缺省的层次式的node-id分为3级，3级上的比特数分别为（10，11,11）。层次式配置可以用下面的方法调用：

```
$ns set-address-format hierarchical
```

这种设置：32位为port-id，32位为node-id，分配了层次式的3个级别，比特数分别为（10,11,11），或者如果多播有效的话，则为（9,11,11）。

15.2.2 特定的层次式设置

第二个选项允许层次式地址设置为特定的级别数，每一个级别分配比特位数。API如下：
$ns set-address-format hierarchical <#n级> <#1级比特数> <#2级比特数> ....<#n级比特数>

一个配置的例子如下：
$ns set-address-format hierarchical 2 8 15

这里层次式地址指定了2个级别，分配8比特为第一个级别，15比特为第二个级别。

15.3 扩展的节点地址格式

注意：由于节点地址和端口地址空间都是32比特长度了，所以请注意这个命令已经过时了。

在地址空间需要更多的比特的情况下，扩展的地址API就可以被用上了：
$ns set-address-format expanded
该命令将地址空间扩展到了30位，分配22个高位给node-id，低8位给port-id。

15.4 扩展port-id域

注意：由于节点地址和端口地址空间都是32位长度，所以请注意这个命令已经过时了。

这个原语可以用在由于需要连接大量的代理到节点上去，因而需要扩展port-id的情况下。这个可以与上面解释的set-addres-format命令（选项不同）一起使用。这个命令的大意为：
expand-port-field-bits <#port-id的比特位>

如果所需的端口尺寸大小不能容纳下（比如，很多自由的比特位不可用），或者如果请求的端口大小小于或者等于已有的端口大小，expand-port-field-bits就会进行检查，并且在随后提交错误。

15.5 设置地址格式的错误

在下面的情况下，set-address-format和expand-port-field-bits都会返回错误：

- 如果指定的比特位数小于0
- 如果比特位置冲突（请求自由比特的邻近位数没有找到）
- 如果比特总位数超过MAXADDRSIZE_
- 如果expand-port-field-bits企图让端口位数小于或者等于已有的端口位数
- 如果层次级别的数目与比特位的数目不匹配
- （对每一级）特定的

15.6 命令一览

下面是在模拟脚本中用到的与地址格式相关的命令列表：

$ns_ set-address-format def

这个命令用于内部设置地址格式为它的缺省值 :低32位为port-id ,1高比特位为多播 ,剩余的31位高比特位为node-id。但是，这个API已经被新的节点API $ns_ node-config -addressType flat所代替。

$ns_ set-address-format hierarchical

这个命令用于设置地址格式为层次式配置，它由3个级别组成，每一级别分配8比特位，低32位为port-id。但是这个API已经被新的节点API $ns_ node-config -addressType hierarchical所代替。

$ns_ set-address-format hierarchical <levels> <args>

这个命令用于设置地址格式为特定的层次式设置。<levels>表示结构中层次式级别的的数目，而args为第一级别定义了比特的位数。一个例子是$ns_ set-address-format hierachical 3 4 4 16，这里4，4,和16定义了在地址空间中级别为1，2,3各自的比特数目。

$ns_ set-address-format expanded

这个命令现在已经过时不用了。这个命令用于将地址空间扩展到30位，分配高22位为node-id，而低8位为port-id。但是，由于现在的寻址长度有32位，这个命令已经过时了，譬如node-id字段就是32位长。

expand-port-field-bits <bits-for-portid>

这个命令现在过时不用了，与上面的命令相似。这个命令用于为port-id域扩展地址空间到<bits-for-portid>指定的比特数目。但是由于现在的端口长度有32位了，这个命令已经过时不用了。

第 16 章

NS中的移动网络

本章描述了无线模型，此模型最初源于CMU的 Monarch 工作组对ns的移动性扩展。本章分两大节，每个大节由数个小节构成。第一节介绍了CMU/Monarch 工作组最初提出的移动模型。在这一节，我们介绍了移动节点（mobilenode）的内部组成，路由机制以及构成移动节点的网络栈（network stack）的网络组件，包括信道（Channel），网络接口（Network-interface），无线传播模型（Radio propagation model），MAC 协议，接口队列（Interface Queue），链路层（Link layer）和地址解析协议模型（ARP）。本节还介绍了CMU的trace支持以及节点移动和流量场景文件的生成等。最初CMU模型仅允许单纯的无线局域网或者多跳（multihop）ad-hoc网络的模拟，随后对其的扩展才允许了无线和有线网络的联合模拟，并且引入了移动IP（MobileIP）。在第二节中，我们会介绍这些扩展。

16.1 NS的基础无线模型

无线模型本质上是由移动节点（MobileNode）作为核心，并附加各种支持特性而构成，以实现对多跳 ad-hoc 网络和无线局域网等的模拟。移动节点对象是一个分裂对象（split object）。在 C++中， MobileNode 类继承自 Node 类，关于 Node 的详细介绍参见第 5 章。因此一个基本的 Node 对象附加上无线和移动节点的功能就构成了 MobileNode，这些附加功能包括在给定拓补（topology）中进行移动的能力，以及通过无线信道接收和发送信号的能力。普通节点和移动节点之间的最大差别是移动节点并不是通过链路（Links）与其他节点或者移动节点相连接。在这一章中，我们将介绍 MobileNode 的内部属性，包括它的路由机制，DSDV，AODV，TORA，DSR 等路由协议，支持信道访问的网络栈的创建，并简要描述无线模拟中每个协议栈的组成结构，trace 支持以及移动/流量场景的建立。

16.1.1 移动节点：创建无线拓扑

MobileNode是基本的ns Node对象，但是又有其自己的特性，例如移动性，以及在信道上发送和接收信号等特点，使其能创建无线移动模拟环境。MobileNode类派生自Node类。MobileNode是一个分裂对象，其移动特性包括节点的移动，周期性位置更新，拓扑边界维护等都在C++中实现，而MobileNode内部的网络构件的组装（例如分类器classifiers,分发器dmux，LL，Mac，Channel等）则是在Otcl中实现。这一小节中所描述的函数和过程的相关源代码参见：
~ns/mobilenode.{cc,h}, ~ns/tcl/lib/ns-mobilenode.tcl, ~ns/tcl/mobility/dsdv.tcl,~ns/tcl/mobility/dsr.tcl, ~ns/tcl/mobility/tora.tcl.。例子参见：~ns/tcl/ex/wireless-test.tcl and ~ns/tcl/ex/wireless.tcl。

我们举的第一个例子使用具有3 个节点的简单拓扑，第二个则具有50 个节点。这些例子可通过键入以下代码运行。
$ns tcl/ex/wireless.tcl （或 /wireless-test.tcl）

目前支持的四个ad-hoc 路由协议分别是： DSDV（Destination Sequence Distance Vector，目的序列距离矢量），DSR（Dynamic Source Routing，动态源路由），TORA（Temporally ordered Routing Algorithm，临时按序路由算法），AODV（Adhoc On-demand Distance Vector，Adhoc按需距离矢量）。创建一个移动节点语句如下所述。注意，创建一个移动节点的旧API依据具体路由协议，例如
set mnode [$opt(rp)-create-mobile-node $id]

其中，

$opt(rp)

定义了“dsdv”，“aodv”，“tora”或者“dsr”，id是移动节点的索引。不过旧版本API 的这种用法很受抨击，因而新版本API创建一个移动节点的方法如下：

```
$ns_ node-config -adhocRouting $opt(adhocRouting)
                 -llType $opt(ll)
                 -macType $opt(mac)
                 -ifqType $opt(ifq)
                 -ifqLen $opt(ifqlen)
                 -antType $opt(ant)
                 -propInstance [new $opt(prop)]
                 -phyType $opt(netif)
                 -channel [new $opt(chan)]
                 -topoInstance $topo
                 -wiredRouting OFF
                 -agentTrace ON
                 -routerTrace OFF
                 -macTrace OFF
```

上面这些API使用给定的值对一个移动节点进行配置，包括：adhoc路由协议，网络栈，信道，拓扑，传输模型以及是否打开有线路由（需要有线--无线通信wired-cum-wireless场景），是否打开各层的trace（路由, mac, 代理）。如果使用了分层寻址（hierarchical addressing）方式，还需要提供节点的hier地址。更多关于这个命令（新的节点API的一部分）参见ns注释和文件的“ns节点重构和新的节点API”（“Restructuring ns node and new Node APIs”in ns Notes and Documentation）内容。

接下来通过以下的方法来真正创建移动节点：

```
for { set j 0 } { $j < $opt(nn)} {incr j} {
    set node_($j) [ $ns_ node ]
    $node_($i) random-motion 0 ;# 关闭随机移动模式
}
```

以上过程创建了一系列移动节点的（分裂）对象，同时根据配置，为每个移动节点对象创建了指定的adhoc路由协议，网络栈（包括LL，IFq，MAC以及具有天线的NetIF），并按照定义的传输模型将协议栈中的各个构件互联并连接到信道上。移动节点结构如图16.1 所示。

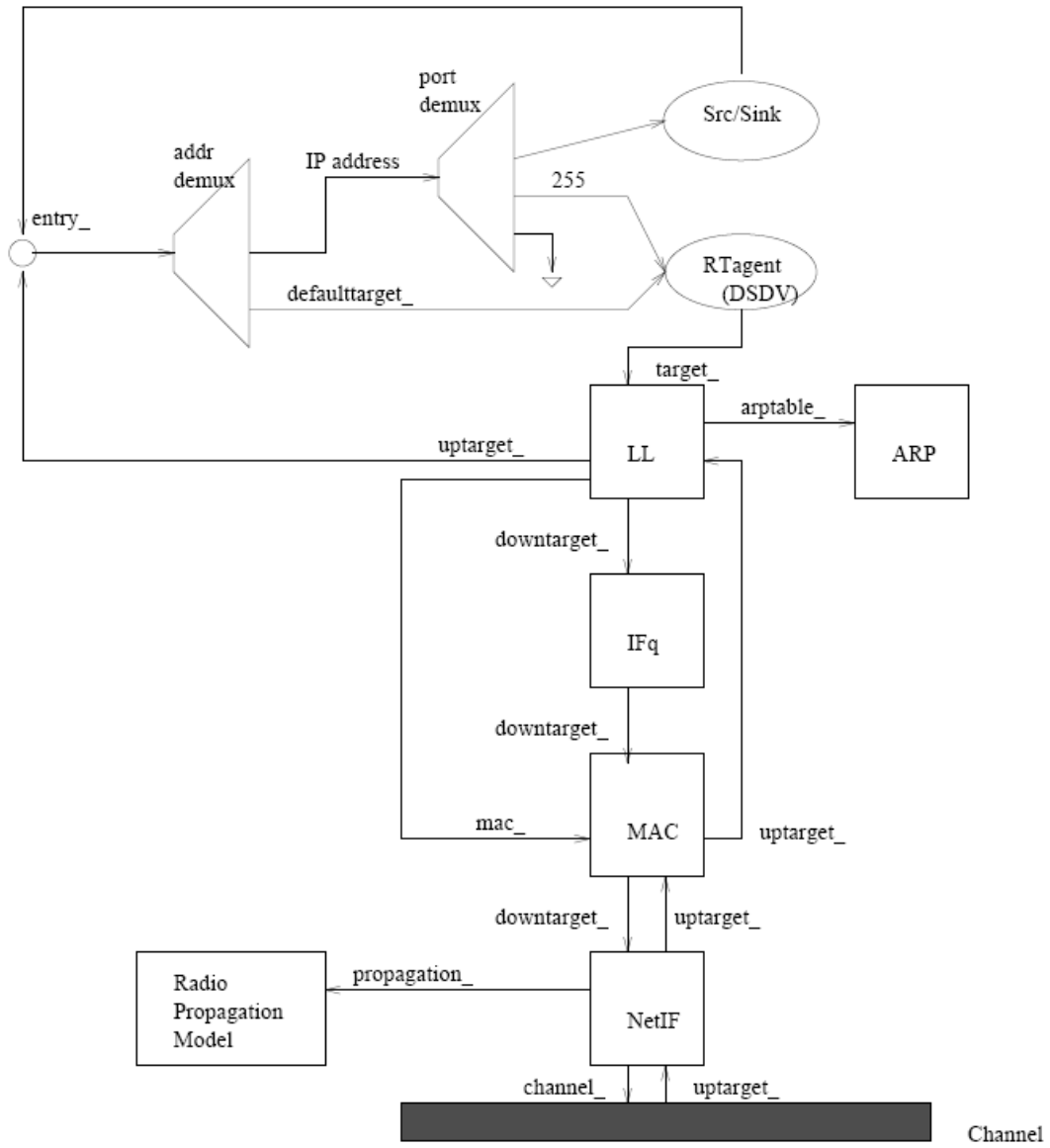


图16.1 CMU monnarch的ns无线扩展定义的移动节点示意图

使用DSR路由协议的移动节点结构与上述节点模型略微不同。SRNode类派生自MobileNode类。SRNode并不使用地址分发器（address demux）或者classifier，同时节点接收到的所有分组都默认交给DSR路由代理进行处理。DSR路由代理或者将接收的分组移交给端口分发器（port dmux），或者通过查询分组头得到源路由以转发该分组，或者发送路由请求/回复信息(若该分组是一新分组)。DSR路由代理的详细描述在16.1.5小节。一个SRNode节点的示意模型如图16.2。

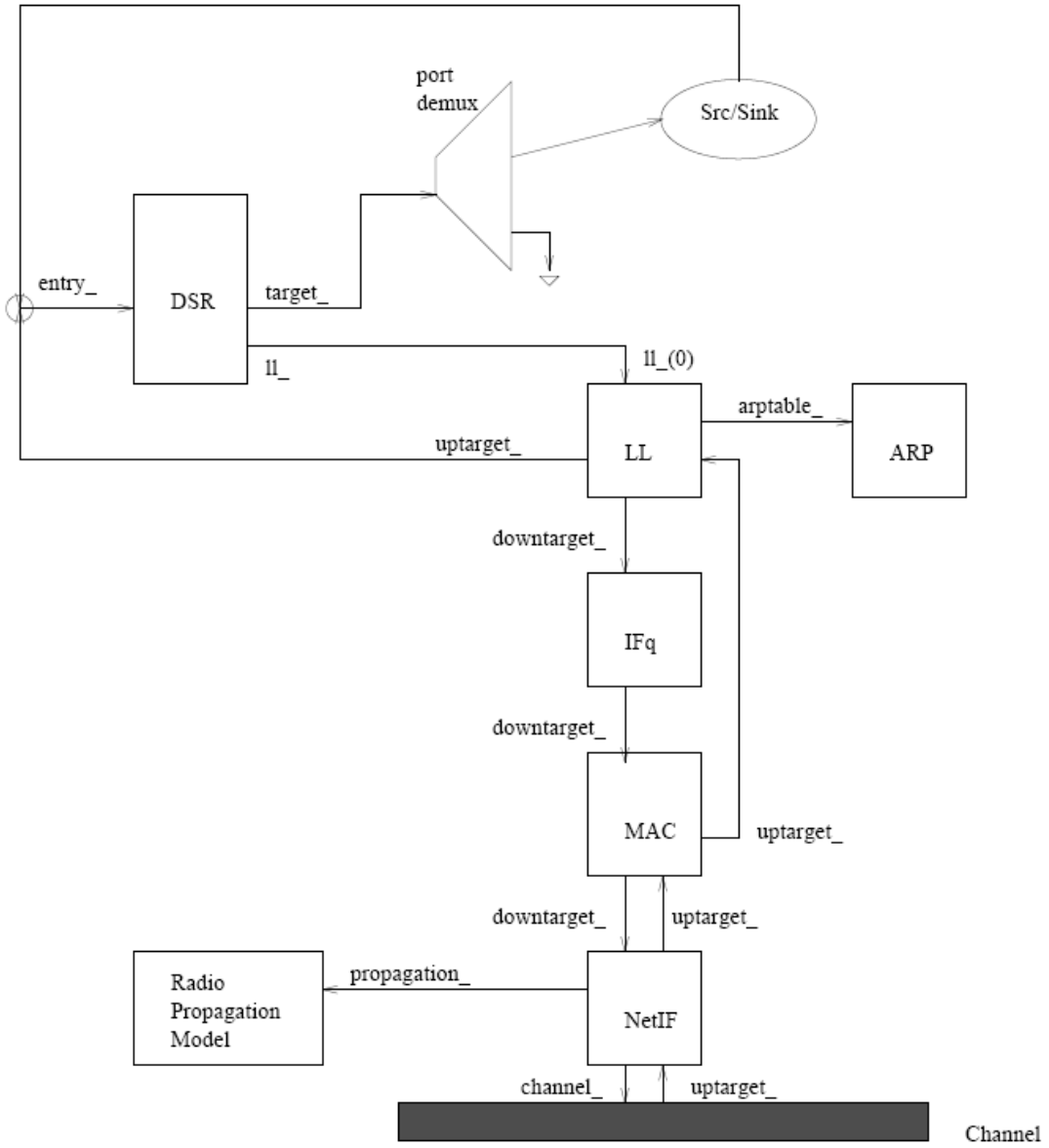


图 16.2 CMU monnarch 对 ns 无线扩展后的 SRNode 示意图

16.1.2 创建节点移动模型

移动节点可以在一个三维的拓扑中运动，然而实际上第三维（z 轴）并没有被使用。这样移动节点就假设为总是在一个 z = 0 的二维平面中运动，它的三维坐标（x，y，z = 0）随着它的运动不断的调整。有两种机制引发移动节点的运动。第一种方法需要明确指定节点的起始位置和终止位置。这些位置指令通常放在一个单独的场景文件中。

移动节点的起始位置和终止位置可以通过下面的 API 函数设定：

```
$node set X_ <x1>
$node set Y_ <y1>
$node set Z_ <z1>

$ns at $time $node setdest <x2> <y2> <speed>
```

在$time 时刻，节点会按设定的速度从起始位置（x1,y1）向目的位置（x2,y2）移动。

按照这种方法，节点移动更新（node-movement-updates）在任何需要知道节点位置的时候触发，例如邻居节点要求查询它们之间的距离时，或者接受上述 setdest 指令改变其目的位置和移动速度时。

上述 APIs 的示例移动场景文件参见~*ns*/tcl/mobility/scene/scen-670x670-50-600-20-0。这里 670x670 定义了拓扑的长和宽，该拓扑具有 50 个以最大速度 20m/s 移动的节点，且节点的平均停止时间为 600 秒。这些节点移动文件可以使用 CMU 的场景发生器来产生，场景发生器参见~*ns*/indep-utils/cmu-scen-gen/setdest。关于节点移动场景生成的详细描述参见 16.1.8 小节。

第二种方法是使用节点随机移动模型，使用语句如下：

```
$mobilenode start
```

该命令使节点从随机位置开始运动，并会定期改变节点的方向和速度。移动目的和速度是随机产生的。本方法的细节留给读者研究。移动节点的移动在 C++中实现，具体实现细节参考源文件~*ns*/mobilenode.{cc.h}。

不论使用何种方法实现节点移动，都需要在创建移动节点之前定义其拓扑图。通常平面拓扑通过指定长和宽来建立，可以使用下面的语句：

```
set topo [new Topography]
$topo load_flatgrid $opt(x) $opt(y)
```

其中 opt(x) 和 opt(y) 是模拟时拓扑的边界。

移动节点的移动情况可以使用如下过程来记录：

```
proc log-movement {} {
    global logtimer ns_ ns

    set ns $ns_
    source ../mobility/timer.tcl
    Class LogTimer -superclass Timer
    LogTimer instproc timeout {} {
        global opt node_;
        for {set i 0} {$i < $opt(nn)} {incr i} {
            $node_($i) log-movement
        }
        $self sched 0.1
    }
    set logtimer [new LogTimer]
    $logtimer sched 0.1
}
```

在以上情况下，移动节点的位置每 0.1 秒就会被记录一次。

16.1.3 移动节点的网络组件

移动节点的网络栈由一系列连接到无线信道的网络构件组成，这些构件包括LL，连接到LL的ARP模块，接口优先级队列（IFq），MAC层，netIF。这些组件由OTcl一起创建和组装。移动节点的add-interface()方法见~*ns*/tcl/lib/ns-mobilenode.tcl，源码如下：

```
#
# The following setups up link layer, mac layer, network interface
# and physical layer structures for the mobile node.
# 以下用于移动节点的创建链路层，mac层，网络接口以及物理层结构
#

Node/MobileNode instproc add-interface { channel pmodel
                        lltype mactype qtype qlen iftype anttype } {
        $self instvar arptable_ nifs_
        $self instvar netif_ mac_ ifq_ ll_
        global ns_ MacTrace opt
        set t $nifs_
        incr nifs_
        set netif_($t) [new $iftype] ;# net-interface
        set mac_($t) [new $mactype] ;# mac layer
        set ifq_($t) [new $qtype] ;# interface queue
        set ll_($t) [new $lltype] ;# link layer
        set ant_($t) [new $anttype]

        #
        # Local Variables    本地变量
        #
        set nullAgent_ [$ns_ set nullAgent_]
        set netif $netif_($t)
        set mac $mac_($t)
        set ifq $ifq_($t)
        set ll $ll_($t)

        #
        # Initialize ARP table only once.    只初始化ARP表一次
        #
        if { $arptable_ == "" } {
        set arptable_ [new ARPTable $self $mac]
        set drpT [cmu-trace Drop "IFQ" $self]
        $arptable_ drop-target $drpT
        }

        #
```

```
# Link Layer
#
$ll arptable $arptable_
$ll mac $mac
$ll up-target [$self entry]
$ll down-target $ifq

#
# Interface Queue
#
$ifq target $mac
$ifq set qlim_ $qlen
set drpT [cmu-trace Drop "IFQ" $self]
$ifq drop-target $drpT

#
# Mac Layer
#
$mac netif $netif
$mac up-target $ll
$mac down-target $netif
$mac nodes $opt(nn)

#
# Network Interface
#
$netif channel $channel
$netif up-target $mac
$netif propagation $pmodel ;# Propagation Model
$netif node $self ;# Bind node <---> interface     将节点和接口绑定
$netif antenna $ant_($t) ;# attach antenna   连接上天线

#
# Physical Channel
#
$channel addif $netif ;# add to list of interfaces  加入接口列表中

# ======================================================================
# Setting up trace objects      设置trace对象
if { $MacTrace == "ON" } {
        #
        # Trace RTS/CTS/ACK Packets
        #
        set rcvT [cmu-trace Recv "MAC" $self]
```

```
            $mac log-target $rcvT

            #
            # Trace Sent Packets
            #
            set sndT [cmu-trace Send "MAC" $self]
            $sndT target [$mac sendtarget]
            $mac sendtarget $sndT

            #
            # Trace Received Packets
            #
            set rcvT [cmu-trace Recv "MAC" $self]
            $rcvT target [$mac recvtarget]
            $mac recvtarget $rcvT

            #
            # Trace Dropped Packets
            #
            set drpT [cmu-trace Drop "MAC" $self]
            $mac drop-target $drpT
        } else {
            $mac log-target [$ns_ set nullAgent_]
            $mac drop-target [$ns_ set nullAgent_]
        }
        # ======================================================================
        $self addif $netif
}
```

以上方法创建了如图16.1所示的网络栈。

下面我们将简单介绍程序中的每个构件，希望将来能有来自CMU的更详细的文档可用。

Link Layer：移动节点使用的LL和第14章介绍的基本一样。唯一的区别是这里讲到的移动节点链路层连接了一个ARP模块，该ARP模块完成所有IP地址到硬件（Mac）地址的转换。对于那些向外发送（发送到信道）的分组，将由路由代理向下传递到LL。LL再将这些分组传递到接口队列。对于接收到的分组（从信道发上来），MAC层将分组向上传递到LL，LL再将分组传递到node_entry_ point。LL类实现参见文件~*ns*/ll.{cc,h}和~*ns*/tcl/lan/ns-ll.tcl。

ARP：地址解析协议（以BSD类型实现）模块从链路层接收请求。如果ARP有目的节点的物理地址，它就把该物理地址写入分组的mac头。否则它会广播一个ARP请求，并且将分组暂时缓存。对于每一个目的节点Mac地址暂时无法查到的分组，ARP模块都将会为其提供一个独立的缓冲区。在这种情况下如果又来更多到同样目的地的分组，旧的分组将被丢弃。一旦ARP知道了分组的下一跳目的节点的物理地址，该分组就被放入接口对列。ARPTable类实现参见文件~*ns*/arp.{cc,h}和~*ns*/tcl/lib/ns-mobilenode.tcl。

Interface Queue：PriQueue类被实现为一个优先级队列，它给每个路由协议分组赋予优先级并将分组插入队列头。它可以

对队列的所有分组进行过滤，删除那些有特定目的地址的分组。源码参见文件~*ns*/priqueue.{cc,h}。

Mac Layer：CMU实现了IEEE 802.11 DCF（distributed coordination function）Mac协议。它使用RTS/CTS/DATA/ACK的模式来发送单播分组和简单数据的广播分组，该实现同时使用了物理和虚拟载波监听。Mac802_11类的实现参见~*ns*/mac-802_11.{cc,h}。

Tap Agents：Trace Analysis Program，Tap类在mac.h中定义声明。Tap类用方法installTap()向mac对象注册它自己。在地址过滤之前，如果Mac协议允许，tap将得到mac层收到的所有分组分组。源代码参见：~*ns*/mac.{cc,h}。

Network Interfaces：网络接口层作为一个物理接口，移动节点可以通过这个物理接口访问信道。Phy/WirelessPhy类实现了无线共享介质接口。这个接口通过冲突（collision）和无线传输模块接收由其它节点接口发送到信道的分组。这个接口为每个分组写上和传输接口相关的meta-data（例如传输波长，传输功率）。接收节点的无线模块通过分组头中的meta-data来判断这个分组是否具有被接收/捕获/检测的最低功率。该模型与DSSS无线接口（Lucent WaveLan direct-sequence spread-spectrum ,朗讯网波直序扩频)类似。源代码参见~*ns*/phy.{cc.h} 和 ~*ns*/wireless-phy.{cc,h}。

Radio Propagation Model：无线传输模型，在较近传输距离使用Friss-space衰减模型（$1/r^2$），而较远距离使用Two ray Ground模型（$1/r^4$）。这种近似值是基于平地为镜面反射的假设。这部分代码位于~ns/tworayground.{cc,h}。（无线传输模型用来计算每个分组在到达接收节点时的信号强度。在移动节点的网络接口层有一个接收功率阈值，当接收到的分组的信号强度小于该阈值时，这个分组就被标记为error并被MAC层丢弃掉。ns中包含了3个无线信号传输模型：Free-apace,模型Two-ray ground reflection 模型和Shadowing模型——译者注）

Antenna：移动节点使用的是单一增益的全向天线，相关代码位于~ns/antenna.{cc, h}

16.1.4 移动网络的各种MAC层协议

在 ns 中，已实现的 MAC 层协议有两种，分别是 802.11 协议和 TDMA。这小节将对它们做一个简单的介绍。

802.11MAC 协议

详细信息见~*ns*/mac-802_11.{cc,h}。

基于前导（Preamble）的 TDMA 协议

注：本部分工作还处于初始阶段，一些实际的问题，例如对于多跳环境下前导阶段的争用以及时隙的复用等尚未考虑。

不像基于竞争的 MAC 协议（例如 802.11），一个 TDMA 协议节点们分配不同的时隙来发送或者接收分组。这些时隙的超集就是 TDMA 帧。

目前，ns 支持单跳，基于前导的 TDMA MAC 协议。一个 TDMA 帧由 preamble 和数据传输时隙组成。在 preamble 中，每一个节点用一个专门的子时隙来广播将要发送的分组的目的节点 ID。其他节点侦听 preamble 并记录它自己需要接收数据的时隙。像其他一般的 TDMA 协议（比如 GSM）一样，每一个节点有一个数据传输时隙以发送分组。

为了避免不必要的能量消耗，每个节点通过显示的调用节点 API set_node_sleep()将其无线电装置设置为开或者关。无线电装置仅在以下条件开启：在前导阶段（占一个时隙）以及需要发送或者接收分组时。

前导阶段被实现为一个中心数据结构 tdma_preamble_，该结构能被所有节点访问。在一帧的起始处，每个要发送分组的节点会将目的节点 ID 写入前导的子时隙中；前导阶段结束后，每一个节点会在各自的数据传输时隙发送分组，并且根据前导内容决定是否需要在其它时隙接收来自其它节点的分组。

用户可以配置以下参数：无线链路带宽 bandwith_,时隙长度 packet_slot_len_，以及节点数目 max_node_num_。更多细节参见~ns/mac-tdma.{cc,h}。

16.1.5 移动网络的各种路由代理

目前已经在 ns 中实现的四种 ad-hoc 路由协议分别为 DSDV，DSR，AODV 以及 TORA。本小节将逐一简要介绍。

DSDV

该协议中，路由消息的交换发生在相邻的移动节点之间（例如彼此处在对方通信范围内的移动节点）。路由更新可以是实时触发或者定期执行。当节点从一个邻居节点得到的路由信息会导致路由表的内容变化时，路由将被更新。如果要发送的分组的目的节点路由信息未知，则节点会将该分组缓存起来，同时发送路由查询消息，直到接收到目的节点的路由答复消息。这些等待路由信息的分组的数量受缓存门限（maximum buffer size）的控制，当缓存的分组数量超过这一门限时，分组将被丢弃。

作为目的地的移动节点收到的所有分组都由地址分发器（address dmux）直接路由给端口分发器（port dmux），端口分发器再将分组传给有关的目的代理。移动节点中的 255 端口就用于连接路由代理。移动节点也可以使用 classifier（或地址 dmux）的默认目标（default-target）。当节点没有在 classifier 中找到合适的目的对象时（这种情况发生在该分组的目的节点并非当前接收节点），就会把该分组交给 default-target，此时即为路由代理。接着路由代理设置好该分组的下一跳地址，并将其向下传给链路层。

路由协议主要是由 C++实现。DSDV 实现源代码参见目录~*ns*/dsdv 和 ~*ns*/tcl/mobility/dsdv.tcl。

DSR

这一部分将简要介绍动态源路由协议的功能。之前已经提到 SRNode 不同于 MobileNode。SRNode 的 entry_变量指向 DSR 路由代理（而通常的情况是先指向端口 dmux），这使得节点收到的所有分组都要交给路由代理处理。今后在实现数据分组捎带回执（piggy-backed）路由信息时便需要该模型，否则分组将不会流经路由代理。

DSR 代理会检查所有数据分组来获得源路由（source-route）信息，并根据这些路由信息转发分组。如果它在分组中找不到路由信息，但自己已经知道了到达目的节点的路由，则会设置分组的源路由；否则它将分组缓存起来并向外发送路由查询消息（route-query）。路由查询消息总是由那些不知道到达目的节点路由的分组所触发，且最初会被广播到所有邻居节点。那些能够提供到达目的节点路由的中继节点，或者目的节点自己，则会发送路由应答消息（route-reply）作为回应。DSR 代理会将所有以本节点为目的地的分组传递给它的端口 dmux，也就是说分组在到达端口 dmux 之前已经被路由代理处理过了，因此实际上 SRNode 的 255 号端口指向的是一个空代理（null agent）。

DSR 协议的实现源代码参见~*ns*/dsr directory 和~*ns*/tcl/mobility/dsr.tcl。

TORA

Tora 是一种基于“链路反转”算法（link reversal algorithm）的分布式路由协议。ns 中使用 Tora 时，每个节点为它的所有目的节点都运行一个独立的 TORA 拷贝。当节点需要到达某个目的节点的路由时，它就广播一个包含目的节点地址的查询（QUERY）消息。这个查询分组在网络内传播，直到被目的节点或者某个知道目的节点路由的中继节点所接收。接收到

查询分组的节点就会广播一个更新（UPDATE）消息，其中列出了该节点相对于目的节点的高度（height）。随着更新消息在网络中传播，所有节点都按照它接收到的更新消息修改自己相对于该目的节点的高度，使得自己记录的高度始终比邻居节点的该高度值大（即目的节点高度为 0，则其邻居节点相对于它的高度为 1，邻居的邻居节点高度为 2，以此类推——译者注）。这样做的结果是形成了从 QUERY 发起节点到目的节点的一系列有向链路（directed link）。如果一个节点发现某个目的节点无法到达了，则它将相对于该目的节点的高度设置为无穷大。当节点无法找到任何邻居节点相对于目的节点的高度为有限值时，它便启动新的路由发现过程。如果出现网络分离时，节点就广播一个清除（CLEAR）消息，重置所有路由状态，并删除无效的路由。

TORA 运行在 IMEP（Internet MANET Encapsulation Protocol）之上，IMEP 主要用来提供路由消息的可靠传送并可以在到邻居节点的链路改变时通知路由协议。IMEP 试图将 IMEP 和 TORA 的消息集成到一个分组中（称作块 block）以减小开销。为了检测链路状态并维护邻居节点列表，IMEP 会定期发送 BEACON 消息，接收到该消息的节点会回复一个 HELLO 消息。TORA 的实现参见 ns/tora 目录和 ns/tcl/mobility/tora.tcl。

AODV

AODV 是 DSR 和 DSDV 协议的混合，它具有 DSR 的路由发现（route-discovery）和路由维护（route-maintenance）功能，同时又使用了 DSDV 采用的逐跳（hop-by-hop）路由，序列号以及 beacons 消息。节点要查询给定目的节点的路由时会生成一个路由请求消息（ROUTE REQUEST）。中继节点在转发路由请求消息的同时，也建立起了一条它自己到达目的节点的反向路由。具有到达目的节点路由的节点在接收到路由请求消息之后，会生成一个路由应答消息（ROUTE REPLY），该消息包含了它到达目的节点的跳数。所有参与转发这个应答消息到源节点的中继节点也都建立了一条到达目的节点的转发路由。这条由源到目的之间各个节点建立起来的路由是一个逐跳路由，只有在源节点处的路由才包含完整路由信息。AODV 实现细节参见 ns/aodv 以及 ns/tcl/lib/ns-lib.tcl。

16.1.6 Trace支持

目前无线模拟的 trace 支持是用 cmu-trace 对象实现的。以后在 ns 中 cmu-trace 将扩展并与 trace 和 monitoring 支持合并，同时包括无线模块的 nam 支持。在此我们将简单介绍 cmu-trace 对象以及在无线模拟场景中如何实现对分组的跟踪。

cmu-trace 有 3 种类型，分别是 CMUTrace/Drop，CMUTrace/Recv 和 CMUTrace/Send，分别用来跟踪被丢弃的分组，被代理/路由器/mac 层/接口队列接收或者发送的分组。无线 trace 支持的方法和过程参见~*ns*/trace.{cc,h}和~*ns*/tcl/lib/ns-cmutrace.tcl。

以下 API 程序可以创建一个 cmu-trace 对象：

```
set sndT [cmu-trace Send "RTR" $self]
```

以上程序创建一个名为 sndT 的 CMUTrace/Send 类型的 trace 对象，用于跟踪由路由器发出的所有分组。trace 对象可以用于在 MAC 层，代理（路由代理或者其它代理），路由器或者其它 NsObject 中跟踪分组。

cmu-trace 对象 CMUTrace 类派生自 Trace 类。在第 26 章将详细介绍 Trace 类。CMUTrace 类定义如下：

```
class CMUTrace : public Trace {
public:
    CMUTrace(const char *s, char t);
    void recv(Packet *p, Handler *h);
    void recv(Packet *p, const char* why);
```

```
private:
        int off_arp_;
        int off_mac_;
        int off_sr_;

        char tracename[MAX_ID_LEN + 1];
        int tracetype;
        MobileNode *node_;

        int initialized() { return node_ && 1; }

        int command(int argc, const char*const* argv);
        void format(Packet *p, const char *why);

        void format_mac(Packet *p, const char *why, int offset);
        void format_ip(Packet *p, int offset);

        void format_arp(Packet *p, int offset);
        void format_dsr(Packet *p, int offset);
        void format_msg(Packet *p, int offset);
        void format_tcp(Packet *p, int offset);
        void format_rtp(Packet *p, int offset);
};
```

类型域(在 Trace 类定义中描述)用来区别不同的 trace 类型。在 cmu-trace 对象中 ,s 表示发送分组 ,r 表示接收分组 , D 表示丢弃分组。第四种类型 f 用来表示转发一个分组 (当该节点不是分组的起始节点)。与 Trace::format()方法类似 , CMUTrace::format()定义和指示了 trace 的文件格式 , 其源代码如下 :

```
void CMUTrace::format(Packet* p, const char *why)
{
        hdr_cmn *ch = HDR_CMN(p);
        int offset = 0;

        /*
        * Log the MAC Header   记录 MAC 分组头
        */
        format_mac(p, why, offset);
        offset = strlen(wrk_);

        switch(ch->ptype()) {

        case PT_MAC:
                break;

        case PT_ARP:
```

```
                format_arp(p, offset);
                break;

        default:
                format_ip(p, offset);
                offset = strlen(wrk_);

                switch(ch->ptype()) {

                case PT_DSR:
                        format_dsr(p, offset);
                        break;

                case PT_MESSAGE:
                case PT_UDP:
                        format_msg(p, offset);
                        break;

                case PT_TCP:
                case PT_ACK:
                        format_tcp(p, offset);
                        break;

                case PT_CBR:
                        format_rtp(p, offset);
                        break;
                ..........
                }
        }
}
```

以上方法根据不同的 trace 类型调用相应的 format 方法。所有的 trace 都写入缓存 wrk_中。同时，缓存的偏移量计数一直被保存并在不同的 trace 函数中传递。函数 format_mac()中定义了最基本的格式，可以用来跟踪所有的 pkt 类型。其他 format 函数则根据分组类型输出附加信息。Mac 层 format 如下：

```
#ifdef LOG_POSITION
        double x = 0.0, y = 0.0, z = 0.0;
        node_->getLoc(&x, &y, &z);
#endif

        sprintf(wrk_ + offset,
#ifdef LOG_POSITION
                "%c %.9f %d (%6.2f %6.2f) %3s %4s %d %s %d [%x %x %x %x] ",
#else
                "%c %.9f _%d_ %3s %4s %d %s %d [%x %x %x %x] ",
```

```
#endif
                op, // s, r, D or f
                Scheduler::instance().clock(), // time stamp  时间戳
                src_, // the nodeid for this node   该节点的节点 id
#ifdef LOG_POSITION
                x, // x co-ord   x 坐标
                y, // y co-ord   y 坐标
#endif
                tracename, // name of object type tracing   跟踪的对象类型名（如 MAC）
                why, // reason, if any     输出理由（如果有的话）

                ch->uid(), // identifier for this event     该事件的标识
                packet_info.name(ch->ptype()), // packet type    分组类型（如 cbr）
                ch->size(), // size of cmn header   公共分组头长度
                mh->dh_duration, // expected time to send data期望的发送数据的时间
                ETHER_ADDR(mh->dh_da), // mac_destination address      mac 目的地址
                ETHER_ADDR(mh->dh_sa), // mac_sender address      mac 源地址
                GET_ETHER_TYPE(mh->dh_body)); // type - arp or IP  类型 –arp 或者 IP
```

如果定义了 LOG_POSITION ,那么 mobilenode 的 X、Y 坐标也会被显示。上面的注释描述了 mac trace 中不同域的。IP 分组的额外 IP 头也需要添加到上述 trace 中。IP trace 如下：

```
sprintf(wrk_ + offset, "------- [%d:%d %d:%d %d %d] ",
                src, // IP src address             源 IP 地址
                ih->sport_, // src port number            源端口号
                dst, // IP dest address    目的 IP 地址
                ih->dport_, // dest port number   目的端口号
                ih->ttl_, // TTL value      TTL 值
                (ch->next_hop_ < 0) ? 0 : ch->next_hop_); // next hopaddress, if any.
                                                          //下一跳地址（如果有的话）
```

下面是一个 tcp 分组的 trace 示例：

```
r 160.093884945 _6_ RTR --- 5 tcp 1492 [a2 4 6 800] ------- [65536:0 16777984:0 31 16777984] [1 0] 2 0
```

例子中我们看到一个 TCP data 分组被 id 号为 6 的节点接收，该分组的 UID 为 5，并且携有一个大小为 1492 的公共分组头。mac 的跟踪细节显示了该分组为一个 IP 分组（ETHERTYPE_IP 定义为 0x0800，ETHERTYPE_ARP 为 0x0806），且接收节点的 mac-id 为 4。发送节点为 6，同时，在无线信道上发送该分组期望时间为 a2（二进制到十进制转换： 160+2 秒)。另外，还有关于源 IP 地址和目的 IP 地址的 trace 信息。源地址（使用 3 层地址 8/8/8）被译为 0 端口的地址串 0.1.0。同理，目的地址 0 端口的地址串 1.0.3。TTL 值是 31，目的地址与源地址相差 1 跳。另外，TCP 格式中 tcp 序列号信息为 1，ack 序号为 0。DSR，DSR，UDP/MESSAGE，TCP/ACK 和 CBR 分组类型等的格式描述参见~*ns*//cmu-trace.cc。

路由代理（TORA 和 DSR）也使用了其他 trace 格式来记录某些特定的路由事件如“发起（originating)”（在分组前面加上 SR 头）或者“超出一个源路由的限度”来表明源路由的一些问题。这些特定的事件 trace，以“S”（DSR 时）或者“T”（Tora 时）作为开头。TORA 代码参见~*ns*/tora/tora.cc，DSR 参见~*ns*/dsr/dsrgent.cc。

16.1.7 无线traces 的修订格式

在融合无线 trace 时提出了一个新的改进的 trace 方法，可用于在 ns 里追踪 cmu-trace 对象。修订后的 trace 支持仍可后向兼容以前的 trace 格式。使用以下命令可开启新的 trace 格式：

```
$ns use-newtrace
```

该命令需要在全局 trace 命令 $ns trace-all <trace-fd> 之前调用。原语 use-newtrace 通过设置一个名为 newTraceFormat 的模拟器变量来建立新的无线 trace 格式。目前这种新的 trace 支持仅对无线模拟有用，在不久的将会扩展到 ns 的其他构件。

以下是一个新 trace 格式的示例：

```
s -t 0.267662078 -Hs 0 -Hd -1 -Ni 0 -Nx 5.00 -Ny 2.00 -Nz 0.00 -Ne
-1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.255 -Id -1.255 -It
message -Il 32 -If 0 -Ii 0 -Iv 32
s -t 1.511681090 -Hs 1 -Hd -1 -Ni 1 -Nx 390.00 -Ny 385.00 -Nz 0.00 -Ne
-1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.255 -Id -1.255 -It
message -Il 32 -If 0 -Ii 1 -Iv 32
s -t 10.000000000 -Hs 0 -Hd -2 -Ni 0 -Nx 5.00 -Ny 2.00 -Nz 0.00 -Ne
-1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.0 -Id 1.0 -It tcp -Il 1000 -If
2 -Ii 2 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 0 -Po 0
r -t 10.000000000 -Hs 0 -Hd -2 -Ni 0 -Nx 5.00 -Ny 2.00 -Nz 0.00 -Ne
-1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.0 -Id 1.0 -It tcp -Il 1000 -If
2 -Ii 2 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 0 -Po 0
r -t 100.004776054 -Hs 1 -Hd 1 -Ni 1 -Nx 25.05 -Ny 20.05 -Nz 0.00 -Ne
-1.000000 -Nl AGT -Nw --- -Ma a2 -Md 1 -Ms 0 -Mt 800 -Is 0.0 -Id 1.0 -It
tcp -Il 1020 -If 2 -Ii 21 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 1 -Po 0
s -t 100.004776054 -Hs 1 -Hd -2 -Ni 1 -Nx 25.05 -Ny 20.05 -Nz 0.00 -Ne
-1.000000 -Nl AGT -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.0 -Id 0.0 -It ack -Il 40
-If 2 -Ii 22 -Iv 32 -Pn tcp -Ps 0 -Pa 0 -Pf 0 -Po 0
```

新 trace 格式介绍：

上述新 trace 格式可以划分成下面的域：

事件类型：上述 trace 中，第一个域（同修订前的 trace 格式）描述了节点事件类型，事件类型为以下 4 种类型之一 ：

s send	发送
r receive	接收
d drop	丢弃
f forward	转发

通用标签：第二个域以“-t”开头，表示时间或全局设置

-t time 时间
-t * (global setting，全局设置)

节点属性标签：这个域表示节点属性如节点 id，trace 所在的层（如 agent，rounter 或者 mac）。标签以“-N”开头，列举如下 ：

-Ni: 节点 id
-Nx: 节点 x 坐标
-Ny: 节点 y 坐标
-Nz: 节点 z 坐标
-Ne: 节点能量等级
-Nl: trace 层次，如 AGT,RTR,MAC
-Nw: 事件理由，不同的丢弃分组理由如下：
"END" DROP_END_OF_SIMULATION
"COL" DROP_MAC_COLLISION
"DUP" DROP_MAC_DUPLICATE
"ERR" DROP_MAC_PACKET_ERROR
"RET" DROP_MAC_RETRY_COUNT_EXCEEDED
"STA" DROP_MAC_INVALID_STATE
"BSY" DROP_MAC_BUSY
"NRTE" DROP_RTR_NO_ROUTE i.e no route is available.
"LOOP" DROP_RTR_ROUTE_LOOP i.e there is a routing loop
"TTL" DROP_RTR_TTL i.e TTL has reached zero.
"TOUT" DROP_RTR_QTIMEOUT i.e packet has expired.
"CBK" DROP_RTR_MAC_CALLBACK
"IFQ" DROP_IFQ_QFULL i.e no buffer space in IFQ.
"ARP" DROP_IFQ_ARP_FULL i.e dropped by ARP
"OUT" DROP_OUTSIDE_SUBNET 例如基站接收到来自其领域之外的节点的路由更新消息时产生的丢弃事件

IP 层的分组信息：本域的标签以“-I”开头，如下所示：

-Is: 源地址.源端口号
-Id: 目的地址.目的端口号
-It: 分组类型
-Il: 分组大小
-If: 流 id
-Ii: 唯一 id
-Iv: ttl 值

下一跳信息：本域提供下一跳的信息，并以“-H”开头，如下所示：

-Hs: 本节点 id
-Hd: 去往目的节点路径上的下一跳节点 id

MAC 层的分组信息：本域提供了 MAC 层的信息，并以“-M”开头，如下所示：

-Ma: 持续时间
-Md: 目的节点以太网地址
-Ms: 源节点以太网地址
-Mt: 以太网类型

“应用层”的分组信息：应用层的分组信息包含 ARP，TCP 等应用类型，adhoc 路由协议类型如 DSDV，DSR，AODV。本域的标签以“-P”开头，如下所示：

-P arp 地址解析协议，如下所示：
-Po: ARP 请求/相应
-Pm: 源 MAC 地址
-Ps: 源地址
-Pa: 目的 MAC 地址
-Pd: 目的地址
-P dsr 动态源路由协议（Dynamic source routing），如下所示：
-Pn: 经过的节点数
-Pq: 路由请求标志
-Pi: 路由请求序列号
-Pp: 路由响应标志
-Pl: 响应长度
-Pe: 源路由起始->源路由目的地 src of srcrouting->dst of the source routing
-Pw: 错误报告标志
-Pm: 错误数量
-Pc: 报告目标
-Pb: 链路 a->链路 b 的链路错误
-P cbr 恒定的比特率（Constant bit rate），CBR 相关标签及解释如下：
-Pi: 序列号
-Pf: 分组已转发次数
-Po: 理想转发次数
-P tcp 关于 TCP 流的信息由下面的标签给出:
-Ps: 序列号
-Pa: ack 号
-Pf: 分组已转发次数
-Po: 理想转发次数

这个域仍处于发展中，新的标签将会随着其他的应用而加入。

16.1.8 无线场景中节点移动和流量连接的生成

为了使用方便，通常把节点移动和流量连接定义放在单独的文件里。这些文件可以使用 CMU 的移动和连接发生器（gennerator）生成。这一小节，我们将分别对二者进行介绍。

节点移动模型

一些关于节点移动的例子参见~*ns*/tcl/mobility/scene/scen-670x670-50-600-20-*。文件中定义了一个 670*670m 的拓扑结构，并在此拓扑中定义了 50 个速度为 20m/s，暂停时间为 600s 的节点移动，并且给每个节点分配一个起始位置。有关节点之间跳数的信息被提供给中心对象“GOD”（XXX 但是这个信息为什么和在哪里被使用呢？期待来自 CMU 的答案）。每个节点都指定了将要运动的方向以及速度。

在~*ns*/indep-utils/cmu-scen-gen/setdest/目录中可以找到生成节点移动文件的发生器代码。在 setdest 下编译此文件可以得到一个可执行文件。带参数的 setdest 运行方式如下：

./setdest -n <num_of_nodes> -p <pausetime> -s <maxspeed> -t <simtime>
-x <maxx> -y <maxy> > <outdir>/<scenario-file>

需要注意的是，因为 ns 传统的节点索引从 0 开始，所以上述节点的索引要从 0 开始而不是像最初的 CMU 版本那样从 1 开始。

生成流量模式文件

一些关于流量模型的例子参见~*ns*/tcl/mobility/scene/cbr-50-{10-4-512, 20-4-512}。

流量发生器在~*ns*/indep-utils/cmu-scen-gen/中可以找到，分别命名为 cbrgen.tcl 和 tcpgen.tcl。这两个发生器分别用来生成 CBR 连接和 TCP 连接。

运行如下程序，可以生成 CBR 连接

ns cbrgen.tcl [-type cbr|tcp] [-nn nodes] [-seed seed]
[-mc connections] [-rate rate]

运行如下代码，可以生成 TCP 连接

ns tcpgen.tcl [-nn nodes] [-seed seed]

你需要将上述代码的结果输出到名为 cbr-* 或者 tcp-*的文件中。

16.2 CMU的移动模型扩展

之前提到过，最初的 CMU 无线模型支持对无线局域网和 ad-hoc 网络的模拟。不过为了使用 CMU 模型同时模拟有线和无线节点我们又加入了一定的扩展，称之为有线-无线通信（wired-cum-wireless）特性。另外 SUN 的 MobileIP（为有线节点实现）也被整合进无线模型来支持无线移动节点上的移动 IP。接下来的两个小节介绍了 ns 无线模型中的这两个扩展。

16.2.1 有线-无线通信场景

目前为止介绍的移动节点主要支持对多跳 ad-hoc 网络或者无线局域网的模拟。但是如果我们需要模拟多个无线局域网通过有线节点连接在一起，或者在无线节点上运行移动 IP 的场景时该怎么办呢？对 CMU 无线模型的扩展则允许了我们这样做。

wired-cum-wireless（有线-无线通信）场景所面临的主要问题是路由。ns 中路由信息是基于拓扑的连接关系生成的，例如节点如何通过链路（Links，此特指有线链路——译者注）彼此连接。然而移动节点里却没有链路的概念，它们使用自己的路由协议在无线拓扑中路由并传输分组。那么分组如何能在这两类节点之间交换呢？

由此一种称为基站节点（BaseStationNode）的节点对象被创建来作为无线和有线域的网关。BaseStationNode实质上是分层节点[16]（HierNode）和MobileNode的混合体。基站节点负责分组在无线域中的进/出，为了实现这一目标我们需要进行分层路由。

每个无线域以及它的基站会被分配一个唯一的域地址。所有以某个无线节点为目的节点的分组都会先到达那个无线节点所在域的基站，由基站最终将分组传输给目的节点（移动节点）。而移动节点则把向（无线）域之外发送的分组先路由给基站节点，基站则知道要如何把这些分组转发到目的（有线）节点。BaseStationNode 结构如图 16.3 所示。

在wired-cum-wireless场景中的移动节点要求支持分层寻址/路由，因此实际上MobileNode和BaseStationNode的结构非常的相似。不过，SRNode则只需要具有自己的分层地址而不需要支持分层路由[17]，因为它不需要任何的地址demux。

DSDV 代理在转发一个分组时会检查分组的目的地址是否在其（无线）子网之外，是的话它便会试图将分组转发给它的基站节点。如果没有找到达到基站的路由则丢弃分组，否则就将分组转发给基站方向路由的下一跳节点。之后基站根据其 classifier 将分组转发到有线网络。

DSR 代理在接收到一个发向子网外的分组时，如果到达基站的路由未知时会先发送一个路由请求以建立路由，此时分组被缓存起来以等待基站的路由应答。接收到路由应答后 DSR 代理就设置好分组头的路由信息并将其发往基站。基站的地址 demux 则会正确的将分组转发到有线网络。

[16] 分层路由和 HierNode 内部结构请参见 32 章。

[17]为了废除这些节点定义变化带来的差异，我们计划修订节点体系结构以支持更加灵活和模块化的节点结构，从而不必受限于特定类。

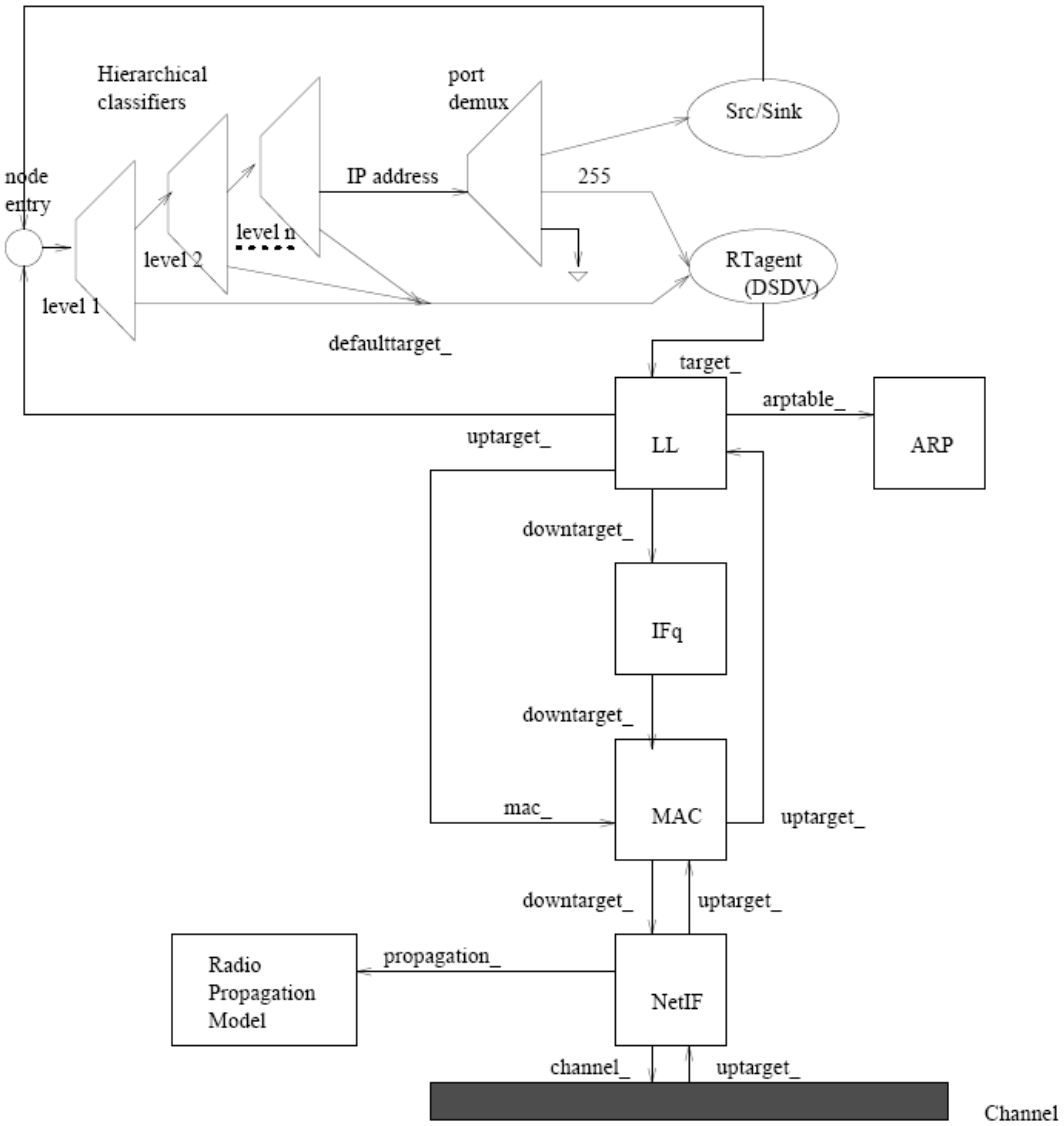


图 16.3 BaseStationNode 示意图

16.2.2 移动IP

ns 中无线模型的 wired-cum-wireless 扩展也为其支持无线 MobileIP 铺好了道路。Sun Microsystem(Charlie Perkins et al) 的 MobileIP 模型就是基于 ns 的有线模型（由 Node 和 Link 组成），因此不需要使用 CMU 的移动模型。

在此简要介绍无线 MobileIP 的实现。希望 Sun 将来能够提供更详细的说明文档。

移动 IP 场景由 HA（Home-Agents，本地代理）和 FA（Foreign-Agents，远程代理），以及在 HA 和 FA 之间运动的 MH（Mobile-Hosts，移动主机）组成。HA 和 FA 本质上就是之前介绍的基站节点，而 MH 则是在 16.1.1 节介绍的移动节点。MobileIP 扩展的方法和过程参见 *~ns*/mip.{cc,h}, *~ns*/mip-reg.cc, *~ns*/tcl/lib/ns-mip.tcl 和 *~ns*/tcl/lib/ns-wireless-mip.tcl。

HA 和 FA 节点被定义为具有一个 regagent_（注册代理）的 MobileNode/MIPBS 类，能够向移动节点发送 beacon，按照需要建立封装器（encapsulator）和解封器（descpsulator），并回复 MH 的请求。MH 节点被定义为

MobileNode/MIPMH 类，它同样具有一个 regagent_来接收和响应 beacon 并向 HA 或 FA 发送请求。图 16.4 为 Mobile/MIPBS 节点示意图。MobileNode/MIPMH 节点结构和该图相似，只是它没有任何封装器和解封器。SRNode 版本的 MH 则没有分层 classifier，而由 RA 代理作为节点的接入点（entry point）。SRNode 模型参见图 16.2。

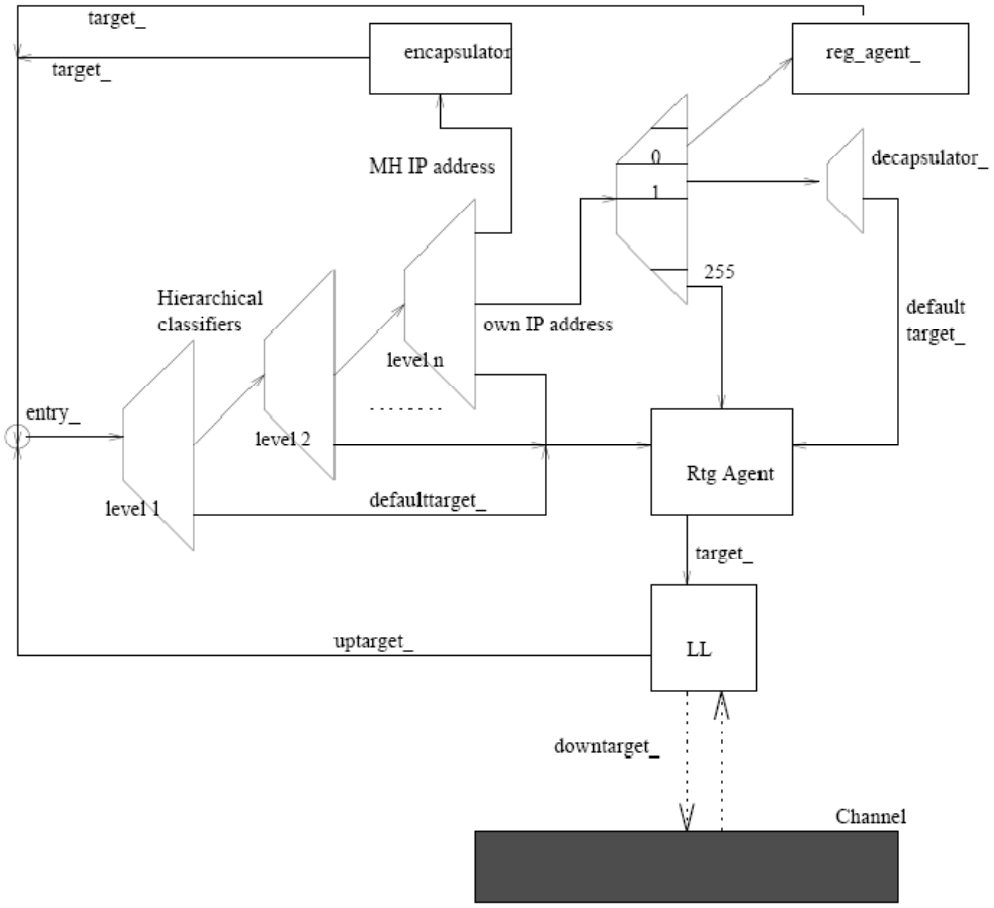


Figure 16.4: Schematic of a Wireless MobileIP BaseStation Node

图 16.4 无线 MobileIP 基站节点示意图

MobileNode/MIPBS 节点定期向 MH 广播 beacon 或者 ad（advertisement，广告）消息。移动节点发来的请求（solicitation）将在基站处产生一个 ad，该 ad 被直接发往请求的 MH。向外发送 beacon 的基站的地址将被 MH 所接收并用作该 MH 的 COA（care-of-address，关注地址）。因此当 MH 从当前域向外移动到外域时，它的 COA 也随之改变。基站在接收到一个 MH 的注册请求 reg_request（作为 ad 的回复）时将检查它自己是否为该 MH 的 HA，若不是，它会建立它的解封器并且将该 reg_request 转发给 MH 真正的 HA。

当基站是发出请求的 MH 的 HA 但 COA 不匹配时，基站就建立一个封装器并把注册请求回复（reg-request-reply）发回给向它转发这个 reg_request 的 FA（以 MH 请求中的 COA 为地址）。 这样所有以该 MH 为目的地址的分组在到达 HA 后，都会经过封装器以隧道（tunneled）的形式先发往 FA（以 COA 为地址），而非直接发往目的 MH。封装器将 IP 分组头（IP pkthdr）封装成 IPinIP 分组头（IPinIP hdr）。FA 的解封器接收到该分组，将其解封装后发送到目的 MH。

当 HA 的 COA 匹配时，说明该 MH 已经回到了 HA 的本地域，于是 HA 就删除之前（在该 MH 移动到其它域时）为其建立的封装器，并直接向 MH 发送回复。

MH 在接收不到基站的任何 ad 时就会向外发送请求。接收到 ad 时，它就将自己的 COA 修改为发送来 ad 的 HA/FA

的地址，并且向 COA 回复注册请求消息（reg-request）。开始时 MH 处于自己的 HA 范围内，并直接从 COA（即此时的 HA）处接收所有的发给它的分组。后来随着 MH 移动出 HA 的范围而进入到一个 FA 的外域，MH 的 COA 也由原来的 HA 变为 FA。然后之前的 HA 会建立一个封装器以便把所有发往 MH 的分组转发给 FA ,而 FA 则将这些分组解封装后发送给 MH。MH 想要发送到有线域的数据总是先交由它当前的 COA 进行路由转发。无线 mobileIP 场景的例子可以参见 *~ns*/tcl/ex/wireless-mip-test.tcl。该模拟由 HA 和 FA 以及在它们之间移动的 MH 组成。HA 和 FA 都分别一端连接到有线域而另一端连接到无线域。在 MH 和一个有线节点之间建立了 TCP 流。

16.3 NS老版本（2.1b5 或其后的）的代码融合到目前版本（2.1b8）的改动列表

由 David Johnhson 的 Monarch 项目所开发的 CMU-wireless 模型于 1998-99 年融合到了当时的 ns-2.1b5 版本中。从那时开始 Monarch 的 ns 版本和我们 ISI 的便差异不大了。最近我们引入了 Monarch 开发的一个新版 DSR 到 ns 中，并在此过程中按照需要为代码融合创建了一个改动列表。希望该列表对那些之前工作在旧版本 ns 上且想要把成果融合进现在版本（ns-2.1b8）的人有帮助。

以下列出将 cmu 版本的 ns（2.1b5）融合到当前版本（2.1b8）所需要做的修改。每条修改都在其后简要说明了修改理由。

访问分组头 pkt hdr 的方法从
(hdr_sr *)p->access(off_sr)
改为为每个 hdr 定义一个静态访问方法，如
hdr_sr::access(p)

hdr_sr 类的静态方法 access()定义如下
```
inline static hdr_sr* access(const Packet* p)
return (hdr_sr*)p->access(offset_);
```

理由：这个改动避免了到处使用类转换。

访问 hdr 方法的改变，使得不再需要显式的绑定 hdr offset 值。这些操作现在是在为每个 hdr 类创建 tcl 链接时完成。因此如下代码需要删除。
```
bind("off_SR_", &off_sr_);
bind("off_ll_", &off_ll_);
bind("off_mac_", &off_mac_);
bind("off_ip_", &off_ip_);
```

AF_枚举类型现在被 NS_AF_代替如下
```
enum ns_af_enum NS_AF_NONE, NS_AF_ILINK, NS_AF_INET ;
```

理由：避免分组头在 ns 和 OS 间产生冲突。

IP hdr（dst/src）地址域之前是 int 现在定义为叫做 ns_addr_t 的结构体，包括两个 int 成员 address_和 prot_。以此如下代码

iph->src()
需要改为
iph->saddr() & iph->sport();
同样类似代码
dst_ = (IP_BROADCAST « 8) | RT_PORT
需要改为
dst_.addr_ = IP_BROADCAST;
dst_.port_ = RT_PORT;

理由：扩展支持了 32 bit 地址。

hdr_sr 类中的 addrs_成员具有一个独立的函数来返回它的值。因此将 hsr.addrs 替换为调用 hsr.addrs()。

理由：addrs_ 现在是一个 private 变量，由 public 函数 addrs()访问。

所有使用<>来引用路径的 include 语句现在都使用”“。因此
<cmu/dsr/dsragent.h>
改为
"cmu/dsr/dsragent.h"

tcl 命令“ip-addr”改为“addr”。
加入像“node”，“port-dmux”以及“trace-target”等新的 tcl 命令。

理由：用于支持 mobileIP 和 wired-cum-wireless 模拟。

现在需要对地址进行 string 到 int 型的转换，因此
atoi(argv[2])
改为
Address::instance().str2addr(argv[2])

理由：用于支持分层寻址/路由。

数组 packet_name[]改为 packet_info.name()

理由：为了删除定义分组类型时的一大堆#defines 语句，现在使用一个枚举类型 packet_t 来描述 ns 中所有的分组类型。所有传入的分组都被 channel 标记为 UP，这些分组在被再次发送到网络前被代理标记为 DOWN。

现在调用 logtarget->pt_->buffer 替代 logtarget-> buffer。

理由：该改动反映了对事件跟踪（event tracing）的支持。跟踪程序发展为分组跟踪和事件跟踪两种类型。Trace 类原本就支持分组跟踪，然而除了从 BaseTrace 类继承来的基础跟踪特性之外，pkt-tracing 还需要继承一些 Connector 类的特性。因此 pt_作为基础 trace 对象，展示了一个纯的 trace 对象所需要的跟踪功能。

以前使用 int 参数来描述丢弃分组的原因，现在改为使用 char*。因此需要用 string 格式定义不同的 pkt-drop 理由。

理由：用于提供更大的扩展性和灵活性。

linkHead 改为 dsrLinkHead。

理由：linkHead 与 ns 其它地方的命名发生冲突。

旧的 cmu 模型通过在所有分组中添加 incoming_标志来指明分组来自于更低层（例如 LL，MAC）。现在改为使用在 cmn_hdr 中加入的 direction_变量，该变量可以被设为 UP，DOWN，或者 NONE。所有分组在创建时都被设置为 DOWN 方向。

理由：过去两个标志都被使用，然而并真正不需要这样。因此现在 incoming_标志被 direction_替代。

16.4 命令一览

以下是无线模拟中使用命令的列表：

```
$ns_ node-config   -addressingType     <通常为平面或分层无线拓扑>
                   -adhocRouting       <adhoc 路由协议如 DSDV, DSR,TORA, AODV 等>
                   -llType             <链路层>
                   -macType            <MAC 类型如 Mac/802_11>
                   -propType           <传输模型如 Propagation/TwoRayGround>
                   -ifqType            <接口队列如 Queue/DropTail/PriQueue>
                   -ifqLen             <接口队列长度如 50>
                   -phyType            <物理层类型 Phy/WirelessPhy>
                   -antType            <天线类型 Antenna/OmniAntenna>
                   -channelType        <信道类型 Channel/WirelessChannel>
                   -topoInstance           <拓扑实例>
                   -wiredRouting       <是否开启有线路由，ON 或 OFF>
                   -mobileIP           <是否开启移动 IP 标志，ON 或 OFF>
                   -energyModel        <能量模型类型>
                   -initialEnergy   <单位焦耳 Joules>
                   -rxPower            <单位瓦特 W>
                   -txPower            <单位瓦特 W>
                   -agentTrace         <是否开启代理层 trace，ON 或 OFF>
                   -routerTrace <是否开启路由层 trace，ON 或 OFF>
                   -macTrace <是否开启 mac 层 trace，ON 或 OFF>
                   -movementTrace <是否开启移动节点移动记录，ON 或 OFF>
```

这些是用来配置一个移动节点的典型命令。要获得更多的关于此命令的说明（新节点 API 部分）可以参见 ns 注释和文件中章节标题为“Restructuring ns node and new Node APIs”的部分。

$ns_ node <optional:hier address>

这个命令被用来在节点配置完成后（像在节点配置命令里展示的那样）创建一个移动节点。如果节点使用分层寻址，则还需要节点的分层地址也作为命令参数。

$node log-movement

这个命令过去用于启用移动节点的移动日志，现在已被$ns_node-config -movementTrace <ON or OFF>替代。

$create-god <num_nodes>

该命令被用来创建一个 GOD 实例。移动节点的数目作为参数传递给 GOD，用于创建一个存储拓扑连接信息的矩阵。

$topo load_flatgrid <X> <Y> <optional:res>

该命令初始化拓扑图的网格。<X> 和<Y>是拓扑的 x-y 坐标并规定了网格的大小。网格清晰度 resolution 作为<res>参数被传入。通常使用 1 作为默认值。

$topo load_demfile <file-descrptor>

用于读取 DEMFile 对到拓扑。DEMFile 参见 *ns*/dem.cc,.h。

$ns_ namtrace-all-wireless <namtrace> <X> <Y>

该命令用于初始化 namtrace 文件。namtrace 用于记录节点移动情况以便用 nam 工具展示。无线拓扑的 X,Y 坐标为 namtrace 文件的描述符，并作为参数通过这个命令传递。

$ns_ nam-end-wireless <stop-time>

这个命令通过使用<stop-time>来告诉 nam 模拟停止的时间。

$ns_ initial_node_pos <node> <size>

这个命令定义了节点在 nam 中的初始位置。<size>表明了在 nam 中节点的大小。这个功能必须在移动模型被定义后被调用。

$mobilenode random-motion <0 or 1>

随机运动（Random-motion）用来启动移动节点的随机运动，在这种情况下，随机目的地被分配给节点。0 为关闭随机运动，1 为开启随机运动。

$mobilenode setdest <X> <Y> <s>

这个命令用于建立一个移动节点的目的地。移动节点开始以<s> m/s 的速度，向以<X>和<Y>为坐标的目的地移动。

$mobilenode reset

此命令用来重置节点中所有对象（例如 LL，MAC，phy 等网络组件）。

内部过程

下面是无线网络中使用的内部程序列表：

$mobilenode base-station <BSnode-hier-addr>

该命令用于 wired-cum-wireless 场景。它提供移动节点所在域的 base-station 节点的信息。wired-cum-wireless 场景的通常使用分级地址，所以这里地址也是分级的。

$mobilenode log-target <target-object>

<target-object>通常来说是一个 trace 对象，用于记录移动节点运动和它们的能量使用情况（如果提供了能量模式的话）

$mobilenode topography <topoinstance>

这个命令用于向拓扑对象提供移动节点的句柄。

$mobilenode addif

一个移动节点可以有不只一个网络接口。这个命令用于向节点传递网络接口句柄。

$mobilenode namattach <namtracefd>

这个命令用于把 namtrace 文件描述符<namtracefd>连接到移动节点。之后节点所有的 nam traces 都被写进这个 namtrace 文件。

$mobilenode radius <r>

半径<r>指明了节点的通信范围。所有进入以该节点为中心，半径为<r>的圆内的移动节点被认为是其邻居。这个信息通常由 gridkeeper 使用。

$mobilenode start

这个命令用来启动移动节点的移动。

第 17 章

NS中的卫星网络

本章介绍了在 ns 中进行卫星网络模拟的扩展。特别地，这些扩展使得 ns 能模拟以下几个方面：1）传统的多用户上/下行链路和不对称链路的同步轨道“弯管”（geostationary “bent-pipe”）卫星；2）带净荷处理（净荷再生或全分组交换）的同步轨道卫星；3）极轨道近地卫星星群（polar orbiting LEO constellations），如 Iridium（铱星）系统和 Teledesic 系统。这些卫星模型的主要目标在于利用 ns 来研究卫星系统中与网络相关的方面，特别地，如 MAC、链路层、路由和传输协议。

17.1 卫星模型概述

精确的卫星网络模拟需要详细的射频特性（干扰、衰退）、协议交互（例如在有检错码情况下链路产生残余脉冲误码的相互影响）和二阶轨道影响（轨道运动，重力异常等）的建模。然而，为了从网络的角度研究卫星网络的基本特性，我们可以抽象一部分特征。例如，为了研究 TCP 在卫星链路上的性能，可以采用一个近似的而非详细的信道模型，即信道性能可以用一阶的总分组丢失概率来刻画。这就是本章中介绍的模拟模型所采用的方法——创建一个基本框架来研究传输、路由和 MAC 层协议在由同步轨道卫星或极轨道近地卫星星群组成的卫星环境下的性能。当然，用户也可以扩展这些模型从而为某些特定协议层提供更加详细的模拟。

17.1.1 同步轨道卫星

同步轨道卫星在赤道上空 22,300 公里高度绕地球运行。卫星的位置根据其天底点 nadir point（在地球表面的子卫星点 subsatellite point）的经度指定。实际上，由于重力的变化，同步轨道卫星会在重力紊乱影响下漂移出指定的位置，这些影响暂时不能在 ns 中模拟。

目前可以模拟的同步轨道卫星有两种：（1）仅仅作为一个轨道中继器的传统“弯管”同步轨道卫星，在这种卫星系统中，所有的分组通过上行链路接收，然后通过 RF 射频频率发送到对应的下行链路，卫星节点对于路由协议是不可见的。（2）新的同步轨道卫星将增加基带处理能力，包括数字信号再生和飞行器上的快速分组交换。在模拟过程中，这一类型的卫星更像是具有分类器和路由代理的传统 ns 节点。

之前，用户可以通过使用传统的 ns 链路和节点模拟一条长延时链路的方式，来实现对同步轨道卫星链路的模拟。这些卫星扩展中与同步轨道卫星相关的一个关键增强之处在于能够模拟 MAC 层协议。现在用户可以在地球表面的不同位置定义多个终端，并且将这些终端连接到相同的上行和下行链路，系统中的传播延时（每一个用户稍微有所差别）可以被真实地模拟。另外，上行链路和下行链路可以分别定义（可能定义不同的带宽或者差错模型）。

