Howto: Wireless network emulation using ns2 and Distributed Applications. Version 1.0 11th December 2004

Daniel Mahrenholz, Svilen Ivanov

University of Magdeburg, Germany

Contents

1 Introduction **3**
- 1.1 Copyright Information . . . 3
- 1.2 Disclaimer . . . 3
- 1.3 Credits . . . 3
- 1.4 Feedback . . . 3
- 1.5 Support . . . 3
- 1.6 Structure . . . 4

2 Quick Walk-through **5**
- 2.1 Configure the network . . . 5
- 2.2 Download . . . 6
- 2.3 Setup Hosts . . . 6
- 2.4 NS-2 Emulation . . . 7
- 2.5 Test Communication . . . 8

3 Network setup **10**
- 3.1 Addressing scheme . . . 10
- 3.2 Connectivity . . . 10
- 3.3 Host Requirements . . . 12

4 NS-2 Emulation **14**
- 4.1 Emulation Modules . . . 14
  - 4.1.1 Network Object . . . 14
  - 4.1.2 Tap Agent . . . 14
- 4.2 Real-Time Scheduler . . . 15
  - 4.2.1 Delayed Execution . . . 15
  - 4.2.2 Transmission Time Compensation . . . 15
- 4.3 Trace Extensions . . . 17
- 4.4 Sample NS-2 Emulation Script . . . 18

5 Disadvantages **19**

A NS-2 Emulation Script **22**

1 Introduction

This document describes how to distribute a set of network applications among several machines in a LAN. Next, it shows how to use the network simulator ns-2 to capture the traffic among these applications and pass it through a simulated network [1]. By "Distributed Applications" here we mean that the applications do not run on the same host as ns-2, but are spread among several machines in a LAN.

This document is based on a work from Daniel Mahrenholz and Svilen Ivanov - "Real-time network emulation with ns-2" [3].

1.1 Copyright Information

1.2 Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your system. Proceed with caution, and although this is highly unlikely, the author does not take any responsibility for that.

1.3 Credits

This work was sponsored by the German Research Foundation (DFG), grand no. NE 837/3-1 and the University of Magdeburg, Germany.

1.4 Feedback

Feedback is most certainly welcome for this document. Without your submissions and input, this document wouldn't exist. Please send your additions, comments and criticisms to the following email address:

<svilen (at) cs.uni-magdeburg.de>.

1.5 Support

If you encounter problems during the following of this document, you can post a request on our "nse – NS-2 Emulation Extension" mailing, located on the following address: http://mail-ivs.cs.uni-magdeburg.de/mailman/listinfo/nse .

1.6 Structure

The document is structured in the following way. Section 2 gives a quick command reference to run ns-2 emulation with distributed applications. In section 3 we describe the structure of the network among the applications and show how they communicate through ns-2. After that, in section 4 we describe short the principles of network emulation in ns-2 and our extensions to it. Section 5 describes the drawbacks of this method and its current implementation.

2 Quick Walk-through

This section gives a step-by-step procedure to run ns-2 emulation. It runs ns-2 on one machine in a LAN, and the applications are distributed among other machines in the same LAN. Multiple applications can be run on one machine. If you encounter problems here, please read sections 3 and 4, which explain the steps in detail.

Here we use a network of three computers. One is running the ns-2 simulator, and the other two hosts are used to run applications. We will call them *Simulator*, *Host2*, and *Host3* for convenience.

2.1 Configure the network

First assign IP addresses to every host in the network.

1. At Simulator:

```
root@simulator:~# ifconfig eth0 192.168.1.1 up
```

2. At Host 2:

```
root@host2:~# ifconfig eth0 192.168.1.2 up
```

3. At Host 3:

```
root@host3:~# ifconfig eth0 192.168.1.3 up
```

Now you should have connection between the simulator and the two hosts:

1. At Simulator:

```
root@simulator:~# ping 192.168.1.2
PING 192.168.1.2 (192.168.1.2): 56 data bytes
64 bytes from 192.168.1.2: icmp_seq=0 ttl=64 time=1.3 ms
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.2 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.2 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.2 ms
--- 192.168.1.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.2/0.4/1.3 ms
```

2. At Simulator

```
root@simulator:~# ping 192.168.1.3
PING 192.168.1.3 (192.168.1.3): 56 data bytes
64 bytes from 192.168.1.3: icmp_seq=0 ttl=64 time=0.7 ms
64 bytes from 192.168.1.3: icmp_seq=1 ttl=64 time=0.2 ms
64 bytes from 192.168.1.3: icmp_seq=2 ttl=64 time=0.2 ms
64 bytes from 192.168.1.3: icmp_seq=3 ttl=64 time=0.2 ms

--- 192.168.1.3 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.2/0.3/0.7 ms
```

