Modeling of data networks by example: ns-2 (I)

Holger Füßler

Course overview

1. Introduction
2. Building block: RNG
3. Building block: Generating random variates I and modeling examples
4. Building block: Generating random variates II and modeling examples
5. Algorithmics: Management of events
6. NS-2: Introduction
7. NS-2: Fixed networks
8. NS-2: Wireless networks
9. Output analysis: single system
10. Output analysis: comparing different configuration
11. Omnet++ / OPNET
12. Simulation lifecycle, summary

Outline of this lecture

- » Part I: What and why of ns-2
- » Part II: Ns-2 overall structure and a basic ns-2 example
  - – Scenario specification with tcl, otcl
  - – Simulator object
  - – Generic structure of a ns-2 simulation script
  - – Ns-2: basic otcl script for UDP traffic
- » Part III: First look into ns-2 internals
- » Part IV: Another example from ns tutorial

I A brief history of … ns-2

- 1989: REAL (‘realistic and large’) network simulator at University of California, Berkeley
- 1995: DARPA VINT (‘Virtual Inter-Network Testbed‘) project; LBL, Xerox PARC, UCB, USC/ISI
  - Developed ns-2 as their simulation tool
  - Nice overview paper: Lee Breslau et al., *Advances in network simulation*, IEEE Computer, May 2000

    “Network researchers must test Internet protocols under varied conditions to determine whether they are robust and reliable. The Virtual Inter-Network Testbed (VINT) project has enhanced its network simulator and related software to provide several practical innovations that broaden the conditions under which researchers can evaluate network protocols.”
- Currently: DARPA SAMAN and NSF CONSER projects develop ns-2

I Goals of ns-2

- Support networking research and education
  - Protocol design, traffic studies, etc
  - Protocol comparison

Network education: visualizations and educational scripts

- Provide a collaborative environment
  - Freely distributed, open source
    - Share code, protocols, models, etc
  - Allow easy comparison of similar protocols
  - Increase confidence in results
    - More people look at models in more situations
    - Experts develop models

http://www.isi.edu/nsnam/dist

- Multiple levels of detail in one simulator
  - Packet level
  - Session level

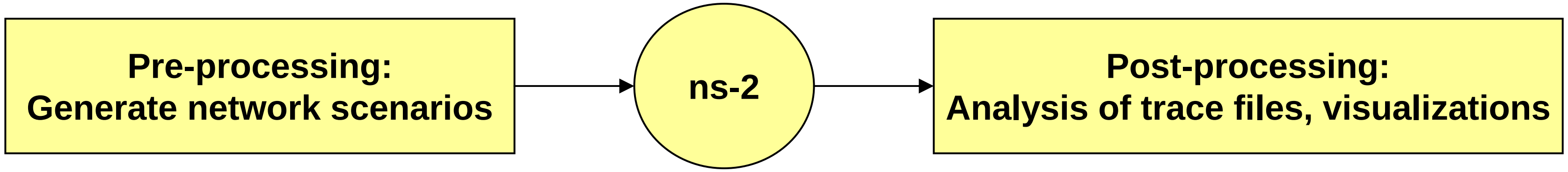
We focus on packet level simulations

[Source: Ns Tutorial 2002, Padmaparna Haldar]

I Elements of ns ‘package’

[Source: Ns Tutorial 2002, Padmaparna Haldar]

- **Ns, the simulator itself**
- **Nam, the network animator**
  - **Visualize ns (or other) output**
  - **Nam editor: GUI interface to generate ns scripts**
- **Pre-processing:**
  - **Traffic and topology generators**
- **Post-processing:**
  - **Simple trace analysis, often in Awk, Perl, or Tcl**


Pre-processing:
Generate network scenarios
ns-2
Post-processing:
Analysis of trace files, visualizations

I Current status of ns-2

- Ns-2: most recent release is ns-2.27
  - Daily snapshots available
  - Full validation suite
- Nam: most recent release is nam-1.10
- Ns-2 is pretty large:
  - Requires about 250 MB disk space
  - More than 200 K lines of code
- Available for Linux, FreeBSD, SunOS, Solaris
  - Also runs on Windows 9x/2000/XP with cygwin
- Functionality:
  - Wired world: various routing methods, multicast, ‘all’ flavors of TCP, UDP, various traffic sources, various queuing disciplines, quality of service mechanisms, …
  - Wireless world: ad hoc routing, mobile IP, directed diffusion, sensor-MAC, …

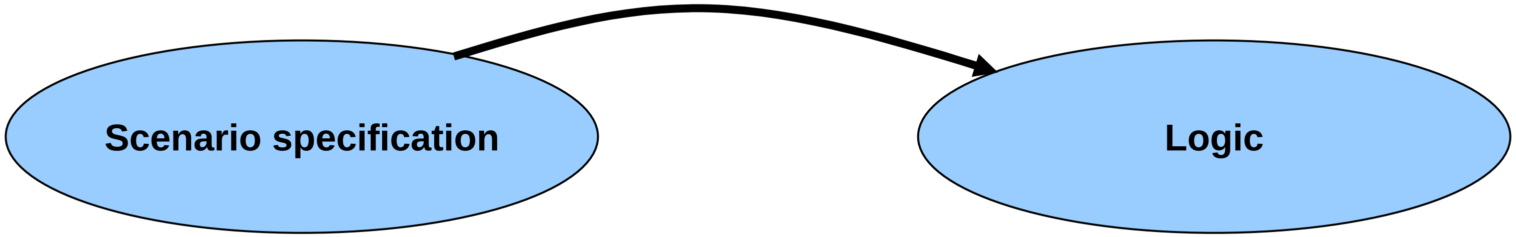
I Our goal

- Learn how to generate network scenarios
- Learn how to run a simulation
- Learn how to analyze simulation output
- Understand how ns-2 works internally

- Our focus is not on how to implement new functionality

- Lecture 6 (today): introduction to ns-2
- Lecture 7: experiments with TCP using ns-2
- Lecture 8: experiments with wireless ad hoc networks using ns-2

II What do we want/have to model?



