The CMU Monarch Project’s
Wireless and Mobility
Extensions to ns

The CMU Monarch Project
http://www.monarch.cs.cmu.edu/
monarch@monarch.cs.cmu.edu

Snapshot Release 1.1.1
August 5, 1999

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

The CMU Monarch extensions to ns [4] provide new elements at the physical, link, and routing layers of the simulation environment. Using these elements, it is possible to construct detailed and accurate simulations of wireless subnets, LANs, or multi-hop ad hoc networks. This document describes each of the pieces we have added to ns, and gives some examples of how they can be used. This document assumes the reader is very familiar with using ns already, and it is not a substitute for reading and understanding the source.

More information on our extensions to ns, and description of some of our results based on the simulator, can be found in a paper published in Mobicom98 [2].

Our extensions to ns are based on the ns-2.1b1 release. We would welcome feedback, improvements, and bug fixes at monarch@monarch.cs.cmu.edu.

In addition to the extensions to ns, we also provide a visualization tool called ad-hockey that can be used to create input scenarios and view the trace playback of a simulation. For more information on ad-hockey, please see the code in ad-hockey/ and the documentation in cmu-docs/ad-hockey.ps.

This documentation is very much under construction. We apologize for the rough edges. Since this is basically a beta release, the assumption is that you, the reader, will jump in and look through the code as it interests you. This document is just to help get you started.
Contents

1 Status 4

2 Overview 4

3 Provided Components 5

3.1 Physical Layer ................................................................. 5
   3.1.1 Radio Propagation Models ........................................ 5
   3.1.2 Antennas ................................................................. 5
   3.1.3 Network Interfaces .................................................. 5

3.2 Link Layer ........................................................................... 5
   3.2.1 Media Access Control Protocols ................................... 5
   3.2.2 Miscellaneous ............................................................. 6

3.3 Network Layer ................................................................. 6
   3.3.1 Routing Protocols ....................................................... 6

3.4 Scenario Creation ............................................................. 6

4 Mobile Node Organization 6

4.1 Outgoing Packets ............................................................... 8

4.2 Incoming Packets ............................................................... 8

4.3 Adding User Level Agents to a Mobile Node ......................... 8

5 Component Details 9

5.1 Classifiers ........................................................................... 9
   5.1.1 Classifier/Addr ............................................................ 9

5.2 class LL .............................................................................. 9

5.3 class Mac ........................................................................... 10

5.4 class PriQueue ................................................................. 10

5.5 class NetIf ........................................................................... 10
   5.5.1 NetIf/SharedMedia .................................................... 10

5.6 Routing Protocols .............................................................. 10
   5.6.1 DSR .......................................................... 11
   5.6.2 How user level agents are added to DSR nodes ............. 13

5.7 Termination Handling ......................................................... 13

6 Writing Packet Sources 13

7 Changes From ns and Other Quirks 13

7.1 Addressing Conventions ..................................................... 14
### Assumptions

<table>
<thead>
<tr>
<th>Building the Simulator</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1 If You Do Not Have ns Already Installed</td>
<td>15</td>
</tr>
<tr>
<td>9.2 If You Already Have ns Installed</td>
<td>16</td>
</tr>
<tr>
<td>9.3 Packing List</td>
<td>16</td>
</tr>
</tbody>
</table>

### Using the Simulator

<table>
<thead>
<tr>
<th>Using the Simulator</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1 Generating Scenario Files</td>
<td>17</td>
</tr>
<tr>
<td>10.1.1 Movement Pattern Files</td>
<td>17</td>
</tr>
<tr>
<td>10.1.2 Communication Pattern Files</td>
<td>19</td>
</tr>
<tr>
<td>10.2 Automating Batches of Runs</td>
<td>20</td>
</tr>
</tbody>
</table>

### Output Trace Format

<table>
<thead>
<tr>
<th>Output Trace Format</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1 Trace File Header</td>
<td>21</td>
</tr>
<tr>
<td>11.2 Log File Entry Types</td>
<td>21</td>
</tr>
<tr>
<td>11.2.1 Packet Motion (s, r or D)</td>
<td>21</td>
</tr>
<tr>
<td>11.2.2 Motion (M)</td>
<td>23</td>
</tr>
<tr>
<td>11.2.3 DSR (S*)</td>
<td>23</td>
</tr>
<tr>
<td>11.2.4 TORA (T*)</td>
<td>23</td>
</tr>
</tbody>
</table>

### Using the Simulator as an Emulator

<table>
<thead>
<tr>
<th>Using the Simulator as an Emulator</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 Setting Up Emulation Channels</td>
<td>24</td>
</tr>
<tr>
<td>12.2 Directing Packets Through the Emulator</td>
<td>24</td>
</tr>
</tbody>
</table>

### Copyright and Disclaimer

| Copyright and Disclaimer | 26 |
1 Status

We are currently (10/98) in the process of helping the VINT project integrate mobility and wireless extensions into the mainline ns code. This integration should be completed sometime in second quarter of 1999.