2.2 Download

Download and extract our ns-2 patch and sample scripts from the web page of the project http://www-ivs.cs.uni-magdeburg.de/eukneu/forschung/projekte/nse/ . Execute the following commands on each computer:

1. Simulator:

```
user@simulator$ cd ~
user@simulator$ wget http://www-ivs.cs.uni-magdeburg.de/eukneu/forschung/ \
projekte/nse/ns2emu-dstapp.tgz
user@simulator$ tar xzf ns2emu-dstapp.tgz
```

2. Host 2:

```
root@host2:~# cd ~
root@host2:~# wget http://www-ivs.cs.uni-magdeburg.de/eukneu/forschung/ \
projekte/nse/ns2emu-dstapp.tgz
root@host2:~# tar xzf ns2emu-dstapp.tgz
```

3. Host 3:

```
root@host3:~# cd ~
root@host3:~# wget http://www-ivs.cs.uni-magdeburg.de/eukneu/forschung/ \
projekte/nse/ns2emu-dstapp.tgz
root@host3:~# tar xzf ns2emu-dstapp.tgz
```

2.3 Setup Hosts

The following commands configure the two hosts, so that each of them can run three applications. The scripts `tapcfg`, `appstart2` and `appstart3` contain paths to some commands. Please edit them and make sure that they reflect the correct paths in your system.

1. Host 2:

```
root@host2:~# cd ns2emu-dstapp/TapUDP/
root@host2:~/ns2emu-dstapp/TapUDP# make
g++ tapudp.cc -o tapudp
root@host2:~/ns2emu-dstapp/TapUDP# ./scripts/tapcfg create 3
Creating 3 TAP devices
Allocated TAP devices: tap0 .. tap2
root@host2:~/ns2emu-dstapp/TapUDP# ./scripts/appstart2
root@host2:~/ns2emu-dstapp/TapUDP# ./scripts/netdelays.pl
root@host2:~/ns2emu-dstapp/TapUDP# scp pdelays.tcl 192.168.1.1:\
 /home/user/ns2emu-dstapp
Password:
pdelays.tcl          100% |**********************************|   606
root@host2:~/ns2emu-dstapp/TapUDP#
```

2. Host 3:

```
root@host3:~# cd ns2emu-dstapp/TapUDP/
root@host3:~/ns2emu-dstapp/TapUDP# make
g++ tapudp.cc -o tapudp
root@host3:~/ns2emu-dstapp/TapUDP# ./scripts/tapcfg create 3
Creating 3 TAP devices
Allocated TAP devices: tap0 .. tap2
root@host3:~/ns2emu-dstapp/TapUDP# ./scripts/appstart3
root@host3:~/ns2emu-dstapp/TapUDP#
```

If you encounter problems here, it is very likely that your systems do not satisfy one of the host requirements. See section 3.3 for details.

2.4 NS-2 Emulation

The following commands introduce the network simulator ns-2 in the network, configured at the previous step. You will need a copy of the source code of ns-2, which you can download from the link http://www.isi.edu/nsnam/dist/ns-allinone-2.27.tar.gz . Then you will have to apply our patch and compile ns-2.

For a successful compilation you need the `zlib` library and header files [4]. If they are not present in your system, you can download them from the zlib home page `http://www.gzip.org/zlib/`.

Another library that you need to compile the emulation modules in ns-2 is the `pcap` library. If it is not present in your system, you can find it on the web page: `http://www.tcpdump.org/`.

Switch to the Simulator machine and execute the following commands:

1. Apply the patch to the ns-2 source code and rebuild it:

```
user@simulator$ cd ns-2
user@simulator:~/ns2$ patch -p1 < ~/ns2emu-dstapp/ns2emu-dstapp.diff
user@simulator:~/ns2$ ./configure
user@simulator:~/ns2$ make clean
user@simulator:~/ns2$ make
user@simulator:~/ns2$ su
root@simulator:# ln -s `pwd`/shmlog /usr/bin/shmlog
```

2. Start the simulator with the sample setup script that we provide (*root* privileges required):

   ```
   root@simulator:# cd ~user/ns2emu-dstapp
   root@simulator:# /path_to_ns-2_directory/nse ns2emu-dstapp.tcl
   ```

   Important is that the previously generated `pdelays.tcl` file is present in the current directory when you start the simulator. It contains timing properties of the communication network, that are considered in the simulation.

