NS-3 manual note – 01 Random Variables

Posted on 07/03/2009 by xa_ceshi

The ns-3 network simulator

ns-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. ns-3 is free software, licensed under the GNU GPLv2 license, and is publicly available for research, development, and use.

ns-3 is intended as an eventual replacement for the popular ns-2 simulator. The project acronym “nsnam” derives historically from the concatenation of ns (network simulator) and nam (network animator).

以上是 NS3 官网的介绍，NS2 还没学好，NS3 就要出来了。。。而且 NS3 的设计初衷就是为了取代 NS2，所以还是早起的鸟儿有虫吃，哦，错了，应该是笨鸟先飞。先研究 manual，翻译的比较拙劣。说句公道话，原文写的也是有些晦涩-_-!。翻译的排版比较乱，后期会出一个完整的 pdf。

NS3 官方网站 http://www.nsnam.org

manual 原文 http://www.nsnam.org/docs/release/manual.html

注意 pdf 版的 manual 和 html 版的 manual 略有不同。

翻译进行时这个 manual 是基于 ns3-3.4 的，原文章节结构为

1. Random variables
2. Callbacks
3. Attributes
4. Object model
5. Real-Time Scheduler
6. Emulation
7. Packets
8. Sockets APIs
9. Node and Internet Stack
10. TCP models in ns-3
11. Routing overview
12. Wifi NetDevice
13. CSMA NetDevice
14. PointToPoint NetDevice
15. Creating a new ns-3 model
16. Troubleshooting

1. 随机变量

ns-3 包含内建的伪随机数生成器（built-in pseudo-random number generator (PRNG)）. 对于要求严格的用户，理解 PRNG 的功能、设置及用法是很重要的，便于判断对于他的研究该 PRNG 是否能够胜任。

————————————————————————————

1.1 概述

ns-3 的随机数是通过 class RandomVariable 的实例来提供的。

默认情况下，ns-3 的模拟过程使用固定的种子；如果在模拟过程中存在随机性，除非改变种子和/或运行次数，否则程序的每次运行都将产生相同的结果。

在 ns-3.3 和早期的版本中，默认情况下 ns-3 的模拟过程使用随机的种子；从 ns-3.4 开始做出了改变。

为了在若干次模拟中获得随机性，必须设置不同的种子或者设置不同的运行次数。在程序的起始处调用 SeedManager::SetSeed(uint32_t) 来设置种子；当要用相同的种子来设置运行次数时，才程序的起始处调用 SeedManager::SetRun (uint32_t)。参考章节 设置种子以及独立的重复实验

每个 ns-3 中使用的随机变量都有一个虚拟随机数发生器与之相关联；所有随机变量都是用固定的或者随机的种子（基于全局种子的使用(previous bullet)）；如果你想对同一个场景使用不同的随机数进行多次运行，请一定阅读一下章节：设置种子以及独立的重复实验.

关于 ns-3 中随机数功能的详细解释。

————————————————————————————

1.2 背景

模拟过程使用大量的随机数； 研究[cite]表明大多数网络模拟花费多达 50%的 CPU 时间来生成随机数。用户需要关注（伪）随机数的质量以及不同随机数流之间的独立性。

用户需要关注一些内容，比如：

随机数生成器的种子化以及某个模拟过程是否是确定性的，

如何获得彼此独立的随机数流，以及

随机数流循环一次的时间有多久。

这里我们引入一些术语：RNG 提供很长的（伪）随机数序列。这个序列的长度叫做 cycle length 或 period，在该循环长度或周期过后，RNG 将重复自身。该序列可以被划分为互不关联的 streams。RNG 的流是 RNG 的一个连续的子集或者块。例如，如果 RNG 周期的长度为 N，且这个 RNG 提供两个流，则第一个流可能使用前 N/2 个值，第二个流可能产生后边的 N/2 个值。一个重要的性质是：这两个流是互不关联的，同样，每个流可以被划分为互不关联的若干个 substreams。底层 RNG 希望能够产生有着非常长的循环长度的伪随机数序列，并以有效地方式将序列划分为流和子流。

ns-3 使用与 ns-2 相同的底层随机数生成器：the MRG32k3a generator from Pierre L’Ecuyer. 详细的描述可以再这里找到：

http://www.iro.umontreal.ca/~lecuyer/myftp/papers/streams00.pdf.

MRG32k3a 生成器提供 1.8×10^19 个独立的随机数流，其中每个流都由 2.3×10^15 个子流组成。每个子流的周期（i.e., the number of random numbers

before overlap）为 7.6×10^22。整个生成器的周期为 3.1×10^57。表 ref-streams 提供了流与子流是如何组合到一起的图形化表示。
类 ns3::RandomVariable 是底层随机数生成器的公共接口。当用户创建新的 RandomVariables（比如 UniformVariable，ExponentialVariable 等）时，将创建一个对象，该对象使用 随机数生成器的彼此独立的流中的某一个流。因此，每个具有类型 RandomVariable 的对象都在概念上有自己的 “虚拟的”RNG. 另外，每个 RandomVariable 都可以被设置为使用由 main stream 获得的 substreams 中的某个子流。
另一个可选的实现是：允许每个 RandomVariable 拥有自己的（被不同的种子种子化过的）RNG。但是，在这种情况下我们无法保证不同的序列将互不关联。因此，我们选择使用单个的 RNG 以及由该 RGG 获得的流与子流。
————————————————————————————
1.3 设置种子以及独立的重复实验
ns-3 的模拟过程可以被设置为来产生确定的或者随机的结果。如果模拟过程被设置为使用固定的、确定的种子以及相同的运行次数，那么每次运行的输出都应该是相同的。
在默认情况下，ns-3 的模拟过程使用固定的种子和运行次数。这些值存储在两个 ns3::GlobalValue 实例中： g_rngSeed 和 g_rngRun。
典型的情况是对某个模拟过程进行多次独立试验。以便根据大量的独立运行来计算统计信息。用户可以改变全局种子并再次运行该模拟过程，也可以推进 RNG 的子流的状态，这被称为增加运行次数。
类 ns3::SeedManager () 提供一个 API 来控制设置种子和运行次数的行为。这个设置种子和子流状态的设置必须在创建任何随机变量之前被调用。例如：

```
SeedManager::SetSeed (3);  // Changes seed from default of 1 to 3
SeedManager::SetRun (7);  // Changes run number from default of 1 to 7
// Now, create random variables
UniformVariable x(0,10);
ExponentialVariable y(2902);
...
```

设置新的种子和推进子流的状态，哪一个更好？无法保证流产生的两个随机种子不是互相重叠的（译者注：即有可能部分一致）。保证两个流不互相重叠的唯一办法是使用由 RNG 的实现提供的子流功能。 因此，使用子流功能来实现同一个模拟过程的多个彼此独立的运行。 换句话说，统计学上，设置独立的重复实验更严格的方法应该是使用固定的种子并改变运行次数。这样的实现考虑到使用子流时，独立的重复实验的最大值为 2.3×10^15。
为了方便使用，用户不必在程序中控制种子和运行次数，可以设置环境变量 NS_GLOBAL_VALUE ：

```
NS_GLOBAL_VALUE=”RngRun=3″ ./waf –run program-name
```

另一个控制的方法是传递一个命令行参数，因为是 ns3 的 GlobalValue，所以等价于如下：

./waf –command-template=”%s –RngRun=3″ –run program-name

或者，如果直接在 waf 之外运行程序：

./build/optimized/scratch/program-name –RngRun=3

上述的各种命令行用法使得由 shell 脚本通过传递不同的 RngRun 系数来运行大量不同的模拟变得很容易。

————————————————————————

1.4 类 RandomVariable

所有随机变量都应该派生自 类 RandomVariable。该基类提供了一些在全局层面设置随机数生成器的行为的静态方法。派生类提供了刻画随机特性的 API，这些随机特性由特殊的分布决定。

模拟过程中被创建的每一个 RandomVariable 都将被给予一个生成器，该生成器是来自于底层 PRNG 的一个新的 RNGStream。按照这种方法来使用，the L’Ecuyer implementation 考虑到了最多 1.8×10^19 个随机变量。每个随机变量在某个单个的实验中可以在出现重叠前产生多达 7.6×10^22 个随机数。

————————————————————————

1.5 基类的公共 API

以下摘录了一些 类 RandomVariable 的公共方法，这些方法处理 RNG 的全局设置和 RNG 的状态。

```
/**
* brief Set seeding behavior
*
* Specify whether the POSIX device /dev/random is to
* be used for seeding.   When this is used, the underlying
* generator is seeded with data from /dev/random instead of
* being seeded based upon the time of day.     Defaults to true.
*/
static   void UseDevRandom(bool udr = true);
/**
* brief Use the global seed to force precisely reproducible results.
*/
static void UseGlobalSeed(uint32_t s0, uint32_t s1, uint32_t s2,
uint32_t s3, uint32_t s4, uint32_t s5);
/**
* brief Set the run number of this simulation
*/
static void SetRunNumber(uint32_t n);
```

```
/**
* brief Get the internal state of the RNG
*
* This function is for power users who understand the inner workings
* of the underlying RngStream method used.  It returns the internal
* state of the RNG via the input parameter.
* param seed Output parameter; gets overwritten with the internal state
* of the RNG.
*/
void GetSeed(uint32_t seed[6]) const;
```

我们已经在上边描述过种子的设置了。

————————————————————————————

1.6 RandomVariable 的类型

目前提供如下随机变量，文档在 ns-3 Doxygen 中，或者阅读 src/core/random-variable.h。用户也可以通过从类 RandomVariable 派生，来创建自定义的随机变量。

```
class UniformVariable
class ConstantVariable
class SequentialVariable
class ExponentialVariable
class ParetoVariable
class WeibullVariable
class NormalVariable
class EmpiricalVariable
class IntEmpiricalVariable
class DeterministicVariable
class LogNormalVariable
class TriangularVariable
```

————————————————————————————

1.7 RandomVariable 对象的语义

RandomVariable 对象有值语义，这意味着他们可以按照值传递的方式传递给函数。他们呢还可以按照引用传递的方式传递给 const。随机变量不是派生自 ns3::Object，并且我们不使用 smart pointers 来管理他们。他们可以被分配在栈上，用户也可以显式控制任何分配在堆上的随机变量。

RandomVariable 对象还可以被 ns-3 的 attributes 使用，这表示可以通过 ns-3 的 attribute system 来给 RandomVariable 对象赋值。以下是一个关于 WifiNetDevice 的 propagation 模型的例子：

TypeId

```
RandomPropagationDelayModel::GetTypeId (void)
{
static TypeId tid = TypeId (“ns3::RandomPropagationDelayModel”)
.SetParent<PropagationDelayModel> ()
.AddConstructor<RandomPropagationDelayModel> ()
.AddAttribute (“Variable”,
“The random variable which generates random delays (s).”,
RandomVariableValue (UniformVariable (0.0, 1.0)),
MakeRandomVariableAccessor
(&RandomPropagationDelayModel::m_variable),
MakeRandomVariableChecker ())
;
return tid;
}
```

这里，ns-3 的用户可以通过 attribute system 改变这个延迟模型的默认随机变量（which is a UniformVariable ranging from 0 to 1）。

————————————————————————

1.8 使用其他 PRNG

目前没有关于替换底层随机数生成器的支持（例如：the GNU Scientific Library or the Akaroa package），欢迎提供程序包。

————————————————————————

1.9 More advanced usage

To be completed

————————————————————————

1.10 Publishing your results

当发布模拟结果时，你需要陈述的一个关键配置信息是你是如何使用随机数生成器的。

你使用了什么样的种子，what seeds you used,

如果没有使用默认的 RNG，那么你使用的 RNG 是什么，

彼此独立的运行时如何执行的，

对于大规模的模拟，你如何检验没有发生循环。

研究者应该对发布的结果提供足够的信息以使得别人能够再现他的结果，这一点是责无旁贷的。研究者还应该充分弄明白所使用的随机数是统计学上合法的，并在文章中陈述为什么做出这样的假定，这一点也是责无旁贷的。

————————————————————————

1.11 小结

让我们回顾一下在创建模拟过程时应该做些什么。

决定使用固定的种子还是随机的种子，默认情况使用随机的种子，

决定如何处理独立的重复实验，对于长时间的模拟，如果可行的话，确认自己没有产生多于循环长度的随机值，
当发布结果时，按照上述的指导来对随机数生成器的使用书写文档。
程序 samples/main-random.cc 包含一些用法的例子。

NS-3 manual note – 01 Random Variables

by gaocong

The ns-3 network simulator

ns-3 is a discrete-event network simulator for Internet systems, targeted primarily for research and educational use. ns-3 is free software, licensed under the GNU GPLv2 license, and is publicly available for research, development, and use.

ns-3 is intended as an eventual replacement for the popular ns-2 simulator. The project acronym “nsnam” derives historically from the concatenation of ns (network simulator) and nam (network animator).

以上是 NS3 官网的介绍，NS2 还没学好，NS3 就要出来了。。。而且 NS3 的设计初衷就是为了取代 NS2，所以还是早起的鸟儿有虫吃，哦，错了，应该是笨鸟先飞。先研究 manual，翻译的比较拙劣。说句公道话，原文写的也是有些晦涩-_-!。翻译的排版比较乱，后期会出一个完整的 pdf。

NS3 官方网站 http://www.nsnam.org

manual 原文 http://www.nsnam.org/docs/release/manual.html

注意 pdf 版的 manual 和 html 版的 manual 略有不同。

翻译进行时这个 manual 是基于 ns3-3.4 的，原文章节结构为

1. Random variables

2. Callbacks

3. Attributes

4. Object model

5. Real-Time Scheduler

6. Emulation

7. Packets

8. Sockets APIs

9. Node and Internet Stack

10. TCP models in ns-3

11. Routing overview

12. Wifi NetDevice

13. CSMA NetDevice

14. PointToPoint NetDevice

15. Creating a new ns-3 model

16. Troubleshooting

1. 随机变量

ns-3 包含内建的伪随机数生成器（built-in pseudo-random number generator (PRNG)）. 对于要求严格的用户，理解 PRNG 的功能、设置及用法是很重要的，便于判断对于他的研究该 PRNG 是否能够胜任。

————————————————————————————

1.1 概述

ns-3 的随机数是通过 class RandomVariable 的实例来提供的。

默认情况下，ns-3 的模拟过程使用固定的种子；如果在模拟过程中存在随机性，除非改变种子和/或运行次数，否则程序的每次运行都将产生相同的结果。

在 ns-3.3 和早期的版本中，默认情况下 ns-3 的模拟过程使用随机的种子；从 ns-3.4 开始做出了改变。

为了在若干次模拟中获得随机性，必须设置不同的种子或者设置不同的运行次数。在程序的起始处调用 SeedManager::SetSeed(uint32_t) 来设置种子；当要用相同的种子来设置运行次数时，才程序的起始处调用 SeedManager::SetRun (uint32_t)。参考章节 设置种子以及独立的重复实验

每个 ns-3 中使用的随机变量都有一个虚拟随机数发生器与之相关联；所有随机变量都是用固定的或者随机的种子（基于全局种子的使用(previous bullet)）；

如果你想对同一个场景使用不同的随机数进行多次运行，请一定阅读一下章节：设置种子以及独立的重复实验.

关于 ns-3 中随机数功能的详细解释。

————————————————————————————

1.2 背景

模拟过程使用大量的随机数； 研究[cite]表明大多数网络模拟花费多达 50%的 CPU 时间来生成随机数。用户需要关注（伪）随机数的质量以及不同随机数流之间的独立性。

用户需要关注一些内容，比如：

随机数生成器的种子化以及某个模拟过程是否是确定性的，

如何获得彼此独立的随机数流，以及

随机数流循环一次的时间有多久。

这里我们引入一些术语：RNG 提供很长的（伪）随机数序列。这个序列的长度叫做 cycle length 或 period，在该循环长度或周期过后，RNG 将重复自身。该序列可以被划分为互不关联的 streams。RNG 的流是 RNG 的一个连续的子集

或者块。例如，如果 RNG 周期的长度为 N，且这个 RNG 提供两个流，则第一个流可能使用前 N/2 个值，第二个流可能产生后边的 N/2 个值。一个重要的性质是：这两个流是互不关联的，同样，每个流可以被划分为互不关联的若干个 substreams。底层 RNG 希望能够产生有着非常长的循环长度的伪随机数序列，并以有效地方式将序列划分为流和子流。

ns-3 使用与 ns-2 相同的底层随机数生成器：the MRG32k3a generator from Pierre L'Ecuyer. 详细的描述可以再这里找到：
http://www.iro.umontreal.ca/~lecuyer/myftp/papers/streams00.pdf.
MRG32k3a 生成器提供 1.8×10^19 个独立的随机数流，其中每个流都由 2.3×10^15 个子流组成。每个子流的周期（i.e., the number of random numbers before overlap）为 7.6×10^22。整个生成器的周期为 3.1×10^57。表 ref-streams 提供了流与子流是如何组合到一起的图形化表示。

类 ns3::RandomVariable 是底层随机数生成器的公共接口。当用户创建新的 RandomVariables（比如 UniformVariable，ExponentialVariable 等）时，将创建一个对象，该对象使用 随机数生成器的彼此独立的流中的某一个流。因此，每个具有类型 RandomVariable 的对象都在概念上有自己的 "虚拟的"RNG. 另外，每个 RandomVariable 都可以被设置为使用由 main stream 获得的 substreams 中的某个子流。

另一个可选的实现是：允许每个 RandomVariable 拥有自己的（被不同的种子种子化过的）RNG。但是，在这种情况下我们无法保证不同的序列将互不关联。因此，我们选择使用单个的 RNG 以及由该 RGG 获得的流与子流。

————————————————————————————

1.3 设置种子以及独立的重复实验

ns-3 的模拟过程可以被设置为来产生确定的或者随机的结果。如果模拟过程被设置为使用固定的、确定的种子以及相同的运行次数，那么每次运行的输出都应该是相同的。

在默认情况下，ns-3 的模拟过程使用固定的种子和运行次数。这些值存储在两个 ns3::GlobalValue 实例中： g_rngSeed 和 g_rngRun。

典型的情况是对某个模拟过程进行多次独立试验。以便根据大量的独立运行来计算统计信息。用户可以改变全局种子并再次运行该模拟过程，也可以推进 RNG 的子流的状态，这被称为增加运行次数。

类 ns3::SeedManager () 提供一个 API 来控制设置种子和运行次数的行为。这个设置种子和子流状态的设置必须在创建任何随机变量之前被调用。例如：

```
SeedManager::SetSeed (3);  // Changes seed from default of 1 to 3

SeedManager::SetRun (7);  // Changes run number from default of 1 to 7

// Now, create random variables

UniformVariable x(0,10);

ExponentialVariable y(2902);

...
```

设置新的种子和推进子流的状态，哪一个更好？无法保证流产生的两个随机种子不是互相重叠的（译者注：即有可能部分一致）。保证两个流不互相重叠的唯一办法是使用由 RNG 的实现提供的子流功能。 因此，使用子流功能来实现同一个模拟过程的多个彼此独立的运行。 换句话说，统计学上，设置独立的重复实验更严格的方法应该是使用固定的种子并改变运行次数。这样的实现考虑到使用子流时，独立的重复实验的最大值为 2.3×10^15。

为了方便使用，用户不必在程序中控制种子和运行次数，可以设置环境变量 NS_GLOBAL_VALUE ：

```
NS_GLOBAL_VALUE=”RngRun=3″ ./waf –run program-name
```

另一个控制的方法是传递一个命令行参数，因为是 ns3 的 GlobalValue，所以等价于如下：

```
./waf –command-template=”%s –RngRun=3″ –run program-name
```

或者，如果直接在 waf 之外运行程序：

./build/optimized/scratch/program-name –RngRun=3

上述的各种命令行用法使得由 shell 脚本通过传递不同的 RngRun 系数来运行大量不同的模拟变得很容易。

————————————————————————————

1.4 类 RandomVariable

所有随机变量都应该派生自 类 RandomVariable。该基类提供了一些在全局层面设置随机数生成器的行为的静态方法。派生类提供了刻画随机特性的 API，这些随机特性由特殊的分布决定。

模拟过程中被创建的每一个 RandomVariable 都将被给予一个生成器，该生成器是来自于底层 PRNG 的一个新的 RNGStream。按照这种方法来使用，the L'Ecuyer implementation 考虑到了最多 1.8×10^19 个随机变量。每个随机变量在某个单个的实验中可以在出现重叠前产生多达 7.6×10^22 个随机数。

————————————————————————————

1.5 基类的公共 API

以下摘录了一些 类 RandomVariable 的公共方法，这些方法处理 RNG 的全局设置和 RNG 的状态。

/**

* brief Set seeding behavior

*

* Specify whether the POSIX device /dev/random is to

* be used for seeding. When this is used, the underlying

* generator is seeded with data from /dev/random instead of

* being seeded based upon the time of day. Defaults to true.

```
*/

static   void UseDevRandom(bool udr = true);

/**

* brief Use the global seed to force precisely reproducible results.

*/

static void UseGlobalSeed(uint32_t s0, uint32_t s1, uint32_t s2,

uint32_t s3, uint32_t s4, uint32_t s5);

/**

* brief Set the run number of this simulation

*/

static void SetRunNumber(uint32_t n);

/**

* brief Get the internal state of the RNG

*

* This function is for power users who understand the inner workings

* of the underlying RngStream method used.   It returns the internal

* state of the RNG via the input parameter.

* param seed Output parameter; gets overwritten with the internal state

* of the RNG.
```

*/

void GetSeed(uint32_t seed[6]) const;