The major changes between this release 1.1.0beta and the previous release 1.0.0beta are:

- This release contains two additional routing protocols: TORA and AODV
- This release fixes some bugs in mac.h and mac-802_*. that prevented the code from running properly on platforms that require aligned access to memory (such as SPARC-based machines).
- A new version of ad-hockey.
- A ns-src/cmu/setdest/calcdest program that can take a movement scenario generated by ad-hockey and annotate it with the “god” information that shows the optimal distance between any two nodes at any point in time.

2 Overview

Figure 1 shows a logical overview of how nodes are connected together using the CMU Monarch extensions to ns. Each mobile node is an independent entity that is responsible for computing its own position and velocity as a function of time. Each mobile node can have one or more network interfaces, each of which is attached to a channel. Channels are the conduits that carry packets between mobile nodes. When a mobile node transmits a packet onto a channel, the channel distributes a copy of the packet to all the other network interfaces on the channel. These interfaces then use a radio propagation model to determine if they are actually able to receive the packet.

Typically, all the network interfaces of the same type should be connected to one channel. A mobile node can have more than one type of network interface, and these interfaces can be connected into different channels. Conceptually, a channel represents a particular radio frequency with a particular modulation.

![Diagram of mobile nodes and channel](image)

**Figure 1** Logical overview of how mobile nodes are connected under the Monarch wireless extensions to ns.
and coding scheme. Channels are treated as orthogonal, meaning that packets sent on one channel do not interfere with the transmission or reception of packets on a another channel.

Although not show in Figure 1, it is also possible to connect nodes together with `class Link` objects as is done in base `ns`. This enables the simulation of networks with both wired and wireless parts.

3 Provided Components

The CMU Monarch extensions to `ns` provide the following additional features, organized by ISO network stack layer.

3.1 Physical Layer

3.1.1 Radio Propagation Models

**Two Ray Ground Reflection Approximation** Uses Friss free-space attenuation ($1/r^2$) at near distances and an approximation to Two Ray Ground ($1/r^4$) at far distances [12]. The approximation assumes specular reflection off a flat ground plane.

3.1.2 Antennas

**Omni-Directional** A unity gain omni-directional antenna.

3.1.3 Network Interfaces

**Shared Media** This network interface implements a shared media model where, subject to collisions and the propagation model, each node can overhear packets transmitted by the others. The default parameters are taken to approximate the Lucent WaveLAN direct-sequence spread-spectrum (DSSS) radio interface.

3.2 Link Layer

3.2.1 Media Access Control Protocols

**IEEE 802.11** The IEEE 802.11 Distributed Coordination Function (DCF) MAC protocol [5]. Uses a “RTS/CTS/Data/ACK” pattern for unicast packets and a “Data” pattern for broadcast packets. Uses both physical and virtual carrier sense.

**CSMA** A carrier-sense multiple access (CSMA) MAC protocol. The protocol is in the file `cmu/mac-802_3.cc`, but since this for use in radio networks, there is no collision detection (a radio receiver can’t hear a collision like a wired transceiver can). The CMU Monarch Project has used this MAC protocol code far less than the 802.11 protocol code and it has not been as extensively verified, so it is proportionally more likely to contain bugs of some sort. *Caveat Emptor*
3.2.2 Miscellaneous

**Address Resolution Protocol** An implementation of ARP [11] similar to the BSD implementation. Like the BSD implementation, for each destination that has an unknown hardware address there is buffering for a single packet to that destination. If an additional packet is sent to the destination before the hardware address is learned, the buffered packet is dropped.

3.3 Network Layer

3.3.1 Routing Protocols

Two example routing protocols showing how the components offered by the CMU Monarch wireless and mobility extensions can be used to simulate multi-hop wireless networks.

**Dynamic Source Routing** An implementation of the DSR protocol [1, 6].

**Destination Sequence Distance Vector** An implementation of the DSDV protocol [10].

**Temporally Ordered Routing Algorithm** An implementation of the TORA protocol [3, 7, 8]. Also includes an implementation of the Internet MANET Encapsulation Protocol (IMEP) which TORA runs on top of.

**Ad hoc On demand Distance Vector** An implementation of the AODV protocol [9].

3.4 Scenario Creation

**ns-src/cmu/setdest/setdest** Creates scenario files using the random way-point model. Annotates them with the “god” information so that at every point in time the length of the shortest path between any two nodes is known.

**ns-src/cmu/setdest/calcdest** Takes a scenario file created by ad-hockey and annotates it with the “god” information.

4 Mobile Node Organization

Figure 2 shows the basic schematic layout of a typical mobile node. This basic layout is created by the Tcl MobileNode super class defined in cmu/scripts/mobile-node.tcl.

As is currently the practice in ns, we use fields in the common header of each packet to store meta-data related to the motion of the packet through the layers of a mobile node. This meta-data is not visible to any of the code that would be part of a real implementation of a mobile node or its routing protocols, but is used only by the internals of the simulator.
Figure 2  Schematic of a mobile node under the CMU Monarch wireless extensions to ns.
4.1 Outgoing Packets

Packets sent by a source on the mobile node are handed to the mobile node’s entry point, which passes them to the address demultiplexer (Section 5.1.1). The address demux tests the destination IP address of the packet, and, if it matches the node’s own address, passes the packet up to the port demux. Since packets sent by a source on this node are typically destined to another node, most packets will match the default target of the address demux and be handed down to the routing protocol (Section 5.6).

