Goals of this tutorial

- Understand the goals of the ns-3 project
- Learn what has been done to achieve these goals
- Identify future work directions

Tutorial schedule

1. 14h00-15h00: Introduction
2. 15h00-16h00: The ns-3 architecture
3. 16h00-17h00: The ns-3 object model

Part I

Introduction
Recent history (1995-2005)

- ns-2 became the main choice for research usage. Search of ACM Digital Library papers citing simulation, 2001-04:

<table>
<thead>
<tr>
<th>Layer</th>
<th>ns-2</th>
<th>OPNET</th>
<th>QualNet/Glomosim</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥ layer 4</td>
<td>123 (75%)</td>
<td>30 (18%)</td>
<td>11 (7%)</td>
</tr>
<tr>
<td>= layer 3</td>
<td>186 (70%)</td>
<td>48 (18%)</td>
<td>31 (12%)</td>
</tr>
<tr>
<td>≤ layer 2</td>
<td>114 (43%)</td>
<td>96 (36%)</td>
<td>55 (21%)</td>
</tr>
</tbody>
</table>

- Funding for ns-2 development dropped in the early 2000’s

What is wrong about ns-2?

- Split object model (OTcl and C++) and use of Tcl:
  - Doesn’t scale well
  - Makes it difficult for students
- Large amount of abstraction at the network layer and below leads to big discontinuities when transitioning from simulation to experiment
- Accretion of unmaintained and incompatible models
- Lack of support for creating methodologically sound simulations
- Lack of, and outdated, documentation
- In ns-2, validation really means regression: no documented validation of the models, outside of TCP
Overheard on e2e-interest mailing list

September 2005 archives of the e2e-interest mailing list:

• "...Tragedy of the Commons..."
• "...around 50% of the papers appeared to be... bogus..."
• “Who has ever validated NS2 code?”
• “To be honest, I’m still not sure whether I will use a simulation in a paper.”
• "...I will have a hard time accepting or advocating the use of NS-2 or any other simulation tool"

A recurring misconception

• Using ns-2 is actively harmful

• Simulation is ns-2

Thus, simulation is actively harmful
Back in 2000’s, the rise of testbeds

- Hardware costs going down
- OS virtualization going up
- Development of control and management software

Result:
- Emulab: http://www.emulab.net
- ORBIT: http://www.orbit-lab.org
- Planetlab: http://planet-lab.org
- ModelNet: https://modelnet.sysnet.ucsd.edu
- ...

Why do we need simulation at all?

- Simulation models are not validated
- Simulation model implementations not verified
- No need for validation and verification in testbeds

However, there are lots of good things about simulation:
- Reproducibility
- Easier to setup, deploy, instrument
- Investigate non-existent systems
- Scalability
But, really, we need both!

We want to get the best from both worlds:
- Simulators: reproducibility, debuggability, ease of setup
- Testbeds: realism

We want an integrated experimentation environment:
- Use each tool separately:
  - Parameter space exploration with simulations
  - More realism with testbeds
- Use both tools together:
  - Simulator for elements of the topology to scale
  - Testbed for other elements to get realism

Summary

We need simulations:
- Easier to use, debug, reproduce than testbeds
- Not constrained by existing hardware/software

We need a special simulator:
- Improves model validation
- Improves model implementation verification
- Allow users to move back and forth between simulation and testbeds

Outline

Starting from ns-2

The biggest reason to start from ns-2 is:
- A large existing userbase
- A large set of existing models

But, we need to address many issues:
- Most existing models lack validation, verification, maintenance
- Bi-language system (C++/tcl) makes debugging complex: removing it would mean dropping backward compatibility
- Core packet data-structure:
  - Inappropriate for emulation
  - Fragmentation unsupported

Re-engineering ns-2 to fix all these issues would make it a new different simulator: we would lose our existing userbase.
Proprietary simulators

There are many of them (google for network simulator):

- Opnet
- QualNet
- Shunra
- etc.

But:

- Terms of use
- Very costly for industrial partners or publicly-funded research which cannot get education licenses.

Omnetpp

- It was not clear in 2005 it would still be alive in 2009
- Major worries over the bi-language architecture: learning curve, debugging, etc.
- Software structure did not seem to lend itself to the realism we sought.