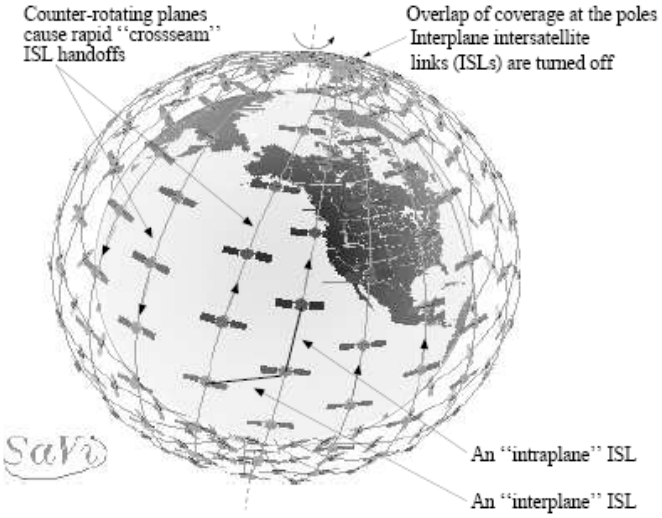


图 17.1 极轨道 LEO 星群示例。本图是采用明尼苏达大学几何学中心的 SaVi 软件包生成的。

17.1.2 近地轨道卫星

极轨道卫星系统（如 Iridium 和 Teledesic 系统）可以在 ns 中模拟。特别地，ns 支持对相邻轨道平面同步旋转的纯圆形平面轨道卫星的详细模拟。除了极轨道卫星星群，也可能对其他非同步轨道卫星星群进行配置与模拟（如 Walker 星群），当然，感兴趣的话用户也可以自己开发新的星群类以模拟其它类型的卫星星群。特别地，这需要定义新的星际链路切换（handoff）过程。

下面是当前可以模拟的卫星星群的参数：

- **基本星群定义：**包括卫星海拔（altitude），卫星数目，轨道平面数目，每个轨道平面的卫星数。

- **轨道：**轨道倾斜可以在从 0 到 180 度之间连续变化（当倾角大于 90 度时即为相对应的反向轨道）。轨道离心率和交点进动（nodal precession）没有被模拟。给定轨道平面中卫星之间的间隔是固定的。轨道平面之间的相对相位是固定的（尽管某些系统可能不控制轨道平面间的相位）。

- **星际链路（Intersatellite link，ISL）:** 对于极轨道星群，可以定义轨道面内（intraplane），轨道面间（interplane）和越缝（crossseam）三种 ISL。轨道面内 ISL 存在于相同轨道面内的相邻或者不相邻的卫星之间，并且永远不会失效或者切换。轨道面间 ISL 存在于相邻同步旋转轨道面内的卫星之间，这些链路在极点（高于表中“ISL 纬度阈值”）处会失效，因为在极点区域天线指向装置不能跟踪这些链路。与轨道面内 ISL 一样，轨道面间 ISL 也不会发生切换。越缝 ISL 可能存在于星群中反向旋转的轨道平面的卫星之间（这些轨道在拓扑上形成一个所谓的“缝”）。GEO ISL 同样也可以被定义为同步轨道卫星星群的星际链路。

- **星地链路（Ground to satellite links，GSL）:** 多个地面终端可以连接到单个 GSL 卫星信道，GEO 卫星的 GSL 链路是静止的，而 LEO 卫星的 GSL 链路周期性地发生切换。

- **截止高度角（Elevation mask）:** GSL 链路可以工作的最小仰角称为截止高度角。通常地，如果正在为终端服务的 LEO 卫星落在了终端的截止高度角之下，则该终端将寻找一个在截止高度角之上的新的卫星。终端根据用户指定的超时间隔检查是否需要发生切换。每一个终端独立地触发切换，也可以定义一个系统使得系统内的切换触发是同步的。

下面的表列出了Iridium[18]和Teledesic[19]系统的示例模拟脚本所使用的有关参数。

	Iridium	Teledesic
Altitude	780 km	1375 km
Planes	6	12
Satellites per plane	11	24
Inclination (deg)	86.4	84.7
Interplane separation (deg)	31.6	15
Seam separation (deg)	22	15
Elevation mask (deg)	8.2	40
Intraplane phasing	yes	yes
Interplane phasing	yes	no
ISLs per satellite	4	8
ISL bandwidth	25 Mb/s	155 Mb/s
Up/downlink bandwidth	1.5 Mb/s	1.5 Mb/s
Cross-seam ISLs	no	yes
ISL latitude threshold (deg)	60	60

表 17.1：用来模拟一个宽带版的 Iridium 系统和被提议的 288 颗卫星组成的 Teledesic 系统所需的模拟参数。两个系统都是极轨道卫星星群的示例。

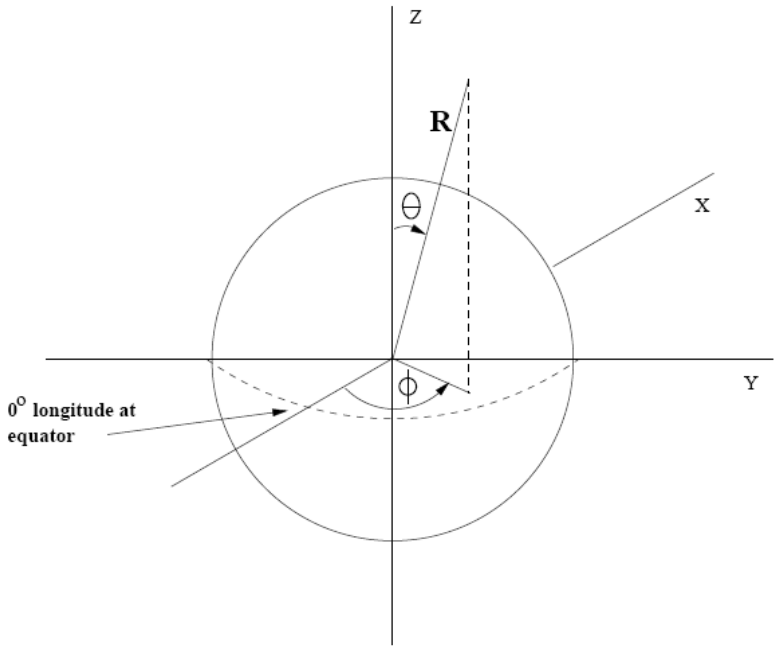


图 17.2 卫星节点使用的球坐标系统

[18]为了废除这些节点定义变化带来的差异，我们计划修订节点体系结构以支持更加灵活和模块化的节点结构，从而不必受限于特定类。

[19]Teledesic 星群的参数是易于被改变的 ;感谢 Teledesic 的 Marie-Jose Montpetit 自 1999 年 2 月起提供试验性参数。链路带宽没有必要精确给出。

17.2 使用卫星扩展

17.2.1 节点和节点位置

ns 中有两种基本的卫星节点类型：geostationary 和 non-geostationary 卫星节点。另外，terminal 节点可以放置在地球表面。正如后面的 17.3 节所解释的，这三种不同类型的节点实际上都是通过同一个 SatNode 类对象实现的，只是拥有不同的位置，切换管理器并连接不同链路对象。位置对象保持跟踪卫星节点的在坐标系统中位置，是模拟历经时间的函数。位置信息用来确定链路传播延时和链路切换的适当时间。5.3 节介绍的“node-config”程序为不同类型卫星节点的配置好节点生成器。

图 17.2 举例说明了球坐标系统和对应的笛卡尔坐标系统。坐标系统的中心为地心，Z 轴与地球自转的轴一致，（R，θ，φ）=（6378km，90°，0°）对应地球赤道上的 0° 经线（本初子午线，prime meridian）。

特别的，ns 中有一个卫星节点类 Node/SatNode，该类可以连接三种类型的位置 Position 对象。每一个 SatNode 和 Position 对象都是一个分裂 OTcl/C++对象，但是大多数代码在 C++中实现，下面是已有的三种类型的位置对象：

- Position/Sat/Term 终端是通过其纬度和经度来指定。纬度范围是[-90,90]，经度范围是[-180，180]，负值分别对应南半球和西半球。随着模拟时间的推移，终端随地球表面而移动。节点产生器能被用来创建一个连接了位置对象的终端，如下所示：

  ```
  $ns node-config -SatNodeType terminal \
  (其它节点配置命令写在这里...)
  set n1[$ns node]
  $n1 set position $lat $lon; # in decimal degrees
  ```

- Position/Sat/Geo 地球同步轨道卫星是通过其在地球赤道上的经度来指定。随着模拟时间的推移，地球同步轨道卫星以与地球自转相同的轨道周期在坐标系统中穿行。经度范围是[-180，180]度。正如我们下面将要描述的，存在两类地球同步轨道卫星：“geo”（带星上处理能力的卫星）和“geo-repeater”（弯管卫星）。节点产生器通过如下方式来创建一个连接了位置对象的地球同步轨道卫星：

  ```
  $ns node-config -SatNodeType geo (或 “ geo-repeater”) \
  (其它节点配置命令写在这里...)
  set n1 [$ns node]
  $n1 set-position $lon; #in decimal degrees
  ```

- Position/Sat/Polar 极轨道卫星有一个纯圆平面轨道，这个平面在坐标系统中是固定的。地球在这个轨道平面下面自转。因此，一个极轨道卫星在地球表面上的覆盖区轨迹同时包含东西方向的和南北方向。严格地讲，极位置对象（polar position object）可以用来模拟一个固定平面内任何圆形轨道的运动。我们在此使用“polar”这一术语是因为后面将使用这样的卫星来模拟极轨道卫星星群。

  卫星轨道通常使用下面六个参数来指定：海拔，半长轴（semi-major），离心率，升交点赤经（right ascension of ascending node），倾角和过近地点时间（time of perigee passage）。ns中的极轨道卫星有一个纯圆形轨道，因此我们将描述参数简化为三个：海拔，倾角和经度，第四个参数alpha制定卫星在轨道中的初始位置，如下所描述。**海拔**指

的是卫星轨道距地球表面的距离，单位是千米；**倾角**可以在[0,180]度之间变化，其中 90 度对应为纯极轨道，角度大于 90 度时对应为“反向”（retrograde）轨道。**升交点**指的是卫星覆盖区轨迹从南到北穿过赤道平面的交点。在这个模型中，**升交点经度**参数指的是卫星天底点从南到北穿过赤道时的地心经度[20]。升交点经度可以在[-180,180]度之间变化。第四个参数，**alpha**给出卫星在其轨道中的初始位置，从升节点开始。例如，alpha为 180 度指示卫星初始是在赤道上空，并由北往南穿过赤道。Alpha可以在[0,360]度之间变化。最后，第五个参数**plane**，用在创建极卫星节点时，在同一轨道平面的所有卫星被指定相同的平面索引，即plane参数。节点生成器通过如下方式来创建一个连接位置对象的极轨道卫星：

```
$ns node-config -satNodeType polar \
(其它节点配置命令写在这里...)
set n1 [$ns node]
$n1 set-position $alt $inc $lon $alpha $plane
```

17.2.2 卫星链路

卫星链路与 16 章所讲到的无线链路类似。一个卫星节点有一个或多个卫星网络接口栈，信道就连接在栈中的物理层对象上。图 17.3 说明了卫星节点的主要组件。卫星链路与 ns 无线链路主要区别在两个方面：1）传输接口和接受接口连接到不同的信道；2）没有 ARP 实现。目前，无线传输模块是一个占位符，这样用户可以根据需要增加更详细的差错模块；目前的代码不使用无线传输模块。

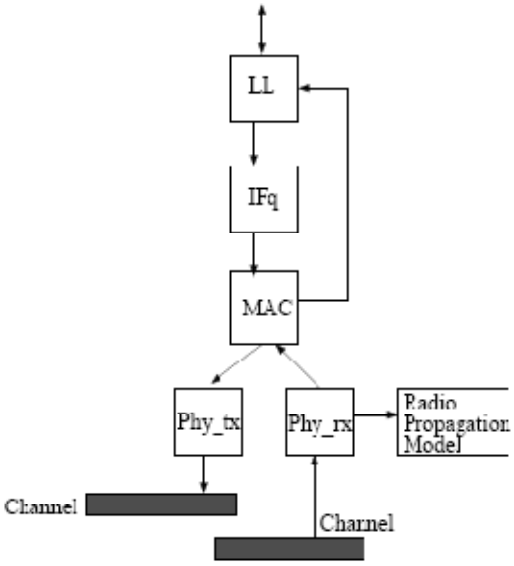


图 17.3 卫星网络接口的主要组件

网络接口可以用 Node/SatNode 类的如下方法实现：

```
$node add-interface $type $ll $qtype $qlim $mac $mac_bw $phy
```

[20]传统的，升交点的“赤经”（right ascension）是为卫星轨道指定的，赤经对应着黄经（celestial longitude）。在 ns 中，我们不关心天体坐标系统中的方向，所以我们指定用地心经度来替代。

方法 add-interface 返回一个索引值，在模拟程序中用此索引访问网络接口栈。按照约定，节点创建的第一个接口连接到卫星或者终端的上行链路和下行链路。以下的参数是必须的：

- **type** :可以指定的链路类型有 :geo 或者 polar ,分别对应来自终端到 geo 或者 polar 卫星的链路 ;gsl 和 gsl-repeater 以及 intraplane、interplane 和 crossseam ISLs 对应从卫星到终端的链路。这些类型用于模拟程序内部区别不同的类型的链路，但是从结构上来讲它们是类似的。
- **ll**：链路层类型（目前只定义了 LL/Sat 类型）。
- **qtype**：队列类型(例如 Queue/DropTail 类型)。任何一个队列类型都可能被使用，但是如果除了队列长度外还需要其他的参数，就需要修改 add-interface 方法以使其可以含有更多的参数。
- **qlim**：接口队列的长度，以分组为单位。
- **mac**：MAC 类型。目前，只定义了两种 mac 层类型： Mac/Sat 类（基本 MAC 类型）和 Mac/Sat/UnslottedAloha 类。Mac/Sat 用于只有一个接收者的链路（不需要做碰撞检测）。Mac/Sat/UnslottedAloha 是无时隙 Aloha 的一种实现。
- **mac_bw**：用于设置链路的带宽以此控制 MAC 层传输数据的速度乃至传输时间。用于计算传输时间的分组大小是公共分组头的 size()值和任何链路层分组头大小值 LINK_HDRSIZE 的总和。LINK_HDRSIZE 的缺省值是 16 字节（在文件 satlink.h 中可以进行设置）。传输时间被编码到分组头中给接收端的 MAC 使用(用来模拟接收一个完整的分组的时间)。
- **phy**：物理层类型，目前定义了两种物理物理层类型（Phy/Sat 类和 Phy/Repeater 类）。Phy/Sat 类仅在协议栈中上下传递信息，这和 16 章讲到的无线代码一样。无线传输模型可以连接到其上。Phy/Repeater 类将接收到的分组直接转发到发送接口。

通过如下方法可以将 ISL 增加到两个节点之间：

```
$ns add-isl $ltype $node1 $node2 $bw $qtype $qlim
```

这个方法创建了两个信道（Channel/Sat 类型）和两个节点上的网络接口，并且将信道和网络接口连接起来。此链路的带宽设置为 bw。链路类型（ltype）必须被指定是 intraplane, interplane, 或者 crossseam。

创建一个 GSL 包括增加网络接口和一条通往卫星的信道（这在下面的典型封装方法中将描述），接着定义了陆地节点上正确的接口，并将它们和卫星链路连接其来，代码如下：

```
$node add-gsl $type $ll $qtype $qlim $mac $bw_up $phy \
[$node_satellite set downlink_] [$node_satellite set uplink_]
```

这里的 type 必须是 geo 或者 polar，并且我们使用了卫星节点的实例变量 downlink_和 uplink_；因此，在这个方法调用之前卫必须创建星的 uplink 和 downlin。

默认情况下 ,卫星节点的节点生成器(在章节 5.3 中描述)将会创建一个给定类型的节点 ,并生成一个 uplink 和 downlink 接口，并根据接口选项创建并连接上（初始化）uplink 和 downlink 信道。

17.2.3 切换

卫星切换模拟是 LEO 卫星网络模拟的一个关键组成部分。要精确预测未来的 LEO 卫星系统中切换怎样发生是非常困难的，这是因为这一主题并没有在公开的文献中得到足够的重视。在 ns 卫星扩展中，我们建立了一定的切换准则，允许节点在需要切换的时候自主地监控。一个可选的方案是让所有的切换事件在整个模拟过程中都同步地发生——修改模拟器至这种工

作方式并不会很困难。

地球同步轨道卫星不涉及链路切换 ,但是有两种类型的极轨道卫星链路必须考虑切换 到极卫星 GSL 和 crossseam ISL。第三种类型的链路，轨道面间 ISL 没有切换，但是在高纬度地区将失效，如下所述。

连接到极轨道卫星的每一个终端都运行一个定时器，一旦超时，将触发 HandoffManger 检查当前卫星是否已经落到终端的截止高度角之下。如果是，切换管理器就将终端从卫星的上行和下行链路分离，并且在卫星节点列表中搜索另一个可能的卫星。首先，当前轨道平面的“下一个”卫星将被检查：每一个极轨道卫星节点的 Position 对象中都存储了一个指向下一个卫星的指针，该指针在进行模拟配置时通过 Node/SatNode 的实例过程“$node set_next $next_node”来设定。如果下一个卫星不合适，切换管理器将在剩下的卫星中继续搜索。如果发现一个合适的极卫星，切换管理器将终端的网络接口连接到卫星的上行和下行链路信道上，并重新启动切换定时器。如果没有发现合适的卫星，将重新启动定时器，并在定时器超时之后再尝试。一旦发生了链路改变，路由代理将被告知。

截止高度角和切换定时器间隔可通过 OTcl 进行设置：

```
HandoffManager/Term set elevation_mask_ 10; # degrees
HandoffManager/Term set term_handoff_int_ 10; # seconds
```

另外，可通过设置下面的变量使切换随机发生以避免相位影响：

```
HandoffManager set handoff_randomization_ 0 ; # 0 表示“否”，1 表示“是”
```

如果 handoff_randomization 为 1 则下一个切换时间间隔将是在(0.5*term_handoff_int_, 1.5*term_handoff_int_ ）区间内服从均匀分布的一个随机值。

Crossseam ISL 是唯一一类有切换的 ISL。Crossseam ISL 切换的标准是在相邻轨道平面上是否存在另一卫星，它相比当前给定卫星连接的卫星而言更为接近。同样，在极卫星的切换管理器中也运行了切换定时器，以决定星群何时检查切换的时机。Crossseam ISL 切换由两颗卫星之中轨道平面编号较低的卫星发起。因此可能会出现一个极轨道卫星有两条 crossseam ISL（连接不同的卫星）的短暂情况。卫星切换时间间隔通过 OTcl 设置，同样也可以是随机的：

```
HandoffManager/Sat set sat_handoff_int_ 10; # 秒
```

轨道面间 ISL 和 crossseam ISL 在靠近极点时失效，因为随着卫星向另一颗卫星靠近移动，链路的指向要求变得越来越高。链路的关闭通过以下参数来管理。

```
HandoffManager/Sat set latitude_threshold_ 70; # 度
```

示例脚本中这个参数的值是随意给出的，准确的值依赖于卫星硬件。一旦切换定时器超时，切换管理器检查其自身和对等卫星的纬度，如果其中任何一颗高于了纬度 latitude_threshold_度（南纬或北纬），链路将被释放直到两颗卫星都下降到这个阈值之下。

最后，如果 crosseam ISL 存在，则在中纬度区域可能会出现两颗卫星彼此过于接近的情况（如果轨道不是非常接近纯极轨道 ）。我们通过下面的参数检查这种轨道重叠情况的发生：

```
HandoffManager/Sat set longitude_threshold_ 10 ; # 度
```

同样，示例脚本中这个参数的值也是随意给出的。如果两颗卫星接近到经度差小于 longitude_threshold_度，它们之间的链路将被释放。这个参数缺省为不可用(设为 0),所有与卫星相关的变量的缺省值都可以在 ~ns/tcl/lib/ns-sat.tcl 中找到。

17.2.4 路由

当前的路由状态还是不完善的。理想情况下，人们可以在卫星链路上运行所有已存在的 ns 路由协议。然而，大多数已存在的路由协议在 OTcl 中实现时需要用到传统的 ns 链路。在这一领域的任何贡献都是受欢迎的，但不幸的是，这并非是一些细微改动就能实现的。

在此情况下，当前已实现的路由协议除了完全用 C++实现以外，其他的都类似于第 29 章描述的 Session 路由。一旦发生任何拓扑变化，一个集中路由 genie 将决定全网拓扑，为所有节点计算新的路由，并使用这些路由在每一个节点上构建一个转发表。当前，slot 表由每一个节点上的路由代理保存，不是以当前节点为目的地的分组将缺省发往这个路由代理。对于已建立起路由的每一个目的节点，转发表中都包含一个去往该目的节点的对应链路的头指针。就像第 29 章所描述的，我们提醒用户，这种类型的集中路由会导致较小的因果关系错误。

路由 genie 是一个 SatRoutrObject 类，通过下面的 Otcl 命令创建和调用：

```
set satrouteobject_ [new SatRouteObject]
$satrouteobject_ compute_routes
```

应该在模拟器中所有的链路和节点都已经实例化以后再调用 compute_routes。像 Scheduler 一样，模拟过程中也只有一个 SatRouteObject 的实例，通过 C++中的一个 instance 变量来访问。例如，在拓扑改变之后的重新计算路由调用如下：

```
SatRouteObject::instance().recompute() ;
```

尽管当前使用集中路由，但是让每个节点具有一个使用分布式路由的路由代理的设计已经基本完成。路由分组可以被发送到每一个节点的 255 端口。分布式路由正确工作的关键之处在于对于路由代理能够确定分组是从哪一个链路来的。这项工作由每一个链路包含的 NetworkInterface 类对象来完成的，该对象唯一标识分组到来的链路。一个助手函数 NsObject* intf_to_target(int label)可以被用来返回给定标识对应链路的头部。通过并行使用路由代理可以获得移动扩展，感兴趣的读者可以从这些例子着手，研究如何在这个框架中实现分布式路由。

最短路径（shortest-path）路由算法使用链路当前的传播延时作为开销矩阵来进行计算。也可以只利用跳数而不是传播延时来计算路由；如果要这样做，则设置下面的变量缺省为“false”：

```
SatRouteObject set metric_delay_ “true”
```

最后，对于每一个大型拓扑（如 Teledesic），集中路由代码将需要非常长的运行时间，这是因为一旦拓扑发生改变，都将对所有的节点对执行最短路径算法，即使当前网络上没有数据要传输。为了在有大量卫星和 ISL，而没有多少数据传输的模拟中加快速度，可以关闭 handoff-driven 而开启 data-driven 路由计算。在 data-driven 计算下，路由只有在有分组发送时才计算，此外，它执行单源（single-source）最短路径算法（只对有分组发送的节点）代替所有节点对最短路径算法。下面的 Otcl 变量可以配置这一选项（缺省设置为“false”）：

```
SatRouteObject set data_driven_computation_  "false"
```

17.2.5 Trace支持

卫星节点和链路的 trace 文件非常类似于第 26 章描述的传统 ns tracing。特别地，SatTrace 对象（SatTrace 类派生自 Trace 类）被用作记录节点的经度和纬度（对于卫星节点，纬度和经度对应于卫星天底点的经度和纬度）。

例如，正常情况下，从节点 66 到节点 22 的链路上的一个分组可能被记录为：

```
+ 1.0000 66 26 cbr 210 ------- 0 66.0 67.0 0 0
```

但是在卫星模拟中，除了上述信息外，还追加了位置信息：

```
+ 1.0000 66 26 cbr 210 ------- 0 66.0 67.0 0 0 37.90 -122.30 48.90 -120.94
```

在这种情况下，节点 66 的位置是北纬 37.90 度，西经 122.30 度，而节点 26 是一个 LEO 卫星，其子卫星点是北纬 48.90 度，西经 120.94 度。（负数纬度对应为南纬，负数经度对应为西经）。

另外还增加了一个 Trace/Sat/Error 类，该类跟踪所有被差错模型设置为错误的分组。如下所示，差错 trace 记录由于差错而丢弃的分组：

```
e 1.2404 12 13 cbr 210 ------- 0 12.0 13.0 0 0 -0.00 10.20 -0.00 -10.00
```

特定情况下，有可能发生一个卫星节点产生的分组不能被转发（如 sat-mixed.tcl 中所示），这种情况在 trace 文件中将显示为丢弃，该分组的目的端字段设为-2，坐标设为-999.00：

```
d 848.0000 14 -2 cbr 210 ------- 1 14.0 15.0 6 21 0.00 10.00 -999.00 -999.00
```

上面的例子指示节点 14 试图发送一个分组到节点 15，但是没有找到有效的路由。

为了在模拟器中跟踪所有的卫星链路，在实例化节点和链路之前使用下面的命令：

```
set f [open out.tr w]
$ns trace-all $f
```

接着，在所有的节点和链路都创建完毕后（且所有的差错模型都已插入，如果有的话），使用下面一行命令跟踪所有卫星链路：

```
$ns trace-all-satlinks $f
```

特别地，这将在所有卫星链路的链路层队列中放置 trace，并且在 mac 和链路层之间放置接收 trace 来接收分组。如果只在特定节点的特定链路上跟踪，则用户可以使用如下命令：

```
$node trace-inlink-queue $f $i
$node trace-outlink-queue $f $i
```

i 是被跟踪的接口的索引。

卫星 trace 对象的实现参见~ns/tcl/lib/ns-sat.tcl 和~ns/sattrace.{cc,h}。

17.2.6 差错模型

我们在第 13 章中已经介绍了 ns 的差错模型。这些差错模型可以控制分组依据各种概率分布产生错误。这些模型非常简单，并且没必要对应实际卫星信道真实的误码情况（特别对于 LEO 信道）。用户可以自由地定义更接近特定卫星环境的复杂的差错模型。

下面的代码给出了向链路中添加差错模型的示例：

```
set em_ [new ErrorModel]
$em_ unit pkt
$em_ set rate_ 0.02
$em_ ranvar [new RandomVariable/Uniform]
$node interface-errormodel $em_
```

上述例子将在节点$node 的第一个接口的接收路径中添加一个差错模型（特别地，该模型在 MAC 和链路层之间），第一个接口通常对应卫星或终端（如果只有一个上行和/或下行链路存在）的上行和下行链路接口。如果需要向不同的栈（通过 i 索引）添加差错模型，可以使用下面的代码：

```
$node interface-errormodel $em_ $i
```

17.2.7 其它配置选项

指定在时间 0 给出卫星的一个初始配置，则通过使用 time_advance_参数，可能从任意时间点开始执行卫星配置（实际上，这只对 LEO 卫星模拟有用）。在模拟运行过程中，这会将对象的位置设置为 Scheduler::instance().clock + time_advance_秒时刻的位置。

```
Position/Sat set time_advance_ 0; # 秒
```

17.2.8 nam支持

目前不支持 nam。为卫星网络模拟添加 nam 支持对所有有兴趣的贡献者都是开放的。

17.2.9 有线与无线节点的综合

最近（2001 年 11 月），ns 增加了连接传统的基于 OTcl 的有线节点和卫星节点的支持。本节将介绍这些代码的能力和局限性。

卫星代码（以及无线代码）正常情况下都是在 C++中执行所有路由模拟，而传统的 ns 代码使用 OTcl 和 C++代码混合

的形式。为了向后兼容，要想全面集成有线和无线代码是非常困难的。目前集成有线和无线代码策略是定义一个特别的网关节点（称作“基站”），使用分层路由，将单个基站节点布置在无线网络，并使基站的协议栈同时位于无线子网和有线子网。因为路由没有全面地集成，模拟的拓扑限制在每个无线子网仅有一个网关节点（也就是说，分组不能够从一个有线网关进入无线网络，而通过另一个有线网关离开）。

卫星/有线代码集成采用与此不同的策略。通过将节点配置选项$ns node-confi-wiredRouting 设置为 ON，C++代码中的卫星路由部分就被关闭，取而代之的是所有卫星拓扑改变将导致调用 OTcl 代码。因此，OTcl 中的 link_数组将在发生任何拓扑变化时被使用，基于 OTcl 的路由由此产生。这样做的代价是要更长的时间来进行大规模的模拟（如 Teledesic 系统），但是对于小规模的模拟，差别不是很明显。

关于如何使用这个新选项的示例脚本在~ns/tcl/ex/sat-wired.tcl 中详细给出，在卫星测试集合（satellite test suite）里的一个类似测试就运行了这个代码。此外，在~ns/tcl/ex 目录下的所有卫星示例脚本都可以通过使用$ns node-config -wiredRouting ON 选项转换成 OTcl 路由。然而，我们还是给出如下声明：

- 卫星的有线路由选项仅仅用（缺省）的静态路由：$ns rtProto Static 进行了测试。这行代码会在卫星拓扑发生任何改变时触发全局路由表的更新。
- 当 wiredRouting 为 ON 时，data_driven_computation_选项不能被设置为“true”。注意打开或关闭 data_driven_computation_选项将使得模拟输出有微妙的差别，这是因为路由是在不同时期计算的（而传播延时是不断变化的）。这个影响可以通过在 Iridium 的示例脚本~ns/tcl/ex/sat-iridium.tcl 中开关这个参数看出来。
- 在 trace 文件中，当一个分组由于“没有到主机的路由”（如当拓扑发生改变）而被丢弃时，trace 随 wiredRouting 是 OFF 还是 ON 稍微有所差别。当 wiredRouting 为 OFF 时，每个丢弃分组都记录一行，目的地址标识为“-2”。而当 wiredRouting 为 ON 时，有三个事件（进队列“+”，出队列“-”和丢弃“d”）对应同一个分组，并且目的地标识为“-1”。
- 在非常罕见的情况下，在执行过程中可能会出现“node out of range”的告警消息。这种情况出现在拓扑中一个节点断开了，而另一个节点试图发送分组给它。例如，可以试试在脚本~ns/tcl/ex/sat-mixed.tcl.中开启 wiredRouting 选项。发生这种情况的原因是因为路由表长度是随拓扑变化而动态计算的，如果一个节点断开了，它可能没有任何入口插入到路由表中（从而路由表的增长与节点数不相适应）。这个告警不影响实际的 trace 输出。
- 目前还没有尝试与无线或移动 IP 代码互进行操作。

17.2.10 示例场景

所有的示例脚本都可以在~ns/tcl/ex 目录下找到，包括：

- sat-mixed.tcl：一个极轨道和地球同步轨道卫星的混合模拟。
- sat-wired.tcl：与上一个脚本类似，但是演示了如何连接有线网络节点到卫星模拟中。
- sat-repeater.tcl：演示了简单“弯管”地球同步轨道卫星的使用，包括差错模型。
- sat-aloha.tcl：模拟了 100 个采用 mesh-VSAT 配置的终端，它们使用无时隙 Aloha MAC 协议来接入一个“弯管”地球同步轨道卫星。终端侦听它们自己的传输（在一段延时之后），如果在超时间隔之后没有成功收到他们自己的分组，它们将执行指数回退，并重传丢失的分组。存在三个变种：basic，basic_tracing 和 poisson。这些变种的详细描述在该脚本文件的文件头注释中给出。
- sat-iridium.tcl：模拟了一个宽带 LEO 星群，该星群的参数与 Iridium 星群的参数类似（需要脚本 sat-iridium-links.tcl，sat-iridium-linkswithcross.tcl，和 sat-iridium-nodes.tcl 的支持）。
- sat-teledesic.tcl：模拟了一个宽带 LEO 星群，该星群的参数与被提议的 288 颗卫星组成的 Teledesic 星群的参数类似（需要脚本 sat-teledesic-links.tcl 和 sat-teledesic-nodes.tcl 的支持）。

此外，还有一个测试集合脚本试图同时检验许多特征，该脚本可以在~ns/tcl/test/testsuite-sat.tcl 中找到。

17.3 实现

卫星扩展实现的代码可以在 ~ns/{sat.h,sathandoff.{cc,h}, satlink.{cc,h}, satnode.{cc,h}, satposition.{cc,h}, satroute.{cc,h}, sattrace.{cc,h}}，以及~ns/tcl/lib/ns-sat.tcl 中找到。几乎所有的机制都是在 C++中实现的。

本节中，我们关注某些关键组件的实现，即链表的使用，节点结构和详细的卫星链路结构。

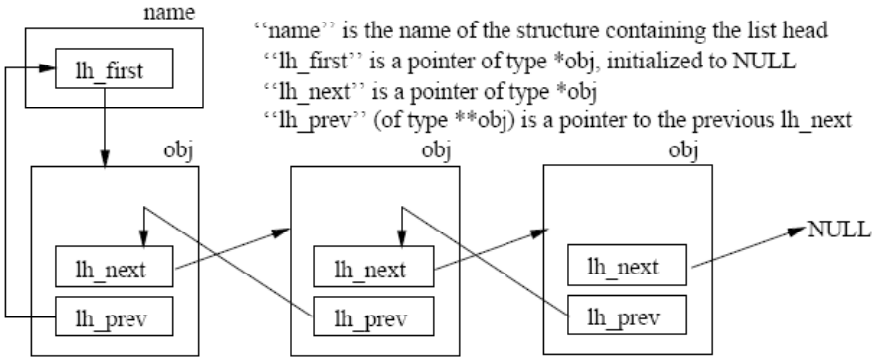


图 17.4 ns 中实现的链表

17.3.1 链表的使用

卫星扩展实现中，许多链表被频繁的使用：

- Node 类维护模拟器中所有 Node 类的对象的一个（静态）列表，变量 Node::nodehead_保存列表头。节点链表用于集中式路由，切换过程中寻找卫星和 tracing。
- Node 类维护节点上所有（卫星）链路的一个列表。特别地，列表是 LinkHead 类的对象的列表。变量 linklisthead_保存列表头。LinkHeads 链表用于检查是否需要切换链路和发现拓扑邻居。
- Channel 类维护信道上所有 Phy 类的对象的一个列表。变量 if_head_保存列表头。这个列表用于决定哪些信道上的接口能够接收分组拷贝。

图 17.4 说明了链表是如何组织的。链表中的每一个对象通过类的一个保护成员“LINK_ENTRY”链接起来。这个入口包含了指向列表中下一项的指针“next”，和一个指向前一个对象的“next”指针的地址的指针。~ns/list.h 中的各种宏可以被用来操作链表，在 ns 中实现的链表类似与在某些变种 BSD UNIX 中实现的队列（queue）。

17.3.2 节点结构

图 17.5 是一个卫星节点 SatNode 的主要组件的图示。SatNode 的结构与无线扩展中 MobileNode 有些类似，但是也有些差别。像所有 ns 节点一样，SatNode 有一个“entry”指向一系列的分类器。地址分类器包含了向外面节点转发分组的 slot 表，但是因为不使用 OTcl 路由，所有不是发向这个节点的分组（从而转发到端口分类器）将被发送到缺省的目标，该目标指向一个路由代理。发往该节点 255 端口的分组被分类为路由分组，并且同样转给路由代理。

每一个节点都包含一个或多个“网络栈”，“网络栈”在链路的入口点包含一个普通的 SatLinkHead。该 SatLinkHead 作为访问链路结构中其他对象的 API，因此它包含了许多指针（尽管这里的 API 还没有最终定下来）。离开网络栈的分组被发送到节点的入口。一个重要的特征是每一个离开网络栈的分组都用一个唯一的与链路相对应的 NetworkInterface 索引将其公共分组头中的 iface_字段编码。这个值可以用来支持分布式路由，如下所述。

路由代理的基类是 SatRouteAgent 类，它可以用来与集中式路由协作。SatRouteAgent 包含一个解析分组地址到特定 LinkHead 目标的转发表——正确地生成转发表则是 SatRouteObject 的工作。SatRouteAgent 生成头部的某些字段，接着将分组下发至适当的链路。为了实现分布式路由，可以定义一个新的 SatRouteAgent，当分组到达该栈时，该 Agent 能够通过记录分组中标记的接口索引来学习拓扑。节点的一个帮助函数 intf_to_target()允许 agent 解析这个索引值到一个特定的 LinkHead。

在一个 SatNode 中，有指针指向三个附加的对象。首先，每一个 SatNode 包含一个位置对象，这在前面已经介绍了。其次，每一个 SatNode 都包含一个 LinkHandoffMgr，用于监控链路切换时机并协调切换。每一个卫星节点和终端节点都有他们特定版本的 LinkHandoffMgr。

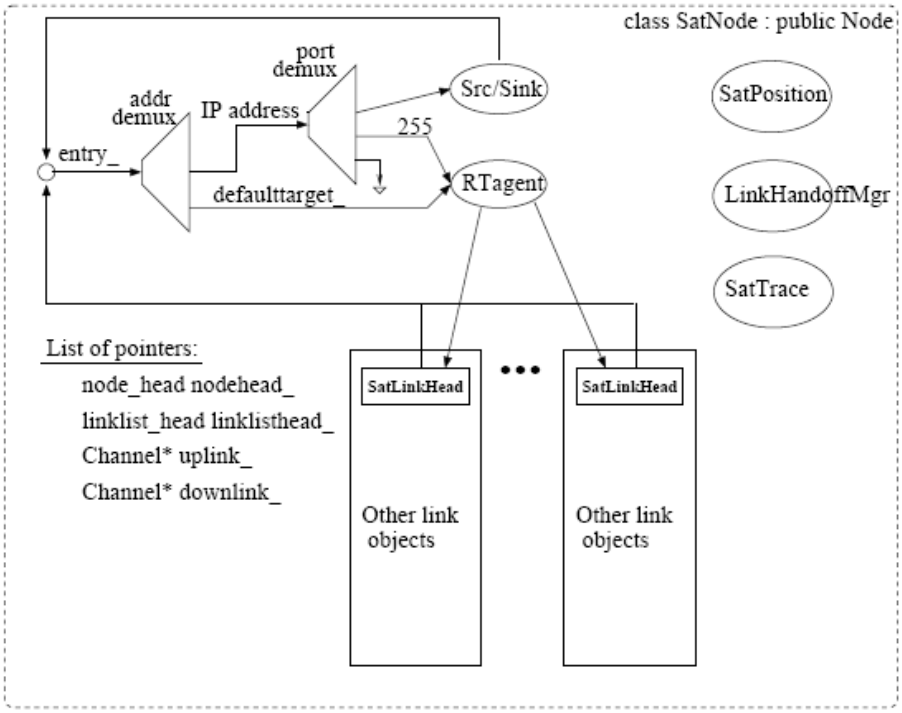


图 17.5 SatNode 类的结构

最后，一个 SatNode 包含了许多对象的指针。我们已经在前面的小节中讨论过 linklisthead_和 nodehead_。为了使用上的方便，采用 uplink_和 downlink_指针指示上/下行信道，当然这是基于大多数模拟中一个卫星或一个终端节点只有一个上行和下行信道的假设。

17.3.3 卫星链路详述

图 17.6 给出了更详细的卫星链路组成图示。在这一节中，我们将介绍分组是如何在栈中上下移动的，以及每一层中需

要注意的关键事情。文件~ns/tcl/lib/ns-sat.tcl 包含了根据图 17.6 组合链路的各种 OTcl 过程。我们描述这种复合结构为“网络栈”，各种链路组件的大多数代码都在~ns/satlink.{cc.h}中。

网络栈的入口点是SatLinkHead对象。SatLinkHead对象派生自LinkHead类；链路头对象的目标是为所有网络栈提供一个统一的API[21]。SatLinkHead对象包含指向LL，Queue，MAC，Error model和Phy等对象的指针。SatLinkHead对象同样能够查询网络栈的类型，如GSL，interplane ISL，crossseam ISL等等。type_字段的有效代码可以在~ns/sat.h中找到。最后，SatLinkHead保存了一个布尔变量linkup_，该变量指示是否已经连接上了信道上的至少一个其它节点。SatLinkHead的C++实现可以在~ns/satlink.{cc,h}中找到。

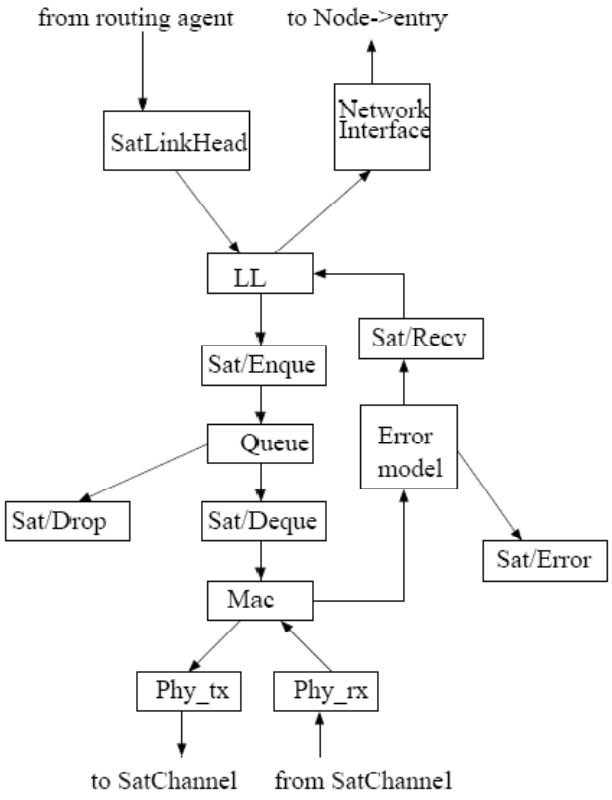


图 17.6 网络接口栈详细图示

分组离开一个节点，透明地通过 SatlinkHead 到达 SatLL 类对象。SatLL 类派生自 LL（链路层）。链路层协议（如 ARQ 协议）可以在这里定义。当前 SatLL 分配一个 MAC 地址给分组。注意，在卫星情况下，我们不使用地址解析协议（ARP）；代而替之，我们简单使用 MAC index_变量作为其地址，并且使用一个帮助函数来查找下一跳节点对应接口的 MAC 地址。因为 LL 类派生自 LinkDelay 类，所以 LinkDelay 的参数 delay_可以被用来模拟链路层中的任何处理延时。缺省这个延时为 0。

输出的分组将遇到的下一个对象是接口队列。然而，如果开启了 tracing，则 tracing 元素可能包围了这个队列，如图 17.6 所示。这一部分的卫星链路功能类似与传统 ns 链路。

接下来的一层是 MAC 层。MAC 层从队列 queue（或者 deque trace）对象提取分组——MAC 和队列之间的一个握手过程允许 MAC 在需要分组时可以从队列中提取他们。分组的传输时间同样在 MAC 层中模拟了，MAC 计算分组的传输延

[21] 在作者看来，ns 中的所有网络栈最终应该在前端有一个 LinkHead 对象，而 SatLinkHead 类将会消失。

时（基于在 satlink.h 中定义的 LINK_HDRSIZE 字段和在公共分组头中的 size 字段），然后直到当前分组已经被“发送”到下面一层才提取下一个分组。因此，在这一层准确设置链路的带宽是非常重要的。方便起见，传输时间编码在 mac 头中，这个信息可以用来在接收方 MAC 计算其必须等待多长时间才探测分组冲突。

接下来，分组被发送到 SatPhy 类的一个发射接口（Phy_tx）。该对象仅将分组发送到其连接的信道上。我们在之前的小节中提到过连接到一个信道的所有接口是该信道链表的一部分。然而，对于发射接口却不是这样的。只有连接到一个信道的接收接口才是这个链表的组成部分，因为只有接收接口需要获得一份发送分组的拷贝。使用独立的发射和接收接口反映了真实世界中全双工卫星链路由不同频率的 RF 信道组成。

输出的分组下一步将发往 SatChannel，SatChannel 再将分组拷贝到信道上的每一个接收接口（SatPhy 类）。然后 Phy_tx 发送分组到（接收端）MAC 层。在 MAC 层，分组先被保持一段时间，即其传输时间，以便进行碰撞检测（如果 MAC 支持，例如 Aloha MAC，则任何适当的碰撞检测可以在 MAC 层执行）。如果确定分组能够安全地到达 MAC 层，则接下来它将被传递到 ErrorModel 对象（如果存在 ErrorModel 对象的话）。否则，分组将经过任何接收 tracing 对象到达 SatLL 对象。SatLL 对象在一段处理延时（同样，delay_的缺省值为 0）之后向上递交该分组。

接收分组最后通过的对象是 NetworkInterface 类的一个对象。该对象用网络栈的唯一索引值来标记公共头中的 iface_ 字段。这可以用来追踪分组到达了哪一个网络栈。分组接着进入 SatNode 的入口（通常是一个地址分类器）。

最后，存在如前面的小节所描述的“geo-repeater”卫星。Geo-repeater 网络栈非常简单，仅仅包含两个 repeater 类对象 Phy_tx 和 Phy_rx，以及一个 SatLinkHead。由 Phy_rx 接收的分组被无延迟地发送到 Phy_tx。Geo-repeater 是一个退化的卫星节点，它不包含如 tracing 元素，切换管理器，路由代理和除中继器接口（repeater interface）外的任何其他链路接口。

17.4 命令一览

下面是卫星网络相关的命令列表：

$ns_ node-config -satNodeType <type>

这个节点配置声明接下来创建的新节点的类型将是<type>，<type>可以是以下列出中的一种：geo，geo-repeater，polar，terminal。卫星节点其他必须的字段（为建立初始链路和信道）如下（参看 5.3 节）：

$ns_ node-config -llType <type>
$ns_ node-config -ifqType <type>
$ns_ node-config -ifqLen <length>
$ns_ node-config -macType <type>
$ns_ node-config -channelType <type>
$ns_ node-config -downlinkBW <value>

（注意：satNodeType 中的 geo-repeater 只需要指定 channelType，所有其他选项都被忽略，参看 tcl/ex/sat-repeater.tcl）。

$ns_ satnode-polar <alt> <inc> <lon> <alpha> <plane> <linkargs> <chan>

这是创建一个极轨道卫星节点的一个模拟器封装方法。两个链路——上行链路和下行链路，与两条信道——上行信道和下行信道一起被创建。<alt>是极轨道卫星的海拔，<inc>是轨道倾斜度，<lon>是升节点的经度，<alpha>给出了卫星在轨道中的初始位置，<plane>定义了极轨道卫星的轨道平面，<linkargs>是一个定义网路接口的链路参数选项列表（如 LL，Qtype，Qlim，PHY 和 MAC 等）。

$ns_ satnode-geo <lon> <linkargs> <chan>

这是创建一个地球同步轨道卫星节点的封装方法，首先创建一个 satnode，然后创建两个链路接口（上行和下行）和两个卫星信道（上行和下行）。<chan>定义信道类型。

$ns_ satnode-geo-repeater <lon> <chan>

这是创建一个地球同步轨道卫星中继器节点的封装方法 ,首先创建一个 satnode ,然后创建两个链路接口(上行和下行) 和两个卫星信道（上行和下行）。

$ns_ satnode-terminal <lat> <lon>

这是创建一个终端节点的简单封装方法。<lat>和<lon>分别定义了终端的纬度和经度。

$ns_ satnode <type> <args>

这是一个创建类型为<type>的 satnode 的更基本的方法，该节点可以是 polar，geo 或终端。

$satnode add-interface <type> <ll> <qtype> <qlim> <mac_bw> <phy>

这是 Node/SatNode 的一个内部方法，用来建立卫星节点的链路层、mac 层、接口队列和物理层结构。

$satnode add-isl <ltype> <node1> <node2> <bw> <qtype> <qlim>

这个方法在两个节点之间创建了一个 ISL（卫星间链路）。链路类型（inter，intra 或者 cross-seam），链路带宽，队列类型和队列限制（queue-limit）都分别被指定。

$satnode add-gsl <ltype> <opt_ll> <opt_ifq> <opt_qlim> <opt_mac> <opt_bw> <opt_phy> <opt_inlink> <opt_outlink>

这个方法创建了一个 GSL（地面到卫星的链路）。首先，通过定义 LL、IfQ、Qlim、MAC、BW 和物理层创建了一个网络栈，接着该节点被连接到信道的入口链路（inlink）和出口链路（outlink）。

第 18 章

无线传播模型

这一章描述了 NS2 中实现的无线传播环境模型。这些模型主要是用来预知每个包的接收信号功率。每个无线节点的物理层都有一个接收的阈值，如果一个包的接收信号功率低于此接收阈值，将其标记为错误并被 MAC 层丢弃。

到目前为止，NS2 实现了三个无线传播模型，分别是：free space（自由空间）模型，two-ray ground reflection 双径地面反射）模型以及 Shadowing（阴影）模型。模型的实现可以在~ns/propagation.{cc,h}, ~ns/tworayground.{cc,h} 和 ~ns/shadowing.{cc,h}中找到。下面描述了 ns-2.1b7 中的 APIs。

18.1 Free space（自由空间） 模型

Free space 传播模型假定了一种理想化的传播环境，在发射方和接受方之间只有一条无障碍的直线路径。H. T. Friis 提出了一种接收信号功率的计算公式：

$$P_r(d)=\frac{P_tG_tG_r\lambda^2}{(4\pi)^2d^2L} \qquad (18.1)$$

d 为发射方和接收方的距离；P_t 为发射的信号功率，G_t和G_r 分别为发送方和接收方的天线增益；L（$L\geq1$）为系统损耗，λ 是波长。在 NS 模拟时，通常取$G_t=G_r=1$和$\mathrm{L}=1$。

在 Free space 模型中，通信范围为以发射方为圆心的一个圆。如果接收方在这个圆里则可以接收所有的包，否则将丢失所有的包。

这个模型的 OTcl 接口是一个 node-config 命令。
使用的一种方法是：
$ns_ node-config -propType Propagation/FreeSpace
另一种方法是：
set prop [new Propagation/FreeSpace]
$ns_ node-config -propInstance $prop

18.2 Two-ray ground reflection（双径地面反射）模型

在两个移动节点之间，单一的直线传播不再是唯一的传播方式。Two-ray ground reflection 模型除了考虑直线传播路径还考虑了地面反射路径。[29]证明了相对于 free space 模型，此模型在长距离传输环境中能有更准确的预测。接收信号功率的公式如下：

$$P_r(d)=\frac{P_tG_tG_rh_t^2h_r^2}{d^4L} \qquad (18.2)$$

d 为距离；h_t 和h_r 分别为发射天线和接收天线的高度。[29]中通常取$\mathrm{L}=1$，L 主要是考虑和 free space 模型相一致

才在公式中加进来的。随着距离的增长，(18.2) 式相对于 (18.1) 式来说有更快的功率损耗。但是因为创建和销毁两个路径的叠加时会产生波动，Two-ray ground reflection 模型在处理短距离的时候效果并不好。相比之下，free space 模型在短距离时还是可用的。

因此，此模型给了一个界限距离 d_c 。当 $d < d_c$ 时，(18.1) 式将使用，$d > d_c$ 时用 (18.2) 式，$d = d_c$ 时 (18.1) 式和 (18.2) 式结果相同。d_c 的公式如下：

$$d_c = \left(4\pi h_t h_r\right)/\lambda \qquad (18.3)$$

在 Otcl 中，调用此模型的接口如下：

```
$ns_ node-config -propType Propagation/TwoRayGround
```

当然，也可以如下调用：

```
set prop [new Propagation/TwoRayGround]
$ns_ node-config -propInstance $prop
```

18.3 Shadowing (阴影) 模型

18.3.1 背景

Free space 模型和 Two-ray ground reflection 模型采用的接收功率公式主要是以距离参数来决定的，其通信范围是一个理想的环。事实上经过一定距离的传播后，由于多径传播效应 (即熟知的阴影效应)，接收功率是随机可变的。由此可见，上两种模型主要是以距离 d 来计算接收功率，而更具综合性且更普遍采用的一种模型是下面要讲的 shadowing(阴影)模型。

环境		β
室外 Outdoor	自由空间 Free space	2
	阴影的乡村地区 Shadowed urban area	2.7 到 5
楼宇 In building	视距 Line-of-sight	1.6 到 1.8
	有阻挡的 Obstructed	4 到 6

表 18.1 一些典型的路径损耗指数 β 值

环境	$\sigma_{dB}\left(DB\right)$
室外 Outdoor	4 到 12
办公室 (墙之类的硬隔开) Office, hard partition	7
办公室 (木板之类的软隔开) Office, soft partition	9.6
工厂 (视距) Factory, line-of-sight	3 到 6
工厂 (有阻挡的) Factory, obstructed	6.8

表 18.2 一些典型的阴影方差 σ_{dB} 值

shadowing 模型包括两个部分。第一部分是路径损耗 (pass loss) 模型，即通过路径 d 来确定接收功率，在这里记为 $\overline{P_r\left(d\right)}$。用接近中心的距离 d_0 作为参考，$\overline{P_r\left(d\right)}$ 可以由 $P_r\left(d_0\right)$ 计算得到：

$$\frac{P_r(d_0)}{P_r(d)} = \left(\frac{d}{d_0}\right)^{\beta} \quad (18.4)$$

β 为路径损耗指数，通常是由环境测量来的经验值。从（18.1）式可以看出在 free space 模型中，β =2。表 18.1 给出了一些典型的 β 值。从表中可以看出越有阻挡的地方其 β 值越大。因此距离越长，平均接收功率降低的越快。$P_r(d_0)$ 可由（18.1）式计算得到。路径损耗经常用 dB 来表示。所以从（18.4）式可以得出：

$$\left[\frac{\overline{P_r(d)}}{P_r(d_0)}\right]_{dB} = -10\beta\log\left(\frac{d}{d_0}\right) \quad (18.5)$$

shadowing 模型的第二部分反映了距离一定时接收功率的变化。它是一个对数正态随机变量，如果以 dB 来作为计量的话，就是一个高斯分布。整个 shadowing 模型表示如下：

$$\left[\frac{\overline{P_r(d)}}{P_r(d_0)}\right]_{dB} = -10\beta\log\left(\frac{d}{d_0}\right) + X_{dB} \quad (18.6)$$

X_{dB} 是一个高斯随机变量，均值为零，方差为 σ_{dB}。σ_{dB} 被称为阴影方差（shadowing deviation），同样也是通过测量获得的。表 18.2 给出了一些 σ_{dB} 的典型值。（18.6）式就是通常的对数正态 shadowing 模型。

这个 shadowing 模型不再是理想的环模型，而是基于统计的模型。所以节点在接近通信范围边界处只是有可能进行通信，而不是绝对的。

18.3.2 Shadowing模型的使用

在使用 shadowing 模型之前，用户应该根据模拟的环境选定路径损耗指数 β 的值和阴影方差 σ_{dB}。此模型的 OTcl 接口仍然是 node-config 命令。下面给出了一种方法（参数的值只是一个例子）：

```
# 第一步设定模型的参数值
Propagation/Shadowing set pathlossExp_ 2.0 ;#设定路径损耗指数值
Propagation/Shadowing set std_db_ 4.0 ;# 设定阴影方差(dB)
Propagation/Shadowing set dist0_ 1.0 ;# 设定参考距离(m)
Propagation/Shadowing set seed_ 0 ;# 设定随机种子
$ns_ node-config -propType Propagation/Shadowing
```

阴影模型新增了随机数发生器（RNG random number generator）对象。RNG有三种类型的种子：raw seed,pre-defined seed (一套已知的好种子) 和huristic seed(具体细节请看25.1)上述的API仅仅采用pre-defined seed。如果用户想使用各种种子生成的方法，可以采用下面的API：

```
set prop [new Propagation/Shadowing]
$prop set pathlossExp_ 2.0
$prop set std_db_ 4.0
$prop set dist0_ 1.0
```

$prop seed <seed-type> 0 ;#用户定义种子生成办法
$ns_ node-config -propInstance $prop
<seed-type>可以是raw, predef or heuristic。

18.4 通信范围

在一些应用程序中，用户希望指定无线节点的通信范围。Ns 中可以在网络接口中通过设定接收阈值来指定通信范围，比如：

Phy/WirelessPhy set RXThresh_ <value>

一个单独的 C 程序被用来计算接收阈值，程序是：~ns/indep-utils/propagation/threshold.cc。这个程序适用于本章讨论的所有传播模型。假定此程序已经编译过得到一个可执行的文件，命名为 threshold。你可以按照如下方法来计算阈值：

threshold -m *<propagation-model> [other-options] distance*

*<propagation-model>*可以是 Freespace、TwoRayGound 或 Shadowing； *distance* 就是通信范围（单位：m）；*[other-options]*用来指定一些参数，而不是默认的参数值。对于 Shadowing 模型，有一个必要的参数即 *:-r <receive-rate>*。这个参数指定了在 *distance* 距离下正确接收的速率。因为在 Shadowing 模型中，通信范围不再是一个理想的环，所以在[29]中采用反向 Q 函数（inverse Q-function）来计算接收阈值。举例来说，如果希望在距离 50m 的地方能正确接收 95%的数据包，你可以按如下方法计算阈值：

threshold -m Shadowing -r 0.95 50
[other-options]中的其他值如下：
-pl <path-loss-exponent 路径损耗指数>
-std <shadowing-deviation 阴影方差>
-Pt <transmit-power 发射功率>
-fr <frequency 频率>
-Gt <transmit-antenna-gain 发射天线增益>
-Gr <receive-antenna-gain 接收天线增益>
-L <system-loss 系统损耗>
-ht <transmit-antenna-height 发射天线高度>
-hr <receive-antenna-height 接收天线高度>
-d0 <reference-distance 参考距离>

18.5 命令一览

下面是传播模型的命令列表：

$ns_ node-config *-propType <propagation-model>*

这个命令用来设定模拟的传播环境模型 *<propagation-model>*。 *<propagation model>*可以是 Propagation/FreeSpace, Propagation/TwoRayGround 或者 Propagation/Shadowing

$ns_ node-config *-propInstance $prop*

这个命令是另一种设定传播模型的方法. *$prop* 是 *<propagation-model>*的实例。

$sprop_ seed *<seed-type> <value>*

这个命令用来设定RNG（随机数发生器）. *$sprop_*是 shadowing模型的一个实例。

threshold *-m <propagation-model> [other-options] distance*

这是一个独立的程序，在~ns/indep-utils/propagation/threshold.cc。用来计算特定通信环境下的接收阈值。

第 19 章

ns中的能量模型

ns中实现的能量模型是一个节点属性。能量模型表示了一个移动主机的能量水平。在模拟开始时，每个节点的能量模型都有一个能量初始值initialEnergy_。此时还产生每个包发射和接收的能耗，分别为txPower_ 和rxPower_。能量模型的定义文件在ns/energymodel[.cc and.h]。其他本章涉及到的函数有 :ns/wireless-phy.cc, ns/cmu-trace.cc, ns/tcl/lib[ns-lib.tcl, nsnode.tcl, ns-mobilenode.tcl]。

19.1 能量模型的C++类

基本的能量模型很简单，类 EnergyModel 定义如下：

```
class EnergyModel : public TclObject
public:
   EnergyModel(double energy) energy_ = energy;
   inline double energy() return energy_;
   inline void setenergy(double e) energy_ = e;
   virtual void DecrTxEnergy(double txtime, double P_tx)
   energy_ -= (P_tx * txtime);
   virtual void DecrRcvEnergy(double rcvtime, double P_rcv)
   energy_ -= (P_rcv * rcvtime);
   protected:
   double energy_;
;
```

从上述的类EnergyModel的定义可以看出，类中只有一个变量energy_表示特定时间节点的能量水平。构造函数EnergyModel（energy）需要初始的能量水平energy作为参数。另外还有：DecrTxEnergy(txtime, P_tx)，表示每个节点发射时一个包减少的能量水平 ;DecrRcvEnergy (rcvtime, P_rcv) ,表示每个节点接收一个包时减少的能量水平 ;P_tx and P_rcv 分别是每个节点接口（或PHY）发射和接收需求的功率。在模拟的一开始，energy_设置为initialEnergy_，其值对每一次的发射和接收都会减少。当节点的能量水平降低到0时 ,节点就不能发射或接收数据包。如果打开了跟踪功能(trace),"DEBUG：node <node-id> dropping pkts due to energy = 0" 这一行将出现在跟踪文件（tracefile）中。

19.2 OTcl 接口

因为能量模型是节点属性，所以其 OTcl 的接口也是在节点配置里定义的：

```
$ns_ node-config -energyModel $energymodel \
                 -rxPower $p_rx \
                    -txPower $p_tx \
                    -initialEnergy $initialenergy
```

上述能量模型的配置参数值列于下表：

属性	可选值	默认值
-energyModel	"EnergyModel"	None
-rxPower	接收功率，单位：瓦特(e.g 0.3)	281.8mW
-txPower	发射功率，单位：瓦特(e.g 0.4)	281.8mW
-initialEnergy	能量，单位：焦耳	0.0

第 20 章

定向扩散（Directed Diffusion）

ns 中的定向扩散模块来自于 SCADDS 组对 USC/ISI 定向扩散的应用。几年前实行的一个旧的 ns 版本已经相对过时。这个旧的版本在 diffusion 目录下还是可以找到的。新一点的 diffusion 版本是保存在~ns//diffusion3。这一章讨论 ns 中新的 diffusion 模型。这里描述的模型和方法可从~ns/tcl/lib/ns-diffusion.tcl 中找到，ns-lib.tcl 和所有相关的 C++代码可从~ns/diffusion3.中得到。关于实现细节可参考 SCADDS 组的网页：
http://www.isi.edu/scaddsfordetailsabouttheirimplementation.

20.1 什么是定向扩散？

定向扩散是一种数据分发方式，特别适合于分布式的感测环境中。它不同于通信中的 IP 方法。对于 IP，“节点被它们的端点识别，而且内部节点的通信被放置在网络内部提供的终端对终端传输服务之上”。而定向扩散是以数据为中心的，传感器节点产生的数据以属性数值对命名，请求数据的接收点或节点发送“兴趣信息”给网络。由“源”节点产生的数据匹配这些兴趣信息，“流”向接收点，中间节点能缓存和转发数据。定向扩散的细节可参考“DirectedDiffusion:A Scalable and Robust Communication Paradigm for Sensor Networks” 一文，作者是 Chalermek Intanagonwiwat,Ramesh Govindan and Deborah Estrin 。出版在 MobiCOM,August2000,Boston,Massachusetts.这篇文章和其他与 diffusion 相关的文章可参考 http://www.isi.edu/scadds/publications.html under publication ssection

20.2 ns中的diffusion模型

定向扩散模型由一个中心diffusion层、一个diffusion库（提供应用程序接口给附加diffusion程序）和应用层（包括diffusion程序和过滤器）。中心diffusion层用作从网络中接收报文或发送报文进入网络，diffusion库提供一个覆盖发布/定购等应用类的接口。这些API在Network Routing API 8.0这个文档中详细介绍了，可以在http://www.isi.edu/scadds/publications.htmlAPI部分找到。在下一段我们将描述diffusion模型的结构。

首先我们开始简要描述 diffusion3 的目录结构。如果读者希望找到与 NS Diffusion 相关的 C++代码以巩固 Otcl 描述命令，可在 ns/diffusion3 中找到，下面是子目录的总结：

Apps 包含样本数据源和接收点应用程序如 gear,ping 和 rmst.

Lib 包含 diffusion 路由类定义和 diffusion 应用类定义。另外有子文件列表叫 main 和 nr，main 里主要包括了各种 diffusion 公用代码。nr 包括属性定义和类 NR，类 NR 是一个 API 的抽象工厂（以便 ISI 或 MIT 可能用它来实现一些类）。

ns 包括 diffusion 代码的 ns 封装。这些封装类允许中心 diffusion 代码和 diffusionAPI 完全的合并成 ns 类分层，DiffRoutingAgent 是一个中心 diffusion 代码的封装，DiffAppAgent 是一个 DiffusionRouting(API) 代码的封装。

filter_core 具有中心 diffusion 代理。

filters 由包括 two-phase-pull,one-phase-pull,gear,rmst,log，tag，srcrt 这些 diffusion 应用程序支持的不同的过滤器

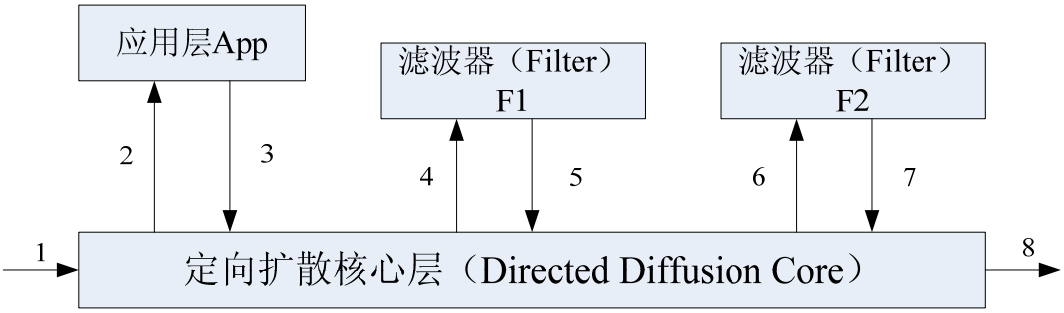


图 20.1 定向扩散的消息流

上图 20.1 是来自 SCADDS 的网络路由 API 文档，它们主页是有效的（URL 给的较早）。文档描述属性因素、属性匹配规则、应用程序与中心 diffusion 层和过滤器以及定时器的 API 接口。所有来自/流向网络的消息都是从中心 diffusion 层收到/发送。消息也可能来自连接到节点的局部应用程序的中心代码层或过滤器。应用层通过发布/定购方法向网路发送兴趣和数据消息。

中心diffusion代理和diffusion应用代理被附加到~ns//tcl/lib/nsdefault.tcl 中定义的两个众所周知的端口。附加到节点上的diffusion应用为发布/定购/发送数据时调用的diffusion应用代理。

20.3 ns中diffusion的一些mac方面的问题

首先，ns 中所有 diffusion 报文是广播形式的，在位于 diffusion 和 ns 之间的 shim 层，(实现这层的 ns 代码目录是 diffusion3/ns)，所有的 diffusion 报文被封装在 ns 报文里并标为广播。在先前的版本，所有的 diffusion 包在 ns 中都被标记为广播。新版本中 ns 的 diffusion 报文采用下一跳信息，这样允许被标记为广播或者单播。

先前只支持广播的报文 diffusion 方式可能会导致 mac 层的一些问题。mac-802.11 在有冲突和报文丢失的情况下不会重发一个广播报文。与这相关的是 mac-802.11 重发一个报文（一个广播数据或单播报文的 rt）之前在争用窗口中不做时隙的随机选择，结果在 mac 层有大量的冲突而且很多报文被丢失。这通过 mac 在发送一个数据报文（或一个 rts 报文）之前添加任意的通道选择来改进。

但是如果我们有大量的很多的拓扑，那么更多的节点在 mac 争用窗口（争用窗口尺寸从 31 到 1023 不等），可能选择相同的 solt，因此现在我们需要在更高的应用层上加一些额外的抖动。diffusion 通过用宏 USE_BROADCAST_MAC 编译 ns 以产生这样一个抖动。所做的就是增加额外的延迟消息 delaying certain messages(以避免冲突)。当在 BROADCASTMAC 层上运行时，diffusion 将采用一个不同延迟和抖动的参数集，这些延迟/抖动参数集在 diffusion3/lib/main/config.h 中定义。因为这可能增加等待时间，所以你可能在这个 h 文件中手动调节延时值。

20.4 diffusion中运用过滤器的APIs

前面已经提到，过滤器可以与不同的 diffusion 代码匹配。有一些基本的 diffusion 过滤器提供 two-phase-pull 和 one-phase-pull diffusion 路由算法。GeoRoutingFilter 也产生特定的基于本地的路由算法。也有其它的路由器用 RMST 算法，logging 算法，源路由和标记算法。关于增加 diffusion 节点的 APIs 详细细节部分看命令一览这个章节。

20.5 Ping一个diffusion应用实现的例子

在 diffusion3/apps/pingsubdir 子目录下有一个 ping 的应用实例。这个应用由一个 ping 发送方和接收方组成。接收端通过在网络中发送“兴趣”请求数据。“兴趣”消息通过网络得到。ping-sender 一旦接收到匹配的兴趣就发送数据。

20.5.1 ping应用程序的c++实现

类 ping-sender 和类 ping-receiver，即为 PingSenderAppand 和 PingReceiverApp，都是源于 DiffApp，所有 diffusion 的父类以应用程序为基础。细节参照 diffusion3/lib/diffapp{.cc,.hh}类实现机制。

类 ping-sender 使用 MySenderReceive 对象处理所有的回叫信号。类 ping-sender 也定义了两个函数 Subscription() 和 setupPublication()。第一个函数创建与数据属性相匹配的兴趣属性。下一步调用 dr 库函数 subscribe()。类 ping-sender 利用这个函数来创建一个内部状态，此状态相对于从网络中收到的兴趣属性，如果匹配，匹配的数据发送到网络。函数 setupPublication()为它提供的数据创建属性并调用库函数 publish()，它返回了一个发布句柄。 ping 发送端使用这个句柄定期的发送数据。一旦找到一个匹配的‘兴趣’，数据通过梯度转发到 diffusion 中心再被发送到网络。

类 ping- receiver 的对象使用相似的调用对象 MyReceiverReceive。它定义一个函数 setupSubscrip-tion()，此函数为接收端要发送的兴趣创建属性。接着它调用 subscribe() 支持的 dr 库，此库发送兴趣进入网络。函数 Recv（ ）用作接收匹配的数据 ,接收端然后计算从 ping 发送端接收过来的每个数据报文的等待时间。类 ping-receiver 可从 ping_receiver.cc,.h 中找到。一些数据/兴趣属性的普通的定义和属性在 ping.h 和 ping_common.c 中定义。

20.5.2 ping应用程序的tcl API

ping 应用程序的一个例子在~ns/tcl/ex/diffusion3/simple-diffusion.tcl 。例子是由 3 个节点组成，一个是 ping-sender，一个是 ping-receiver，数据源和接收点彼此远离，只是通过第三方节点通信，选择的 adhocRoutin 被定义为定向扩散。这就使得一个核心的定向扩散代理在节点创建的期间被创建。如果它已经不存在的话它也创建一个 diffusion 应用代理。选项 diffusionFilter 需要在节点配置期间被提供定义将被添加到节点上的一个或多个过滤器。还有一个选项是指定模拟终止的。此时有一个函数的回调将所有静态数据输出到/tmp/diffusion-*.out。

节点配置方法如下：

```
$ns_node-config   -adhocRouting   $opt(adhocRouting)
                  -llType          $opt(ll)
                    -diffusionFilter   $opt(filters)
                   -stopTime         $opt(prestop)
```

Ping 的发送端的应用程序按照下列方法创建：

```
set src_(0) [new Application/DiffApp/PingSender]
$ns_ attach-diffapp $node_(0) $src_(0)
$ns_ at 0.123 "$src_(0) publish"
```

第一行创建一个 ping-sender 的对象。模拟类方法 attach-diffapp 基本上附加到给定节点隐藏的 diffusion 应用代理上，命令 publish 启动了发送端的应用程序。

类似 ping 的接收端按照下列方法创建：

```
#Diffusion sink application
set snk_(0) [new Application/DiffApp/PingReceiver]
$ns_ attach-diffapp $node_(2) $snk_(0)
$ns_ at 1.456 "$snk_(0) subscribe"
```

此命令指定了启动 ping 接收端的应用。

由此可以看出创建你自己的应用程序，你需要：

1. 为应用程序的兴趣/数据定义属性仓库和属性。
2. 创建应用类（使用 dr-library APIs）。
3. 添加 tcl 命令来启动应用程序。
4.

