ns-3 Rastreamento

Versão ns-3.15

ns-3 project

13/11/2012

Sumário

1 Introdução **3**
- 1.1 Para os usuários do ns-2 . . . 3
- 1.2 Contribuindo . . . 4
- 1.3 Organização do Tutorial . . . 4

2 Recursos **5**
- 2.1 A Internet . . . 5
- 2.2 Mercurial . . . 5
- 2.3 Waf . . . 6
- 2.4 Ambiente de Desenvolvimento . . . 6
- 2.5 Programando com Soquetes (Sockets) . . . 6

3 Iniciando **9**
- 3.1 Baixando o ns-3 . . . 9
- 3.2 Construindo o ns-3 . . . 12
- 3.3 Testando o ns-3 . . . 14
- 3.4 Executando um código (*Script*) . . . 15

4 Visão Conceitual **17**
- 4.1 Principais Abstrações . . . 17
- 4.2 O primeiro código no ns-3 . . . 19
- 4.3 Código fonte do Ns-3 . . . 27

5 Aprofundando Conhecimentos **29**
- 5.1 Usando o Módulo de Registro . . . 29
- 5.2 Usando Argumentos na Linha de Comando . . . 34
- 5.3 Usando o Sistema de Rastreamento . . . 38

6 Construindo topologias **43**
- 6.1 Construindo uma rede em barramento . . . 43
- 6.2 Modelos, atributos e a `sua` realidade . . . 51
- 6.3 Construindo uma rede sem fio . . . 52

7 Rastreamento **61**

7.1 Introdução 61
7.2 Visão Geral 63
7.3 Um Exemplo Real 75
7.4 Usando Classes Assistentes para Rastreamento 90
7.5 Considerações Finais 102

8 Conclusão **103**
8.1 Para o futuro 103
8.2 Finalizando 103

Este é o *Tutorial do ns-3* baseado no *ns-3 Tutorial* (inglês), versão 3.14. A documentação para o projeto ns-3 esta disponível da seguinte forma:

- ns-3 Doxygen: Documentação das APIs públicas do simulador;
- Tutorial *(este documento)*, manual, modelos de bibliotecas para a última release e árvore de desenvolvimento;
- ns-3 wiki.

Este documento é escrito em reStructuredText para Sphinx e é mantido no diretório `doc/tutorial-pt-br` do código fonte do ns-3.

CAPÍTULO 1

Introdução

O *ns-3* é um simulador de redes baseado em eventos discretos desenvolvido especialmente para pesquisa e uso educacional. O projeto ns-3 iniciou em 2006 e tem seu código aberto.

O objetivo deste tutorial é apresentar o *ns-3* de forma estruturada aos usuários iniciantes. Algumas vezes torna-se difícil obter informações básicas de manuais detalhados e converter em informações úteis para as simulações. Neste tutorial são ilustrados vários exemplos de simulações, introduzindo e explicando os principais conceitos necessários ao longo do texto.

A documentação completa do *ns-3* e trechos do código fonte são apresentados para os interessados em aprofundar-se no funcionamento do sistema.

Alguns pontos importantes para se observar:

- O *ns-3* não é uma extensão do ns-2; O *ns-3* é um simulador novo. Ambos são escritos em C++, mas o *ns-3* é totalmente novo e não suporta as APIs da versão anterior. Algumas funcionalidades do ns-2 já foram portadas para o *ns-3*. O projeto continuará mantendo o ns-2 enquanto o *ns-3* estiver em fase de desenvolvimento e formas de integração e transição estão em estudo.
- O *ns-3* é código aberto e existe um grande esforço para manter um ambiente aberto para pesquisadores que queiram contribuir e compartilhar software com o projeto.

1.1 Para os usuários do ns-2

Para aqueles familiarizados com o ns-2 a mudança mais visível é a escolha da linguagem de codificação (*scripting*). O ns-2 utiliza a linguagem OTcl e os resultados das simulações podem ser visualizados utilizando o *Network Animator - nam*. Entretanto, não é possível executar uma simulação escrita inteira em C++ no ns-2 (por exemplo, com um `main()` sem nenhum código OTcl). Assim sendo, no ns-2 alguns componentes são escritos em C++ e outros em OTcl. No *ns-3*, todo o simulador é escrito em C++ com suporte opcional a Python. Desta forma, os códigos de simulação podem ser escritos somente em C++ ou Python. Os resultados de algumas simulações podem ser visualizados pelo *nam*, mas novas formas de visualização estão sendo desenvolvidas. O *ns-3* gera arquivos de rastreamento de pacotes *(packet trace)* no formato *pcap*, assim, é possível utilizar outras ferramentas para a análise de pacotes. Neste tutorial iremos nos concentrar inicialmente nos códigos de simulação escritos em C++ e na interpretação dos pacotes nos arquivos de rastreamento.

Também existem semelhanças entre o ns-2 e o *ns-3*. Ambos, por exemplo, são orientados a objeto e parte do código do ns-2 já foi portado para o *ns-3*. As diferenças entre as versões serão destacadas ao longo deste tutorial.

Uma questão que frequentemente aparece é: "Eu devo continuar usando o ns-2 ou devo migrar para o *ns-3*?". A resposta é: depende. O *ns-3* não tem todos os modelos do ns-2, contudo, possui novas funcionalidades (tais como: trabalha corretamente com nós de rede com múltiplas interfaces de rede (por exemplo, computadores com várias placas de rede), usa endereçamento IP, é mais consistente com arquiteturas e protocolos da Internet, detalha mais o modelo 802.11, etc.). Em todo o caso, os modelos do ns-2 podem ser portados para o *ns-3* (um guia está em desenvolvimento). Atualmente existem várias frentes de trabalho para o desenvolvimento do simulador. Os desenvolvedores acreditam (e os primeiros usuários concordam) que o *ns-3* está pronto para o uso e é uma ótima alternativa para usuários que querem iniciar novos projetos de simulação.

1.2 Contribuindo

O *ns-3* é um simulador para pesquisas e de uso educacional, feito por e para pesquisadores. Este conta com contribuições da comunidade para desenvolver novos modelos, corrigir erros ou manter códigos e compartilhar os resultados. Existem políticas de incentivo para que as pessoas contribuam com o projeto, assim como foi feito no ns-2, tais como:

- Licença de código aberto compatível com GNU GPLv2;
- Wiki;
- Página para contribuição com o código, similar a página de contribuição do ns-2;
- Registro de erros (bugs) aberto.

Se você está lendo este documento, provavelmente contribuir diretamente não seja o foco neste momento, mas esteja ciente que contribuir está no espírito do projeto, mesmo que seja deixando uma mensagem descrevendo suas experiências com o *ns-3* (por exemplo, você pode relatar qual seção deste tutorial não esta clara), reportar a desatualização da documentação, etc. Toda ajuda será muita bem vinda.

1.3 Organização do Tutorial

Este tutorial assume que os novos usuários podem iniciar da seguinte forma:

- Baixar e compilar uma cópia do *ns-3*;
- Executar alguns programas exemplo;
- Analisar as saídas de simulação e ajustá-las.

Assim, tentamos organizar este tutorial nesta sequência.

CAPÍTULO 2

Recursos

2.1 A Internet

Há vários recursos importantes que um usuário do *ns-3* deve conhecer. O principal está em http://www.nsnam.org e fornece acesso a informações básicas sobre o *ns-3*. A documentação detalhada esta disponível no sítio principal através do endereço http://www.nsnam.org/documentation/. Nesta página, também podem ser encontrados documentos relacionados a arquitetura do sistema.

Também existe um *Wiki* que completa o sítio do *ns-3* e pode ser encontrado em http://www.nsnam.org/wiki/. Nesta página são encontradas perguntas freqüentes - FAQs (do inglês, *Frequently Asked Questions*) para usuários e desenvolvedores, guias para resolução de problemas, código de terceiros, artigos, etc.

O código fonte também pode ser encontrado e explorado em http://code.nsnam.org/. Neste encontra-se a árvore de código em desenvolvimento em um repositório chamado `ns-3-dev`. Repositórios antigos e experimentais do núcleo de desenvolvimento podem ser encontrados neste sítio também.

2.2 Mercurial

Sistemas complexos precisam gerenciar a organização e alterações do código, bem como a documentação. Existem várias maneiras de fazer isto e o leitor provavelmente já ouviu falar de algumas. O *Concurrent Version System (CVS)* — em português, Sistema de Versões Concorrentes — é provavelmente o mais conhecido.

O *ns-3* utiliza o Mercurial para isto. Embora não seja necessário conhecer muito sobre o Mercurial para entender este tutorial, recomenda-se a familiarização com o uso da ferramenta para acessar o código fonte do sistema. O Mercurial tem um sítio em http://www.selenic.com/mercurial/, no qual pode-se baixar diretamente os executáveis ou o código fonte deste sistema de *Software Configuration Management (SCM)* — em português, Software de Gerenciamento de Configuração. Selenic (o desenvolvedor do Mercurial), também fornece tutoriais em http://www.selenic.com/mercurial/wiki/index.cgi/Tutorial/, e um guia rápido em http://www.selenic.com/mercurial/wiki/index.cgi/QuickStart/.

Informações vitais de como usar o Mercurial e o *ns-3* são encontradas no sítio principal do projeto.

2.3 Waf

Uma vez baixado o código fonte para o seu sistema de arquivos local, será necessário compilar estes fontes para criar os executáveis. Para esta tarefa existem várias ferramentas disponíveis. Provavelmente a mais conhecida é o Make. Além de mais conhecido, também deve ser o mais difícil de usar em grandes sistemas e com muitas opções de configuração. Por este motivo, muitas alternativas foram desenvolvidas, utilizando principalmente a linguagem Python.

O Waf é utilizado para gerar os binários no projeto *ns-3*. Ele faz parte da nova geração de sistemas de compilação e contrução baseados em Python. O leitor não precisa entender nada de Python para compilar o *ns-3*, e terá que entender um pequeno e intuitivo subconjunto da linguagem se quiser estender o sistema.

Para os interessados em mais detalhes sobre o Waf, basta acessar o sítio http://code.google.com/p/waf/.

2.4 Ambiente de Desenvolvimento

Como mencionado anteriormente, a programação no *ns-3* é feita em C++ ou Python. A partir do ns-3.2, a maioria das APIs já estão disponíveis em Python, mas os modelos continuam sendo escritos em C++. Considera-se que o leitor possui conhecimento básico de C++ e conceitos de orientação a objetos neste documento. Somente serão revistos conceitos avançados, possíveis características pouco utilizadas da linguagem, dialetos e padrões de desenvolvimento. O objetivo não é tornar este um tutorial de C++, embora seja necessário saber o básico da linguagem. Para isto, existe um número muito grande de fontes de informação na Web e em materiais impressos (livros, tutoriais, revistas, etc).

Se você é inexperiente em C++, pode encontrar tutoriais, livros e sítios Web para obter o mínimo de conhecimento sobre a linguagem antes de continuar. Por exemplo, pode utilizar este tutorial.

O *ns-3* utiliza vários componentes do conjunto de ferramentas GNU — *"GNU toolchain"* — para o desenvolvimento. Um *software toolchain* é um conjunto de ferramentas de programação para um determinado ambiente. Para uma breve visão do que consiste o *GNU toolchain* veja http://en.wikipedia.org/wiki/GNU_toolchain. O *ns-3* usa o *gcc*, *GNU binutils* e *gdb*. Porém, não usa as ferramentas GNU para compilar o sistema, nem o Make e nem o Autotools. Para estas funções é utilizado o Waf.

Normalmente um usuário do *ns-3* irá trabalhar no Linux ou um ambiente baseado nele. Para aqueles que usam Windows, existem ambientes que simulam o Linux em vários níveis. Para estes usuários, o projeto *ns-3* fornece suporte ao ambiente Cygwin. Veja o sítio http://www.cygwin.com/ para detalhes de como baixá-lo (o MinGW não é suportado oficialmente, embora alguns mantenedores do projeto trabalhem com ele). O Cygwin fornece vários comandos populares do Linux, entretanto podemos ter problemas com a emulação, às vezes a interação com outros programas do Windows pode causar problemas.

Se você usa o Cygwin ou MinGW e usa produtos da Logitech, evite dores de cabeça e dê uma olhada em MinGW FAQ.

Busque por "Logitech" e leia a entrada com o assunto: "why does make often crash creating a sh.exe.stackdump file when I try to compile my source code.". Acredite ou não, o `Logitech Process Monitor` influencia todas as DLLs do sistema. Isto pode ocasionar problemas misteriosos durante a execução do Cygwin ou do MinGW. Muita cautela quando utilizar software da Logitech junto com o Cygwin.

Uma alternativa ao Cygwin é instalar um ambiente de máquina virtual, tal como o VMware server e criar uma máquina virtual Linux.

2.5 Programando com Soquetes (Sockets)

Neste tutorial assume-se, nos exemplos utilizados, que o leitor está familiarizado com as funcionalidades básicas da API dos soquetes de Berkeley. Se este não for o caso, recomendamos a leitura das APIs e alguns casos de uso comuns.

Uma API — do Inglês, *Application Programming Interface* — é um é um conjunto de rotinas e padrões estabelecidos por um software para a utilização das suas funcionalidades. Para uma boa visão geral sobre a programação de soquetes TCP/IP sugerimos TCP/IP Sockets in C, Donahoo and Calvert.

O sítio http://cs.baylor.edu/~donahoo/practical/CSockets/ contém os códigos fontes dos exemplos do livro.

Se o leitor entender os primeiros quatro capítulos do livro (ou para aqueles que não têm acesso ao livro, os exemplos de cliente e servidor de eco mostrado no sítio anterior) estará apto para compreender o tutorial. Existe também um livro sobre soquetes multidifusão, Multicast Sockets, Makofske and Almeroth. que é um material que cobre o necessário sobre multidifusão caso o leitor se interesse.

CAPÍTULO 3

Iniciando

3.1 Baixando o ns-3

O *ns-3*, como um todo, é bastante complexo e possui várias dependências. Isto também é verdade para as ferramentas que fornecem suporte ao *ns-3* (exemplos, *"GNU toolchain"*, Mercurial e um editor para a programação), desta forma é necessário assegurar que várias bibliotecas estejam presentes no sistema. O *ns-3* fornece um Wiki com várias dicas sobre o sistema. Uma das páginas do Wiki é a página de instalação (*"Installation"*) que está disponível em: http://www.nsnam.org/wiki/index.php/Installation.

A seção de pré-requisitos (*"Prerequisites"*) do Wiki explica quais pacotes são necessários para a instalação básica do *ns-3* e também fornece os comandos usados para a instalação nas variantes mais comuns do Linux. Os usuários do Cygwin devem utilizar o `Cygwin installer`.

Seria interessante explorar um pouco o Wiki, pois lá existe uma grande variedade de informações.

A partir deste ponto considera-se que o leitor está trabalhando com Linux ou em um ambiente que o emule (Linux, Cygwin, etc), que tenha o *"GNU toolchain"* instalado, bem como os pré-requisitos mencionados anteriormente. Também assume-se que o leitor tenha o Mercurial e o Waf instalados e funcionando em seu sistema.

Os códigos fonte do *ns-3* estão disponíveis através dos repositórios do Mercurial no servidor http://code.nsnam.org. Os fontes compactados podem ser obtidos em http://www.nsnam.org/releases/. No final desta seção há instruções de como obter uma versão compactada. No entanto, a não ser que se tenha uma boa razão, recomenda-se o uso do Mercurial para acesso ao código.

A maneira mais simples de iniciar com o Mercurial é usando o ambiente `ns-3-allinone`. Trata-se de um conjunto de *scripts* que gerencia o baixar e o construir de vários sub-sistemas do *ns-3*. Recomenda-se que os pouco experientes iniciem sua aventura neste ambiente, pois irá realmente facilitar a jornada.

3.1.1 Obtendo o ns-3 usando o Mercurial

Para iniciar, uma boa prática é criar um diretório chamado `repos` no diretório *home* sobre o qual será mantido o repositório local do Mercurial. *Dica: iremos assumir que o leitor fez isto no restante deste tutorial, então é bom executar este passo!* Se o leitor adotar esta abordagem é possível obter a cópia do `ns-3-allinone` executando os comandos a seguir no *shell* Linux (assumindo que o Mercurial está instalado):

```
cd
mkdir repos
cd repos
hg clone http://code.nsnam.org/ns-3-allinone
```

Quando executarmos o comando *hg* (Mercurial), teremos como saída algo como:

```
destination directory: ns-3-allinone
requesting all changes
adding changesets
adding manifests
adding file changes
added 31 changesets with 45 changes to 7 files
7 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Depois que o comando *clone* for executado com sucesso, teremos um diretório chamado `ns-3-allinone` dentro do diretório `~/repos`. O conteúdo deste diretório deve ser algo como:

```
build.py*  constants.py  dist.py*  download.py*  README  util.py
```

Até agora foram baixados alguns *scripts* em Python. O próximo passo será usar estes *scripts* para baixar e construir a distribuição *ns-3* de sua escolha.

Acessando o endereço: http://code.nsnam.org/, observa-se vários repositórios. Alguns são privados à equipe de desenvolvimento do *ns-3*. Os repositórios de interesse ao leitor estarão prefixados com "ns-3". As *releases* oficiais do *ns-3* estarão enumeradas da seguinte forma: `ns-3.<release>.<hotfix>`. Por exemplo, uma segunda atualização de pequeno porte (*hotfix*) de uma hipotética *release* 42, seria enumerada da seguinte maneira: `ns-3.42.2`.

A versão em desenvolvimento (que ainda não é uma *release* oficial) pode ser encontrada em http://code.nsnam.org/ns-3-dev/. Os desenvolvedores tentam manter este repositório em um estado consistente, mas podem existir códigos instáveis. Recomenda-se o uso de uma *release* oficial, a não ser que se necessite das novas funcionalidades introduzidas.

Uma vez que o número das versões fica mudando constantemente, neste tutorial será utilizada a versão ns-3-dev, mas o leitor pode escolher outra (por exemplo, ns-3.10). A última versão pode ser encontrada inspecionando a lista de repositórios ou acessando a página "ns-3 Releases" e clicando em *latest release*.

Entre no diretório `ns-3-allinone` criado anteriormente. O arquivo `download.py` será usado para baixar as várias partes do *ns-3* que serão utilizadas.

Execute os seguintes comandos no *shell* (lembre-se de substituir o número da versão no lugar de `ns-3-dev` pela que escolheu, por exemplo, se você optou por usar a décima *release* estável, então deve usar o nome `"ns-3.10"`).

```
./download.py -n ns-3-dev
```

O `ns-3-dev` é o padrão quando usamos a opção `-n`, assim o comando poderia ser `./download.py -n`. O exemplo redundante é apenas para ilustra como especificar repositórios alternativos. Um comando mais simples para obter o `ns-3-dev` seria:

```
./download.py
```

Com o comando *hg* (Mercurial) em execução devemos ver a seguinte saída:

```
    #
    # Get NS-3
    #

Cloning ns-3 branch
 =>  hg clone http://code.nsnam.org/ns-3-dev ns-3-dev
requesting all changes
adding changesets
adding manifests
adding file changes
added 4634 changesets with 16500 changes to 1762 files
870 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Esta é a saída do *script* de download obtendo o código atual do repositório `ns-3`.

O *script* de download reconhece que partes do ns-3 não são suportadas na plataforma. Dependendo do sistema, pode ser que a saída não seja exatamente como a mostrada a seguir. Porém, a maioria dos sistemas apresentarão algo como:

```
    #
    # Get PyBindGen
    #

Required pybindgen version:  0.10.0.640
Trying to fetch pybindgen; this will fail if no network connection is available.
Hit Ctrl-C to skip.
 =>  bzr checkout -rrevno:640 https://launchpad.net/pybindgen pybindgen
Fetch was successful.
```

Este é o *script* de download obtendo um gerador de *bindings* Python — um *binding* é literalmente a ligação ou ponte entre dois sistemas, chamaremos aqui de extensões Python. Também será necessário o Bazaar (brz) para baixar o PyBindGen. O Bazaar é um sistema de controle de versões. Em seguida, o leitor deve ver (com algumas variações devido as plataformas) algo parecido com as seguintes linhas:

```
    #
    # Get NSC
    #

Required NSC version:  nsc-0.5.0
Retrieving nsc from https://secure.wand.net.nz/mercurial/nsc
 =>  hg clone https://secure.wand.net.nz/mercurial/nsc nsc
requesting all changes
adding changesets
adding manifests
adding file changes
added 273 changesets with 17565 changes to 15175 files
10622 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Neste momento, o *script* de download baixa o *Network Simulation Cradle* - NSC. Note que o NSC não é suportado no OSX ou Cygwin e trabalha melhor com o gcc-3.4, gcc-4.2 ou superiores.

Depois que o *script* `download.py` tiver completado sua tarefa, veremos vários diretórios novos dentro de `~/repos/ns-3-allinone`:

```
build.py*     constants.pyc  download.py*  nsc/        README      util.pyc
constants.py  dist.py*       ns-3-dev/     pybindgen/  util.py
```

Por fim, no diretório `ns-3-dev` que está dentro do diretório `~/repos/ns-3-allinone` deve existir, depois dos passos anteriores, o seguinte conteúdo:

```
AUTHORS       doc       ns3            scratch   testpy.supp  VERSION   waf-tools
bindings      examples  README         src       utils        waf*      wscript
CHANGES.html  LICENSE   RELEASE_NOTES  test.py*  utils.py     waf.bat*  wutils.py
```

Agora está tudo pronto para a construção da distribuição do *ns-3*.

3.1.2 Obtendo o ns-3 compactado (*Tarball*)

O processo de download do *ns-3* compactado é mais simples do que o processo usando o Mercurial, porque tudo que precisamos já vem empacotado. Basta escolher a versão, baixá-la e extraí-la.

Como mencionado anteriormente, uma boa prática é criar um diretório chamado `repos` no diretório *home* para manter a cópia local dos repositórios do Mercurial. Da mesma forma, pode-se manter também um diretório chamado

`tarballs` para manter as versões obtidas via arquivo compactado. *Dica: o tutorial irá assumir que o download foi feito dentro do diretório ‘‘repos‘‘.* Se a opção for pelo método do arquivo compactado, pode-se obter a cópia de uma versão digitando os seguintes comandos no *shell* Linux (obviamente, substitua os números de versões do comando para o valor apropriado):

```
cd
mkdir tarballs
cd tarballs
wget http://www.nsnam.org/releases/ns-allinone-3.10.tar.bz2
tar xjf ns-allinone-3.10.tar.bz2
```

Dentro do diretório `ns-allinone-3.10` extraído, deverá haver algo como:

```
build.py      ns-3.10/      pybindgen-0.15.0/    util.py
constants.py  nsc-0.5.2/    README
```

Agora está tudo pronto para a construção da distribuição do *ns-3*.

3.2 Construindo o ns-3

3.2.1 Construindo com o `build.py`

A primeira construção do *ns-3* deve ser feita usando o ambiente `allinone`. Isto fará com que o projeto seja configurado da maneira mais funcional.

Entre no diretório criado na seção “Obtendo o ns-3”. Se o Mercurial foi utilizado, então haverá um diretório chamado `ns-3-allinone` localizado dentro de `~/repos`. Se foi utilizado o arquivo compactado, haverá um diretório chamado `ns-allinone-3.10` dentro do diretório `~/tarballs`. Lembre-se de adequar os nomes conforme os arquivos obtidos e diretórios criados. Agora, digite o seguinte comando:

```
./build.py --enable-examples --enable-tests
```

Foram utilizadas as opções `--enable-examples` e `--enable-tests` pois o tutorial irá trabalhar com exemplos e testes, e, por padrão, eles não são construídos. Futuramente, o leitor poderá construir sem estas opções.

Serão exibidas muitas saídas típicas de um compilador conforme o código é construído. Finalmente, no final do processo, deverá aparecer uma saída como esta:

```
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (2m30.586s)

Modules built:
aodv                      applications              bridge
click                     config-store              core
csma                      csma-layout               dsdv
emu                       energy                    flow-monitor
internet                  lte                       mesh
mobility                  mpi                       netanim
network                   nix-vector-routing        ns3tcp
ns3wifi                   olsr                      openflow
point-to-point            point-to-point-layout     propagation
spectrum                  stats                     tap-bridge
template                  test                      tools
topology-read             uan                       virtual-net-device
visualizer                wifi                      wimax
```

Uma vez que o projeto foi construído, pode-se deixar de lado os *scripts* `ns-3-allinone`. O leitor já obteve o que precisava e agora irá interagir diretamente com o Waf no diretório `ns-3-dev`. Mude para o diretório `ns-3-dev` (ou para o diretório apropriado de sua versão).

```
cd ns-3-dev
```

3.2.2 Construindo com o Waf

O Waf é utilizado para configurar e construir o projeto do *ns-3*. Não é estritamente necessário neste ponto, mas será valioso quando se forem necessárias alterações nas configurações do projeto. Provavelmente a mudança mais útil que será feita futuramente é a construção de uma versão do código otimizado. Por padrão, o projeto é construído com a opção de depuração (*debug*), para verificação de erros. Então, para construir um projeto otimizado, deve-se executar o seguinte comando (ainda com suporte a testes e exemplos):

```
./waf -d optimized --enable-examples --enable-tests configure
```

Isto executa o Waf fora do diretório local (o que é bem conveniente). Como o sistema em construção verifica várias dependências, deverá aparecer uma saída similar com a que segue:

```
Checking for program g++                 : ok /usr/bin/g++
Checking for program cpp                 : ok /usr/bin/cpp
Checking for program ar                  : ok /usr/bin/ar
Checking for program ranlib              : ok /usr/bin/ranlib
Checking for g++                         : ok
Checking for program pkg-config          : ok /usr/bin/pkg-config
Checking for -Wno-error=deprecated-declarations support : yes
Checking for -Wl,--soname=foo support                   : yes
Checking for header stdlib.h                            : ok
Checking for header signal.h                            : ok
Checking for header pthread.h                           : ok
Checking for high precision time implementation         : 128-bit integer
Checking for header stdint.h                            : ok
Checking for header inttypes.h                          : ok
Checking for header sys/inttypes.h                      : not found
Checking for library rt                                 : ok
Checking for header netpacket/packet.h                  : ok
Checking for pkg-config flags for GSL                   : ok
Checking for header linux/if_tun.h                      : ok
Checking for pkg-config flags for GTK_CONFIG_STORE      : ok
Checking for pkg-config flags for LIBXML2               : ok
Checking for library sqlite3                            : ok
Checking for NSC location                               : ok ../nsc (guessed)
Checking for library dl                                 : ok
Checking for NSC supported architecture x86_64          : ok
Checking for program python                             : ok /usr/bin/python
Checking for Python version >= 2.3                      : ok 2.5.2
Checking for library python2.5                          : ok
Checking for program python2.5-config                   : ok /usr/bin/python2.5-config
Checking for header Python.h                            : ok
Checking for -fvisibility=hidden support                : yes
Checking for pybindgen location                         : ok ../pybindgen (guessed)
Checking for Python module pybindgen                    : ok
Checking for pybindgen version                          : ok 0.10.0.640
Checking for Python module pygccxml                     : ok
Checking for pygccxml version                           : ok 0.9.5
Checking for program gccxml                             : ok /usr/local/bin/gccxml
Checking for gccxml version                             : ok 0.9.0
```

```
Checking for program sudo                                 : ok /usr/bin/sudo
Checking for program hg                                   : ok /usr/bin/hg
Checking for program valgrind                             : ok /usr/bin/valgrind
---- Summary of optional NS-3 features:
Threading Primitives          : enabled
Real Time Simulator           : enabled
Emulated Net Device           : enabled
GNU Scientific Library (GSL)  : enabled
Tap Bridge                    : enabled
GtkConfigStore                : enabled
XmlIo                         : enabled
SQlite stats data output      : enabled
Network Simulation Cradle     : enabled
Python Bindings               : enabled
Python API Scanning Support   : enabled
Use sudo to set suid bit      : not enabled (option --enable-sudo not selected)
Build tests                   : enabled
Build examples                : enabled
Static build                  : not enabled (option --enable-static not selected)
'configure' finished successfully (2.870s)
```

Repare a última parte da saída. Algumas opções do ns-3 não estão habilitadas por padrão ou necessitam de algum suporte do sistema para funcionar corretamente. Por exemplo, para habilitar XmlTo, a biblioteca libxml-2.0 deve estar presente no sistema. Se a biblioteca não estiver instalada esta funcionalidade não é habilitada no *ns-3* e uma mensagem é apresentada. Note também que existe uma funcionalidade que utiliza o Sudo para configurar o *suid* de certos programas. Isto não está habilitado por padrão, então esta funcionalidade é reportada como não habilitada (`not enabled`).

Vamos configurar uma construção do *ns-3* com suporte a depuração, bem como, vamos incluir exemplos e testes. Para isto devemos executar:

```
./waf -d debug --enable-examples --enable-tests configure
```

Pronto o sistema está configurado, agora podemos construir nossa versão digitando:

```
./waf
```

Alguns comandos do Waf são válidos apenas na fase de construção e outros são válidos na fase de configuração do sistema. Por exemplo, se o leitor espera usar características de emulação do *ns-3*, deve habilitar o *suid* usando o Sudo como descrito anteriormente, isto na fase de configuração. O comando utilizado, incluindo exemplos e testes, será:

```
./waf -d debug --enable-sudo --enable-examples --enable-tests configure
```

Com esta configuração, o Waf executará o Sudo para alterar programas que criam soquetes para executar o código de emulação como *root*. Existem várias outras opções de configuração e construção disponíveis no Waf. Para explorar estas opções, digite:

```
./waf --help
```

Alguns comandos de teste serão utilizados na próxima seção.

Como pôde ser notado, a construção do *ns-3* foi feita duas vezes. Isto para que o leitor saiba como trocar a configuração para construir código otimizado no futuro.

3.3 Testando o ns-3

Para executar os testes de unidade do *ns-3*, basta chamar o arquivo `./test.py` da seguinte forma:

```
./test.py -c core
```

Estes testes são executados em paralelo pelo Waf. No final, uma mensagem como a que segue deve aparecer.

```
47 of 47 tests passed (47 passed, 0 failed, 0 crashed, 0 valgrind errors)
```

Esta é uma mensagem importante.

Também haverá saídas da execução do teste e estas geralmente são algo como:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (1.799s)

Modules built:
aodv                      applications              bridge
click                     config-store              core
csma                      csma-layout               dsdv
emu                       energy                    flow-monitor
internet                  lte                       mesh
mobility                  mpi                       netanim
network                   nix-vector-routing        ns3tcp
ns3wifi                   olsr                      openflow
point-to-point            point-to-point-layout     propagation
spectrum                  stats                     tap-bridge
template                  test                      tools
topology-read             uan                       virtual-net-device
visualizer                wifi                      wimax

PASS: TestSuite ns3-wifi-interference
PASS: TestSuite histogram
PASS: TestSuite sample
PASS: TestSuite ipv4-address-helper
PASS: TestSuite devices-wifi
PASS: TestSuite propagation-loss-model

...

PASS: TestSuite attributes
PASS: TestSuite config
PASS: TestSuite global-value
PASS: TestSuite command-line
PASS: TestSuite basic-random-number
PASS: TestSuite object
PASS: TestSuite random-number-generators
95 of 95 tests passed (95 passed, 0 failed, 0 crashed, 0 valgrind errors)
```

Este comando é normalmente executado pelos usuários para verificar se o *ns-3* foi construído corretamente.

3.4 Executando um código (*Script*)

Os códigos normalmente são executados sob o controle do Waf. Isto assegura que os caminhos das bibliotecas compartilhadas estejam corretas e que estarão disponíveis em tempo de execução. Para executar um programa, basta usar a opção `--run` no Waf. Para executar um equivalente ao "Olá mundo" (*Hello world*) no *ns-3*, utilizamos o comando:

```
./waf --run hello-simulator
```

O Waf primeiro verifica se o programa foi construído corretamente e se necessário, o constrói. Então executa o programa, que fornece a seguinte saída:

```
Hello Simulator
```

Parabéns. Você agora é um usuário ns-3

O que fazer se o comando não gerar uma saída?

Se a mensagem `Hello Simulator` não aparece, mas o Waf gera saídas indicando que a construção do sistema foi executada com sucesso, é possível que o leitor tenha trocado o sistema para o modo otimizado na seção *Construindo com o Waf*, e tenha esquecido de voltar ao modo de depuração. Todas as saídas utilizadas neste tutorial são feitas com um componente especial de registro (*logging*) do *ns-3* que é muito útil para mostrar mensagens na tela. As saídas deste componente são automaticamente desabilitadas quando o código é contruído na forma otimizada. Para produzir as saídas, digite o seguinte comando,

```
./waf -d debug --enable-examples --enable-tests configure
```

para dizer ao Waf para construir o *ns-3* com a versão de depuração e incluir exemplos e testes. Ainda é necessário digitar o seguinte comando para a construção:

```
./waf
```

Agora, ao executar o programa `hello-simulator` devemos ter a seguinte a saída.

Se o leitor for executar seus programas sob outras ferramentas, tais como Gdb ou Valgrind, é recomendável que leia a seguinte entrada no Wiki.

CAPÍTULO 4

Visão Conceitual

Antes de escrever códigos no *ns-3* é extremamente importante entender um pouco dos conceitos e abstrações do sistema. Muitos conceitos poderão parecer óbvios, mas a recomendação geral é que esta seção seja lida por completo para assegurar que o leitor inicie com uma base sólida.

4.1 Principais Abstrações

Nesta seção, são revistos alguns termos que são comumente usados por profissionais de redes de computadores, mas que tem um significado específico no *ns-3*.

4.1.1 Nó (*Node*)

No jargão da Internet, um dispositivo computacional que conecta-se a uma rede é chamado de *host* ou em alguns casos de *terminal*. Devido ao fato do *ns-3* ser um simulador de *rede*, e não um simulador da *Internet*, o termo *host* é intencionalmente não utilizado, pois está intimamente associado com a Internet e seus protocolos. Ao invés disso, é utilizado o termo *node* — em português, *nó* — que é um termo mais genérico e também usado por outros simuladores que tem suas origens na Teoria dos Grafos.

A abstração de um dispositivo computacional básico é chamado então de nó. Essa abstração é representada em C++ pela classe `Node`. Esta classe fornece métodos para gerenciar as representações de dispositivos computacionais nas simulações.

O nó deve ser pensado como um computador no qual se adicionam funcionalidades, tal como aplicativos, pilhas de protocolos e periféricos com seus *drivers* associados que permitem ao computador executar tarefas úteis. No *ns-3* é utilizado este mesmo conceito básico.

