GppCom

Grupo de Pesquisa em Prototipagem Rápida de Soluções para Comunicação.

DCO

Departamento de Engenharia de Comunicações

UFRN

Hands-On 04: O Sistema de Rastreamento

Jéssika Cristina da Silva
GppCom/DCO/UFRN

III Workshop Gppcom

Natal, 26/07/2018

Universidade Federal do Rio Grande do Norte (UFRN)

Objetivos

- Entender o funcionamento do sistema de rastreamento do ns-3, vantagens e objetivos.
- Analisar 2 implementações simples.
- Aprender a fazer o rastreamento.

Introdução

- 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 (ex: uso NS_LOG);

  -Desenvolver mecanismos de saída que resultam somente ou exatamente na informação pretendida;

- O rastreamento (Tracing) torna possível a segunda estratégia sem alteração do código fonte. Vantagens:

  - Redução da quantidade de dados para gerenciar;

  - O formato da saída pode ser controlado diretamente;

- O sistema de rastreamento do ns-3 é a melhor forma de se obter informações fora da simulação.

Introdução

Funciona a partir de 3 conceitos principais:

- **Source** (origem de rastreamento):

  Fornece acesso aos dados do sistema (ex: janela de congestionamento do modelo tcp/ip, momento em que um pacote é recebido)

  Mantém uma lista interna de “funções”.
- **Sink** (destino de rastreamento):

  Função que faz algo útil com os dados do sistema.
- **Conexão:**

  Função de destino é adicionada a lista interna de funções da origem.

  Conexão entre source e sink:
- **Sobrecarga:**

  Quando a variável de rastreamento é utilizada, todas as funções da lista interna da origem são chamadas!

  funções de destino recebem como parâmetro dados fornecidos pela origem.

Existem várias origens, mas só geram saída quando conectadas a algum destino!

Introdução

O sistema funciona através do uso de callbacks:

- Permite que parte de um código invoque uma função ou método de forma indireta.
- O endereço de chamada de uma função é tratada como uma variável (um ponteiro para função).
- Ex:

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

ao criar a função:

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

é possivel fazer pfi apontar para essa função:

```
pfi = MyFunction;
```

a função MyFunction é chamada de forma indireta:

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

Classes e Objetos

```
class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area (void){return width*height;};
};
int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

Sobrecarga de métodos

- Métodos com nomes iguais, mas argumentos diferentes.

```
class Ponto{
private:
int x, y;
public:
Ponto(){
  x=0;
  Y=0;
};
Ponto(int x1, int y1){
  x=x1;
  y=y1;
}
}
```

```
void main{
Ponto A();
Ponto B(2,3);
}
```

Resultado:

A.x= A.y=0

B.x=2 ; B.y=3

Sobrecarga de Operadores

- Redefinição de operadores já existentes através de um método.

```
class Ponto{
private:
int x, y;
public:
Ponto(){
   x=0;
   Y=0;
}
Ponto(int x1, int y1){
   x=x1;
   y=y1;
}
Ponto operador+ (Ponto a){
Ponto c;
c.x = a.x + x;
c.y = a.y + y
return c;
}}
```

```
Void main(){
Ponto A(1,2);
Ponto B(2,1);
Ponto x();
X = A + B;
}
```

Resultado:

X.x=3;

X.y=3;

Classes Genéricas com Templates

- Os tipos de dados usados pelos métodos e atributos da classe são parâmetros do template.
- Ao se instanciar um objeto da classe deve-se passar os parâmetros do template

```
template<typename
T1, typename T2>
Class Ponto{
public:
T1  x
T2  y;
public:
Ponto(T1 x1, T2 y1){
  x=x1;
  y=y1;
}
}
```

```
Void main(){
Ponto<int32_t, float>
A(2, 2.3);
}
```

Resultado:

A.x = 2, tipo inteiro.

A.y = 2.3, tipo floar

Sobrecarga de Métodos e Operadores + Templates

```
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
{
public:

  ...

