

The MIT uAMPS ns Code Extensions

Version 1.0

*Massachusetts Institute of Technology
Cambridge, MA 02139
uamps@mit.edu*

August 7, 2000

The MIT uAMPS extensions to ns [5] add support for large-scale wireless sensor networks. These extensions include models for node energy dissipation and node state, as well as several routing protocols. This document describes the uAMPS additions to ns and gives details on how to use the code and run the simulator. This document assumes the reader is familiar with the operation of ns.

1 Overview of ns Mobile Node

The work described here is based on the ns-2.1b5 release, which includes the CMU Wireless and Mobility platform. The CMU additions to the baseline ns simulator include mobile nodes, MAC protocols, and channel propagation models [1]. Figure 1 shows the implementation of a mobile node. The Application creates “data packets” that are sent to the Agent¹. The Agent performs the transport- and network-layer functions of the protocol stack. The Agent sends packets of data to CMUTrace, which writes statistics about the packets to trace files. The packets are then sent to a Connector, which passes them to the Link-Layer for data-link processing. After a small delay, the packets are sent from the Link-Layer to the Queue, where they are queued if there are packets ahead that have not yet been transmitted. Once a packet is removed from the Queue, it is sent to the MAC, where media access protocols are run. Finally, the packet is sent to the Network Interface, where the correct transmit power is added to the packet and it is sent through the Channel. The Channel sends a copy of the packet to each node connected to the channel. The packets are received by each node’s Network Interface and then passed up through the MAC, Link-Layer,

¹In ns, data are not actually transferred to the nodes in the network. Rather, virtual data, in the form of a packet of a given length, is transmitted. Therefore, packets have a certain size but do not actually contain any data.

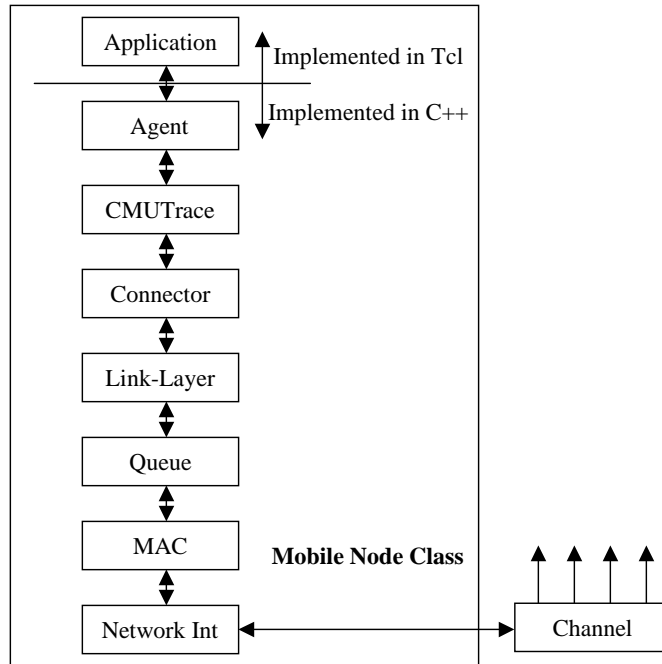


Figure 1: Block diagram of an ns Mobile Node.

Connector, CMUTrace, and Agent functions. The Agent de-packetizes the data and sends notification of packet arrival to the Application.

2 Resource-Adaptive Node

We added a Resource-Adaptive Node [4] to ns, as shown in Figure 2. The new features of a Resource-Adaptive Node are the Resources and the Resource Manager. The Resource Manager provides a common interface between the application and the individual resources. The Resources can be anything that needs to be monitored, such as energy and node neighbors. The Application updates the status of the node's resources through the Resource Manager using the functions:

- *add*: add more of a resource to the node's supply.
- *remove*: remove some of a resource from the node's supply.
- *query*: find out what amount of the resource the node currently has.