The routing protocol is responsible for setting the \texttt{next\_hop} field in the packet’s common header to the address of the next node that should process the packet, and then passing the packet down to the link layer (LL) (Section 5.2). If the next hop address is an IP address, the LL object queries the ARP object to translate the IP address to a hardware address. If the ARP object does not already have a mapping for the IP address, it will queue the packet and instruct the LL object to send an ARP Request packet with a broadcast hardware address in order to perform address resolution. Once the hardware address of a packet’s next hop is known, the packet is inserted into the interface queue (IFq) (Section 5.4). The media access control object (MAC) takes packets from the head of the interface queue and sends them to the networking interface (NetIF) when appropriate.

The network interface (Section 5.5) stamps the common header of the packet with properties such as the power and position of the transmitting interface, and then passes the packet to the channel, where a copy is made for all the other interfaces on the channel.

4.2 Incoming Packets

A copy of each packet sent onto a channel is delivered to each of the network interfaces on the channel at the time at which the first bit of the packet would begin arriving at the interface in a physical system, based on the distance between the nodes and the speed of light. Each network interface stamps the packet with the receiving interface’s properties and then invokes the propagation model (Prop Model). The propagation model uses the transmit and receive stamps and the properties of the receiving interface to determine the power with which the interface will receive the packet. The receiving network interfaces then use their properties to determine if they actually successfully receive the packet, and hand the packet to their MAC layer if appropriate. If the MAC layer receives the packet collision- and error-free, it passes the packet to the mobile node’s entry point.

If this is the final destination of the packet, the address demux will pass the packet to the port demux, which will hand the packet to the proper sink agent. If this is not the packet’s final destination, it will go to the default target of the address demux, and the routing agent at this node will be called on to assign the packet a next hop and pass the packet back to the link layer.

4.3 Adding User Level Agents to a Mobile Node

There are several ways to attach the agents that actually send and receive packets to the MobileNodes that will carry them around and provide them with network layer services. In the CMU Monarch simulation scheme, we place the commands that create, configure, and control these agents in a communication pattern.
file. These communication pattern files can be created directly, or by ad-hockey using the Construction-Tools::SchedulePackets menu option.

As an illustrative example, the following line of Tcl code is from one of our example CBR communication pattern files.

```tcl
set cbr_(4) [\$ns_ create-connection CBR $node_(6) CBR $node_(7) 0]
```

`create-connection` is a method of the simulator object that is defined in `ns-src/tcl/lib/ns-lib.tcl`. The method first creates the CBR source and sink, and then calls the `attach` method of the nodes to associate them with the respective node. For MobileNodes, this invokes the code below in `cmu/scripts/mobile_node.tcl:MobileNode::attach` which: 1) sets the target to which the agent sends packets to be the value returned by calling the `entry` method of the MobileNode, and 2) sets the port demux to send to the agent any packets with a destination port number equalling that assigned to the agent.

```
$agent target [$self entry]
$dmux_ install $port agent
```

For normal MobileNodes, the `entry` method returns a reference to the address classifier.

5 Component Details

5.1 Classifiers

We added a default target to the classifier, rather than overriding the Tcl `no-slot` method, since invoking the Tcl interpreter during a simulation run increases run time by roughly an order of magnitude.

<table>
<thead>
<tr>
<th>Table I</th>
<th>Tcl methods added to the Classifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>file: classifier.cc</td>
<td>new Tcl methods:</td>
</tr>
<tr>
<td>defaulttarget</td>
<td>set/query the object to handle packets to an address without a specifically installed handler</td>
</tr>
</tbody>
</table>

5.1.1 Classifier/Addr

It’s a bit of a hack, but if the `shift_` field is set to 0, the classifier will demux based on the destination port number, and if the `shift_` field is set to 8, the classifier will demux based on the destination IP address. Other values for `shift_` will cause an abort.

5.2 class LL

The class LL is based on code by the Daedalus Research Group at UCB [13].
Table II  Tcl methods added to the LL

<table>
<thead>
<tr>
<th>file: cmu/ll.cc</th>
</tr>
</thead>
<tbody>
<tr>
<td>new Tcl methods:</td>
</tr>
<tr>
<td><strong>ifq</strong></td>
</tr>
<tr>
<td><strong>arptable</strong></td>
</tr>
<tr>
<td><strong>mac</strong></td>
</tr>
<tr>
<td>new Tcl variables:</td>
</tr>
<tr>
<td><strong>mindelay_</strong></td>
</tr>
</tbody>
</table>

5.3 **class Mac**

All objects implementing a MAC protocol are derived from **class** **Mac**. The Mac object will normally pass to its `recvtarget` only packets matching the `id` set for the object. Agents that subclass themselves from **class** **Tap** defined in `cmu/mac.h` can also register themselves with the Mac object using `installTap()`. If the particular MAC protocol in use permits it, the tap will be promiscuously given all packets received by the Mac layer, before address filtering is done.