定向扩散 OTcl 分支实现的命令可看~ns/tcl/lib/ns-lib.tcl,ns-diffusion.tcl。也可看手册中的移动这一章节以便更进一步了解移动模型和无线模型。

20.6 给ns添加yr diffusion应用程序的需求

假如你已经有了运行在测试版本上的应用程序。（它可能甚至是一个特定的过滤器，而且有它的类层次或是一个源于类 DiffApp 的 diffusion 应用程序）。现在你想在 ns 上运行 diffusion 并且在 ns 环境中使用 yr。你需要在 ns 环境中注明几行说明你要使用 yr。

我们考虑 onePhasePullFilter 对象(在 diffusion3/filters/misc/log.*)。第一步你需要给应用类对象创建一个分裂对象，作为一个纯 c++类定义。分裂对象是由用户在解释器中（在 OTcl 空间）创建的对象，而且在编译层（在 c++）空间）也有一个影子对象。在 ns 中，一个分裂对象源于类 TclClass，如下所示：

```
#ifdef NS_DIFFUSION
static class LogFilterClass : public TclClass
public:
LogFilterClass() : TclClass("Application/DiffApp/LogFilter")
TclObject * create(int argc, const char*const* argv)
return(new LogFilter());
class_log_filter;
 #endif //DIFFUSION
```

注意到在构造或调用的过程，所有特定的过滤器对象有一个对 DiffAppAgent（diffusion 路由对象）的句柄。过滤器对象从 ns-diffusion.tcl 中定义的 create-diffusionApp-agentdiffFilters 函数中创建。当它在基于节点配置参数的节点创建期间被调用时，用户不需要专门调用 oTcl 函数 create-diffusionApp-agent 。大致看一下有关的部分就明白过滤器怎么在节点配置的命令中被定义。但是应用程序不是过滤对象（如 ping_sender, push_receiver）的应用对象由用户直接从用户脚本中创建。并且在那种情况下 DiffAppAgent 的句柄通过$ns attach-diffapp $node $app 传递，此处应用程序$app 直接被附加到节点对象$node 上。

以上解释的构造器的原因与非 ns 定向扩散的情况不同之处可以从下面看出：

```
#ifdef NS_DIFFUSION
LogFilter::LogFilter()
#else
```

```
LogFilter::LogFilter(int argc, char **argv)
#endif // NS_DIFFUSION
// 创建Diffusion路由类
#ifndef NS_DIFFUSION
parseCommandLine(argc, argv);
dr_ = NR::createNR(diffusion_port_);
#endif // !NS_DIFFUSION
filter_callback_ = new LogFilterReceive(this);
#ifndef NS_DIFFUSION
//建立滤波器
filter_handle_ = setupFilter();
...
...
#endif // !NS_DIFFUSION...
```

下一步你需要添加 c++命令（。。）允许 tcl 命令通过编译的映像对象执行。例如 otcl 命令 start 用来启动一个过滤器应用程序如$appstart 。同时命令 publishand 和 subscribe 分别用来启动发送端和接收端应用程序。命令函数被添加，NS_DIFFUSION 范围使用 ifdef 说明，具体程序如下：

```
#ifdef  NS_DIFFUSION
Int LogFilter::command(intargc,const char *const *argv)
if(argc==2)
if(strcmp(argv[1],"start")==0)
run();
return TCL_OK;
return  DiffApp::command(argc,argv);
#endif // NS_DIFFUSION
```

注意如果 command 这个字串找不到的话，如何调用父类 command 函数。参照 lib/diffapp.* 可知道所有 DiffApp 类支持的 octl 命令。

一旦这些改变添加到你 C++代码里，你将需要使用你的 diffusion 程序和合适的 tclAPI 写一个 tcl 脚本（参照 test-suite 部分可看到 tcl 脚本的例子）。

20.7 定向扩散的测试组

我们开始一个简单的测试例子，其中有 3 个节点，一个 ping 源，1 个 ping 接收端。当然还有一些测试例子，比如 2phase-pull(2pp)，1phase-pul，push 和 gear 的场景。以后我们计划扩展 test-suite 以测试不同的定向扩散的组件和功能。所有与测试相关的 diffusion3 的例子可在~ns/tcl/test/test-suite-diffusion3.tcl.中找到。

20.8 命令一览

下列是用作与 ns 模拟的 diffusion 相关的命令列表。

```
$ns_ node-config -adhocRouting $opt(adhocRouting)
                 -llType $opt(ll)
                 ...
                 -diffusionFilter $opt(filters)
                 -stopTime $(pre-stop)
                 ...
```

其中 opt(adhocRouting)的值设为 Directed_Diffusion。这个命令使得定向扩散在无线节点中的使用有效。

Opt（filters）的值可以是一个过滤器的列表，它需要通过一个 diffusion 机制或被分别附在每个定向扩散的节点上。 这个命令用于将过滤器对象添加到 diffusion 激活的节点上。

Opt（pre-stop）的值通常是当所有的统计数据被添加到文件中时将所有的时间模拟停止。这个命令在运行一个定向扩散仿真后将统计数据加入一个输出文件中。

Set src [new Application/DiffApp/PingSender]

这个命令用作创建 ping-sender 应用程序。

Set snk[newApplication/DiffApp/PingReceiver]

这个命令用作创建 ping-receiver 应用程序。

Set src[newApplication/DiffApp/PushSender]

这个命令用作创建 push-sender 应用程序

Set snk[newApplication/DiffApp/PushReceiver]

这个命令用作创建 push-receiver 应用程序

Set src[newApplication/DiffApp/GearSenderApp]

这个命令用作创建 gear-sender 应用程序

Set snk[newApplication/DiffApp/GearReceiverApp]

这个命令用作创建 gear-receiver 应用程序

$gearApp push-pull-options <push/pull> <point/region> <co-ordinatesX1><X2> <Y1><Y2>

这个命令用来定义路由算法合适的类型。在第二个选项被定义为 region 的情况下，所有的四个坐标应该被定义。而如果 point 被选择，只有 X1 和 Y1 可以被定义。

$ns_attach-diffapp$node_$src_

在此 diffusion 应用程序$src_gets 被附加到给定的节点$node_上

$src_(0) publish

命令用来开始一个 ping 数据源（发送端）

$snk_(0) subscribe

命令用来开始一个 ping 接收点（接收端）

第 21 章

XCP：显式拥塞控制协议

XCP 是一个基于反馈的拥塞控制系统，它利用路由器显示的反馈网络的拥塞状况来避免网络拥塞。其设计目标为既具有可扩展性又具有一般性。该协议是由 Dina Katabi 在 Mark Handley 的建议下开始开发的（参考文献[31]详细地描述了有关情况）。最初由 Dina Katabi 为 XCP 开发的 ns 代码已经在 USC/ISI 被修改、扩展和集成到 ns-2.28 版本中。直到今天它还在不断完善。如果你有兴趣看一看 Dina 最初的源代码，请访问她的 website：http://www.ana.lcs.mit.edu/dina/XCP/。

21.1 什么是 XCP？

XCP 是一个可以应用到任何传输协议中的拥塞控制协议。它在高带宽延时积（BDP：bandwidth-delay product）网络中具有特别好的性能。在高带宽延时积网络中，在出现丢包情况下，则 TCP 的性能会急剧下降，并且在任何队列机制下会使网络都会变得不稳定。但是在相同的环境下，XCP 由于采用了使用基于控制理论的反馈模型，以及通过网络反馈的拥塞状况决定发送速率的办法使其更高的效率、稳定性和公平性，

XCP 的可扩张性在于路由器上不需要通过保留每流状态（per-flow state）来计算反馈信息的特性。大多数路由器辅助的拥塞控制系统需要维持每流状态来预留资源。而 XCP 只需要在路由器上保存非常少的信息，这些信息用来最小化维持路由器最基本的信息和刷新每包信息。

概括地讲，XCP 将资源分配功能划分为两个控制器：一个是效率控制器确保流使用所有的有效链路容量（在 xcp 的原文中用的是 Efficiency Controller 而不是 congestion Controller，故翻译中全部纠正为效率控制器），一个公平控制器确保在流之间公平地分配链路容量。大多数拥塞控制系统在都没有划分这两个功能，更不用说作为两个概念上松耦合的部分来实现。这个划分使得 XCP 中两个基本的资源分配功能得到了一个清晰的阐述和实现。

尽管 XCP 可以在任何传输协议中实现，然而，作为一个初步的测试，我们在 TCP 中实现了它。下一节详细介绍 ns 中 XCP 实现的途径。

21.2 在 ns 中实现 XCP

在 ns 中，XCP 的实现可以在~ns/xcp 目录下找到。这个协议需要配置在 TCP 终端（源端和接收端）和中间节点，这里的中间节点大多数情况下是路由器，某些时候也可能是一个链路层交换机。终端部分的 XCP 代码可以在 xcp-end-sys.{cc,h}中找到，路由器部分的代码可以在 xcp.{cc,h}和 xcpq.{cc,h}中找到。

21.2.1 XCP 终端

XCP 中的终端由采用 XCP 拥塞控制机制的 TCP 源和 sink 代理组成。中间节点或路由器在每一个分组头写入反馈信息，作为 delta_throughput 值，这是一个关于发送速率的值，如果反馈为正，则增加发送速率（cwnd），

如果反馈为负，则应该减少发送速率（cwnd）。当分组到达接收端后，delta_throughput 值将被拷贝到返回分组的拥塞头的 reverse_feedback 字段，从而返回到发送端，这个返回的分组在 TCP 的 ACK 分组中。 发送端依据这个负反馈值，通过增加或减少其拥塞窗口大小（视情况而定）来调整其发送速率。

XCP 采用的分组头在 ns 中实现为一个名为 hdr_xcp 的结构，定义如下：

```
double    throughput_;
double    rtt_;
enum {
      XCP_DISABLED = 0,
      XCP_ENABLED,
      XCP_ACK,
}     xcp_enabled_;       // to indicate that the flow is XCP enabled
Bool xcp_sparse_;         // flag used with xcp_sparse extension
int   xcpId_;             // Sender's ID (debugging only)
double    cwnd_;                  // The current window (debugging only)
double    reverse_feedback_;

// --- Initialized by source and Updated by Router
double delta_throughput_;
unsigned int controlling_hop_;   // The AQM ID of the controlling router
```

XCP 的接收端负责拷贝 dalta_throughput 值到 ack 分组中的 reverse_feedback 字段中。在某些情况下，接收端采用延迟应答(delay ack),则接收端计算接收到的分组的 delta_throughput 值之和，然后将其拷贝到发出 ack 分组的 reverse_feedback 字段。 controlling_hop_字段携带最后一个更新反馈信息的路由器的地址，这一信息仅用于调试。

当网络中出现分组丢失的情况，TCP 的 Van Jacobson 拥塞控制（VJCC）应该很可能优于 XCP。然而，在 ns 中这种情况稍微有点不同。在接收到重复 ack 指示分组丢失的假定下，VJCC 将 cwnd（拥塞窗口）减半，接着执行快速重传和快速恢复算法。然而，XCP 路由器继续发送反馈信息到源端，基于此，源端试图打开其 cwnd。因此，看起来 VJCC 和 XCP 算法有些混乱。然而，对于大多数情况，这个问题并没有出现，因为 XCP 路由器很少会出现分组丢弃的情况，这是由于被 XCP 路由器会根据持续队列大小不断的调整反馈值使其减少。理解 XCP 在面对分组丢失时的正确行为是当前需要研究的领域，也是将来应该实现的。

21.2.2 XCP 路由器

XCP 路由器由一个封装类组成，这个类保存了 XCP、TCP 和 OTHER 业务流的虚拟队列。OTHER 可能为除 XCP 和 TCP 以外的任何流。在当前实现中，XCP 队列采用的是丢尾队列（drop-tail queue），而 TCP 和 OTHER 用 RED。

这些基础队列被捆绑到一个封装类 XCPWrapQ，该类为 XCP 路由器提供必要的接口。XCP/TCP/OTHER 队列的调度方式为加权轮循（WRR：Weight Round-Robin）。每一个队列都有一个权值以决定其服务的优先级。当前队列实现中，XCP 和 TCP 的权值均为 0.5，因此允许在两者之间平等享受服务。保留给 OTHER 流的第三个队列还没有使用，因此权值为 0.0。

OTCL 类 Queue/XCP 有一个名为 tcp_xcp_on_的标记，这个标记缺省值设置为 0。如果仿真过程中同时使用 XCP 和 TCP 流 则该标记应该设为 1 ,从而用该标记来在 XCP 和 TCP 队列之间分割路由器的容量。这是一个临时约定的假设，将来一旦动态队列权值开始生效，则应该去除这个标记。一个关于 tcp_xcp 标记的说明是，它不是作为每一个实例变量而是作为一个 OTcl 类变量设置的。这在拓扑上使用混合间歇 xcp 和 tcp 流的情况可能会出现某些问题，这是因为某些路由器将需要同时支持 TCP 和 XCP，而某些不需要。 由封装队列类接收的每一个分组首先根据其分组的类型被标记或被分配一个码点（code point）。对于当前的 TCP 实现，分组可被标记为 XCP、TCP/TCP-ACK 和 OTHER 分组。码点用于使分组进入正确的队列。封装类是在 xcp.{cc,h}中实现的。

21.2.3 XCP 队列

XCP 队列负责在每一个分组头中返回反馈信息，接收端利用这些反馈信息控制向网络发送数据分组的速率。XCP 使用两个控制器，即效率控制器和公平控制器，他们每隔一个估计控制时间间隔 Te 执行一次。

在 ns 中，估计定时器 estimation_timer 的时间间隔为通过路由器的所有（xcp）流的平均 rtt。、然而，可能有更好的方法来定义这个时间间隔。路由器中的持续队列由 queue_timer 每隔 Tq 时间测量一次。最后 rtt_timer 来测量给定时间间隔 Tr 内路由器的某些参数如分组丢弃、队列长度和利用率等，Tr 值既可以是用户通过 tcl 脚本来设置，也可以采用经过路由器的所有流的 rtt 的最高值。

rtt 定时器 rtt_timer 的时间间隔值 Tr 可以使用下面的 API 从 tcl 脚本中设置： $queue queue-sample-everyrtt $rtt_value

此处，queue 是 xcp 路由器的一个句柄，$rtt_value 是时间间隔，每隔该时间 值 测 量 一 次 xcp 队 列 参 数 如 分 组 丢 弃 、队 列 长 度 等 。请 参 看 在~ns/tcl/ex/xcp/ parking_lot_topo/parking_lot_topo.tcl 中的示例脚本。

当分组到达时，在 XCP 队列计算总的输入流增加了的新分组的大小。sum_inv_throughput 以及 rtt 与吞吐量乘积累加和（sum_rtt_by_throughput）也随之增加。从 xcp 分组头中读取的吞吐量和 rtt 值由 TCP 源设置。每一个值都由分组大小值归一化处理了。 一旦估计定时器超时，使用上面提到的两个累加和计算所有流的平均 rtt，如

avg_rtt = sum_rtt_by_throughput/ sum_inv_throughput

聚合反馈是基于路由器的可用带宽容量、到达路由器的流量和路由器的持续队列长度计算得到的。有关XCP路由器算法使 用 的 计 算 算 法 的 进 一 步 详 细 解 释 可 以 在 XCP 的 说 明 文 档 中 找 到 ， 请 参 看：http://www.isi.edu/isi-xcp/docs/draft-falk-xcp-spec-00.txt。

每一个分组在其分组头中携带了流的当前吞吐量值和一个吞吐量调整值 delta_throughput。XCP 路由器基于其在估计定时器超时时计算的反馈值为每一个分组计算吞吐量调整反馈值。正反馈时将剩余带宽公平分配给每个流，而负反馈根据每个流的容量成比例地减少。同样，当计算得到的反馈值小于分组头中的时（有上游负载较轻的路由器写入），下游路由器可以在分组头中改变 delta_through 值。在 ns 中的 XCP 队列的实现可以在 xcpq.{cc,h}中找到。

21.3 XCP 示例脚本

首先让我们演练一个类似于~ns/tcl/ex/xcp/xcp_test.tcl 的简单 xcp 脚本，该脚本采用 3 个 xcp 源共享一个瓶颈链路的哑铃状拓扑。 拓扑通过节点和链路创建 API 建立，瓶颈链路是一个双向链路，两个方向都有一个 xcp 路由器。关于在ns中如何创建节点、链路等的详细介绍请参看Marc Greis的NS tutorial ,网址 :http://www.isi.edu/nsnam/ns/tutorial。带有一个 XCP 队列的瓶颈链路的创建方法如下：

```
set   R0   [$ns node]       ;# create Bottleneck between nodes R0 and R1
set   R1
[$ns node] $ns   duplex-link   $R0   $R1   <BW>Mb   <delay>ms   XCP
```

连接源节点和瓶颈链路的链路也有一个 XCP 队列。API 接口 queue-limit 允许用户设置队列的缓存大小。

xcp 源和 sink 创建方法如下（非常类似于 tcp）：

```
set   xcp [new Agent/TCP/Reno/XCP]
$ns   attach-agent   $src_node $xcp
set   xcp_sink   [new Agent/XCPSink]
$ns   attach-agent   $rcvr_node $xcp_sink
$ns   connect   $xcp $xcp_sink
 ...
 ...
```

示例脚本中有一个 tcl 类 class GeneralSender，使用该类建立源端节点中的 xcp agents，然后将他们与目的节点中的 xcp receiver 连接起来。3 个源都采用一个 FTP 业务源。

注意，一旦拓扑建立，则链路带宽信息需要被传播到 xcp 队列，因为这些信息将被 xcp 路由器用来计算反馈。因此对每一个 xcp 队列，使用下面的 tcl 脚本：

```
$xcp_queue set-link-capacity <bandwidth_in_bits_per_sec>
```

接下来，我们需要跟踪 xcp 路由器和 xcp 源端的变量。在 ns 中，GeneralSender 类的过程 trace-xcp 使用变量跟踪（variable-tracing）为 xcp 源建立跟踪。

```
GeneralSender instproc trace-xcp parameters {
$self instvar tcp_ id_ tcpTrace_
global ftracetcp$id_
set ftracetcp$id_ [open xcp$id_.tr w]
set tcpTrace_ [set ftracetcp$id_]
$tcp_ attach-trace [set ftracetcp$id_]
if { -1 < [lsearch $parameters cwnd] } { $tcp_ tracevar cwnd_ }
if { -1 < [lsearch $parameters seqno] } { $tcp_ tracevar t_seqno_ }
}
```

为了跟踪 xcp 队列，需要绑定一个文件描述符到 xcp 队列：

```
$xcpq attach <file-descriptor>
```

下面是在 xcp 源端获得的跟踪文件的示例：

```
0.00000 2 0 1 0 cwnd_ 1.000
0.00000 2 0 1 0 t_seqno_ 0
0.079 x x x x throughput 0.1
0.07900 2 0 1 0 t_seqno_ 1
0.119064 x x x x reverse_feedback_ 0
0.119064 x x x x controlling_hop_ 0
0.119064 x x x x newcwnd 1
0.11906 2 0 1 0 cwnd_ 2.000
```

0.119064 x x x x throughput 50000
0.11906 2 0 1 0 t_seqno_ 2
0.119064 x x x x throughput 50000
0.11906 2 0 1 0 t_seqno_ 3

第一个字段给出了时间戳；接下来 4 个字段分别是 xcp 流的源 ID（节点/端口）和目的 ID（节点/端口）。接下来的字段是被跟踪的变量名，接着是变量的值。注意变量如 cwnd_，t_seqno_使用的变量跟踪（variable tracing）是由 OTcl lib 支持的一个函数。而像变量 throughput 和 reverse_feedback 采用在 xcp-end-sys.cc 中定义的 XCPAgent 类的函数 trace_var。更多关于 ns 中变量跟踪的信息请阅读本手册的 3.4.3 节。

在 xcp 瓶颈链路路由器的跟踪输出示例如下所示：

Tq_ 0.0472859 0.025
queue_bytes_ 0.0472859 0
routerId_ 0.0472859 0
pos_fbk 0.053544 0
neg_fbk 0.053544 0
delta_throughput 0.053544 0
Thruput2 0.053544 60000
pos_fbk 0.054024 0
neg_fbk 0.054024 0
delta_throughput 0.054024 0
Thruput2 0.054024 60000
residue_pos_fbk_not_allocated 0.0638023 0
residue_neg_fbk_not_allocated 0.0638023 0
input_traffic_bytes_ 0.0638023 2480
avg_rtt_ 0.0638023 0.04

第一个字段描述了变量的名称。第二个字段给出了时间戳，接下来第三行给出了变量的值。XCPQueue 类的函数 trac_var()被用来跟踪 xcp 队列中的变量。 其他的分组跟踪可以使用下面的 tcl API 接口在 ns 中创建：

```
set f_all [open out.tr w]
$ns trace-all $f_all
```

首先打开一个文件，然后绑定文件描述符到 ns 跟踪对象，这样当每一个分组经过网路时，其踪迹都被记录并存储在输出文件中。 下面是输出文件的一个示例：

+ 0.003 4 0 xcp 40 ------- 2 4.0 1.2 0 0
- 0.003 4 0 xcp 40 ------- 2 4.0 1.2 0 0
r 0.013016 4 0 xcp 40 ------- 2 4.0 1.2 0 0
+ 0.013016 0 1 xcp 40 ------- 2 4.0 1.2 0 0
- 0.013016 0 1 xcp 40 ------- 2 4.0 1.2 0 0
r 0.023032 0 1 xcp 40 ------- 2 4.0 1.2 0 0
+ 0.023032 1 0 ack 40 ------- 2 1.2 4.0 0 1
- 0.023032 1 0 ack 40 ------- 2 1.2 4.0 0 1

r 0.033048 1 0 ack 40 ------- 2 1.2 4.0 0 1
+ 0.033048 0 4 ack 40 ------- 2 1.2 4.0 0 1
- 0.033048 0 4 ack 40 ------- 2 1.2 4.0 0 1
r 0.043064 0 4 ack 40 ------- 2 1.2 4.0 0 1
+ 0.043064 4 0 xcp 1200 ------- 2 4.0 1.2 1 2
- 0.043064 4 0 xcp 1200 ------- 2 4.0 1.2 1 2
+ 0.043064 4 0 xcp 1200 ------- 2 4.0 1.2 2 3
- 0.043544 4 0 xcp 1200 ------- 2 4.0 1.2 2 3

我们以第一行为例介绍跟踪文件记录的含义：
+ 0.003 4 0 xcp 40 ----- 2 4.0 1.2 0 0

+表示一个分组进入（节点 4 的）队列，此时分组正从节点 4 向节点 0 转发。你将会发现分组进队列（+），然后当其被发送到链路上等待下一个节点接收之后，又出队列（-）。分组类型是 xcp，大小为 40 字节。xcp 流的 id 为 2，分组头有一个为源节点/端口 id 为 4.0，目的节点/端口 id 为 1.2，唯一的分组 id 为 0。

21.4 XCP 测试集

xcp 的测试集有 3 个测试脚本。第一个类似于我们在前面小节中介绍过的一个脚本，其拓扑为 3 个 xcp 流共享一个瓶颈链路组成的哑铃型拓扑。第二个测试脚本类似于 3 个 xcp 流和一个 tcp 流共享相同的瓶颈链路。最后一个是在 Dina Katabi 的停车场试验基础上构建的，请参看她在 SIGCOMM’02 上发表的论文，这是 Dina 的示例的一个下载版本。测试采用 9 跳串行拓扑。10 个长 XCP 流，并且每个流都经过整个链状拓扑。10 个 XCP 流通过每一跳，从第（n-1）跳开始，在第 n 跳结束，从而创建断续流。最后，在反方向路径上也有长 XCP 流，从最后一跳（第 10 跳）开始，在第 1 跳结束。瓶颈链路在链状拓扑的中间。总的来说，这三个测试方案采用大型的和复杂的拓扑，并且给出了每一条链路的利用率、队列长度和分组丢弃率等值。

命令一览

下面是在 ns 中与 xcp 仿真相关的了命令列表：

set xcp_src [new Agent/TCP/Reno/XCP]
这个命令创建了一个 XCP 源代理

set xcp_dst [new Agent/XCPSink]
这个命令创建了一个 XCP sink

$ns duplex-link $R0 $R1 <BW>Mb <delay>ms XCP
这段代码在节点 R0 和 R1 之间创建了一个指定带宽和链路延时，并采用 XCP 路由器的双向链路。

$xcp_queue set-link-capacity <bandwidth_in_bits_per_sec>
这个命令将链路的带宽信息传播到 xcp 队列，路由器将利用这些信息来计算反馈。

set tfile [open tfile w] $xcp_queue attach $tfile
这个 Tcl 命令允许绑定一个文件来跟踪 xcp 队列参数。

$xcp_src attach-trace <file-descriptor>

$xcp_src tracevar <var-to-traced>

这个命令允许 xcp 源跟踪变量。

$queue queue-sample-everyrtt $rtt_value

这个命令允许用户设置 rtt 时间间隔值，每隔这一时间将在 xcp 队列中测量一次分组丢弃（packet_drops）和队列长度（queue length）等参数。

Queue/XCP set tcp_xcp_on_ 1

这个标记使得 tcp 和 xcp 流使用共同的 xcp 路由器。这是一个暂时的约定，将来一旦动态队列权值开始生效，则应该去除这个标记。

第 22 章

延迟器（DelayBox）：每条数据流的时延与丢包

延迟器是一个ns节点，应该放在源节点和目的节点的中间。通过延迟器，TCP流上的数据包在被传递到下一个节点之前，可以被延迟，丢弃，或者被迫通过一个瓶颈链路。一项分布可以为源 - 目标对指定时延，丢包，瓶颈链路速度。在源 - 目标对之间的每条数据流用这个分布来决定它们的特性。时延在延迟器中是以每条数据流计的，而不是每个数据包。既然延迟器区发了数据流，模拟中的fid_变量（流标识）应该为每条数据流单独设置。延迟器可用于TCP和FullTCP两种代理。

22.1 实现细节

延迟器维护了两个表：一个规则表和一个数据流表。规则表的入口由用户在OTcl模拟脚本中添加，并且给出了如何处理从源到目标节点的数据流的概况。其字段包括源，目标，随机变量(单位为ms), loss rate随机变量(丢失的分组的分数)，以及瓶颈链路速度随机变量（单位为Mbps)。其中瓶颈链路速度字段是可选的。数据流表的入口是内部创建的，具体指定了每条数据流怎样去处理。其字段包括源，目标，流ID，delay，loss，以及瓶颈链路速度（如果可应用的话）。Full-TCP 数据流这样定义：始于对一个新的流ID的第一个SYN消息的接收，止于第一个FIN消息发送后。第一个FIN消息发送后的数据包被立即转发（即，它们既没被延迟器延迟，也没被丢弃）。对于TcpAgent，其数据流定义为：始于对一个新流ID的第一个40个字节的数据包的接收。因为在TcpAgent中没有FIN数据包，TcpAgent数据流永远也不会结束，也不会从数据流表中移出。

延迟器同时维护了一些队列的集合，用于处理延迟的数据包。在数据流表的每个入口处都有一个对应的队列。这些队列都用delta队列实现，因为传送数据包的时间仅仅用来标识头一个数据包。其它的数据包以不同的时间存储，介于被传送和前一个分组被传送的时间之间。前一个分组应该被传送的实际时间存储在变量deltasum_，如此命名是因为这是队列中所有的delta值的总和（包括头数据包的传送时间）。如果瓶颈链路速度已经被指定，就会按照其速度值来计算每个数据包的delay，通过分割分组的大小。

当一个数据包被接收，它的传送时间（当前时间 + delay）就会被计算出来。（这个传送时间是分组的第一个比特将要传送的时间。在瓶颈链路上在此分组之后等候在队列中的分组必须延迟一定的时间来传送）。可以考虑两种场景来决定如何设定分组的delta值：

1. 如果分组要在队列中最后一个分组的最后一个比特之前传送，它的delta值（在传送前一个分组和这个分组之间的时间）应该被设为前一个分组的处理时延。这个分组必须在前一个分组后面排队，一旦前一个分组传送完毕就准备开始传送。
2. 如果分组要在队列中最后一个分组的最后一个比特之后传送，它的delta值就是这个分组的传送时间和上一个分组传送时间之差。

如果队列中的当前分组是唯一的，延迟器就会为分组的接收调度一个定时器。当这个定时器终止时，延迟器就会将分组传递到标准的转发器处理。一旦这个分组被传递，延迟器将查找队列中的下一个分组，并为其传送调度定时器。所有的分组，包括数据和ACK信号都以这种方式被延迟。

那些没有排队和被传递的分组应该被丢弃。相同连接的一个队列的所有分组的延迟值是相同的（除非由于分组大小产生的延迟），被丢弃的概率也是相同的。注：延迟器中的丢包不被trace-queue文件所记录。

22.2 示例

更多例子可在tcl/ex/delaybox/这个ns源代码的目录下找到。有效性脚本test-suite-delaybox在tcl/test/下，并且可在该目录下运行test-all-delaybox命令获得。

```
# test-delaybox.tcl - NS file transfer with DelayBox
# setup ns
remove-all-packet-headers; # removes all packet headers
add-packet-header IP TCP; # adds TCP/IP headers
set ns [new Simulator]; # instantiate the simulator
global defaultRNG
$defaultRNG seed 999
# create nodes
set n_src [$ns node]
set db(0) [$ns DelayBox]
set db(1) [$ns DelayBox]
set n_sink [$ns node]
# setup links
$ns duplex-link $db(0) $db(1) 100Mb 1ms DropTail
$ns duplex-link $n_src $db(0) 100Mb 1ms DropTail
$ns duplex-link $n_sink $db(1) 100Mb 1ms DropTail
set src [new Agent/TCP/FullTcp]
set sink [new Agent/TCP/FullTcp]
$src set fid_ 1
$sink set fid_ 1
# attach agents to nodes
$ns attach-agent $n_src $src
$ns attach-agent $n_sink $sink
# make the connection
$ns connect $src $sink
$sink listen
# create random variables
set recvr_delay [new RandomVariable/Uniform]; # delay 1-20 ms
$recvr_delay set min_ 1
$recvr_delay set max_ 20
set sender_delay [new RandomVariable/Uniform]; # delay 20-100 ms
$sender_delay set min_ 20
$sender_delay set max_ 100
set recvr_bw [new RandomVariable/Constant]; # bw 100 Mbps
$recvr_bw set val_ 100
set sender_bw [new RandomVariable/Uniform]; # bw 1-20 Mbps
$sender_bw set min_ 1
$sender_bw set max_ 20
set loss_rate [new RandomVariable/Uniform]; # loss 0-1% loss
```

```
$loss_rate set min_ 0
$loss_rate set max_ 0.01
# setup rules for DelayBoxes
$db(0) add-rule [$n_src id] [$n_sink id] $recvr_delay $loss_rate $recvr_bw
$db(1) add-rule [$n_src id] [$n_sink id] $sender_delay $loss_rate $sender_bw
# output delays to files
$db(0) set-delay-file "db0.out"
$db(1) set-delay-file "db1.out"
# schedule traffic
$ns at 0.5 "$src advance 10000"
$ns at 1000.0 "$db(0) close-delay-file; $db(1) close-delay-file; exit 0"
# start the simulation
$ns run
```

22.3 命令一览

下述基于DelayBox类的命令可通过OTcl来访问：

[$ns DelayBox]

创建了一个新的DelayBox节点。

$delaybox add-rule <srcNodeID> <dstNodeID> <delayRV> [<lossRV>] [<linkSpeedRV>]

在规则表中添加一条规则，指定时延，丢包率，瓶颈链路速度随机变量。时延是必须的，后两者是可选的。

$delaybox list-rules

列表规则表中的所有规则。

$delaybox list-flows

列表数据流表中的所有数据流。

$delaybox set-asymmetric

指定了时延必须只在数据路径上，而不是应用在数据和ACK共同路径上。

$delaybox set-delay-file <filename>

为每条数据流的时延输出到指定的文件名。格式为： srcNode dstNode fid delay

$delaybox close-delay-file

关闭写入delay的文件。

$delaybox set-debug <int>

设定调试级：

- 1: 当延迟器中有分组丢弃时输出
- 2: Level 1 +

在每次队列操作时队列的内容。

第 23 章

在NS-2.31 中实现IEEE 802.15.4 的变化

在下面，在ns2.31发布版中，对IEEE 802.15.4模块所做的改变，连同其修改所影响到的一系列文件作介绍。本文件由Iyappan Ramachandran编写。

23.1 关闭无线电（radio）

在这个发布版中，添加了WPAN节点的睡眠功能：

1. 在./wpan/p802_15_4def.h中定义了一个叫做SHUTDOWN的宏（macro），用于提供当没有任何分组传送的时候对节点的关闭功能。目前，还没有直接在tcl接口上对无线电关闭的支持，但间接的方法已经出现（见第4点）。

2. 添加了两个函数Phy802_15_4::wakeupNode()和Phy802_15_4::putNodeToSleep()，可以调用它们来完成对节点的关闭和唤醒。这些函数主要用来降低在休眠状态下的能量损耗。影响到的文件包括：./wpan/p802_15_4phy.cc, ./wpan/p802_15_4phy.h

3. 添加了一个新的叫做macWakeupTimer的定时器，用于为节点设定一个闹钟在其关闭前将其唤醒（用于信标节点接收，等等）。定时器终止时调用Phy802_15_4::wakeupNode()。影响到的文件包括： ./wpan/p802_15_4mac.cc, ./wpan/p802_15_4mac.h, ./wpan/p802_15_4timer.cc, ./wpan/p802_15_4timer.h, ./wpan/p802_15_4csmaca.h。

4. 变量P_sleep_（休眠状态的能量损耗），P_transition _（在休眠 - 唤醒的转化过程中的损耗）以及T_transition_（休眠 - 唤醒的转化时间）已经出现中mac/wireless-phy.h中。T_transition_以前并没有初始化，但是现在有了。另外，一个新的变量T_sleep_被加入到wireless-phy，用来指示哪个无线电可以被关闭的时间。这可用变量名字sleepTime（见./tcl/ex/wpan_demo_sleep.tcl）通过tcl接口来设定。因此，可以保持对SHUTDOWN的定义，但是要将sleepTime设为一个很大的值来来使得在整个模拟中无线的关闭功能都处于无效中。这提供了一种从tcl接口打开/关闭无线电的方法。影响到的文件包括：mac/wireless-phy.h。

5. 当MAC接收到一个要传送的分组，无线电如果处于睡眠中应该被唤醒。类似的，一个休眠的节点要被唤醒以接收信标，而不管它们何时才能到达。如果无线电的关闭功能被激活，无线电必须要等到分组传送完后才能进入睡眠。Mac802_15_4::recv()通过调用 Phy802_15_4::wakeupNode()和Phy802_15_4::putNodeToSleep()实现了这个功能，通过休眠降低了能量损耗，影响到的文件包括：./wpan/p802_15_4mac.cc。

6. 在每次信标节点接收之后，当无线电关闭功能激活时，如果没有分组需要传输，节点就可以关闭自身了。这是由Mac802_15_4::recvBeacon()通过调用Phy802_15_4::putNodeToSleep()来完成的，影响到的文件包括：./wpan/p802_15_4mac.cc。

7. 如果节点由于没被使用而进入休眠时，休眠 - 空闲转化必须被计算。这是由CsmaCA802_15_4::start()来完成的，第一

个退避阶段的退避时间这样计算：aswtime=MAX(wtime,ceil(phy→T_transition_local_，影响到的文件包括：./wpan/p802_15_4csmaca.cc。

23.2 其它变化

1. 在退避了macMaxCSMABackoffs后不能发送数据包，MAC层必须报告一个通道访问的错误。过去的实现试图不确定地传送数据包，而不是报告通道访问错误。这在Mac802_15_4::mcps_data_request()函数中已经修正了。同时节点也在这个阶段进入休眠（如果有必要的话）。影响到的文件包括：./wpan/p802_15_4mac.cc。

2. 添加了一个新的叫做aCCATime的常量，用来指示在symbol周期的CCA持续时间。影响到的文件包括：./wpan/p802_15_4const.h。

3. CCA过程被指定了8个symbol过程。在过去的实现中，CCA在第8个symbol过程的结尾完成，以用来决定通道的空闲状态。结果，如果通道在第一个8个symbol过程后依然空闲（这很有可能），实现就会指示为通道为空闲，当然这在实际上是不应该的。现在已经被修改为在第4个CCA symbol过程结尾处完成，但是在第8个报告通道状态。为了达到这个目的，添加了一个新的定时器CCAReportH，在终止时调用CCAReportHandler用来产生报告。影响到的文件包括：./wpan/p802_15_4phy.cc, ./wpan/p802_15_4phy.h。

4. Phy802_15_4::PD_DATA_indication()函数调用了WirelessChannel::sendUp()用来检查分组是否被正确的接收，以及在分组接收过程中是否能量的损耗降低了。SendUp()函数已经被recv()调用，并且再度被其调用，用来二次降低能量损耗。这个错误已经在Phy802_15_4::PD_DATA_indication()中得以修正。影响到的文件包括：./wpan/p802_15_4phy.cc。

5. 从通道接收分组的Phy802_15_4::recv()函数用WirelessPhy::sendUp()检查分组是否被正确接收，当分组被释放时便失败。sendUp()在节点休眠时，或者当分组接收能量低于CS阈值时返回一个0值。在前一种情况下，变量rxTotPower和rxTotNum由于CS目的，需要在丢包前被更新；而后一种情况下，分组只是需要简单的被丢弃。Zheng的实现没有更新变量而丢弃了所有的包。这点已经在Phy802_15_4::recv()得以，影响到的文件包括： ./wpan/p802_15_4phy.cc 。

6. 接收器必须被打开用于载波侦听操作，因此在这段时期的消耗为接收能量Pr。早期的实现由于载波侦听而没有减少接收能量。这已经加入了Phy802_15_4::CarrierSenser()函数。同时，能量也在发送 - 接收转化中被消化。这同样被计入了。影响到的文件包括：./wpan/p802_15_4phy.cc。

第三篇

支持

第 24 章

ns调试

ns 是一个用 C++编写的仿真引擎，并且使用 OTcl（面向对象的 Tcl）语言作为配置和命令行交互的接口。然而为了调试 ns，我们需要同时关注 OTcl 和 C++这两种语言。本章分别给出了 Tcl 层次以及 C++层次的调试说明，并且展示如何在 Tcl 和 C++这两个层次之间，进行来回的交互调试。我们还会简短的介绍 ns 中内存调试以及内存开销的内容。

24.1 Tcl层次的调试

Ns支持Don Libs' Tcl调试工具（这个调试工具的附录文档可以从http://expect.nist.gov/tcl-debug/tcl-debug.ps.Z上获得，调试工具的源代码则可以在http://expect.nist.gov/tcl-debug/tcl-debug.tar.gz上获得）。直接安装或者将调试工具的源代码放在与ns-2平行的目录下并编译它。和tcl-debug文档中说明的不一样的是，这个调试工具并不支持-D选项。在你的脚本的合适位置加入一行“debug 1”，就可以进入调试器。在调试ns时，你可以打开调试选项，即键入命令的时候使用选项“-enable-debug”，如果你的Tcl调试工具没有安装在ns-2所在的目录下，你可以使用选项“–with-tcldebug=<give/your/path/to/tcldebug/library>”来指明你的调试工具所在的位置。

一个有用的调试命令是$ns_gen-map，这个命令会以原始的形式列出所有的OTcl对象。你可以方便的通过给定一个对象的名字，将这个对象的位置和它的函数关联在一起。一个对象的名字是一个OTcl句柄，通常的形式是_o###。对于TclObjects，在C++的调试工具中也是可见的，比如在gdb中使用this->name_。

24.2 C++层次的调试

C++层次有很多标准的调试工具可以使用。使用下面的 gdb 中的宏，可以简单的看到一个 Tcl 子程序内部运行的情况。

```
## for Tcl code
define pargvc
set $i=0
      while $i < argc
            p argv[$i]
            set $i=$i+1
      end
end
document pargvc
Print out argc argv[i]' s common in Tcl code.
(presumes that argc and argv are defined)
end
```

24.3 混合调试Tcl和C

一个痛苦的实现是，当你正在编辑Tcl代码并且调试Tcl层次的内容时，你常常想去看看C层次的类，同样当调试C代码时也会有这样的经历。下面我们会给出一个简略的提示让这个任务容易一些。如果你使用gdb来调试ns，下面的代码会让你访问Tcl调试器。至于接下来应该如何联合使用这些调试工具，我们在这章的前面列出的URL中的文档将会给你答案(http://expect.nist.gov/tcl-debug/tcl-debug.ps.Z)。

```
(gdb) run
Starting program: /nfs/prot/kannan/PhD/simulators/ns/ns-2/ns
...
Breakpoint 1, AddressClassifier::AddressClassifier (this=0x12fbd8)
at classifier-addr.cc:47
(gdb) p this->name_
$1 = 0x2711e8 "_o73"
(gdb) call Tcl::instance().eval("debug 1")
15: lappend auto_path $dbg_library
dbg15.3> w
*0: application
15: lappend auto_path /usr/local/lib/dbg
dbg15.4> Simulator info instances
_o1
dbg15.5> _o1 now
0
dbg15.6> # and other fun stuff
dbg15.7> _o73 info class
Classifier/Addr
dbg15.8> _o73 info vars
slots_ shift_ off_ip_ offset_ off_flags_ mask_ off_cmn_
dbg15.9> c
(gdb) w
Ambiguous command "w": while, whatis, where, watch.
(gdb) where
#0 AddressClassifier::AddressClassifier (this=0x12fbd8)
at classifier-addr.cc:47
#1 0x5c68 in AddressClassifierClass::create (this=0x10d6c8, argc=4,
argv=0xefffcdc0) at classifier-addr.cc:63
...
(gdb)
```

同样的 如果你用 gdb 调试 ns ,你可以随时发送一个中断来引起 gdb 的关注 ,通常在 berkeloidrones 上按下<Ctrl-c>。但是，这有时候会打乱栈的指示器，有时（这种情况非常罕见）甚至会弄乱整个栈，这会导致程序无法继续运行下去。幸运的是，gdb 现在已经作的非常灵活，出现这种情况时会给出警告，这时你应该更加小心翼翼的进行调试。

24.4 内存调试

当你内存耗尽时，首先要做的事是确认你可以使用系统的所有内存。有些系统限制了单个程序能使用的内存总数。为了解除这个限制，你应该使用 limit 或者 ulimit 命令。这些都是 shell 命令———详细信息请参看 shell 手册。Limit 是 csh 的命令，ulimit 则是 sh/bash 的命令。

大型网络的仿真会消耗非常多的内存。Ns-2.0b17 支持Gray Watson的dmalloc用户库（这个库的文档参看：http://www.letters.com/dmalloc/，库的源代码则在：ftp://ftp.letters.com/src/dmalloc/dmalloc.tar.gz）。要使用这个库，你可以安装它或者直接将它放在ns-2 所在的文件夹下，在运行ns时则使用配置选项-with-dmalloc。然后build所有你想获得内存信息并且被标注了调试符号的组件（至少应该包括ns-2，可能会有tclcl和otcl，也许还会有tcl）。

24.4.1 使用dmalloc

使用 dmalloc 的步骤如下：

- 定义一个别名
  csh：’ eval ‘\dmalloc -C \!* ‘’ ，
  bash: function dmalloc { eval ‘command dmalloc -b $* ‘ }%$

- 下一步打开调试器输入 dmalloc -l logfile low

- 运行你的程序（配置和 built 的方法见前面的描述）

- 解释 logfile 则需要运行 dmalloc_summarize ns <logfile。（你还需要单独下载 damlloc_summarize）

在有些平台上，你或许需要静态的链接一些东西使得 dmalloc 能够工作。在 Solaris 中，则要连接这些选项："-Xlinker -B -Xlinker -static libraries -Xlinker -B -Xlinker –dynamic -ldl -lX11 -lXext"。（你需要改变 Makefile。感谢 Haobo Yu 和 Doug Simth 指出了这一点）。

我们可以解释一个由 ns-2/tcl/ex/newmcast/cmcast-100.tcl 脚本产生的样本摘要，它报告了第 200 个 duplex-link-of-interfaces 状态后的退出状态。

Ns allocates 6MB of memory.
1MB is due to TclObject::bind
900KB is StringCreate, all in 32-byte chunks
700KB is NewVar, mostly in 24-byte chunks

Dmalloc_summarize 必须根据函数的地址来映射函数的名字。它经常不能解析共享库的地址，因此如果你看到很多内存被分配到以“ra=”开头的项时，就是因为这个原因。解决这个问题的最好方法就是静态的 build ns（如果不能全部构建，就尽可能多的构建）。

Dmalloc 的内存分配模式开销是比较大的，外加簿记（bookkeeping）开销。dmalloc 链接的程序比起标准的 mallocs 链接的内存开销要大。

Dmalloc 能分析其它内存错误（重复释放，缓冲区溢出，等等），详细信息请参看文档。

24.4.2 内存开销说明

这里有一些节约内存的技巧（这些技巧中的一些使用 cmcast-100.tcl 脚本中的例子）：如果你有很多链路和节点：

避免 trace-all: $ns trace-all $f 会导 trace 对象跟踪所有的链路。如果你只想跟踪一个链路，则没必要使用这个命令。这能为你节省 14kb/link。

用数组存储顺序变量：每一个变量，比如集合 n$i [$ns node]中的变量 n$i，都会有一定开销。如果节点序列由数组存储，比如 n($i)，则只会创建一个变量，这样就会减少内存消耗。可以节省 40+Byte/variable

避免不必要的变量：如果一个对象只使用一次，则不需要为它声明。比如，直接 new CtrMcast $ns $n(1) $ctrmcastcomp [list 1 1]比起 set cmcast(1) [new CtrMcast $ns $n(1) $ctrmcastcomp [list 1 1]]要好的多。将会节省 80Byte/variable。

在 FreeBSD 上运行：FreeBSD 的 malloc()开销比其它系统都要少。我们最后会将这个函数发布到所有平台。

动态绑定：C++中为每一个你创建的对象使用 bind()都会产生开销。如果你创建了很多一样的对象，这种方法的开销将会非常庞大。将每一个类的 bind()都改成 delay_bind()会改善内存需求。请参看 ns/object.cc 的绑定的例子。

禁用分组头部：对于分组很多的仿真，如果禁用你不需要使用的分组头部将会节省很多内存开销。详细信息参看 12.1 节。

24.4.3 Dmalloc收集的数据统计

在最近的仿真中出现了内存开销问题（cmcast-[150,200,250].tcl），因此我们决定仔细研究一下此问题。在http://www-mash.cs.berkeley.edu/ns/ns-scaling.html中列出了我们找到的一些瓶颈。

下表总结了我们找到的瓶颈：

KBytes	*cmcast-50.tcl(217 Links)*	*cmcast-100.tcl(950 Links)*
trace-all	8,084	28,541
turn off trace-all	5,095	15,465
use array	5,091	15,459
remove unnecessay variables	5,087	15,451
on SunOS	5,105	15,484

24.5 内存泄漏

这一节将处理 ns 中 Tcl 方面和 C/C++方面的内存泄漏问题。

24.5.1 OTcl

OTcl，尤其 TclCL，提供了分配新对象的方法。但是，它并没有提供一个垃圾回收机制。这很容易导致内存泄漏。重要的是：像 dmalloc 和 purify 这样的工具是不能检查出这种内存泄漏问题的。例如，下面这段 OTcl 脚本：

```
set ns [new Simulator]
for set i 0 $i < 500 incr i
    set a [new RandomVariable/Constant]
```

你可能会认为在这段代码执行后只会消耗一个变量的内存。然而，由于 OTcl 没有垃圾回收机制，当生成第二个 RandomVariable 时，前一个并没有被释放掉因此导致了内存泄漏。不幸的是，并没有一个简单的方法解决这个问题，因为垃圾回收机制和 Tcl 的精神背道而驰。唯一的方法就是显示地释放你脚本中的每一个 OTcl 对象，同样你需要注意 C/C++里的 malloc-ed 对象。

24.5.2 C/C++

另一个内存泄漏的源头是 C/C++。使用特定的工具来进行泄露跟踪，比如 dmalloc 和 purity，相比 OTcl 要容易得多。ns 有一个特定的目标 ns-pure 可以编译生成一个“纯的”可执行 ns。首先确认，对于你的连接器，ns 的 Makefile 中的宏 PURIFY 包含了正确的-collector（如果你不知道这是什么，请参考 purify 的 man 帮助）。然后简单的键入 make ns-pure。请参看这章前面介绍过的使用 ns 和 libdmalloc。

第 25 章

对数学的支持

模拟器包含了一些用来执行随机变量生成和整合的数学函数，现在，这方面内容正在不断调整中。

这章描述的过程和函数可以在以下地方找到：~*ns*/tools/rng.{cc, h}, ~*ns*/tools/random.{cc, h},~*ns*/tools/ranvar.{cc, h}, ~*ns*/tools/pareto.{cc, h}, ~*ns*/tools/expoo.{cc, h}, ~*ns*/tools/integrator.{cc, h}, and ~*ns*/tcl/lib/nsrandom.tcl。

25.1 随机数生成器

RNG类包含了一个由L’Ecuyer[16]提议的合成多重递归发生器MRG32k3a。它的C++代码从[18]改编而来。这代替了RNG类的前一实现版本，前一版本使用了Park和Miller[27]的最低标准乘法线性同余发生器。新的RNG（MRG32k3a）使用在ns 2.1b9及以后版本。

MRG32k3a发生器能产生 $1.8\mathbf{x}10^{19}$ 个不同的随机数字流。每个流包含 $2.3\mathbf{x}10^{15}$ 个子流。每个子流有一个长度为 $7.6\mathbf{x}10^{22}$ 的周期（例如，随机数字重复之前的数字），生成器的全部周期为 $3.1\mathbf{x}10^{57}$ ，图25.1图示了流如何与子流结合。

一个默认的RNG（defaultRNG），在模拟器初始化时被创建。如果仿真中使用了多个随机变量，那么每个随机变量都应该是一个单独的RNG对象。当一个RNG对象被创建时，该对象自动地种入到下一个独立随机数字流的开头。使用这种特性，允许使用最大 $1.8\mathbf{x}10^{19}$ 个随机变量。

有时候需要使用一个仿真的多份不同的拷贝（比如，为了执行统计分析多次运行固定参数的仿真），对于每一份拷贝，应该使用不同的子流来保证随机数字流是独立的（后面给出了一个此过程的OTcl例子）。这允许启用最大 $2.3\mathbf{x}10^{15}$ 个独立拷贝。一个拷贝中的每一个随机变量在重复之前能够产生最多 $7.6\mathbf{x}10^{22}$ 个随机数字。

注：此处只描述了最常见的函数，更多的信息请参见[18]和tools/rng.h 、tools/rng.cc中的源码。此处RNG和更常见的LCG16807 RNG的比较（以及为什么LCG16807不是一个好的RNG）请参见[17]。

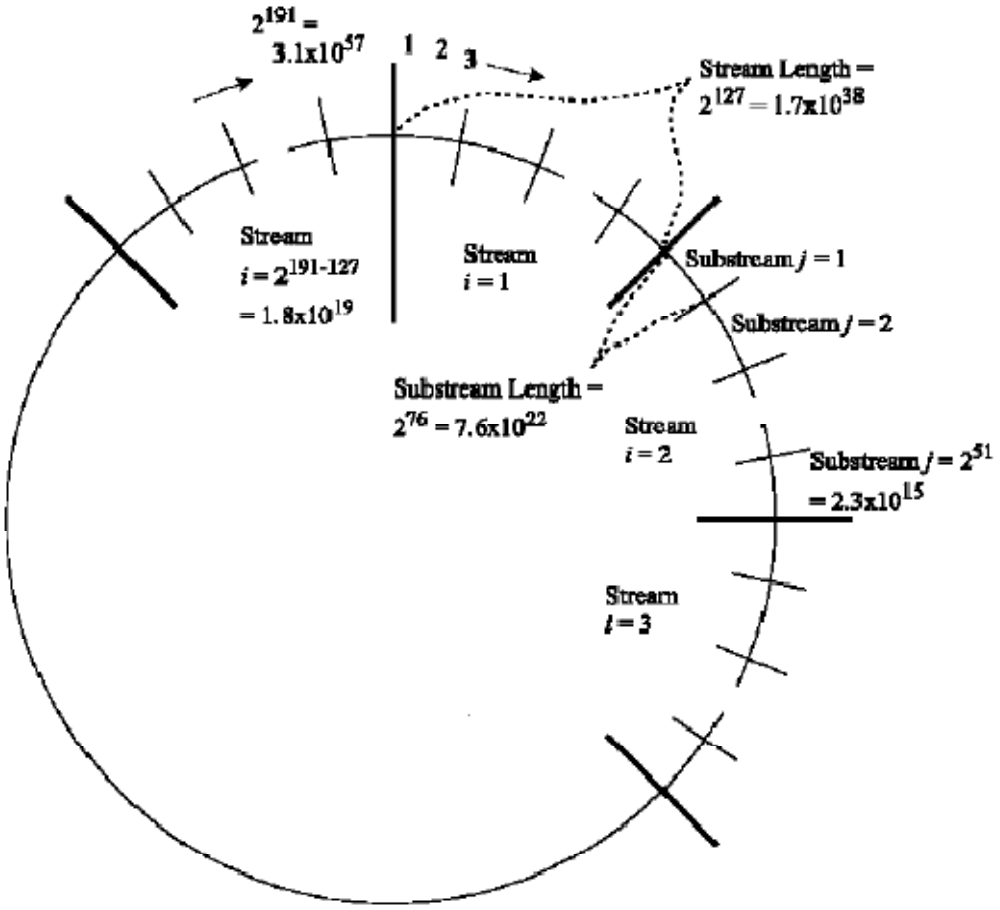


图25.1：流和子流的总体布局[18]

25.1.1 RNG种子

由于RNG的特性和它的实现方式，没有必要为它设置种子（默认种子是12345）。如果你想改变种子，有许多函数可用。你应该只设置默认的RNG种子，你创建的其它RNG会自动地种入它们产生的独立流。种子的有效范围是1到MAXINT。

为了得到非确定的行为，可以设置默认RNG的种子为0。这将根据当前的时间和计算器设置种子。这种方法不应该用来设置独立拷贝的种子。因为这不能保证两个随机种子产生的流不会重复。唯一能够保证两个流不重复的方法是使用由RNG实现提供的子流能力。

例子：

```
# Usage: ns rng-test.tcl [replication number]

if {$argc > 1} {
	puts "Usage: ns rng-test.tcl \[replication number\]"
	exit
}
set run 1
if {$argc == 1} {
	set run [lindex $argv 0]
```

```
}
if {$run < 1} {
        set run 1
}
# 为默认的RNG设置种子
global defaultRNG
$defaultRNG seed 9999

# 创建RNG并把它们配置到正确的子流上
set arrivalRNG [new RNG]
set sizeRNG [new RNG]
for {set j 1} {$j < $run} {incr j} {
        $arrivalRNG next-substream
        $sizeRNG next-substream
}

# arrival_是一个描述连续包到达时间的指数型随机变量
set arrival_ [new RandomVariable/Exponential]
$arrival_ set avg_ 5
$arrival_ use-rng $arrivalRNG

# size_ 是一个描述包大小的均匀分布随机变量
set size_ [new RandomVariable/Uniform]
$size_ set min_ 100
$size_ set max_ 5000
$size_ use-rng $sizeRNG

# 打印前5个到达包的时间和大小
for {set j 0} {$j < 5} {incr j} {
        puts [format "%-8.3f %-4d" [$arrival_ value] \
                [expr round([$size_ value])]]
}
```

输出：

```
% ns rng-test.tcl 1
6.358 4783
5.828 1732
1.469 2188
0.732 3076
4.002 626
% ns rng-test.tcl 5
0.691 1187
0.204 4924
8.849 857
```

2.111 4505
3.200 1143

25.1.2 OTcl支持

支持命令：

OTcl中可以访问RNG类中的下列命令，它们位于tools/rng.cc中：
seed n – 设置RNG的种子为n，如果n == 0，这个种子将被依据当前的时间和计算器设置next-random – 返回下一个随机数字
seed – 返回当前种子的值
next-substream –前进到下一个子流
reset-start-substream – 重置流为当前子流的初始。
normal avg std –返回一个给定均值及标准差的正态分布的抽样数字
lognormal avg std –返回一个给定均值及标准差的对数正态分布的抽样数字

OTcl中可以访问RNG类中的下列命令，它们位于tcl/lib/ns-random.tcl中：
exponential mu –返回一个均值为mu的指数分布的抽样数字
uniform min max – 返回一个在[min,max]上均匀分布的抽样整数
integer k – 返回一个在[0,k-1]上均匀分布的抽样整数

例子：

```
# Usage: ns rng-test2.tcl [replication number]
if {$argc > 1} {
puts "Usage: ns rng-test2.tcl \[replication number\]"
exit
}
set run 1
if {$argc == 1} {
set run [lindex $argv 0]
}
if {$run < 1} {
set run 1
}
#  默认的RNG种子是12345
#创建RNG并把它们配置到正确的子流上
set arrivalRNG [new RNG]
set sizeRNG [new RNG]
for {set j 1} {$j < $run} {incr j} {
$arrivalRNG next-substream
  $sizeRNG next-substream
}
#打印前5个到达包的时间和大小
```

```
for {set j 0} {$j < 5} {incr j} {
puts [format "%-8.3f %-4d" [$arrivalRNG lognormal 5 0.1] \
[expr round([$sizeRNG normal 5000 100])]]
}
```

输出

```
% ns rng-test2.tcl 1
142.776 5038
174.365 5024
147.160 4984
169.693 4981
187.972 4982
% ns rng-test2.tcl 5
160.993 4907
119.895 4956
149.468 5131
137.678 4985
158.936 4871
```

25.1.3 C++支持

成员函数：

随机数生成器在RNG类中实现，它的定义在tools/rng.h中。
注：tools/random.h中的Random类是标准随机数字流的原有（旧的）接口。
成员函数提供了下列操作：

void set_seed (long seed) – 设置RNG的种子，如果 seed == 0，这个种子将被依据当前的时间和计算器设置
long seed (void) – 返回当前的种子
long next (void) –返回在[0,MAXINT]上的下一个随机整数
double next_double (void) – 返回在[0, 1]上的下一个随机数
void reset_start_substream (void) –重置流为当前子流的开始
void reset_next_substream (void) –前进到下一个子流
int uniform (int k) –返回一个在[0,k-1]上均匀分布的抽样整数
double uniform (double r) –返回一个在[0,r]上均匀分布的抽样数字
double uniform (double a, double b) –返回一个在[a,b]上均匀分布的抽样数字
double exponential (void) –返回一个均值为1.0的指数分布的抽样数字
double exponential (double k) –返回一个均值为k的指数分布的抽样数字
double normal (double avg, double std) –返回一个给定均值及标准差的正态分布的抽样数字
double lognormal (double avg, double std) –返回一个给定均值及标准差的对数正态分布的抽样数字

例子：

```
/* 创建新的RNGs */
RNG arrival (23456);
RNG size;
```

```
/* 把RNGs配置的合适的子流 */
for (int i = 1; i < 3; i++) {
arrival.reset_next_substream();
size.reset_next_substream();
}
/* 打印前5个到达包的时间和大小 */
for (int j = 0; j < 5; j++) {
printf ("%-8.3f %-4d\n", arrival.lognormal(5, 0.1),
int(size.normal(500, 10)));
}
```

输出

161.826 506
160.591 503
157.145 509
137.715 507
118.573 496

25.2 随机变量

RandomVariable类在基本的随机数字发生器和默认的随机数字流之上加了一层薄薄的功能。它的定义在~ns/ranvar.h中：

```
class RandomVariable : public TclObject {
    public:
        virtual double value() = 0;
        int command(int argc, const char*const* argv);
        RandomVariable();
    protected:
        RNG* rng_;
};
```

从这个抽象类派生出的类实现了不同的分布算法。每种分布都被恰当的参数值参数化了。返回值函数用来从分布中返回一个值。

现在已定义的分布，它们相关的参数是：

class UniformRandomVariable min_, max_
class ExponentialRandomVariable avg_
class ParetoRandomVariable avg_, shape_
class ParetoIIRandomVariable avg_, shape_
class ConstantRandomVariable val_
class HyperExponentialRandomVariable avg_, cov_
class NormalRandomVariable avg_, std_
class LogNormalRandomVariable avg_, std_

RandomVariable类在OTcl中是可用的，比如，产生一个[10,20]上服从均匀分布的随机变量：

```
set u [new RandomVariable/Uniform]
$u set min_ 10
$u set max_ 20
$u value
```

默认情况下，RandomVariable对象使用上一节介绍的默认的随机数字发生器。use-rng方法可以把一个RandomVariable和一个非默认的RNG联系起来：

```
set rng [new RNG]
$rng seed 0
set e [new RandomVariable/Exponential]
$e use-rng $rng
```

25.3 积分

Integrator类支持通过（离散）相加的集成逼近（连续）。它定义在~ns/integrator.h中，integrator.h中的有：

```
class Integrator : public TclObject {
    public:
        Integrator();
        void set(double x, double y);
        void newPoint(double x, double y);
        int command(int argc, const char*const* argv);
    protected:
        double lastx_;
        double lasty_;
        double sum_;
};
```

Integrator.cc中有：

```
Integrator::Integrator() : lastx_(0.), lasty_(0.), sum_(0.)
{
    bind("lastx_", &lastx_);
    bind("lasty_", &lasty_);
    bind("sum_", &sum_);
}
void Integrator::set(double x, double y)
{
    lastx_ = x;
    lasty_ = y;
    sum_ = 0.;
```

```
}
void Integrator::newPoint(double x, double y)
{
	sum_ += (x - lastx_) * lasty_;
	lastx_ = x;
	lasty_ = y;
}
int Integrator::command(int argc, const char*const* argv)
{
	if (argc == 4) {
		if (strcmp(argv[1], "newpoint") == 0) {
			double x = atof(argv[2]);
			double y = atof(argv[3]);
			newPoint(x, y);
			return (TCL_OK);
		}
	}
	return (TclObject::command(argc, argv));
}
```

这个类是一个基础类，其他的类比如QueueMonitor可以使用它来保存运行中的相加结果。每一个相加操作中的新元素都被函数newPoint(x,y)加起来。newPoint执行第k次以后，运行相加结果为 $\sum_{i=1}^{k} y_{i-1}(x_i - x_{i-1})$ ，这里如果lastx_，lasty_，或者sum_没有通过OTcl重置，则 $x_0 = y_0 = 0$ 。注意，相加操作中的一个新点，既可以被C++成员函数newPoint也可以被OTcl成员函数newpoint相加。使用积分计算特定类型（比如，平均队列长度）的均值可以在（pp. 429–430, [15]）找到。

25.4 ns-random

ns-random 是一个已经废弃了的产生随机数的方法。提供这个信息只是为了保持向后兼容性。ns-random的实现在~ns/misc.{cc,h}。无参数调用时，它使用在[0,MAXINT]上服从均匀分布的方式生成一个随机数。当提供一个参数时，它把这个参数作为随机生成器的种子。

特别地，当ns-random 0被调用时，它根据当前时间随机产生生成器的种子。这个特性可以用来在运行过程中产生非确定的结果。

25.5 一些数学支持相关的对象

INTEGRATOR OBJECTS

积分对象支持使用离散相加的连续积分的近似计算。运行相加(积分) 计算如下： sum_ += [lasty_ * (x lastx_)] 这里 (x, y) 是最后传进来的元素，而(lastx_, lasty_)是先前被加入 sum 的元素。 lastx_和 lasty_被更新作为新的元素加入。第一个例子的点默认为（0,0）,这可以通过改变(lastx_,lasty_)的值来改变它。

$integrator newpoint <x> <y>
把点(x,y) 加入sum。 注意x比lastx_小是没有意义的。

状态变量：
lastx_ 最后取样点的x坐标
lasty_ 最后取样点的y坐标
sum_ 所有取样点的运行和（比如积分）

SAMPLES OBJECT
样本对象支持给定样本的均值和方差统计的计算。
$samples mean
返回样本的均值
$samples variance
返回样本的方差
$samples cnt
返回一个被考察的取样点的计数
$samples reset
重置样本对象，以监测一套新的样本。

这个对象没有特殊的配置参数或状态变量。

25.6 命令一览

下面列出仿真脚本中经常使用的数学支持相关的命令列表：
set rng [new RNG]
这个命令创建了一个新的随机数产生器。

$rng seed <0 or n>
这个命令为RNG设置种子。如果0被指定，则随机（试探性）地为RNG产生种子。否则设置RNG的种子为n

$rng next-random
这个命令从RNG返回下一个随机数字

$rng uniform <a> <b>
这个命令返回在<a>到<b>上服从均匀分布的一个数字。

$rng integer <k>
这个命令返回在0，k-1上服从均匀分布的一个整数。

$rng exponential
这个命令返回一个数字，它服从均值为1的指数分布。

set rv [new Randomvariable/<type of random-variable>]
这个命令创建了一个随机变量的实例，这个实例可以产生服从特定分布的随机变量。

不同类型的随机变量来源于下面的基类：
RandomVariable/Uniform, RandomVariable/Exponential,
RandomVariable/Pareto, RandomVariable/Constant,
RandomVariable/HyperExponential.
每种分布类型都被恰当的参数值参数化了，详情请参见本章的25.2部分。

$rv use-rng <rng>
这个方法用来把一个随机变量和一个非默认RNG关联起来，否则，默认情况下，这个随机变量会和默认的随机数字发生器关联。

第 26 章

对跟踪和监控的支持

本章中描述的过程和函数可以在以下地方找到：~*ns*/trace.{cc, h}, ~*ns*/tcl/lib/ns-trace.tcl, ~*ns*/queuemonitor.{cc, h}, ~*ns*/tcl/lib/ns-link.tcl, ~*ns*/packet.h, ~*ns*/flowmon.cc, 和~*ns*/classifier-hash.cc。

在一个仿真器中有许多方法收集输出或者跟踪数据。一般地，跟踪数据或者在仿真执行过程中直接地显示出来，或者存储在一个文件中以备以后处理和分析 ,后一种方法较常用。目前仿真器主要支持两种不同类型的监控方法。第一种叫做traces，记录每一个在链路或者队列中到达、离开或者丢弃的数据包。trace对象被当作网络拓扑中的节点配置到仿真中，并用一个Tcl “Channel”对象链接它们，这个对象代表收集数据的目的地（当前目录下的trace文件是典型情况）。另一种叫做monitors（监控），记录各种有用的数据，例如包或字节的到达、离开等等。monitors 或者基于单流的flow monitor（见下面26.7部分）能够监控与包有关的数值。

为了支持跟踪，每一个包里都有一个特殊的common包头（作为hdr_cmn结构体被定义在~*ns*/packet.h中）。该包头中一般包含一个独一无二的标识符，一个包类型域（在agents产生包的时候被设置），一个包大小域（单位bytes，用于计算包的传输时间），一个接口标签（用于计算多播分布树）。

监控由一些单独的对象支持，这些对象在队列周围被创建并插入到网络拓扑中。它们提供了一个聚集接收数据统计和时间信息的地方，并用类Integrator（25.3部分） 来计算超过时间间隔的统计信息。

26.1 对跟踪的支持

Otcl中对跟踪的支持，包括一些特殊的在Otcl 中可见但却在C++中执行的类，以及一套Tcl 的帮助程序和ns 库中定义的类。

以下所有Otcl 类都由在~ns/trace.cc 中定义的C++类支持。下面所有类型的对象被直接内嵌到网络拓扑中。

Trace/Hop 跟踪一个“跳”
Trace/Enque 一个包到达（通常用在队列上）
Trace/Deque 一个包离开（通常用在队列上）
Trace/Drop 包丢失(即包传给drop-target对象)
Trace/Recv 链路的目的节点的包接收事件
SnoopQueue/In 输入、采集到一个时间/大小样本（传递数据包）
SnoopQueue/Out 输出、采集到一个时间/大小样本（传递数据包）
SnoopQueue/Drop 丢弃、采集到一个时间/大小样本（传递数据包）
SnoopQueue/EDrop “提前”丢弃、采集到一个时间/大小样本（传递数据包）

参考上面列出的那些对象，下面这些类型的对象被加入到仿真中，它们用来收集由SnoopQueue 对象采集到的统计数字：

QueueMonitor 接收并聚集从探测器采集到的样本

QueueMonitor/ED 队列监视的一种，具有区别“提前”和标准的包丢失能力
QueueMonitor/ED/Flowmon 每个流的统计信息的监测器（管理者）
QueueMonitor/ED/Flow 每个流的统计信息的容器
QueueMonitor/Compat 当使用兼容的ns v1时，对标准QueueMonitor 的代替

26.1.1 Otcl 的帮助函数

下面的帮助函数可以在仿真脚本中使用，可以帮助添加跟踪元素(请参考 ~*ns*/tcl/lib/ns-lib.tcl)；它们是Simulator 类的实例程序：

flush-trace{} 清空仿真中的所有跟踪对象的缓冲区

create-trace{type file src dst} 在给定的src 和dst 节点之间创建一个类型为type的跟踪对象，如果file 是非空的，那么它将被当作一个Tcl通道处理，并被附加到新创建的跟踪对象上。该程序返回新创建的跟踪对象的句柄。

trace-queue{n1 n2 file} 在节点n1 和n2 之间的链路上安排跟踪。这个函数调用了create-trace，因此同样的规则可以适用于file。

trace-callback{ns command} 当一行被跟踪时安排调用command 命令。程序把command 看作一个字符串并用它评估每个跟踪的行，详细的使用情况请参考~ns/tcl/ex/calolback_demo.tcl

monitor-queue{n1 n2} 这个函数在节点n1 和n2 之间的链路上调用了init-monitor函数。

drop-trace{n1 n2 trace} 给定的trace 对象被传给n1 和n2之间的链路相关联队列的drop-target。

create-trace{} 程序用来创建一个新的trace 对象 ,该对象具有指定的种类并能附带一个Tcl I/O 的通道(通常是文件句柄)。src_和dst_域被相应的C++对象用来产生跟踪输出文件的，因此该跟踪输出中可以包含节点地址，节点地址标识了被跟踪的链路的端点。需要注意的是，它们不是用来匹配的。另外，这些值与包头中的src_及dst_域无关，虽然包头中这些域也在跟踪中被显示出来。参考下面Trace 类的描述（26.3部分）。

trace-queue 函数使得节点n1 和n2 之间链路上的Enque 、Deque和Drop 跟踪有效。链路跟踪程序在下面有讲解（26.2部分）。

monitor-queue 函数功能和trace-queue类似，它通过调用链路的init-monitor 程序，组织对象（SnoopQueue和QueueMonitor对象 ）的创建，这些对象可以轮流地探知时间聚集队列的统计信息。

drop-trace 函数提供了一种不用直接获得队列句柄就可以使用Queue 的drop target的方法。

26.2 类库的支持和示例

上面描述的Simulator 方法需要与Otcl Link 类相关联的trace 和init-monitor 方法的支持。Link有许多子类被定义，其中最常见的是SimpleLink， trace 和init-monitor 方法实际上是SimpleLink而不是它的基类link的一部分。trace 函数被定义如下（在ns-link.tcl中）：

```
#
# Build trace objects for this link and
# update the object linkage
#
SimpleLink instproc trace { ns f } {
    $self instvar enqT_ deqT_ drpT_ queue_ link_ head_ fromNode_ toNode_
    $self instvar drophead_
    set enqT_ [$ns create-trace Enque $f $fromNode_ $toNode_]
    set deqT_ [$ns create-trace Deque $f $fromNode_ $toNode_]
    set drpT_ [$ns create-trace Drop $f $fromNode_ $toNode_]
    $drpT_ target [$drophead_ target]
    $drophead_ target $drpT_
    $queue_ drop-target $drpT_
    $deqT_ target [$queue_ target]
    $queue_ target $deqT_
    if { [$head_ info class] == "networkinterface" } {
        $enqT_ target [$head_ target]
        $head_ target $enqT_
        # puts "head is i/f"
    } else {
        $enqT_ target $head_
        set head_ $enqT_
        # puts "head is not i/f"
    }
}
```

这个函数在仿真$ns 中建立了Enque、Deque 和Drop 跟踪，并输出到I/O 句柄$f。该函数假定一个队列已经和链路相关联。它的操作是首先创建了三个新的跟踪对象，在队列前插入Enque 对象，在队列之后插入Deque 对象，在队列和它的前一个drop target之间插入Drop。注意，所有的跟踪输出都连到同一个I/O 句柄。

这个函数还完成一个额外的任务。它检查看是否一个链路包含了一个网络接口，如果是的话，把它放在链路对象链的第一个对象的位置，否则在第一个的位置插入Enque 对象。

下面的函数init-monitor 和attach-monitor 用来创建一套对象，这些对象用来监视与链路相关联的一个队列的大小。它们定义如下：

```
SimpleLink instproc attach-monitors { insnoop outsnoop dropsnoop qmon } {
    $self instvar queue_ head_ snoopIn_ snoopOut_ snoopDrop_
    $self instvar drophead_ qMonitor_
    set snoopIn_ $insnoop
    set snoopOut_ $outsnoop
    set snoopDrop_ $dropsnoop
    $snoopIn_ target $head_
    set head_ $snoopIn_
```

```
        $snoopOut_ target [$queue_ target]
        $queue_ target $snoopOut_
        $snoopDrop_ target [$drophead_ target]
        $drophead_ target $snoopDrop_
        $snoopIn_ set-monitor $qmon
        $snoopOut_ set-monitor $qmon
        $snoopDrop_ set-monitor $qmon
        set qMonitor_ $qmon
}
#
# Insert objects that allow us to monitor the queue size
# of this link. Return the name of the object that
# can be queried to determine the average queue size.
#
SimpleLink instproc init-monitor { ns qtrace sampleInterval} {
        $self instvar qMonitor_ ns_ qtrace_ sampleInterval_
        set ns_ $ns
        set qtrace_ $qtrace
        set sampleInterval_ $sampleInterval
        set qMonitor_ [new QueueMonitor]
        $self attach-monitors [new SnoopQueue/In] \
        [new SnoopQueue/Out] [new SnoopQueue/Drop] $qMonitor_
        set bytesInt_ [new Integrator]
        $qMonitor_ set-bytes-integrator $bytesInt_
        set pktsInt_ [new Integrator]
        $qMonitor_ set-pkts-integrator $pktsInt_
        return $qMonitor_
}
```

这些函数在仿真ns 中的SimpleLink 对象上建立了队列监视。队列监视通过构造三个SnoopQueue 对象和一个QueueMonitor 对象来执行。SnoopQueue 对象连接在一个某种程度上类似Trace 对象的Queue 周围。SnoopQueue/In(Out)对象监视包的到达(离去)并报告给一个相关的QueueMonitor agent。此外，一个SnoopQueue/out 对象也经常把聚集起来的包丢失的统计信息，传给一个相关的QueueMonitor 对象。对init-monitor 来说，同样的QueueMonitor 对象可以用于所有地方。SnoopQueue 和QueueMonitor 类的C++定义在下面的部分。

26.3 C++的跟踪类

潜在的C++对象被创建以支持在26.3 节有详细说明的接口，该对象也被作为网络元素链接到网络中。单个的C++的Trace 类用来执行Otcl 类Trac/Hop，Trace/Enque，Trace/Deque 和Trace/Drop。Type_域用来区分任何特定Trace 对象可能要执行的各种类型的跟踪。目前，这个域可能包含下面的一种象征性字符：+表示enque，-表示deque，h表示hop，d 表示drop。整个类在~ns/trace.cc 中定义如下：

```
class Trace : public Connector {
    protected:
        int type_;
        nsaddr_t src_;
        nsaddr_t dst_;
        Tcl_Channel channel_;
        int callback_;
        char wrk_[256];
        void format(int tt, int s, int d, Packet* p);
        void annotate(const char* s);
        int show_tcphdr_; // bool flags; backward compat
        public:
        Trace(int type);
        ~Trace();
        int command(int argc, const char*const* argv);
        void recv(Packet* p, Handler*);
        void dump();
        inline char* buffer() { return (wrk_); }
};
```

内部状态Src_和dst_用来标识跟踪输出，它和包头中相应域的名字各自独立。主要的recv()方法定义如下：

```
void Trace::recv(Packet* p, Handler* h)
{
    format(type_, src_, dst_, p);
    dump();
    /* hack: if trace object not attached to anything free packet */
    if (target_ == 0)
        Packet::free(p);
    else
        send(p, h); /* Connector::send() */
}
```

函数仅仅使用源、目的和特殊的跟踪类型字符来格式化一个跟踪的入口。dump 函数把格式化了的跟踪入口写到与channel_相关联的I/O 句柄。format 函数实际规定了跟踪的文件格式。

26.4 跟踪文件格式

Trace::format()方法定义了由Trace 类产生的跟踪文件的格式。为了让早期ns版本（比如ns v1）上的脚本能够继续执行，该函数被设计成与早期仿真器的输出文件保持向后兼容。它实现中的重要的部分如下：

```
// this function should retain some backward-compatibility, so that
// scripts don't break.
void Trace::format(int tt, int s, int d, Packet* p)
```

```
{
    hdr_cmn *th = (hdr_cmn*)p->access(off_cmn_);
    hdr_ip *iph = (hdr_ip*)p->access(off_ip_);
    hdr_tcp *tcph = (hdr_tcp*)p->access(off_tcp_);
    hdr_rtp *rh = (hdr_rtp*)p->access(off_rtp_);
    packet_t t = th->ptype();
    const char* name = packet_info.name(t);
        if (name == 0)
            abort();
    int seqno;
    /* XXX */
    /* CBR's now have seqno's too */
    if (t == PT_RTP || t == PT_CBR)
        seqno = rh->seqno();
    else if (t == PT_TCP || t == PT_ACK)
        seqno = tcph->seqno();
    else
    seqno = -1;
    ...
    if (!show_tcphdr_) {
        sprintf(wrk_, "%c %g %d %d %s %d %s %d %d.%d %d.%d %d %d",
                    tt,
                    Scheduler::instance().clock(),
                    s,
                    d,
                    name,
                    th->size(),
                    flags,
                    iph->flowid() /* was p->class_ */,
                    iph->src() >> 8, iph->src() & 0xff, // XXX
                    iph->dst() >> 8, iph->dst() & 0xff, // XXX
                    seqno,
                    th->uid() /* was p->uid_ */);
    } else {
        sprintf(wrk_,
                    "%c %g %d %d %s %d %s %d %d.%d %d.%d %d %d %d 0x%x %d",
                    tt,
                    Scheduler::instance().clock(),
                    s,
                    d,
                    name,
                    th->size(),
                    flags,
                    iph->flowid() /* was p->class_ */,
```

```
                iph->src() >> 8, iph->src() & 0xff, // XXX
                iph->dst() >> 8, iph->dst() & 0xff, // XXX
                seqno,
                th->uid(), /* was p->uid_ */
                tcph->ackno(),
                tcph->flags(),
                tcph->hlen()
            );
    }
}
```