我们已经在上边描述过种子的设置了。

————————————————————————————

1.6 RandomVariable 的类型

目前提供如下随机变量，文档在 ns-3 Doxygen 中，或者阅读 src/core/random-variable.h。用户也可以通过从类 RandomVariable 派生，来创建自定义的随机变量。

class UniformVariable

class ConstantVariable

class SequentialVariable

class ExponentialVariable

class ParetoVariable

class WeibullVariable

class NormalVariable

class EmpiricalVariable

class IntEmpiricalVariable

class DeterministicVariable

class LogNormalVariable

class TriangularVariable

————————————————————————————

1.7 RandomVariable 对象的语义

RandomVariable 对象有值语义，这意味着他们可以按照值传递的方式传递给函数。他们呢还可以按照引用传递的方式传递给 const。随机变量不是派生自 ns3::Object，并且我们不使用 smart pointers 来管理他们。他们可以被分配在栈上，用户也可以显式控制任何分配在堆上的随机变量。

RandomVariable 对象还可以被 ns-3 的 attributes 使用，这表示可以通过 ns-3 的 attribute system 来给 RandomVariable 对象赋值。以下是一个关于 WifiNetDevice 的 propagation 模型的例子：

```
TypeId

RandomPropagationDelayModel::GetTypeId (void)

{

static TypeId tid = TypeId (“ns3::RandomPropagationDelayModel”)

.SetParent<PropagationDelayModel> ()

.AddConstructor<RandomPropagationDelayModel> ()

.AddAttribute (“Variable”,

“The random variable which generates random delays (s).”,

RandomVariableValue (UniformVariable (0.0, 1.0)),

MakeRandomVariableAccessor
(&RandomPropagationDelayModel::m_variable),

MakeRandomVariableChecker ())

;
```

return tid;

}

这里，ns-3 的用户可以通过 attribute system 改变这个延迟模型的默认随机变量（which is a UniformVariable ranging from 0 to 1）。

————————————————————————————

1.8 使用其他 PRNG

目前没有关于替换底层随机数生成器的支持（例如：the GNU Scientific Library or the Akaroa package），欢迎提供程序包。

————————————————————————————

1.9 More advanced usage

To be completed

————————————————————————————

1.10 Publishing your results

当发布模拟结果时，你需要陈述的一个关键配置信息是你是如何使用随机数生成器的。

你使用了什么样的种子，what seeds you used,

如果没有使用默认的 RNG，那么你使用的 RNG 是什么，

彼此独立的运行时如何执行的，

对于大规模的模拟，你如何检验没有发生循环。

研究者应该对发布的结果提供足够的信息以使得别人能够再现他的结果，这一点是责无旁贷的。研究者还应该充分弄明白所使用的随机数是统计学上合法的，并在文章中陈述为什么做出这样的假定，这一点也是责无旁贷的。

————————————————————————————

1.11 小结

让我们回顾一下在创建模拟过程时应该做些什么。

决定使用固定的种子还是随机的种子，默认情况使用随机的种子，

决定如何处理独立的重复实验，对于长时间的模拟，如果可行的话，确认自己没有产生多于循环长度的随机值，

当发布结果时，按照上述的指导来对随机数生成器的使用书写文档。

程序 samples/main-random.cc 包含一些用法的例子。

NS-3 manual note – 02 Callbacks

Posted on 07/03/2009 by xa_ceshi

NS-3 manual note – 02 Callbacks

by gaocong

PS: 3.4 的 manual 很多章节都空缺，等原文更新后会补上。

2. Callbacks 回调
某些 ns-3 的新手对代码中广泛使用的编程习惯不熟悉：“ns-3 callback”。本章提供回调的设计动机、使用方法以及他的实现细节。

2.1 动机
2.2 使用回调 API
2.3 回调在 ns-3 中的位置
2.4 实现细节

————————————————————————————

2.1 动机
考虑有两个模拟模型 A 和 B，并希望使他们在模拟过程中互相传递信息。一种方法是使得 A 和 B 彼此知道对方，这样他们就可以彼此调用对方的方法。

```
class A {
public:
void ReceiveInput ( // parameters );
...
}
```

(in another source file:)

```
class B {
public:
void ReceiveInput ( // parameters);
void DoSomething (void);
...
```

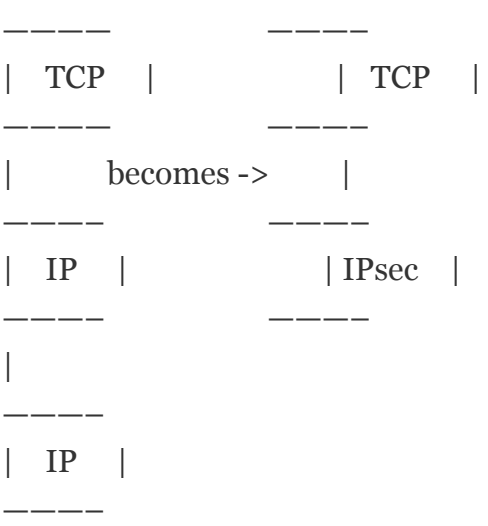
```
private:
A* a_instance; // pointer to an A
}

void
B::DoSomething()
{
// Tell a_instance that something happened
a_instance->ReceiveInput ( // parameters);
...
}
```

这样当然可行，但有缺点：为了使 A 和 B 彼此知道对方，在编译期间引入了 A 与 B 的依赖（这使得在模拟器很难获得独立的编译单元），并且不是一般化的。如果在后期的使用情景下，B 需要与另一个完全不同的对象 C 来交流，那么 B 的源代码就需要添加一个“c_instance”，等等。容易看出这是一种通信的蛮力机制，他将在模型中导致无用的、笨拙的编码。

但这并不表示对象不应该彼此知道对方，尤其是彼此之间存在牢固的依赖时。但如果他们之间的交流在编译期间限制较少的话，模型可以被制定的更加灵活。

这不是网络模拟研究的一个抽象问题，当研究者想要扩展或修改系统以便来做一些不同的事情时（as they are apt to do in research），它是先前一些模拟器的一个问题的源泉。例如，用户想要在 TCP 层和 IP 层之间加入一个 IPsec 安全协议子层：



如果模拟器被假定并编码为：IP 层始终与上层的传输层协议对话，那么用户就可能被迫对系统进行修改来获得需要内部连接。

提供这个灵活性的另一种方法是使用在编程中称为回调的间接手段。回调函数不是由调用者显式调用的，而是被委托给接收回调函数地址的另一个函数，由该函数来调用回调函数。

你可能熟悉 C 或 C++中的函数指针，函数指针可以被用来实现回调。更多关于回调的介绍可以在线获得： 声明函数指针并实现回调 和 回调（计算机科学）– Wikipedia。

ns-3 中的回调 API 被设计为使得模拟器的不同部件间的耦合度最小化，使每个模块都依赖于回调 API 而不是依赖于其他模块。回调作为一种第三方机制，任务被指派并转发到适当的目标模块。这些回调 API 是基于 C++模板的，是类型安全（type-safe）的，即执行静态类型检查来保证调用者与被调用者之间合适的签名兼容性。因此他比传统的函数指针要更加类型安全，但他的语法可能初看起来有些难度。本节来介绍回调系统，以便使你能够在 ns-3 中轻松地使用他。

————————————————————————————

2.2 使用回调 API
回调 API 很精简，他提供两类服务：

回调类型声明：用给定的签名声明回调的类型。
回调实例化：实例化由模板生成的转发回调，该回调能够将所有调用转发至另一个 C++的类成员方法或 C++函数。
最好通过例子 samples/main-callback.cc 来理解：

————————————————————————————

2.2.1 通过静态函数使用回调 API Using the Callback API with static functions
考虑函数：

```
static double
CbOne (double a, double b)
{
std::cout << “invoke cbOne a=” << a << “, b=” << b << std::endl;
return a;
}
```

考虑以下主程序片段：

```
int main (int argc, char *argv[])
{
// return type: double
// first arg type: double
// second arg type: double
Callback<double, double, double> one;
}
```

这个类模板回调实现了 Functor Design Pattern。他被用来声明回调的类型。他包含 1 个必须的参数(指派给该回调的函数的返回类型)和最多 5 个可选的参数，分别指定了参数的类型（如果你的函数包含的参数多于 5 个，可以通过扩展回调的实现来处理）。

我们在上述内容中声明了一个叫做”one” 的回调，该回调将保留一个函数指针。他保留的函数必须返回 double 型，并且必须支持两个 double 型参数。如果试图传递一个函数，该函数的签名不与所声明的该回调匹配，那么编译将失败。

现在我们需要将这个回调实例与实际的目标函数（CbOne）联系起来。注意 CbOne 的函数签名类型与回调的类型是相同的，这一点很重要。我们可以将类型符合的任意函数传递给该回调。让我们更严密地研究一下：

```
static double CbOne (double a, double b) {}
^          ^          ^
|        —|     ——|
|        |       |
Callback<double, double, double> one;
```

只有在签名匹配的情况下才能将函数与回调进行绑定。模板的第一个参数是返回值，其他参数是函数签名的参数的类型。

现在我们将回调”one” 与签名匹配的函数进行绑定：

```
// build callback instance which points to cbOne function
one = MakeCallback (&CbOne);
```

在后边的程序中如果需要用到回调，则使用方法如下：

```
// this is not a null callback
NS_ASSERT (!one.IsNull ());
// invoke cbOne function through callback instance
double retOne;
```

retOne = one (10.0, 20.0);
检查函数 IsNull() 确保该回调不为 Null，即该回调的背后存在一个函数来调用。函数 one() 返回的结果与直接调用函数 CbOne() 返回的结果是相同的。

————————————————————————————

2.2.2 通过成员函数使用回调 API Using the Callback API with member functions
通常情况下，你调用的是对象的公共成员函数，而不是静态函数。在这种情况下，MakeCallback 函数就需要一个额外的参数，该参数告诉系统 该函数应该在哪个对象上被调用。考虑如下来自 main-callback.cc 的例子：

```
class MyCb {
public:
int CbTwo (double a) {
std::cout << “invoke cbTwo a=” << a << std::endl;
return -5;
}
};

int main ()
{
...
// return type: int
// first arg type: double
Callback<int, double> two;
MyCb cb;
// build callback instance which points to MyCb::cbTwo
two = MakeCallback (&MyCb::CbTwo, &cb);
...
}
```

这里我们传递一个（裸）指针至函数 MakeCallback<> ，即当函数 two () 被调用时，调用由 &cb 所指的对象上的函数 CbTwo。

另一种变形使用是 当对象 被 ns-3 的 smart pointers 引用时，MakeCallback API 接受裸指针，所以我们需要调用函数 PeekPointer () 来获得该裸指针。则上述的例子将类似于：

```
class MyCb : public Object {
public:
int CbTwo (double a) {
std::cout << “invoke cbTwo a=” << a << std::endl;
return -5;
}
};

int main ()
{
...
// return type: int
// first arg type: double
Callback<int, double> two;
Ptr<MyCb> cb = CreateObject<MyCb> ();
// build callback instance which points to MyCb::cbTwo
two = MakeCallback (&MyCb::CbTwo, PeekPointer (cb));
...
}
```

————————————————————————————

2.2.3 构建 Null 回调

回调可以是空的，因此在使用他们之前进行检查是明智的。有一种构建空回调的方法，将”0″作为参数传递即可。MakeNullCallback<> 构建：

```
two = MakeNullCallback<int, double> ();
// invoking a null callback is just like
// invoking a null function pointer:
// it will crash at runtime.
//int retTwoNull = two (20.0);
NS_ASSERT (two.IsNull ());
```

————————————————————————————

2.3 回调在 ns-3 中的位置

回调通常在 ns-3 中的什么地方被用到？ns-3? 这里是一些对于典型用户比较容易可见的例子：

————————————————————————————

2.3.1 Socket API

————————————————————————————

2.3.2 Layer-2/Layer-3 API

————————————————————————————

2.3.3 Tracing subsystem

————————————————————————————

2.3.4 Routing
Route Reply

————————————————————————————

2.4 实现细节
本节为对实现感兴趣的 C++专家提供进阶解释，大多数用户可以略过。

代码编写最初基于这里描述的技术，它通常随着如下体系结构被重写： Modern C++ Design: Generic Programming and Design Patterns Applied– Alexandrescu, chapter 5, “Generalized Functors”。

代码使用：

默认的模板参数，这使得当实际的参数数量小于最大支持的参数数量时，用户不必指定空参数。
the pimpl idiom：类 Callback 被通过值来传递，并且类 Callback 将任务的关键指派给他的 pimpl pointer。
由 CallbackImpl FunctorCallbackImpl 派生的两个 pimpl 实现可以通过任意 functor-type 来使用，MemPtrCallbackImpl 可以通过指向成员函数的指针来使用。
引用列表来实现回调的值语义。
代码与 Alexandrescu 的实现显著不同，即没有使用类型列表来指定和传递回调

参数的类型。当然，也没有使用 copy-destruction 语义，并依赖于引用列表而不是 autoPtr 来保留指针。

NS-3 manual note – 03 Attributes

Posted on 07/03/2009 by xa_ceshi

NS-3 manual note – 03 Attributes

by gaocong

3. 属性

3.1 对象概述
3.2 属性概述
3.3 对属性进行扩展
3.4 给属性系统添加新的类
3.5 ConfigStore

在 ns-3 的模拟中，主要有两个方面的设置：

模拟拓扑和对象是如何连接的。
在拓扑中实例化的模型所使用的值。

本章的重点在于上述的第二项：ns-3 中使用的大量的值是如何组织的、记录的以及 ns-3 的用户如何来修改这些值。对于跟踪和统计信息是如何在模拟器中收集的，ns-3 的属性系统也起到了支柱作用。

在深入研究属性值系统之前，回顾 class ns3::Object 的一些基本属性会很有帮助。
————————————————————————————

3.1 对象概述
ns-3 本质上是一个基于 C++对象的系统。这意味着新的 C++类（类型）可以像往常一样被声明、定义以及子类化。

许多 ns-3 的对象继承基类 ns3::Object。这些对象有很多附加的属性，这些属性是我们为了对系统进行组织和以及改进对象的内存管理而开发的：

“metadata” 系统将类名与大量的 meta-information 进行链接，这些 meta-informatino 是与对象有关的，包括子类的基类、子类中可访问的构造函

数集以及子类的“attributes”集。
a reference counting smart pointer implementation, for memory management.
使用属性系统的 ns-3 对象是从 ns3::Object 或 ns3::ObjectBase 派生的。我们将要讨论的大多数 ns-3 对象派生自 ns3::Object，但 一些在 smart pointer 内存管理框架之外的对象派生自 ns3::ObjectBase。

让我们回顾一下这些对象的一些属性。
————————————————————————————

3.1.1 Smart pointers
正如 ns-3 tutorial 中介绍的，ns-3 的对象的内存管理师通过 reference counting smart pointer implementation，class ns3::Ptr 实现的。

Smart pointers 在 ns-3 的 API 被广泛的使用，来避免 将引用传递给堆分配的对象而可能引起内存泄漏。对于大多数基本的用法（语法），将 smart pointer 看待为常规指针：

```
Ptr<WifiNetDevice> nd = ...;
nd->CallSomeFunction ();
// etc.
```

————————————————————————————

3.1.2 创建对象
正如我们在@ref{Object Creation}中讨论过的，在最底层的 API 中，具有 ns3::Object 类型的对象不是像通常一样用 operator new 实例化的，而是通过一个叫做 CreateObject() 的模板函数实例化的。

创建这样的对象的典型方法如下：

```
Ptr<WifiNetDevice> nd = CreateObject<WifiNetDevice> ();
```
你可以认为这在功能上等价于：

```
WifiNetDevice* nd = new WifiNetDevice ();
```
由 ns3::Object 派生的对象一定是通过 CreateObject()被分配在堆上。而由 ns3::ObjectBase 派生的对象，比如 ns-3 的 helper functions and packet headers and trailers, 可以被分配在栈上。

在一些脚本中，你可能看不到大量的 CreateObject()调用，那时因为存在很多 helper objects，他们为你执行了 CreateObject()s。
————————————————————————————

3.1.3 TypeId
ns-3 中由类 ns3::Object 派生出的类可以包含一个叫做 TypeId 的元数据类，该类记录关于类的元信息，以便在对象聚合以及构件管理系统中使用：

识别该类的一个独一无二的字符串。a unique string identifying the class
在元数据系统中，子类的基类。
子类中 可访问的构造函数的集合。

————————————————————————————

3.1.4 对象小结
将所有这些概念综合在一起，让我们研究一个特定的例子：class ns3::Node。

公共头文件 node.h 中的一个声明包含了一个静态的 GetTypeId 函数调用：

```
class Node : public Object
{
public:
static TypeId GetTypeId (void);
...
```

该函数在文件 node.cc 中定义如下：

```
TypeId
Node::GetTypeId (void)
{
static TypeId tid = TypeId (“ns3::Node”)
.SetParent<Object> ()
;
return tid;
}
```

最终，当用户要创建节点时，可以这样调用：

```
Ptr<Node> n = CreateObject<Node> ();
```

下边我们会讨论属性（与类的成员变量以及成员函数相关联的值）是如何被加入

上述 TypeId 的。
————————————————————————————

3.2 属性概述
属性系统的目标是组织对模拟的内部成员对象的访问。这个目标的产生式因为：通常在模拟中，用户将剪切、粘贴或修改现存的模拟脚本，或者将使用更高层的模拟构造，但通常对研究或跟踪某些特别的内部变量很有兴趣。例如：

“我只想跟踪第一个接入点的无线接口上的包。”
“我想跟踪某个特定 TCP 套接字上 TCP 拥塞窗口的值（每次当它发生变化时）”
“我想获取并记录模拟中所有被使用到的值。”
类似地，用户可能想对模拟中的内部变量进行细致的访问，或者可能想广泛地改变某个特定参数的初始值，以便涉及到所有随后创建的对象。用户还可能希望知道在模拟配置中哪些变量是可以设置的和可以获得的。这不仅仅是为了命令行下直接交互，还考虑到（将来的）图形用户界面，该界面可能提供让用户在节点上右击鼠标就能获得信息的功能，这些信息可能是一个层次性组织的参数列表，显示该节点上可以设置的参数以及构成节点的成员对象，还有帮助信息和每个参数的默认值。
————————————————————————————

3.2.1 功能概述 Functional overview
我们给用户提供可以访问系统深处的值的方法，而不用在系统中加入存取器（指针）并通过指针链来获取想要的值。考虑类 DropTailQueue，该类有一个叫做 m_maxPackets 的无符号整型成员变量，该成员变量控制队列的大小。

查看 DropTailQueue 的声明，我们看到：

```
class DropTailQueue : public Queue {
public:
static TypeId GetTypeId (void);
...

private:
std::queue<Ptr<Packet> > m_packets;
uint32_t m_maxPackets;
};
```

考虑用户可能对 m_maxPackets 的值想要做的事情：

为系统设置一个默认值，以便无论何时一个新的 DropTailQueue 被创建时，这个成员变量都被初始化成该默认值。
对一个已经被实例化过的队列，设置或获取队列的该值。
上述情况通常需要提供 Set()和 Get()函数，以及某些类型的全局默认值。

在 ns-3 的属性系统中，这些值定义和存取器函数被移入类 TypeId，例如：

```
TypeId DropTailQueue::GetTypeId (void)
{
static TypeId tid = TypeId (“ns3::DropTailQueue”)
.SetParent<Queue> ()
.AddConstructor<DropTailQueue> ()
.AddAttribute (“MaxPackets”,
“The maximum number of packets accepted by this DropTailQueue.”,
UintegerValue (100),
MakeUintegerAccessor (&DropTailQueue::m_maxPackets),
MakeUintegerChecker<uint32_t> ())
;

return tid;
}
```

方法 AddAttribute()对该值进行一系列处理：

将变量 m_maxPackets 绑定到一个字符串”MaxPackets”。
提供默认值（100 packets）
提供为该值下定义的帮助信息。
提供”checker”（本例中未使用），可以用来设置所允许的上下界。
关键的一点是，现在该变量的值以及它的默认值在属性名字空间中是可访问的，基于字符串”MaxPackets”和 TypeId 字符串。在下一节，我们将提供一个例子来说明用户如何来操纵这些值。
————————————————————————————

3.2.2 基本用法
我们研究一下用户脚本如何访问这些值，基于文件
samples/main-attribute-value.cc，略去了一些细节。

```
//
// This is a basic example of how to use the attribute system to
```

```
// set and get a value in the underlying system; namely, an unsigned
// integer of the maximum number of packets in a queue
//

int
main (int argc, char *argv[])
{

// By default, the MaxPackets attribute has a value of 100 packets
// (this default can be observed in the function DropTailQueue::GetTypeId)
//
// Here, we set it to 80 packets.  We could use one of two value types:
// a string-based value or a Uinteger value
Config::SetDefault ("ns3::DropTailQueue::MaxPackets", StringValue ("80"));
// The below function call is redundant
Config::SetDefault ("ns3::DropTailQueue::MaxPackets", UintegerValue (80));

// Allow the user to override any of the defaults and the above
// SetDefaults() at run-time, via command-line arguments
CommandLine cmd;
cmd.Parse (argc, argv);
```

需要注意的是对 Config::SetDefault 的两次调用。这表明了我们如何设置随后被实例化的 DropTailQueues 的默认值。我们举例说明了两种类型的 Value 类：类 StringValue 和类 UintegerValue，这两个类可以被用来将值赋给叫做“ns3::DropTailQueue::MaxPackets”的属性。

现在我们使用底层 API 来创建一些对象。由于上述对默认值的操作，新建的队列的 m_maxPackets 值将被初始化为 80 而不是 100。

```
Ptr<Node> n0 = CreateObject<Node> ();

Ptr<PointToPointNetDevice> net0 = CreateObject<PointToPointNetDevice>
();
n0->AddDevice (net0);

Ptr<Queue> q = CreateObject<DropTailQueue> ();
net0->AddQueue(q);
```

这里我们创建了一个单独的节点（节点 0）和一个单独的

PointToPointNetDevice（NetDevice 0），并将一个 DropTailQueue 加入到了该 PointToPointNetDevice 上。