For example, the Energy Resource is used to keep track of a node's energy. The initial energy level is set at the beginning of the simulation, and throughout the simulation, energy is removed by the Application as the node performs

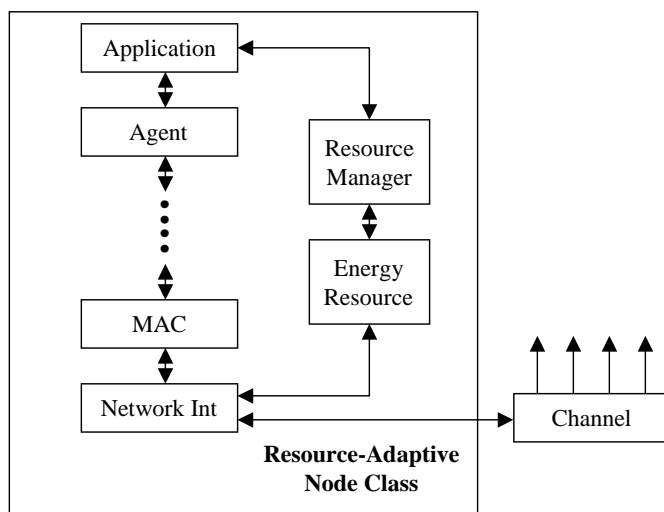


Figure 2: Block diagram of a Resource-Adaptive Node.

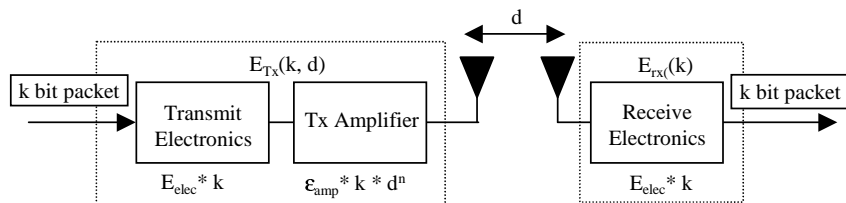


Figure 3: Radio energy dissipation model.

computation of data and by the Network Interface during packet transmission and reception. The Application (and any lower-layer functions) can also find out how much energy remains at any given time. This is useful for implementing *resource-adaptive* protocols, that change their behavior based on the current level of the resource.

We implemented a simple radio energy dissipation model, as shown in Figure 3. In this model, the transmitter dissipates energy to run the radio electronics and the power amplifier, and the receiver dissipates energy to run the radio electronics. See Chapter 4 of [2] for more details on this model. The amount of energy removed for computation or communication is defined by the user at the beginning of the simulation.

3 Network Interface

The Network Interface performs the physical-layer functions. When it receives a packet from the MAC-layer, it sets the transmit power based on an approximation of the distance to the receiver (assuming power control is used, as in all the protocols we implemented), removes the appropriate amount of energy to send the packet, and sends the packet to the Channel. If the node has used up all its energy after transmitting the packet, the node is dead and will be removed from the channel. Nodes that have died do not have any impact on the routing protocols, and any data sent to a node that is dead is thrown away.

When receiving data, the packet enters the node's Network Interface from the Channel. If the node is in the sleep state, the Network Interface discards the packet, since sleeping nodes cannot receive or transmit any packets. Therefore, there is no energy cost to these nodes even when packets are being transmitted in their vicinity. However, if data are being sent to a sleeping node, they will be lost since the node has no way of knowing that it missed a packet. Therefore, the routing protocols must ensure that a transmitting node only sends data to a receiving node when the receiving node is awake to guarantee delivery of the data.

If the node is awake, the Network Interface determines the received power of the packet. If the received power is below a detection threshold ($P_{r-detect}$), the packet is thrown away, as the node would not have been able to detect that a packet was transmitted. If the received power is above the detection threshold but below a successful reception threshold ($P_{r-thresh}$), the packet is marked as erroneous and passed up the stack. It is not thrown away because reception of this packet affects the ability to successfully receive other packets at the same time. Finally, if the received power is above $P_{r-thresh}$, the packet has been received successfully and is passed up the stack to the MAC class.

The code that implements the functions described in this section is found in **wireless-phy.cc**.