2.5 Test Communication

Now switch again to one of the hosts and test the connection between the applications. This time all the traffic should pass through the simulator. You can have three possible communication paths:

1. Intra-network communication

   This is communication in the same network in terms of figure 1. Here the packets flow between TAP1 in host 2 and TAP1 in host 3.

   Host 2:

   ```
   root@host2:~/ns2emu-dstapp/TapUDP# ping 10.1.0.3 -c 4
   PING 10.1.0.3 (10.1.0.3): 56 octets data
   64 octets from 10.1.0.3: icmp_seq=0 ttl=64 time=94.6 ms
   64 octets from 10.1.0.3: icmp_seq=1 ttl=64 time=10.2 ms
   64 octets from 10.1.0.3: icmp_seq=2 ttl=64 time=9.8 ms
   64 octets from 10.1.0.3: icmp_seq=3 ttl=64 time=10.3 ms

   --- 10.1.0.3 ping statistics ---
   4 packets transmitted, 4 packets received, 0% packet loss
   round-trip min/avg/max = 9.8/31.2/94.6 ms
   ```

2. Inter-network communication

   This is communication between different networks in terms of figure 1. Here the packets flow between TAP1 in host 2 and TAP3 in host 3.

   Host 2:

```
root@host2:~/ns2emu-dstapp/TapUDP# ping 10.1.3.3 -c 4
PING 10.1.3.3 (10.1.3.3): 56 octets data
64 octets from 10.1.3.3: icmp_seq=0 ttl=64 time=82.8 ms
64 octets from 10.1.3.3: icmp_seq=1 ttl=64 time=10.1 ms
64 octets from 10.1.3.3: icmp_seq=2 ttl=64 time=10.1 ms
64 octets from 10.1.3.3: icmp_seq=3 ttl=64 time=10.7 ms

--- 10.1.3.3 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 10.1/28.4/82.8 ms
```

3. Intra-host communication

   This is communication between TAP interfaces on the same host (see figure 1). Here the packets are exchanged between TAP1 and TAP2 on host 2. Because of the used addressing scheme (see section 3) the packets are not delivered locally, but pass through the simulator.

   Host 2:

```
root@host2:~/ns2emu-dstapp/TapUDP# ping 10.1.2.2 -c 4
PING 10.1.2.2 (10.1.2.2): 56 octets data
64 octets from 10.1.2.2: icmp_seq=0 ttl=64 time=93.5 ms
64 octets from 10.1.2.2: icmp_seq=1 ttl=64 time=9.8 ms
64 octets from 10.1.2.2: icmp_seq=2 ttl=64 time=9.8 ms
64 octets from 10.1.2.2: icmp_seq=3 ttl=64 time=10.3 ms

--- 10.1.2.2 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 9.8/30.8/93.5 ms
```

Congratulations! You have successfully used ns-2 to emulate a wireless network among a set of distributed applications in a local area network. If you have encountered problems through this procedure, or you want to send us a remark, do not hesitate to contact us (see section 1.5 for details).

3 Network setup

This section describes how to setup the hosts in the network so that each host can embed several applications.

The setup is based on the TAP virtual Ethernet interfaces [2]. The kernel treats them in the same way as real network interfaces. The difference to the real ones is that user-space programs act as "network cable" for these virtual interfaces. When the kernel sends an Ethernet frame through a tapXX interface, an application receives it via the /dev/tapXX device file. On the other way, when a user-space program writes an Ethernet frame to the /dev/tapXX device the kernel receives it through the corresponding tapXX interface.

In this setup we use a single host to accommodate several applications. Every application has a corresponding node in ns-2 on which it is simulated to be running. On the hosts we allocate one TAP interface per node in the ns-2 simulation. In this way, all the applications that use the IP address of the TAP interface are simulated to be running on the same node in ns-2.

On each host runs also an additional program called *tapudp* that connects the applications, running on the host with ns-2. It collects all packets that the applications send through the TAP devices and forwards them to ns-2 via UDP. In the simulator these frames are encapsulated in simulator packets and sent through a simulated network. When the packets are received in the simulated network, they are sent to the `tapudp` program on one of the remote hosts. This program delivers the packet to the destination application through the corresponding TAP interface.

