Tracing in NS-3

May, 2014

Walid Younes

Physical and Life Sciences

Lawrence Livermore National Laboratory

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Security, LLC, Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Chapter 7: Tracing

Source material:

- Online tutorial: http://www.nsnam.org/docs/release/3.14/tutorial/singlehtml/index.html#tracing
- T. Predojev (2012): http://wikienergy.cttc.es/images/2/2d/Ns-3-tutorial-complete.pdf
- M. Lacage (2009): http://www.nsnam.org/tutorials/ns-3-tutorial-tunis-apr09.pdf
- G. Riley (2008): http://www.wns2.org/docs/wns_tutorial-handout.pdf
- S. Kristiansen (2010): http://www.uio.no/studier/emner/matnat/ifi/INF5090/v11/undervisningsmateriale/INF5090-NS-3-Tutorial-2011-Oslo-slides.pdf

More advanced tracing with NS3

- You can use ascii, pcap tracing or logging to get info from your sim
  - Need to write code to parse output
  - The info you want may not be obtainable by pre-defined mechanisms
- There is another way in NS3
  - Add your own traces to events you care about
  - Produce output in a convenient form
  - Add hooks to the core that can be accessed by other users later

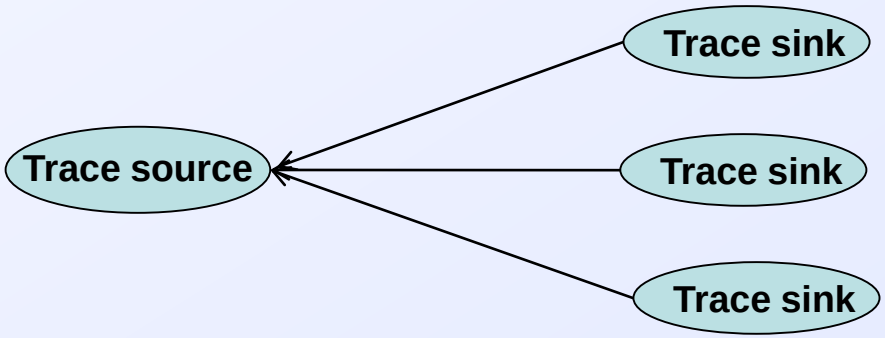
You could use print statements, but I wouldn't recommend it

- You may have to dig deep inside the NS3 core to find the info you want
- More print statements ⇒ need way to enable/disable specific ones
  - Congratulations! You've just re-invented the NS3 logging system!
- You could add logging statements to the core
  - Remember: NS3 is open-source, evolving system
    - Core will bloat to include all possible log messages
    - Unwieldy gigantic log files ⇒ effectively useless
    - No guarantee specific log messages will survive releases

Logging ≠ Tracing

The basic idea: trace sources and sinks

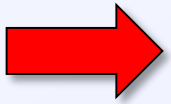
- You need:
  1. Trace source: signals sim event and provides access to the data
  2. Trace sink: consumes the trace info, do something useful with it
  3. Mechanism to connect trace source to trace sink
- Note: there can be many sinks connected to the same trace source



A simple low-level example: callbacks

- This is the key to the way tracing works
- In C/C++ you can pass a pointer to a function: callback mechanism

Ex: a function to integrate functions



```
float myFunction(float x)
{
  return x*x;
}
                ⋮
float Integrate(float (*func)(float), float lo, float hi)
{
  …
}
                ⋮
Integrate(&myFunction,0,1);
```

- Trace source maintains a list of callback functions added by the sinks
- When an event of interest happens
  - Trace source passes data to the callback functions
  - Callback functions are executed

A simple low-level example: fourth.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:
  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;
};

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;
}
```

Goal: trace changes made to a variable

- i.e., notify user whenever
  - +, ++, =, etc.

Trace source

Trace sink

Connection

A simple-low level example: the trace source

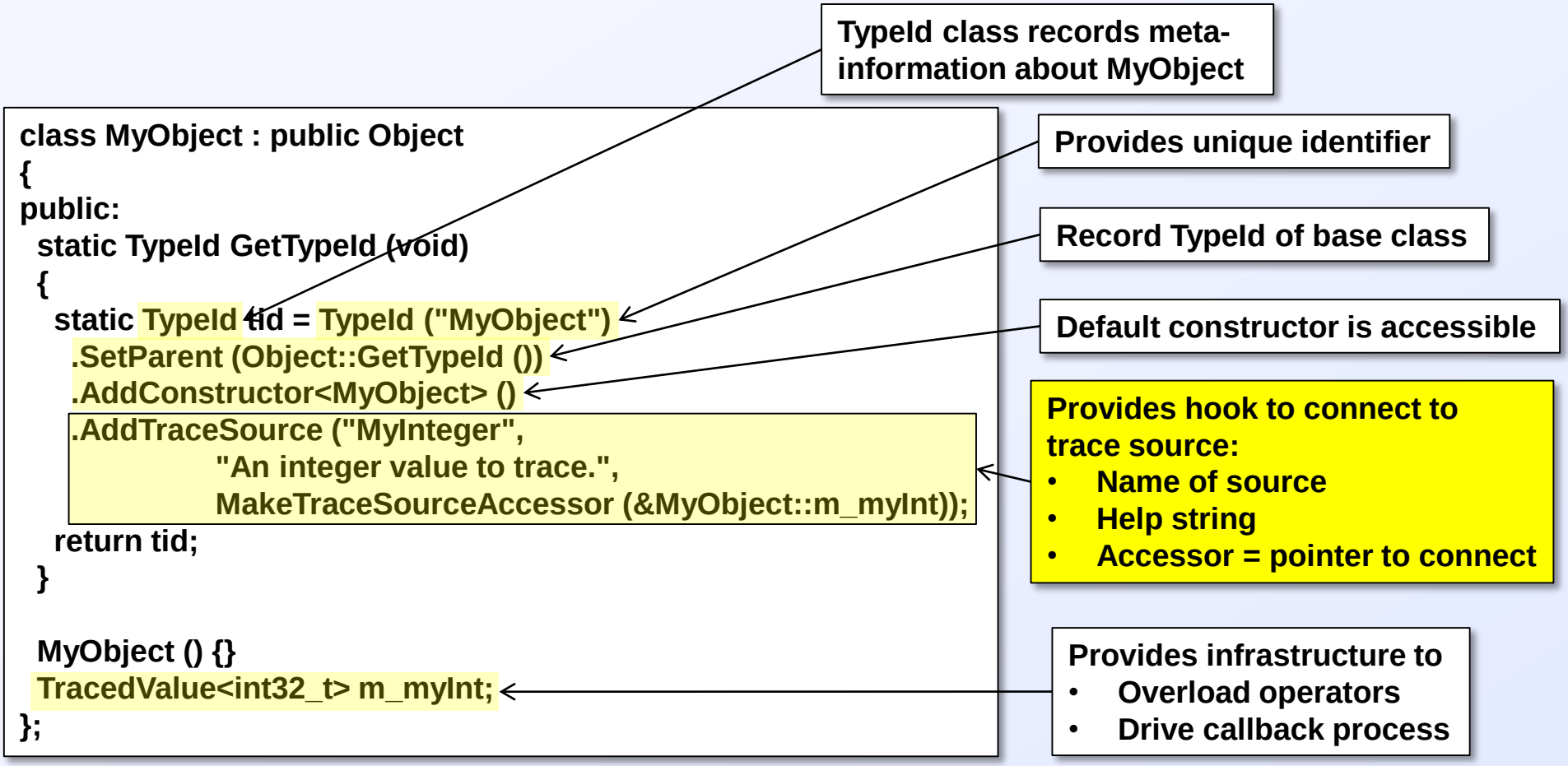
MyObject is derived from Object, inherits public members

```
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;
};
```

always need GetTypeId method for an object

- provides run-time info on type
- allows objects to located via path (more on this later)
- provides objects with attributes
- provides objects with trace sources

A simple-low level example: the trace source



A simple low-level example: the trace sink

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

A simple void function

- Takes in the old and new values of the traced variable
  - Will be supplied by the trace source
- Prints them to the screen

A low-level example: the main function

NS3 smart pointers provide garbage collection via reference counting
- Track number of pointers to an object
- Helps avoid memory leaks when you forget to delete an object

Wrapper for C++ “new”

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

  myObject->m_myInt = 1234;
}
```

If all goes as planned: should trigger callback

Connecting source to sink
- “MyInteger” = name of source
- Callback made from our sink function
- We’ll talk about “Context” next…

A low-level example: running fourth.cc

```
cp examples/tutorial/fourth.cc scratch/myfourth.cc
./waf --run scratch/myfourth
```

Traced 0 to 1234

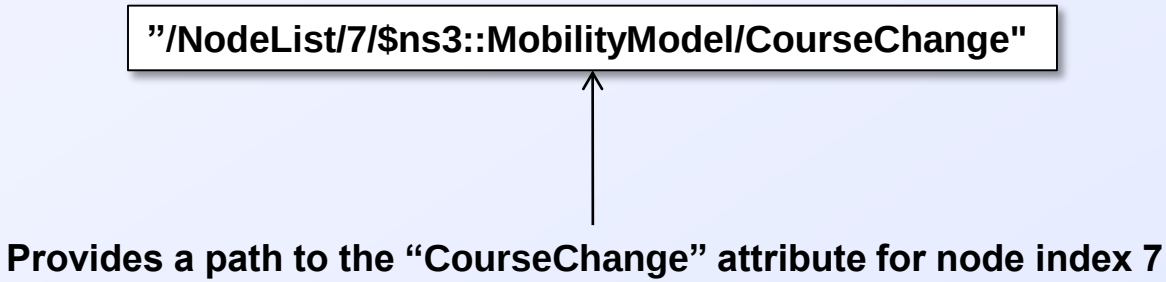
Assignment in: myObject->m_myInt = 1234;
triggers callback

Seems almost anticlimactic, but we've illustrated the main steps in tracing

- **Define trace source**
- **Define trace sink**
- **Connect trace source to trace sink**

Using Config to connect to trace sources

- TraceConnectWithoutContext is not typical way to connect source to sink
- Normally done via the Config subsystem in ns3, by using a config path
  - A string that looks like a file path
  - Represents chain of objects leading to the desired event (or attribute)
- We encountered this in third.cc