  void operator() (T1 a1) const;
  void operator() (T1 a1, T2 a2) const;
  void operator() (T1 a1, T2 a2, T3 a3) const;
  void operator() (T1 a1, T2 a2, T3 a3, T4 a4) const;
  void operator() (T1 a1, T2 a2, T3 a3, T4 a4, T5 a5) const;
  void operator() (T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6) const;
  void operator() (T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6, T7 a7) const;
  void operator() (T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6, T7 a7, T8 a8) const;
```

Introdução

O sistema de rastreamento é integrado com Atributos e métodos.
Obrigatoriamente existe um “Object ns-3” para cada origem do rastreamento;

Origem:

- Mantém lista interna de callbacks;
- Possui um método operador ();
- Possui método de conexão;

Conexão:

- Funções de destino são adicionadas à lista de callbacks da origem;

Sobrecarga:

- Sempre que ocorre o evento de rastreamento o método operador() é chamado, que chama todas as funções da lista de callbacks.

Entendendo um exemplo simples de rastreamento

- Abra o arquivo fouth.cc:

```
#include "ns3/object.h"
#include "ns3/uinteger.h"
#include "ns3/traced-value.h"
#include "ns3/trace-source-accessor.h"
#include <iostream>
using namespace ns3;
class MyObject : public Object
{
public:
  /**
   * Register this type.
   * \return The TypeId.
   */
  static TypeId GetTypeId (void)
  {
    static TypeId tid = TypeId ("MyObject")
      .SetParent<Object> ()
      .SetGroupName ("Tutorial")
      .AddConstructor<MyObject> ()
      .AddTraceSource ("MyInteger",
                       "An integer value to trace.",
                       MakeTraceSourceAccessor (&MyObject::m_myInt),
                       "ns3::TracedValueCallback::Int32")
    ;
    return tid;
  }

  MyObject () {}
  TracedValue<int32_t> m_myInt;
};
void
IntTrace (int32_t oldValue, int32_t newValue)
{
  std::cout << "Traced " << oldValue << " to " << newValue << std::endl;
}
int
main (int argc, char *argv[])
{
  Ptr<MyObject> myObject = CreateObject<MyObject> ();
  myObject->TraceConnectWithoutContext ("MyInteger", MakeCallback (&IntTrace));

  myObject->m_myInt = 1234;
}
```

Módulos

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

#include <iostream>

using namespace ns3;
```

Declarando o objeto (1/2)

```
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;
};
```

Esse trecho do código declara e define uma classe.

Declarando o Objeto (2/2)

```
{
  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;
};
```

- AddTraceSource: define que a variável rastreada é m_myInt, com nome de identificação “My Integer”.
- Declaração:

  - A última linha define o atributo “m_myInt” como do tipo TracedValue<int32_t>.

  - TracedValue: classe, cujos métodos descritos nela podem gerar uma lista de callbacks, fazer conexão entre origem e destino e define função operador().

Destino de Rastreamento

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

- Esta definição corresponde diretamente a função de callback;
- Nós temos a origem e o destino do rastreamento. Agora, precisamos conectá-los.

Conexão

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

  myObject->m_myInt = 1234;
}
```

O objeto "myobject" da classe "MyObject é criado".

O método “TraceConnectWithoutContext” conecta a origem ao destino.

A função MakeCallback() cria o objeto callback associado a função de destino IntTrace()

O método “TraceConnectWithoutContext” adiciona esta callback à lista de callbacks da origem.

Conexão

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

- O atributo m_myint é igualado à 1234, é quando ocorre a sobrecarga de operadores

- O inteiro 1234 é passado como parâmetro da função operador();

- TracedValue para executa as callbacks da lista, retornando void e possuindo dois inteiros como parâmetros (um valor antigo e um novo valor). Isto é exatamente a assinatura da função para a função de callback que foi fornecida “IntTrace”.

Conexão

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