现在，我们可以操纵已经实例化过的 DropTailQueue 的 MaxPackets 值了，有多种不同的方法来达到这个目的。

————————————————————————————

3.2.2.1 基于指针的存取 Pointer-based access

假定存在一个指向相关网络设备的智能指针（Ptr），例如这里 net0 的指针。

改变该值的一种方法是 通过存取指向底层的队列的指针，并修改该队列的属性。

首先，我们能够通过 PointToPointNetDevice 的属性获得指向队列（基类）的指针，该属性叫做 TxQueue。

```
PointerValue tmp;
net0->GetAttribute (“TxQueue”, tmp);
Ptr<Object> txQueue = tmp.GetObject ();
```

使用函数 GetObject ，我们能够执行到 DropTailQueue 的安全的 downcast，MaxPackets 是 DropTailQueue 的成员。

```
Ptr<DropTailQueue> dtq = txQueue->GetObject <DropTailQueue> ();
NS_ASSERT (dtq != 0);
```

现在我们能够获取该队列上属性的值。由于属性系统存储的是值，而不是类型，所以与 Java 类似地，我们为底层数据类型引入了封装的 “Value” 类。这里，属性值被赋值给了一个 UintegerValue ，对这个值应用 Get()方法会得到 (unwrapped)uint32_t。

```
UintegerValue limit;
dtq->GetAttribute (“MaxPackets”, limit);
NS_LOG_INFO (“1.  dtq limit: ” << limit.Get () << ” packets”);
```

注意上述的 downcast 不是必须的。尽管该属性是该子类的成员，我们依然能够使用 Ptr<Queue> 来做同样的事情。

```
txQueue->GetAttribute (“MaxPackets”, limit);
NS_LOG_INFO (“2.  txQueue limit: ” << limit.Get () << ” packets”);
```

现在，让我们将他设置为另一个值（60）。

```
txQueue->SetAttribute("MaxPackets", UintegerValue (60));
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("3.  txQueue limit changed: " << limit.Get () << " packets");
```

————————————————————————————

3.2.2.2 基于名字空间的存取 Namespace-based access

另一种获取属性的方法是使用 configuration namespace。属性位于这个名字空间的已经路径上。如果用户无法访问底层指针但又想要使用一条语句来配置某个特定的属性时，这个方法很有用。

```
Config::Set ("/NodeList/0/DeviceList/0/TxQueue/MaxPackets",
UintegerValue (25));
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("4.  txQueue limit changed through namespace: " <<
limit.Get () << " packets");
```

我们还可以使用通配符来设置所有节点和所有网络设备的该值（例子如下）。

```
Config::Set ("/NodeList/*/DeviceList/*/TxQueue/MaxPackets",
UintegerValue (15));
txQueue->GetAttribute ("MaxPackets", limit);
NS_LOG_INFO ("5.  txQueue limit changed through wildcarded namespace: "
<<
limit.Get () << " packets");
```

————————————————————————————

3.2.3 通过构造函数和 helper classes 来设置 Setting through constructors helper classes

任意的属性组合都可以由 helper 和底层 APIs 来设置和获得。通过构造函数本身：

```
Ptr<Object> p = CreateObject<MyNewObject> ("n1", v1, "n2", v2, ...);
```

通过高层 helper APIs，比如：

```
mobility.SetPositionAllocator ("GridPositionAllocator",
"MinX", DoubleValue (-100.0),
"MinY", DoubleValue (-100.0),
"DeltaX", DoubleValue (5.0),
```

“DeltaY”, DoubleValue (20.0),
“GridWidth”, UintegerValue (20),
“LayoutType”, StringValue (“RowFirst”));

————————————————————————

3.2.4 值类 Value classes
读者将注意到新的某 Value 类是 AttributeValue 基类的子类。这些类可以被看做中间类，这些中间类可以被用来将 raw types 转换为可以被属性系统使用的值。属性系统的数据库用一种一般类型来存储许多类型的对象，到该一般类型的转换可以使用中间类（IntegerValue, DoubleValue for “floating point”）来完成，也可以通过字符串来完成。从类型到值的直接隐式转换不是很可行，所以用户可以选择使用字符串还是值：

p->Set (“cwnd”, StringValue (“100″)); // string-based setter
p->Set (“cwnd”, IntegerValue (100)); // integer-based setter
对于用户想引入属性系统的新的类型，系统提供一些宏来帮助用户为新的类型声明和定义新的 AttributeValue 子类。

ATTRIBUTE_HELPER_HEADER
ATTRIBUTE_HELPER_CPP

————————————————————————

3.3 对属性进行扩展 Extending attributes
ns-3 系统在属性系统下边放置了许多内部值，但毫无疑问，用户将对系统不完善的地方进行扩展以及加入用户自己的类。
————————————————————————

3.3.1 将现存的内部变量加入元数据系统 Adding an existing internal variable to the metadata system
考虑类 TcpSocket 中的这个变量：

uint32_t m_cWnd; // Congestion window
假设 使用 Tcp 的某个人想要使用元数据系统获得或设置该变量的值。如果 ns-3 还没有提供这个，用户可以再元数据系统中添加如下声明（在 TcpSocket 的 TypeId 声明中）：

```
.AddParameter (“Congestion window”,
“Tcp congestion window (bytes)”,
Uinteger (1),
MakeUintegerAccessor (&TcpSocket::m_cWnd),
MakeUintegerChecker<uint16_t> ());
```

现在，用户可以使用指向该 TcpSocket 的指针来执行设置和获取操作，而不用显式添加这些函数。此外，访问控制可以被应用，比如使得该参数只读不可写和对参数进行上下界检查。

————————————————————————————

3.3.2 添加新的 TypeId Adding a new TypeId

现在我们讨论用户如何往 ns-3 系统中添加新的类。

我们已经介绍过类似如下的 TypeId 定义：

```
TypeId
RandomWalk2dMobilityModel::GetTypeId (void)
{
static TypeId tid = TypeId (“ns3::RandomWalk2dMobilityModel”)
.SetParent<MobilityModel> ()
.SetGroupName (“Mobility”)
.AddConstructor<RandomWalk2dMobilityModel> ()
.AddAttribute (“Bounds”,
“Bounds of the area to cruise.”,
RectangleValue (Rectangle (0.0, 0.0, 100.0, 100.0)),
MakeRectangleAccessor (&RandomWalk2dMobilityModel::m_bounds),
MakeRectangleChecker ())
.AddAttribute (“Time”,
“Change current direction and speed after moving for this delay.”,
TimeValue (Seconds (1.0)),
MakeTimeAccessor (&RandomWalk2dMobilityModel::m_modeTime),
MakeTimeChecker ())
// etc (more parameters).
;
return tid;
}
```

类声明中与此相关的声明是一行公共成员方法：

public:
static TypeId GetTypeId (void);
典型的错误包括：

没有调用 SetParent 方法，或者使用了错误的类型来调用他。
没有调用 AddConstructor 方法，或者使用了错误的类型来调用他。
在 TypeId 的构造函数中对于 TypeId 的名字引入了印刷错误。
没有使用封装类的全限定 C++类型名作为 TypeId 的名字。
以上错误都无法被 ns-3 探测到，所以用户应当多次检查以确保正确性。
————————————————————————————

3.4 给属性系统中添加新的类
从用户的角度来看，编写新的类并将其加入属性系统主要是编写字符串与属性值之间的转换。多数可以通过“宏化的“（macro-ized）代码来复制/粘贴。例如目录 src/mobility/ 下的类 Rectangle：

类声明中加入一行：

```
/**
* brief a 2d rectangle
*/
class Rectangle
{
...

};
```

在类声明的下边加入一个宏调用和两个操作符：

```
std::ostream &operator << (std::ostream &os, const Rectangle &rectangle);
std::istream &operator >> (std::istream &is, Rectangle &rectangle);

ATTRIBUTE_HELPER_HEADER (Rectangle);
```

类定义的代码类似于：

```
ATTRIBUTE_HELPER_CPP (Rectangle);

std::ostream &
operator << (std::ostream &os, const Rectangle &rectangle)
```

```
{
os << rectangle.xMin << "|" << rectangle.xMax << "|" << rectangle.yMin <<
"|" << rectangle.yMax;
return os;
}
std::istream &
operator >> (std::istream &is, Rectangle &rectangle)
{
char c1, c2, c3;
is >> rectangle.xMin >> c1 >> rectangle.xMax >> c2 >> rectangle.yMin >>
c3 >> rectangle.yMax;
if (c1 != '|' ||
c2 != '|' ||
c3 != '|')
{
is.setstate (std::ios_base::failbit);
}
return is;
}
```

这些流操作符 将 字符串表示形式的 Rectangle ("xMin|xMax|yMin|yMax")转化为底层的 Rectangle，模块的编写者必须指定新类的这些操作符以及该类的实例的字符串句法表示形式。

————————————————————————————

3.5 ConfigStore

请求反馈： 这是 ns-3 的一个试验性的特色。他不在主要的代码树中。如果您喜欢该特色并愿意提供关于他的反馈，请给我们写电子邮件。

ns-3 的属性的值可以被存储在 ascii 文本文件中，并在将来的模拟中加载。这个特色被认为是 ns-3 的 ConfigStore。ConfigStore 的代码在 src/contrib/ 下。因为我们还在寻求用户的反馈，所以目前还不在主要的代码树中。

我们用一个例子来探索这个系统。将文件 csma-bridge.cc 复制到 scratch 目录：

```
cp examples/csma-bridge.cc scratch/
./waf
```

我们编辑该文件以加入 ConfigStore 特色。首先，添加一个 include 语句，然后加入以下行：

```
#include “contrib-module.h”
…
int main (…)
{
// setup topology

// Invoke just before entering Simulator::Run ()
ConfigStore config;
config.Configure ();

Simulator::Run ();
}
```

存在一个控制 Configure()的属性，他决定 Configure()是将模拟的配置存储在文件中并退出，还是加载模拟的配置文件并继续执行。首先，属性 LoadFilename 被检查，如果不为空，则程序从所提供的文件名来加载配置；如果为空，且属性 StoreFilename 被提供，则配置将被写入指定的输出文件。

虽然生成一个配置文件的样本并修改一些值是可能的，但有些情况这种方法是行不通的，因为对于同一个自动生成的配置文件，同一个对象上的同一个值可能在不同的配置路径上出现多次。

同样地，使用这个类的最好方法是用他生成一个初始的配置文件，仅从该文件中提取严格必须得元素，并将这些元素移动一个新的配置文件。这个新的配置文件在随后的模拟中可以被安全地编辑和加载。

以此为例运行一次程序来创建一个配置文件。如果你使用的是 bash shell，那么下边的命令应该能够工作（阐明了如何从命令行设置属性）：

```
./build/debug/scratch/csma-bridge –
ns3::ConfigStore::StoreFilename=test.config
```

如果上述命令不起作用（上述命令需要 rpath 的支持），试试如下：

```
./waf –command-template=”%s –
ns3::ConfigStore::StoreFilename=test.config” –run scratch/csma-bridge
```

运行该程序将产生一个叫做”test.config”的输出配置文件，类似于如下：

/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNet
Device/Addre
ss 00:00:00:00:00:01
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNet
Device/Frame
Size 1518
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNet
Device/SendE
nable true
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNet
Device/Recei
veEnable true
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNet
Device/TxQue
ue/$ns3::DropTailQueue/MaxPackets 100
/$ns3::NodeListPriv/NodeList/0/$ns3::Node/DeviceList/0/$ns3::CsmaNet
Device/Mtu 1
500
...
上边列出了拓扑脚本中的每一个对象以及每一个注册过的属性的值。此文件的语法是每行都标明了属性独一无二的名字，名字后边是值。

该文件是某个给定模拟中的参数的一个方便的记录，可以使用模拟输出文件来存储。此外，该文件还可以被用来将模拟参数化，而不是编辑脚本或传递命令行参数。比如：检查并调整一个已经存在的配置文件中的值，然后将该文件传递给模拟程序。相关的命令：

./build/debug/scratch/csma-bridge –
ns3::ConfigStore::LoadFilename=test.config
如果上述命令不起作用（上述命令需要 rpath 的支持），试试如下：

./waf –command-template=”%s –
ns3::ConfigStore::LoadFilename=test.config” –run scratch/csma-bridge

————————————————————————

3.5.1 基于 GTK 的 ConfigStore GTK-based ConfigStore

对于 ConfigStore，存在一个基于 GTK 的前端。这使得用户可以使用 GUI 来存取和修改变量。该特色的屏幕截图可以在 ns-3 Overview 找到。

要使用这个特色，必须安装 libgtk 和 libgtk-dev。Ubuntu 下安装命令的示例：

```
sudo apt-get install libgtk2.0-0 libgtk2.0-dev
```

通过 ./waf configure 阶段的输出来检验是否已经配置好：

```
—- Summary of optional NS-3 features:
Threading Primitives          : enabled
Real Time Simulator           : enabled
GtkConfigStore                : not enabled (library 'gtk+-2.0 >= 2.12' not found)
```

在上述例子中，GtkConfigStore 没有开启，要想使用他，必须安装合适的版本，并且再次执行 ./waf configure; ./waf。

用法与 non-GTK-based 版本几乎一样：

```
// Invoke just before entering Simulator::Run ()
GtkConfigStore config;
config.Configure ();
```

现在，当你运行脚本时将弹出一个 GUI，使得你可以打开不同节点/对象上属性的菜单，配置好之后启动模拟。

————————————————————————————

3.5.2 将来的工作 Future work

可能存在的改进：

在文件起始处保存一个包含日期与时间的独一无二的版本号。
在某处保存 rng 的初始种子。
使每个 RandomVariable 都连续化（serialize）自己的初始种子并在后期重新读取。
加入默认值。

NS-3 manual note – 04 Object model

Posted on 07/07/2009 by xa_ceshi

4. 对象模型 Object model

ns-3 本质上是一个基于 C++对象的系统。对象可以像往常一样依照 C++的规则被声明以及实例化。ns-3 还给传统的 C++对象添加了一些新的特色来提供更强的功能。本章为读者介绍 ns-3 的对象模型。

本节描述针对 ns-3 的对象的 C++类设计。总的来说，已经使用的设计模式包括经典的 object-oriented 设计（多态的接口和实现）、接口与实现相分离、非虚拟的公共接口设计模式、对象聚合设施以及 reference counting for memory management。尽管 ns-3 的设计与上述的任意一个都不严格一致，但熟悉类似 COM 或 Bonobo 等构件模型的人将能够识别 ns-3 对象聚合模型中的设计元素。

————————————————————————————

4.1 面向对象行为 Object-oriented behavior

一般而言，C++对象提供常见的面向对象功能（抽象、封装、继承以及多态），这些功能是经典的面相对象设计的一部分。ns-3 对象使用了这些特性。例如：

```
class Address
{
public:
Address ();
Address (uint8_t type, const uint8_t *buffer, uint8_t len);
Address (const Address & address);
Address &operator = (const Address &address);
...
private:
uint8_t m_type;
uint8_t m_len;
...
};
```

————————————————————————————

4.2 对象的基类 Object base classes

ns-3 中有 3 个特殊的基类。由 3 个基类继承的类能够用特殊的特性来实例化对象。这 3 个基类是：

```
class Object
class ObjectBase
class RefCountBase
```

没有要求 ns-3 的对象都继承自这 3 个类，但对于具有特殊特性的类，是这样的。由 class Object 派生的类具有如下特性。

ns-3 的类型和属性系统（参看 Attributes）。

对象聚合系统
a smart-pointer reference counting system (class Ptr)
由 class ObjectBase 派生的类具有上述前两个特性，不具有 smart pointers。由 class RefCountBase 派生的类只具有 the smart-pointer reference counting system.
在实际中，ns-3 的开发者碰到最多的是上述 3 个中的 class Object。
————————————————————————————

4.3 内存管理和类 Ptr Memory management and class Ptr
C++程序中的内存管理是一个复杂的过程，并且经常被不正确地处理或者被不一致地处理。以下介绍我们决定使用的一个引用计数设计。
所有使用引用计数的对象都维护一个内部引用值，根据该值来决定对象什么时候可以安全地删除他自身。每当有接口获得对象的指针时，该对象的引用计数值就增加 1，这个增加是通过调用 calling Ref() 完成的。当用户不再使用该指针时，该指针的用户负责显式调用 Unref() 来解除对该指针的引用。当引用计数减到 0 时，对象被删除。
当用户代码通过创建对象从该对象获得指针或者通过 QueryInterface 获得指针时，没有必要增加引用计数值。
当用户代码从其他的源（比如，对指针进行复制）获得指针时，用户代码必须调用 Ref() 来增加引用计数值。
该对象的指针的所有用户都必须调用 Unref() 来释放引用。
通过使用下边描述的 reference counting smart pointer class，调用 Unref() 的负担有了一些缓解。
通过底层 API，并想在堆上显式分配 non-reference-counted 对象的用户使用 new 操作符，用户负责删除这类对象。
————————————————————————————

4.3.1 引用计数智能指针（Ptr）Reference counting smart pointer (Ptr)
因为始终调用 Ref() 和 Unref() 很麻烦，所以 ns-3 提供类似于 Boost::intrusive_ptr 的智能指针 class Ptr。该智能指针类假定底层类型提供一对 Ref 和 Unref 方法，且该对方法增加和减少对象的内部引用计数值。
这种实现使得你能够像操纵普通指针一样操纵智能指针：可以将他和 0 比较、将他和其他指针比较以及给他赋 0 值，等等。
通过 GetPointer 和 PeekPointer 方法有可能从智能指针中提取出裸指针。
如果你想把用 new 产生的对象存储到一个智能指针。为了避免内存泄漏，我们建议你使用 CreateObject 模板函数来创建对象并将他存储到智能指针。这些函数是很小的便利函数，他们的目标仅仅是使你少敲些键盘。
————————————————————————————

4.3.2 CreateObject 和 Create CreateObject and Create

C++的对象可以被静态地创建、动态地创建以及自动地创建。这在 ns-3 中同样适用，但系统中的一些对象有一些附加的框架。特别地，引用计数的对象通常使用模板化的 Create 或 CreateObject 方法被分配。
对于由 class Object 派生的对象：
Ptr<WifiNetDevice> device = CreateObject<WifiNetDevice> ();
不要使用 operator new 来创建这类对象。应该使用 CreateObject() 来创建。
对于由 class RefCountBase 派生的对象，或其他支持智能指针类用法的对象（特别地，比如 ns-3 Packet class），建议使用模板化的 helper function：
Ptr<B> b = Create<B> ();
这是一个对 new 操作符的封装，他正确地处理了引用计数系统。
————————————————————————————

4.3.3 聚合 Aggregation
ns-3 的对象聚合系统很大程度上是由一个认识促成的，即 ns-2 中一个很普遍的用法是通过继承和多态来扩展协议模型。例如，TCP 的特殊版本 RenoTcpAgent 是由类 TcpAgent 派生的，并对基类的函数进行覆盖（override）。尽管如此，ns-2 模型中出现的两个问题是 downcasts 和“weak base class” 。Downcasting 是指一个过程，即使用指向某个基类对象的指针并在程序运行时查询该指针来获得类型信息，然后将该指针显式转换为子类的指针，以便子类的 API 能够使用。Weak base class 是指当某个类无法被有效地重用（由他进行派生）出现的问题，因为他缺少必要的功能，导致开发者不得不修改基类，这将导致基类 API 的增生，某些 API 可能并不是对所有子类都在语义上正确。
ns-3 使用查询接口设计模式来避免这些问题。该设计基于 Component Object Model 和 GNOME Bonobo 的基础。尽管现在替代构件的完全的二进制兼容性还不被支持，但我们努力简化语法和对模型编写者的影响。
————————————————————————————

4.3.3.1 聚合的例子 Aggregation example
ns-3 中，class Node 是使用聚合的一个很好的例子。注意 ns-3 中没有类 Node 的派生类（比如类 InternetNode 等），而是将构件（各种协议）聚合到节点中。我们来研究一些 Ipv4 协议是如何被加入节点的。

```
static void
AddIpv4Stack(Ptr<Node> node)
{
Ptr<Ipv4L3Protocol> ipv4 = CreateObject<Ipv4L3Protocol> ();
ipv4->SetNode (node);
node->AggregateObject (ipv4);
Ptr<Ipv4Impl> ipv4Impl = CreateObject<Ipv4Impl> ();
ipv4Impl->SetIpv4 (ipv4);
node->AggregateObject (ipv4Impl);
```

}
注意 Ipv4 协议是用 CreateObject() 创建的。接着 Ipv4 协议被聚合到节点中。这样，基类 Node 就不需要被编辑来使得用户使用指向基类 Node 的指针来访问 Ipv4 接口；用户可以在程序运行时来向节点请求指向该节点的 Ipv4 接口的指针。用户如何向节点提出请求在下一小节描述。
注意：将多于一个的同一类型的对象聚合到某个 ns3::object 是编程错误。所以，如果想要存储一个节点的所有活动的 sockets，聚合是不可选的。
————————————————————————————

4.3.3.2 GetObject 的例子 GetObject example
GetObject 是一个获得安全 downcasting 的类型安全的方法，并且使得接口能够在对象上被找到。
考虑一个节点的指针 m_node，该指针指向一个节点对象，且先前已经将 Ipv4 的实现聚合到了该节点。客户代码想要配置一个默认的路由。为了实现这点，必须访问该节点内的一个对象，且该对象具有 IP 转发配置的接口。如下：

```
Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4> ();
```

如果实际上没有 Ipv4 的对象被聚合到该节点，那么该方法将返回 null。因此，检查该函数调用的返回值是一个好习惯。如果成功，则用户可以使用 Ptr，该指针指向先前被聚合到该节点的 Ipv4 对象。
另一个如何使用聚合的例子是给对象添加可选的模型。例如，一个现存的 Node 对象可以具有一个在运行时被聚合到该节点对象的“Energy Model”对象（不需要对节点类进行修改和重新编译）。一个现存的模型（比如一个无线网络设备）可以通过”GetObject”来获得该能量模型并表现地就像该接口是内建在 Node 对象的底层或者该接口是在运行时被聚合到该节点的。而其他节点却不需要知道能量模型的任何事情。
我们希望这样的编程模式可以大量减小开发者修改各种基类的必要。
————————————————————————————

4.4 Downcasting
有一个经常出现的问题：”假设我有一个基类的指针（Ptr），该指针指向一个对象，如果我想要派生类的指针，那么我应该进行 downcast（通过 C++的动态类型转换）来获得派生的指针还是应该使用对象聚合系统进行 GetObject<> () 来找到一个 Ptr，该指针指向子类 API 的接口”？
这个问题的答案是：在多数情况下，两种技术都行的通。ns-3 提供一个模板化的函数，该函数使得对象动态类型转换的语法更加友好：

```
template <typename T1, typename T2>
Ptr<T1>
DynamicCast (Ptr<T2> const&p)
{
return Ptr<T1> (dynamic_cast<T1 *> (PeekPointer (p)));
```

}
当程序员有一个基类的指针，想和一个子类的指针进行测试时，DynamicCast 行的通。当寻找被聚合的不同对象时，GetObject 行的通。但对于子类，GetObject 也行的通，和 DynamicCast 一样。如果不确定，那么程序员应该使用 GetObject，因为他在所有情况下都适用。如果程序员知道所考虑的对象的类层次结构，使用 DynamicCast 更加直接。
————————————————————————————

NS-3 manual note – 04 Object model

by gaocong

4. 对象模型 Object model

ns-3 本质上是一个基于 C++对象的系统。对象可以像往常一样依照 C++的规则被声明以及实例化。ns-3 还给传统的 C++对象添加了一些新的特色来提供更强的功能。本章为读者介绍 ns-3 的对象模型。

本节描述针对 ns-3 的对象的 C++类设计。总的来说，已经使用的设计模式包括经典的 object-oriented 设计（多态的接口和实现）、接口与实现相分离、非虚拟的公共接口设计模式、对象聚合设施以及内存管理中的引用计数(reference counting)。尽管 ns-3 的设计与上述的任意一个都不严格一致，但熟悉类似 COM 或 Bonobo 等构件模型的人将能够识别 ns-3 对象聚合模型中的设计元素。

————————————————————————————

4.1 面向对象行为 Object-oriented behavior

一般而言，C++对象提供常见的面向对象功能（抽象、封装、继承以及多态），这些功能是经典的面相对象设计的一部分。ns-3 对象使用了这些特性。例如：

class Address

{

public:

Address ();

Address (uint8_t type, const uint8_t *buffer, uint8_t len);

Address (const Address & address);

Address &operator = (const Address &address);

...

private:

uint8_t m_type;

uint8_t m_len;

...

