

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Overview of SWAN</b>	<b>4</b>
2.1	Protocol Graph . . . . .	5
2.2	Radio Propagation . . . . .	6
<b>3</b>	<b>Using SWAN</b>	<b>6</b>
3.1	Installation . . . . .	6
3.2	Running Some Examples . . . . .	7
3.2.1	The Simplest Example: 2 Nodes . . . . .	7
3.2.2	A Simple 3 Node Example . . . . .	9
<b>4</b>	<b>SWAN and SOS</b>	<b>15</b>
4.1	Creating Experiments . . . . .	15
4.2	Analyzing Experiments . . . . .	16
4.3	Related Files . . . . .	17
<b>5</b>	<b>Radio Propagation Models</b>	<b>18</b>
5.1	A Brief Introduction . . . . .	18
5.2	DML Configuration . . . . .	19
5.3	Radio Propagation Models . . . . .	20
5.3.1	Friis Free-Space Model . . . . .	20
5.3.2	Two-Ray Ground Reflection Model . . . . .	21
5.3.3	General Distance Fading Model . . . . .	22
<b>6</b>	<b>Protocols</b>	<b>24</b>
6.1	IEEE 802.11 MAC Layer . . . . .	25

6.1.1	Overview of 802.11 MAC Protocol . . . . .	25
6.1.2	The Model . . . . .	27
6.1.3	DML Configuration . . . . .	28
6.2	IEEE 802.11 PHY Layer . . . . .	31
6.2.1	Overview of the 802.11 PHY Layer Model . . . . .	31
6.2.2	DML Configuration . . . . .	34
6.3	Address Resolution Protocol . . . . .	38
6.3.1	Overview of the ARP Model. . . . .	38
6.3.2	DML Configuration of ARP . . . . .	38
6.3.3	Related Files . . . . .	39
6.4	Internet Protocol . . . . .	40
6.4.1	Overview of the IP Model . . . . .	40
6.4.2	DML Configuration of IP . . . . .	40
6.4.3	Related Files . . . . .	41
6.5	Internet Control Message Protocol . . . . .	42
6.5.1	Overview of the ICMP Model . . . . .	42
6.5.2	DML Configuration of ICMP . . . . .	42
6.5.3	Related Files . . . . .	42
6.6	Ad-hoc On-demand Distance Vector Protocol . . . . .	43
6.6.1	Overview of the AODV Model . . . . .	43
6.6.2	DML Configuration of AODV . . . . .	44
6.6.3	Related Files . . . . .	48
6.7	Dynamic Source Routing Protocol . . . . .	49
6.7.1	Overview of the DSR Model . . . . .	49
6.7.2	DML Configuration of DSR . . . . .	51
6.7.3	Related Files . . . . .	55

6.8	Test Application Sessions . . . . .	56
6.8.1	Overview of Test Application Sessions . . . . .	56
6.8.2	DML Configuration of Test Application Session . . . . .	56
6.8.3	DML Configuration of NS-2 Application Session . . . . .	57
6.8.4	Related Files . . . . .	59

# 1 Introduction

The Dartmouth Simulator for Wireless Ad-hoc Networks (SWAN) is a configurable wireless network simulator that is designed to be easy to use, fast, and scalable. In order to accomplish these goals it draws on previous work in this group on the Scalable Simulation Framework (SSF) API [3], the Dartmouth SSF implementation (DaSSF) [8], and the SSFNet fixed network simulator [4] in Java. Being a wireless ad-hoc network simulator it will naturally share certain features with other simulators, such as ns-2 [2] and GloMoSim [11]. But in contrast to ns-2, we started from the ground up building a simulator that will execute on parallel or distributed computers in order to make it as scalable as possible. In this sense, it has goals similar to GloMoSim, but also has some differences because of its different roots, for example:

- Implemented in C++, since it is based on DaSSF which is written in C++.
- Uses the Domain Modeling Language (DML) for simulator configuration, in the same vein as the SSFNet simulator.
- Modular protocol graph structure based on the SSFNet design, which in turn, was inspired by the X-kernel design [5].

This document describes the design of the SWAN simulator and how one uses it, both at a high level and in depth. We start by presenting an overview of the simulator and the protocols implemented to date in Section 2. Section 3 explains how to install and use SWAN through some of the example models included in the distribution. After this follows detailed descriptions of the implemented protocols in Section 6 along with descriptions of the configuration options available for each of the protocol modules.

## 2 Overview of SWAN

SWAN relies on the DaSSF simulation kernel for all the common tasks of discrete event simulators such as: process context switching, event scheduling, random number generation, statistics collection, and also for model configuration support. Moreover, since DaSSF is

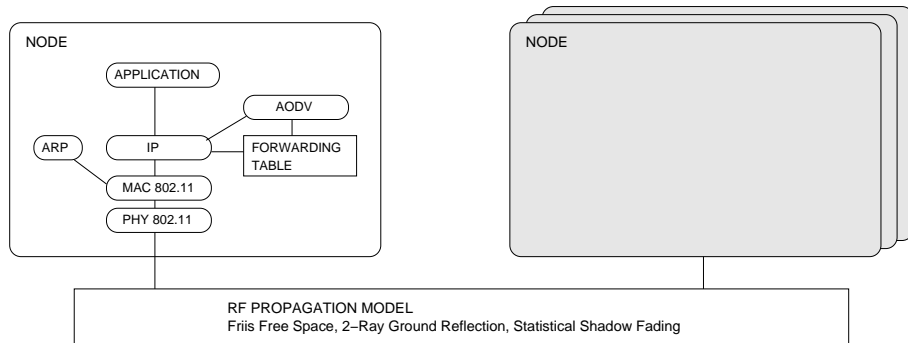


Figure 1: Currently implemented protocol graph in SWAN nodes.

designed to exploit parallel or distribution computers to scale up simulation model size, the design of SWAN is geared towards exploiting these features. However, efficiently parallelizing models of wireless communication networks is a non-trivial problem that is the subject of currently ongoing research. In particular the SWAN simulator is a research vehicle for exploring issues related to parallelization. For this reason, the publicly released version of SWAN has a number of limitations with regards to parallel execution.

## 2.1 Protocol Graph

Each (mobile) node in the simulation implements a per-node configurable protocol graph. Figure 1 illustrates the protocol graph currently implemented in SWAN and the infrastructure support for packet delivery subject to RF propagation models.

SWAN includes a model of the 802.11 protocol implemented in two parts: a protocol session for the MAC protocol, and a “pseudo-protocol-session” for the physical layer. These models were ported from the GloMoSim simulator code (version ?? \*\*\*). It also includes models of IP and ARP ported from the SSFNet simulator code. The IP layer includes a module for the forwarding table that can be accessed and manipulated by routing protocols. The current implementation includes a model of the Ad-hoc On Demand Distance Vector (AODV) routing protocol [10], derived from GloMoSim source code but updated to comply with Draft version 10 of the AODV specifications. The application layer is modeled through another protocol session, also in a similar manner to the SSFNet simulator.

## 2.2 Radio Propagation

As indicated in Figure 1, SWAN implements a few simple models of signal attenuation for radio propagation, similar to models found in ns-2:

- **Friis Free Space model.**
- **2-Ray Ground Reflection model.**
- **Statistical Shadow Fading model.**

These models are described in more detail in Section 5.

## 3 Using SWAN

In this section we explain how to install the SWAN simulator and how to use it at a “high level”, i.e. how to run simulations. More information on details and options for the various protocols follows in later sections.

### 3.1 Installation

Since SWAN relies on DaSSF, a working installation of DaSSF is a prerequisite to installing and running SWAN. We refer the reader to the SWAN User’s Manual [8] for instructions on how to install DaSSF and verify that the installation is working. However, **please note** that SWAN requires DaSSF version 3.2.4 or later, and that the following configuration options need to be enabled when installing DaSSF: `--with-ltime-longlong` and `--with-stl`. So, at a minimum the following should be done to configure and compile DaSSF:

```
> ./configure --with-ltime-longlong --with-stl
> make
```

but depending on machine architecture and OS, more information may have to be supplied to configure the installation.

Once the DaSSF installation has been successfully completed, proceed by downloading and unpacking the SWAN distribution into a directory of your choice. At this point it is sufficient to verify that the environment variable `DaSSF_HOME` points to the DaSSF installation (root) directory, and then from the SWAN distribution top directory type `make depend` to create source file dependencies, followed by `make` to build SWAN. This will generate the executable code as well as the documentation. The executable will have a name starting with `swan` and extensions depending on the machine architecture and operating system. For example, if SWAN is compiled on a Linux PC, the executable will be named `swan-X86-LINUX`. The following section describes a few bundled example models that can be used to verify that the simulator is working.

## 3.2 Running Some Examples

The SWAN distribution includes a few example models in subdirectories of the `models` directory. Each example directory has a README file that explains the model and how to run it.

### 3.2.1 The Simplest Example: 2 Nodes

Let us look at a model that illustrates the simplest possible configuration, two communicating nodes. This is in the `models/2nodes/` directory. Let us assume that the SWAN was compiled on a PC running Linux. To run this model, from the installation root directory type

```
> swan-X86-LINUX 100 models/2nodes/2nodes.dml
```

This produces a report of packet statistics per node, as in Figure ??, followed by some performance statistics for the simulator. The per-node packet statistics shows statistics at multiple levels of the protocol graph, including application packets sent/received, various routing packets, and MAC-level statistics.

As mentioned previously the model is defined by using the Domain Modeling Language (DML) which is read by the simulator to configure the model. More detail on DML in general and the DML parsing supported by DaSSF is provided in the DaSSF User's Manual [8].

```

...
-- MAC report, address: 0x10000001--
  packets to be sent: 101
  packets lost due to buffer overflow: 0
  packets sent via unicast: 100
  packets sent via broadcast: 1
  packets received via unicast: 99
  packets received via broadcast: 1
  packets retransmissions due to CTS lost: 0
  packets retransmissions due to ACK lost: 0
  packets dropped: 0
-- Node 1 ARP report --
  numRequestSent: 0
  numReplySent: 1
  numReplyRcv: 0
  numPktDropped: 0
  numQuery: 99
-----

SessAppSession[hostid=1]: #sent=99, #rcvd=98, #sent_fail=0

----- AODV NODE 10000001 REPORT -----
  number of rreq sent: 1
  number of rreq relayed: 0
  number of rrep sent: 0
  number of rrep relayed: 0
  number of rerr sent: 0
  number of rerr relayed: 0
  number of hello msg sent: 0
  number of rrep-ack sent: 0
  number of data sent as src: 99
  number of data forwarded: 0
  number of data received as dst: 98
  number of pkts dropped: 0
  number of data pkts dropped by MAC: 0
  number of link broken detected: 0
  number of bootup: 0
  number of boot success: 0
  number of abort: 0
-----
...

```

Figure 2: Part of output from 3node example.