Not Invented Here

Yes, we did fall prey to that syndrome too: we thought we could do it better than the others

Outline

Simulation considered harmful

Why not reuse an existing simulator?

What is so special about ns-3?

What we learned along the way
Good debuggability

C++-only simulations: no need to debug two languages at the same time
- ns-3 is a library written in C++
- Simulation programs are C++ executables
- Bindings in Python for python simulations

Long term project lifetime

A open source community:
- An open license (GPLv2)
- All design and implementation discussions in the open on mailing-lists (even flame wars)
- Everyone can (should) become a maintainer
This is critical to allow:
- The project to scale to many models
- The project to last beyond initial seed funding
- Model/implementations reviews in the open:
  Given enough eyeballs, all bugs are shallow

Low cost of model validation

Make models close to the real world:
- Models are less abstract: easier to validate
- Makes it easy to perform direct execution of real code
- Emulation is native and robust against changes in models
How ?
- Real IP addresses
- Multiple interfaces per node
- Bsd-like sockets
- Packets contain real network bytes

A usecase: NSC
NSC implementation

- Globalizer: per-process kernel source code and add indirection to all global variable declarations and accesses
- Glue: per-kernel (and per-kernel-version) glue to provide kernel APIs for kernel code:
  - kmalloc: memory allocation
  - NetDevice integration
  - Socket integration
- Provides glue for:
  - Linux 2.6.18, 2.6.26, 2.6.28
  - FreeBSD 5
  - lwip 1.3
  - OpenBSD 3

NSC accuracy

- Have shown NSC to be very accurate – able to produce packet traces that are almost identical to traces measured from a test network

Summary

ns-3 has a strong focus on realism:
- Makes models closer to the real world: easier to validate
- Allows direct code execution: no model validation
- Allows robust emulation for large-scale and mixed experiments

ns-3 also cares about good software engineering:
- Single-language architecture is more robust in the long term
- Open source community ensures long lifetime to the project

Outline

- Simulation considered harmful
- Why not reuse an existing simulator?
- What is so special about ns-3?
- What we learned along the way
Things You Should Never Do

It's an old axiom of software engineering: **Don't rewrite from scratch, ever.**

We did not really start from scratch:
- Stole code and concepts from *GTNetS* (applications)
- Stole code and concepts from *yans* (wifi)
- Stole code and concepts from *ns-2* (olsr, error models)

Even then, it took us 2 years to get to a useful state.

Need for integrated statistical tools

Initially, we thought we could:
- Allow users to easily instrument the system
- Delegate analysis to third-party tools such as *R*

It does not work that way though:
- Lack of methodology documentation
- Fancy statistical tools are too complex for most users

Future work: integrate tools to automatically measure and improve confidence intervals on simulation output.

Building an open source community is hard

It’s a lot of work to attract contributors and keep them: they want to have fun, they want to have impact on the project:
- Never flame people on mailing-lists:
  - Always answer questions kindly, point out manuals and FAQ
  - Don’t answer provocative statements
  - English is not the native language of most users
- We need to do the boring work (release management, bug tracking, server maintenance)
- No discussion *behind closed doors*: increases communication cost
- It’s a meritocracy: those who contribute the most should have power to decide for the project

Need for a high-level experimentation environment

*ns-3* provides low-level functionality:
- Tap devices
- Realtime simulation core

But we want to allow easy switching and mixing of simulation and testbeds. We need higher-level abstractions for:
- Experiment description (topology, application traffic)
- Experiment configuration
- Tracing configuration
- Deployment automation

Work towards this is underway with NEPI (ROADS’09: **NEPI: Using Independent Simulators, Emulators, and Testbeds for Easy Experimentation**:

http://www-sop.inria.fr/members/Mathieu.Lacage/roads09-nepi.pdf)
Need for more direct code execution I

Integrate normal POSIX network applications in the simulator:
- No source code modifications
- Easy to debug (great network application development platform!)