```
TracedValue<int32_t> m_myInt;
```

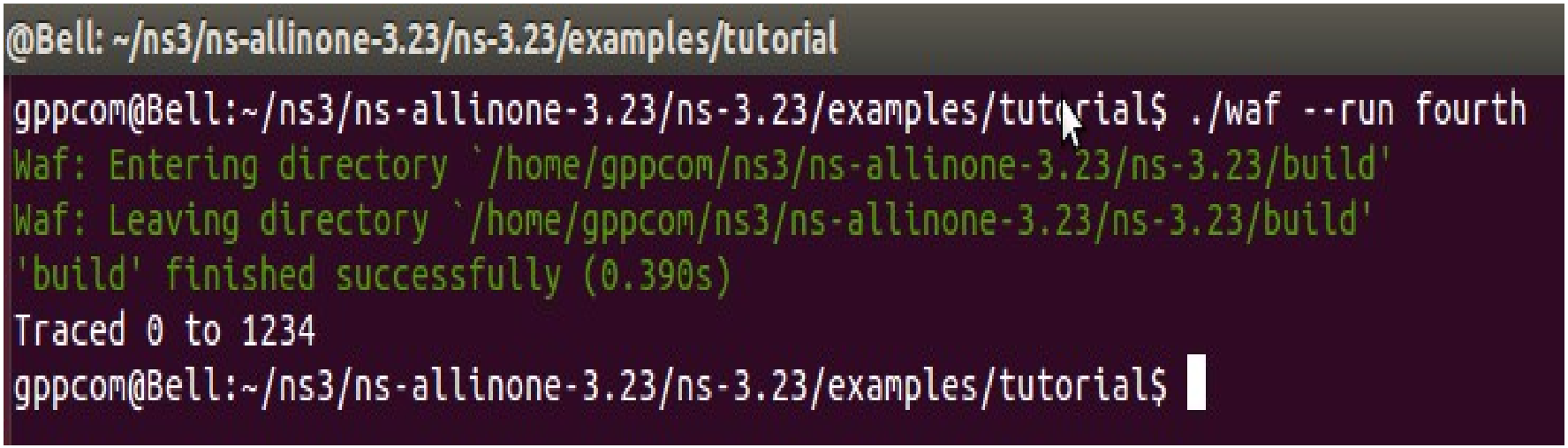
Os parâmetros definidos pela origem devem corresponder com os parâmetros de entrada das funções de destino. EX:

- m_myInt é do tipo tracedvalue.

- tracedvalue define callbacks com 2 parâmetros (um antigo e um novo) do valor de m_myInt (tipo int32_t).

- As funções de destino devem receber dois valores de entrada do tipo int32_t.

Saída no Terminal



Entendendo rastreamento no hands-on 3 em baixo nível.

- No hands-on 3 foi adicionada a seguinte função de destino:

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

- O destino foi conectado à origem CourseChange definida na classe MobilityModel:

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

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

A origem de rastreamento

- Em src/mobility/model/mobility-model.cc podemos encontrar uma definição de origem:

```
MobilityModel::GetTypeId (void)
{ ….
.AddTraceSource ("CourseChange",
                 "The value of the position and/or velocity vector changed",
                 MakeTraceSourceAccessor (&MobilityModel::m_courseChangeTrace),
                 "ns3::MobilityModel::TracedCallback");
} return tid;
```

- Atributo m_courseChangeTrace da classe MobilityModel é rastreado e identificado por “CourseChange”.

A origem de rastreamento

- O atributo m_courseChangeTrace está definido em Mobility_model.h como:

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

- Esta é uma declaração que utiliza templates;
- m_courseChangeTrace é do tipo TracedCallback;
- TracedCallback é uma classe genérica (utiliza templates) definida em traced_callback.h;

A origem de rastreamento

- Traced_callback.h: o código começa com a definição do template, definindo 8 possíveis tipos de argumentos.

```
 * \tparam T1 \explicit Type of the first argument to the functor.
 * \tparam T2 \explicit Type of the second argument to the functor.
 * \tparam T3 \explicit Type of the third argument to the functor.
 * \tparam T4 \explicit Type of the fourth argument to the functor.
 * \tparam T5 \explicit Type of the fifth argument to the functor.
 * \tparam T6 \explicit Type of the sixth argument to the functor.
 * \tparam T7 \explicit Type of the seventh argument to the functor.
 * \tparam T8 \explicit Type of the eighth argument to the functor.
 */
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
{
```

- Seus métodos e atributos podem utilizar quaisquer um desses 8 tipos.
- Ao se criar um objeto da classe TracedCallbak, deve-se passar os parâmetros do templates.
- Neste caso: T1 = Ptr<const Mobility Model>.

Os métodos e atributos da origem

- São declarados os métodos de conexão,o construtor e operador():

```
class TracedCallback
{
public:
  /** Constructor. */
  TracedCallback ();

  void ConnectWithoutContext (const CallbackBase & callback);

  void Connect (const CallbackBase & callback, std::string path);

  void DisconnectWithoutContext (const CallbackBase & callback);

  void operator() (void) const;

  void operator() (T1 a1) const;

  void operator() (T1 a1, T2 a2) const;

  void operator() (T1 a1, T2 a2, T3 a3) const;

  void operator() (T1 a1, T2 a2, T3 a3, T4 a4) const;

  void operator() (T1 a1, T2 a2, T3 a3, T4 a4, T5 a5) const;

  void operator() (T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6) const;

  void operator() (T1 a1, T2 a2, T3 a3, T4 a4, T5 a5, T6 a6, T7 a7) const;
```

Os métodos e atributos da origem

- Definição do tipo callbacklist;
- Criação do atributo do tipo callbacklist: m_callbacklist;
- m_callbacklist pode guardar uma lista de callbacks que retornam void e recebem os tipos T1 - T8.