Figure 3 lists the DML used to define this model.

As shown in Figure 3, all SWAN models must start with the keyword `model` and the argument to this keyword is a list, the contents of which defines the model. The `arena` keyword defines the space that is being simulated, in this case an area of 50 meters by 50 meters. Here we also define the mobility model, in this case no mobility, and one or more “networks” which can represent and separate different radio technologies. After the `arena`, we specify the propagation model, and this concludes the global specifications.

Next, each mobile node is defined through the `host` keyword. Each host must be given an id number, as `id <integer>`, and may be given an initial position using `xpos` and `ypos`. Let us focus on node number 1. The next important part is defining which protocols the node should run, this is done by defining the protocol graph using the `graph` keyword. Inside the graph each protocol session is defined as `session [ . . . ]` with some parameters. We specify the protocols to run from the top level down, starting with the application layer. In this example we use a simple test application `tstapp.sess-app-session` that (by default) sends packets according to a Poisson process. We specify the packet size, the mean packet inter-arrival time `iat`, and who to send the packets to through the `peer` keyword.

Each node runs the Ad-hoc On Demand Distance Vector (AODV) routing protocol `routing.aodv_sim.swan-aodv-session`, the IP protocol, the Address Resolution Protocol (ARP), and a specification of the physical interface card `interface`. Since different interface cards may run different MAC and physical level protocols, the `interface` keyword actually specifies a “sub-protocol-graph” inside. In this case by using the 802.11 protocol.

Node 2 is the same as node 1, with the exception of its location and its definition of communication “peer”.

### 3.2.2 A Simple 3 Node Example

We can extend the previous example, slightly, to 3 nodes, by looking into the `models/3nodes/` directory. The DML definitions for this example are shown in Figures 4 to 6. One powerful feature of DML is its support for including and inheriting definitions. In this example it is manifested through the use of a `dictionary` of definitions that are used and reused in

```

model [
  arena [ # Define an arena of 50m x 50m; everybody is stationary.
    mobility [ model "mobility.stationary" deployment "preset" xdim 50 ydim 50 ]
    network [ netid 1 # Wireless network 1:
      model "network.fixed-range"
      cutoff_max_signal_loss 126 # cutoff dist def by max power loss of 126 dB
    ]
  ]

propagation [ model "propagation.friis-free-space"
  carrier_frequency 2.4e9 system_loss 1.0
  temperature 290 noise_figure 10.0 ambient_noise_factor 0
]

# node 1 at (0,25,0), sending packets to node 3
host [ id 1 xpos 0 ypos 25
  graph [
    session [ name "app" use "tstapp.ssess-app-session"
      packet_size 512 iat 1.0 show_report true
      peer [ netid 1 hostid 2 iface 0 ]
    ]
    session [ name "aadv" use "routing.aadv_sim.swan-aadv-session" netid 1
      show_report true # show report at the end
    ]
    session [ name "net" use "net.ip-session" ]
    session [ name "arp" use "net.arp-session" show_report true ]
    interface [ id 0 netid 1
      session [ name "mac" use "mac.mac-802-11-session" show_report true ]
      session [ name "phy" use "phy.phy-802-11-session"
        bandwidth 11e6 accumulative_noise true interference_threshold -111.0
      ]
    ]
  ]
]

# node 2 at (50,25,0), sending packets to node 1
host [ id 2 xpos 50 ypos 25
  graph [
    session [ name "app" use "tstapp.ssess-app-session"
      packet_size 512 iat 1.0 show_report true
      peer [ netid 1 hostid 1 iface 0 ]
    ]
    session [ name "aadv" use "routing.aadv_sim.swan-aadv-session" netid 1
      show_report true # show report at the end
    ]
    session [ name "net" use "net.ip-session" ]
    session [ name "arp" use "net.arp-session" show_report true ]
    interface [ id 0 netid 1
      session [ name "mac" use "mac.mac-802-11-session" show_report true ]
      session [ name "phy" use "phy.phy-802-11-session"
        bandwidth 11e6 accumulative_noise true interference_threshold -111.0
      ]
    ]
  ]
]
]]]

```

multiple nodes. For example, all three nodes inherit their protocol session definitions from the `wireless_graph` definition in the `dictionary`.

But let us start by looking at the definition of the `arena`, at the top. The `_find` keyword substitutes in a definition from some other place, much like the use of a macro definition. Thus, the arena defined in the dictionary is copied into the model definition. The protocol sessions of the nodes, on the other hand, are created by *inheriting* from definitions in the dictionary using the `_extends` keyword. This keyword essentially copies all the keywords within the given definition into place but also allows these definitions to be subsequently overridden as desired. For instance, all the definitions for the `application_session` are copied into the application session for the node, but the peer definition is added at the end. Had the dictionary contained a peer definition, this could have been overridden with some other settings for the specific node.

Since this example uses a dictionary for definitions, in a separate DML file, we need to also supply the dictionary file to the simulator to run the model, as follows:

```
> swan-X86-LINUX 100 models/3nodes/3nodes.dml models/3nodes/dictionary.dml
```

This brief look at some example models concludes the overview description of model configuration in SWAN using DML. The following sections will explain all the protocols and modules and provide more detailed information of configuration options for each module.

```

model [
  _find .my_dictionary.arena

  propagation [
    _extends .my_dictionary.propagation_dist_fading
    temperature 290
    noise_figure 10.0
    ambient_noise_factor 0
  ]

  # node 1 at (0,0,0), sending packets to node 3
  host [ id 1 xpos 0 ypos 0
    graph [
      session [
        _extends ".my_dictionary.application_session"
      ]
      peer [ netid 1 hostid 3 iface 0 ]
    ]
    _extends ".my_dictionary.wireless_graph"
  ]
]

# node 2 at (50,0,0), doing nothing other than forwarding packets
host [ id 2 xpos 50
  graph [
    _extends ".my_dictionary.wireless_graph"
  ]
]

# node 3 at (100,0,0), sending packets to node 1
host [ id 3 xpos 100
  graph [
    session [
      _extends ".my_dictionary.application_session"
    ]
    peer [ netid 1 hostid 1 iface 0 ]
  ]
  _extends ".my_dictionary.wireless_graph"
]
]

```

Figure 4: DML for 3node example model.

```

my_dictionary [

arena [

    # Define an arena of 500m x 500m; everybody is stationary.

    mobility [
        model "mobility.stationary"
        deployment "preset"
        xdim 500 ydim 500
    ]

    # Wireless network 1: cutoff distance is defined by a maximum
    # power loss of 126 dB (sender power is 15 dBm, receiver
    # sensitivity is -111 dBm).

    network [
        netid 1
        model "network.fixed-range"
        cutoff_max_signal_loss 126
    ]
]

# Propagation models with parameters.

propagation_friis_free_space [
    model "propagation.friis-free-space"
    carrier_frequency 2.4e9
    system_loss 1.0
]

propagation_2_ray [
    model "propagation.2-ray"
    carrier_frequency 2.4e9
    system_loss 1.0
    antenna_height 1.5
    #shadowing_stdev 6
    #seed 12345
]

propagation_dist_fading [
    model "propagation.dist-fading"
    carrier_frequency 2.4e9
    system_loss 1.0
    reference_distance 1.0
    pathloss_exponent 3.0
    shadowing_stdev 6
    seed 12345
]

...

```

Figure 5: DML for 3node example model — “dictionary” part (beginning).

```

...

# Application layer sending packets of size 512 bytes with an
# exponential time interval of 1 second.

application_session [
  name "app" use "tstapp.sess-app-session"
  packet_size 512 iat 1.0 show_report true
]

# Each node is running AODV, IP, ARP protocols.

wireless_graph [
  session [
    name "aodv" use "routing.aodv_sim.swan-aodv-session"
    netid 1
    show_report true           # show report at the end
    #show_rt_table_change true # show routing table change
    #use_hello_msg true       # use hello msg
    #rreq_flag                # set default rreq flag, G-1, R-2, J-4
  ]
  session [ name "net" use "net.ip-session" ]
  session [ name "arp" use "net.arp-session" show_report true ]
  interface [ id 0 _extends ".my_dictionary.wireless_interface" ]
]

# Wireless interface card contains MAC and PHY sessions.

wireless_interface [
  netid 1
  session [
    name "mac" use "mac.mac-802-11-session"
    show_report true
    #queue_priorities 3
    #queue_size 100
    #rts_threshold 512
  ]
  session [
    name "phy" use "phy.phy-802-11-session"
    bandwidth 11e6
    #show_report true
    accumulative_noise true
    interference_threshold -111.0
  ]
]
]
]

```

Figure 6: DML for 3node example model — “dictionary” part (continued).

## 4 SWAN and SOS

SOS (Scripts for Organizing 'Speriments) is a set of Perl scripts designed to “ease the process of running a large number of experiments and working with the resulting data (including plotting)”[?]. It allows simulator parameters to be easily defined and declared as ranges, and then automatically runs all combinations of the parameters and stores the results in a MySQL database. Further scripts allow particular data to be extracted and graphed in **gnuplot**.

Sample scripts and DML generators have been prepared for *SWAN* to integrate it with SOS. This section describes how to use and modify them to quickly create large experiments, but please refer to the SOS README[?] for installation instructions, design concepts, and other details.

### 4.1 Creating Experiments

Create a new family directory in `sos/families/` with the name of your work. Copy the files `schema.pl` and `simschema.pl` from `families/swan_sample` into your family directory. These files contain definitions of simulator parameters; `schema.pl` variables can vary across a range or list of values, while `simschema.pl` parameters must stay constant for a group of experiments. Depending on which parameters you wish to experiment on, you may need to move definition lines between these files (changing `$simschema` to `$schema` as appropriate), but please note that MySQL will not allow more than fourteen parameters in the `schema` database. Also copy over the file `prepe.pl`, which will take the parameters and pass them to a DML file generator.

The *SWAN* -specific script `sos/swan/mkdml-lib.pl` will create a DML file for every combination of parameters. If some *SWAN* parameter is not listed in the schema files, you will need to edit this script in addition to adding the parameter to the schema file. See the script for more details.

Create a directory inside this family directory with the name of a particular experiment group. Inside, create two files named `vals.expr` and `vals.sim` for declarations of the exper-

iment (schema) and simulator (simschema) parameters. Use the files in `families/swan_sample/expr_sample` as models. The variable in `vals.expr` may be declared as:

- constant: `param=x`
- range: `param=[x-z]`
- list: `param=[x,y,z]`

An extractor script is necessary to parse *SWAN*'s standard report and insert derived data into the MySQL database. In the extractor directory, create two new files for the extractor and the field definitions. Use `swanThruput_def.pl` as a model for field definitions (meaningful output data), and replace the pattern-matching lines and SQL command at the end of `swanThruput.pl`. (The extractor `swanThruput` compares packet throughput, as measured by packets received over packets sent, to number of nodes).