Needs:
- Globalization: global variables must be virtualized for each instance of the application running in the simulator
- Filesystem virtualization: each application needs a separate filesystem (to get different configuration and log files for example)
- Socket library: need a complete implementation of sockets in the simulator, including all the crazy ioctls

Need for more direct code execution II

Status:
- Running demonstrations with ping, traceroute
- Simple socket applications can run: a couple of threads, select, tcp server/client
- Larger applications using fancy socket ioctls don’t work very well yet

Part II

The ns-3 architecture

Outline

Introduction

Fundamental network model structure

Topology construction
Environment setup

Install all needed tools:

Ubuntu

```
sudo apt-get install build-essential g++ python mercurial
```

Windows

- cygwin
- python
- mercurial

Getting ns-3

Availability (linux, osx, cygwin, mingw):
- Released tarballs: http://www.nsnam.org/releases
- Development version: http://code.nsnam.org/ns-3-dev

The development version is usually stable: a lot of people use it for daily work:

```
hg clone http://code.nsnam.org/ns-3-dev
```

Running ns-3

Use waf to build it (similar to make):

```
./waf
./waf shell
./build/debug/examples/csma-broadcast
```
Exploring the source code

A typical simulation

- Create a bunch of C++ objects
- Configure and interconnect them
- Each object creates events with Simulator::Schedule
- Call Simulator::Run to execute all events

A (fictional) simulation

```
Node *a = new Node();
Node *b = new Node();
Link *link = new Link(a,b);
Simulator::Schedule(Seconds(0.5), // in 0.5s from now
 &Node::StartCbr, a, // call StartCbr on 'a'
   "100bytes", "0.2ms", b); // pass these arguments
Simulator::Run();
```

Outline

- Introduction
- Fundamental network model structure
- Topology construction

The basic model
The fundamental objects

- **Node**: the motherboard of a computer with RAM, CPU, and IO interfaces
- **Application**: a packet generator and consumer which can run on a Node and talk to a set of network stacks
- **Socket**: the interface between an application and a network stack
- **NetDevice**: a network card which can be plugged into an IO interface of a Node
- **Channel**: a physical connector between a set of NetDevice objects

**Important remark**

NetDevices are strongly bound to Channels of a matching type:

Existing models

- **Network stacks**: arp, ipv4, icmpv4, udp, tcp (ipv6 under review)
- **Devices**: wifi, csma, point-to-point, bridge
- **Error models and queues**
- **Applications**: udp echo, on/off, sink
- **Mobility models**: random walk, etc.
- **Routing**: olsr, static global

For example, the wifi models

- New model, written from 802.11 specification
- Accurate model of the MAC
- DCF, beacon generation, probing, association
- A set of rate control algorithms (ARF, ideal, AARF, Minstrel, etc.)
- Not-so-slow models of the 802.11a PHY
Development of wifi models

New contributions from many developers:

- University of Florence: 802.11n, EDCA, frame aggregation, block ack
- Russian Academy of Sciences: 802.11s, HWMP routing protocol
- Boeing: 802.11b channel models, validation
- Deutsche Telekom Laboratories: PHY modelization, validation
- Karlsruhe Institute of Technology: PHY modelization (Rayleigh, Nakagami)

Summary

- Core models are based on well-known abstractions: sockets, devices, etc.
- An active community of contributors

Outline

- Introduction
  - Fundamental network model structure
  - Topology construction
**The Helper/Container API**

We want to:
- Make it easy to build topologies with repeating patterns
- Make the topology description more high-level (and less verbose) to make it easier to read and understand

The idea is simple:
- Sets of objects are stored in Containers
- One operation is encoded in a Helper object and applies on a Container

Helper operations:
- Are not generic: different helpers provide different operations
- Do not try to allow code reuse: just try to minimize the amount of code written
- Provide *syntactical sugar*: make the code easier to read

**Typical containers and helpers**

Example containers:
- NodeContainer
- NetDeviceContainer
- Ipv4AddressContainer

Example helper classes:
- InternetStackHelper
- WifiHelper
- MobilityHelper
- OlsrHelper
- etc. Each model provides a helper class

**Create a couple of nodes**

Create empty node container
Create two nodes
Create empty node container
Add existing node to it
And then create some more nodes

**Then, the csma network**

Create empty device container
Create csma helper
Set data rate
Set delay
Create csma devices and channel
And a couple of wifi interfaces