3.1 Addressing scheme

This network setup uses a specific addressing scheme. Each host is assigned a *host number*, and each TAP interface within a host is assigned a *network number*. These two numbers are encoded in the IP and MAC addresses of each TAP interface. The network number is the second byte in the IP address, and the host number is the last byte. For example the IP address of host 1 in network 2 is 10.2.0.1. Figure 1 shows an example addressing with three networks and three hosts.

Each node in ns-2 represents a TAP interface (respectively an application). We use the hierarchical addressing scheme in ns-2. The address of each node is determined by the network number and host number of the corresponding TAP interface (e.g. for interface 10.2.0.1 the address of the node is 2.1).

With this fixed address mapping scheme the agents in ns-2 can determine the source and destination nodes from the source and destination addresses of the real packets.

3.2 Connectivity

In this setup we want to provide means for communication between each couple of applications (TAP interfaces). We also want to pass through ns-2 all the communication between applications.

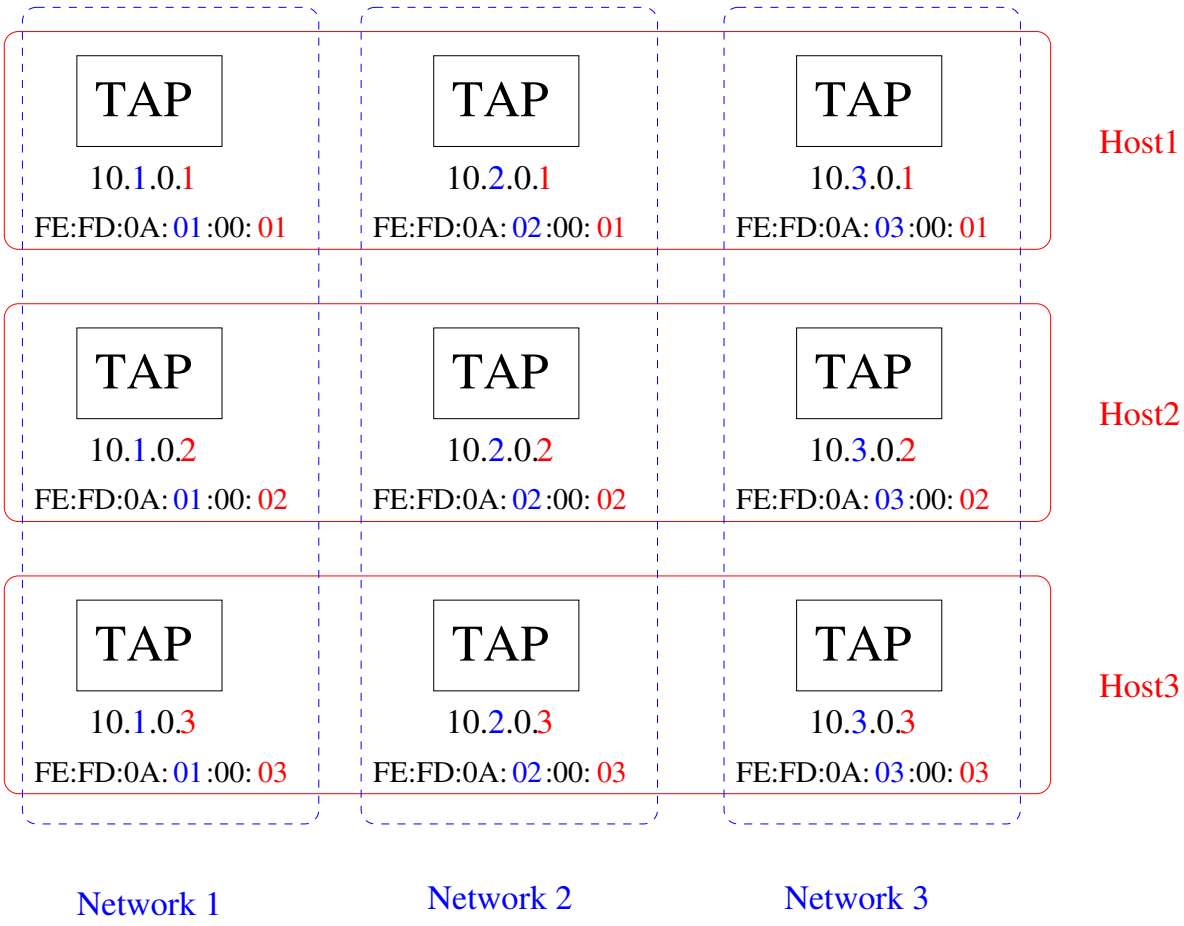

Figure 1: Addressing scheme

There are different communication types in this setup. First is communication between interfaces in the same network (*Intra-network*). Second is communication between interfaces in different networks (*Inter-network*). This includes even the case when the source and destination TAPs are on the same host (*Intra-host*). In all these cases we want to pass all the communication through ns-2.