这个函数不是特别的完美，主要是因为需要维持后向兼容性。它格式化了源、目的和在跟踪对象中定义的type域（不是在包头中），当前时间，然后是一些包头的域，包括包的类型、大小、标记，流的识别，源和目的的包头区域，序列码和唯一的标识符。Show_tcphdr 变量指示跟踪的输出是否应该在每一个输出行的最后追加tcp 头信息。这对使用FullTCP agents 的仿真尤其有用（见34.3部分）。一个跟踪文件的例子（无tcp包头域）可能会是下面这样：

```
+ 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
- 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
r 1.84471 2 1 cbr 210 ------- 1 3.0 1.0 195 600
r 1.84566 2 0 ack 40 ------- 2 3.2 0.1 82 602
+ 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
- 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
r 1.84609 0 2 cbr 210 ------- 0 0.0 3.1 225 610
+ 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
d 1.84609 2 3 cbr 210 ------- 0 0.0 3.1 225 610
- 1.8461 2 3 cbr 210 ------- 0 0.0 3.1 192 511
r 1.84612 3 2 cbr 210 ------- 1 3.0 1.0 196 603
+ 1.84612 2 1 cbr 210 ------- 1 3.0 1.0 196 603
- 1.84612 2 1 cbr 210 ------- 1 3.0 1.0 196 603
+ 1.84625 3 2 cbr 210 ------- 1 3.0 1.0 199 612
```

这里我们看到14 条跟踪数据，5 个enque 操作（第一列标“+”），4 个deque 操作（标“-”），4 个接收事件（标“r”）和一个丢包事件（标”d”）。(这只是一个trace片段，许多包凭空消失了)。每一个事件发生的仿真时间（单位s）列在了第二列。接下来的两个域表明了在哪两个结点之间发生的跟踪。下一个域是一个已知的包类型的名字（见26.5部分）。接下来的域是包的大小，编码在它的IP 的头里。

下一个域包含标识信息，在这里例子里没有使用。标识信息在trace.cc 里的flags[]队列里定义。有四个标志是为ECN 所用的：“E”表示发生的拥塞（CE），“N”表示IP 头里的ECN-Capable-Transport（ECT）指示，“C”表示ECN-Echo，“A”表示TCP 头里的拥塞窗口缩小（CWR）。关于其他的标志，“P”是优先权，“F”表示TCP 的快速启动。

下一个域给出了为Ipv6 定义的IP flow identifier。随后的两个域分别表示了包的源和目的结点的地址。接下来的域表示序列号码（为了与ns v1兼容）。最后的域是一个包的唯一标识符。仿真中创建的每一个新包都会被分配一个新的唯一标识符。

26.5 包类型

每一个包都包含一个包类型域，该域被Trace::format 用来打印遇到的包的类型。类型域是在TraceHeader 类中定义的，被认为是跟踪支持的一个部分；仿真中任何部分都没有对它的解释。包中类型域的初始化由Agent::allocpkt(void)方法来执行。类型域在Agent 构造函数中被传进来的参数设置为整数值（见10.6.3部分）。当前支持的包类型的定义、值以及相关联的象征性名字如下所示：

```
enum packet_t {
        PT_TCP,
        PT_UDP,
        PT_CBR,
        PT_AUDIO,
        PT_VIDEO,
        PT_ACK,
        PT_START,
        PT_STOP,
        PT_PRUNE,
        PT_GRAFT,
        PT_GRAFTACK,
        PT_JOIN,
        PT_ASSERT,
        PT_MESSAGE,
        PT_RTCP,
        PT_RTP,
        PT_RTPROTO_DV,
        PT_CtrMcast_Encap,
        PT_CtrMcast_Decap,
        PT_SRM,
        /* simple signalling messages */
        PT_REQUEST,
        PT_ACCEPT,
        PT_CONFIRM,
        PT_TEARDOWN,
        PT_LIVE,// packet from live network
        PT_REJECT,
        PT_TELNET,// not needed: telnet use TCP
        PT_FTP,
        PT_PARETO,
        PT_EXP,
        PT_INVAL,
        PT_HTTP,
        /* new encapsulator */
        PT_ENCAPSULATED,
```

```
        PT_MFTP,
        /* CMU/Monarch’ s extnsions */
        PT_ARP,
        PT_MAC,
        PT_TORA,
        PT_DSR,
        PT_AODV,
        // insert new packet types here
        PT_NTYPE // This MUST be the LAST one
};
```

The constructor of class p_info glues these constants with their string values:

```
p_info() {
        name_[PT_TCP]= "tcp";
        name_[PT_UDP]= "udp";
        name_[PT_CBR]= "cbr";
        name_[PT_AUDIO]= "audio";
        ...
        name_[PT_NTYPE]= "undefined";
}
```

更多细节请参见12.2.2部分。

26.6 队列监测

队列监测指的是动态跟踪一个队列中的包的能力。队列监测跟踪包的到达/离开/丢包统计信息，还可以有选择的计算这些值的平均值。监测可以适用于所有的包（综合统计），或者每个流的统计（使用一个Flow Monitor）。

许多类被用来支持队列监测。当一个包到达启用了队列监测的链路时，它到达、离开或者丢弃时一般都会通过一个SnoopQueue 对象。该对象包含了一个到QueueMonitor 对象的引用。一个QueueMonitor的定义如下（*~ns*/queue-monitor.cc）：

```
class QueueMonitor : public TclObject {
        public:
                QueueMonitor() : bytesInt_(NULL), pktsInt_(NULL),                                delaySamp_(NULL),
                        size_(0), pkts_(0),
                        parrivals_(0), barrivals_(0),
                        pdepartures_(0), bdepartures_(0),
                        pdrops_(0), bdrops_(0),
                        srcId_(0), dstId_(0), channel_(0) {
                                bind("size_", &size_);
                                bind("pkts_", &pkts_);
                                bind("parrivals_", &parrivals_);
                                bind("barrivals_", &barrivals_);
                                bind("pdepartures_", &pdepartures_);
```

```
                bind("bdepartures_", &bdepartures_);
                bind("pdrops_", &pdrops_);
                bind("bdrops_", &bdrops_);
                bind("off_cmn_", &off_cmn_);
        };
    int size() const { return (size_); }
    int pkts() const { return (pkts_); }
    int parrivals() const { return (parrivals_); }
    int barrivals() const { return (barrivals_); }
    int pdepartures() const { return (pdepartures_); }
    int bdepartures() const { return (bdepartures_); }
    int pdrops() const { return (pdrops_); }
    int bdrops() const { return (bdrops_); }
    void printStats();
    virtual void in(Packet*);
    virtual void out(Packet*);
    virtual void drop(Packet*);
    virtual void edrop(Packet*) { abort(); }; // not here
    virtual int command(int argc, const char*const* argv);
    ...

// packet arrival to a queue
void QueueMonitor::in(Packet* p)
{
    hdr_cmn* hdr = (hdr_cmn*)p->access(off_cmn_);
    double now = Scheduler::instance().clock();
    int pktsz = hdr->size();
    barrivals_ += pktsz;
    parrivals_++;
    size_ += pktsz;
    pkts_++;
    if (bytesInt_)
    bytesInt_->newPoint(now, double(size_));
    if (pktsInt_)
    pktsInt_->newPoint(now, double(pkts_));
    if (delaySamp_)
    hdr->timestamp() = now;
    if (channel_)
    printStats();
}
... in(), out(), drop() are all defined similarly ...
```

除了包和字节的计算之外，一个队列监测可以有选择地引用对象，这些对象使用Integrator 对象保存一个积分队列的大小，其中Intrgrator 对象的定义在25.3 节中。Intrgrator 类提供了一个简单的离散总和积分逼近法的实现。

所有的以p 开头的绑定变量都涉及包的数目，所有的以b 开头的变量都涉及字节数目。变量size_记录了实例队列的大小（字节数），变量pkts_记录了包中的相同的值。当一个QueueMonitor被配置成包含积分函数（对字节或者包）时，它计算在区间[to,now]外的队列大小的近似积分，这里to要么是仿真的开始，要么是潜在Integrator 类的sum_域重置时最后时间。

QueueMonitor 类不是Connector的派生类，也并不直接连接到网络中。然而，SnoopQueue 类（或者它的派生类）的对象是被插入到网络中的，这些对象包含了一个相关队列监测的引用。通常，多个SnoopQueue 对象会指向同一个队列监测。由这些类构建的对象被像上面那样连接到仿真系统中，它们会根据探测队列的特殊类型，调用QueueMonitor out、in 或者drop 程序。

26.7 Per-flow 监测

一些特殊的类被用来完成per-flow的集合统计。这些类包括 :QueueMonitor/ED/Flowmon ,QueueMonitor/ED/Flow和Classifier/Hash。一般地，一个到达的包会被检查属于哪个流。这种检查和流的映射由一个classifier 对象（参见26.7.1部分）完成。一旦确定了正确的流，包就被传给一个负责管理per-flow 状态的flow monitor。Per-flow 状态包含在flow 对象中，该flow对象与流检测器所知道的流保持一一对应。通常，当一个包到达后没有已知的和它匹配的流时，一个流监测器会创建流对象。

26.7.1 流监测

当包到达当前未知的流时，QueueMonitor/ED/Flowmon 类负责管理新的流对象的创建和更新已经存在的流对象。因为它是一个QueueMonitor 的子类，每一个流监测都包含包和字节的到达、离开和丢弃的总体统计。因此，没有必要创建一个独立的队列监测来记录总体的统计信息。QueueMonitor/ED/Flowmon类提供了下面的Otcl 接口：

classifier	取得（设置）classifier把包映射到流
attach	把一个Tcl I/O 通道连接到这个监测器
dump	把流监测内容转储到Tcl 通道
flows	返回这个监测器知道的流对象的名称字符串

classifier 函数可以设置或取得以前分配的对象的名称，这些对象会为流监测完成包到流的映射。通常，以前用的classifier 类型将不得不处理用户持有的“流”的观念。一个基于hash的监测不同IP层包头域的classifier一般用在这里（例如fid,src/dst,src/dst/fid）。注意到，当classifier 接收到包并把它们转发给下游对象时，流监测只用classifier 来完成包映射的功能，因此流监测器只是一个被动的监测器，它不主动地转发包。

attach 和dump 函数用来把Tcl I/O流和流监测关联起来，并按要求转储它的内容。dump 命令所用的文件格式在下面有描述。

flows 函数以一种Tcl 理解的方式来返回流监测所知的流的名称列表。这允许tcl 代码询问一个流监测器，以获得它维护的每个流的句柄。

26.7.2 流监测的跟踪格式

流监测定义了一种跟踪格式，该格式可以由后期处理脚本使用以确定per-flow 基础上各种变量的计数。该格式定义在~ns/flowmon.cc 中见下面的代码：

```
void
FlowMon::fformat(Flow* f)
{
	double now = Scheduler::instance().clock();
	sprintf(wrk_, "%8.3f %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d",
			now,
			f->flowid(), // flowid
			0, // category
			f->ptype(), // type (from common header)
			f->flowid(), // flowid (formerly class)
			f->src(),
			f->dst(),
			f->parrivals(), // arrivals this flow (pkts)
			f->barrivals(), // arrivals this flow (bytes)
			f->epdrops(), // early drops this flow (pkts)
			f->ebdrops(), // early drops this flow (bytes)
			parrivals(), // all arrivals (pkts)
			barrivals(), // all arrivals (bytes)
			epdrops(), // total early drops (pkts)
			ebdrops(), // total early drops (bytes)
			pdrops(), // total drops (pkts)
			bdrops(), // total drops (bytes)
			f->pdrops(), // drops this flow (pkts) [includes edrops]
			f->bdrops() // drops this flow (bytes) [includes edrops]
		);
};
```

大部分域在代码注释中都有解释。“category”是历史遗留的，但它被用来维持和ns v1 中流管理格式松散的后向兼容性。

26.7.3 流的类

QueueMonitor/ED/Flow 被流监测器用来保管per-flow 数目。作为QueueMonitor 的一个子类，它继承了包和字节中为到达、离开、丢弃而设置的标准计数器。此外，每个流一般由包的源、目的和流标识域等这些混合因素来唯一的确定，所以QueueMonitor/ED/Flow对象都包含这些域。它的Otcl接口仅包含绑定的变量：

src_ 到达该流的包的源地址
dst_ 到达该流的包的目的地址
flowid_ 到达该流的包的流id

注意到，包可以用非src/dst/flowid 三要素的标准被映射(使用classifiers)到流上。在这种情况下，只有那些实际被classifier用来执行包和流的映射的域，才被认为是可靠的。

26.8 命令一览

下面是在仿真脚本中经常用到的跟踪相关的命令列表：

$ns_ trace-all <tracefile>
这是一个用来在ns 中建立跟踪的命令。所有的跟踪记录都写入到<tracefile>中。

$ns_ namtrace-all <namtracefile>
这个命令在ns 里建立起一个nam 跟踪，所有的nam 跟踪记录都写入到<namtracefile>里。

$ns_ namtrace-all-wireless <namtracefile> <X> <Y>
这个命令建立了无线的nam 跟踪。<X><Y>是无线系统中的x-y 的坐标，所有的无线nam 跟踪记录被写到<namtracefile>中。

$ns_ nam-end-wireless <stoptime>
这个命令告诉nam仿真结束的时间在<stoptime>中给定。

$ns_ trace-all-satlinks <tracefile>
这个方法用来跟踪卫星链路，并把跟踪写到<tracefile>里

$ns_ flush-trace
这个命令清空跟踪的缓冲区，通常在仿真运行结束时调用这个命令。

$ns_ get-nam-traceall
返回存储在Simulator 实例变量namtraceAllFile_中的namtrace 文件描述符。

$ns_ get-ns-traceall
和上一个命令类似。它返回ns 跟踪文件的文件描述符，这个跟踪文件存储在Simulator 的实例变量traceAllFile_中。

$ns_ create-trace <type> <file> <src> <dst> <optional:op>
这个命令在结点<src><dst>之间创建了一个类型为<type>的跟踪对象.跟踪记录被写入<file>中。<op>是一个可以用来详细说明跟踪类型的参数,比如nam。如果<op>没有定义的话，那么默认的跟踪对象将创建成nstraces。

$ns_ trace-queue <n1> <n2> <optional:file>
这是一个create-trace 的封装方法，这个命令在结点<n1><n2>之间的链路上创建了用来跟踪事件的跟踪对象。

$ns_ namtrace-queue <n1> <n2> <optional:file>
这是用来在结点<n1>和<n2>之间的链路上给namtracing 创建跟踪对象的，这个方法和trace-queue 类似，是对应于namtrace的部分。

$ns_ drop-trace <n1> <n2> <trace>
这个命令使给定的<trace>对象成为与<n1><n2>之间的链路相关联的队列的丢弃对象

$ns_ monitor-queue <n1> <n2> <qtrace> <optional:sampleeinterval>
这条命令建立了一个监测器，它跟踪在结点<n1>和<n2>之间的链路上的队列的平均队列长度。Sampleeinterval 的默认值是0.1

$link trace-dynamics <ns> <fileID>
跟踪这个链路的动态情况，把输出写入到fileID 的文件句柄。ns 是一个Simulator 的实例，或是一个被创建用来调用仿真的Multisim 对象。

跟踪文件的格式和ns v1 中的输出文件格式保持后向兼容性 ,因此ns-1处理脚本仍可以被使用。对带有Enque、Deque、receive和Drop 跟踪的链路对象的跟踪记录，一般有下面的格式：
<code> <time> <hsrc> <hdst> <packet>
这里：
<code> := [hd+-r] h=跳 d=丢弃 +=入队 -=出队 r=接收
<time> := 模拟时间，单位秒
<hsrc> := 跳 / 队列链路的第一个节点地址
<hdst> := 跳 / 队列链路的第二个节点地址
<packet> := <type> <size> <flags> <flowID> <src.sport> <dst.dport> <seq>
<pktID>
<type> := tcp|telnet|cbr|ack etc.
<size> := 包大小，单位字节
<flags> := [CP] C=拥塞, P=优先级
<flowID> := 为IPv6定义的流标识符域
<src.sport> := 传输地址 (src=node,sport=agent)
<dst.sport> := 传输地址 (dst=node,dport=agent)
<seq> := 包序列号
<pktID> := 每个新包的唯一标识符

只有那些对排序感兴趣的agent 才会产生序列号，因此这个域（seq）对一些agents 产生的包无用。对于应用RED 网关的链路，有如下附加的跟踪记录：
<code><time><value>
这里：
<code> := [Qap] Q=队列大小, a=平均队列大小, p=丢包概率
<time> := 模拟时间，单位秒
<value> := 值

动态链路的跟踪记录有如下格式：
<code><time><state><src><dst>
这里：
<code> := [v]
<time> := 模拟时间，单位秒

<state> := [link-up | link-down]
<src> := 链路第一个节点地址
<dst> := 链路第二个节点地址

第 27 章

对Test Suite 的支持

在 ~ns/tcl/test 目录下包括许多 ns 的测试套件，这些测试套件被验证程序（~*ns*/validate, validatewired,validate-wireless, and validate.win32）用来检查ns 是否正确安装了。如果你向ns 中修改或者添加了新的模型，你最好运行验证程序，以保证你做的改变没有影响ns 的其他部分。

27.1 Test Suite 组件

~ns/tcl/test 下的每一个test suite 都是用来验证ns 中特定模块的正确性的，它有三个组件：

- 一个shell 脚本（test-all-xxx）来启动test
- 一个ns tcl 脚本(test-suite-xxx.tcl)来运行已经定义的测试
- ~ns/tcl/test 的一个子目录(test-output-xxx)，它包含了由test suite 产生的正确的跟踪文件。这些文件用来验证test suite 是否在你的ns 上正确运行。

（说明：xxx代表test suite的名称）

27.2 编写一个test suite

当你要编写自己的test suite 时，你可以从~ns/tcl/test 中取出一个当作模板，例如无线局域网的test suite (test-all-wireless-lan, test-suite-wireless-lan.tcl, and test-output-wireless-lan)。

要编写一个test suite，首先需要写shell 脚本(test-all-xxx)。在shell 脚本中，你必须指明将被测试的模块，ns tcl 脚本的名字和输出子目录。你可以在静音模式下运行这个shell 脚本。下面是一个例子（test-all-wireless-lan）：

```
# To run in quiet mode: "./test-all-wireless-lan quiet".
f="wireless-lan" # 指明要测试的模块名称
file="test-suite-$f.tcl" # ns脚本名称
directory="test-output-$f" # 盛放输出结果的子目录
version="v2" # 指明ns版本
# Pass the arguments to test-all-template1, which will run through
# all the test cases defined in test-suite-wireless-lan.tcl.
./test-all-template1 $file $directory $version $@
```

你也可以通过给每个不同的测试定义一个TestSuite 的子类的方法，在ns 的脚本中(test-suite-xxx.tcl)创建几个测试用例。例如在test-suite-wireless-lan.tcl中，每个测试用例使用不同的Ad Hoc 路由协议，它们的定义如下：

```
Class TestSuite
```

```
# wireless model using destination sequence distance vector
Class Test/dsdv -superclass TestSuite
# wireless model using dynamic source routing
Class Test/dsr -superclass TestSuite
… …
```

每个测试用例基本上就是一个仿真场景。在父类TestSuite 中，你可以定义一些函数像init 和finish ，去做每个测试用例需要做的工作，比如设置网络拓扑和ns 跟踪。测试的具体配置定义在相应的子类中。每个子类也有一个运行函数来启动仿真：

```
TestSuite instproc init {} {
        global opt tracefd topo chan prop
        global node_ god_
        $self instvar ns_ testName_
        set ns_ [new Simulator]
        … …
}
TestSuite instproc finish {} {
        $self instvar ns_
        global quiet
        $ns_ flush-trace
        puts "finishing.."
        exit 0
}
Test/dsdv instproc init {} {
        global opt node_ god_
        $self instvar ns_ testName_
        set testName_ dsdv
        … …
        $self next
        … …
        $ns_ at $opt(stop).1 "$self finish"
}
Test/dsdv instproc run {} {
        $self instvar ns_
        puts "Starting Simulation..."
        $ns_ run
}
```

ns 脚本中所有的测试都是从函数runset 开始的：

```
proc runtest {arg} {
        global quiet
        set quiet 0
```

```
    set b [llength $arg]
    if {$b == 1} {
        set test $arg
    } elseif {$b == 2} {
        set test [lindex $arg 0]
        if {[lindex $arg 1] == "QUIET"} {
            set quiet 1
        }
    } else {
        usage
    }
    set t [new Test/$test]
    $t run
}
global argv arg0
runtest $argv
```

当你运行测试的时候，跟踪文件就生成了，并被保存到输出子目录里。把这些跟踪文件和那些test suite 中保留的正确跟踪相比较，如果有不同，则表示测试失败。

第 28 章

ns代码风格

我们推荐下面的ns编码指导风格。

28.1 缩进风格

- 我们推荐使用BSD内核正常形式（KNF）编码风格，参见：http://cvsweb.netbsd.org/bsdweb.cgi/sharesrc/share/misc/style?rev=HEAD&content-type=text/x-cvsweb-markup
- 虽然 KNF 是针对 C 语言的 ,但它应用于 C++和 Tcl 的效果也不错。大部分 ns 已经使用 KNF ,KNF 也被广泛的用于 BSD 和 Linux 内核。
- 高位缩进 8 位。使用 8 缩进避免了与一个“tab”字符冲突，不过它的坏处是加宽了嵌套循环结构，使之很难在 80 列内结束。（有些人认为这是一个特性:-）

28.2 变量命名约定

- 类的实例变量都应该以_结束，这有利于分清，实例变量是全局还是局部变量。
- C++和 Tcl 绑定变量应该具有相同的名字，这有利于快速辨认出绑定变量，减少了复杂性。

28.3 其他约定

- 避免使用C++模板。NS支持多平台，模板不是很轻便，而且常常很难调试。虽然有些导进来的代码违反了这条约定，但那只是一个例外，ns核心代码应该在没有模板的情况下调试和运行。
- 对于NsObjects，使用debug_实例变量来激活调试函数。这避免了调试代码的重复定义,并允许在不进行重新编译的情况下调试某一Nsobject。
  例如：
  为了激活队列对象中的调试，请把下面的代码加入你的tcl脚本：
  Queue set debug_ true
  调试代码可以像下面这样被插入到Queue的子类中：
  debug("This is a debug statement %d",variable_to_debug);
  所有的调试信息都会输出到stdout（屏幕）。
- 导致程序退出的错误信息会被输出到stderr，其他的错误信息被输出到stdout。

第四篇

路由

第 29 章

单播路由

本章节介绍 ns 中单播路由的结构。首先，通过 Simulator 类和 RouteLogic 类的方法，来介绍面向用户的接口（29.1 节）。其次，我们将介绍特定路由（29.2 节）的配置机制，比如非对称路由或等代价路由的配置机制。稍后部分介绍单个路由策略和协议（29.3 节）的配置机制。最后，总结给出一个 ns 路由整体的内部结构框架（29.4 节）。

本章节介绍的过程和函数可以在：~ns/tcl/lib/ns-route.tcl,~ns/tcl/rtglib/route-proto.tcl, ~ns/tcl/mcast/McastProto.tcl,和~ns/rtProtoDV.{CC,H}中找到。

29.1 模拟管理的接口（The API）

用户级的模拟脚本需要一个命令：该命令用来指定模拟单播路由的策略或协议。路由策略是 ns 用来计算模拟路由的一种通用机制。在 ns 中有四种路由策略：即静态路由策略、Session 路由策略、动态路由策略和手工路由策略。相对而言，路由协议则是一个特定算法的实现。目前，静态路由策略和 Session 路由策略使用的是 Dijkstra 的 all-pairs SPF 算法[]；而动态路由策略使用是：分布式的 Bellman-Ford 算法[]；在 ns 中，我们将淡化静态和 session 路由策略和协议二者之间的区别，仅将它们简化看作协议。

Rtproto{}是 Simulator 类的实例化过程，用来指定模拟过程中所应用到的单播路由协议。它带有若干参数，第一种是强制性的，该参数标识所使用路由的协议，后面的参数则是指定运行这个协议实例的节点。第二种是默认的：即在一个拓扑结构中，所有的节点都运行同一个路由协议。举个例子，下面的程序说明了 rtptoto{}命令的用法。

```
$ns rtproto Static                    ;# Enable static route strategy for the simulation
$ns rtproto Session                   ;# Enable session routing for this simulation
$ns rtproto DV $n1 $n2 $n3            ;# Run DV agents on nodes $n1,$n2,and $n3
$ns rtproto LS $n1 $n2                ;# Run link state routing on specified nodes
```

如果一个模拟脚本没有指定任何一个 rtproto{}命令，那么，在同一个拓扑结构中，ns 将在所有节点上运行静态路由策略。

在一个模拟脚本中，针对同一个或不同的路由协议可能会出现多种 rtproto{}命令行，然而，一种模拟不可能同时使用诸如静态或 session 路由的集中式控制路由机制，又同时使用诸如 DV 的分层动态路由协议。

在静态路由策略中，每一个节点可能运行多种路由协议。这种情况下，每种路由协议都可能有到达同一目标节点的一条路径，因此，给每一种协议的路由分配了不同的优先值。这些值是在 0 - 255 范围内的非负整数，优先值越低，该路径具有优先级越大。当每个路由 agent 都能够发现有到达同一目标节点的路由时，那么优先级别越高的路径就会被选中，并加入到节点的转发路由表中。如果多个 agent 同时拥有最高优先级别的路由时，那些含有最低代价的路由将会被选中。我们将来自最高优先级别协议中的那些最低代价路由称为“候选”路由，如果存在来自同一个或不同协议的多个候选路由，那么，就目前来讲，将会随机的选择 agent 中的一个路由。

优先级分配和控制 每种协议的 agent 中都存放着路由优先级的一个列表 rtpref_。一个目标节点对应着一个按该节点句柄进行索引的元素。默认的优先级值有类变量 preference_来决定。目前的默认值是：

```
Agent/rtproto set preference_200                ;#global default preference
Agent/rtproto/Direct set preference_100
Agent/rtproto/DV set    preference_120
```

满足以下三个方面之一时，一个模拟脚本可以通过改变路径的优先级值来控制路由：改变通过特殊协议 agent 获得的专有路由优先级；改变由 agent 获得所有路由的优先级；在 agent 创建之前，改变 agent 的类变量。

链路代价的分配和控制 在目前应用的路由协议中，从一个节点到达目标节点的路由长度也就是从这个节点到目标节点的代价。 有可能在每一条链路中改变其某个链路的总代价。

实例进程 cost{}作为$ns cost <node1> <node2> <cost>的调用，并且设置从<node1> 到<node2> 到<cost>链路的代价。

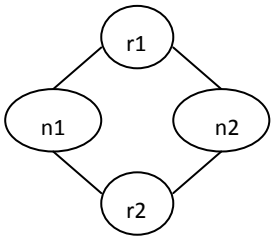
```
$ns   cost   $n1   $ns2   10                ;# set cost of link from $n1 to $n2 to 10
$ns   cost   $n2   $ns1   5                 ;# set cost of link in reverse direction to 5
[$ns   link   $n1   $ns2]   cost ?          ;# query cost of link from $n1 to $n2
[$ns   link   $n2   $ns1]   cost ?          ;# query cost of link in reverse dierction
```

注意：该程序只沿一个方向设置代价。同时，程序 cost?{}返回的是遍历特定无方向链路的代价，默认的链路代价为 1。

29.2 特殊路由的其他配置机制

调整优先级和代价机制可以完成两种特殊类型的路由配置机制：非对称路由和多径路由。

非对称路由 当节点 n1 到节点 n2 的路径不同于 n2 到 n1 的路径时，就产生了非对称路由。下面是一个简单的拓扑结构和能到达这种效果的代价配置：



Nodes n1 and n2 use different paths to reach each other. All other pairs of nodes use symmetric paths to reach each other

```
$ns cost $n1 $r1 2
$ns cost $n2 $r2 2
$ns cost $r1 $n2 3
```

如果合理配置链路的代价，那么任何一条使用链路代价作为单位的路由协议都可认为是非对称路由。

多径路由 每一个节点可以独立的进行配置，使用多条分段路径到达特定的目标节点。实例变量 multiPath_决定一个节点是否使用多条路径来达到目标节点。每个节点负责初始化同名类变量的实例变量。如果存在到达目标节点的多条候选路径，那么所有路径都可通过同一协议互相获悉。然后，节点可能同时利用所有不同的路径到达目标节点，典型配置如下所示：

```
Node set multipath_1          ;# All new nodes in the simulation use multiPaths where applicable or alternately
set n1  [$ns  Node ]          ;# only enable $n1 to use multiPaths where applicable
$n1 set multipath_1
```

就目前来讲，只有 DV 路由能够产生多径路由。

29.3 协议专有的配置参数

静态路由 静态路由计算策略是 ns 默认路由计算机制。该策略使用 Dijstra 的 all_pairs SPF 算法[]。路由计算算法在模拟开始之前已被严格的运行。在拓扑结构中，使用邻接矩阵和所有链路的链路代价来计算路径。

(注意：静态路由中的节点在一定程度上是静态的，即路由的计算先于模拟的开始。不同于 session 和 DV 路由允许在模拟中间更改路由。不需要计算但要有用户来设置路径的地方，手工路由就是针对静态路由的一种选择。)

session 路由 在一个模拟过程中，静态路由策略仅为拓扑结构计算路径时，才被提前应用。然而在模拟进行中，如果以上的静态路由正在被使用和拓扑结构的更改，那么在短时间内，一些源节点和目的节点可能暂时互相不可达。

Session 路由策略几乎与静态路由一样，在模拟开始之前运行，session 路由运行 Dijkstra all_pairs SPF 算法，使用邻接矩阵和拓扑结构链路代价来计算路由。不过，在模拟过程中，拓扑结构发生变化时，session 路由也会应用相同的算法进

行重新计算。换句话说，路由及时地进行了再计算和再恢复，并且像在静态路由一样不存在短暂的路由损耗。

在动态拓扑结构中，session 路由提供全面和适时的路由变化，如果拓扑结构总能够连接，那么在模拟过程中端到端连接就一直存在。然而，用户应该注意到，session 路由的瞬时路由的重新计算并不能阻止短暂的违背因果关系的现象，如当拓扑结构急剧变化时，包的重排。

DV 路由 DV 路由是 ns 中分布式 Bellman-Ford 路由的一个实现。这种实现每隔一个广播间隙就周期性的发送一个路由更新信息，这个变量是 Agent/rtProto/DV 中的一个变量，它的默认值是 2 秒。

除了周期更新之外，每个 agent 也发送触发更新；当节点的转发表改变时，会引起触发更新。同样由于拓扑结构的改变，或在一个节点的 agent 收到了一个路由更新，并重新计算和设置了新的路径时，也会导致触发更新的产生。

每个 agent 都用一个带有标记反向分割线机制向它的临近的路由进行广播。“分割线”是一种机制，agent 根据该机制，不向到正被使用到达目标节点接口以外的节点广播到达目标节点的路由。在“带有标记反向分割线”机制中，agent 将会以无限单位量向接口以外进行路由广播。

每个 DV agent 使用的默认值为 120 的 preference_of ，这个值由同名的类变量来决定。

每个 agent 都是用类变量 INFINITY（设为 32）来决定一条路由的有效性。

手工路由 手工路由不是一种路由计算协议（如前所述），但对用户来讲是一种简单方式，用户可以手工配置路由表，就像在一个工作站上进行路由操作一样。

可以与 rtproto 配合使用手工路由，然后，用 add-route-to-adj-node 命令来设置每一个节点的路由表。例如：

```
$ns rtproto Manual
set n1 [$ns node]
set n2 [$ns node]
$ns duplex-link $n1 $n2 10Mb 100ms DropTail
$n1 add-route-to-adj-node -default $n2
$n2 add-route-to-adj-node -default $n1
```

如果想看一个更完整的例子，请看 tcl/ex/many_tcp.tcl

29.4 路由的实质和体系结构

我们将开始讨论相关单播的路由类，以及用来配置和执行每一个不同路由协议的代码路径。稍后我们介绍单播路由和动态网络之间、单播路由和多播路由之间的接口。

29.4.1 类

主要有四种类：RouteLogic 类，rtObjict 类 rtPeer 类和所有协议的基类 Agent/rtProto 类，路由体系结构还涉及到 Simulator， Link Node 和 Classifier 类。

RouteLogic 类 该类定义了两个方法来配置单播路由，一个方法用来查询路由信息。当拓扑结构动态变化时，它也定义了一个可应用的实例化进程。我们将讨论面向动态网络接口相关的关键过程。

- 实例化过程 register{}被 Simulator::rtproto{}所调用。该实例化过程获得协议和节点列表，并创建一个实例化变量队列 rtprotos_；这个队列的索引恰好是协议的名字，索引值就是运行在该协议下节点的序列。
- Configure{}读取 rtprotos_实例变量，并为队列每一个元素调用路由协议方法，进行恰当的初始化。同样它也能被模拟运行过程所调用。

对 rtproto_队列中，每一个被索引的协议<rt-proto>来讲，该路由调用 Agent/rtProto/<rt-photo> init-all rtproto_(<rt-proto>)。

如果在 roproto_中没有元素存在，则该路由调用静态路由，Agent/rtProto/Staic init-all。

- 实例化过程 lookup{}带有两个节点号： $nodeId_1$ 和 $nodeId_2$ ，它返回 $nodeId_1$ 到达 $nodeId_2$ 邻居节点的 id。

该过程常用在每个节点上，面向用于查询计算路由和组装路由的静态路由计算过程中，同样它也应用在进行恰当的 RPF 监测的多播路由中。

注意：该过程重载了一个同名的 instproc-like。如果恰当的实体对象 rtObject 存在，就访问它们；另为该过程也调用 instproc-like 来获得其他相关信息。

rtObject 类 应用在动态路由的模拟中。每一个节点都有一个和它相联系的 rtObject 类，它作为一个节点内运行不同的路由协议的等同物。在任何一个节点上，节点上的 rtObject 会跟踪每一个在节点上运行的协议；通过每一个协议，它计算并设置可达的目标节点的下一条路由。如果路由表发生改变或拓扑结构发生变化，rtObject 将会更改协议以采取恰当的行为。

该类定义了 init-all{}过程；这个过程获得众多节点的一个列表，并且在每一个描述列表中创建节点的 rtObject。随之，调用它的 compute-routes。

如果每一个新对象的构造函数直接初始化每个节点的路由协议，而路由协议负责计算最近邻居节点的路由。则当 init-all{}运行 compute-routes{}时，那些直接路由将恰当设置其路由对象。

该类中其他实例化过程：

- init{} 构造函数在它的实例变量 node_中设置由它本身指向节点的指针，通过节点实例过程 init-routing{}和节点实例变量 rtObject_，设置由节点指向构造函数本身的指针。然后，初始化 nextHop_、repref_、metric_、rtVia_的队列，每一个队列的索引是目标节点的句柄。

- nextHop_包含了到达特定目标节点的链路；rtpref_和 metric_是节点路由建立的优先值和单位；rtVia_是设置在节点上路由 agent 的名字。

构造函数同样也会创建直接路由协议的实例，并为它调用 compute-routes{}。

- add-proto{}创建协议的一个实例，保存指向它的协议队列的引用 rtProtos_。队列的索引是该协议的名字，它捆绑在节点的协议对象上，并返回协议对象的句柄。

- lookup{}具有目标节点的句柄，返回用于到达目标节点的邻居节点的 id。如果使用多条路径，那么它将返回使用邻居节点的一个列表。

如果节点没有到达目标节点的路径，则该过程将返回-1。

- compute-route{}是该类的核心过程。它首先检测节点任意路由协议是否计算过任何新的路径。如果计算过，它将决定从所有协议中选择到达目标节点的最佳路由。如果路径已经改变，该过程会把大量的变化通知给每个协议，确保使任何协议能够发送最近的更新。最后，它也将把已经计算完成的新单播路由表通知给所有的多播协议。

路由策略会检查协议 agent 的实例变量 rtsChanged_，看该协议自最近一次检查后，是否所有的路由已经发生改变。然后 ,应用协议的实例化变量队列 nextHop、rtpref_和 metric_来计算自己的队列。rtObject 将会设置或修改发现变化的路由。

如果节点上的所有路由都发生了改变，则 rtObject 将调用带有变化量字段的 send-updates{}的协议 agent 实例过程。之后，如多播路由对象存在，则进行调用。

下面部分的路由过程用来查询 rtObject，获得变量状态的信息。

- dump-routes{}带有一个输出文件描述符的参数，并从节点的路由表读出文件的描述符。

典型的 dump 输出是：

- rtProto?{} 带有一个路由协议的参数，并返回运行在节点上协议的实例句柄。
- nextHop?{} 带有目标节点的句柄，并返回用于到达目标节点的链路。

同样，rtpref?{}和 metric?{}带有一个目标节点句柄的参数，并返回在节点设置到达目标节点路由的优先级和单位。

rtPeer 类； 是协议 agents 应用的一个容器类，每个对象保存 peer agent 的地址、单位和由 peer 所广播路径的优先级。协议 agent 将保存对象的每一个 peer。该类维护实例化变量 addr_和实例化变量队列 metric_和 rtpref_；队列中的元素是目标节点的句柄。

该类的实例化过程 metric{}和 preference{}获得目标节点和值 ,并设置各自的队列变量。过程 metric?{}和 preference?{}获得目标节点并返回目标节点的当前单位和优先值。实例 addr?{}返回 peer agent 的地址。

Agent/rtProto 类 该类是所有路由协议 agents 继承的基类。每种协议的 agent 必须定义 init-all{}过程来对整个协议进行初始化，并有可能定义实例过程 init{}、compute-routes{}和 send-updates{}。另为，如果拓扑结构是动态的，并且协议支持路由计算以适应拓扑结构的变化，那么协议就应该定义 compute-all{}过程，也有可能定义 intf-changed{}实例化过

程。在本章节里，我们暂时先介绍基类的接口。然后介绍在动态网络中的 compute-all{}和 intf-changed{}。本章结尾，我们将详细介绍每种协议的其他部分。

init-all{}过程是类的一个全局初始化过程。有可能给出节点（节点应该是运行在该路由协议下节点）的一个列表参数。然而，诸如 static 和 session 等集中式路由协议将忽略这个参数；分散的路由如 DV 将使用这个参数列表，在每个特殊的节点上初始化协议的 agent。

注意，派生类 OTcl 并不继承基类中定义的过程，因此，每一个派生路由协议类必须明确地定义它自己的过程。

实例化过程 init{}是创建协议 agent 的构造函数。在这个类中，基类的构造函数初始化对象的优先级，并确定节点的接口事件和接口事件的当前状态。邻居节点句柄索引该接口 ,并保存到实例化变量队列 ifs_中,而相应的状态实例化队列是 ifstat_。

诸如 static 和 session 的集中式路由不能为每个节点创建单独的 agent ,因此 ,不要访问这些实例化过程任何一个过程。

实例化过程 compuet-routes{}为协议计算实际的路由。计算是基于协议已获悉的路径和协议之间的变化。

只要拓扑结构变化，rtObject 将调用该过程。同样，当节点接收到协议更新消息时，也会调用该过程。

该过程计算新的路径时，需要调用 rtObject::compute-routes{}进行重算，并有可能在节点上设置新的路径，首次调用该路由的过程将实时调用 rtObject。

实例化过程 send-updates{} 只要节点的路由表发生改变和新的更新消息发送给所有协议时 ,该过程就会被 rtObject 所调用。rtObject 传输已发生的变化量参数。也可能是路径没有发生变化，但接口事件的状态发生变化时，该过程也可能被调用。变化的数值用于决定必须发送路由更新消息的 peers 列表。

稍候介绍其他涉及到拓扑结构变化的相关过程。

Simulator、Node、Link 和 Classifier 的其他扩展部分

在前面(29.1 节)我们已经讨论了类 Simulator 的 rtproto{}和 cost{}方法。另外一个方法 get-routelogic{}在内部使用，该方法在模拟中返回路由的逻辑实例。通常，类 Simulator、单播路由和多播路由使用该方法。

Node 类包含额外的实例化过程来支持动态单播路由 如 init-routing{}、add-routes{}、delete-routes{}和 rtObject{}。

实例化过程 init-routing{}在节点上被 rtObject 调用。为了稍后的修改和恢复，在它的实例化变量 rtObject 中，保存了一个指向 rtObject 指针。它同样会检测它的类变量，看它是否应该使用多径路由和实现实例化变量的设置。如果能够使用多径路由，则实例化队列 routes_保存了一个为每个目标节点设置的路径数的计算器。在单播路由中，这是一个唯一不是被节点的句柄，而是被节点 id 所索引的队列。

实例化进程 rtObject?{}返回节点的 rtObject 句柄。

实例化进程 add-routes{}获得定点的 id 和链路列表。它向链路列表里添加那些到达由节点 id 识别目标节点的路径。应用单独的 Classifier/multiPath 来实现 multiPath 的路由。对任意给定 id 的目标节点 d，如果该节点有多条链路到 d，那么主分类器将指向该多径分类器，而不是指向到达目标节点的链路。使用的接口在多径分类器中进行设置，并标记每一个多径

路由。为了成功的转发包到分类器，多径分类器将设置在其内部的每一个链路。

实例化进程 delete-routes{}带有一个节点的 id，一个接口的列表和一个 nullAgent。从已设置的接口列表中删除每一个接口。如果实体先前没有使用多径分类器，那么它必须仅有一条路径，并且该路径实体被设置指向特定的 nullAgent。

问题：为什么在多径分类器里实体趋于零时，它不指向 nullAgent?

单播路由中 ,对 Link 类主要的扩展就是支持链路代价的概念。实例化变量 cost_包含无方向链路的代价。实例化过程 cost{}和 cost?{}设置和获取链路上的代价。

注意，cost{}带有代价参数。优先使用模拟方法来设置代价变量，类似于用模拟实例过程来设置链路上的队列和延迟。

Classifier 类包含三个新的过程，两个重载现有的 instproc-like，另一个提供新的功能。

实例化过程 install{}重载了现有的同名的 instproc-like。过程保存了在实例变量队列 elements_中正在设置的实体,然后调用 instproc-like。

实例化过程 installNext{}也同样重载了现有的同名的 instproc-like。该 instproc-like 简单的设置进入下一可得间隙的实体。

实例化过程 adjacents{}返回所有设置在 classifier 中元素的<key，value>序列对的一个列表。

29.4.2 动态网络和多径的接口

本章节介绍单播路由中响应拓扑结构变化的方法。在其他章节里介绍造成拓扑结构变化的一系列行为以及所激发的相应响应。拓扑结构变化的响应分为两类：在每一个节点由独自的 agent 引发的响应和由整个拓扑结构全局引发的响应。

诸如 DV 分层路由协议的执行需要有不同节点各自的协议 agents 来激发响应。而如 static 和 session 集中式路由协议则专属于后一种。分层路由协议能够应用一些技术来获得与路由操作相关的统计信息。然而，在目前 ns 中，这样的代码并没有被实现。

单个节点的行为 任何拓扑结构发生变化后，网络动态模型将首次在每一个影响到的节点上调用 rtobject::intf-changed{}，对运行在受影响节点上的每一个单播路由协议来讲，rtobject::intf-changed{}将调用各个独自协议的实例化过程，先调用协议的 compute-routes{}，随后调用过程 intf-changed{}。

各种协议完成独自路径的计算后，rtobject::intf-changed{}调用 compute-routes{}有可能设置新的路径。如果在节点上设置了新的路径，rtobject-changed{}将会为运行在节点上每一种路由协议调用 send-updates{}，该过程也同样将标记节点路由变化的多径路由对象，并指出已经执行过的变化数量。rtobject::flag-multicast{}将依次通知多径路由对象采取恰当的行为。

单播路由和多播路由之间接口的一种特例是：在动态密集模型多径传输和分层路由之间的相互作用。在 ns 中，动态密集模型的执行的前提假设是：当路径变化时，邻居节点将发送一个一个特定更新消息，而实时并没有发送更新消息。然后，利

用特有的信息为多路径生成树计算恰当的父子关系。因此，当分层路由收到路由更新消息，甚至是更新消息在自己的路由表中并没有引起任何变化时，也将调用 rtobject_flag-multicast。

全局行为 受到影响的节点一旦完成其全部行为。动态网络模型将向拓扑结构通知变化的 RouteLogic 实例 RouteLogic::notify{}。该过程将为曾经设置在每个节点上的任何协议调用 compute-all{}过程。像 session 一样的集中式路由协议应用该信号重新计算拓扑结构中的路径。最后，RouteLogic::notify{}过程将通知运行在节点上任何集中式多径路由的实例。

29.5 内部协议

本节我们将介绍各个路由协议 agent 的剩余其余细节。注意 :仅在本节对内部路由协议 agent 进行了讨论 ,即直接路由。

直接路由 该协议跟踪事变链路的状态，仅维持到最近邻居节点的路径。像其他路由协议一样，它维持 nextHop_，rtpref_和 metric_的实例变量队列，在拓扑结构中被可能的每一个目标节点的句柄索引。

实例化过程 compute-routes{}基于当前的链路状态和先前已知的事变链路状态来计算路径。

该协议没有定义其他过程或实例化过程。

Static 路由 类 RouteLogic 中 compute-routes{}过程首先会创建邻接矩阵，然后，调用映像对象的 C++方法 compute_routes()，最后，该过程重新获得路由计算的结果，并且为拓扑结构的各个节点插入恰当的路径。

该类仅定义了调用 compute-routes{}的过程 init-all{}。

Session 路由 类定义了 init-all{}过程，用来计算模拟开始的路径，同样当拓扑发生变化时，它也定义了 compute-all{}过程来计算路径。这两个过程直接调用 compute-router{}。

DV 路由 动态路由策略中，在信息互相交换的基础上，进行节点发送和接收信息，以及计算拓扑结构的路径。过程 init-all{}带有节点列表的参数；默认拓扑结构中节点的列表。在参数中每一个节点上，该过程启动类 rtobject 和类 Agent/rtproto/DV agents。然后，为每一个最新创建的 DV agents 决定 DV peers，并创建相关的 rtpeer 对象。

DV agent 构造函数初始化许多实例化变量，每个 agent 保存一个由目标节点句柄索引的队列、优先级和单位、下一跳的接口以及在接口上远方的 peer 事件，以便 agent 计算出到达每个目标节点的最优路径。Agent 创建完这些实例化变量，然后，确定时间在模拟开始的 0.5 秒内发送首次更新。

每个 agent 保存有 peer 节点句柄索引的它的 peers 列表。每个 peer 都是一个单独的结构，拥有 peer agent 的地址以及由 peer 广播的到达目标节点路由的优先级和单位。在稍后讨论路由结构时 ,我们讨论 rtpeer 结构。Peer 结构由过程 init-all{}调用 add-peer{}来进行初始化。

路由 send-periodic-update{}调用 send-updates{}来发送实际的更新消息。然后，在广播间隙稍后，重新制定发送下一阶段更新的时间表，以避免受到时钟同步的影响。

Send-updates{}将发送更新消息到选择的 peers 集。如果节点上的所有路由发生变化，或周期性更新，那么该过程将发送更新消息到所有 peers。另外，如果一些事变链路只是恢复，那么该过程将只发送更新消息到事变链路的临近 peers。

send-updates{}将使用过程 send-to-peer{}来发送实际的更新消息。该过程包含了更新，并把分隔线和标记反向机制考虑在内。它调用了 instproc-like ,send-update{}(注意类似的事件)来发送实际的更新。当前的路由更新保存在类变量 msg_ 中，由一个非负整数作为索引值。Instproc-like 仅向远方的 peer 发送索引到 msg_。这就不再需要来回的把 Otcl 的字符串进行格式变化。

当一个 peer 接收一个路由更新时，它首先检测是否与先前的更新有差别，并决定是否更新，如果更新包含新的消息，agent 则将计算新的路径。

29.6 单播路由对象

ns 单播路由中，Routelogic 和 rtObject 是两个重要的对象。在任意一个单播模拟中，尤其是 Routelogic 声明并维护了所创建的路由表。rtObject 是每个加入动态单播路由节点的对象，如果 Session 路由像一个没有正被模拟的分层路由协议一样的话，注意：节点将不具有该对象的实例。RouteLogic 和 rtObject 的方法将在下一章介绍。

29.7 命令一览

下面是应用在模拟脚本中，单播路由相关命令的一个列表。

$ns_ rtproto <routing-proto> <args>

那些定义应用的路由协议类型像 Static、Manual、Session 和 DV 的地方，args 也许定义将要运行协议的节点的列表。节点列表默认为拓扑结构中的所有节点。

内部方法：

$ns_ compute-routes

该命令为拓扑结构中所有应用拓扑连通性的节点计算下一跳信息。下一跳信息被普通节点的分类器或路由表所使用。依靠模拟中正在使用的寻址类型，compute-routes 调用 compute-flat-routes 或者调用 computer-heir-routes。

$ns_ get-routelogic

它返回指向 RouteLogic 对象（路由表）的一个句柄，如果句柄被创建，那么一个新的路由表对象也就被创建了。

$ns_ dump-routelogic-nh

它删除路由表中的下一跳信息

$ns_ dump-routelogic-distance

它删除路由表中的距离信息

$node add-route <dst> <Target>

它是用来在节点路由表中添加路由实体（下一跳信息）的一个方法。从该节点到<dst>的下一跳是<target>对象，并且这个消息将被添加到节点的分类器中。

$routelogic lookup <srcid> <destid>

返回从 id 是 srcid 的节点到 id 是 destid 的节点的下一跳的节点的 id

$routelogic dump <node id>

删除 id 不是 nodeid 的所有节点的路由表。节点的 id 是按照创建顺序从 0 开始以递增的方式赋值给每个节点的

$rtobject dump-routes <fileID>

删除由 fileID 声明的输出通道的路由表。FileID 必须是由 Tcl 打开命令返回的一个文件句柄，它必须为写入打开。

$rtobject rtProto? <proto>

如果路由协议 agent 存在的话，那么该命令返回有 proto 声明的路由协议的一个句柄。否则，返回一个空字符串。

$rtobject nextHop? <destID>

返回指向由 id 为<destID>节点指明下一跳目标节点的 id

$rtobject rtpref? DestID

返回到达由 destid 指定目标节点的路由优先级

$ rtobject metric? DestID

返回到达 destid 的路由单位

第 30 章

组播路由

本章节介绍 ns 中组播路由执行的内部细节和使用方法。我们首先介绍形成组播路由的用户接口（30.1 节），详细说明使用的组播路由协议，以及目前 ns 专门支持的各种协议的多种方法和配置参数。然后，我们介绍 ns 组播路由详细的内部细节和体系结构（30.2 节）。

这个章节里的过程和函数模块可以在目录~ns/tcl/mcastm，~ns/tcl/ctrmcat 中的各种文件中找到；附加支持的程序可以在~ns/mcast_ctrl.{cc,h},~ns/tcl/lib/ns-lib.tcl, 和~ns/tcl/lib/ns-node.tcl 中找到。

30.1 组播API

在拓扑结构中，组播转发要求增加节点数和链路数。因此，用户在创建拓扑结构前，必须详细说明 Simulator 类的组播要求条件，下面是两种实现方法中的一种：

```
set ns [new Simulator -multicast on]
```

or

```
set ns [new Simulator]
$ns multicast
```

当组播扩展有效时 ,将为组播转发创建带有额外的 classifiers 和 replicators 的节点 ,链路将包含到达一个节点所有包的、指派引入接口列表的元素。

组播路由策略是一种机制，在模拟中组播分布树通过该机制进行计算。Ns 支持三种组播路由计算策略：集中式、密集模式（DM）和共享树模式（ST）。

Simulator 类的 mrtproto{}方法指明了每一种路由策略，是使用集中式组播路由，还是使用特殊的分层组播路由协议。

下面是 ns 组播路由中有效命令的例子：

```
set cmc [$ns mrtproto CrrMcast]   ;# specify centralized multicast for all nodes
$ns mrtproto DM                   ;# cmc is the handle for multicast protocol object
                                  ;# specify dense mode multicast for all nodes
$ns mrtproto ST                   ;# specify shared tree mode to run on all nodes
```

注意：在以上的例子中，CtrMcast 返回一个句柄，该句柄用来对集中式组播路由进行额外的配置。其他路由协议将返回一个空字符串。拓扑结构中的所有节点将运行相同协议的实例。

多个组播路由协议也可以在一个节点上运行，但在这种情况下，用户必须指明那个协议拥有的引入接口，使用 mrtproto-iifs{}以便进行更好的控制。

使用过程 allocaddr{}分配新的或没有使用的多播地址。

Agents 使用 join-group{}和 leave-group{}过程的实例，在 Node 类中加入或离开组播组。这两个过程都带有两个命令行参数。第一个参数确定相应的 agent，第二个参数指定组地址。

下面是一个相对配置比较简单的例子：

```
set ns [new Simulator -multicast on]          ;# enable multicast routing
set group [node allocaddr]                    ;# allocate a multicast address
set node0 [$ns node]                          ;# create multicast capable nodes
set node1 [$ns node]
$ns duplex-link $node0 $node1 1.5Mb 10ms DropTail

 set mproto DM                                ;# configure multicast protocol
 set mrthandle [$ns mrtproto $mprotd]         ;# all nodes will contain multicast protocol agents
set udp [new Agent/UDP]                       ;# create a source agent at node 0
$ns attach-agent $node0 $udp
set src [new Application/Traffic/CBR]
$src attach-agent $udp
$udp set dst_addr_$group
$udp set dst_port_0

set rcvr [new Agent/LossMonitor]              ;# create a receiver agent at node 1
$ns attach-agent $node1 $rcvr
$ns at 0.3 "$node1 join-group $rcvr $group" ;# join the group at simulation time 0.3(sec)
```

30.1.1 组播行为的监控配置

ns 支持一种组播检测模式，该模式可以跟踪用户定义包的行为。并周期性计算通过的包的数目，将结果存入到指定的文

件里。attach{}能够使检测模型打印其输出结果到一个文件。trace-topo{}把检测模式插入到所有链路。filter{}是基于特殊的包头、头部填充域和区域值的，在过滤情况下，重复调用 filter{}将产生 AND 的效果。print-trace{}通知检测模型以开始清除数据。ptype( )是一个全局指针，它带有包的类型名字并把它映射到相应的数值上。在一个特殊组上，过滤 cbr 包的简单配置如下：

```
set   mcastmonitor   [new   McastMonitor]
set   chan   [open   cbr.tr   w]                              ;# open trace file
$mmonitor attach $chan1                ;# attach trace file to McastMoniotor object
$mcastmonitor set period_ 0.02              ;# default 0.02(sec)
$mmonitor trace-topo                              ;# trace entire topology
$mmonitor filter Common ptype_ $ptype(cbr)    ;# filter on ptype_ in Common header
$mmonitor filter IP   dst_$group          ;# AND filter on dst_address in IP header
$mmonitor print-trace              ;# begin dumping periodic traces to specified files
```

下面以简单输出为例说明输出文件的格式（时间，数量）

```
0.16  0
0.17999999999999999   0
0.19999999999999998   0
0.21999999999999997   6
0.23999999999999996   11
0.25999999999999995   12
0.27999999999999997   12
```

30.1.2 协议的特殊配置

在这一节中，我们简要解释执行在 ns 中所有协议的特殊配置机制。

集中式组播 集中式组播是一种类似于 PIM-SM[9]组播的稀疏模型执行方式。用源于共享树的 Rendezvous Point(RP)来创建组播组。实际上删除、添加等消息的发送在节点建立状态下是不能模拟的。用集中式计算 agent 来计算转发树，并设置组播转发的状态，相关节点的<S,G>作为新的接收者加入到组里。从一个发送者到一个组的数据包对 RP 来说是单播的，注意：即使没有组的接受者，从发送者到 RP 的数据包也是单播的。

在模拟中，集中式组播路由的可行的方法是：

```
set mproto    CtrMcast                                    ;# set multicast protocol
set mrthandle    [$ns    mrtproto    $mprotp]
```

该命令过程 mrtproto{}返回一个指向组播协议对象的句柄。这个句柄用来控制 RP 和 boot-strap-router(BSR)，转换特殊组的树类型，从共享树到源特定树和重新计算多播路径。

```
$mrthandle set_c_rp $node0 $node1                ;# set the RPs
$mrthandle set_c_bsr $node0: 0 $node1 :1   ;# set the BSR,specified as list of node :priority

$mrthandle get_c_rp $node0 $group                ;# get the current RP ???
$mrthandle get_c_bsr $node0                      ;# get the current BSR

$mrthandle switch-treetype $group                ;# to source specific or shared tree
$mrthandle compute-mroutes               ;# recomputed routes. usually invoked antomatically as needid
```

注意：当网络动态变化或单播路由变化时，可能会调用 compute-mroutes{}重新计算多播路径。在过渡阶段，瞬间集中式算法的重算特性可能导致暂时违背因果关系。

密集模式 密集模式协议（DM.tcl）是 dense-mode-like 协议的一种实现。根据 DM 类变量 CacheMissMode 的值，它运行在两种模式之一上。如果 CacheMissMode 被设置为 pimdm（默认）的话，将使用 PIM-DM-like 转发规则。否则，CacheMissMode 被设置为 dvmrp(非严格的基于 DVMRP[30])。这两种模式主要的不同点是：DVMRP 为了减少广播数据包链路的数量，而在节点中维护节点之间的父子关系。该实现工作与局域网上点到点的链路上实现相同，并适应网络的动态变化（链路的增加或减少）。

任何一个节点如果收到了去往一个的特定组的数据包，而该节点却没有下游的接受者，那么它将向上游发送删除信息。删除信息引起上游节点激活该节点自身的删除状态。删除状态阻止该节点发送到特殊组下游节点的数据，并在状态有效期内，发送最初的删除消息。通过 DM 类变量 ProuneTimeout 可以在删除状态有效期内进行时间的配置。下面是一个典型的 DM 配置：

```
DM set PruneTimeout 0.3                          ;# default 0.5 (sec)
DM set CacheMissMode dvmrp                       ;# default pimdm
$ns mrtproto DM
```

共享树模式 简化的稀疏模式 ST.tcl 是共享树多播协议的一个版本。由组索引的类变量队列 RP_ 决定了一个特殊组的哪个节点是 RP。举个例子：

ST set RP_($group) $node0

$ns mrtptoto ST

在组播模拟开始时，协议将会在拥有多播发射器的节点上，创建并设置压缩对象，在 RPs 上建立并设置解压缩对象，然后连接它们。加入一个组，节点要向特定组的 RP 发送一个移植消息。离开一个组，节点要发送一个删除消息。目前，就协议而言，并不支持动态的变化和局域网。

双向共享树模式 BST.tcl 是双向共享树协议的一个试验版本。在共享树模式下，必须通过使用类队列 RP_ 来配置 RPs。目前的协议并不支持动态变化和局域网。

30.2 多播路由的内部细节

我们用三个部分来介绍其内部细节：首先是实现和支持多播路由的类；其次是特殊协议实现的详细细节；最后，给出在实现过程中使用的变量列表。

30.2.1 类

组播路由的实现，主要用到类有：mrtObject 类和 McastProtocol 类。它们同样是基类：Simulator、Node、Classifier 等的扩展。这些类和扩展将在下一节里进行介绍。特殊类协议的实现同样要使用附加数据结构来完成特定的任务，诸如分层密集形式中的计时器机制、集中式多路径中的解压缩 agents 等；稍后在介绍特殊类协议自身基础上，我们来介绍这些对象。

mrtObject 类 在每个具有组播能力的节点上，都有一个 mrtObject（aka Arbiter）对象。该对象通过维护由接入接口索引的多播协议的一个队列，来支持节点运行多种组播路由协议的能力。因此，如果一个节点上运行几种组播协议，那么每一个接口仅属于一种协议。所以，该对象支持一个节点运行多种组播路由协议的能力。节点使用仲裁器(arbiter)来执行协议的行为。要么节点上一个特殊协议的实例在运行，要么是所有的协议的实例在运行。

addproto{instance, [iiflist]} 添加协议实例的句柄到它自己协议队列中去。第二个可选参数是由协议控制的接入接口列表。如果该参数为空表，那么就假定协议运行在所有接口上（仅指一个协议的所有接口）。

getType{protocol} 返回与该点特定类型（第一个和第二个参数类型）相匹配的运行协议实例的句柄。该函数通常被应用于其他节点，来查找一个协议的 peer。如果给定的协议类型没有找到，那么它将返回一个空串。

all-mprotos{os,args} 在该节点所有协议实例上，内部路由执行带有 args 的“op”。

start{}

stop{} 开始/停止所有协议的执行

notify{dummy} 在拓扑结构改变发生时被调用，目前 dummy 参数并没有使用

dump-mroutes{file-handle,[grp],[src]} 把组播路径删除到指定的 file-handle 中

join-group{G, S} 向所有协议实例发出加入<S, G>的信号

leave-group{G, S} 向所有协议实例发出退出<S, G>的信号

upcall{code, s, g, iif} 分类器转发错误时，节点发出该信号。通过为特定代码段调用恰当的句柄函数，该程序依次向拥有接入接口（iif）的协议实例发出信号。

drop{rep, s, iif } 包丢失时调用，可能要删除一个接口

另外，类 mrtObject 支持已知组（well known groups）的概念，例如，ns 中预先定义的不需要外在协议支持的 ALL_ROUTERS 和 ALL_PIM_ROUTERS 两个已知组。

类 mrtObject 定义了两个类过程，用来设置和获悉已知组的消息。

registerWellKnownGroups{name} 给已知组的地址指派一个名字

getWellKnownGroup{name} 返回已知组分配的地址 name，如果 name 没有被注册为一个已知组，那么它返回 ALL_ROUTERS 的地址

McastProtocol 类 这是所有组播协议实现的一个基类。它包含基本的组播函数：

start{}, stop{} 设定该协议执行的 status_

getStatus{} 返回该协议执行的状态

getType{} 返回由该实例执行协议的类型

upcall{code args } 对一个将要到的包，要么由于 cache-miss，要么由于 wrong-ii，节点分类器发出一个错误信号时调用。该程序调用协议指定带有用于处理信号的特殊的 arg 的 shandle-<code>{}

对于接口而言，ns 中假定多播实现采取是双工链路。举个例子，node 1 到 node2 有一条单工链路，那么必定存在从 node2 到 node1 的一条反向单工链路。为了能够区别从不同链路接收的包，组播模拟器在每个链路的终点配置带有接口标签的链路，用带有独特的、唯一的标签（id）来标记包。因此，“流入接口”指的就是该标签，它是一个大于等于零的数 。当本地节点发送包到指定的节点 agent 的特殊情况下，流入接口值可能为负值（-1）。

比较而言，“流出接口”指的是一个对象的句柄， 通常是一个 replicator 可以设置的链路的头部。这种设计的目的是重要的：流入接口是一个包的数字标签，而流出接口是一个能够接收包的对象的句柄，如链路头部。

30.2.2 ns中其它类的扩展

在 ns 先前的章节中（第五章）介绍了节点，现在我们介绍 ns 节点的内部体系结构。为了简要回顾一下，多播节点的接

口入口是 switn_。它监视最高位以决定目标是一个组播包，还是一个单播包。组播包转发到维护一个 replicators 表的组播的 classifier 中，每一个<source,group>tuple 中有一个 replicator，Replicator 复制流入的包并转发到所有的流出接口。

Node 类 Node 支持两种主要方式来实现组播：在地址分配的范围内 ，它作为核心服务于组播协议的访问、控制和管理，并聚合动态的数据成员；其次，它提供了一种简单的方法，用来访问和控制该点链路事件上的接口。

expandaddr{}

allocaddr{} 地址管理的类过程。现在 expandaddr{}不太常用，allocaddr{}分配下一个变量的组播地址

start-mcast{},

stop-mcast{} 开始/停止该点的组播路由

notify-mcast{} 当拓扑结构变化或来自邻居节点的单播路由更新时，notify-mcast{}向该点的 mrtObject 发出重算组播路径的信号

getArbiter{} 返回该点运行的 mrtObject 的一个句柄

dump-routes{file-handle} 删除该点的组播转发表

new-group{s g iif code} 当收到一个组播包，而组播 classifier 却不能够为该包找到相应的时隙，那么它就调用 Node nstproc new-group{}来创建相应的入口。该代码指明了没有找到时隙的原因。目前有两种可能：cache-miss 和 wrong-iif。该过程通知仲裁器（arbiter）实例来建立新组

join-group{a g} 加入特定组节点的一个 agent 调用“node join-group<agent> <group>”。该节点向 mrtObject 发出加入特定组的信号，并将 agent 添加到特定组自身的 agents 表中；然后，添加 agent 到与组相联的所有 replicators 里。

leave-group{a g} Node instproc leave-group 与上一个过程正好相反。对组所有的 replicators 来说，它阻止到接收者 agents 的流出接口，从本地 Agents_表中删除接收者的 agents；然后，调用仲裁器（arbiter）实例的 leave-group{}.

add-mfc{s g iif oiflist} Node instproc add-mfc 为一特定<source,group,iif>添加一个组播转发高速缓冲区入口，机制是：

- 创建一个新的 replicator(如果一个都没有存在)
- 更新该节点的 replicator_实例变量队列
- 添加全部的流出接口和本地 agent 到相应的 replicator 中
- 调用组播 classifier 的 add-rep{}来为组播 classifier 里的 replicator 创建一个时隙

del-mfc{s g oiflist} 删除来自<s,g>的 replicator 中，oiflist 里的每一个 oif

控制节点接口的可访问原始表如下：

add-iif{ifid link},

add-oif{link if} 在每个节点链路创建其流入接口标识和流出接口对象期间调用

get-all-oifs{} 返回该节点所有的 oifs

get-all-iifs{} 返回该节点所有的 iifs

iif2link{ifid} 返回带有指定接口标签标识的链路对象

link2iif{link} 返回指定链路的流入接口标签

oif2link{} 返回与指定的流出接口相对应的链路对象

link2oif{link} 返回链路的流出接口（在该点易于发生的 ns 对象）

rpf-nbr{src} 返回指向特定 src 下一跳邻居节点的句柄

getReps{s g} 返回与<s,g>相对应的 replicator 的句柄，任何参数都可能是一个通配符（*）

getReps-raw{s g} 与以上类似，但返回<key,handle>对的一个列表

clearReps{s g} 删除与<s,g>相关的所有 replicators

Link 和 SimpleLink 类 这是放置在每一个链路的简单连接器（connector），它把自身标签的 id 标记在转发过它的包上。该链路上目标节点事件用包的 id 来标识包到达目标节点的链路。由 Link 构造函数来配置标签的 id，该对象是一个内部对象，并不是为用户模拟脚本操作来设计的。该对象只支持两种方法：

label{ifid} 指派该对象将给每一个包标记的 ifid.

label{} 返回该对象标记在每个包上的标签

全局类变量 ifacenum_，指定下一个可用的 ifid 号

Multicast Classifier 类 Classifier/Multicast 维护了一个组播转发的高速缓存。每个节点都有一个组播 classifier。节点在实例化变量 multiclassifier_保存了一个指向该 classifier 的引用。当该 classifier 接收到一个包时，它察看包头部的<source,group>和来自流入接口或 iif 包的接口；在 MFC 里进行检测和确定用于转发包的时隙，在时隙内将指明合适的 replicator。

然而，如果 classifier 没有该<source,group>的入口，或该入口的 iif 不同，它将为 classifier 向上调用 new-group{}，用下面两种代码中一种来标识这个问题：

- cache-miss 指出 classifier 没有找到任何的<source,group>入口
- wrong-iif 指出 classifier 找到了<source,group>入口，但没有与到达包对应的接口

这些 TCL 回调提供了改正状态的机会：安置一个恰当的 MFC-entry(for cache-miss)或为存在的 MFC-entry(for

wrong-iif)改变流入接口。回调的返回值将决定 classifier 怎样处理包。如果返回值是 1，它将假定 TCL 回调已经恰当修改了 MFC，并尝试对下一次包进行分类。如果返回值是 0，不做任何深入检测，将该包丢弃。

add-rep{}在 classifier 里创建一个时隙，并为那个时隙的<source,group,iif>添加一个 replicator。

Replicator 类 当 replicator 收到一个包时，它拷贝该包到它所有的时隙。每一个时隙为特定的<source,group>指向其流出接口。

如果时隙不存在，C++中的 replicator 则调用类实例化过程 drop{}来触发协议的特定行为。在下一节里，介绍每个多播路由协议内部过程时，我们将介绍协议的特定行为。

下面是每个时隙中控制元素的实例过程：

insert{oif} 为下一可得时隙插入一个新的流出接口

disable{oif} 使该时隙不指向特定的 oif

enable{oif} 使该时隙指向特定的 oif

is-active{} 如果 replicator 至少有一个活动的时隙，则返回真

exists{oif} 如果时隙指向特定的 oif 在活动，则返回真

change-iface{source,group,oldiif,newiif} 为特定的 replicator 修改 iif 入口

30.2.3 协议的内部细节

现在我们来介绍不同组播路由协议 agents 的实现。

集中式组播

CtrMcast 继承于 McastProtocol，每个节点需要创建一个 CtrMcast agent。集中式的 CtrMcastComp agent 为整个拓扑结构计算和建立组播路径。每个 CtrMcast agent 处理成员的动态命令，并为重新计算合适的路径而重新定向 CtrMcastComp agent。

join-group{} 用 CtrMcastComp agent 注册新的成员，并调用该 agent 为该新成员计算路径

leave-group{} 恰好与 join-group{}相反

handle-cache-miss{} 当特定的包源没有找到其正确的转发入口时调用。集中式组播情况下，它预示着新的包源已经开始发送包，因此，CtrMcast agent 用 CtrMcastComp agent 来注册新的包源。如果组中的任何成员都计算新的组播树或改组在 RPT（共享树）模式下，则

1. 在包的源点上创建一个压缩的 agent
2. 在 RP 上创建一个对应的解压缩的 agent
3. 通过 unicast 连接两个 agent

4. <S,G>入口向压缩的 agent 指出其流出接口

CtrMcastComp 是集中式组播路由的计算 agent.

reset-mroutes{} 重新设定所有的组播转发入口

compute-mroutes{} (重新)计算所有的组播转发入口

compute-tree{source,group} 在特定组中，计算一个到达所有接收者的源组播树

compute-branch{source,group,member} 当接收者加入一个组播组时执行该过程。或当它本身正在重新计算组播树时，由 compute-tree{}来调用执行，且必须重新定义所有接收者的父节点。算法开始于接收者，递归地找到连续的下一跳，直到到达源或 RP，或到达已经是相关组播树部分的一个节点。在处理过程中，将创建几个新的 replicators 和一个流出接口。

prune-branch{source,group,member} 除了流出接口无效外，该过程与 compute-branch{}相类似；如果该节点出的流出接口为空，那么，它将遍历组播树，删除每一个中间节点，直到找到该特定组播树一个流出接口表非空的节点为止。

密集式模式

join-group{group} 如果<S,G>没有包含任何一个活动的流出时隙(如下游没有接收者)，则它向上游发出移植消息。如果下一跳指向的源是一个局域网，则为该局域网的特定组接收者增加一个计数器。

leave-group{group} 减少局域网的计数器

handle-cache-miss{srcID group iface} 依赖于 CacheMissMode 的值，要么调用 handle-cache-miss-pimdm，要么调用 handle-cache-miss-dvmrp

handle-cache-miss-pimdm{srcID group iface} 如果正确的 iif(来自指向源的下一跳节点)接收到包，就向所有的 oifs 广播包，但反馈 next-hop-neighbor 和不是局域网转发者节点的 oifs 除外。否则，接口不正确，发送一个删除反馈。

handle-cache-miss-dvmp{srcID group iface} 仅向下一跳指向源或父的节点广播包

drop{replicator source group iface} 发送一个删除消息，反馈给上一跳

recv-prune{from source group iface} 如果先前没有删除接口，则重新设置删除计时器，否则，它启动删除计时器并删除接口；进一步来讲，如果流出接口表变空，则向上游广播删除消息。

recv-graft{from source group iface} 取消任何存在的删除计时器和恢复以删除的接口。如流出接口表已经为空，则它向上游转发移植消息

handle-wrong-iif{srcID group iface} 当由于包到达错误的接口，而导致组播 classifier 丢包时，调用该过程和 new-group 过程。由 mrtObject instproc new-group{}调用该路由。调用时，它发送一个删除消息回馈给源

30.2.4 内部变量

mrtObject

protocols_ 节点上活动的协议实例的巨柄队列，该节点的协议操作由流入接口来索引

mask-wkgroups 类变量——定义经常用于标识已知组的 mask

wkgroups 类队列变量——已分配的已知组地址的队列，由组名来索引。Wkgroup(已分配)是一个特殊的变量，该变量指示了当前最高级别的分配组

McastProtocol

status_ 取值“up”或“down”，来指示协议实例执行的状态

type_ 包含由实例来执行协议的类型（类名），如 DM 或 ST

Simulator

multiSim_ 如果能够进行组播模拟，则是 1，否则为 0

MrtHandle_ 集中式组播模拟对象的句柄

Node

switch_ classifier 的句柄 ,用来监测每个包的目标地址最高位 ,以决定它是一个组播包(bit=1)或是一个单播包(bit=0)

multiclassifier_ 与执行<s,g,iif>相匹配的 classifier 的句柄

replicator_ 由句柄<s,g>索引的队列，该句柄向必需的链路复制组播包

Agents_ 在监听特殊组的本地节点上，由 agents 表的组播组所索引的队列

outLink_ 该节点流出接口的高速缓冲表

inLink_ 该节点上流入接口的高速缓冲表

Link 和 SimpeLink

iif_ 设置在该链路上的网络接口对象的句柄

head_ 链路上的第一个对象，不是一个 no-op 连接器，然而，该对象包含实例化变量 link_，用以指向 Link 对象的容器

NetworkInterface

ifacenum_ 类变量——含有下一变量的接口号

30.3 命令一览

以下是组播模拟中，使用的命令列表

set ns [new Simulator -mcast on]

在指定的模拟中，该命令开启组播标志，同时创建模拟对象

ns_multicast

该命令如上面的开启组播标志命令一样

ns_multicast?