4.1.2 Aplicações (*Application*)

Normalmente, programas de computador são divididos em duas classes. *Programas de sistema* organizam recursos do computador, tais como: memória, processador, disco, rede, etc., de acordo com algum modelo computacional. Tais programas normalmente não são utilizados diretamente pelos usuários. Na maioria das vezes, os usuários fazem uso de *aplicações*, que usam os recursos controlados pelos programas de sistema para atingir seus objetivos.

Geralmente, a separação entre programas de sistema e aplicações de usuários é feita pela mudança no nível de privilégios que acontece na troca de contexto feita pelo sistema operacional. No *ns-3*, não existe um conceito de sistema

operacional real, não há o conceito de níveis de privilégios nem chamadas de sistema. Há apenas aplicações que são executadas nos nós para uma determinada simulação.

No *ns-3*, a abstração básica para um programa de usuário que gera alguma atividade a ser simulada é a aplicação. Esta abstração é representada em C++ pela classe `Application`, que fornece métodos para gerenciar a representação de suas versões de aplicações a serem simuladas. Os desenvolvedores devem especializar a classe `Application` para criar novas aplicações. Neste tutorial serão utilizadas duas especializações da classe `Application`, chamadas `UdpEchoClientApplication` e `UdpEchoServerApplication`. Estas aplicações compõem um modelo cliente/servidor usado para gerar pacotes simulados de eco na rede.

4.1.3 Canal de Comunicação (*Channel*)

No mundo real, computadores estão conectados em uma rede. Normalmente, o meio sobre o qual os dados trafegam é chamada de canal (*channel*). Quando um cabo Ethernet é ligado ao conector na parede, na verdade está se conectando a um canal de comunicação Ethernet. No mundo simulado do *ns-3*, um nó é conectado a um objeto que representa um canal de comunicação. A abstração de canal de comunicação é representada em C++ pela classe `Channel`.

A classe `Channel` fornece métodos para gerenciar objetos de comunicação de sub-redes e nós conectados a eles. Os `Channels` também podem ser especializados por desenvolvedores (no sentido de programação orientada a objetos). Uma especialização de `Channel` pode ser algo como um simples fio. Pode também ser algo mais complexo, como um comutador Ethernet ou ainda ser uma rede sem fio (*wireless*) em um espaço tridimensional com obstáculos.

Neste tutorial, são utilizadas versões especializadas de `Channel` chamadas `CsmaChannel`, `PointToPointChannel` e `WifiChannel`. O `CsmaChannel`, por exemplo, é uma versão do modelo de rede que implementa controle de acesso ao meio CSMA (*Carrier Sense Multiple Access*). Ele fornece uma funcionalidade similar a uma rede Ethernet.

4.1.4 Dispositivos de Rede (*Net Device*)

No passado, para conectar computadores em uma rede, era necessário comprar o cabo específico e um dispositivo chamado (na terminologia dos PCs) de *periférico*, que precisava ser instalado no computador. Se a placa implementava funções de rede, era chamada de interface de rede (*Network Interface Card* - NIC). Atualmente, a maioria dos computadores vêm com a placa de rede integrada à placa mãe (*on-board*) e os usuários não vêem o computador como uma junção de partes.

Uma placa de rede não funciona sem o *driver* que a controle. No Unix (ou Linux), um periférico (como a placa de rede) é classificado como um dispositivo (*device*). Dispositivos são controlados usando drivers de dispositivo (*device drivers*) e as placas de rede são controladas através de drivers de dispositivo de rede (*network device drivers*), também chamadas de dispositivos de rede (*net devices*). No Unix e Linux estes dispositivos de rede levam nomes como *eth0*.

No *ns-3* a abstração do *dispositivo de rede* cobre tanto o hardware quanto o software (*drive*). Um dispositivo de rede é "instalado" em um nó para permitir que este se comunique com outros na simulação, usando os canais de comunicação (*channels*). Assim como em um computador real, um nó pode ser conectado a mais que um canal via múltiplos dispositivos de rede.

A abstração do dispositivo de rede é representado em C++ pela classe `NetDevice`, que fornece métodos para gerenciar conexões para objetos `Node` e `Channel`. Os dispositivos, assim como os canais e as aplicações, também podem ser especializados. Várias versões do `NetDevice` são utilizadas neste tutorial, tais como: `CsmaNetDevice`, `PointToPointNetDevice` e `WifiNetDevice`. Assim como uma placa de rede Ethernet é projetada para trabalhar em uma rede Ethernet, um `CsmaNetDevice` é projetado para trabalhar com um `CsmaChannel`. O `PointToPointNetDevice` deve trabalhar com um `PointToPointChannel` e o `WifiNetNevice` com um `WifiChannel`.

4.1.5 Assistentes de Topologia (*Topology Helpers*)

Em redes reais, os computadores possuem placas de rede, sejam elas integradas ou não. No *ns-3*, teremos nós com dispositivos de rede. Em grandes simulações, será necessário arranjar muitas conexões entre nós, dispositivos e canais de comunicação.

Visto que conectar dispositivos a nós e a canais, atribuir endereços IP, etc., são todas tarefas rotineiras no *ns-3*, são fornecidos os Assistentes de Topologia (*topology helpers*). Por exemplo, podem ser necessárias muitas operações distintas para criar um dispositivo, atribuir um endereço MAC a ele, instalar o dispositivo em um nó, configurar a pilha de protocolos no nó em questão e por fim, conectar o dispositivo ao canal. Ainda mais operações podem ser necessárias para conectar múltiplos dispositivos em canais multiponto e então fazer a interconexão das várias redes. Para facilitar o trabalho, são disponibilizados objetos que são Assistentes de Topologia, que combinam estas operações distintas em um modelo fácil e conveniente.

4.2 O primeiro código no ns-3

Se o sistema foi baixado como sugerido, uma versão do *ns-3* estará em um diretório chamado `repos` dentro do diretório *home*. Entrando no diretório dessa versão, deverá haver uma estrutura parecida com a seguinte:

```
AUTHORS       examples       scratch        utils      waf.bat*
bindings      LICENSE        src            utils.py   waf-tools
build         ns3            test.py*       utils.pyc  wscript
CHANGES.html  README         testpy-output  VERSION    wutils.py
doc           RELEASE_NOTES  testpy.supp    waf*       wutils.pyc
```

Entrando no diretório `examples/tutorial`, vai haver um arquivo chamado `first.cc`. Este é um código que criará uma conexão ponto-a-ponto entre dois nós e enviará um pacote de eco entre eles. O arquivo será analisado linha a linha, para isto, o leitor pode abri-lo em seu editor de textos favorito.

4.2.1 Padronização

A primeira linha do arquivo é uma linha de modo emacs, que informa sobre a convenção de formatação (estilo de codificação) que será usada no código fonte.

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
```

Este é sempre um assunto um tanto quanto controverso. O projeto *ns-3*, tal como a maioria dos projetos de grande porte, adotou um estilo de codificação, para o qual todo o código deve conformar. Se o leitor quiser contribuir com o projeto, deverá estar em conformidade com a codificação que está descrita no arquivo `doc/codingstd.txt` ou no sítio.

A equipe do *ns-3* recomenda aos novos usuários que adotem o padrão quando estiverem tralhando com o código. Tanto eles, quanto todos os que contribuem, tiveram que fazer isto em algum momento. Para aqueles que utilizam o editor Emacs, a linha de modo torna mais facil seguir o padrão corretamente.

O *ns-3* é licenciado usando a *GNU General Public License* - GPL. No cabeçalho de todo aquivo da distribuição há as questões legais associadas à licença. Frequentemente, haverá também informações sobre os direitos de cópia de uma das instituições envolvidas no projeto e o autor do arquivo.

```
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307 USA
 */
```

4.2.2 Inclusão de Módulos

O código realmente começa pelo carregamento de módulos através da inclusão dos arquivos:

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
```

Os arquivos a serem incluídos são agrupados em módulos relativamente grandes, de forma a ajudar os usuários. Também é possível fazer referência a um único arquivo que irá recursivamente carregar todas as bibliotecas de cada módulo. Ao invés de procurar pelo arquivo exato, e provavelmente, ter que resolver dependências, é possível carregar um grupo de arquivos de uma vez. Esta com certeza não é a abordagem mais eficiente, mas permite escrever códigos de forma bem mais fácil.

Durante a construção, cada um dos arquivos incluídos é copiado para o diretório chamado `ns3` (dentro do diretório `build`), o que ajuda a evitar conflito entre os nomes dos arquivos de bibliotecas. O arquivo `ns3/core-module.h`, por exemplo, corresponde ao módulo que está no diretório `src/core`. Há um grande número de arquivos neste diretório. No momento em que o Waf está construindo o projeto, copia os arquivos para o diretório `ns3`, no subdiretório apropriado — `build/debug` ou `build/optimized` — dependendo da configuração utilizada.

Considerando que o leitor esteja seguindo este tutorial, já terá feito:

```
./waf -d debug --enable-examples --enable-tests configure
```

Também já terá feito

```
./waf
```

para construir o projeto. Então, no diretório `../../build/debug/ns3` deverá haver os quatro módulos incluídos anteriormente. Olhando para o conteúdo destes arquivos, é possível observar que eles incluem todos os arquivos públicos dos seus respectivos módulos.

4.2.3 Ns3 *Namespace*

A próxima linha no código `first.cc` é a declaração do *namespace*.

```
using namespace ns3;
```

O projeto *ns-3* é implementado em um *namespace* chamado *ns3*. Isto agrupa todas as declarações relacionadas ao projeto em um escopo fora do global, que ajuda na integração com outros códigos. A declaração `using` do C++ insere o *namespace ns-3* no escopo global, evitando que se tenha que ficar digitando `ns3::` antes dos códigos *ns-3*. Se o leitor não esta familiarizado com *namesapaces*, consulte algum tutorial de C++ e compare o *namespace* `ns3` e o *namespace* `std`, usado com frequência em C++, principalmente com `cout` e `streams`.

4.2.4 Registro (*Logging*)

A próxima linha do código é o seguinte,

```
NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");
```

Nós iremos utilizar esta declaração em um lugar conveniente para conversar com o sistema de documentação Doxygen. Se você procurar no *web site* do projeto ns-3, você encontrará um *link* para a documentação (*Documentation*) na barra de navegação. Se selecionarmos este *link*, veremos a página de documentação. Lá tem um *link* para as últimas *releases* (*Latest Release*) que irão apresentar a documentação da última *release* do *ns-3*. Se você também pode selecionar o *link* para a documentação das APIs (*API Documentation*).

Do lado esquerdo, você achará uma representação gráfica da estrutura da documentação. Um bom lugar para começar é com o livro de módulos do NS-3 (`NS-3 Modules`) na árvore de navegação, você pode expandir para ver a lista de documentação de módulos do *ns-3*. O conceito de módulo aqui está diretamente ligado com a inclusão de bibliotecas apresentadas anteriormente. O sistema de registro (*logging*) é discutido na seção `C++ Constructs Used by All Modules`, vá em frente e expanda a documentação. Agora expanda o livro da depuração (`Debugging`) e selecione a página de `Logging`.

Agora você deve procurar na documentação Doxygen pelo módulo de `Logging`. Na lista de `#define` bem no topo da página você verá uma entrada para `NS_LOG_COMPONENT_DEFINE`. Antes de ir para lá, dê uma boa olhada na descrição detalhada (*Detailed Description*) do módulo de *logging*.

Uma vez que você tem uma ideia geral, prossiga e olhe a documentação de `NS_LOG_COMPONENT_DEFINE`. Não esperamos duplicar a documentação, mas para resumir esta linha declara o componente de *logging* chamado `FirstScriptExample` que permite habilitar e desabilitar mensagens de *logging* referenciando pelo nome.

4.2.5 Função Principal

As próximas linhas do código são,

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

Esta é a declaração da função principal (`main`) do programa. Assim como em qualquer programa em C++, você precisa definir uma função principal que é a primeira função que será executada no programa. Não há nada de especial e seu código *ns-3* é apenas um programa C++.

As próximas duas linhas do código são usadas para habilitar dois componentes de registro que são construídos com as aplicações de *Echo Client* e *Echo Server*:

```
LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
```

Se você leu a documentação do componente de *logging* você viu que existem vários níveis de detalhamento de *logging* e que você pode habilitá-los para cada componente. Essas duas linhas de código habilitam a depuração de *logging* com o nível `INFO` para o cliente e servidor. Isto ira fazer com que as aplicações mostrem as mensagens dos pacotes sendo enviados e recebidos durante a simulação.

Agora nós iremos direto ao ponto, criando uma topologia e executando uma simulação. Nós usaremos o Assistente de Topologia para fazer isto da forma mais simples possível.

4.2.6 Assistentes de Topologia

Contêineres de nós (*NodeContainer*)

As próximas duas linhas de código cria os objetos do tipo `Node` que representarão os computadores na simulação.

```
NodeContainer nodes;
nodes.Create (2);
```

Antes de continuar vamos pesquisar a documentação da classe `NodeContainer`. Outra forma de obter a documentação é através aba `Classes` na página do Doxygen. No Doxygen, vá até o topo da página e selecione `Classes`. Então, você verá um novo conjunto de opções aparecendo, uma delas será a sub-aba `Class List`. Dentro desta opção você verá uma lista com todas as classes do *ns-3*. Agora procure por `ns3::NodeContainer`. Quando você achar a classe, selecione e veja a documentação da classe.

Lembre-se que uma de nossas abstrações é o nó. Este representa um computador, ao qual iremos adicionar coisas, como protocolos, aplicações e periféricos. O assistente `NodeContainer` fornece uma forma conveniente de criar, gerenciar e acessar qualquer objeto `Node` que criamos para executar a simulação. A primeira linha declara um `NodeContainer` que chamamos de `nodes`. A segunda linha chama o método `Create` sobre o objeto `nodes` e pede para criar dois nós.

Os nós, como estão no código, não fazem nada. O próximo passo é montar uma topologia para conectá-los em uma rede. Uma forma simples de conectar dois computadores em uma rede é com um enlace ponto-a-ponto.

PointToPointHelper

Construiremos um enlace ponto-a-ponto e para isto usaremos um assistente para configurar o nível mais baixo da rede. Lembre-se que duas abstrações básicas são `NetDevice` e `Channel`. No mundo real, estes termos correspondem, a grosso modo, à placa de rede e ao cabo. Normalmente, estes dois estão intimamente ligados e não é normal ficar trocando. Por exemplo, não é comum placas Ethernet conectadas em canais sem fio. O assistente de topologia acopla estes dois conceitos em um simples `PointToPointHelper` para configurar e conectar objetos `PointToPointNetDevice` e `PointToPointChannel` em nosso código.

As próximas três linhas no código são,

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

A primeira linha,

```
PointToPointHelper pointToPoint;
```

instancia o objeto `PointToPointHelper` na pilha. De forma mais superficial a próxima linha,

```
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
```

diz ao objeto `PointToPointHelper` para usar o valor de "5Mbps" (cinco *megabits* por segundo) como "DataRate" (taxa de transferência) quando criarmos um objeto `PointToPointNetDevice`.

De uma perspectiva mais detalhada, a palavra "DataRate" corresponde ao que nós chamamos de atributo (`Attribute`) do `PointToPointNetDevice`. Se você olhar no Doxygen na classe `ns3::PointToPointNetDevice` e procurar a documentação para o método `GetTypeId`, você achará uma lista de atributos definidos por dispositivos. Dentro desses está o atributo "DataRate". A maioria dos objetos do *ns-3* tem uma lista similar de atributos. Nós usamos este mecanismo para facilitar a configuração das simulações sem precisar recompilar, veremos isto na seção seguinte.

Parecido com o "DataRate" no `PointToPointNetDevice` você achará o atributo "Delay" (atraso) associado com o `PointToPointChannel`. O final da linha,

```
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

diz ao `PointToPointHelper` para usar o valor de "2ms" (dois milissegundos) como valor de atraso de transmissão para o canal ponto-a-ponto criado.

NetDeviceContainer

Até agora no código, temos um `NodeContainer` que contém dois nós. Também temos `PointToPointHelper` que carrega e prepara os objetos `PointToPointNetDevices` e `PointToPointChannel`. Depois, usamos o assistente `NodeContainer` para criar os nós para a simulação. Iremos pedir ao `PointToPointHelper` para criar, configurar e instalar nossos dispositivos. Iremos necessitar de uma lista de todos os objetos *NetDevice* que são criados, então nós usamos um *NetDeviceContainer* para agrupar os objetos criados, tal como usamos o *NodeContainer* para agrupar os nós que criamos. Nas duas linhas de código seguintes,

```
NetDeviceContainer devices;
devices = pointToPoint.Install (nodes);
```

vamos terminar configurando os dispositivos e o canal. A primeira linha declara o contêiner de dispositivos mencionado anteriormente e o segundo faz o trabalho pesado. O método `Install` do `PointToPointHelper` utiliza um `NodeContainer` como parâmetro. Internamente, um `NetDeviceContainer` é criado. Para cada nó no `NodeContainer` (devem existir dois para um enlace ponto-a-ponto) um `PointToPointNetDevice` é criado e salvo no contêiner do dispositivo. Um `PointToPointChannel` é criado e dois `PointToPointNetDevices` são conectados. Quando os objetos são criados pelo `PointToPointHelper`, os atributos, passados anteriormente, são configurados pelo assistente (*Helper*).

Depois de executar a chamada `pointToPoint.Install (nodes)` iremos ter dois nós, cada qual instalado na rede ponto-a-ponto e um único canal ponto-a-ponto ligando os dois. Ambos os dispositivos serão configurados para ter uma taxa de transferência de dados de cinco megabits por segundo, que por sua vez tem um atraso de transmissão de dois milissegundos.

InternetStackHelper

Agora temos os nós e dispositivos configurados, mas não temos qualquer pilha de protocolos instalada em nossos nós. As próximas duas linhas de código irão cuidar disso.

```
InternetStackHelper stack;
stack.Install (nodes);
```

O `InternetStackHelper` é um assistente de topologia inter-rede. O método `Install` utiliza um `NodeContainer` como parâmetro. Quando isto é executado, ele irá instalar a pilha de protocolos da Internet (TCP, UDP, IP, etc) em cada nó do contêiner.

Ipv4AddressHelper

Agora precisamos associar os dispositivos dos nós a endereços IP. Nós fornecemos um assistente de topologia para gerenciar a alocação de endereços IP's. A única API de usuário visível serve para configurar o endereço IP base e a máscara de rede, usado para alocação de endereços.

As próximas duas linhas de código,

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
```

declara um assistente de endereçamento e diz para ele iniciar a alocação de IP's na rede 10.1.1.0 usando a máscara 255.255.255.0. Por padrão, os endereços alocados irão iniciar do primeiro endereço IP disponível e serão incrementados um a um. Então, o primeiro endereço IP alocado será o 10.1.1.1, seguido pelo 10.1.1.2, etc. Em um nível mais baixo, o *ns-3* mantém todos os endereços IP's alocados e gera um erro fatal se você acidentalmente usar o mesmo endereço duas vezes (esse é um erro muito difícil de depurar).

A próxima linha de código,

```
Ipv4InterfaceContainer interfaces = address.Assign (devices);
```

realiza efetivamente o endereçamento. No *ns-3* nós fazemos a associação entre endereços IP e dispositivos usando um objeto `Ipv4Interface`. As vezes precisamos de uma lista dos dispositivos de rede criados pelo assistente de topologia para consultas futuras. O `Ipv4InterfaceContainer` fornece esta funcionalidade.

Agora que nós temos uma rede ponto-a-ponto funcionando, com pilhas de protocolos instaladas e endereços IP's configurados. O que nós precisamos são aplicações para gerar o tráfego de rede.

4.2.7 Aplicações

Outra abstração do núcleo do *ns-3* são as aplicações (`Application`). Neste código são utilizadas duas especializações da classe `Application` chamadas `UdpEchoServerApplication` e `UdpEchoClientApplication`. Assim como nas explicações anteriores que nós utilizamos assistentes para configurar e gerenciar outros objetos, nós usaremos os objetos `UdpEchoServerHelper` e `UdpEchoClientHelper` para deixar a nossa vida mais fácil.

UdpEchoServerHelper

As linhas seguintes do código do exemplo `first.cc`, são usadas para configurar uma aplicação de eco (*echo*) UDP em um dos nós criados anteriormente.

```
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (nodes.Get (1));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

Um fragmento da primeira linha do código declara o `UdpEchoServerHelper`. Esta não é a própria aplicação, é o objeto usado para ajudar na criação da aplicação. Uma de nossas convenções é colocar os atributos *obrigatórios* no construtor do assistente de topologia. Neste caso, o assistente não pode fazer nada se não colocarmos um número de porta que o cliente conhece. O construtor, por sua vez, configura o atributo "Port" usando `SetAttribute`.

De forma semelhante aos outros assistentes, o `UdpEchoServerHelper` tem o método `Install`. É este método que instancia a aplicação de servidor de eco (`echo server`) e a associa ao nó. O método `Install` tem como parâmetro um `NodeContainter`, assim como o outro método `Install` visto. Isto é o que é passado para o método, mesmo que não seja visível. Há uma *conversão implícita* em C++, que pega o resultado de `nodes.Get (1)` (o qual retorna um ponteiro para o objeto *node* — `Ptr<Node>`) e o usa em um construtor de um `NodeContainer` sem nome, que então é passado para o método `Install`.

Agora vemos que o `echoServer.Install` instala um `UdpEchoServerApplication` no primeiro nó do `NodeContainer`. O `Install` irá retornar um contêiner que armazena os ponteiros de todas as aplicações (neste caso passamos um `NodeContainer` contendo um nó) criadas pelo assistente de topologia.

As aplicações requerem um tempo para "iniciar" a geração de tráfego de rede e podem ser opcionalmente "desligadas". Estes tempos podem ser configurados usando o `ApplicationContainer` com os métodos `Start` e `Stop`, respectivamente. Esses métodos possuem o parâmetro `Time`. Em nosso exemplo, nós usamos uma convenção explicita do C++ para passar 1.0 e converter em um objeto `Time` usando segundos. Esteja ciente que as regras de conversão podem ser controladas pelo autor do modelo e o C++ tem suas próprias regras, desta forma, você não pode assumir que o parâmetro sempre vai ser convertido para você. As duas linhas,

```
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

irão iniciar (`Start`) a aplicação de servidor de eco um segundo após o início da simulação e depois desligar (`Stop`) em dez segundos. Em virtude de termos declarado que um evento de simulação (o evento de desligamento da aplicação) deve ser executado por dez segundos, a simulação vai durar pelo menos dez segundos.

UdpEchoClientHelper

A aplicação cliente de eco é configurada de forma muito similar ao servidor. Há o `UdpEchoClientApplication` que é gerenciado por um `UdpEchoClientHelper`.

```
UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps = echoClient.Install (nodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

Para o cliente de eco, precisamos configurar cinco diferentes atributos. Os dois primeiros são configurados durante a construção do `UdpEchoClientHelper`. Passamos os parâmetros que são usados (internamente pelo Assistente) para configurar os atributos "RemoteAddress" (endereço remoto) e "RemotePort" (porta remota).

Lembre-se que usamos um `Ipv4InterfaceContainer` para configurar o endereço IP em nossos dispositivos. A interface zero (primeira) no contêiner corresponde ao endereço IP do nó zero no contêiner de nós. A primeira interface corresponde ao endereço IP do primeiro nó. Então, na primeira linha do código anterior, nós criamos um assistente e dizemos ao nó para configurar o endereço remoto do cliente conforme o IP do servidor. Nós dizemos também para enviar pacotes para a porta nove.

O atributo "MaxPackets" diz ao cliente o número máximo de pacotes que são permitidos para envio durante a simulação. O atributo "Interval" diz ao cliente quanto tempo esperar entre os pacotes e o "PacketSize" informa ao cliente qual é o tamanho da área de dados do pacote. Com esta combinação de atributos que nós fizemos teremos clientes enviando pacotes de 1024 bytes.

Assim como no caso do servidor de eco, nós dizemos para o cliente de eco iniciar e parar, mas aqui nós iniciamos o cliente um segundo depois que o servidor estiver funcionando (com dois segundos de simulação).

4.2.8 Simulador (*Simulator*)

O que nós precisamos agora é executar o simulador. Isto é feito usando a função global `Simulator::Run`.

```
Simulator::Run ();
```

Quando nós chamamos os métodos

```
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
...
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

agendamos os eventos no simulador em 1 segundo, 2 segundos e dois eventos em 10 segundos. Quando chamamos `Simulator::Run`, o sistema verificará a lista de eventos agendados e os executará no momento apropriado. Primeiramente, ele vai executar o evento de 1 segundo que inicia a aplicação de servidor de eco. Depois executa o evento agendado com dois segundos (t=2,0) que iniciará a aplicação do cliente de eco. Estes eventos podem agendar muitos

outros eventos. O evento *start* do cliente irá iniciar a fase de transferência de dados na simulação enviando pacotes ao servidor.

O ato de enviar pacotes para o servidor vai disparar uma cadeia de eventos que serão automaticamente escalonados e executarão a mecânica do envio de pacotes de eco de acordo com os vários parâmetros de tempo que configuramos no código.

Considerando que enviamos somente um pacote (lembre-se que o atributo `MaxPackets` foi definido com um), uma cadeia de eventos será disparada por este único pedido de eco do cliente até cessar e o simulador ficará ocioso. Uma vez que isto ocorra, os eventos restantes serão o `Stop` do servidor e do cliente. Quando estes eventos forem executados, não havendo mais eventos para processar, o `Simulator::Run` retorna. A simulação está completa.

Tudo que resta é limpar. Isto é feito chamando uma função global chamada `Simulator::Destroy`. Uma das funções dos assistentes (ou do código de baixo nível do *ns-3*) é agrupar todos os objetos que foram criados e destruí-los. Você não precisa tratar estes objetos — tudo que precisa fazer é chamar `Simulator::Destroy` e sair. O *ns-3* cuidará desta difícil tarefa para você. As linhas restantes do código fazem isto:

```
  Simulator::Destroy ();
  return 0;
}
```

4.2.9 Construindo o código

É trivial construir (criar os binários de) seu código. Tudo que tem a fazer é copiar seu código para dentro do diretório `scratch` e ele será construído automaticamente quando executar o Waf. Copie `examples/tutorial/first.cc` para o diretório `scratch` e depois volte ao diretório principal.

```
cd ../..
cp examples/tutorial/first.cc scratch/myfirst.cc
```

Agora construa seu primeiro exemplo usando o Waf:

```
./waf
```

Você deve ver mensagens reportando que o seu exemplo `myfirst` foi construído com sucesso.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
[614/708] cxx: scratch/myfirst.cc -> build/debug/scratch/myfirst_3.o
[706/708] cxx_link: build/debug/scratch/myfirst_3.o -> build/debug/scratch/myfirst
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (2.357s)
```

Você agora pode executar o exemplo (note que se você construiu seu programa no diretório `scratch`, então deve executar o comando fora deste diretório):

```
./waf --run scratch/myfirst
```

Você deverá ver algumas saídas:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.418s)
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

O sistema verifica se os arquivos foram construídos e então executa-os. Através do componente de registro vemos que o cliente enviou 1024 bytes para o servidor através do IP 10.1.1.2. Também podemos ver que o servidor diz ter recebido 1024 bytes do IP 10.1.1.1 e ecoa o pacote para o cliente, que registra o seu recebimento.

4.3 Código fonte do Ns-3

Agora que você já utilizou alguns assistentes do *ns-3*, podemos dar uma olhada no código fonte que implementa estas funcionalidades. Pode-se navegar o código mais recente no seguinte endereço: http://code.nsnam.org/ns-3-dev. Lá você verá a página de sumário do Mercurial para a árvore de desenvolvimento do *ns-3*.

No início da página, você verá vários *links*,

```
summary | shortlog | changelog | graph | tags | files
```

selecione `files`. Aparecerá o primeiro nível do repositório:

```
drwxr-xr-x                                [up]
drwxr-xr-x                                bindings python  files
drwxr-xr-x                                doc              files
drwxr-xr-x                                examples         files
drwxr-xr-x                                ns3              files
drwxr-xr-x                                scratch          files
drwxr-xr-x                                src              files
drwxr-xr-x                                utils            files
-rw-r--r-- 2009-07-01 12:47 +0200 560     .hgignore        file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 1886    .hgtags          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 1276    AUTHORS          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 30961   CHANGES.html      file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 17987   LICENSE          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 3742    README           file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 16171   RELEASE_NOTES    file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 6       VERSION          file | revisions | annotate
-rwxr-xr-x 2009-07-01 12:47 +0200 88110   waf              file | revisions | annotate
-rwxr-xr-x 2009-07-01 12:47 +0200 28      waf.bat          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 35395   wscript          file | revisions | annotate
-rw-r--r-- 2009-07-01 12:47 +0200 7673    wutils.py        file | revisions | annotate
```

Os códigos exemplo estão no diretório `examples`. Se você clicar verá uma lista de subdiretórios. Um dos arquivos no subdiretório `tutorial` é o `first.cc`. Clicando nele você encontrará o código que acabamos de analisar.

O código fonte é mantido no diretório `src`. Você pode vê-lo clicando sobre o nome do diretório ou clicando no item `files` a direita do nome. Clicando no diretório `src`, obterá uma lista de subdiretórios. Clicando no subdiretório `core`, encontrará um lista de arquivos. O primeiro arquivo é o `abort.h`, que contém macros caso condições anormais sejam encontradas.

O código fonte para os assistentes utilizados neste capítulo podem ser encontrados no diretório `src/applications/helper`. Sinta-se à vontade para explorar a árvore de diretórios e ver o estilo de código do *ns-3*.

CAPÍTULO 5

Aprofundando Conhecimentos

5.1 Usando o Módulo de Registro

Já demos uma breve olhada no módulo de Registro (do inglês, *log* ou *logging*) do *ns-3*, enquanto analisávamos o código `first.cc`. Agora iremos aprofundar nossos conhecimento sobre este módulo para utilizá-lo de forma mais eficiente.

5.1.1 Visão Geral Sobre o Sistema de Registro

A maioria dos sistemas de grande porte suportam mensagens de registros (mensagens de 'log' que informam o que esta ocorrendo no sistema) e o *ns-3* não é nenhuma exceção. Em alguns casos, somente mensagens de erros são reportadas para o "operador do console" (que é geralmente o `stderr` dos sistemas baseados no Unix). Em outros sistemas, mensagens de avisos podem ser impressas detalhando informações do sistema. Em alguns casos, o sistema de registro fornece mensagens de depuração que podem rapidamente mostrar o que está ocorrendo de errado.

O *ns-3* permite que o usuário tenha visões de todos os níveis do sistema através das mensagens de registro. Podemos selecionar o nível de saída das mensagens de registro(*verbosty*), através de um abordagem multinível. As mensagens de registro podem ser desabilitadas completamente, habilitadas componente por componente ou habilitada globalmente. Ou seja, permite selecionar o nível de detalhamento das mensagens de saída. O módulo de registro do *ns-3* fornece uma forma correta e segura de obtermos informações sobre as simulações.

No *ns-3* foi implementado um mecanismo de — rastreamento — de propósito geral, que permite a obtenção de saídas de dados dos modelos simulados (veja a seção Usando o Sistema de Rastreamento do tutorial para mais detalhes). O sistema de registro deve ser usado para depurar informações, alertas, mensagens de erros ou para mostrar qualquer informação dos *scripts* ou modelos.

Atualmente existem sete níveis de mensagens de registro definidas no sistema.

- NS_LOG_ERROR — Registra mensagens de erro;
- NS_LOG_WARN — Registra mensagens de alertas;
- NS_LOG_DEBUG — Registra mensagens mais raras, mensagens de depuração *ad-hoc*;
- NS_LOG_INFO — Registra mensagens informativas sobre o progresso do programa;
- NS_LOG_FUNCTION — Registra mensagens descrevendo cada função chamada;
- NS_LOG_LOGIC — Registra mensagens que descrevem o fluxo lógico dentro de uma função;
- NS_LOG_ALL — Registra tudo.

Também é fornecido um nível de registro incondicional, que sempre é exibido independente do nível de registro ou do componente selecionado.

- NS_LOG_UNCOND — Registra mensagens incondicionalmente.

Cada nível pode ser requerido individualmente ou de forma cumulativa. O registro pode ser configurado usando uma variável de ambiente (`NS_LOG`) ou através de uma chamada ao sistema de registro. Já havíamos abordado anteriormente o sistema de registro, através da documentação Doxygen, agora é uma boa hora para ler com atenção esta documentação no Doxygen.

Depois de ler a documentação, vamos usar nosso conhecimento para obter algumas informações importante do código de exemplo `scratch/myfirst.cc`.

5.1.2 Habilitando o Sistema de Registro

Utilizaremos a variável de ambiente `NS_LOG` para habilitar o sistema de Registro, mas antes de prosseguir, execute o código feito na seção anterior,

```
./waf --run scratch/myfirst
```

Veremos a saída do nosso primeiro programa *ns-3* de exemplo, tal como visto anteriormente.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.413s)
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

As mensagens de envio e recebimentos apresentadas anteriormente são mensagens de registro, obtidas de `UdpEchoClientApplication` e `UdpEchoServerApplication`. Podemos pedir para a aplicação cliente, por exemplo, imprimir mais informações configurando o nível de registro através da variável de ambiente `NS_LOG`.

Aqui estamos assumindo que você está usando um *shell* parecido com o `sh`, que usa a sintaxe "VARIABLE=value". Se você estiver usando um *shell* parecido com o `csh`, então terá que converter os exemplos para a sintaxe "setenv VARIABLE value", requerida por este *shell*.

Agora, a aplicação cliente de eco UDP irá responder a seguinte linha de código `scratch/myfirst.cc`,

```
LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
```

Essa linha de código habilita o nível `LOG_LEVEL_INFO` de registro. Quando habilitamos um dado nível de registro, estamos habilitando este nível e todos os níveis inferiores a este. Neste caso, habilitamos `NS_LOG_INFO`, `NS_LOG_DEBUG`, `NS_LOG_WARN` e `NS_LOG_ERROR`. Podemos aumentar o nível de registro e obter mais informações sem alterar o *script*, ou seja, sem ter que recompilar. Conseguimos isto através da configuração da variável de ambiente `NS_LOG`, tal como:

```
export NS_LOG=UdpEchoClientApplication=level_all
```

Isto configura a variável de ambiente `NS_LOG` no *shell* para,

```
UdpEchoClientApplication=level_all
```

Do lado esquerdo do comando temos o nome do componente de registro que nós queremos configurar, no lado direito fica o valor que estamos passando. Neste caso, estamos ligando todos os níveis de depuração para a aplicação. Se executarmos o código com o `NS_LOG` configurado desta forma, o sistema de registro do *ns-3* observará a mudança e mostrará a seguinte saída:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.404s)
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
UdpEchoClientApplication:HandleRead(0x6241e0, 0x624a20)
Received 1024 bytes from 10.1.1.2
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
```

As informações de depuração extras, apresentadas aqui, são fornecidas pela aplicação no nível de registros `NS_LOG_FUNCTION`. Isto é apresentado toda vez que a aplicação chamar a função. Não é obrigatório que o modelo forneça suporte a registro, no *ns-3*, esta decisão cabe ao desenvolvedor do modelo. No caso da aplicação de eco uma grande quantidade de saídas de *log* estão disponíveis.