Application layer protocol

Transport protocol

Routing protocol

Queues

Packets

Nodes

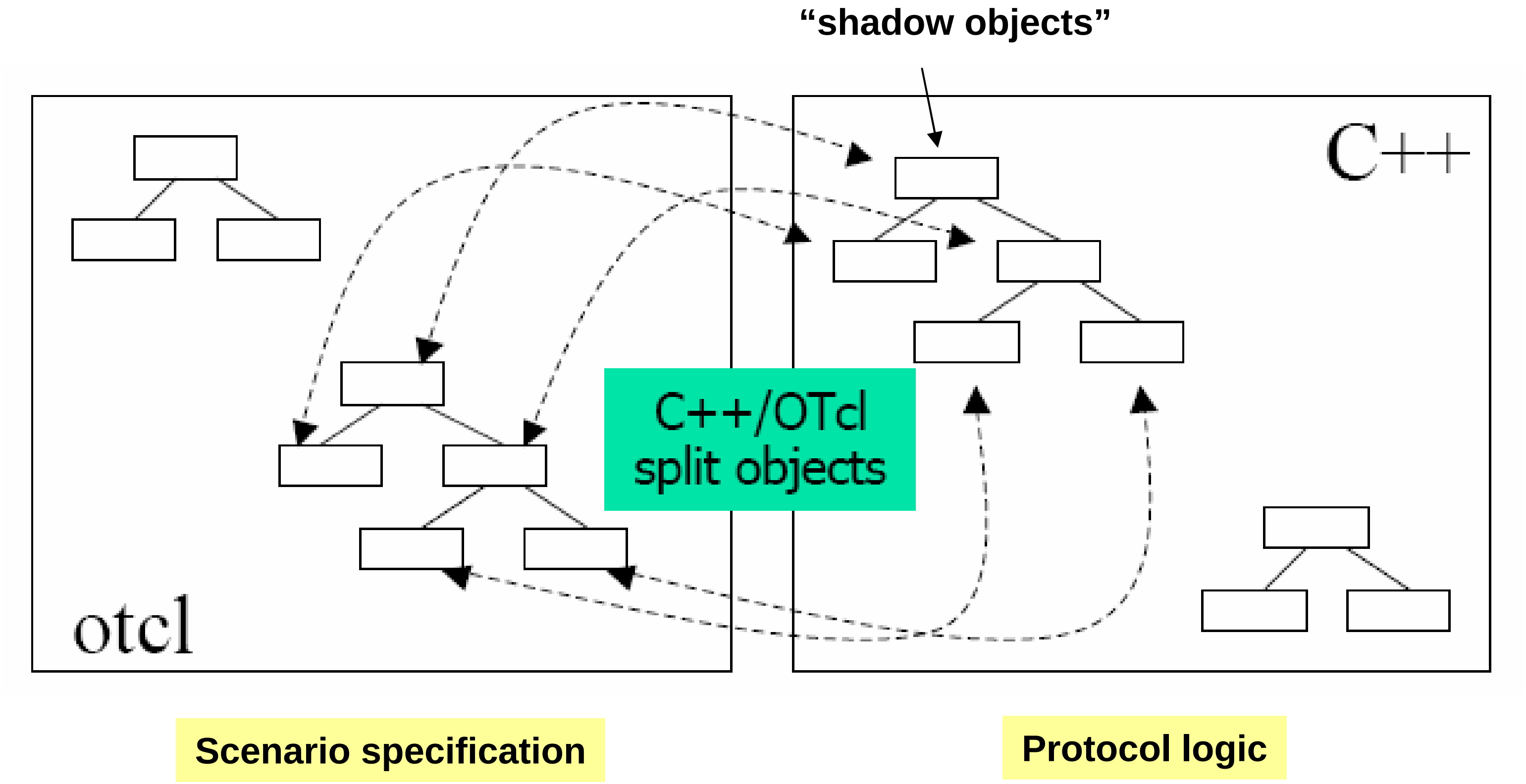
Links

Implementation of
- **Application layer protocol**
- **Transport protocol**
- **Routing protocol**
- **Queue behavior**
- **Link behavior**
- **…**

II Requirements

- Scenario specification ‘language’
  - We want to experiment easily with various scenarios without recompiling protocol logic
- Language for implementing protocol logic
  - Speed is important aspect
- Both should be object-oriented
  - Reusability
  - Extensibility (method overloading)

II NS-2 overall architecture



» OTcl: Object version of the 'Tool Command Language'

– Tcl intro: http://www.tcl.tk/scripting/

II Tcl: Ousterhout on scripting languages

Scripting: Higher Level Programming for the 21st Century, John K. Ousterhout
IEEE Computer magazine, March 1998

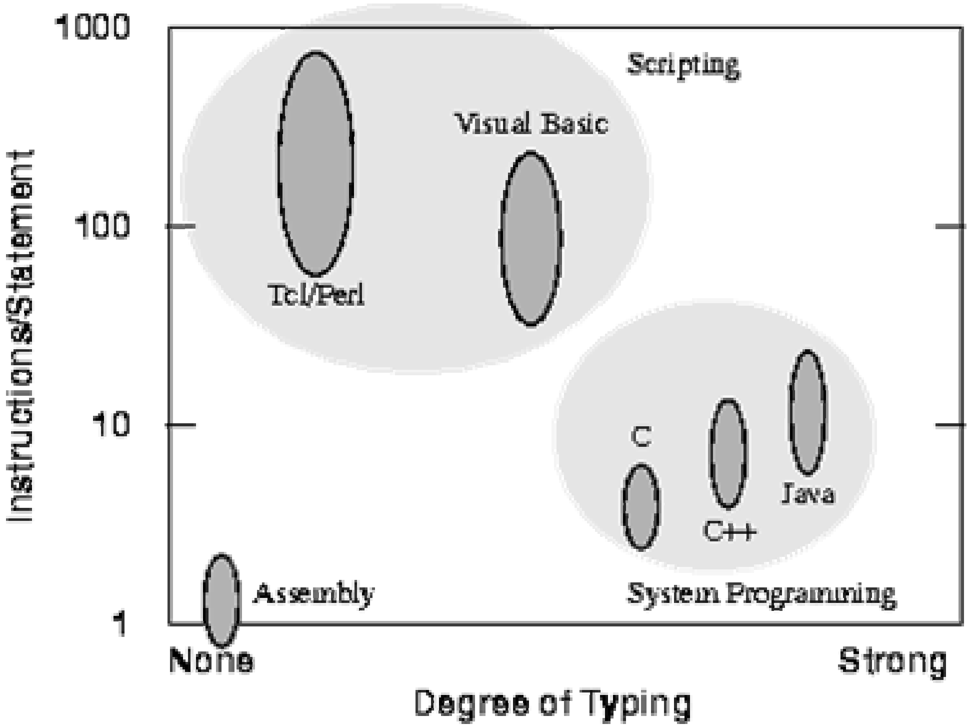


Figure 1. A comparison of various programming languages based on their level (higher level languages execute more machine instructions for each language statement) and their degree of typing. System programming languages like C tend to be strongly typed and medium level (5-10 instructions/statement). Scripting languages like Tcl tend to be weakly typed and very high level (100-1000 instructions/statement).

Scipting:

- **System integration language**
- **Interpreted**
- **Typeless**

II Tcl: basic commands

» Variables

- `set x 10`
- `puts “x is $x”`

» Functions and expressions

- `set y [pow x 2]`
- `set y [expr x*x]`

» Procedures

-
  ```
  proc pow {x n} {
    if {$n == 1} { return $x }
    set part [pow x [expr $n-1]]
    return [expr $x*$part]
    }
  ```

» Control flow

```
if {$x > 0} { return $x } else {
    return [expr -$x]
    }
while { $x > 0 } {
    puts $x
    incr x –1
    }
```

II Object Tcl (Otcl): basic commands

```
Class Person
```

# constructor:

```
Person instproc init {age} {
   $self instvar age_
   set age_ $age
}
```

# method:

```
Person instproc greet {} {
   $self instvar age_
   puts “$age_ years old: How
   are you doing?”
}
```

# subclass:

```
Class Kid -superclass Person
Kid instproc greet {} {
   $self instvar age_
   puts “$age_ years old kid:
   What’s up, dude?”
}
```

```
set a [new Person 45]
set b [new Kid 15]
$a greet
$b greet
```

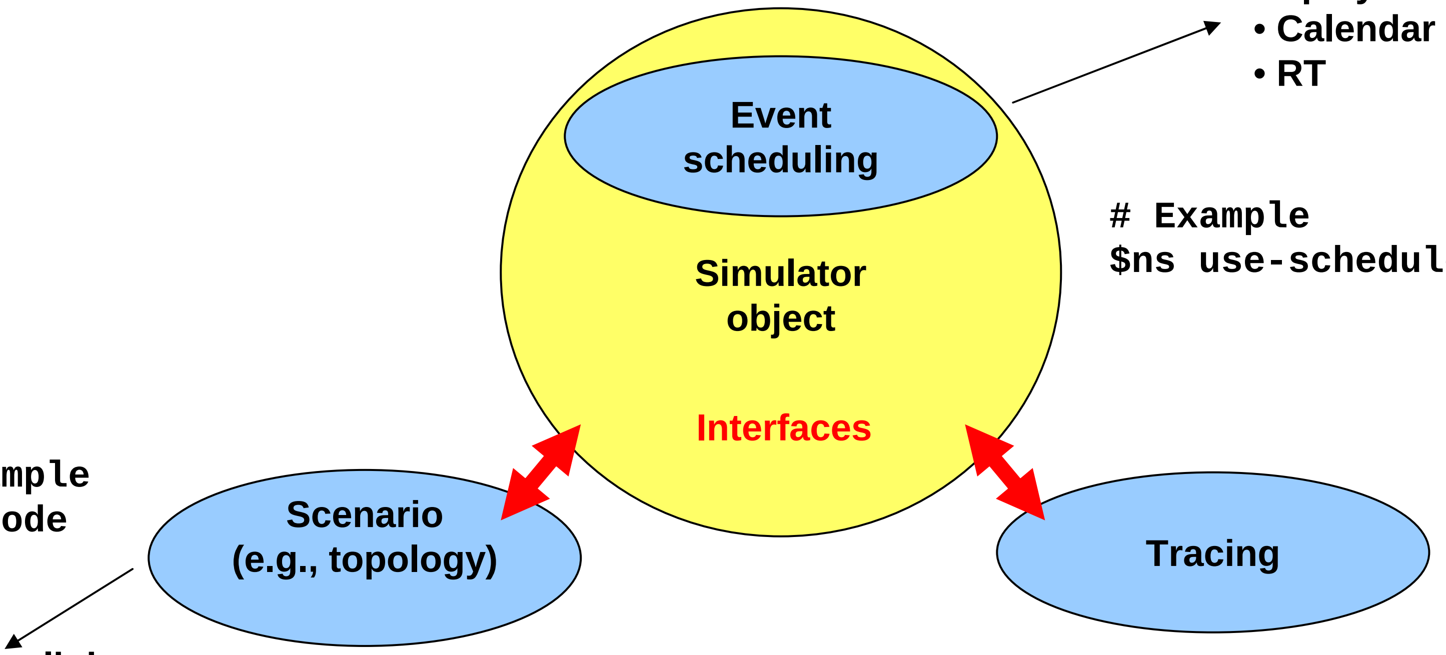
=> can easily make variations of existing things (TCP, TCP/Reno)