如果模拟中组播标志已经开启，该命令返回真，相反，没有开启，则返回假

$ns_mrtproto <mproto> <optional:nodelist>

该命令指定像 DM,CtrMcast 等使用的组播协议<mproto>的类型，作为附加的参数，将运行分层路由协议（不是集中式路由协议）实例的节点列表也能够通过

$ns_mrtproto-iifs <mproto> <node> <iifs>

该命令允许存在比 mrtptoto 更好的控件。自从结点能够运行多径路路由协议起 ,该命令就指定不同的组播协议<mproto>运行在由<node>中<iifs>给定的接入接口上

Node allocaddr

该命令返回一个新的没有使用过的组播地址，用于给一个组来指派组播地址

Node expandaddr

该命令现在不太常用，这个命令由 16 位增到 30 位来拓展地址空间。然而，这个命令已经被 ns_set-address-format-expanded 所代替

$node_join-group <agent> <grp>

该命令是在该点<agent>加入一个特定的组<grp>时使用

$node_leave-group <agent> <grp>

该命令是在该点<agent>决定离开一个特定的组<grp>时使用

内部方法：

$ns_ run-mcast

该命令在所有的节点上开启组播路由

$ns_ clear-mcast

该命令停止所有节点的组播路由

$ns_ enable-mcast <sim>

该命令允许使用特定的组播支持机制（想一个组播 classifier），用以添加一个组播使能的节点。<sim>是一个指向模拟对象的句柄

另外，对于列在此处的内部方法外，还有其他一些特殊方法用于如集中式组播（CtrMcast）、密集式模式（DM），共享树模式（ST）或双向共享树模式（BST）协议，有 Node 和 Link 类方法、网络接口和组播路由中特定的组播 classifier 方法，所有的组播的相关文件可以在 ns/tcl/mcast 目录里找到

集中式组播 当协议在 mrtproto 里指定为 CtrMcast 时，返回一个指向 CtrMcastComp 的句柄

$ctrmcastcomp switch-threetype group-addr

为 group-addr 声明的组，把基于共享树的 Rendezvous Point 转变成 source-specific 树。注意：该方法不能用于为一组播组将 source-specific 树转回到共享树

$ctrmcastcomp set_c_rp <node-list>

该命令用来设置 RPS

$ctrmcastcomp set_c_bsr <node0:0> <node1:1>

该命令用来设置 BSR，由优先级来指定节点列表

$ctrmcastcomp get_c_rp <node> <group>

返回 RP，该 RP 由带有地址 group-addr 组播组的节点可见。注意：如果分割网络，节点可能被分隔在不同的区域内，那么组中不同的节点有可能看到不同的 RPs

$ctrmcastcomp get_c_bsr <node>

返回组目前的 BSR

$ctrmcastcomp compute-mroutes

如果发生网络动态变化或单播路经改变，则该命令重新计算组播路径

密集式模式 密集式模式（DM）协议能够像 PIM-DM 或 DVMRP 一样依赖类变量 CacheMissMode 来运行，没有其他特殊的方法用于这个组播协议对象。类变量有：

PruneTimeout 节点处删除状态消耗的时间，默认值为 0.5

CacheMissMode 用来设置 PIM-DIM 或 DVMRP 类型的转发规则

共享树 该类没有方法，变量有：

RP_ 有组来索引 RP_以决定哪一个节点是特定组的 RP

双向共享树 该类没有方法，变量和介绍过的共享树变量一样。

第 31 章

动态网络（Network Dynamics）

本章描述了 ns 在仿真动态拓扑方面的能力，分为 4 个部分。在 31.1 部分我们通过类 Simulator 实例程序来贯穿本章，这将有助于我们理解仿真脚本。接下来的 31.2 部分介绍包括不同类和实例变量以及过程的内部结构；31.3 部分叙述了与单播路由的 interaction。本章节部分仍然处于实验阶段，因此本章最后部分将简要描述当前版本中存在的不足之处，相信不久的将来这些不足能够被解决。

本章所涉及的过程和函数在 ~ns/tcl/rtglib/dynamics.tcl 和 ~ns/tcl/rtglib/routeproto.tcl （在原文 ~ns/tcl/lib/routeproto.tcl 中是找不到 routeproto.tcl 的）中可以找到。

31.1 用户层接口API

动态网络用户层接口是一个在类 Simulator 中实例过程的集合，是一个跟踪（trace）和记录（log）变化行为的过程。这些过程是 rtmodel，rtmodel-delete 和 rtmodel-at。还有一个过程 rtmodel-configure 是类 Simulator 在仿真开始之前内部用来配置 rtmodels 的。我们将在 31.2 部分介绍。

——实例过程 rtmodel{}定义了一个应用于拓扑中节点和链接的模型。以下是可以用在仿真脚本中的例子：

```
$ns rtmodel Exponential 0.8 1.0 1.0 $n1
$ns rtmodel Trace dynamics.trc $n2 $n3
$ns rtmodel Deterministic 20.0 20.0 $node(1) $node(5)
```

此过程至少需要三个参数（arguments）：

头两个参数（arguments）定义了将被使用的 model，以及配置 model 的参数（parameters）。

在目前 ns 版本中的 models 是 Exponential(On/Off)，Deterministic(On/Off)，Trace(driven)或 Manual(one-shot)模型。

配置参数的数目、格式和解释对于特定的 model 是明确的：

1、Exponential(On/Off) model 有四个参数：[start time]，up interval，down interval，[finish time]。

<start time> 默认值是从仿真开始后 0.5s；

<finish time>默认值是仿真结束的时刻；

<up interval>、<down interval>指明了指数分布的意义 :定义了节点或链接将要各自 up 和 down 的时间。up 和 down 间隔值默认分别为 10s 和 1s。这些值也可以用“-”来表示默认值。如下例子：

```
0.8 1.0 1.0          ;#start at 0.8s, up/down = 1.0s, finish is default
5.0 0.5              ;#start is default, up/down = 5.0s 0.5s, finish is default
- 0.7                ;#start, up interval are default, down = 0.7s, finish is default
- - - 10             ;#start,up,down are default, finish at 10s
```

2. Deterministic(On/Off) model 与 Exponential model 相似，同样也有四个参数：[start time]，up interval，down

interval，[finish time]。

<start time>缺省为仿真的开始，其值为 0.5s；

<finish time>缺省为仿真的结束，其值为仿真的持续时间；

注意 up 和 down 的时间间隔是不同前面的：<up interval> 和 <down interval>分别指明了节点和链接将要 up 和 down 的确切的持续时间。<up interval>的缺省值为 2.0s，<down interval>缺省值为 1.0s。

3.Trace driven model 只有一个参数：trace 文件的名称。输入 trace 文件的格式与由动态 trace 模块产生的输出的格式是一样的，即，v <time> link-<operation> <node1> <node2>。没有反映指定的 node 或 link 的行是被忽略的。如：

v 0.8123 link-up 3 5

v 3.5124 link-down 3 5

4. Manual one-shot model 有两个参数：被执行的操作和被执行操作的时间。

最后的参数（arguments）定义了将使用 model 的节点或链接。如果只有一个节点被指定，那么假设这个节点将失效。通过使节点上的链接事件失效来实现这个假设。如果有两个节点被假定，那么命令（command）假设这两个节点是相邻的，并且模型被应用于这两个节点的链接事件上。如果超过 2 个节点被指定，那么只有第一个 argument 被考虑，后续的 arguments 将被忽略。实例变量，traceAllFile_被设定。

命令返回模型的句柄，这个模型是在此次调用中创建的。

在 ns 内部，rtmodel{}存储了在类 Simulator 实例变量(rtModel_)中创建的路由模型的列表。

—实例过程 rtmodel-delete{}把路由模型的句柄作为参数获取，接着将它从 rtModel_列表中移除，并且删除这个路由模型。

—实例过程 rtmodel-at{}是一个特殊的网络动态化的 Manual 模型的接口。

命令把时间，操作和节点或链接作为参数获取，并且在指定时间针对节点或链接执行操作。命令的例子如下：

```
$ns rtmodel-at 3.5 up $n0
$ns rtmodel-at 3.9 up $n(3) $n(5)
$ns rtmodel-at 40 down $n4
```

最后，类 rtModel 的实例过程 trace-dynamics{}使被这个模型影响的动态变化（dynamics）变得可行。例子如下：

```
set fh [open "dyn.tr" w]
$rtmodel1 trace-dynamics $fh
$rtmodel2 trace-dynamics $fh
$rtmodel1 trace-dynamics stdout
```

在这个例子中，$rtmodel1 将 trace 实体输出到 dyn.tr 和 stdout 中；$rtmodel2 只把 trace 实体输出到 dyn.tr 中。一个典型的被任一个模型写出的 trace 实体的序列可能如下：

```
v 0.8123 link-up 3 5
v 0.8123 link-up 5 3
v 3.5124 link-down 3 5
v 3.5124 link-down 5 3
```

以上几行表明 Link<3,5>在 0.8123s 的时候失效，在 3.5124s 的时候被恢复。

31.2 内部构造

每种动态网络模型是作为独立的类来实现的，它们的基类是类 rtModel。我们首先介绍基类 rtModel 和 31.2.1 部分的驱动类。动态网络模型使用内部队列结构类 rtQueue 来确保仿真事件被正确的执行。接下来 31.2.2 部分将描述这个结构的内部本质。最后，在 31.3.1 部分我们介绍几个已存在的类：节点类，链接类及其他。

31.2.1 类rtModel

为了使用一个新的路由模型，常规 rtmodel{}创建一个相应类型的实例，定义模型操作的对象——节点或链接，配置模型以及可能允许跟踪；不同的分块完成以上操作的实例过程是：

基类的构造函数在它的实例变量中存储了一个指向 Simulator 的 reference，ns_。同时它从同名类变量中初始化了 startTime_和 finishTime。

实例过程 set-elements 用来确定模型操作对象——节点或链接。这个命令存储了两个数组：links_，将被模型操作的那些链接；nodes_，将被由模型引起的那些受链接失效或恢复的事件节点。

在基类中被用来设置模型配置参数的缺省过程是 set-parms。此过程假定合适的 start time ,up interval ,down interval 和 finish time ,并且为一些模型的类设置了配置参数。它在实例变量中存储了这些值 :startTime ,upInterval_ ,downInterval_ , finishTime_。Exponential 和 Deterministic 模型使用这个缺省的值（routine），而 Trace based 和 Manual 模型使用自己定义的过程。

实例过程 trace{}允许 trace-dynamics{}在每一个它所影响的链接上执行。具体的 trace-dynamics{}将在 31.3.1 部分介绍。

接下来的配置步骤仅比仿真开始的时刻早。ns 仅在仿真开始之前调用 rtmodel-configure{}。这个实例过程首先需要一个类 rtQueue 的实例，接着在它的列表 rtModel_中为每一个路由模型调用 configure{}。

实例过程 configure{}生成每一条可被应用于动态的链接；这个链接集被存储在它的实例变量数组 link_中。然后过程调度它的第一个事件。

缺省的实例过程 set-first-event{}调用第一个事件来使所有链接在$startTime_ + upInterval_时刻失效。基于这个基类的不同类型的路由模型应该重定义这个过程。

在基类中的两个实例过程，set-event{}和 set-event-exact{}，在路由队列中能够被用于调度事件。

set-event{interval,operation}在从目前时间后经过 interval 秒后调度操作；它使用了以下的过程 set-event-exact{}。

set-event-exact{fireTime,operation}在 fireTime 时刻执行操作。

如果操作的时间超出 finishTime_,那么只有重新启用失效的 link。

最后，基类提供一个方法用来使链接 up{}或 down{}。每一个方法在实例变量 links_中的每个链接上调用相应的过程。

Exponetial 这个模型在 startTime_ + E(upInterval_)时刻调度它的第一个事件用来使链接失效。

同时，它定义了 up{}和 down{}过程；每个过程调用基类过程用以执行实际的操作。然后这个例程（routine）分别在 E(upInterval)或 E(downInterval_)时刻重新调度下一个事件。

Deterministic 这个模型定义了 up{}和 down{}过程；每一个过程调用基类过程来执行实际的操作。然后这个例程分别在 upInterval 或 downInterval_时刻重新调度下一个事件。

Trace 这个模型重新定义了实例过程 set-parms{}用以操作一个 trace 文件，同时基于输入参数来设置事件。

实例过程 get-next-event{} 从 trace 文件返回下一个有效事件。一个有效事件是指一个事件可适用于在这个对象变量的 links_中的诸多链接中的一个链接。

实例过程 set-trace-events{}使用 get-next-event{}来调度下一个有效事件。

Trace 模型为了使用 set-trace-events{}重新定义了 set-first-event{},up{}和 down{}。

Manual 这个模型被设计来正确的激发（fire）一次。实例过程 set-parms{}使用一次操作和时间来将此操作当作参数（arguments）来执行。set-first-event{}将在合适的时间调度这个事件。

这个例程重新定义 notify{}用在当操作完成时删除对象实例。这个对象自己删除自己的概念是一段脆弱的代码（fragile code）

由于这个对象只被激发一次，同时不能被再次调用，所以它不会重载 up{}或 down{}过程。

31.2.2 类rtQueue

这个仿真器需要并列的同时发生的多个网络动态事件，特别是确保正确的连贯的（coherent）行为（behaviour）。因此，网络动态模型使用它们自己的间隔路由队列来调度动态事件。在仿真器中有一个这个对象的实例，在类 Simulator 中的实例变量 rtq_。

这个队列对象在 rtq_中存储了一个队列操作的数组。其队列索引是指事件将要执行的时间。队列中的每个元素是在某个时刻将被执行的操作的列表。

实例过程 insq{}和 insq-i{}能将元素插入队列中。

第一个参数是当前操作将被执行的时刻。insq{}将确切时间作为参数；insq-i{}将间隔作为参数，同时在当前时刻之后调度操作间隔。

余下的参数指定对象（$obj），对象的实例过程（$iproc）和过程的参数（$args）。

这些参数被放在路由队列中，以便在适当的时刻被执行。

实例过程 runq{}在适当时刻执行 eval $obj $iproc $args。在所有这个实例的事件被执行完成之后，runq{}将要 notify{}每个执行的对象。

最后，实例过程 delq{}能删除一个队列动作以及它的时间和对象名。

31.3 与单播路由的互动

在前面的章节中，我们介绍了单播路由如何应对网络拓扑的变化。本小节详细描述了具体的步骤，由网络动态代码将通知节点和路由拓扑结构变化的情况。

1、rtQueue::runq{}将调用被每个路由模型实例指定的过程。在所有动作完成后，runq{}将通知每一个 models。

2、接着 notify{}将在所有易被影响的链接节点上调用实例过程。每一个路由模型在它的变量数组 nodes_中存储节点列表。

然后它将通知实例 RouteLogic 拓扑发生变化。

3、rtModel 对象为每一个受影响的节点调用类 Node 的实例过程 intf-changed{}。

4、Node::intf-changed{}将通知任何一个在拓扑中可能发生变化的节点上的 rtObject。

再调用那些当仿真使用具体的动态单播路由时被创建的路由对象。

31.3.1 其他类的扩展

默认情况下，已有的类假定拓扑是静态的。在本节中，我们介绍一些为了支持动态拓扑而做了必要改变的类。

我们此前已经在 31.2.1 描述了在类 Simulator 中为了创建和操作路由模型的实例过程，比如 rtmodel{}，rtmodel-at{}，rtmodel-delete{}和 rtmodel-configure{}。同样的，类 Node 包含了我们在 31.3 介绍过的实例过程 intf-changed{}。

动态网络的代码在各自的链接上操作。每一个模型普遍的将它的规范解释成为在适当链接上的操作。接下来的我们介绍类 Link 及相关类。

类 DynamicLink 这个类在动态网络代码中是唯一的 TclObject。其影子（shadow）类是类 DynaLink。这个类支持支持边界变量，status_。当链接连接起来的时候 status_是 1，而链接失效时是 0。影子对象的 recv()方法检查 status_变量，为了决定一个分组是否要被转发。

类 Link 这个类支持原语(primitives)：up,down，同时 up 用来设置查询 status_。这些原语是这个类的实例过程。

实例过程 up{}和 down{}分别设置 status_为 0 和 1。

此外，当链接失效时，down{}将重设所有连接器用以包装链接。每一个连接器，包括所有队列和延迟对象将清洗缓冲区和丢弃所有现存的分组。这样做是仿真由于链接失效而引起分组的丢弃。

接着两个过程把在列表 dynT_中的每个文件句柄写入跟踪实体中。

实例过程 up?{}返回当前的 status_值。

此外，这个类包含了实例过程 all-connectors{}。这个过程把一个操作作为参数，并且一律将此操作应用于所有为 TclObject 而被处理的类实例变量。

类 SimpleLink 该类支持两个实例过程 dynamic{}和 trace-dynamics{}。我们已经在类 rtModel 的过程 trace{}中介绍过后者。

实例过程 dynamic{}在队列头插入一个 DynamicLink 对象。实例表明如果对象被定义，则对象的下行为 drpT_，链接的丢弃目标，或者是仿真器中的 nullAgent_。同时实例标识每个链接中的连接器，链接现在是动态的。

大多数连接器忽略这个标识动态的标记除了 DelayLink 对象。这个对象通常将调用每个分组，而这些分组是被目的节点在适当的时刻所接收的。当链接是动态的时候，对象将在内部把每个包排队；它只调度下一个将要发送数据包的事件，而不是通常地每包事件。如果链接失效，路由模型将发送一个 reset 信号，影子对象接收到这个信号后，将执行它的相应的 reset 过程，清空内部队列中所有的数据包。具体的关于 DelayLink 的介绍在第八章中。

31.4 目前网络动态API中的缺陷

目前网络动态 API 中的缺陷如下所示：

1、不能在锁定步骤(lock-step)的动态同步中指定一个节点集或链接集

2、节点失效本该由其自身机制来处理，而不应该由链接失效的二级成员来决定。这可以在以下几种情况下见到：

（1） 仿效链路中断时的节点失败的方法。理想的情况下，节点失败应该引起将要被重新设置的节点上的所有 agent 事件。

（2）没有和节点失败相关联的跟踪。

如果两个不同的路由模型适用于一个普通节点上的两个独立链路事件，而且两个链路同时经历了系统变化，那么将会再次通知节点。

31.5 命令一览

以下命令列表是在 ns 中仿真动态网络场景中将会使用到的：

$ns_ rtmodel <model> <model-params> <args>

这个命令定义了适用于系统节点和链路的动态模型。前两个参数包括 rtmodel 和配置模型的参数。<args>代表了不同类型的参数，这些参数是不同动态模型所需要的。这个命令返回一个和指定模型相对应的模型对象的句柄。

- 在确定性模型中，<model-params>可以是<start time>,<up-interval>,<down-interval>和<finish-time>。从 start-time 开始启动，链路打开 up-interval 和关闭 down-interval，直到 finish-time 结束。start time,up-interval,down-interval 的缺省值分别是 0.5s,2.0s,1.0s。finish-time 默认是仿真结束的时间。start-time 默认为 0.5s 是为了使路由协议能够计算完成。
- 如果是指数模型的话，<model-params>可以是<up-interval>,<down-interval>。这里链路的 up-time 是一个关于平均 up-interval 的指数分布，而链路的 down-time 是一个关于平均 down-interal 的指数分布。up-interval 和 down-interval 的默认值分别是 10s，1s。

- 如果是手工分布的话，<model-params>可以是<at>,<op>。其指的是在特定的时刻，哪一个操作 op 应该执行。op 是一个 up 或是 down。手工分布可以使用后面章节中介绍的 rtmodel-at 方法来声明。
- 如果跟踪当作模型来声明的话，将从一个跟踪文件中读取动态链路或节点。<model-params>在这种情况下将是拥有动态信息的跟踪文件的句柄。跟踪文件的格式和跟踪动态链路方法产生的跟踪输出是一样的。

$ns_ rtmodel-delete <model>

这个命令把路由模型<model>的句柄当作一个参数，从仿真维护的 rtmodel 列表中清除并且删除模型。

$ns_ rtmodel-at <at> <op> <args>

这个命令是动态网络的手工模型的一个特定接口。它有时间<at>、操作类型<op>和把操作<args>当作变量的节点和链路。在时间<at>，可能开启或关闭的操作<op>被应用于一个节点或链路。

$rtmodel trace <ns> <f> <optional:op>

这个命令能使链路中该模型影像的动态跟踪生效。<ns>是一个仿真的实例，<f>是跟踪信息写入的文件，而<op>是一个用来定义操作类型的可选参数。这是类 trace-dynamic 的一个封装。

$link trace-dynamic <ns> <f> <optional:op>

这是一个类的连接实例程序，用来在特定链路上建立动态跟踪。参数和上面讲的类 rtModel 程序的一样。

$link dynamic

这个命令在队列的首部插入一个对象 DynamicLink，通知链路中所有链接目前的链路是动态的。

内部过程：

$ns_ rtmodel-configure

这是一个配置动态模型的内部过程，而且这些动态模型必须是在目前仿真所维护的模型中已经存在的。

第 32 章

分层路由(Hierarchical Routing)

本章讲述了 ns 中分层路由的内部机理，包括两个部分。在第一部分中我们概括了分层路由；第二部分，我们逐个介绍用来设置分层路由的 API，并描述了其结构、内部机理和相应的代码路径。

本章涉及的函数和过程可以在~ns/tcl/lib/ns-hiernode.tcl,tcl/lib/ns-address.tcl,tcl/lib/ns-route.tcl 和 route.cc 中找到。

32.1 分层路由概述

分层路由主要是在大型拓扑网络的仿真中用来减少内存需求的。一个拓扑被分成几个层次，因而缩小了路由表。使用分层结构，能使一张二维路由表的复杂度从 n^2 降低到 log*n*。但是分层同时也带来了开销的增加。最合适的结构是分层三层路由，而且目前 ns 也最多只支持 3 层。

为了能在仿真中使用分层路由，我们需要定义拓扑的层次和提供给节点分层的地址。在平面路由中，每个节点都知道拓扑中的其他节点，因此路由表的信息长度为 n^2。对于分层路由，每个节点仅知道同层节点的信息。为了能够跟其它层次的节点通信，节点需要通过同层中的边界路由来转发分组。由此其路由表可以缩小到 log*n*。

32.2 分层路由的使用

分层路由需要一些 ns 的附加功能和机制。举例来说，要为 hier rtg 定义一个新的被称为"HierNode"的节点对象。因此用户必须在创建网络之前详细说明分级路由的要求。可以做到这一点，如下所示：

首先，节点和端口地址使用的地址格式或者地址空间要在分层模型中设置好。有两种方法，其中一种是：

```
set ns [new Simulator]
$ns set - address -format hierarchical
```

这把节点地址空间设置成三个级别，每个级别分配了 8 个 bit。或者可以如下：

```
$ns set -address -format hierarchical <n hierarchy levels> <# bits in level 1>...< # bits in nth level>
```

为 n 个层的网络创建了一个节点地址空间，并为每个层分配了所要求的 bit 数。

这不仅创建了分层的地址空间，也设置了一个称为 EnableHierRt_的标志，并把 Simulator 的类变量 node_factory_设置成了 HierNode。因此当通过调用 Simulator 方法来创建节点"node"时，可以：

```
$ns node 0.0.1 # a HierNode is created with an address of 0.0.1;
```

AddrParams 类可以用来存放拓扑的层次（如层次的数目），每一个层中区域的数目（域的数目），簇的数目和每一个簇中节点的数目。

为 AddrParams 提供以上信息的 API 如下所示：

```
AddrParams set domain_num_ 2
lappend cluster_num 2 2
AddrParams set cluster_num_ $cluster_num
lappernd eilastlevel 2 3 2 3
AddrParams set nodes_num_ $eilastlevel
```

以上定义了一个拥有两个域的拓扑，记为 D1 和 D2，每个域各有两个簇（C11 和 C12 在 D1，C21 和 C22 在 D2）。接着 4 个簇分别有 2,3,2,3 个节点。

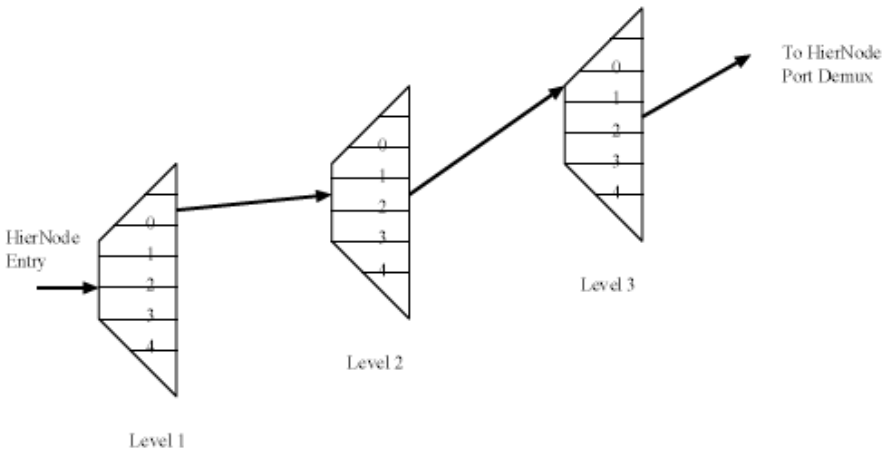
默认情况下，AddrParams 提供的拓扑有一个域，该域有 4 个簇，每个簇有 5 个节点。

同时 AddrParams 为每个层次的每个节点自动分配掩码和 shift value。

每个 HierNode 节点在其创建时刻都会调用方法"mk-default-classifier"来为 n 个层建立 n 个地址分类器。

```
HierNode instproc mk-default-classifier
  $self instvar np_ id_ classifiers_ agents_ dmux_ neighbor_ address_
  # puts "id=$id_"
  set levels [AddrParams set hlevel_]
  for set n 1 $n <= $levels incr n
    set classifiers_($n) [new Classifier/Addr]
    $classifiers_($n) set mask_ [AddrParams set NodeMask_($n)]
    $classifiers_($n) set shift_ [AddrParams set NodeShift_($n)]
```

在路由计算时刻，需要调用 add-route。如下图所示，在 otcl 方法中 add-route 如何在分类器中转移：



3-Level classifiers for HierNode (hier-addr:0.2.1)

Figure 32.1: Hierarchical classifiers

```
Node instproc add-route dst target
  $self instvar rtnotif_
# Notify every module that is interested about this
# route installation
if $rtnotif_ != ""
```

$rtnotif_ add-route $dst $target
$self incr-rtgtable-size

对于一个三层例子来说，第一层的分类器是域，第二层是节点所在域中的所有簇，最后一层是在某个簇中所有的节点。对于这样一个拓扑结构，带有地址 0.1.2 的 hierNode 看起来像下面的数字：

因此路由表的大小可以缩小到 logn，而不是平面路由中的 n^2，这样就每个节点就不需存储拓扑中所有节点的下一条信息。相反，在分层路由中，每个节点只要存储同一簇中其他相邻节点的信息和所处域中所有簇的信息，以及所有域的信息就可以了。这样大大节省了运行期间仿真所需的内存。

32.3 创建大规模分层拓扑

前面部分已经介绍了手工创建分层拓扑的方法。然而，在 ns 中有一个可用的脚本，它可以把 Georgia-tech 的 SGB-graphs 转换成 ns 的兼容分级网络。这部分请参考 http://wwwmash.CS.Berkeley.EDU/ns/ns-topogen.html。

见~ns/tcl/ex 中的示例脚本 hier-rtg-10.tcl 和 hier-rtg-100.tcl。

32.4 带SessionSim的分层路由

分层路由可以和 Session 仿真(见第 42 章)联用。如果 Session 级别并且在大规模拓扑中使用多播路由的仿真采用分层路由，将会大大节省内存。参见在~ns/tcl/ex/newmcast/session-hier.tcl 的示例脚本。

32.5 命令一览

以下是在仿真脚本中的分层路由和地址的相关命令列表：

$ns_ set-address-format hierarchical

这个命令被用来在 ns 中建立分层路由。然而，在最近的节点 API 的改变中，这个命令被替换成：

$ns_ node-config -addressType hierarchichal

这个命令创建了一个缺省的 3 层路由拓扑结构，每层分配 8bit。

$ns_ set -address-format hierarchical <nlevels> <#bits in level1>...<#bits in level n>

这个命令创建了一个有<nlevels>的层级，按照参数中说明的那样给每个层次分配了一定的 bit。

AddrParams set domain_num_ <n_domains>
AddrParams set cluster_num_ <n_clusters>
AddrParams set nodes_num_ <n_nodes>

以上 API 用来指定分层网络，比如拓扑结构中域、簇及节点的数目。其缺省值为一个拓扑中有一个域，此域中有 4 个簇，每个簇中有 5 个节点。

内部过程

$Node add-route <dst> <target>

这个过程用来为给定的<target>增加一个到目的<dst>的下一跳的实体。

$hiernode_ split-addrstr <str>

这个命令在每层中将一个分层地址字符串（如 a.b.c）分成一个地址列表（如 a,b 和 c）

第五篇

传送（Transport）

第 33 章

UDP Agent

33.1 UDP Agents

UDP Agents 是在 udp.{cc,h}里实现的。一个 UDP Agent 从应用层程序可变大小的数据块 ,如果需要 ,则将它们分段。UDP 分组也包含一个单调增加的序列号和一个 RTP 时间戳。真实的 UDP 分组并不包含序列号和时间戳。序列号和时间戳都不计算在分组的长度内，主要用于 trace 文件的分析和一些使用 UDP 的应用层程序。

默认的 UDP 代理的最大段长（MSS）是 1000 字节。

```
Agent/UDP set packetSize_ 1000 ;# max segment size
```

packetSize_这个 OTcl 的实例变量是和 C++中的 size_变量绑定的。

应用层程序可以通过 c++类中的 sendmsg()访问 UDP agents，或者通过 Otcl 类中的 send 和 sendmsg，详见 38.2.3

下面是一个使用 UDP Agents 的简单例子。在这个例子中，CBR 流量产生器在 1.0s 时启动，然后周期性地调用 UDP 的 sendmsg（ ）函数

```
set ns [new Simulator]
set n0 [$ns node]
set n1 [$ns node]
$ns duplex-link $n0 $n1 5Mb 2ms DropTail
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
$udp0 set packetSize_ 536 ;# set MSS to 536 bytes
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0
$ns connect $udp0 $null0
$ns at 1.0 "$cbr0 start"
```

33.2 命令一览

下面这些命令用来在模拟脚本中设置 UDP 代理：

```
set udp0 [new Agent/UDP]
```

建立一个 UDP agent.的实例

```
$ns_ attach-agent <node> <agent>
```

将<agnet>绑定到<node>的常用命令

$traffic-gen attach-agent <agent>

这是类 Application/Traffic/<traffictype>中将流量产生器和<agent>连接的方法，比如为一个 UDP Agents udp1 建立一个 CBR 流量就可以采用下面的命令：

set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $udp1
$ns_ connect <src-agent> <dst-agent>
这个命令为两个端节点（在运输层）建立了连接

$udp set packetSize_ <pktsize>
$udp set dst_addr_ <address>
$udp set dst_port_ <portnum>
$udp set class_ <class-type>
$udp set ttl_ <time-to-live>
..... etc

上面是可能用来设置 udp 代理的不同的参数值，默认值见 ns/tcl/lib/ns-default.tcl ，至于在模拟中用来建立一个 UDP 代理的例子，见上面的 33.1 章。

第 34 章

TCP Agents

本章介绍 ns 中的 TCP Agent 的操作。TCP Agent 主要有两种：单向 agent 和双向 agent。单向 agent 可以进一步分为一系列的 TCP 发送端（采用不通的拥塞控制机制技术）和接收端（“sink”）；而双向 agent 即可以做为一个发送器也可以做为接收端，它还有待于进一步的发展。

本节所要介绍的文件太多，不可能在此一一列举。它们基本覆盖了形式为～ns/tcp*.{cc,h}的大部分文件。 目前现存的单向 TCP 发送 agent 如下：

- Agent/TCP - a “tahoe” TCP sender
- Agent/TCP/Reno - a “Reno” TCP sender
- Agent/TCP/Newreno - Reno with a modification
- Agent/TCP/Sack1 - TCP with selective repeat (follows RFC2018)
- Agent/TCP/Vegas - TCP Vegas
- Agent/TCP/Fack - Reno TCP with “forward acknowledgment

单项接收 TCP Agent 如下：

- Agent/TCPSink - TCP sink with one ACK per packet
- Agent/TCPSink/DelAck - TCP sink with configurable delay per ACK
- Agent/TCPSink/Sack1 - selective ACK sink (follows RFC2018)
- Agent/TCPSink/Sack1/DelAck - Sack1 with DelAck

目前现存的实验性双向发送器只有 TCP Reno 形式的：

·Agent/Tcp/FullTcp

这一章包括三部分：第一部分简要概述并列举说明基本 TCP 发送/接收 agent 的构造（接收端不需要构造）；第二部分介绍基本发送 agent 的内核；最后一部分介绍仿真中现存的扩展了的 agent。

34.1 One-Way TCP Senders

仿真器支持多个版本的抽象 TCP 发送端。这些对象试图抓住 TCP 拥塞差错控制操作的精髓，但不是完全复制实际使用的 TCP 设置。它们不含动态窗口通知，而只是在信息包单元中 记入 segment 号和 ACK 号。它没有 SYN/FIN 链接的建立/断开过程，也没有数据被传输（例如：没有检查和 和 紧急数据）。

34.1.1 The Base TCP Sender (Tahoe TCP)

“Tahoe” TCP agent Agent/TCP 采用了和 UC Berkeley 编写的 4.3BSD “Tahoe” UN’X system 类似的拥塞控制机制和 rtt 估算方法。在慢启动阶段(cwnd_<ssthresh_)时 ,每收到一个新的 ACK ,拥塞窗口就增加 1 ;在拥塞避免(cwnd>

= ssthresh_ ）阶段,每收到一个新的 ACK，拥塞窗口就增加 1/cwnd_。

Responses to Congestion： 在 Tahoe TCP 中，当出现 3 次（ NUMDUPACKS ）重复 ACK 时或者重传计时器超时，就认为包（ 因为拥塞 ）已丢失。此时，Tahoe 的响应是将 ssthresh_设定为当前窗口大小（ cwnd_和 window_中的最小值 ）的一半与 2 之中的较大值。并把 cwnd_变为初始值 windowInit_。这样使得 TCP 又进入慢启动（ slow_start ）阶段。

Round-Trip Time Estimation and RTO Timeout Selection： rtt_，srtt_，rttvar_，tcpTick_和 backoff_用于估算 RTT(往返时延)，TCP 初始化 rttvar = 3/tcpT ick_，backoff = 1。重传定时器设为当前时间 + max(bt(a + 4v + 1), 64)秒。其中 b 为当前 backoff 值，，t 是 tcpTick 的值，x 是 srtt 的值，v 是 rttvar 的值。

往返时延的取样值(sample)在新 ACK 到来时计算。RTT 取样值是当前时间与 ACK 信息包的响应时间之差。在得到第一个取样值之后，它将会被用作为 strr_的初始值，它的 1/2 被用作为 rttvar_的初始值。但是对于随后的取样值，它的数值更新公式如下

$$srtt = \frac{7}{8} \times srtt + \frac{1}{8} \times sample$$

$$rtt\ \mathrm{var} = \frac{3}{4} \times rtt\ \mathrm{var} + \frac{1}{4} \times \left| sample - srtt \right|$$

34.1.2 配置

实现一个 TCP 仿真需要创建构造 agent，绑定应用层数据源（ 业务发生器 ），启动 agent 和业务发生器。

34.1.3 简单的配置

创建一个 agent

```
set ns [new Simulator] ;# preamble initialization
set node1 [$ns node] ;# agent to reside on this node
set node2 [$ns node] ;# agent to reside on this node
set tcp1 [$ns create-connection TCP $node1 TCPSink $node224]
$tcp set window_ 50 ;# configure the TCP agent
set ftp1 [new Application/FTP]
$ftp1 attach-agent $tcp1
$ns at 0.0 "$ftp start"
```

这个例子说明了仿真器内部函数 create-connection 的使用方法。它的参数有：源 agent、源节点、目标 agent、目标节点、链接所用的流 id。它的运行过程为 :创建两个 agent、设定 agent 中的流 id 域、在它们相关的节点上添加源 agent 和目标 agent、最后连接 agent（ 即设定合适的源地址、目的地址和端口 ）。函数的返还值是所创建的源 agent 的名称。

TCP Data Source TCP Agent 并不产生应用数据，但是我们可以把任何业务发生模块连接到 TCP Agent 以产生数据。TCP 的两个常见应用为 FTP 和 Telnet。FTP 中传送大量的数据；而 Telnet 中则是从 tcplib（ 见文件 tcplib-telnet.cc ）中随机选择器传送数据量的大小。这些应用的源对象的构建见 38.4 节。

34.1.4 其它配置参数

除了上面 window_参数之外，TCP agent 还支持其他的配置变量。这一分节所介绍的变量既是类变量又是实例变量。

随着类变量的改变，随后所创建的所以 agent 的缺省值也将发生变化；而一个指定 agent 的实例变量的改变仅仅影响那个 agent 所用的值。例如：

```
Agent/TCP set window_ 100 ;# Changes the class variable
$tcp set window_ 2.0 ;# Changes window_ for the $tcp object only
```

每个 TCP agent 的缺省参数为：

```
Agent/TCP set window_ 20 ;# max bound on window size
Agent/TCP set windowInit_ 1 ;# initial/reset value of cwnd
Agent/TCP set windowOption_ 1 ;# cong avoid algorithm (1: standard)
Agent/TCP set windowConstant_ 4 ;# used only when windowOption != 1
Agent/TCP set windowThresh_ 0.002 ;# used in computing averaged window
Agent/TCP set overhead_ 0 ;# !=0 adds random time between sends
Agent/TCP set ecn_ 0 ;# TCP should react to ecn bit
Agent/TCP set packetSize_ 1000 ;# packet size used by sender (bytes)
Agent/TCP set bugFix_ true ;# see explanation
Agent/TCP set slow_start_restart_ true ;# see explanation
Agent/TCP set tcpTick_ 0.1 ;# timer granulatiry in sec (.1 is NONSTANDARD)
Agent/TCP set maxrto_ 64 ;# bound on RTO (seconds)
Agent/TCP set dupacks_ 0 ;# duplicate ACK counter
Agent/TCP set ack_ 0 ;# highest ACK received
Agent/TCP set cwnd_ 0 ;# congestion window (packets)
Agent/TCP set awnd_ 0 ;# averaged cwnd (experimental)
Agent/TCP set ssthresh_ 0 ;# slow-stat threshold (packets)
Agent/TCP set rtt_ 0 ;# rtt sample
Agent/TCP set srtt_ 0 ;# smoothed (averaged) rtt
Agent/TCP set rttvar_ 0 ;# mean deviation of rtt samples
Agent/TCP set backoff_ 0 ;# current RTO backoff factor
Agent/TCP set maxseq_ 0 ;# max (packet) seq number sent
```

对许多仿真而言，几乎不需要改变任何配置参数。经常改动的参数有 window_和 packetsize_：前者限制了 发送端的拥塞窗口大小，它的作用是真实的 TCP 连接中接收端宣布的接收窗口（尽管它保持不变）；而 packetsize_必然起着与实际 TCP 的 MSS 大小类似的作用。改变这些参数可对 TCP 的表现产生深远的影响。一般来说，具有较大的包头、较大的窗口、较小的往返时延（拓扑和拥塞的结果）的 TCP 连接在利用网络带宽方面将具有优势。

34.1.5 Other One-Way TCP Senders

Reno TCP： Reno Tcp 与 Tahoe Tcp 很相似。他们的一个不同在于 Reno Tcp 具有快速恢复功能。在新分组到达以前 Reno 用重复 ACK 同步即将发出的分组。新的 ACK 是指比已经收到的 ACK 号中的最大值更大的号的 ACK。此外，Reno Tcp agent 在快速重传时不采用慢启动。但是，它把拥塞窗口设定为当前窗口的 1/2，复位 ssthresh_以匹配此值。

Newreno TCP： 他是在 Reno TCP agent 的基础上改进而来的，它修改了接收新 ACK 后的操作。，只有当发送器所接收的 ACK 序号是进入快速恢复前的最好序号时，Newreno 才退出快速恢复。因此当一个数据窗口出现多个丢包时不需要出现多个重传超时。

Vegas TCP：此 agent 配置了“Vegas” TCP（[4,5]）。由 Ted Kuo 提供。

Sack TCP：此 agent 基于接收器所提供的选择性 ACK 而采用了选择重传机制。它在[23]中所介绍的 ACK 方案的后面，并且被 Matt Mathis and Jamshid Mahdavi 改进了。

Fack TCP： 此 agent 采用了“前向 ACK” TCP，是由对 Sack TCP 的修改而得到。见[22].

34.2 TCP Receivers (sinks)

TCP Sender 只是发送端单方面的行为，它需要与 TCP Receivers 相配合

34.2.1 The Base TCP Sink

基础的 TCP 接收器对象（Agent/TcpSink）负责将 ACK 发送给到对应的 TCP 源对象。它每收到一个信息包就产生一个 ACK，ACK 的大小可以设置。TCP 接收器对象的创建构造一般是通过调用库函数来自动实现的(见上述 create_connection)。

它的结构参数：

```
Agent/TCPSink set packetSize_ 40
```

34.2.2 Delayed-ACK TCP Sink

Delayed-ACK 接收器对象（Agent/Agent/TCPSink/DelACK）仿真 的 TCP 接收端可以产生小于收到包的个数。此对象包含一个捆绑变量 interval_，它表示 发送相隔 ACK 之间的等待时间（单位为秒）。Delayed ACK 接收端在收到顺序 ACK 时采用了 aggressive ACK 策略，当接收端收到无序包时产生 ACK。

它的结构参数为：

```
Agent/TCPSink/DelACK set interval_ 100ms
```

34.2.3 Sack TCP Sink

选择性确认（selecttive-acknowledgement）TCP 接收器（Agent/TCPSink/Sack1）采用了基于 RFC2018 的 SACK 发生策略。它具有一个捆绑变量 maxSackBlocks_，它给出了 ACK 信息的最大 block 数目（available for holding SACK information），缺省值为 3（与 RTTM 的期望值一致）(见 RFC2018 第三节)。 Agent/TCPSink/Sack1/DelAck 型对象采用延迟了的选择性 ACK。

结构参数：

```
Agent/TCPSink set maxSackBlocks_ 3
```

34.3 Two-Way TCP Agents (FullTcp)

对象 Agent/TCP/FullTcp 是仿真器所提供的一套 TCP agent 之外的又一新对象，而且它还有待于进一步的发展。它与其它的 agent 不同而且不兼容，但是它们都使用某些相同的结构。它与其它 agent 的区别如下：

·连接可被建立或拆除（交换 SYN/FIN 信息包）。

·支持双向数据传输。

·队列长度以字节（而非包序号）为单位。

在试图通过进行简短的数据传输的仿真行为时，SYN 包及其 ACK 的产生显得至关重要。在默认的情况下，目前的仿真器中的 TCP 版本是从在三次握手的第三部分(segment)发送数据的。这与实际情况有所不同。一次典型的 TCP 连接过程为：发起方(客户端)发送 SYN 包；被动方(服务器)作出响应并发送 SYN + ACK 包；发送方发送一个 ACK 包相应， 经过一段时间之后，(服务器)根据第一次请求记录开始发送第一部分数据。因而，现有版本 TCP 发送数据时间要比实际配置的早一点。此 TCP 也可以被构建为在初始 SYN 部分发送数据。将来，对 FullTCP 的改进内容可能包括：对延迟第一部分数据发送的修改；可能对 T/TCP 功能配置的修改。

目前，FullTCP 仅仅采用了 Reno 拥塞控制机制，但是它最终还将采用全范围的拥塞控制算法（如“Tahoe、Sack、Vegas 等）。

34.3.1 Simple Configuration

进行 FullTCP 仿真需要创建构造 agent、添加应用层数据源（业务发生器）、启动 agent 和业务发生器。

```
Creating the Agent
# set up connection (do not use "create-connection" method because
# we need a handle on the sink object)
set src [new Agent/TCP/FullTcp] ;# create agent
set sink [new Agent/TCP/FullTcp] ;# create agent
$ns_ attach-agent $node_(s1) $src ;# bind src to node
$ns_ attach-agent $node_(k1) $sink ;# bind sink to node
$src set fid_ 0 ;# set flow ID field
$sink set fid_ 0 ;# set flow ID field
$ns_ connect $src $sink ;# active connection src to sink
# set up TCP-level connections
$sink listen ;# will figure out who its peer is
$src set window_ 100;
```

FullTcp 的创建与其它 agent 的相似。但是，接收器被程序 listen 设定在收听状态。建立呼叫需要使用指向接收器的指针。因而 create-connection 将不能使用。

Configuration Parameters：Tcl 中 FullTcp agent 的配置参数如下：

```
Agent/TCP/FullTcp set segsperack_ 1 ;# segs received before generating ACK
Agent/TCP/FullTcp set segsize_ 536 ;# segment size (MSS size for bulk xfers)
Agent/TCP/FullTcp set tcprexmtthresh_ 3 ;# dupACKs thresh to trigger fast rexmt
Agent/TCP/FullTcp set iss_ 0 ;# initial send sequence number
Agent/TCP/FullTcp set nodelay_ false ;# disable sender-side Nagle algorithm
```

Agent/TCP/FullTcp set data_on_syn_ false ;# send data on initial SYN?
Agent/TCP/FullTcp set dupseg_fix_ true ;# avoid fast rxt due to dup segs+acks
Agent/TCP/FullTcp set dupack_reset_ false ;# reset dupACK ctr on !0 len data segs containing dup ACKs
Agent/TCP/FullTcp set interval_ 0.1 ;# as in TCP above, (100ms is non-std)

34.3.2 BayFullTcp

Kathy Nichole/Van Jacobson 小组已经向 ns 中添加了一个不同的双向 TCP，即 Bay FullTcp。它与 FullTcp 的基本不同点如下：

- BayTcp 支持 client-server 应用模型，但是 FullTcp 不提供应用层。
- 两着的 TCP 应用接口不同。
- FullTcp 支持 partial ACK，但是 Bay Tcp 则不支持。
- FullTcp 支持多种 TCP（tahoe、reno 等），但是 Bay Tcp 不支持。
- 两者的 API 不同。

这两者之间或许还有其它的差别。我们将来的计划之一就是要重新定义 API 来让 FullTcp 使用 client-server 模型。

34.4 Architecture and Internals

基类 TCP agent（类 Agent/TCP）被构建为实例程序(routines)的集合：发送信息包，处理 ACK，管理发送窗口，处理超时。一般来说，每一个实例程序(routines)都可以在派生类中重写。这正是配置多种不同的 TCP 发送器的方法所在。

The TCP header： 文件~ns/tcp.h 中结构参数 hdr_tcp 定义了 TCP 报头。基本 agent 仅使用下列域的子集：
ts_ /* current time packet was sent from source */
ts_echo_ /* for ACKs: timestamp field from packet associated with this ACK */
seqno_ /* sequence number for this data segment or ACK (Note: overloading!) */
reason_ /* set by sender when (re)transmitting to trace reason for send */

Functions for Sending Data 注：发送 TCP 并不真正发送数据，而只是设置数据包的长度
send_much(force, reason, maxburst) 它试图发送 当前发送窗口所允许的最大数量的信息包。他会跟踪他发送的包的数目，发送总量被限定为 maxburst_。
output(seqno, reason) 他发送给定序号的包，并且更新发送的最大序号(maxseq_),这个函数会给 TCP 包头赋值(sequence number, timestamp, reason for transmission)。这个函数还会为发送包设置重传定时器。

Functions for Window Management 有效发送窗口在任何时刻都由函数 window（ ）所给出。它给出最小拥塞窗口和变量 wnd_。其中 wnd_表示接收端的告知的接收窗口大小。
opencwnd() 在接收到新 ACK 的时候，可以调用此函数来打开(open 应该指的是增加拥塞窗口)拥塞窗口。在慢启动的情况下，每收到一个 ACK，此函数使 cwnd_加 1。在拥塞避免的情况下，标准配置把 cwnd_增大 1/cwnd。在拥塞避免阶段，还设置了其它窗口增长选项，但是它们只是试验性的（并非正式的）；要了解其详解，请联系 Sally Floyed。
closecwnd(int how) 此函数可以用来缩小拥塞窗口。当进入快速重传阶段、定时器超时或则拥塞报告（ECN 比特标记）时调用此函数。它的参数 how 表明将会怎样缩小拥塞窗口。0 值用于 Tahoe TCP 重传超时和快速重传，但是它常常会导致 TCP 进入慢启动状态 并把 ssthresh_减小为当前窗口的 1/2。值 1 在 Reno TCP 中用于快速恢复（避免返回到慢启动状态）。值 2 用于 ECN 标记时减小拥塞窗口，将拥塞窗口复位到其初始值（常导致慢启动）但不改变 ssthresh_。
Functions for Processing ACKs

recv() 是 ACK 的主要接收路径。注：只用到信息的单向传输。故该方法只有纯 ACK 包（即无数据）到来时才会被调用。它在 ts_peer_中保存了来向 ACK 的时间标记（timestamp），检测 ECN 比特（在合适的时候缩小发送窗口）。如果 ACK 是新的，则调用 newack(),否则检查他是否为前一个 ACK 的重复。若如此，它则进入快速重传，关闭窗口，复位重传计时器，调用 send_much 发送包。

newack() 处理新 ACK（新 ACK 要具有比此前 ACK 号更大号的序号）。它调用 newtimer()设立新的重传计时器，调用 rtt_update 来更新 RTT 的估值。更新最高的前一 ACK 序号。

Functions for Managing the Retransmission Timer

这些函数有两种用途：估算往返时延，设置重传计时器。

rtt_init 把 srtt_和 rtt_初始化为零。把 rttvar_设立为 3/tcp_tick_把 backoff multiplier 设为 1。

rtt_timeout 此函数给出了用于安排下一重传计时器的间隔时间(秒)，这个值是基于对当前往返时延时间平均值和偏差的估算得到的。

rtt_update 以测量的 RTT 为参数，并根据上述所介绍计算平均值和偏差。注：t_srtt_和 t_rttvar 均存储于固定点（整数）。它们分别占有 3 个和 2 个比特。

reset_rtx_timer 快速重传或超时的时候调用此函数，它通过调用 ser_rtx_timer 来设置重传计时器，如果是因为超时被调用的则还调用 rtt_backoff。

rtt_backoff 此函数通过倍增重传计时器。

newtimer 只有在新 ACK 到达的时候才调用，如果发送器的左窗边缘超出了 ACK，则调用 set_rtx_timer,否则如果重传计时器未定则取消。

34.5 Tracing TCP Dynamics

TCP 的行为通常是通过构建时间-序号图形来观测的。一般而言，trace 是通过激活 TCP 包所通过的链路上的 tracing 来实现的。现存的两个 trace 方法：默认的（用于追踪 TCP agent）和扩展了的（仅用于 FullTCP）。

34.6 One-Way Trace TCP Trace Dynamics

由 TCP agent 产生发送到 TCP Sink 的 TCP 包通过一个 traced link(见第 26 章)时会产生格式如下的 trace 文件：

```
+ 0.94176 2 3 tcp 1000 ------ 0 0.0 3.0 25 40
+ 0.94276 2 3 tcp 1000 ------ 0 0.0 3.0 26 41
d 0.94276 2 3 tcp 1000 ------ 0 0.0 3.0 26 41
+ 0.95072 2 0 ack 40 ------ 0 3.0 0.0 14 29
- 0.95072 2 0 ack 40 ------ 0 3.0 0.0 14 29
- 0.95176 2 3 tcp 1000 ------ 0 0.0 3.0 21 36
+ 0.95176 2 3 tcp 1000 ------ 0 0.0 3.0 27 42
```

在 26.4 节给出了这个 trace 文件的确切格式。在 tracing TCP 的时候 tcp 或 ack 包是相关的。其类型、大小、序列号（ack 包的 ack 号）、到达/出发/丢包时间分别在位置 5、6、11 和 2 处给出；“+”表示信息包的到达；“d”表示丢包；“-”表示离开。有些脚本程序处理这些文件以产生图形输出或统计结果(例如：见文件~ns/test-suit.tcl 中的程序 finish)。

34.7 One-Way Trace TCP Trace Dynamics

FullTcp 所产生的 并通过 Traced 链路的 TCP 包 包含了一些附加的信息。这些信息在默认情况下未显示的(using 常规的 trace 对象)。通过设置 trace 对象中的标志符 show_tcphdr_(见 refsec:tracmat 这一节) 向 trace 文件中写入 3 个其它的 header 域：ack 号、tcp 指定标志、header 长度。

34.8 命令一览

在仿真中下列命令用于建立操纵 tcp 流：

```
set tcp0 [new Agent/TCP]
```

这个命令用于闯将一个 TCP Agent，有多种形式的 TCP 发送端和接收端在 ns 中实现，TCP 发送端有：Agent/TCP, Agent/TCP/Reno, Agent/TCP/Newreno, Agent/TCP/Sack1, Agent/TCP/Vegas, Agent/TCP/Fack。

TCP 接收端有：Agent/TCPSink, , Agent/TCPSink/DelAck , Agent/TCPSink/Sack1 , Agent/TCPSink /Sack1/DelAck。

双向 TCP agent 有：Agent/TCP/FullTcp，他们的不同请参看本章的前面。

TCP 流的结构参数可设置如下：

```
$tcp set window_<wnd-size>
```

在 34.1.4 中有所有的可能的 TCP 结构参数，缺省的结构参数见文件~ns/tcl/lib /ns-default.tcl。

以下是一个简单的建立 TCP 建立的例子：

```
set tcp [new Agent/TCP] ;# create tcp agent
$ns_ attach-agent $node_(s1) $tcp ;# bind src to node
$tcp set fid_ 0 ;# set flow ID field
set ftp [new Application/FTP] ;# create ftp traffic
$ftp attach-agent $tcp ;# bind ftp traffic to tcp agent
set sink [new Agent/TCPSink] ;# create tcpsink agent
$ns_ attach-agent $node_(k1) $sink ;# bind sink to node
$sink set fid_ 0 ;# set flow ID field
$ns_ connect $ftp $sink ;# active connection src to sink
$ns_ at $start-time "$ftp start" ;# start ftp flow
```

建立 full-tcp 的例子请参看 34.3.1。

第 35 章

SCTP代理

本章描述由特拉华（Delaware）州立大学的协议工程实验室为ns开发的SCTP代理。SCTP代理均为双向（two-way）代理，这就意味着发送端和接收端是对称的。然而，双向数据还没有实现。SCTP代理的每个实例只是一个发送端或者一个接收端。

SCTP代理在文件~ns/sctp/sctp*.{cc, h}中实现，用于匹配正则表达式。

SCTP目前支持的有：

●Agent/SCTP
-RFC2960+draft-ietf-tsvwg-sctpimpguide-09.txt+draft-ietf-tsvwg-usctp-01.txt

●Agent/SCTP/HbAfterRto – 实验扩展（RTO后的HEARTBEAT）

●Agent/SCTP/MultipleFastRtx –实验扩展（UD PEL的多重快速重传算法）

●Agent/SCTP/Timestamp –实验扩展（时间戳）

●Agent/SCTP/MfrHbAfterRto –结合MultipleFastRtx和HbAfterRto的实验扩展

●Agent/SCTP/MfrTimestamp –结合MultipleFastRtx和Timestamp的实验扩展

35.1节对基类SCTP代理的详细的配置参数和命令作一个简要的概述。35.2节描述了可行的SCTP扩展。分组trace文件中用到的详细的SCTP的trace格式将在35.3节中解释。35.4节解释了怎样使用SCTP的遗传应用，怎样编写探测SCTP全部特征的SCTP感知应用。35.5节提供了单穴和多穴端点的脚本例子。

35.1 基类SCTP代理

基类SCTP代理Agent/SCTP支持RFC2960的下面一些部分的特征，包括提交到draft-ietf-tsvwg-sctpimpguide-13.txt的修正。

5.1 一个联系（Association）的正常建立（实质的握手）
6.1 大块数据的传输
6.2 大块数据接收的确认
6.3 重传计时器的管理
6.4 多穴的SCTP端点
6.5 流标识符与流序列号
6.6 有序与无序分发
6.7 接收到的数据TSNs的报告差距（Report Gaps）
7.2 SCTP的慢启动（Slow-Start）与拥塞避免（Congestion Avoidance）
8.1 端点故障检测（Failure Detection）
8.2 路径故障检测
8.3 路径Heartbeat（不受上层控制）

该代理同时支持draft-ietf-tsvwg-usctp-01.txt的局部可靠性扩展。

联系建立 SCTP代理用一个四路握手（four-way handshake）建立一个association，但是这个握手被简化了，并没有严格的遵照RFC2960。这个握手并不交换标签，而INIT和COOKIEECHO块也未用于更新RTT。取而代之的是，RTT估计（estimation）以第一个数据块开始。

联系关闭 目前，SCTP代理没有完成一个正确的关闭（shutdown）。当模拟的连接结束的时候，联系会突然终止。未来的发行版中将会加入shutdown过程。

多穴（Multihoming） ns-2的下部构造（underlying infrastructure）并不为一个单一的节点提供多接口支持。为了突破这个限制，我们的方法为有多穴传输层的逻辑多穴节点提供全面的支持，例如SCTP。每个多穴节点实际上由不止一个节点组成。如图35.1所示，一个逻辑多穴节点由一个单一的核心节点（core node）和多个接口节点（interface node）组成，每个节点对应一个模拟接口。核心节点通过一个单向链路连接到每个接口节点上，这条链路由核心节点指向接口节点，但是流量从未经过这些链路。这些链路只适合用于为核心节点提供路由决定。一个SCTP代理同时附着在这些节点上（例如，核心节点和接口节点），但是实际流量仅由接口节点流入或流出。只要SCTP代理需要发送数据到目的地，并且不知道使用哪个输出接口，代理首先与核心节点协商一个路由查询（lookup）。然后，SCTP代理从适当的接口节点执行发送动作。输入的数据直接被一个接口节点接收，并被传递到SCTP代理上。这种方法适用于ns-2中要求多穴功能的任何传输协议。注意：用户必须使用35.1.2节中的命令来配置多穴节点（例子如35.5.2节中所示）。

分组编号与TSN编号方式（TSN Numbering） 当ns开始对分组从0开始编号时，SCTP模块则从1开始对DATA块TSNs进行编号，并分配未定义的TSN值（－1）用来控制块（例如，INIT，SACK，HEARTBEAT，等等）。联系建立过程中交换的四个分组计入分组枚举中，但未在图表中表示出。当做一些事如为ErrorModel对象指定一个drop链表时，这种信息是很重要的。例如，编号为2的分组实际上指的是第1个有DATA块的SCTP分组。

35.1.1 配置的参数

SCTP支持一些配置变量，这些变量都是TCL可绑定的。本小节中描述的每个变量都既是一个类变量，又是一个实例变量。改变类变量会改变随后创建的所有代理的缺省值。改变一个特殊代理的实例变量却只会影响那个代理使用到的值。例如，

```
Agent/SCTP set pathMaxRetrans_ 5                    ;# 改变类变量
$sctp set pathMaxRetrans_ 5                         ;# 仅改变$sctp对象的pathMaxRetrans_值
```

对于每个SCTP代理，缺省参数为：

```
Agent/SCTP set debugMask_ 0          ;# 模块拴牢调试控制的32位掩码（见解释）
Agent/SCTP set debugFileIndex_ -1    ;# 指定调试输出文件（见解释）
Agent/SCTP set associationMaxRetrans_ 10    ;# RFC2960的Association.Max.Retrans
Agent/SCTP set pathMaxRetrans_ 5       ;# RFC2960的Path.Max.Retrans
Agent/SCTP set changePrimaryThresh_ -1    ;# 如果error计数超过域值(缺省为无穷大)，则改变Primary
Agent/SCTP set maxInitRetransmits_ 8            ;# RFC2960的Max.Init.Retransmits
Agent/SCTP set oneHeartbeatTimer_ 1          ;# 为所有的目的地拴牢一个HB计时器
Agent/SCTP set heartbeatInterval_ 30       ;# RFC2960的HB.interval，单位为秒
Agent/SCTP set mtu_ 1500                    ;# MTU的值，单位为字节，包括IP头
Agent/SCTP set initialRwnd_ 65536       ;# 初始的接收端窗口，单位为字节（在接收端设置）
Agent/SCTP set initialSsthresh_ 65536          ;# 初始的ssthresh值，字节为单位
Agent/SCTP set initialCwnd_ 2                ;# 初始的cwnd，若干的(MTU - SCTP/IP头)
Agent/SCTP set initialRto_ 3.0               ;# RTO缺省的初始值 = 3秒
Agent/SCTP set minRto_ 1.0                   ;# RTO缺省的最小值= 1秒
```

```
Agent/SCTP set maxRto_ 60.0                ;# RTO缺省的最大值 = 60秒
Agent/SCTP set fastRtxTrigger_ 4           ;# 4个缺失的报告触发快速rtx
Agent/SCTP set numOutStreams_ 1            ;# 输出流的条数
Agent/SCTP set numUnrelStreams_ 0          ;# 部分可靠流的条数（从0条开始聚合）
Agent/SCTP set reliability_ 0              ;# 所有部分可靠流的k-rtx值
Agent/SCTP set unordered_ 0                ;# 拴牢所有块是有序的/无序的
Agent/SCTP set ipHeaderSize_ 20            ;# IP头的大小
Agent/SCTP set dataChunkSize_ 1468         ;# 包括数据块头和4个字节边界限制
Agent/SCTP set useDelayedSacks_ 1    ;# delayed sack算法的拴牢开/关（在接收端设置）
Agent/SCTP set sackDelay_ 0.200            ;# rfc2960推荐为200毫秒
Agent/SCTP set useMaxBurst_ 1              ;# max burst的拴牢开/关
Agent/SCTP set rtxToAlt_ 1         ;# rtxs的发送目的地，0为same，1为alt，2为快速rtx到same加上超时到alt
Agent/SCTP set dormantAction_ 0    ;# 0=改变dest, 1=使用primary, 2=在静止前使用上一个dest
Agent/SCTP set routeCalcDelay_ 0   ;# 计算一条路由的时间(见解释)
Agent/SCTP set routeCacheLifetime_ 1.2     ;# 一条路由保存缓存的时间（见解释）
Agent/SCTP set trace_all_ 0        ;# 在一个trace事件中打印出所有变量开/关
```

参数debugMask_是一个32位掩码，用来为特殊的函数打开/关闭调试。参见~ns/sctp/sctpDebug.h中的比特掩码的映射。一个-1可以被用来清空所有的比特位，0被用来关闭所有的调试。如果debug_（标准的ns的调试标识符）设定为1，那么debugMask_中所有的比特位都被设定。注意：ns必须打开-DDEBUG选项编译才能工作。

参数debugFileIndex_是一个整型，用来指定由一个SCTP代理用来调试输出的文件。一个SCTP代理的每个实例都能独立地向不同的文件输出调试信息。例如，数据发送器能将调试日志输出到一个文件中，而接收器则将日志输出到另一个文件中。如果debugFileIndex_置为0，用到的文件名将被命令为debug.SctpAgent.0。如果置为 - 1，那么调试输出被送到stderr中。为了避免混乱，两个SCTP代理不应发送调试输出到同一个文件中。缺省值为 - 1。注意：ns必须打开-DDEBUG选项编译才能工作。

涉及排序（ordering）和可靠性（reliability）选项的配置参数可以被一个SCTP感知（SCTP-aware）应用控制（见35.4节）。

参数routeCalcDelay_和routeCacheLifetime_只用来可选地模拟MANET中的reactive路由协议的高架（overhead），并没有实际模拟一个MANET。（如果你要实际模拟一个MANET，不要用这一个特征）routeCalcDelay_的缺省设置为0秒，这就意味着该特征是关闭的。routeCacheLifetime_的缺省设置为1.2秒（如果这个特征关闭，则可以忽略），这种有意的设置其稍微大于缺省的最小RTO值，用来避免一个单一的timeout事件后出现的“缓存丢失”（cache miss）。

35.1.2 命令

SCTP提供了某些可用于TCL脚本中的命令：

trace 跟踪给定的变量。这个变量（和相关信息）在每次值改变时打印出来。占用1个参数：

cwnd_ 用于跟踪所有路径的cwnd。

rto_ 用于跟踪所有路径的RTO。

errorCount_ 用于跟踪所有路径的error计数器。

frCount_ 快速重传算法被调用时，用于跟踪重传次数。

mfrCount_ 多重快速重传算法被调用时 ,用于跟踪重传次数。这个跟踪变量只能用于扩展的MultipleFastRtx代理(见35.2.2)。
timeoutCount_ 在所有路径上当一个timeout事件发生时，用于跟踪总次数。
rcdCount_ 在所有路径上当一个路由计算延迟发生时，用于跟踪总次数。

注意：这些跟踪变量的实际值是没有意义的。它们只是用于为潜在的多穴端点跟踪相应的变量。例如，如果一个发送端的对等（peer）端点有两个目的端，这个发送端将同时维护两个cwnd。跟踪变量cwnd_将同时跟踪这两个cwnd。

print 提供跟踪的采样方法。这个命令在每次调用时打印出一个给定的变量（和相关信息）。占用1个参数：上面列出的跟踪变量的其中一种。
set-multihome-core 为多穴端点设定核心节点。占用type节点的1个参数。强制性设定 ,并且每个端点的最多设定一次。
multihome-add-interface 为一个多穴端点添加一个接口。占用type节点的2个参数。参数1是多穴端点的核心节点。参数2是需要添加的接口节点。强制性设定。所有的接口必须在set-multihome-core被调用之后 ,在multihome-attach-agent调用之前添加。
multihome-attach-agent 将一个SCTP代理附在一个多穴端点上。占用2个参数。参数1是核心节点。参数2是SCTP代理。强制性设定。
set-primary-destination 将对等端点的接口节点设定为主要目的端(primary destination)。占用type节点的1个参数。是可选的并可为每个端点设定多次。如果不用这个参数，primary目的端会自动被选取。
force-source 设定分组将从接口节点发送出来。占用type节点的1个参数。是可选的并可为每个节点设定可次。如果不用这个参数，路由会为每个分组自动选取源。

35.2 扩展

35.2.1 HbAfterRto SCTP

HbAfterRto SCTP代理扩展了当前的重传策略（retransmission policy）。除SCTP的当前重传策略在一个timeout期间交替变换（alternate）目的端之外，一个heartbeat被立即发送到timeout事件发生的目的端。多余的heartbeat为发送端提供一种机制，用于频繁更新一个alternate目的端的RTT估计值，因此带来一个更好的基于RTO值的RTT估计值。

例如 ,假设一个分组在向primary目的端传送过程中丢失了 ,随后重传到一个alternate目的端。同样假设重传时间耗尽。丢失的分组会被重传到另一个alternate目的端（如果有一个出现的话；否则，重传到primary目的端）。更为重要的是，当时间耗尽时，一个heartbeat会同时被发送到alternate目的端。如果heartbeat成功地返回确认信息，那个目的端会获得一个额外的RTT方法，用来帮助减少它最近的两倍的RTO[?]。

35.2.2 MultipleFastRtx SCTP

MultipleFastRtx SCTP代理试图将timeout事件发生的次数降到最低。在没有多重快速重传算法的支持下，SCTP可能一次只能快速重传一个TSN。如果一个快速重传的TSN丢失了，一个timeout事件会被再一次的用来重传TSN。多重快速重传算法允许同样的TSN在必要的情况下可以重传多次。如果没有多重快速重传算法的话，巨大的数据窗口可能会产生足够多的SACK，在一个单独的RTT，错误的激发同样的TSN的多次重传。为了避免这些伪造的快速重传，多重快速重传算法为每个需要快速重传的TSN引进了一个fastRtxRecover状态变量。这个变量在一个TSN快速重传时，存储一个最高级的TSN。于是，只有离fastRtxRecover较远的最新确认TSN的SACK才能增加快速重传TSN的丢失（missing）报告。如果再一次达到快速重传TSN的missing报告的阈值（threshold），发送端会有足够的迹象表明这个TSN已经丢失，可以再次快速重传[?]。

35.2.3 Timestamp SCTP

Timestamp SCTP代理将时间戳（timestamps）引进了每个分组中，因此可以使发送端将重新传输与原始传输区别开。通过消除重传的不确定性（ambiguity），Karn的算法能够除去不确定性，并且在alternate路径上成功的重传能够用来更新RTT估计值，使得RTO值更为准确。通过时间戳，发送端可以有更多的样本用于更新alternate目的端的RTT估计值[?]。

35.2.4 MfrHbAfterRto SCTP

MfrHbAfterRto SCTP代理结合了HbAfterRto SCTP代理和MultipleFastRtx SCTP代理的功能。

35.2.5 MfrHbAfterRto SCTP

MfrTimestamp SCTP代理结合了Timestamp SCTP代理和MultipleFastRtx SCTP代理的功能。

35.3 动态跟踪SCTP

SCTP分组由一个SCTP代理产生，并且通过一个跟踪的链路发送到对等的SCTP代理（见26小节），它产生的trace文件的格式如下面各行：

```
+ 0.5 1 4 sctp 56 -------I 0 1.0 4.0 1 -1 4 65535 65535
- 0.5 1 4 sctp 56 -------I 0 1.0 4.0 1 -1 4 65535 65535
r 0.700896 1 4 sctp 56 -------I 0 1.0 4.0 1 -1 4 65535 65535
+ 0.700896 4 1 sctp 56 -------I 0 4.0 1.0 1 -1 5 65535 65535
- 0.700896 4 1 sctp 56 -------I 0 4.0 1.0 1 -1 5 65535 65535
r 0.901792 4 1 sctp 56 -------I 0 4.0 1.0 1 -1 5 65535 65535
+ 0.901792 1 4 sctp 36 -------I 0 1.0 4.0 1 -1 6 65535 65535
- 0.901792 1 4 sctp 36 -------I 0 1.0 4.0 1 -1 6 65535 65535
r 1.102368 1 4 sctp 36 -------I 0 1.0 4.0 1 -1 6 65535 65535
+ 1.102368 4 1 sctp 36 -------I 0 4.0 1.0 1 -1 7 65535 65535
- 1.102368 4 1 sctp 36 -------I 0 4.0 1.0 1 -1 7 65535 65535
r 1.302944 4 1 sctp 36 -------I 0 4.0 1.0 1 -1 7 65535 65535
+ 1.302944 1 4 sctp 1500 -------D 0 1.0 4.0 1 1 8 0 0
- 1.302944 1 4 sctp 1500 -------D 0 1.0 4.0 1 1 8 0 0
+ 1.302944 1 4 sctp 1500 -------D 0 1.0 4.0 1 2 9 0 1
- 1.326624 1 4 sctp 1500 -------D 0 1.0 4.0 1 2 9 0 1
r 1.526624 1 4 sctp 1500 -------D 0 1.0 4.0 1 1 8 0 0
r 1.550304 1 4 sctp 1500 -------D 0 1.0 4.0 1 2 9 0 1
+ 1.550304 4 1 sctp 48 -------S 0 4.0 1.0 1 2 11 65535 65535
- 1.550304 4 1 sctp 48 -------S 0 4.0 1.0 1 2 11 65535 65535
r 1.751072 4 1 sctp 48 -------S 0 4.0 1.0 1 2 11 65535 65535
...
+ 19.302944 4 1 sctp 56 -------H 0 2.0 5.0 1 -1 336 65535 65535
```

- 19.302944 4 1 sctp 56 -------H 0 2.0 5.0 1 -1 336 65535 65535
r 19.303264 4 1 sctp 56 -------H 0 4.0 1.0 1 -1 322 65535 65535
+ 19.303264 1 4 sctp 56 -------B 0 1.0 4.0 1 -1 337 65535 65535
- 19.327584 1 4 sctp 56 -------B 0 1.0 4.0 1 -1 337 65535 65535
r 19.52848 1 4 sctp 56 -------B 0 1.0 4.0 1 -1 337 65535 65535

当跟踪SCTP时，分组的SCTP类型是对应的。它们的分组类型，（数据）块类型，分组大小，TSN（SACK块的CumAck指针），流标识，SSN，以及到达/离开/丢弃时间分别为字段的位置5,7（标识符位置8）,6，12,14,15和2给定。如果一个分组的chunk不止一个，会为每个chunk打印出一行。将来的发行版本中应该包括一个指示一行代表哪个分组的chunk的字段（例如，2:3可以用来表示一个包含3个chunk的分组的第2个chunk）。既然控制chunk没有使用TSN，流标识，SSN，这些chunk的trace行便可以使用未定义的数字（ - 1或是65535）来表示这些字段。 + 表示分组的到达，d表示分组的丢失，-a表示分组的离开。

字段7的标识符位置8表明chunk的类型如下：I标识表示控制chunk初始化的集合（INIT， INIT-ACK, COOKIE-ECHO, COOKIE-ACK）。将来的发行版本中应该将I用于INIT和INIT-ACK chunk中。D，S，H以及B标识符分别表示一个数据块，一个SACK，一个HEARTBEAT chuck以及一个HEARTBEAT-ACK chunk。

许多的脚本对这个文件进行处理，用来产生图表输出或者统计概要（例如，参考~ns/tcl/test/test-suite-sctp.tcl中的finish过程）。

35.4 SCTP应用

SCTP支持对ns应用的继承，但是很明显的是，它们并不能完全使用SCTP的所有特征。对于这些应用，TCL绑定的SCTP配置参数被用来设定可靠性和排序选项。然而，使用这些参数对于每个message这些选项是不受控制的。只有SCTP感知应用可以这样做。ns应用要成为SCTP感知应用可以使用下面的sendmsg API（参考~ns/apps/sctp_app1.{cc, h}作为例子）。

1. 创建并填充AppData_S结构的一个实例（参见~ns/sctp/sctp.h）。The AppData_S结构有如下字段：
usNumStreams 是在协商（negotiation）建立过程中的输出流的数目。虽然这个字段随着每次sendmsg的调用而传递，它却只能用于联系建立的过程中。一旦联系被创建，这个字段就会被忽略。
usNumUnreliable 是不可靠的输出流的数目（现在叫做部分可靠的）。发送端只是简单地将最小的输出流设为不可靠/部分可靠；剩下的都是可靠的。这个字段同样只能用于联系建立的过程中。
usStreamId 是一个message的流标识。
usReliability 是一个message的可靠级别（k-rtx值）。这个字段当message在一个可靠的流上发送时可以忽略。
eUnordered 是一个message的无序的布尔标识符。
uiNumBytes 是一个message中的字节数。

2. 将这个对象传递给SCTP的sendmsg函数的第2个参数。
sctpAgent->sendmsg(numBytes, (char *)appData);

35.5 脚本例子

35.5.1 单穴例子

```
Trace set show_sctphdr_ 1                ;# 需要设定这行来跟踪SCTP分组
set ns [new Simulator]
set allchan [open all.tr w]
$ns trace-all $allchan
proc finish {
exit 0
}
set n0 [$ns node]
set n1 [$ns node]
$ns duplex-link $n0 $n1 .5Mb 200ms DropTail
# 注意：调试文件（在本例中，为debug.SctpAgent.0和debug.SctpAgent.1）包含大量有用＃信息。它们可以被用来trace所有分组的发送，接收和处理。
set sctp0 [new Agent/SCTP]
$ns attach-agent $n0 $sctp0
$sctp0 set debugMask_ 0x00303000          ;# 参考sctpDebug.h设定掩码映射
$sctp0 set debugFileIndex_ 0
set trace_ch [open trace.sctp w]
$sctp0 set trace_all_ 0                   ;# 不去trace一行中的所有变量
$sctp0 trace cwnd_                        ;# trace所有目的端的cwnd
$sctp0 attach $trace_ch
set sctp1 [new Agent/SCTP]
$ns attach-agent $n1 $sctp1
$sctp1 set debugMask_ -1                  ;# 使用-1来打开所有的调试
$sctp1 set debugFileIndex_ 1
$ns connect $sctp0 $sctp1
set ftp0 [new Application/FTP]
$ftp0 attach-agent $sctp0
$ns at 0.5 "$ftp0 start"
$ns at 4.5 "$ftp0 stop"
$ns at 5.0 "finish"
$ns run
```