Let's dissect this config path

Dissecting the Config path in third.cc

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

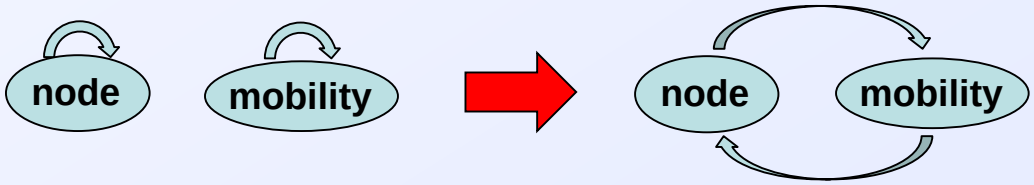
- "NodeList" = predefined namespace in Config, lists all the nodes in the sim
- "NodeList/7" = node 7 (i.e., the eigth node)
- "$" = what follows designates an aggregated object

Let's take a short detour to discuss aggregated objects in NS3

Object aggregation in NS3

- Makes it possible for objects to access each other, and for users to easily access objects in an aggregation
- Avoids need to modify a base class to provide pointers to all possible connected objects
  - Class definitions would bloat uncontrollably
  - Solution: separate functionality belongs to separate classes

```
Node->AggregateObject(mobility);
```



Retrieving an aggregated object:

```
Ptr<MobilityModel> mob = node->GetObject<MobilityModel> ();
```

Dissecting the Config path in third.cc

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

- “NodeList” = predefined namespace in Config, lists all the nodes in the sim
- "NodeList/7" = node 7 (i.e., the eigth node)
- "$" = what follows designates an aggregated object
- "/NodeList/7/$ns3::MobilityModel" = get the mobility model object aggregated to node 7 in the sim
- "CourseChange" = attribute of MobilityModel we are interested in
- Config will follow the chain of pointers to the attribute you specified

```
Config::Connect(”/NodeList/7/$ns3::MobilityModel/CourseChange”,MakeCallback(&CourseChange))
```

- Ties the trace source to the trace sink callback function
- Passes the context (= config path) to the callback function

How to Find and Connect Trace Sources, and Discover Callback Signatures

- What trace sources are available (besides CourseChange)?
- How do I figure out the config path I need to connect to this source?
- What are the return type and arguments of my callback function?

```
void CourseChange(std::string context,Ptr<const MobilityModel> model)
```

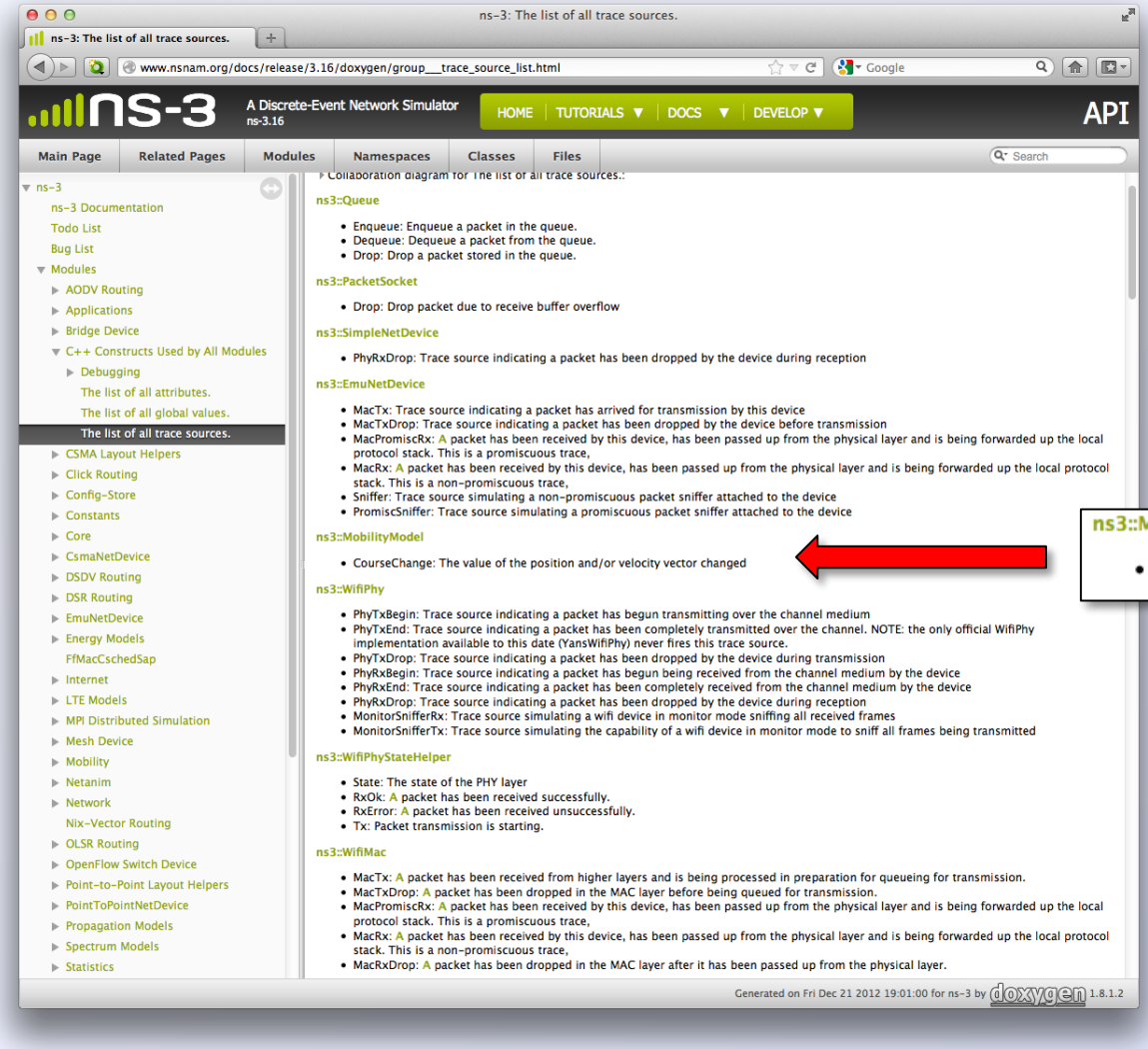
```
Void IntTrace (int32_t oldValue, int32_t newValue)
```

Doxygen and a little detective work will go a long way here

What trace sources are available?

- Doxygen has the answer!
  - Trace sources: http://www.nsnam.org/docs/release/3.16/doxygen/group___trace_source_list.html
  - Attributes: http://www.nsnam.org/docs/release/3.16/doxygen/group___attribute_list.html
  - Global values: http://www.nsnam.org/docs/release/3.16/doxygen/group___global_value_list.html

What trace sources are available?



ns3::MobilityModel

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

What string do I use to connect?

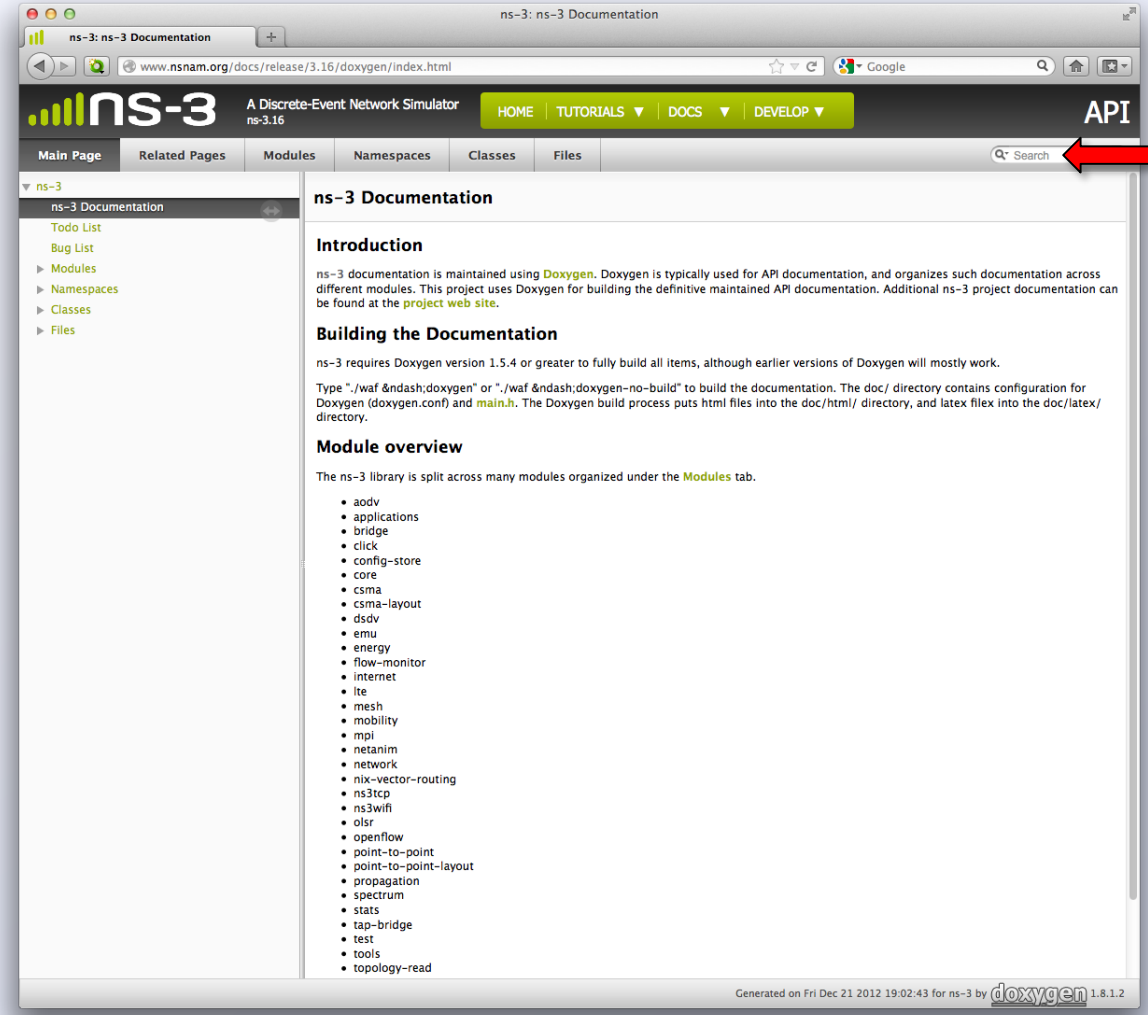
- method 1: look for config path in someone else's code
  - find -name "*.cc" | xargs grep CourseChange | grep Connect

```
                ⋮
./src/mobility/examples/main-random-walk.cc:  Config::Connect
("/NodeList/*/$ns3::MobilityModel/CourseChange",
                ⋮
```

  - Config path = "/NodeList/*/$ns3::MobilityModel/CourseChange"
    - you can use regular expressions in the path, so "*" ⇒ any node
    - "/NodeList/[3-5]|8|[0-1]" would match node indices 0,1,3,4,5,8

What string do I use to connect?