4 MAC Protocol

We created a new MAC protocol type, called MacSensor. This protocol is a combination of carrier-sense multiple access (CSMA), time-division multiple access (TDMA), and a simple model of direct-sequence spread spectrum (DS-SS). TDMA is implemented within the Application by only having the Application send data to the Agent during the specified TDMA time-slot². CSMA is implemented in the MacSensor class, and DS-SS is implemented jointly within the application and the MacSensor class.

²In this implementation of TDMA, it is assumed that the clocks of all the nodes are synchronized. However, it would be more accurate if the clock drift was modeled and explicit synchronization was performed. This is an area for future work.

Non-persistent CSMA is implemented in MacSensor. To perform CSMA, the node senses the channel before transmission. If the channel is currently being used by someone else, the node sets a back-off timer to expire after a random amount of time, where the timer is chosen uniformly with a maximum time equal to the transmit time of the packet it is waiting to transmit. This back-off policy for CSMA is effective for our protocols because all nodes are transmitting packets with the same length during a given time. Therefore, the maximum amount of time that the channel will be busy is equal to the amount of time it would take to transmit the node's packet. Once the back-off timer expires, the node again senses the channel. If it is still busy (presumably someone else captured the channel first), the node again sets a back-off timer. This continues until the node senses a free channel. Once the channel is free, the state of the node is set to indicate that the node is currently transmitting data and the node passes the packet to the Network Interface. The node must also set a transmit-timer so it knows when it has finished transmitting the packet and can reset its state to idle. It is important that the state of the node is set accurately because a node cannot transmit two packets at the same time, and a node cannot receive a packet while it is transmitting.

The Application is responsible for determining the pseudo-random noise sequence to use for DS-SS and performing data-spreading (modeled as an increase in data size). Since data are not actually transmitted in ns, the chosen spreading sequence is just listed in the header of the packet.

When MacSensor receives a packet from the Network Interface, it first determines whether the packet was sent using the spreading code which the node is currently receiving (listed in the packet header). This models the correlator stage of a spread-spectrum system, where packets sent using the node's pseudo-random noise sequence will correlate, whereas packets sent using some other pseudo-random noise sequence will not correlate and hence be dropped. However, the packet still adds to the noise floor. If too many packets with different codes are being transmitted, all the packets will be received in error, since the noise floor will be too high to receive the packets sent with the correct code. Therefore, the node keeps track of which codes are being transmitted in its vicinity at all times. There is a maximum number of simultaneous transmissions that can occur using DS-SS, based on the amount of spreading. As long as there are fewer than this number of transmissions heard at the receiving node, it is assumed that the reception of the packet is successful.

A node cannot receive a packet while transmitting. If the node is transmitting as a packet arrives, the received packet is marked as erroneous. However, it is not dropped because the receiver may be busy receiving this erroneous packet after the transmitter has finished. All packets that contain errors are dropped in the link-layer of the stack.

If the node is currently receiving another packet (P_{current}) when a new packet (P_{new}) arrives, two situations can occur. First, P_{current} might be sent with enough power to swamp out the reception of the new packet, P_{new} . In

this case, capture occurs and P_{new} is dropped. On the other hand, if $P_{current}$ does not have enough power to swamp out P_{new} , both packets collide. In this case, the packet that will last the longest for reception is kept and marked as erroneous and the other packet is dropped. By keeping the packet that will last the longest, the receiver is busy for the maximum amount of time. Again, the packet that is kept will be dropped in the link-layer of the stack.

If the node is neither transmitting nor receiving when the new packet arrives, the node's state is set to indicate that the node is currently receiving a packet and a timer is set that expires after the length of time required to receive the packet. When the timer expires, the node checks the address field of the packet. If the address is the node's address (or the broadcast address), the packet is sent up the stack to the Queue. Otherwise, the packet is not intended for this node and is dropped.

The code that implements the functions described in this section is found in `mac-sensor.cc` and `mac-sensor-timer.cc`³.

5 LEACH Protocols

LEACH [3] is implemented exactly as described in Chapter 3 of [2]. Since LEACH is an application-specific protocol architecture, it is implemented as a subclass of `ns`'s Application class. The code that implements these functions is in `ns-leach.tcl`. LEACH-C is also implemented as described in Chapter 3 of [2]. LEACH-C uses many of the same functions as LEACH (only the set-up phase differs), so it is implemented as a sub-class of LEACH. The code that implements LEACH-C is found in `ns-leach-c.tcl`.