5.4 **class PriQueue**

Of Tcl type `Queue/DropTail/PriQueue`, **class** **PriQueue** inserts routing protocol packets at the head of the queue, and all other packets at the back. It also supports running a filter over all the packets in the queue and removing those with a specified destination address.

5.5 **class NetIf**

**class NetIf** is the superclass from which all network interfaces must be subclassed. Currently we provide only `NetIf/SharedMedia` interfaces.

5.5.1 **NetIf/SharedMedia**

Implements a generic radio interface characterized by the property that it behaves as a shared media between nodes. What one node transmits, the other nodes can receive (subject to the propagation model, of course).

Figure 3 shows the lines from `cmu/scripts/run.tcl` that configure the SharedMedia interface to behave like the Lucent WaveLAN DSSS radio interface.

5.6 **Routing Protocols**

This section is still under construction.

As a brief heads-up, while the DSDV and AODV mobile nodes closely follow the pattern of Figure 2, the DSR mobile node incorporates the address demux and the routing agent into a single object (the DSRAgent). This is because future implementations of the DSRAgent may want to piggy-back routing information on
Table III  Tcl variables for SharedMedia interfaces

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rb_</td>
<td>raw bitrate (bit per second)</td>
</tr>
<tr>
<td>Pt_</td>
<td>power of transmission (W)</td>
</tr>
<tr>
<td>freq_</td>
<td>frequency (Hz)</td>
</tr>
<tr>
<td>L_</td>
<td>system loss factor</td>
</tr>
<tr>
<td>CSThresh_</td>
<td>carrier sense threshold (W): min power required to detect another node’s transmission</td>
</tr>
<tr>
<td>RXThresh_</td>
<td>receive threshold (W): min power required to receive a packet</td>
</tr>
<tr>
<td>CPThresh_</td>
<td>capture threshold (dbm): signal ratio required to maintain receiver capture of incoming packet in face of collision</td>
</tr>
</tbody>
</table>

5.6.1  DSR

The architecture of a DSR node is shown in Figure 4. The DSR mobile node incorporates the address demultiplexer and the routing protocol agent into a single object (the DSRAgent). This is because future implementations of the DSRAgent may want to piggy-back routing information on data packets that would not otherwise flow through the routing agent. The implication is that the DSR agent does need to know its own IP address so that packet destined to that address can be passed up the stack to the port demultiplexer.

The RTR level receive tracer (Section 11) is moved to see all packets coming through the entry point (don’t forget this when working with DSR nodes!).

Since nodes that use the DSR routing protocol have a different architecture than other mobile nodes, `cmu/dsr/dsr.tcl` defines a subclass of MobileNode called SRNode that is used to create DSR mobile

```tcl
NetIf/SharedMedia set CPThresh_ 10.0
NetIf/SharedMedia set CSThresh_ 1.559e-11
NetIf/SharedMedia set RXThresh_ 3.652e-10
NetIf/SharedMedia set Rb_ 2*1e6
NetIf/SharedMedia set Pt_ 0.2818
NetIf/SharedMedia set freq_ 914e+6
NetIf/SharedMedia set L_ 1.0
```

**Figure 3**  Class variable values that make Sharedmedia interfaces model the Lucent WaveLAN radio interface.
Figure 4  Schematic of a DSR mobile node.
nodes. That is, cmu/dsr/dsr.tcl:create-mobile-node calls set node_($id) [new SRNode] and not set node_($id) [new MobileNode].

5.6.2 How user level agents are added to DSR nodes

Because of the different node architecture, agents are attached to DSR nodes differently from how they are attached in basic MobileNodes (Section 4.3). For SRNodes, the entry method is overloaded to return a reference to the DSR object created in SRNode::init, which causes packets sent by user level agents on a DSR node to be fed directly into the routing protocol, bypassing the address classifier.

As a result, after the create-connection Tcl code has finished being interpreted during the initial set-up of the simulation, the target field of the CBR source object will be set to a C++ pointer to the DSR object (which is instantiated by class DSRAgent). When the CBR source sends a packet via target_->recv(p), this packet is then handed to the function DSRAgent::recv(), which will begin network layer processing on it.

5.7 Termination Handling

6 Writing Packet Sources

This next bit is not sanity checked yet

> In particular, I would like to be able
> to broadcast and unicast messages via appropriate DSR facilities.

The 802.11 MAC layer uses a field in the packet common header to determine where to send each packet. See packet.h

To unicast a packet, simply set the ip_dst field to the desired destination, and hand the packet to the node’s entry point (please look at how existing packet sources work).

I believe packets with ip_dst set to 255.255.255.255 are converted by the ARP code to be local broadcasts (e.g., received by all neighbors, but not repropagated.) Please note, however, that the DSR layer itself does not understand the broadcast address 255.255.255.255, so giving it a packet with ip_dst = 255.255.255.255 will result in DSR doing a Route Request for target 255.255.255.255, which, of course, none of the nodes will answer.

The flood fill/multicast scheme described in the current DSR draft is implemented in our actual testbed code, but is not in the simulator code. If you do implement it, I’d appreciate it if you send us your changes for inclusion in the next simulator release.