Finally, setup the wifi channel:

```cpp
YansWifiChannelHelper wifiChannel = YansWifiChannelHelper::Default();
YansWifiPhyHelper wifiPhy = YansWifiPhyHelper::Default();
wifiPhy.SetChannel(wifiChannel.Create());
```

And create adhoc devices on this channel:

```cpp
NetDeviceContainer wifiDevices;
WifiHelper wifi = WifiHelper::Default();
wifiDevices = wifi.Install(wifiPhy, wifiNodes);
```

Comparison with low-level version

Fire up editor for tutorial-helper.cc and tutorial-lowlevel.cc

Summary

- It's always possible to create objects by hand, interconnect and configure them
- But it can be easier to reuse the for loops encapsulated in Helper classes

Summary

- It's always possible to create objects by hand, interconnect and configure them

Summary

• It's always possible to create objects by hand, interconnect and configure them
• But it can be easier to reuse the for loops encapsulated in Helper classes
• Helper classes make scripts less cluttered and easier to read and modify

Outline

A coherent memory management scheme
Maximizing model reuse
Getting the right object
A uniform configuration system
Controlling trace output format
The underlying type metadata database

Part III
The ns-3 object model

It’s easy to build a network simulator

It’s just a matter of:
• Provide an event scheduler
• Implement a couple of models to create and consume events
But it’s much harder to build a network simulator which:
• Allows models to be reusable independently
• Ensures API coherence between models
• Automates common tasks (tracing, configuration)
Why are objects so complicated to create?

We do:

```cpp
Ptr<Node> node0 = CreateObject<Node>();
```

Why not:

```cpp
Node *node0 = new Node();
```

Or:

```cpp
Node node0 = Node();
```

Templates: the Nasty Brackets

- Contain a list of type arguments
- Parameterize a class or function from input type
- In ns-3, used for:
  - Standard Template Library
  - Syntactical sugar for low-level facilities
- Saves a lot of typing
- No portability/compiler support problem
- Sometimes painful to decipher error messages.

Memory Management

It is hard in C++:

- No garbage collector
- Easy to forget to delete an object
- Pointer cycles
- Ensure coherency and uniformity

So, we use:

- Reference counting: track number of pointers to an object (Ref+Unref)
- Smart pointers: Ptr<>, Create<> and, CreateObject<>
Where is my MobileNode?

- Some nodes need an IPv4 stack, a position, an energy model.
- Some nodes need just two out of three.
- Others need other unknown features.
- The obvious solution: add everything to the Node base class:
  - The class will grow uncontrollably over time
  - Everyone will need to patch the class
  - Slowly, every piece of code will depend on every other piece of code (cannot reuse anything without dragging in everything)
  - A maintenance nightmare...
- A better solution:
  - Separate functionality belongs to separate classes
  - Objects can be aggregated at runtime to obtain extra functionality

Outline

- A coherent memory management scheme
- Maximizing model reuse
- Getting the right object
- A uniform configuration system
- Controlling trace output format
- The underlying type metadata database

Object aggregation

- A circular singly linked-list
- AggregateObject is a constant-time operation
- GetObject is a O(n) operation
- Aggregate contains only one object of each type
The traditional approach

In C++, if you want to call methods on an object, you need a pointer to this object. To get a pointer, you need to:

- keep local copies of pointers to every object you create
- walk pointer chains to get access to objects created within other objects

For example, in ns-3, you could do this:

```cpp
Ptr<NetDevice> dev = NodeList::Get (5)->GetDevice (0);
Ptr<WifiNetDevice> wifi = dev->GetObject<WifiNetDevice> ();
Ptr<WifiPhy> phy = dev->GetPhy ();
phy->SetAttribute("TxGain", ...);
phy->ConnectTraceSource (...);
```

It’s not fun to do...

Use a namespace string!