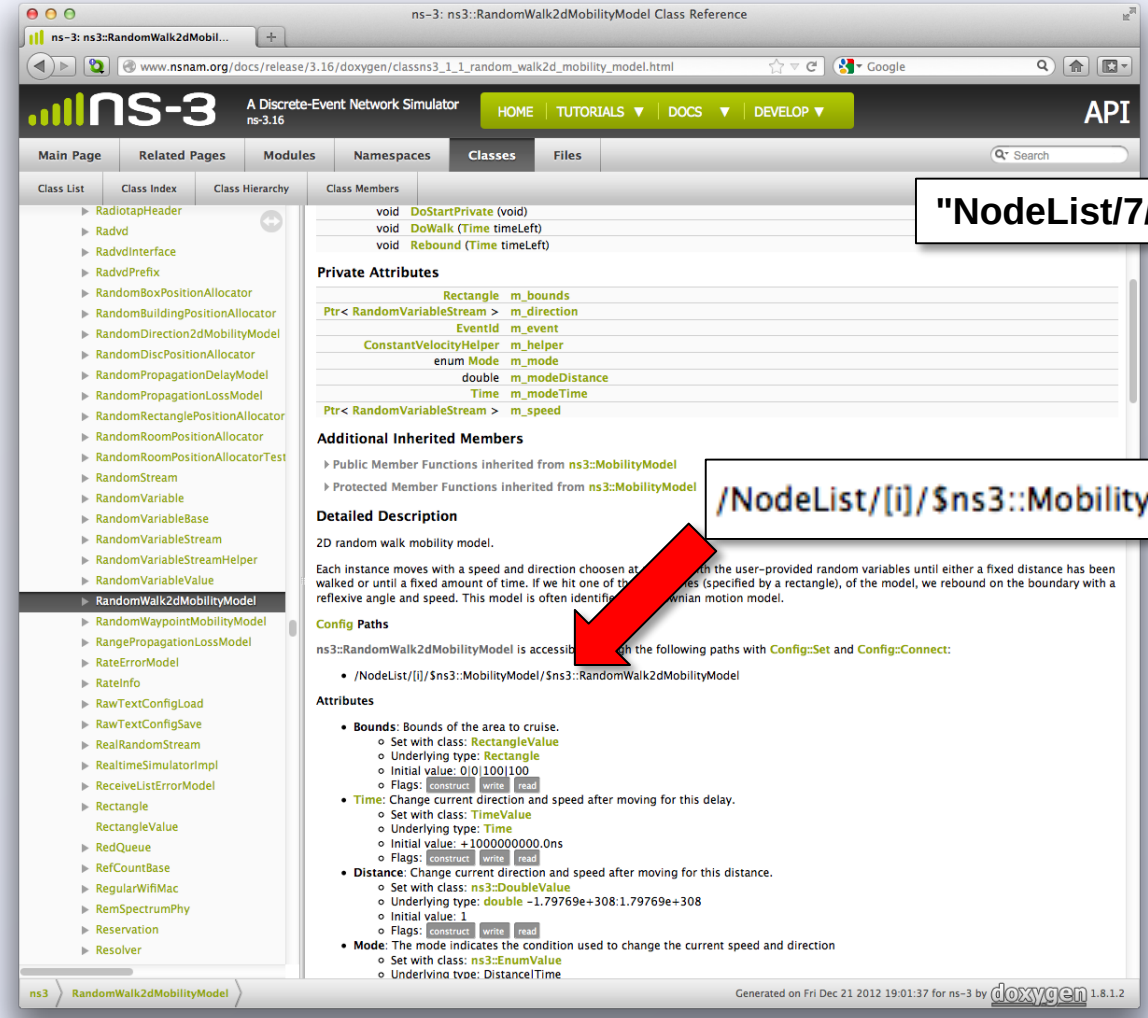
- **method 2: Doxygen**



RandomWalk2dMobilityModel

What string do I use to connect?

- **method 2**: Doxygen



Remember: before, we used:

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

We're almost there! We found:

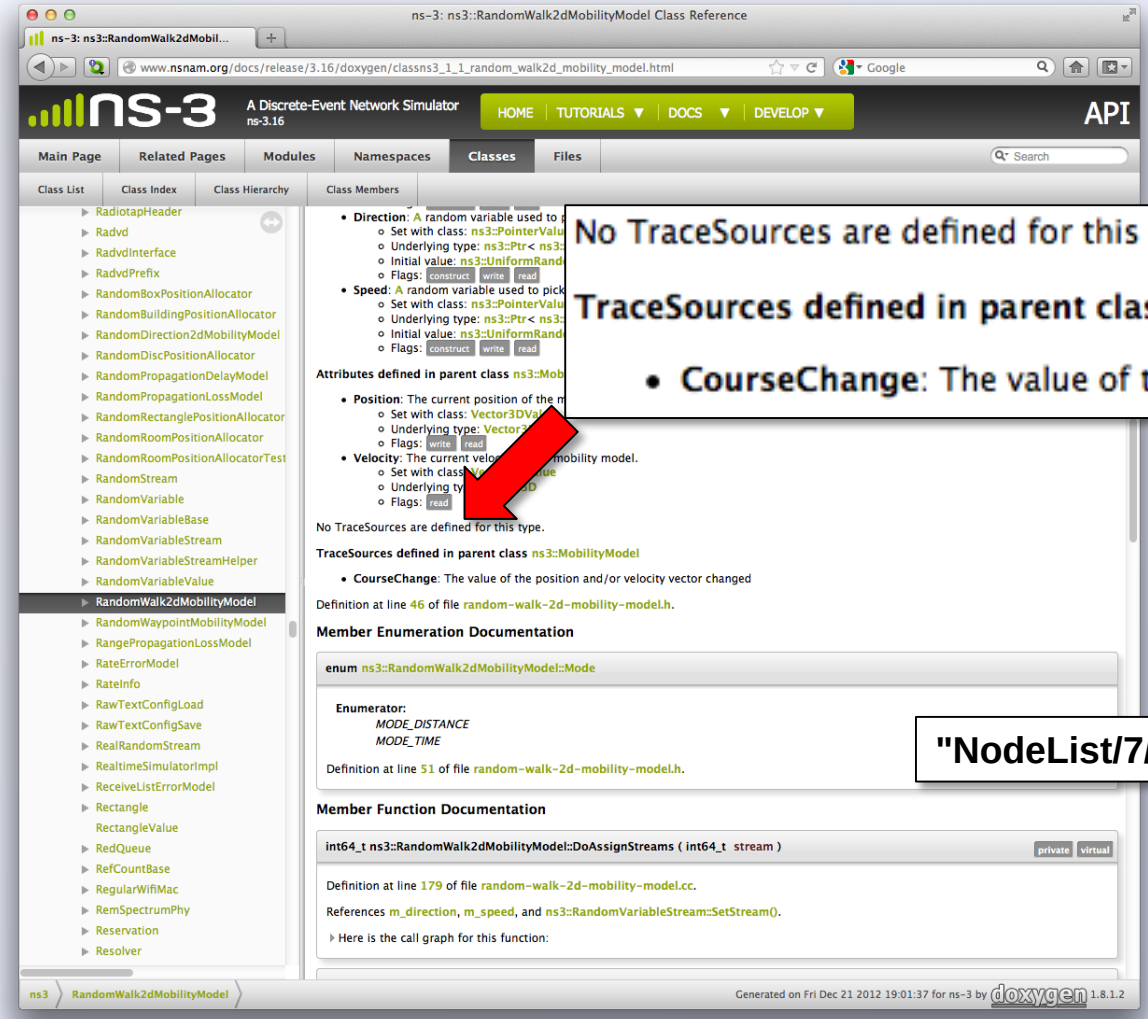
/NodeList/[i]/$ns3::MobilityModel/$ns3::RandomWalk2dMobilityModel

But what's this bit?

Scroll further down

What string do I use to connect?

- **method 2**: Doxygen



No TraceSources are defined for this type.

TraceSources defined in parent class ns3::MobilityModel

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

This tells you to access the "CourseChange" attribute from the parent class, so:

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

What are the return value and arguments for the callback function?

- The easy way: copy someone else's answer
  - find -name "*.cc" | xargs grep CourseChange | grep Connect

```
./src/mobility/examples/main-random-walk.cc:  Config::Connect
("/NodeList/*/$ns3::MobilityModel/CourseChange",
```

  - Inside "./src/mobility/examples/main-random-walk.cc":

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

  - And we then find the header of the "CourseChange" function:

```
static void CourseChange (std::string foo, Ptr<const MobilityModel> mobility)
```

But what if you can't find what you need in someone else's code?

What return value and arguments for the callback function?

- The somewhat easy way
  - The return value is always “void”
  - To get the list of formal arguments, look in the ".h" file for the model
    - Find the "TracedCallback" declaration
    - For ex,in "src/mobility/model/mobility-model.h”

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

This is the argument we need!

    - So for callback using Config::ConnectWithoutContext

```
void CourseChange(Ptr<const MobilityModel> model)
```

    - And for callback using Config::Connect

```
void CourseChange(std::string path,Ptr<const MobilityModel> model)
```

Context (= config path) gets passed here

What about TracedValue?