Podemos ver agora registros de várias funções executadas pela aplicação. Se olharmos mais de perto veremos que as informações são dadas em colunas separadas por (`::`), do lado esquerdo está o nome da aplicação (no exemplo, `UdpEchoClientApplication`) e do outro lado o nome do método esperado pelo escopo C++. Isto é incremental.

O nome que está aparece no registro não é necessariamente o nome da classe, mas sim o nome do componente de registro. Quando existe uma correspondência um-para-um, entre código fonte e classe, este geralmente será o nome da classe, mas isto nem sempre é verdade. A maneira sutil de diferenciar esta situação é usar `:` quando for o nome do componente de registro e `::` quando for o nome da classe.

Em alguns casos pode ser complicado determinar qual método gerou a mensagem. Se olharmos o texto anterior, veremos a mensagem "`Received 1024 bytes from 10.1.1.2`", nesta não existe certeza de onde a mensagem veio. Podemos resolver isto usando um "OU" (*OR*) entre o nível de registro e o `prefix_func`, dentro do `NS_LOG`.

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func'
```

As aspas são requeridas quando usamos o | (*pipe*) para indicar uma operação de OU.

Agora, se executarmos o *script* devemos ver que o sistema de registro informa de qual componente de registro vem a mensagem.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.417s)
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
UdpEchoClientApplication:HandleRead(0x6241e0, 0x624a20)
UdpEchoClientApplication:HandleRead(): Received 1024 bytes from 10.1.1.2
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
```

Podemos ver, depois da configuração, que todas as mensagens do cliente de eco UDP estão identificadas. Agora a mensagem "Received 1024 bytes from 10.1.1.2" é claramente identificada como sendo do cliente de eco. O restante

das mensagens devem estar vindo do servidor de eco UDP. Podemos habilitar mais do que um componente usando `:`, para separá-los na variável `NS_LOG`.

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func:
               UdpEchoServerApplication=level_all|prefix_func'
```

Atenção: não podemos quebrar a entrada da variável em várias linhas como foi feito no exemplo, tudo deve estar em uma única linha. O exemplo ficou assim por uma questão de formatação do documento.

Agora, se executarmos o *script* veremos todas as mensagens de registro tanto do cliente quando do servidor. Isto é muito útil na depuração de problemas.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.406s)
UdpEchoServerApplication:UdpEchoServer()
UdpEchoClientApplication:UdpEchoClient()
UdpEchoClientApplication:SetDataSize(1024)
UdpEchoServerApplication:StartApplication()
UdpEchoClientApplication:StartApplication()
UdpEchoClientApplication:ScheduleTransmit()
UdpEchoClientApplication:Send()
UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
UdpEchoServerApplication:HandleRead(): Received 1024 bytes from 10.1.1.1
UdpEchoServerApplication:HandleRead(): Echoing packet
UdpEchoClientApplication:HandleRead(0x624920, 0x625160)
UdpEchoClientApplication:HandleRead(): Received 1024 bytes from 10.1.1.2
UdpEchoServerApplication:StopApplication()
UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoServerApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
UdpEchoServerApplication:~UdpEchoServer()
```

As vezes também é útil registrar o tempo em que uma mensagem é gerada. Podemos fazer isto através de um OU com o *prefix_time*, exemplo:

```
export 'NS_LOG=UdpEchoClientApplication=level_all|prefix_func|prefix_time:
               UdpEchoServerApplication=level_all|prefix_func|prefix_time'
```

Novamente, teremos que deixar tudo em uma única linha e não em duas como no exemplo anterior. Executando o *script*, veremos a seguinte saída:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.418s)
0s UdpEchoServerApplication:UdpEchoServer()
0s UdpEchoClientApplication:UdpEchoClient()
0s UdpEchoClientApplication:SetDataSize(1024)
1s UdpEchoServerApplication:StartApplication()
2s UdpEchoClientApplication:StartApplication()
2s UdpEchoClientApplication:ScheduleTransmit()
2s UdpEchoClientApplication:Send()
2s UdpEchoClientApplication:Send(): Sent 1024 bytes to 10.1.1.2
2.00369s UdpEchoServerApplication:HandleRead(): Received 1024 bytes from 10.1.1.1
2.00369s UdpEchoServerApplication:HandleRead(): Echoing packet
2.00737s UdpEchoClientApplication:HandleRead(0x624290, 0x624ad0)
2.00737s UdpEchoClientApplication:HandleRead(): Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
```

```
10s UdpEchoClientApplication:StopApplication()
UdpEchoClientApplication:DoDispose()
UdpEchoServerApplication:DoDispose()
UdpEchoClientApplication:~UdpEchoClient()
UdpEchoServerApplication:~UdpEchoServer()
```

Podemos ver que o construtor para o `UdpEchoServer` foi chamado pelo simulador no segundo 0 (zero). Isto acontece antes do simulador ser iniciado, mas o tempo é mostrado como zero, o mesmo acontece para o construtor do `UdpEchoClient`.

Lembre-se que o *script* `scratch/first.cc` inicia a aplicação servidor de eco no primeiro segundo da simulação. Repare que o método `StartApplication` do servidor é, de fato, chamado com um segundo. Também podemos notar que a aplicação cliente de eco é iniciada com dois segundos de simulação, como nós pedimos no *script*.

Agora podemos acompanhar o andamento da simulação: `ScheduleTransmit` é chamado no cliente, que invoca o `Send` e o `HandleRead`, que é usado na aplicação servidor de eco. Repare que o tempo decorrido entre o envio de cada pacote é de 3.69 milissegundos. Veja que a mensagem de registro do servidor diz que o pacote foi ecoado e depois houve um atraso no canal. Podemos ver que o cliente recebeu o pacote ecoado pelo método `HandleRead`.

Existe muita coisa acontecendo por baixo dos panos e que não estamos vendo. Podemos facilmente seguir as entradas de processo configurando todos os componentes de registro do sistema. Configure a variável de `NS_LOG` da seguinte forma,

```
export 'NS_LOG=*=level_all|prefix_func|prefix_time'
```

O asterisco é um componente coringa, que ira ligar todos os componentes de registro usados na simulação. Não vamos reproduzir a saída aqui (cada pacote de eco produz 1265 linhas de saída), mas podemos redirecionar esta informação para um arquivo e visualizá-lo depois em um editor de textos,

```
./waf --run scratch/myfirst > log.out 2>&1
```

utilizamos uma versão extremamente detalhada de registro quando surge um problema e não temos ideia do que está errado. Assim, podemos seguir o andamento do código e depurar o erro. Podemos assim visualizar a saída em um editor de texto e procurar por coisas que nós esperamos e principalmente por coisa que não esperávamos. Quando temos uma ideia geral sobre o que está acontecendo de errado, usamos um depurador de erros para examinarmos de forma mais detalhada o problema. Este tipo de saída pode ser especialmente útil quando nosso *script* faz algo completamente inesperado. Se estivermos depurando o problema passo a passo, podemos nos perder completamente. O registro pode tornar as coisas mais visíveis.

5.1.3 Adicionando registros ao Código

Podemos adicionar novos registros nas simulações fazendo chamadas para o componente de registro através de várias macros. Vamos fazer isto em nosso código `myfirst.cc` no diretório `scratch`

Lembre-se que nós temos que definir o componente de registro em nosso código:

```
NS_LOG_COMPONENT_DEFINE ("FirstScriptExample");
```

Agora que sabemos habilitar todos os registros em vários níveis configurando a variável `NS_LOG`. Vamos adicionar alguns registros ao código. O macro usado para adicionar uma mensagem ao nível de informação é `NS_LOG_INFO`, então vamos adicionar uma mensagem dessas (pouco antes de criar os nós de rede) que diz que a "Topologia foi Criada". Isto é feito como neste trecho do código,

Abra o arquivo `scratch/myfirst.cc` e adicione a linha,

```
NS_LOG_INFO ("Creating Topology");
```

antes das linhas,

```
NodeContainer nodes;
nodes.Create (2);
```

Agora construa o código usando o Waf e limpe a variável `NS_LOG` desabilite o registro que nós havíamos habilitado anteriormente:

```
./waf
export NS_LOG=
```

Agora, se executarmos o código,

```
./waf --run scratch/myfirst
```

veremos novas mensagens, pois o componente de registro não está habilitado. Agora para ver a mensagem devemos habilitar o componente de registro do `FirstScriptExample` com um nível maior ou igual a `NS_LOG_INFO`. Se só esperamos ver um nível particular de registro, devemos habilita-lo,

```
export NS_LOG=FirstScriptExample=info
```

Agora se executarmos o código veremos nossa mensagem de registro "Creating Topology",

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.404s)
Creating Topology
Sent 1024 bytes to 10.1.1.2
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.1.2
```

5.2 Usando Argumentos na Linha de Comando

5.2.1 Sobrepondo Atributos Padrões

Podemos alterar o comportamento dos códigos do *ns-3* sem precisar editar ou construir códigos, isto é feito através de linhas de comandos. Para isto o *ns-3* fornece um mecanismo de analise de argumentos de linha de comando (*parse*), que configura automaticamente variáveis locais e globais através desses argumentos.

O primeiro passo para usar argumentos de linha de comando é declarar o analisador de linha de comandos. Isto é feito com a seguinte linha de programação, em seu programa principal,

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

  CommandLine cmd;
  cmd.Parse (argc, argv);

  ...
}
```

Estas duas linhas de programação são muito uteis. Isto abre uma porta para as variáveis globais e atributos do *ns-3*. Adicione estas duas linhas no código em nosso exemplo `scratch/myfirst.cc`, bem no inicio da função principal (`main`). Na sequencia construa o código e execute-o, mas peça para o código "ajudar" da seguinte forma,

```
./waf --run "scratch/myfirst --PrintHelp"
```

Isto pede ao Waf para executar o `scratch/myfirst` e passa via linha de comando o argumento `--PrintHelp`. As aspas são necessárias para ordenar os argumentos. O analisador de linhas de comandos agora tem como argumento o `--PrintHelp` e responde com,

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.413s)
TcpL4Protocol:TcpStateMachine()
CommandLine:HandleArgument(): Handle arg name=PrintHelp value=
--PrintHelp: Print this help message.
--PrintGroups: Print the list of groups.
--PrintTypeIds: Print all TypeIds.
--PrintGroup=[group]: Print all TypeIds of group.
--PrintAttributes=[typeid]: Print all attributes of typeid.
--PrintGlobals: Print the list of globals.
```

Vamos focar na opção `--PrintAttributes`. Já demos a dica sobre atributos no *ns-3* enquanto explorávamos o código do `first.cc`. Nós olhamos as seguintes linhas de código,

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));
```

e mencionamos que `DataRate` é um atributo de `PointToPointNetDevice`. Vamos usar os argumentos de linha de comando para ver os atributos de `PointToPointNetDevice`. A lista de ajuda diz que nós devemos fornecer um `TypeId`, este corresponde ao nome da classe do atributo. Neste caso será `ns3::PointToPointNetDevice`. Seguindo em frente digite,

```
./waf --run "scratch/myfirst --PrintAttributes=ns3::PointToPointNetDevice"
```

O sistema irá mostrar todos os atributos dos tipos de dispositivos de rede (*net device*). Entre os atributos veremos,

```
--ns3::PointToPointNetDevice::DataRate=[32768bps]:
  The default data rate for point to point links
```

32768 bits por segundos é o valor padrão que será usado quando criarmos um `PointToPointNetDevice` no sistema. Vamos alterar este valor padrão do `PointToPointHelper`. Para isto iremos usar os valores dos dispositivos ponto-a-ponto e dos canais, deletando a chamada `SetDeviceAttribute` e `SetChannelAttribute` do `myfirst.cc`, que nós temos no diretório `scratch`.

Nosso código agora deve apenas declarar o `PointToPointHelper` sem configurar qualquer operação, como no exemplo a seguir,

```
...

NodeContainer nodes;
nodes.Create (2);

PointToPointHelper pointToPoint;

NetDeviceContainer devices;
devices = pointToPoint.Install (nodes);

...
```

Agora construa o novo código com o Waf (`./waf`) e depois vamos habilitar alguns registros para o servidor de eco UDP e ligar o prefixo de informações sobre tempo de execução.

```
export 'NS_LOG=UdpEchoServerApplication=level_all|prefix_time'
```

Agora ao executar o código veremos a seguinte saída,

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.405s)
0s UdpEchoServerApplication:UdpEchoServer()
1s UdpEchoServerApplication:StartApplication()
Sent 1024 bytes to 10.1.1.2
2.25732s Received 1024 bytes from 10.1.1.1
2.25732s Echoing packet
Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
UdpEchoServerApplication:DoDispose()
UdpEchoServerApplication:~UdpEchoServer()
```

Lembre-se que o último tempo que vimos na simulação quando recebemos um pacote de eco no servidor, foi de 2.00369 segundos.

```
2.00369s UdpEchoServerApplication:HandleRead(): Received 1024 bytes from 10.1.1.1
```

Agora o pacote é recebido em 2.25732 segundos. Isto porque retiramos a taxa de transferência do `PointToPointNetDevice` e portanto foi assumido o valor padrão 32768 bits por segundos ao invés de cinco megabits por segundo.

Se nós passarmos uma nova taxa de dados usando a linha de comando, podemos aumentar a velocidade da rede novamente. Nós podemos fazer isto da seguinte forma, usando um *help*:

```
./waf --run "scratch/myfirst --ns3::PointToPointNetDevice::DataRate=5Mbps"
```

Isto ira configurar o valor do atributo `DataRate` para cinco megabits por segundos. Ficou surpreso com o resultado? Acontece que para obtermos o resultado do código original, teremos que configurar também o atraso do canal de comunicação. Podemos fazer isto via linha de comandos, tal como fizemos com o dispositivo de rede:

```
./waf --run "scratch/myfirst --PrintAttributes=ns3::PointToPointChannel"
```

Então descobrimos que o atributo `Delay` do canal esta configurado com o seguinte valor padrão:

```
--ns3::PointToPointChannel::Delay=[0ns]:
  Transmission delay through the channel
```

Podemos configurar ambos valores via linha de comando,

```
./waf --run "scratch/myfirst
  --ns3::PointToPointNetDevice::DataRate=5Mbps
  --ns3::PointToPointChannel::Delay=2ms"
```

neste caso voltamos com os tempos de `DataRate` e `Delay` que tínhamos inicialmente no código original:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.417s)
0s UdpEchoServerApplication:UdpEchoServer()
1s UdpEchoServerApplication:StartApplication()
Sent 1024 bytes to 10.1.1.2
2.00369s Received 1024 bytes from 10.1.1.1
2.00369s Echoing packet
Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
```

```
UdpEchoServerApplication:DoDispose()
UdpEchoServerApplication:~UdpEchoServer()
```

Repare que o pacote é recebido novamente pelo servidor com 2.00369 segundos. Então desta forma, podemos configurar qualquer atributo usado no código. Em particular nós podemos configurar o atributo `MaxPackets` do `UdpEchoClient` para qualquer outro valor.

É importante lembrar que devemos retirar todas as configurações com valores explícitos do código. Depois disto devemos re-construir o código (fazer novamente os binários). Também teremos que achar a sintaxe do atributo usando o *help* da linha de comando. Uma vez que tenhamos este cenário estaremos aptos para controlar o números de pacotes ecoados via linha de comando. No final a linha de comando deve parecer com algo como:

```
./waf --run "scratch/myfirst
  --ns3::PointToPointNetDevice::DataRate=5Mbps
  --ns3::PointToPointChannel::Delay=2ms
  --ns3::UdpEchoClient::MaxPackets=2"
```

5.2.2 Conectando Seus Próprios Valores

Podemos também adicionar conectores (opções que alteram valores de variáveis) ao sistema de linha de comando. Isto nada mais é do que criar uma opção na linha de comando, a qual permitirá a configuração de uma variável dentro do código. Isto é feito usando o método `AddValue` no analisador da linha de comando.

Vamos usar esta facilidade para especificar o número de pacotes de eco de uma forma completamente diferente. Iremos adicionar uma variável local chamada `nPackets` na função `main`. Vamos iniciar com nosso valor anterior. Para permitir que a linha de comando altere este valor, precisamos fixar o valor no *parser*. Fazemos isto adicionando uma chamada para `AddValue`. Altere o código `scratch/myfirst.cc` começando pelo seguinte trecho de código,

```
int
main (int argc, char *argv[])
{
  uint32_t nPackets = 1;

  CommandLine cmd;
  cmd.AddValue("nPackets", "Number of packets to echo", nPackets);
  cmd.Parse (argc, argv);

  ...
```

Dê uma olhada um pouco mais para baixo, no código e veja onde configuramos o atributo `MaxPackets`, retire o `1` e coloque em seu lugar a variável `nPackets`, como é mostrado a seguir:

```
echoClient.SetAttribute ("MaxPackets", UintegerValue (nPackets));
```

Agora se executarmos o código e fornecermos o argumento `--PrintHelp`, deveremos ver nosso argumento de usuário (*User Arguments*), listado no *help*.

Execute,

```
./waf --run "scratch/myfirst --PrintHelp"

Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.403s)
--PrintHelp: Print this help message.
--PrintGroups: Print the list of groups.
--PrintTypeIds: Print all TypeIds.
--PrintGroup=[group]: Print all TypeIds of group.
```

```
--PrintAttributes=[typeid]: Print all attributes of typeid.
--PrintGlobals: Print the list of globals.
User Arguments:
    --nPackets: Number of packets to echo
```

Agora para especificar o número de pacotes de eco podemos utilizar o argumento `--nPackets` na linha de comando,

```
./waf --run "scratch/myfirst --nPackets=2"
```

Agora deveremos ver,

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.404s)
0s UdpEchoServerApplication:UdpEchoServer()
1s UdpEchoServerApplication:StartApplication()
Sent 1024 bytes to 10.1.1.2
2.25732s Received 1024 bytes from 10.1.1.1
2.25732s Echoing packet
Received 1024 bytes from 10.1.1.2
Sent 1024 bytes to 10.1.1.2
3.25732s Received 1024 bytes from 10.1.1.1
3.25732s Echoing packet
Received 1024 bytes from 10.1.1.2
10s UdpEchoServerApplication:StopApplication()
UdpEchoServerApplication:DoDispose()
UdpEchoServerApplication:~UdpEchoServer()
```

Agora, ecoamos dois pacotes. Muito fácil, não é?

Usando o *ns-3*, podemos usar argumentos via linha de comando para controlar valores de atributos. Ao construir modelos de simulação podemos adicionar novos atributos aos objetos e isto ficará disponível, automaticamente para ajustes através do sistema via linha de comando. Ao criarmos nossos códigos poderemos adicionar novas variáveis e conectá-las ao sistema de forma simples.

5.3 Usando o Sistema de Rastreamento

O ponto principal da simulação é gerar informações de saída para estudos futuros e o sistema de rastreamento (*Tracing System*) do *ns-3* é o mecanismo primário para isto. Devido ao fato do *ns-3* ser um programa escrito em C++, as funcionalidades para se gerar saídas podem ser utilizadas:

```
#include <iostream>
...
int main ()
{
  ...
  std::cout << "The value of x is " << x << std::endl;
  ...
}
```

Podemos usar o módulo de registro (visto anteriormente) para verificar pequenas estruturas de nossas soluções. Porém, os problemas gerados por esta abordagem já são bem conhecidos e portanto fornecemos um subsistema para rastrear eventos genéricos para localizar problemas importantes.

Os objetivos básicos do sistema de rastreamento *ns-3* são:

- Para as tarefas básicas, o sistemas de rastreamento fornece ao usuário um rastreamento padrão através de rastreamentos conhecidos e customização dos objetos que geram o rastreamento;

- Os usuários podem estender o sistema de rastreamento para modificar os formatos das saídas geradas ou inserir novas fontes de rastreamento, sem modificar o núcleo do simulador;
- Usuários avançados podem modificar o núcleo do simulador para adicionar novas origens de rastreamentos e destino do rastreamento.

O sistema de rastreamento do *ns-3* é feito através de conceitos independentes de rasteamento na origem e no destino, e um mecanismo uniforme para conectar a origem ao destino. O rastreador na origem são entidades que podem demonstrar eventos que ocorrem na simulação e fornece acesso aos dados importantes. Por exemplo, um rastreador de origem podem indicar quando um pacote é recebido por um dispositivo de rede e prove acesso aos comentários de pacote para os interessados no rastreador do destino.

Os rastreadores de origem não são usados sozinhos, eles devem ser "conectados" a outros pedaços de código que fazem alguma coisa útil com a informação fornecida pelo destino. Rastreador de destino são consumidores dos eventos e dados fornecidos pelos rastreadores de origem. Por exemplo, pode-se criar um rastreador de destino que (quando conectado ao rastreador de origem do exemplo anterior) mostrará saídas de partes importantes de pacotes recebidos.

A lógica desta divisão explicita é permitir que os usuários apliquem novos tipos de rastreadores de destinos em rastreadores de origem existentes, sem precisar editar ou recompilar o núcleo do simulador. Assim, no exemplo anterior, o usuário pode definir um novo rastreador de destino em seu código e aplicar isto a um rastreador de origem definido no núcleo da simulação editando, para isto, somente o código do usuário.

Neste tutorial, abordamos alguns rastreadores de origem e de destino já predefinidos e demonstramos como esses podem ser customizados, com um pouco de esforço. Veja o manual do ns-3 ou a seção de *how-to* para informações avançadas sobre configuração de rastreamento incluindo extensão do *namespace* de rastreamento e criação de novos rastreadores de origem.

5.3.1 Rastreamento ASCII

O *ns-3* fornece uma funcionalidade de ajuda (*helper*) que cobre rastreamento em baixo nível e ajuda com detalhes envolvendo configuração e rastros de pacotes. Se habilitarmos essa funcionalidade, veremos as saídas em arquivos ASCII (texto puro) — daí o nome. Isto é parecido com a saída do *ns-2*. Este tipo de rastreamento é parecido com o `out.tr` gerado por outros códigos.

Vamos adicionar algumas saídas de rastreamento ASCII em nosso código `scratch/myfirst.cc`. Antes de chamar `Simulator::Run ()` adicione as seguintes linhas de código:

```
AsciiTraceHelper ascii;
pointToPoint.EnableAsciiAll (ascii.CreateFileStream ("myfirst.tr"));
```

Tal como já vimos no *ns-3*, este código usa o objeto Assistente para criar o rastreador ASCII. A segunda linha aninhada duas chamadas para métodos. Dentro do método `CreateFileStream()` é utilizado um objeto para criar um outro objeto, que trata um arquivo, que é passado para o método. Iremos detalhar isto depois, agora tudo que precisamos saber é que estamos criando um objeto que representa um arquivo chamado "myfirst.tr". Estamos dizendo para o `ns-3` tratar problemas de criação de objetos e também para tratar problemas causados por limitações do C++ com objetos relacionados com cópias de construtores.

Fora da chamada, para `EnableAsciiAll()`, dizemos para o Assistente que esperamos habilitar o rastreamento ASCII para todo dispositivo ponto-a-ponto da simulação; e esperamos rastrear destinos e escrever as informações de saída sobre o movimento de pacotes no formato ASCII.

Para queles familiarizados com *ns-2*, os eventos rastreados são equivalentes aos populares pontos de rastreadores (*trace points*) que registram eventos "+", "-", "d", e "r"

Agora podemos construir o código e executa-lo:

```
./waf --run scratch/myfirst
```

Veremos algumas mensagens do Waf, seguida da mensagem "'build' finished successfully", bem como algumas mensagens do programa.

Quando isto for executado, o programa criará um arquivo chamado `myfirst.tr`. Devido a forma que o Waf trabalha, o arquivo não é criado no diretório local, mas sim no diretório raiz do repositório. Se você espera controlar o que é salvo, então use a opção `-cwd` do Waf. Agora mude para o diretório raiz do repositório e veja o arquivo `myfirst.tr` com um editor de texto.

Análise de Rastros ASCII

Uma grande quantidade de informação é gerada pelo sistema de rastreamento e pode ser difícil analisá-las de forma clara e consistente.

Cada linha do arquivo corresponde a um evento de rastreamento (*trace event*). Neste caso são eventos rastreados da fila de transmissão (*transmit queue*). A fila de transmissão é um lugar através do qual todo pacote destinado para o canal ponto-a-ponto deve passar. Note que cada linha no arquivo inicia com um único caractere (com um espaço depois). Este caractere tem o seguinte significado:

- `+`: Uma operação de enfileiramento (bloqueio) ocorreu no dispositivo de fila;
- `-`: Uma operação de desenfileiramento (desbloqueio) ocorre no dispositivo de fila;
- `d`: Um pacote foi descartado, normalmente por que a fila está cheia;
- `r`: Um pacote foi recebido por um dispositivo de rede.

Vamos detalhar mais a primeira linha do arquivo de rastreamento. Vamos dividi-la em seções com números de dois dígitos para referência:

```
00 +
01 2
02 /NodeList/0/DeviceList/0/$ns3::PointToPointNetDevice/TxQueue/Enqueue
03 ns3::PppHeader (
04   Point-to-Point Protocol: IP (0x0021))
05   ns3::Ipv4Header (
06     tos 0x0 ttl 64 id 0 protocol 17 offset 0 flags [none]
07     length: 1052 10.1.1.1 > 10.1.1.2)
08     ns3::UdpHeader (
09       length: 1032 49153 > 9)
10       Payload (size=1024)
```

A primeira linha do evento expandido (referência número 00) é a operação. Temos um caractere +, que corresponde a uma operação de enfileiramento na fila de transmissão. A segunda linha (referência 01) é o tempo da simulação em segundos. Lembre-se que pedimos ao `UdpEchoClientApplication` para iniciar o envio de pacotes depois de dois segundos (aqui podemos confirmar que isto está acontecendo).

A próxima linha do exemplo (referência 02) diz qual rastreador de origem iniciou este evento (expressado pelo *namespace* de rastreamento). Podemos pensar no *namespace* do rastreamento como algo parecido com um sistema de arquivos. A raiz do *namespace* é o `NodeList`. Este corresponde a um gerenciador de *container* no núcleo *ns-3* que contém todos os nós de rede que foram criados no código. Assim, como um sistema de arquivos pode ter diretórios dentro da raiz, podemos ter nós de rede no `NodeList`. O texto `/NodeList/0` desta forma refere-se ao nó de rede 0 (zero) no `NodeList`, ou seja é o "node 0". Em cada nós existe uma lista de dispositivos que estão instalados nestes nós de rede. Esta lista aparece depois do *namespace*. Podemos ver que este evento de rastreamento vem do `DeviceList/0` que é o dispositivo 0 instalado neste nó.

O próximo texto, `$ns3::PointToPointNetDevice` informa qual é o tipo de dispositivo na posição zero da lista de dispositivos para o nó 0 (*node 0*). Lembre-se que a operação + significa que uma operação de enfileiramento está acontecendo na fila de transmissão do dispositivo. Isto reflete no segmento final do caminho de rastreamento, que são `TxQueue/Enqueue`.

As linhas restantes no rastreamento devem ser intuitivas. As referências 03-04 indicam que o pacote é encapsulado pelo protocolo ponto-a-ponto. Referencias 05-07 mostram que foi usado o cabeçalho do IP na versão 4, o endereço IP de origem é o 10.1.1.1 e o destino é o 10.1.1.2. As referências 08-09 mostram que o pacote tem um cabeçalho UDP e finalmente na referência 10 é apresentado que a área de dados possui 1024 bytes.

A próxima linha do arquivo de rastreamento mostra que o mesmo pacote inicia o desenfileiramento da fila de transmissão do mesmo nó de rede.

A terceira linha no arquivo mostra o pacote sendo recebido pelo dispositivo de rede no nó que representa o servidor de eco. Reproduzimos o evento a seguir.

```
00 r
01 2.25732
02 /NodeList/1/DeviceList/0/$ns3::PointToPointNetDevice/MacRx
03   ns3::Ipv4Header (
04     tos 0x0 ttl 64 id 0 protocol 17 offset 0 flags [none]
05     length: 1052 10.1.1.1 > 10.1.1.2)
06     ns3::UdpHeader (
07       length: 1032 49153 > 9)
08       Payload (size=1024)
```

A operação agora é o `r` e o tempo de simulação foi incrementado para 2.25732 segundos. Se você seguiu os passos do tutorial isto significa que temos o tempo padrão tanto para `DataRate` quanto para o `Delay`. Já vimos este tempo na seção anterior.

Na referência 02, a entrada para o *namespace* foi alterada para refletir o evento vindo do nó 1 (`/NodeList/1`) e o recebimento do pacote no rastreador de origem (`/MacRx`). Isto deve facilitar o acompanhamento dos pacotes através da topologia, pois basta olhar os rastros no arquivo.

5.3.2 Rastreamento PCAP

Também podemos usar o formato `.pcap` para fazer rastreamento no *ns-3*. O pcap (normalmente escrito em letras minúsculas) permite a captura de pacotes e é uma API que inclui a descrição de um arquivo no formato `.pcap`. O programa mais conhecido para ler o mostrar este formato é o Wireshark (formalmente chamado de Etherreal). Entretanto, existem muitos analisadores de tráfego que usam este formato. Nós encorajamos que os usuários explorem várias ferramentas disponíveis para análise do pcap. Neste tutorial nos concentraremos em dar uma rápida olhada no tcpdump.

O código usado para habilitar o rastreamento pcap consiste de uma linha.

```
pointToPoint.EnablePcapAll ("myfirst");
```

Insira esta linha depois do código do rastreamento ASCII, no arquivo `scratch/myfirst.cc`. Repare que passamos apenas o texto "myfirst" e não "myfirst.pcap", isto ocorre por que é um prefixo e não um nome de arquivo completo. O assistente irá criar um arquivo contendo um prefixo e o número do nó de rede, o número de dispositivo e o sufixo ".pcap".

Em nosso código, nós iremos ver arquivos chamados "myfirst-0-0.pcap" e "myfirst-1-0.pcap" que são rastreamentos pcap do dispositivo 0 do nó 0 e do dispositivo 0 do nó de rede 1, respectivamente.

Uma vez que adicionamos a linha de código que habilita o rastreamento pcap, podemos executar o código da forma habitual:

```
./waf --run scratch/myfirst
```

Se olharmos no diretório da distribuição, veremos agora três novos arquivos de registro: `myfirst.tr` que é o arquivo ASCII, que nós examinamos na seção anterior. `myfirst-0-0.pcap` e `myfirst-1-0.pcap`, que são os novos arquivos pcap gerados.

Lendo a saída com o tcpdump

A forma mais confortável de olhar os arquivos pcap é usando o `tcpdump`.

```
tcpdump -nn -tt -r myfirst-0-0.pcap
reading from file myfirst-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 1024
2.514648 IP 10.1.1.2.9 > 10.1.1.1.49153: UDP, length 1024

tcpdump -nn -tt -r myfirst-1-0.pcap
reading from file myfirst-1-0.pcap, link-type PPP (PPP)
2.257324 IP 10.1.1.1.49153 > 10.1.1.2.9: UDP, length 1024
2.257324 IP 10.1.1.2.9 > 10.1.1.1.49153: UDP, length 1024
```

Podemos ver no primeiro *dump* do arquivo `myfirst-0-0.pcap` (dispositivo cliente), que o pacote de eco é enviado com dois segundos de simulação. Olhando o segundo *dump* veremos que o pacote é recebido com 2.257324 segundos. O pacote é ecoado de volta com 2.257324 segundos e finalmente é recebido de volta pelo cliente com 2.514648 segundos.

Lendo saídas com o Wireshark

Podemos obter o Wireshark em http://www.wireshark.org/, bem como sua documentação.

O Wireshark é um programa que usa interface gráfica e pode ser usado para mostrar os arquivos de rastreamento. Com o Wireshark, podemos abrir cada arquivo de rastreamento e visualizar conteúdo como se tivéssemos capturando os pacotes usando um analisador de tráfego de redes (*packet sniffer*).

CAPÍTULO 6

Construindo topologias

6.1 Construindo uma rede em barramento

Nesta seção, o conhecimento sobre dispositivos de rede e canais de comunicação são expandidos de forma a abordar um exemplo de uma rede em barramento. O *ns-3* fornece um dispositivo de rede e canal que é chamado de CSMA (*Carrier Sense Multiple Access*).

O dispositivo CSMA modela uma rede simples no contexto da Ethernet. Uma rede Ethernet real utiliza CSMA/CD (*Carrier Sense Multiple Access with Collision Detection*) com recuo binário exponencial para lidar com o meio de transmissão compartilhado. O dispositivo e o canal CSMA do *ns-3* modelam apenas um subconjunto deste.

Assim como foi visto nos assistentes ponto-a-ponto (objetos) na construção de topologias ponto-a-ponto, veremos assistentes (objetos) equivalentes da topologia CSMA nesta seção. O formato e o funcionamento destes assistentes serão bastante familiares para o leitor.

Um novo código exemplo é fornecido na pasta `examples/tutorial`. Este baseia-se no código `first.cc` e adiciona uma rede CSMA a simulação ponto-a-ponto já considerada. O leitor pode abrir o arquivo `examples/tutorial/second.cc` em seu editor favorito para acompanhar o restante desta seção. Embora seja redundante, o código será analisado em sua totalidade, examinando alguns de seus resultados.

Assim como no exemplo `first.cc` (e em todos os exemplos ns-3), o arquivo começa com uma linha de modo Emacs e algumas linhas do padrão GPL.

O código começa com o carregamento de módulos através da inclusão dos arquivos.