[Source: Ns-2 tutorial, P. Haldar, X. Chen, 2002]

II Ns-2: Class Simulator

```
# Create an instance of Class Simulator

set ns [new Simulator]
```

Scheduler:
- List
- Heap
- Splay
- Calendar
- RT



```
# Example
$ns use-scheduler Heap
```

```
# Example
$ns node
```

Nodes, links

set ns [new Simulator] ↔ Otcl interpreter ↔ Interpreted hierarchy ↔ Compiled hierarchy

Otcl C++

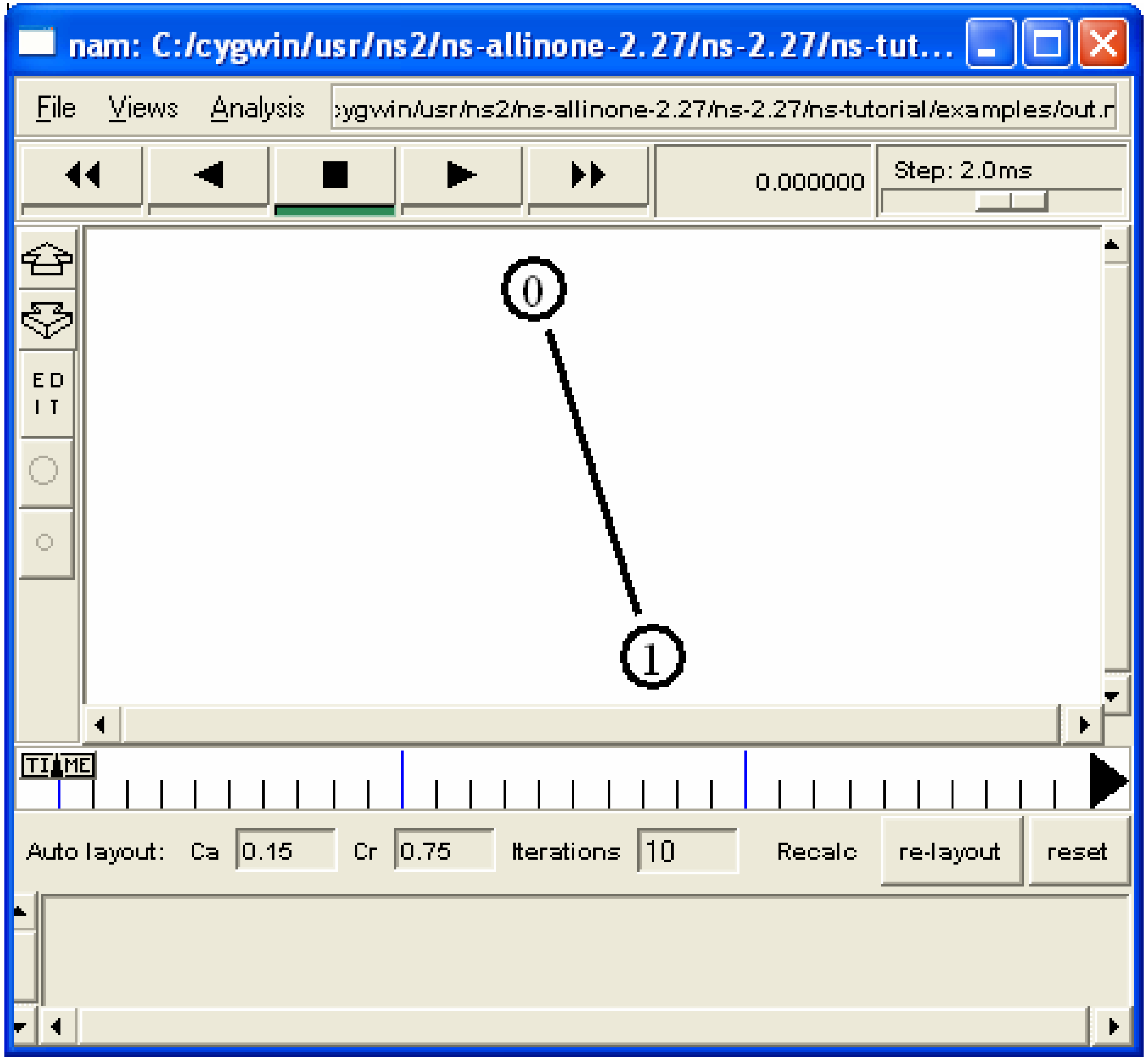
II Ns-2: a simple otcl script

```
#Create a simulator object
set ns [new Simulator]

#Open the nam trace file
set nf [open out.nam w]
$ns namtrace-all $nf

#Define a 'finish' procedure
proc finish {} {
        global ns nf
        $ns flush-trace
    #Close the trace file
        close $nf
    #Execute nam on the trace file
        exec nam out.nam &
        exit 0
}
```

```
#Create two nodes
set n0 [$ns node]
set n1 [$ns node]

#Create a duplex link between the nodes
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

#Call the finish procedure after 5 seconds
    of simulation time
$ns at 5.0 "finish"

#Run the simulation
$ns run
```

[Source: example1a.tcl, ns-tutorial]

II Nam output



nam: C:/cygwin/usr/ns2/ns-allinone-2.27/ns-2.27/ns-tut...

:ygwin/usr/ns2/ns-allinone-2.27/ns-2.27/ns-tutorial/examples/out.r

0.000000

Step: 2.0ms

Auto layout: Ca 0.15 Cr 0.75 Iterations 10

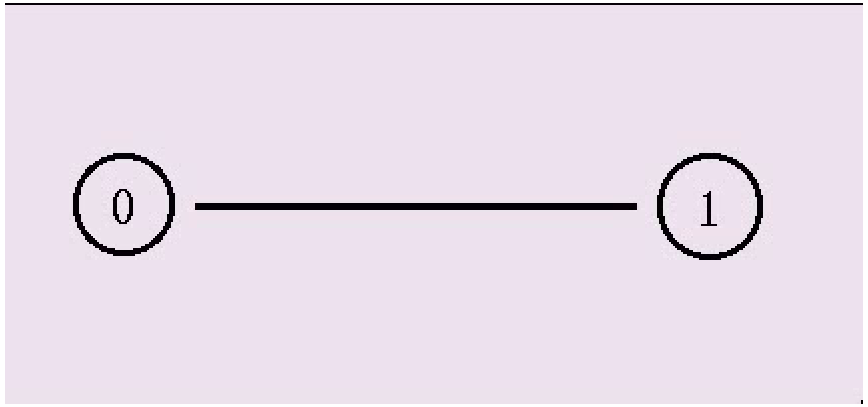
II Ns-2: add data traffic (UDP)

```
#Create a UDP agent and attach it to
    node n0

set udp0 [new Agent/UDP]

$ns attach-agent $n0 $udp0


# Create a CBR traffic source and attach
    it to udp0

set cbr0 [new Application/Traffic/CBR]

$cbr0 set packetSize_ 500

$cbr0 set interval_ 0.005

$cbr0 attach-agent $udp0


#Create a Null agent (a traffic sink)
    and attach it to node n1

set null0 [new Agent/Null]

$ns attach-agent $n1 $null0
```

```
#Connect the traffic source with the
    traffic sink

$ns connect $udp0 $null0


#Schedule events for the CBR agent

$ns at 0.5 "$cbr0 start"

$ns at 4.5 "$cbr0 stop"
```

[Source: example1b.tcl, ns-tutorial]

II Nam output



II How do you get information on ns-2 commands?