- Remember we used the callback function

```
void IntTrace(int32_t oldValue, int32_t newValue)
```

- How did we guess the right formal arguments to use?
- Find "TracedCallback" declaration in "src/core/model/traced-value.h"

```
template <typename T> class TracedValue
{
public:
  ...
private:
  T m_v;
  TracedCallback<T,T> m_cb;
};
```

Template class ⇒ you get to choose its type

- In fourth.cc, inside the MyObject class, we declared

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

⇒ T = int32_t

⇒ Two arguments in the callback, both of type T (i.e., int32_t)

A real example

From W. Richard Stevens, "TCP/IP Illustrated, Vol. I: The Protocols", Addison-Wesley (1994)

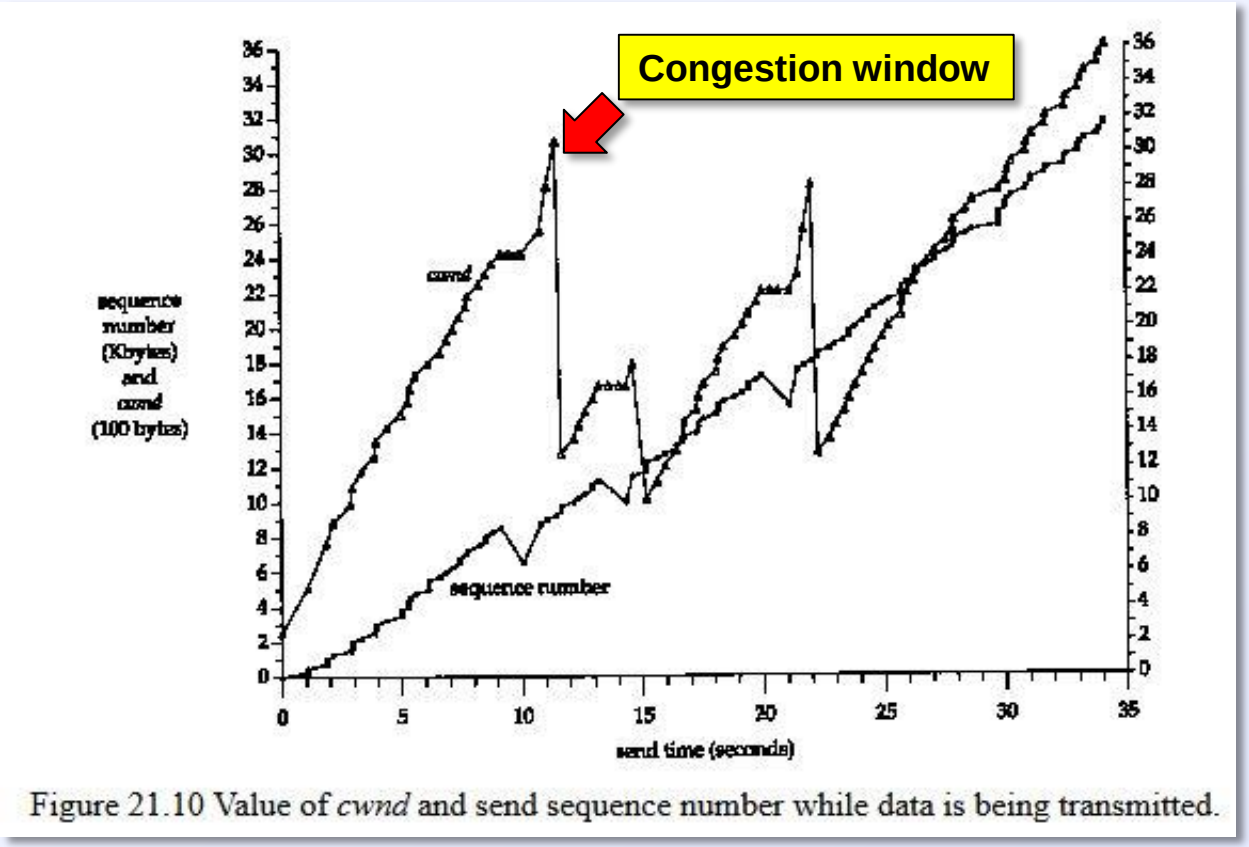


Figure 21.10 Value of *cwnd* and send sequence number while data is being transmitted.

Goal: simulate a TCP congestion window and look at the effect of dropped packets

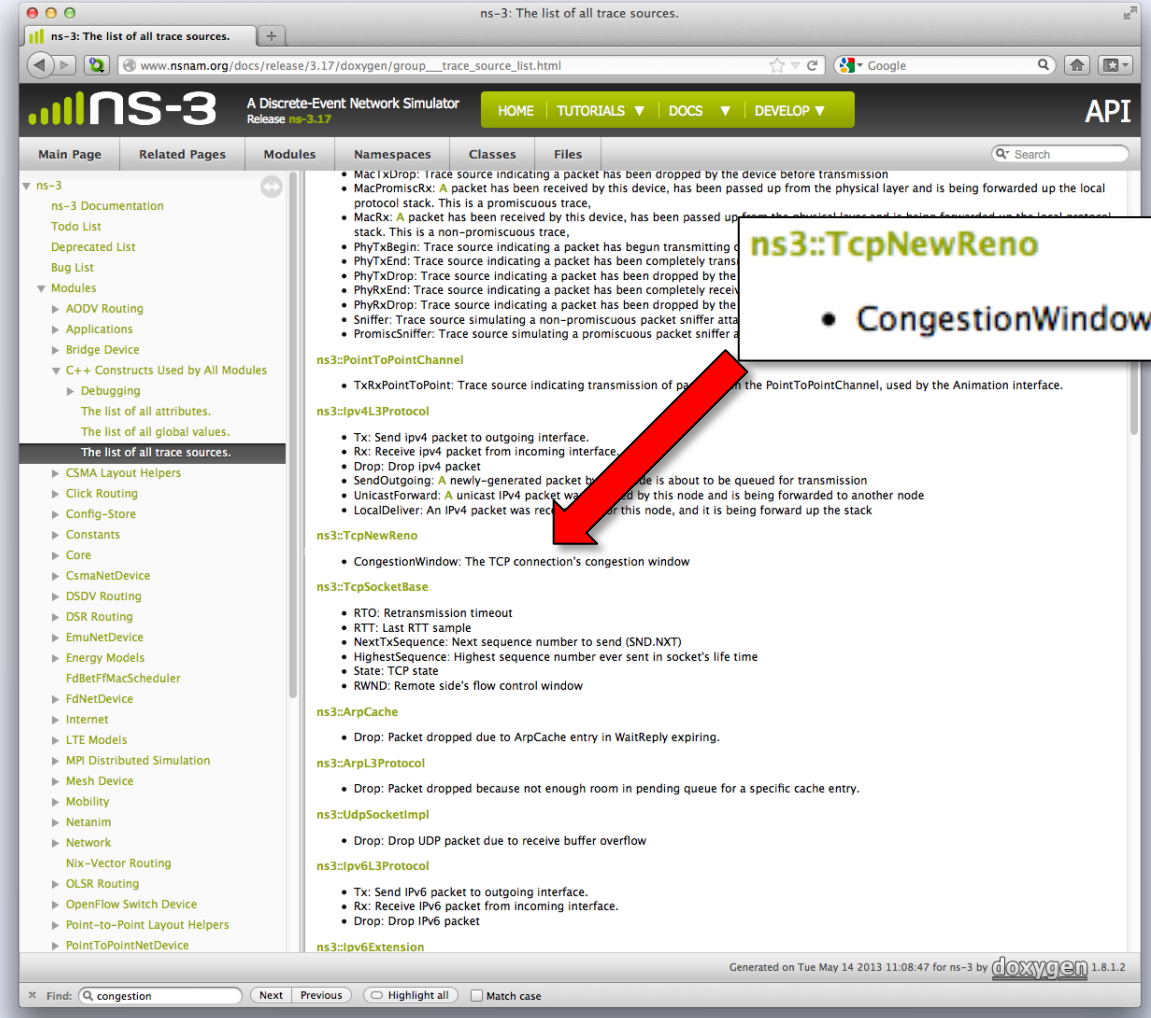
Congestion windows 101 (from Wikipedia)

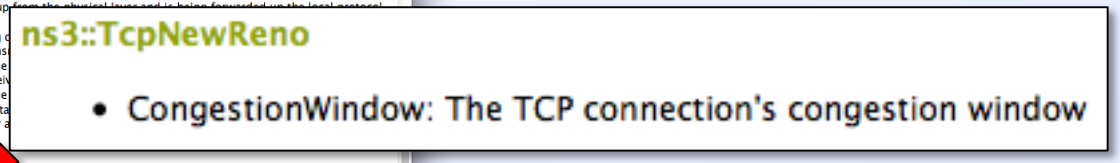
- Congestion collapse occurs at choke points in the network
  - Wherever total incoming traffic to a node > outgoing bandwidth
  - e.g., connection points between LAN and WAN
- TCP uses network congestion avoidance algorithm
- Congestion window is part of TCP strategy to avoid congestion collapse
  - Limits total number of unacknowledged packets that may be in transit
  - The size of the window is adjusted dynamically by the sender
    - Based on how much congestion between sender and receiver
    - If all segments are received and the acks reach the sender on time
      - Add a constant to the window (typically 1 MSS = Max Segment Size)
    - Otherwise
      - Scale back by a set factor (typically 1/2)

Explains "sawtooth" shape of congestion window over time

Are there trace sources available?