35.5.2 多穴例子

这个例子展示了多穴的情况。两个SCTP端点，分别有2个接口，每对端口之间直接连接。在＃联系的中间， primary（目的端）有一个改变。运行nam可以帮助用来可视化多穴体系结构。

```
#                host0_if0 O===========O host1_if0
#                       /                       \
#     host0_core O                                O host1_core
#                       \                       /
#                host0_if1 O===========O host1_if1
Trace set show_sctphdr_ 1
set ns [new Simulator]
set nf [open sctp.nam w]
$ns namtrace-all $nf
set allchan [open all.tr w]
$ns trace-all $allchan
proc finish {
exec nam sctp.nam &
exit 0
}
set host0_core [$ns node]
set host0_if0 [$ns node]
set host0_if1 [$ns node]
$host0_core color Red
$host0_if0 color Red
$host0_if1 color Red
$ns multihome-add-interface $host0_core $host0_if0
$ns multihome-add-interface $host0_core $host0_if1
set host1_core [$ns node]
set host1_if0 [$ns node]
set host1_if1 [$ns node]
$host1_core color Blue
$host1_if0 color Blue
$host1_if1 color Blue
$ns multihome-add-interface $host1_core $host1_if0
$ns multihome-add-interface $host1_core $host1_if1
$ns duplex-link $host0_if0 $host1_if0 .5Mb 200ms DropTail
$ns duplex-link $host0_if1 $host1_if1 .5Mb 200ms DropTail
set sctp0 [new Agent/SCTP]
$ns multihome-attach-agent $host0_core $sctp0
set trace_ch [open trace.sctp w]
$sctp0 set trace_all_ 1                    ;# 在一个单一的trace事件上打印出所有
$sctp0 trace rto_
$sctp0 trace errorCount_
$sctp0 attach $trace_ch
set sctp1 [new Agent/SCTP]
$ns multihome-attach-agent $host1_core $sctp1
$ns connect $sctp0 $sctp1
set ftp0 [new Application/FTP]
```

```
$ftp0 attach-agent $sctp0
$sctp0 set-primary-destination $host1_if0            ;#  在联系启动前设定primary目的端
$ns at 7.5 "$sctp0 set-primary-destination $host1_if1"                ;#  改变primary目的端
$ns at 7.5 "$sctp0 print cwnd_" ;# print all dests’  cwnds at time 7.5
$ns at 0.5 "$ftp0 start"
$ns at 9.5 "$ftp0 stop"
$ns at 10.0 "finish"
$ns run
```

第 36 章

Agent/SRM

本章介绍了ns中SRM实现的内部。本章包括三个部分：第一部分概述了最简SRM配置，并对基本SRM代理的配置参数作了“全面”描述。第二部分介绍了基本SRM代理的结构（architecture），内部（internals）以及代码路径。本章的最后一部分介绍了截止目前为止对其它类型SRM代理所作的扩展。

本章所介绍的过程和函数可以在~ns/tcl/mcast/srm.tcl，~ns/tcl/mcast/srm-adaptive.tcl，~ns/tcl/mcast/srm-nam.tcl, ~ns/tcl/mcast/srm-debug.tcl以及~ns/srm.{cc, h}中找到。

36.1 配置

运行一个SRM模拟需要创建和配置代理，添加一个应用层数据源（一个流量发生器），并启动代理和流量发生器。

36.1.1 琐细的配置

创建代理：

```
set ns [new Simulator]                          ;# 预先的初始化
$ns enableMcast
set node [$ns node]                             ;# 附在节点上的代理
set group [$ns allocaddr]                       ;# 这个代理的多播组（group）
set srm [new Agent/SRM]
$srm set dst_ $group                ;# 配置SRM代理
$ns attach-agent $node $srm
$srm set fid_ 1                     ;# 可选配置
$srm log [open srmStats.tr w]                   ;# 在该文件中生成统计日志
$srm trace [open srmEvents.tr w]                ;# 为该代理跟踪事件
```

配置一个原始的SRM代理的关键步骤是分配它的多播组，并把它连接到节点上。

其它有用的配置参数可以用来给源自此代理的流量分配一个独立的流ID，打开一个log文件以写入统计资料，打开一个trace文件以跟踪数据[22]。

tcl/mcast/srm-nam.tcl文件包含了重载这个代理的send方法的定义；它对源自此代理的控制流量进行分类。每种类型被分配一个独立的流ID。这些控制流量被分为会议消息（session messages）（流标识为40）、请求消息（流标识为41）以及修复消息（流标识为42）。基本流标识的改变可以在使用srm-nam.tcl源之前，通过将全局变量ctrlFid设定为比预定流标识小的值来实现。为了做到这一点，模拟脚本中在创建任何SRM代理前调用srm-nam.tcl源。它可用于流量踪迹的分析，或在nam中可视化。

[22] 注意这个 trace 数据同样能够用于收集一定类型的 trace 数据。我们稍后将予以解释。

应用数据的处理 代理本身并不产生任何应用数据；相反，模拟用户可以将任何流量发生器模拟连接到SRM代理上，用来产生数据。下述代码说明了如何将一个流量发生模块连接在SRM代理上：

```
set packetSize 210
set exp0 [new Application/Traffic/Exponential]          ;# 配置流量发生器
$exp0 set packetSize_ $packetSize
$exp0 set burst_time_ 500ms
$exp0 set idle_time_ 500ms
$exp0 set rate_ 100k
$exp0 attach-agent $srm0                                ;# 将应用连接到代理上
$srm0 set packetSize_ $packetSize                       ;# 产生适当大小的修复分组
$srm0 set tg_ $exp0                                     ;# 指向流量发生器对象的指针
$srm0 set app_fid_ 0                                    ;# 由流量发生器产生的分组的fid值
```

用户可以在一个SRM代理上连接任何流量发生器。SRM代理将添加SRM头，把目的地地址设定为多播组，并将分组分发到其目的地。SRM头包含消息的类型，发送器的标识（identity），消息的序列号，（控制消息），消息被发送的轮数。SRM中的各个数据单元被识别为<发送器的标识，消息序列号>。

SRM代理本身并不产生数据；它也不保持对发送出的数据的跟踪，除了记录在需要做错误修复的事件中接收到的消息的序列号。既然这个代理并不真正记录过去的数据，它需要知道传送每条修复消息的分组的大小。因此，实例变量packetSize_指定了代理所产生的修复消息的大小。

启动代理和流量发生器 代理和流量发生器的启动必须分别执行：

```
$srm start
$exp0 start
```

或者，也可以从SRM代理来启动流量发生器：

```
$srm0 start-source
```

启动时，代理连接多播组，开始产生session消息。start-source触发流量发生器来发送数据。

36.1.2 其它配置参数

除了上述参数外，SRM代理还支持其它的结构变量。本节所介绍的各个变量既是OTcl类变量，同时又是OTcl对象的实例变量。类变量的改变将会引起随后所创建的所有代理的缺省变量值的变化。而一个特定的代理的实例变量的变化只影响该代理用到的值。例如：

```
Agent/SRM set D1_ 2.0                                   ;# 改变类变量
$srm set D1_ 2.0                ;# 为特定的$srm对象改变D1_
```

其中每个SRM代理默认的请求和修复定时器参数为：

```
Agent/SRM set C1_ 2.0                                   ;# 请求参数
Agent/SRM set C2_ 2.0
Agent/SRM set D1_ 1.0                                       ;# 修复参数
Agent/SRM set D2_ 1.0
```

因而基于代理是否通过使用下述定义来利用可能性或确定性的抑制可以得到两种具体的SRM代理：

```
Class Agent/SRM/Deterministic -superclass Agent/SRM
Agent/SRM/Deterministic set C2_ 0.0
Agent/SRM/Deterministic set D2_ 0.0
Class Agent/SRM/Probabilistic -superclass Agent/SRM
Agent/SRM/Probabilistic set C1_ 0.0
Agent/SRM/Probabilistic set D1_ 0.0
```

在后面的一小节（36.7小节）里，我们将介绍其它扩展SRM的方法。

与定时器相关的函数是由属于SRM类的对象来单独处理的。对于丢报恢复和周期性发送session消息而言，定时器是必需的。可以用来发送请求修复信息的丢包恢复对象有两个。这个代理分别创建请求或修复对象来处理丢包。相反的，这个代理只创建一个session对象来发送周期性session消息。实现每个功能的缺省类为：

```
Agent/SRM set requestFunction_ "SRM/request"
Agent/SRM set repairFunction_ "SRM/repair"
Agent/SRM set sessionFunction_ "SRM/session"
Agent/SRM set requestBackoffLimit_ 5          ;# requestFunction_的参数
Agent/SRM set sessionDelay_ 1.0               ;# sessionFunction_的参数
```

实例过程requestFunction{}, repairFunction{}以及sessionFunction{}可用于改变个别代理的默认函数。最后两行是request和session对象所用的特定参数。下面的小节（36.2节）较详细介绍了这些对象的实现。

36.1.3 统计资料（Statisties）

每个代理都有两组统计资料，即测量对数据丢失的响应的统计资料和每个请求/修复的总体统计资料。此外，还有访问源自这个代理的其它信息的方法。

数据丢失：测量对数据丢失的响应的统计资料，跟踪了duplicate请求（修复）及平均请求（修复）时延。所用算法见Floyd etal（9）。在此算法中，每个新请求（修复）启动一个新的请求（修复）周期，在请求（修复）周期中，直到此轮结束，代理才根据所接收到的请求（修复），或根据所发送请求（修复）测量第一轮duplicate请求（修复）的数目。下列代码说明了用户怎样简单恢复代理中的当前值：

```
set statsList [$srm array get statistics_]
array set statsArray [$srm array get statistics_]
```

第一种形式返回了键值对（key-value pair）列表；第二种形式将列表装入startsArray以进行进一步处理。此数组的关键字为dup-req, ave-dup-req, req-delay, ave-req-delay,dup-rep, ave-dup-rep,rep-delay以及ave-rep-delay.Each。

总体统计资料：此外，每个loss recovery和session 对象都跟踪时间和统计值，尤其每个对象都记录了与其相关的startTime、serviceTime、distance。startTime 是对象创建时间，serviceTime 是对象完成任务所需时间，distance是到达远程对等端的单向时间。

对request对象而言，startTime是检测到丢包的时间，serviseTime是最终接收到分组的时间，distance是到分组的初始发送器距离；而对repair对象而言，startTime是收到重传请求的时间，serviseTime是发送修复信息的时间，distance是到最初请求器的距离。对这两种类型的对象而言，serviceTime都使用distance来正常化；对session对象而言，startTime是代理连接到多播组的时间。serviceTime和distance是不相关的。

每个对象都同时维护此类对象特有的统计资料。request对象跟踪了所收到的duplicate请求和duplicate修复的数目、所发送的请求的数目，以及在最终接收数据之前此对象必须退避（backoff）的时间。repair对象跟踪了duplicate请求和duplicate修复的数目，以及此对象是否为这个代理发送修复信息。

session对象仅简单记录所发送session消息的数目。

每当对象结束其规定的错误恢复函数的时候，每个对象的定时器及统计值均被写入log文件。这各trace 文件格式如下：

<prefix> <id> <times> <stats>

其中

<prefix>为<time> n <node id> m <msg id> r <round>

<msg id>可以表示成<source id:sequence number>

<id>为（对象的）类型

<times>是startTime, serviceTime, distance键值对的一个列表

<stats>是每个对象统计资料的键值对的列表

dupRQST, dupREPR, #request对象的发送，退避

dupRQST, dupREPR, #repair对象的发送

#session对象的发送

下面的抽样输出说明了输出文件的格式（每行已经折叠过来适应页面）：

```
3.6274 n 0 m <1:1> r 1 type repair serviceTime 0.500222 \
startTime 3.5853553333333332 distance 0.0105 #sent 1 dupREPR 0 dupRQST 0
3.6417 n 1 m <1:1> r 2 type request serviceTime 2.66406 \
startTime 3.5542666666666665 distance 0.0105 backoff 1 #sent 1 dupREPR 0 dupRQST 0
3.6876 n 2 m <1:1> r 2 type request serviceTime 1.33406 \
startTime 3.5685333333333333 distance 0.021 backoff 1 #sent 0 dupREPR 0 dupRQST 0
3.7349 n 3 m <1:1> r 2 type request serviceTime 0.876812 \
startTime 3.5828000000000002 distance 0.032 backoff 1 #sent 0 dupREPR 0 dupRQST 0
3.7793 n 5 m <1:1> r 2 type request serviceTime 0.669063 \
startTime 3.5970666666666671 distance 0.042 backoff 1 #sent 0 dupREPR 0 dupRQST 0
3.7808 n 4 m <1:1> r 2 type request serviceTime 0.661192 \
startTime 3.5970666666666671 distance 0.0425 backoff 1 #sent 0 dupREPR 0 dupRQST 0
```

混杂信息（Miscellaneous Information） 最后，用户可以用下列方法来收集这个代理的其它信息：

- groupSize?{}返回对代理的当前多播组大小的估计值
- distances?{}返回distance的键值对列表，key是代理的地址，value是到代理的距离估计值。第一个元素是这个代理的地址，并且其距离为0。
- distance?{}返回到参数指定的代理的距离。任何模拟中的缺省起始距离值为1。

```
$srm(i) groupSize?          ;# 返回对$srm(i)的组大小的估计值
$srm(i) distances?          ;# 返回<address, distance>二元组的列表
$srm(i) distance? 257       ;# 在257号地址处返回distance至代理
```

36.1.4 跟踪（Tracing）

每个对象都在其错误恢复时记录用于跟踪对象进程的trace信息。跟踪入口的形式为：

<prefix> <tag> <type of entry> <values>

Prefix已经在前述有关统计值的章节中介绍过了。当tag为Q时，代表request对象；当tag为P时，代表repair对象，当tag为S时，代表session对象。下面按对象列出了跟踪入口类型和参数：

Tag	Type of Object	Other values	Comments
Q	DETECT		
Q	INTERVALS	C1 ⟨C1_⟩ C2 ⟨C2_⟩ dist ⟨distance⟩ i ⟨backoff_⟩	
Q	NTIMER	at ⟨time⟩	Time the request timer will fire
Q	SENDNACK		
Q	NACK	IGNORE-BACKOFF ⟨time⟩	Receive NACK, ignore other NACKs until ⟨time⟩
Q	REPAIR	IGNORES ⟨time⟩	Receive REPAIR, ignore NACKs until ⟨time⟩
Q	DATA		Agent receives data instead of repair. Possibly indicates out of order arrival of data.
P	NACK	from ⟨requester⟩	Receive NACK, initiate repair
P	INTERVALS	D1 ⟨D1_⟩ D2 ⟨D2_⟩ dist ⟨distance⟩	
P	RTIMER	at ⟨time⟩	Time the repair timer will fire
P	SENDREP		
P	REPAIR	IGNORES ⟨time⟩	Receive REPAIR, ignore NACKs until ⟨time⟩
P	DATA		Agent receives data instead of repair. Indicates premature request by an agent.
S	SESSION		logs session message sent

下面给出了单次loss和recovery的一个典型的trace格式：

```
3.5543 n 1 m <1:1> r 0 Q DETECT
3.5543 n 1 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.0105 i 1
3.5543 n 1 m <1:1> r 1 Q NTIMER at 3.57527
3.5685 n 2 m <1:1> r 0 Q DETECT
3.5685 n 2 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.021 i 1
3.5685 n 2 m <1:1> r 1 Q NTIMER at 3.61053
3.5753 n 1 m <1:1> r 1 Q SENDNACK
3.5753 n 1 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.0105 i 2
3.5753 n 1 m <1:1> r 2 Q NTIMER at 3.61727
3.5753 n 1 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.59627
3.5828 n 3 m <1:1> r 0 Q DETECT
3.5828 n 3 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.032 i 1
3.5828 n 3 m <1:1> r 1 Q NTIMER at 3.6468
3.5854 n 0 m <1:1> r 0 P NACK from 257
3.5854 n 0 m <1:1> r 1 P INTERVALS D1 1.0 D2 0.0 d 0.0105
3.5854 n 0 m <1:1> r 1 P RTIMER at 3.59586
3.5886 n 2 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.021 i 2
3.5886 n 2 m <1:1> r 2 Q NTIMER at 3.67262
```

3.5886 n 2 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.63062
3.5959 n 0 m <1:1> r 1 P SENDREP
3.5959 n 0 m <1:1> r 1 P REPAIR IGNORES 3.62736
3.5971 n 4 m <1:1> r 0 Q DETECT
3.5971 n 4 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.0425 i 1
3.5971 n 4 m <1:1> r 1 Q NTIMER at 3.68207
3.5971 n 5 m <1:1> r 0 Q DETECT
3.5971 n 5 m <1:1> r 1 Q INTERVALS C1 2.0 C2 0.0 d 0.042 i 1
3.5971 n 5 m <1:1> r 1 Q NTIMER at 3.68107
3.6029 n 3 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.032 i 2
3.6029 n 3 m <1:1> r 2 Q NTIMER at 3.73089
3.6029 n 3 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.66689
3.6102 n 1 m <1:1> r 2 Q REPAIR IGNORES 3.64171
3.6172 n 4 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.0425 i 2
3.6172 n 4 m <1:1> r 2 Q NTIMER at 3.78715
3.6172 n 4 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.70215
3.6172 n 5 m <1:1> r 2 Q INTERVALS C1 2.0 C2 0.0 d 0.042 i 2
3.6172 n 5 m <1:1> r 2 Q NTIMER at 3.78515
3.6172 n 5 m <1:1> r 2 Q NACK IGNORE-BACKOFF 3.70115
3.6246 n 2 m <1:1> r 2 Q REPAIR IGNORES 3.68756
3.6389 n 3 m <1:1> r 2 Q REPAIR IGNORES 3.73492
3.6533 n 4 m <1:1> r 2 Q REPAIR IGNORES 3.78077
3.6533 n 5 m <1:1> r 2 Q REPAIR IGNORES 3.77927

请求和修复文件的记录由SRM::evTrace{}来完成。然而，常规方法srm/session::evTrace{}忽略了srm::evTrace{}的基类定义，不记录任何内容。为了在logging选项中得到更强的适应性，在个别模拟脚本中可以忽略这些方法。忽略这些方法的一个可能原因就是为了减少所产生的数据量；因而新程序就可以产生压缩的处理过的输出。

注意trace 文件含有足够的信息和细节以导出大部分统计资料。这些统计资料记录于log文件或存储在统计数组中。

36.2 体系结构（Architecture）及内部（Internals）

SRM代理实现把协议功能分成分组处理，丢包恢复（recovery）和会话消息行为（session message activity）。

分组处理包括转发应用数据信息、收发控制信息。这些操作是用C++方法来实现的。错误检测（Error detection）是在C++中根据所收信息进行。但丢包恢复完全是用OTcl程序来实现的。发送处理信息是在C++中完成的；但这些消息何时发送的策略是由OTcl中的实例程序决定的。

我们将首先介绍基于接收信息而进行的C++处理（36.3小节）。错误恢复和session消息的发送涉及基于定时器的处理。代理用一个单独SRM类实现基于定时器的函数。对于每一个丢包而言，代理既可以进行请求处理，也可以进行修复处理。每个代理将对每个丢包实例化一个单独的丢包恢复对象。在接下来的一小节里介绍基于定时器的基本函数和丢包恢复机制（36.5小节）。最后介绍代理周期性地用基于定时器的函数来发送周期性的session消息。

36.3 分组处理：处理接收到的消息（messages）

方法recv()接收4种类型的信息：数据、请求、修复和会话消息。

数据包：代理不产生任何数据消息。用户必须指定一个外部代理来产生流量。方法recv()必须能够区分两种数据：本地所产生的必须发往多播组的数据与来自多播组的需要处理的数据。因此应用代理必须把分组的目的地地址设为0。

对于本地所产生的数据而言，代理添加合适的SRM报头。设置目的地地址为多播组并把分组转发至其目的地。

当接收到来自多播组的数据信息时，recv-data (sender,msgid)就立即更新其状态、标记所收信息<sender,msgid>。如果它检测到丢包，还可能触发请求。此外，如果所收到的消息是一个旧的失常信息，则必然有个有待于清除的请求或修复。此时，编译的对象将调用OTcl实例程序recv-data{sender,msgid}[23]。

目前，接收器并不真正接收应用数据。代理也不存储任何用户数据。它只是产生适当规模的修复消息，其定义于实例变量packetsize_。但是代理假设应用数据置于分组的数据部分，并且用packet_accessdata()指向此应用数据。

请求分组 当接收到请求分组时，recv_rqst(sender,msgid)将会立即检查它是否需要为其它的丢失数据调度请求。若它在了解到源已经产生此数据信息之前就接收到此请求（即请求序列号大于来自此源的前一个数据序列号），则代理推断它已丢失此请求信息和来自上一已知队列号的数据。它在调度所有丢失数据的请求之后返回。另一方面，如果请求队列号小于来自此源的上一已知队列号，则代理可以处于下列3个状态之一：(1).不具备此数据，且请求准备接收；(2).具备此数据，且已经收到一个请求并准备修复之；(3).具备此数据，且修复之。所有这些错误恢复机制都是在OTcl 中实现的。Recv_rqst()调用实例过程recv_rqst{sender,msgid,requester}以进行进一步处理。

修复分组 当接收到修复信息时，recv_repr(sender,msgid)就立即检查它是否需要为其它丢失数据调度请求。如果它了解到源已经产生此数据信息之前就已经收到此修复信息（即修复序列号大于来自此源的前一个数据队列号），则代理推断它将丢失上一个已知队列号与修复队列号之间的所有数据。为所有丢失数据发出请求，在标记这些信息后返回。另一方面，如果请求队列号小于来自此源的上一已知队列号，则代理将会处于下列3个状态之一：(1).不具备此数据，且请求准备接收；(2).具备此数据 ,且已经收到一个请求并准备修复之 ;(3).它具有此数据 ,而且可能在某时刻调度修复。在错误恢复之后 ,它的hold down定时器（等于它与某请求则者距离的3倍）便终止了，此时未决的对象将被清除。在这最后一种情况下，代理将忽略修复。因为任何操作都没有意义。所有这些错误恢复机制都是在OTcl中实现的。recv_repr()调用Recv-repr{sender,msgid}来为特殊分组完成错误恢复状态。

session分组 **当接**收到session消息时，代理就更新所有激活源的序列号，计算到发送代理的瞬时距离。若代理收到一个无序的session消息，则忽略来自此源的所有的旧的session消息。

session消息的处理在Recv-session()中完成。Session消息的格式是〈count of tuples in this message,list of tuples〉。其中tuple表示<发送器的id，来自此源的上一序列号，来自此发送器的前一个session消息发送时间，消息的发送时间>。第一个tuple是关于本地代理[24]的信息。

---

[23] 从技术上看，recv_data()调用了实例过程 recv data<sender><msgid>，然后调用 recv-data{}。这就间接地允许了单独的模拟脚本在必要时可以忽略 recv{}。

[24] 注：session 消息处理的这种实现与 wb 或文献[11]中描述的有细微的差别。从原理上来看，一个代理将它实际接收到的数据散布开来。另一方面，我们的实现只散布每个源的最后一个消息序列号的总和，并且代理知道那个源已经发送。当研究丢失恢复的分割和还原方面时，这是一个限制。有理由期望这部分代码的维护人员在其丰富的消遣时间中将其修复。

36.4 丢包检测（Loss Detection）—SRMinfo类

一个微型的纯C++的封装类跟踪了多种状态信息。组里的每一个成员n_i为组里的其它每个成员使用了一个SRMinfo块。与组成员n_i上n_j有关的SRMinfo对象包含从n_j发出的为n_i所接收的session信息。n_i可通过这个信息计算出到n_j的距离。若n_j处于动态发送数据流量状态，SRMinfo对象也要含有接收数据的信息，包括一比特的向量用以表明所有来自n_j的分组。

代理在其成员变量sip_中为每个组成员维持着一个SRMinfo对象列表，其方法get_state(int sender)将根据相应的发送器返回对象。如果不存在对象，则可能创建它。SRMinfo类有两个访问及设置比特向量的方法，如：
ifReceived(int id) 表明n_i是否收到来自合适的发送器，具有号为id的信息。
setReceived(int id) 设定标明n_i收到来自合适的发送器，具有号为id的特定信息的比特。

访问计时信息的session消息变量是公用的；未提供封装了的方法。这些变量有：

```
int lsess_;                                    /* # 最后接收到的session消息 */
int sendTime_;                                 /* 发送时间 */
int recvTime_;                                 /* 接收时间 */
double distance_;
/* 数据消息 */
int ldata_;                                    /* # of last data msg sent */
```

36.5 丢包恢复（Loss Recovery）对象

在上一小节里，我们介绍了当接收到一条消息后代理的行为。定时器用于控制何时发送什么特定的控制消息。SRM代理用独立的SRM类来进行基于定时器的处理。这一节我们将介绍SRM类的基本知识和丢包恢复对象。下一节将介绍怎样用SRM类发送周期性的session消息。SRM代理将实例化一个对象用于从一个丢失的数据分组中恢复。检测丢包的代理将实例化SRM/request类中的一个对象；接收请求并具有被请求数据的代理将实例化SRM/repair中的一个对象。

请求机制 SRM代理在接收消息时检测丢包状况，并基于所有的消息序列号来推测丢包。因为分组的接收完全是由编译对象来处理的，那么丢包检测也是用C + +方法来实现的。但是丢包恢复则完全是由OTcl中的相应的解释对象的实例过程来完成的。

当任何方法检测到新的丢包的时候，都将调用Agent/SRM::request{}，连同一个丢失消息序列号列表。request{}将为每个丢失的消息创建一个新对象requestFunction_。代理在其pending_对象的数组中存储了对象句柄。数组的关键字是消息的标识符<sender>:<msgid>。

缺省的requestFunction_是SRM/request类。该类的构造函数调用其基类构造函数来初始化模拟的实例（ns_），SRM代理（agent_），trace文件（trace_）以及数组times_。然后用相关组件初始化其statistics_数组。

对set-params{}的一次单独调用为request对象设定实例变量ender_, msgid_, round_。对象通过查询其agent_来决定C1 _ 和C2 _ 。设定到达发送器的距离times_(distance)，并固定其它调度的参数：backoff常量（backoff_），当前的backoff数目（(backoffCtr_）和代理所确定的限制（backoffLimit_）。set-params{}记录trace入口“Q DETECT”。

request{}的最后一步是调度定时器，用来在合适的时候发送实际请求。实例过程SRM/request::schedule{}用

compute-delay{}及其当前的backoff变量来决定时延。对象调度send-request{}，使其在delay秒之后执行。实例变量eventID_存储了指向调度事件的指针。缺省的compute-delay{}函数返回一个平均分布在[C1ds, (C1 + C2)ds]区间的值。其中ds是$times_(distance)的两倍。schedule{}在计算好的时延之后，调度一个发送请求的事件。常规写入trace入口”Q NTIMER at<time>”。

当被定了时的定时器启动时，常规send-request{}发出合适的消息。它调用“$agent_ send request <args>”来发送请求。注意send{}是一个由编译对象的command()方法来执行的instproc-like。然而，为了特定的配置，有可能用指定的实例过程send{}来重载此instproc-like。例如，回想文件tcl/mcast/srm-nam.tcl重载了send{}命令来设定基于所发送消息类型的流标识。send-request{}更新了统计资料，并写入trace入口”Q SENDNACK“。

当代理为出现的等候对象的分组接收到一条控制消息时，这个代理将会把消息传递给某对象以便进行处理。当接收一条特殊分组的请求时，request对象可能会处于下面两种状态之一：将请求视作duplicate品而忽略，或者取消发送事件并在退避定时器之后重新调度发送。如果忽略请求，则它将更新统计资料，写入trace入口”Q NACK dup“。否则，设立一个基于当前delay_估计值的时间，到此时间结束后再忽略进一步的请求。这个时隙由实例变量ignore_来标识。如果对象重新调度其定时器，它将写入trace入口”Q NACKIGNOREBACKOFF <ignore>”。注意重新调度的请求是代理已经被加入到多播组，而且将会收到它所发出的每一条消息的拷贝。

当request对象接收到一个特殊分组的修复消息时，它可能会处于下面两种状态之一：仍然等待修复消息，或者已经收到一个早期的修复消息。如果是前者的话，则将有一个事件被挂起来发送修复消息，而且eventID_也将指向该事件。对象将计算其serviceTime，取消此事件，建立hold-down周期，在这个周期当中，它将忽略其它的请求。在hold-down周期之末，对象将要求其代理将其清除。它将写入trace入口” Q REPAIR IGNORES <ignore>”。另一方面，如果这是一个duplicate的修复消息，对象将更新其统计值，写入trace入口”Q REPAIR dup”。

当对象完成丢包恢复阶段之后，Agent/SRM::clear{}将从pending_对象的数组中删除其数据，并将其放入done_对象的列表当中去。代理将周期性地整理删除done_对象。

修复机制 如果代理接收到一个分组的请求消息，它将初始化一个修复，而且没有这个分组的request对象pending_。缺省的repair对象属于SRM/repair类。除了较小的差别以外，事件序列和类中的实例过程与SRM/request的均相同。我们不一一介绍每个程序，而只是简述它与request对象的区别。

Repair对象使用修复参数D1_，D2 _。它不会重复定时其定时器。因而它不需要使用像request对象所用的任何backoff变量。Repair对象忽略来自相同分组的所有请求。它不使用request对象所使用的ignore_变量。Repair对象所写入的trace入口有明显的不同；它们是“P NACK from <requester>”，“P RTIMER at <fireTime>”，“P SENDREP”，“P REPAIR ES <holddown>”。

除了这些差别以外，repair对象的事件调用顺序与request对象相似。

统计机制 代理和request对象，以及repair对象一起，收集它们对丢失数据[11]响应的统计资料。每次调用request{}过程都要标记一个新的周期。在这个新周期的开始，mark-period{}计算上一个周期duplicate品总数的动态平均值。每当代理收到来自另一个代理的每一轮请求，并且它已经在那一轮中发出请求的时候，于是它便认为这个请求是一个duplicate过的请求，并增大适当的计数器。如果request对象在第一轮中未发出请求，则它不会考虑duplicate这个请求。如果代理的repair对象未就续，由不会考虑那个分组的duplicate请求的到达。此对象的方法SRM/request::dup-request?{}以及SRM/repair::dup-request?{}用代码描述了这些策略，并按要求返回0或1。

Request对象同时也计算检测丢包与第一次请求之间的时间间隔。代理计算此段时间间隔的动态平均值。对象计算它在取消第一轮调度事件之后的时间间隔（或时延）。时延动态平均值的计算是通过调用Agent/SRM::update-ave来实现的。

代理保存了duplicate修复和修复时延的相似统计资料。

代理存储了完成一次丢包恢复的轮数，以保证随后的丢包恢复阶段不计入统计资料。代理在数组old_中存储了一个阶段的路由数目。当实例化一个新的丢包恢复对象时，该对象将使用代理的实例过程round?{}来决定此分组此前一轮丢包恢复阶段的轮数。

36.6 Session对象

就像丢包恢复对象（36.5小节）一样，session对象也是由SRM基类继承而来。但是，与丢包恢复对象不同的是，代理在其生存期内仅仅创建一个session对象。构造函数像之前一样调用其基类构造函数；然后设定其实例变量sessionDelay_。代理在启动之后创建session对象。此时，它还要调用SRM/session::schedule，用于在sessionDelay_秒后发送一个session消息。

当对象发送一个session消息时，它将调度在一段时间之后发送下一个session消息，并更新其统计资料。send-session{}删除trace入口”S SESSION”。

类忽略记录trace入口的实例过程evTrace{}。而SRM/session::evTrace不能够记录session消息的trace入口。

目前已有的session调度策略有两种 :基类中调度每隔一个固定的sessionDelay_时间间隔就发送一次session消息的函数，围绕着一个很小的值不断变化以避免所有节点的所有的代理同步。 SRM/session/logScaledchedules类调度每隔sessionDelay乘以log2(groupSize_)时间间隔后发送一次session消息。因而，session消息发送的频率与group的大小成反比。

代理在固定间隔发送消息的基类缺省为sessionFunction_。

36.7 扩展Agent基类

在有关配置参数的前几节（36.1.2）中，我们表明了如何一般地扩展代理，以得到确定性和可能性的协议行为。在这一节中，我们将介绍如何继承较为复杂的扩展协议以实现固定的或者适应性定时器机制。

36.7.1 固定的定时器（Fixed Timers）

固定定时器机制是在继承类Agent/SRM/Fixed中实现的。其与固定定时器的主要差别就在于其修复参数被设定为log(groupSize_)。因而，在调度repair对象之前，固定定时器代理的repair过程将设定的D1与D2值与group的大小成比例。

36.7.2 适应性定时器（Adaptive Timers）

采用适应性定时器机制的代理在下面三种情况下修改它们的修复参数：(1)每次新的丢包对象被创建时；(2)发送消息的时候；(3)如果与丢包的相对距离比与发送duplicate的距离短，而且收到duplicate的时候。这三种情况都需要扩展代理和loss对象。Agent/SRM/Adaptive类分别用SRM/request/Adaptive类和SRM/repair/Adaptive类来作为request和repair对象。此外，在最后一个条件下还需要扩展报头以告知它们与丢包的距离。代理相应的编译类为ASRMAgent类。

重新计算每个新的loss对象 每次创建新的request对象时，SRM/request/Adaptive::set-params都要调用$agent_ recompute-request-params。Agent方法recompute-request-params()使用关于duplicate和时延的统计资料来修改用于当前和将来进行请求的C1和C2。

类似的，每次创建新的repair对象时，SRM/request/Adaptive::set-params都要调用$agent_ recompute-repair-params。Agent方法recompute-repair-params()使用关于duplicate和时延的统计资料来修改用于当前和将来进行修复的D1和D2。

发送消息 如果loss对象在其第一轮就发送request消息，则实例过程sending-request{}中的代理将减小C1,并将其实例变量closest_（requestor）设为1。

类似的，如果loss对象在其第一轮就发送repair消息，就将调用代理的实例过程sending-repair{}，以用来减小D1，并将closest_（repairor）设为1。

告知距离(Advertising the Distance) 每个代理都必须向它所发出的每一个请求/修复消息中添加附加信息。基类SRMAgent为它所发送的每一个SRM分组调用虚方法addExtendedHeaders()。这个方法在分组传送前，且在添加SRM分组头后被调用。适应性SRM代理重载了addExtendedHeaders()，用来在附加的报头中指定其距离。当发送请求消息时，代理清楚地了解到发送端的id。例如，适应性SRM代理的addExtendedHeaders()定义为：

```
void addExtendedHeaders(Packet* p) {
SRMinfo* sp;
hdr_srm* sh = (hdr_srm*) p->access(off_srm_);
hdr_asrm* seh = (hdr_asrm*) p->access(off_asrm_);
switch (sh->type()) {
case SRM_RQST:
sp = get_state(sh->sender());
seh->distance() = sp->distance_;
break;
...
}
}
```

类似的，每次接收到一个SRM分组时， parseExtendedHeaders()方法就会被调用。它将成员变量pdistance _ 设定为发送消息的对等端（peer）所告知的距离。该成员变量被捆绑于同名的实例变量。因而可以使用合适的实例过程来访问peer距离。相应的适应性SRM代理方法parseExtendedHeaders()很简单：

```
void parseExtendedHeaders(Packet* p) {
hdr_asrm* seh = (hdr_asrm*) p->access(off_asrm_);
pdistance_ = seh->distance();
}
```

最后，适应性SRM代理的扩展报头被定义为struct hdr_asrm。与大多数其它的分组头不同的是，它们是不能够在分组中自动获取的。第一个适应性代理的解释构造函数将向分组格式中添加报头。例如，Agent/SRM/Adaptive代理构造函数的开头部分为：

```
Agent/SRM/Adaptive set done_ 0
Agent/SRM/Adaptive instproc init args {
if ![$class set done_] {
set pm [[Simulator instance] set packetManager_]
TclObject set off_asrm_ [$pm allochdr aSRM]
$class set done_ 1
```

```
}
eval $self next $args
...
}
```

36.8 SRM对象

SRM对象是实现了SRM可靠多播传输协议的agent对象的子类，它们继承了所有常用的agent函数。在36.9小节里将介绍此对象的方法，此对象的**配置参数**有：

packetSize_ 用于修复消息的分组的大小，缺省值为1024。

requestFunction_ 用于重传请求算法，例如设定请求定时器，缺省器是SRM/request。其它可能的请求函数有适应性SRM代码所用的SRM/request/Adaptive。

repairFunction_ 用于产生修复的算法，例如计算修复定时器。缺省值为SRM/repair。其它可能的请求函数有适应性SRM代码所用的SRM/repair/Adaptive。

sessionFunction_ 用于产生session消息的算法，缺省值为SRM/session。

sessionDelay_ session消息之间的基本区间，此区间加上一个小的随机变量以避免session消息的全局同步，用户可根据其特定的模拟需求来调整此变量，缺省值为1.0。

C1_, C2_ 用于控制请求定时器的参数，详见[8]，缺省值C1 _ = C2 _ = 2.0。

D1_, D2_ 用于控制修复定时器的参数，详见[8]，缺省值D1 _ = D2 _ = 1.0。

requestBackoffLimit_ 最大潜在的backoff的数目。缺省值为5。

状态变量有：

stats_ 包含适应性SRM代理所必须的多种统计资料的数组，其中包括：当前请求/修复周期中的duplicate请求/修复，duplicate请求/修复的平均数目，当前请求/修复周期的请求/修复时延，平均请求/修复时延。

SRM/ADAPTIVE OBJECTS SRM/Adaptive对象是SRM对象的子类，它实现了SRM可靠多播传输协议。它们都继承了SRM对象的所有函数，其状态变量有：

(更详细信息，请参考Sally et al [Fall, K., Floyd, S., and Henderson, T., Ns Simulator Tests for Reno FullTCP. URL ftp://ftp.ee.lbl.gov/papers/fulltcp.ps. July 1997.]的SRM论文)

pdistance_ 该变量用于传递远端代理在请求/修复消息中所提供的距离估计值。

D1_, D2_ 除了第一次修复时初始值为log10(group size)之外与SRM代理均相同。

MinC1_, MaxC1_,MinC2_, MaxC2_ C1和C2的最大值，最小值，缺省初始值定义于[8]，它们规定了C1和C2的动态变化范围。

MinD1_, MaxD1_,MinD2_, MaxD2_ D1和D2的最大值，最小值，缺省初始值定义于[8]，它们规定了D1和D2的动态变化范围。

AveDups 平均duplicates的上限。

AveDelay 平均时延的上限。

eps AveDups -当我们调整参数以减少的时候，dups决定了duplicates的下限。

36.9 命令一览

下在是一个在模拟中用于创建/操作SRM代理的命令列表：

set srm0 [new Agent/SRM]
创建了一个SRM代理的实例，除基类以外，还配置了两个扩展了的SRM代理。它们是Agent/SRM/Fixed和Agent/SRM/Adaptive。其扩展详见36.7小节。

ns_ attach-agent <node> <srm-agent>
向给定的<node>添加SRM代理实例。

set grp [Node allocaddr]
$srm set dst_ $grp
它向用mcast地址<grp>表示的多播组分配了一个SRM代理。

SRM代理的配置参数可以设置如下：
$srm set fid_ <flow-id>
$srm set tg_ <traffic-generator-instance>
..等等

要了解所有可能用到的参数及其缺省值，请查询ns/tcl/mcast/srm.tcl和ns/tcl/mcast/srm-adaptive.tcl。
set exp [new Application/Traffic/Exponential]
$exp attach-agent $srm
该命令向SRM代理添加了一个流量发生器（本例中是一个幂函数发生器）。

$srm start; $exp start
这两个命令用于启动SRM代理和流量发生器。注意，SRM代理和流量发生器必须分别启动，或者，可用代理来启动流量发生器，如下：
$srm start-source。
创建SRM代理的简例请参照ns/tcl/ex/srm.tcl文件。

第 37 章

PLM

本章介绍了PLM协议[19]的ns实现。PLM协议的代码用C＋＋和OTcl两种语言共同编写。PLM分组对（Packet Pair）产生器用C＋＋语言编写，而PLM的核心机制（core machinery）则用OTcl语言编写。本章简单来说，包括三个部分：第一部分展示了如何去创建并配置一个PLM会话（session）；第二部分描述了分组对产生器；第三部分描述了PLM协议的体系结构及其内部。在最后这个部分里，我们将介绍每个功能性（functionality）的主要过程（PLM数据源的实例，接收端的实例，分组的接收，损耗的检测，等等）。

本章中介绍的过程，函数均可在下述文件中找到：~ns/plm/cbr-traffic-PP.cc, ~ns/plm/lossmonitor-plm.cc, ~ns/tcl/plm/plm.tcl, ~ns/tcl/plm/plm-ns.tcl, ~ns/tcl/plm/plm-topo.tcl, ~ns/tcl/lib/ns-default.tcl。

37.1 配置

用一个PLM流（只有一个接收器）创建一个简单的场景

这个简单的例子可以不经修改便可运行（更为复杂的场景见文件~ns/tcl/ex/simple-plm.tcl）。

```
set packetSize 500                ;#分组大小(单位为字节)
set plm_debug_flag 2              ;#调试输出
set rates "50e3 50e3 50e3 50e3 50e3"          ;#第一层的速率
set rates_cum [calc_cum $rates]           ;#所有层的累加速率(强制性的)
set level [llength $rates]                ;#层的数目(强制性的)
set Queue_sched_ FQ                       ;#队列的调度
set PP_burst_length 2                     ;#PP脉冲时间（单位为分组个数）
set PP_estimation_length 3    ;#作估计所需要的最小的PP的个数
Class Scenario0 -superclass PLMTopology
Scenario0 instproc init args {
eval $self next $args
$self instvar ns node
$self build_link 1 2 100ms 256Kb          ;#建立一条链路
set addr(1) [$self place_source 1 3]          ;#创建一个PLM数据源
$self place_receiver 2 $addr(1) 5 1               ;#创建一个PLM接收器
#创建一条多播路由
DM set PruneTimeout 1000       ;#所需的PruneTimeout值
set mproto DM
set mrthandle [$ns mrtproto $mproto {} ]
}
set ns [new Simulator -multicast on]          ;#PLM需要多播路由
$ns multicast
$ns namtrace-all [open out.nam w]                     ;#Nam输出
```

```
set scn [new Scenario0 $ns]                ;#场景的调用
$ns at 20 "exit 0"
$ns run
```

这个例子中引进了几个变量。它们都必须在模拟脚本中设定（这些变量都没有缺省值）。下面两行是强制性的，一定不能漏掉：

```
set rates_cum [calc_cum $rates]
set level [llength $rates]
```

现在我们来详细地介绍各个变量：

packetSize 代表由PLM数据源发送的分组的大小，单位为字节。

plm_debug_flag 代表调试输出的详细层，从0到3,0为无输出，3为全部输出。当plm_debug_flag设为3(全部输出)时，会产生与nam可视化不相兼容的长的行输出。

rates 是指定每层带宽的一个列表（这不是累积带宽！）

rates_cum 是指定层的累积带宽的列表：rates_cum的第一个元素是第一层的带宽，第二个元素是第一层和第二层带宽之和，依此类推。calc_cum{}过程用于计算累积rates。

level 是层的数目。

Queue_sched_ 代表了队列的调度。它用于PLMTopology instproc build_link。PLM需要FQ调度或改变。

PP_burst_length 代表了数据包中突然闯入的分组对（Packet Pair）的大小。

PP_estimation_length 代表了为了计算一个估计值（见37.3.3小节），所需要的分组对的最小个数。

PLM的所有模拟都应该在PLMTopology环境下建立（正如我们定义所谓Scenario0的超类PLMTopology的脚本中所示）。用户接口如下所示（所有的instproc可以在~ns/tcl/plm/plm-topo.tcl中找到）：

build_link a b d bw 创建了节点a和节点b之间的一条双向链路，其延迟为d，带宽为bw。如果这两个节点没有出现，build_link将会创建它。

place_source n t 在节点n上创建并放置了PLM数据源，并在时间为t时启动它。place_source返回addr以允许将接收器和该数据源连接起来。

place_receiver n addr C nb 在节点n上创建并放置一个PLM接收器，并将它与返回地址addr的数据源连接起来。这个PLM接收器的检验值为C。一个可选参数nb用于获得一个PLM接收器的实例，该实例只用于取得与该接收器相关的一些特定的统计资料（主要是接收或丢失的分组数目）。

37.2 分组对（Packet Pair）数据源发生器

本小节介绍了分组对数据源发生器；相关的文件为：~ns/plm/cbr-traffic-PP.cc, ~ns/tcl/lib/nsdefault.tcl。PP数据源的OTcl类名为Application/Traffic/CBR_PP。分组头（PP）数据源发生器在文件~ns/plm/cbr-traffic-PP.cc中。这个数据源发生器由CBR数据源发生器变化而来，CBR数据源发生器见~ns/cbr_traffic.cc。我们仅仅介绍CBR数据源和PP数据源代码之间的显著差异。PP数据源发生器和CBR数据源发生器一样，其缺省值在~ns/tcl/lib/ns-default.tcl中。我们需要为PP数据源发生器设定一个新的参数PBM _ ：

```
Application/Traffic/CBR_PP set PBM_ 2          ;#缺省值
```

OTcl instvar绑定的变量PBM_(C + + 中与OTcl中同名)指定了被分送的背靠背（back-to-back）分组的数目。当PBM _ = 1时，我们用CBR数据源，当PBM _ = 2时，我们用PP数据源（一种背靠背发送两个分组的数据源），等等。PP数据源的平均速率是rate_，但分组却以PBM _ 脉冲分组发送。注意我们也使用术语PP数据源和PP脉冲来表示PBM_>2。我们这样计算next_interval：

```
double CBR_PP_Traffic::next_interval(int& size)

/*(PP_ - 1)是当前脉冲顺分组的个数*/
if (PP_ >= (PBM_ - 1))
interval_ = PBM_*(double)(size_ << 3)/(double)rate_;
PP_ = 0;
else
interval_ = 1e-100; //zero
PP_ += 1 ;
...
```

timeout{}方法将NEW_BURST标志插入一个脉冲的第一个数据包中。这有助于PLM协议来识别出一个PP脉冲的开头。

```
void CBR_PP_Traffic::timeout()
...
if (PP_ == 0)
agent_->sendmsg(size_, "NEW_BURST");
else
agent_->sendmsg(size_);
...
```

37.3 PLM协议的体系结构

PLM协议的代码分布在三个文件当中：~ns/tcl/plm/plm.tcl，包含了没有ns中任何特定接口的PLM协议机制；~ns/tcl/plm/plm-ns.tcl ,包含了特定的ns接口。然而 ,我们并不保证这些接口的严格性和有效性 ;~ns/tcl/plm/plm-topo.tcl，包含一个用户接口，用于PLM流模拟场景的建立。下面我们不去讨论每个对象不同的过程（例如：PLM类所有的instproc），而去讨论每个功能（functionality）（例如：涉及一个PLM接收器的不同类中的instproc）不同的过程。对于一个给定的功能，我们不去详细解释所有相关的代码，而是给出主要的步骤。

37.3.1 PLM数据源（Source）的实例化

为了创建一个PLM数据源，将其放在节点n上，并在t_0时启动它，我们可以调用PLMTopology instproc place_source n t_0。这个instproc返回一个将接收器连接在该数据源上所需的地址addr。place_source调用Simulator instproc PLMbuild_source_set用来创建与层（下面我们将Application/Traffic/CBR_PP类的一个实例称作一层）一样多的Application/Traffic/CBR_PP实例。每一层对应一个不同的多播组。

为了在PLM数据源启动时，能够加速模拟，我们可以使用如下的手段：在t=0时刻，PLMbuild_source_set限定每一层只能发送一个数据包（将maxpkts_设为1）。这就使得我们可以建立多播树（每一层都有一个多播树），而不用洪泛整个网络。事实上，每一层只需要发送一个数据包就可以建立相应的多播树。

多播树在建立时占用最多最大化的RTT值，而且必须在t_0时刻之前建立，其中t_0为PLM数据源的启动时间。因此，t_0值必须谨慎地选取，否则PLM数据源就会发送大量无用的数据包。然而，当我们需要PLM数据源在多播树建立以后启动，t_0可以被过高的估计。在t_0时刻，我们将每层的maxpkts_设定为268435456。

基本上，为了有一个稳固的多播树，prune timeout应该设定为一个很大的值，以DM路由为例：

DM set PruneTimeout 1000

同样的PLM数据源的每一层有一个同样的流标识。因此，每个PLM数据源被看作是惟一流的公平队列调度器。PLM代码自动管理fid_以阻止不同的数据源有相同的fid_。fid_在第一个数据源启动时为1，然后随着每个新数据源的建立而依次增加1。小心避免其它流（例如：当前的TCP流）有相同的fid_。如果你觉得fid_会大于32，别忘了将~ns/fq.cc（MAXFLOW必须被设为模拟中最大的fid_）中的MAXFLOW值增加。

37.3.2 PLM接收器的实例化

所有的PLM机制均在接收端实现。在这一小节里，我们将介绍一个接收器的实例化。为了创建这个接收器，放在节点n上，连接在数据源S上，并在t_1时刻启动，我们调用PLMTopology instproc build_receiver n addr t1 C，其中addr为S创建时，由place_source返回的地址，C为校验值。由build_receiver创建的接收器是PLM/ns类的一个实例，PLM机制的ns接口。在接收器的初始化中，PLM instproc init由于继承关系而被调用。

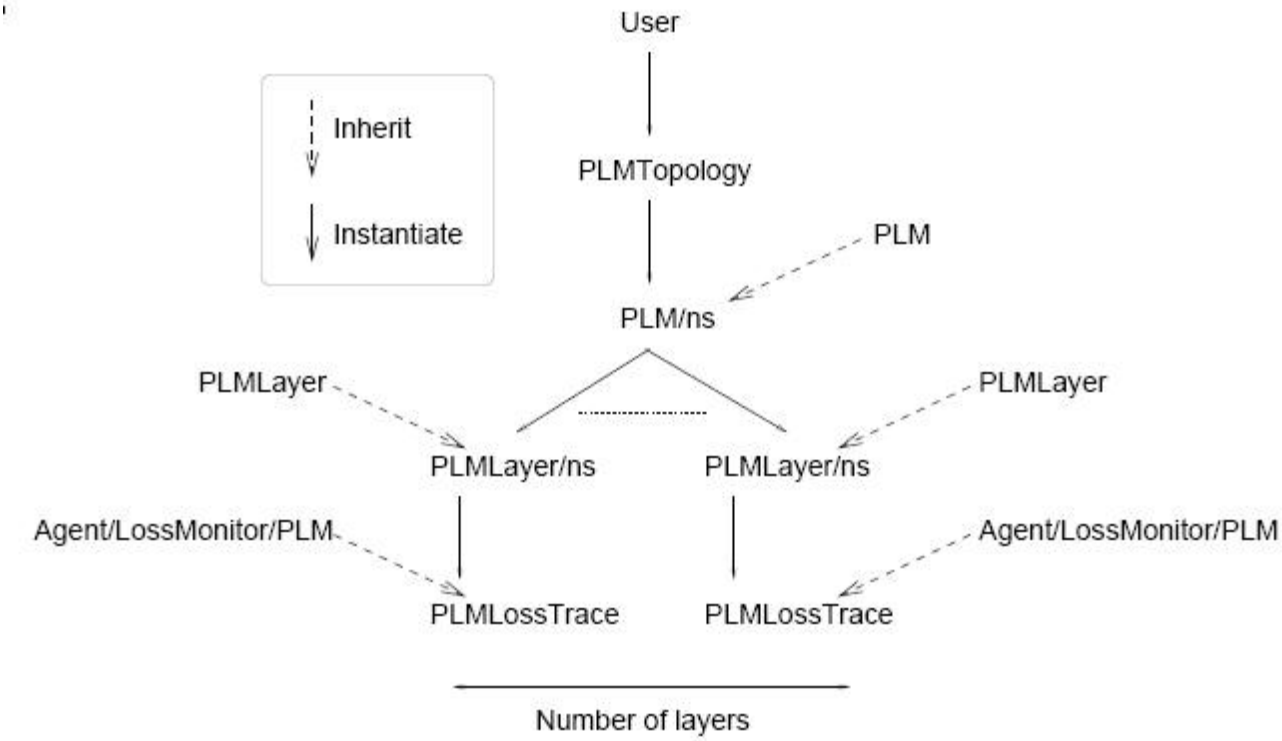


图37.1 创建接收器时的继承和实例化

Init调用了PLM/ns instproc create-layer，并且通过这种方式创建了与层数一样多的PLMLayer/ns类（PLMLayer类的ns接口）的实例。PLMLayer/ns类的实例创建了一个PLMLossTrace类的实例，用于监视接收到和丢失的数据包，这是由于PLMLossTrace类是Agent/LossMonitor/PLM类继承而来。图37.1示意性地描述了一个PLM接收器实例化的过程。下面我们就来讲述当一个PLM接收器接收到一个数据包时和检测到一个数据包丢失时的行为。

37.3.3 分组的接收（Reception）

我们创建了一个新的C＋＋类PLMLossMonitor (~ns/plm/loss-monitor-plm.cc)，它是由 LossMonitor类继承而来。

其OTcl类的名字为Agent/LossMonitor/PLM。

```
class PLMLossMonitor : public LossMonitor
public:
PLMLossMonitor();
virtual void recv(Packet* pkt, Handler*);
protected:
// PLM only
int flag_PP_;
double packet_time_PP_;
int fid_PP_;
;
static class PLMLossMonitorClass : public TclClass
public:
PLMLossMonitorClass() : TclClass("Agent/LossMonitor/PLM")
TclObject* create(int, const char*const*)
return (new PLMLossMonitor());
class_loss_mon_plm;
```

我们在每次接收到数据包的时候，将void PLMLossMonitor::recv(Packet* pkt, Handler*) 的一个Tcl调用添加进Agent/LossMonitor/PLM instproc log-PP中。

```
void LossMonitor::recv(Packet* pkt, Handler*)
...
if (expected_ >= 0)
...
Tcl::instance().evalf("%s log-PP", name());
```

Agent/LossMonitor/PLM instproc log-PP是空的。事实上，我们为PLMLossTrace类定义了log-PP instproc。Log-PP计算了基于单PP脉冲（数据包中PP_burst_length的长度）的可用带宽的估计值。一旦log-PP接收到脉冲的PP_burst_length数据包，它便计算估计值，并且以该估计值为参数调用PLM instproc make_estimate。

make_estimate将基于单PP（PP_value）的估计值放入一个估计样本（PP_estimate）数组中。如果PP_value低于当前订阅级别（例如：低于通过当前预订层数完成的吞吐量），make_estimate就会调用PLM instproc stability-drop，用来简单丢弃层，直到当前订阅级别变得比PP_value低。make_estimate通过在上一个check_estimate周期（如果至少有一个接收到的单一PP估计值PP_estimation_length）获取的一个最小的PP_value来估计PP_estimate_value。一旦make_estimate有一个PP_estimate_value，它便会通过当前订阅级别和PP_estimate_value来调用PLM instproc choose_layer，用于连接和丢弃层，其代码请参考~ns/tcl/plm/plm.tcl。

37.3.4 丢包检测

每次PLMLossMonitor类的实例检测出一个丢包，便会触发对Agent/LossMonitor/PLM instproc log-loss的调用。Agent/LossMonitor/PLM instproc log-loss是空的。事实上，我们为PLMLossTrace类定义了log-loss instproc。PLMLossTrace instproc log-loss简单调用了PLM instproc log-loss，后者包含了丢包的PLM机制。大体上，log-loss只在丢包率超过10%（这项测试由PLM instproc exeed_loss_thresh来执行）时才会丢弃一层。在一层丢弃后，log-loss将在500ms内由于丢包而排除其它层。有关PLM instproc log-loss的详情，请参考~ns/tcl/plm/plm.tcl中的代码。

37.3.5 层的连接与脱离

TPLM instproc add-layer被调用用于连接一层。这个instproc调用了PLMLayer instproc join-group，后者调用了PLMLayer/ns instproc join-group。PLM instproc drop-layer被调用用于脱离一层。这个instproc调用了PLMLayer instproc leave-group，后者调用了PLMLayer/ns instproc leave-group。

37.4 命令一览

注：本小节是由37.1小节结尾部分的复制粘贴而来。我们增加这一节是为了保持ns手册的各个章节的一致性。

PLM的所有模拟都应该在PLMTopology环境下建立（正如我们定义所谓Scenario0的超类PLMTopology的脚本中所示）。用户接口如下所示（所有的instproc可以在~ns/tcl/plm/plm-topo.tcl中找到）：

build_link a b d bw 创建了节点a和节点b之间的一条双向链路，其延迟为d，带宽为bw。如果这两个节点没有出现，build_link将会创建它。

place_source n t 在节点n上创建并放置了PLM数据源，并在时间为t时启动它。place_source返回addr以允许将接收器和该数据源连接起来。

place_receiver n addr C nb 在节点n上创建并放置一个PLM接收器，并将它与返回地址addr的数据源连接起来。这个PLM接收器的检验值为C。一个可选参数nb用于获得一个PLM接收器的实例，该实例只用于取得与该接收器相关的一些特定的统计资料（主要是接收或丢失的分组数目）。

第六篇

应用（Application）

第 38 章

应用程序和传输代理API

在 ns 中应用程序位于传输代理之上。有两种基本类型的应用程序：流量产生器和模拟应用程序。图 38.1 说明了两个应用程序如何组成并被绑在传输代理之上。传输代理将在第 V（五）部分（传输）讲述。

本章首先描述 Application 基类。然后讲述了传输 API，通过它，应用程序可以使用底层传输代理的服务。最后，讲解了目前的流量产生器和数据源的实现。

38.1 Application类

Application 是一个定义如下的 C++类：

```
class Application : public TclObject {
public:
        Application();
        virtual void send(int nbytes);
        virtual void recv(int nbytes);
        virtual void resume();
protected:
        int command(int argc, const char*const* argv);
        virtual void start();
        virtual void stop();
        Agent *agent_;
        int enableRecv_; // 调用OTcl recv（接收）或者不调用
        int enableResume_; // 调用OTcl resume（重新开始）或者不调用
};
```

尽管Application类不一定需要例子演示，但我们也并没有把它当作一个抽象的类，因此它对于OTcl层来说是可见的。这个类提供了应用程序动作（send()，recv()，resume()，start()，stop()）的基本原型，指向它所连接的传输代理的一个指针，还有一个指示OTcl层的调用是否应该响应recv()和resume()事件的标记。

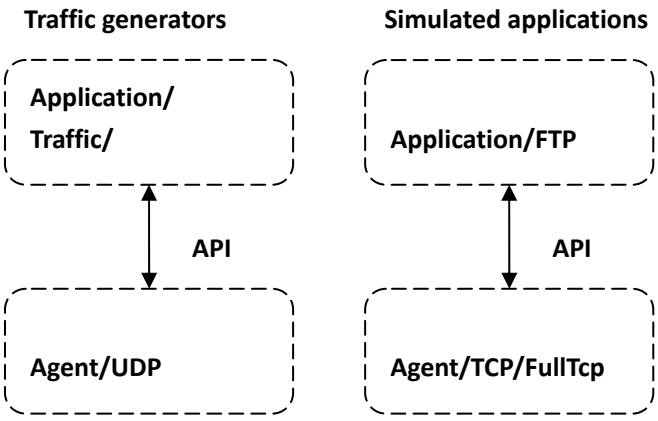


图 38.1：应用程序组成的例子

38.2 传输代理API

在实际的系统中，应用程序一般通过应用编程接口（API）来访问网络的服务，这些 API 中最流行的是“socket”。在 ns 中，我们通过一套已经定义好的 API 函数来模拟 socket API 的行为。这些函数然后被映射到对应的内部代理函数上（比如，通过调用 send(numBytes)可使 TCP 代理增加相应字节数量的“send buffer”）。

本节讲述代理和应用程序如何捆绑在一起，以及如何通过 API 来彼此通信。

32.2.1 把传输代理放在节点之上

这一步通常是在 OTcl 层完成。代理管理在 5.2 节有简要的描述。

```
set src [new Agent/TCP/FullTcp]
set sink [new Agent/TCP/FullTcp]
$ns_ attach-agent $node_(s1) $src
$ns_ attach-agent $node_(k1) $sink
$ns_ connect $src $sink
```

上面的代码用例子说明了，在 ns 中，首先通过 attach-agent 方法将代理放在节点之上。然后，使用 connect 方法把每一对作为源和目的的代理连接起来。注意，比起常规的 socket 方法，ns 中的 connect()有不同的意思。在 ns 中，connect()仅仅为一个代理建立目的地址，并不建立连接，这样的话上面的应用程序就不需要知道它的对手（peer）的地址。对于 TCP 交换 SYN 部分来说，首先调用 send()方法将会触发 SYN 的交换。

为了把一个代理从节点分离开来，可以使用实例函数 detach-agent；这样重置的目的是使这个代理变成一个空代理。

38.2.2 将应用程序放在代理之上

应用程序定义之后 ,必须被连接到一个传输代理之上。attach-agent 方法可以用来把一个应用程序放在代理之上 ,如下所示：

```
set ftp1 [new Application/FTP]
$ftp1 attach-agent $src
```

下面是实现上述功能的快捷方法：

```
set ftp1 [$src attach-app FTP]
```

attach-agent 方法是在 C++中实现的，它也能通过使用 attach-app 而被调用。它把 Application 类中的 agent_指针指向了传输代理，然后它调用了 agent.cc 中的 attachApp()来设置 app_指针指回应用程序。通过维护这个仅存在于 C++中的捆绑，OTcl 层的 instvar 指针就被避免了，从而保证了 OTcl 和 C++之间的一致性。应用程序可以使用 OTcl 层命令[$ftp1 agent]来获得传输代理的句柄。

38.2.3 通过系统调用使用传输代理

一旦传输代理配置好以及应用程序捆绑好之后，应用程序就可以通过下列的系统调用来使用传输服务了。这些调用可以在 OTcl 或者 C++中被激活，从而允许应用程序在 C++或者 OTcl 中编码。这些函数已经作为 Agent 基类中的虚函数被实现了，如果需要的话可以通过重载来重新定义。

- send(int nbytes) — 发送nbytes的数据到peer（对手，对等体，与该节点相连的节点）。对于TCP代理来说，如果nbytes等于-1的话，就对应一个无限的发送；也就是说，这个TCP代理将会像是不断地由应用程序来补充它的发送缓冲区那样来动作。
- sendmsg(int nbytes, const char* flags = 0) — 和上面的send(int nbytes)函数一样，不过它要传送一个附加的字符串flags。目前一个标志值“MSG_EOF”已经被定义；MSG_EOF详细说明了这是最后一组由应用程序提交的数据，作为一个关闭的暗示（让TCP在数据中发送FIN）。
- close() — 要求代理关闭连接（仅仅适用于TCP）。
- listen() — 要求代理监听新的连接（仅仅适用于TCP）。
- set_pkttype(int pkttype) — 这个函数设置了代理中的type_变量为pkttype。包类型定义在packet.h中。这个函数是为了跟踪时可以用来废除传输层包类型。

注意，有些调用不适用于有些代理；比如调用 close()关闭一个 UDP 连接会导致一个空操作。其他的调用可以在特定的代理中实现，只要它们是公开的成员函数。

38.2.4 代理调用应用程序

既然在当前的 ns 中没有实际的数据在应用程序之间传递，那么代理可以通过“upcalls”通知应用程序在传输层发生了某一事件。例如，应用程序可以被告知有一个字节的数据已经到达了；这个信息可以帮助应用程序更紧密地模拟实际的应用程序行为。两个基本的“upcalls”已经在 Application 基类和传输代理中实现了：

- recv(int nbytes) — 宣布代理已经收到了nbytes的数据。对于UDP代理，这标志着一个单一的包的到达。对于TCP

代理，这意味着有大量连续数据的传递，这些数据或许要比一个单一的包的容量大（由于有可能网络重新排序）。

- resume() — 指出传输代理已经把所有的提交给应用程序的数据按时发送给它们了。对于TCP，它并不指示数据是否已经被确认，只是说明第一次发送出去了。

默认的动作如下 根据应用程序是否可以在 C++或者 OTcl 中实现 这些 C++函数调用一个名字类似叫做(recv ,resume ）的函数，如果有这样一个函数已经定义了的话。

尽管严格地说没有对应用程序的回叫，某些代理还是实现了一个从 C++到 OTcl 层的回叫，而且已经被使用到了像 HTTP 模拟器的应用中。这个在 TCP 代理中使用的回叫方法叫做 done{}。在 TCP 中，当一个 TCP 发送器已经接收到所有数据的确认时，会调用 done{}，然后关闭；因此它用来模拟一个阻塞的 TCP 连接。done{}方法主要在 API 完成之前使用，但对于那些不想使用 resume()的应用程序还是有用的。

为了在 FullTcp 中使用 done{}，举例来说，你可以这样试试：

```
set myagent [new Agent/TCP/FullTcp]
$myagent proc done
    ... code you want（你的代码） ...
```

如果你希望所有的 FullTcp 有同样的代码，可以这样：

```
Agent/TCP/FullTcp instproc done
    ... code you want（你的代码） ...
```

默认 done{}不会做任何事情。

38.2.5 示例

这里有一个关于 API 如何在一个 FullTcp 连接之上实现简单应用（FTP）的例子。

```
set src [new Agent/TCP/FullTcp]
set sink [new Agent/TCP/FullTcp]
$ns attach-agent $node_(s1) $src
$ns attach-agent $node_(k1) $sink
$ns connect $src $sink

# 建立TCP-level连接
$sink listen;
$src set window_ 100

set ftp1 [new Application/FTP]
$ftp1 attach-agent $src

$ns at 0.0 "$ftp1 start"
```

在配置脚本中，前五行代码分配了两个新的 FullTcp 代理，并把它们放在正确的节点之上，然后连接起来（给每个代理指定正确的目的地址）。接下来的两行进一步地配置了 TCP 代理，并把它们当中的其中一个设置为 LISTEN（监听）模式。然后，ftp1 定义成为一个新的 FTP 应用，同时在 C++（app.cc）中调用 attach-agent 方法。
ftp1 应用程序在时刻 0 开始：

```
Application/FTP instproc start {} {
    [$self agent] send -1; # 无限发送
}
```

另外，FTP 应用程序在 C++中像下面这样执行：

```
void FTP::start()
{
    agent_->send(-1); // 无限发送
}
```

因此，FTP 应用程序没有利用回叫，这些函数在 C++中是空的，这样 OTcl 中也就没有回叫了。

38.3 TrafficGenerator类

TrafficGenerator 是一个 C++抽象类，定义如下：

```
class TrafficGenerator : public Application {
public:
    TrafficGenerator();
    virtual double next_interval(int &) = 0;
    virtual void init() {}
    virtual double interval() { return 0; }
    virtual int on() { return 0; }
    virtual void timeout();
    virtual void recv() {}
    virtual void resume() {}
protected:
    virtual void start();
    virtual void stop();
    double nextPkttime_;
    int size_;
    int running_;
    TrafficTimer timer_;
};
```

纯虚函数 next_interval()返回时间直到下一个包创建为止，同时设置了下一个包的字节数目。函数 start()调用 init(void)并启动 timer（定时器）。函数 timeout()发送一个包，并安排下一个 timeout。函数 stop()取消所有等待的传输。回叫信号一般不用于流量产生器，因此这些函数（recv，resume）都是空的。

目前，有四个 C++类从 TrafficGenerator 类派生：

1. EXPOO_Traffic — 根据指数On/Off分布产生流量。在On周期包以一个固定的速率发送出去，在Off周期没有包发送。On和Off周期来源于一个指数分布，包的大小固定。
2. POO_Traffic — 根据排列On/Off分布产生流量。除了On和Off周期来自于一个排列分布外，它和指数On/Off分布一样。这些源可以用来产生经历时间较长的聚集流量。
3. CBR_Traffic — 根据确定的速率产生流量。包的大小是不变的，在内部包离开的间隙可能会产生一些随机的抖动。
4. TrafficTrace — 根据一个跟踪文件产生流量。跟踪文件中的每一个记录以网络（big-endian）字节规则包含2个32位的字段。第一个字段包含着下一个包的微秒计产生时间，第二个字段包含着下一个包的字节数。

这些类可以在 OTcl 中创建，OTcl 类的名字和相关参数如下：

Exponential On/Off 指数 On/Off 对象包含在 OTcl 类 Application/Traffic/Exponential 之中，参数化这个对象的成员变量如下：

packetSize_	产生包的恒定大小
burst_time_	产生器的平均“on”时间
idle_time_	产生器的平均“off”时间
rate_	“on”时间内的发送速率

因此一个新的指数 On/Off 流量产生器可以如此创建并参数化：

```
set e [new Application/Traffic/Exponential]
$e set packetSize_ 210
$e set burst_time_ 500ms
$e set idle_time_ 500ms
$e set rate_ 100k
```

注：通过把变量 burst_time_设置为 0 和把变量 rate_设置为非常大的值，可以把指数 On/Off 产生器配置成像一个泊松过程那样。C++的代码保证了即使脉冲时间是 0 的话，也至少发送一个包。此外，下一个间隔时间是所有假定包的传输时间（由变量 rate_决定）的总和，并根据 idle_time_随机变化。因此，为了使总数中的第一个阶段比较小，必须使脉冲速度非常大，以致于相对于一般的闲置时间来说传输时间可以忽略不计。

Parteo On/Off 排列 On/Off 对象包含 OTcl 类 Application/Traffic/Pareto 之中，参数化这个对象的成员变量如下：

packetSize_	产生包的恒定大小
burst_time_	产生器的平均“on”时间
idle_time_	产生器的平均“off”时间
rate_	“on”时间内的发送速率
shape_	排列分布的“shape”参数

一个新的排列 On/Off 流量产生器可以如下创建：

```
set p [new Application/Traffic/Pareto]
$p set packetSize_ 210
```

```
$p set burst_time_ 500ms
$p set idle_time_ 500ms
$p set rate_ 200k
$p set shape_ 1.5
```

CBR CBR 对象包含在 OTcl 类 Application/Traffic/CBR 之中，参数化这个对象的成员变量有：

rate_	发送速率
interval_	（可选的）包之间的间隔
packetSize _	产生包的恒定大小
random_	在计划离开时间内指示是否引入随机“noise”的标志（缺省是off）
maxpkts_	发送包的最大数目（缺省是 228）

因此一个新的 CBR 流量产生器可以如下创建并参数化：

```
set e [new Application/Traffic/CBR]
$e set packetSize_ 48
$e set rate_ 64Kb
$e set random_ 1
```

CBR 对象的 rate_和 interval_的设置是相互排斥（包之间的间隔被作为 C++中的一个内部变量来维护，而且一些 ns 的脚本例子指定间隔而不是速率）。在仿真中，应该为一个 CBR 对象指定一个速率或者一个间隔（但不同时指定两者）。

Traffic Trace 流量跟踪对象由 OTcl 类 Application/Traffic/Trace 初始化。相关的 Tracefile 类可以使得多个 Traffic/Trace 对象和一个跟踪文件联系起来。Traffic/Trace 类用 attach-tracefile 方法把一个 Traffic/Trace 对象和一个特定 Tracefile 对象关联起来。Tracefile 类的 filename 方法将 Tracefile 对象与一个跟踪文件关联在一起。下面的例子演示了如何创建两个 Application/Traffic/Trace 对象，每一个都和同样的跟踪文件（这个例子中叫做“example-trace”）相关联。为了避免流量的同步产生，每一个 Traffic/Trace 对象都要在跟踪文件中选择一个随机的启动位置。

```
set tfile [new Tracefile]
$tfile filename example-trace

set t1 [new Application/Traffic/Trace]
$t1 attach-tracefile $tfile
set t2 [new Application/Traffic/Trace]
$t2 attach-tracefile $tfile
```

38.3.1 示例

下面的代码演示了在一个 UDP 代理上配置一个指数流量源的步骤，通信流从节点 s1 流向节点 k1：

```
set src [new Agent/UDP]
set sink [new Agent/UDP]
$ns_ attach-agent $node_(s1) $src
$ns_ attach-agent $node_(k1) $sink
```

```
$ns_ connect $src $sink

set e [new Application/Traffic/Exponential]
$e attach-agent $src
$e set packetSize_ 210
$e set burst_time_ 500ms
$e set idle_time_ 500ms
$e set rate_ 100k

$ns_ at 0.0 "$e start"
```

38.4 模拟应用程序：Telnet和FTP

目前有两个“模拟应用程序”派生于 Application 类：Application/FTP 和 Application/Telnet。这些类通过由 TCP 传输代理增加有效包的数量来进行工作。实际上有效包的传输仍然由 TCP 流和拥塞算法来控制。

Application/FTP Application/FTP 在 OTcl 中执行，可以模拟大批数据的传输。下面是 Application/FTP 类的一些方法：

attach-agent	把一个Application/FTP对象放在一个代理上
start	通过调用TCP代理的send(-1)函数来启动Application/FTP ,使得TCP看起来像在不断地发送新数据
stop	停止发送
produce n	把要发送包的数目设置为n
producemore n	把要发送包的数目增加n
send n	和producemore类似，但是是发送n个字节而不是n个包

Application/Telnet Application/Telnet 对象利用两种方法中的一种产生包。如果成员变量 interval_非零的话，那么就从指数分布中选择 inter-packet 时间，平均等于 interval_。如果 interval_等于零的话，那么就根据 tcplib 分布（参见 tcplib-telnet.cc 文件）来选择 inter-arrival 时间。Start 方法启动包的产生过程。

38.5 应用程序对象

一个应用程序对象有两种类型，流量产生器或者模拟应用程序。流量产生器对象产生通信流，可以分为四种类型：指数、排列、CBR 和业务跟踪。

Application/Traffic/Exponential 指数流量对象产生 On/Off 通信流。在“on”周期，以一个恒定的脉冲速率产生包。在“Off”周期，不产生通信流。脉冲时间和闲散时间都来自指数分布，配置参数如为：

PacketSize_ 产生包的恒定大小
burst_time_ 产生器on周期的平均时间
idle_time_ 产生器off周期的平均时间
rate_ on周期的发送速率

Application/Traffic/Pareto Application/Traffic/Pareto 对象根据服从排列分布的脉冲时间和闲散时间来产生 On/Off 通信流。配置参数为：

PacketSize_ 产生包的恒定大小
burst_time_ 产生器on周期的平均时间
idle_time_ 产生器off周期的平均时间
rate_ on周期的发送速率
shape_ 排列分布使用的成形参数

Application/Traffic/CBR CBR 对象以一个恒定的比特速率产生包。

$cbr start
引发数据源开始产生包

$cbr stop
引发数据源停止产生包

配置参数为：

PacketSize_ 产生包的恒定大小
rate_ 发送速率
interval_ （可选的）包之间的间隔
random_ 在计划的离开时间里确定是否引入随机噪声，默认是不产生噪声
maxpkts_ 要发送包的最大数目

Application/Traffic/Trace Application/Traffic/Trace 对象根据一个跟踪文件来产生通信流。
$trace attach-tracefile tfile
把跟踪文件对象 tfile 放在一个跟踪上。跟踪文件对象声明了跟踪文件，它来自于被读取的通信数据。多个 Application/Traffic/Trace 对象可以放在同一个跟踪文件对象之上。每一个 Application/Traffic/Trace 对象要选择跟踪文件中的一个随机启动位置。这个对象没有专门的配置参数。

一个模拟应用程序对象也有两种类型：Telnet 和 FTP。

Application/Telent Telnet 对象按如下方式在 inter-arrival 时间内产生单个包。如果 interval_不等于零，从一个指数分布中选择 inter-arrival 时间，平均值是 interval_。如果 interval_等于零，使用 “tcplib” telnet 分布来选择 inter-arrival 时间。