Install the family, extractor, and experiment into your MySQL database and begin execution of the group of experiments by running the appropriate SOS scripts:

```
> mkfamily.pl <family-name> -x <extr-name>
> mkexpr.pl <family-name> <expr-name>
> do_run.pl <family-name> <expr-name>
```

Please note that if, after running either of the make-scripts, changes are made to the user-defined experiment or family scripts, the experiment or family will need to be removed and re-made.

## 4.2 Analyzing Experiments

Running an experiment fills your MySQL database with output data extracted from *SWAN*'s reports that can then be manipulated by standard SQL commands. SOS has also provided a script to run a file of SQL statements and print out the results. As an example, `extractors/swanThruput.sql` prints out a table with number of nodes compared to throughput and standard deviation.

Data tables may be easily graphed using SOS' `mkplot.pl` script. Using the `-v` flag, you may choose the gnuplot variable file `swan/mkplot_vars.pl`, which plots a line graph with error bars across the specified x-range.

### 4.3 Related Files

- Sample Family
  - `families/swan_sample/prepe.pl`
  - `families/swan_sample/schema.pl`
  - `families/swan_sample/simschema.pl`
- Sample Experiment
  - `families/swan_sample/expr_sample/vals.expr`
  - `families/swan_sample/expr_sample/vals.sim`
- Sample Extractor
  - `extractors/swanThruput.pl`
  - `extractors/swanThruput_def.pl`
  - `extractors/swanThruput.sql`
- Utilities
  - `swan/mkdml-lib.pl`
  - `swan/mkplot_vars.pl`

## 5 Radio Propagation Models

### 5.1 A Brief Introduction

The radio propagation model is used to calculate radio signal attenuation as it travels from the transmitter to the receiver. The design of the propagation model plays an important role in achieving accuracy of the simulation model.

In radio propagation model, the term *fading* is used to describe the fluctuation in the envelope of a transmitted radio signal. In general, a radio propagation model can be divided into two categories: small-scale fading and large-scale fading. In small-scale fading, a.k.a. short-term fading, we are interested in the characteristics of radio signal fluctuation over a short period of time or a small distance. In this case, the signal power shows rapid fluctuations. Small-scale fading is primarily caused multipath effect. A radio signal may travel via different paths from the transmitter to the receiver and cause. Due to varying path lengths, two or more versions of the same signal may add or subtract at the receiver and interference with each other. Also, signals suffer from the Doppler effect due to movement of the mobile host or the surrounding objects. Small-scale fading in general is hard to predict. Fortunately, radio engineers have invented ways to correct it with adaptive equalizers or robust modulation and error correction techniques. There are classic statistic models that predict small-scale fading, such as Rayleigh fading and Rice fading. The current version of *SWAN* does not include models for calculating small-scale fading.

Large-scale fading, a.k.a. long-term fading, describes the signal power level over a long time interval or large distance. The signal power, in this case, is the average power level and slowly varying. In large-scale fading, we consider two factors. One is related to signal attenuation as a function of signal traveling distance, also called distance path loss. The other is called shadowing, which is due to environmentally scattered fields along the transmission path. It turns out that the shadowing effect can be simulated by a log-normal distribution with a mean equal to the distance path loss value.

```

model [
  arena [ ... ]
  propagation [
    model "propagation.friis-free-space"
    # model specific parameters
    carrier_frequency 2.4e9
    system_loss 1.0
    # parameters used to compute background noise
    temperature 290
    noise_figure 10.0
    ambient_noise_factor 0
  ]
  host [ ... ]
  ...
]

```

Figure 7: A DML Script for Setting Up A Propagation Model

## 5.2 DML Configuration

In *SWAN*, we implemented three propagation models to calculate radio propagation for large-scale fading effect: Friis free-space propagation model, two-ray ground reflection model, and a general distance path loss model.

The user can choose which model to use and select appropriate parameters for the chosen model using DML script. The parameters are specified within `propagation` attribute in the model specification. Figure 7 shows an example of setting up a propagation model for an ad hoc network. The network uses Friis free-space propagation model.

We leave the description of model specific parameters until the next section. There are parameters common to all propagation models. These parameters are used to calculate the background noise level for the entire network. In particular, the background noise is a sum of two kinds of noises: thermal noise and ambient noise. Thermal noise  $N_t$  is calculated

from the following formula:  $N_t = B * T * F$ , where B is the Boltzmann constant (1.379e-23), T is the temperature of the environment (in Kelvin), and F is the noise figure. The user can specify all parameters needed for calculating the background noise:

- `temperature` is the temperature of the environment (in Kelvin). The default temperature is 290 (or 17C).
- `noise_figure` is the noise figure. The default is 10.
- `ambient_noise_factor` is the ambient noise level (in Watts). The default is zero.

## 5.3 Radio Propagation Models

### 5.3.1 Friis Free-Space Model

Friis free-space model assumes ideal propagation condition. The model assumes that signals travel in a vacuum space without obstacles from the transmitting antenna in an isotropic fashion. That is, the transmitting antenna radiates its signal power uniformly in all directions and there must be a line-of-sight path between the transmitter and receiver.

Friis free-space model can be summarized as follows. Assuming the receiver is at a distance  $d$  away from the transmitter, the receive power  $P_R(d)$  can be obtained by the following equation:

$$P_R(d) = \frac{P_T G_T G_R \lambda^2}{(4\pi d)^2 L}, \quad (1)$$

where  $P_T$  is the transmitted signal power,  $G_T$  and  $G_R$  are the antenna gains for the transmitter and the receiver respectively.  $\lambda$  is the carrier wave length, and  $L \geq 1$  is the system loss factor.

In *SWAN*, all parameters of the above equation can be set at runtime using DML script. In particular, the transmission power ( $P_T$ ), the antenna gains ( $G_T$  and  $G_R$ ) can be set when configuring the PHY layer of a mobile host (see 6.2). Others can be obtained or calculated indirectly from the following DML attributes:

- `carrier_frequency` specifies frequency (in Hz) of the carrier. The default is 2.4e9, that is, 2.4 GHz.

- `system_loss` sets the system loss factor. It must be no less than 1. The default is 1.

### 5.3.2 Two-Ray Ground Reflection Model

The two-ray ground reflection model is a more realistic model than the highly ideal free-space model for mobiles close to the Earth surface. In this model, we consider two paths from the transmitter to the receiver: a direct path assuming a signal line-of-sight between the two, and a ground reflection path assuming a perfectly flat ground. The receive power at a distance  $d$  from the transmitter  $P_R(d)$  can be calculated as follows:

$$P_R(d) = \frac{P_T G_T G_R h_T^2 h_R^2}{d^4 L}, \quad (2)$$

where  $P_T$  is the transmitted signal power,  $G_T$  and  $G_R$  are the antenna gains for the transmitter and the receiver respectively.  $h_T$  and  $h_R$  are the height of the transmitter antenna and receiver antenna.

This two-ray ground reflection model is more accurate than the free-space model when the distance between the transmitter and receiver is large. However, the above equation does not apply well to situations when the distance is small. In that case, we switch back to free space model when the distance is smaller than a crossover distance. To achieve continuity, we equalize the right side of Equation (1) and that of Equation (2). We obtain the crossover distance  $d_c$ :  $P_R(d)$  can be calculated as follows:

$$d_c = \frac{4\pi h_T h_R}{\lambda}. \quad (3)$$

That is, if the distance between the transmitter and the receiver is larger than  $d_c$ , we apply two-ray ground reflection model using Equation (2). Otherwise, we apply Friis model using Equation (1).

The model also considers shadowing effect. It reflects the variation of received signal power at a given distance. We use a log-normal distribution. That is, we have a Gaussian distribution if we compute the received power in dB. The Gaussian random variable has zero mean and a standard deviation, the value of which depends on the characteristic of the environment. Note that the shadowing model introduces probability in computing the

received signal power: each node can only communicate with its neighbors in a probabilistic manner.

The model accepts the following DML parameters:

- `carrier_frequency` specifies frequency (in Hz) of the carrier. The default is 2.4e9, that is, 2.4 GHz.
- `system_loss` sets the system loss factor. It must be no less than 1. The default is 1.
- `antenna_height` is the height of the antenna for all mobile nodes. We simplify the calculation by assuming all antennas in mobiles are of the same height.
- `shadowing_stdev` is the standard deviation used for simulating the Gaussian shadow fading effect. The value is in dB. The default is 0, i.e. no shadowing.
- `seed` is the initial random seed to the random stream used to sample Gaussian distribution for the shadowing effect.

### 5.3.3 General Distance Fading Model

The general distance fading model describes radio signal attenuation as a combination of two effects: a path loss, which is used to predict the mean signal power at a given distance from the transmitter, and a shadowing effect, which is used to reflect signal power fluctuation at a given site. The path loss is inversely proportional to the distance raised to a specified exponent. The shadowing effect is obtained from a log-normal distribution. The power gain  $G_{dB}(d)$  in dB at a distance  $d > d_0$  away from the transmitter can be calculated as follows:

$$G_{dB}(d) = G_{dB}(d_0) - 10\beta \log_{10}(d/d_0) + X_{dB}, \quad (4)$$

where  $G_{dB}(d_0)$  is the power gain in dB at the reference distance  $d_0$  computed by the Friis free-space propagation model,  $\beta \geq 2$  is the path loss exponent,  $X_{dB}$  is random variable of a Gaussian distribution with zero mean and a standard deviation of  $\delta_{dB}$ .

Typical values of the path loss exponent  $\beta$  and the shadowing standard deviation  $\delta_{dB}$  for a given type of environment can be obtained from the literature.

The general distance fading model accepts the following DML parameters:

- `carrier_frequency` specifies frequency (in Hz) of the carrier. The default is 2.4e9, that is, 2.4 GHz.
- `system_loss` sets the system loss factor. It must be no less than 1. The default is 1.
- `reference_distance` is the reference distance ( $d_0$ ) in meters. The default is 1.
- `pathloss_exponent` specifies the path loss exponent ( $\beta$ ). The default is 2 for free space.
- `shadowing_stdev` is the standard deviation used for simulating the Gaussian shadow fading effect. The value is in dB. The default is 0, i.e. no shadowing.
- `seed` is the initial random seed to the random stream used to sample Gaussian distribution for the shadowing effect.

## 6 Protocols

## 6.1 IEEE 802.11 MAC Layer

### 6.1.1 Overview of 802.11 MAC Protocol

IEEE 802.11 is a protocol standard for wireless local area networks (WLAN), which consists of both the physical (PHY) layer and the medium access control (MAC) layer specifications [6]. It provides asynchronous and time bounded delivery service for wireless connectivity of fixed, portable, and mobile stations moving at pedestrian and vehicular speeds within a local area.