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/csma-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/ipv4-global-routing-helper.h"
```

Algo que pode ser surpreendentemente útil é uma pequena arte ASCII que mostra um desenho da topologia da rede construída. Um "desenho" similar é encontrado na maioria dos exemplos no projeto.

Neste caso, é possível perceber que o exemplo ponto-a-ponto (a ligação entre os nós n0 e n1 abaixo) está sendo estendido, agregando uma rede em barramento ao lado direito. Observe que esta é a topologia de rede padrão, visto que o número de nós criados na LAN pode ser mudado. Se o atributo `nCsma` for configurado para um, haverá um total de dois nós na LAN (canal CSMA) — um nó obrigatório e um nó "extra". Por padrão, existem três nós "extra", como pode ser observado:

```
// Default Network Topology
//
//       10.1.1.0
// n0 -------------- n1   n2   n3   n4
//    point-to-point  |    |    |    |
//                    ================
//                      LAN 10.1.2.0
```

Em seguida, o *namespace* do ns-3 é *usado* e um componente de registro (*log*) é definido. Até aqui, tudo é exatamente como em `first.cc`, não há nada novo ainda.

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("SecondScriptExample");
```

O programa principal começa com um toque ligeiramente diferente. A variável 'verbose' é usada para determinar se os componentes de registro de `UdpEchoClientApplication` e `UdpEchoServerApplication` estarão habilitados. O valor padrão é verdadeiro (os componentes de registro estão ativados), mas é possível desligar durante os testes de regressão deste exemplo.

Você verá códigos familiares que lhe permitirão mudar o número de dispositivos na rede CSMA via linha de comando. Fizemos algo semelhante, quando permitimos que o número de pacotes enviados em uma sessão fosse alterado. A última linha garante que você tenha pelo menos um nó "extra".

O código consiste em variações de APIs abordadas anteriormente neste tutorial.

```
bool verbose = true;
uint32_t nCsma = 3;

CommandLine cmd;
cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
cmd.AddValue ("verbose", "Tell echo applications to log if true", verbose);

cmd.Parse (argc, argv);

if (verbose)
  {
    LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
  }

nCsma = nCsma == 0 ? 1 : nCsma;
```

O próximo passo é a criação de dois nós que iremos conectar através da ligação ponto-a-ponto. O `NodeContainer` é usado para fazer isto, assim como foi feito em `first.cc`.

```
NodeContainer p2pNodes;
p2pNodes.Create (2);
```

Em seguida, declaramos outro `NodeContainer` para manter os nós que serão parte da rede em barramento (CSMA). Primeiro, instanciamos somente o contêiner.

```
NodeContainer csmaNodes;
csmaNodes.Add (p2pNodes.Get (1));
csmaNodes.Create (nCsma);
```

Depois, na próxima linha de código, `obtém-se` o primeiro nó do contêiner ponto-a-ponto e o adiciona ao contêiner de nós que irão receber dispositivos CSMA. O nó em questão vai acabar com um dispositivo ponto-a-ponto *e* um dispositivo CSMA. Em seguida, criamos uma série de nós "extra" que compõem o restante da rede CSMA. Visto que

já temos um nó na rede CSMA – aquele que terá tanto um dispositivo ponto-a-ponto quanto um dispositivo de rede CSMA, o número de nós "extras" representa o número desejado de nós na seção CSMA menos um.

Instanciamos um `PointToPointHelper` e definimos os atributos padrões de forma a criar uma transmissão de cinco megabits por segundo e dois milésimos de segundo de atraso para dispositivos criados utilizando este assistente.

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);
```

Em seguida, instanciamos um `NetDeviceContainer` para gerenciar os dispositivos ponto-a-ponto e então `Instalamos` os dispositivos nos nós ponto-a-ponto.

Mencionamos anteriormente que abordaríamos um assistente para dispositivos e canais CSMA, as próximas linhas o introduzem. O `CsmaHelper` funciona como o `PointToPointHelper`, mas cria e conecta dispositivos e canais CSMA. No caso de um par de dispositivos e canais CSMA, observe que a taxa de dados é especificada por um atributo do canal, ao invés de um atributo do dispositivo. Isto ocorre porque uma rede CSMA real não permite que se misture, por exemplo, dispositivos 10Base-T e 100Base-T em um mesmo meio. Primeiro definimos a taxa de dados a 100 megabits por segundo e, em seguida, definimos o atraso do canal como a velocidade da luz, 6560 nano-segundos (escolhido arbitrariamente como 1 nanossegundo por 30,48 centímetros sobre um segmento de 100 metros). Observe que você pode definir um atributo usando seu tipo de dados nativo.

```
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install (csmaNodes);
```

Assim como criamos um `NetDeviceContainer` para manter os dispositivos criados pelo `PointToPointHelper`, criamos um `NetDeviceContainer` para gerenciar os dispositivos criados pelo nosso `CsmaHelper`. Chamamos o método `Install` do `CsmaHelper` para instalar os dispositivos nos nós do `csmaNodes NodeContainer`.

Agora temos os nossos nós, dispositivos e canais criados, mas não temos nenhuma pilha de protocolos presente. Assim como no exemplo `first.cc`, usaremos o `InternetStackHelper` para instalar estas pilhas.

```
InternetStackHelper stack;
stack.Install (p2pNodes.Get (0));
stack.Install (csmaNodes);
```

Lembre-se que pegamos um dos nós do contêiner `p2pNodes` e o adicionamos ao contêiner `csmaNodes`. Assim, só precisamos instalar as pilhas nos nós `p2pNodes` restantes e todos os nós do contêiner `csmaNodes` para abranger todos os nós na simulação.

Assim como no exemplo `first.cc`, vamos usar o `Ipv4AddressHelper` para atribuir endereços IP para as interfaces de nossos dispositivos. Primeiro, usamos a rede 10.1.1.0 para criar os dois endereços necessários para os dispositivos ponto-a-ponto.

```
Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);
```

Lembre-se que salvamos as interfaces criadas em um contêiner para tornar mais fácil a obtenção de informações sobre o endereçamento para uso na criação dos aplicativos.

Precisamos agora atribuir endereços IP às interfaces dos dispositivo CSMA. A operação é a mesma realizada para o ponto-a-ponto, exceto que agora estamos realizando a operação em um contêiner que possui um número variável de dispositivos CSMA — lembre-se que fizemos o número de dispositivos CSMA serem passados na linha de comando. Os dispositivos CSMA serão associados com endereços IP da rede 10.1.2.0, como visto a seguir.

```
address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);
```

Agora a topologia já está construída, mas precisamos de aplicações. Esta seção é muito similar a seção de aplicações do exemplo `first.cc`, mas vamos instanciar o servidor em um dos nós que tem um dispositivo CSMA e o cliente em um nó que tem apenas um dispositivo ponto-a-ponto.

Primeiro, vamos configurar o servidor de eco. Criamos um `UdpEchoServerHelper` e fornecemos o atributo obrigatório do construtor que é o número da porta. Lembre-se que esta porta pode ser alterada posteriormente, utilizando o método `SetAttribute`.

```
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

Lembre-se que o `csmaNodes NodeContainer` contém um dos nós criados para a rede ponto-a-ponto e os `nCsma` nós "extra". O que queremos é o último dos nós "extra". A entrada zero do contêiner `csmaNodes` será o nó ponto-a-ponto. O jeito fácil de pensar nisso é, ao criar um nó CSMA "extra", este será o nó um do contêiner `csmaNodes`. Por indução, se criarmos `nCsma` nós "extra", o último será o de índice `nCsma`. Isto ocorre no `Get` da primeira linha de código.

A aplicação cliente é criada exatamente como fizemos no exemplo `first.cc`. Novamente, fornecemos os atributos necessários no construtor do `UdpEchoClientHelper` (neste caso, o endereço e porta remotos). Dizemos ao cliente para enviar pacotes para o servidor. Instalamos o cliente no nó ponto-a-ponto mais à esquerda visto na ilustração da topologia.

```
UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps = echoClient.Install (p2pNodes.Get (0));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

Visto que construímos uma inter-rede, precisamos de alguma forma de roteamento. O *ns-3* fornece o que chamamos de roteamento global para simplificar essa tarefa. O roteamento global tira proveito do fato de que toda a inter-rede é acessível na simulação — ele realiza a disponibilização do roteamento sem a necessidade de configurar roteadores individualmente.

Basicamente, o que acontece é que cada nó se comporta como se fosse um roteador OSPF que se comunica instantaneamente e magicamente com todos os outros roteadores transparentemente. Cada nó gera anúncios de ligações e os comunica diretamente a um gerente de rota global. O gerente, por sua vez, utiliza esta informação para construir as tabelas de roteamento de cada nó. A configuração deste tipo de roteamento é realizada em uma linha:

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Em seguida, vamos habilitar o rastreamento pcap. A primeira linha de código para habilita o rastreamento pcap no assistente ponto-a-ponto. A segunda linha habilita o rastreamento pcap no assistente CSMA e há um parâmetro extra que ainda não havíamos usado.

```
pointToPoint.EnablePcapAll ("second");
csma.EnablePcap ("second", csmaDevices.Get (1), true);
```

A rede CSMA é uma rede multi-ponto-a-ponto. Isto significa que pode (e neste caso, de fato há) vários nós em um meio compartilhado. Cada um destes nós tem um dispositivo de rede associado. Existem duas alternativas para a coleta de informações de rastreamento em uma rede desse tipo. Uma maneira é criar um arquivo de rastreamento para cada dispositivo de rede e armazenar apenas os pacotes que são enviados ou recebidos por esse dispositivo. Outra maneira é escolher um dos dispositivos e colocá-lo em modo promíscuo. Esse dispositivo então "sniffs" a rede por todos os pacotes e os armazena em um único arquivo pcap. Isto é como o `tcpdump` funciona, por exemplo. O último parâmetro informa ao assistente CSMA se deve ou não capturar pacotes em modo promíscuo.

Neste exemplo, vamos selecionar um dos dispositivos CSMA e pedir para realizar uma captura promíscua na rede, emulando, assim, o que o `tcpdump` faria. Se você estivesse em uma máquina Linux faria algo como `tcpdump -i eth0` para obter o rastreamento. Neste caso, especificamos o dispositivo usando `csmaDevices.Get(1)`, que seleciona o primeiro dispositivo no contêiner. Configurando o último parâmetro para verdadeiro habilita a captura no modo promíscuo.

A última seção do código apenas executa e limpa a simulação como no exemplo `first.cc`.

```
  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

Para executar este exemplo, copie o arquivo de `second.cc` para o diretório "scratch" e use o comando waf para compilar exatamente como você fez com `first.cc`. Se você estiver no diretório raiz do repositório, digite,

```
cp examples/tutorial/second.cc scratch/mysecond.cc
./waf
```

Atenção: Usamos o arquivo `second.cc` como um dos nossos testes de regressão para verificar se ele funciona exatamente como achamos que deve, a fim de fazer o seu tutorial uma experiência positiva. Isto significa que um executável chamado `second` já existe no projeto. Para evitar qualquer confusão sobre o que você está executando, renomeie para `mysecond.cc` como sugerido acima.

Se você está seguindo o tutorial religiosamente (você está? certo?), ainda vai ter a variável `NS_LOG` definida, então limpe a variável e execute o programa.

```
export NS_LOG=
./waf --run scratch/mysecond
```

Uma vez que configuramos aplicações UDP de eco para rastrear, assim como fizemos em `first.cc`, você verá uma saída semelhante quando executar o código.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.415s)
Sent 1024 bytes to 10.1.2.4
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.2.4
```

Lembre-se que a primeira mensagem, "`Sent 1024 bytes to 10.1.2.4`," é o cliente UDP enviando um pacote de eco para o servidor. Neste caso, o servidor está em uma rede diferente (10.1.2.0). A segunda mensagem, "`Received 1024 bytes from 10.1.1.1`," é do servidor de eco, gerado quando ele recebe o pacote de eco. A mensagem final, "`Received 1024 bytes from 10.1.2.4`," é do cliente de eco, indicando que ele recebeu seu eco de volta.

Se você olhar no diretório raiz, encontrará três arquivos de rastreamento:

```
second-0-0.pcap  second-1-0.pcap  second-2-0.pcap
```

Vamos gastar um tempo para ver a nomeação desses arquivos. Todos eles têm a mesma forma, <nome>-<nó>-<dispositivo>.pcap. Por exemplo, o primeiro arquivo na listagem é second-0-0.pcap `` que é o rastreamento pcap do nó zero, dispositivo zero. Este é o dispositivo na rede ponto-a-ponto no nó zero. O arquivo ``second-1-0.pcap é o rastreamento pcap para o dispositivo zero no nó um, também um dispositivo ponto-a-ponto. O arquivo second-2-0.pcap é o rastreamento pcap para o dispositivo zero no nó dois.

Se remetermos para a ilustração da topologia no início da seção, vai ver que o nó zero é o nó mais à esquerda da ligação ponto-a-ponto e o nó um é o nó que tem tanto um dispositivo ponto-a-ponto quanto um CSMA. Observamos que o nó dois é o primeiro nó "extra" na rede CSMA e seu dispositivo zero foi selecionado como o dispositivo para capturar pacotes de modo promíscuo.

Agora, vamos seguir o pacote de eco através das redes. Primeiro, faça um tcpdump do arquivo de rastreamento para o nó ponto-a-ponto mais à esquerda — nó zero.

```
tcpdump -nn -tt -r second-0-0.pcap
```

Teremos o conteúdo do arquivo pcap:

```
reading from file second-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.007602 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

A primeira linha do despejo (*dump*) indica que o tipo da ligação é PPP (ponto-a-ponto). Você então vê o pacote de eco deixando o nó zero através do dispositivo associado com o endereço IP 10.1.1.1, destinado para o endereço IP 10.1.2.4 (o nó CSMA mais à direita). Este pacote vai passar pela ligação ponto-a-ponto e será recebido pelo dispositivo ponto-a-ponto no nó um. Vamos dar uma olhada:

```
tcpdump -nn -tt -r second-1-0.pcap
```

Observamos agora a saída de rastreamento pcap do outro lado da ligação ponto-a-ponto:

```
reading from file second-1-0.pcap, link-type PPP (PPP)
2.003686 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Aqui vemos que o tipo de ligação também é PPP. Vemos nesta interface o pacote do endereço IP 10.1.1.1 (que foi enviado a 2,000000 segundos) endereçado ao IP 10.1.2.4. Agora, internamente a este nó, o pacote será enviado para a interface CSMA e devemos vê-lo saindo nesse dispositivo a caminho de seu destino final.

Lembre-se que selecionamos o nó 2 como o "sniffer" promíscuo para a rede CSMA, por isso, vamos analisar o arquivo second-2-0.pcap.

```
tcpdump -nn -tt -r second-2-0.pcap
```

Temos agora o despejo do nó dois, dispositivo zero:

```
reading from file second-2-0.pcap, link-type EN10MB (Ethernet)
2.003696 arp who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003707 arp reply 10.1.2.4 is-at 00:00:00:00:00:06
2.003801 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.003811 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4
2.003822 arp reply 10.1.2.1 is-at 00:00:00:00:00:03
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Observamos que o tipo de ligação agora é "Ethernet". Algo novo apareceu. A rede em barramento necessicita do ARP, o "Address Resolution Protocol". O nó um sabe que precisa enviar o pacote para o endereço IP 10.1.2.4, mas não sabe o endereço MAC do nó correspondente. Ele transmite na rede CSMA (ff:ff:ff:ff:ff:ff) pedindo ao dispositivo que tem o

endereço IP 10.1.2.4. Neste caso, o nó mais à direita responde dizendo que está no endereço MAC 00:00:00:00:00:06. Note que o nó dois não está diretamente envolvido nesta troca, mas está capturando todo o tráfego da rede.

Esta troca é vista nas seguintes linhas,

```
2.003696 arp who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003707 arp reply 10.1.2.4 is-at 00:00:00:00:00:06
```

Em seguida, o nó um, dispositivo um, envia o pacote de eco UDP para o servidor no endereço IP 10.1.2.4.

```
2.003801 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
```

O servidor recebe a solicitação de eco e tenta enviar de volta para a origem. O servidor sabe que este endereço está em outra rede que chega através do endereço IP 10.1.2.1. Isto porque inicializamos o roteamento global. Entretanto, o nó servidor de eco não sabe o endereço MAC do primeiro nó CSMA, por isso tem que solicitar via ARP assim como o primeiro nó CSMA teve que fazer.

```
2.003811 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4
2.003822 arp reply 10.1.2.1 is-at 00:00:00:00:00:03
```

O servidor então envia o eco de volta ao nó de encaminhamento.

```
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Analisando o nó mais à direita da ligação ponto-a-ponto,

```
tcpdump -nn -tt -r second-1-0.pcap
```

Observamos o pacote que ecoou vindo de volta para a lingação ponto-a-ponto na última linha do despejo.

```
reading from file second-1-0.pcap, link-type PPP (PPP)
2.003686 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.003915 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Finalmente, analisando o nó que originou o eco,

```
tcpdump -nn -tt -r second-0-0.pcap
```

vericamos que o pacote eco chega de volta na fonte em 2,007602 segundos

```
reading from file second-0-0.pcap, link-type PPP (PPP)
2.000000 IP 10.1.1.1.49153 > 10.1.2.4.9: UDP, length 1024
2.007602 IP 10.1.2.4.9 > 10.1.1.1.49153: UDP, length 1024
```

Finalmente, lembre-se que adicionamos a habilidade de controlar o número de dispositivos CSMA na simulação por meio da linha de comando. Você pode alterar esse argumento da mesma forma como quando alteramos o número de pacotes de eco no exemplo `first.cc`. Tente executar o programa com o número de dispositivos "extra" em quatro:

```
./waf --run "scratch/mysecond --nCsma=4"
```

Você deve ver agora:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.405s)
Sent 1024 bytes to 10.1.2.5
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.2.5
```

Observe que o servidor de eco foi agora transferido para o último dos nós CSMA, que é 10.1.2.5 em vez de o caso padrão, 10.1.2.4.

É possível que você não se satisfaça com um arquivo de rastreamento gerado por um espectador na rede CSMA. Você pode querer obter o rastreamento de um único dispositivo e pode não estar interessado em qualquer outro tráfego na rede. Você pode fazer isso facilmente.

Vamos dar uma olhada em `scratch/mysecond.cc` e adicionar o código permitindo-nos ser mais específicos. Os assistentes do `ns-3` fornecem métodos que recebem um número de nó e um número de dispositivo como parâmetros. Substitua as chamadas `EnablePcap` pelas seguites:

```
pointToPoint.EnablePcap ("second", p2pNodes.Get (0)->GetId (), 0);
csma.EnablePcap ("second", csmaNodes.Get (nCsma)->GetId (), 0, false);
csma.EnablePcap ("second", csmaNodes.Get (nCsma-1)->GetId (), 0, false);
```

Sabemos que queremos criar um arquivo pcap com o nome base "second" e sabemos também que o dispositivo de interesse em ambos os casos vai ser o zero, então estes parâmetros não são interessantes.

A fim de obter o número do nó, você tem duas opções: primeiro, os nós são numerados de forma crescente a partir de zero na ordem em que você os cria. Uma maneira de obter um número de nó é descobrir este número "manualmente" através da ordem de criação. Se olharmos na ilustração da topologia da rede no início do arquivo, perceberemos que foi o que fizemos. Isto pode ser visto porque o último nó CSMA vai ser o de número `nCsma + 1`. Esta abordagem pode tornar-se muito difícil em simulações maiores.

Uma maneira alternativa, que usamos aqui, é perceber que os `NodeContainers` contêm ponteiros para objetos `Node` do *ns-3*. O objeto `Node` tem um método chamado `GetId` que retornará o ID do nó, que é o número do nó que buscamos. Vamos dar uma olhada por `Node` no Doxygen e localizar esse método, que está mais abaixo no código do núcleo do que vimos até agora. Às vezes você tem que procurar diligentemente por coisas úteis.

Consulte a documentação em Doxygen para a sua distribuição do ns (lembre-se que você pode encontrá-la no site do projeto). Você pode chegar a documentação sobre o objeto `Node` procurando pela guia "Classes", até encontrar `ns3::Node` na "Class List". Selecione `ns3::Node` e você será levado a documentação para a classe `Node`. Se você ir até o método `GetId` e selecioná-lo, será levado a documentação detalhada do método. Usar o método `getId` pode tornar muito mais fácil determinar os números dos nós em topologias complexas.

Vamos limpar os arquivos de rastreamento antigos do diretório raiz para evitar confusão sobre o que está acontecendo,

```
rm *.pcap
rm *.tr
```

Se você compilar o novo código e executar a simulação com `nCsma` em 100,

```
./waf --run "scratch/mysecond --nCsma=100"
```

você vai observar a seguinte saída:

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.407s)
Sent 1024 bytes to 10.1.2.101
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.2.101
```

Observe que o servidor de eco está agora em 10.1.2.101, que corresponde a ter 100 nós CSMA "extras" com o servidor de eco no último. Se você listar os arquivos pcap no diretório principal, você verá,

```
second-0-0.pcap  second-100-0.pcap  second-101-0.pcap
```

O arquivo de rastreamento `second-0-0.pcap` é o dispositivo ponto-a-ponto "mais à esquerda" que é a origem do pacote de eco. O arquivo `second-101-0.pcap` corresponde ao dispositivo CSMA mais à direita que é onde o servidor de eco reside. O leitor deve ter notado que o parâmetro final na chamada para ativar o rastreamento no nó servidor era falso. Isto significa que o rastreamento nesse nó estava em modo não-promíscuo.

Para ilustrar a diferença entre o rastreamento promíscuo e o não promíscuo, também solicitamos um rastreamento não-promíscuo para o nó vizinho ao último. Dê uma olhada no `tcpdump` para `second-100-0.pcap`.

```
tcpdump -nn -tt -r second-100-0.pcap
```

Agora observamos que o nó 100 é realmente um espectador na troca de eco. Os únicos pacotes que ele recebe são os pedidos ARP que são transmitidos para a rede CSMA inteira (em broadcast).

```
reading from file second-100-0.pcap, link-type EN10MB (Ethernet)
2.003696 arp who-has 10.1.2.101 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003811 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.101
```

Agora, dê uma olhada no `tcpdump` para `second-101-0.pcap`.

```
tcpdump -nn -tt -r second-101-0.pcap
```

Observamos que o nó 101 é realmente o participante na troca de eco.

```
reading from file second-101-0.pcap, link-type EN10MB (Ethernet)
2.003696 arp who-has 10.1.2.101 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1
2.003696 arp reply 10.1.2.101 is-at 00:00:00:00:00:67
2.003801 IP 10.1.1.1.49153 > 10.1.2.101.9: UDP, length 1024
2.003801 arp who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.101
2.003822 arp reply 10.1.2.1 is-at 00:00:00:00:00:03
2.003822 IP 10.1.2.101.9 > 10.1.1.1.49153: UDP, length 1024
```

6.2 Modelos, atributos e a `sua` realidade

Este é um local conveniente para fazer uma pequena excursão e fazer uma observação importante. Pode ou não ser óbvio para o leitor, mas sempre que alguém está usando uma simulação, é importante entender exatamente o que está sendo modelado e o que não está. É tentador, por exemplo, pensar nos dispositivos e canais CSMA utilizados na seção anterior como se fossem dispositivos Ethernet reais, e esperar um resultado que vai refletir diretamente o que aconteceria em uma Ethernet real. Este não é o caso.

Um modelo é, por definição, uma abstração da realidade. Em última análise, é responsabilidade do autor do código da simulação determinar a chamada "faixa de precisão" e "domínio de aplicabilidade" da simulação como um todo e, portanto, suas partes constituintes.

Em alguns casos, como no `Csma`, pode ser bastante fácil de determinar o que é *não* modelado. Ao ler a descrição do fonte (`csma.h`) você descobrirá que não há detecção de colisão e poderá decidir sobre quão aplicável a sua utilização será em sua simulação ou quais ressalvas pode querer incluir em seus resultados. Em outros casos, pode ser razoavelmente fácil configurar comportamentos que podem não existir em qualquer equipamento real que possa ser comprado por aí. Vale a pena gastar algum tempo investigando tais casos e quão facilmente pode-se desviar para fora dos limites da realidade em suas simulações.

Como você viu, o *ns-3* fornece atributos que um usuário pode facilmente configurar para mudar o comportamento do modelo. Considere dois dos `atributos` do `CsmaNetDevice`: `Mtu` e `EncapsulationMode`. O atributo `Mtu` indica a unidade máxima de transmissão para o dispositivo. Este é o tamanho máximo do Protocol Data Unit (PDU) que o dispositivo pode enviar.

O valor padrão para o MTU é 1500 bytes na `CsmaNetDevice`. Este padrão corresponde a um número encontrado na RFC 894, "Um padrão para a transmissão de datagramas IP sobre redes Ethernet". O número é derivado do tamanho máximo do pacote para redes 10Base5 (full-spec Ethernet) – 1518 bytes. Se você subtrair a sobrecarga de encapsulamento DIX para pacotes Ethernet (18 bytes), você vai acabar com o tamanho máximo possível de dados (MTU) de 1500 bytes. Pode-se também encontrar que o `MTU` para redes IEEE 802.3 é 1492 bytes. Isto é porque o encapsulamento LLC/SNAP acrescenta oito bytes extra de sobrecarga para o pacote. Em ambos os casos, o hardware subjacente pode enviar apenas 1518 bytes, mas o tamanho dos dados é diferente.

A fim de definir o modo de encapsulamento, o `CsmaNetDevice` fornece um atributo chamado `EncapsulationMode` que pode assumir os valores `Dix` ou `Llc`. Estes correspondem ao enquadramento Ethernet e LLC/SNAP, respectivamente.

Se deixarmos o `Mtu` com 1500 bytes e mudarmos o encapsulamento para `Llc`, o resultado será uma rede que encapsula PDUs de 1500 bytes com enquadramento LLC/SNAP, resultando em pacotes de 1526 bytes, o que seria ilegal em muitas redes, pois elas podem transmitir um máximo de 1518 bytes por pacote. Isto resultaria em uma simulação que, de maneira sutil, não refletiria a realidade que você possa estar esperando.

Só para complicar o cenário, existem quadros jumbo (1500 < MTU <= 9000 bytes) e quadros super-jumbo (MTU > 9000 bytes) que não são oficialmente especificados pela IEEE, mas estão disponíveis em alguns equipamentos de redes de alta velocidade (Gigabit) e NICs. Alguém poderia deixar o encapsulamento em `Dix`, e definir o atributo `Mtu` em um dispositivo `CsmaNetDevice` para 64000 bytes – mesmo que o atributo `CsmaChannel DataRate` associado esteja fixado em 10 megabits por segundo. Isto modelaria um equipamento Ethernet 10Base5, dos anos 80, suportando quadros super-jumbo. Isto é certamente algo que nunca foi feito, nem é provável que alguma vez seja feito, mas é bastante fácil de configurar.

No exemplo anterior, usamos a linha de comando para criar uma simulação que tinha 100 nós `CSMA`. Poderíamos ter facilmente criado uma simulação com 500 nós. Se fossemos de fato modelar uma rede com conectores de pressão *(vampire-taps)* 10Base5, o comprimento máximo do cabo Ethernet é 500 metros, com um espaçamento mínimo entre conectores de 2,5 metros. Isso significa que só poderia haver 200 máquinas em uma rede real. Poderíamos facilmente construir uma rede ilegal desta maneira também. Isto pode ou não resultar em uma simulação significativa dependendo do que você está tentando modelar.

Situações similares podem ocorrer em muitos momentos no *ns-3*, assim como em qualquer simulador. Por exemplo, você pode posicionar os nós de tal forma que ocupem o mesmo espaço ao mesmo tempo ou você pode ser capaz de configurar amplificadores ou níveis de ruído que violam as leis básicas da física.

O *ns-3* geralmente favorece a flexibilidade, e muitos modelos permitirão a configuração de atributos sem impor qualquer consistência ou especificação especial subjacente.

Em resumo, o importante é que o *ns-3* vai fornecer uma base super flexível para experimentações. Depende de você entender o que está requisitando ao sistema e se certificar de que as simulações tem algum significado e alguma ligação com a sua realidade.

6.3 Construindo uma rede sem fio

Nesta seção, vamos expandir ainda mais nosso conhecimento sobre dispositivos e canais do *ns-3* para cobrir um exemplo de uma rede sem fio. O *ns-3* fornece um conjunto de modelos 802.11 que tentam fornecer uma implementação precisa a nível MAC da especificação 802.11 e uma implementação "não-tão-lenta" a nível PHY da especificação 802.11a.

Assim como vimos assistentes de topologia (objetos) ponto-a-ponto e CSMA quando da construção destes modelos, veremos assistentes `Wifi` similares nesta seção. O formato e o funcionamento destes assistentes devem parecer bastante familiar ao leitor.

Fornecemos um exemplo no diretório `examples/tutorial`. Este arquivo baseia-se no código presente em `second.cc` e adiciona uma rede Wifi a ele. Vá em frente e abra `examples/tutorial/third.cc` em seu editor favorito. Você já deve ter visto código *ns-3* suficiente para entender a maior parte do que está acontecendo neste exemplo, existem algumas coisas novas, mas vamos passar por todo o código e examinar alguns dos resultados.

Assim como no exemplo `second.cc` (e em todos os exemplos ns-3), o arquivo começa com uma linha de modo emacs e algumas linhas do padrão GPL.

Dê uma olhada na arte ASCII (reproduzida abaixo) que mostra topologia de rede construída no exemplo. Você pode ver que estamos ampliando ainda mais nosso exemplo agregando uma rede sem fio do lado esquerdo. Observe que esta é uma topologia de rede padrão, pois você pode variar o número de nós criados nas redes com fio e sem fio. Assim

como no exemplo `second.cc`, se você mudar `nCsma`, ele lhe dará um número "extra" de nós CSMA. Da mesma forma, você pode definir `nWifi` para controlar quantos nós (estações) `STA` serão criados na simulação. Sempre haverá um nó `AP` (*access point*) na rede sem fio. Por padrão, existem três nós "extra" no CSMA e três nós sem fio `STA`.

O código começa pelo carregamento de módulos através da inclusão dos arquivos, assim como no exemplo `second.cc`. Há algumas novas inclusões correspondentes ao módulo Wifi e ao módulo de mobilidade que discutiremos a seguir.

```
#include "ns3/core-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/wifi-module.h"
#include "ns3/mobility-module.h"
#include "ns3/csma-module.h"
#include "ns3/internet-module.h"
```

A ilustração da topologia da rede é a seguinte:

```
// Default Network Topology
//
//   Wifi 10.1.3.0
//                 AP
//  *    *    *    *
//  |    |    |    |    10.1.1.0
// n5   n6   n7   n0 -------------- n1   n2   n3   n4
//                   point-to-point  |    |    |    |
//                                   ================
//                                     LAN 10.1.2.0
```

Observamos que estamos acrescentando um novo dispositivo de rede ao nó à esquerda da ligação ponto-a-ponto que se torna o ponto de acesso da rede sem fios. Alguns nós STA sem fio são criados para preencher a nova rede 10.1.3.0, como mostrado no lado esquerdo da ilustração.

Após a ilustração, o namespace ns-3 é *usado* e um componente de registro é definido.

```
using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("ThirdScriptExample");
```

O programa principal começa exatamente como em `second.cc`, adicionando parâmetros de linha de comando para habilitar ou desabilitar componentes de registro e para alterar o número de dispositivos criados.

```
bool verbose = true;
uint32_t nCsma = 3;
uint32_t nWifi = 3;

CommandLine cmd;
cmd.AddValue ("nCsma", "Number of \"extra\" CSMA nodes/devices", nCsma);
cmd.AddValue ("nWifi", "Number of wifi STA devices", nWifi);
cmd.AddValue ("verbose", "Tell echo applications to log if true", verbose);

cmd.Parse (argc,argv);

if (verbose)
  {
    LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
    LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);
  }
```

Assim como em todos os exemplos anteriores, o próximo passo é a criação de dois nós que irão se ligar através da ligação ponto-a-ponto.

```
NodeContainer p2pNodes;
p2pNodes.Create (2);
```

Em seguida, instanciamos um `PointToPointHelper` e definimos os atributos padrões para criar uma transmissão de cinco megabits por segundo e dois milésimos de segundo de atraso para dispositivos que utilizam este assistente. Então `instalamos` os dispositivos nos nós e o canal entre eles.

```
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

NetDeviceContainer p2pDevices;
p2pDevices = pointToPoint.Install (p2pNodes);
```

Em seguida, declaramos outro `NodeContainer` para manter os nós que serão parte da rede em barramento (CSMA).

```
NodeContainer csmaNodes;
csmaNodes.Add (p2pNodes.Get (1));
csmaNodes.Create (nCsma);
```

A próxima linha de código `obtém` o primeiro nó do contêiner ponto-a-ponto e o adiciona ao contêiner de nós que irão receber dispositivos CSMA. O nó em questão vai acabar com um dispositivo ponto-a-ponto e um dispositivo CSMA. Em seguida, criamos uma série de nós "extra" que compõem o restante da rede CSMA.

Em seguida, instanciamos um `CsmaHelper` e definimos seus atributos assim como fizemos no exemplo anterior. Criamos um `NetDeviceContainer` para gerenciar os dispositivos CSMA criados e então `instalamos` dispositivos CSMA nos nós selecionados.

```
CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6560)));

NetDeviceContainer csmaDevices;
csmaDevices = csma.Install (csmaNodes);
```

Em seguida, vamos criar os nós que farão parte da rede Wifi. Vamos criar alguns nós "estações", conforme especificado na linha de comando, e iremos usar o nó "mais à esquerda" da rede ponto-a-ponto como o nó para o ponto de acesso.

```
NodeContainer wifiStaNodes;
wifiStaNodes.Create (nWifi);
NodeContainer wifiApNode = p2pNodes.Get (0);
```

A próxima parte do código constrói os dispositivos Wifi e o canal de interligação entre esses nós. Primeiro, vamos configurar os assistentes PHY e de canal:

```
YansWifiChannelHelper channel = YansWifiChannelHelper::Default ();
YansWifiPhyHelper phy = YansWifiPhyHelper::Default ();
```

Para simplificar, este código usa a configuração padrão da camada PHY e modelos de canais que estão documentados na API doxygen para os métodos `YansWifiChannelHelper::Default` e `YansWifiPhyHelper::Default`. Uma vez que esses objetos são instanciados, criamos um objeto de canal e associamos ele ao nosso gerente de objetos da camada PHY para nos certificarmos de que todos os objetos da camada PHY criados pelo `YansWifiPhyHelper` compartilham o mesmo canal subjacente, isto é, eles compartilham o mesmo meio físico sem fio e podem comunicar-se e interferir:

```
phy.SetChannel (channel.Create ());
```

Uma vez que o assistente PHY está configurado, podemos nos concentrar na camada MAC. Aqui escolhemos trabalhar com MACs não-Qos, por isso usamos um objeto `NqosWifiMacHelper` para definir os parâmetros MAC.

```
WifiHelper wifi = WifiHelper::Default ();
wifi.SetRemoteStationManager ("ns3::AarfWifiManager");

NqosWifiMacHelper mac = NqosWifiMacHelper::Default ();
```

O método `SetRemoteStationManager` diz ao assistente o tipo de algoritmo de controle de taxa a usar. Aqui, ele está pedindo ao assistente para usar o algoritmo AARF — os detalhes estão disponíveis no Doxygen.