7 Changes From ns and Other Quirks

Changes to the base ns code are typically surrounded by #ifdef MONARCH..#endif preprocessor commands.

The majority of the CMU Monarch mobility and wireless extensions are contained in the new subdirectory called cmu/.
We have commented out several packet header types in tcl/lib/ns-packet.tcl in order to decrease the minimum memory footprint of a class Packet object. If you need these headers, simply uncomment them.

7.1 Addressing Conventions

IP addresses Under base ns, the data-type used for storing addresses, nsaddr_t, represents an IP address as a single word containing the truncated IP address in the upper 24 bits of the address and the port number in the lower 8 bits of the address. We have unpacked this into separate address and port fields everywhere we found it. Outside code using the old representation might work (using the old classifier to demux packets), but we cannot guarantee it.

MAC addresses Although the packet headers derived from the definitions in cmu/mac.h use 6 bytes to store MAC addresses, only the high 4 bytes are actually accessed or saved. This means that you can only represent \(2^{23} - 1\) MAC Addresses, but is convenient since working with 6 byte quantities is painful. The only place our chosen representation is likely to cause problems would be representing 802.3 style multicast MAC addresses. The MAC and LL modules provided in this distribution do not support 802.3 style LL multicast, and some care would be required to add such a feature.

8 Assumptions

All simulations contain assumptions to make their calculations tractable. The most important assumptions made by the CMU Monarch wireless extensions to ns are listed here:

- Nodes do not move significantly over the length of time they transmit or receive a packet. This assumption is used by the propagation model in order to compute a single signal strength with which each packet is received. It represents what is probably the single most unrealistic and hard-to-fix assumption in the wireless physical layer.

The assumption is typically true for moderate to fast bit-rate interfaces, but a node sending a 512-byte packet over a 9.6Kbps link while moving at 20m/s with respect to the receiver will have moved 10m during the packet transmission. The problem becomes more severe when the propagation model attempts to take sharp-edged obstacles or Rayleigh fades into account. In such cases, even small movements that are irrelevant to the distance-based signal attenuation metrics could have an impact on the success or failure of a packet’s reception.

The propagation model could potentially sample the signal strength several times over the period during which a packet is received and combine the measurements, but we have not looked into this further. Statistical noise sources could be used to model the effect of changes in power level during a packet reception, but this would reduce the repeatability of a simulation run.

- Node velocity is insignificant compared to the speed of light. In particular, none of the provided propagation models include Doppler effects, although they could. Additionally, when a node receives
a packet the transmitter’s location is assumed to be effectively the same as when it transmitted the packet. If this assumption couldn’t be made, then the class PacketStamp could be extended to include the transmitter’s position.

9 Building the Simulator

The CMU Monarch extensions to ns are distributed as a complete ns source tree based on ns-2.1b1 (the most current ns release as of 8/11/98 is ns-2.1b3). The source tree should build on any platform that ns will build on. It has been tested on various versions of FreeBSD from 2.2.2 on up, and on Red Hat 5.0 and 5.1 Linux distributions.

Before building the CMU Monarch ns source tree, please first familiarize yourself with the build process for base UCB/LBL/VINT ns given on their web page: http://www-mash.CS.Berkeley.EDU/ns/.

Of particular help is the installation information page:
http://www-mash.CS.Berkeley.EDU/ns/ns-build.html

9.1 If You Do Not Have ns Already Installed

We recommend the following process.

1. FTP the ns-allinone package from:
   http://www-mash.CS.Berkeley.EDU/ns/ns-build.html
   and build it using the install script it includes. The CMU Monarch ns source tree should still build okay if the optional parts of allinone do not build correctly at this point.

2. Unpack the CMU Monarch ns tarball in the ns-allinone-2.1b3 directory. ns-src/ is the CMU Monarch ns source tree, and ns-2/ is the unmodified ns-2.1b3 source tree. cmu-docs/ contains this file, and ad-hockey/ contains the ad-hockey scenario and trace file visualization tool. See the file cmu-docs/ad-hockey.ps for information on installing and using the visualizer tool.

3. cd ns-src

4. Edit the Makefile.in file to change the line
   TCLSH = ../tcl7.6/unix/tclsh
   to
   TCLSH = ../tcl8.0/unix/tclsh

5. cp ../ns-2/configure configure-2

6. sh configure-2

7. make depend; make
9.2 If You Already Have ns Installed

Unpack the CMU Monarch ns tarball parallel to your existing ns source tree. You may have to alter the included Makefile.in and configure scripts so they can find where you installed the ns support code such as otcl, tcl, tk, and tclcl. Configure and make the sources.

9.3 Packing List

When untared, the cmu-extendedns-1.0.0-beta.tgz file produces the following subdirectories:

- ns-src/ — the complete source tree for ns version 2.1b1 including the CMU Monarch wireless and mobility extensions
  - cmu/ — the majority of the CMU Monarch ns extensions
    * dsdv/ — implementation of DSDV protocol
    * dsr/ — implementation of DSR protocol
    * scripts/ — perl and Tcl scripts
    * setdest/ — program for creating scenario files
- ad-hockey/ — the scenario creation and visualization tool
- cmu-docs/ — documentation for the CMU Monarch ns code

10 Using the Simulator

You can, of course, use the pieces we’ve provided and implement your own mobile nodes and simulations from scratch. To help get you started, here are some examples of how we use our extended version of ns.