The 802.11 MAC layer protocol provides shared access to a wireless channel. In particular, the distributed coordination function (DCF) is designed as the primary access method for contention-based shared access to the medium in wireless ad hoc networks. DCF is based on CSMA/CA, which stands for Carrier Sense Multiple Access with Collision Avoidance. We briefly describe how DCF works in the following. Users who are familiar with the 802.11 can skip the rest of the session and proceed to the next.

The core mechanism used by DCF is called *Basic Access Method*, which is summarized in Figure 8. Before a station initiates transmission of a data frame (called MAC protocol data unit, or MPDU), it needs to sense the channel in order to determine whether another station is currently transmitting. The station can proceed with its transmission if the medium is determined to be idle for a time interval of DIFS (DCF Inter-Frame Space). Once a data frame has been successfully received at the receiver, it must send an acknowledgment frame (ACK). This is because that the transmitter cannot determine whether a frame is faithfully delivered to its destination by simply listening to the channel—the sender may not be able to observe frame collisions at the receiver, which may detect other transmissions not observable by the first (this is the so-called “hidden terminal problem”). To transmit the ACK, the receiver waits for the channel to be idle for another time interval called SIFS (Short Inter-Frame Space). If the sender does not receive an acknowledgment within a certain time-out period, it presumes that the data frame is lost and schedules a re-transmission.

If the medium is busy upon transmitting a data frame or an ACK, the transmission must be deferred until the end of the ongoing transmission. In this case, a random backoff

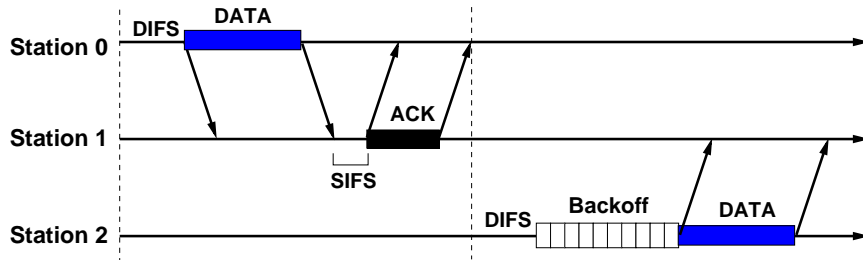


Figure 8: Basic Access Method of IEEE 802.11 MAC Protocol

interval is selected, as follows. A backoff timer is set with a random backoff integer ( $BV$ ) drawn from a uniform distribution over the interval  $[0, CW - 1]$ , where  $CW$  (Collision Window) is an integer within the range of  $CW_{min}$  and  $CW_{max}$ .  $BV$  is the number of idle “slots” the station must wait until it is allowed to transmit—there is a specified and understood slot duration. The value is decremented by one for each idle slot detected. The backoff timer suspends when the medium becomes busy before  $BV$  reaches zero. The timer resumes only after the medium has been idle longer than the designated inter-frame space interval. The station starts transmitting the frame when the backoff timer reaches zero. For each successive re-transmissions, the value of  $CW$  increases exponentially (i.e.  $CW_{new} = CW_{old} * 2 - 1$ ), until it reaches and then stays at  $CW_{max}$ .  $CW$  will be reset to  $CW_{min}$  after a successful transmission. The backoff method is used to minimize collisions and maximize throughput at both low and high network utilizations.

DCF also provides an alternative way of transmitting data frames that involves transmission of RTS (request to send) and CTS (clear to send) prior to the actual data transmission. RTS and CTS are used to reserve the channel between the transmitter and receiver. An RTS frame is transmitted by a station which needs to transmit a data packet. The receiving station responds with an CTS frame. The rules for transmission of RTS and CTS frames are the same as those of the data frame and the acknowledgment frame. RTS and CTS frames contain a duration field that tells the period of time the channel is to be reserved for transmitting the data frame. This information is picked up by other stations in the area that are sensing the channel. It helps them to construct NAV (network allocation vector) – the period of time a station is required to be kept silent. The technique is referred to as virtual carrier sense mechanism and is used to reduce contentions due to hidden terminals.

### 6.1.2 The Model

The IEEE 802.11 MAC layer model in *SWAN* was originally ported from UCLA’s Glo-MoSim [1]. We modified the original implementation to take advantage of *SWAN* framework and its supporting simulator kernel. The model is implemented as a protocol session (Section ??) as part of the network interface within a mobile host. The session is named “mac.mac-802-11-session” and its source code can be found in `include/mac/mac-802-11-session.h` and `src/mac/mac-802-11-session.cc`.

The IEEE 802.11 MAC layer model provides detailed implementation of DCF, having both basic access method and the extended method using RTS/CTS frames. Whether to use RTS/CTS frames during a packet transmission is determined by the size of the packet. If the packet size is larger than a given threshold value, the MAC layer will first use RTS/CTS method to reserve the radio channel. Otherwise, the packet is transmitted directly. The model is a fairly detailed state machine, having implemented important aspects of the protocol standard. Still, it is important to mind for its limitations:

1. The model implements only DCF. Another important access method of 802.11 known as point coordination function (PCF) is useful only for infrastructure network configurations (i.e. with access points), and therefore is not implemented.
2. The model does not support fragmentation. Fragmentation is used by 802.11 MAC protocol to provide flexible transmission. An 802.11 transmitter can optionally break messages into small fragments. It is possible that shorter data bursts can be more robust and may reduce the chance of errors due to signal fading or noise. The current MAC model does not have that.

The MAC session accepts packets *pushed* down from the network session above it, assuming it’s an IP protocol. The packets are first queued up in an internal buffer. Packets may have priorities and each priority maintains a separate FIFO queue with prespecified queue size. Packets at the the front of the queue of the highest priority will be transmitted first once the channel is detected idle. Transmission is simulated by pushing the packet down the protocol stack to the PHY session below it. The 802.11 PHY session is responsi-

ble for transforming the packet into signal representation (called radio frames in *SWAN* ) and sending it out to neighboring mobile stations within radio transmission range. These are taken care of by the *SWAN* framework. Once a mobile station successfully receives a radio frame, the PHY layer of the mobile station transforms the radio frame back to packet representation and *pops* it up to the MAC session. The MAC session handles the receiving packet and finally pops it up to the network layer above.

The MAC layer also keeps track of packet delivery and packet drop and reports their occurrences to the network layer. For example, once a unicast message has been successfully sent out, a control message is sent to the session above. Or, if a unicast message is dropped due to buffer overflow or due to the maximum number of retransmissions having been exceeded, the MAC session sends a control message to the network session as well. Routing protocols may use these control messages (by registering as listeners at the IP session) to adjust their behavior accordingly.

The 802.11 MAC session model supports power cycling and signal jamming to take part in simulations for attack scenarios in wireless ad hoc networks. See Section ?? for details on how to set up experiments to simulate attacks.

### 6.1.3 DML Configuration

The IEEE 802.11 MAC session can be configured automatically at runtime from the DML script. The DML script for a typical setup of a mobile host with 802.11 MAC session given is shown in Figure 9. As mentioned earlier, we require that an 802.11 MAC session be a protocol session defined within the definition of a network interface. Also, 802.11 MAC session assumes the presence of IP session and 802.11 PHY session above and below it. That is, there must be an IP session (Section 6.4) in the protocol graph of the mobile host. And, the protocol session below the 802.11 MAC session must be an 802.11 PHY session (Section 6.2). Both 802.11 MAC session and PHY session are protocol sessions of a network interface within a mobile host.

Since the MAC session can be configured from the DML script at runtime, a user can provide parameters—different from the default values—to set up the MAC session for a

```

host [
  id 1 # assume this is host no. 1
  graph [
    ... # other protocol sessions
    session [
      name "net" use "net.ip-session"
      # other parameters for the IP session
    ]
  interface [
    id 0 # this eth0
    netid 1 # this interface interact with wireless network of id 1
    graph [
      session [
        name "mac" use "mac.mac-802-11-session"
        # other parameters for the 802.11 mac session
      ]
      session [
        name "phy" use "phy.phy-802-11-session"
        # other parameters for the 802.11 phy session
      ]
    ]
  ]
]
]

```

Figure 9: DML Setup for A Mobile Node Running IEEE 802.11 Protocol

```

session [
    name "mac" use "mac.mac-802-11-session"
    queue_size 500
    rts_threshold 1024
    show_report true
]

```

Figure 10: An Example DML Script for 802.11 MAC Session

particular simulation run without having to recompile the simulator. These parameters are described in the following:

- `queue_priorities` is the total number of different packet priorities. In the MAC session, each priority has a separate queue. The MAC session always retrieves packet from the queue with highest priority first. The allowable packet priority is from 0 to (`queue_priorities-1`). If the attribute is omitted, the default value is 3.
- `queue_size` is the size of the queue for each packet priority. By default, the queue size is 100.
- `rts_threshold` is the threshold whether to use RTS/CTS before transmitting a unicast data packet. If the size of a unicast packet is larger than `rts_threshold`, the MAC session uses RTS/CTS method before transmitting the packet. The default value is 3000 bytes.
- `show_report` determines whether to show statistic report. If it is `true`, the MAC layer will print out the statistics at the end of the simulation. The default is `false`.

Figure 10 shows a snippet of a DML script for a MAC session. The MAC session has a total of 3 priorities (default), each with a queue of 500 packets in size. The session uses RTS/CTS method when the packet to be sent is larger than 1KB. We ask the simulator to print out the statistics of the MAC session.

## 6.2 IEEE 802.11 PHY Layer

### 6.2.1 Overview of the 802.11 PHY Layer Model

The 802.11 PHY layer together with the MAC layer implements the IEEE 802.11 protocol standard for wireless local area networks [6]. The PHY layer model is used to represent the radio transceiver of the mobile station running 802.11 protocol. The model is tightly coupled with the radio channel model of the *SWAN* framework. A radio frame—a train of radio signals representing a packet at the PHY layer—is sent out from the PHY layer of a mobile station and delivered to all neighboring stations that the radio signal can reach. The PHY layer interacts with the *SWAN* framework to simulate the packet delivery. This includes interacting with the radio propagation model and determining radio reception based on signal receiving power and prevailing noise and interference. The IEEE 802.11 PHY layer model in *SWAN* was originally ported from UCLA’s GloMoSim [1]. We modified the original implementation to fit in the *SWAN* framework and take advantage of its underlying simulator engine. The model is implemented as a protocol session called “phy.phy-802-11-session”. The source code of the session can be found in `include/phy/phy-802-11-session.h` and `src/phy/phy-802-11-session.cc` under the source tree.

Radio signal that propagates through the air suffers from power loss when travelling from the transmitter to the receiver. Moreover, the noise and the interference at the receiver may affect the reception of the signal. The design of radio signal propagation and interference model plays an important role for both accuracy and efficiency of the wireless ad hoc simulation. In a classical channel model, signal attenuation can be represented as a product of long-scale and short-scale fadings. Long-scale fading depends on distance and shadowing by large objects. Short-scale fading encompasses the effect of multi-path propagation interference and varying traveling time. In the current version of *SWAN*, we consider only long-scale fading.