Em seguida, configuramos o tipo de MAC, o SSID da rede de infraestrutura, e nos certificamos que as estações não realizam sondagem ativa (active probing):

```
Ssid ssid = Ssid ("ns-3-ssid");
mac.SetType ("ns3::StaWifiMac",
  "Ssid", SsidValue (ssid),
  "ActiveProbing", BooleanValue (false));
```

Este código primeiro cria um objeto de um identificador de conjunto de serviços (SSID 802.11) que será utilizado para definir o valor do atributo "Ssid" da implementação da camada MAC. O tipo particular da camada MAC que será criado pelo assistente é especificado como sendo do tipo "ns3::StaWifiMac". O uso do assistente `NqosWifiMacHelper` irá garantir que o atributo "QosSupported" para os objetos MAC criados será falso. A combinação destas duas configurações implica que a instância MAC criada em seguida será uma estação (STA) não-QoS e não-AP, em uma infraestrutura BSS (por exemplo, uma BSS com um AP). Finalmente, o atributo "ActiveProbing" é definido como falso. Isto significa que as solicitações de sondagem não serão enviados pelos MACs criados por este assistente.

Depois que todos os parâmetros específicos das estações estão completamente configurados, tanto na camada MAC como na PHY, podemos invocar o nosso método já familiar `Instalar` para criar os dispositivos Wifi destas estações:

```
NetDeviceContainer staDevices;
staDevices = wifi.Install (phy, mac, wifiStaNodes);
```

Já configuramos o Wifi para todos nós STA e agora precisamos configurar o AP. Começamos esse processo mudando o atributo padrão `NqosWifiMacHelper` para refletir os requisitos do AP.

```
mac.SetType ("ns3::ApWifiMac",
  "Ssid", SsidValue (ssid)));
```

Neste caso, o `NqosWifiMacHelper` vai criar camadas MAC do "ns3::ApWifiMac", este último especificando que uma instância MAC configurado como um AP deve ser criado, com o tipo de assistente implicando que o atributo "QosSupported" deve ser definido como falso - desativando o suporte a Qos do tipo 802.11e/WMM nos APs criados.

As próximas linhas criam um AP que compartilha os mesmos atributos a nível PHY com as estações:

```
NetDeviceContainer apDevices;
apDevices = wifi.Install (phy, mac, wifiApNode);
```

Agora, vamos adicionar modelos de mobilidade. Queremos que os nós STA sejam móveis, vagando dentro de uma caixa delimitadora, e queremos fazer o nó AP estacionário. Usamos o `MobilityHelper` para facilitar a execução desta tarefa. Primeiro, instanciamos um objeto `MobilityHelper` e definimos alguns atributos controlando a funcionalidade de "alocação de posição".

```
MobilityHelper mobility;

mobility.SetPositionAllocator ("ns3::GridPositionAllocator",
  "MinX", DoubleValue (0.0),
  "MinY", DoubleValue (0.0),
  "DeltaX", DoubleValue (5.0),
  "DeltaY", DoubleValue (10.0),
```

```
  "GridWidth", UintegerValue (3),
  "LayoutType", StringValue ("RowFirst"));
```

Este código diz ao assistente de mobilidade para usar uma grade bidimensional para distribuir os nós STA. Sinta-se à vontade para explorar a classe `ns3::GridPositionAllocator` no Doxygen para ver exatamente o que está sendo feito.

Arranjamos os nós em uma grade inicial, mas agora precisamos dizer-lhes como se mover. Escolhemos o modelo `RandomWalk2dMobilityModel` em que os nós se movem em uma direção aleatória a uma velocidade aleatória dentro de um delimitador quadrado.

```
mobility.SetMobilityModel ("ns3::RandomWalk2dMobilityModel",
  "Bounds", RectangleValue (Rectangle (-50, 50, -50, 50)));
```

Agora dizemos ao `MobilityHelper` para instalar os modelos de mobilidade nos nós STA.

```
mobility.Install (wifiStaNodes);
```

Queremos que o ponto de acesso permaneça em uma posição fixa durante a simulação. Conseguimos isto definindo o modelo de mobilidade para este nó como `ns3::ConstantPositionMobilityModel`:

```
mobility.SetMobilityModel ("ns3::ConstantPositionMobilityModel");
mobility.Install (wifiApNode);
```

Agora temos os nossos nós, dispositivos e canais e modelos de mobilidade escolhidos para os nós Wifi, mas não temos pilhas de protocolo. Assim como já fizemos muitas vezes, usaremos o `InternetStackHelper` para instalar estas pilhas.

```
InternetStackHelper stack;
stack.Install (csmaNodes);
stack.Install (wifiApNode);
stack.Install (wifiStaNodes);
```

Assim como no exemplo `second.cc`, vamos usar o `Ipv4AddressHelper` para atribuir endereços IP para as interfaces de nossos dispositivos. Primeiro, usamos a rede 10.1.1.0 para criar os dois endereços necessários para os dois dispositivos ponto-a-ponto. Então, usamos rede 10.1.2.0 para atribuir endereços à rede CSMA e, por último, atribuir endereços da rede 10.1.3.0 para ambos os dispositivos STA e o ponto de acesso na rede sem fio.

```
Ipv4AddressHelper address;

address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer p2pInterfaces;
p2pInterfaces = address.Assign (p2pDevices);

address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer csmaInterfaces;
csmaInterfaces = address.Assign (csmaDevices);

address.SetBase ("10.1.3.0", "255.255.255.0");
address.Assign (staDevices);
address.Assign (apDevices);
```

Vamos colocar o servidor de eco no nó “mais à direita” na ilustração no início do arquivo.

```
UdpEchoServerHelper echoServer (9);

ApplicationContainer serverApps = echoServer.Install (csmaNodes.Get (nCsma));
serverApps.Start (Seconds (1.0));
serverApps.Stop (Seconds (10.0));
```

E colocamos o cliente de eco no último nó STA criado, apontando-o para o servidor na rede CSMA.

```
UdpEchoClientHelper echoClient (csmaInterfaces.GetAddress (nCsma), 9);
echoClient.SetAttribute ("MaxPackets", UintegerValue (1));
echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.)));
echoClient.SetAttribute ("PacketSize", UintegerValue (1024));

ApplicationContainer clientApps =
  echoClient.Install (wifiStaNodes.Get (nWifi - 1));
clientApps.Start (Seconds (2.0));
clientApps.Stop (Seconds (10.0));
```

Uma vez que construímos uma inter-rede aqui, precisamos ativar o roteamento inter-redes, assim como fizemos no exemplo `second.cc`.

```
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
```

Algo que pode surpreender alguns usuários é o fato de que a simulação que acabamos de criar nunca vai parar "naturalmente". Isto acontece porque pedimos para o ponto de acesso gerar *beacons*. Ele irá gerar *beacons* para sempre, e isso irá resultar em eventos sendo escalonados no futuro indefinidamente, por isso devemos dizer para o simulador parar, ainda que hajam eventos de geração de *beacons* agendados. A seguinte linha de código informa ao simulador para parar, para que não simulemos *beacons* para sempre e entremos no que é essencialmente um laço sem fim.

```
Simulator::Stop (Seconds (10.0));
```

Criamos rastreamento suficiente para cobrir todas as três redes:

```
pointToPoint.EnablePcapAll ("third");
phy.EnablePcap ("third", apDevices.Get (0));
csma.EnablePcap ("third", csmaDevices.Get (0), true);
```

Estas três linhas de código irão iniciar o rastreamento do pcap em ambos os nós ponto-a-ponto que funcionam como nosso *backbone*, irão iniciar um modo promíscuo (monitor) de rastreamento na rede Wifi e na rede CSMA. Isto vai permitir ver todo o tráfego com um número mínimo de arquivos de rastreamento.

Finalmente, executamos a simulação, limpamos e, em seguida, saimos do programa.

```
  Simulator::Run ();
  Simulator::Destroy ();
  return 0;
}
```

Para executar este exemplo, você deve copiar o arquivo `third.cc` para o diretório `scratch` e usar o Waf para compilar exatamente como com o exemplo `second.cc`. Se você está no diretório raiz do repositório você deverá digitar,

```
cp examples/tutorial/third.cc scratch/mythird.cc
./waf
./waf --run scratch/mythird
```

Novamente, uma vez que configuramos aplicações de eco UDP, assim como fizemos no arquivo `second.cc`, você verá uma saída similar.

```
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone/ns-3-dev/build'
'build' finished successfully (0.407s)
Sent 1024 bytes to 10.1.2.4
Received 1024 bytes from 10.1.3.3
Received 1024 bytes from 10.1.2.4
```

Lembre-se que a primeira mensagem, `Sent 1024 bytes to 10.1.2.4`," é o cliente de eco UDP enviando um pacote para o servidor. Neste caso, o cliente está na rede wireless (10.1.3.0). A segunda mensagem, "`Received 1024 bytes from 10.1.3.3`," é do servidor de eco UDP, gerado quando este recebe o pacote de eco. A mensagem final, "`Received 1024 bytes from 10.1.2.4`," é do cliente de eco, indicando que este recebeu o seu eco de volta do servidor.

No diretório raiz, encontraremos quatro arquivos de rastreamento desta simulação, dois do nó zero e dois do nó um:

```
third-0-0.pcap  third-0-1.pcap  third-1-0.pcap  third-1-1.pcap
```

O arquivo "third-0-0.pcap" corresponde ao dispositivo ponto-a-ponto no nó zero – o lado esquerdo do *backbone*. O arquivo "third-1-0.pcap" corresponde ao dispositivo ponto-a-ponto no nó um – o lado direito do *backbone*. O arquivo "third-0-1.pcap" corresponde o rastreamento em modo promíscuo (modo monitor) da rede Wifi e o arquivo "third-1 1.pcap" ao rastreamento em modo promíscuo da rede CSMA. Você consegue verificar isto inspecionando o código?

Como o cliente de eco está na rede Wifi, vamos começar por aí. Vamos dar uma olhada para a saída de rastreamento no modo promíscuo (modo monitor) capturada nessa rede.

```
tcpdump -nn -tt -r third-0-1.pcap
```

Você deverá ver alguns conteúdos ''wifi" que não tinham antes:

```
reading from file third-0-1.pcap, link-type IEEE802_11 (802.11)
0.000025 Beacon (ns-3-ssid) [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
0.000263 Assoc Request (ns-3-ssid) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.000279 Acknowledgment RA:00:00:00:00:00:09
0.000552 Assoc Request (ns-3-ssid) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.000568 Acknowledgment RA:00:00:00:00:00:07
0.000664 Assoc Response AID(0) :: Succesful
0.001001 Assoc Response AID(0) :: Succesful
0.001145 Acknowledgment RA:00:00:00:00:00:0a
0.001233 Assoc Response AID(0) :: Succesful
0.001377 Acknowledgment RA:00:00:00:00:00:0a
0.001597 Assoc Request (ns-3-ssid) [6.0 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit]
0.001613 Acknowledgment RA:00:00:00:00:00:08
0.001691 Assoc Response AID(0) :: Succesful
0.001835 Acknowledgment RA:00:00:00:00:00:0a
0.102400 Beacon (ns-3-ssid) [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
0.204800 Beacon (ns-3-ssid) [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
0.307200 Beacon (ns-3-ssid) [6.0* 9.0 12.0 18.0 24.0 36.0 48.0 54.0 Mbit] IBSS
```

Observamos que o tipo de ligação agora é 802.11, como esperado. Você provavelmente vai entender o que está acontecendo e encontrar o pacote de pedido de eco e a resposta nesta saída de rastreamento. Vamos deixar como um exercício a análise completa da saída.

Agora, analisando o arquivo pcap do lado direito da ligação ponto-a-ponto,

```
tcpdump -nn -tt -r third-0-0.pcap
```

Novamente, temos algumas saídas familiares:

```
reading from file third-0-0.pcap, link-type PPP (PPP)
2.002160 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.009767 IP 10.1.2.4.9 > 10.1.3.3.49153: UDP, length 1024
```

Este é o pacote de eco indo da esquerda para a direita (do Wifi para o CSMA) e de volta através da ligação ponto-a-ponto.

Agora, analisando o arquivo pcap do lado direito da ligação ponto-a-ponto,

```
tcpdump -nn -tt -r third-1-0.pcap
```

Novamente, temos algumas saídas familiares:

```
reading from file third-1-0.pcap, link-type PPP (PPP)
2.005846 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.006081 IP 10.1.2.4.9 > 10.1.3.3.49153: UDP, length 1024
```

Este é o pacote de eco indo da esquerda para a direita (do Wifi para o CSMA) e depois voltando através do ligação ponto-a-ponto com tempos um pouco diferentes, como esperado.

O servidor de eco está na rede CSMA, vamos olhar para a saída de rastreamento promíscua:

```
tcpdump -nn -tt -r third-1-1.pcap
```

Temos algumas saídas familiares:

```
reading from file third-1-1.pcap, link-type EN10MB (Ethernet)
2.005846 ARP, Request who-has 10.1.2.4 (ff:ff:ff:ff:ff:ff) tell 10.1.2.1, length 50
2.005870 ARP, Reply 10.1.2.4 is-at 00:00:00:00:00:06, length 50
2.005870 IP 10.1.3.3.49153 > 10.1.2.4.9: UDP, length 1024
2.005975 ARP, Request who-has 10.1.2.1 (ff:ff:ff:ff:ff:ff) tell 10.1.2.4, length 50
2.005975 ARP, Reply 10.1.2.1 is-at 00:00:00:00:00:03, length 50
2.006081 IP 10.1.2.4.9 > 10.1.3.3.49153: UDP, length 1024
```

Isto deve ser de fácil entendimento. Se esqueceu, volte e olhe para a discussão em `second.cc`. Este exemplo segue a mesma seqüência.

Passamos algum tempo com a criação de modelos de mobilidade para a rede sem fio e por isso seria uma vergonha encerrar sem mostrar que os nós STA estão realmente se movendo durante a simulação. Vamos fazer isto ligando o `MobilityModel` à fonte de rastreamento. Isto é apenas uma visão geral da seção que detalha o rastreamento, mas é um ótimo lugar para um exemplo.

Como mencionado na seção "Aperfeiçoando", o sistema de rastreamento é dividido em origem de rastreamento e destino de rastreamento, com funções para conectar um ao outro. Vamos usar a mudança de curso padrão do modelo de mobilidade para originar os eventos de rastreamento. Vamos precisar escrever um destino de rastreamento para se conectar a origem que irá exibir algumas informações importantes. Apesar de sua reputação como difícil, é de fato muito simples. Antes do programa principal do código `scratch/mythird.cc`, adicione a seguinte função:

```
void
CourseChange (std::string context, Ptr<const MobilityModel> model)
{
  Vector position = model->GetPosition ();
  NS_LOG_UNCOND (context <<
    " x = " << position.x << ", y = " << position.y);
}
```

Este código obtém as informações de posição do modelo de mobilidade e registra a posição x e y do nó. Vamos criar o código para que esta função seja chamada toda vez que o nó com o cliente de eco mude de posição. Fazemos isto usando a função `Config::Connect`. Adicione as seguintes linhas de código no código antes da chamada `Simulator::Run`.

```
std::ostringstream oss;
oss <<
  "/NodeList/" << wifiStaNodes.Get (nWifi - 1)->GetId () <<
  "/$ns3::MobilityModel/CourseChange";

Config::Connect (oss.str (), MakeCallback (&CourseChange));
```

O que fazemos aqui é criar uma *string* contendo o caminho do *namespace* de rastreamento do evento ao qual se deseja conectar. Primeiro, temos que descobrir qual nó que queremos usando o método `GetId` como descrito anteriormente. No caso de usar o número padrão do CSMA e dos nós de rede sem fio, este acaba sendo o nó sete e o caminho do *namespace* de rastreamento para o modelo de mobilidade seria:

```
/NodeList/7/$ns3::MobilityModel/CourseChange
```

Com base na discussão na seção de rastreamento, você pode inferir que este caminho referencia o sétimo nó na lista *NodeList* global. Ele especifica o que é chamado de um objeto agregado do tipo `ns3::MobilityModel`. O prefixo cifrão implica que o *MobilityModel* é agregado ao nó sete. O último componente do caminho significa que estamos ligando ao evento "CourseChange" desse modelo.

Fazemos uma conexão entre a origem de rastreamento no nó sete com o nosso destino de rastreamento chamando `Config::Connect` e passando o caminho do *namespace* como parâmetro. Feito isto, cada evento de mudança de curso no nó sete será capturado em nosso destino de rastreamento, que por sua vez irá imprimir a nova posição.

Se você executar a simulação, verá as mudanças de curso assim que elas ocorrerem.

```
Build finished successfully (00:00:01)
/NodeList/7/$ns3::MobilityModel/CourseChange x = 10, y = 0
/NodeList/7/$ns3::MobilityModel/CourseChange x = 9.41539, y = -0.811313
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.46199, y = -1.11303
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.52738, y = -1.46869
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.67099, y = -1.98503
/NodeList/7/$ns3::MobilityModel/CourseChange x = 5.6835, y = -2.14268
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.70932, y = -1.91689
Sent 1024 bytes to 10.1.2.4
Received 1024 bytes from 10.1.3.3
Received 1024 bytes from 10.1.2.4
/NodeList/7/$ns3::MobilityModel/CourseChange x = 5.53175, y = -2.48576
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.58021, y = -2.17821
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.18915, y = -1.25785
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.7572, y = -0.434856
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.62404, y = 0.556238
/NodeList/7/$ns3::MobilityModel/CourseChange x = 4.74127, y = 1.54934
/NodeList/7/$ns3::MobilityModel/CourseChange x = 5.73934, y = 1.48729
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.18521, y = 0.59219
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.58121, y = 1.51044
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.27897, y = 2.22677
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.42888, y = 1.70014
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.40519, y = 1.91654
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.51981, y = 1.45166
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.34588, y = 2.01523
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.81046, y = 2.90077
/NodeList/7/$ns3::MobilityModel/CourseChange x = 6.89186, y = 3.29596
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.46617, y = 2.47732
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.05492, y = 1.56579
/NodeList/7/$ns3::MobilityModel/CourseChange x = 8.00393, y = 1.25054
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.00968, y = 1.35768
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.33503, y = 2.30328
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.18682, y = 3.29223
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.96865, y = 2.66873
```

CAPÍTULO 7

Rastreamento

7.1 Introdução

Como abordado na seção Usando o Sistema de Rastreamento, o objetivo principal de uma simulação no *ns-3* é a geração de saída para estudo. Há duas estratégias básicas: usar mecanismos predefinidos de saída e processar o conteúdo para extrair informações relevantes; ou desenvolver mecanismos de saída que resultam somente ou exatamente na informação pretendida.

Usar mecanismos predefinidos de saída possui a vantagem de não necessitar modificações no *ns-3*, mas requer programação. Geralmente, as mensagens de saída do pcap ou `NS_LOG` são coletadas durante a execução da simulação e processadas separadamente por códigos (*scripts*) que usam *grep*, *sed* ou *awk* para reduzir e transformar os dados para uma forma mais simples de gerenciar. Há o custo do desenvolvimento de programas para realizar as transformações e em algumas situações a informação de interesse pode não estar contida em nenhuma das saídas, logo, a abordagem falha.

Se precisarmos adicionar o mínimo de informação para os mecanismos predefinidos de saída, isto certamente pode ser feito e se usarmos os mecanismos do *ns-3*, podemos ter nosso código adicionado como uma contribuição.

O *ns-3* fornece outro mecanismo, chamado Rastreamento (*Tracing*), que evita alguns dos problemas associados com os mecanismos de saída predefinidos. Há várias vantagens. Primeiro, redução da quantidade de dados para gerenciar (em simulações grandes, armazenar toda saída no disco pode gerar gargalos de Entrada/Saída). Segundo, o formato da saída pode ser controlado diretamente evitando o pós-processamento com códigos *sed* ou *awk*. Se desejar, a saída pode ser processada diretamente para um formato reconhecido pelo *gnuplot*, por exemplo. Podemos adicionar ganchos ("*hooks*") no núcleo, os quais podem ser acessados por outros usuários, mas que não produzirão nenhuma informação exceto que sejam explicitamente solicitados a produzir. Por essas razões, acreditamos que o sistema de rastreamento do *ns-3* é a melhor forma de obter informações fora da simulação, portanto é um dos mais importantes mecanismos para ser compreendido no *ns-3*.

7.1.1 Métodos Simples

Há várias formas de obter informação após a finalização de um programa. A mais direta é imprimir a informação na saída padrão, como no exemplo,

```
#include <iostream>
...
void
SomeFunction (void)
{
```

```
  uint32_t x = SOME_INTERESTING_VALUE;
  ...
  std::cout << "The value of x is " << x << std::endl;
  ...
}
```

Ninguém impedirá que editemos o núcleo do *ns-3* e adicionemos códigos de impressão. Isto é simples de fazer, além disso temos controle e acesso total ao código fonte do *ns-3*. Entretanto, pensando no futuro, isto não é muito interessante.

Conforme aumentarmos o número de comandos de impressão em nossos programas, ficará mais difícil tratar a grande quantidade de saídas. Eventualmente, precisaremos controlar de alguma maneira qual a informação será impressa; talvez habilitando ou não determinadas categorias de saídas, ou aumentando ou diminuindo a quantidade de informação desejada. Se continuarmos com esse processo, descobriremos depois de um tempo que, reimplementamos o mecanismo `NS_LOG`. Para evitar isso, utilize o próprio `NS_LOG`.

Como abordado anteriormente, uma maneira de obter informação de saída do *ns-3* é processar a saída do `NS_LOG`, filtrando as informações relevantes. Se a informação não está presente nos registros existentes, pode-se editar o núcleo do *ns-3* e adicionar ao fluxo de saída a informação desejada. Claro, isto é muito melhor que adicionar comandos de impressão, desde que seguindo as convenções de codificação do *ns-3*, além do que isto poderia ser potencialmente útil a outras pessoas.

Vamos analisar um exemplo, adicionando mais informações de registro ao *socket* TCP do arquivo `tcp-socket-base.cc`, para isto vamos acrescentando uma nova mensagem de registro na implementação. Observe que em `TcpSocketBase::ReceivedAck()` não existem mensagem de registro para casos sem o ACK, então vamos adicionar uma da seguinte forma:

```
/** Processa o mais recente ACK recebido */
void
TcpSocketBase::ReceivedAck (Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION (this << tcpHeader);

  // ACK Recebido. Compara o número ACK com o mais alto seqno não confirmado
  if (0 == (tcpHeader.GetFlags () & TcpHeader::ACK))
    { // Ignora se não há flag ACK
    }
  ...
```

para adicionar um novo `NS_LOG_LOGIC` na sentença apropriada:

```
/** Processa o mais recente ACK recebido */
void
TcpSocketBase::ReceivedAck (Ptr<Packet> packet, const TcpHeader& tcpHeader)
{
  NS_LOG_FUNCTION (this << tcpHeader);

  // ACK Recebido. Compara o número ACK com o mais alto seqno não confirmado
  if (0 == (tcpHeader.GetFlags () & TcpHeader::ACK))
    { // Ignora se não há flag ACK
      NS_LOG_LOGIC ("TcpSocketBase " << this << " sem flag ACK");
    }
  ...
```

Isto pode parecer simples e satisfatório a primeira vista, mas lembre-se que nós escreveremos código para adicionar ao `NS_LOG` e para processar a saída com a finalidade de isolar a informação de interesse. Isto porque o controle é limitado ao nível do componente de registro.

Se cada desenvolvedor adicionar códigos de saída para um módulo existente, logo conviveremos com a saída que outro

desenvolvedor achou interessante. É descobriremos que para obter uma pequena quantidade de informação, precisaremos produzir uma volumosa quantidade de mensagens sem nenhuma relevância (devido aos comandos de saída de vários desenvolvedores). Assim seremos forçados a gerar arquivos de registros gigantescos no disco e processá-los para obter poucas linhas de nosso interesse.

Como não há nenhuma garantia no *ns-3* sobre a estabilidade da saída do `NS_LOG`, podemos descobrir que partes do registro de saída, que dependíamos, desapareceram ou mudaram entre versões. Se dependermos da estrutura da saída, podemos encontrar outras mensagens sendo adicionadas ou removidas que podem afetar seu código de processamento.

Por estas razões, devemos considerar o uso do `std::cout` e as mensagens `NS_LOG` como formas rápidas e porém sujas de obter informação da saída no *ns-3*.

Na grande maioria dos casos desejamos ter um mecanismo estável, usando APIs que permitam acessar o núcleo do sistema e obter somente informações interessantes. Isto deve ser possível sem que exista a necessidade de alterar e recompilar o núcleo do sistema. Melhor ainda seria se um sistema notificasse o usuário quando um item de interesse fora modificado ou um evento de interesse aconteceu, pois o usuário não teria que constantemente vasculhar o sistema procurando por coisas.

O sistema de rastreamento do *ns-3* é projetado para trabalhar seguindo essas premissas e é integrado com os subsistemas de Atributos (*Attribute*) e Configuração (*Config*) permitindo cenários de uso simples.

7.2 Visão Geral

O sistema de rastreamento do *ns-3* é baseado no conceito independente origem do rastreamento e destino do rastreamento. O *ns-3* utiliza um mecanismo uniforme para conectar origens a destinos.

As origens do rastreamento (*trace source*) são entidades que podem assinalar eventos que ocorrem na simulação e fornecem acesso a dados de baixo nível. Por exemplo, uma origem do rastreamento poderia indicar quando um pacote é recebido por um dispositivo de rede e prove acesso ao conteúdo do pacote aos interessados no destino do rastreamento. Uma origem do rastreamento pode também indicar quando uma mudança de estado ocorre em um modelo. Por exemplo, a janela de congestionamento do modelo TCP é um forte candidato para uma origem do rastreamento.

A origem do rastreamento não são úteis sozinhas; elas devem ser conectadas a outras partes de código que fazem algo útil com a informação provida pela origem. As entidades que consomem a informação de rastreamento são chamadas de destino do rastreamento (*trace sinks*). As origens de rastreamento são geradores de eventos e destinos de rastreamento são consumidores. Esta divisão explícita permite que inúmeras origens de rastreamento estejam dispersas no sistema em locais que os autores do modelo acreditam ser úteis.

Pode haver zero ou mais consumidores de eventos de rastreamento gerados por uma origem do rastreamento. Podemos pensar em uma origem do rastreamento como um tipo de ligação de informação ponto-para-multiponto. Seu código buscaria por eventos de rastreamento de uma parte específica do código do núcleo e poderia coexistir com outro código que faz algo inteiramente diferente com a mesma informação.

Ao menos que um usuário conecte um destino do rastreamento a uma destas origens, nenhuma saída é produzida. Usando o sistema de rastreamento, todos conectados em uma mesma origem do rastreamento estão obtendo a informação que desejam do sistema. Um usuário não afeta os outros alterando a informação provida pela origem. Se acontecer de adicionarmos uma origem do rastreamento, seu trabalho como um bom cidadão utilizador de código livre pode permitir que outros usuários forneçam novas utilidades para todos, sem fazer qualquer modificação no núcleo do *ns-3*.

7.2.1 Um Exemplo Simples de Baixo Nível

Vamos gastar alguns minutos para entender um exemplo de rastreamento simples. Primeiramente precisamos compreender o conceito de *callbacks* para entender o que está acontecendo no exemplo.

Callbacks

O objetivo do sistema de *Callback*, no *ns-3*, é permitir a uma parte do código invocar uma função (ou método em C++) sem qualquer dependência entre módulos. Isto é utilizado para prover algum tipo de indireção – desta forma tratamos o endereço da chamada de função como uma variável. Esta variável é denominada variável de ponteiro-para-função. O relacionamento entre função e ponteiro-para-função não é tão diferente que de um objeto e ponteiro-para-objeto.

Em C, o exemplo clássico de um ponteiro-para-função é um ponteiro-para-função-retornando-inteiro (PFI). Para um PFI ter um parâmetro inteiro, poderia ser declarado como,

```
int (*pfi)(int arg) = 0;
```

O código descreve uma variável nomeada como "pfi" que é inicializada com o valor 0. Se quisermos inicializar este ponteiro com um valor significante, temos que ter uma função com uma assinatura idêntica. Neste caso, poderíamos prover uma função como,

```
int MyFunction (int arg) {}
```

Dessa forma, podemos inicializar a variável apontando para uma função:

```
pfi = MyFunction;
```

Podemos então chamar `MyFunction` indiretamente, usando uma forma mais clara da chamada,

```
int result = (*pfi) (1234);
```

É uma forma mais clara, pois é como se estivéssemos dereferenciando o ponteiro da função como dereferenciamos qualquer outro ponteiro. Tipicamente, todavia, usa-se uma forma mais curta pois o compilador sabe o que está fazendo,

```
int result = pfi (1234);
```

Esta forma é como se estivessemos chamando uma função nomeada "pfi", mas o compilador reconhece que é uma chamada indireta da função `MyFunction` por meio da variável `pfi`.

Conceitualmente, é quase exatamente como o sistema de rastreamento funciona. Basicamente, uma origem do rastreamento *é* um *callback*. Quando um destino do rastreamento expressa interesse em receber eventos de rastreamento, ela adiciona a *callback* para a lista de *callbacks* mantida internamente pela origem do rastreamento. Quando um evento de interesse ocorre, a origem do rastreamento invoca seu `operator()` provendo zero ou mais parâmetros. O `operator()` eventualmente percorre o sistema e faz uma chamada indireta com zero ou mais parâmetros.

Uma diferença importante é que o sistema de rastreamento adiciona para cada origem do rastreamento uma lista interna de *callbacks*. Ao invés de apenas fazer uma chamada indireta, uma origem do rastreamento pode invocar qualquer número de *callbacks*. Quando um destino do rastreamento expressa interesse em notificações de uma origem, ela adiciona sua própria função para a lista de *callback*.

Estando interessado em mais detalhes sobre como é organizado o sistema de *callback* no *ns-3*, leia a seção *Callback* do manual.

Código de Exemplo

Analisaremos uma implementação simples de um exemplo de rastreamento. Este código está no diretório do tutorial, no arquivo `fourth.cc`.

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
```

```
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include "ns3/object.h"
#include "ns3/uinteger.h"
#include "ns3/traced-value.h"
#include "ns3/trace-source-accessor.h"

#include <iostream>

using namespace ns3;
```

A maior parte deste código deve ser familiar, pois como já abordado, o sistema de rastreamento faz uso constante dos sistemas Objeto (*Object*) e Atributos (*Attribute*), logo é necessário incluí-los. As duas primeiras inclusões (*include*) declaram explicitamente estes dois sistemas. Poderíamos usar o cabeçalho (*header*) do módulo núcleo, este exemplo é simples.

O arquivo `traced-value.h` é uma declaração obrigatória para rastreamento de dados que usam passagem por valor. Na passagem por valor é passada uma cópia do objeto e não um endereço. Com a finalidade de usar passagem por valor, precisa-se de um objeto com um construtor de cópia associado e um operador de atribuição. O conjunto de operadores predefinidos para tipos de dados primitivos (*plain-old-data*) são ++, —, +, ==, etc.

Isto significa que somos capazes de rastrear alterações em um objeto C++ usando estes operadores.

Como o sistema de rastreamento é integrado com Atributos e este trabalham com Objetos, deve obrigatoriamente existir um `Object` *ns-3* para cada origem do rastreamento. O próximo código define e declara um Objeto.

```
class MyObject : public Object
{
public:
  static TypeId GetTypeId (void)
  {
    static TypeId tid = TypeId ("MyObject")
      .SetParent (Object::GetTypeId ())
      .AddConstructor<MyObject> ()
      .AddTraceSource ("MyInteger",
                       "An integer value to trace.",
                       MakeTraceSourceAccessor (&MyObject::m_myInt))
      ;
    return tid;
  }

  MyObject () {}
  TracedValue<int32_t> m_myInt;
};
```

As duas linhas mais importantes com relação ao rastreamento são `.AddTraceSource` e a declaração `TracedValue` do `m_myInt`.

O método `.AddTraceSource` provê a "ligação" usada para conectar a origem do rastreamento com o mundo externo, por meio do sistema de configuração. A declaração `TracedValue` provê a infraestrutura que sobrecarrega os operadores abordados anteriormente e gerencia o processo de *callback*.

```
void
IntTrace (int32_t oldValue, int32_t newValue)
{
  std::cout << "Traced " << oldValue << " to " << newValue << std::endl;
}
```

Esta é a definição do destino do rastreamento. Isto corresponde diretamente a função de *callback*. Uma vez que está conectada, esta função será chamada sempre que um dos operadores sobrecarregados de `TracedValue` é executado.

Nós temos a origem e o destino do rastreamento. O restante é o código para conectar a origem ao destino.