We define three main inputs to ns. The first is a movement pattern file that describes the movement that all nodes should undergo during the simulation. The second is a communication pattern file that describes the packet workload that is offered to the network layer during the simulation. Together, the communication pattern file and the movement pattern file comprise a description of the simulation scenario. For historical reasons, the movement pattern file is also called a scenario file. The third input to the simulation is a router configuration file that defines which ad hoc network routing protocol should be be used during the simulation.

These inputs are brought together by the main Tcl script that coordinates the simulation, cmu/scripts/run.tcl. To understand how we run our simulations, you should start by reading through it. It first defines default values for many simulation constants, and then processes the command line arguments to ns. Based on the arguments, it reads in the communication and movement (scenario) files, builds and configures the appropriate number of nodes, and then starts the simulation. Each of the routing protocols comes with a router configuration Tcl file that defines the procedure create-mobile-node used by run.tcl to build a mobile node that uses the routing protocol.

For example:
will run ns using the DSR protocol on the specified scenario, which is a site 1500m×300m with 50 nodes moving at a maximum speed of 20m/s with pause time 0 (meaning constant motion). The communication will be driven by 20 sources among the 50 nodes each sending 4 512-byte packets per second. The simulation will end after simulating 900 seconds (which also happens to be when the scenario file stops scripting movement). The output will be written to the trace file out.tr.

10.1 Generating Scenario Files

Both the movement and communication pattern files can contain arbitrary Tcl code, and you are free to write them by hand. However the ad-hockey program expects them to use a stylized form, and will not be able to read or write them unless they adhere to the form described below. This typically means that if you load a manually generated file into ad-hockey, it will read the parts of the file it can, and when you save the scenario it will delete from your files all the lines it could not recognize.

You are not required to use ad-hockey to work with scenario files, so you should feel free to generate them by some other means. I often write perl code to generate complicated movement patterns, but output the pattern in the format ad-hockey can read so that I can view and edit the node motions with ad-hockey later on.

10.1.1 Movement Pattern Files

File Header Each movement pattern file should have the following header on it. This header is not required, but provides information that can be read by ad-hockey and cmu/scripts/run.tcl. If the movement pattern does not contain this header, you will need to provide the information in it to run.tcl via command line arguments to ns.

#
# nodes: 14, max time: 900.00, max x: 86550.00, max y: 74190.00
# nominal range: 250.000000 link bw: 2000000.000000
# comm pattern:
# background bitmap: bay-area.gif 695 593
#

The fields are:

- **nodes** number of nodes
- **max time** time at which simulation should stop (in seconds)
- **max x, y** size of the simulation site in meters
nominal range  range of the radios in meters WARNING: changing this value in the header DOES NOT change the effective range of the radios in the simulator, since the range of the radios depends on parameters like Tx power and receiver gain that can not be easily calculated from a single range number. This value is used by ad-hockey to draw circles showing the nominal range, and by calcdest to compute the minimum number of hops between nodes.

link bw  sets the raw bit rate of the channel. WARNING: you must ensure that the MAC protocol you have selected will work properly at the bit rate you set.

comm pattern  used by ad-hockey to record the filename of the communication pattern associated with this movement pattern

background bitmap  used by ad-hockey to load in a bitmap for the background of the node display. Should be the filename of a X bitmap or GIF file, followed by the size of the image in pixels as X Y.

Movement Commands  The actual movement of the nodes is controlled by lines of the following form:

\$node_{(1)}\ set\ Z_ 0.0
\$node_{(1)}\ set\ Y_ 45665.00
\$node_{(1)}\ set\ X_ 33779.59

Establishes the initial location of node 1 to be at 33779.59m, 45665.00m, 0.0m.

\$ns\ at\ 445.5\ "$node_{(1)}\ setdest\ 510.0\ 545.0\ 1.0"
\$ns\ at\ 341.3\ "$node_{(1)}\ setdest\ 480.0\ 515.0\ 1.0"
\$ns\ at\ 331.3\ "$node_{(1)}\ setdest\ 585.0\ 470.0\ 0.0"
\$ns\ at\ 0.0\ "$node_{(1)}\ setdest\ 585.0\ 470.0\ 1.0"

Each command of the form

\$ns\ at\ time\ "$node_{(N)}\ setdest\ X\ Y\ speed"

means that the given time, node N should begin moving towards location X,Y at the given speed. As described in the ad-hockey documentation, ns is capable of processing an arbitrary string of setdest commands. When the command is delivered to the node, the node simply starts heading to the newly commanded location from whatever location it currently happens to be at. In order to support the ability to move both forwards and backwards in time, however, ad-hockey requires that the node actually reach each destination before being commanded to go to another destination. Failure to obey this constraint will make nodes appear to teleport on the ad-hockey display, although the movement pattern will be properly processed by ns itself.