It is important to mind the limitations of *SWAN* in calculating radio signal propagation and interference. We consider only packet-level transmissions, instead of finer levels, like bits or chips. An entire packet (in the format of a radio frame) is transmitted and delivered to neighboring mobile stations. The characteristic of the entire transmission of a packet is

sampled at the time of the transmission. This includes the position of the sender mobile station and its transmission power. We assume the position and the transmission power do not change during the entire packet transmission. At the receiver side, the receiving power of the entire packet is calculated by the propagation model (Section 5). Similar to the transmitter, we assume that the position of the receiving mobile station and the signal receiving power do not change during the period of reception. The receiving power is compared against the prevailing noise level; the radio frame is correctly received if the signal-to-noise ratio (SNR) is larger than a given threshold and no error is detected due to channel fading. Channel fading is defined as a probability of packet loss and is related to modulation and error correction characteristics of the radio receiver.

The 802.11 PHY layer model implements a two-threshold scheme [7]. The carrier detect threshold (CDT) is defined as a carrier signal level, below which the receiver will not do a receive. In *SWAN*, the received radio frame is considered as a detectable transmission, only if the receiving signal power is above the threshold. Otherwise, it can only contribute to the noise at the receiver. Of course, whether the radio frame is truly received error-free further depends on the signal to noise ratio. Another threshold called the defer threshold or DT is used to determine the quality of reception. If the receiving carrier signal power level is observed above the threshold, the PHY layer can hold up a pending transmission request. The combination of this threshold together with RTS/CTS can be used to reduce the effect of hidden terminals. Both CDT and DT are determined by the receiver circuitry and configurable in the simulation model.

The propagation model determines the receiving signal power of a packet transmission. The interference calculations treat all transmissions other than the one that dominates as noise. Conceptually, all transmissions in the entire system should be accounted for to derive accurate noise level. In practice, this is unnecessary and time-consuming, since, if there is significant signal attenuation, the received signal power becomes too low to affect the logic of the receiver. In *SWAN*, the radio network (Section ??) defines a cut-off distance beyond which transmissions are considered irrelevant. We use a threshold called interference threshold to determine whether a received signal should be considered in the interference calculation at the receiver mobile station. The cut-off distance of the

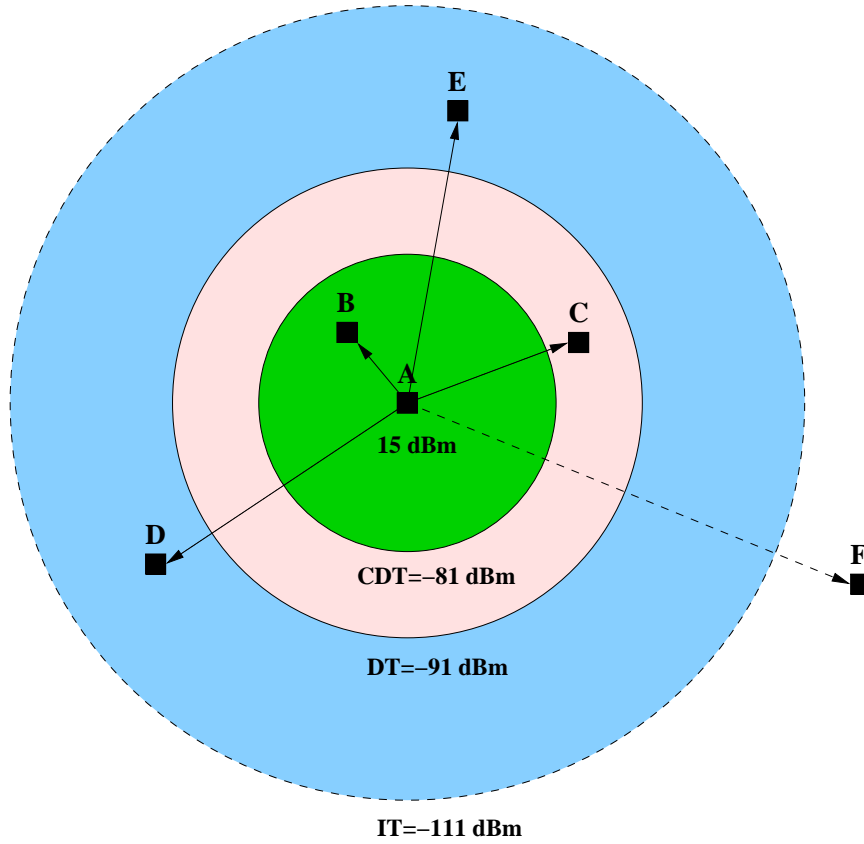


Figure 11: Thresholds for Radio Signal Transmission and Reception

system is determined by the difference between maximum packet transmission power and the interference threshold. For example, if the maximum packet transmission power is 15 dBm, and the interference threshold is set to be -111 dBm, the maximum power loss is at most 126 dB. The maximum power loss is then used to calculate the cut-off distance from the propagation model.

Figure 11 shows an example where station A is transmitting a radio frame. The transmission power is 15 dBm. Assuming only distance path loss for propagation, the power of the signal degrades to -81 dBm when it reaches the inner circle, -91 dBm at the middle circle, and -111 dBm at the outer circle. In this case, station B will be able to receive the signal as a correct transmission as long as there are no other stations transmitting and causing interference at station B. Station C cannot pick up enough signal power to receive the signal faithfully. Yet, it can defer its pending transmissions since the received signal power is above DT. That is, according to IEEE 802.11 PHY specification, the radio of station

C should remain silent during the ongoing transmission from A. Station D and E cannot recognize the transmission at all. The received signal power for A's transmission at D and E can only be used to add to the noise level, which could potentially affect the receptions at D and E during the packet's transmission. The power loss for the transmission is too large for Station F to even consider the transmission exist at all. For efficiency reasons, the *SWAN* framework will not deliver the radio frame to station F in this case.

The 802.11 PHY session model supports power cycling and signal jamming to take part in simulations for attack scenarios in wireless ad hoc networks. See Section ?? for details on how to set up experiments to simulate attacks. Upon the current release, the PHY model does not yet include power management support specified by the IEEE 802.11 Standard.

### 6.2.2 DML Configuration

A DML script for a typical setup of a mobile host with 802.11 PHY session is shown in Figure 9. We require that an 802.11 PHY session should be a protocol session defined within the definition of a network interface. That is, both 802.11 MAC session and PHY session are protocol sessions of a network interface within a mobile host. 802.11 PHY session assumes the presence of 802.11 MAC session above it (Section 6.1).

Many parameters can be used to adjusted the runtime behavior of the PHY layer. These parameters are given in the DML script to take effect at runtime. This saves the user from having to recompile the simulator. The parameters are described in the following:

- `medium_access_method` specifies whether the PHY layer is using DSSS (direct sequence spread spectrum) or FHSS (frequency hop spread spectrum). The default is DSSS.
- `bandwidth` specifies the bandwidth of the radio channel (in bits per second). The default value is 11e6.
- `propagation_delay` is the average propagation time (in nanoseconds) for a radio frame to travel from one mobile station to another. In *SWAN* , we assume a fixed propagation delay for all transmission regardless of the distance between the trans-

mitter and receiver. The assumption should not be a problem for the accuracy of the model since all propagation delays are in a much smaller scale with respect to other operations in the model. The default propagation delay is 1 microsecond.

- `synchronization_time` is the time (in nanoseconds) associated with the transceiver logic and is used for each packet transmission. The default value is 192 microseconds (from GloMoSim).
- `signal_airborne_delay` is the time for a radio signal to go through the transmitter's circuitry. The default value is 5 microseconds if the medium access method is DSSS, or 20 microseconds if FHSS.
- `flakiness` is a flat error rate (between 0 and 1) applied to an entire packet transmission. Once a packet is received, we sample a Bernoulli random variable. We reject the received packet with a probability of the given flakiness. The default is 0.
- `tx_antenna_gain` is the antenna gain of the transmitter (in dB). The default is 0.
- `rx_antenna_gain` is the antenna gain of the receiver (in dB). The default is 0.
- `tx_power` is the carrier signal power (in dBm) of all transmissions from this radio. The default is 15 dBm. The PHY layer model supports runtime adjustment of the transmission power as control messages from the upper protocol layers.
- `defer_threshold` sets the defer threshold or DT (in dBm). The default is -91 dBm.
- `carrier_detect_threshold` sets the carrier detect threshold or CDT (in dBm). The default is -81 dBm.
- `capture_snr_threshold` is the threshold value for radio capture. It is the threshold for signal to noise ratio (in DB). The default is 10. If the newly received signal power is stronger than the noise and the ratio is above the threshold, the new transmission is captured by the receiver. If the newly received signal power is weaker with respect to this threshold. There are two possibilities. If the new receiving signal power is much lower than the current receiving signal power, it is treated as part of noise. If

the signal to noise ratio remains larger than the threshold, the current transmission is not disturbed by the new one. If, however, the received signal power levels of the two transmissions are more or less the same, that is, when SNR is below the capture threshold no matter which one is treated as noise, the two transmissions collide one another and no one can be faithfully received in this case.

- `accumulative_noise` specifies whether the received signal power of the transmissions that are considered as noise should be accumulative or not. If `true`, the noise level at the receiver end increases at the start of the transmission and decreases when the transmission finishes. Otherwise, the model only considers the last transmission as the current noise level. This parameter is set false only to achieve better efficiency at the cost of accuracy. The default is true.
- `interference_threshold` is the threshold (in dBm) used to determine interference calculations. If the received signal power is lower than the threshold, the transmission will not even be considered as part of noise at all. The default is -111 dBm. The interference threshold is used only when `accumulative_noise` is true. The transmission power and the interference threshold can be used to calculate maximum power loss value, which is used to compute the cutoff distance of the radio network.

Figure 12 shows an example of a DML configuration of the PHY session. In the example, we have a radio channel with a bandwidth of 2 Mbps. CDT and DT are set to be -85 dBm and -100 dBm, respectively. Although we do not specify the rest of the parameters explicitly in DML script, the PHY layer model takes the default. For example, the PHY session uses DSSS method and accumulative noise mode for interference calculations.

```
session [  
  name "phy" use "phy.phy-802-11-session"  
  bandwidth 2e6  
  carrier_detect_threshold -85  
  defer_threshold -100  
]
```

Figure 12: An Example DML Script for 802.11 PHY Session

## 6.3 Address Resolution Protocol

### 6.3.1 Overview of the ARP Model.

ARP (Address Resolution Protocol) is the protocol that resolves IP address / MAC address translation.