};

————————————————————————————

4.2 对象的基类 Object base classes

ns-3 中有 3 个特殊的基类。由 3 个基类继承的类能够用特殊的特性来实例化对象。这 3 个基类是：

class Object

class ObjectBase

class RefCountBase

没有要求 ns-3 的对象都继承自这 3 个类，但对于具有特殊特性的类，是这样的。由 class Object 派生的类具有如下特性。

ns-3 的类型和属性系统（参看 Attributes）。

对象聚合系统

a smart-pointer reference counting system (class Ptr)

由 class ObjectBase 派生的类具有上述前两个特性，不具有 smart pointers。由 class RefCountBase 派生的类只具有 the smart-pointer reference counting system.

在实际中，ns-3 的开发者碰到最多的是上述 3 个中的 class Object。

————————————————————————————

4.3 内存管理和类 Ptr Memory management and class Ptr

C++程序中的内存管理是一个复杂的过程，并且经常被不正确地处理或者被不一致地处理。以下介绍我们决定使用的一个引用计数设计。

所有使用引用计数的对象都维护一个内部引用值，根据该值来决定对象什么时候可以安全地删除他自身。每当有接口获得对象的指针时，该对象的引用计数值就增加 1，这个增加是通过调用 calling Ref() 完成的。当用户不再使用该指针时，该指针的用户负责显式调用 Unref() 来解除对该指针的引用。当引用计数减到 0 时，对象被删除。

当用户代码通过创建对象从该对象获得指针或者通过 QueryInterface 获得指针时，没有必要增加引用计数值。

当用户代码从其他的源（比如，对指针进行复制）获得指针时，用户代码必须调用 Ref() 来增加引用计数值。

该对象的指针的所有用户都必须调用 Unref() 来释放引用。

通过使用下边描述的 reference counting smart pointer class，调用 Unref() 的负担有了一些缓解。

通过底层 API，并想在堆上显式分配 non-reference-counted 对象的用户使用 new 操作符，用户负责删除这类对象。

————————————————————————————

4.3.1 引用计数智能指针（Ptr）Reference counting smart pointer (Ptr)

因为始终调用 Ref() 和 Unref() 很麻烦，所以 ns-3 提供类似于 Boost::intrusive_ptr 的智能指针 class Ptr。该智能指针类假定底层类型提供一对 Ref 和 Unref 方法，且该对方法增加和减少对象的内部引用计数值。

这种实现使得你能够像操纵普通指针一样操纵智能指针：可以将他和零比较、将他和其他指针比较以及给他赋零值，等等。

通过 GetPointer 和 PeekPointer 方法有可能从智能指针中提取出裸指针。

如果你想把用 new 产生的对象存储到一个智能指针。为了避免内存泄漏，我们建议你使用 CreateObject 模板函数来创建对象并将他存储到智能指针。这些函数是很小的便利函数，他们的目标仅仅是使你少敲些键盘。

————————————————————————————

4.3.2 CreateObject 和 Create CreateObject and Create

C++的对象可以被静态地创建、动态地创建以及自动地创建。这在 ns-3 中同样适用，但系统中的一些对象有一些附加的框架。特别地，引用计数的对象通常使用模板化的 Create 或 CreateObject 方法被分配。

对于由 class Object 派生的对象：

Ptr<WifiNetDevice> device = CreateObject<WifiNetDevice> ();

不要使用 operator new 来创建这类对象。应该使用 CreateObject() 来创建。

对于由 class RefCountBase 派生的对象，或其他支持智能指针类用法的对象（特别地，比如 ns-3 Packet class），建议使用模板化的 helper function：

Ptr<B> b = Create<B> ();

这是一个对 new 操作符的封装，他正确地处理了引用计数系统。

————————————————————————————

4.3.3 聚合 Aggregation

ns-3 的对象聚合系统很大程度上是由一个认识促成的，即 ns-2 中一个很普遍的用法是通过继承和多态来扩展协议模型。例如，TCP 的特殊版本 RenoTcpAgent 是由类 TcpAgent 派生的，并对基类的函数进行覆盖（override）。

尽管如此，ns-2 模型中出现的两个问题是 downcasts 和“weak base class” 。Downcasting 是指一个过程，即使用指向某个基类对象的指针并在程序运行时查询该指针来获得类型信息，然后将该指针显式转换为子类的指针，以便子类的 API 能够使用。Weak base class 是指当某个类无法被有效地重用（由他进行派生）出现的问题，因为他缺少必要的功能，导致开发者不得不修改基类，这将导致基类 API 的增生，某些 API 可能并不是对所有子类都在语义上正确。

ns-3 使用查询接口设计模式来避免这些问题。该设计基于 Component Object Model 和 GNOME Bonobo 的基础。尽管现在替代构件的完全的二进制兼容性还不被支持，但我们努力简化语法和对模型编写者的影响。

————————————————————————————

4.3.3.1 聚合的例子 Aggregation example

ns-3 中，class Node 是使用聚合的一个很好的例子。注意 ns-3 中没有类 Node 的派生类（比如类 InternetNode 等），而是将构件（各种协议）聚合到节点中。我们来研究一些 Ipv4 协议是如何被加入节点的。

```
static void

AddIpv4Stack(Ptr<Node> node)

{

Ptr<Ipv4L3Protocol> ipv4 = CreateObject<Ipv4L3Protocol> ();

ipv4->SetNode (node);

node->AggregateObject (ipv4);

Ptr<Ipv4Impl> ipv4Impl = CreateObject<Ipv4Impl> ();

ipv4Impl->SetIpv4 (ipv4);
```

node->AggregateObject (ipv4Impl);

}

注意 Ipv4 协议是用 CreateObject() 创建的。接着 Ipv4 协议被聚合到节点中。这样，基类 Node 就不需要被编辑来使得用户使用指向基类 Node 的指针来访问 Ipv4 接口；用户可以在程序运行时来向节点请求指向该节点的 Ipv4 接口的指针。用户如何向节点提出请求在下一小节描述。

注意：将多于一个的同一类型的对象聚合到某个 ns3::object 是编程错误。所以，如果想要存储一个节点的所有活动的 sockets，聚合是不可选的。

————————————————————————————

4.3.3.2 GetObject 的例子 GetObject example

GetObject 是一个获得安全 downcasting 的类型安全的方法，并且使得接口能够在对象上被找到。

考虑一个节点的指针 m_node，该指针指向一个节点对象，且先前已经将 Ipv4 的实现聚合到了该节点。客户代码想要配置一个默认的路由。为了实现这点，必须访问该节点内的一个对象，且该对象具有 IP 转发配置的接口。如下：

Ptr<Ipv4> ipv4 = m_node->GetObject<Ipv4> ();

如果实际上没有 Ipv4 的对象被聚合到该节点，那么该方法将返回 null。因此，检查该函数调用的返回值是一个好习惯。如果成功，则用户可以使用 Ptr，该指针指向先前被聚合到该节点的 Ipv4 对象。

另一个如何使用聚合的例子是给对象添加可选的模型。例如，一个现存的 Node 对象可以具有一个在运行时被聚合到该节点对象的“Energy Model”对象（不需要对节点类进行修改和重新编译）。一个现存的模型（比如一个无线网络设备）可以通过”GetObject”来获得该能量模型并表现地就像该接口是内建在 Node 对象的底层或者该接口是在运行时被聚合到该节点的。而其他节点却不需要知道能量模型的任何事情。

我们希望这样的编程模式可以大量减小开发者修改各种基类的必要。

————————————————————————————

4.4 Downcasting

有一个经常出现的问题：”假设我有一个基类的指针（Ptr），该指针指向一个对象，如果我想要派生类的指针，那么我应该进行 downcast（通过 C++的动态类型转换）来获得派生的指针还是应该使用对象聚合系统进行 GetObject<> () 来找到一个 Ptr，该指针指向子类 API 的接口”？

这个问题的答案是：在多数情况下，两种技术都行的通。ns-3 提供一个模板化的函数，该函数使得对象动态类型转换的语法更加友好：

```
template <typename T1, typename T2>

Ptr<T1>

DynamicCast (Ptr<T2> const&p)

{

return Ptr<T1> (dynamic_cast<T1 *> (PeekPointer (p)));

}
```

当程序员有一个基类的指针，想和一个子类的指针进行测试时，DynamicCast 行的通。当寻找被聚合的不同对象时，GetObject 行的通。但对于子类，GetObject 也行的通，和 DynamicCast 一样。如果不确定，那么程序员应该使用 GetObject，因为他在所有情况下都适用。如果程序员知道所考虑的对象的类层次结构，使用 DynamicCast 更加直接

NS-3 manual note – 05 Real-Time Scheduler

Posted on 07/07/2009 by xa_ceshi

5. 实时调度者 Real-Time Scheduler

ns-3 被设计为可以集成到测试床以及虚拟机环境。要与真实的网络栈集成并发出/消耗包，则需要一个实时调度者来尝试将模拟时钟与硬件时钟进行锁定。我们描述这样一个构件：实时调度者。

实时调度者的目标是使模拟时钟的前进与外部的时基同步。当没有外部时基（比如挂钟）时，模拟时间从某个模拟时刻瞬间跳转到下一个模拟时刻。

————————————————————————————

5.1 行为 Behavior

当使用非实时调度者时（ns-3 中的默认情况），模拟器将模拟时间推进到下一个排定的事件。在事件执行期间，模拟时间是停滞不动的。当使用实时调度者时，从模拟模型的角度来看，模拟时间在事件执行期间是停滞不动的；但在事件与事件之间，模拟器将尝试把模拟时钟与机器时钟保持一致。

当一个事件执行完成后，调度程序转移到下一个事件，调度程序将下一个事件的执行时间与机器时钟进行比较。如果下一个事件被排定在某个将来的时刻，则模拟器进行睡眠直到该时刻，然后执行下一个事件。

由于模拟事件执行时固有的处理过程，有可能发生模拟器无法跟上实际时间。在这种情况下，由用户配置决定怎么做。ns-3 中有两个属性控制该行为。第一个是 ns3::RealTimeSimulatorImpl::SynchronizationMode。该属性的两个可能的条目是 BestEffort（默认的）和 HardLimit。在 quot;BestEffort”模式下，模拟器将努力赶上实际时间，具体的做法是模拟器执行事件直到下一个时间点来临（这个时间点是指下一个事件在将来的某个实际时间的来临），否则模拟结束。在 BestEffort 模式下，模拟有可能消耗比挂钟走过的时间更多的时间。

在”HardLimit”模式下，如果超过容忍阈值，则模拟在未完成前中止。这个属性是： ns3::RealTimeSimulatorImpl::HardLimit，默认值为 0.1 秒。

另一种操作模式是 在事件执行期间，模拟时间不停滞。这种模式的实时模拟被实现了，但是由于他是否有用的问题，他被从 ns-3 的代码结构中删除了。如果用户对模拟时间在事件执行期间不停滞的的实时模拟（比如，每次调用 Simulator::Now() 都返回当前的挂钟时刻，而不是事件开始执行的时刻）感兴趣，请联系 ns-developer 邮件列表。

————————————————————————————

5.2 用法 Usage

从脚本的角度来看，实时模拟器的用法直截了当。用户只需要给实时模拟器设置属性 SimulatorImplementationType ，具体如下：

GlobalValue::Bind (“SimulatorImplementationType”,
StringValue (“ns3::RealtimeSimulatorImpl”));

文件 examples/realtime-udp-echo.cc 中有一个如何配置实时行为的例子，尝试执行：
./waf –run realtime-udp-echo
模拟器使用 best effort 还是 hard limit 方式来工作取决于上一节中解释的属性。
————————————————————————————

5.3 实现 Implementation
具体实现包含在如下文件中：
src/simulator/realtime-simulator-impl.{cc,h
src/simulator/wall-clock-synchronizer.{cc,h
创建一个实时调度者，并希望模拟时间能够消耗实际时间。我们主张使用 sleep-wait 与 busy-wait 的组合来实现。Sleep-waits 导致调用进程（线程）让出处理器一段时间。即使这个指定量的时间可以通过纳秒来传递，他实际上被转化为操作系统指定的粒度。在 Linux 下，这个粒度叫做 Jiffy。通常该解析率不能满足我们的需要（与 10 毫秒相似），所以我们下舍入并休眠较少个数的 Jiffies。指定个数的 Jiffies 过去之后，线程被唤醒。此时，我们还有一些残留的时间需要等待。这个时间通常都小于休眠时间的最小值，所以我们对剩余的时间进行 busy-wait。这意味着线程保持在一个消耗性的 for 循环中，直到想要的时间来临。通过 sleep-wait 与 busy-wait 的组合，逝去的实际时间（挂钟时间）将与下一个事件的模拟时刻相吻合，则模拟继续进行。

NS-3 manual note – 05 Real-Time Scheduler

by gaocong

5. 实时调度者 Real-Time Scheduler

ns-3 被设计为可以集成到测试床以及虚拟机环境。要与真实的网络栈集成并发出/消耗包，则需要一个实时调度者来尝试将模拟时钟与硬件时钟进行锁定。我们描述这样一个构件：实时调度者。

实时调度者的目标是使模拟时钟的前进与外部的时基同步。当没有外部时基（比如挂钟）时，模拟时间从某个模拟时刻瞬间跳转到下一个模拟时刻。

————————————————————————————

5.1 行为 Behavior

当使用非实时调度者时（ns-3 中的默认情况），模拟器将模拟时间推进到下一个排定的事件。在事件执行期间，模拟时间是停滞不动的。当使用实时调度者时，从模拟模型的角度来看，模拟时间在事件执行期间是停滞不动的；但在事件与事件之间，模拟器将尝试把模拟时钟与机器时钟保持一致。

当一个事件执行完成后，调度程序转移到下一个事件，调度程序将下一个事件的执行时间与机器时钟进行比较。如果下一个事件被排定在某个将来的时刻，则模拟器进行睡眠直到该时刻，然后执行下一个事件。

由于模拟事件执行时固有的处理过程，有可能发生模拟器无法跟上实际时间。在这种情况下，由用户配置决定怎么做。ns-3 中有两个属性控制该行为。第一个是 ns3::RealTimeSimulatorImpl::SynchronizationMode。该属性的两个可能的条目是 BestEffort（默认的）和 HardLimit。在 quot;BestEffort”模式下，模拟器将努力赶上实际时间，具体的做法是模拟器执行事件直到下一个时间点来临（这个时间点是指下一个事件在将来的某个实际时间的来临），否则模拟结束。在 BestEffort 模式下，模拟有可能消耗比挂钟走过的时间更多的时间。
在”HardLimit”模式下，如果超过容忍阈值，则模拟在未完成前中止。这个属性是： ns3::RealTimeSimulatorImpl::HardLimit，默认值为 0.1 秒。

另一种操作模式是 在事件执行期间，模拟时间不停滞。这种模式的实时模拟被实现了，但是由于他是否有用的问题，他被从 ns-3 的代码结构中删除了。如果用户对模拟时间在事件执行期间不停滞的的实时模拟（比如，每次调用 Simulator::Now() 都返回当前的挂钟时刻，而不是事件开始执行的时刻）感兴趣，请联系 ns-developer 邮件列表。

————————————————————————————

5.2 用法 Usage

从脚本的角度来看，实时模拟器的用法直截了当。用户只需要给实时模拟器设置属性 SimulatorImplementationType ，具体如下：

GlobalValue::Bind (“SimulatorImplementationType”,

StringValue (“ns3::RealtimeSimulatorImpl”));

文件 examples/realtime-udp-echo.cc 中有一个如何配置实时行为的例子，尝试执行：

./waf –run realtime-udp-echo

模拟器使用 best effort 还是 hard limit 方式来工作取决于上一节中解释的属性。

————————————————————————

5.3 实现 Implementation

具体实现包含在如下文件中：

src/simulator/realtime-simulator-impl.{cc,h

src/simulator/wall-clock-synchronizer.{cc,h

创建一个实时调度者，并希望模拟时间能够消耗实际时间。我们主张使用 sleep-wait 与 busy-wait 的组合来实现。Sleep-waits 导致调用进程（线程）让出处理器一段时间。即使这个指定量的时间可以通过纳秒来传递，他实际上被转化为操作系统指定的粒度。在 Linux 下，这个粒度叫做 Jiffy。通常该解析率不能满足我们的需要（与 10 毫秒相似），所以我们下舍入并休眠较少个数的 Jiffies。指定个数的 Jiffies 过去之后，线程被唤醒。此时，我们还有一些残留的时间需要等待。这个时间通常都小于休眠时间的最小值，所以我们对剩余的时间进行 busy-wait。这意味着线程保持在一个消耗性的 for 循环中，直到想要的时间来临。通过 sleep-wait 与 busy-wait 的组合，逝去的实际时间（挂钟时间）将与下一个事件的模拟时刻相吻合，则模拟继续进行。

NS-3 manual note – 06 Emulation

Posted on 07/09/2009 by xa_ceshi

6. 仿真 Emulation

ns-3 被设计为可以集成到测试床以及虚拟机环境。我们通过提供两类网络设备来陈述这个需求。第一类，Emu @code {NetDevice 使得 ns-3 模拟过程能够 在“real”网络上发送数据。第二类，Tap NetDevice 使得“real”主机能够参与 ns-3 的模拟过程，就像其他模拟节点一样。ns-3 的模拟过程可以由任意组合的 Emu 以及 Tap 设备来构造。

我们想要支持的用例（use-case）之一是测试床。这类环境的一个具体例子是 ORBIT 测试床。ORBIT 是一个实验室性质的仿真试验网络，该网络由 400 个 802.11 无线节点以二维网格分布而成。我们使用 ORBIT 的“imaging”进程在 ORBIT 阵列上加载和运行 ns-3 的模拟过程，这便是与 ORBIT 的集成。我们使用 Emu NetDevices 来驱动测试床上的硬件，而且我们可以使用 ns-3 的跟踪函数和日志记录函数来积累结果，也可以使用 ORBIT 的数据收集技术。参看 http://www.orbit-lab.org/ 来获得 ORBIT 测试床的细节。

下图显示了一个这样的模拟过程：

图 6.1：Figure 6.1: 测试床仿真的实现举例 Example Implementation of Testbed Emulation.

可以看到有彼此分开的主机，每个主机都运行“global”模拟的一个子集。我们没有使用 ns-3 的 channel 来连接主机，而是使用了由测试床提供的真实的硬件。这使得 ns-3 中附着于模拟节点的 application 和协议栈能够通过真实的硬件来通信。

我们期望这个配置的主要用途是在现实世界的网络环境中生成反复的实验结果，该实验结果包括 ns-3 所有的跟踪、日志记录、可视化以及统计信息收集工具。在本质上相反的配置中，我们使运行应用程序和协议栈的“real”机器来与 ns-3 的模拟过程集成。这考虑到与真实机器相连接的大型网络模拟，并且开启了虚拟化。

下图显示了一个这样的模拟过程：

图 6.2：Figure 6.2: 仿真信道的实现概述 Implementation overview of emulated channel.