- “Just a matter of language.”
- Ns manual
- Plenty of examples in

  `ns-2.27/tcl`
- Ns-2 tutorial by Marc Greis

  `ns-2.27/ns-tutorial`

Excerpt from ns manual:

The following is a list of simulator commands commonly used in simulation scripts:

set ns_ [new Simulator]

This command creates an instance of the simulator object.

set now [$ns_ now]

The scheduler keeps track of time in a simulation. This returns scheduler’s notion of current time.

$ns_ halt

This stops or pauses the scheduler.

$ns_ run

This starts the scheduler.

$ns_ at <time> <event>

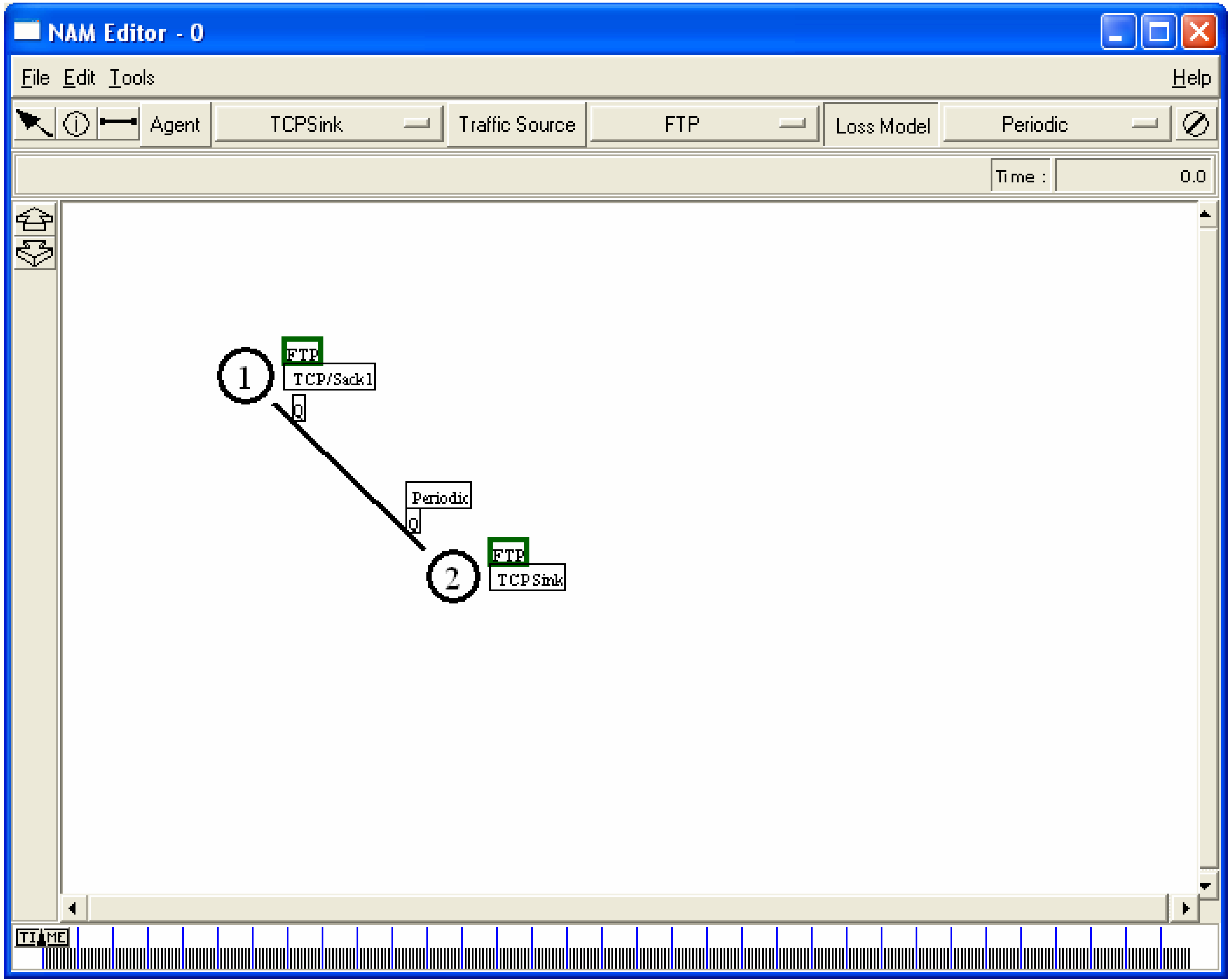
…

II Generic ns-2 script

```
set ns [new Simulator]

# [Turn on tracing]

# Create topology

# Setup packet loss, link dynamics

# Create routing agents

# Create:

# - multicast groups

# - protocol agents

# - application and/or setup traffic sources

# Post-processing procs

# Start simulation
```

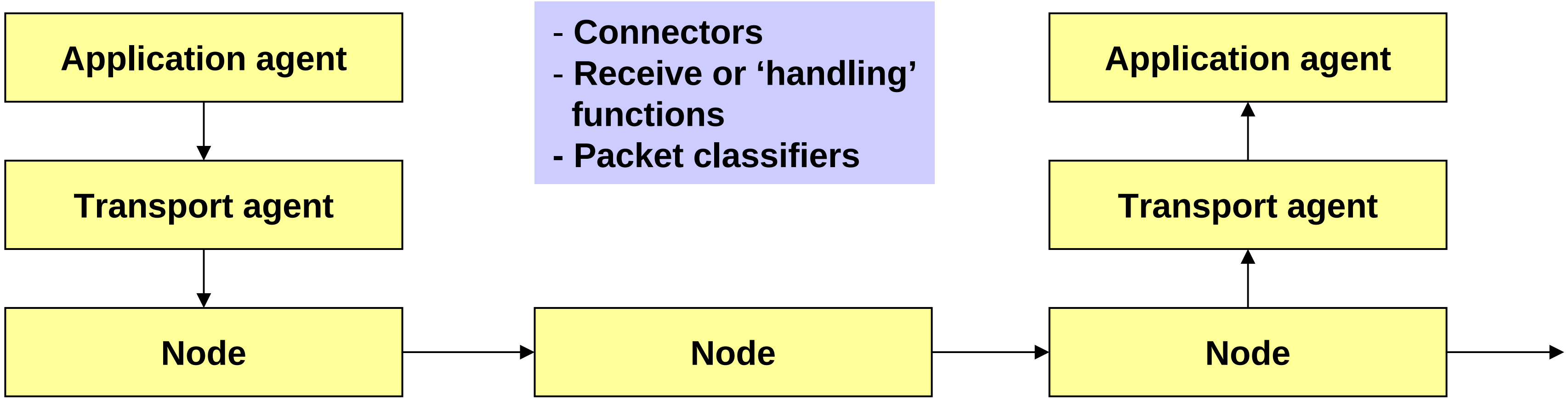
[Source: Ns-2 tutorial, P. Haldar, X. Chen, 2002]

II Nam editor for generating simple set-ups



III A first look into ns-2 internals

- We now have an basic understanding of the language used to specify a network scenario, but:
- How are all these network elements represented/coded on the C++ side of ns-2?
- What do we need as ‘primitives’?