ARP in *SWAN* is trying to mimic the real implementation very closely. When a packet is pushed down to the MAC layer, the MAC layer first consults the ARP to translate the next hop IP to a recognizable MAC address. If the ARP does not have such information in its cache, it will do a broadcasting request to its neighborhood. The host that has that IP address, if it is within one-hop distance, will send back an ARP reply. This information will be cached in the host for 20 minutes. While the ARP is trying to resolve the address, the pending packet will be kept by the ARP. The ARP has an one-entry buffer for each address, if another packet arrives with the same unresolved address, the previous packet will be erased. A simple mechanism that prevents ARP flooding is applied: no more than one ARP request for a given address can be sent out in a given second. After several timeouts of the requests, that address will be rejected for a while when queried.

One can choose not to install ARP in the protocol graph in the simulation to save some resource. In that case the IP address / MAC address translation is faked in the MAC layer, and no network traffic will be generated because of address translation.

### 6.3.2 DML Configuration of ARP

Figure 13 shows an example of the DML configuration of the ARP session.

```
session [  
    name "icmp" use "net.icmp-session"  
    show_report true  
]
```

Figure 13: An Example DML Script for ARP Session

- **show\_report**: This is an option to output a summary of the ARP session at the end of the simulation. It's a boolean switch and the default is **false**. The report includes:
  - numRequestSent*: number of ARP requests sent
  - numReplySent*: number of ARP reply sent.
  - numReplyRcv*: number of ARP reply received.
  - numPktDropped*: number of packets dropped by the ARP.
  - numQuery*: number of query processed.
- **pkt\_priority**: This option set the priority of the ARP packets. It's used by the MAC layer that supports multiple priority queues. When using with MAC802.11, the valid priority range is [0, 1, 2]. 0 is the highest priority, and 2 is the default priority.

### 6.3.3 Related Files

- Header files
  - include/net/arp.h
- Source files
  - src/net/arp.cc

## 6.4 Internet Protocol

### 6.4.1 Overview of the IP Model

Internet Protocol is one of the most basic protocols of the current Internet. The model developed in *SWAN* is a general purpose network layer protocol. Usually in the protocol graph, a network layer protocol is necessary. In *SWAN*, the network layer is a merge point of multiple upper layers and multiple interfaces (MAC layers).

The basic functions of the IP is straightforward. When a packet arrives at IP:

1. It checks whether the packet is sent to the local host. If it is, send it to the proper upper layer protocol. If no proper upper layer protocol has been installed, an error will be generated.
2. If the packet is not destined to the local host, IP tries to forward the packet. It first checks the TTL (Time To Live) of the packet, if it's zero, the packet will be dropped with `NET_ERROR_TTL_EXPIRE`. if the TTL hasn't expired, it's decremented by one and IP consults the forwarding table to find a route to forward the packet. If no route is found, a `NET_ERROR_NO_ROUTE` error will be returned. Otherwise IP uses the proper route entry information and pushes the packet to the proper MAC layer.

IP in *SWAN* provides API for protocols that need to intercept packets passing the IP layer. It also enables other protocol to register to listen to packet drop / delivered notification from the MAC layers, if such functionalities exist in the MAC layer.

### 6.4.2 DML Configuration of IP

The configuration of IP in *SWAN* is simple and straightforward. Figure 14 shows an example of the configuration. It configs the IP session to dump the kernel forwarding table every one second.

- **forwarding\_table**: IP can use different instances of forwarding table as long as they are extended class of `RoutingTable` and use `RouteInfo` as its routing entry type. Its

```

session [
    name "net" use "net.ip-session"
    dump_kernel_fwd_table true
    dump_interval 1.0
]

```

Figure 14: An Example DML Script for IP Session

default value is `hash`, and it's also the only currently available class provided.

- `dump_kernel_fwd_table`: IP provides the option to dump the kernel routing table every `dump_interval` second to the output. The current output format is “kernel\_fwd\_table: time %g src  $\implies$  dest next\_hop”  
It's a boolean switch and the default value is `false`.
- `dump_interval`: If `dump_kernel_fwd_table` is set to be true, this option specifies the interval (in second) of dumping the kernel forwarding table. It's a float variable and the default is 1.0.
- `send_remote_icmp`: If “ICMPSession” is installed in the protocol graph, IPSession has the option to send ICMP error messages (host unreachable, port unreachable, etc.) to the remote host. This is a boolean switch and the default is `false`.

### 6.4.3 Related Files

- Header files  
include/net/ip-session.h    include/net/ip-header.h
- Source files  
src/net/ip-session.cc    src/net/ip-header.cc

## 6.5 Internet Control Message Protocol

### 6.5.1 Overview of the ICMP Model

ICMP (Internet Control Message Protocol) in *SWAN* is a much simplified version of its counterpart in reality. The role of ICMP here is just to provide an error reporting mechanism for other protocols in the protocol graph. Commands like “ping”, “echo”... are not implemented for now. By default, sending ICMP error report to remote host is suppressed at the network layer (IPSession). In order to provide error reports (host unreachable, port unreachable, etc.) to higher level protocols, it's recommended to have it in the protocol graph. The restriction of ICMP is simple in *SWAN* : no icmp message should be generated for icmp packets. Some other restrictions exist in reality.

### 6.5.2 DML Configuration of ICMP

As shown in Figure 15, no special configuration option is provided for ICMP. Other protocol sessions using ICMP control method to register to ICMP and listen to their own error reports.

```
session [ name "icmp" use "net.icmp-session" ]
```

Figure 15: An Example DML Script for ICMP Session

### 6.5.3 Related Files

- Header files

```
include/net/icmp-session.h    include/net/icmp-header.h
```

- Source files

```
src/net/icmp-session.cc    src/net/icmp-header.cc
```

## 6.6 Ad-hoc On-demand Distance Vector Protocol

### 6.6.1 Overview of the AODV Model

The AODV (Ad-hoc On-demand Distance Vector) model in *SWAN* is initially developed based on AODV Internet Draft 10 [9]. It is a reactive routing protocol for ad-hoc networks. In the current model, supported features include: **Expanding Ring Search**, **Local Repair**, **Hello Message**, and **Blacklist**. Only unicast is implemented, and it doesn't have enough support for the aggregated networks. The support for multiple network cards is in place but hasn't been extensively tested. In the real implementation, AODV uses UDP to communicate, but in the current simulation, AODV only requires IP to operate.

AODV intercepts packets that pass the IP layer when it operates. In AODV, there are four types of control packets: RREQ (Routing Request), RREP (Routing Reply), RERR (Routing Error), RREP-ACK (Routing Reply ACK). Below is a brief description of the basic operations of AODV.

When there is a packet push down to IP, AODV intercepts the packet and check whether there is an active route for the destination of the packet. If there is one, the packet is released to go through the normal procedure of the IP forwarding. If no active route is found, AODV initialize a request (RREQ) for the destination IP address. The RREQ is sent out by broadcasting to 255.255.255.255. In the same time, a timer is set for the RREQ sent out. When it times out, another round of RREQ for the same address will be sent out. This repeats until it has done network-wide broadcasting for RREQ\_RETRIES times or a reply (RREP) has been received for the requested address.

When a host receives a request, it checks whether it knows an active fresh <sup>1</sup> route for the request destination address. If it does, it unicasts back a reply (RREP). Otherwise it will forward the request by re-broadcasting it. The request will not be forwarded when its TTL (Time To Live) in the IP header expires.

In ad-hoc network, instability is often expected. When a route broken is detected via

---

<sup>1</sup>Whether a route is fresh enough is decided by a sequence number associated with the destination of the route.

either the notification from other protocol sessions or AODV control messages, a route error (RERR) message will be sent to the upstream host of the route.

When a host suspects that some link to a peer is uni-directional, it may set an **A** flag in the reply (RREP) it sends out to that peer. That peer should respond with a reply ACK (RREP-ACK) upon receiving the RREP.

### 6.6.2 DML Configuration of AODV

- **iface**: This is an attribute that can appear more than once in the configuration. By default, AODV controls all available interfaces on the hosts. If not all interfaces should be controlled, use this attribute to specify the ID of each interface that should be controlled.
- **local\_repair**: This is the boolean switch of the optional feature: **local repair**. The default value is **false**. If **local\_repair** is set to be true, when a link broken is detected, the upstream host may choose to try repairing the broken route before sending an error (RERR) message back. The repairing is done by sending out a request trying to fix the route. If the route can't be fixed, RERR will be sent back to upstream hosts. If the route is fixed but became longer in hops, an RERR will be sent to upstream hosts with **N** flag.
- **use\_eps**: This is the boolean switch of the feature: **Expanding Ring Search**, and its default value is **true**. When EPS is applied, the host tries to do less network-wide request flooding. When a fresh request is needed, the host first set the TTL of the request to be **TTL\_START**. Each timeout the **ttl** is increased by **TTL\_INCREMENT** until it reaches **TTL\_THRESHOLD**, after which it will do network-wide broadcasting (**TTL == NET\_DIAMETER**) for **RREQ\_RETRIES** times.
- **rreq\_flag**: RREQ in AODV draft 10 has flag **J**, **R** and **G**. **J** and **R** flags are reserved for multicast. **G** (**Gratuitous RREP**) flag may be used when bi-directional path is desirable for upper layers. When the **G** flag of a RREQ is set, an intermediate host that tries to send back a reply on behalf of the requested destination will also send a

RREP on behalf of the originator of RREQ to the destination.

- **use\_hello\_msg**: This is a boolean switch of using **Hello Message**, and the default value is **false**. When the link layer ACK as in MAC802.11 is not available, this option can be set to be true, and the host will periodically send out RREP for itself with TTL set to 1. Other hosts can use the Hello Message to maintain their views of local connectivity. If more than **ALLOWED\_HELLO\_LOSS** Hello Messages from a host are lost, the link to that host is assumed to be lost.
- **use\_rrep\_ack**: This is a boolean switch of using **RREP\_ACK**, and the default value is **false**. **RREP\_ACK** is used to detect uni-directional link when link layer ACK is not available. When a host sends out a RREP over a link that is suspected to be uni-directional, it sets the **A** flag and asks for an acknowledgement from the receiver of the RREP. If no **RREP\_ACK** is received within **NODE\_TRAVERSAL\_TIME**, the link is assumed to be uni-directional for a short period of time.
- **show\_report**: This is a switch of outputting a summary report at the end of the simulation. The default value is **false**. The summary includes:
  - numRequestSent* - Number of RREQ sent (including retries).
  - numRequestRelay* - Number of RREQ relayed.
  - numAbort* - Number of aborted request attempts.
  - numReplySent* - Number of RREP sent (Hello Message not included).
  - numReplyRelay* - Number of Reply relayed.
  - numRerrSent* - Number of RERR sent.
  - numRerrRelay* - Number of RERR relayed.
  - numRrepAckSent* - Number of RREP-ACK sent.
  - numDataSent* - Data (non-aodv pkts) Sent as the source of the route.
  - numDataTxed* - Data (non-aodv pkts) Forwarded.
  - numDataReceived* - Data (non-aodv pkts) Received as the destination of the route.
  - numPacketsDropped* - Number of packets dropped by aodv.
  - numDataPktDroppedByMAC* - Number of data (non-aodv) packets dropped by MAC
  - numBrokenLinks* - Number of link breaks (route expire not included.)