Typically in a real network, an application sends a packet specifying only the destination address. The source address is usually determined by the operating system. In our setup we want to keep this behaviour and require from the applications to specify only the destination address when they send packets. The problem here is that on our real host we have many possible TAP interfaces through which a packet may go out. So we decide to encode the outgoing TAP interface in the destination IP address.

For example destination address 10.1.2.3 means: "send a packet through the TAP interface in network 1, to the TAP interface in network 2 of host 3".

To implement this addressing scheme we have to change the source and destination addresses of the IP frames in the simulator.

3.3 Host Requirements

This section describes the requirements to the hosts, that are used to run the applications (`host2` and `host3` in the above examples).

1. TUN/TAP driver

   The TUN/TAP (option `CONFIG_TUN`) driver should be included in the kernel. In 2.4.26 kernel you can find this option under *"Network device support -> Universal TUN/TAP device driver support"*. If your kernel does not provide you this option you can download a patch from the TUN/TAP home page `http://vtun.sourceforge.net/tun/`. The driver supports kernel series 2.2.x, 2.4.x and 2.6.x.

2. /dev/net/tun device

   If the device `/dev/net/tun` (in some systems `/dev/tun`) does not exist on your system, you have to create it in order to access the TUN/TAP driver. You can use the following commands:

   ```
   root@host2~# mkdir /dev/net
   root@host2~# mknod /dev/net/tun c 10 200
   ```

3. tunctl tool

   You need also the `tunctl` tool, which is used to configure TUN/TAP devices. If it is not present in your system, you can download it from the following link: http://www.user-mode-linux.org/cvs/tools/tunctl/ .

4. ping tool

You need the `ping` tool from the iputils package. We use the its possibility for adaptive ping (option -A) to test the network throughput before simulation. If the iputils is not repserent in your system you can obtain it from `ftp://ftp.inr.ac.ru/ip-routing/iputils-current.tar.gz`.

5. Perl Interpretator

   We use a simple perl script to measure the throughput of the network. You can obtain perl from `http://www.perl.com/`.

4 NS-2 Emulation

The ns-2 emulation feature is used to introduce the simulator into a live network. It can grab packets from a real network, pass them through a simulated network, and then inject them back into the real one. This section shows how to introduce ns-2 in the network, described in the previous section. In this example ns-2 is running on a separated machine from the applications. It receives packets from the remote hosts (that embed the applications). Then it passes them through a simulated wireless network and sends them back to the applications. Each ns-2 node in the simulated network represents a TAP interface in the real network.

4.1 Emulation Modules

The ns-2 emulation facility mainly consists of two kinds of modules: *network objects* and *tap agents*. The network objects are used to send and receive packets to and from a network. The tap agents are connected to network objects and to ns-2 nodes. They obtain network packets from the network objects and send them through the simulated network. Each tap agent can be connected to at most one network object.

4.1.1 Network Object

In this setup we use the `Network/IP/UDP` network object. It receives Ethernet frames from remote hosts through UDP and sends them, encapsulated into ns-2 packets through the simulated network.

4.1.2 Tap Agent

In this setup we allocate a single network object per host to minimise the number of UDP connections. Since multiple TAP interfaces are on the same host, multiple Tap agents should cooperatively use the same network object. So, we implemented a new Tap agent that maintains the multiple access of tap agents to a single network object. It is called `Agent/Tap/Coop` (for cooperative) and it is a descendant of `Agent/Tap`.

At the creation of `Agent/Tap/Coop` you have to specify the *ns-2* IP address of the node on which it will be situated. For example:

```
set a12 [new Agent/Tap/Coop 1.2]
```

So, the tap agent will process only packets coming from the corresponding TAP interface. In this example it is the interface in network 1 of host 2. Then you have to attach the agent to a network object and to a node. For example:

```
$a12 network $ipudp2
$ns attach-agent $node_(1.2) $a12
```

These two commands attach the tap agent `$a12` to the network object `$ipudp` and to the ns-2 node `$node_(1.2)`.