6 Base Station Application

The base station node has no energy constraints and is the node to which all data are eventually sent. Therefore, the base station node must keep track of all the data that it receives, as determining when the base station receives the data provides an estimate the latency of different protocols, and determining how much data is received during a given time provides quality information about the different protocols. To perform these functions, we created a `BSApp` Application.

For most of the protocols, this is the only function the base station serves. However, for LEACH-C, the base station must also receive small information packets from each node at the beginning of each round that contain the node's location and current energy level. Once the base station receives this data from all nodes, it must determine optimal clusters. As discussed in Chapter 3 of [2], the base station performs simulated annealing to determine these clusters. Since

³This code borrows heavily from the implementation of the `Mac802_11Class` from `ns`.

this is a computationally intense algorithm, the simulated annealing algorithm was implemented in C++, using an Agent called BSAgent. BSAgent determines the optimal clusters and sends this information up the stack to BSApplication. The base station node then broadcasts this information to the nodes in the network. These functions are implemented in the files **ns-bsapp.tcl** and **bsagent.cc**.

7 MTE Routing

For MTE routing, routes from each node to the base station were chosen such that each node's next-hop neighbor is the closest node that is in the direction of the base station. Each node requires a certain amount of energy to determine their next-hop neighbor. When a node dies, all of that node's upstream neighbors (i.e., all the nodes that send their data to this node) begin transmitting their data to the node's next-hop neighbor.

Nodes adjust their transmit power to the minimum required to reach their next-hop neighbor. This reduces interference with other transmissions and reduces the nodes' energy dissipation. Communication with the next-hop neighbor occurs using a CSMA MAC protocol, and when collisions occur, the data are dropped. When a node receives data from one of its upstream neighbors, it forwards the data to its next-hop neighbor. This continues until the data reach the base station. These functions are implemented in **ns-mte.tcl**.

8 Static-Clustering

The static clustering protocol is identical to LEACH except the clusters are chosen a-priori and fixed. The clusters are formed using the simulated annealing algorithm as in LEACH-C. Static clustering includes scheduled data transmissions from the cluster members to the cluster-head and data aggregation at the cluster-head. Static-clustering is implemented in **ns-stat-cluster.tcl**.

9 Statistics Collection

We added statistics collection to keep track of the internal state of the network and the individual sensors during the simulation. At periodic intervals, the following data are collected:

1. Amount of energy consumed by each node.
2. Amount of data received at the base station from each node.
3. Number of nodes still alive.

Using these statistics, we can evaluate the effectiveness of the different communication protocols.

10 The Code

We added `#ifdef uAMPS ... #endif` wrappers around all code we added to existing `ns` files, including: `app.[cc,h]`, `channel.cc`, `cmu-trace.[cc,h]`, `mac.cc`, `packet.[cc,h]`, `phy.[cc,h]`, and `wireless-phy.[cc,h]`. We also added the files `mac-sensor.[cc,h]` and `mac-sensor-timers.[cc,h]`.

The files to implement Resource-Adaptive nodes, agents, and link-layer functions are in the directory `mit/rca` and include: `ns-ranode.tcl`, `rcagent.[cc,h]`, `rca-ll.[cc,h]`, `resource.[cc,h]`, `energy.[cc,h]`.

The routing protocol files are in the directory `mit/uAMPS` and include: `ns-leach.tcl`, `ns-leach-c.tcl`, `ns-mte.tcl`, and `ns-stat-clus.tcl`. In addition, the files `ns-bsapp.tcl`, `extras.tcl`, and `stats.tcl` contain functions needed to run the routing protocols. The files `bsagent.[cc,h]` contain the base station agent functions.