- From Doxygen's "The list of all trace sources"





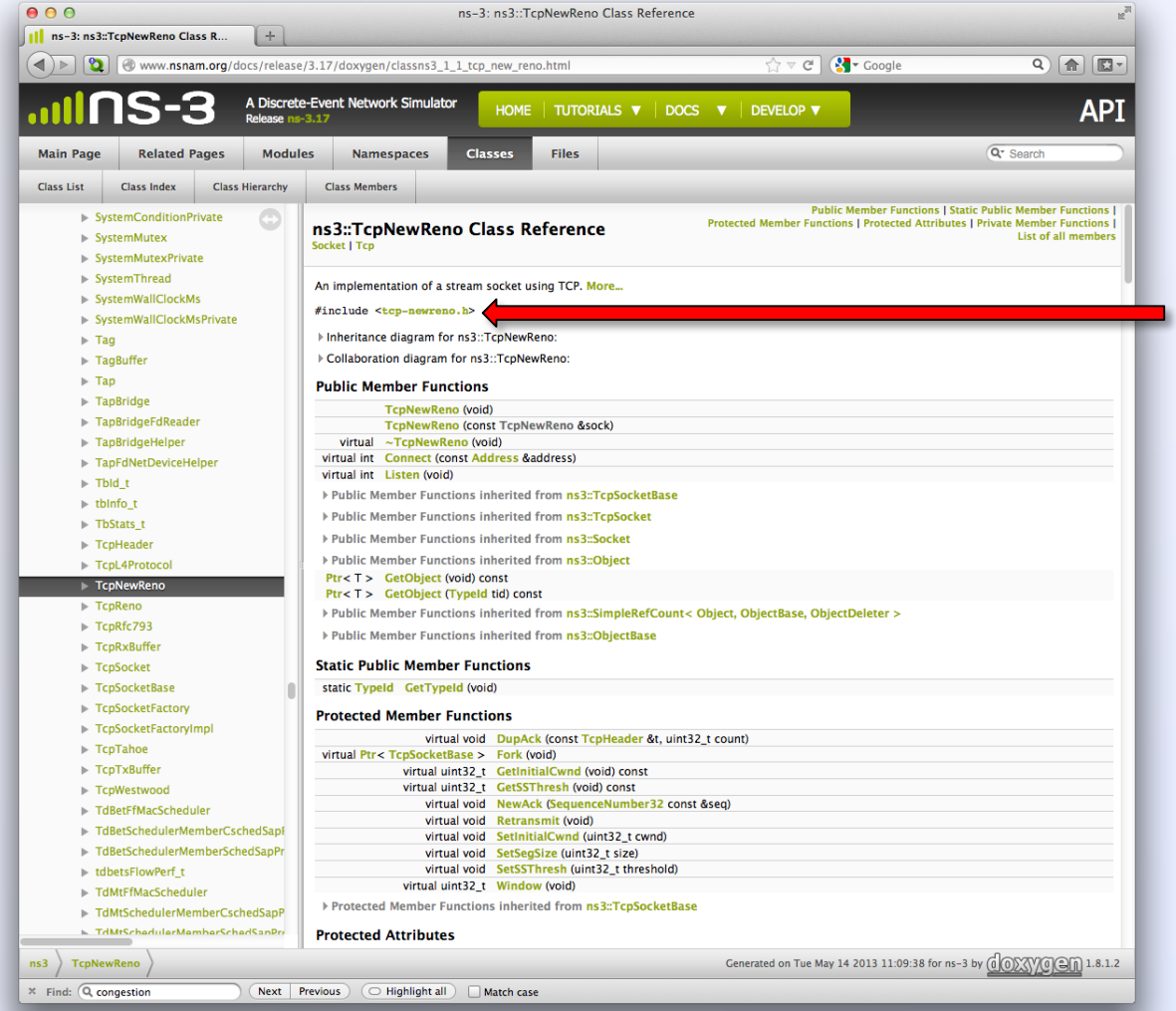
Look for "congestion":

We have our trace source: CongestionWindow

Click on "ns3::TcpNewReno" to access class info

Are there trace sources available?

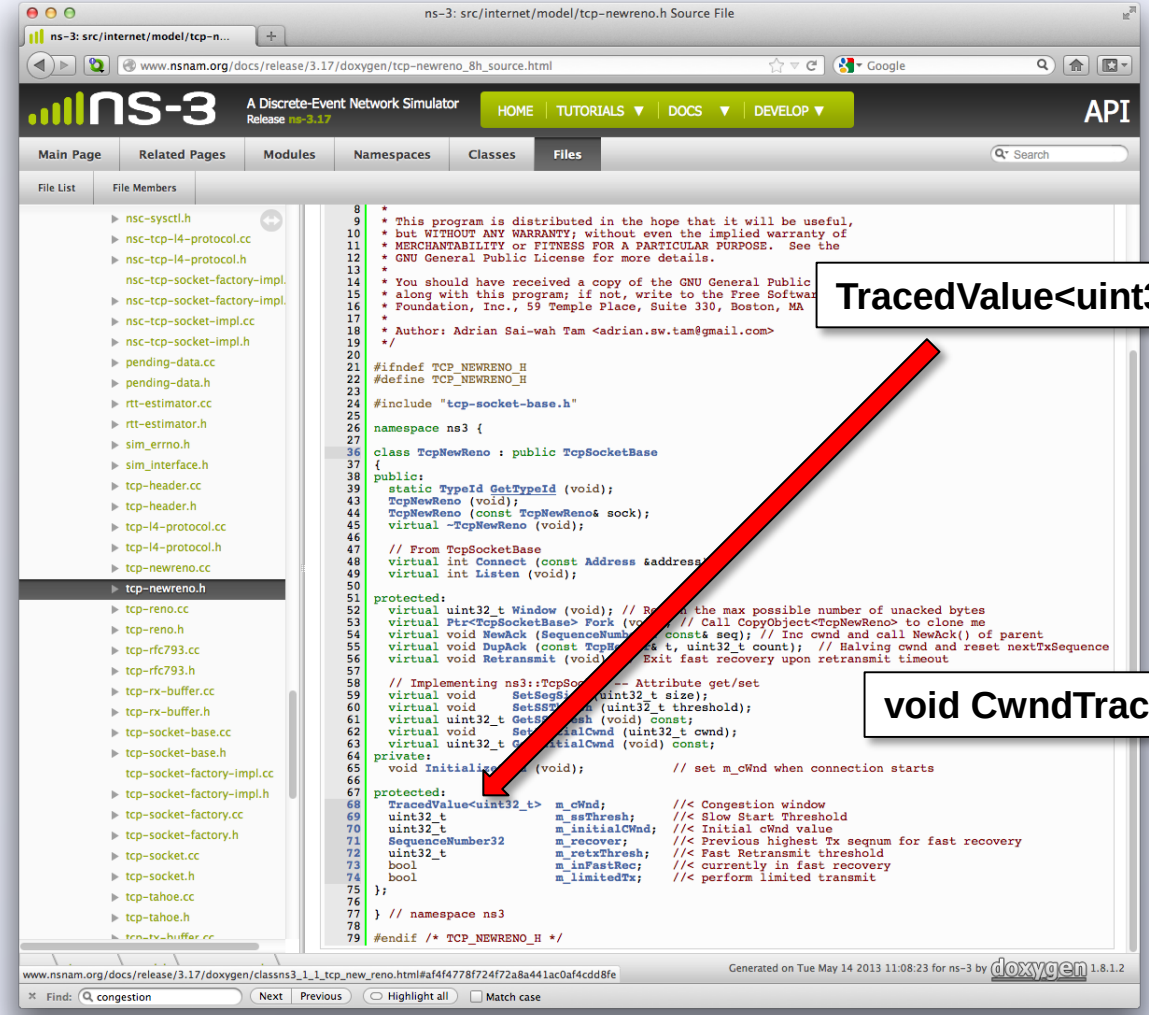
- From Doxygen's "The list of all trace sources"



- Click on link to header file "tcp-newreno.h"
- Look for keyword "congestion"

Are there trace sources available?

- From Doxygen's "The list of all trace sources"



TracedValue<uint32_t> m_cWnd; //< Congestion window

this looks a lot like our fourth.cc example

we'll need a callback function of the form

void CwndTrace(uint32_t oldValue,uint32_t newValue)

How do we get started writing our script?

- The time-honored way: steal from someone else's code
  - find . -name "*.cc" | xargs grep CongestionWindow
  - Look, e.g., in "./src/test/ns3tcp/ns3tcp-cwnd-test-suite.cc"
  - The connection between trace and source is done by

```
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow",
MakeCallback (&Ns3TcpCwndTestCase1::CwndChange, this));
```

This is the callback function

```
void CwndChange (uint32_t oldCwnd, uint32_t newCwnd)
```

  - We can also copy code from the function

```
void Ns3TcpCwndTestCase1::DoRun (void)
```

This is how fifth.cc was put together

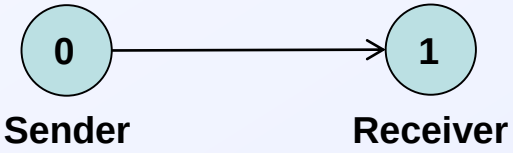
Avoiding a common mistake in NS3

- NS3 scripts execute in three separate stages
  - Configuration time
  - Simulation time (i.e., Simulator::Run)
  - Teardown time
- TCP uses sockets to connect nodes
  - Sockets are created dynamically during simulation time
    - Want to hook the CongestionWindow on the socket of the sender
  - Connection to trace sources is established during configuration time
  - Can't put the cart before the horse!
- Solution:
  1. Create socket at configuration time
  2. Hook trace source then
  3. Pass this socket object to system during simulation time

Next: fifth.cc walkthrough

Overview of fifth.cc

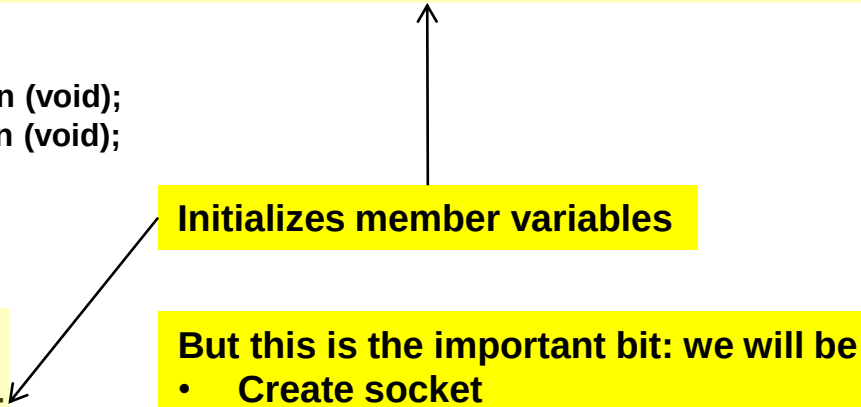
- Dumbbell topology, point-to-point network, like first.cc



- We will create our own application and socket
  - So we can access socket at configuration time
  - No helper, so we'll have to do the work manually
  - Connect to CongestionWindow trace source in sender socket
- Introduce errors into the channel between nodes
  - Dropped packets ⇒ interesting behavior in congestion window

Creating our own application: the MyApp class

```
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;
};
```



Initializes member variables

But this is the important bit: we will be able to

- **Create socket**
- **Hook its CongestionWindow trace source**
- **Pass the socket to the application**

All this at configuration time!