```
int
main (int argc, char *argv[])
{
  Ptr<MyObject> myObject = CreateObject<MyObject> ();
  myObject->TraceConnectWithoutContext ("MyInteger", MakeCallback(&IntTrace));

  myObject->m_myInt = 1234;
}
```

Criamos primeiro o Objeto no qual está a origem do rastreamento.

No próximo passo, o `TraceConnectWithoutContext` conecta a origem ao destino do rastreamento. Observe que a função `MakeCallback` cria o objeto *callback* e associa com a função `IntTrace`. `TraceConnectWithoutContext` faz a associação entre a sua função e o `operator()`, sobrecarregado a variável rastreada referenciada pelo Atributo `"MyInteger"`. Depois disso, a origem do rastreamento "disparará" sua função de callback.

O código para fazer isto acontecer não é trivial, mas a essência é a mesma que se a origem do rastreamento chamasse a função `pfi()` do exemplo anterior. A declaração `TracedValue<int32_t> m_myInt;` no Objeto é responsável pela mágica dos operadores sobrecarregados que usarão o `operator()` para invocar o *callback* com os parâmetros desejados. O método `.AddTraceSource` conecta o *callback* ao sistema de configuração, e `TraceConnectWithoutContext` conecta sua função a fonte de rastreamento, a qual é especificada por um nome Atributo.

Vamos ignorar um pouco o contexto.

Finalmente a linha,

```
myObject->m_myInt = 1234;
```

deveria ser interpretada como uma invocação do operador = na variável membro `m_myInt` com o inteiro `1234` passado como parâmetro.

Por sua vez este operador é definido (por `TracedValue`) para executar um *callback* que retorna `void` e possui dois inteiros como parâmetros — um valor antigo e um novo valor para o inteiro em questão. Isto é exatamente a assinatura da função para a função de *callback* que nós fornecemos — `IntTrace`.

Para resumir, uma origem do rastreamento é, em essência, uma variável que mantém uma lista de *callbacks*. Um destino do rastreamento é uma função usada como alvo da *callback*. O Atributo e os sistemas de informação de tipo de objeto são usados para fornecer uma maneira de conectar origens e destinos do rastreamento. O ação de "acionar" uma origem do rastreamento é executar um operador na origem, que dispara os *callbacks*. Isto resulta na execução das *callbacks* dos destinos do rastreamento registrados na origem com os parâmetros providos pela origem.

Se compilarmos e executarmos este exemplo,

```
./waf --run fourth
```

observaremos que a saída da função `IntTrace` é processada logo após a execução da origem do rastreamento:

```
Traced 0 to 1234
```

Quando executamos o código, `myObject->m_myInt = 1234;` a origem do rastreamento disparou e automaticamente forneceu os valores anteriores e posteriores para o destino do rastreamento. A função `IntTrace` então imprimiu na saída padrão, sem maiores problemas.

7.2.2 Usando o Subsistema de Configuração para Conectar as Origens de Rastreamento

A chamada `TraceConnectWithoutContext` apresentada anteriormente é raramente usada no sistema. Geralmente, o subsistema `Config` é usado para selecionar uma origem do rastreamento no sistema usando um caminho de configuração (*config path*). Nós estudamos um exemplo onde ligamos o evento "CourseChange", quando estávamos brincando com `third.cc`.

Nós definimos um destino do rastreamento para imprimir a informação de mudança de rota dos modelos de mobilidade de nossa simulação. Agora está mais claro o que está função realizava.

```
void
CourseChange (std::string context, Ptr<const MobilityModel> model)
{
  Vector position = model->GetPosition ();
  NS_LOG_UNCOND (context <<
    " x = " << position.x << ", y = " << position.y);
}
```

Quando conectamos a origem do rastreamento "CourseChange" para o destino do rastreamento anteriormente, usamos o que é chamado de caminho de configuração ("*Config Path*") para especificar a origem e o novo destino do rastreamento.

```
std::ostringstream oss;
oss <<
  "/NodeList/" << wifiStaNodes.Get (nWifi - 1)->GetId () <<
  "/$ns3::MobilityModel/CourseChange";

Config::Connect (oss.str (), MakeCallback (&CourseChange));
```

Para entendermos melhor o código, suponha que o número do nó retornado por `GetId()` é "7". Neste caso, o caminho seria,

```
"/NodeList/7/$ns3::MobilityModel/CourseChange"
```

O último segmento de um caminho de configuração deve ser um Atributo de um Objeto. Na verdade, se tínhamos um ponteiro para o Objeto que tem o Atributo "CourseChange" , `poderíamos escrever como no exemplo anterior. Nós já sabemos que guardamos tipicamente ponteiros para outros nós em um ''NodeContainer.` No exemplo `third.cc`, os nós de rede de interesse estão armazenados no `wifiStaNodes NodeContainer`. De fato enquanto colocamos o caminho junto usamos este contêiner para obter um `Ptr<Node>`, usado na chamada `GetId()`. Poderíamos usar diretamente o `Ptr<Node>` para chamar um método de conexão.

```
Ptr<Object> theObject = wifiStaNodes.Get (nWifi - 1);
theObject->TraceConnectWithoutContext ("CourseChange", MakeCallback (&CourseChange));
```

No exemplo `third.cc`, queremos um "contexto" adicional para ser encaminhado com os parâmetros do *callback* (os quais são explicados a seguir) então podemos usar o código equivalente,

```
Ptr<Object> theObject = wifiStaNodes.Get (nWifi - 1);
theObject->TraceConnect ("CourseChange", MakeCallback (&CourseChange));
```

Acontece que o código interno para `Config::ConnectWithoutContext` e `Config::Connect` permite localizar um Ptr<Object> e chama o método `TraceConnect`, no nível mais baixo.

As funções `Config` aceitam um caminho que representa uma cadeia de ponteiros de Objetos. Cada segmento do caminho corresponde a um Atributo Objeto. O último segmento é o Atributo de interesse e os seguimentos anteriores devem ser definidos para conter ou encontrar Objetos. O código `Config` processa o caminho até obter o segmento final. Então, interpreta o último segmento como um Atributo no último Objeto ele encontrou no caminho. Então as funções `Config` chamam o método `TraceConnect` ou `TraceConnectWithoutContext` adequado no Objeto final.

Vamos analisar com mais detalhes o processo descrito.

O primeiro caractere "/" no caminho faz referência a um *namespace*. Um dos *namespaces* predefinidos no sistema de configuração é "NodeList" que é uma lista de todos os nós na simulação. Itens na lista são referenciados por índices , logo "/NodeList/7" refere-se ao oitavo nó na lista de nós criados durante a simulação. Esta referência é um `Ptr<Node>`, por consequência é uma subclasse de um `ns3::Object`.

Como descrito na seção Modelo de Objeto do manual *ns-3*, há suporte para Agregação de Objeto. Isto permite realizar associação entre diferentes Objetos sem qualquer programação. Cada Objeto em uma Agregação pode ser acessado a partir de outros Objetos.

O próximo segmento no caminho inicia com o carácter "$". O cifrão indica ao sistema de configuração que uma chamada `GetObject` deveria ser realizada procurando o tipo especificado em seguida. É diferente do que o `MobilityHelper` usou em `third.cc` gerenciar a Agregação, ou associar, um modelo de mobilidade para cada dos nós de rede sem fio. Quando adicionamos o "$", significa que estamos pedindo por outro Objeto que tinha sido presumidamente agregado anteriormente. Podemos pensar nisso como ponteiro de comutação do `Ptr<Node>` original como especificado por "/NodeList/7" para os modelos de mobilidade associados — quais são do tipo "$ns3::MobilityModel". Se estivermos familiarizados com `GetObject`, solicitamos ao sistema para fazer o seguinte:

```
Ptr<MobilityModel> mobilityModel = node->GetObject<MobilityModel> ()
```

Estamos no último Objeto do caminho e neste verificamos os Atributos daquele Objeto. A classe `MobilityModel` define um Atributo chamado "CourseChange". Observando o código fonte em `src/mobility/model/mobility-model.cc` e procurando por "CourseChange", encontramos,

```
.AddTraceSource ("CourseChange",
                 "The value of the position and/or velocity vector changed",
                 MakeTraceSourceAccessor (&MobilityModel::m_courseChangeTrace))
```

o qual parece muito familiar neste momento.

Se procurarmos por declarações semelhantes das variáveis rastreadas em `mobility-model.h` encontraremos,

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

A declaração de tipo `TracedCallback` identifica `m_courseChangeTrace` como um lista especial de *callbacks* que pode ser ligada usando as funções de Configuração descritas anteriormente.

A classe `MobilityModel` é projetada para ser a classe base provendo uma interface comum para todas as subclasses. No final do arquivo, encontramos um método chamado `NotifyCourseChange()`:

```
void
MobilityModel::NotifyCourseChange (void) const
{
  m_courseChangeTrace(this);
}
```

Classes derivadas chamarão este método toda vez que fizerem uma alteração na rota para suportar rastreamento. Este método invoca `operator()` em `m_courseChangeTrace`, que invocará todos os *callbacks* registrados, chamando todos os *trace sinks* que tem interesse registrado na origem do rastreamento usando a função de Configuração.

No exemplo `third.cc` nós vimos que sempre que uma mudança de rota é realizada em uma das instâncias `RandomWalk2dMobilityModel` instaladas, haverá uma chamada `NotifyCourseChange()` da classe base `MobilityModel`. Como observado, isto invoca `operator()` em `m_courseChangeTrace`, que por sua vez, chama qualquer destino do rastreamento registrados. No exemplo, o único código que registrou interesse foi aquele que forneceu o caminho de configuração. Consequentemente, a função `CourseChange` que foi ligado no Node de número sete será a única *callback* chamada.

A peça final do quebra-cabeça é o "contexto". Lembre-se da saída de `third.cc`:

```
/NodeList/7/$ns3::MobilityModel/CourseChange x = 7.27897, y = 2.22677
```

A primeira parte da saída é o contexto. É simplesmente o caminho pelo qual o código de configuração localizou a origem do rastreamento. No caso, poderíamos ter qualquer número de origens de rastreamento no sistema correspondendo a qualquer número de nós com modelos de mobilidade. É necessário uma maneira de identificar qual origem do rastreamento disparou o *callback*. Uma forma simples é solicitar um contexto de rastreamento quando é usado o `Config::Connect`.

7.2.3 Como Localizar e Conectar Origens de Rastreamento, e Descobrir Assinaturas de *Callback*

As questões que inevitavelmente os novos usuários do sistema de Rastreamento fazem, são:

1. "Eu sei que existem origens do rastreamento no núcleo da simulação, mas como eu descubro quais estão disponíveis para mim?"
2. "Eu encontrei uma origem do rastreamento, como eu defino o caminho de configuração para usar quando eu conectar a origem?"
3. "Eu encontrei uma origem do rastreamento, como eu defino o tipo de retorno e os argumentos formais da minha função de *callback*?"
4. "Eu fiz tudo corretamente e obtive uma mensagem de erro bizarra, o que isso significa?"

7.2.4 Quais Origens de Rastreamento são Disponibilizadas

A resposta é encontrada no Doxygen do *ns-3*. Acesse o sítio Web do projeto, ns-3 project, em seguida, "Documentation" na barra de navegação. Logo após, "Latest Release" e "API Documentation".

Acesse o item "Modules" na documentação do NS-3. Agora, selecione o item "C++ Constructs Used by All Modules". Serão exibidos quatro tópicos extremamente úteis:

- The list of all trace sources
- The list of all attributes
- The list of all global values
- Debugging

Estamos interessados em "*the list of all trace sources*" - a lista de todas origens do rastreamento. Selecionando este item, é exibido uma lista com todas origens disponíveis no núcleo do *ns-3*.

Como exemplo, `ns3::MobilityModel`, terá uma entrada para

```
CourseChange: The value of the position and/or velocity vector changed
```

No caso, esta foi a origem do rastreamento usada no exemplo `third.cc`, esta lista será muito útil.

7.2.5 Qual String eu uso para Conectar?

A forma mais simples é procurar na base de código do *ns-3* por alguém que já fez uso do caminho de configuração que precisamos para ligar a fonte de rastreamento. Sempre deveríamos primeiro copiar um código que funciona antes de escrever nosso próprio código. Tente usar os comandos:

```
find . -name '*.cc' | xargs grep CourseChange | grep Connect
```

e poderemos encontrar um código operacional que atenda nossas necessidades. Por exemplo, neste caso, `./ns-3-dev/examples/wireless/mixed-wireless.cc` tem algo que podemos usar:

```
Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
  MakeCallback (&CourseChangeCallback));
```

Se não localizamos nenhum exemplo na distribuição, podemos tentar o Doxygen do *ns-3*. É provavelmente mais simples que percorrer o exemplo "CourseChanged".

Suponha que encontramos a origem do rastreamento "CourseChanged" na "The list of all trace sources" e queremos resolver como nos conectar a ela. Você sabe que está usando um `ns3::RandomWalk2dMobilityModel`. Logo, acesse o item "Class List" na documentação do *ns-3*. Será exibida a lista de todas as classes. Selecione a entrada para `ns3::RandomWalk2dMobilityModel` para exibir a documentação da classe.

Acesse a seção "Member Function Documentation" e obterá a documentação para a função `GetTypeId`. Você construiu uma dessas em um exemplo anterior:

```
static TypeId GetTypeId (void)
{
  static TypeId tid = TypeId ("MyObject")
    .SetParent (Object::GetTypeId ())
    .AddConstructor<MyObject> ()
    .AddTraceSource ("MyInteger",
                     "An integer value to trace.",
                     MakeTraceSourceAccessor (&MyObject::m_myInt))
    ;
  return tid;
}
```

Como abordado, este código conecta os sistemas *Config* e Atributos à origem do rastreamento. É também o local onde devemos iniciar a busca por informação sobre como conectar.

Você está observando a mesma informação para `RandomWalk2dMobilityModel`; e a informação que você precisa está explícita no Doxygen:

```
This object is accessible through the following paths with Config::Set
              and Config::Connect:

/NodeList/[i]/$ns3::MobilityModel/$ns3::RandomWalk2dMobilityModel
```

A documentação apresenta como obter o Objeto `RandomWalk2dMobilityModel`. Compare o texto anterior com o texto que nós usamos no código do exemplo:

```
"/NodeList/7/$ns3::MobilityModel"
```

A diferença é que há duas chamadas `GetObject` inclusas no texto da documentação. A primeira, para `$ns3::MobilityModel` solicita a agregação para a classe base. A segunda, para `$ns3::RandomWalk2dMobilityModel` é usada como *cast* da classe base para a implementação concreta da classe.

Analisando ainda mais o `GetTypeId` no Doxygen, temos

```
No TraceSources defined for this type.
TraceSources defined in parent class ns3::MobilityModel:

CourseChange: The value of the position and/or velocity vector changed
Reimplemented from ns3::MobilityModel
```

Isto é exatamente o que precisamos saber. A origem do rastreamento de interesse é encontrada em `ns3::MobilityModel`. O interessante é que pela documentação não é necessário o *cast* extra para obter a classe concreta, pois a origem do rastreamento está na classe base. Por consequência, o `GetObject` adicional não é necessário e podemos usar o caminho:

```
/NodeList/[i]/$ns3::MobilityModel
```

que é idêntico ao caminho do exemplo:

```
/NodeList/7/$ns3::MobilityModel
```

7.2.6 Quais são os Argumentos Formais e o Valor de Retorno?

A forma mais simples é procurar na base de código do *ns-3* por um código existente. Você sempre deveria primeiro copiar um código que funciona antes de escrever seu próprio. Tente usar os comandos:

```
find . -name '*.cc' | xargs grep CourseChange | grep Connect
```

e você poderá encontrar código operacional. Por exemplo, neste caso, `./ns-3-dev/examples/wireless/mixed-wireless.cc` tem código para ser reaproveitado.

```
Config::Connect ("/NodeList/*/$ns3::MobilityModel/CourseChange",
  MakeCallback (&CourseChangeCallback));
```

como resultado, `MakeCallback` indicaria que há uma função *callback* que pode ser usada. Para reforçar:

```
static void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

Acredite em Minha Palavra

Se não há exemplos, pode ser desafiador descobrir por meio da análise do código fonte.

Antes de aventurar-se no código, diremos algo importante: O valor de retorno de sua *callback* sempre será *void*. A lista de parâmetros formais para uma `TracedCallback` pode ser encontrada no lista de parâmetro padrão na declaração. Recorde do exemplo atual, isto está em `mobility-model.h`, onde encontramos:

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

Não há uma correspondência de um-para-um entre a lista de parâmetro padrão na declaração e os argumentos formais da função *callback*. Aqui, há um parâmetro padrão, que é um `Ptr<const MobilityModel>`. Isto significa que precisamos de uma função que retorna *void* e possui um parâmetro `Ptr<const MobilityModel>`. Por exemplo,

```
void
CourseChangeCallback (Ptr<const MobilityModel> model)
{
  ...
}
```

Isto é tudo que precisamos para `Config::ConnectWithoutContext`. Se você quer um contexto, use `Config::Connect` e uma função *callback* que possui como um parâmetro uma *string* de contexto, seguido pelo argumento.

```
void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

Para garantir que `CourseChangeCallback` seja somente visível em seu arquivo, você pode adicionar a palavra chave `static`, como no exemplo:

```
static void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

exatamente o que é usado no exemplo `third.cc`.

A Forma Complicada

Esta seção é opcional. Pode ser bem penosa para aqueles que conhecem poucos detalhes de tipos parametrizados de dados (*templates*). Entretanto, se continuarmos nessa seção, mergulharemos em detalhes de baixo nível do *ns-3*.

Vamos novamente descobrir qual assinatura da função de *callback* é necessária para o Atributo "CourseChange". Isto pode ser doloroso, mas precisamos fazê-lo apenas uma vez. Depois de tudo, você será capaz de entender um `TracedCallback`.

Primeiro, verificamos a declaração da origem do rastreamento. Recorde que isto está em `mobility-model.h`:

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

Esta declaração é para um *template*. O parâmetro do *template* está entre <>, logo estamos interessados em descobrir o que é `TracedCallback<>`. Se não tem nenhuma ideia de onde pode ser encontrado, use o utilitário *grep*.

Estamos interessados em uma declaração similar no código fonte do *ns-3*, logo buscamos no diretório `src`. Então, sabemos que esta declaração tem um arquivo de cabeçalho, e procuramos por ele usando:

```
find . -name '*.h' | xargs grep TracedCallback
```

Obteremos 124 linhas, com este comando. Analisando a saída, encontramos alguns *templates* que podem ser úteis.

```
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::TracedCallback ()
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::ConnectWithoutContext (c ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::Connect (const CallbackB ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::DisconnectWithoutContext ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::Disconnect (const Callba ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (void) const ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1) const ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1, T2 a2 ...
```

Observamos que todas linhas são do arquivo de cabeçalho `traced-callback.h`, logo ele parece muito promissor. Para confirmar, verifique o arquivo `mobility-model.h` e procure uma linha que corrobore esta suspeita.

```
#include "ns3/traced-callback.h"
```

Observando as inclusões em `mobility-model.h`, verifica-se a inclusão do `traced-callback.h` e conclui-se que este deve ser o arquivo.

O próximo passo é analisar o arquivo `src/core/model/traced-callback.h` e entender sua funcionalidade.

Há um comentário no topo do arquivo que deveria ser animador:

```
An ns3::TracedCallback has almost exactly the same API as a normal ns3::Callback but
instead of forwarding calls to a single function (as an ns3::Callback normally does),
it forwards calls to a chain of ns3::Callback.
```

Isto deveria ser familiar e confirma que estamos no caminho correto.

Depois deste comentário, encontraremos

```
template<typename T1 = empty, typename T2 = empty,
         typename T3 = empty, typename T4 = empty,
         typename T5 = empty, typename T6 = empty,
         typename T7 = empty, typename T8 = empty>
class TracedCallback
{
  ...
```

Isto significa que TracedCallback é uma classe genérica (*templated class*). Possui oito possíveis tipos de parâmetros com valores padrões. Retorne e compare com a declaração que você está tentando entender:

```
TracedCallback<Ptr<const MobilityModel> > m_courseChangeTrace;
```

O `typename T1` na declaração da classe corresponde a `Ptr<const MobilityModel>` da declaração anterior. Todos os outros parâmetros são padrões. Observe que o construtor não contribui com muita informação. O único lugar onde há uma conexão entre a função *callback* e o sistema de rastreamento é nas funções `Connect` e `ConnectWithoutContext`. Como mostrado a seguir:

```
template<typename T1, typename T2,
         typename T3, typename T4,
         typename T5, typename T6,
         typename T7, typename T8>
void
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::ConnectWithoutContext ...
{
  Callback<void,T1,T2,T3,T4,T5,T6,T7,T8> cb;
  cb.Assign (callback);
  m_callbackList.push_back (cb);
}
```

Você está no olho do furação. Quando o *template* é instanciado pela declaração anterior, o compilador substitui `T1` por `Ptr<const MobilityModel>`.

```
void
TracedCallback<Ptr<const MobilityModel>::ConnectWithoutContext ... cb
{
  Callback<void, Ptr<const MobilityModel> > cb;
  cb.Assign (callback);
  m_callbackList.push_back (cb);
}
```

Podemos observar a implementação de tudo que foi explicado até este ponto. O código cria uma *callback* do tipo adequado e atribui sua função para ela. Isto é equivalente a `pfi = MyFunction` discutida anteriormente. O código

então adiciona a *callback* para a lista de *callbacks* para esta origem. O que não observamos ainda é a definição da *callback*. Usando o utilitário *grep* podemos encontrar o arquivo `./core/callback.h` e verificar a definição.

No arquivo há muito código incompreensível. Felizmente há algum em Inglês.

```
This class template implements the Functor Design Pattern.
It is used to declare the type of a Callback:
 - the first non-optional template argument represents
   the return type of the callback.
 - the second optional template argument represents
   the type of the first argument to the callback.
 - the third optional template argument represents
   the type of the second argument to the callback.
 - the fourth optional template argument represents
   the type of the third argument to the callback.
 - the fifth optional template argument represents
   the type of the fourth argument to the callback.
 - the sixth optional template argument represents
   the type of the fifth argument to the callback.
```

Nós estamos tentando descobrir o que significa a declaração

```
Callback<void, Ptr<const MobilityModel> > cb;
```

Agora entendemos que o primeiro parâmetro, `void`, indica o tipo de retorno da *callback*. O segundo parâmetro, `Ptr<const MobilityModel>` representa o primeiro argumento da *callback*.

A *callback* em questão é a sua função que recebe os eventos de rastreamento. Logo, podemos deduzir que precisamos de uma função que retorna `void` e possui um parâmetro `Ptr<const MobilityModel>`. Por exemplo,

```
void
CourseChangeCallback (Ptr<const MobilityModel> model)
{
  ...
}
```

Isto é tudo que precisamos no `Config::ConnectWithoutContext`. Se você quer um contexto, use `Config::Connect` e uma função *callback* que possui como um parâmetro uma *string* de contexto, seguido pelo argumento.

```
void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

Se queremos garantir que `CourseChangeCallback` é visível somente em seu arquivo, você pode adicionar a palavra chave `static`, como no exemplo:

```
static void
CourseChangeCallback (std::string path, Ptr<const MobilityModel> model)
{
  ...
}
```

o que é exatamente usado no exemplo `third.cc`. Talvez seja interessante reler a seção (Acredite em Minha Palavra).

Há mais detalhes sobre a implementação de *callbacks* no manual do *ns-3*. Elas estão entre os mais usados construtores das partes de baixo-nível do *ns-3*. Em minha opinião, algo bastante elegante.

7.2.7 E quanto a TracedValue?

No início desta seção, nós apresentamos uma parte de código simples que usou um `TracedValue<int32_t>` para demonstrar o básico sobre código de rastreamento. Nós desprezamos os métodos para encontrar o tipo de retorno e os argumentos formais para o `TracedValue`. Acelerando o processo, indicamos o arquivo `src/core/model/traced-value.h` e a parte relevante do código:

```
template <typename T>
class TracedValue
{
public:
  ...
  void Set (const T &v) {
    if (m_v != v)
      {
      m_cb (m_v, v);
      m_v = v;
      }
  }
  ...
private:
  T m_v;
  TracedCallback<T,T> m_cb;
};
```

Verificamos que `TracedValue` é uma classe parametrizada. No caso simples do início da seção, o nome do tipo é int32_t. Isto significa que a variável membro sendo rastreada (`m_v` na seção privada da classe) será `int32_t m_v`. O método `Set` possui um argumento `const int32_t &v`. Você deveria ser capaz de entender que o código `Set` disparará o *callback* `m_cb` com dois parâmetros: o primeiro sendo o valor atual do `TracedValue`; e o segundo sendo o novo valor.

A *callback* `m_cb` é declarada como um `TracedCallback<T, T>` que corresponderá a um `TracedCallback<int32_t, int32_t>` quando a classe é instanciada.

Lembre-se que o destino da *callback* de um TracedCallback sempre retorna `void`. Lembre também que há uma correspondência de um-para-um entre a lista de parâmetros polimórfica e os argumentos formais da função *callback*. Logo, a *callback* precisa ter uma assinatura de função similar a:

```
void
MyCallback (int32_t oldValue, int32_t newValue)
{
  ...
}
```

Isto é exatamente o que nós apresentamos no exemplo simples abordado anteriormente.

```
void
IntTrace (int32_t oldValue, int32_t newValue)
{
  std::cout << "Traced " << oldValue << " to " << newValue << std::endl;
}
```

7.3 Um Exemplo Real

Vamos fazer um exemplo retirado do livro "TCP/IP Illustrated, Volume 1: The Protocols" escrito por W. Richard Stevens. Localizei na página 366 do livro um gráfico da janela de congestionamento e números de sequência versus

tempo. Stevens denomina de "Figure 21.10. Value of cwnd and send sequence number while data is being transmitted." Vamos recriar a parte *cwnd* daquele gráfico em *ns-3* usando o sistema de rastreamento e `gnuplot`.

7.3.1 Há Fontes de Rastreamento Disponibilizadas?

Primeiro devemos pensar sobre como queremos obter os dados de saída. O que é que nós precisamos rastrear? Consultamos então *"The list of all trace sources"* para sabermos o que temos para trabalhar. Essa seção encontra-se na documentação na seção *"Module"*, no item *"C++ Constructs Used by All Modules"*. Procurando na lista, encontraremos:

```
ns3::TcpNewReno
CongestionWindow: The TCP connection's congestion window
```

A maior parte da implementação do TCP no *ns-3* está no arquivo `src/internet/model/tcp-socket-base.cc` enquanto variantes do controle de congestionamento estão em arquivos como `src/internet/model/tcp-newreno.cc`. Se não sabe a priori dessa informação, use:

```
find . -name '*.cc' | xargs grep -i tcp
```

Haverá páginas de respostas apontando para aquele arquivo.

No início do arquivo `src/internet/model/tcp-newreno.cc` há as seguintes declarações:

```
TypeId
TcpNewReno::GetTypeId ()
{
  static TypeId tid = TypeId("ns3::TcpNewReno")
    .SetParent<TcpSocketBase> ()
    .AddConstructor<TcpNewReno> ()
    .AddTraceSource ("CongestionWindow",
                     "The TCP connection's congestion window",
                     MakeTraceSourceAccessor (&TcpNewReno::m_cWnd))
    ;
  return tid;
}
```

Isto deveria guiá-lo para localizar a declaração de `m_cWnd` no arquivo de cabeçalho `src/internet/model/tcp-newreno.h`. Temos nesse arquivo:

```
TracedValue<uint32_t> m_cWnd; //Congestion window
```

Você deveria entender este código. Se nós temos um ponteiro para `TcpNewReno`, podemos fazer `TraceConnect` para a origem do rastreamento "CongestionWindow" se fornecermos uma *callback* adequada. É o mesmo tipo de origem do rastreamento que nós abordamos no exemplo simples no início da seção, exceto que estamos usando `uint32_t` ao invés de `int32_t`.

Precisamos prover uma *callback* que retorne `void` e receba dois parâmetros `uint32_t`, o primeiro representando o valor antigo e o segundo o novo valor:

```
void
CwndTrace (uint32_t oldValue, uint32_t newValue)
{
  ...
}
```

7.3.2 Qual código Usar?

É sempre melhor localizar e modificar um código operacional que iniciar do zero. Portanto, vamos procurar uma origem do rastreamento da "CongestionWindow" e verificar se é possível modificar. Para tal, usamos novamente o *grep*:

```
find . -name '*.cc' | xargs grep CongestionWindow
```

Encontramos alguns candidatos: `examples/tcp/tcp-large-transfer.cc` e `src/test/ns3tcp/ns3tcp-cwnd-test-suite.cc`.

Nós não visitamos nenhum código de teste ainda, então vamos fazer isto agora. Código de teste é pequeno, logo é uma ótima escolha. Acesse o arquivo `src/test/ns3tcp/ns3tcp-cwnd-test-suite.cc` e localize "CongestionWindow". Como resultado, temos

```
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow",
  MakeCallback (&Ns3TcpCwndTestCase1::CwndChange, this));
```

Como abordado, temos uma origem do rastreamento "CongestionWindow"; então ela aponta para `TcpNewReno`, poderíamos alterar o `TraceConnect` para o que nós desejamos. Vamos extrair o código que precisamos desta função (`Ns3TcpCwndTestCase1::DoRun (void)`). Se você observar, perceberá que parece como um código *ns-3*. E descobre-se exatamente que realmente é um código. É um código executado pelo *framework* de teste, logo podemos apenas colocá-lo no `main` ao invés de `DoRun`. A tradução deste teste para um código nativo do *ns-3* é apresentada no arquivo `examples/tutorial/fifth.cc`.

7.3.3 Um Problema Comum e a Solução

O exemplo `fifth.cc` demonstra um importante regra que devemos entender antes de usar qualquer tipo de Atributo: devemos garantir que o alvo de um comando `Config` existe antes de tentar usá-lo. É a mesma ideia que um objeto não pode ser usado sem ser primeiro instanciado. Embora pareça óbvio, muitas pessoas erram ao usar o sistema pela primeira vez.

Há três fases básicas em qualquer código *ns-3*. A primeira é a chamada de "Configuration Time" ou "Setup Time" e ocorre durante a execução da função `main`, mas antes da chamada `Simulator::Run`. O segunda fase é chamada de "Simulation Time" e é quando o `Simulator::Run` está executando seus eventos. Após completar a execução da simulação, `Simulator::Run` devolve o controle a função `main`. Quando isto acontece, o código entra na terceira fase, o "Teardown Time", que é quando estruturas e objetos criados durante a configuração são analisados e liberados.

Talvez o erro mais comum em tentar usar o sistema de rastreamento é supor que entidades construídas dinamicamente durante a fase de simulação estão acessíveis durante a fase de configuração. Em particular, um `Socket` *ns-3* é um objeto dinâmico frequentemente criado por Aplicações (`Applications`) para comunicação entre nós de redes. Uma Aplicação *ns-3* tem um "Start Time" e "Stop Time" associado a ela. Na maioria dos casos, uma Aplicação não tentar criar um objeto dinâmico até que seu método `StartApplication` é chamado em algum "Start Time". Isto é para garantir que a simulação está completamente configurada antes que a aplicação tente fazer alguma coisa (o que aconteceria se tentasse conectar a um nó que não existisse durante a fase de configuração). A resposta para esta questão é:

1. criar um evento no simulador que é executado depois que o objeto dinâmico é criado e ativar o rastreador quando aquele evento é executado; ou
2. criar o objeto dinâmico na fase de configuração, ativá-lo, e passar o objeto para o sistema usar durante a fase de simulação.

Nós consideramos a segunda abordagem no exemplo `fifth.cc`. A decisão implicou na criação da Aplicação `MyApp`, com o propósito de passar um `Socket` como parâmetro.

7.3.4 Analisando o exemplo fifth.cc

Agora, vamos analisar o programa exemplo detalhando o teste da janela de congestionamento. Segue o código do arquivo localizado em `examples/tutorial/fifth.cc`:

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation;
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Include., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <fstream>
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("FifthScriptExample");
```

Todo o código apresentado já foi discutido. As próximas linhas são comentários apresentando a estrutura da rede e comentários abordando o problema descrito com o `Socket`.

```
// ===========================================================================
//
//         node 0                 node 1
//   +----------------+    +----------------+
//   |    ns-3 TCP    |    |    ns-3 TCP    |
//   +----------------+    +----------------+
//   |    10.1.1.1    |    |    10.1.1.2    |
//   +----------------+    +----------------+
//   | point-to-point |    | point-to-point |
//   +----------------+    +----------------+
//           |                      |
//           +----------------------+
//                5 Mbps, 2 ms
//
//
// We want to look at changes in the ns-3 TCP congestion window.  We need
// to crank up a flow and hook the CongestionWindow attribute on the socket
// of the sender.  Normally one would use an on-off application to generate a
// flow, but this has a couple of problems.  First, the socket of the on-off
// application is not created until Application Start time, so we wouldn't be
// able to hook the socket (now) at configuration time.  Second, even if we
// could arrange a call after start time, the socket is not public so we
// couldn't get at it.
//
```

```
// So, we can cook up a simple version of the on-off application that does what
// we want.  On the plus side we don't need all of the complexity of the on-off
// application.  On the minus side, we don't have a helper, so we have to get
// a little more involved in the details, but this is trivial.
//
// So first, we create a socket and do the trace connect on it; then we pass
// this socket into the constructor of our simple application which we then
// install in the source node.
// ============================================================================
//
```

A próxima parte é a declaração da Aplicação `MyApp` que permite que o `Socket` seja criado na fase de configuração.

```
class MyApp : public Application
{
public:

  MyApp ();
  virtual ~MyApp();

  void Setup (Ptr<Socket> socket, Address address, uint32_t packetSize,
    uint32_t nPackets, DataRate dataRate);

private:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

  void ScheduleTx (void);
  void SendPacket (void);

  Ptr<Socket>     m_socket;
  Address         m_peer;
  uint32_t        m_packetSize;
  uint32_t        m_nPackets;
  DataRate        m_dataRate;
  EventId         m_sendEvent;
  bool            m_running;
  uint32_t        m_packetsSent;
};
```

A classe `MyApp` herda a classe `Application` do *ns-3*. Acesse o arquivo `src/network/model/application.h` se estiver interessado sobre detalhes dessa herança. A classe `MyApp` é obrigada sobrescrever os métodos `StartApplication` e `StopApplication`. Estes métodos são automaticamente chamado quando `MyApp` é solicitada iniciar e parar de enviar dados durante a simulação.

Como Aplicações são Iniciadas e Paradas (Opcional)

Nesta seção é explicado como eventos tem início no sistema. É uma explicação mais detalhada e não é necessária se não planeja entender detalhes do sistema. É interessante, por outro lado, pois aborda como partes do *ns-3* trabalham e mostra alguns detalhes de implementação importantes. Se você planeja implementar novos modelos, então deve entender essa seção.

A maneira mais comum de iniciar eventos é iniciar uma Aplicação. Segue as linhas de um código *ns-3* que faz exatamente isso:

```
ApplicationContainer apps = ...
apps.Start (Seconds (1.0));
```

```
apps.Stop (Seconds (10.0));
```

O código do contêiner aplicação (src/network/helper/application-container.h) itera pelas aplicações no contêiner e chama,

```
app->SetStartTime (startTime);
```

como um resultado da chamada apps.Start e

```
app->SetStopTime (stopTime);
```

como um resultado da chamada apps.Stop.

O último resultado destas chamadas queremos ter o simulador executando chamadas em nossa Applications para controlar o inicio e a parada. No caso MyApp, herda da classe Application e sobrescreve StartApplication e StopApplication. Estas são as funções invocadas pelo simulador no momento certo. No caso de MyApp, o MyApp::StartApplication faz o Bind e Connect no *socket*, em seguida, inicia o fluxo de dados chamando MyApp::SendPacket. MyApp::StopApplication interrompe a geração de pacotes cancelando qualquer evento pendente de envio e também fechando o socket.

