# User's and developer's manual of BitSimulator

BitSimulator's authors

April 24, 2025

## Contents



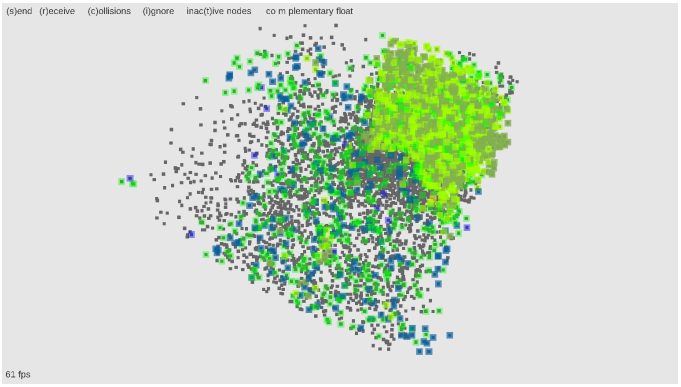


Figure 1: Screenshot of VisualTracer in 3D, showing nodes in sending (blue) or receiving (green) state.

# 1 Introduction

BitSimulator is a fast wireless 3D nanonetwork bit-level simulator for routing and transport levels. It comes with a companion program, VisualTracer, allowing to show simulation results on a 2D or 3D map.

You need to read BitSimulator's article [4] too, which presents the simulator from another point of view. TODO: integrate all its information in this document



# 2 Features and limitations

Some features:

- It can simulate and visualise 2D and 3D environments, cf. figure 1.
- Given that it deals only with nanocommunications, its design is simple and efficient. Tens of thousands of nodes with a density of hundreds of nodes can be simulated on a laptop.
- Propagation delay: packet arrival time on a node depends on its distance from the sender (extremely important given the tiny time length of TS-OOK pulses).
- Computation of collisions uses the TS-OOK model by checking the actual bit value of each packet currently being received at each node.
- While it allows for multiple concurrent receptions on a given node, it also acknowledges that there is a limit to their number (caused by hardware processing power or buffer space for instance); in particular, this parameter has a strong impact on the behaviour of upper

layer protocols, as packets otherwise correct can be completely ignored.

- The simulator's behaviour is deterministic, and uses several random number generator instances with their own seed.
- Nodes can sleep (duty cycling) with a cycle equal to $T_s$.
- Two propagation models are provided: the well-known disc model, and shadowing, where the border is blurred, i.e. random losses can occur at the border of the communication range. The communication range used by nodes can change runtime.
- It implements several protocols: SLR routing protocol, probabilistic flooding, backoff flooding, and others.
- It comes with an extremely useful visualisation tool, VisualTracer.
- Use of XML configuration files along with command line parameters allows easy batch runs. Changing parameters and seeds allows to get statistical results and explore the effect of various parameters.

Some limitations are the following:

- Nodes are static
- There is no energy model
- Antennas are omnidirectional (both for sending and receiving)
- It simulates only one kind of networks: nanonetworks, e.g. it cannot simulate a nanonetwork–IP mix of networks.
- A node always sends packets *sequentially* (one after the other), even if it schedules to send several packets at the same time.

# 3 Scenario specification

The scenario is specified in `scenario.xml` file and through command line options. The latter have precedence. All the XML elements and command line options are discussed in their sections.

## 3.1 XML configuration file

### 3.1.1 Hello World example

```
<scenario>
  <world sizeX_nm="6000000" sizeY_nm="0" sizeZ_nm="6000000">
    <genericNodes count="1000" positionRNGSeed="1"/>
  </world>

  <modulation>
    <ts-ook pulseDuration_fs="100" defaultBeta="1000"
            defaultCommRange_nm="500000"
            maxConcurrentReceptions="10"
            minIntervalBetweenSends="0"
            minIntervalBetweenReceiveAndSend="0"/>
  </modulation>

  <routing defaultBackoffWindow="10000" backoffRNGSeed="1">
    <PureFloodingRouting/>
  </routing>

  <applications>
    <cbr flowId="0" srcId="3" dstId="10" port="1"
         packetSize="1000" repetitions="3" interval_ns="100"
         startTime_ns="1000000"/>
  </applications>

  <log/>
</scenario>
```