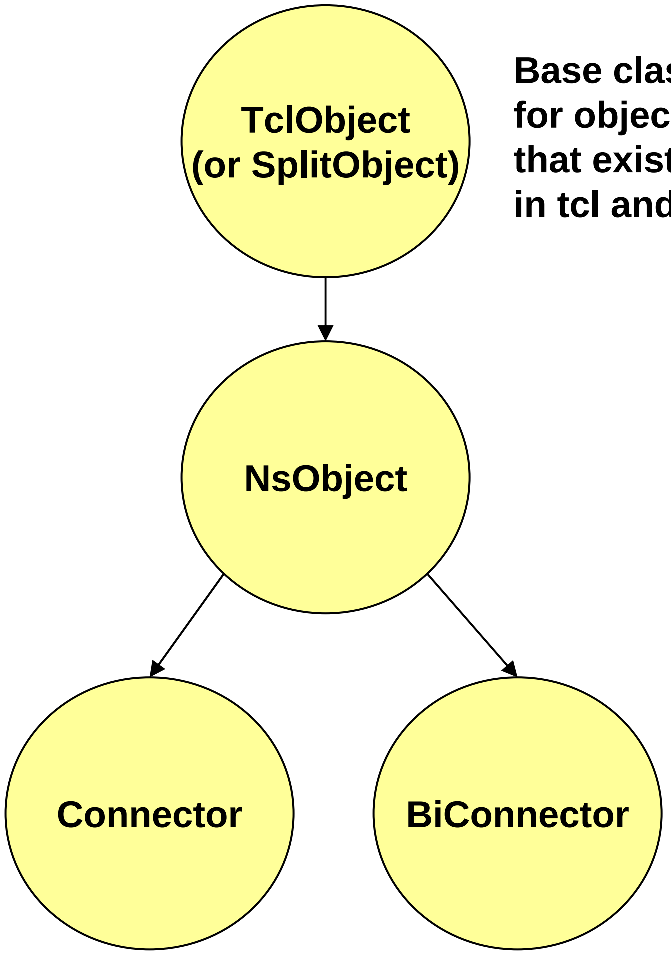


- Connectors
- Receive or ‘handling’ functions
- Packet classifiers

III Basic Ns-2 internals

- Every NsObject has `recv()` method
- Connector: has `target()` and `drop()`
- BiConnector: has `uptarget()` and `downtarget()`

Class hierarchy



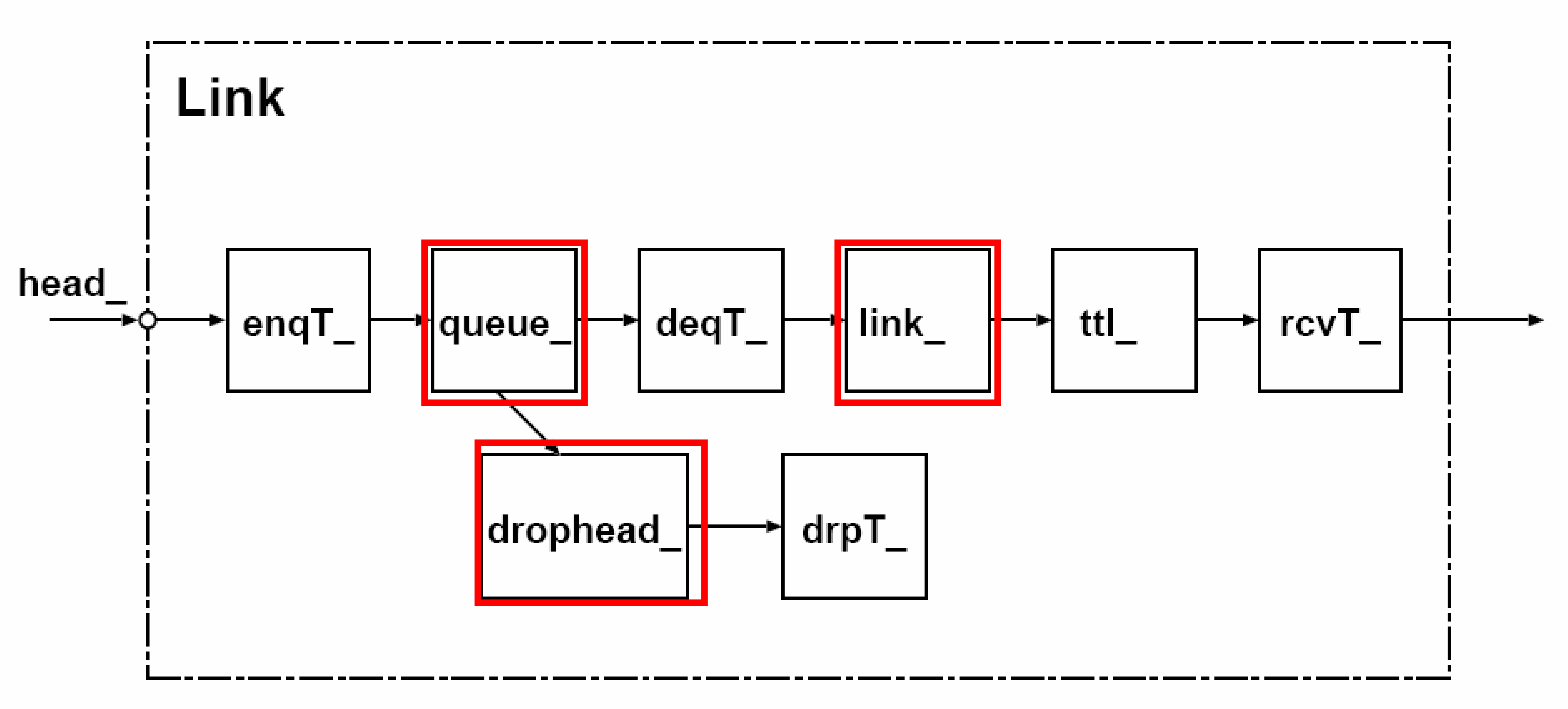
Base class
for objects
that exists
in tcl and C++

III Example: connector

```
class Connector : public NsObject {

public:

    Connector();

    inline NsObject* target() { return target_; }

    virtual void drop(Packet* p);

protected:

    virtual void drop(Packet* p, const char *s);

    int command(int argc, const char*const*
    argv);

    void recv(Packet*, Handler* callback = 0);

    inline void send(Packet* p, Handler* h) {
    target_->recv(p, h); }


    NsObject* target_;

    NsObject* drop_; // drop target for this
    connector

};
```