Uma das coisas legais sobre o *ns-3* é que podemos ignorar completamente os detalhes de implementação de como sua Aplicação é "automaticamente" chamada pelo simulador no momento correto. De qualquer forma, detalhamos como isso acontece a seguir.

Se observarmos em src/network/model/application.cc, descobriremos que o método SetStartTime de uma Application apenas altera a variável m_startTime e o método SetStopTime apenas altera a variável m_stopTime.

Para continuar e entender o processo, precisamos saber que há uma lista global de todos os nós no sistema. Sempre que você cria um nó em uma simulação, um ponteiro para aquele nó é adicionado para a lista global NodeList.

Observe em src/network/model/node-list.cc e procure por NodeList::Add. A implementação public static chama uma implementação privada denominada NodeListPriv::Add. Isto é comum no *ns-3*. Então, observe NodeListPriv::Add e encontrará,

```
Simulator::ScheduleWithContext (index, TimeStep (0), &Node::Start, node);
```

Isto significa que sempre que um Node é criado em uma simulação, como uma implicação, uma chamada para o método Start do nó é agendada para que ocorra no tempo zero. Isto não significa que o nó vai iniciar fazendo alguma coisa, pode ser interpretado como uma chamada informacional no Node dizendo a ele que a simulação teve início, não uma chamada para ação dizendo ao Node iniciar alguma coisa.

Então, o NodeList::Add indiretamente agenda uma chamada para Node::Start no tempo zero, para informar ao novo nó que a simulação foi iniciada. Se olharmos em src/network/model/node.h não acharemos um método chamado Node::Start. Acontece que o método Start é herdado da classe Object. Todos objetos no sistema podem ser avisados que a simulação iniciou e objetos da classe Node são exemplos.

Observe em seguida src/core/model/object.cc. Localize por Object::Start. Este código não é tão simples como você esperava desde que Objects *ns-3* suportam agregação. O código em Object::Start então percorre todos os objetos que estão agregados e chama o método DoStart de cada um. Este é uma outra prática muito comum em *ns-3*. Há um método pública na API, que permanece constante entre implementações, que chama um método de implementação privada que é herdado e implementado por subclasses. Os nomes são tipicamente algo como MethodName para os da API pública e DoMethodName para os da API privada.

Logo, deveríamos procurar por um método Node::DoStart em src/network/model/node.cc. Ao localizar o método, descobrirá um método que percorre todos os dispositivos e aplicações no nó chamando respectivamente device->Start e application->Start.

As classes Device e Application herdam da classe Object, então o próximo passo é entender o que acontece quando Application::DoStart é executado. Observe o código em

src/network/model/application.cc:

```
void
Application::DoStart (void)
{
  m_startEvent = Simulator::Schedule (m_startTime, &Application::StartApplication, this);
  if (m_stopTime != TimeStep (0))
    {
      m_stopEvent = Simulator::Schedule (m_stopTime, &Application::StopApplication, this);
    }
  Object::DoStart ();
}
```

Aqui finalizamos nosso detalhamento. Ao implementar uma Aplicação do *ns-3*, sua nova aplicação herda da classe `Application`. Você sobrescreve os métodos `StartApplication` e `StopApplication` e provê mecanismos para iniciar e finalizar o fluxo de dados de sua nova `Application`. Quando um `Node` é criado na simulação, ele é adicionado a uma lista global `NodeList`. A ação de adicionar um nó na lista faz com que um evento do simulador seja agendado para o tempo zero e que chama o método `Node::Start` do `Node` recentemente adicionado para ser chamado quando a simulação inicia. Como um `Node` herda de `Object`, a chamada invoca o método `Object::Start` no `Node`, o qual, por sua vez, chama os métodos `DoStart` em todos os `Objects` agregados ao `Node` (pense em modelos móveis). Como o `Node` `Object` tem sobrescritos `DoStart`, o método é chamado quando a simulação inicia. O método `Node::DoStart` chama o método `Start` de todas as `Applications` no nó. Por sua vez, `Applications` são também `Objects`, o que resulta na invocação do `Application::DoStart`. Quando `Application::DoStart` é chamada, ela agenda eventos para as chamadas `StartApplication` e `StopApplication` na `Application`. Estas chamadas são projetadas para iniciar e parar o fluxo de dados da `Application`.

Após essa longa jornada, você pode entende melhor outra parte do *ns-3*.

A Aplicação MyApp

A Aplicação `MyApp` precisa de um construtor e um destrutor,

```
MyApp::MyApp ()
  : m_socket (0),
    m_peer (),
    m_packetSize (0),
    m_nPackets (0),
    m_dataRate (0),
    m_sendEvent (),
    m_running (false),
    m_packetsSent (0)
{
}

MyApp::~MyApp()
{
  m_socket = 0;
}
```

O código seguinte é a principal razão da existência desta Aplicação.

```
void
MyApp::Setup (Ptr<Socket> socket, Address address, uint32_t packetSize,
                     uint32_t nPackets, DataRate dataRate)
{
  m_socket = socket;
  m_peer = address;
```

```
  m_packetSize = packetSize;
  m_nPackets = nPackets;
  m_dataRate = dataRate;
}
```

Neste código inicializamos os atributos da classe. Do ponto de vista do rastreamento, a mais importante é `Ptr<Socket> socket` que deve ser passado para a aplicação durante o fase de configuração. Lembre-se que vamos criar o `Socket` como um `TcpSocket` (que é implementado por `TcpNewReno`) e associar sua origem do rastreamento de sua *"CongestionWindow"* antes de passá-lo no método `Setup`.

```
void
MyApp::StartApplication (void)
{
  m_running = true;
  m_packetsSent = 0;
  m_socket->Bind ();
  m_socket->Connect (m_peer);
  SendPacket ();
}
```

Este código sobrescreve `Application::StartApplication` que será chamado automaticamente pelo simulador para iniciar a `Application` no momento certo. Observamos que é realizada uma operação `Socket Bind`. Se você conhece Sockets de Berkeley isto não é uma novidade. É responsável pelo conexão no lado do cliente, ou seja, o `Connect` estabelece uma conexão usando TCP no endereço `m_peer`. Por isso, precisamos de uma infraestrutura funcional de rede antes de executar a fase de simulação. Depois do `Connect`, a `Application` inicia a criação dos eventos de simulação chamando `SendPacket`.

```
void
MyApp::StopApplication (void)
{
  m_running = false;

  if (m_sendEvent.IsRunning ())
    {
      Simulator::Cancel (m_sendEvent);
    }

  if (m_socket)
    {
      m_socket->Close ();
    }
}
```

A todo instante um evento da simulação é agendado, isto é, um `Event` é criado. Se o `Event` é uma execução pendente ou está executando, seu método `IsRunning` retornará `true`. Neste código, se `IsRunning()` retorna verdadeiro (*true*), nós cancelamos (`Cancel`) o evento, e por consequência, é removido da fila de eventos do simulador. Dessa forma, interrompemos a cadeia de eventos que a `Application` está usando para enviar seus `Packets`. A Aplicação não enviará mais pacotes e em seguida fechamos (`Close`) o *socket* encerrando a conexão TCP.

O socket é deletado no destrutor quando `m_socket = 0` é executado. Isto remove a última referência para Ptr<Socket> que ocasiona o destrutor daquele Objeto ser chamado.

Lembre-se que `StartApplication` chamou `SendPacket` para iniciar a cadeia de eventos que descreve o comportamento da `Application`.

```
void
MyApp::SendPacket (void)
{
  Ptr<Packet> packet = Create<Packet> (m_packetSize);
```

```
  m_socket->Send (packet);

  if (++m_packetsSent < m_nPackets)
    {
      ScheduleTx ();
    }
}
```

Este código apenas cria um pacote (`Packet`) e então envia (`Send`).

É responsabilidade da `Application` gerenciar o agendamento da cadeia de eventos, então, a chamada `ScheduleTx` agenda outro evento de transmissão (um `SendPacket`) até que a `Application` decida que enviou o suficiente.

```
void
MyApp::ScheduleTx (void)
{
  if (m_running)
    {
      Time tNext (Seconds (m_packetSize * 8 / static_cast<double> (m_dataRate.GetBitRate ())));
      m_sendEvent = Simulator::Schedule (tNext, &MyApp::SendPacket, this);
    }
}
```

Enquanto a `Application` está executando, `ScheduleTx` agendará um novo evento, que chama `SendPacket` novamente. Verifica-se que a taxa de transmissão é sempre a mesma, ou seja, é a taxa que a `Application` produz os bits. Não considera nenhuma sobrecarga de protocolos ou canais físicos no transporte dos dados. Se alterarmos a taxa de transmissão da `Application` para a mesma taxa dos canais físicos, poderemos ter um estouro de *buffer*.

Destino do Rastreamento

O foco deste exercício é obter notificações (*callbacks*) do TCP indicando a modificação da janela de congestionamento. O código a seguir implementa o destino do rastreamento.

```
static void
CwndChange (uint32_t oldCwnd, uint32_t newCwnd)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);
}
```

Esta função registra o tempo de simulação atual e o novo valor da janela de congestionamento toda vez que é modificada. Poderíamos usar essa saída para construir um gráfico do comportamento da janela de congestionamento com relação ao tempo.

Nós adicionamos um novo destino do rastreamento para mostrar onde pacotes são perdidos. Vamos adicionar um modelo de erro.

```
static void
RxDrop (Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
}
```

Este destino do rastreamento será conectado a origem do rastreamento "PhyRxDrop" do `NetDevice` ponto-a-ponto. Esta origem do rastreamento dispara quando um pacote é removido da camada física de um `NetDevice`. Se olharmos rapidamente `src/point-to-point/model/point-to-point-net-device.cc` verificamos que a origem do rastreamento refere-se a `PointToPointNetDevice::m_phyRxDropTrace`. E se procurarmos em `src/point-to-point/model/point-to-point-net-device.h` por essa variável, encontraremos que

ela está declarada como uma `TracedCallback<Ptr<const Packet> >`. Isto significa que nosso *callback* deve ser uma função que retorna `void` e tem um único parâmetro `Ptr<const Packet>`.

O Programa Principal

O código a seguir corresponde ao início da função principal:

```
int
main (int argc, char *argv[])
{
  NodeContainer nodes;
  nodes.Create (2);

  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
  pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms"));

  NetDeviceContainer devices;
  devices = pointToPoint.Install (nodes);
```

São criados dois nós ligados por um canal ponto-a-ponto, como mostrado na ilustração no início do arquivo.

Nas próximas linhas, temos um código com algumas informações novas. Se nós

rastrearmos uma conexão que comporta-se perfeitamente, terminamos com um

janela de congestionamento que aumenta monoliticamente. Para observarmos um

comportamento interessante, introduzimos erros que causarão perda de pacotes,

duplicação de ACK's e assim, introduz comportamentos mais interessantes a

janela de congestionamento.

O *ns-3* provê objetos de um modelo de erros (`ErrorModel`) que pode ser adicionado aos canais (`Channels`). Nós usamos o `RateErrorModel` que permite introduzir erros no canal dada uma *taxa*.

```
Ptr<RateErrorModel> em = CreateObjectWithAttributes<RateErrorModel> (
  "RanVar", RandomVariableValue (UniformVariable (0., 1.)),
  "ErrorRate", DoubleValue (0.00001));
devices.Get (1)->SetAttribute ("ReceiveErrorModel", PointerValue (em));
```

O código instancia um objeto `RateErrorModel`. Para simplificar usamos a função `CreateObjectWithAttributes` que instancia e configura os Atributos. O Atributo "RanVar" foi configurado para uma variável randômica que gera uma distribuição uniforme entre 0 e 1. O Atributo "ErrorRate" também foi alterado. Por fim, configuramos o modelo erro no `NetDevice` ponto-a-ponto modificando o atributo "ReceiveErrorModel". Isto causará retransmissões e o gráfico ficará mais interessante.

```
InternetStackHelper stack;
stack.Install (nodes);

Ipv4AddressHelper address;
address.SetBase ("10.1.1.0", "255.255.255.252");
Ipv4InterfaceContainer interfaces = address.Assign (devices);
```

Neste código configura a pilha de protocolos da internet nos dois nós de rede, cria interfaces e associa endereços IP para os dispositivos ponto-a-ponto.

Como estamos usando TCP, precisamos de um nó de destino para receber as conexões e os dados. O `PacketSink Application` é comumente usado no *ns-3* para este propósito.

```
uint16_t sinkPort = 8080;
Address sinkAddress (InetSocketAddress(interfaces.GetAddress (1), sinkPort));
PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory",
  InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
ApplicationContainer sinkApps = packetSinkHelper.Install (nodes.Get (1));
sinkApps.Start (Seconds (0.));
sinkApps.Stop (Seconds (20.));
```

Este código deveria ser familiar, com exceção de,

```
PacketSinkHelper packetSinkHelper ("ns3::TcpSocketFactory",
  InetSocketAddress (Ipv4Address::GetAny (), sinkPort));
```

Este código instancia um `PacketSinkHelper` e cria sockets usando a classe `ns3::TcpSocketFactory`. Esta classe implementa o padrão de projeto "fábrica de objetos". Dessa forma, em vez de criar os objetos diretamente, fornecemos ao `PacketSinkHelper` um texto que especifica um `TypeId` usado para criar um objeto que, por sua vez, pode ser usado para criar instâncias de Objetos criados pela implementação da fábrica de objetos.

O parâmetro seguinte especifica o endereço e a porta para o mapeamento.

As próximas duas linhas do código criam o *socket* e conectam a origem do rastreamento.

```
Ptr<Socket> ns3TcpSocket = Socket::CreateSocket (nodes.Get (0),
  TcpSocketFactory::GetTypeId ());
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow",
  MakeCallback (&CwndChange));
```

A primeira declaração chama a função estática `Socket::CreateSocket` e passa um `Node` e um `TypeId` para o objeto fábrica usado para criar o *socket*.

Uma vez que o `TcpSocket` é criado e adicionado ao `Node`, nós usamos `TraceConnectWithoutContext` para conectar a origem do rastreamento "CongestionWindow" para o nosso destino do rastreamento.

Codificamos uma `Application` então podemos obter um `Socket` (durante a fase de configuração) e usar na fase de simulação. Temos agora que instanciar a `Application`. Para tal, segue os passos:

```
Ptr<MyApp> app = CreateObject<MyApp> ();
app->Setup (ns3TcpSocket, sinkAddress, 1040, 1000, DataRate ("1Mbps"));
nodes.Get (0)->AddApplication (app);
app->Start (Seconds (1.));
app->Stop (Seconds (20.));
```

A primeira linha cria um Objeto do tipo `MyApp` – nossa `Application`. A segunda linha especifica o *socket*, o endereço de conexão, a quantidade de dados a ser enviada em cada evento, a quantidade de eventos de transmissão a ser gerados e a taxa de produção de dados para estes eventos.

> Next, we manually add the `MyApp Application` to the source node and explicitly call the `Start` and `Stop` methods on the `Application` to tell it when to start and stop doing its thing.

Depois, adicionamos a `MyApp Application` para o nó origem e chamamos os métodos `Start` e `Stop` para dizer quando e iniciar e parar a simulação.

Precisamos agora fazer a conexão entre o receptor com nossa *callback*.

```
devices.Get (1)->TraceConnectWithoutContext("PhyRxDrop", MakeCallback (&RxDrop));
```

Estamos obtendo uma referência para o `Node NetDevice` receptor e conectando a origem do rastreamento pelo Atributo "PhyRxDrop" do dispositivo no destino do rastreamento `RxDrop`.

Finalmente, dizemos ao simulador para sobrescrever qualquer `Applications` e parar o processamento de eventos em 20 segundos na simulação.

```
  Simulator::Stop (Seconds(20));
  Simulator::Run ();
  Simulator::Destroy ();

  return 0;
}
```

Lembre-se que quando `Simulator::Run` é chamado, a fase de configuração termina e a fase de simulação inicia. Todo o processo descrito anteriormente ocorre durante a chamada dessa função.

Após o retorno do `Simulator::Run`, a simulação é terminada e entramos na fase de finalização. Neste caso, `Simulator::Destroy` executa a tarefa pesada e nós apenas retornamos o código de sucesso.

7.3.5 Executando fifth.cc

O arquivo `fifth.cc` é distribuído no código fonte, no diretório `examples/tutorial`. Para executar:

```
./waf --run fifth
Waf: Entering directory `/home/craigdo/repos/ns-3-allinone-dev/ns-3-dev/build
Waf: Leaving directory `/home/craigdo/repos/ns-3-allinone-dev/ns-3-dev/build'
'build' finished successfully (0.684s)
1.20919 1072
1.21511 1608
1.22103 2144
...
1.2471  8040
1.24895 8576
1.2508  9112
RxDrop at 1.25151
...
```

Podemos observar o lado negativo de usar "prints" de qualquer tipo no rastreamento. Temos mensagens `waf` sendo impressas sobre a informação relevante. Vamos resolver esse problema, mas primeiro vamos verificar o resultado redirecionando a saída para um arquivo `cwnd.dat`:

```
./waf --run fifth > cwnd.dat 2>&1
```

Removemos as mensagens do `waf` e deixamos somente os dados rastreados. Pode-se também comentar as mensagens de "RxDrop...".

Agora podemos executar o gnuplot (se instalado) e gerar um gráfico:

```
gnuplot> set terminal png size 640,480
gnuplot> set output "cwnd.png"
gnuplot> plot "cwnd.dat" using 1:2 title 'Congestion Window' with linespoints
gnuplot> exit
```

Devemos obter um gráfico da janela de congestionamento pelo tempo no arquivo "cwnd.png", similar ao gráfico 7.1:

figure:: figures/cwnd.png

Gráfico da janela de congestionamento versus tempo.

7.3.6 Usando Auxiliares Intermediários

Na seção anterior, mostramos como adicionar uma origem do rastreamento e obter informações de interesse fora da simulação. Entretanto, no início do capítulo foi comentado que imprimir informações na saída padrão não é uma boa prática. Além disso, comentamos que não é interessante realizar processamento sobre a saída para isolar a informação

de interesse. Podemos pensar que perdemos muito tempo em um exemplo que apresenta todos os problemas que propomos resolver usando o sistema de rastreamento do *ns-3*. Você estaria correto, mas nós ainda não terminamos.

Uma da coisas mais importantes que queremos fazer é controlar a quantidade de saída da simulação. Nós podemos usar assistentes de rastreamento intermediários fornecido pelo *ns-3* para alcançar com sucesso esse objetivo.

Fornecemos um código que separa em arquivos distintos no disco os eventos de modificação da janela e os eventos de remoção. As alterações em cwnd são armazenadas em um arquivo ASCII separadas por TAB e os eventos de remoção são armazenados em um arquivo *pcap*. As alterações para obter esse resultado são pequenas.

Analisando sixth.cc

Vamos verificar as mudanças do arquivo `fifth.cc` para o `sixth.cc`. Verificamos a primeira mudança em `CwndChange`. Notamos que as assinaturas para o destino do rastreamento foram alteradas e que foi adicionada uma linha para cada um que escreve a informação rastreada para um fluxo (*stream*) representando um arquivo

```
static void
CwndChange (Ptr<OutputStreamWrapper> stream, uint32_t oldCwnd, uint32_t newCwnd)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);
  *stream->GetStream () << Simulator::Now ().GetSeconds () << "\t"
              << oldCwnd << "\t" << newCwnd << std::endl;
}

static void
RxDrop (Ptr<PcapFileWrapper> file, Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
  file->Write(Simulator::Now(), p);
}
```

Um parâmetro "stream" foi adicionado para o destino do rastreamento `CwndChange`. Este é um objeto que armazena (mantém seguramente vivo) um fluxo de saída em C++. Isto resulta em um objeto muito simples, mas que gerência problemas no ciclo de vida para fluxos e resolve um problema que mesmo programadores experientes de C++ tem dificuldades. Resulta que o construtor de cópia para o fluxo de saída (*ostream*) é marcado como privado. Isto significa que fluxos de saída não seguem a semântica de passagem por valor e não podem ser usados em mecanismos que necessitam que o fluxo seja copiado. Isto inclui o sistema de *callback* do *ns-3*. Além disso, adicionamos a seguinte linha:

```
*stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" << oldCwnd
              << "\t" << newCwnd << std::endl;
```

que substitui `std::cout` por `*stream->GetStream ()`

```
std::cout << Simulator::Now ().GetSeconds () << "\t" << oldCwnd << "\t" <<
              newCwnd << std::endl;
```

Isto demostra que o `Ptr<OutputStreamWrapper>` está apenas encapsulando um `std::ofstream`, logo pode ser usado como qualquer outro fluxo de saída.

Uma situação similar ocorre em `RxDrop`, exceto que o objeto passado (`Ptr<PcapFileWrapper>`) representa um arquivo pcap. Há uma linha no *trace sink* para escrever um marcador de tempo (*timestamp*) eo conteúdo do pacote perdido para o arquivo pcap.

```
file->Write(Simulator::Now(), p);
```

É claro, se nós temos objetos representando os dois arquivos, precisamos criá-los em algum lugar e também passá-los aos *trace sinks*. Se observarmos a função `main`, temos o código:

```
AsciiTraceHelper asciiTraceHelper;
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("sixth.cwnd");
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow",
              MakeBoundCallback (&CwndChange, stream));

...

PcapHelper pcapHelper;
Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap",
              std::ios::out, PcapHelper::DLT_PPP);
devices.Get (1)->TraceConnectWithoutContext("PhyRxDrop",
              MakeBoundCallback (&RxDrop, file));
```

Na primeira seção do código, criamos o arquivo de rastreamento ASCII e o objeto responsável para gerenciá-lo. Em seguida, usando uma das formas da função para criação da *callback* permitimos o objeto ser passado para o destino do rastreamento. As classes assistentes para rastreamento ASCII fornecem um vasto conjunto de funções para facilitar a manipulação de arquivos texto. Neste exemplo, focamos apenas na criação do arquivo para o fluxo de saída.

A função `CreateFileStream()` instancia um objeto `std::ofstream` e cria um novo arquivo. O fluxo de saída `ofstream` é encapsulado em um objeto do *ns-3* para gerenciamento do ciclo de vida e para resolver o problema do construtor de cópia.

Então pegamos o objeto que representa o arquivo e passamos para `MakeBoundCallback()`. Esta função cria um *callback* como `MakeCallback()`, mas "associa" um novo valor para o *callback*. Este valor é adicionado ao *callback* antes de sua invocação.

Essencialmente, `MakeBoundCallback(&CwndChange, stream)` faz com que a origem do rastreamento adicione um parâmetro extra "fluxo" após a lista formal de parâmetros antes de invocar o *callback*. Esta mudança está de acordo com o apresentado anteriormente, a qual inclui o parâmetro `Ptr<OutputStreamWrapper> stream`.

Na segunda seção de código, instanciamos um `PcapHelper` para fazer a mesma coisa para o arquivo de rastreamento pcap. A linha de código,

```
Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap", "w",
              PcapHelper::DLT_PPP);
```

cria um arquivo pcap chamado "sixth.pcap" no modo "w" (escrita). O parâmetro final é o "tipo da ligação de dados" do arquivo pcap. As opções estão definidas em `bpf.h`. Neste caso, `DLT_PPP` indica que o arquivo pcap deverá conter pacotes prefixado com cabeçalhos ponto-a-ponto. Isto é verdade pois os pacotes estão chegando de nosso *driver* de dispositivo ponto-a-ponto. Outros tipos de ligação de dados comuns são DLT_EN10MB (10 MB Ethernet) apropriado para dispositivos CSMA e DLT_IEEE802_11 (IEEE 802.11) apropriado para dispositivos sem fio. O arquivo `src/network/helper/trace-helper.h"` define uma lista com os tipos. As entradas na lista são idênticas as definidas em `bpf.h`, pois foram duplicadas para evitar um dependência com o pcap.

Um objeto *ns-3* representando o arquivo pcap é retornado de `CreateFile` e usado em uma *callback* exatamente como no caso ASCII.

É importante observar que ambos objetos são declarados de maneiras muito similares,

```
Ptr<PcapFileWrapper> file ...
Ptr<OutputStreamWrapper> stream ...
```

Mas os objetos internos são inteiramente diferentes. Por exemplo, o Ptr<PcapFileWrapper> é um ponteiro para um objeto *ns-3* que suporta `Attributes` e é integrado dentro do sistema de configuração. O Ptr<OutputStreamWrapper>, por outro lado, é um ponteiro para uma referência para um simples objeto contado. Lembre-se sempre de analisar o objeto que você está referenciando antes de fazer suposições sobre os "poderes" que o objeto pode ter.

Por exemplo, acesse o arquivo `src/network/utils/pcap-file-wrapper.h` e observe,

```
class PcapFileWrapper : public Object
```

que a classe `PcapFileWrapper` é um `Object` *ns-3* por herança. Já no arquivo `src/network/model/output-stream-wrapper.h`, observe,

```
class OutputStreamWrapper : public SimpleRefCount<OutputStreamWrapper>
```

que não é um `Object` *ns-3*, mas um objeto C++ que suporta contagem de referência.

A questão é que se você tem um Ptr<alguma_coisa>, não necessariamente significa que "alguma_coisa" é um `Object` *ns-3*, no qual você pode modificar `Attributes`, por exemplo.

Voltando ao exemplo. Se compilarmos e executarmos o exemplo,

```
./waf --run sixth
```

Veremos as mesmas mensagens do "fifth", mas dois novos arquivos aparecerão no diretório base de sua distribuição do *ns-3*.

```
sixth.cwnd  sixth.pcap
```

Como "sixth.cwnd" é um arquivo texto ASCII, você pode visualizar usando *cat* ou um editor de texto.

```
1.20919 536     1072
1.21511 1072    1608
...
9.30922 8893    8925
9.31754 8925    8957
```

Cada linha tem um marcador de tempo, o valor da janela de congestionamento e o valor da nova janela de congestionamento separados por tabulação, para importar diretamente para seu programa de plotagem de gráficos. Não há nenhuma outra informação além da rastreada, logo não é necessário processamento ou edição do arquivo.

Como "sixth.pcap" é um arquivo pcap, você pode visualizar usando o `tcpdump` ou `wireshark`.

```
reading from file ../../sixth.pcap, link-type PPP (PPP)
1.251507 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 17689:18225(536) ack 1 win 65535
1.411478 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 33808:34312(504) ack 1 win 65535
...
7.393557 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 781568:782072(504) ack 1 win 65535
8.141483 IP 10.1.1.1.49153 > 10.1.1.2.8080: . 874632:875168(536) ack 1 win 65535
```

Você tem um arquivo pcap com os pacotes que foram descartados na simulação. Não há nenhum outro pacote presente no arquivo e nada mais para dificultar sua análise.

Foi uma longa jornada, mas agora entendemos porque o sistema de rastreamento é interessante. Nós obtemos e armazenamos importantes eventos da implementação do TCP e do *driver* de dispositivo. E não modificamos nenhuma linha do código do núcleo do *ns-3*, e ainda fizemos isso com apenas 18 linhas de código:

```
static void
CwndChange (Ptr<OutputStreamWrapper> stream, uint32_t oldCwnd, uint32_t newCwnd)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);
  *stream->GetStream () << Simulator::Now ().GetSeconds () << "\t" <<
              oldCwnd << "\t" << newCwnd << std::endl;
}

...

AsciiTraceHelper asciiTraceHelper;
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("sixth.cwnd");
```

```
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow",
              MakeBoundCallback (&CwndChange, stream));

...

static void
RxDrop (Ptr<PcapFileWrapper> file, Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
  file->Write(Simulator::Now(), p);
}

...

PcapHelper pcapHelper;
Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap", "w",
              PcapHelper::DLT_PPP);
devices.Get (1)->TraceConnectWithoutContext("PhyRxDrop",
              MakeBoundCallback (&RxDrop, file));
```

7.4 Usando Classes Assistentes para Rastreamento

As classes assistentes (*trace helpers*) de rastreamento do *ns-3* proveem um ambiente rico para configurar, selecionar e escrever diferentes eventos de rastreamento para arquivos. Nas seções anteriores, primeiramente em "Construindo Topologias", nós vimos diversas formas de métodos assistentes para rastreamento projetados para uso dentro de outras classes assistentes.

Segue alguns desses métodos já estudados:

```
pointToPoint.EnablePcapAll ("second");
pointToPoint.EnablePcap ("second", p2pNodes.Get (0)->GetId (), 0);
csma.EnablePcap ("third", csmaDevices.Get (0), true);
pointToPoint.EnableAsciiAll (ascii.CreateFileStream ("myfirst.tr"));
```

O que não parece claro é que há um modelo consistente para todos os métodos relacionados à rastreamento encontrados no sistema. Apresentaremos uma visão geral desse modelo.

Há dois casos de uso primários de classes assistentes em *ns-3*: Classes assistentes de dispositivo e classes assistentes de protocolo. Classes assistentes de dispositivo tratam o problema de especificar quais rastreamentos deveriam ser habilitados no domínio do nó de rede. Por exemplo, poderíamos querer especificar que o rastreamento pcap deveria ser ativado em um dispositivo particular de um nó específico. Isto é o que define o modelo conceitual de dispositivo no *ns-3* e também os modelos conceituais de várias classes assistentes de dispositivos. Baseado nisso, os arquivos criados seguem a convenção de nome *<prefixo>-<nó>-<dispositivo>*.

As classes assistentes de protocolos tratam o problema de especificar quais rastreamentos deveriam ser ativados no protocolo e interface. Isto é definido pelo modelo conceitual de pilha de protocolo do *ns-3* e também pelos modelos conceituais de classes assistentes de pilha de rede. Baseado nisso, os arquivos criados seguem a convenção de nome *<prefixo>-<protocolo>-<interface>*.

As classes assistentes consequentemente encaixam-se em uma taxinomia bi-dimensional. Há pequenos detalhes que evitam todas as classes comportarem-se da mesma forma, mas fizemos parecer que trabalham tão similarmente quanto possível e quase sempre há similares para todos métodos em todas as classes.

```
                                                    | pcap | ascii |
----------------------------------------------------+------+-------|
Classe Assistente de Dispositivo (*Device Helper*)   |      |       |
```

```
-------------------------------------------------+------+-------|
Classe Assistente de Protocolo (*Protocol Helper*)   |      |       |
-------------------------------------------------+------+-------|
```

Usamos uma abordagem chamada `mixin` para adicionar funcionalidade de rastreamento para nossas classes assistentes. Uma `mixin` é uma classe que provê funcionalidade para aquela que é herdada por uma subclasse. Herdar de um `mixin` não é considerado uma forma de especialização mas é realmente uma maneira de colecionar funcionalidade.

Vamos verificar rapidamente os quatro casos e seus respectivos `mixins`.

7.4.1 Classes Assistentes de Dispositivo para Rastreamento Pcap

O objetivo destes assistentes é simplificar a adição de um utilitário de rastreamento pcap consistente para um dispositivo *ns-3*. Queremos que opere da mesma forma entre todos os dispositivos, logo os métodos destes assistentes são herdados por classes assistentes de dispositivo. Observe o arquivo `src/network/helper/trace-helper.h` para entender a discussão do código a seguir.

A classe `PcapHelperForDevice` é um `mixin` que provê a funcionalidade de alto nível para usar rastreamento pcap em um dispositivo *ns-3*. Todo dispositivo deve implementar um único método virtual herdado dessa classe.

```
virtual void EnablePcapInternal (std::string prefix, Ptr<NetDevice> nd,
              bool promiscuous, bool explicitFilename) = 0;
```

A assinatura deste método reflete a visão do dispositivo da situação neste nível. Todos os métodos públicos herdados da classe `PcapUserHelperForDevice` são reduzidos a chamada da implementação deste simples método dependente de dispositivo. Por exemplo, o nível mais baixo do método pcap,

```
void EnablePcap (std::string prefix, Ptr<NetDevice> nd, bool promiscuous = false,
              bool explicitFilename = false);
```

chamaremos diretamente a implementação do dispositivo de `EnablePcapInternal`. Todos os outros métodos de rastreamento pcap públicos desta implementação são para prover funcionalidade adicional em nível de usuário. Para o usuário, isto significa que todas as classes assistentes de dispositivo no sistema terão todos os métodos de rastreamento pcap disponíveis; e estes métodos trabalharão da mesma forma entre dispositivos se o dispositivo implementar corretamente `EnablePcapInternal`.

Métodos da Classe Assistente de Dispositivo para Rastreamento Pcap

```
void EnablePcap (std::string prefix, Ptr<NetDevice> nd,
              bool promiscuous = false, bool explicitFilename = false);
void EnablePcap (std::string prefix, std::string ndName,
              bool promiscuous = false, bool explicitFilename = false);
void EnablePcap (std::string prefix, NetDeviceContainer d,
              bool promiscuous = false);
void EnablePcap (std::string prefix, NodeContainer n,
              bool promiscuous = false);
void EnablePcap (std::string prefix, uint32_t nodeid, uint32_t deviceid,
              bool promiscuous = false);
void EnablePcapAll (std::string prefix, bool promiscuous = false);
```

Em cada método apresentado existe um parâmetro padrão chamado `promiscuous` que é definido para o valor "false". Este parâmetro indica que o rastreamento não deveria coletar dados em modo promíscuo. Se quisermos incluir todo tráfego visto pelo dispositivo devemos modificar o valor para "true". Por exemplo,

```
Ptr<NetDevice> nd;
...
helper.EnablePcap ("prefix", nd, true);
```

ativará o modo de captura promíscuo no `NetDevice` especificado por `nd`.

Os dois primeiros métodos também incluem um parâmetro padrão chamado `explicitFilename` que será abordado a seguir.

É interessante procurar maiores detalhes dos métodos da classe `PcapHelperForDevice` no Doxygen; mas para resumir ...