### 3.1.2 All the elements

Here are all the optional elements, with all their attributes.

```
<name>Pure flooding test</name>

<description>Pure flooding test description</description>

<world>
  <node id="0" posX_nm="100" posY_nm="0" posZ_nm="100"/>
  <area shape="rectangle" x_nm="3000000" y_nm="0" z_nm="3000000"
        sizeX_nm="3000000" sizeY_nm="0" sizeZ_nm="3000000"
        distribution="uniform" nodesCount="100" positionRNGSeed="5">
    <area shape="rectangle" x_nm="0" y_nm="0" z_nm="0"
          sizeX_nm="3000000" sizeY_nm="0" sizeZ_nm="3000000"
          distribution="normal" meanX_nm="3000000" meanY_nm="0"
          meanZ_nm="3000000" deviationX_nm="500000"
          deviationY_nm="0" deviationZ_nm="500000"
          nodesCount="2000" positionRNGSeed="50"/>
    <area shape="ellipse" x_nm="1000000" y_nm="0" z_nm="1000000"
          sizeX_nm="2000000" sizeY_nm="0" sizeZ_nm="2000000"
          distribution="uniform" nodesCount="1000"/>
    <area shape="rectangleHole" x_nm="100000" y_nm="0" z_nm="100000"
          sizeX_nm="1000000" sizeY_nm="0" sizeZ_nm="100000"/>
  </area>
</world>

<modulation>
  <ts-ook ... commRangeStandardDeviation_nm="50000"/>
</modulation>

<routing>
  <!-- one of the following routing agents -->
  <PureFloodingRouting/>
  <ProbaFloodingRouting probability=".2"/>
  <BackoffFloodingRouting/>
  <RayTracingRouting/>
  <SLRRouting commRangeSetup_nm="250000"
              useCounterBasedForwarding="true"
              useDeviation="true"
              firstBeaconStartTime_ns="200000"
              intervalBetweenBeaconStart_fs="200000000000"
              anchor1id="0" anchor2id="2" anchor3id="5"/>
  ...
</routing>

<applications>
  <DEDeN enabled="true" startTime_fs="1000000"/>
  <cbr ... beta="1000"/>
</applications>

<log prefix="new"/>
```

## 3.2 Command line

The command line options for BitSimulator are the following:

```
[--scenarioFile <string>] [-D <string>] [--prefix
<string>] [--routing <string>] [--neighboursList] [--ecc
<int>] [--genericNodesRNGSeed <int>] [--genericNodesCount
<int>] [--defaultBeta <long>] [--defaultBackoffWindow
<int>] [--backoffRNGSeed <int>] [--nodePositionNoise
<int>] [--deden] [--dedenRNGSeed <int>]
[--dedenLoadEstimationFromFile] [--awakenDuration <int>]
[--awakenNodes <int>] [--sleepRNGSeed <int>]
[--disableLogsAtRoutingLevel] [--disableLogsAtNodeLevel]
[--backoffMultiplier <float>] [--backoffRedundancy <int>]
[--hcdPathWidth <int>] [--rangeSmall <distance_t>]
[--rangeBig <distance_t>] [--probability <float>]
```

```
[--slrPathWidth <int>] [--loadSLRPositionsFromFile]
[--D1DoNotSimulateFlood] [--D1MinPacketsForTermination
<int>] [--D1RemainingFreeRounds <int>] [--D1MaxRound
<int>] [--D1MaxTic <int>] [--D1ErrorMax <float>]
[--D1GrowRate <float>] [--payloadRNGSeed <int>] [-g] [--]
[--version] [-h] <string>
```

Execute `bitsimulator -h` for more information.

By default, BitSimulator looks for `scenario.xml` file in the current directory. The file name and the directory can be changed with a file name at the end and `-D` parameters.

# 4 Simulation information

## 4.1 World specification

### 4.1.1 Node deployment

The world (network) is rectangular, its parameters being specified in the XML file or in the command line. The world can be 2D (when the `0y` dimension is 0) or 3D. Nodes must be inside, and can be created in three manners:

- manually, by specifying their position, in the XML file, e.g. `<node id="0" posX_nm="100" posY_nm="0" posZ_nm="100"/>`.
- randomly inside the whole network, by specifying their number in the XML file or in the command line, using a uniform distribution, e.g. `<genericNodes count="1000" positionRNGSeed="1"/>`.
- using areas, see below.

Note that in the XML file the random placement must come after all the manual placements (for id assignment reasons).

Also, no check is done if two nodes are in the same position, since this is not a problem in the simulator.

`--genericNodesRNGSeed` is used for the random placement of the nodes.

### 4.1.2 Specifying heterogeneity

Allowing the specification of only one distribution is not sufficient in some cases. For example, SLR anchors should be located at the border of the network; however, in distributions like Gauss or Poisson the borders have few nodes, and beacons generated by anchors on borders might have too few neighbours and not propagate through the network. Also, empty zones cannot be specified through a distribution. Hybrid scenarios, like a homogeneous zone, an empty one, and an attenuation one, cannot be specified either.

BitSimulator is flexible in this respect, allowing to specify heterogeneous densities around three notions: areas, distributions, and holes:

- An *area* can be a rectangle (rectangular cuboid in 3D) or an ellipse (ellipsoid in 3D). The XML file can specify several areas. When several areas are present in a region, they have cumulative effect (nodes from all of those areas are placed in the region). A nested area has its coordinates expressed relatively to its parent area.

```
  |--------|------------|---------------------|----------->
  0      200*10^9     500*10^9              1000*10^9
DEDeN  SLR setup  Sleep enabling           CBR start
```

Figure 2: Default order of protocol execution during simulation (times are in fs). The starting times can be modified in the XML file.