这里你将看到一个单独的主机，该主机上运行着一些虚拟机。图中部的一个虚拟机上运行着一个 ns-3 模拟过程。该模拟有一些节点，这些节点与一些 ns-3 的 applications 和协议栈相关联，这些 applications 和协议栈通过 ns-3 模拟的网络设备与一个 ns-3 的 channel 交流。

图的左边和右边还有两个虚拟机。这些虚拟机正在运行本地（Linux 的）应用程序和协议栈。虚拟机通过 Linux Tap 网络设备被连接到模拟过程中。Tap 设备的用户模式的处理程序在模拟过程中被实例化，并被附着到一个代理节点，该代理节点代表模拟过程中的虚拟机。这些处理程序使得本地虚拟机上的 Tap 设备像

模拟过程中的虚拟机上的 ns-3 网络设备一样工作。这点也使得本地虚拟机上的软件和协议套件确信他们是与 ns-3 的模拟信道连接着的。
我们期望这个环境的典型用例是 分析本地应用程序和协议套件在大规模的 ns-3 模拟网络面前的行为。

6.1 行为 Behavior

6.1.1 Emu 网络设备 Emu Net Device

Emu 网络设备使得模拟节点能够在真实的网络中发送和接收包。仿真的网络设备依赖一个特定的处在混杂（promiscuous）模式的接口。网络设备打开一个原始套接字（raw socket）并与该接口绑定。我们执行 MAC 欺骗来将模拟网络流量与可能流进/出该主机的其他网络流量分开。
通常，仿真的网络设备的用例在较小的模拟中，这些模拟通过特定的接口与外界连接。例如，你可以构造一些虚拟机并通过主机模式（host-only）的网络来连接他们。为了使用仿真的网络设备，你需要将所有 host-only 模式的接口都设置为混杂模式并且提供适当的设备名称，比如"eth1"。
你还可以在测试床环境下使用 Emu 网络设备，运行模拟的主机有着驱动该测试床硬件的特定接口。你还需要将这个特定接口设置为混杂模式并为 ns-3 的仿真网络设备提供一个适当的设备名称。这样环境的一个例子是上述的 ORBIT 测试床。
只有当底层接口启动并处在混杂模式时，Emu 网络设备才能工作。包将通过该设备被发送出去，但是我们使用 MAC 欺骗。默认情况下，MAC 地址使用 Organizationally Unique Identifier (OUI) 00:00:00 作为基准来产生（译者注：Organizationally Unique Identifier）。这个制造商代码没有分配给任何组织，所以应该不会与任何真实硬件产生冲突。
总是又用户来判断是否可以在网络中使用这些 MAC 地址，即不会与网络中的其他东西（包括其他使用 Emu 设备的模拟过程）产生冲突。如果你在彼此分开的模拟中使用仿真网络设备，那么你必须考虑全局的 MAC 地址分配问题，并确保所有模拟中的 MAC 地址都是独一无二的。仿真网络设备遵从方法 SetAddress 提供的 MAC 地址，所以你可以手工来完成。对于更大规模的模拟，你可能想在 MAC 地址分配函数中来设置 OUI。
与仿真网络设备对应的 IP 地址是在模拟过程中生成的地址，这些地址是通过 helper functions 用普通的方法生成的。由于我们使用了 MAC 欺骗，所以 ns-3 的网络栈不会和任意本地网络栈发生冲突。
与所有的 ns-3 设备一样，仿真网络设备也具有一个 helper function。一个独特的方面是没有与底层媒介相关联的 channel。我们不知道外部的媒介是什么，所以没有对他进行抽象建模。需要清楚的首要事情是这对于静态全局路由蕴含的影响。全局路由器模块尝试在 channels 上"行走"来寻找相邻的网络。由于没有 channel，所以全局路由器将无法做到这一点，则你必须使用动态路由协议来将路由包含到 Emu-based 网络中，比如 OLSR（译者注：OLSR）。

6.1.2 Tap 网络设备 Tap Net Device

The Tap Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the Tap device is merged.

6.2 用法 Usage

6.2.1 Emu Net Device

一旦模拟的网络配置完成，Emu 网络设备的用法是直截了当的。由于涉及到这个设备的多数工作都是在模拟开始之前的网络配置中，所以你可能需要回顾一些 ns-3 wiki 上的 HOWTO 页面，这些页面描述了 如何 使用 WMware 构建虚拟的测试网络以及如何运行使用 Emu 网络设备的模拟的例子（客户机 服务器）。

http://www.nsnam.org/wiki/index.php/HOWTO_use_VMware_to_set_up_virtual_networks_(Windows)

http://www.nsnam.org/wiki/index.php/HOWTO_use_ns-3_scripts_to_drive_real_hardware_(experimental)

一旦完成了配置，使用 Emu 设备需要脚本修改就很少了。主要的结构上的不同是你将需要为每个节点创建一个 ns-3 模拟脚本。在上述 HOWTOs 的情况中，有一个客户机脚本和一个服务器脚本。唯一的“challenge”是将地址设置正确。

与其他所有的 ns-3 设备一样，我们为 Emu 提供一个 helper class。下边的代码段举例说明了 如何声明一个 EmuHelper 并用他将属性“DeviceName”设置为“eth1”，以及如何把 Emu 设备安装到一组节点上。在上述 HOWTO 的情况中，客户端和服务器端都需要这样做。

```
EmuHelper emu;
emu.SetAttribute (“DeviceName”, StringValue (“eth1″));
NetDeviceContainer d = emu.Install (n);
```

另一个可能要求的仅有的变化是保证客户机和服务器的地址空间（MAC 和 IP）是互相兼容的。首先，MAC 地址被设置为一个独一无二的值，该值在两端都准所周知的（这里以一端为例）。

```
//
// We’ve got the devices in place.  Since we’re using MAC address
// spoofing under the sheets, we need to make sure that the MAC addresses
// we have assigned to our devices are unique.  Ns-3 will happily
// automatically assign the same MAC addresses to the devices in both halves
// of our two-script pair, so let’s go ahead and just manually change them
// to something we ensure is unique.
//
Ptr<NetDevice> nd = d.Get (0);
Ptr<EmuNetDevice> ed = nd->GetObject<EmuNetDevice> ();
ed->SetAddress (“00:00:00:00:00:02″);
```

接着客户机或服务器的 IP 将使用 helpers 用普通方法设置。

```
//
// We've got the "hardware" in place.  Now we need to add IP addresses.
// This is the server half of a two-script pair.  We need to make sure
// that the addressing in both of these applications is consistent, so
// we use provide an initial address in both cases.  Here, the client
// will reside on one machine running ns-3 with one node having ns-3
// with IP address "10.1.1.2" and talk to a server script running in
// another ns-3 on another computer that has an ns-3 node with IP
// address "10.1.1.3"
//
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0", "0.0.0.2");
Ipv4InterfaceContainer i = ipv4.Assign (d);
```

你将使用 application helpers 来生成流量，就像你在任何 ns-3 模拟脚本做的一样。注意，如下代码片段中显示的服务器地址必须与另一段代码中分配给服务器节点的 IP 地址相对应，该段代码类似于上述片段。

```
uint32_t packetSize = 1024;
uint32_t maxPacketCount = 2000;
Time interPacketInterval = Seconds (0.001);
UdpEchoClientHelper client ("10.1.1.3", 9);
client.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));
client.SetAttribute ("Interval", TimeValue (interPacketInterval));
client.SetAttribute ("PacketSize", UintegerValue (packetSize));
ApplicationContainer apps = client.Install (n.Get (0));
apps.Start (Seconds (1.0));
apps.Stop (Seconds (2.0));
```

与其他的 ns-3 网络设备一样，Emu 网络设备和 helper 提供对 ASCII 跟踪功能和 pcap（译者注：Packet Capture）跟踪功能的访问。你可以用与其他网络设备类似的方法开启跟踪：

```
EmuHelper::EnablePcapAll ("emu-udp-echo-client");
```

如下代码是使用了 Emu 网络设备的一个客户机脚本的例子：examples/emu-udp-echo-client.cc 和 examples/emu-udp-echo-server.cc，他们在 http://code.nsnam.org/craigdo/ns-3-emu/中。

6.2.2 Tap Net Device

The Tap Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the Tap device is merged.

6.3 实现 Implementation

或许 Emu 和 Tap 的实现中最不寻常的部分是要求以超级用户权限来执行一些代码。我们提供了一个以 root 身份运行的“creator”小程序来处理高权限要求的套接字工作，而不是强迫用户以 root 身份运行整个模拟过程。

我们为 Emu 和 Tap 设备做类似的工作。从较高的层次来看，方法 CreateSocket 创建一个本地进程间的（Unix）套接字，进行 fork 调用并执行 creation 程序。该创建程序以 suid root 来运行（译者注：suid 可以理解为”set uid”），该程序创建一个原始套接字并通过一个 Unix 套接字将创建的原始套接字的文件描述符发回，该 Unix 套接字是先前作为参数传递给创建程序的那个套接字。该原始套接字作为一个 SCM_RIGHTS 类型的控制消息（也叫做辅助数据(ancillary data)）来传递。

6.3.1 Emu 网络设备 Emu Net Device

Emu 网络设备使用了 ns-3 的线程和多线程实时调度器的扩展部分。Emu 网络设备中需要做的工作是在该设备启动时来完成的（EmuNetDevice::StartDevice ()）。属性“Start”提供一个启动该网络设备的模拟时间。在这个特定的时刻（默认设置为 t=0），套接字创建函数被调用并如上所述来执行。还可以用属性“Stop”指定一个时刻来停止该设备。

一旦该（混杂模式）的套接字被创建完成，我们将他绑定到一个接口名称，该接口名称也作为属性来提供（“DeviceName”），该属性在内部被存储为 m_deviceName：

```
struct ifreq ifr;
bzero (&ifr, sizeof(ifr));
strncpy ((char *)ifr.ifr_name, m_deviceName.c_str (), IFNAMSIZ);
int32_t rc = ioctl (m_sock, SIOCGIFINDEX, &ifr);
struct sockaddr_ll ll;
bzero (&ll, sizeof(ll));
ll.sll_family = AF_PACKET;
ll.sll_ifindex = m_sll_ifindex;
ll.sll_protocol = htons(ETH_P_ALL);
rc = bind (m_sock, (struct sockaddr *)&ll, sizeof (ll));
```

在混杂模式的原始套接字创建完成后，将酿成另一个单独的进程来从该套接字进行读取操作，并且链路状态被置为 Up。

```
m_readThread = Create<SystemThread> (
MakeCallback (&EmuNetDevice::ReadThread, this));
m_readThread->Start ();
NotifyLinkUp ();
```

函数 EmuNetDevice::ReadThread 在无限循环中对该混杂模式大的原始套接字进行读取操作，并使用实时模拟器的扩展来调度包的接受。

```
for (;;)
```

```
{
...
len = recvfrom (m_sock, buf, bufferSize, 0, (struct sockaddr *)&addr,
&addrSize);
...
DynamicCast<RealtimeSimulatorImpl> (Simulator::GetImplementation ())->
ScheduleRealtimeNow (
MakeEvent (&EmuNetDevice::ForwardUp, this, buf, len));
...
}
```

由模板化的 DynamicCast 函数开始的行需要注释一下。该行代码用方法 Simulator::GetImplementation 获得对模拟器实现对象的访问，然后转化为实时模拟器实现来使用实时调度方法 ScheduleRealtimeNow。这个函数将导致一个处理新近收到的包的处理程序被排定，该排定为了在当前的真实时间值来执行。这同样将导致如下情况，即当该排定的事件（EmuNetDevice::ForwardUp）被启动时，模拟时钟将被推进至该真实时间值。

函数 ForwardUp 的运作与多数的 ns-3 网络设备的方法相类似。首先基于目标地址对包进行过滤。对于 Emu 设备，MAC 目标地址将是该设备的地址，而不是真实设备的硬件地址。然后 headers 将被剥离，命中跟踪钩子（trace hook）。最后，该包由 ns-3 的协议栈向上传递，这个过程使用该网络设备的接收回调函数（receive callback function）。

发送包的过程显示如下。我们首先给要发送的 ns-3 Packet 添加以太网头部（header）和尾部（trailer）。源地址对应于 Emu 网络设备的地址，而不是底层本地设备的 MAC 地址。这便是 MAC 地址欺骗被完成的地方。在添加完尾部后，我们将包在网络设备的队列上入队（enqueue）和出队（dequeue）来命中跟踪钩子。

```
header.SetSource (source);
header.SetDestination (destination);
header.SetLengthType (packet->GetSize ());
packet->AddHeader (header);
EthernetTrailer trailer;
trailer.CalcFcs (packet);
packet->AddTrailer (trailer);
m_queue->Enqueue (packet);
packet = m_queue->Dequeue ();
struct sockaddr_ll ll;
bzero (&ll, sizeof (ll));
ll.sll_family = AF_PACKET;
```

ll.sll_ifindex = m_sll_ifindex;
ll.sll_protocol = htons(ETH_P_ALL);
rc = sendto (m_sock, packet->PeekData (), packet->GetSize (), 0,
reinterpret_cast<struct sockaddr *> (&ll), sizeof (ll));
最后，我们将包发送给原始套接字，由套接字将包发送至真实的网络。

6.3.2 Tap Net Device

The Tap Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the Tap device is merged.

NS-3 manual note – 06 Emulation

by gaocong

6. 仿真 Emulation

ns-3 被设计为可以集成到测试床以及虚拟机环境。我们通过提供两类网络设备来陈述这个需求。第一类，Emu NetDevice 使得 ns-3 模拟过程能够在“real”网络上发送数据。第二类，Tap NetDevice 使得“real”主机能够参与 ns-3 的模拟过程，就像其他模拟节点一样。ns-3 的模拟过程可以由任意组合的 Emu 以及 Tap 设备来构造。

我们想要支持的用例（use-case）之一是测试床。这类环境的一个具体例子是 ORBIT 测试床。ORBIT 是一个实验室性质的仿真试验网络，该网络由 400 个 802.11 无线节点以二维网格分布而成。我们使用 ORBIT 的“imaging”进程在 ORBIT 阵列上加载和运行 ns-3 的模拟过程，这便是与 ORBIT 的集成。我们使用 Emu NetDevices 来驱动测试床上的硬件，而且我们可以使用 ns-3 的跟踪函数和日志记录函数来积累结果，也可以使用 ORBIT 的数据收集技术。参看 http://www.orbit-lab.org/ 来获得 ORBIT 测试床的细节。

下图显示了一个这样的模拟过程：(译者注：原图缺)

图 6.1：Figure 6.1: 测试床仿真的实现举例 Example Implementation of Testbed Emulation.

可以看到有彼此分开的主机，每个主机都运行“global”模拟的一个子集。我们没有使用 ns-3 的 channel 来连接主机，而是使用了由测试床提供的真实的硬件。这使得 ns-3 中附着于模拟节点的 application 和协议栈能够通过真实的硬件来通信。

我们期望这个配置的主要用途是在现实世界的网络环境中生成反复的实验结果，该实验结果包括 ns-3 所有的跟踪、日志记录、可视化以及统计信息收集工具。

在本质上相反的配置中，我们使运行应用程序和协议栈的“real”机器来与 ns-3 的模拟过程集成。这考虑到与真实机器相连接的大型网络模拟，并且开启了虚拟化。下图显示了一个这样的模拟过程：(译者注：原图缺)

图 6.2：Figure 6.2: 仿真信道的实现概述 Implementation overview of emulated channel.

这里你将看到一个单独的主机，该主机上运行着一些虚拟机。图中部的一个虚拟机上运行着一个 ns-3 模拟过程。该模拟有一些节点，这些节点与一些 ns-3 的 applications 和协议栈相关联，这些 applications 和协议栈通过 ns-3 模拟的网络设备与一个 ns-3 的 channel 交流。

图的左边和右边还有两个虚拟机。这些虚拟机正在运行本地（Linux 的）应用程序和协议栈。虚拟机通过 Linux Tap 网络设备被连接到模拟过程中。Tap 设备的用户模式的处理程序在模拟过程中被实例化，并被附着到一个代理节点，该代理节点代表模拟过程中的虚拟机。这些处理程序使得本地虚拟机上的 Tap 设备像模拟过程中的虚拟机上的 ns-3 网络设备一样工作。这点也使得本地虚拟机上的软件和协议套件确信他们是与 ns-3 的模拟信道连接着的。

我们期望这个环境的典型用例是 分析本地应用程序和协议套件在大规模的 ns-3 模拟网络面前的行为。

6.1 行为 Behavior

6.1.1 Emu 网络设备 Emu Net Device

Emu 网络设备使得模拟节点能够在真实的网络中发送和接收包。仿真的网络设备依赖一个特定的处在混杂（promiscuous）模式的接口。网络设备打开一个原始套接字（raw socket）并与该接口绑定。我们执行 MAC 欺骗来将模拟网络流量与可能流进/出该主机的其他网络流量分开。

通常，仿真的网络设备的用例在较小的模拟中，这些模拟通过特定的接口与外界连接。例如，你可以构造一些虚拟机并通过主机模式（host-only）的网络来连接他们。为了使用仿真的网络设备，你需要将所有 host-only 模式的接口都设置为混杂模式并且提供适当的设备名称，比如”eth1″。

你还可以在测试床环境下使用 Emu 网络设备，运行模拟的主机有着驱动该测试床硬件的特定接口。你还需要将这个特定接口设置为混杂模式并为 ns-3 的仿真网络设备提供一个适当的设备名称。这样环境的一个例子是上述的 ORBIT 测试床。

只有当底层接口启动并处在混杂模式时，Emu 网络设备才能工作。包将通过该设备被发送出去，但是我们使用 MAC 欺骗。默认情况下，MAC 地址使用 Organizationally Unique Identifier (OUI) 00:00:00 作为基准来产生（译者注：Organizationally Unique Identifier）。这个制造商代码没有分配给任何组织，所以应该不会与任何真实硬件产生冲突。

总是又用户来判断是否可以在网络中使用这些 MAC 地址，即不会与网络中的其他东西（包括其他使用 Emu 设备的模拟过程）产生冲突。如果你在彼此分开的模拟中使用仿真网络设备，那么你必须考虑全局的 MAC 地址分配问题，并确保所有模拟中的 MAC 地址都是独一无二的。仿真网络设备遵从方法 SetAddress 提供的 MAC 地址，所以你可以手工来完成。对于更大规模的模拟，你可能想在 MAC 地址分配函数中来设置 OUI。

与仿真网络设备对应的 IP 地址是在模拟过程中生成的地址，这些地址是通过 helper functions 用普通的方法生成的。由于我们使用了 MAC 欺骗，所以 ns-3 的网络栈不会和任意本地网络栈发生冲突。

与所有的 ns-3 设备一样，仿真网络设备也具有一个 helper function。一个独特的方面是没有与底层媒介相关联的 channel。我们不知道外部的媒介是什么，所以没有对他进行抽象建模。需要清楚的首要事情是这对于静态全局路由蕴含的影响。全局路由器模块尝试在 channels 上“行走”来寻找相邻的网络。由于没有 channel，所以全局路由器将无法做到这一点，则你必须使用动态路由协议来将路由包含到 Emu-based 网络中，比如 OLSR（译者注：OLSR）。

6.1.2 Tap 网络设备 Tap Net Device

The Tap Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the Tap device is merged.

6.2 用法 Usage

6.2.1 Emu Net Device

一旦模拟的网络配置完成，Emu 网络设备的用法是直截了当的。由于涉及到这个设备的多数工作都是在模拟开始之前的网络配置中，所以你可能需要回顾一些 ns-3 wiki 上的 HOWTO 页面，这些页面描述了 如何 使用 WMware 构建虚拟的测试网络以及如何运行使用 Emu 网络设备的模拟的例子（客户机 服务器）。

http://www.nsnam.org/wiki/index.php/HOWTO_use_VMware_to_set_up_virtual_networks_(Windows)
http://www.nsnam.org/wiki/index.php/HOWTO_use_ns-3_scripts_to_drive_real_hardware_(experimental)

一旦完成了配置，使用 Emu 设备需要脚本修改就很少了。主要的结构上的不同是你将需要为每个节点创建一个 ns-3 模拟脚本。在上述 HOWTOs 的情况中，有一个客户机脚本和一个服务器脚本。唯一的“challenge”是将地址设置正确。

与其他所有的 ns-3 设备一样，我们为 Emu 提供一个 helper class。下边的代码段举例说明了 如何声明一个 EmuHelper 并用他将属性“DeviceName”设置为“eth1”，以及如何把 Emu 设备安装到一组节点上。在上述 HOWTO 的情况中，客户端和服务器端都需要这样做。

```
EmuHelper emu;

emu.SetAttribute (“DeviceName”, StringValue (“eth1″));

NetDeviceContainer d = emu.Install (n);
```

另一个可能要求的仅有的变化是保证客户机和服务器的地址空间（MAC 和 IP）是互相兼容的。首先，MAC 地址被设置为一个独一无二的值，该值在两端都准所周知的（这里以一端为例）。

```
//

// We’ve got the devices in place.  Since we’re using MAC address

// spoofing under the sheets, we need to make sure that the MAC addresses

// we have assigned to our devices are unique.  Ns-3 will happily

// automatically assign the same MAC addresses to the devices in both halves
```

// of our two-script pair, so let's go ahead and just manually change them

// to something we ensure is unique.

//

Ptr<NetDevice> nd = d.Get (0);

Ptr<EmuNetDevice> ed = nd->GetObject<EmuNetDevice> ();

ed->SetAddress ("00:00:00:00:00:02");

接着客户机或服务器的 IP 将使用 helpers 用普通方法设置。

//

// We've got the "hardware" in place. Now we need to add IP addresses.

// This is the server half of a two-script pair. We need to make sure

// that the addressing in both of these applications is consistent, so

// we use provide an initial address in both cases. Here, the client

// will reside on one machine running ns-3 with one node having ns-3

// with IP address "10.1.1.2" and talk to a server script running in

// another ns-3 on another computer that has an ns-3 node with IP

// address "10.1.1.3"

//

Ipv4AddressHelper ipv4;

ipv4.SetBase ("10.1.1.0", "255.255.255.0", "0.0.0.2");

Ipv4InterfaceContainer i = ipv4.Assign (d);

你将使用 application helpers 来生成流量，就像你在任何 ns-3 模拟脚本做的一样。注意，如下代码片段中显示的服务器地址必须与另一段代码中分配给服务器节点的 IP 地址相对应，该段代码类似于上述片段。

uint32_t packetSize = 1024;

uint32_t maxPacketCount = 2000;

Time interPacketInterval = Seconds (0.001);

UdpEchoClientHelper client ("10.1.1.3″, 9);

client.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));

client.SetAttribute ("Interval", TimeValue (interPacketInterval));

client.SetAttribute ("PacketSize", UintegerValue (packetSize));

ApplicationContainer apps = client.Install (n.Get (0));

apps.Start (Seconds (1.0));

apps.Stop (Seconds (2.0));

与其他的 ns-3 网络设备一样，Emu 网络设备和 helper 提供对 ASCII 跟踪功能和 pcap（译者注：Packet Capture）跟踪功能的访问。你可以用与其他网络设备类似的方法开启跟踪：

EmuHelper::EnablePcapAll ("emu-udp-echo-client");

如下代码是使用了 Emu 网络设备的一个客户机脚本的例子：examples/emu-udp-echo-client.cc 和 examples/emu-udp-echo-server.cc，他们在 http://code.nsnam.org/craigdo/ns-3-emu/中。

6.2.2 Tap Net Device

The Tap Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the Tap device is merged.