*numHelloMsgSent* - Number of Hello messages sent.

*numBootup* - Number of reboots (not including the initial one.)

*numBootSucc* - Number of reboots that finished wait\_reboot period.

- **show\_rt\_table\_change**: This is a boolean switch and the default value is **false**. When it's set to be true, the host will output some information when 1) a data (non-aodv) packet is initialized from this host and pushed to IP; 2) there is any change in the AODV routing table.
- **show\_stat\_trace**: This is a boolean switch and the default values is **false**. When it's set to be true, the host outputs the statistical information every second. There will also be additional output when 1) a reply has been received for a RREQ initialized from this host; 2) a route is removed from the kernel forwarding table.
- **show\_error**: This is a boolean switch and the default value is **false**. When it's set to be true, the host prints out "error" messages for failing to send out some control packets.
- **active\_route\_timeout**: It specifies the lifetime in second of an active route. The default value is 10.0 second. If the link layer layer ACK is not available, This value should be adjusted according to the draft.
- **net\_diameter**: This is an estimation of the maximum hops between any two hosts in the network. The default value is 35.
- **rreq\_retries**: It specifies the maximum number of network-wide RREQ tries. (TTL == NET\_DIAMETER). The default value is 2.
- **node\_traversal\_time** This is an estimation of the delay of one-hop roundtrip time of a packet in millisecond. The default is 40 ms. Using a too small value may result in route loops.
- **pkt\_priority** It specifies the priority of the AODV packets. It's used by the MAC layer that supports multiple priority queues. For MAC802.11, the valid range is [0, 1, 2]. 0 is the highest priority, and 2 is the default value.

```

session [
    name "aodv" use "routing.aodv_sim.swan-aodv-session"
    iface 0          # control interface 0
    iface 1          # control interface 1
    pkt_priority 0   # control pkt priority
    net_diameter 40  # maximum hop distance between two hosts
    show_report true # output summary at the end of simulation
]

```

Figure 16: An Example DML Script for SWAN AODV Session

- **rreq\_record\_time**: When a host processes an RREQ, it keeps a record about the originator IP and the rreq ID for RREQ\_RECORD\_TIME second so that the same request won't be processed again. AODV draft used PATH\_DISCOVERY\_TIME for it. When testing with MAC802.11, we found out that it's often not long enough and may result in route loops under heavy traffic. Thus we re-used the RREQ\_RECORD\_TIME that has been used in earlier draft. The default value is computed based on NODE\_TRAVERSAL\_TIME. It has a maximum value of 50 second.
- **allowed\_hello\_loss**: It specifies the maximum number of allowed hello losses. The default value is 2.
- **hello\_interval**: The time interval in second of broadcasting Hello Messages. The default value is 1.0 second.

Figure 16 shows an example of DML configuration of SWAN AODV when the MAC layer is MAC802.11 (not shown here). By default, it doesn't use **Hello Message** or **RREP\_ACK**, but **Expanding Ring Search** is applied, and AODV controls all available interfaces. The control packets uses the highest priority 0, set the **NET\_DIAMETER** to be 40, and at the end a summary is reported for each host.

### 6.6.3 Related Files

- Header Files

include/routing/aodv/sim-aodv/swan-aodv-session.h

include/routing/aodv/sim-aodv/swan-aodv-header.h

include/routing/aodv/sim-aodv/swan-aodv-routing-table.h

- Source Files

src/routing/aodv/sim-aodv/swan-aodv-session.cc

src/routing/aodv/sim-aodv/swan-aodv-timeout.cc

src/routing/aodv/sim-aodv/swan-aodv-header.cc

src/routing/aodv/sim-aodv/swan-aodv-routing-table.cc

## 6.7 Dynamic Source Routing Protocol

### 6.7.1 Overview of the DSR Model

The DSR (Dynamic Source Routing) model in *SWAN* is implemented according to the DSR Internet Draft 7 [?]. It is a reactive routing protocol for ad-hoc wireless networks, similar to AODV. The two chief differences between DSR and AODV are that DSR copies the entire source route into the header for each data packet transmitted, and that it often piggybacks different options on a single header. The protocol inserts a DSR header between the IP header and the data packet to route the packet across multiple hops. All routing work happens on-demand with *Route Discovery*—routes and neighbors are only discovered by a node when it has data it needs to transmit—so nodes may join, move around, and leave the network with no setup cost. However, data packets are often sent down broken links due to outdated route tables, necessitating *Route Maintenance*.

DSR *options* specify the type of control data stored in the header. The RREQ (Route Request) is a broadcast option that propagates across the network in all directions, building up routes that fan out from the Route Discovery initiator in search of a particular target node. As each node receives the RREQ packet, it adds its own address to the route stored in the header and rebroadcasts the RREQ.

When an RREQ option is received by the target destination, the target node replies with an RREP (Route Reply) option that contains the source route built-up by the RREQ. In an 802.11 network, the RREP is unicast back to the initiator of the Route Discovery on the reverse of the source route to ensure that the intermediate links are bidirectional.

When the initiator receives the RREP, it caches the route and then transmits the data packet awaiting this route. As long as the route remains in the Route Cache, further packets pushed to DSR, e.g. from the Application layer, will be immediately transmitted on this route.

Data packets are unicast using a Source Route option, which includes the entire route of IP addresses for intermediate nodes. Each intermediate node checks the header to determine the next node and retransmits the packet.

In an ad-hoc network, instability is expected because of such factors as interference and mobility. Therefore, for every packet transmitted or retransmitted in unicast mode, a node must receive next-hop confirmation. In an 802.11 network, the MAC layer provides this confirmation after it receives its own confirmation signal (ACK) <sup>2</sup>. While waiting for this confirmation, transmitted packets are stored in the Maintenance Buffer. If confirmation is not received after several retransmission attempts, Route Maintenance is invoked (either by a timer or by notification from the MAC layer of next-hop failure). All packets awaiting confirmation to this lost node are removed from the Maintenance Buffer <sup>3</sup>, and if other routes to their respective destinations exist in this node's Route Cache, these packets are salvaged on the alternate routes. The broken link is removed from the Route Cache and an RERR (Route Error) option is transmitted back to the originator of each failed packet (but limited to one RERR per unique source). Upon receipt of an RERR option, a node removes the bad link from its Route Cache; the RERR reaches the originator of the bad route by using either a Source Route option or an RREQ option (if the node reporting the error does not have a route to the originator of the bad route).

These four options may be piggybacked on one another in many combinations. The most common instance is the RREP option routed back to the initiator of Route Discovery piggybacked on a Source Route option. The only combination specifically not allowed by DSR is an RREQ and a Source Route, because one must be broadcast and the other unicast. Each option gets processed by every node that receives the packet.

---

<sup>2</sup>*SWAN*'s DSR implementation also provides DSR-level acknowledgement with ACK REQ and ACK header options. These will be used automatically if DSR detects that the MAC layer does not provide next-hop confirmation. However, they have not been tested extensively

<sup>3</sup>In an 802.11 network, only one packet is transmitted at a time until either confirmation is received, or until several retransmission attempts fail. All other outbound packets are sitting in the MAC buffer waiting for a transmission attempt (the MAC buffer in *SWAN* is equivalent to the Network Interface Queue specified by the DSR draft in section 4.5). Failure of one packet must stop all MAC buffered packets to the same next-hop address from being transmitted. Note that in *SWAN*'s implementation, DSR does not have this level of control over the MAC buffer, so only one packet to a particular next-hop may be pushed to the MAC layer at a time, and subsequent packets are stored only in the Maintenance Buffer. Once this first packet has been successfully sent, all the others are pushed to the MAC layer and removed from the Maintenance Buffer, because confirmation of one is assumed to equal confirmation of all.

*SWAN* 's implementation of DSR supports many optimizations: `Promiscuous Caching of Overheard Routing Information`, `Replying to Route Requests using Cached Routes`, `Route Request Hop Limits`, `Packet Salvaging`, `Automatic Route Shortening`, `Preventing Route Reply Storms`, and `Multiple Interfaces`. Please see the appropriate sections in the DSR draft [?] for further information. Some of these optimizations are customizable from the DML script. Also, a link-based table is used for the Route Cache, so that individual links are stored, rather than entire routes. Dijkstra's shortest path algorithm is used to find the best route. The cost of a link is fixed at 1, except external links cost 100 to so that routes internal to the DSR network are always preferred. There is also a simple route-based table, which can be used by commenting out a macro flag in the file `/include/routing/dsr/dsr-globals.h`.

### 6.7.2 DML Configuration of DSR

- `send_timeout`: The delay time, in seconds, before data waiting for Route Discovery to find a route to its requested destination is silently discarded from the Send Buffer. Default is 30 seconds.
- `req_timeout`: The delay time, in milliseconds, before a propagating Route Request is retransmitted if no Route Reply has been received. Subsequent retransmissions will occur with exponential backoff. Default is 500 milliseconds.
- `req_nonprop_timeout`: The delay time, in milliseconds, before a nonpropagating Route Request (i.e. one with a time to live of 1, so that the initiator's neighbors do not retransmit the Request) is retransmit with the `req_timeout` delay. Default is 30 milliseconds.
- `maint_timeout`: The delay time, in milliseconds, before a packet is retransmitted or a Route Error is generated. This should be set to 0 (i.e. unused) if the MAC layer provides next-hop confirmation. Default is 0.
- `maint_rexmt`: This is the integer value of the number of times the Maintenance buffer will retransmit a packet before salvaging or discarding it. A message should be re-

transmitted at least 2 times, according to the DSR draft (section 9), including re-transmissions by the MAC layer. 802.11 retransmits packets 4 or 7 times, depending on size, so the default for this count is 0.