4.2 Real-Time Scheduler

The emulation facility of ns-2 should be used together with the real-time scheduler. This scheduler synchronises the simulation virtual clock with the system time and tries to dispatch events at actual moments in time. In this way it ensures that the packets, passing the simulated network, are delayed a proper amount of time. The real-time scheduler in ns-2 is used with the commands:

```
set ns [new Simulator]
$ns use-scheduler RealTime
```

4.2.1 Delayed Execution

The real-time scheduler however introduces a problem of *delayed* execution of events. This comes simply because the software implementation of the wireless model in ns-2 can hardly execute with the same speed as hardware devices. The delays accumulate during simulation and can lead to strange behaviour of the network protocols inside ns-2. One effect is for example the change of chronological order of ns-2 events, which leads to false performance of the IEEE 802.11 protocol.

We developed several techniques to increase the real-time performance of ns-2 and they are described in the paper [3]. These are mainly changes to the real-time scheduler - in synchronisation with the system time and in execution of events.

4.2.2 Transmission Time Compensation

The applications in the current setup are distributed in a network. Then the packets that they send are delivered to ns-2 not instantly, but with a certain delay from the network. So, when the ns-2 receives a packet from an application, it has already been sent in some moment in the past real-time. This delay happens also in the other direction (from the simulator to the applications). If a simulated node receives a packet in some moment in time, the corresponding application will receive it after a transmission delay. So, the transmission times in the network interfere in the end-to-end transmission time between applications in the emulated wireless network.

If we knew the exact transmission times from and to the applications, we could consider them in the simulation model and compensate these delays. We can do this by simply inserting external events in the simulator past with some *offset* from the current time. Then, the reception of the packet within the simulator will happen sooner, but the transmission times will make the end-to-end transmission time between applications the same as it is computed by the model.

However, we can hardly measure this offset in the past. First because we need synchronised clocks to measure the time from the applications to the simulator. And second, because in the other direction (simulator -> applications) it is not possible to measure the transmission time before the packet is actually transmitted. So, we have to estimate the offset.

We do a simple estimation by sending ICMP echo requests with different sizes. Then, for each packet size we compute the average round trip time and use it as an offset in the past for that packet size. This information is provided to the Cooperative TAP agents, described in section 4.1.2. When a TAP agent receives a packet, it looks into a table for the closest packet size present there and takes the corresponding offset. Then it inserts the event "sent packet" in the past of the simulator.

One can of course argue that by inserting events in the past we can produce causality errors in the model (i.e. errors caused by not-in-order execution of events). However, the system is a *virtual environment* for applications. Important is here the effect of the emulated wireless network on the applications. Therefore, *small* errors in the execution of the model can be tolerated. By small here we mean errors that do not have a noticeable effect on the properties of the wireless network.

OTcl Syntax: The Agent/Tap/Coop (section 4.1.2) has the task to provide the scheduler with the transmission time of each packet. Therefore the Tap agent has the following two commands:

1. $Agent set_packet_granularity <number>

   This command specifies the granularity of the packet size in the lookup table of the Tap agent. It means that neighbouring records in the table have difference in the packet size in <number> bytes. The smaller the granularity, the bigger is the accuracy of the estimated transmission time of the packets.

   Example usage:

   ```
   $a12 set_packet_granularity 100
   ```

2. $Agent set_packet_delay <packet_size> <delay>

   This command sets the measured delay in seconds for the transmission of packet of the given size. When the Tap agent becomes a packet from a remote application, it assigns the delay of the closest packet size in the table.

   Example usage:

   ```
   $a12 set_packet_delay 250 0.000157
   ```

   This command sets a delay of 157ms for the packets of size 250 bytes.

Note that you do not need to set packet granularity and packet delays for all the Tap agents. It is enough to set these parameters only for one agent, because these information is shared among them. We provide a perl script `netdelays.pl` which can be used to compute delays for different packet sizes and specified granularity. The script uses ICMP echo requests to measure round trip times and we assume that they have relatively small differences to the UDP round trip times. The script `netdelays.pl` generates a `pdelays.tcl` file, which you can directly include in your simulation script. If you avoid the above commands, the Tap agents will not apply any time compensation technique.

4.3 Trace Extensions

The trace subsystem in ns-2 is used to store event information in log files. The current implementation buffers the trace data in main memory and then performs disk writes at certain periods. These writes block the simulator and may cause serious delays in execution of events in real-time mode.

To improve the real-time performance of ns-2 we developed a new trace subsystem. Here the simulator process only stores trace data in main memory and another low priority process empties the buffer when the simulator has nothing to do. To use the main memory efficiently and reduce disk writes in the system during simulation we perform in-memory compression of the trace file with the `zlib` library [4]. In this case the output of the simulator is a file in `gzip` format.