6.3 实现 Implementation

或许 Emu 和 Tap 的实现中最不寻常的部分是要求以超级用户权限来执行一些代码。我们提供了一个以 root 身份运行的“creator”小程序来处理高权限要求的套接字工作，而不是强迫用户以 root 身份运行整个模拟过程。

我们为 Emu 和 Tap 设备做类似的工作。从较高的层次来看，方法 CreateSocket 创建一个本地进程间的（Unix）套接字，进行 fork 调用并执行 creation 程序。该创建程序以 suid root 来运行（译者注：suid 可以理解为”set uid”），该程序创建一个原始套接字并通过一个 Unix 套接字将创建的原始套接字的文件描述符发回，该 Unix 套接字是先前作为参数传递给创建程序的那个套接字。该原始套接字作为一个 SCM_RIGHTS 类型的控制消息（也叫做辅助数据 (ancillary data)）来传递。

6.3.1 Emu 网络设备 Emu Net Device

Emu 网络设备使用了 ns-3 的线程和多线程实时调度器的扩展部分。Emu 网络设备中需要做的工作是在该设备启动时来完成的（EmuNetDevice::StartDevice ()）。属性“Start”提供一个启动该网络设备的模拟时间。在这个特定的时刻（默认设置为 t=0），套接字创建函数被调用并如上所述来执行。还可以用属性“Stop”指定一个时刻来停止该设备。

一旦该（混杂模式）的套接字被创建完成，我们将他绑定到一个接口名称，该接口名称也作为属性来提供（“DeviceName”），该属性在内部被存储为 m_deviceName：

```
struct ifreq ifr;

bzero (&ifr, sizeof(ifr));

strncpy ((char *)ifr.ifr_name, m_deviceName.c_str (), IFNAMSIZ);

int32_t rc = ioctl (m_sock, SIOCGIFINDEX, &ifr);

struct sockaddr_ll ll;
```

bzero (&ll, sizeof(ll));

ll.sll_family = AF_PACKET;

ll.sll_ifindex = m_sll_ifindex;

ll.sll_protocol = htons(ETH_P_ALL);

rc = bind (m_sock, (struct sockaddr *)&ll, sizeof (ll));

在混杂模式的原始套接字创建完成后，将酿成另一个单独的进程来从该套接字进行读取操作，并且链路状态被置为 Up。

m_readThread = Create<SystemThread> (

MakeCallback (&EmuNetDevice::ReadThread, this));

m_readThread->Start ();

NotifyLinkUp ();

函数 EmuNetDevice::ReadThread 在无限循环中对该混杂模式大的原始套接字进行读取操作，并使用实时模拟器的扩展来调度包的接受。

for (;;)

{

...

len = recvfrom (m_sock, buf, bufferSize, 0, (struct sockaddr *)&addr,

&addrSize);

...

DynamicCast<RealtimeSimulatorImpl> (Simulator::GetImplementation ())->

```
ScheduleRealtimeNow (

MakeEvent (&EmuNetDevice::ForwardUp, this, buf, len));

...

}
```

由模板化的 DynamicCast 函数开始的行需要注释一下。该行代码用方法 Simulator::GetImplementation 获得对模拟器实现对象的访问，然后转化为实时模拟器实现来使用实时调度方法 ScheduleRealtimeNow。这个函数将导致一个处理新近收到的包的处理程序被排定，该排定为了在当前的真实时间值来执行。这同样将导致如下情况，即当该排定的事件（EmuNetDevice::ForwardUp）被启动时，模拟时钟将被推进至该真实时间值。

函数 ForwardUp 的运作与多数的 ns-3 网络设备的方法相类似。首先基于目标地址对包进行过滤。对于 Emu 设备，MAC 目标地址将是该设备的地址，而不是真实设备的硬件地址。然后 headers 将被剥离，命中跟踪钩子（trace hook）。最后，该包由 ns-3 的协议栈向上传递，这个过程使用该网络设备的接收回调函数（receive callback function）。

发送包的过程显示如下。我们首先给要发送的 ns-3 Packet 添加以太网头部（header）和尾部（trailer）。源地址对应于 Emu 网络设备的地址，而不是底层本地设备的 MAC 地址。这便是 MAC 地址欺骗被完成的地方。在添加完尾部后，我们将包在网络设备的队列上入队（enqueue）和出队（dequeue）来命中跟踪钩子。

```
header.SetSource (source);

header.SetDestination (destination);

header.SetLengthType (packet->GetSize ());

packet->AddHeader (header);

EthernetTrailer trailer;

trailer.CalcFcs (packet);
```

```
packet->AddTrailer (trailer);

m_queue->Enqueue (packet);

packet = m_queue->Dequeue ();

struct sockaddr_ll ll;

bzero (&ll, sizeof (ll));

ll.sll_family = AF_PACKET;

ll.sll_ifindex = m_sll_ifindex;

ll.sll_protocol = htons(ETH_P_ALL);

rc = sendto (m_sock, packet->PeekData (), packet->GetSize (), 0,

reinterpret_cast<struct sockaddr *> (&ll), sizeof (ll));
```

最后，我们将包发送给原始套接字，由套接字将包发送至真实的网络。

6.3.2 Tap Net Device

The Tap Net Device is scheduled for inclusion in ns-3.4 at the writing of this section. We will include details as soon as the Tap device is merged.

NS-3 manual note – 07 Packets

Posted on 07/10/2009 by xa_ceshi

NS-3 manual note – 07 Packets

by gaocong

7. 包 Packets
ns 中 Packet 框架的设计主要侧重以下重要的用例：
在引入新的包头部和尾部类型时避免修改模拟器的核心。
使之与真实世界的代码和系统进行集成的容易度最大化。
使之更好的支持 fragmentation（译者注：分组，在 IP 层将打包文件切成适当大小的程序)，defragmentation（译者注：重组）以及在无线系统中很重要的 concatenation。
使得包对象的内存管理具有高效率。
仿真应用程序中的实际的（actual）应用程序数据和虚假的（dummy）应用程序字节。
ns 的包对象包含一个字节缓冲区：协议头部和尾部由用户提供的串行化（serialization）与还原串行化（deserialization）例程在这个字节缓冲区被串行化。我们期望该字节缓冲区的内容与实现该协议的真实网络中的包的内容做到 bit 匹配。
分组和重组在这样的上下文中可以很自然的实现：由于我们拥有一个真实字节的缓冲区，我们可以将他拆分成多个片段以及重组这些片段。我们期望这样的选择将使得 我们的包数据结构可以很容易地在 Linux-style skbuff 或 BSD-style mbuf 的范围内进行包装，进而在模拟器中与真实世界的内核代码进行集成。我们还希望将模拟器实时接入真实世界的网络可以很容易。
因为模拟过程的开发者经常要在包对象中存储真实包中不存在的数据（比如时间戳或类似的 in-band 数据），所以 ns 的包类能够给每个包存储额外的 “Tags”，该 Tags 是 16 字节的数据。任何包都可以存储任意个数的独一无二的 Tags，其中的每个 Tag 都由自身的 C++类型独一无二地标识。这些 tags 使得将每个模型的数据附加到包变得容易，而且不用为主要的包类和包设施打补丁。
包对象的内存管理是完全自动的和极为高效的：应用程序层的内存有效载荷可以用由零填充的虚拟字节缓冲区来建模，该内存只有在被用户显示请求或者包被分组时才被分配。此外，通过写时复制（Copy on Write）的技术，复制、添加以及删除包的头部或尾部已经被优化到 virtually free。
包（消息）是模拟器的基本对象。从性能和资源管理的角度来看，包（消息）的设计是很重要的。有很多设计包的方法，不同方法中也有折中方案。特别地，易用性、性能以及安全的接口设计是需要互相权衡的。

包对象的设计上有如下一些需求：
对象的创建、管理以及删除应该尽量简单，避免内存泄漏（memory leak）和堆溢出（heap corruption）
包应该支持串行化（serialization）和还原串行化（serialization），以支持网络仿真（network emulation）。
包应该支持分组（fragmentation）和 concatenation(multiple packets in a data link frame), 尤其是对无线的支持。
包应该很自然地携带实际的应用程序数据，或者如果仅仅是仿真应用程序且没必要携带虚假的（dummy）字节，此时应该使用较小的包，这些较小的包仅包含头部和有效载荷大小的记录，而没有实际的应用程序数据在包中传送。
包应该使得在 mbuf 上进行 BSD-like operations 变得容易，以支持可移植的操作系统栈。
附加信息应该被支持，比如跨层信息（cross-layer）的 tag。
7.1 包设计概述 Packet design overview
ns-2 的包对象包含一个与协议头部对应的 C++结构缓冲区，与 ns-2 不同，ns-3 中的每个包都包含一个字节缓冲区和一个 Tag 列表：
该字节缓冲区存储串行化过的内容，该内容是被加入到包的组块（chunks，译者注：有意义的信息单元）。这些组块的串行化表示被期望与真实网络的包做到 bit 匹配（尽管你不必这样做），这意味着包缓冲区的内容被期望和真实的包一样。包还可以用任意大小的由零填充的内存有效载荷来创建，因为并没有实际的内存被分配。
tag 列表在包中存储一个任意大小的集合，该集合由任意的用户提供的数据结构组成。每个 Tag 都是由他的类型来独一无二地标识。tag 列表中只允许每个数据结构的一个实例。这些 tags 通常包含包的跨层信息（cross-layer）或者流标识符（flow identifier）（例如一些你在实际线路中不会找到的比特）。tag 列表中存储的每个 tag 至多可以有 16 字节。附加更大的数据结构的尝试将触发运行时的崩溃。16 字节的限制是一个可修改的编译常量。
Figure 7.1: 包类的实现概述 Implementation overview of Packet class.
图 fig:packets 是包实现的一个高层概述，关于字节缓冲区实现的细节在图 fig:buffer 中提供。在 ns-3 中，包的字节缓冲区与 Linux skbuff 或 BSD mbuf 是类似的。他是包中实际数据的串行化表示。tag 列表是一些额外项的容器，这些项可以方便模拟的进行。如果包对象被转化成仿真包，并放到真实的网络中，那么 tags 将被剥离，字节缓冲区将被直接复制到真实的包。
包类拥有值语义（value semantics）：他可以被自由地复制、分配到栈上以及作为实参传递给函数。无论何时复制一个实例，所有底层的数据都不被复制，因为使用的是 “写时复制 copy-on-write” (COW)语义。包实例可以通过值传递来作为函数的实参，不产生任何性能上的开销。

对字节缓冲区进行添加和删除的基本类是 class Header 和 class Trailer。头部更为常见，但一下的讨论也大量适用于使用尾部的协议。需要加入包实例或者从包实例中删除的所有协议头部都应该从抽象 Header 基类派生并实现如下私有的纯虚方法：

ns3::Header::SerializeTo()

ns3::Header::DeserializeFrom()

ns3::Header::GetSerializedSize()

ns3::Header::PrintTo()

前三个函数用来将协议控制信息串行化进缓冲区以及从缓冲区还原串行化。例如，可以定义这样一个类 class TCPHeader : public Header。TCPHeader 的对象通常由一些私有数据（比如序列号）和公共接口访问函数（比如对输入进行边界检查），包缓冲区中 TCPHeader 的底层表示是 20 个串行化的字节（加上 TCP 选项）。函数 TCPHeader::SerializeTo()将被设计为把这 20 个字节以网络字节顺序合适地写入包中。最后一个函数用来定义 Header 对象如何将自己打印至输出流（output stream）。

类似地，用户定义的 Tags 也可以被附加到包。与 Headers 不同，tags 没有被串行化进邻近的缓冲区，而是存储在一个数组中。默认情况下，tag 的大小限制为 16 字节。Tags 可以被灵活地定义为任意类型，但是无论在任何时候，Tags 缓冲区中只能存在某个类型的一个实例（译者注：即可以有任意多个类型的数据，但同一个类型的实例只能有一个）。具体实现使用模板来为所有 Tag 类型生成合适的 Add()，Remove()，Peek()函数集合。

7.2 包接口 Packet interface

包对象的公共成员函数如下：

7.2.1 构造函数

```
/**
* Create an empty packet with a new uid (as returned
* by getUid).
*/
Packet ();
/**
* Create a packet with a zero-filled payload.
* The memory necessary for the payload is not allocated:
* it will be allocated at any later point if you attempt
* to fragment this packet or to access the zero-filled
* bytes. The packet is allocated with a new uid (as
* returned by getUid).
*
* param size the size of the zero-filled payload
```

```
*/
Packet (uint32_t size);
```

7.2.2 添加和删除缓冲区数据

以下代码仅为 Header 类生成，类似的函数在 Trailer 类中也存在。

```
/**
* Add header to this packet. This method invokes the
* ns3::Header::serializeTo method to request the header to serialize
* itself in the packet buffer.
*
* param header a reference to the header to add to this packet.
*/
void Add (Header const &header);
/**
* Deserialize header from this packet. This method invokes the
* ns3::Header::deserializeFrom method to request the header to deserialize
* itself from the packet buffer. This method does not remove
* the data from the buffer. It merely reads it.
*
* param header a reference to the header to deserialize from the buffer
*/
void Peek (Header &header);
/**
* Remove a deserialized header from the internal buffer.
* This method removes the bytes read by Packet::peek from
* the packet buffer.
*
* param header a reference to the header to remove from the internal buffer.
*/
void Remove (Header const &header);
/**
* Add trailer to this packet. This method invokes the
* ns3::Trailer::serializeTo method to request the trailer to serialize
* itself in the packet buffer.
*
* param trailer a reference to the trailer to add to this packet.
*/
```

7.2.3 添加和删除 Tags

```
/**
```

```
* Attach a tag to this packet. The tag is fully copied
* in a packet-specific internal buffer. This operation
* is expected to be really fast.
*
* param tag a pointer to the tag to attach to this packet.
*/
template <typename T>
void AddTag (T const &tag);
/**
* Remove a tag from this packet. The data stored internally
* for this tag is copied in the input tag if an instance
* of this tag type is present in the internal buffer. If this
* tag type is not present, the input tag is not modified.
*
* This operation can be potentially slow and might trigger
* unexpectedly large memory allocations. It is thus
* usually a better idea to create a copy of this packet,
* and invoke removeAllTags on the copy to remove all
* tags rather than remove the tags one by one from a packet.
*
* param tag a pointer to the tag to remove from this packet
* returns true if an instance of this tag type is stored
*          in this packet, false otherwise.
*/
template <typename T>
bool RemoveTag (T &tag);
/**
* Copy a tag stored internally to the input tag. If no instance
* of this tag is present internally, the input tag is not modified.
*
* param tag a pointer to the tag to read from this packet
* returns true if an instance of this tag type is stored
*          in this packet, false otherwise.
*/
template <typename T>
bool PeekTag (T &tag) const;
/**
* Remove all the tags stored in this packet. This operation is
```

* much much faster than invoking removeTag n times.
*/
void RemoveAllTags (void);

7.2.4 分组 Fragmentation

/**
* Create a new packet which contains a fragment of the original
* packet. The returned packet shares the same uid as this packet.
*
* param start offset from start of packet to start of fragment to create
* param length length of fragment to create
* returns a fragment of the original packet
*/
Packet CreateFragment (uint32_t start, uint32_t length) const;
/**
* Concatenate the input packet at the end of the current
* packet. This does not alter the uid of either packet.
*
* param packet packet to concatenate
*/
void addAtEnd (Packet packet);
/oncatenate the input packet at the end of the current
* packet. This does not alter the uid of either packet.
*
* param packet packet to concatenate
*/
void AddAtEnd (Packet packet);
/**
* Concatenate the fragment of the input packet identified
* by the offset and size parameters at the end of the current
* packet. This does not alter the uid of either packet.
*
* param packet to concatenate
* param offset offset of fragment to copy from the start of the input packet
* param size size of fragment of input packet to copy.
*/
void AddAtEnd (Packet packet, uint32_t offset, uint32_t size);
/**
* Remove size bytes from the end of the current packet

```
* It is safe to remove more bytes that what is present in
* the packet.
*
* param size number of bytes from remove
*/
void RemoveAtEnd (uint32_t size);
/**
* Remove size bytes from the start of the current packet.
* It is safe to remove more bytes that what is present in
* the packet.
*
* param size number of bytes from remove
*/
void RemoveAtStart (uint32_t size);
```

7.2.5 其他 Miscellaneous

```
/**
* returns the size in bytes of the packet (including the zero-filled
*          initial payload)
*/
uint32_t GetSize (void) const;
/**
* If you try to change the content of the buffer
* returned by this method, you will die.
*
* returns a pointer to the internal buffer of the packet.
*/
uint8_t const *PeekData (void) const;
/**
* A packet is allocated a new uid when it is created
* empty or with zero-filled payload.
*
* returns an integer identifier which uniquely
*          identifies this packet.
*/
uint32_t GetUid (void) const;
```

7.3 使用 Headers Using Headers

walk through an example of adding a UDP header

7.4 使用 Tags Using Tags

walk through an example of adding a flow ID

7.5 使用分组 Using Fragmentation

walk through an example of link-layer fragmentation/reassembly

7.6 实例程序 Sample program

以下实例程序(ns3/samples/main-packet.cc)举例说明了 Packet，Header 以及 Tag 类的一些用法。

```
/* -*-	Mode:C++; c-basic-offset:4; tab-width:4; indent-tabs-mode:nil -*- */
#include “ns3/packet.h”
#include “ns3/header.h”
#include <iostream>
using namespace ns3;
/* A sample Header implementation
*/
class MyHeader : public Header {
public:
MyHeader ();
virtual ~MyHeader ();
void SetData (uint16_t data);
uint16_t GetData (void) const;
private:
virtual void PrintTo (std::ostream &os) const;
virtual void SerializeTo (Buffer::Iterator start) const;
virtual void DeserializeFrom (Buffer::Iterator start);
virtual uint32_t GetSerializedSize (void) const;
uint16_t m_data;
};
MyHeader::MyHeader ()
{}
MyHeader::~MyHeader ()
{}
void
MyHeader::PrintTo (std::ostream &os) const
{
os << “MyHeader data=” << m_data << std::endl;
}
uint32_t
MyHeader::GetSerializedSize (void) const
{
```

```
return 2;
}
void
MyHeader::SerializeTo (Buffer::Iterator start) const
{
// serialize in head of buffer
start.WriteHtonU16 (m_data);
}
void
MyHeader::DeserializeFrom (Buffer::Iterator start)
{
// deserialize from head of buffer
m_data = start.ReadNtohU16 ();
}
void
MyHeader::SetData (uint16_t data)
{
m_data = data;
}
uint16_t
MyHeader::GetData (void) const
{
return m_data;
}
/* A sample Tag implementation
*/
struct MyTag {
uint16_t m_streamId;
};
static TagRegistration<struct MyTag> g_MyTagRegistration (“ns3::MyTag”,
0);
static void
Receive (Packet p)
{
MyHeader my;
p.Peek (my);
p.Remove (my);
std::cout << “received data=” << my.GetData () << std::endl;
```

```
struct MyTag myTag;
p.PeekTag (myTag);
}
int main (int argc, char *argv[])
{
Packet p;
MyHeader my;
my.SetData (2);
std::cout << “send data=2″ << std::endl;
p.Add (my);
struct MyTag myTag;
myTag.m_streamId = 5;
p.AddTag (myTag);
Receive (p);
return 0;
}
```

7.7 实现细节 Implementation details

7.7.1 私有成员变量

包对象的接口提供对一些私有数据的访问：

```
Buffer m_buffer;
Tags m_tags;
uint32_t m_uid;
static uint32_t m_global_uid;
```

每个包都拥有一个缓冲区，一个 Tags 对象以及一个 32-bit 的独一无二地 ID(m_uid)。由一个静态成员变量记录已分配的 UID。注意真实网络的包没有 UID。因此 UID 通常是包对象中作为 Tag 存储的一个数据。尽管如此，由于 UID 是一个特例，他在模拟过程中经常被用到，所以将他存储在成员变量中更为方便。

7.7.2 缓冲区的实现

类 Buffer 代表字节缓冲区。他的大小是自动调整的，以存放用户预先提供的和后期添加的数据。他的实现通过创建大小为已经使用过的最大缓冲区的新缓冲区来使得缓冲区调整大小的次数达到最小值。该最大缓冲区的大小通过运行时记录包的最大尺寸来获得。

类 Header 和类 Trailer 的作者需要知道类 Buffer 的公共 API(add summary here)。

字节缓冲区的实现如下：

```
struct BufferData {
uint32_t m_count;
uint32_t m_size;
```

uint32_t m_initialStart;
uint32_t m_dirtyStart;
uint32_t m_dirtySize;
uint8_t m_data[1];
};
struct BufferData *m_data;
uint32_t m_zeroAreaSize;
uint32_t m_start;
uint32_t m_size;
BufferData::m_count: BufferData 结构的引用计数
BufferData::m_size: BufferData 结构中存储的数据缓冲区的大小
BufferData::m_initialStart: 从数据缓冲区开始到数据最先被插入的地方的偏移量
BufferData::m_dirtyStart: 从数据缓冲区开始（所有 Buffer 都对该开始拥有一个引用）到该 BufferData 实例目前已写数据的地方的偏移量
BufferData::m_dirtySize: 目前已写数据区域的大小
BufferData::m_data: 指向数据缓冲区的指针
Buffer::m_zeroAreaSize: size of zero area which extends before m_initialStart
Buffer::m_start: 从缓冲区开始到被这个实例缓冲区所使用区域的偏移量
Buffer::m_size: BufferData 结构中被该 Buffer 使用的区域的大小
Figure 7.2: 包的字节缓冲区的实现概述
此数据结构总结在图 fig:buffer 中。每个 Buffer 都拥有一个指向某个 BufferData 实例的指针。多数 Buffer 能够共享同一个底层的 BufferData 并因此增加该 BufferData 的引用计数。如果需要改变在 Dirty Area 中里 BufferData 的内容，且引用计数值不是 1，那么将首先创建该 BufferData 的一个副本，然后再完成改变状态的操作。