$telnet start
引发Application/Telnet对象开始产生包
$telnet stop
引发Application/Telnet对象停止产生包
$telnet attach <agent>
把一个Telnet对象放在代理上

配置参数为：
interval_ Telnet对象产生的包的平均以秒计inter-arrival时间
Application FTP FTP 对象为要发送的 TCP 对象产生大量的数据。

$ftp start
引发数据源产生maxpkts_包
$ftp produce <n>
使得FTP对象同时产生n个包
$ftp stop
使得已经配置好的TCP对象停止数据发送

$ftp attach agent
把一个Application/FTP放在代理上
$ftp producemore <count>
使得Application/FTP对象产生更多数量的包

配置参数为：

maxpkts 由数据源产生的包的最大的数目

TRACEFILE OBJECTS Tracefile 对象被使用来声明跟踪文件，它被用来产生通信流（参见本节前面的Application/Traffic/Trace 对象描述）。$tracefile 是 Tracefile 对象的实例。

$tracefile filename <trace-input>

设置从通信流读取跟踪数据的 filename（文件名）为 trace-input

这个对象没有配置参数。一个跟踪文件包含许多固定长度的记录，每个记录包含 2 个 32 位字段。第一个字段指示下一个包产生的微秒计间隔时间，第二个字段指示下一个包的字节长度。

38.6 命令一览

下面是一个在模拟脚本中用到的和传输代理以及应用程序相关的命令。

```
set tcp1 [new Agent/TCP]
$ns_ attach-agent $node_(src) $tcp1
set sink1 [new Agent/TCPSink]
$ns_ attach-agent $node_(snk) $sink1
$ns_ connect $tcp1 $sink1
```

这段代码创建了一个源的tcp代理和一个目的的sink代理。两个传输代理被放在他们的各自的（原文为resoective，译者认为写错了，应该是respective）节点上。最后一条点到点的连接在src和sink之间建立起来。

```
set ftp1 [new Application/FTP]
$ftp1 attach-agent $agent
```

或者是set ftp1 [$agent attach-app FTP]，以上的命令都能得到同样的效果。一个应用程序被创建并放在源代理之上。一个应用程序可以有两种类型，一个流量产生器或者是一个模拟应用程序。流量产生器的当前类型有：Application/Traffic/Exponential，Application/Traffic/Pareto ,Application/Traffic/CBR和Application/Traffic/Trace，详见38.3。模拟应用的类型当前可以实现的有：Application/FTP和Application/Telnet，详见38.4。

第 39 章

像应用程序的网络高速缓存

上面讲的应用程序都是“虚拟”应用程序，在某种意义上它们在模拟时并不传输它们自己的数据；数据传输只是和大小与时间有关。有时候或许我们需要应用程序在模拟中传输它们自己的数据。这样的一个例子就是网络高速缓存。我们需要HTTP服务器发送HTTP的头文件到缓冲区和客户端。这些头文件包含页面修改时间的信息和其他的缓存指令，这些对于缓存一致性算法来说是非常重要的。

接下来，我们首先描述ns中关于传送应用程序级数据的常见事项，然后我们讲解特殊的问题，比如API，这些问题和传送应用数据时使用TCP作为传输端口有关。我们还会进一步地介绍HTTP客户端、服务器和代理高速缓存的内部设计。

39.1 在ns中使用应用程序级数据

为了在ns中传输应用程序级（application-level）数据，我们提供了一个统一的结构来在应用程序中传输数据，以及从应用程序传输数据到传输端口代理（图39.1）。它主要有三个部分：一个统一的应用程序级数据单元（ADU），一个在应用程序间传输数据的公用接口，两种在应用程序和传输端口代理之间传输数据的机制。

39.1.1 ADU

ADU 的功能和 Packet 的类似。它需要把用户的数据打包放在一个序列里，该队列然后被放在一个由 Agent 创建的 ns packet 的用户数据区里（这不被目前的代理支持。用户必须由应用程序衍生新的代理来接收用户数据，或者使用像 TcpApp 这样的载体。我们将在后面讨论这些）。

和Packet相比，ADU以不同的方式提供了这个功能。在Packet中，要为所有的包的头部分配一个公共区域；还需要一个偏移来访问该区域中的不同头部。而在ADU中这不可行，因为一些ADU根据用户数据的可用性动态地分配区域。比如说，如果我们想要在应用程序间传输一个OTcl脚本，必须事先定好脚本的大小。因此，我们选择一个效率较低但更灵活的方法。每一个ADU决定了它自己的数据成员，并提供串行它们的方法（也就是，把数据打包到一个序列中和从一个序列中提取数据）。例如，在所有ADU中，都含抽象基类AppData，定义如下：

```
class AppData {
```

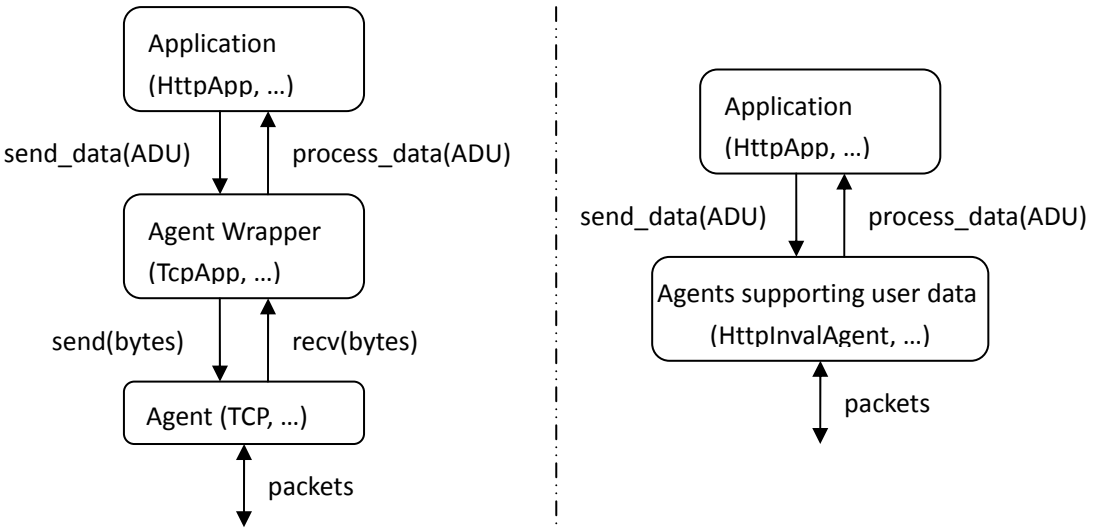


图 39.1: 应用程序级数据流的例子

```
private:
    AppDataType type_; // ADU类型
public:
    struct hdr {
        AppDataType type_;
    };
public:
    AppData(char* b) {
        assert(b != NULL);
        type_ = ((hdr *)b)->type_;
    }
    virtual void pack(char* buf) const;
}
```

这里pack(char* buf)被用来向一个序列中添写AppData对象，AppData(char* b)被用来从一个序列中的“串行化”拷贝对象建立新的AppData。

当从基类衍生新的 ADU 时，用户可以增加更多数据，但同时必须提供新的 pack(char *b)和新的构造函数来读写序列中的新数据成员。关于怎样由 ADU 导出实例，参见文件 ns/webcache/http-aux.h。

39.1.2 在应用程序之间传递数据

Application 的基类 Process 允许应用程序之间相互传递或请求数据，其定义如下：

```
class Process {
public:
```

```
        Process() : target_(0) {}
        inline Process*& target() { return target_; }

        virtual void process_data(int size, char* data) = 0;
        virtual void send_data(int size, char* data = 0);
    protected:
        Process* target_;
    };
```

Process 使 Application 互相连接在一起。

39.1.3 在UDP上传输用户数据

目前在Agent类中不支持用户数据传输。通过传输代理来传输串行化ADU有两种方法。首先，对UDP代理（所有的代理都从那里衍生的），我们从UDP类衍生新类并增加一个新的方法send(int nbytes, char *userdata)来实现从Application传递用户数据到Agent中。从Agent传递用户数据到Application则更具技巧 :每个代理都有一个指向其被附着的应用程序的指针，我们动态地把这个指针指向AppConnector，然后调用AppConnector::process_data()。

作为一个例子，我们说明了怎样实现 HttpInvalAgent 类。它基于 UDP，被用来传递网络缓冲区失效信息（ns/webcache/inval-agent.h）。它被定义如下：

```
    class HttpInvalAgent : public Agent {
    public:
        HttpInvalAgent();
        virtual void recv(Packet *, Handler *);
        virtual void send(int realsize, AppData* data);
    protected:
        int off_inv_;
    };
```

这里 recv(Packet*, Handler*)被重载用来提取用户数据，一个新的 send(int, AppData*)方法也被提出来用来在包中包含用户数据。使用 Agent::attachApp()方法将一个应用程序（HttpApp）放在 HttpInvalAgent 之上（需要动态指派）。在 send()中，下面的代码被用来在包中将用户数据从 AppData 写到用户数据区：

```
    Packet *pkt = allocpkt(data->size());
    hdr_inval *ih = (hdr_inval *)pkt->access(off_inv_);
    ih->size() = data->size();
    char *p = (char *)pkt->accessdata();
    data->pack(p);
```

在 recv()中，下面的代码被用来从包中读取用户数据，并传递给其上的应用程序：

```
    hdr_inval *ih = (hdr_inval *)pkt->access(off_inv_);
```

```
((HttpApp*)app_)->process_data(ih->size(), (char *)pkt->accessdata());
Packet::free(pkt);
```

39.1.4 在TCP上传输用户数据

用 TCP 传送数据比用 UDP 传送数据更富有技巧性，其主要原因是 TCP 的重组队列仅能用于 FullTcp，我们解决这一问题的办法是精简 TCP 连接作为一个 FIFO 通道来使用。

如 38.2.4 节所介绍，应用程序数据的传输可通过代理的上行调用（upcall）来实现。假设我们正在使用 TCP 代理，所有的数据都在队列传递，这就表示我们可以把 TCP 连接看做一个 FIFO 通道。我们仿真 TCP 上的用户数据传输如下：首先在发送器为应用程序数据提供高速缓存，然后在接收器统计所接收字节数。当接收器得到当前传输数据的所有字节之后，它直接从发送器获取数据。Application/TcpApp 类可用来实现这一功能。

对象TcpApp包含一个指向传送代理的指针，可能是FullTcp或者SimpleTcp [25]（当前TcpApp不支持非对称TCP代理，即发送器与接收器分离）。它提供了下列OTcl接口：

- connect：把另外的TcpAPP连接到本身，这个连接是双向的，既只需一个连接命令就可以在两个方向同时发送数据
- send：它有两个参数（nbytes, str）。nbytes是应用数据“名义上”的大小，str是字符串格式的应用数据

为了能够以二进制形式发送应用数据 ,TcpApp 提供了一个 C++的虚函数 send(int nbyte, int dsize, const char* data)。事实上 ,这是用来实现 OTcl 函数 send 的方法。因为在 Tcl 中 ,不容易处理二进制数据 ,所以没有提供处理二进制数据的 OTcl 接口。nbytes 是要传送的字节数；dsize 是数组 data 的大小。

TcpApp 提供了一个 C++虚函数 process-data(int size, char* data)来处理所有接收到的数据。其默认的操作就是把数据看成为一个 TCL 脚本来评估。但是，衍生一个类来处理其它类型数据也是容易的。
下面是一个使用 Application/TcpApp 的例子，在文件 ns/tcl/test/test-suit-webcache.tcl 中还有一个类似的例子：Test/TcpApp-2node。首先，我们创建 FullTcp agent，并把它们连接起来：

```
set tcp1 [new Agent/TCP/FullTcp]
set tcp2 [new Agent/TCP/FullTcp]
# 这里设置TCP参数，比如window_, iss_, . . .

$ns attach-agent $n1 $tcp1
$ns attach-agent $n2 $tcp2
$ns connect $tcp1 $tcp2
$tcp2 listen
```

然后，创建 TcpApp 并把它们连接起来：

```
set app1 [new Application/TcpApp $tcp1]
set app2 [new Application/TcpApp $tcp2]
```

[25]一个 SimpleTcp 代理被单独用来模拟网络高速缓存，它实际上是一个 UDP 代理。它既没有错误恢复也没有流量/拥塞控制，它也不处理包分段。假定一个无损网络中，且按序传递包，SimpleTcp 代理简化了跟踪文件，因此有助于应用协议的调试，在我们的应用中，应用协议是网络高速缓存一致性协议。

```
$app1 connect $app2
```

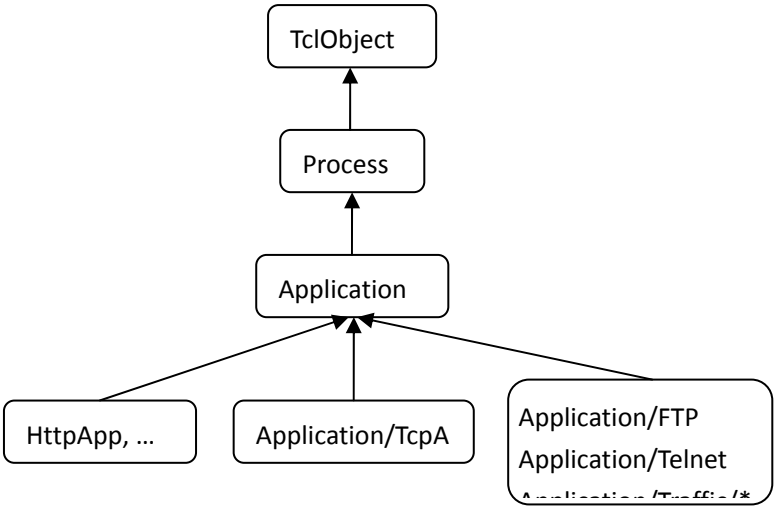


图 39.2：与应用程序级数据处理相关的类的层次结构

现在我们令$app1 是发送器，$app2 是接收器：

```
$ns at 1.0 "$app1 send 100 \"$app2 app-recv 100\""
```

这里的 app-recv 定义如下：

```
Application/TcpApp instproc app-recv { size } {
    global ns
    puts "[$ns now] app2 receives data $size from app1"
}
```

39.1.5 与用户数据处理相关的类的层次结构

我们用本节涉及到的类的层次结构来总结本节（图 39.2）。

39.2 网络高速缓存类的概貌

与实际所使用的一样，和网络缓存有关的类主要有三个：client（浏览器）、server 和 cache。因为它们具有一个共同的特点，即 HTTP 协议，所有它们都是由同一个基类 Http（OTcl 中的名字，在 C++中叫 HttpApp）所衍生出来的。因为下列原因，它们不是一个真正的应用程序。首先，一个 HTTP 对象（即 client/cache/server）可能会同时保持多个 HTTP 连接，但一个 Application 仅仅只有一个 agent_。同时，HTTP 对象需要发送真实的数据（如 HTTP 报头），这是由 TcpApp 提供的，并不是 Agent。因此，我们选择由 TclObject 衍生的一个独立类来处理所有 HTTP 对象的公共特性，这个类正用于管理 HTTP 连接和一系列页面。在本节的剩余部分，我们将介绍这些 Http 的功能。在随后的三节当中，将依次介绍 HTTP client、cache 和 server。

39.2.1 管理HTTP连接

每个 HTTP 连接都被具体化为一个 TcpApp 对象。Http 保存了 TcpApp 对象的 hash，它是其所有的活动连接。这里假设它与任何其他 Http 只有一条 HTTP 连接。它还允许动态的建立或拆除连接。只有 OTcl 接口可用来建立、拆除连接和通过连接发送数据。

OTcl 方法 与 Http 对象中的连接管理有关的 OTcl 接口如下：

id	返回Http对象的id，即此对象所依附节点的id
get-cnc <client>	返回与$client（Http对象）有关的TCP代理
is-connected <server>	如果连接到$server返回0，否则返回1。
send <client> <bytes> <callback>	发送数据$bytes到$client，完成之后执行$callback（OTcl命令）
connect <client> <TCP>	连接TCP 代理和$client（Http对象），这个代理将被用于向$client发送信息包
disconnect <client>	拆除TCP代理与$client的连接，注意：仅仅删除连接，而不删除TCP代理和$client

配置参数 在默认的状态下，Http 对象采用 Agent/SimpleTcp 作为发送 agent（39.1.4 节）。它们也可以使用 Agent/FullTcp 代理，这种代理允许 Http 对象在松散网络中工作。类变量代码 transport_被使用来达到这一目的。例如，Http 通过设置 TRANSPORT_FullTcp 来告之所有的 Http 对象使用 FullTcp agent。

因为FullTcp代理不对SimpleTcp代理做任何内部操作，所以应该在模拟开始之前进行配置并且在模拟过程中不能改变。

39.2.2 管理网络页面

Http 也提供 OTcl 接口来管理一系列的页面。真正的页面管理是由 PagePool 类及其子类来处理的。因为不同的 HTTP 对象具有不同的页面管理要求，所以我们允许不同的 Http 类的子类连接到不同的 PagePool 子类。同时还必须通过 Http 向 OTcl 提供一套公用的 PagePool 接口。例如，一个浏览器可能仅仅使用 PagePool 产生请求信息流，这样它的 PagePool 只需要包含一列 URL。但是，缓冲可能需要保存页面长度和每个页面的最后修改时间，而不仅仅是一列 URL。但是在当前的实现中，这种分离是不清晰的。

页面 URL 的表达格式为：<ServerName>: <SequenceNumber>。其中 ServerName 是 OTcl 对象的名称，而且每个服务器上的每个页面都有一个唯一的 SequenceNumber。页面的内容被忽略了，代之的是每个页面包含若干个属性，这些属性在 OTcl 中表示为下面的（<name> <value>）对："modtime <val>"（页面修改时间），"size <val>"（页面大小）和 "age <val>"。这些值对的排序是无关紧要的。

下面是相关 OTcl 方法的列表。

set-pagepool <pagepool>	设定page pool
enter-page <pageid> <attributes>	向pool中添加id为$pageid的页面。像上面描述的那$attributes是$pageid的属性
get-page <pageid>	用上面描述的格式返回页面属性
get-modtime <pageid>	返回页面$pageid的最后更新时间
exist-page <pageid>	如果Http对象中不存在$pageid，返回0，否则返回1
get-size <pageid>	返回$pageid大小
get-cachetime <pageid>	返回页面$pageid 被加入到缓冲的时间

39.2.3 调试

HttpApp 提供了两种调试方法。日志为特别的 HttpApp 跟踪注册了一个作为跟踪文件的文件操作，它的跟踪格式在 39.9 节描述。evTrace 往跟踪文件中记录特殊事件日志，它连接时间和 HttpApp 的 id 到给定的字符串，然后输出这个串。详细的信息可以参见文件 ns/webcache/http.cc。

39.3 表示网页

我们把网页表示为一个抽象类 Page，定义如下：

```
class Page {
public:
    Page(int size) : size_(size) {}
    int size() const { return size_; }
    int& id() { return id_; }
    virtual WebPageType type() const = 0;

protected:
    int size_;
    int id_;
};
```

它把网页的基本属性描述为：大小和 URL。由它我们衍生出了另外两个网页类：ServerPage 和 ClientPage。前者包含一系列的页面修改时间，被期望使用于服务器。它最初被设计用来与特殊的网络服务跟踪协作；目前它在 ns 中没有被广泛使用。后者，ClientPage，是下面所有 page pools（页面池）的缺省网页。

一个 ClientPage 有下面的主要属性（我们省略了一些被缓存使用的无效变量，因为它们有太多细节，不可能在这里全部介绍）：

- HttpApp* server_　　页面的原始服务器指针
- double age_　　页面的生命周期
- int status_　　页面的状态，它的内容将在下面解释

一个 ClientPage 的状态（32 位）分为两个 16 位的部分。第一部分（格式为 0X00FF）用来保存页面状态；第二部分（格式为 0XFF00）用来保存缓冲所执行的预期页面操作。有效的页面状态有（同样，我们省略了那些与无效页面非常相关东西）：

HTTP_VALID_PAGE　　页面是有效的
HTTP_UNCACHEABLE　　页面是不能缓冲的，这个操作可用来模拟 CGI 页面或动态服务器页面

ClientPage 主要有如下的 C++方法：

- type()　　返回页面的类型。如果相同类型的页面具有同样的操作，那么我们让所有的ClientPage是“HTML”类型。如果以后需要其他类型的网页，则从ClientPage（或Page）衍生出一个类，且具有期望的类型。

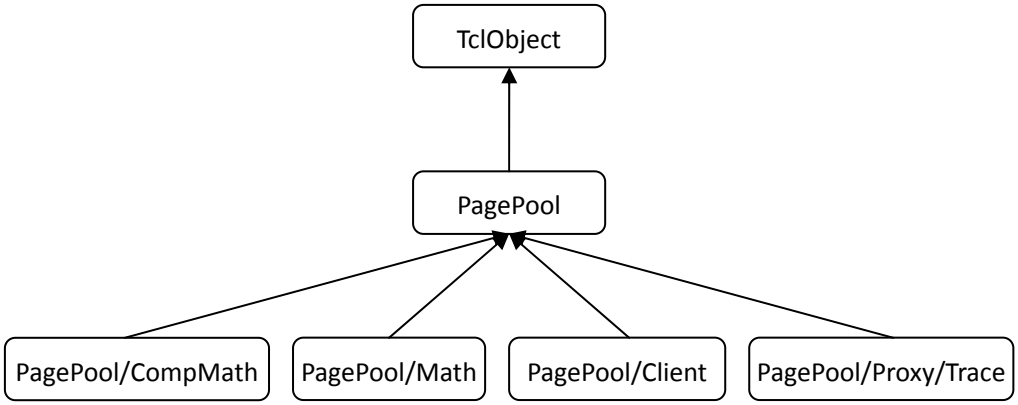


图 39.3：页面池的类的层次结构

- name(char *buf)　把页面名称写入给定的缓冲中。一个页面名称使用的格式为：<ServerName>: <PageID>
- split_name(const char *name, PageID& id)　把给定的页面名称分为两部分，这是一个静态的方法
- mtime()　返回页面的最后修改时间
- age()　返回页面的生命周期

39.4 页面池

服务器用 PagePool 及其衍生出来的类产生页面信息（名称，大小，修改时间，生命周期等），高速缓存用来描述所存储的页面，客户端用来产生请求数据流。图 39.3 提供了这里的类的层次结构的一个概貌。

在这些类中，PagePool/Client 是最常被高速缓存用来存储页面和其他高速缓存相关信息的类；其他三个类被服务器和客户端使用。下面我们将一个接一个地描述这些类。

39.4.1 PagePool/Math

这是页面池的简单类型。它仅仅有一个页面，大小由一个给定的随机变量生成。通过使用两个给定的随机变量生成页面修改队列和请求队列。它有如下的 OTcl 方法：

gen-pageid	返回下一个被请求页面的ID。因为它仅仅有一个页面，所以总是返回值0
gen-size	返回页面的大小。它可以用一个指定的随机变量来产生
gen-modtime <pageID> <mt>	返回页面的下一次修改时间。<mt>给出了上一次的修改时间。它还使用了生命周期随机变量
ranvar-age <rv>	设置文件生命周期随机变量为<rv>
ranvar-size <rv>	设置文件大小随机变量为<rv>

注意：这里有两种方法产生一个请求队列。对于除了 PagePool/Proxy/Trace 以外的所有页面池，请求队列由描述请求间隔的随机变量产生，其他的页面池的 gen-pageid 方法给下一个请求返回页面 ID。PagePool/Proxy/Trace 在初始化阶段装载请求数据流，因此对于请求间隔它不需要随机变量；它的描述参见下面。

使用 PagePool/Math 的例子在 39.8 节，脚本可参见 *ns*/tcl/ex/simple-webcache.tcl。

39.4.2 PagePool/CompMath

通过引入一个复合页面模型，在 PagePool/Math 基础上得以改进。我们把一个主要由文本页面和大量嵌入对象（例如 GIF）所组成的页面表示为复合页面。我们把复合页面模拟为一个主页面和若干个组件对象。其中主页面通常被指定 ID 为 0。所有组件页面的大小相同。主页面的长度和组件对象的长度都是固定的，但是它们可以分别通过 OTcl 捆绑变量 main_size_ 和 comp_size_来调节。组件对象的数目可以用 OTcl 捆绑变量 num_page_来设定。

PagePool/CompMath 有下列主要的 OTcl 方法：

gen-size <pageID>	如果<pageID>是，返回main_size_，否则返回comp_size_
ranvar-main-age <rv>	为主页面生命周期设置随机变量。两外一个方法，ranvar-obj-age，为组件对象来设置
gen-pageid	总是返回0，它是主页面ID
gen-modtime <pageID> <mt>	返回给定页面<pageID>的下一个修改时间。如果给定 ID 是 0，它使用主页面生命周期随机变量；否则使用组件对象生命周期随机变量

关于使用 PagePool/CompMath 的例子参见文件 *ns*/tcl/ex/simple-webcache-comp.tcl。

39.4.3 PagePool/Proxy/Trace

上面的两个页面池综合两个变量来表示对一个简单网页的请求流：一个请求请求时间间隔；另一个用于被请求页面 ID。有时，用户可能会需要较复杂的请求流，这个请求流由复合页面、exhibits 空间位置和临时位置组成。现在已经有生成这样请求流的提议（SURGE[3]）了，我们选择了提供一个替代的解决方法：使用真正的网络代理高速缓存跟踪（或服务器跟踪）。

类PagePool/Proxy/Trace使用真正的跟踪来进行模拟。因为现存的许多网络跟踪具有不同的格式，所以它们在注入页面池之前就必须被转变为中间格式。有效的转换器在http://mash.cs.berkeley.edu/dist/vint/webcache-trace-conv.tar.gz中。它可以容纳四种不同格式的跟踪：DEC代理跟踪（1996），UCB Home-IP跟踪，NLANR代理跟踪和EPA网络服务器跟踪。它把给定的跟踪转换成两个文件：pglog和reqlog。Pglog中的每一行的格式如下：

[<serverID> <URL_ID> <PageSize> <AccessCount>]

除了最后一行，reqlog的每一行有如下格式：

[<time> <clientID> <serverID> <URL_ID>]

reqlog的最后一行记录了整个跟踪持续时间和唯一URL的总数：

i <Duration> <Number_of_URL>

PagePool/ProxyTrace 用这两个文件作为输入，然后用他们来驱动模拟。因为大部分现存的网络代理跟踪不具有完整的页面修改信息，我们选择使用双模态的页面修改模式[7]。我们允许用户选择页面中的 *x*%来拥有一个随机的页面修改间隔发生器，剩余的页面有另外一个发生器。通过这个方法，就可使 *x*%的页面变成动态的，即频繁的修改，而其余的则是静态的。热门的页面被均匀地分散到所有页面中。例如：假定 10%的页面是动态的，那么如果我们根据其声望值，把页面存储到一个列表中，则页面 0、10、20…是动态的，剩余的是静态的。由于采用这种选择机制，在这 10%的单元中我们只能改变双模态比率。

为了把请求分布到模拟器中的不同请求器中，PagePool/ProxyTrace 用模操作把跟踪中的客户 ID 与模拟器中的请求器对应了起来。

PagePool/ProxyTrace 的主要 OTcl 方法如下：

get-poolsize	返回页面的总数
get-duration	返回跟踪的持续时间
bimodal-ratio	返回双模态比率
set-client-num <num>	设置模拟中的请求器数量
gen-request <ClientID>	为给定的请求器产生下一次的请求
gen-size <PageID>	返回给定页面的大小
bimodal-ratio <ratio>	设置动态页面为<ratio>*10%。注意，这个比率在10%的范围内波动
ranvar-dp <ranvar>	设置动态页面的修改时间间隔发生器。同样，ranvar-sp <ranvar>是设置静态页面的发生器
set-reqfile <file>	设置上面所介绍的请求流文件
set-pgfile <file>	设置上面所介绍的页面信息文件
gen-modtime <PageID> <LastModTime>	建立给定页面的下一次修改时间

在文件 *ns*/tcl/ex/simple-webcache-trace.tcl 中有一个关于使用 PagePool/ProxyTrace 的例子。

39.4.4 PagePool/Client

PagePool/Client 类帮助高速缓存跟踪缓存中的页面，存储各种与高速缓存有关的关于页面的信息。因为主要用于内部，并且用户几乎不需要使用什么功能，它大都用 C++来实现。它具有下列主要的 C++方法：

- get_page(const char* name) 返回具有指定名称的页面指针
- add_page(const char *name, int size, double mt, double et, double age)添加一个具有给定长度、最后修改时间（mt）、高速缓存加入时间、页面生命周期（age）的页面
- remove_page(const char* name) 从高速缓存中删除一个页面

此页面池应该能够支持多种高速缓存替换算法，但是它仍然还没有实现。

39.4.5 PagePool/WebTraf

PagePool/WebTraf 类是一个独立网络流量模型，它利用 PagePool 框架。然而，这个类已经完全不同于 HttpApp 类。因为在这里我们仅仅对使用它研究网络流量模式感兴趣，所以不希望与传输 HTTP 头等这样的问题搅在一起。它有下面两个主要的数据结构。细节可以参见 *ns*/webcache/webtraf.cc 和 *ns*/webcahe.webtraf.h，WebTraf 模型的体系结构在[10]的 2.4 节 3-4 段中有描述，在附录 A.1 中也有。

- WebTrafSession 一个模仿网络用户会话的类，定义如下：

```
class WebTrafSession : public TimerHandler {
public:
    WebTrafSession(WebTrafPool *mgr, Node *src, int np, int id) : rvInterPage_(NULL),
    rvPageSize_(NULL), rvInterObj_(NULL), rvObjSize_(NULL), mgr_(mgr), src_(src),
    nPage_(np), curPage_(0), donePage_(0), id_(id), interPageOption_(1) {}
```

```
        virtual ~WebTrafSession();

        // 单独页面/对象的查询
        inline RandomVariable*& interPage() { return rvInterPage_; }
        inline RandomVariable*& pageSize() { return rvPageSize_; }
        inline RandomVariable*& interObj() { return rvInterObj_; }
        inline RandomVariable*& objSize() { return rvObjSize_; }

        void donePage(void* ClntData);   // 这个集合中的所有网页
                                         // 会话已经发送
        void launchReq(void* ClntData, int obj, int size);
        inline int id() const { return id_; }
        inline WebTrafPool* mgr() { return mgr_; }
    private:
        virtual void expire(Event *e = 0); // 对一个页面发动请求
        virtual void handle(Event *e); // 对下一个页面计划定时器

        RandomVariable *rvInterPage_, *rvPageSize_, *rvInterObj_, *rvObjSize_;
        WebTrafPool* mgr_;
        Node* src_; // 每个会话的一个网络客户（请求源）
        nt nPage_; // 每个会话的页面数量
        int curPage_; // 已经发送的页面数量
        int id_; // 页面ID
        int interPageOption_;
    }
```

- WebPage 一个模仿网页的类，定义如下：

```
    class WebPage : public TimerHandler {
    public:
        WebPage(int id, WebTrafSession* sess, int nObj, Node* dst) :
                id_(id), sess_(sess), nObj_(nObj), curObj_(0),
                doneObj_(0), dst_(dst) {}
        virtual ~WebPage() {}
        inline void start() { // 调用expire()，如果有需要，计划下一次动作
        void doneObject() { // 本页面的所有对象已经发送
        inline int id() const { return id_; }
        Node* dst() { return dst_; }
        inline int curObj() const { return curObj_; }
        inline int doneObj() const { return doneObj_; }
    private:
        virtual void expire(Event* = 0) { // 对一个对象发动请求
        virtual void handle(Event *e) { // 对下一个对象计划定时器
        int id_; // 对象ID
        WebTrafSession* sess_; // 请求本页面的会话
        int nObj_; // 本页面中的对象数量
        int curObj_ ; // 已经发送的对象数量
```

```
        Node* dst_; // 请求本页面的服务器
    }
```

下面是 WebTraf 类的相关 OTcl 方法的列表。

set-num-session <number-of-session>	设置WebTraf池中的会话总数
set-num-server <number-of-server>	设置服务器总数
set-num-client <number-of-client>	设置客户端总数
set-interPageOption <option>	有两种方法解释*inter-page*时间：一种是被相同用户下载的两个连续页面的开始时间之间的间隔；另外一种是被相同用户下载的前一个页面的结束和后续页面的开始之间的时间间隔。$option可以设置为0或者1（缺省是1）。当$option被置为1时，用第二种方法解释“inter-page”时间。第一种解释适用于$option被置为0的情况。注意，相比第二种解释方法，第一种更多被用在在解释被引发的流量大小
doneObj <webpage>	所有在$webpage中的对象已经发送
set-server <id> <node>	设置$node作为服务器$id
set-client <id> <node>	设置$node客户端$id
recycle <tcp> <sink>	回收TCP发送/接收（source/sink）对
create-session <session-index> <pages-per-sess> <launch-time> <inter-page-rv> <page-size-rv> <inter-obj-rv> <obj-size-rv>	建立一个网络会话。$session-index是会话索引， $pages-per-sess是每个会话中页面的总数，$launch-time是会话开始时间$inter-page-rv是产生页面内部到达（inter-arrival）时间的随机变量，$page-size-rv是产生每个页面对象数量的随机变量，$inter-obj-rv是产生对象内部到达时间的随机变量，$obj-size-rv是产生对象大小的随机变量

例子脚本参见 ns/tcl/ex/web-traffic.tcl（也可以参考 ns/tcl/ex/large-scale-web-traffic.tcl 了解大范围网络流量模拟的使用）

39.5 网络客户端

HttP/Client 类模仿简单的网络浏览器。它产生一个的页面请求序列，其中页面 ID 和请求间隔是随机的。它是从 Http 继承过来的纯 OTcl 类。接下来，我们将介绍它的功能和用法。

建立一个客户端 首先，我们建立一个客户端，并把它连接到高速缓存和网络服务器。目前一个客户端仅仅被允许连接一个单一的高速缓存，但是它被允许连接多个服务器。注意，这必须在模拟开始之后才能调用（即在$ns run 之后才能调用）。除非明确说明，它将用于下列的所有方法和 Http 代码示例及其衍生类。

```
# 假定$server是一个配置好的Http/Server
set client [new Http/Client $ns $node]          # 客户端寄居在节点之上
$client connect $server                         # 连接客户端和服务器
```

配置请求生成 对每一个请求，Http/Client使用PagePool生成一个随机页面ID，并使用一个随机变量来生成两个连续页面之

间的间隔：[26]

```
$client set-page-generator $pgp          # 附上一个配置好的PagePool
$client set-interval-generator $ranvar   # 附上一个随机变量
```

这里我们假定 Http/Client 的 PagePool 和服务器的 PagePool 一起共享同样的页面集合。通常我们通过让所有客户端和服务器共享相同的 PagePool 来简化模拟，即它们有同样的页面集合。当存在多个服务器，或者服务器的 PagePool 与它们客户端的 PagePool 相分离的时候，我们必须注意确保每个客户端与它们相连的服务器看到相同的页面集合。

启动 按上面的设置好后，启动请求是非常简单的：

```
$client start-session $cache $server     # 假定$cache是一个配置好的Http/Cache
```

OTcl 接口 下面是它的 OTcl 方法清单（除了那些从 Http 继承的以外）。这不是一个完整的清单，更加详细的可以参见 *ns*/tcl/webcache/http-agent.tcl。

send-request <server> <type> <pageid> <args>	发送一个页面请求$pageid和类型$type到$server。对于客户端，唯一允许的请求类型是GET。$args个格式与Http::enter-page中所描述的$attributes 相同
start-session <cache> <server>	开始通过$cache发送一个随机页面请求到$server
start <cache> <server>	在发送请求前，向$cache中添加客户端的PagePool中的所有页面。在假定高速缓存大小不限和要观测在稳定状态下高速缓存一致性算法的行为的时候，这个方法是用的
set-page-generator <pagepool>	连接PagePool来产生随机页面ID
set-interval-generator <ranvar>	连接一个随机变量来产生随机请求间隔

39.6 网络服务器

Http/Server类用来模仿Http服务器。它的配置非常简单。用户所有要做的就是建立一个服务器，连接到一个PagePool，然后就是等待：

```
set server [new Http/Server $ns $node]   # 把$server放在节点$node之上
$server set-page-generator $pgp          # 连接一个页面池
```

一个Http/Sever对象在模拟开始之后等待流入请求。通常，由客户端和高速缓存初始化与Http/Server的连接。但是它（Http/Server）也有自己的connect方法，这个方法允许Http/Server对象动态地连接到某个客户端和高速缓存。有时候这是非常有用的，正如文件ns/tcl/test/test-suite-webcache.tcl中的方法Test/TLC1::set-groups{}所阐述的。

一个 Http/Server 对象接收两种类型的请求：GET 和 IMS。GET 请求模仿正常的客户端请求。对于每个 GET 请求，它返回请求页面的属性。IMS 请求为高速缓存一致性模仿被 TTL 算法使用的 If-Modified-Since。对于每个 IMS（If-Modified-Since）请求，它比较由请求给定的页面修改时间和它的 PagePool 中的页面修改时间。如果这个时间指示请

[26]一些 PagePool，比如 PagePool/Math，仅仅有一个页面，因此它总是返回相同的页面。一些其他的 PagePool，比如 PagePool/Trace，有多个页面，因此需要一个随机变量来选出一个随机页面。

求是较前的，它发回一个很小大小的回复，否则它返回一个与真实页面大小相同的回复，这个回复包含页面的所有属性。

39.7 网络高速缓存

目前实现了六中类型的网络高速缓存，包括Http/Cache基类。它的五个衍生子类实现了五种高速缓存一致性算法：简单的旧版本TTL，适应的TTL，无所不知的TTL，层次多播失效算法和层次多播失效加直接请求算法。

因为 Http/Cache 的所有子类都包括了关于高速缓存一致性算法的讨论，这里介绍似乎不太合适，所以下面仅仅描述了 Http/Cache 基类

39.7.1 Http/Cache

Http/Cache 类用来模仿有限大小的简单 HTTP 高速缓存。它不包含排除算法，也不包含一致性算法。我们没有打算让它单独使用，而只是作为用来实验不同的高速缓存一致性算法和其他高速缓存算法的基类。

建立和启动 建立一个 Http/Cache 要求和 Http/Client 与 Http/Server 同样的参数集。建立之后，一个高速缓存需要连接到一个确定的服务器上。注意，当一个请求进来，高速缓存发现它没有连接到服务器时，这个建立也可以是动态的。然而，在目前的代码里我们不能模拟这样的行为。下面的代码是一个例子：

```
set cache [new HttpCache $ns $node]          # 把高速缓存放在节点$node上
$cache connect $server                          # 连接到服务器$server
```

像 Http/Server，在如上的初始化后，一个 Http/Cache 对象等待请求（和来自服务器的包）。当使用层次高速缓存时，下面的代码能用来建立层次：

```
$cache set-parent $parent                    # 设置高速缓存的父缓存
```

目前所有的 TTL 和多播失效高速缓存都支持层次高速缓存。然而，仅仅有两种多播失效高速缓存允许多播高速缓存层次进行内部操作。

OTcl 方法 尽管 Http/Cache 是一个分裂对象，它的所有方法都用 OTcl 实现。它们大多用来处理流入请求。它们的关系可以用下面的流程图和接下来的解释来说明。

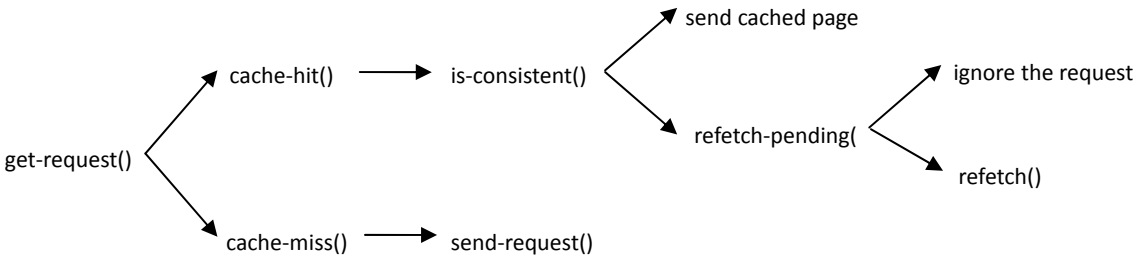


图 39.4：Http/Cache 中处理流入请求

get-request <client> <type> <pageid> 处理请求的入口点。它检查高速缓存的页面池中是否存在所请求的页面

	$pageid，然后调用cache-hit和cache-miss
cache-miss <client> <type> <pageid>	这个高速缓存没有页面。如果还没向服务器(或父缓存)发送请求来获取页面，便发出获取页面请求。把$client记录在列表中，以便当高速缓存得到页面的时候，转发页面到所有请求此页面的客户端
cache-hit <client> <type> <pageid>	检查已缓存页面的有效性。如果有效，给客户端发送已缓存页面，否则重新获取页面
is-consistent <client> <type> <pageid>	如果$pageid有效则返回1。在子类中要重载
refetch <client> <type> <pageid>	从服务器重新获取无效页面。在子类中要重载

39.8 整理成一个简单的例子

我们已经了解到了所有的部分，现在我们把所有的部分列在一起作一个概述。首先，我们建立拓扑和其它的初始化操作：

```
set ns [new Simulator]

# 建立拓扑/路由
set node(c) [$ns node]
set node(e) [$ns node]
set node(s) [$ns node]
$ns duplex-link $node(s) $node(e) 1.5Mb 50ms DropTail
$ns duplex-link $node(e) $node(c) 10Mb 2ms DropTail
$ns rtproto Session
```

接下来，我们建立 Http 对象：

```
# HTTP日志
set log [open "http.log" w]

# 建立页面池来作为中心页面生成器。使用PagePool/Math
set pgp [new PagePool/Math]
set tmp [new RandomVariable/Constant]          # 页面大小生成器
$tmp set val_ 1024                             # 平均页面大小
$pgp ranvar-size $tmp
set tmp [new RandomVariable/Exponential]               # 页面生存时间生成器
$tmp set avg_ 5                                        # 平均页面生存时间
$pgp ranvar-age $tmp

set server [new Http/Server $ns $node(s)]              # 建立一个服务器并连接到中心页面池
$server set-page-generator $pgp
$server log $log

set cache [new Http/Cache $ns $node(e)]                # 建立高速缓存
$cache log $log
```

```
set client [new Http/Client $ns $node(c)]          # 建立客户端
set tmp [new RandomVariable/Exponential]           # 使用泊松过程作为请求序列
$tmp set avg_ 5                                    # 平均请求间隔
$client set-interval-generator $tmp
$client set-page-generator $pgp
$client log $log

set startTime 1                                    # 模拟开始时间
set finishTime 50                                  # 模拟结束时间
$ns at $startTime "start-connection"
$ns at $finishTime "finish"
```

然后，我们定义一个过程，它在模拟开始后将被调用。这个过程将在所有 Http 对象之间建立连接。

```
proc start-connection {} {
    global ns server cache client
    $client connect $cache
    $cache connect $server
    $client start-session $cache $server
}
```

在结尾，通常的结束方法为：

```
proc finish {} {
    global ns log
    $ns flush-trace
    flush $log
    close $log
    exit 0
}
$ns run
```

在文件*ns*/tcl/ex/simple-webcache.tcl中也有这个脚本。检查其输出结果http.log，我们将发现如果缺少高速缓存一致性算法，将会导致大量的失效命中。用“set cache [new Http/Cache/TTL $ns $node(e)]”替换“new Http/Cache”可以修复此错误。较复杂的高速缓存一致性算法的例子参见文件*ns*/tcl/test/test-suit-webcache.tcl。

39.9 Http跟踪的格式

Http 代理的跟踪文件的构造方法和 SRM 跟踪文件的相似。它由多个入口组成，每个入口占一行，其格式如下：

Time | ObjectID | Object Values

有三种类型的对象：客户端（**C**），高速缓存（**E**）和服务器（**S**）。下面是与这三种类型相关的所有可能事件和值类型的完整列举。

对象类型	*事件类型*	*Values（值）*
E	**HIT**	<Prefix>
E	**MISS**	<Prefix> z <RequestSize>
E	**IMS**	<Prefix> z <Size> t <CacheEntryTime>
E	**REF**	p <PageID> s <ServerID> z <Size>
E	**UPD**	p <PageID> m <LastModifiedTime> z <PageSize> s <ServerID>
E	**GUPD**	z <PageSize>
E	**SINV**	p <PageID> m <LastModTime> z <PageSize>
E	**GINV**	p <PageID> m <LastModTime>
E	**SPF**	p <PageID> c <DestCache>
E	**RPF**	p <PageID> c <SrcCache>
E	**ENT**	p <PageID> m >LastModifiedTime> z <PageSize> s <ServerID>
C	**GET**	p <PageID> s <PageServerID> z <RequestSize>
C	**STA**	p <PageID> s <OrigServerID> l <StaleTime>
C	**RCV**	p <PageID> s <PageServerID> l <ResponseTime> z <PageSize>
S	**INV**	p <PageID> m <LastModifiedTime> z <Size>
S	**UPD**	p <PageID> m <LastModifiedTime> z <Size>
S	**SND**	p <PageID> m <LastModifiedTime> z <PageSize> t <Requesttype>
S	**MOD**	p <PageID> n <NextModifyTime>

<Prefix>是对所有跟踪入口的公共信息，它包括：

p <PageID> c <RequestClientID> s <PageServerID>

事件操作的简要介绍：

对象类型	事件类型	介绍
E	HIT	高速缓存命中。PageSererID是页面“拥有者”的id
E	MISS	高速缓存没命中。这样的话，缓存将发送请求到服务器获取页面
E	IMS	If-Modified-Since。使用TTL协议来验证过期页面
E	REF	页面重新获取。使用失效协议来重新获取一个无效页面
E	UPD	页面更新。使用失效协议来从父高速缓存到子高速缓存“push（推）”更新 发送失效信息 获取失效信息
E	SINV	发送pro forma（形式化信息，也可写为proforma）
E	GINV	接收pro forma（形式化信息，也可写为proforma）
E	SPF	输入一个页面到本地页面高速缓存
E	RPF	
E	ENT	
C	GET	客户端发送一个页面请求
C	STA	客户端获得一个失效命中。OrigModeTime是网络服务器上修改时间，CurrModTime是本地页面的修改时间
C	RCV	客户端接收一个页面
S	INV	服务器发送一个回应
S	UPD	服务器把页面更新送到它的“主高速缓存”。仅由失效协议使用
S	SND	服务器发送一个失效信息。仅由失效协议使用
S	MOD	服务器修改一个页面。页面下次将在<NextModifyTime>被修改

39.10 命令一览

下面是网络高速缓存应用程序的相关命令：

set server [new Http/Server <sim> <s-node>]
这条命令在指定的<s-node>建立Http服务器的一个实例。需要传递一个模拟器<sim>实例的参数

set client [new Http/Client <sim> <c-node>]
这条命令在给定的<c-node>建立Http客户端的一个实例

set cache [new Http/Cache <sim> <e-node>
这条命令建立一个高速缓存

set pgp [new PagePool/<type-of-pagepool>]
这条命令建立一个指定类型的页面池（pagepool）。目前已经实现的不同类型页面池有：PagePool/Math，PagePool/CompMath，PagePool/ProxyTrace和PagePool/Client。参见39.4节了解Pagepool每种类型的Otcl接口详情

$server set-page-generator <pgp>
$server log <handle-to-log-file>
上面的命令是服务器配置。首先服务器被连接到中心页面池<pgp>，接下来连接到一个日志文件

client set-page-generator <pgp>
$client set-interval-generator <ranvar>
$client log <handle-to-log-file>
这些是Http客户端的配置。它连接到一个中心页面池<pgp>，接下来一个随机变量<ranvar>被连到客户端，这个变量被它（客户端）使用来生成两个连续请求之间的间隔，最后为了记录事件这个客户端被连接到日志文件

$cache log <log-file>
这是高速缓存配置部分，它允许高速缓存在日志文件中记录其事件

$client connect <cache>
$cache connect <server>
一旦客户端，高速缓存和服务器配置好了，它们需要使用上面的命令连接起来

$client start-session <cache> <server>
开始从客户端向服务器<server>，通过高速缓存<cache>发送一个随机页面请求

第 40 章

蠕虫模型

在这一章，我们描述一个ns中可升级的蠕虫传播模型，也就是具体网络和抽象网络（DN-AN）模型。它结合了包级别的模拟与分析的蠕虫传播模型。如图40.1所示，我们把Internet模拟成两个部分：具体的和抽象的部分。具体网络可以是企业网或者ISP管理的网络。它模拟网络的连通性和包的传送。在具体网络中，用户可以评估蠕虫检测算法。另一方面，我们抽象Internet的剩余部分为数学模型，也就是susceptible- infectious-removal（SIR，易感染性排除）模型（详细描述参见[13]）。相比于具体网络，在抽象网络中，我们仅仅跟踪几个状态变量，比如被感染主机的数量。DN（具体网络）和AN（抽象网络）之间的的交互是通过实际的包传送进行的，也就是说，探测流量由两边的折衷主机生成。

更详细的DN-AN模型描述请参见我们的论文的初稿。我们把蠕虫传输模型作为应用程序实现了。源程序代码在文件*~ns*//apps/worm.{cc,h}中。在文件*~ns*/tcl/ex/worm.tcl中有一个说明DN-AN模型的例子脚本。

40.1 概述

我们用三个类实现的蠕虫传播模型：WormApp、DnhWormApp和AnWormApp类。WormApp和DnhWormApp类在具体网络中使用，分别代表无懈可击和易受攻击的主机。AnWormApp类用在抽象网络。目前，我们的模型仅仅支持基于UDP的蠕虫。

一个易受攻击的主机在接收一个探测包时是缺乏免疫的。然后，它选择一个目的主机（随机或者根据本地邻居的某种偏好）来扫描。探测包对于无懈可击的主机没有效果。当抽象网络接收探测包时，它更新它目前的状态。

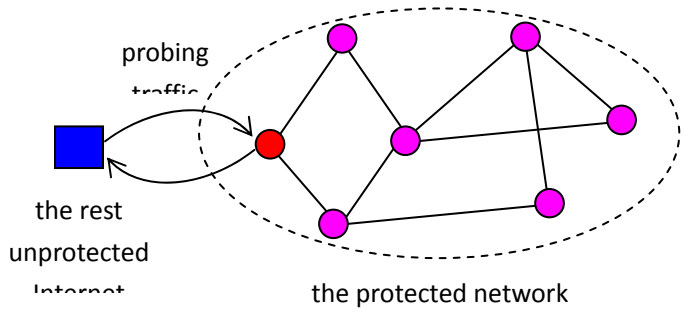


图40.1: DN-AN模型

40.2 配置

为了建立模拟场景，我们首先构造具体网络。我们也需要建立一个额外的节点来代表抽象网络，然后把它连接到具体网络。

对于具体网络中的节点，我们首先放一个MessagePassing代理在每个节点之上：

```
set a [new Agent/MessagePassing]
$n attach $a $probing_port
```

如果节点表示易受攻击的主机，我们使用DnhWormApp类

```
set w [new Application/Worm/Dnh]
$w attach-agent $a
```

否则，我们配置节点为无懈可击的：

```
set w [new Application/Worm]
$w attach-agent $a
```

按如下方式配置抽象网络：

```
set a [new Agent/MessagePassing]
$na attach $a $probing_port
set w [new Application/Worm/An]
$w attach-agent $a
```

对于抽象网络，为了接收由具体网络中的节点产生的探测包，我们需要使用手动路由。对于抽象网络的节点有一些额外的配置：

```
set p [$na set dmux_]
$p defaulttarget $a
[$na entry] defaulttarget $p
```

40.3 命令一览

通过TCL脚本可以配置一些公共参数：

```
ScanRate        # 折衷主机发送探测包的速率
ScanPort        # 易受攻击服务端口号
ScanPacketSize  # 蠕虫探测包的大小
```

缺省情况下，折衷主机随机地扫描Internet。我们也能通过设置局部扫描概率来模拟局部扫描蠕虫：

```
$w local-p 0.5
```

下面是抽象网络中配置参数的一些命令：

```
$w beta 0.1                    # 感染参数
$w gamma 0                     # 排除参数
$w addr-range 2000 200000      # 抽象网络的地址空间
$w dn-range 0 1999             # 具体网络的地址空间
$w v_percent 0.01              # 抽象网络中的易受攻击主机的百分比
```

第 41 章

PackMime-HTTP：网络流量生成

基于目前的Internet流量跟踪，PackMime互联网流量模型是由贝尔实验室的互联网流量研究组的研究员开发出来的。PackMime包括一个称为PackMime-HTTP的HTTP流量模型。由PackMime-HTTP产生的流量强度受*rate*参数控制，这个参数是每秒启动的新HTTP连接的平均数量。PackMime-HTTP在ns-2中实现，在UNC-Chaple Hill（北卡大学教堂山分校）被开发，能够生成HTTP/1.0和HTTP/1.1（持久的，非流水线的）连接。

PackMime-HTTP的目标不是模拟一个单一网络客户端和网络服务器之间的交互，而是模拟发生在一条由许多网络客户端和服务器共享的链路上的TCP级别流量。

一个典型的PackMime-HTTP实例包含ns节点：服务器节点和客户端节点。这些节点不会回应一个单一的网络服务器或者网络客户端，关于这点是很重要的。一个单一的PackMime-HTTP客户端节点产生来自网络客户端组成的“cloud（子网，或者由很多节点组成的连接？）”的HTTP连接。同样，一个单一的PackMime-HTTP服务器节点接收和服务于那些目的地是由网络服务器组成的“cloud（子网）”的HTTP连接。一个单一网络客户端由一个单一的PackMime-HTTP客户端应用程序表示，一个单一网络服务器由一个单一的PackMime-HTTP服务器应用程序组成。有许多客户端应用程序被指派给一个单一的客户端ns节点，许多服务器应用程序被指派给一个单一的服务器ns节点。

为了模拟每个连接的不同RTT、瓶颈链路和（或）丢失率，PackMime-HTTP经常在具有DelayBox（参见第22章）的连接中使用。DelayBox是一个在北卡大学教堂山分校开发的模块，目的是根据给定的分布，在流量中延迟和（或）丢弃包。在41.3节有更多一起使用PackMime-HTTP和DelayBox的信息。

PackMime-HTTP流量模型在下面的论文中有详细的描述：J. Cao, W.S. Cleveland, Y. Gao, K. Jeffay, F.D. Smith, and M.C. Weigle，“Stochastic Models for Generating Synthetic HTTP Source Traffic”，*Proceedings of IEEE INFOCOM*, Hong Kong, March 2004。

41.1 实现细节

PackMimeHTTP是一个驱动HTTP流量生成的ns对象。每个PackMimeHTTP对象控制着Application两种类型的操作，PackMimeHTTP服务器Application和PackMimeHTTP客户端Application。这些Application的每一个都连接到一个TCP Agent（Full-TCP）。**注意**：PackMime-HTTP仅仅支持Full-TCP代理。

每个网络服务器或客户端子网（cloud）由一个单一的ns节点表示，这个节点能够在同一时刻产生和消耗大量HTTP连接（图41.1）。对于每个HTTP连接，PackMimeHTTP建立（或者从不活动池指派，下面有描述）服务器和客户端Application和它们的关

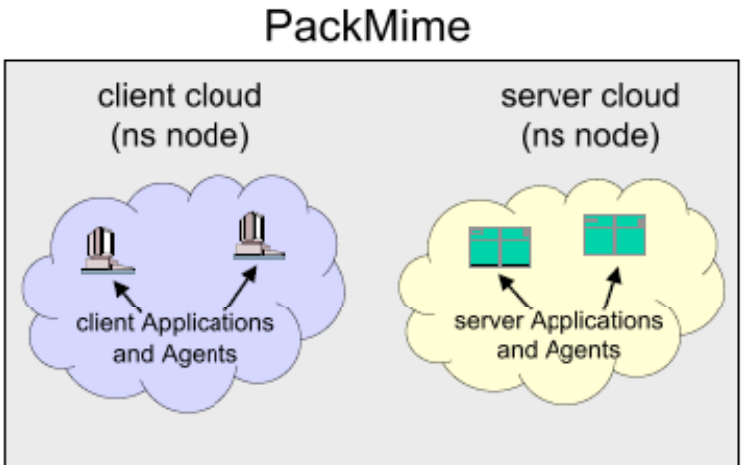


图41.1：PackTimeHTTP体系结构。每个PackMimeHttp对象控制一个服务器和客户端子网。每个子网由多个客户端或服务器Application表示。每个Application既表示一个单一的网络服务器也表示一个单一的网络客户端

联TCP Agent。在设置和开始每条连接之后，PackeMimeHTTP设置一个定时器在下一个新的连接应该开始的时候终止当前连接。新连接根据连接到达时间来启动，不用考虑以前请求的完成情况，但是相同客户端/服务器对（使用HTTP1.1）之间的新请求仅仅在以前的请求/回应对已经完成之后才开始。

PackMimeHTTP处理那些已经完成数据发送的Application和Agent的重新使用。有5个池（pool）使用来维护Application和Agent，其中，一个池处理不活动的TCP Agent，然后另外四个池分别处理活动、不活动的客户端和服务器Application。那些处理活动Application的池确保所有活动Application在模拟结束时被销毁。活动TCP Agent不需要放在一个池中，因为每个活动Application包含一个指向它关联的TCP Agent的指针。新的对象仅仅在当非活动池中没有有效Agent或Application时才建立。

41.1.1 PackMimeHTTP客户端应用程序

每个PackMimeHTTP客户端控制被传送HTTP请求的大小。每个PackMimeHttp客户端采用如下的步骤：

- 如果连接是继续的，并且由大于一个的请求组成，那么客户端为了连接抽取所有请求大小，回应大小和内部请求时间。
- 如果连接仅仅由一个请求组成，那么客户端抽取请求大小和回应大小。
- 发送第一个 HTTP 请求给服务器。
- 监听 HTTP 回应。
- 当接收到完整 HTTP 回应后，客户端设置定时器来在下一请求开始时终止当前请求。
- 当定时器终止时，下一个 HTTP 请求被发送，并且上面的过程将被重复，直到所有饿请求都完成了。

41.1.2 PackMimeHTTP服务器应用程序

每个网络服务器控制被传送请求的大小。当一个新的TCP连接开始后服务器被启动。每个PackMimeHTTP服务器采用如下的步骤：

- 监听来自相关客户端的 HTTP 请求。
- 当完整请求到达时，服务器从服务器延迟分布中抽取延迟时间。
- 当服务器超过延时时，设置定时器来终止。
- 当定时器终止后，服务器发送回应（回应大小由客户端抽取，并传递给服务器）。
- 这个过程将一直重复，直到请求处理完毕 – 服务器将知道有多少请求会在连接中被发送。
- 发送一个 FIN 来关闭连接。

41.2 PackMimeHTTP随机变量

对于PackMimeHTTP连接变量的特殊分布，PackMimeHTTP的实现提供了几个ns随机变量对象。这个实现采用的方法来自于贝尔实验室提供的源代码，修改后适应ns随机变量框架。这些允许PackMimeHTTP连接变量被任何类型的ns随机变量特制，它现在包括PackMimeHTTP特制随机变量。如果在TCL脚本中没有随机变量定制，那么PackMimeHTTP将自动设置这些。

TCL脚本中的PackMimeHTTP定制随机变量语法如下：

- $ns [new RandomVariable/PackMimeHTTPFlowArrive <rate>]，这里rate是特制PackMimeHTTP连接的速率（每秒新的连接数）
- $ns [new RandomVariable/PackMimeHTTPReqSize <rate>]，这里rate是特制PackMimeHTTP连接的速率
- $ns [new RandomVariable/PackMimeHTTPRspSize <rate>]，这里rate是特制PackMimeHTTP连接的速率
- $ns [new RandomVariable/PackMimeHTTPPersistRspSize]
- $ns [new RandomVariable/PackMimeHTTPPersistent <probability>]，这里probability是连接持续性的概率
- $ns [new RandomVariable/PackMimeHTTPNumPages <probability> <shape> <scale>]，这里probability是连接中有一个单一页面的概率，shape和scale是Weibull分布的参数，这决定连接中页面的数量
- $ns [new RandomVariable/PackMimeHTTPSingleObjPages <probability>]，这里probability是在当前页面中有一个单一对象的概率
- $ns [new RandomVariable/PackMimeHTTPObjsPerPage <shape> <scale>]，这里shape和scale是Gamma分布的参数，这决定在一个单一页面中有多少对象
- $ns [new RandomVariable/PackMimeHTTPTimeBtwnObjs]
- $ns [new RandomVariable/PackMimeHTTPTimeBtwnPages]

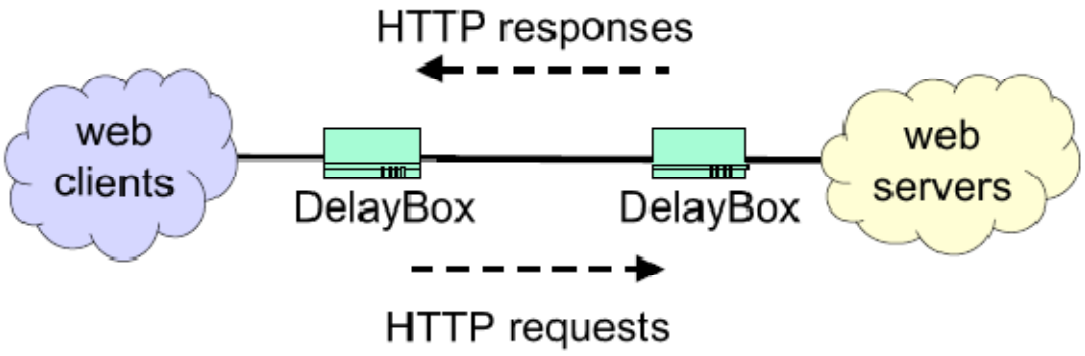


图41.2: 使用PackMimeHTTP和DelayBox的拓扑例子。网络客户端子网是一个单一的ns节点，网络服务器子网也是一个单一的ns节点。每个DelayBox节点是一个单一的ns节点

- $ns [new RandomVariable/PackMimeHTTPServerDelay <shape> <scale>], 这里shape和scale是Weibull分布的参数，这决定服务器的延迟
- $ns [RandomVariable/PackMimeHTTPXmit <rate> <type>], 这里对于客户端一方的延迟是type等于0，对于服务器一方的延迟type等于1。**注意：**这个随机变量仅仅在具有DelayBox的连接中使用。它返回实际延迟的1/2，因为这以为着将在两个DelayBox节点使用，每个节点应该延迟包1/2个实际延迟

41.3 使用DelayBox和PackMime-HTTP

PackMimeHTTP使用ns来模仿一条模拟链路上客户端和服务器之间的TCP层交换。使用DelayBox（见22）模拟通过子网传送HTTP的网络层效果。DelayBox是ns中虚拟网络模拟量，在网络实验中经常用于延迟和丢弃包。延迟时间模仿从源到子网边缘（或者子网边缘到目的）的传播和排队延迟。因为所有PackMimeHTTP中的HTTP连接仅仅发生在两个节点之间，所有必须有一个ns对象来延迟每个流中的包，而不是仅仅在两个节点间的链路上有一个静态延迟。DelayBox也模拟单个连接基础上的瓶颈链路和包丢失率。两个DelayBox节点的使用如图41.3所示。一个节点一个放在由ns节点表示的网络客户端子网前面来处理客户端方的迟延、丢失率和链路瓶颈。另外一个DelayBox放在由ns节点表示的网络服务器子网前面来处理服务器方的延迟、丢失率和链路瓶颈。

41.4 示例

更多的例子（包括那些展示DelayBox和PackMime使用的）可以在tcl/ex/packmime/目录下的ns源代码中找到。有效脚本test-suite-packmine.tcl在tcl/test/目录中，可以在哪个目录中使用命令test-all-packtime来运行。

注意：仅仅PackMime-HTTP必须设置的参数是rate，client，server，flow_arrive，req_size和rsp_size。下面的例子显示了必须设置的最少参数，但其他的参数能被设置用来改变缺省动作（参见“命令一览”）。

```
# test-packmime.tcl
#  有用的常量
set CLIENT 0
set SERVER 1
remove-all-packet-headers;          #  去掉所有的包头
add-packet-header IP TCP;           #  增加TCP/IP包头
set ns [new Simulator];             #  建立模拟实例Simulator
```

```
$ns use-scheduler Heap;                    # 使用Heap（堆）方案

# 建立拓扑
# create nodes
set n(0) [$ns node]
set n(1) [$ns node]
# create link
$ns duplex-link $n(0) $n(1) 10Mb 0ms DropTail

# 建立PACKMIME
set rate 15
set pm [new PackMimeHTTP]
$pm set-client $n(0);                      # 把节点$n(0)作为客户端
$pm set-server $n(1);                      # 把节点$n(1)作为服务器
$pm set-rate $rate;                            # 每秒产生一个新连接
$pm set-http-1.1;                          # 使用HTTP/1.1协议

# 设置PACKMIME随机变量
global defaultRNG

# 建立RNGs（适当的RNG种子被自动分配）
set flowRNG [new RNG]
set reqsizeRNG [new RNG]
set rspsizeRNG [new RNG]

# 建立RandomVariables
set flow_arrive [new RandomVariable/PackMimeHTTPFlowArrive $rate]
set req_size [new RandomVariable/PackMimeHTTPFileSize $rate $CLIENT]
set rsp_size [new RandomVariable/PackMimeHTTPFileSize $rate $SERVER]

# 分配RNGs给RandomVariables
$flow_arrive use-rng $flowRNG
$req_size use-rng $reqsizeRNG
$rsp_size use-rng $rspsizeRNG

# 设置PackMime变量
$pm set-flow_arrive $flow_arrive
$pm set-req_size $req_size
$pm set-rsp_size $rsp_size

# 记录HTTP统计信息
$pm set-outfile "data-test-packmime.dat"
$ns at 0.0 "$pm start"
```

```
$ns at 30.0 "$pm stop"
$ns run
```

41.5 命令一览

下面是从OTcl可以访问PackMimeHTTP类的命令：

[new PackMimeHTTP]
建立一个新PackMimeHTTP对象

$packmime start
开始产生连接

$packmime stop
停止产生新连接

$packmime set-client <node>
把节点node和PackMimeHTTP客户端子网联系在一起

$packmime set-server <node>
把节点node和PackMimeHTTP服务器子网联系在一起

$packmime set-rate <float>
设置每秒启动的新连接平均数量

$packmime set-req_size <RandomVariable>
设置HTTP请求大小的分布

$packmime set-rsp_size <RandomVariable>
设置HTTP回应大小的分布

$packmime set-flow_arrive <RandomVariable>
设置两个连续启动连接之间的时间

$packmime set-server_delay <RandomVariable>
设置网络服务器获取页面的延迟

$packmime set-run <int>
为了用于随机变量的RNGs使用相同的子流，设置运行号（详细说明参加第25章的RNG）

$packmime get-pairs
返回已完成的HTTP请求/应答对数量，tcl/ex/packmime/pm-end-pairs.tcl是一个在一定数量的请求/应答对完成后，使用get-pairs结束模拟的例子

$packmime set-TCP <protocol>
设置在客户端和服务器子网中所有连接的TCP类型（Reno，Newreno或者Sack），缺省类型是Reno。

和HTTP/1.1相关的特殊命令

$packmime set-http-1.1
对持续连接，使用HTTP/1.1代替HTTP/1.0

$packmime no-pm-persistent-reqsz
缺省时，PackMime-HTTP设置持续连接中所有请求大小一样。但是这个命令不这样设置持续连接中的每个请求，而是从请求大小的分布中抽取一个新的请求大小。

$packmime no-pm-persistent-rspsz
缺省时，PackMime-HTTP使用算法（参见packmime_ranvar.h中的方法PackMimeHTTPPersistRspSizeRandomVariable::value()了解细节）设置持续连接中的应答大小。这个操作不这样动作，而是为持续连接中的每个应答从应答大小分布中抽取应答大小

$packmime set-prob_persistent <RandomVariable>
设置持续连接的概率

$packmime set-num_pages <RandomVariable>
设置每个连接的页面数量

$packmime set-prob_single_obj <RandomVariable>
设置页面包含单一对象的概率

$packmime set-objs_per_page <RandomVariable>
设置每个页面中对象的数量

$packmime set-time_btwn_pages <RandomVariable>
设置页面请求之间的时间间隔（也就是，思考时间）

$packmime set-time_btwn_objs <RandomVariable>
设置对象请求之间的时间间隔

和输出相关的特殊命令

$packmime active-connections
向标准错误输出设备输出当前活动HTTP连接的数量

$packmime total-connections
向标准错误输出设备输出已完成HTTP连接的总数量

$packmime set-warmup <int>
设置输出的开始的时间。仅仅用来设置输出文件

$packmime set-outfile <filename>
输出下面的字段（每个HTTP请求/应答对的一行）到输出文件filename：

- HTTP应答完成的时间
- HTTP请求大小（字节）
- HTTP应答大小（字节）
- HTTP应答时间（ms），此时间是客户端发送HTTP请求到收到已完成的HTTP答应的时间
- 源节点和端口分类器
- 这个HTTP请求/应答对完成时的活动连接数量

$packmime set-filesz-outfile <filename>
发送一个应答后，输出下面的字段（每个HTTP请求/应答对的一行）到输出文件filename：

- HTTP应答发送的时间

- HTTP请求大小（字节）
- HTTP应答大小（字节）
- 服务器节点和地址

$packmime set-samples-outfile <filename>

发送一个请求之间，输出下面的字段（每个HTTP请求/应答对的一行）到输出文件filename：

- HTTP请求发送时间
- HTTP请求大小（字节）
- HTTP应答大小（字节）
- 服务器节点和端口地址

$packmime set-debug <int>

设置调试级别：

- 1：输出在模拟结束时建立的连接总数
- 2：Level 1 +

输出TCP代理和应用程序的建立/管理

在一个新连接开始时输出

客户端发送的字节数量和期望的应答大小

服务器发送的字节数量

- 3：Level 2 +

当TCP代理和应用程序被移到到pool时输出

- 4：Level 3 +

输出每次客户端或者服务器接收到一个包时的所接收字节数量

第七篇

规模（Scale）

第 42 章

会话级（Session-level）分组分发

本章讲述ns中会话级分组分发实现的内部。本章分为两个部分：第一部分概述了会话配置（42.1小节），并”完全”描述了一个会话的配置参数。第二部分描述了结构，内部，以及会话级分组的分发代码路径。

本章描述的过程与函数均可以在~ns/tcl/session/session.tcl中找到。

会话级分组分发导向完成大型拓扑结构的多播模拟。对一些使用会话级模拟的拓扑结构的内存要求是：

2048 个节点, 连通度 = 8 ≈ 40MB

2049–4096 个节点 ≈ 167MB

4097–8194 个节点 ≈ 671MB

然而要注意的是，会话级模拟忽略了查询时延。因此，模拟所使用的高速率数据源或使用的网络内某点上聚集的复合源，其模拟的确切性受到怀疑。

42.1 配置

会话级模拟的配置包含两个部分：会话级细节本身的配置（42.1.1小节）和添加丢失和错误模式到会话级抽象模拟具体的行为（42.1.2小节）。

42.1.1 基本配置

基本配置包括创建和配置一个多播会话。每个会话（例如，一个多播树）必须严格按如下顺序配置：(1)创建和配置一个会话源;(2)创建一个会话助手（helper）并把它系在会话源上；(3)最后，让会话成员加入会话。

```
set ns [new SessionSim]                          ;# 前置初始化
set node [$ns node]
set group [$ns allocaddr]
set udp [new Agent/UDP]              ;# 创建和配置源
$udp set dst_ $group
set src [new Application/Traffic/CBR]
$src attach-agent $udp
$ns attach-agent $node $udp
$ns create-session $node $udp              ;# 创建helper并附在源上
set rcvr [new Agent/NULL]                        ;# 配置接收器
$ns attach-agent $node $rcvr
$ns at 0.0 "$node join-group $rcvr $group"          ;# 将会话连接起来
$ns at 0.1 "$src start"
```

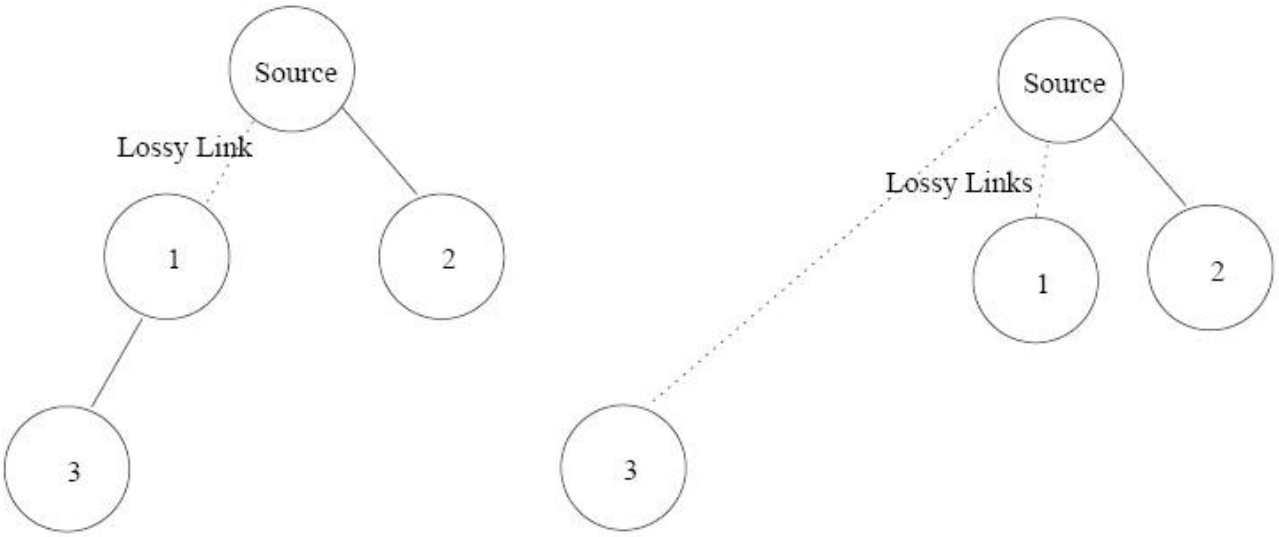


图41.1 两种多播树的对比

一个会话级模拟依据将拓扑转化为虚拟网格（virtual mesh）拓扑来刻画。做这些涉及到的相关步骤为：

1. 所有的分类器和复制器都被删除。每个节点只存储实例变量，以用来跟踪节点的ID以及端口的ID。
2. 链接不包含多重组件。每个链接只存储实例变量，以跟踪带宽和时延特性。
3. 包含链接的拓扑结构被转化为一个虚拟的网格。

图42.1显示了一个细节模拟中的多播树和一个会话级模拟中的多播树之间的差异。注意转化处理过程导致一个会话级模拟忽略排队时延。对于大多数模拟，ns已经忽略了所有节点上的处理时延。

42.1.2 插入丢失（loss）模块

当研究一个协议（譬如，SRM错误恢复机制）时，在丢失链路上研究协议可能是有用的。然而，既然一个会话级的模拟通过抽象出内部拓扑而刻画，我们就需要附加的机制来适当地插入一个丢失模块。这一小节描述了怎样创建这些loss模块来对错误场景进行建模。