III Ns-2 simple links

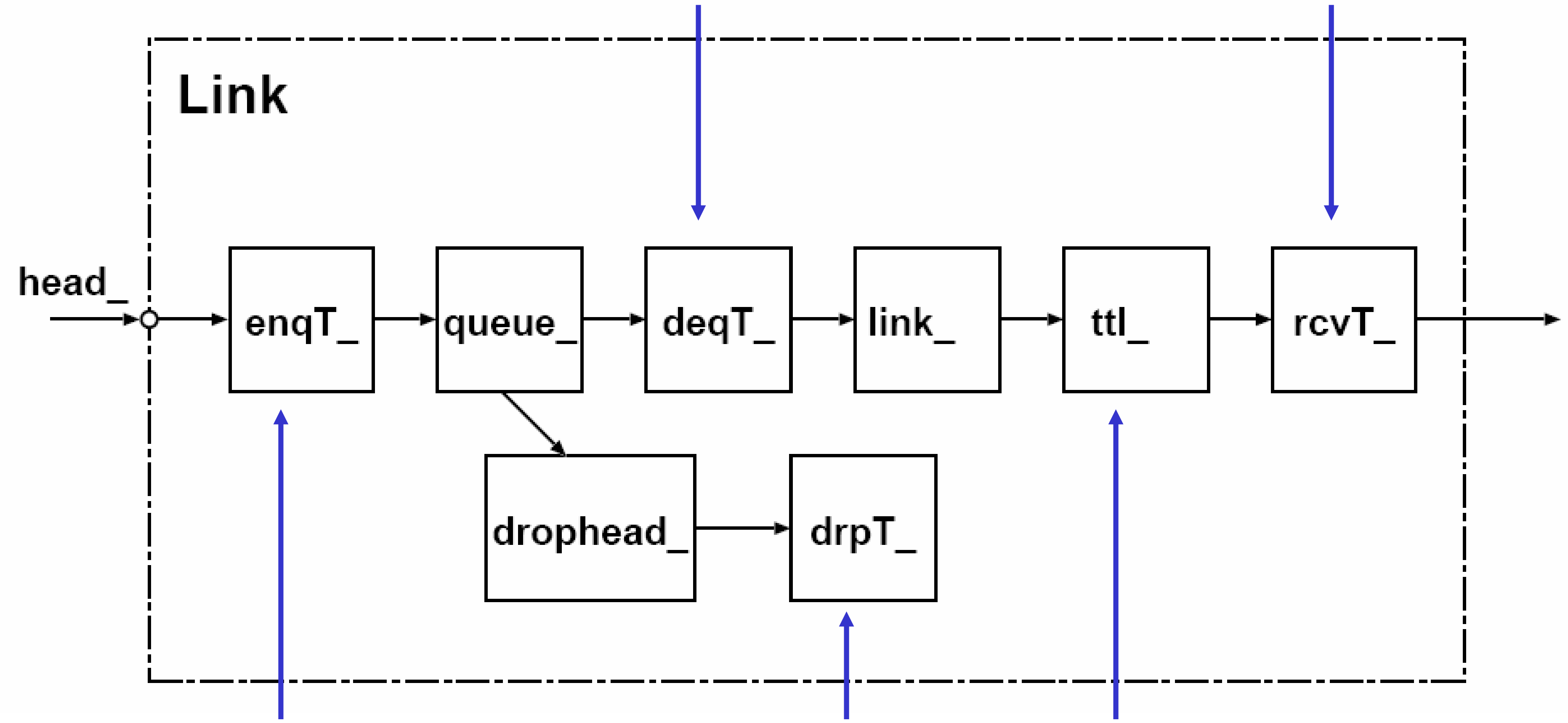
- A simple link is a sequence of connectors.



III Ns-2 link basics

Trace dequeue operation

Trace ‘receive’ by next node



Trace enqueue operation

Trace drops

TTL decrement operator

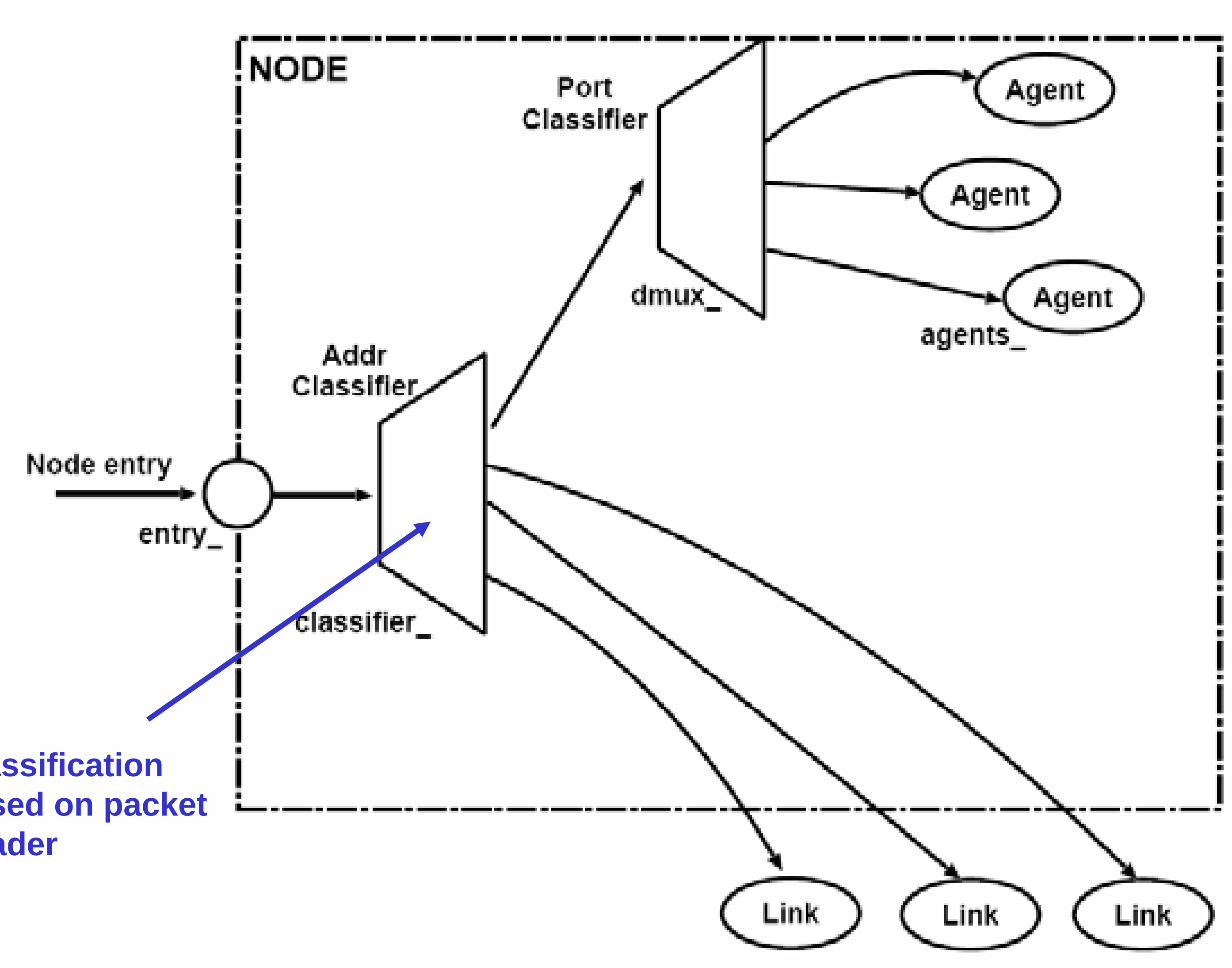
III Classifier

From ns manual:

- The function of a node when it receives a packet is to examine the packet’s fields, usually its destination address, and on occasion, its source address. It should then map the values to an outgoing interface object that is the next downstream recipient of this packet.
- In *ns*, this task is performed by a simple *classifier* object. Multiple classifier objects, each looking at a specific portion of the packet forward the packet through the node. A node in *ns* uses many different types of classifiers for different purposes.