Podemos ativar o rastreamento pcap em um par nó/dispositivo-rede específico provendo um `Ptr<NetDevice>` para um método `EnablePcap`. O `Ptr<Node>` é implícito, pois o dispositivo de rede deve estar em um `Node`. Por exemplo,

```
Ptr<NetDevice> nd;
...
helper.EnablePcap ("prefix", nd);
```

Podemos ativar o rastreamento pcap em um par nó/dispositivo-rede passando uma `std::string` que representa um nome de serviço para um método `EnablePcap`. O `Ptr<NetDevice>` é buscado a partir do nome da *string*. Novamente, o `Ptr<Node>` é implícito pois o dispositivo de rede deve estar em um `Node`.

```
Names::Add ("server" ...);
Names::Add ("server/eth0" ...);
...
helper.EnablePcap ("prefix", "server/eth0");
```

Podemos ativar o rastreamento pcap em uma coleção de pares nós/dispositivos usando um `NetDeviceContainer`. Para cada `NetDevice` no contêiner o tipo é verificado. Para cada dispositivo com o tipo adequado, o rastreamento será ativado. Por exemplo,

```
NetDeviceContainer d = ...;
...
helper.EnablePcap ("prefix", d);
```

Podemos ativar o rastreamento em uma coleção de pares nó/dispositivo-rede usando um `NodeContainer`. Para cada `Node` no `NodeContainer` seus `NetDevices` são percorridos e verificados segundo o tipo. Para cada dispositivo com o tipo adequado, o rastreamento é ativado.

```
NodeContainer n;
...
helper.EnablePcap ("prefix", n);
```

Podemos ativar o rastreamento pcap usando o número identificador (*ID*) do nó e do dispositivo. Todo `Node` no sistema tem um valor inteiro indicando o *ID* do nó e todo dispositivo conectado ao nó tem um valor inteiro indicando o *ID* do dispositivo.

```
helper.EnablePcap ("prefix", 21, 1);
```

Por fim, podemos ativar rastreamento pcap para todos os dispositivos no sistema, desde que o tipo seja o mesmo gerenciado pela classe assistentes de dispositivo.

```
helper.EnablePcapAll ("prefix");
```

Seleção de um Nome de Arquivo para o Rastreamento Pcap da Classe Assistente de Dispositivo

Implícito nas descrições de métodos anteriores é a construção do nome de arquivo por meio do método da implementação. Por convenção, rastreamento pcap no *ns-3* usa a forma "<prefixo>-<id do nó>-<id do dispositivo>.pcap"

Como mencionado, todo nó no sistema terá um *id* de nó associado; e todo dispositivo terá um índice de interface (também chamado de id do dispositivo) relativo ao seu nó. Por padrão, então, um arquivo pcap criado como um resultado de ativar rastreamento no primeiro dispositivo do nó 21 usando o prefixo "prefix" seria "prefix-21-1.pcap".

Sempre podemos usar o serviço de nome de objeto do *ns-3* para tornar isso mais claro. Por exemplo, se você usa o serviço para associar o nome "server" ao nó 21, o arquivo pcap resultante automaticamente será, "prefix-server-1.pcap" e se você também associar o nome "eth0" ao dispositivo, seu nome do arquivo pcap automaticamente será denominado "prefix-server-eth0.pcap".

Finalmente, dois dos métodos mostrados,

```
void EnablePcap (std::string prefix, Ptr<NetDevice> nd,
              bool promiscuous = false, bool explicitFilename = false);
void EnablePcap (std::string prefix, std::string ndName,
              bool promiscuous = false, bool explicitFilename = false);
```

tem um parâmetro padrão `explicitFilename`. Quando modificado para verdadeiro, este parâmetro desabilita o mecanismo automático de completar o nome do arquivo e permite criarmos um nome de arquivo abertamente. Esta opção está disponível nos métodos que ativam o rastreamento pcap em um único dispositivo.

Por exemplo, com a finalidade providenciar uma classe assistente de dispositivo para criar um único arquivo de captura pcap no modo promíscuo com um nome específico ("my-pcap-file.pcap") em um determinado dispositivo:

```
Ptr<NetDevice> nd;
...
helper.EnablePcap ("my-pcap-file.pcap", nd, true, true);
```

O primeiro parâmetro `true` habilita o modo de rastreamento promíscuo e o segundo faz com que o parâmetro `prefix` seja interpretado como um nome de arquivo completo.

7.4.2 Classes Assistentes de Dispositivo para Rastreamento ASCII

O comportamento do assistente de rastreamento ASCII `mixin` é similar a versão do pcap. Acesse o arquivo `src/network/helper/trace-helper.h` para compreender melhor o funcionamento dessa classe assistente.

A classe `AsciiTraceHelperForDevice` adiciona funcionalidade em alto nível para usar o rastreamento ASCII para uma classe assistente de dispositivo. Como no caso do pcap, todo dispositivo deve implementar um método herdado do rastreador ASCII `mixin`.

```
virtual void EnableAsciiInternal (Ptr<OutputStreamWrapper> stream,
              std::string prefix, Ptr<NetDevice> nd, bool explicitFilename) = 0;
```

A assinatura deste método reflete a visão do dispositivo da situação neste nível; e também o fato que o assistente pode ser escrito para um fluxo de saída compartilhado. Todos os métodos públicos associados ao rastreamento ASCII herdam da classe `AsciiTraceHelperForDevice` resumem-se a chamada deste único método dependente de implementação. Por exemplo, os métodos de rastreamento ASCII de mais baixo nível,

```
void EnableAscii (std::string prefix, Ptr<NetDevice> nd,
              bool explicitFilename = false);
void EnableAscii (Ptr<OutputStreamWrapper> stream, Ptr<NetDevice> nd);
```

chamarão uma implementação de `EnableAsciiInternal` diretamente, passando um prefixo ou fluxo válido. Todos os outros métodos públicos serão construídos a partir destas funções de baixo nível para fornecer funcionalidades

adicionais em nível de usuário. Para o usuário, isso significa que todos os assistentes de dispositivo no sistema terão todos os métodos de rastreamento ASCII disponíveis e estes métodos trabalharão do mesmo modo em todos os dispositivos se estes implementarem `EnableAsciiInternal`.

Métodos da Classe Assistente de Dispositivo para Rastreamento ASCII

```
void EnableAscii (std::string prefix, Ptr<NetDevice> nd,
              bool explicitFilename = false);
void EnableAscii (Ptr<OutputStreamWrapper> stream, Ptr<NetDevice> nd);

void EnableAscii (std::string prefix, std::string ndName,
              bool explicitFilename = false);
void EnableAscii (Ptr<OutputStreamWrapper> stream, std::string ndName);

void EnableAscii (std::string prefix, NetDeviceContainer d);
void EnableAscii (Ptr<OutputStreamWrapper> stream, NetDeviceContainer d);

void EnableAscii (std::string prefix, NodeContainer n);
void EnableAscii (Ptr<OutputStreamWrapper> stream, NodeContainer n);

void EnableAsciiAll (std::string prefix);
void EnableAsciiAll (Ptr<OutputStreamWrapper> stream);

void EnableAscii (std::string prefix, uint32_t nodeid, uint32_t deviceid,
              bool explicitFilename);
void EnableAscii (Ptr<OutputStreamWrapper> stream, uint32_t nodeid,
              uint32_t deviceid);
```

Para maiores detalhes sobre os métodos é interessante consultar a documentação para a classe `AsciiTraceHelperForDevice`; mas para resumir ...

Há duas vezes mais métodos disponíveis para rastreamento ASCII que para rastreamento pcap. Isto ocorre pois para o modelo pcap os rastreamentos de cada par nó/dispositivo-rede são escritos para um único arquivo, enquanto que no ASCII todo as as informações são escritas para um arquivo comum. Isto significa que o mecanismo de geração de nomes de arquivos *<prefixo>-<nó>-<dispositivo>* é substituído por um mecanismo para referenciar um arquivo comum; e o número de métodos da API é duplicado para permitir todas as combinações.

Assim como no rastreamento pcap, podemos ativar o rastreamento ASCII em um par nó/dispositivo-rede passando um `Ptr<NetDevice>` para um método `EnableAscii`. O `Ptr<Node>` é implícito pois o dispositivo de rede deve pertencer a exatamente um `Node`. Por exemplo,

```
Ptr<NetDevice> nd;
...
helper.EnableAscii ("prefix", nd);
```

Os primeiros quatro métodos também incluem um parâmetro padrão `explicitFilename` que opera similar aos parâmetros no caso do pcap.

Neste caso, nenhum contexto de rastreamento é escrito para o arquivo ASCII pois seriam redundantes. O sistema pegará o nome do arquivo para ser criado usando as mesmas regras como descritas na seção pcap, exceto que o arquivo terá o extensão ".tr" ao invés de ".pcap".

Para habilitar o rastreamento ASCII em mais de um dispositivo de rede e ter todos os dados de rastreamento enviados para um único arquivo, pode-se usar um objeto para referenciar um único arquivo. Nós já verificamos isso no exemplo "cwnd":

```
Ptr<NetDevice> nd1;
Ptr<NetDevice> nd2;
```

```
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream
              ("trace-file-name.tr");
...
helper.EnableAscii (stream, nd1);
helper.EnableAscii (stream, nd2);
```

Neste caso, os contextos são escritos para o arquivo ASCII quando é necessário distinguir os dados de rastreamento de dois dispositivos. É interessante usar no nome do arquivo a extensão ".tr" por motivos de consistência.

Podemos habilitar o rastreamento ASCII em um par nó/dispositivo-rede específico passando ao método `EnableAscii` uma `std::string` representando um nome no serviço de nomes de objetos. O `Ptr<NetDevice>` é obtido a partir do nome. Novamente, o `<Node>` é implícito pois o dispositivo de rede deve pertencer a exatamente um `Node`. Por exemplo,

```
Names::Add ("client" ...);
Names::Add ("client/eth0" ...);
Names::Add ("server" ...);
Names::Add ("server/eth0" ...);
...
helper.EnableAscii ("prefix", "client/eth0");
helper.EnableAscii ("prefix", "server/eth0");
```

Isto resultaria em dois nomes de arquivos - "prefix-client-eth0.tr" e "prefix-server-eth0.tr" - com os rastreamentos de cada dispositivo em seu arquivo respectivo. Como todas as funções do `EnableAscii` são sobrecarregadas para suportar um *stream wrapper*, podemos usar da seguinte forma também:

```
Names::Add ("client" ...);
Names::Add ("client/eth0" ...);
Names::Add ("server" ...);
Names::Add ("server/eth0" ...);
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream
              ("trace-file-name.tr");
...
helper.EnableAscii (stream, "client/eth0");
helper.EnableAscii (stream, "server/eth0");
```

Isto resultaria em um único arquivo chamado "trace-file-name.tr" que contém todosos eventos rastreados para ambos os dispositivos. Os eventos seriam diferenciados por *strings* de contexto.

Podemos habilitar o rastreamento ASCII em um coleção de pares nó/dispositivo-rede fornecendo um `NetDeviceContainer`. Para cada `NetDevice` no contêiner o tipo é verificado. Para cada dispositivo de um tipo adequado (o mesmo tipo que é gerenciado por uma classe assistente de dispositivo), o rastreamento é habilitado. Novamente, o `<Node>` é implícito pois o dispositivo de rede encontrado deve pertencer a exatamente um `Node`.

```
NetDeviceContainer d = ...;
...
helper.EnableAscii ("prefix", d);
```

Isto resultaria em vários arquivos de rastreamento ASCII sendo criados, cada um seguindo a convenção `<prefixo>-<id do nó>-<id do dispositivo>.tr`.

Para obtermos um único arquivo teríamos:

```
NetDeviceContainer d = ...;
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream
              ("trace-file-name.tr");
```

```
...
helper.EnableAscii (stream, d);
```

Podemos habilitar o rastreamento ASCII em um coleção de pares nó/dispositivo-rede fornecendo um `NodeContainer`. Para cada `Node` no `NodeContainer`, os seus `NetDevices` são percorridos. Para cada `NetDevice` associado a cada nó no contêiner, o tipo do dispositivo é verificado. Para cada dispositivo do tipo adequado (o mesmo tipo que é gerenciado pelo assistente de dispositivo), o rastreamento é habilitado.

```
NodeContainer n;
...
helper.EnableAscii ("prefix", n);
```

Isto resultaria em vários arquivos ASCII sendo criados, cada um seguindo a convenção `<prefixo>-<id do nó>-<id do dispositivo>.tr`.

Podemos habilitar o rastreamento pcap na base da *ID* do nó e *ID* do dispositivo tão bem como com um `Ptr`. Cada `Node` no sistema possui um número identificador inteiro associado ao nó e cada dispositivo conectado possui um número identificador inteiro associado ao dispositivo.

```
helper.EnableAscii ("prefix", 21, 1);
```

Os rastreamentos podem ser combinados em um único arquivo como mostrado acima.

Finalmente, podemos habilitar o rastreamento ASCII para todos os dispositivos no sistema.

```
helper.EnableAsciiAll ("prefix");
```

Isto resultaria em vários arquivos ASCII sendo criados, um para cada dispositivo no sistema do tipo gerenciado pelo assistente. Todos estes arquivos seguiriam a convenção `<prefixo>-<id do nó>-<id do dispositivo>.tr`.

Selecionando Nome de Arquivo para as Saídas ASCII

Implícito nas descrições de métodos anteriores é a construção do nome de arquivo por meio do método da implementação. Por convenção, rastreamento ASCII no *ns-3* usa a forma "`<prefixo>-<id do nó>-<id do dispositivo>.tr`".

Como mencionado, todo nó no sistema terá um *id* de nó associado; e todo dispositivo terá um índice de interface (também chamado de id do dispositivo) relativo ao seu nó. Por padrão, então, um arquivo ASCII criado como um resultado de ativar rastreamento no primeiro dispositivo do nó 21 usando o prefixo "prefix" seria "prefix-21-1.tr".

Sempre podemos usar o serviço de nome de objeto do *ns-3* para tornar isso mais claro. Por exemplo, se usarmos o serviço para associar o nome `server` ao nó 21, o arquivo ASCII resultante automaticamente será, `prefix-server-1.tr` e se também associarmos o nome `eth0` ao dispositivo, o nome do arquivo ASCII automaticamente será denominado `prefix-server-eth0.tr`.

Diversos métodos tem um parâmetro padrão `explicitFilename`. Quando modificado para verdadeiro, este parâmetro desabilita o mecanismo automático de completar o nome do arquivo e permite criarmos um nome de arquivo abertamente. Esta opção está disponível nos métodos que possuam um prefixo e ativem o rastreamento em um único dispositivo.

7.4.3 Classes Assistentes de Protocolo para Rastreamento Pcap

O objetivo destes `mixins` é facilitar a adição de um mecanismo consistente para da facilidade de rastreamento para protocolos. Queremos que todos os mecanismos de rastreamento para todos os protocolos operem de mesma forma, logo os métodos dessas classe assistentes são herdados por assistentes de pilha. Acesse `src/network/helper/trace-helper.h` para acompanhar o conteúdo discutido nesta seção.

Nesta seção ilustraremos os métodos aplicados ao protocolo `Ipv4`. Para especificar rastreamentos em protocolos similares, basta substituir pelo tipo apropriado. Por exemplo, use um `Ptr<Ipv6>` ao invés de um `Ptr<Ipv4>` e chame um `EnablePcapIpv6` ao invés de `EnablePcapIpv4`.

A classe `PcapHelperForIpv4` provê funcionalidade de alto nível para usar rastreamento no protocolo `Ipv4`. Cada classe assistente de protocolo devem implementar um método herdado desta. Haverá uma implementação separada para `Ipv6`, por exemplo, mas a diferença será apenas nos nomes dos métodos e assinaturas. Nomes de métodos diferentes são necessário para distinguir a classe `Ipv4` da `Ipv6`, pois ambas são derivadas da classe `Object`, logo os métodos compartilham a mesma assinatura.

```
virtual void EnablePcapIpv4Internal (std::string prefix, Ptr<Ipv4> ipv4,
              uint32_t interface, bool explicitFilename) = 0;
```

A assinatura desse método reflete a visão do protocolo e interface da situação neste nível. Todos os métodos herdados da classe `PcapHelperForIpv4` resumem-se a chamada deste único método dependente de dispositivo. Por exemplo, o método do pcap de mais baixo nível,

```
void EnablePcapIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface,
              bool explicitFilename = false);
```

chamará a implementação de dispositivo de `EnablePcapIpv4Internal` diretamente. Todos os outros métodos públicos de rastreamento pcap são construídos a partir desta implementação para prover funcionalidades adicionais em nível do usuário. Para o usuário, isto significa que todas as classes assistentes de dispositivo no sistema terão todos os métodos de rastreamento pcap disponíveis; e estes métodos trabalharão da mesma forma entre dispositivos se o dispositivo implementar corretamente `EnablePcapIpv4Internal`.

Métodos da Classe Assistente de Protocolo para Rastreamento Pcap

Estes métodos são projetados para terem correspondência de um-para-um com o `Node` e `NetDevice`. Ao invés de restrições de pares `Node` e `NetDevice`, usamos restrições de protocolo e interface.

Note que como na versão de dispositivo, há seis métodos:

```
void EnablePcapIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface,
              bool explicitFilename = false);
void EnablePcapIpv4 (std::string prefix, std::string ipv4Name,
              uint32_t interface, bool explicitFilename = false);
void EnablePcapIpv4 (std::string prefix, Ipv4InterfaceContainer c);
void EnablePcapIpv4 (std::string prefix, NodeContainer n);
void EnablePcapIpv4 (std::string prefix, uint32_t nodeid, uint32_t interface,
              bool explicitFilename);
void EnablePcapIpv4All (std::string prefix);
```

Para maiores detalhes sobre estes métodos é interessante consultar na documentação da classe `PcapHelperForIpv4`, mas para resumir ...

Podemos habilitar o rastreamento pcap em um par protocolo/interface passando um `Ptr<Ipv4>` e `interface` para um método `EnablePcap`. Por exemplo,

```
Ptr<Ipv4> ipv4 = node->GetObject<Ipv4> ();
...
helper.EnablePcapIpv4 ("prefix", ipv4, 0);
```

Podemos ativar o rastreamento pcap em um par protocolo/interface passando uma `std::string` que representa um nome de serviço para um método `EnablePcapIpv4`. O `Ptr<Ipv4>` é buscado a partir do nome da *string*. Por exemplo,

```
Names::Add ("serverIPv4" ...);
...
helper.EnablePcapIpv4 ("prefix", "serverIpv4", 1);
```

Podemos ativar o rastreamento pcap em uma coleção de pares protocolo/interface usando um `Ipv4InterfaceContainer`. Para cada par‘‘Ipv4‘‘/interface no contêiner o tipo do protocolo é verificado. Para cada protocolo do tipo adequado, o rastreamento é ativado para a interface correspondente. Por exemplo,

```
NodeContainer nodes;
...
NetDeviceContainer devices = deviceHelper.Install (nodes);
...
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = ipv4.Assign (devices);
...
helper.EnablePcapIpv4 ("prefix", interfaces);
```

Podemos ativar o rastreamento em uma coleção de pares protocolo/interface usando um `NodeContainer`. Para cada `Node` no `NodeContainer` o protocolo apropriado é encontrado. Para cada protocolo, suas interfaces são enumeradas e o rastreamento é ativado nos pares resultantes. Por exemplo,

```
NodeContainer n;
...
helper.EnablePcapIpv4 ("prefix", n);
```

Pode ativar o rastreamento pcap usando o número identificador do nó e da interface. Neste caso, o *ID* do nó é traduzido para um `Ptr<Node>` e o protocolo apropriado é buscado no nó. O protocolo e interface resultante são usados para especificar a origem do rastreamento resultante.

```
helper.EnablePcapIpv4 ("prefix", 21, 1);
```

Por fim, podemos ativar rastreamento pcap para todas as interfaces no sistema, desde que o protocolo seja do mesmo tipo gerenciado pela classe assistente.

```
helper.EnablePcapIpv4All ("prefix");
```

Seleção de um Nome de Arquivo para o Rastreamento Pcap da Classe Assistente de Protocolo

Implícito nas descrições de métodos anterior é a construção do nome de arquivo por meio do método da implementação. Por convenção, rastreamento pcap no *ns-3* usa a forma `<prefixo>-<id do nó>-<id do dispositivo>.pcap`. No caso de rastreamento de protocolos, há uma correspondência de um-para-um entre protocolos e `Nodes`. Isto porque `Objects` de protocolo são agregados a *Node Objects‘*. Como não há um *id* global de protocolo no sistema, usamos o *ID* do nó na nomeação do arquivo. Consequentemente há possibilidade de colisão de nomes quando usamos o sistema automático de nomes. Por esta razão, a convenção de nome de arquivo é modificada para rastreamentos de protocolos.

Como mencionado, todo nó no sistema terá um *id* de nó associado. Como há uma correspondência de um-para-um entre instâncias de protocolo e instâncias de nó, usamos o *id* de nó. Cada interface tem um *id* de interface relativo ao seu protocolo. Usamos a convenção “<prefixo>-n<id do nó>-i<id da interface>.pcap” para especificar o nome do arquivo de rastreamento para as classes assistentes de protocolo.

Consequentemente, por padrão, uma arquivo pcap criado como um resultado da ativação de rastreamento na interface 1 do protocolo ipv4 do nó 21 usando o prefixo `prefix` seria `prefix-n21-i1.pcap`.

Sempre podemos usar o serviço de nomes de objetos do *ns-3* para tornar isso mais claro. Por exemplo, se usamos o serviço de nomes para associar o nome “serverIpv4” ao Ptr<Ipv4> no nó 21, o nome de arquivo resultante seria `prefix-nserverIpv4-i1.pcap`.

Diversos métodos tem um parâmetro padrão `explicitFilename`. Quando modificado para verdadeiro, este parâmetro desabilita o mecanismo automático de completar o nome do arquivo e permite criarmos um nome de arquivo abertamente. Esta opção está disponível nos métodos que ativam o rastreamento pcap em um único dispositivo.

7.4.4 Classes Assistentes de Protocolo para Rastreamento ASCII

O comportamento dos assistentes de rastreamento ASCII é similar ao do pcap. Acesse o arquivo `src/network/helper/trace-helper.h` para compreender melhor o funcionamento dessa classe assistente.

Nesta seção apresentamos os métodos aplicados ao protocolo `Ipv4`. Para protocolos similares apenas substitua para o tipo apropriado. Por exemplo, use um `Ptr<Ipv6>` ao invés de um `Ptr<Ipv4>` e chame `EnableAsciiIpv6` ao invés de `EnableAsciiIpv4`.

A classe `AsciiTraceHelperForIpv4` adiciona funcionalidade de alto nível para usar rastreamento ASCII para um assistente de protocolo. Todo protocolo que usa estes métodos deve implementar um método herdado desta classe.

```
virtual void EnableAsciiIpv4Internal (Ptr<OutputStreamWrapper> stream,
                                      std::string prefix,
                                      Ptr<Ipv4> ipv4,
                                      uint32_t interface,
                                      bool explicitFilename) = 0;
```

A assinatura deste método reflete a visão central do protocolo e interface da situação neste nível; e também o fato que o assistente pode ser escrito para um fluxo de saída compartilhado. Todos os métodos públicos herdados desta classe `PcapAndAsciiTraceHelperForIpv4` resumem-se a chamada deste único método dependente de implementação. Por exemplo, os métodos de rastreamento ASCII de mais baixo nível,

```
void EnableAsciiIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface,
              bool explicitFilename = false);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, Ptr<Ipv4> ipv4,
              uint32_t interface);
```

chamarão uma implementação de `EnableAsciiIpv4Internal` diretamente, passando um prefixo ou fluxo válido. Todos os outros métodos públicos serão construídos a partir destas funções de baixo nível para fornecer funcionalidades adicionais em nível de usuário. Para o usuário, isso significa que todos os assistentes de protocolos no sistema terão todos os métodos de rastreamento ASCII disponíveis e estes métodos trabalharão do mesmo modo em todos os protocolos se estes implementarem `EnableAsciiIpv4Internal`.

Métodos da Classe Assistente de Protocolo para Rastreamento ASCII

```
void EnableAsciiIpv4 (std::string prefix, Ptr<Ipv4> ipv4, uint32_t interface,
                      bool explicitFilename = false);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, Ptr<Ipv4> ipv4,
                      uint32_t interface);

void EnableAsciiIpv4 (std::string prefix, std::string ipv4Name, uint32_t interface,
                      bool explicitFilename = false);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, std::string ipv4Name,
                      uint32_t interface);

void EnableAsciiIpv4 (std::string prefix, Ipv4InterfaceContainer c);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, Ipv4InterfaceContainer c);

void EnableAsciiIpv4 (std::string prefix, NodeContainer n);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, NodeContainer n);
```

```
void EnableAsciiIpv4All (std::string prefix);
void EnableAsciiIpv4All (Ptr<OutputStreamWrapper> stream);

void EnableAsciiIpv4 (std::string prefix, uint32_t nodeid, uint32_t deviceid,
                      bool explicitFilename);
void EnableAsciiIpv4 (Ptr<OutputStreamWrapper> stream, uint32_t nodeid,
                      uint32_t interface);
```

Para maiores detalhes sobre os métodos consulte na documentação a classe `PcapAndAsciiHelperForIpv4`; mas para resumir ...

Há duas vezes mais métodos disponíveis para rastreamento ASCII que para rastreamento pcap. Isto ocorre pois para o modelo pcap os rastreamentos de cada par protocolo/interface são escritos para um único arquivo, enquanto que no ASCII todo as as informações são escritas para um arquivo comum. Isto significa que o mecanismo de geração de nomes de arquivos "<prefixo>-n<id do nó>-i<interface>" é substituído por um mecanismo para referenciar um arquivo comum; e o número de métodos da API é duplicado para permitir todas as combinações.

Assim, como no rastreamento pcap, podemos ativar o rastreamento ASCII em um par protocolo/interface passando um `Ptr<Ipv4>` e uma `interface` para um método `EnableAsciiIpv4`. Por exemplo,

```
Ptr<Ipv4> ipv4;
...
helper.EnableAsciiIpv4 ("prefix", ipv4, 1);
```

Neste caso, nenhum contexto de rastreamento é escrito para o arquivo ASCII pois seriam redundantes. O sistema pegará o nome do arquivo para ser criado usando as mesmas regras como descritas na seção pcap, exceto que o arquivo terá o extensão `.tr` ao invés de `.pcap`.

Para habilitar o rastreamento ASCII em mais de uma interface e ter todos os dados de rastreamento enviados para um único arquivo, pode-se usar um objeto para referenciar um único arquivo. Nós já verificamos isso no exemplo "cwnd":

```
Ptr<Ipv4> protocol1 = node1->GetObject<Ipv4> ();
Ptr<Ipv4> protocol2 = node2->GetObject<Ipv4> ();
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream
              ("trace-file-name.tr");
...
helper.EnableAsciiIpv4 (stream, protocol1, 1);
helper.EnableAsciiIpv4 (stream, protocol2, 1);
```

Neste caso, os contextos são escritos para o arquivo ASCII quando é necessário distinguir os dados de rastreamento de duas interfaces. É interessante usar no nome do arquivo a extensão `.tr` por motivos de consistência.

Pode habilitar o rastreamento ASCII em protocolo específico passando ao método `EnableAsciiIpv4` uma `std::string` representando um nome no serviço de nomes de objetos. O `Ptr<Ipv4>` é obtido a partir do nome. O `<Node>` é implícito, pois há uma correspondência de um-para-um entre instancias de protocolos e nós. Por exemplo,

```
Names::Add ("node1Ipv4" ...);
Names::Add ("node2Ipv4" ...);
...
helper.EnableAsciiIpv4 ("prefix", "node1Ipv4", 1);
helper.EnableAsciiIpv4 ("prefix", "node2Ipv4", 1);
```

Isto resultaria em dois nomes de arquivos `prefix-nnode1Ipv4-i1.tr` e `prefix-nnode2Ipv4-i1.tr`, com os rastreamentos de cada interface em seu arquivo respectivo. Como todas as funções do `EnableAsciiIpv4` são sobrecarregadas para suportar um *stream wrapper*, podemos usar da seguinte forma também:

```
Names::Add ("node1Ipv4" ...);
Names::Add ("node2Ipv4" ...);
...
```

```
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream
              ("trace-file-name.tr");
...
helper.EnableAsciiIpv4 (stream, "node1Ipv4", 1);
helper.EnableAsciiIpv4 (stream, "node2Ipv4", 1);
```

Isto resultaria em um único arquivo chamado `trace-file-name.tr` que contém todos os eventos rastreados para ambas as interfaces. Os eventos seriam diferenciados por *strings* de contexto.

Podemos habilitar o rastreamento ASCII em um coleção de pares protocolo/interface provendo um `Ipv4InterfaceContainer`. Para cada protocolo no contêiner o tipo é verificado. Para cada protocolo do tipo adequado (o mesmo tipo que é gerenciado por uma classe assistente de protocolo), o rastreamento é habilitado para a interface correspondente. Novamente, o `<Node>` é implícito, pois há uma correspondência de um-para-um entre protocolo e seu nó. Por exemplo,

```
NodeContainer nodes;
...
NetDeviceContainer devices = deviceHelper.Install (nodes);
...
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = ipv4.Assign (devices);
...
...
helper.EnableAsciiIpv4 ("prefix", interfaces);
```

Isto resultaria em vários arquivos de rastreamento ASCII sendo criados, cada um seguindo a convenção `<prefixo>-n<id do nó>-i<interface>.tr`.

Para obtermos um único arquivo teríamos:

```
NodeContainer nodes;
...
NetDeviceContainer devices = deviceHelper.Install (nodes);
...
Ipv4AddressHelper ipv4;
ipv4.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interfaces = ipv4.Assign (devices);
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream
              ("trace-file-name.tr");
...
helper.EnableAsciiIpv4 (stream, interfaces);
```

Podemos habilitar o rastreamento ASCII em uma coleção de pares protocolo/interface provendo um *NodeContainer*‘. Para cada `Node` no `NodeContainer` os protocolos apropriados são encontrados. Para cada protocolo, sua interface é enumerada e o rastreamento é habilitado nos pares. Por exemplo,

```
NodeContainer n;
...
helper.EnableAsciiIpv4 ("prefix", n);
```

Podemos habilitar o rastreamento pcap usando o número identificador do nó e número identificador do dispositivo. Neste caso, o *ID* do nó é traduzido para um `Ptr<Node>` e o protocolo apropriado é procurado no nó de rede. O protocolo e interface resultantes são usados para especificar a origem do rastreamento.

```
helper.EnableAsciiIpv4 ("prefix", 21, 1);
```

Os rastreamentos podem ser combinados em um único arquivo como mostrado anteriormente.

Finalmente, podemos habilitar o rastreamento ASCII para todas as interfaces no sistema.

```
helper.EnableAsciiIpv4All ("prefix");
```

Isto resultaria em vários arquivos ASCII sendo criados, um para cada interface no sistema relacionada ao protocolo do tipo gerenciado pela classe assistente.Todos estes arquivos seguiriam a convenção `<prefix>-n<id do node>-i<interface>.tr`.

Seleção de Nome de Arquivo para Rastreamento ASCII da Classe Assistente de Protocolo

Implícito nas descrições de métodos anteriores é a construção do nome do arquivo por meio do método da implementação. Por convenção, rastreamento ASCII no sistema *ns-3* são da forma `<prefix>-<id node>-<id do dispositivo>.tr`.

Como mencionado, todo nó no sistema terá um número identificador de nó associado. Como há uma correspondência de um-para-um entre instâncias de protocolo e instâncias de nó, usamos o *ID* de nó. Cada interface em um protocolo terá um índice de interface (também chamando apenas de interface) relativo ao seu protocolo. Por padrão, então, um arquivo de rastreamento ASCII criado a partir do rastreamento no primeiro dispositivo do nó 21, usando o prefixo "prefix", seria `prefix-n21-i1.tr`. O uso de prefixo distingue múltiplos protocolos por nó.

Sempre podemos usar o serviço de nomes de objetos do *ns-3* para tornar isso mais claro. Por exemplo, se usarmos o serviço de nomes para associar o nome "serverIpv4" ao Ptr<Ipv4> no nó 21, o nome de arquivo resultante seria `prefix-nserverIpv4-i1.tr`.

Diversos métodos tem um parâmetro padrão `explicitFilename`. Quando modificado para verdadeiro, este parâmetro desabilita o mecanismo automático de completar o nome do arquivo e permite criarmos um nome de arquivo abertamente. Esta opção está disponível nos métodos que ativam o rastreamento em um único dispositivo.

7.5 Considerações Finais

O *ns-3* inclui um ambiente completo para permitir usuários de diversos níveis personalizar os tipos de informação para serem extraídas de suas simulações.

Existem funções assistentes de alto nível que permitem ao usuário o controle de um coleção de saídas predefinidas para uma granularidade mais fina. Existem funções assistentes de nível intermediário que permitem usuários mais sofisticados personalizar como as informações são extraídas e armazenadas; e existem funções de baixo nível que permitem usuários avançados alterarem o sistema para apresentar novas ou informações que não eram exportadas.

Este é um sistema muito abrangente e percebemos que é muita informação para digerir, especialmente para novos usuários ou aqueles que não estão intimamente familiarizados com C++ e suas expressões idiomáticas. Consideramos o sistema de rastreamento uma parte muito importante do *ns-3*, assim recomendamos que familiarizem-se o máximo possível com ele. Compreender o restante do sistema *ns-3* é bem simples, uma vez que dominamos o sistema de rastreamento.

Conclusão

8.1 Para o futuro

Este documento é um trabalho em andamento. Espera-se que cresça com o tempo e cubra mais e mais funcionalidades do *ns-3*.

Para as próximas versões, os seguintes capítulos são esperados:

- O sistema de *callback*
- O sistema de objetos e o gerenciamento de memória
- O sistema de roteamento
- Adicionando novos dispositivos de rede e canais de comunicação
- Adicionando novos protocolos
- Trabalhando com redes e *hosts* reais

Escrever capítulos de manuais e tutoriais não é fácil, mas é muito importante para o projeto. Se você, leitor, é especialista em uma dessas áreas, por favor, considere contribuir com o *ns-3* escrevendo um desses capítulos; ou com qualquer outro capítulo que julgue importante.

8.2 Finalizando

O *ns-3* é um sistema grande e muito complexo. Por isto, é impossível cobrir todos os aspectos relevantes em um tutorial.

Ao leitor foi apresentada apenas uma introdução ao *ns-3*, entretanto, espera-se que tenha abordado o suficiente para o início de trabalhos e pesquisas relevantes com o simulador.

– Atenciosamente, equipe de desenvolvimento do *ns-3*.

Tradução

Traduzido para o português pelos alunos do programa de doutorado inter institucional do Instituto de Matemática e Estatística da Universidade de São Paulo — IME-USP em parceria com a Universidade Tecnológica Federal do Paraná - Câmpus Campo Mourão — UTFPR-CM:

- Frank Helbert (frank@ime.usp.br);

- Luiz Arthur Feitosa dos Santos (luizsan@ime.usp.br);
- Rodrigo Campiolo (campiolo@ime.usp.br).