- **jitter\_timeout**: The time, in milliseconds, upon which a random delay value is calculated before a packet is pushed down to the MAC layer for transmission. Used to help avoid collisions and to prevent Route Reply storms. Default is 10 milliseconds.
- **route\_timeout**: The delay time, in seconds, before a route or link is removed from the Route Cache. This value gets reset each time a route or link is chosen as the best available for an outgoing packet. Default is 300 seconds.
- **gratrep\_timeout**: The delay time, in seconds, before a node forgets that it has sent a Gratuitous Route Reply, and hence can send another to the same node. Default is 1 second.
- **blacklist\_timeout**: The delay time, in seconds, before a node is removed from the Blacklist, which is used to avoid unidirectional links; if **bi\_directional\_required** is set to **false**, the Blacklist will not be used. Default is 25 seconds.
- **bi\_directional\_required**: This is a boolean switch (default **true**): if set to **true**, the DSR session will be required to use only bidirectional links. All Route Replies will be routed back to the Route Discovery initiator on the reverse of the built-up source route to test it for bidirectionality. If set to **false**, DSR allows unidirectional links by routing next-hop confirmation acknowledgments over an indirect path. Some MAC protocols, including 802.11, require links to be bidirectional in order to function properly.
- **bi\_directional\_frequent**: This is a boolean switch (default **true**): if set to **true**, the DSR session will assume most links in the network are bidirectional and will cache discovered routes in both directions. If set to **false**, the DSR session will assume that there are many unidirectional links in the network and hence will not cache the reverse of routes because these are not necessarily valid. Note that if

`bi_directional_required` is set to `true`, this option will be ignored and all reverse routes will be cached because they must be bidirectional to be cached at all.

- `promiscuous`: This is a boolean switch (default `false`): if set to `true`, all packets physically received by a node will reach the DSR session, regardless of the IP header's next-hop address. DSR will process promiscuously-received packets for several optimizations, including `Caching Overheard Routing Information`, `Automatic Route Shortening`, and `Preventing Route Reply Storms`. If set to `false`, the DSR session of a node will only receive broadcast packets and those packets specifically addressed to it.
- `salvage`: This is a boolean switch (default `true`) to prevent DSR from salvaging packets while performing Route Maintenance.
- `max_ttl`: This is the integer value of the maximum/standard time-to-live of an RREQ, in number of hops (default 50). Once the RREQ packet has been forwarded by `max_ttl` nodes, it will be discarded. Note that *SWAN* allows a maximum TTL of 127.
- `init_ttl`: This is the integer value of the initial time-to-live of an RREQ, in number of hops (default is equal to `max_ttl`). If no RREP is received, the first time the RREQ is rebroadcast its `time-to-live` field will be increased to `max_ttl`.
- `seed`: An integer from which to derive a random number generator. Using the same seed gives the same random numbers for each run. Random numbers are used for initiating the RREQ sequence ID's at each node and for jitter delays. If a value is not given, DSR uses DaSSF's random number generator.
- `show_report`: This is a boolean switch (default `false`): if set to `true`, at the end of the simulation each DSR session will output a summary report, which includes:
  - number of rreq sent.*
  - number of rreq rebroadcast.*
  - number of rreq received.*
  - number of rreq aborted.*
  - number of rrep sent.*

*number of grat rrep sent.*

*number of rrep received.*

*number of rerr sent.*

*number of rerr received.*

*number of ack req sent.*

*number of ack sent.*

*number of source route sent.*

*number of source route received.*

*number of packets received, not including promiscuous packets.*

*number of packets received promiscuously.*

*number of packets transmitted, includes retransmissions.*

*number of packets dropped, incremented when the Send Table times out before transmitting a packet and when the Maintenance Buffer times out before receiving next-hop acknowledgement and cannot salvage a packet.*

*number of salvage attempts, including multiple attempts to salvage a single packet.*

*number of broken links, incremented by 2 if both a-b and b-a are removed.*

*number of bootups, incremented after a node shuts down and restarts.*

*number of routing packets, i.e. packets transmitted without a payload.*

*number of routing bytes, a total count of the size of each DSR header transmitted or retransmitted.*

*average hop count of all source routes created by this node. If a packet is salvaged, the number of hops not yet traversed downstream of a broken link is deleted from this statistic, and the number of hops in the alternate route is added; note that this will give an incorrect statistic for individual nodes when any node besides the originator salvages a packet, but a correct global statistic.*

Figure 17 shows an example of DML configuration of DSR when the MAC layer is MAC802.11 (not shown here).

```
session [  
    name "dsr" use "routing.dsr-session"  
    seed 12345  
    jitter_timeout 0 # because MAC randomizes transmission time  
    promiscuous true # examine packets received in promiscuous mode  
    init_ttl 1      # RREQs first sent only to neighbors  
    show_report true # show statistics at the end  
]
```

Figure 17: An Example DML Script for a SWAN DSR Session

### 6.7.3 Related Files

- Header Files

include/routing/dsr/dsr-globals.h

include/routing/dsr/dsr-header.h

include/routing/dsr/dsr-session.h

include/routing/dsr/dsr-tables.h

- Source Files

include/routing/dsr/dsr-header.cc

include/routing/dsr/dsr-session.cc

include/routing/dsr/dsr-tables.cc

## 6.8 Test Application Sessions

### 6.8.1 Overview of Test Application Sessions

As of writing this document, three simple test applications have been developed in *SWAN* to test other protocols: `rand-app-session`, `sess-app-session`, and `ns-app-session`. In fact, `sess-app-session` is a more general form of `rand-app-session`, so `rand-app-session` is not individually described. `Ns-app-session` performs much like the other two, but takes as input an NS-2 traffic file generated by CMU's utility `cbrgen.tcl` (available with the NS-2 all-in-one package); it's one difference in functionality is that `ns-app-session` allows multiple sessions at one time from one host.

The test application session is a very simple session. It chooses a start time with some randomness, and start to send out packets to one of the peers specified at a given rate. After some number of packets, another peer is selected as the destination. This process repeats until the end of the simulation.

### 6.8.2 DML Configuration of Test Application Session

- `peer` or `peer_range`: It's required that the peers need to be specified in the DML. The peers can be specified with either but not both of these two attributes. `peer` can be used more than once to specify multiple peers. For each of them, one needs to specify `netid`, `hostid` and `iface` (interface ID). The peer IP address can be constructed out of the three attributes provided above. When using `peer_range`, instead of using `hostid`, one specifies `hostid_min` and `hostid_max`, assuming that they all use the same `netid` and `iface`. The user can use either way of specifying the peers.
- `packet_size`: The test session uses fixed size packets, and this attribute specifies the packet size in bytes. The default value is 1024.
- `iat`: Inter-Arrival-Time of the packets. The default value is 1.0 second. This is in fact the mean value of the inter arrival time.
- `start_phase`: When the session starts, it will pick a start time from 0 to `start_phase`

second randomly. The default value is 2.0 second.

- **fixed\_rate**: This is a boolean switch and the default is **true**. If it's set to be false, the real inter arrival time of the packets in the simulation will be sampled from an exponential distribution using **iat** as the mean. By default, **iat** is the real inter arrival time.
- **session\_len**: It specifies the session length in the number of packets. After this number of packets have been sent to a destination, the host will select another destination from the available peers. By default, this value is  $2^{31} - 1$ .
- **show\_report**: This is a boolean switch of outputting a summary at the end of the simulation, and the default value is **false**. The summary includes:
  - #sent*: number of sent packets.
  - #rcvd*: number of received packets.
  - #sent\_fail*: number of packets the local host failed to send out. Notice that this doesn't count those packets that are dropped by some other hosts in the middle of the path.
  - #ave time*: average end-to-end delay time for packet delivery.
- **show\_stat\_trace**: This is a boolean switch of outputting the statistics every simulated second during the simulation. The content is the same as in the final summary. It also print out the chosen peer when it changes the destination. The default value of this switch is **false**.

Figure 18 shows an example of the DML configuration of a sess-app-session. It chooses its destination from the hosts with ID range [10 20]. The packet size is 512 bytes, and the sending rate is not fixed. The session length is 100 packets, and a summary will be printed out at the end of the simulation.

### 6.8.3 DML Configuration of NS-2 Application Session

- **nsfile**: Specify a filename (with path relative to the *SWAN* executable) that contains a traffic model generated by CMU's TCL script `cbrgen.tcl`. To function properly

```

session [
    name "app" use "tstapp.sess-app-session"
    peer_range [
        netid 1
        hostid_min 10 hostid_max 20
        iface 0
    ]
    iat 0.5          # iat is 0.5 second
    fixed_rate false # not fixed rate
    packet_size 512 # packet size
    session_len 100 # session length
    show_report true # print out summary at the end.
]

```

Figure 18: An Example DML Script for SessAppSession

with *SWAN*, *cbrgen.tcl* must be given only the CBR option (`-type cbr`), not the TCP option. Also, note that a bug in the *cbrgen.tcl* utility creates  $x + 1$  nodes given the parameter `-nn x`, numbered 0 -  $x$ . In the host `id_range [...]` attribute of the *SWAN* DML, the nodes must be numbered `id_range [ from 1 to x + 1 ]`.

- `netid`: The network ID, which should match the ID specified in `network [ netid x ]`. Used to construct IP addresses.
- `iface`: The interface ID, which should match the ID specified in `interface [ iface x ]`. Used to construct IP addresses.
- `show_stat_trace`: Equivalent to the other test applications' option.
- `show_report`: Equivalent to the other test applications' option.

Figure 18 shows an example of the DML configuration of a ns-app-session.

```
session [  
    name "app" use "tstapp.ns-app-session"  
    nsfile "traffic"  
    netid 1  
    iface 0  
    show_report true  
]
```

Figure 19: An Example DML Script for NSAppSession

#### 6.8.4 Related Files

- Header files

include/tstapp/sess-app-session.h

include/tstapp/rand-app-session.h

include/tstapp/ns-app-session.h

- Source files

src/tstapp/sess-app-session.cc

src/tstapp/rand-app-header.cc

src/tstapp/ns-app-session.cc

## References

- [1] L. Bajaj, M. Takai, R. Ahuja, K. Tang, R. Bagrodia, and M. Gerla. GloMoSim: A Scalable Network Simulation Environment. Technical Report 990027, UCLA, Computer Science Department, 1999.
- [2] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.
- [3] J. H. Cowie. *Scalable Simulation Framework API reference manual, Version 1.0*, 1999.
- [4] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the Global Internet. *Computing in Science & Engineering*, 1(1):42–50, 1999.
- [5] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
- [6] IEEE Computer Society, L. M. S. *Wireless LAN medium access control (MAC) and physical layer (PHY) specification*, 1997.
- [7] A. Kamerman and L. Monteban. WaveLAN-II: A high-performance wireless LAN for the unlicensed band. *Bell Labs Technical Journal*, 2(3):118–33, Summer 1997.
- [8] J. Liu and D. M. Nicol. *DaSSF 3.1 User’s Manual*, April 2001.
- [9] Mobile Ad Hoc Networking Working Group, IETF. *Ad hoc On-Demand Distance Vector (AODV) Routing (draft 10)*, June 2002.
- [10] Charles E. Perkins, Elizabeth M. Belding-Royer, and Samir Das. Ad hoc on demand distance vector (aodv) routing. IETF Internet draft (Work in Progress), draft-ietf-manet-aodv-10.txt, November 2001.
- [11] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: a library for Parallel Simulation of Large-scale Wireless Networks. Proceedings of the 12th Workshop on Parallel and Distributed Simulations – PADS ’98, Banff, Alberta, Canada, May 1998.