The line with a speed of 0.0 shows how ad-hockey denotes a pause in motion. For the 10 seconds from 331.3 to 341.3, node 1 remains stationary at the location the setdest at time 0.0 sent it to.
Since we often use the List based Scheduler object, movement pattern files load into \textit{ns} faster if the events are in reverse order. For long and complicated movement patterns, the time savings is significant. When writing to a movement pattern file, \texttt{ad-hockey} saves the movement commands in reverse order for this reason.

**Node Attributes** The following node attributes can be manually inserted into the movement pattern file. They are comments as far as Tcl and \textit{ns} is concerned, but they effect how \texttt{ad-hockey} displays a visualization of the scenario. \texttt{ad-hockey} can read, process, and save these attributes declarations, but currently (5/30/99) has no way to insert or change them via the GUI.

```
# attribute: node 1 text: 'Team A' color: red
# attribute: node 2 text: 'Team B' color: red
# attribute: node 4 text: 'EOC-2' color: red
```

Each attribute specifies the node to which it applies. Currently supported attributes are:

- **text**: 'string': Label the node marker with string.
- **color**: color: Draw node text and marker in color from rgb.txt file.

```
# attribute: node 3 text: 'EOC-1' color: red
# attribute: node 3 text: 'EOC-1 down' after: 10.0 color: grey20
# attribute: node 3 text: 'EOC-1 coming up' after: 30.0 color: navyblue
# attribute: node 3 text: 'EOC-1' after: 50.0 color: red
```

The attributes are processed in the order they appear in the file, so specifying more than one attribute for a node provides a way to make the node change appearance during the visualization. Attribute lines with the \texttt{after:} tag are only applied after the simulation clock has passed the listed time.

**Automatically generating movement patterns** The program \texttt{cmu/setdest/setdest} can be used to generate additional movement scenarios that use the Random Waypoint algorithm described in our MobiCom'98 paper. Running the program without arguments will cause it to print out usage information.

**10.1.2 Communication Pattern Files**

Like the movement pattern file, the communication pattern file can contain arbitrary Tcl code to configure the traffic load in your simulation. Below are two stylized pieces of code that create CBR and a TCP connection, respectively. The comments are inserted by \texttt{ad-hockey}, and are actually the only part of the communication file that it knows how to process. Information about the available types of traffic agents and their parameters is available in the \textit{ns} documentation.
The file `cmu/scripts/cbrgen.tcl` is an example of how we create communication patterns. This Tcl script can be run using ns as an interpreter, and outputs peer-to-peer communication patterns as described in our MobiCom'98 paper.

```
nst cmu/scripts/cbrgen.tcl
```

### 10.2 Automating Batches of Runs

To coordinate large runs over multiple scenarios, we use the following script files:

- `run-setup` Describes a series of simulations to run in terms of input scenario files
- `cmu/scripts/run.csh -auto` Run the set of simulations described by the run-setup
- `cmu/scripts/totals.csh` Extract and analyse data from the log files created for the run-setup

### 11 Output Trace Format

The trace lines that show the motion of packets between the layers of a node and between nodes are produced by the file `ns-src/cmum/cmum-trace.cc`. As always, the source is the best documentation available. The other trace lines are used to debug and perform internal analysis of the routing protocols, and are produced by sites throughout the routing protocols.
As a quick walkthrough, (almost) all lines have the basic format

\(<\text{type}> \ <\text{time}> \ [\_<\text{node}>\_] \ \text{stuff}\)

Places where the trace output is not of this form are gradually being converted to be this form. Please keep this basic form if you modify the trace routines, since the form makes it easy to take slices through a trace file using egrep. For example, to see all the activity at node 4, we can do `egrep ‘\_4\_’ out.tr`.

### 11.1 Trace File Header

An initial header is placed on the trace file by `cmu/scripts/run.tcl`:

```
M 0.0 nn 50 x 1500 y 300 rp cmu/dsr/dsr.tcl
M 0.0 sc scen/scen-1500x300-50-0-20-1 cp scen/cbr-50-20-4-64 seed 0.0
M 0.0 prop Propagation/TwoRayGround ant Antenna/OmniAntenna
```

### 11.2 Log File Entry Types

The following subsections are divided into the types of trace file entries. The type of an entry is denoted by the first token on the trace file line.

#### 11.2.1 Packet Motion (s, r or D)

To track the motion of packets through the system, we insert tracers at various points in the mobile node’s network stack as shown in Figure 5. The tracers are inserted by `cmu/scripts/mobile_node.tcl`, though at least the DSR configuration file `cmu/dsr/dsr.tcl` moves them around as required by the particular routing protocol. Which tracers are actually inserted on a particular run is controlled by `run.tcl`.