创建一个Loss模块 在我们能够将loss模块插入一个源 - 接收器对（source-receiver pair）之前，我们必须首先创建这个模块。一个loss模块主要通过比较两个值来决定是否丢弃一个分组。第一个值是在每次loss模块从一个随机变量接收到一个分组时被赋值的。第二个值是固定的，并且在loss模块创建时被配置。下述代码给出了一个创建固定的丢包率为0.1的例子。

```
# 创建统一的分发随机变量
set loss_random_variable [new RandomVariable/Uniform]
$loss_random_variable set min_ 0                    ;# 设定随机变量的范围
$loss_random_variable set max_ 100
set loss_module [new ErrorModel]                    ;# 创建error模型
$loss_module drop-target [new Agent/Null]
$loss_module set rate_ 10                           ;# 设定错误率为0.1 = 10 / (100 – 0)
$loss_module ranvar $loss_random_variable           ;# 将随机的var.附在loss模块上
```

随机变量分布的目录已经在前面（第25章）描述过。对error模型更为详细的讨论也已经在前面不同的章节（第13章）进行描述过。

插入一个loss模块 一个loss模块只能在相应的接收器已经加入组后才能被插入。下面的代码解释了一个模拟脚本怎样引入一个loss模块。

```
set sessionhelper [$ns create-session $node $src]    ;#为loss模块保存一个句柄
$ns at 0.1 "$sessionhelper insert-depended-loss $loss_module $rcvr"
```

42.2 体系结构

会话级分发的目的是在保持一定精度的同时，加快模拟速度并减少内存的消耗。第一个瓶颈（bottleneck）是在链路和节点很多的情况下，内存的消耗也变得很大。因此，在SessionSim（针对会话级分组分发的模拟器）中，我们只保留链路和节点的最少数量的状态，同时使用适当的延迟和loss模块来把高层的源和接收器应用连接起来。

一个组中特定的数据源发送它的数据分组到复制器（replicator），这个replicator负责为所有的接收器复制分组。在replicator和接收器中间的loss和delay模块，保证了合适的端对端特性。也就是说，一个会话级的模拟将拓扑，路由以及排队时延都抽象化了。在SessionSim中的分组，不需要路由。它们只是沿着已经建立好的会话，进行分发。

42.3 内部

这一节介绍会话级分组分发的内部。我们首先配置一个简单的会话级模拟所用的OTcl脚本（42.3.1小节）；接着我们对分组转发是怎样实现的做一个简要的解释，作为总结（42.3.2小节）。

42.3.1 对象链接

我们将介绍在ns中构造会话级模拟的三个方面：修改网络拓扑规则，允许抽象节点和链路的生成；为每个活动的数据源建立会话helper；为会话添加接收器，在接收器加入适当的组后，再插入适当的loss和delay模块。

节点和链路 节点只包含它自己的节点id和针对下一个代理的端口号。链路只包含它自己的带宽和延迟的值。

```
SessionNode instproc init {} {
$self instvar id_ np_
set id_ [Node getid]
set np_ 0
}
SessionSim instproc simplex-link { n1 n2 bw delay type } {
$self instvar bw_ delay_
set sid [$n1 id]
set did [$n2 id]
set bw_($sid:$did) [expr [string trimright $bw Mb] * 1000000]
set delay_($sid:$did) [expr [string trimright $delay ms] * 0.001]
}
```

会话Helper 每个会话中的活动的数据源都需要一个“会话helper”。Ns中的会话helper是通过一个replicator来实现的。

会话helper是当用户使用create-session{}标识源代理的时候创建的。模拟器本身在它的实例变量数组（session_）中保存了一个指向会话helper的指针，session_以源和源的目的地址作为索引。

注意必须在调用create-session{}之前就设定数据源代理的目的端。

```
SessionSim instproc create-session { node agent } {
$self instvar session_
set nid [$node id]
set dst [$agent set dst_]
set session_($nid:$dst) [new Classifier/Replicator/Demuxer]
$agent target $session_($nid:$dst)          ;# 将replicator附在源上
return $session_($nid:$dst)    ;# 在SessionSim实例变量数组session_中保存一个replicator
}
```

Delay和Loss模块 在一个组里每个接收器都需要一个delay模块，用以反映它针对与特定数据源的时延情况。当接收器加入到一个组中时，join-group{}标识出session_中所有的会话helper。如果目的索引与接收器所加入的组地址匹配的话，将执行下列动作：

1. 会话helper的一个新的时隙被建立，并被分配至接收器。
2. 例行程序计算在源和接收器间总共的带宽和时延，这是通过SessionSim的实例过程get-bw{}以及get-delay{}来完成的。
3. 创建一个常量随机变量；它将使用累计时延作为平均时延的估计值来生成随机发送时间。
4. 通过端到端的带宽特性创建一个新的delay模块，随机变量生成器提供时延的估计值。
5. 插入的delay模块首先插入会话helper当中，也就是插入会话helper和接收器之间。
   在接收器中插入loss模块的情况也类似，请参考42.1.2小节。

```
SessionSim instproc join-group { agent group } {
$self instvar session_
foreach index [array names session_] {
set pair [split $index :]
if {[lindex $pair 1] == $group} {
# 注：必须以这个顺序插入一系列loss, delay以及目标代理。
$session_($index) insert $agent             ;# 向session复制器中插入目标代理
set src [lindex $pair 0]                    ;# 查找accum. b/w和delay
set dst [[$agent set node_] id]
set accu_bw [$self get-bw $dst $src]
set delay [$self get-delay $dst $src]
set random_variable [new RandomVariable/Constant]          ;# 设定时延变量
$random_variable set avg_ $delay
set delay_module [new DelayModel]                          ;# 配置时延模块
$delay_module bandwidth $accu_bw
$delay_module ranvar $random_variable
$session_($index) insert-module $delay_module $agent    ;# 插入时延模块
}
}
}
```

42.3.2 分组转发

分组分发行为在C++中执行。一个数据源应用产生了一个分组，并将它转发到它的目的地，这个目的地必须是一个复

制器（session helper）。复制器复制了分组并转发至活动时隙中的目的地，要么是时延模式，要么是丢失模式。如果是丢失模式，则作出是否丢弃分组的决定，如果决定丢弃，则分组被转发到丢失模式的丢弃目的地；如果决定不丢弃，则丢失模拟将它转发至目的地（必须是时延模式）。时延模式将有时延地把分组转发至其目的地（必须是一个接收装置）。

42.4 命令一览

下面是与session-level有关的命令列表：

set ns [new SessionSim]
该命令创建了一个session模式的模拟器的实例。

$ns_ create-session <node> <agent>
该命令创建了并系上了一个session-helper，它主要是在节点上的代理的一个复制器。

第 43 章

Asim:近似的分析模拟（analytical simulation）

本章介绍一个快速的近似的网络模拟器，Asim。Asim使用近视的固定的点，解决了网络状态的稳定问题。总体结构如图43.1所示。用户输入一个常规的ns脚本，并打开sim标志。Asim能够对网络场景进行快速的近似性模拟，同时能够给用户显示路由器的丢包概率，也能够显示链路和数据流延迟和总的吞吐率。

特别地，ns支持如下链路/流量模型：

- Drop Tail队列
- RED队列
- FTP数据流中的成块TCP数据流
- 短周期的TCP数据流

Asim的数据结构通过ns的Tcl空间的一个模块访问，这是通过用户的Tcl脚本来实现的。在Asim执行时，执行结果能够通过Tcl的例行程序访问。为了在脚本中使用Asim，用户必须使用Simulator set useasim_ 1。

这个标志的缺省值为0。

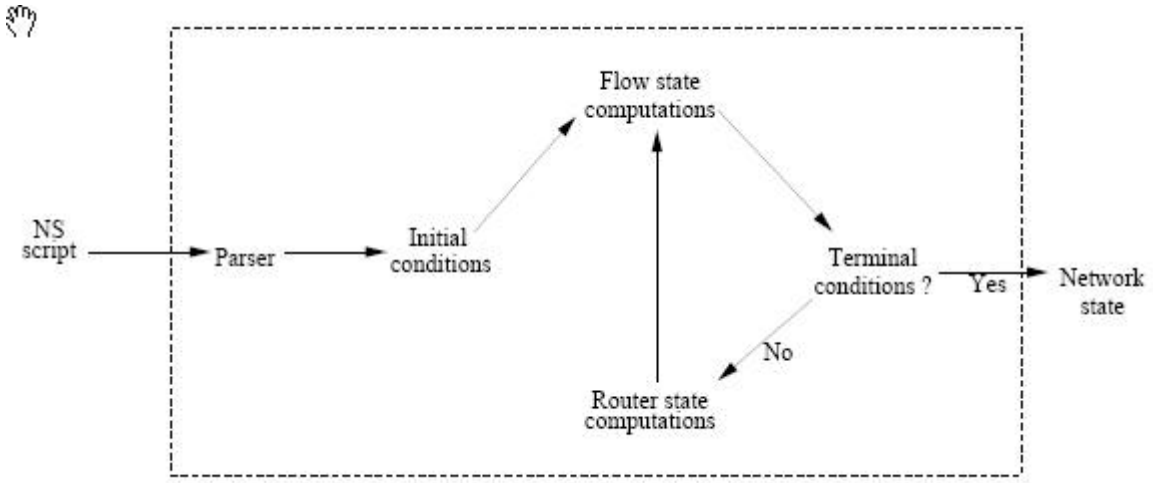


图43.1 Asim的结构

一个简单的脚本如下：

```
proc addsrc { s } {
global ns
set t [$ns set src_]
lappend t $s
$ns set src_ $t
}
proc adddst { src } {
global ns
set t [$ns set dst_]
```

```
lappend t $src
$ns set dst_ $t
}
proc finish {} {
global ns fmon
set drops [$fmon set pdrops_]
set pkts [$fmon set parrivals_]
set notDroped [$fmon set pdepartures_]
set overflow_prob [expr 1.0 * $drops / $pkts]
puts [format "tdrops $drops tpkts $pkts o_prob. %7.4f" $overflow_prob]
exit 0
}
set N_ 100000
set arrival 0
set available $N_
set endTime_ 200
set ns [new Simulator]
$ns set useasim_ 1
$ns at $endTime_ "finish"
set src_ ""
set dst_ ""
$ns set src_ $src_
$ns set dst_ $dst_
set n(0) [$ns node]
set n(1) [$ns node]
set link(0:1) [$ns duplex-link $n(0) $n(1) 1Mbps 50ms RED]
for {set i 0} { $i < 4} {incr i} {
set ltcp($i) [new Agent/TCP]
set ltcpsink($i) [new Agent/TCPSink]
$ns attach-agent $n(0) $ltcp($i)
$ns attach-agent $n(1) $ltcpsink($i)
$ns connect $ltcp($i) $ltcpsink($i)
set lftp($i) [new Application/FTP]
$lftp($i) attach-agent $ltcp($i)
$ns at 0 "$lftp($i) start"
}
# 短期数据流
addsrc 1
adddst 0
set pool [new PagePool/WebTraf]
#创建服务器和客户节点
$pool set-num-client [llength [$ns set src_]]
$pool set-num-server [llength [$ns set dst_]]
global n
```

```
set i 0
foreach s [$ns set src_] {
$pool set-client $i $n($s)
incr i
}
set i 0
foreach s [$ns set dst_] {
$pool set-server $i $n($s)
incr i
}
# 每个会话的页面数目
set numPage 100000
$pool set-num-session 1
set interPage [new RandomVariable/Exponential]
$interPage set avg_ 0.5
set pageSize [new RandomVariable/Constant]
$pageSize set val_ 1
set interObj [new RandomVariable/Exponential]
$interObj set avg_ 1
set objSize [new RandomVariable/Constant]
$objSize set val_ 20
# 这是必须的
$pool use-asim
$pool create-session 0 $numPage 0 $interPage $pageSize $interObj $objSize
# 将内部数据结构放入这个dump文件
$ns asim-dump dumpfile
# Calls asim-run
$ns asim-run
# 访问asim统计资料
set l [$ns link $n(0) $n(1)]
puts "after asim run, link bw = [$ns asim-getLinkTput $l] packets"
puts "after asim run, flow bw = [$ns asim-getFlowTput $ltcp(0)] packets"
```

第八篇

仿真（Emulation）

第 44 章

仿真

本章主要描述ns的仿真功能。仿真是指把模拟器引进到一个真实的网络上。模拟器中有专门的对象，它们有能力把真实的网络流量引入到模拟器中，并能够把处理过的流量发回到真实的网络中。

关于仿真器的注意点：

- 尽管用户不希望下面描述的接口发生剧烈的变化，但是这种仿真功能正处于发展当中，应该用实验的眼光来看待，受变化的影响。
- ns的仿真功能是在FreeBSD 2.2.5操作系统上开发的，作者并没有在其它系统上进行测试。
- 由于当前仿真功能的有限的可移植性，因此仿真功能被编译到nse中（使用”make nse”编译选项），而不是在标准的ns中。

44.1 概述

仿真功能可以细分为两种模式：

1. 不透明模式（opaque mode） – 将实时数据视作不透明的数据包
2. 协议模式（protocol mode） – 模拟器能够解释/生成实时数据

在不透明模式下，模拟器将网络数据视作未经解释（uninterpreted）的数据包。特别的是，真实网络中数据包的协议字段不能被模拟器直接的操作。在不透明模式下，实时数据可能会被模拟器丢弃，延迟，重新排序或者复制。但是因为没有执行协议的处理过程，所以一些特定协议的流量处理场景（例如：丢弃一个含有重传序列号为23045的TCP数据段）可能不会被执行。在协议模式下，模拟器能够解释并/或生成实时网络流量，包括任意相关的字段。**直到现在（1998.3），ns只实现了不透明模式下的仿真功能。**

一些对象的集合包括tap代理和网络对象提供了模拟器和实时网络之间的接口。Tap代理把真实的网络数据嵌入到模拟的数据包中，反之亦然。网络对象被安装在tap代理中，它们为真实数据的发送与接收提供了入口。两者（Tap代理和网络对象）都会在下面的小节中进行讲述。

当ns工作在仿真模式下，使用了一个特别版本的系统调度器：实时调度器。这个调度器使用了和标准的基本calendar-queue的调度器相同的底层结构，但是它能够实时地执行事件。描述如下：

44.2 实时调度器（Real-Time Scheduler）

实时调度器实现了将一个事件在真实时间上运行的软件调度调度。假使CPU的速度足够的快，能与到达的分组保持同步，那么模拟器的虚拟时间就会与真实时间很接近。如果模拟器太慢而落后于真实时间，模拟器的虚拟时间就会与真实时间有一个偏差。当这个偏差大于某个预告设定的常量“溢出系数”（slop factor）(当前是10ms)时，模拟器就会不断地产生警告。

调度器分发事件的主循环在文件scheduler.cc中的RealTimeScheduler::run()函数里，它本质上遵循下面的算法：

- 只要模拟器没有中断就执行下面的操作：
  - 获得当前真实时间（“now”）。
  - 分发所有时间戳在当前时间之前的未处理的模拟器事件。
  - 如果有下一个（将来的）事件到来，就取得该事件。
  - 延迟直到下一个模拟器事件准备好或有一个Tcl事件发生。
  - 如果有一个Tcl事件发生，重新把下一个事件插入模拟器事件队列并且返回到主循环开始处继续执行。
  - 否则，就分发模拟器事件，回到主循环开始处继续执行。
  - 如果没有将来的事件，就检查Tcl事件，并且返回到主循环开始处继续执行。

实时调度器一定要与仿真功能一起使用。否则就会导致模拟器的虚拟时间快于真实时间。在这种情况下，经过模拟网络的流量就不会被延迟适当的时间。使用实时调度器需要在模拟脚本的开始处做如下说明：

```
set ns [new Simulator]
$ns use-scheduler RealTime
```

44.3 Tap代理

TapAgent类一个由Agent基类派生的简单类。这样，它就能够生成模拟器的数据包，同时在ns的数据包公共（common）头中包含任意指定的值。Tap代理处理数据包公共头中包大小字段和包类型字段。它把插入到模拟器中的包的类型字段批定为PT _ LIVE。每个tab代理只能具有一个相关的网络对象，然而一个模拟器的节点中却可以实例化多个tap代理。

配置 Tap代理能够从一个相关的网络对象接收数据包，也能把数据包发送到一个相关的网络对象。假设$netobj代表一个网络对象，一个tap代理需要使用network方法进行配置：

```
set a0 [new Agent/Tap]
$a0 network $netobj
$a0 set fid_ 26
$a0 set prio_ 2
$ns connect $a0 $a1
```

注意流ID和优先级的配置是由Agent基类来处理的。在数据包公共头中设置流ID的目的是在真实数据的特定流中标识出属于它的数据包。针对这些数据包能够进行特别的处理，例如丢弃或重新排序，等等。Connect方法要求代理$a0通过模拟器的网络拓扑的当前路由发送实时流量到代理$a1。

44.4 网络对象

网络对象提供对真实网络的访问。根据要求访问真实网络协议层的不同和运行模拟器主机的操作系统所提供的服务的不同，网络对象分为几中不同的类别。要使用网络对象，需要具有特殊的访问权限，这些权限要事先设定好。一般来说，在一个真实网络的特定协议层（例如：链路层，IP层，UDP层等等），网络对象提供了一个入口和特殊的访问模式（只读，只写，可读写）。

一些网络对象提供了特殊的功能。如:对数据包进行过滤，或者以混杂模式接收网络数据包（例如:pcap/bpf网络对象），或者在一组节点间进行通信（例如UDP/IP的多播）的功能。C++中的Network类是所有派生的网络对象的基类。当前ns支持三种网络对象：pcap/bpf, raw IP，以及UDP/IP。下面分别介绍这几种。

44.4.1 Pcap/BPF网络对象

这些对象提供了LBNL包捕获程序库（libpcap）的扩展接口。（更多信息请参考ftp://ftp.ee.lbl.gov/libpcap.tar）这个库能在混杂模式下从网络接口驱动程序中捕获链路层的帧（例如：使用libpcap的程序也会得到数据包，这些数据包是从网卡驱动程序所获得的数据包中复制而来的）。它也提供了以”tcpdump”格式读写数据包跟踪文件的功能。如果网络接口驱动程序允许，ns所提供的扩展接口，就能把帧通过网络接口驱动程序写到真实的网络上。系统管理员可能会限制使用这个库来捕获和生成网络实时流量。如果你要使用这个库，你至少必须具有对系统包过滤服务的读的权限，这些权限都是管理员来指定的。

这个包捕获库在多个基于UNIX的平台上都能很好的运行。它特别针对Berkerley包过滤器（BPF）[25]，进行了优化。同时包捕获库也提供了一个对BPF伪机器码的过滤器编译器。由于大多系统支持BPF，一段驻留内核的BPF代码处理过滤条件，并对接收到的帧进行模式匹配。与过滤条件相匹配的帧通过BPF接收，与过滤条件不匹配的帧不受影响。BPF也能够发送链路层的帧。但一般不建议这样做，这是因为，一个正确初始化的帧一定要在被传递给BPF之前被创建。这就会导致无法为下一个目标地址指定正确的链路层的头部信息。一般来说，用“原始IP“（raw IP）网络对象发送IP包比较好，因为系统路由功能能够为链路层封装的头部指定正确的信息。

配置 Pcap网络对象可以被配置成与真实网络相关，或者与跟踪文件相关。如果与真实网络相关，则要指明将要用到的特定的网络接口，也要指明混杂模式标志。对所有的网络对象，可以用读或写的方法打开。这里有个例子：

```
set me [exec hostname]
set pf1 [new Network/Pcap/Live]
$pf1 set promisc_ true
set intf [$pf1 open readonly]
puts "pf1 configured on interface $intf"
set filt "(ip src host foobar) and (not ether broadcast)"
set nbytes [$pf1 filter $filt]
puts "filter compiled to $nbytes bytes"
puts "drops: [$pf1 pdrops], pkts: [$pf1 pkts]"
```

这个例子首先确定了本地系统的名字，用于构造BPF/libpcap过滤器的条件。第二行Network/Pcap/Live调用创建了pcap网络对象的一个实例，用于捕获实时流量。第三行promisc_标志告诉包过滤器把网络接口模式设置成混杂模式（如果底层网络接口支持的话）。第四行调用网络对象$pf1的方法open激活包过滤器，open的参数可以指定为只读，只写，或者可读写。Open方法返回与过滤器相关的网络接口的名字。对open方法也能传递额外的参数（我们没有阐述），用于指明想要使用的网络接口的名称。这在一个主机有多个接口的情况下使用。Filter方法创建了一个BPF兼容的包过滤条件，这个条件被加载到底层的BPF模块中。Filter方法返回过滤条件所使用的字节数。Pdrops和pkts方法用于统计包的个数。它们分别返回由于过滤器队列溢出而丢弃的包的总数和到达过滤器的包的总数（不是被过滤器接收的包的总数）。

44.4.2 IP网络对象

这些对象具有IP协议的原始访问能力，并且能够处理完整规范的IP数据包（包括包头部分）。这些对象使用一个原始的套接字（raw socket）来实现。在大多数UNIX系统上，要访问这些sockets，需要具有超级用户的权限。另外，raw sockets的接口不如其它类型的sockets标准。Network/IP类提供了原始IP的功能，并具有Network基类所具有的功能。所有实现更高层协议的网络对象都是从Network基类所派生的。

配置 原始的IP网络对象的配置是相当简单的。它与任何特定的网络接口都没有关联。在需要发生数据包时，可以使用系统的IP路由功能。在这些数据包的头部中包含了目的地址的网络接口的物理地址。这里有一个配置一个IP对象的例子：

```
set ipnet [new Network/IP]
$ipnet open writeonly
...
$ipnet close
```

IP网络对象只支持open和close方法。

44.4.3 IP/UDP网络对象

这些对象封装了系统UDP系统实现的功能，也提供了IP的组播功能。这个对象正在**开发中。**

44.5 示例

下面的代码展示了一个很小但却很完整的模拟脚本，在脚本中使用了BPF和IP网络对象，可以使用这个脚本来进行仿真测试。Ns在一个多接口的主机上运行，它具有路由能力，它能够从一个接口读取帧，并使这些帧通过模拟网络，最后使用原始的IP网络把它们再发到真实的网络上。

```
set me "10.0.1.1"
set ns [new Simulator]
$ns use-scheduler RealTime
#
# 我们希望测试机器禁用ip转发功能，所以检测这些（这就是至少在FreeBSD系统上如何去做）
#
set ipforw [exec sysctl -n net.inet.ip.forwarding]
if $ipforw
puts "can not run with ip forwarding enabled"
exit 1
#
# 分配一个BPF类型网络对象和一个raw-IP对象
#
set bpf0 [new Network/Pcap/Live]
set bpf1 [new Network/Pcap/Live]
$bpf0 set promisc_ true
$bpf1 set promisc_ true
set ipnet [new Network/IP]
set nd0 [$bpf0 open readonly fxp0]
set nd1 [$bpf1 open readonly fxp1]
$ipnet open writeonly
#
# 试着过滤出怪异的流量譬如： netbios pkts, arp requests, dns等等
# 同样，不要捕捉来自本机或广播的数据流
#
set notme "(not ip host $me)"
```

```
set notbcast "(not ether broadcast)"
set ftp "and port ftp-data"
set f0len [$bpf0 filter "(ip dst host bit) and $notme and $notbcast"]
set f1len [$bpf1 filter "(ip src host bit) and $notme and $notbcast"]
puts "filter lengths: $f0len (bpf0), $f1len (bpf1)"
puts "dev $nd0 has address [$bpf0 linkaddr]"
puts "dev $nd1 has address [$bpf1 linkaddr]"
set a0 [new Agent/Tap]
set a1 [new Agent/Tap]
set a2 [new Agent/Tap]
puts "install nets into taps..."
$a0 network $bpf0
$a1 network $bpf1
$a2 network $ipnet
set node0 [$ns node]
set node1 [$ns node]
set node2 [$ns node]
$ns simplex-link $node0 $node2 10Mb 10ms DropTail
$ns simplex-link $node1 $node2 10Mb 10ms DropTail
$ns attach-agent $node0 $a0
$ns attach-agent $node1 $a1
$ns attach-agent $node2 $a2
$ns connect $a0 $a2
$ns connect $a1 $a2
puts "okey"
$ns run
```

44.6 命令一览

下面是一些仿真相关的命令：

$ns_ use-scheduler RealTime

这个命令创建了一个实时调度器。注意，实时调度器应该总是和仿真功能一起使用，否则就会导致模拟网络的虚拟时间快于真实时间。

set netob [new Network/<network-object-type>]

这个命令创建了一个网络对象的实例。网络对象用于访问一个真实的网络。当前可用的几种网络对象的类型分别为：Network/Pcap/Live, Network/IP和Network/IP/UDP。关于网络对象的详细信息，请参考44.4小节。

第九篇

用Nam来可视化 - - 网络动画

第 45 章

NAM

45.1 简介

Nan 是基于 Tcl/Tk 的动画演示工具，用以观测网络模拟中的 traces 和真实世界中 trace 数据包。Nam 设计的理念是创建一个动画模拟工具 ,使其能够读取大批的动画演示数据并具有良好的扩展性 ,以适应不同网络环境下的场景可视化的要求。在此前提下，nam 应当能够从大量的 trace 文件中提取简单的动画演示命令。为了处理大量的动画数据信息，需要在计算机存储器中记录尽可能简化的信息，这些事件命令被记录在案以便能够随时调用。

进行 nam 演示的第一个步骤是创建 trace 文件，文件应当包括网络的拓扑信息，如 nodes，links 以及数据包的 trace 数据等等，详细格式信息见 46.1 小节。通常，trace 文件是在运行 ns 的过程中产生的，在 ns 网络模拟过程中，用户可以通过事件跟踪来确立拓扑信息、场景布局及数据跟踪。此外，其他的 application 也能够产生 trace 文件。

一但 trace 文件生成，就可以运用 nam 动画模拟了。Nam 启动，读取 trace 文件以后，就会产生拓扑结构并弹出一个窗口，默认标记停留在时刻 0(动画开始)，如果需要的话还可以对其布局重新设定。凭借其用户界面，nam 实现了各方面的动画演示控制，详细的功能介绍参照‘用户界面’部分。

尽管nam已经推出了一系列的版本并且性能也越来越稳定，但任然存在一些bug。如果您在使用中遇到任何bug或有什么好的性能改进的话，请发email至nsuser@isi.edu。

45.2 Nam命令选项

nam [-g <geometry>] [-t <graphInput>] [-i <interval>]
[-j <startup time>] [-k <intial socket port number>]
[-N <application name>] [-c <cache size>]
[-f <configuration file>] [-r initial animation rate]
[-a] [-p] [-S][<tracefile(s)>]

命令选项

-g 指明 nan 窗口的几何位置

-t 指定 nam 使用 tkgraph，并为 tkgraph 确定输入文件

-i [信息不一定精确]指定屏幕刷新率(单位:毫秒)，默认值为 50 毫秒(20 帧/秒)。

-N 为 nam 实例命名，它可能在此后用于对等的同步。

-c 进行反向演示时，缓存区所能存储活动对象的最大值(size).

-f 演示启动时的所载入的初始文件。在该文件中，用户可以定义在 trace 文件中将被引用的函数。例如在动态链路(dynamic links)中的'link-up'和'link-down'(详见$ns rtmodel 和 ns 目录下 tcl/ex/simple-dyn.tcl)。初始化例子可以在 ex/sample.nam.tcl 和 ex/dynamicnam.conf 找到。

-a 创建一个独立的 nam 实例。

-p 打印 trace 的文件输出格式

-s 开启 X 同步以便于图形的调试，仅用于 X 环境的 Unix 系统。

<tracefile> 文件名，包含用于动画模拟的跟踪数据。如果<tracefile>不能被读取，nam 将会打开<tracefile>.nam。

45.3 用户界面

启动 nam 后将会出现 nam 的主界面 ,你可以在同一个 nam 实例下运行多个 nam 动画。主界面的顶部是工具栏 ,有’ File’和’ Help’两个菜单。’ File’菜单下有;

’ New’命令，利用 nam 编辑器，创建 ns 拓扑；

‘Open’命令，打开已经存在的 nam 格式的 trace 文件；

‘Winlist’命令，打开 Active Animations 视窗列出所有已打开过的 trace 文件；

‘Quit’命令，退出 nam 模拟。

‘Help’菜单下有:

‘Help’,显示部分信息窗口中显示部分帮助信息；

‘About nam’ ，在显示信息窗口中显示版本、历史背景与版权信息。

当 trace 文件被加载到 nam 后(可通过’ Open’菜单或命令方式)，就会出现一个动画演示窗口：

File 菜单：

[Save layout]—用于保存当前的网络布局

[Print]—打印当前的网络布局。

Views 菜单：

[New view]—新增主显示窗口，所有的主显示窗口会同步显示。

[Show monitors]—在动画演示窗口中显示监测区。

[Show autolatout]—在动画演示显示窗口中显示自动布局设定区(autolayout)，当采用线路方位(link orientain)网络布局(layout)时,此选项无法勾选。

[Show annotations]—在动画演示窗口显示备注注解区。

在菜单工具栏下面有个控制列 ，包含六个按钮、一个标签和一个滑动条。它们可以随意点击，下面将从左到右解释它们的作用：

按钮 1 ’ <<’ ：倒回。点击后，以当前屏幕刷新率的 25 倍回放动画。

按钮 2 ‘<’ ：回放。点击后，逆时播放动画。

按钮 3 ‘■’：暂停。点击后，停止动画演示。

按钮 4 ‘>’ ：播放。点击后，顺时播放动画。

按钮 5 ‘>>’ ：快进。点击后，以当前屏幕刷新率的 25 倍播放动画。

按钮 6 ‘^’ ：关闭当前演示窗口。(译者注:貌似 2.30 版本没有此按钮)

时间标签：显示当前的演示时间(trace 文件中的模拟时刻)。

速率控制条：控制屏幕的刷新率(演示粒度)，当前的刷新率显示在滑动标签上方。

在控制列下面是主显示区，包括一个工具条和带有两个滚动条的主视图面板，所有由’ View/New view’命令生成的视图都包含以上三个组件。工具条包含两个缩放按钮：上箭头表示放大，下箭头表示缩小，两个滚动条用来滚动主动画视图。

在主显示区内左键单击任何一个对象(译者注:网络模拟中的节点)，将会弹出一个信息窗口，对于’ packet’ 和’ agent’ 对象，信息窗口包含有’ monitor’ 按钮，单击该按钮将会打开监视器面板并为该对象添加监视器；对于’ link’ 对象将会是’ Graph’ 按钮，单击出现新窗口可供选择带宽或链路丢失率(必须有事件提供此操作)。

对于目前讨论的用户界面，主显示区不是必要的，这取决于复选框’ Views/show monitors’ 是否被设置，默认情况是未设置。所有的监视器都会显示在这个区域里，一个监视器就像一个很大的按钮。目前只有’ packet’ 和’ agent’ 对象有监视器功能。

一个’ packet’ 监视器显示其大小、id、和发送时间。当 packet 到达目的地后，监视器并不会消失，但会提示该 packet 不可见；一个’ agent’ 监视器显示代理的名称，如果该代理绑定了 trace 变量，则该 trace 变量也会被显示。

在监视区下方(如果没有监视区，则在监视区出现的区域)，有一个看起来象刻度尺的时间滑动条，’ TIME’ 标签可以在’ 刻度尺’ 上移动，用于设置当前动画演示的时间。当拖动’ TIME’ 标签时，动画演示的时间将会显示在滑动条上。滑动条左端代表 trace 文件的开始时间，右端代表结束时间，点击滑动条的其它位置(非标签位置)，则可以产生与“倒回”或“快进”相同的效果。

自动布局设定区可以被隐藏或可见。如果可见的话，其出现在时间滑动条下方，它包括三个输入编辑框和一个自动布局按钮，三个输入框供设定两个自动布局常数和产生下次布局的反复次数。当在输入框下按下 ENTER 或’ relayout’ 按钮后，则开始按照设定的参数做反复动作。

Nam 窗口的底部是注解区，注解的形式是(time，sting)，用于描述事件及其发生时间。双击位于列表框的某个注解，动画模拟就会停留在注解标志的时刻。当鼠标停留在列表框内时，单击右键停止动画并弹出一个菜单，其中有三个选项：’ add’ 、’ delete’ 和’ info’ ，’ add’ 允许用户在弹出的对话框中添加注解，’ delete’ 删除当前注解，’ info’ 显示注解的细节—时间或事件等等。

45.4 键盘命令

窗口上大多数的按钮都有等效的快捷键，需要注意的是，只有当鼠标停留在 nam 窗口中时，这些快捷键才有效。

回车 —暂停 nam 动画，如果当前动画处于暂停状态，则步进一次。

‘**p**’ 或’ **P**’ —暂停，但没有步进功能。

‘**c**’ 或’ **C**’ —恢复动画放映。

‘**b**’ 或’ **B**’ —降低屏幕刷新率。

‘**r**’ 或’ **R**’ —倒回。

‘**f**’ 或’ **F**’ —快进。

‘**n**’ 或’ **N**’ —动画跳至下一个 trace 事件。

‘**x**’ 或’ **X**’ —取消上一次的刷新率变换。

‘**u**’ 或’ **U**’ —取消上一次的滑动条拖动。

‘**>**’ 或’ **.**’ —增大刷新速率 5%

‘**<**’ 或’ **,**’ —减小刷新速率 5%

空格键—对暂停和放映两种状态进行切换

‘**q**’ 、’ **Q**’ 或’ **Ctrl+C**’ —退出。

45.5 由Nam生成演示动画

Nam 可以保存并转化为 gifs 或 MPEG 格式的短片。

要保存你所想要的短片，首先启动 trace 动画，设置起始时间和其它一些参数(步进速率，长短等等)，从’ File’ 菜单中选择’ Record Animation’ 开始短片，演示的细节将会保存为一个名为” nam%d” 的 X-window 文件，其中’ %d’ 表示短片序号。经过后期处理，保存下来的一系列文件可以聚集成 GIF 或 MPEG 动画。

下面的 shell 脚本程序(sh,而不是 csh)将这些文件转化为 GIF：

```
for i in *.xwd; do
   xwdtoppm <$i |
   ppmtogif -interlace -transparent’ #e5e5e5’ > ‘basename $i .xwd ‘.gif;
done
gifmerge -l0 -2 -229,229,229 *.gif >movie.gif
```

需要注意的是xwdtoppm, ppmtogif, 和gifmerge不是NS自带的工具，前两个可以在http://download.sourceforge.net/netpbm/中下载，gifmerge可以在http://www.the-labs.com/GIFMerge/下载。

45.6 网络布局

在 Nam 演示中，可以通过边缘对象改变 node 对象以设定网络拓扑，但是想显示地表达网络拓扑还需要一个有效的布局机制。现行的 nam 提供了三种布局方法：

第一种方式通过链路方位来布局。链路方位一般用链路边缘和水平的夹角来表示，范围[0,2 π]。在布局的过程中，nam 会参照给定的链路方位，选择一个参考节点，然后依照方位和长度来对其它节点布局，链路的长度取决于时延和节点大小，这种布局方式适用于小型和手工生成的拓扑。

第二方式主要应用于随机拓扑的情况，自动生成网络布局。这种方式中，采用并实现了一种自动的图形布局算法。该算法的基本思想是将拓扑用一种模型来描绘，节点用球表示，链路用弹簧表示。球与球之间相互排斥，而弹簧起相反作用。进行多次迭代运算后，该算法将会收敛，实验发现，对于中小型的布局图，进行不多的迭代后(几十次或几百次)，将形成可视的、通用的布局结构。要实现大型的布局图，需要综合多个自动布局结果并手动配置。布局过程中需要调整三个参数：Ca 引力常数，用于控制球之间弹簧的力量；Cr 斥力常数，控制球之间的斥力；迭代次数，运行自动布局的次数。对于几十个节点的小型网络，使用默认参数(20-30 次迭代)就可以完成一个较好的布局。对于大型网络则需要仔细调整参数。以下是 Georgia Tech’ s ITM 网络拓扑建模者提出的经验方法，用于对 100 个节点的网络进行布局：首先，设定 Ca、Cr 值为 0.2，进行约 30 次迭代，然后设定 Cr 值为 1.0，Ca 值约为 0.01 再做 10 次迭代，最后设 Ca 为 0.5，Cr 为 1.0，进行 6 次迭代。

第三种方式为(x,y)坐标方式布局，它被用来进行无线网络拓扑构造，这种环境下不存在固定的链路，节点被加以(x,y)坐标值在坐标系中进行定位。

45.7 动画构件

Nam 主要运用以下实体构件进行动画模拟：

Node(节点) 在 trace 文件中，节点由’n’来标记，用于代表一个源、主机或路由。Nam 会略去对一个节点的重复定义，节点可以有三种情况(圆形、方形和六边形)，一旦节点被构造，就不能再改变形状，演示过程中还可以改变节点的颜色。

Link(链路) 链路构建于节点之间，用于形成网络拓扑。Nam 链路包含两条单向链路，trace 事件’1’创建两条单向链路并进行其它必要的设置。因此，从用户的角度来看，所有链路都是双向的，演示过程中链路也可以改变颜色。

Queue(队列) 队列应构建于节点之间，对应于双向链路的一条边，队列在演示中显示为堆积的数据包，数据包沿着一条直线堆积，这条线和水平线之间的角度，可以在队列的 trace 事件中规定。

Packet(数据包) 数据包在演示中显示为带箭头的方块，箭头的方向显示了数据包的流向，队列的数据包显示为小方块。有时数据包会从队列或链路中溢出，此时显示为坠落的方块并消失于屏幕中。不足的是埋在 nam 当前的设计中，方向演示过程无法显现溢出的数据包。

Agent(代理) 代理被用来区分节点的协议状态，它们都是特定于某个节点的，代理有特定的名字，用于唯一确定该代理。在 nam 中。它显示为一个小方块，位置与所连接的节点相近。

第 46 章

Nam Trace

Nam 是一种基于 Tcl/Tk 的演示 ns 仿真和跟踪真实数据包的动画工具。运行 nam 的第一步是要生成 nam trace 文件，该文件应当像数据 trace 文件一样包含拓扑信息(例如 node、link、queue 和 node connectivity 等等)。在本章将会介绍 nam trace 的格式和简单的 ns commands/API(用于产生拓扑布局和控制 nam 动画)。

Nam 设计的基础是：能够处理大量的 trace 数据，其原始数据能改变以适应不同的场景可视化。因而，nam 需要从文件中提取信息并存储尽可能简化且连贯的动画信息以适应不同的场景。

46.1 Nam Trace的格式

C++中的 trace 类同样适用于 nam trace，这而类的详细的描述见 26.3 节。方法 Trace::format()定义 nam 格式，这些格式被 nam trace 文件用于 ns 仿真中的可视化需求，Trace::format()的描述见第 26 章的 26.4 小节。如果已定义了宏 NAM_TRACE(默认在 trace.h 中定义)，那么下列的代码为 Trace::format()的一部分：

```
  if (namChan_ != 0)
          sprintf(nwrk_,
              "%c -t "TIME_FORMAT" -s %d -d %d -p %s -e %d -c %d
-i %d -a %d -x %s.%s %s.%s %d %s %s",
                          tt,
                          Scheduler::instance().clock(),
                          s,
                          d,
                          name,
                          th->size(),
                          iph->flowid(),
                          th->uid(),
                          iph->flowid(),
                          src_nodeaddr,
                          src_portaddr,
                          dst_nodeaddr,
                          dst_portaddr,
                          seqno,flags,sname);
```

Nam trace 文件都具有自己的基本格式，每一行都是一次 nam 事件，行首的字符定义了事件，随后的标签标志了该事件的其它可选项。每个事件都在新的行首字符处结束。

```
<event-type> -t <time> <more flags>...
```

针对不同的事件类型，时间标签后有不同标签。

在trace文件中分两个部分 静态初始化的配置事件和动画事件。有标志’ -t *’ 的事件都是配置事件并位于文件的开头。值得注意的是 nam 也可以在实时应用(realtime applications)中用流(stream)来反馈,更多信息见 ‘**Using Streams with Realtime Applications**’ 章节。

接下来将描述不同类事件和动画对象的 nam trace 文件格式。

46.1.1 初始化事件(Initialization Events)

Trace 文件的第一个部分一定包含初始化信息，所有的初始化事件都包含标识’ -t *’ ，表示这个事件在动画开始前需要解析。

Version(版本) 下面定义了 trace 文件可视化所需的 nam 版本：

V -t <time> -v <version> -a <attr>

通常只有一个版本在 trace 文件中给出，一般在文件的第一行。例如：V -t * -v 1.0a5 -a 0 标签-v 1.0a5 表示这个脚本需要的 nam 版本不低于 1.0a5。该事件的更多信息见文件 tcl/stats.tcl，在进程 nam_analysis 下。

Wireless(无线) 在 nam 中无线节点时需要无线初始化事件：

W -t * -x 600 -y 600

给出无线场景中的布局大小 宽为’ -x’ 高为’ -y’ ,更多信息见进程 infer-network-model 中的 animator.tcl 文件。

Hierarchy(分层) 分层地址信息定义如下：

A -t <time> -n <levels> -o <address-space size> -c <mcastshift> -a <mcastmask>
-h <nth level> -m <mask in nth level> -s <shift in nth level>

如果在仿真中有分层地址的话，trace 文件会给出详细信息。

‘-n <levels>’ 表示分层的总级数，’ 1’ 表示平面，’ 2’ 表示 2 层...依次类推；’ -o <address-space size>’ 表示地址总的位数；’ -h <nth level>’ 细化了分层地址的等级；’ -m <mask>’ 和’ -s <shift>’ 分别描述了地址的掩码和分层地址中的位数转移信息。下面是一个 3 级分层拓扑 trace 实例：

A -t * -n 3 -p 0 -o 0xffffffff -c 31 -a 1
A -t * -h 1 -m 1023 -s 22
A -t * -h 2 -m 2047 -s 11
A -t * -h 3 -m 2047 -s 0

更多信息见进程 nam_addressing 下 tcl/netModel.tcl 文件。

Color Table Entry (色表入口) 详细定义如下：

c -t <time> -i <color id> -n <color name>

Nam 中允许颜色名用整数来表示，这对带颜色的 packet 来说非常有用。一个 packet 的流 id 与相对应的色表入口映射来表示颜色。应注意的是颜色名必须在颜色数据库 X11(/usr/X11/lib/rgb.txt)中有相关记录。

除此之外，事件 node 和 link 布局也包括在初始化部分。

46.1.2 Nodes(节点)

描述 node 状态的 nam trace 如下：

n -t <time> -a <src-addr> -s <src-id> -S <state> -v <shape> -c <color> -i <I-color> -o<color>

‘**n**’表示 node 的状态；’ **-t**’表示时间；’ **-a**’表示地址；’ **-s**’表示 id；
’ **-S**’给出了节点的传输状态：·UP，DOWN 节点生效和失效；
·COLOR 节点颜色变化，如果颜色已给定，’ -c <color>’表示变换为新的颜色，’ -o’回复原来的颜色。·DLABEL 给节点添加标签，如果有 DLABEL，则’ -l <old-label> -L <new-label>’指老标签和当前标签。’ **shape**’给出了节点形状。Node 的标签颜色可以用’ **-i**’来指明。
‘**-v**’代表节点的形状，有’ • circle’ 、’ • box’ 、’ • hexagon’ 三种情况。

一个简单例子 :n -t * -a 4 -s 4 -S UP -v circle -c tan -i tan ,定义了一个地址和id为4的节点 ,其颜色及其标签的颜色是tan(棕褐色)，形状为圆形。

46.1.3 Links(链路)

描述 link 状态的 nam trace 如下：
l -t <time> -s <src> -d <dst> -S <state> -c <color> -o orientation -r <bw> -D <delay>

<color> 与<state>的涵义与上一节node中意思一样，’ -o’表示link的方向(link和水平的夹角)，已有的方向值有：
• up(上) • down(下) • right(右) • left(左) • up-right(右上)
• down-right(右下) • up-left(左上) • down-left(左下)
• angle between 0 and 2pi([0, 2π]之间任一角度值)
‘-r’ ,’ -d’ 分别表示带宽和延时。

一个简单的例子：l -t * -s 0 -d 1 -S UP -r 1500000 -D 0.01 -c black -o right

46.1.4 Queues(队列)

描述 queue 状态的 nam trace 如下：
q -t <time> -s <src> -d <dst> -a <attr>

queue 在在 nam 中是以直线形式显现的，表示 packet(小方块)的堆积。在 queue trace 事件中，’ -a’表示 queu line 的方向(queue line 与水平夹角，顺时针方向)。例如，下面的例子表述了一个与屏幕垂直且向上的的 queue(0.5 表示 queue line 的角度是π/2)： q -t * -s 0 -d 1 -a 0.5

46.1.5 Packets(数据包)

当 trace 行描述 packet 时，事件类型可能是：+ (enqueue),- (dequeue), r (receive), d (drop), 或 h (hop)。
<type> -t <time> -e <extent> -s <source id> -d <destination id> -c <conv> -i <id> <type>

h Hop: packet开始在链路上从<source id>传输至<destination id>并按接下来的路由传递。
r Receive: packet 完成传递并在终点开始接收。
d Drop:packet 在队列或链路中被丢弃，在此并不能确定是在队列或链路中丢弃，而是由丢弃的时间来确定。
+ Enter queue:由<source id>到<destination id>过程中进入队列。
- **Leave queue:** 退出队列。

其它标签的意思如下：
-t <time> 事件发生时间；-s <source id>；源节点；-d <destination id>目的节点；-p <pkt-type>数据包类型，详解章节26.5；-e <extent>数据包大小 ;-c <conv>该小节de会话id或flow-id ;-a <attr>数据包属性 ,用于颜色id ;**-x <src-na.pa> <dst-sa.na> <seq> <flags> <sname>**源节点和目的节点的入口地址，序列号，标识符，消息类型。
例如：-x {0.1 -2147483648.0 -1 ----- SRM_SESS}表示SRM消息由节点0发送。

不同协议对应不同的标识符：
-P <packet type> 字符串表示不同的包类型，如下所示：
TCP TCP数据包；
ACK 应答信号；
NACK 否定应答；
SRM SRM数据包；
-n <sequence number>给定包的序列号。

46.1.6 Node Marking(节点标记)

节点标记由同心圆、方形或六边形表示，其构造如下：
m -t <time> -n <mark name> -s <node> -c <color> -h <shape> -o <color>
或由 m -t <time> -n <mark name> -s <node> -X 删除。一旦节点被创建，形状就不可变(同心圆、方形或六边形)，它们由上述小写的字符串表示。节点标志在 nam trace 中表示如下：m -t 4 -s 0 -n m1 -c blue -h circle，表示节点 0 在事件 4.0 出为蓝色圆形，其名字为 m1.

46.1.7 Agent Tracing(代理跟踪)

代理跟踪被用于显现协议种类，它们与节点相连。代理跟踪有区别于其它代理的名字，名字是唯一的。
代理跟踪由以下格式表述：
a -t <time> -n <agent name> -s <src>

在 ns 中，代理有可能被解除，在 nam 中表述如下：
a -t <time> -n <agent name> -s <src> -X

例如，下面的例子表示在 0 时刻一个名为 srm(5)的代理连接到节点 0：
a -t 0.00000000000000000 -s 5 -n srm(5)

46.1.8 Variable Tracing(变量跟踪)

为显现连接在代理上的不同变量，可以用特征跟踪。通常我们可以用某一**特征**显示一个简单变量，例如，一个单个数值的变量，该数值是一个简单的不带空格的字符串。所有的**特征**需连接到代理之上，则在该代理上可以随时添加和删除这些**特征**，这些**特征**的trace表述如下：
f -t <time> -s <src> -a <agentname> -T <type> -n <varname> -v <value> -o <prev value>

标识 <type> 可以为：**V** 简单变量；**I** 列表；**s** 停止计时器；**u** 前向计时器；**d** 后退计时器。然而仅仅只有 V 在 ns 中是

可执行的，-v <value>给出了一个新的变量，变量的值是一个简单的 ASCII 字符串并符合 TCL 语言的字符串标准；列表值也符合 TCL 列表标准；计数器值是 ASCII 数字；-o <prev value>表示先前变量的值，在回放动画时会用到。

一个简单的**特征** trace 事件如下：

f -t 0.00000000000000000 -s 1 -n C1_ -a srm(1) -v 2.25000 -T v

删除**特征**的 trace 格式为：

f -t <time> -a <agent name> -n <var name> -o <prev value> -X

46.1.9 Executing Tcl Procedures and External Code from within Nam(在Nam中执行TCL代码和外部代码)

在 Nam 的 trace 文件中有一种特殊的情况来允许运行不同的 TCL 程序，该情况以’ V’ 来标识：v -t <time> -e <nam tcl procedure call>。

上述程序具有普遍性，在给出的时间中可以在一行(不超过 256 字符)几个不同的程序，中间要以空格来隔开。不同于其它的情况，标识命令和 TCL 调用命令非常重要。

下面是这种情况的几个例子：

Setting playback speed(设置重放速度)

通常可以在 nam 窗口中控制速率条来选择回放速度，trace 文件也可以通过 set_rate_ext 命令来设置回放速度：

v -t <time> -e set_rate_ext <time> <update-peers?>

例如：v -t 2.5 -e set_rate_ext 20ms 1

为了兼容早期的 nam 版本，也支持 set_rate 命令。可以用

10 × log10 *<time-in-seconds>*来代替直接指定步数大小，例如：

v -t 2.5 -16.9897000433602 1。

为了获取更多的可读文件，建议使用 set_rate_ext 命令。

Annotation(注解)

用于在给定的时间显示文本注释：

v -t <time> -e sim_annotation <time> <unique sequence id> <text to display>

例如：v -t 4 -e sim_annotation 4 3 node 0 added one mark，在 nam 中调用 TCL 函数 sim_annotation，将节点 0 的注释 one mark 添加到动画面板中，具体细节见 tcl/annotation.tcl 中 sim_annotation。

Node Exec Button(节点执行按钮)

在 nam 的版本中，1.0a10 及其后期的版本支持运行用户自定义的脚本和程序，通过点击 nam 上的节点按钮来执行这一功能。

v -t 1.0 -e node_tclscript <node_id> <button label> <tcl script>

该行表示：读取 trace 文件时，这一行向节点对象添加一个新的按钮，点击该按钮执行 TCL 脚本。例如：v -t 1.0 -e node_tclscript 2 "Echo Hello" {puts [exec echo hello]}：向节点 2 的信息窗口添加一个’ Echo Hello’ 按钮，当点击’ Echo Hello’ 按钮时，将会向终端输出 TCL 脚本的输出。

实现上述不同的 nam trace 功能可以在 *ns*/trace.cc, , *ns*/trace.h, *ns*/tcl/lib/ns-namsupp.tcl 中找到。

46.1.10 Using Streams for Realtime Applications(实时应用的流事件)

除了可以从文件中，nam 还可以从流如 STDIN 中读取数据，下面将给出一个简单的教程讲述如何将 nam 跟踪流发送给 nam 以操作实时数据。首先先介绍一点关于 nam 内部如何工作的背景知识，基本上可以认定 nam 是从 nam trace 文件中读取数据的。这个文件有特定的格式，每一行就是一个 nam 事件，第一个字符定义了事件的类型，后继的字符用来设定事件相关的选项，事件以换行符结束。一个 nam trace 文件分两部分，静态配置和活动事件，所有带’ -t*’ 的都是配置事件，必须同时发给 nam，以’ #’ 开始的行是注释行，注释行只能放在动画场景中。

首先你必须把数据输送到nam的STDIN中，并以’ -’ 标记。例如：% cat wireless.nam | nam – 则nam将会从STDIN中读取数据。

接下来是一个简单的无线活动的例子，脚本的初始部分带有’ -t*’ ，说明这是初始配置信息：

V -t * -v 1.0a5 -a 0
W -t * -x 600 -y 600

前两行用来初始化，从你的程序传到 nam 的头两行必须是这两行，V 表示所需的最低 nam 版本，W 说明脚本中包含有无线节点，场景大小范围为：高 y 宽 x

n -t * -s 0 -x 0.0 -y 0.0 -z 20 -v circle -c black -w
n -t * -s 1 -x 0.0 -y 200.0 -z 20 -v box -c black –w

接下来的是网络布局，第一行的 n 构造一个无线(-w)环路(-v)节点，它的 ID 是 0(-s 0)，位置在(0.0，0.0)(-x 0.0 –y 1.0)，大小是 20(-z),颜色是黑色(-c)。第二行是无线方形节点（-v box)，ID 为 1(-s 1),方位(0.0 200.0),每个节点有唯一的 ID 号，由标志’ -s’ 指明。

A -t * -n 1 -p 0 -o 0xffffffff -c 31 -a 1
A -t * -h 1 -m 2147483647 -s 0

A 事件与 nam 的分层结构有关，这在无线 nam 中是必须的，因为，每个 packet 都是广播的。

当 nam 的初始配置部分完成后，接下来就是动画事件。为了使 nam 能够实时运行，它必需要不断接受到新的更新，它会跟踪 nam 的每一行并能够返回。事件的进行必需按时间顺序，例如下面的例子是把节点 1 的颜色由黑色变为绿色然会回到黑色、最终变成黑色的过程：

n -t 0.0 -s 1 -S COLOR -c green -o black
n -t 0.01 -s 1 -S COLOR -c black -o green
n -t 0.02 -s 1 -S COLOR -c black -o black

需要注意的是’ -t <time>’ 的值是递增的，不可能在执行完’ -t 0.2’ 的事件后又执行’ -t 0.1’ 事件。Nam有一个内部的计时器，事件是和内部计时器一一对应的，只用顺时的事件才会被执行，所以下面的代码就不起作用： n -t 0.0 -s 1 -S COLOR -c black -o black

n -t 0.02 -s 1 -S COLOR -c green -o black
n -t 0.01 -s 1 -S COLOR -c black -o green

因为 nam 有自己的内部时间表示，但它的表示方法与现实的时间不同，所以必须尝试与 nam 的时间同步，现在并没有完全精确的方法与之同步，但可以用一个粗略的方法：周期性的向 nam 发送事件，该事件并不一定存在，为此构造一个 dump

或’ no-op’ 事件(T)来实现：

T -t 0.5 。

由上所述，你提供给 nam 的事件必须是时间上非递减的，在同一时间内也可以有一系列事件。在对一个时间内赋予活动以前，nam 需要知道它所获得的这个时间的所有事件，而它事实上可以在这个时间之后读取事件，因此如果按照实时的时间顺序来赋予事件的话，nam 处理事件将会有延时。为了更好的让 nam 来执行事件，需要制造 dummy 事件(中间时间槽)，由此 nam 可以继续。事件(T)就是 dummy 事件，这可能会产生不平稳的动画:

n -t 1.0 -s 1 -S COLOR -c green -o black
n -t 2.0 -s 1 -S COLOR -c black -o green
n -t 3.0 -s 1 -S COLOR -c black -o black

下面的将会平稳一些：

T -t 0.0
T -t 0.5
n -t 1.0 -s 1 -S COLOR -c green -o black
T -t 1.5
n -t 2.0 -s 1 -S COLOR -c black -o green
T -t 2.5
n -t 3.0 -s 1 -S COLOR -c black -o black
T -t 3.5
T -t 4.0
...

如果 nam 在一个事件流结束时阻塞，程序这时就象冻结了一样，屏幕画面就会静止不前，直到遇到另一事件，因此必须确保赋予 nam 的事件速度高于或等于 nam 读取事件速度。这个技术可以让 nam 看起来接近实时运行的事件，中间产生的微小时延是可以接受的。

还有一个需要注意的事情是:应用程序发送事件的时间和其开始的时间是相对应的 ,而且 ,发送给 nam 的事件不能被缓存，这样就不得不执行该事件。

另一个可以发送给nam的事件就是note了，它将会显示在nam窗口的底部：v -t 0.08 -e sim_annotation 0.08 1 Time is 0.08

v -t 0.09 -e sim_annotation 0.09 2 Time is 0.09
v -t 0.10 -e sim_annotation 0.08 3 Time is 0.10

‘V’ 事件将 packet 传送到传输队列;’ -’ 事件从队列中删除 packet 并准备广播该 packet ;’ h’ 表示将 packet 传送到下一跳(动画过程)。’ -t’ 时间 ;’ -s’ 传输节点 id ;’ -d’ 目的节点 id(‘-l’ 表示广播) ;’ -e’ 传输的大小 ;’ -c’ 最终的目的地。

将 packet 由一个节点发送到另一个节点表述如下：

r -t 0.255 -s 1 -d -1 -p MAC -e 512 -c 0 -a 0 -i 0 -k MAC
+ -t 0.255 -s 1 -d 0 -p AODV -e 512 -c 0 -a 0 -i 0 -k MAC
- -t 0.255 -s 1 -d 0 -p AODV -e 512 -c 0 -a 0 -i 0 -k MAC
h -t 0.255 -s 1 -d 0 -p AODV -e 512 -c 0 -a 0 -i 0 -k MAC
r -t 0.255 -s 0 -d 1 -p AODV -e 512 -c 0 -a 0 -i 0 -k MAC

首先节点 1 接受(‘r’)由无线广播传送的 packet，然后将该 packet 加入(‘+’)到节点 1 的输出队列中，然后从队列中删除(‘-’)packet 并准备发送给节点 0，然后节点被发送至下一跳(‘h’)节点 0,这将触发 packet 由节点 1 传送至节点 0 的动画，最终 packet 由节点 1 转移至节点 0.

更多的 nam 事件可以见 ns 手册的 nam 章节。

你也可以使用 unix 中的命令’tee’来保存 trace 事件，例如：

```
% cat wireless.nam | tee saved_tracefile.nam | nam –
```

在 nam 中有时会出现 bug 或 trace 文件的格式问题(自定义的格式以实现自己的需求，这些在 nam 中本来是没有的)。Nam 设计的一个原则是由 trace 文件来控制动画事件，这样使得 nam 更加灵活，但同时增加了用于生成 trace 文件的程序的复杂性。

46.1.11 Nam Trace File Format Lookup Table(Nam trace速查表)

下面是所有可能出现的 nam trace 事件代码及其标识列表，来源于源文件 parser.cc。你也可以运用’nam –p’命令生成自己的列表：
(译者注：具体的注解就不翻译了，相信大家都懂)

```
# : comment – this line is ignored(注释，该行可忽略)
T : Dummy event to be used in time synchronization(dummy事件，用于同步)
      -t    <time> time
n : node
   -t <time> time时间
   -s <int> node id节点id
   -v <shape> shape (circle, box, hexagon)
   -c <color> color
   -z <double> size of node
   -a <int> address
   -x <double> x location
   -y <double> y location
   -i <color> label color
   -b <string> label
   -l <string> label
   -o <color> previous color
   -S <string> state (UP, DOWN, COLOR)
   -L <string> previous label
   -p <string> label location
   -P <string> previous label location
   -i <color> inside label color
   -I <color> previous inside label color
   -e <color> label color
   -E <color> previous label color
   -u <string> x velocity
```

-U <string> x velocity
-V <string> y velocity
-T <double> node stop time
-w <flag> wireless node

l : link

-t <time> time
-s <int> source id
-d <int> destination id
-r <double> transmission rate
-D <double> delay
-h <double> length
-O <orientation> orientation
-b <string> label
-c <color> color
-o <color> previous color
-S <string> state (UP, DOWN)
-l <string l> label
-L <string> previous label
-e <color> label color
-E <color> previous label color

+ : enqueue packet

-t <time> time
-s <int> source id
-d <int> destination id
-e <int> extent
-a <int> packet color attribute id
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type

- : dequeue packet

-t <time> time
-s <int> source id
-d <int> destination id
-e <int> extent
-a <int> attribute
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type

h : hop

-t <time> time
-s <int> source id
-d <int> destination id
-e <int> extent
-a <int> attribute
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type
-R <double> wireless broadcast radius
-D <double> wireless broadcast duration

r : receive

-t <time> time
-s <int> source id
-d <int> destination id
-e <int> extent
-a <int> attribute
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type
-R <double> wireless broadcast radius
-D <double> wireless broadcast duration

d : drop line

-t <time> time
-s <int> source id
-d <int> destination id
-e <int> extent
-a <int> attribute
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type

E : session enqueue

-t <time> time
-s <int> source id
-d <int> destination id

-e <int> extent
-a <int> attribute
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type

D : session dequeue

-t <time> time
-s <int> source id
-d <int> destination id
-e <int> extent
-a <int> attribute
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type

P : session drop

-t <time> time
-s <int> source id
-d <int> destination id
-e <int> extent
-a <int> attribute
-i <int> id
-l <int> energy
-c <string> conversation
-x <comment> comment
-p <string> packet type
-k <string> packet type

a : agent

-t <time> time
-s <int> source id
-d <int> destination id
-x <flag> remove agent
-n <string> agent name

f : feature

-t <time> time
-s <int> source id
-d <int> destination id
-x <flag> remove feature
-T <char> type

-n <string> name
-a <string> agent
-v <string> value
-o <string> previous value

G : group

-t <time> time
-n <string> name
-i <int> node id
-a <int> group id
-x <flag> remove from group
L : lan link
-t <time> time
-s <int> source id
-d <int> destination id
-o <orientation> orientation
-O <orientation> orientation

m : mark node

-t <time> time
- n <string> name
-s <int> node id
-c <string> color
-h <string> shape (circle, square, hexagon)
-X <flag> remove mark

R : routing event

-t <time> time
-s <int> source id
-d <int> destination id
-g <int> multicast group
-p <packet source> packet source id or *
-n <flag> negative cache
-x <flag> this route timed out
-T <double> timeout
-m <string> mode (iif or oif)

v : execute tcl expression

-t <time> time
-e <tcl expression> tcl script

V : set trace file version

-t <time> time
-v <string> time
-a <int> time

N : use nam graph

W : wireless range

-t <time> time
-x <int> X

-y <int> Y
g : energy status – for future use
-t <time> time
A : hierarchical address space configuration – initilization only
-t <time> time
-n <int> hierarchy
-p <int> port shift
-o <hex> port mask
-c <int> mulitcast shift
-a <int> multicast mask
-h <int> hierarchy
-m <int> node shift
-s <int> node mask
c : color table configuration – initialization only
-t <time> time
-i <int> id
-n <string> color
q : create packet queue – initialization only
-t <time> time
-s <int> source id
-d <int> destination id
-a <orientation> orientation
X : layout lan
-t <time> time
-n <string> name
-r <double> rate
-D <double> delay
-o <orientation> orientation
-O <orientation> orientation

46.2 Ns commands for creating and controlling nam animations(创建和控制Nam的外部命令)

本节介绍了在 ns 中各种对象如节点、链路、对列、代理等操作 nam 动画的 API，这些 API 大部分在 *ns*/tcl/lib/ns-namsupp.tcl 中实现，具体的范例在 *ns*/tcl/ex/nam-example.tcl 中。

46.2.1 Node(节点)

节点在trace文件中以’n’开始创建，每一个节点都代表一个主机或路由，如果有重复定义的节点的话，nam将会终止，这是因为节点有自己的颜色、形状、标签、标签颜色、标签位置和添加/删除标记等特征。每个节点可以有三种可选形状：圆形(默认)、方形和六边形，一但节点创建，则形状不可变。不同的节点有不同的颜色，颜色在仿真中可以变化。下面的OTcl脚本用以设置节点特征，属于节点类的方法：

$node color [color] ;# sets color of node(设置颜色)
$node shape [shape] ;# sets shape of node(设置形状)
$node label [label] ;# sets label on node(设置标签)
$node label-color [lcolor] ;# sets color of label(标签颜色)
$node label-at [ldirection] ;# sets position of label(标签位置)
$node add-mark [name] [color] [shape] ;# adds a mark to nod(e添加标识)
$node delete-mark [name] ;# deletes mark from node(删除标识)

46.2.2 Link/Queue(链路/队列)

链路在节点之间创建形成网络拓扑，nam 链路是内在单向的，对用户不可见。Trace 事件’l’创建两条简单链路和其它必要步骤，因此对用户来说链路看起来是双向的，链路有很多可选颜色且在仿真中颜色可变。队列在两节点之间形成，队列连接到单向链路上，这点与链路不同。Trace 事件’q’仅仅创建一个单向链路，在 nam 中，队列是可见的堆积在节点上的数据包且在一条直线上，该直线与水平的夹角可在’q’中细化。创建不同链路特征的代码如下：

```
$ns duplex-link-op <attribute> <value>
```

<attribute>可以是以下的一种：方向、颜色、队列位置或标签。方向定义了链路与水平的夹角，可选的方向的值可以是角度值或文本如right(0), right-up(45), right-down (-45), left (180), left-up (135), left-down (-135), up (90), down (-90)。队列位置定义了队列线与水平的夹角，各特征定义脚本如下：

```
$ns duplex-link-op orient right ; #方向设置为右，link创建的顺序取决于该脚本#调用次序
$ns duplex-link-op color "green"
$ns duplex-link-op queuePos 0.5
$ns duplex-link-op label "A"
```

46.2.3 Agent and Features(代理和特征)

代理用来在节点上区别协议，它们通常与节点想关联，每个代理有自己的名字，这是它的唯一标识，通常显示为一个带名字的方框，连接到相关节点，下面是支持代理跟踪的命令：

```
$ns add-agent-trace <agent> <name> <optional:tracefile>
$ns delete-agent-trace <agent>
$ns monitor-agent-trace <agent>
```

一但创建代理后，就可以用nsagent的trace方法来创建代理的假定变量**特征**trace，例如，一下代码段创建SRM代理$srm(0)的变量C1_的trace：

```
$ns attach-agent $n($i) $srm(0)
$ns add-agent-trace $srm($i) srm(0)
$ns monitor-agent-trace $srm(0) ;# turn nam monitor on from the start
$srm(0) tracevar C1_
```

46.2.4 Some Generic Commands(其它命令)

$ns color <color-id>定义了颜色索引，一但创建，color-id 可以在 nam trace 中代替颜色名。

$ns trace-annotate <annotation>在nam中插入批注。注意: 如果<annotation>中含有空格键，则必须用双注解来批注，例如

$ns at $time ” $ns trace-annotate ” Event A happened” ” ，这个注解用于控制事件驱动的动画，显示在nam窗口。
$ns set-animation-rate <timestep>设定播放速率为给定步长，单位为秒，前缀可选(如，1代表1秒，2ms代表0.002秒)。

第十篇

其它(Other)

第 47 章

NS和NAM的教学用途

本章是关于ns和nam的教学用途。ns是一种离散事件驱动的模拟器，支持TCP协议的多种格式，包含多种单播（unicast）和多播（multicast）路由模型，以及多种不同的多播协议。它支持移动网络，包括局域网和卫星网络。同时它也支持像网页缓存（web caching）这样的应用。ns使用nam（一种由Tcl/Tk开发的动画工具）来可视化运行ns脚本所产生的packet踪迹。因此ns和nam结合使用的话，在教室环境下就可以很容易的验证不同的网络问题。在这一章节中，我们将主要讨论已经开发出来的教学用的脚本数据库。同时我们也将讨论如何使用nam来运行namtrace文件。

47.1 以教学为目的的NS使用

为了实现上面提到的在教室环境下使用ns的这个教学需求，我们已经开发出了一个基于网络的接口。这个网络接口由ns的脚本数据库来提供服务，使得可以实现教室环境下的验证或其它教学目的。可以在这里找到这个接口http://www.isi.edu/nsnam/script_inv。该网页也提供一个向库中上传或递交类似脚本的接口。因此尽管我们目前开始时在这个脚本库里仅有少数几个脚本，但我们希望在您的贡献下这个脚本库会越来越大。接下来这几段，我们将更多地讨论这个用于教学的脚本索引网页。

47.1.1 安装/创建/运行ns

为了运行前一小节提到的教学脚本，你的机器里需要有一个能够运行的ns版本。ns的主页位于http://www.isi.edu/nsnam/ns。参考http://www.isi.edu/nsnam/ns/nsbuild.html网页，下载并在你的机器上创建ns。如果你想了解编写/运行脚本相关的情况，参考ns新手指南，位于http://www.isi.edu/nsnam/ns/tutorial/index.html。

47.1.2 教学脚本库的页面

教学用的脚本的库页面位于http://www.isi.edu/nsnam/script_inv。在这里可以查找、浏览并下载一个或多个模拟脚本（或者一些经过或未经过模拟验证过的相关文件，如namtrace，screenshot，webpage describing）或者向库中递交模拟脚本。接下来的几段我们将讨论这两种情况的选项：

查询/浏览/下载NS脚本：你可以转到“Search database”页面，用关键字查询数据库。你也可以转到“View database”页面浏览整个数据库。目前都是非常基本的查询功能，我们希望随着数据库的增长来扩展查询功能。数据库中的每个脚本入口有如下信息：
脚本名称
作者姓名、作者的E-mail和主页（如果提供的话）
模拟描述
运行此脚本所需的ns版本
关于脚本的其它注解

脚本的种类　目前我们已有的脚本的种类包括：应用，传输（TCP及其它），路由（单播及多播），多播协议，队列管理，无线及其它（包含其它任何种类）。

其它相关的文件　如果作者把NamTrace文件，screenshot，webpage这些文件/信息连同脚本一起递交的话，那么在每个入口的右下角会有这些相关文件的链接。

为了下载任何脚本或任何相关文件，只要在文件名上单击弹出的下载对话框输入文件的下载路径即可。

向库中递交NS脚本　一旦你有适合教室环境下验证的模拟脚本，你可以向库中递交。为了能够顺利上传你的脚本，必须至少递交如下信息：

作者姓名
作者E-mail
投搞的脚本名称（以及脚本的位置路径）
脚本的简要描述
脚本所需的NS版本
脚本的种类

下面的信息你可以有选择和上述信息一起递交：

作者的主页
Namtrace文件（模拟脚本时产生的namdump）
Screenshot（nam屏幕的截图）
网页（指向你脚本文件的网页）
其它注解，如果有的话

重要提示：我们建议你把namtrace文件和脚本一起提供，因为许多用户为了可视化可能只用namtrace，而只有当他们想更改模拟时才需要下载脚本。

47.2 以教学为目的的NAM使用

为了使ns模拟可视化，需要安装NAM工具。你可以为你的平台简单地下载nam binary，也可以下载nam distribution，然后在机器中创建。获取nam binaries即nam源文件的链接为http://www.isi.edu/nsnam/nam，同时它也是nam的主页。在powerpoint中使用nam的步骤：打开powerpoint后，在“Slide Show”（顶层菜单可见）下单击action buttons”。选择你喜欢的按钮类型，这会在你的幻灯片中创建一个按纽。这个按钮会弹出一个“Action Setting”窗口。在窗口中，有一个“Run Program”，在这里你可以定义你的nam程序并运行。

参考书目

[1] C. Alaettino˘glu, A.U. Shankar, K. Dussa-Zeiger, and I. Matta. Design and implementation of MaRS: A routing testbed.
Internetworking: Research and Experience, 5:17–41, 1994.
[2] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang, Satish Kumar, StevenMcCanne,Reza Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu,
Haobo Yu, and Daniel Zappala. Improving simulation for network research. Technical Report 99-702b, University of
Southern California, March 1999. (revised September 1999).
[3] Paul Barford and Mark Crovella. Generating representative web workloads for network and server peformance evaluation.
In Proceedings of the ACM SIGMETRICS, pages 151–160, June 1998.
[4] L.S. Brakmo, S. O’ Malley, and L.L. Peterson. TCP vegas: New techniques for congestion detection and avoidance. In
Proceedings of the ACM SIGCOMM, pages 24–35, October 1994.
[5] L.S. Brakmo, S. O’ Malley, and L.L. Peterson. TCP vegas: New techniques for congestion detection and avoidance.
Technical Report TR 94 04, Department of Computer Science, The University of Arizona, Tucson, AZ 85721, February
1994.
[6] R. Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. Communications
of the ACM, 31(10):1220–1227, October 1988.
[7] Pei Cao and Chengjie Liu. Maintaining strong cache consistency in the World-Wide Web. In Proceedings of the IEEE
ICDCS, pages 12–21, May 1997.
[8] N. Christin, J. Liebeherr, and T. Abdelzaher. A quantitative assured forwarding service. In Proceedings of IEEE INFOCOM 2002, volume 2, pages 864–873, New York, NY, June 2002.
[9] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, Ching-Gung Liu, and L. Wei. An architecture for wise-area multicast
routing. Technical Report USC-SC-94-565, Computer Science Department, University of Southern California, Los Angeles, CA 90089., 1994.
[10] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of IP traffic: A study of the role of
variability and the impact of control. pages 301–313, Cambridge, MA, USA, August 1999.
[11] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and
application level framing. In Proceedings of the ACM SIGCOMM, pages 342–356, August 1995.
[12] H. T. Friis. A note on a simple transmission formula. Proc. IRE, 34, 1946.
[13] H. W. Hethcote. The mathematics of infectious diseases. SIAM Review, 42(4):599–653, October 2000.

[14] A. Heybey. Netsim Manual. MIT, 1989.
[15] R. Jain. The Art of Computer Systems Performance Analysis. John Wiley and Sons, Inc., 1996.
[16] Pierre L’ Ecuyer. Good parameters and implementations for combined multiple recursive random number generators.
Operations Research, 47(1):159–164, 1999.
[17] Pierre L’ Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In Proceedings
of the 2001 Winter Simulation Conference, pages 95–105, December 2001.
[18] Pierre L’ Ecuyer, Richard Simard, E. Jack Chen, andW. David Kelton. An object-oriented random number package with
many long streams and substreams. Operations Research, 2001.
[19] A. Legout and E.W. Biersack. PLM: Fast convergence for cumulative layered multicast transmission schemes. In Proceedings of the ACM SIGMETRICS, Santa Clara, CA, U.S.A., June 2000.
[20] J. Liebeherr and N. Christin. JoBS: Joint buffer management and scheduling for differentiated services. In Proceedings
of IWQoS 2001, pages 404–418, Karlsruhe, Germany, June 2001.
[21] J. Liebeherr and N. Christin. Rate allocation and buffer management for differentiated services. Computer Networks,
40(1):89–110, September 2002.
[22] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP congestion control. In Proceedings of the ACM
SIGCOMM, August 1996.
[23] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanov. TCP Selective Acknowledgement Options, RFC 2018 edition, 1996.
[24] S. McCanne and S. Floyd. ns—Network Simulator. http://www-mash.cs.berkeley.edu/ns/.
[25] S. McCanne and V. Jacobson. The bsd packet filter: A new architecture for user-level packet capture. pages 259–269,
January 1993.
[26] John Ousterhout. Scripting: Higher-level programming for the 21st century. IEEE Computer,
31(3):23–30,March 1998.
[27] S.K. Park and R.W. Miller. Random number generation: Good ones are hard to find. Communications of the ACM,
31(10):1192–1201, October 1988.
[28] Peter Pieda, Jeremy Ethridge, Mandeep Baines, and Farhan Shallwani. A Network Simulator, Differentiated Services
Implementation. Open IP, Nortel Networks, 2000.
[29] T. S. Rappaport. Wireless communications, principles and practice. Prentice Hall, 1996.
[30] D. Waitzman, C. Partridge, and S.E. Deering. Distance Vector Multicast Routing Protocol, RFC 1075 edition, 1988.