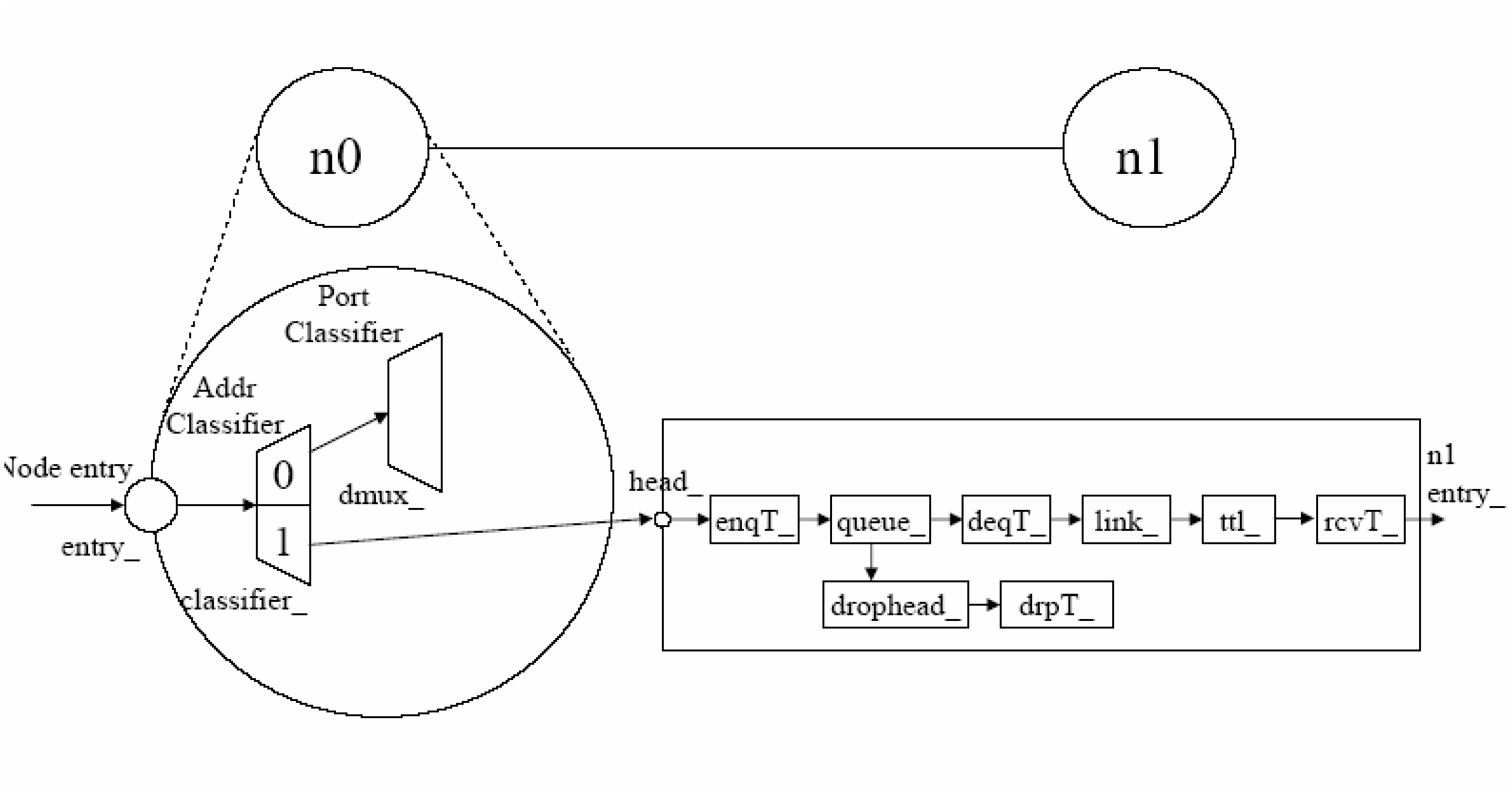
```
class Classifier : public NsObject {
public:
        ~Classifier();
        void recv(Packet*, Handler* h = 0);
protected:
        Classifier();
        void install(int slot, NsObject*);
        void clear(int slot);
        virtual int command(int argc, const
        char*const* argv);
        virtual int classify(Packet *const) = 0;
        void alloc(int);
        NsObject** slot_; /* table that maps slot
        number to a NsObject */
        int nslot_;
        int maxslot_;
};
```

III Ns-2 node basics (unicast)