7.7.3 Tags 的实现 Tags implementation

Tags 通过一个指针来实现，该指针指向 TagData 数据结构的链表的开始。每个 TagData 结构指向链表中的下一个 TagData（当指针包含零时表示链表的结尾）。每个 TagData 都包含一个整型的独一无二的 id，该 id 标识存储在该 TagData 中的 tag 的类型。

struct TagData {
struct TagData *m_next;
uint32_t m_id;
uint32_t m_count;
uint8_t m_data[Tags::SIZE];
};
class Tags {

struct TagData *m_next;
};
添加一个 tag 即是在链表头部插入一个新的 TagData。考虑某个 tag，用户需要在链表中找到相关的 TagData，并将他的数据复制到用户的数据结构中。删除 tag 以及更新 tag 的内容要求在执行操作前对链表进行深复制（deep copy，译者注：与之对应的是 shallow copy）。另一方面，复制包对象和他的 tags 即是要复制 TagData 的头指针以及增加包的引用计数。
Tags 通过 Tag 类型与底层 id 之间独一无二的映射来寻找。这就是为什么一个包对象中任何 Tag 至多只能有一个实例。Tag 类型与底层 id 之间的映射通过以下注册过程来执行：
/* A sample Tag implementation
*/
struct MyTag {
uint16_t m_streamId;
};
add description of TagRegistration for printing
7.7.4 内存管理
Describe free list.
Describe dataless vs. data-full packets.
7.7.5 写时复制的语义 Copy-on-write semantics
字节缓冲区和 tag 链表的当前实现基于写时复制（COW，Copy On Write）。关于 COW 的介绍可以在 Scott Meyer's "More Effective C++", items 17 and 29 中找到。本设计特性以及公共接口方面借鉴了 Georgia Tech Network Simulator 中的包设计。我们对 COW 的实现使用了用户自定义的引用计数智能指针类（reference counting smart pointer class）。
COW 意味着复制包而不进行修改开销很小（指 CPU 和内存的使用），此外对包进行修改的开销也很小。COW 的实现的关键是要能够在对包的修改发生之前检测到该修改会对包的状态做出修改，进而触发一个对数据的完全复制：即 COW 系统需要能够检测到某个操作是"dirty"，然后必须执行一个真正的复制。
Dirty operations:
Packet::RemoveTag()
Packet::Add()
both versions of ns3::Packet::AddAtEnd()
Non-dirty operations:
Packet::AddTag()
Packet::RemoveAllTags()
Packet::PeekTag()
Packet::Peek()

Packet::Remove()
Packet::CreateFragment()
Packet::RemoveAtStart()
Packet::RemoveAtEnd()
Dirty operations 总是比 non-dirty operations 要慢，有时要慢好几个数量级。尽管如此，对于常见的用例，dirty operations 也已经被优化过，这意味着多数情况下，这些操作将不会触发数据的复制，因此也就比较快。

7. 包 Packets

ns 中 Packet 框架的设计主要侧重以下重要的用例：

在引入新的包头部和尾部类型时避免修改模拟器的核心。

使之与真实世界的代码和系统进行集成的容易度最大化。

使之更好的支持 fragmentation（译者注：分组，在 IP 层将打包文件切成适当大小的程序)，defragmentation（译者注：重组）以及在无线系统中很重要的 concatenation。

使得包对象的内存管理具有高效率。

仿真应用程序中的实际的（actual）应用程序数据和虚假的（dummy）应用程序字节。

ns 的包对象包含一个字节缓冲区：协议头部和尾部由用户提供的串行化（serialization）与还原串行化（deserialization）例程在这个字节缓冲区被串行化。我们期望该字节缓冲区的内容与实现该协议的真实网络中的包的内容做到 bit 匹配。

分组和重组在这样的上下文中可以很自然的实现：由于我们拥有一个真实字节的缓冲区，我们可以将他拆分成多个片段以及重组这些片段。我们期望这样的选择将使得 我们的包数据结构可以很容易地在 Linux-style skbuff 或 BSD-style mbuf 的范围内进行包装，进而在模拟器中与真实世界的内核代码进行集成。我们还希望将模拟器实时接入真实世界的网络可以很容易。

因为模拟过程的开发者经常要在包对象中存储真实包中不存在的数据（比如时间戳或类似的 in-band 数据），所以 ns 的包类能够给每个包存储额外的 “Tags”，

该 Tags 是 16 字节的数据。任何包都可以存储任意个数的独一无二的 Tags，其中的每个 Tag 都由自身的 C++类型独一无二地标识。这些 tags 使得将每个模型的数据附加到包变得容易，而且不用为主要的包类和包设施打补丁。

包对象的内存管理是完全自动的和极为高效的：应用程序层的内存有效载荷可以用由零填充的虚拟字节缓冲区来建模，该内存只有在被用户显示请求或者包被分组时才被分配。此外，通过写时复制（Copy on Write）的技术，复制、添加以及删除包的头部或尾部已经被优化到 virtually free。

包（消息）是模拟器的基本对象。从性能和资源管理的角度来看，包（消息）的设计是很重要的。有很多设计包的方法，不同方法中也有折中方案。特别地，易用性、性能以及安全的接口设计是需要互相权衡的。

包对象的设计上有如下一些需求：

对象的创建、管理以及删除应该尽量简单，避免内存泄漏（memory leak）和堆溢出（heap corruption）包应该支持串行化（serialization）和还原串行化（serialization），以支持网络仿真（network emulation）。

包应该支持分组（fragmentation）和 concatenation(multiple packets in a data link frame), 尤其是对无线的支持。

包应该很自然地携带实际的应用程序数据，或者如果仅仅是仿真应用程序且没必要携带虚假的（dummy）字节，此时应该使用较小的包，这些较小的包仅包含头部和有效载荷大小的记录，而没有实际的应用程序数据在包中传送。

包应该使得在 mbuf 上进行 BSD-like operations 变得容易，以支持可移植的操作系统栈。

附加信息应该被支持，比如跨层信息（cross-layer）的 tag。

7.1 包设计概述 Packet design overview

ns-2 的包对象包含一个与协议头部对应的 C++结构缓冲区，与 ns-2 不同，ns-3 中的每个包都包含一个字节缓冲区和一个 Tag 列表：

该字节缓冲区存储串行化过的内容，该内容是被加入到包的组块（chunks，译者注：有意义的信息单元）。这些组块的串行化表示被期望与真实网络的包做到

bit 匹配（尽管你不必这样做），这意味着包缓冲区的内容被期望和真实的包一样。包还可以用任意大小的由零填充的内存有效载荷来创建，因为并没有实际的内存被分配。

tag 列表在包中存储一个任意大小的集合，该集合由任意的用户提供的数据结构组成。每个 Tag 都是由他的类型来独一无二地标识。tag 列表中只允许每个数据结构的一个实例。这些 tags 通常包含包的跨层信息（cross-layer）或者流标识符（flow identifier）（例如一些你在实际线路中不会找到的比特）。tag 列表中存储的每个 tag 至多可以有 16 字节。附加更大的数据结构的尝试将触发运行时的崩溃。16 字节的限制是一个可修改的编译常量。

(译者注：原图缺)

Figure 7.1: 包类的实现概述 Implementation overview of Packet class.

图 fig:packets 是包实现的一个高层概述，关于字节缓冲区实现的细节在图 fig:buffer 中提供。在 ns-3 中，包的字节缓冲区与 Linux skbuff 或 BSD mbuf 是类似的。他是包中实际数据的串行化表示。tag 列表是一些额外项的容器，这些项可以方便模拟的进行。如果包对象被转化成仿真包，并放到真实的网络中，那么 tags 将被剥离，字节缓冲区将被直接复制到真实的包。

包类拥有值语义（value semantics）：他可以被自由地复制、分配到栈上以及作为实参传递给函数。无论何时复制一个实例，所有底层的数据都不被复制，因为使用的是 “写时复制 copy-on-write” (COW)语义。包实例可以通过值传递来作为函数的实参，不产生任何性能上的开销。

对字节缓冲区进行添加和删除的基本类是 class Header 和 class Trailer。头部更为常见，但一下的讨论也大量适用于使用尾部的协议。需要加入包实例或者从包实例中删除的所有协议头部都应该从抽象 Header 基类派生并实现如下私有的纯虚方法：

ns3::Header::SerializeTo()

ns3::Header::DeserializeFrom()

ns3::Header::GetSerializedSize()

ns3::Header::PrintTo()

前三个函数用来将协议控制信息串行化进缓冲区以及从缓冲区还原串行化。例如，可以定义这样一个类 class TCPHeader : public Header。TCPHeader 的对象通常由一些私有数据（比如序列号）和公共接口访问函数（比如对输入进行边界检查），包缓冲区中 TCPHeader 的底层表示是 20 个串行化的字节（加上 TCP 选项）。函数 TCPHeader::SerializeTo()将被设计为把这 20 个字节以网络字节顺序合适地写入包中。最后一个函数用来定义 Header 对象如何将自己打印至输出流（output stream）。

类似地，用户定义的 Tags 也可以被附加到包。与 Headers 不同，tags 没有被串行化进邻近的缓冲区，而是存储在一个数组中。默认情况下，tag 的大小限制为 16 字节。Tags 可以被灵活地定义为任意类型，但是无论在任何时候，Tags 缓冲区中只能存在某个类型的一个实例（译者注：即可以有任意多个类型的数据，但同一个类型的实例只能有一个）。具体实现使用模板来为所有 Tag 类型生成合适的 Add()，Remove()，Peek()函数集合。

7.2 包接口 Packet interface

包对象的公共成员函数如下：

7.2.1 构造函数

```
/**

* Create an empty packet with a new uid (as returned

* by getUid).

*/

Packet ();

/**

* Create a packet with a zero-filled payload.

* The memory necessary for the payload is not allocated:

* it will be allocated at any later point if you attempt
```

```
* to fragment this packet or to access the zero-filled

* bytes. The packet is allocated with a new uid (as

* returned by getUid).

*

* param size the size of the zero-filled payload

*/

Packet (uint32_t size);
```

7.2.2 添加和删除缓冲区数据

以下代码仅为 Header 类生成，类似的函数在 Trailer 类中也存在。

```
/**

* Add header to this packet. This method invokes the

* ns3::Header::serializeTo method to request the header to serialize

* itself in the packet buffer.

*

* param header a reference to the header to add to this packet.

*/

void Add (Header const &header);

/**

* Deserialize header from this packet. This method invokes the
```

```
* ns3::Header::deserializeFrom method to request the header to deserialize
* itself from the packet buffer. This method does not remove
* the data from the buffer. It merely reads it.
*
* param header a reference to the header to deserialize from the buffer
*/
void Peek (Header &header);
/**
* Remove a deserialized header from the internal buffer.
* This method removes the bytes read by Packet::peek from
* the packet buffer.
*
* param header a reference to the header to remove from the internal buffer.
*/
void Remove (Header const &header);
/**
* Add trailer to this packet. This method invokes the
* ns3::Trailer::serializeTo method to request the trailer to serialize
* itself in the packet buffer.
```

*

* param trailer a reference to the trailer to add to this packet.

*/

7.2.3 添加和删除 Tags

/**

* Attach a tag to this packet. The tag is fully copied

* in a packet-specific internal buffer. This operation

* is expected to be really fast.

*

* param tag a pointer to the tag to attach to this packet.

*/

template <typename T>

void AddTag (T const &tag);

/**

* Remove a tag from this packet. The data stored internally

* for this tag is copied in the input tag if an instance

* of this tag type is present in the internal buffer. If this

* tag type is not present, the input tag is not modified.

*

```
* This operation can be potentially slow and might trigger
* unexpectedly large memory allocations. It is thus
* usually a better idea to create a copy of this packet,
* and invoke removeAllTags on the copy to remove all
* tags rather than remove the tags one by one from a packet.
*
* param tag a pointer to the tag to remove from this packet
* returns true if an instance of this tag type is stored
*          in this packet, false otherwise.
*/
template <typename T>
bool RemoveTag (T &tag);
/**
* Copy a tag stored internally to the input tag. If no instance
* of this tag is present internally, the input tag is not modified.
*
* param tag a pointer to the tag to read from this packet
* returns true if an instance of this tag type is stored
*          in this packet, false otherwise.
```

```
*/
template <typename T>
bool PeekTag (T &tag) const;
/**
* Remove all the tags stored in this packet. This operation is
* much much faster than invoking removeTag n times.
*/
void RemoveAllTags (void);
```

7.2.4 分组 Fragmentation

```
/**
* Create a new packet which contains a fragment of the original
* packet. The returned packet shares the same uid as this packet.
*
* param start offset from start of packet to start of fragment to create
* param length length of fragment to create
* returns a fragment of the original packet
*/
Packet CreateFragment (uint32_t start, uint32_t length) const;
/**
```

```
* Concatenate the input packet at the end of the current

* packet. This does not alter the uid of either packet.

*

* param packet packet to concatenate

*/

void addAtEnd (Packet packet);

/oncatenate the input packet at the end of the current

* packet. This does not alter the uid of either packet.

*

* param packet packet to concatenate

*/

void AddAtEnd (Packet packet);

/**

* Concatenate the fragment of the input packet identified

* by the offset and size parameters at the end of the current

* packet. This does not alter the uid of either packet.

*

* param packet to concatenate

* param offset offset of fragment to copy from the start of the input packet
```

```
* param size size of fragment of input packet to copy.

*/

void AddAtEnd (Packet packet, uint32_t offset, uint32_t size);

/**

* Remove size bytes from the end of the current packet

* It is safe to remove more bytes that what is present in

* the packet.

*

* param size number of bytes from remove

*/

void RemoveAtEnd (uint32_t size);

/**

* Remove size bytes from the start of the current packet.

* It is safe to remove more bytes that what is present in

* the packet.

*

* param size number of bytes from remove

*/

void RemoveAtStart (uint32_t size);
```

7.2.5 其他 Miscellaneous

```
/**
* returns the size in bytes of the packet (including the zero-filled
*              initial payload)
*/
uint32_t GetSize (void) const;
/**
* If you try to change the content of the buffer
* returned by this method, you will die.
*
* returns a pointer to the internal buffer of the packet.
*/
uint8_t const *PeekData (void) const;
/**
* A packet is allocated a new uid when it is created
* empty or with zero-filled payload.
*
* returns an integer identifier which uniquely
*              identifies this packet.
```

*/

uint32_t GetUid (void) const;

7.3 使用 Headers Using Headers

walk through an example of adding a UDP header

7.4 使用 Tags Using Tags

walk through an example of adding a flow ID

7.5 使用分组 Using Fragmentation

walk through an example of link-layer fragmentation/reassembly

7.6 实例程序 Sample program

以下实例程序(ns3/samples/main-packet.cc)举例说明了 Packet，Header 以及 Tag 类的一些用法。

/* -*- Mode:C++; c-basic-offset:4; tab-width:4; indent-tabs-mode:nil -*- */

#include “ns3/packet.h”

#include “ns3/header.h”

#include <iostream>

using namespace ns3;

/* A sample Header implementation

*/

class MyHeader : public Header {

public:

```
MyHeader ();

virtual ~MyHeader ();

void SetData (uint16_t data);

uint16_t GetData (void) const;

private:

virtual void PrintTo (std::ostream &os) const;

virtual void SerializeTo (Buffer::Iterator start) const;

virtual void DeserializeFrom (Buffer::Iterator start);

virtual uint32_t GetSerializedSize (void) const;

uint16_t m_data;

};

MyHeader::MyHeader ()

{}

MyHeader::~MyHeader ()

{}

void

MyHeader::PrintTo (std::ostream &os) const

{

os << “MyHeader data=” << m_data << std::endl;
```

```
}

uint32_t

MyHeader::GetSerializedSize (void) const

{

return 2;

}

void

MyHeader::SerializeTo (Buffer::Iterator start) const

{

// serialize in head of buffer

start.WriteHtonU16 (m_data);

}

void

MyHeader::DeserializeFrom (Buffer::Iterator start)

{

// deserialize from head of buffer

m_data = start.ReadNtohU16 ();

}

void
```

```
MyHeader::SetData (uint16_t data)

{

m_data = data;

}

uint16_t

MyHeader::GetData (void) const

{

return m_data;

}

/* A sample Tag implementation

*/

struct MyTag {

uint16_t m_streamId;

};

static TagRegistration<struct MyTag> g_MyTagRegistration (“ns3::MyTag”,
0);

static void

Receive (Packet p)

{

MyHeader my;
```

```
p.Peek (my);

p.Remove (my);

std::cout << “received data=” << my.GetData () << std::endl;

struct MyTag myTag;

p.PeekTag (myTag);

}

int main (int argc, char *argv[])

{

Packet p;

MyHeader my;

my.SetData (2);

std::cout << “send data=2″ << std::endl;

p.Add (my);

struct MyTag myTag;

myTag.m_streamId = 5;

p.AddTag (myTag);

Receive (p);

return 0;

}
```

7.7 实现细节 Implementation details

7.7.1 私有成员变量

包对象的接口提供对一些私有数据的访问：

Buffer m_buffer;

Tags m_tags;

uint32_t m_uid;

static uint32_t m_global_uid;

每个包都拥有一个缓冲区，一个 Tags 对象以及一个 32-bit 的独一无二地 ID(m_uid)。由一个静态成员变量记录已分配的 UID。注意真实网络的包没有 UID。因此 UID 通常是包对象中作为 Tag 存储的一个数据。尽管如此，由于 UID 是一个特例，他在模拟过程中经常被用到，所以将他存储在成员变量中更为方便。

7.7.2 缓冲区的实现

类 Buffer 代表字节缓冲区。他的大小是自动调整的，以存放用户预先提供的和后期添加的数据。他的实现通过创建大小为已经使用过的最大缓冲区的新缓冲区来使得缓冲区调整大小的次数达到最小值。该最大缓冲区的大小通过运行时记录包的最大尺寸来获得。

类 Header 和类 Trailer 的作者需要知道类 Buffer 的公共 API(add summary here)。

字节缓冲区的实现如下：

struct BufferData {

uint32_t m_count;

uint32_t m_size;

uint32_t m_initialStart;

uint32_t m_dirtyStart;

uint32_t m_dirtySize;

uint8_t m_data[1];

};

struct BufferData *m_data;

uint32_t m_zeroAreaSize;

uint32_t m_start;

uint32_t m_size;

BufferData::m_count: BufferData 结构的引用计数

BufferData::m_size: BufferData 结构中存储的数据缓冲区的大小

BufferData::m_initialStart: 从数据缓冲区开始到数据最先被插入的地方的偏移量

BufferData::m_dirtyStart: 从数据缓冲区开始（所有 Buffer 都对该开始拥有一个引用）到该 BufferData 实例目前已写数据的地方的偏移量

BufferData::m_dirtySize: 目前已写数据区域的大小

BufferData::m_data: 指向数据缓冲区的指针

Buffer::m_zeroAreaSize: size of zero area which extends before m_initialStart

Buffer::m_start: 从缓冲区开始到被这个实例缓冲区所使用区域的偏移量

Buffer::m_size: BufferData 结构中被该 Buffer 使用的区域的大小

(译者注：原图缺)

Figure 7.2: 包的字节缓冲区的实现概述

此数据结构总结在图 fig:buffer 中。每个 Buffer 都拥有一个指向某个 BufferData 实例的指针。多数 Buffer 能够共享同一个底层的 BufferData 并因此增加该 BufferData 的引用计数。如果需要改变在 Dirty Area 中里 BufferData 的内容，且引用计数值不是 1，那么将首先创建该 BufferData 的一个副本，然后再完成改变状态的操作。

7.7.3 Tags 的实现 Tags implementation

Tags 通过一个指针来实现，该指针指向 TagData 数据结构的链表的开始。每个 TagData 结构指向链表中的下一个 TagData（当指针包含零时表示链表的结尾）。每个 TagData 都包含一个整型的独一无二的 id，该 id 标识存储在该 TagData 中的 tag 的类型。

```
struct TagData {

struct TagData *m_next;

uint32_t m_id;

uint32_t m_count;

uint8_t m_data[Tags::SIZE];

};

class Tags {

struct TagData *m_next;

};
```

添加一个 tag 即是在链表头部插入一个新的 TagData。考虑某个 tag，用户需要在链表中找到相关的 TagData，并将他的数据复制到用户的数据结构中。删除 tag 以及更新 tag 的内容要求在执行操作前对链表进行深复制（deep copy，译者注：与之对应的是 shallow copy）。另一方面，复制包对象和他的 tags 即是要复制 TagData 的头指针以及增加包的引用计数。

Tags 通过 Tag 类型与底层 id 之间独一无二的映射来寻找。这就是为什么一个包对象中任何 Tag 至多只能有一个实例。Tag 类型与底层 id 之间的映射通过以下注册过程来执行：

/* A sample Tag implementation

*/

struct MyTag {

uint16_t m_streamId;

};

add description of TagRegistration for printing

7.7.4 内存管理

Describe free list.

Describe dataless vs. data-full packets.