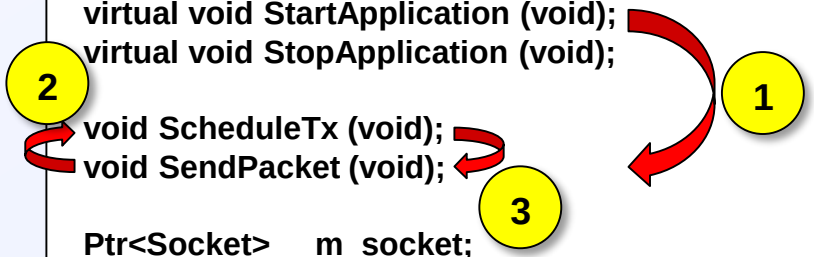
Creating our own application: the MyApp class

```
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;
};
```

We also need to override these with our own implementations that will be called by the simulator to start and stop

Creating our own application: the MyApp class

```
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;
};
```



1. StartApplication calls SendPacket
2. SendPacket calls ScheduleTx
3. ScheduleTx set up next SendPacket call

…

1. Until StopApplication

The trace sinks

- We want to trace 2 types of events
  - Updates to the congestion window:

```
static void CwndChange (uint32_t oldCwnd, uint32_t newCwnd)
{
  NS_LOG_UNCOND (Simulator::Now ().GetSeconds () << "\t" << newCwnd);
}
```

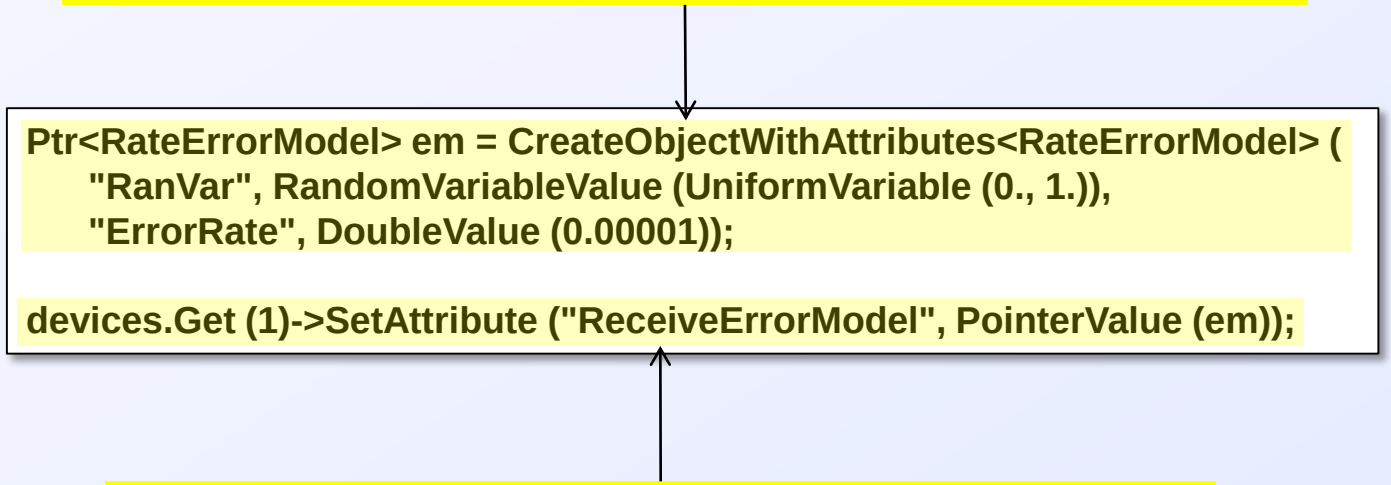
  - Dropped packets

```
static void RxDrop (Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
}
```

Introducing errors in the channel

Determines which packets are errored corresponding to underlying

- **Distribution = random variable**
- **Rate ↔ mean duration between errors**
- **Unit = per-bit, per-byte, or per-packet**



```
Ptr<RateErrorModel> em = CreateObjectWithAttributes<RateErrorModel> (
    "RanVar", RandomVariableValue (UniformVariable (0., 1.)),
    "ErrorRate", DoubleValue (0.00001));

devices.Get (1)->SetAttribute ("ReceiveErrorModel", PointerValue (em));
```

Will cause randomly dropped packets in the receiver device

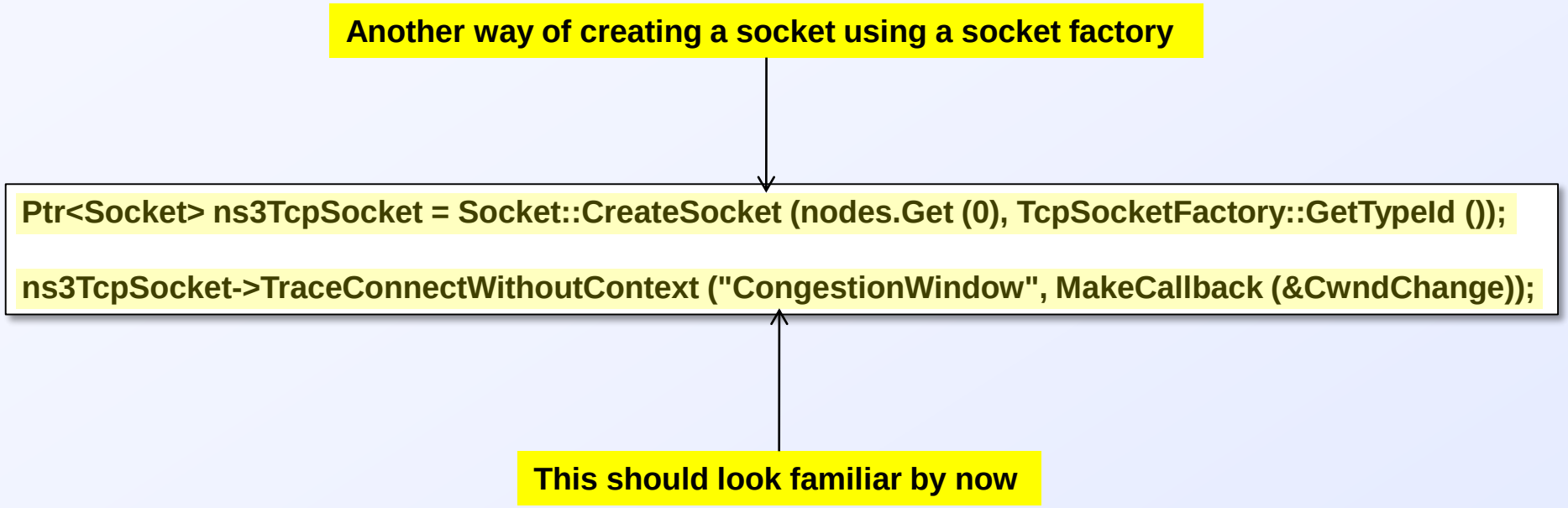
Setting the application on the receiver node

```
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.));
```


- PacketSink receives and consumes traffic generated to an IP address and port
- PacketSinkHelper creates sockets using an “object factory”
  - Object factories are used to mass produce similarly configured objects
  - Factory method doesn’t require you to know the type of the objects created

Connecting the sender's socket, and connecting the trace source

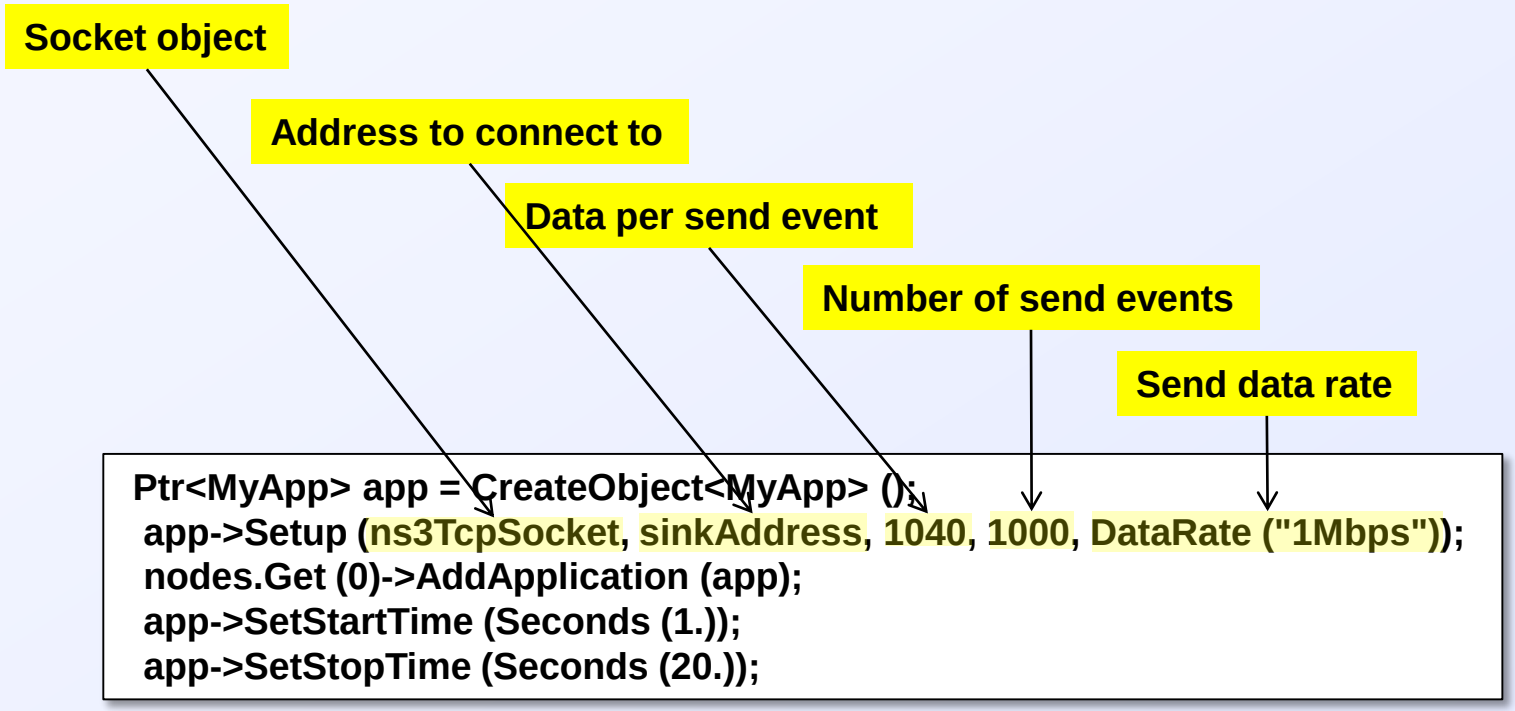
Another way of creating a socket using a socket factory



This should look familiar by now

Because we created our own app and socket at configuration time we can hook into its CongestionWindow trace source