- \(s\) = packet sent
- \(r\) = packet received
- \(D\) = packet dropped

```
s 12.7 _18_ AGT --- 0 cbr 64 [0 0 0 0 0] -------- [18:0 19:0 32 0] [0] 0 1
```

\(<\text{level}> \ <\text{drop reason}> \ <\text{uid}> \ <\text{pkt type}> \ <\text{len}> \ [\text{mac hdr}] \ ——– \ [\text{ip addrs}] \ <\text{stuff}>\)

- \(<\text{level}>\) is one of AGT,RTR,MAC which refer to the level at which the tracer is. To really understand what’s going on, you’ll need to understand what it is the tracers measure and what packets are flowing on the paths they instrument. You’ve got to read the source for this. It’s the only way to validate your experiments. Figure 5 shows the locations we inserted the tracers.
- \(\text{AGT} = \) agent level (packet going between higher layers and layer 3)
Figure 5  Schematic of a mobile node showing the location of the packet tracers inserted by mobilenode.tcl.
• RTR = packets entering or leaving layer 3
• MAC = packets entering or leaving layer 2
• <uid> = unique packet id
• <pkt type> =
  – message: DSDV routing packets
  – cbr, tcp: data packets
  – DSR, TORA, IMEP, AODV: routing protocol packets
• <stuff> = extra information that is specific to the particular packet type

11.2.2 Motion (M)

M = movement of nodes (logged when a node changes direction)
M 0.00000 50 (117.78, 294.63, 0.00), (1135.88, 261.16), 3.19
(current X Y Z), (destination X Y), speed

11.2.3 DSR (S*)

S* = DSR information of some type

• Sconfig = basic node configuration. In order to experiment with DSR we added lots of variables that
control the features of the protocol. For example,

  Sconfig 0.00000 tap: on snoop: rts? on errs? on

• SRC = route cache performance analysis messages
• SRR = DSR route discovery
• SF = packet forwarded according to source route

11.2.4 TORA (T*)

T* = tora information

12 Using the Simulator as an Emulator

This is new work, and strictly “Use the Source, Luke” We are still in the process of validating the emulator,
and you yourself will need to validate any emulation runs you perform. We have some slides on our research
papers web page that might help some.

1. Build the target “nse”
2. See files in ns-src/cmu/emulate/*.pl for hints on how we’ve been running things.

3. You can use ad-hockey to visualize the current state of the scenario by starting up ad-hockey with the correct scenario loaded. Setting it to slave to remote ns server, and hitting start. When you then start nse and provide it with the ip address of the machine running the ad-hockey\(^1\), it will send ad-hockey periodic time packets which ad-hockey will use to sync to.

### 12.1 Setting Up Emulation Channels

If added to the communication pattern file, the following code will arrange for any packet intercepted by the emulator on interface xl0 from 1.1.1.1 to 1.1.1.2 or 1.1.1.2 to 1.1.1.1 to be subject to network emulation. The physical packet is injected into the simulation as a packet event given to the network layer of the simulated node representing the physical IP source, and the packet is not released back onto the wire until, in simulation, it reaches the simulation node representing the physical IP destination. Simulation node 1 is the stand-in (aka dopelganger) for physical machine 1.1.1.1, and simulation node 3 is the stand-in for machine 1.1.2.

```
source emulate/ns-emulate.tcl
AddEmulationChannel 1 "1.1.1.1" 3 "1.1.1.2" "xl0"
```

### 12.2 Directing Packets Through the Emulator

We’ve been forcing packets through the emulator by creating host routes on the physical machines that list the emulator as the gateway. If 1.1.1.1 and 1.1.1.2 are the physical machines, and 1.1.1.3 the emulator, we would

```
route add 1.1.1.2 1.1.1.3
```
on 1.1.1.1, and

```
route add 1.1.1.1 1.1.1.3
```
on 1.1.2.

You will need to make sure your emulation box is not sending ICMP redirects. If you set IP FORWARDING off, that’s often enough (on *BSD boxes, see `sysctl -a | grep ip`). Be paranoid. Use tcpdump to make sure packets are going where you expect and that the emulator isn’t flipping out and generating trash.

### References


\(^1\)see the command line options in run.tcl


A Copyright and Disclaimer

All new files or revisions to existing files, including source code and documentation, included in the CMU Monarch Project’s wireless and mobility extensions to \textit{ns} are covered by the notice included below, whether or not the notice appears in the text of the file.

If you will be using one of the files and/or any part of any file in another system, it is your responsibility to ensure this notice appears both in the file and/or its parts and in the documentation for your system.

All Rights Reserved.

Permission to use, copy, modify, and distribute this software and its documentation is hereby granted (including for commercial or for-profit use), provided that both the copyright notice and this permission notice appear in all copies of the software, derivative works, or modified versions, and any portions thereof, and that both notices appear in supporting documentation, and that credit is given to Carnegie Mellon University in all publications reporting on direct or indirect use of this code or its derivatives.

THESE EXTENSIONS TO NS ARE EXPERIMENTAL AND ARE KNOWN TO HAVE BUGS, SOME OF WHICH MAY HAVE SERIOUS CONSEQUENCES. CARNEGIE MELLON PROVIDES THIS SOFTWARE IN ITS “AS IS” CONDITION, AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Carnegie Mellon encourages (but does not require) users of this software to return any improvements or extensions that they make, and to grant Carnegie Mellon the rights to redistribute these changes without encumbrance.