In order to use this trace system, first you have to specify a standard trace file with the `trace-all` command. Then create an instance of the `BaseTrace/ShmGZ` object to redirect the output from the standard file to the new trace system. The constructor of the `BaseTrace/ShmGZ` object has the following syntax:

```
new BaseTrace/ShmGZ <File Name> <Shm Size> <Max Line> <Compressed Buffer>
```

- `<File Name>` is the output file name
- `<Shm Size>` determines the number of trace lines which can be stored in the buffer between the simulator and the logging process. This number is computed as $2^{<ShmSize>}$. For example <Shm Size> of 10 determines a buffer which can store up to 1024 trace lines.
- `<Max Line>` is the expected maximum length of a line in the ns-2 trace file.
- `<Compressed Buffer>` is the size of memory in bytes, used to buffer the compressed output file. If you use for example 104857600 (100MB), you could easily fit up to 1GB text information there, because the trace files contain many similar patterns and the `zlib` library compresses them significantly.

For example you can use this trace system by these commands:

```
set tracefd [open dummy.tr w]
$ns trace-all $tracefd
set tr0 [new BaseTrace/ShmGZ output.tr.gz 16 400 104857600]
```

Here we first create a dummy trace file and point all the trace data to it. After that we create a `BaseTrace/ShmGZ` object, which redirects the output to the compressed file `output.tr.gz`. The trace object here can hold up to 65536 lines of maximum 400 length in the buffer between the simulator and the low priority log process. The buffer for the in-memory compressed file is 100MB.

Note: The trace system uses IPC shared memory for communication between the simulator and the logger. The requested shared memory segment in the above example is 25MB and most systems will require special privileges(root) for this allocation.

4.4 Sample NS-2 Emulation Script

Appendix A contains an example ns-2 emulation script. It creates a wireless ad-hoc network with six nodes inside. These 6 nodes correspond to 6 TAP interfaces on two hosts. It uses the ns-2 extensions, described in this section. It uses a `Tap/Coop` and a node in ns-2 to represent each virtual Ethernet TAP interface.

5 Disadvantages

This section describes the known drawbacks in the current implementation, and gives points for further investigation.

1. Unreliable transport protocol

   In the current implementation we use UDP to transport the Ethernet frames from the remote hosts (applications) to the simulator, and on the other way round. UDP is fast and has a small overhead, but it is an unreliable protocol. Packets may be reordered, or even lost.

   That is why our solution has to be used only in high throughput and low error rate networks (like Fast Ethernet LANs) to reduce the probability of errors. In this case if the applications do not flood the network, the packets lost in the LAN due to UDP will be much smaller than the packets lost in the emulated wireless network.

2. Serialised Broadcasts

   In the current implementation, each host accommodates several TAP interfaces and each interface corresponds to a node in ns-2. When several nodes receive the same broadcast packet it is sent through the real network via UDP for each per TAP interface. In the model all the broadcasts are *should be received* at nearly at the same time (due to small propagation delay of the network). But the broadcasts sent to the applications are sent one by one through the UDP socket, and so they are serialised.

   A more effective solution would be to send the broadcast once and then deliver it to each TAP interface locally. Care must be taken however that some nodes might not receive packets due to transmission errors. A list of all receivers should be also sent together with the packet.

3. Execution of past events

   When we compensate the transmission delay from the network, we insert events in the simulator past. On the other hand we do not do any rollback of the simulator, because it is quite a complicated procedure (as it is described in [5]). So, we execute past events with the current state of the model. This can cause an incorrect execution of the model.

   For example, if the medium is free now, but was not free in the true moment of sending, the packet will be sent with no delay regardless of the state of the medium in the past. On the other hand, if the medium is busy now, but was free in the past the packet will be delayed with no reason i.e. it could have been sent in its true moment. We assume that these errors diminish in the average case with a large number of packets and do not affect considerably the statistical results.

The presented system is a virtual environment, i. e. it mimics a wireless network among a set of applications. Important here is to see the effect of the wireless network on the applications. So, errors in the execution of the model, that do not have a noticeable impact on of the wireless network (as it is seen by the applications), can be tolerated.

References

[1] The Network Simulator - ns-2 `http://www.isi.edu/nsnam/ns/`.

[2] Universal TUN/TAP driver `http://vtun.sf.net/tun/`.

[3] Daniel Mahrenholz, Svilen Ivanov. Real-time Network Emulation with ns-2.

[4] Zlib home site: `http://www.gzip.org/zlib/`.

[5] Fujimoto, R. M. (2000). Parallel and Distributed Simulation Systems, Wiley Interscience.