Setting up the application on the sender node



Running fifth.cc: text output

```
./waf –run scratch/myfifth > cwnd.dat 2>&1
```

```
Waf: Entering directory `/mypath/ns-3-dev/build'
Waf: Leaving directory `/mypath/ns-3-dev/build'
'build' finished successfully (1m58.520s)
1       536
1.00919 1072
1.01511 1608
1.02163 2144
1.02995 2680
1.03827 3216
1.04659 3752
1.05491 4288
1.06323 4824
1.07155 5360
1.07987 5896
1.08819 6432
1.09651 6968
1.10483 7504
1.11315 8040
1.12147 8576
1.12979 9112
RxDrop at 1.13692
1.13811 9648
1.15475 2900
1.15563 3436
```

Eliminate these lines by hand

Running fifth.cc: plotting the results

Original by W. Richard Stevens

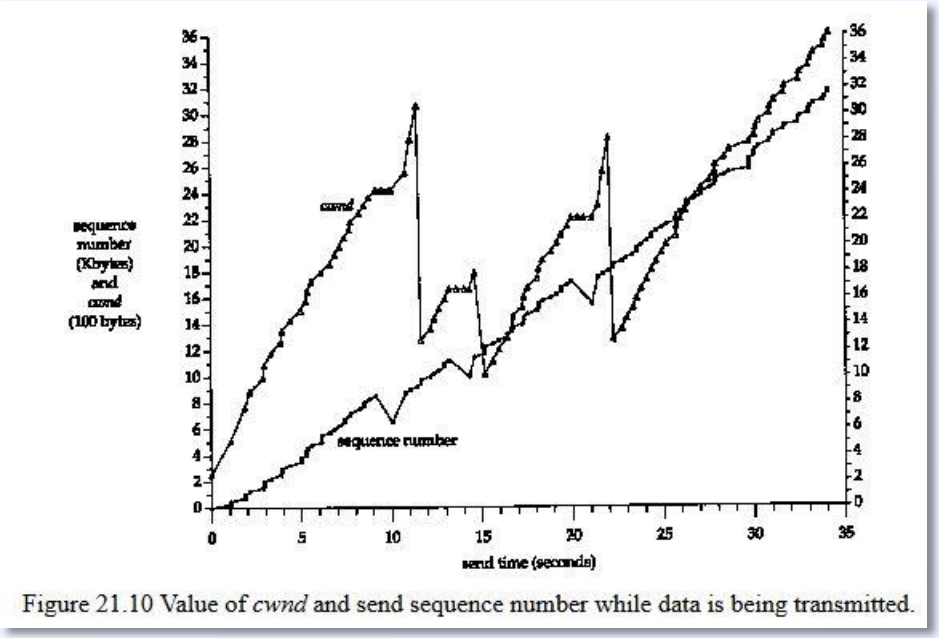
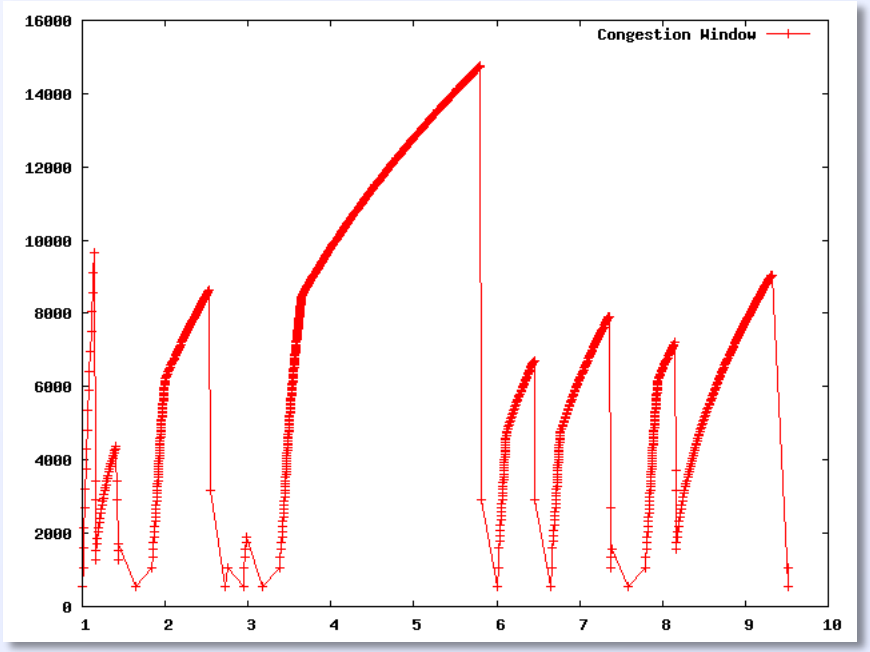


Figure 21.10 Value of *cwnd* and send sequence number while data is being transmitted.

Our result, using gnuplot



That's nice, but…

- Remember after

```
./waf –run scratch/myfifth > cwnd.dat 2>&1
```

- We had to edit the file by hand to remove "junk" lines…

- But we said tracing gives you control over output format
- Is there a cleaner way to produce the output we need?
- Yes! Use trace helpers

Let's tweak fifth.cc to produce cleaner output ⇒ sixth.cc

A sixth.cc walkthrough: CwndChange callback

Modify the callback function:

```
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;
}
```

Added to fifth.cc

Formatted output to the stream

Add lines in the main to create stream:

Filename attached to stream

```
AsciiTraceHelper asciiTraceHelper;
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("sixth.cwnd");
ns3TcpSocket->TraceConnectWithoutContext ("CongestionWindow", MakeBoundCallback (&CwndChange, stream));
```

Causes the stream argument to be added to the function callback

A sixth.cc walkthrough: RxDrop callback

Modify the callback function:

```
static void RxDrop (Ptr<PcapFileWrapper> file, Ptr<const Packet> p)
{
  NS_LOG_UNCOND ("RxDrop at " << Simulator::Now ().GetSeconds ());
  file->Write (Simulator::Now (), p);
}
```

Added to fifth.cc

Formatted output to the pcap file

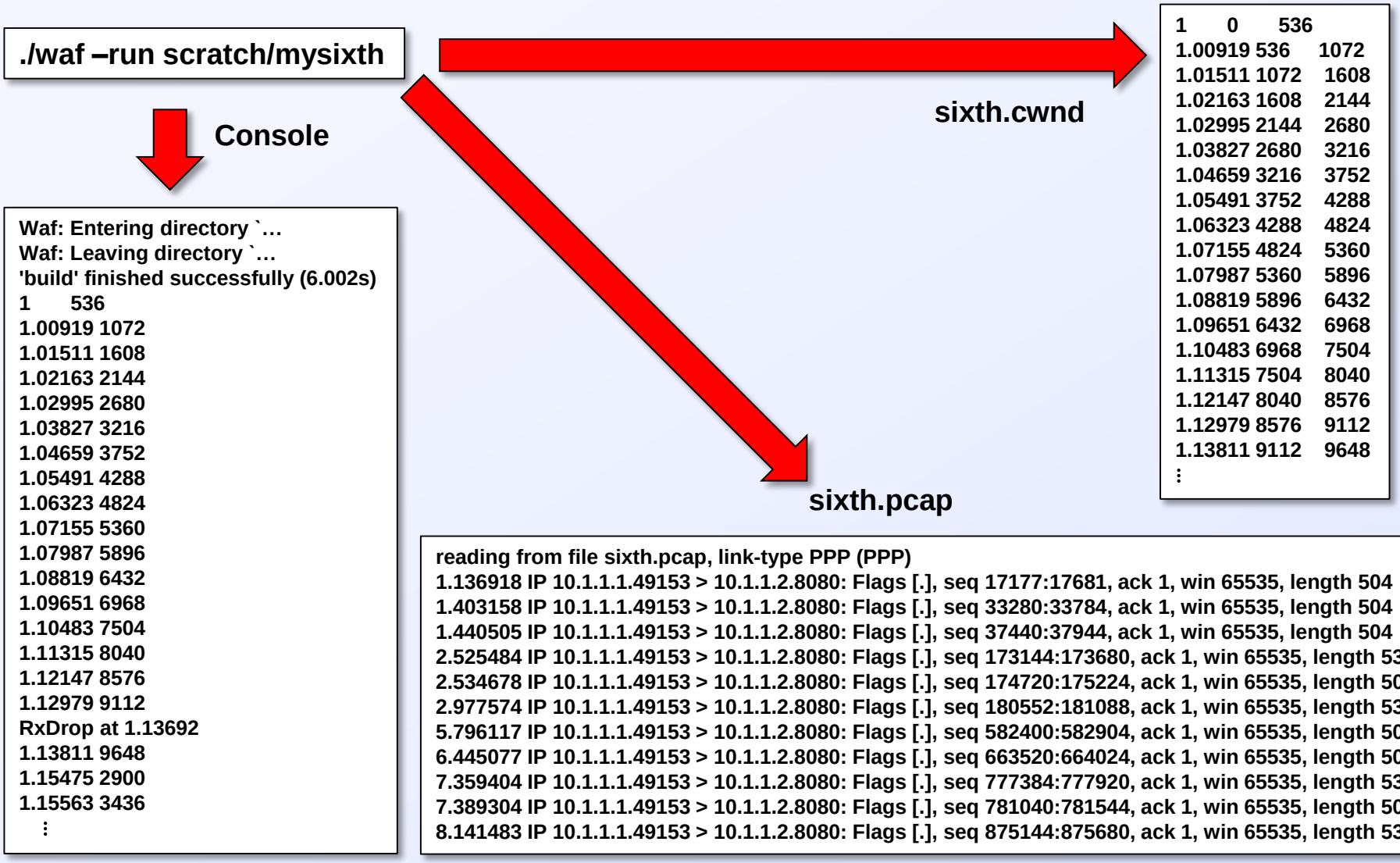
Add lines in the main to create stream:

Filename attached to file

```
PcapHelper pcapHelper;
Ptr<PcapFileWrapper> file = pcapHelper.CreateFile ("sixth.pcap", std::ios::out, PcapHelper::DLT_PPP);
devices.Get (1)->TraceConnectWithoutContext ("PhyRxDrop", MakeBoundCallback (&RxDrop, file));
```