```
private:


  typedef std::list<Callback<void,T1,T2,T3,T4,T5,T6,T7,T8> > CallbackList;

  CallbackList m_callbackList;
};
```

Os métodos e atributos da origem: conexão

- callback' guarda o endereço de uma callback base.
- Callback cb é criada de acordo com os tipos passados no template.
- Se cb e callback são correspondentes, callbackbase é adicionada à lista de callbacks: m_callbacklist.

```
void
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::ConnectWithoutContext (const CallbackBase & callback)
{
  Callback<void,T1,T2,T3,T4,T5,T6,T7,T8> cb;
  if (!cb.Assign (callback))
    NS_FATAL_ERROR_NO_MSG();
  m_callbackList.push_back (cb);
}
```

Os métodos e atributos da origem: conexão

- No caso analisado o que acontece é:

```
Void
TracedCallback<Ptr<const MobilityModel>::ConnectWithoutContext ...
{
Callback<void, Ptr<const MobilityModel> > cb;
If(!cb.Assign (callback));
NS_FATAL_ERROR_NO_MSG();
m_callbackList.push_back (cb);
}
```

- Se a callbackbase não receber o tipo Ptr<const MobilityModel> e não retornar void, o cb.Assign falhará.

Conexão

No momento da conexão, é chamado o método analisado anteriormente.

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

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

Os métodos e atributos da origem: sobrecarga de operadores

- As próximas definições são dos métodos operador():

```
void
TracedCallback<T1,T2,T3,T4,T5,T6,T7,T8>::operator() (T1 a1) const
{
  for (typename CallbackList::const_iterator i = m_callbackList.begin ();
       i != m_callbackList.end (); i++)
    {
      (*i)(a1);
    }
}
```

- Chama todas as callbacks presentes na lista
  - No nosso caso: T1 = ptr<const mobility model>
  - Se criarmos um objeto da classe MobilityModel e instanciarmos o atributo m_courseChangeTrace fazendo m_courseChangeTrace = algumobjeto
  - Então a1 = algumobjeto.

Sobrecarga de operadores

- O instanciamento do nosso caso é definido na própria classe MobilityModel.
- Em mobility_model.cc encontra-se:

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

- É passado como parâmetro o próprio objeto.
- Sempre que uma mudança de rota era realizada, havia uma chamada NotifyCourseChange()

O Destino

- A função de destino imprime as posições sempre que ocorre a sobrecarga de operadores.

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

Resumo

- Uma origem de 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 de rastreamento;
- A ação de “acionar” uma origem de rastreamento é executar um operador na origem que dispara as callbacks;

Isto resulta na execução das callbacks dos destinos registrados na origem, com os parâmetros providos pela origem.

Quais origens estão disponíveis no ns-3?

- A resposta é encontrada no Doxygen do ns-3.

  Link:https://www.nsnam.org/doxygen/

  Acesse: ns-3 Documentation > All trace Sources.

Sobre o GppCom

- **A meta do GppCom é criar na UFRN um ambiente de P&D&I através de prototipagem rápida baseada em simulação via software e hardware nas áreas de sistemas de comunicação e processamento digital de sinais e imagens. O Grupo é formado pelos professores: Vicente Angelo de Sousa Junior (coordenador), Luiz Gonzaga de Queiroz Silveira Junior (vice-coordenador), Luiz Felipe de Queiroz Silveira, Marcio Eduardo da Costa Rodrigues, Adaildo Gomes D'Assunção (pesquisador associado), Cláudio Rodrigues Muniz da Silva (pesquisador associado), Cristhianne de Fátima Linhares de Vasconcelos (pesquisador associado). O GppCom está de portas abertas para novas parcerias, conheça o portifolio do grupo.**

- **Contato:** vicente.gppcom@gmail.com