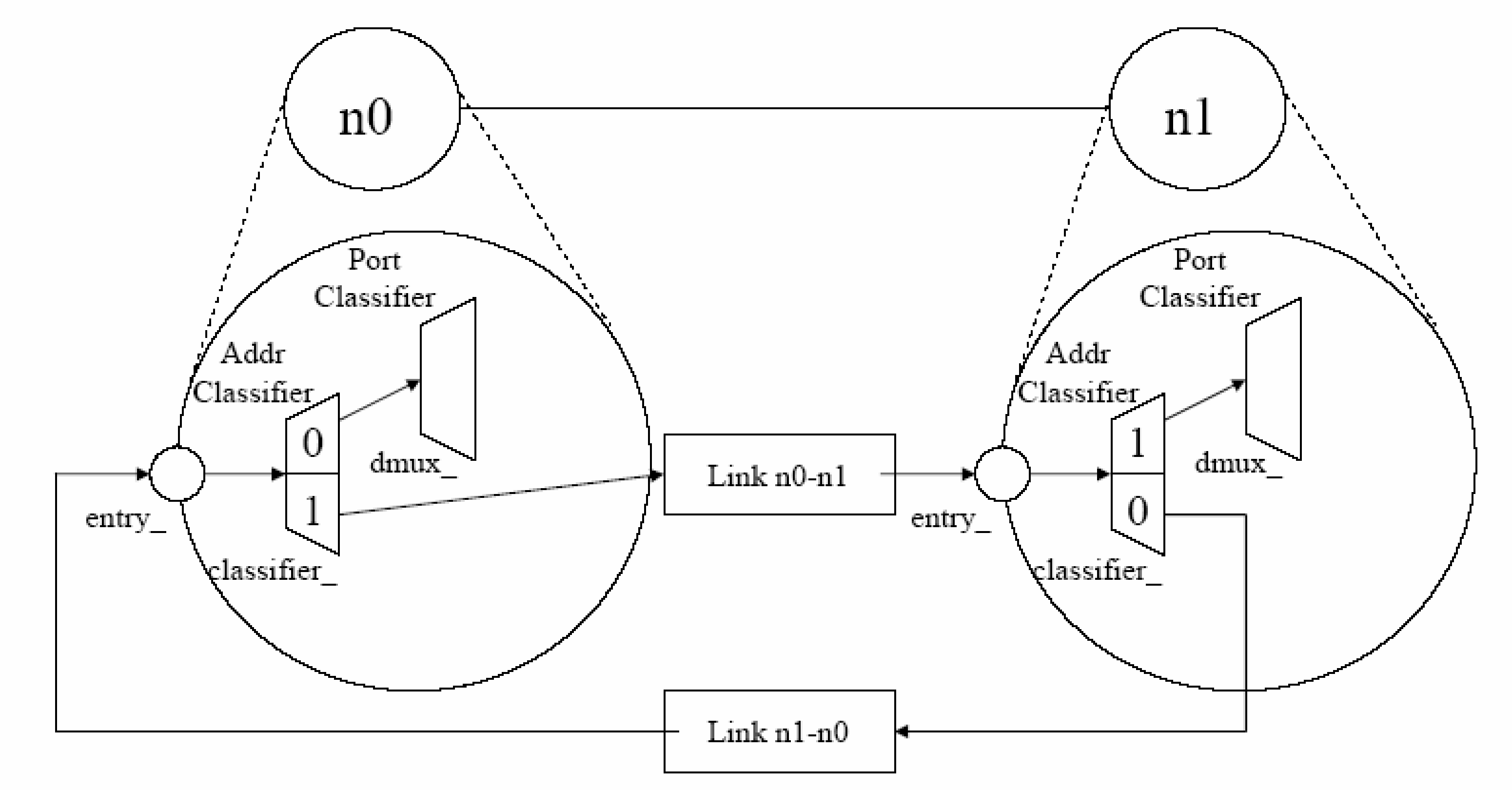
Classification based on packet header

III Our first ns-2 scripts revisited



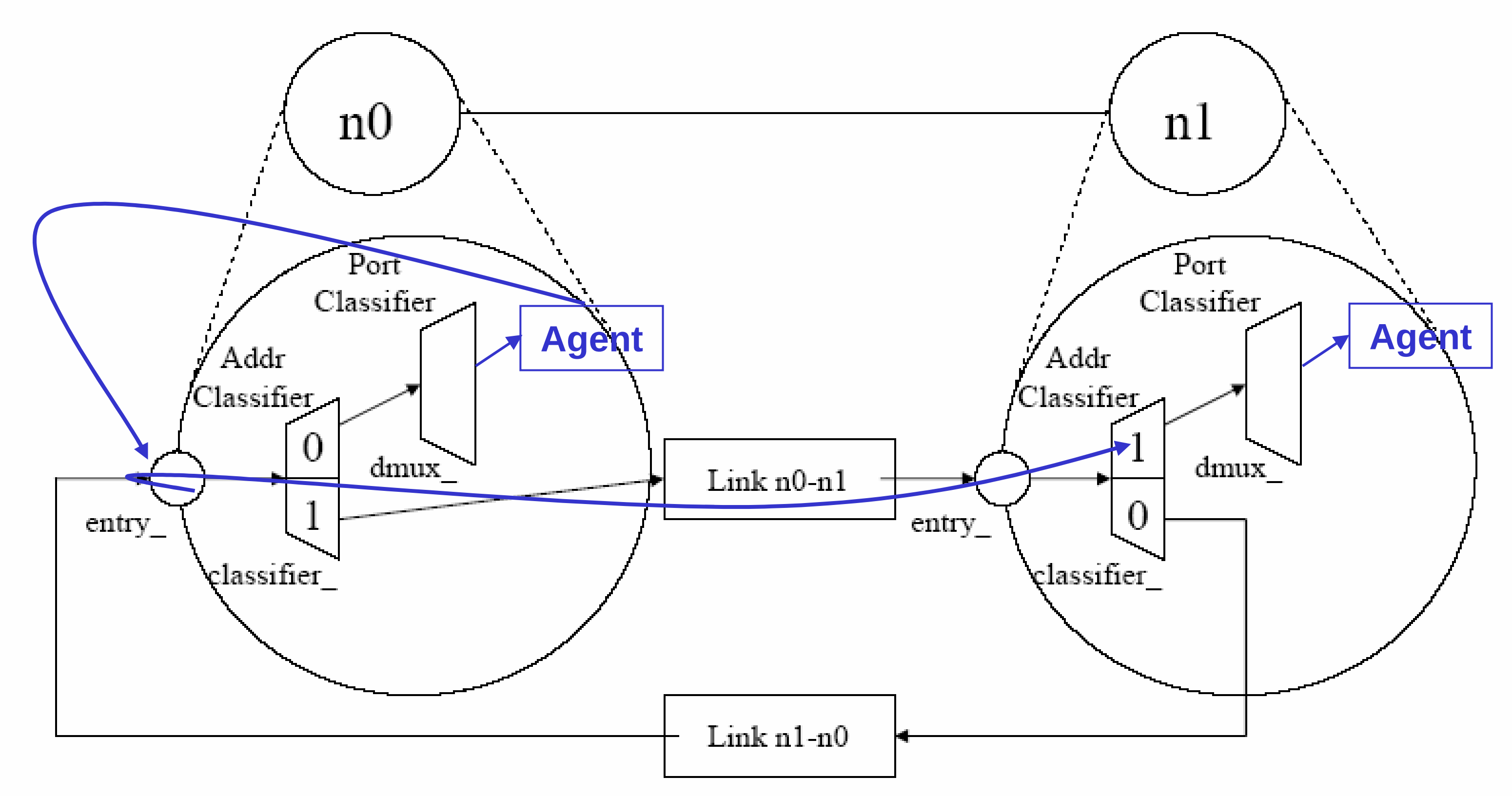
[Source: Dr. A. Kirstädter]

III Our first ns-2 scripts revisted (2)



[Source: Dr. A. Kirstädter]

III Our first ns-2 scripts revisted (3)



III Ns-2 events and packets (coarse overview)

- Events: packets and ‘at-events’
  - Class Packet is derived from class event
  - Objects in the class Packet are the fundamental unit of exchange between objects in the simulation.

Examples:

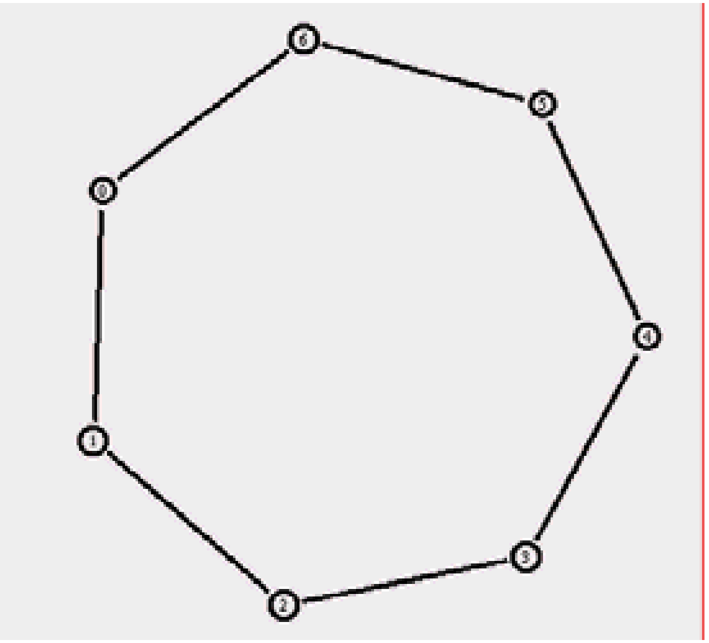
-
  ```
  void schedule(Handler*,
  Event*, double delay);
       // sched later event
  ```

-
  ```
  s.schedule(target_, p, txt +
  delay_); // from delay.cc
  ```

```
class Event {

public:

Event* next_; /* event list */

Handler* handler_; /* handler to
   call when event ready */

double time_; /* time at which
   event is ready */

int uid_; /* unique ID */

Event() : time_(0), uid_(0) {}

}; // from scheduler.cc
```

IV Another example from ns tutorial (1)

```
#Create seven nodes

for {set i 0} {$i < 7} {incr i} {

        set n($i) [$ns node]

}

#Create links between the nodes

for {set i 0} {$i < 7} {incr i} {

   $ns duplex-link $n($i)

  $n([expr ($i+1)%7]) 1Mb 10ms DropTail

}

…
```



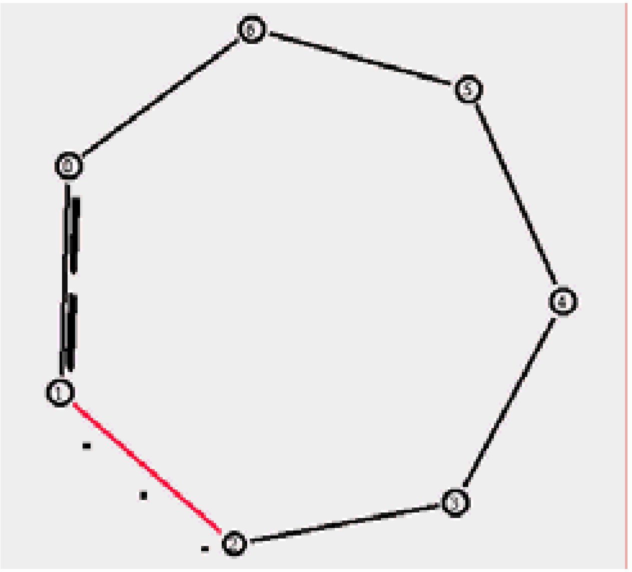
IV Another example from ns tutorial (2)

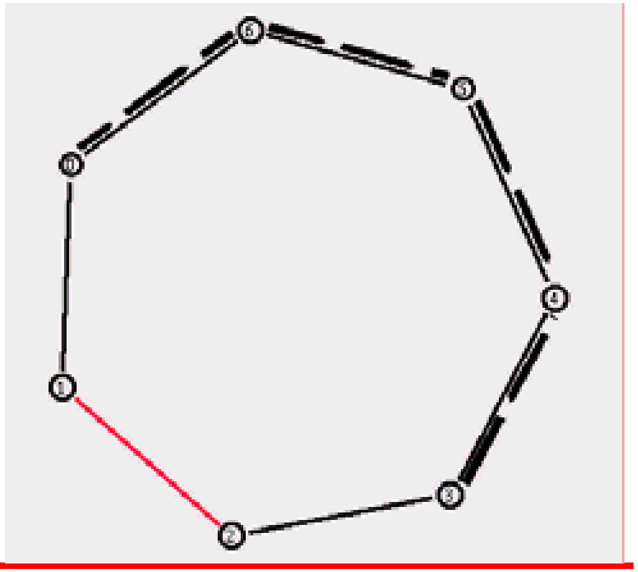
Send data from n0 to n3.

```
#Tell the simulator to use dynamic
   routing

$ns rtproto DV


$ns rtmodel-at 1.0 down $n(1) $n(2)

$ns rtmodel-at 2.0 up $n(1) $n(2)
```

To play with this example, go to

ns-2.27/ns-tutorial/example3.tcl





Wrap-up

- Introduction to ns-2
- Specify scenario via Otcl, specifiy protocol logic via C++
- Some first ns-2 scripts that show generic ns-2 script structure
- Nam: to visualize simulations
- Some internals of ns-2: connectors, recv functions, classifiers as basis for links and nodes

Discussion

- What is the better approach to network simulation: top-down or bottom-up?