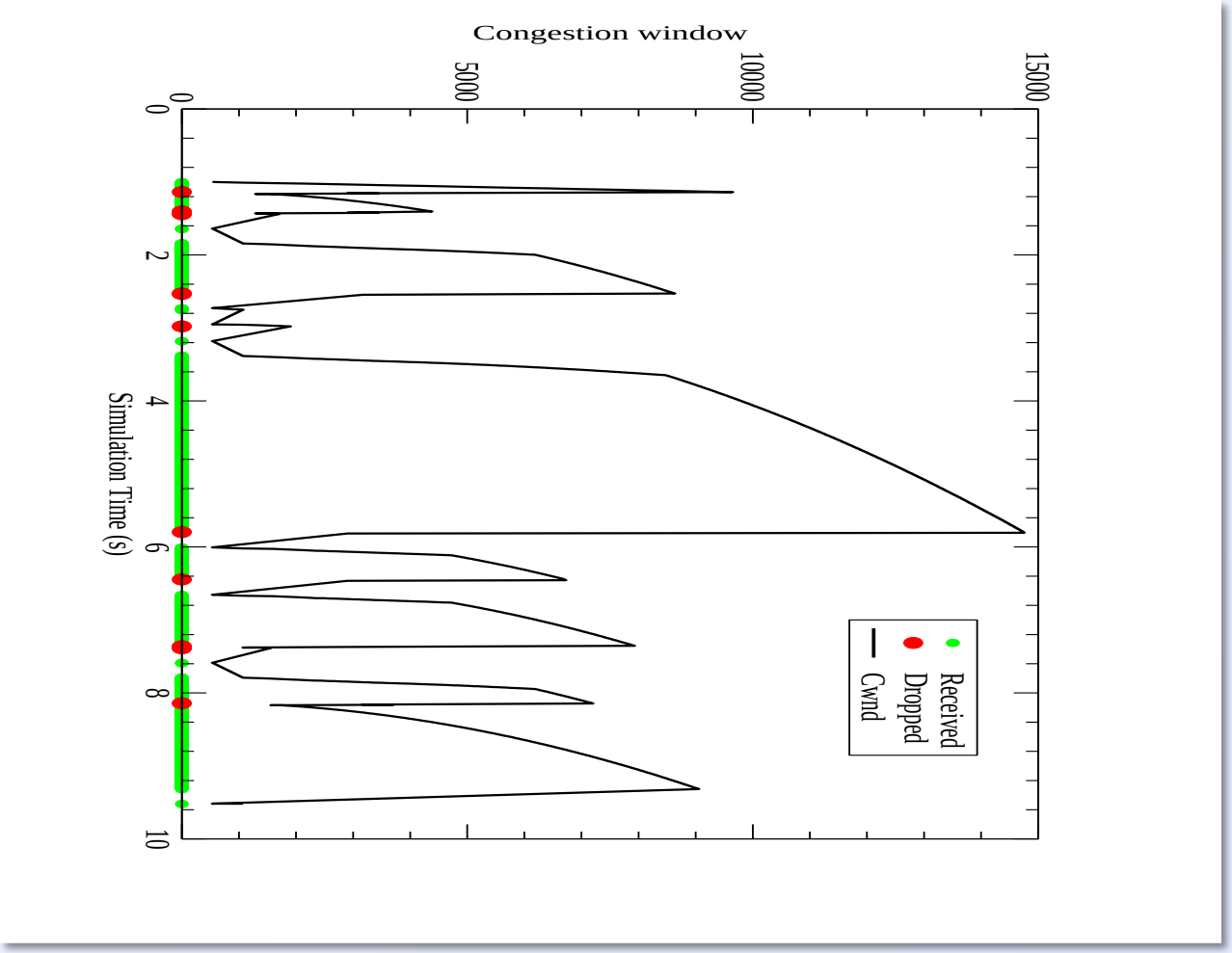
Causes the file argument to be added to the function callback

Running sixth.cc



Just for fun: what happens to uncorrupted packets?

PhyRxEnd: Trace source indicating a packet has been completely received by the device



Using trace helpers

- We've encountered trace helpers before

```
pointToPoint.EnablePcapAll ("second");
pointToPoint.EnablePcap ("second", p2pNodes.Get (0)->GetId (), 0);
csma.EnablePcap ("third", csmaDevices.Get (0), true);
pointToPoint.EnableAsciiAll (ascii.CreateFileStream ("myfirst.tr"));
```

- What other trace helpers are available?
- How do we use them?
- What do they have in common?

Two categories of trace helpers

- Device helpers
  - Trace is enabled for node/device pair(s)
  - Both pcap and ascii traces provided
  - Conventional filenames: <prefix>-<node id>-<device id>
- Protocol helpers
  - Trace is enabled for protocol/interface pair(s)
  - Both pcap and ascii traces provided
  - Conventional filenames: <prefix>-n<node id>-i<interface id>
    - "n" and "i" to avoid filename collisions with node/device traces

NS3 uses “mixin” classes to ensure tracing works the same way across all devices or interfaces

- **Mixin classes in NS3 (see Doxygen for more info):**

	PCAP	ASCII
Device Helper	PcapHelperForDevice	AsciiTraceHelperForDevice
Protocol Helper	PcapHelperForIpv4	AsciiTraceHelperForIpv4

- **These mixin classes each provide a single virtual method to enable trace**
  - **All device or protocols must implement this method**
  - **All other methods of the mixin class call this one method**
  - **Provides consistent functionality across different devices & interfaces**

The PcapHelperForDevice mixin class

- Every device must implement

```
virtual void EnablePcapInternal (std::string prefix,
                                 Ptr<NetDevice> nd,
                                 bool promiscuous,
                                 bool explicitFilename) = 0;
```

Pcap filename prefix

Device to trace

Type of trace

Override prefix

- All other methods of PcapHelperForDevice call this one

⇒ Consistency across devices

Pcap Tracing Device Helper: EnablePcap methods

- EnablePcap for various node/device pair(s)
  - Provide Ptr<NetDevice>
  - Provide device name using the NS3 object name service

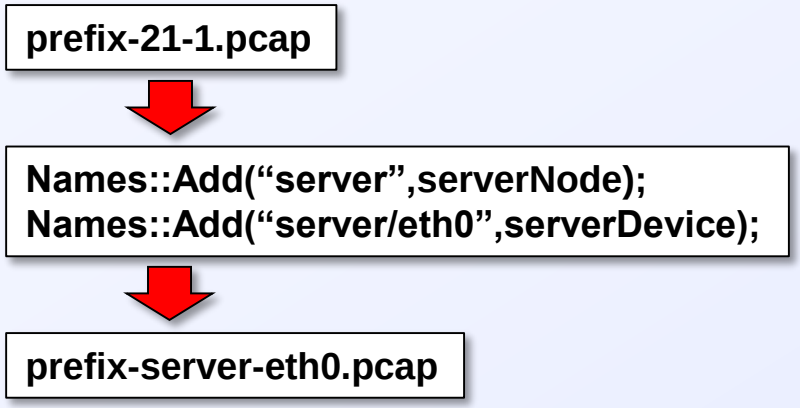
```
Names::Add ("server" ...);
Names::Add ("server/eth0" ...);
...
helper.EnablePcap ("prefix", "server/ath0");
```

  - Provide a NetDeviceContainer
  - Provide a NodeContainer
  - Provide integer node and device ids
- Enable pcap tracing for all devices in the simulation

```
helper.EnablePcapAll ("prefix");
```

Pcap Tracing Device Helper: filename selection

- By convention: <prefix>-<node id>-<device id>.pcap
- Can use the NS3 object name service to replace ids with meaningful names e.g.,



- You can override the naming convention, e.g.

```
void EnablePcap(std::string prefix, Ptr<NetDevice> nd, bool promiscuous, bool explicitFilename);
```

Set this to true
- prefix becomes filename

Ascii Tracing Device Helpers

- The mixin class is AsciiTraceHelperForDevice
  - All device implement virtual EnableAsciiInternal method
  - All other methods of AsciiTraceHelperForDevice will call this one
- Can provide EnableAscii with Ptr<NetDevice>, string from name service, NetDeviceContainer, NodeContainer, integer node/device ids
- Or helper.EnableAsciiAll("prefix");
- Can also dump ascii traces to a single common file, e.g.,

```
Ptr<NetDevice> nd1;
Ptr<NetDevice> nd2;
...
Ptr<OutputStreamWrapper> stream = asciiTraceHelper.CreateFileStream ("trace-file-name.tr");
...
helper.EnableAscii (stream, nd1);
helper.EnableAscii (stream, nd2);
```

- So there are twice as many trace methods as for pcap

Ascii Tracing Device Helpers: filename selection

- By convention: <prefix>-<node id>-<device id>.tr
- Using NS3 object name service, can assign names to the id, then e.g.
  - "prefix-21-1.tr" → "prefix-server-eth0.tr"
- Many EnableAscii methods offer a explicitFilename option
  - set to true ⇒ override naming convention, use your own file name

Pcap Tracing Protocol Helpers

- The mixin class is PcapHelperForIpv4
  - All device implement virtual EnablePcapIpv4Internal method
  - All other methods of PcapHelperForIpv4 will call this one
- Can provide EnablePcapIpv4 with Ptr<Ipv4>, string from name service, Ipv4InterfaceContainer, NodeContainer, integer node/device ids
- Or helper.EnablePcapIpv4All("prefix");
- Filename selection
  - Convention is <prefix>-n<node id>-i<interface id>.pcap
  - Can also use the NS3 object name service clarity
  - explicitFilename parameter lets you impose your own filenames

Ascii Tracing Protocol Helpers

- The mixin class is AsciiTraceHelperForIpv4
  - All device implement virtual EnableAsciiIpv4Internal method
  - All other methods of AsciiTraceHelperForIpv4 will call this one
- Can provide EnableAsciiIpv4 with Ptr<Ipv4>, string from name service, Ipv4InterfaceContainer, NodeContainer, integer node/device ids
- Or helper.EnableAsciiIpv4All("prefix");
- Can also dump ascii traces to a single common file
  - So there are twice as many trace methods as for pcap
- Filename selection
  - Convention is <prefix>-n<node id>-i<interface id>.pcap
  - Can also use the NS3 object name service clarity
  - explicitFilename parameter lets you impose your own filenames

Conclusion

- NS3 is very powerful/comprehensive
- Lots of tools for you to use
  - Helpers
  - Containers
  - Logging
  - Tracing
  - Models
- Doxygen is your friend!

Practice! Practice! Practice!