7.7.5 写时复制的语义 Copy-on-write semantics

字节缓冲区和 tag 链表的当前实现基于写时复制（COW，Copy On Write）。关于 COW 的介绍可以在 Scott Meyer's "More Effective C++", items 17 and 29 中找到。本设计特性以及公共接口方面借鉴了 Georgia Tech Network Simulator 中的包设计。我们对 COW 的实现使用了用户自定义的引用计数智能指针类（reference counting smart pointer class）。

COW 意味着复制包而不进行修改开销很小（指 CPU 和内存的使用），此外对包进行修改的开销也很小。COW 的实现的关键是要能够在对包的修改发生之前检测到该修改会对包的状态做出修改，进而触发一个对数据的完全复制：即 COW 系统需要能够检测到某个操作是"dirty"，然后必须执行一个真正的复制。

Dirty operations:

Packet::RemoveTag()

Packet::Add()

both versions of ns3::Packet::AddAtEnd()

Non-dirty operations:

Packet::AddTag()

Packet::RemoveAllTags()

Packet::PeekTag()

Packet::Peek()

Packet::Remove()

Packet::CreateFragment()

Packet::RemoveAtStart()

Packet::RemoveAtEnd()

Dirty operations 总是比 non-dirty operations 要慢，有时要慢好几个数量级。尽管如此，对于常见的用例，dirty operations 也已经被优化过，这意味着多数情况下，这些操作将不会触发数据的复制，因此也就比较快。

NS-3 manual note – 08 Sockets APIs

Posted on 07/11/2009 by xa_ceshi

NS-3 manual note – 08 Sockets APIs

by gaocong

8. 套接字 APIs

套接字 API 是一套经久不衰的 API，用户的应用程序用之来访问内核中的网络服务。类似于 Unix 的文件句柄，“socket”是一个抽象，他使得应用程序能够与互联网上的其他主机连接并在服务中进行可靠的字节流（reliable byte streams）以及不可靠的数据报（unreliable datagrams）的交换。

ns-3 提供两种类型的套接字 API，理解二者的不同很重要。第一类是 native ns-3 API，第二类使用 native API 的服务来提供 POSIX-like API，作为整个应用程序进程的一部分。这两类 API 都努力和 Unix 系统上应用程序作者所熟悉的典型套接字 API 相近，但是 POSIX 更加近似于真实系统的套接字 API。

8.1 ns-3 的套接字 API sockets API

ns-3 的 native 套接字 API 为不同的传输层协议(TCP, UDP)以及包套接字提供接口。在将来还可能为 Netlink-like 套接字提供接口。尽管如此，用户需要注意这些 API 的语义和真实系统并不是完全相同的(参看下一节中的一个和真实系统对应的一个 API)。

class ns3::Socket 定义在 src/node/socket.cc,h 中。读者将发现很多公共成员函数和真实的套接字函数调用是对应的，并且所有其他东西也同样对待，我们试图与 Posix 套接字 API 来对应。需要注意以下问题：

ns-3 的应用程序掌握 Socket 对象的智能指针，而不是文件描述符。
没有同步(synchronous)API 的以及“blocking”API 的概念。事实上，应用程序与套接字之间交互的模型是异步(asynchronous)I/O，这种方式通常在真实系统中不存在。
没有使用 C-style 的套接字地址结构。
该 API 不是一套完整的套接字 API，例如并不支持全部套接字选项或是全部函数变型。
许多调用使用类 ns3::Packet 来在应用程序和套接字之间转移数据。对于将

“Packets”通过流套接字传递的人，这多少有些滑稽，因为想着这些包仅仅是这个层次上的空想的字节缓冲区。

8.1.1 基本操作和调用

(译者注：原图缺)

图 8.1: native 套接字 API 的实现概述

8.1.1.1 创建套接字

要使用套接字的应用程序必须先创建套接字。在真实系统中，可以调用 socket()来实现：

```
int
socket(int domain, int type, int protocol);
```

该函数在系统中创建一个套接字并返回一个整数描述符。

在 ns-3 中，在底层没有和系统调用等价的东西，所以我们使用如下模型：factory 对象可以创建套接字。每个 factory 能够创建某一类型的套接字，如果某个特定类型的套接字能够在某个给定节点上被创建，那么能够创建此类套接字的 factory 一定被集成(aggregate)进了该节点。

```
static Ptr CreateSocket (Ptr node, TypeId tid);
```

可以传递给这个方法的 TypeIds 的例子有：TcpSocketFactory、PacketSocketFactory、以及 UdpSocketFactory。

该方法返回一个指向 Socket 对象的智能指针。举例：

```
Ptr n0;
// Do some stuff to build up the Node’s internet stack
Ptr localSocket = Socket::CreateSocket (n0, TcpSocketFactory::GetTypeId ());
```

在某些 ns-3 代码中，套接字将不是由用户的主程序显式创建的，而是由某个 ns-3 的应用程序创建的。例如，对于 class ns3::OnOffApplication，函数 StartApplication()执行套接字的创建，该应用程序掌握该套接字的指针。

8.1.1.2 使用套接字

以下是真实的实现中为 TCP client 进行的典型的套接字调用序列：

```
sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
bind(sock, ...);
connect(sock, ...);
```

send(sock, ...);
recv(sock, ...);
close(sock);
所有这些调用在 ns-3 中都有类似的东西，这里我们只关注两个方面。真实系统中的套接字的多数使用需要一种在应用程序和内核之间管理 I/O 的方法。这些模型包括 blocking sockets、signal-based I/O，以及带有轮询(polling)的 non-blocking sockets。在 ns-3 中，我们使用回调机制来支持第四种模式，类似于 POSIX 中的 asynchronous I/O。

在这个模型中，对于发送方，如果由于缓冲区不足而导致 send()调用将要失败，则应用程序暂停对更多数据的发送直到一个注册在 SetSendCallback()的回调函数被调用。应用程序还可以通过调用 GetTxAvailable()来询问套接字有多少可用空间。发送数据的典型事件序列(忽略了连接的建立)可能为：
SetSendCallback (MakeCallback(&HandleSendCallback));
Send ();
Send ();
...
// Send fails because buffer is full
(wait until HandleSendCallback() is called)
(HandleSendCallback() is called by socket, since space now available)
Send (); // Start sending again
类似地，对于接收方，套接字的用户并不阻塞 recv()调用，而是由应用程序用 SetRecvCallback()来设置一个回调，在该回调中套接字将通知应用程序什么时候、多大量的数据需要被读，接着应用程序调用 Recv()来读数据直到不再有数据可读。

8.1.2 Packet vs. buffer variants

Send()和 Recv()本别支持两种变型：
virtual int Send (Ptr
p) = 0;
int Send (const uint8_t* buf, uint32_t size);
Ptr
Recv (void);
int Recv (uint8_t* buf, uint32_t size);
non-Packet 的变型是为旧的 API 留下的。当调用 Send()的原始缓冲区变型时，该缓冲区被立即写入一个 Packet，然后 Send (Ptr
p)被调用。

用户可能觉得将 Packet 传递给流套接字(比如 TCP)在语义上比较奇怪。不必让名字困扰你，将 ns3::Packet 想成空想的字节缓冲区。存在一些原因使得 Packet 的变型在 ns-3 里成为首选：

用户可以使用 packet 的 Tags 功能来做一些事情，比如对流 ID 或是其他的 helper 数据来进行编码。
用户可以开发 copy-on-write 的实现来避免内存复制(对于接收方，到 uint8_t* buf 的转化可能导致额外的拷贝)。
Packet 的使用与剩余的 ns-3 API 更加对应一致。

8.1.3 发送伪(dummy)数据

有时用户想要模拟器仅仅假装在包中存在实际的数据载荷(比如需要计算传输延迟 transmission delay)，但却不想真正产生或是消耗数据。这在 ns-3 中被直接支持：让应用程序调用 Create
(size);而不是 Create
(buffer, size);。类似地，给原始缓冲区变型的指针参数传入零值可以起到同样的效果。注意，如果随后有代码试图读该 Packet 的数据缓冲区，则该虚假的缓冲区将在该点被转变为一个真实的缓冲区(由零填充)，虚假缓冲区的效率也就随之丧失。

8.1.4 Socket options

to be completed (译者注：原文缺)

8.1.5 Socket errno

to be completed (译者注：原文缺)

8.1.6 Example programs

to be completed (译者注：原文缺)

8.2 POSIX-like 套接字 API

这个性能正在开发中，预定于在 ns-3.5 中包含。参看
http://code.nsnam.org/mathieu/ns-3-simu 来获得细节。

以下是从 Mathieu 2008 年 4 月 4 日发布在 ns-developers list 上的帖子摘录的。

“总的来说，目标是所有 posix/socket API 要定义在 src/process/simu.h 中：每个 posix 类型和函数需要用 simu_或 SIMU_作为前缀来重新定义，进而避免名字冲突(请尽管提出更好的前缀)。

进程通过调用 ProcessManager::Create 来被创建，并被附加到该 ProcessManager 实例上。所以，如果该 ProcessManager(ProcessManager 被集成进 Node，见 src/helper/process-helper.cc)在模拟终止时被删除，那么系统将自动回收与每个 manager 相关联的每个进程的所有资源。当某个应用程序从他的 main 函数”exits”时将发生同样的事情。

用来举例的应用程序定义了两个 posix “processes”：函数 ClientProgram 在 localhost 的 2000 端口上创建一个 udp 套接字，函数 ServerProgram 在 localhost 的 2000 端口上创建一个 udp 套接字。代码并不能正常工作，因为我目前还没有正确的理解 simu_read 的细节，我计划在某个时间解决这些问题。

我认为这个努力是值得的，以下是一些原因的要点：

使得移植真实世界的应用程序变得_非常_容易。
使得新用户编写应用程序变得非常容易，因为他们可以阅读 bsd socket api 的参考手册和文档来直接编写代码。
能够用来编写同时在模拟和真实世界都能工作的应用程序。为了做到这一点，你只需要使用 simu_ API 来编写的你的应用程序，然后你可以再编译时选择你想要使用该 API 的哪个实现：你可以选择一个将所有调用都转发到系统的 BSD socket API 的实现，或者另一个将所有调用都转发到被附加上的 ProcessManager。我将不实现转发到系统 BSD 套接字的版本，因为那个价值不大。
欢迎对所有的 API 进行评论。对 gsoc 项目中真实世界的代码整合感兴趣的同学也应该考虑留意一下这里的讨论。”

NS-3 manual note – 09 Node and Internet Stack

Posted on 07/13/2009 by xa_ceshi

NS-3 manual note – 09 Node and Internet Stack

by gaocong

9. 节点和 Internet 栈

本章描述 ns-3 的节点是如何装配起来的，并简单介绍包是如何通过基于 Internet 的节点的。
(译者注：原图缺)
图 9.1 高层的节点体系结构

在 ns-3 中，节点是 class Node 的实例。该类可以被继承，但概念上的模型是我们给该类集成(aggregate)或插入对象，而不是定义他的子类。可以把单纯的 ns-3 节点想象成一个计算机的空壳，可以给他添加 NetDevices(比如各种外设卡)以及其他内部结构，包括协议和应用程序。fig:node 展示了节点对象包含应用程序的列表(初始状态时列表为空)，NetDevices 的列表(初始状态时列表为空)，一个独一无二的整数 ID，以及一个系统 ID(用在分布式模拟中 distributed simulation)。

我们的设计试图避免在以下方面给 Node 基类，应用程序以及 NetDevice 增加过多的依赖：

IP 版本，或者 IP 是否在 Node 中被使用到。
IP 栈的实现细节。
从软件的角度看，应用程序的底层接口对应于基于 C 的套接字 API。NetDevice 对象的上层接口对应于 Linux 栈的设备无关子层。在这之间的任何东西都可以根据需要被集成起来。

让我们了解一下协议多路分配器(protocol demultiplexer)。我们想让来自第二层的帧被传递至正确的第三层协议，比如 IPv4。该多路分配器的功能是为接收包而注册回调。回调是基于第二层中的 EtherType 来被索引的。

许多不同类型的高层协议可能被连接到 NetDevice，比如 IPv4、IPv6、ARP、MPLS、IEEE 802.1x，以及 packet 套接字。因此，使用基于回调的多路分配器可以避免为所有这些协议使用一个公共的基类，该基类容易产生问题，因为可能需要注册很多不同类型的对象(包括 packet 套接字)。

NetDevice 将包带上如下签名传递给某个回调：

/**
* param device a pointer to the net device which is calling this callback
* param packet the packet received
* param protocol the 16 bit protocol number associated with this packet.
* This protocol number is expected to be the same protocol number
* given to the Send method by the user on the sender side.
* param address the address of the sender
* returns true if the callback could handle the packet successfully,
* false otherwise.
*/
typedef Callback,Ptr,uint16_t,const Address &> ReceiveCallback;
类 Node 中有一个函数与该签名匹配：

private:bool ReceiveFromDevice (Ptr device, Ptr,uint16_t protocol, const Address &from);

尽管如此，用户不必直接访问该函数。当用户调用 uint32_t AddDevice (Ptr device)时，这个函数的实现设置回调(该函数返回该节点上那个 NetDevice 的 ifIndex)。函数 ReceiveFromDevice 做什么呢？这里，对应于所匹配的 EtherType，他在自己的回调列表查找另一个回调。该回调称为 ProtocolHandler，定义如下：
typedef Callback,Ptr,uint16_t,const Address &> ProtocolHandler;
上层协议或对象应该提供这样一个函数，并通过调用 Node::RegisterProtocolHandler ();将他注册至 ProtocolHandlers 的列表。 例如，如果将 Ipv4 集成进 Node，则可以通过如下调用来将 Ipv4 接收函数与 protocol handler 进行注册：

RegisterProtocolHandler (MakeCallback (&Ipv4L3Protocol::Receive, ipv4),Ipv4L3Protocol::PROT_NUMBER, 0);对于 Ipv6、Arp、等等也是一样的。

9.1 NodeList

每个被创建的节点都被自动加入 NodeList。类 NodeList 提供一个 Add()方法和 C++ iterators 来使得用户可以遍历链表并通过节点的整数标识符来获取该节点的指针。

9.2 Internet 栈的集成

上述的class Node很有用，因为对象必须被集成进Node来提供所需的节点功能。

ns-3 源代码中目录 src/internet-stack 提供了与 TCP/IPv4 相关的构件的实现，包括 IPv4、ARP、UDP、TCP，以及其他相关的协议。Internet 节点不是类 Node 的子类，他们只是集成了一些与 IPv4 相关的对象的 Node。他们可以被手动放在一起，后者通过一个 helper 函数 AddInternetStack ()，该函数进行如下操作：

```
void AddInternetStack (Ptr node)
{
// Create layer-3 protocols
Ptr ipv4 = CreateObject ();
Ptr arp = CreateObject ();
ipv4->SetNode (node);
arp->SetNode (node);

// Create an L4 demux
Ptr ipv4L4Demux = CreateObject ();

// Create transport protocols and insert them into the demux
Ptr udp = CreateObject ();
Ptr tcp = CreateObject ();

ipv4L4Demux->SetNode (node);
udp->SetNode (node);
tcp->SetNode (node);

ipv4L4Demux->Insert (udp);
ipv4L4Demux->Insert (tcp);

// Add factories for instantiating transport protocol sockets
Ptr udpFactory = CreateObject ();
Ptr tcpFactory = CreateObject ();
Ptr ipv4Impl = CreateObject ();

udpFactory->SetUdp (udp);
tcpFactory->SetTcp (tcp);
ipv4Impl->SetIpv4 (ipv4);
```

```
// Aggregate all of these new objects to the node
node->AggregateObject (ipv4);
node->AggregateObject (arp);
node->AggregateObject (ipv4Impl);
node->AggregateObject (udpFactory);
node->AggregateObject (tcpFactory);
node->AggregateObject (ipv4L4Demux);
}
```

9.2.1 Internet 节点的结构

Internet 节点(一类 ns-3 节点，通过集成来拥有一个或多个 IP 栈)的内部结构如下。

9.2.1.1 第三层协议

在最底层，即紧挨 NetDevice 之上，是”第三层”协议，包括 IPv4、IPv6，以及 ARP。这些协议提供如下关键的方法和数据成员：

```
class Ipv4L3Protocol : public Object
{
public:
// Add an Ipv4 interface corresponding to the provided NetDevice
uint32_t AddInterface (Ptr device);

// Receive function that can be bound to a callback, for receiving
// packets up the stack
void Receive( Ptr device,Ptr p,uint16_t protocol,const Address &from);

// Higher-level layers call this method to send a packet
// down the stack to the MAC and PHY layers
//
void Send (Ptr packet,Ipv4Address source,Ipv4Address destination,uint8_t
protocol);

private:
Ipv4InterfaceList m_interfaces;
```

```
// Protocol handlers
}
```
还有很多函数(比如 Forward())，但从体系结构的角度，我们只关注上述 4 项。

首先，注意函数 Receive()拥有一个与 class Node 中 ReceiveCallback 项匹配的签名。当 AddInterface()被调用时，这个函数指针被插入到节点的协议处理器。实际的注册过程通过如下的语句来完成：

```
RegisterProtocolHandler ( MakeCallback (&Ipv4Protocol::Receive,
ipv4),Ipv4L3Protocol::PROT_NUMBER, 0);
```
对象 Ipv4L3Protocol 被集成进 Node。只存在一个 Ipv4L3Protocol 这样的对象。需要给 Ipv4L3Protocol 对象发送包的高层协议可以调用 GetObject()来获得一个指针：

```
Ptr ipv4 = m_node->GetObject ();
if (ipv4 != 0)
{
ipv4->Send (packet, saddr, daddr, PROT_NUMBER);
}
```
这个类很好地展示了 我们在 ns-3 中用来绑定对象的两种技术：回调(callbacks)和对象集成(object aggregation)。

一旦 IPv4 判断出某个包是给本地节点的，那么他将把包向上转发至栈。这用如下函数来完成：

```
void
Ipv4L3Protocol::ForwardUp (Ptr p,Ipv4Header const&ip,Ptr
incomingInterface)
{
NS_LOG_FUNCTION (this << p << &ip);

Ptr demux = m_node->GetObject ();
Ptr protocol = demux->GetProtocol (ip.GetProtocol ());
protocol->Receive (p, ip.GetSource (), ip.GetDestination (),
incomingInterface);
}
```
第一步是找到集成进的 Ipv4L4Demux 对象，接着基于 IP 协议号码询问该对象来查找合适的 Ipv4L4Protocol。例如，TCP 是以协议号码 6 在 demux 中注册的。最后，Ipv4L4Protocol 上的 Receive()函数被调用，即 TcpL4Protocol::Receive。我们目前还没有引入类 Ipv4Interface。基本来说，所有 NetDevice 都要与这样

的设备的某种 IPv4 表示形式配对。在 Linux 中，class Ipv4Interface 大致上对应于 struct in_device，主要的目的是提供关于接口的 address-family 详细信息。

9.2.1.2 第四层协议和套接字

我们下来描述传输层协议、套接字，以及应用程序是如何连接在一起的。概括地说，每个传输层协议的实现都是一个套接字工厂。

例如，对于需要新的套接字的应用程序，要创建一个 UDP 套接字，应用程序将使用如下一段代码：

```
Ptr udpSocketFactory = GetNode ()->GetObject ();
Ptr m_socket = socketFactory->CreateSocket ();
m_socket->Bind (m_local_address);
...
```

上述代码将查询该节点并获得指向他的 UDP 套接字工厂的指针，并将用一个类似于 C 套接字 API 的 API 来使用该套接字，例如 Connect()和 Send()。参看 ns-3 套接字这一章来获得更多信息。

我们描述了套接字工厂(比如 class Udp)和套接字(可以被专门化，比如 class UdpSocket)。还有一些对象与将包多路分配给一个或多个接收套接字有关。其中关键的是 class Ipv4EndPointDemux。该多路分配器存储 class Ipv4EndPoint 的对象。这个类存储与套接字相关的地址/端口的元组(本地端口、本地地址、目的地端口、目的地地址)以及一个接收回调。这个接收回调拥有一个由该套接字注册的接收函数。针对 Ipv4EndPointDemux 的 Lookup()函数返回一个 Ipv4EndPoint 对象的列表(可能是一个列表，因为可能有多个套接字与该包匹配)。第四层协议复制该包到每个 Ipv4EndPoint，并调用每个 Ipv4EndPoint 对象自己的 ForwardUp()方法，该函数然后调用由套接字注册的 Receive()函数。

在对真实系统上的套接字 API 进行操作时出现一个问题，即使用某种类型的 I/O 对从套接字的读操作进行管理(比如阻塞、非阻塞、异步，等等)。ns-3 为套接字 I/O 实现了一个异步模型，应用程序设置回调，该回调被用来通知接收到的数据已经可以被读，当数据可获得时，该回调由传输层协议来调用。回调的细节如下：

```
void Socket::SetRecvCallback (Callback,Ptr,const Address&> receivedData);
```

收到的数据是在包的数据缓冲区被运送的。例如，class PacketSink 中的用法如下：

m_socket->SetRecvCallback (MakeCallback(&PacketSink::HandleRead, this));
概括来说，UDP 实现的内部组织如下：

类 UdpImpl 实现 Udp 套接字工厂的功能。
类 UdpL4Protocol 实现套接字无关的协议逻辑。
类 UdpSocketImpl 实现 UDP 的套接字细节相关的方面。
类 Ipv4EndPoint 存储与包相关的地址/端口的元组(本地端口、本地地址、目的地端口、目的地地址)以及一个针对该套接字的接收回调。

9.2.2 Internet 节点接口

很多 Internet 节点对象的实现细节以及内部对象本身是模拟器的公共 API 无法访问的，这考虑到了多种不同的实现，例如，用可移植的 TCP/IP 栈代码替换 ns-3 自带的模型。

所有这些对象的 C++公共 API 在目录 src/node 下，主要包括：

socket.h
tcp.h
udp.h
ipv4.h
这些是典型的基类对象，他们实现了实现中用到的默认值，实现了 get/set 状态变量、主机属性的访问方法，还实现了客户可以访问公共方法，比如 CreateSocket。

9.2.3 包的路径举例

以下两个栈跟踪图显示了包是如何流经 Internet 节点对象的。
(译者注：原图缺)
图 9.2：包的发送路径
(译者注：原图缺)
图 9.3：包的接收路径