11 Running the Simulator

The following environment variables must be set: `RCA_LIBRARY=mit/rca` and `uAMPS_LIBRARY=mit/uAMPS`. Each of the protocols can be run by setting the `rp` option to “leach”, “leach-c”, “mte”, or “stat-clus”. An example for running LEACH is:

```
./ns tcl/ex/wireless.tcl \  
-sc mit/uAMPS/sims/100nodescen \  
-rp leach \  
-x 1000 \  
-y 1000 \  
-nn 101 \  
-stop 100 \  
-eq_energy 1 \  
-init_energy 2 \  
-filename leach_file \  
-dirname leach_dir \  
-num_clusters 5 \  
-bs_x 0 \  
-bs_y 0 \  
      ;# file containing node locations \  
      ;# routing protocol \  
      ;# x-size of network \  
      ;# y-size of network \  
      ;# number of nodes (including base station) \  
      ;# length of simulation (in seconds) \  
      ;# have all nodes begin with equal energy \  
      ;# amount of initial node energy (J) \  
      ;# output statistics file name \  
      ;# directory for output statistics file \  
      ;# number of clusters desired (k parameter) \  
      ;# x-location of base station \  
      ;# y-location of base station
```

The file `tcl/ex/wireless.tcl` sets some of the simulation parameters and sources the file `tcl/mobility/leach.tcl` (or `leach-c.tcl`, `mte.tcl`, or `stat-clus.tcl`). These files are linked to files with the same names in `mit/uAMPS/sims`. Each of these files sets parameters specific to that protocol and sources the file

mit/uAMPS/sims/uamps.tcl, which contains the parameters that are the same for all the routing protocols (e.g., channel bandwidth, data signal size, etc.). Table 1 shows a list of parameters that are set at the beginning of a simulation.

12 Contact Information

For more information on the MIT uAMPS project, go to <http://www-mtl.mit.edu/research/icsystems/uamps>. Please email any comments or bug fixes to uamps@mtl.mit.edu.

References

- [1] J. Broch, D. Maltz, D. Johnson, Y. Hu, and J. Jetcheva. A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols. In *Proc. 4th ACM International Conference on Mobile Computing and Networking (Mobicom'98)*, Oct. 1998.
- [2] W. Heinzelman. *Application-Specific Protocol Architectures for Wireless Networks*. PhD thesis, Massachusetts Institute of Technology, 2000.
- [3] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-Efficient Routing Protocols for Wireless Microsensor Networks. In *Proc. 33rd Hawaii International Conference on System Sciences (HICSS '00)*, Jan. 2000.
- [4] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive Protocols for Information Dissemination in Wireless Sensor Networks. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '99)*, pages 174–185, Aug. 1999.
- [5] UCB/LBNL/VINT Network Simulator - ns. <http://www-mash.cs.berkeley.edu/ns>, 1998.

Table 1: Parameters that are set at the beginning of a simulation.

Parameter	Description
x	maximum x location (m)
y	maximum y location (m)
nn	number of nodes in the simulation (including the base station)
rp	routing protocol (either leach, leach-c, mte, or stat-clus)
stop	length of the simulation (s)
eq_energy	1 if all nodes begin with equal energy, 0 otherwise
init_energy	initial amount of energy given to each node (J)
num_clusters	number of clusters desired for the protocols
ch_change	time between rounds in LEACH
check_energy	time between trace updates
hdr_size	packet header size (bytes)
sig_size	data packet size (bytes)
bw	radio bitrate (bps)
delay	link-layer delay (s)
prop_speed	channel propagation speed (m/s)
ll	link-layer protocol
mac	mac-layer protocol
ifq	internal packet queue type
netif	wireless channel
ant	type of antenna
spreading	amount of data-spreading for DS-SS
freq	carrier frequency (Hz)
L	system (non-propagation) loss
Gt	Tx antenna gain
Gr	Rx antenna gain
ht	antenna height (m)
CSThresh	received power threshold for detecting a packet (W)
RXThresh	received power threshold for receiving an error-free packet (W)
EXcvr	radio electronics energy dissipation (J/bit)
e_bf	beamforming energy dissipation (J/bit/signal)
Efriss_amp	transmit amplifier energy dissipation (J/bit/m ²)
Etwo_ray_amp	transmit amplifier energy dissipation (J/bit/m ⁴)
Pidle	power dissipated in idle mode (W)
Psleep	power dissipated in sleep mode (W)