A NS-2 Emulation Script

```
# An example script for the usage of ns-2 in emulation mode
# with distributed applications:
# Uses:
# 1. Network/IP/UDP agents to receive Ethernet frames
# from remote hosts via UDP
# 2. Tap/Coop agents to cooperatively use a Network object
# 3. In-memory compression of the trace file
# Common variables
#
set val(chan) Channel/WirelessChannel ;# Channel Type
set val(prop) Propagation/TwoRayGround ;# radio-propagation model
set val(netif) Phy/WirelessPhy ;# network interface type
set val(mac) Mac/802_11 ;# MAC type
set val(ifq) Queue/DropTail/PriQueue ;# interface queue type
set val(ll) LL ;# link layer type
set val(ant) Antenna/OmniAntenna ;# antenna model
set val(ifqlen) 50 ;# max packet in ifq
set val(x) 100 ;# x range in meters
set val(y) 100 ;# y range in meters
set val(rp) DumbAgent ;# routing protocol
set val(nn) 6 ;# number of mobile nodes
set val(stime) 30.0 ;# simulation time

set ns [new Simulator]
$ns use-scheduler RealTime
set tracefd [open nsemu_dstapp.tr w]
$ns trace-all $tracefd
set tr0 [new BaseTrace/ShmGZ nsemu_dstapp.tr.gz 16 400 104857600]

set topo [new Topography]
$topo load_flatgrid $val(x) $val(y)

# Create GOD
create-god $val(nn)
# Create channel
set chan_1_ [new $val(chan)]

# Configure nodes
$ns node-config -adhocRouting $val(rp) \
-llType $val(ll) \
-macType $val(mac) \
```

```
-ifqType $val(ifq) \
-ifqLen $val(ifqlen) \
-antType $val(ant) \
-propType $val(prop) \
-phyType $val(netif) \
-topoInstance $topo \
-agentTrace ON \
-routerTrace ON \
-macTrace ON \
-movementTrace OFF \
-channel $chan_1_

$ns set-address-format hierarchical 2 24 8

proc setup_node {id x y z color} {
#Procedure to configure an ns-2 node initially
global ns node_
set node_($id) [$ns node $id]
$node_($id) set X_ $x
$node_($id) set Y_ $y
$node_($id) set Z_ $z
$node_($id) color $color
$ns at 0 "$node_($id) setdest $x $y 0"
$ns at 0 "$node_($id) color $color"
$node_($id) random-motion 0
}

setup_node 1.2 10 20 0 "red"
setup_node 1.3 10 30 0 "red"
setup_node 2.2 20 20 0 "red"
setup_node 2.3 20 30 0 "red"
setup_node 3.2 30 20 0 "red"
setup_node 3.3 30 30 0 "red"

#Network object IPUDP
set ipudp2 [new Network/IP/UDP]
$ipudp2 open readwrite
$ipudp2 bind 192.168.1.1 10002
$ipudp2 connect 192.168.1.2 10000

set ipudp3 [new Network/IP/UDP]
$ipudp3 open readwrite
$ipudp3 bind 192.168.1.1 10003
```

```
$ipudp3 connect 192.168.1.3 10000

#Tap Agent for each node
set a12 [new Agent/Tap/Coop 1.2]
set a13 [new Agent/Tap/Coop 1.3]
set a22 [new Agent/Tap/Coop 2.2]
set a23 [new Agent/Tap/Coop 2.3]
set a32 [new Agent/Tap/Coop 3.2]
set a33 [new Agent/Tap/Coop 3.3]

source "pdelays.tcl"

puts "install nets into taps..."
#Assign network objects to TAP agents
$a12 network $ipudp2
$a22 network $ipudp2
$a32 network $ipudp2

$a13 network $ipudp3
$a23 network $ipudp3
$a33 network $ipudp3

#Assign TAP agents to ns-2 nodes
$ns attach-agent $node_(1.2) $a12
$ns attach-agent $node_(1.3) $a13
$ns attach-agent $node_(2.2) $a22
$ns attach-agent $node_(2.3) $a23
$ns attach-agent $node_(3.2) $a32
$ns attach-agent $node_(3.3) $a33

$ns at $val(stime) "stop"
$ns at $val(stime) "puts \"NS EXITING ...\" ; $ns halt"

proc stop {} {
global ns tracefd ipudp2 ipudp3
$ns flush-trace
close $tracefd
$ipudp2 close
$ipudp3 close
}

puts "okey"
$ns run
```