Set an attribute:

```cpp
Config::SetAttribute("/NodeList/5/DeviceList/0/Phy/TxGain", StringValue("10"));
```

Connect a trace sink to a trace source:

```cpp
Config::Connect("/NodeList/5/DeviceList/0/Phy/TxGain", MakeCallback(&LocalSink));
```

Just get a pointer:

```cpp
Config::MatchContainer match; match = Config::LookupMatches("/NodeList/5/DeviceList/0/Phy/");
Ptr<WifiPhy> phy = match.Get (0)->GetObject<WifiPhy> ();
```

The object namespace I

Object namespace strings represent a path through a set of object pointers:

- `/NodeList/0` represents the first node
- `/DeviceList/0` represents the first device
- `/SendEnable` represents the SendEnable attribute of the device
- `/FrameSize` represents the FrameSize attribute of the device
- `/DataRate` represents the DataRate attribute of the device
- `/InterframeGap` represents the InterframeGap attribute of the device

For example, `/NodeList/x/DeviceList/y/InterframeGap` represents the InterframeGap attribute of the device number `y` in node number `x`.

The object namespace II

Navigating the attributes using paths:

- `/NodeList/[3-5]8[0-1]`: matches nodes index 0, 1, 3, 4, 5, 8
- `/NodeList/*`: matches all nodes
- `/NodeList/3/Sns3::Ipv4`: matches object of type `ns3::Ipv4` aggregated to node number 3
- `/NodeList/3/DeviceList/*/Sns3::CsmaNetDevice`: matches all devices of type `ns3::CsmaNetDevice` within node number 3
- `/NodeList/3/DeviceList/0/RemoteStationManager`: matches the object pointed to by attribute RemoteStationManager in device 0 in node 3.
Outline

- A coherent memory management scheme
- Maximizing model reuse
- Getting the right object
- A uniform configuration system
- Controlling trace output format
- The underlying type metadata database

Traditionally, in C++

- Export attributes as part of a class's public API
- Use static variables for defaults

For example:

```cpp
class MyModel {
    public:
        MyModel () : m_foo (m_defaultFoo) {}
        void SetFoo (int foo) {m_foo = foo;}
        int GetFoo (void) {return m_foo}
        static void SetDefaultFoo (int foo) {m_defaultFoo = foo;}
        static int GetDefaultFoo (void) {return m_defaultFoo;}
    private:
        int m_foo;
        static int m_defaultFoo = 10;
};
```

In ns-3, done automatically I

- Set a default value:
  ```cpp```
  Config::SetDefaultValue ("ns3::WifiPhy::TxGain", StringValue ("10");
  ```cpp```

- Set a value on a specific object:
  ```cpp```
  phy->SetAttribute ("TxGain", StringValue ("10");
  ```cpp```

- Set a value from the command-line -ns3::WifiPhy::TxGain=10:
  ```cpp```
  CommandLine cmd;
  cmd.Parse (argc, argv);
  ```cpp```

In ns-3, done automatically II

- Load, Change, and Save all values from and to a raw text or xml file with or without a GUI:
  ```cpp```
  GtkConfigStore config;
  config.ConfigureDefaults ();
  ...
  config.ConfigureAttributes ();
  ```cpp```

- Set a value with an environment variable
  ```cpp```
  NS_ATTRIBUTE_DEFAULT=ns3::WifiPhy::TxGain=10
  ```cpp```
Outline

A coherent memory management scheme
Maximizing model reuse
Getting the right object
A uniform configuration system
Controlling trace output format
The underlying type metadata database

Tracing requirements

- Tracing is a structured form of simulation output
- Example (from ns-2):
  
  + 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
  - 1.84375 0 2 cbr 210 ------- 0 0.0 3.1 225 610
  r 1.84471 2 1 cbr 210 ------- 1 3.0 1.0 195 600
  r 1.84566 2 0 ack 40 ------- 2 3.2 0.1 82 602
  + 1.84566 0 2 tcp 1000 ------- 2 0.1 3.2 102 611
- Problem: tracing needs vary widely
  - Would like to change tracing output format without editing the core
  - Would like to support multiple output formats
Tracing overview

- Simulator provides a set of pre-configured trace sources
  - Users may edit the core to add their own
- Users provide trace sinks and attach to the trace source
  - Simulator core provides a few examples for common cases
- Multiple trace sources can connect to a trace sink

Ns-3 trace sources

- Various trace sources (e.g., packet receptions, state machine transitions) are plumbed through the system
- Organized with the rest of the attribute system

The ns-3 tracing model

Decouple trace sources from trace sinks:

Benefit: Customizable trace sinks

Multiple levels of tracing

- High-level: use a helper to hook a predefined trace sink to a trace source and generate simple tracing output (ascii, pcap)
- Mid-level: hook a special trace sink to an existing trace source to generate adhoc tracing
- Low-level: add a new trace source and connect it to a special trace sink
High-level tracing

- Use predefined trace sinks in helpers
- All helpers provide ascii and pcap trace sinks

```cpp
CsmaHelper::EnablePcap("filename", nodeid, deviceid);
std::ofstream os;
os.open("filename.tr");
CsmaHelper::EnableAscii(os, nodeid, deviceid);
```

Mathieu Lacage (INRIA)  Experimentation with ns-3  Trilogy'2009  86 / 95

Mid-level tracing

- Provide a new trace sink
- Use attribute/trace namespace to connect trace sink and source

```cpp
void
DevTxTrace (std::string context,
            Ptr<const Packet> p, Mac48Address address)
{
    std::cout << "TX to=" << address << " p: " << *p << std::endl;
}
Config::Connect("/NodeList/*/DeviceList/*/Mac/MacTx",
                MakeCallback(&DevTxTrace));
```

Mathieu Lacage (INRIA)  Experimentation with ns-3  Trilogy'2009  87 / 95

Pcap output

The trace sink:

```cpp
static void PcapSnifferEvent (Ptr<PcapWriter> writer, 
                              Ptr<const Packet> packet)
{
    writer->WritePacket (packet);
}
```

Prepare the pcap output:

```cpp
oss << filename << "." << nodeid << "." << deviceid << ".pcap";
Ptr<PcapWriter> pcap = ::ns3::Create<PcapWriter> ();
pcap->Open (oss.str ());
pcap->WriteWifiHeader ();
```

Finally, connect the trace sink to the trace source:

```cpp
oss << "/NodeList/" << nodeid << "/DeviceList/" << deviceid;
oss << "/Ns3::WifiNetDevice/Phy/PromiscSniffer";
Config::ConnectWithoutContext (oss.str ());
    MakeBoundCallback (&PcapSnifferEvent, pcap);
```

Mathieu Lacage (INRIA)  Experimentation with ns-3  Trilogy'2009  88 / 95

Outline

- A coherent memory management scheme
- Maximizing model reuse
- Getting the right object
- A uniform configuration system
- Controlling trace output format
- The underlying type metadata database
The ns-3 type system

- The aggregation mechanism needs information about the type of objects at runtime
- The attribute mechanism needs information about the attributes supported by a specific object
- The tracing mechanism needs information about the trace sources supported by a specific object

All this information is stored in `ns3::TypeId`:
- The parent type
- The name of the type
- The list of attributes (their name, their type, etc.)
- The list of trace sources (their name, their type, etc.)

Mathieu Lacage (INRIA)
Experimentation with ns-3
Trilogy'2009 91 / 95

Summary

- Memory management is uniform and simple
Summary

- Memory management is uniform and simple
- Dynamic aggregation makes models easier to reuse
- Path strings allow access to every object in a simulation
- Attributes allow powerful and uniform configuration
- Trace sources allow arbitrary output file formats
Summary

- Simulation is a key component of network research
  - Debuggability
  - Reproducibility
  - Parameter exploration
  - No dependency on existing hardware/software
- ns-3 has a strong focus on realism:
  - Makes models closer to the real world: easier to validate
  - Allows direct code execution: no model validation
  - Allows robust emulation for large-scale and mixed experiments
- ns-3 also cares about good software engineering:
  - Single-language architecture is more robust in the long term
  - Open source community ensures long lifetime to the project

Resources

- Web site: http://www/nsnam.org
- Developer mailing list: http://mailman.isi.edu/mailman/listinfo/ns-developers
- User mailing list: http://groups.google.com/group/ns-3-users
- IRC: #ns-3 at irc.freenode.net
- Tutorial: http://www.nsnam.org/docs/tutorial/tutorial.html
- Code server: http://code.nsnam.org

Acknowledgments

- Many slides stolen from other’s presentations and tutorials
- Many contributors to the ns-3 codebase (developers, testers)
- Google summer of code students (2008,2009)