- Each area has a *distribution* attached to it, along with its parameters if any. The implemented distributions are: uniform (or homogeneous) and Gauss (or normal, in rectangle only). No node will be placed outside the area of the distribution. No node will be placed outside the world.
- Rectangular *holes*, where no node exists, can be specified. The place where a hole is specified inside the `<area>` element matters: a rectangleHole puts a hole in its parent shape, but only in the siblings that have been defined *before* the rectangleHole.

An example of network density is to use a homogeneous distribution everywhere, and above it a normal law. "Borders" in terms of density, generated by two adjacent areas, can be attenuated by specifying a gradient between the two areas for example.

### 4.1.3 Implementation details

**World** `World` is the class in charge of the world's parameters such as the size and all the placement of all nodes. It also initialises routing agents, nodes etc., and handles the neighbour list (section 4.2.4).

Nodes can be placed in the closed interval `[0,size]`, where `size` is the size of the world in that direction.

**Node** Nodes are responsible for the transmissions and the reception code. They can store their neighbours list in the static mode.

**SleepingNode** This class derives from the `Node` class. Sleeping (duty cycling) nodes are able to sleep (to save resources); while sleeping, all the packets received by it are discarded. Sleeping nodes sleep in a cyclic way for each $T_s$.

The duration is specified in two manners, via command line options: either absolute (`--awakenDuration`), or inferred from the average number of awake neighbours desired (`--awakenNodes`). For the latter option, the duration of each node will be different, because it depends on either the number of neighbours, or its estimated neighbours (if DEDeN is used). The sleeping mechanism is presented in [6, section III].

Once the duration is known, the `--sleepRNGSeed` command line parameter sets the seed for the RNG used to set the beginning of the sleeping period for sleeping nodes.

Nodes do not sleep at the beginning of simulation, but starting from $500 * 10^9$ fs. This is to allow the setup phase of some protocols to run in non sleeping mode. The overall order of algorithms is shown in figure 2; of course, they are executed only when enabled in the scenario.

## 4.2 Modulation specification

Only one protocol is implemented, TS-OOK. This means that BitSimulator cannot simulate a nanonetwork–IP or

nanonetwork–Bluetooth network for example.

### 4.2.1 TS-OOK main concepts

TS-OOK [5] is a pulse-based modulation. For more information about TS-OOK see for example [4, section 2.1].

### 4.2.2 Sending and receiving packets

Sending is done sequentially, even if several packets are scheduled to be sent. Attributes `minIntervalBetweenSends` and `minIntervalBetweenReceiveAndSend` specify the minimum time between the beginning of two consecutive sent packets, and between the beginning and the reception of the last packet, respectively, for any nodes. NB: these attributes do not work reliably currently, let them be 0!!

All the nodes have a maximum concurrent receptions (MCR) value, configurable in XML file. An MCR of `n` means that nodes can process at most `n` packets in parallel, so it ignores all the other packets received at the same time as the `n` packets being processed. More information is given in [2, section 3].

### 4.2.3 Propagation model

BitSimulator proposes two propagation models, both statistical (unlike COMSOL Multiphysics for example, where reception considers the characteristics of the space): the free space model (also known as all or nothing model, or disc model), and an improved model, known as shadowing in other simulators.

By default, a node receives a packet iff the distance between it and sender is smaller than or equal to the communication range. The range is configurable in the XML file.

The communication range can be modified runtime, using `setCommunicationRange` function in a node or in a packet. However, note that it cannot be greater than the initial communication range, specified in the XML file. It is important to note that, when using the node function, the range used is checked when the packet is injected in the network, which is not necessarily the same as when the packet is sent by the application in case several packets wait in node's queue (remember that packets are sent sequentially). This is why this function exists in a packet too. To conclude, it is safer to use node for the beginning of the simulation, and packet when the range changes during the simulation. Contact us for further details.

When using the `commRangeStandardDeviation_nm` attribute in the XML file (by default, it is 0), BitSimulator uses a propagation model which blurs the border, i.e. allows random losses near the border, depending on the distance, as shown in figure 3. The model, also called *shadowing* in ns2 network simulator, uses the normal distribution, is moved to left 3*stddev (3 is chosen so that the probability beyond the communication range be very small, 0.5%), and is cut (changes sharply) before commrange–6*stddev (i.e. *all* packets are received for a distance less than this value, which represents 0.5% of them) and beyond communication range (i.e. no packet is received by nodes at distance greater than or equal to the communication range, which represents 0.5% of them). In this model, losses change over time, to avoid that the same node systematically losses or

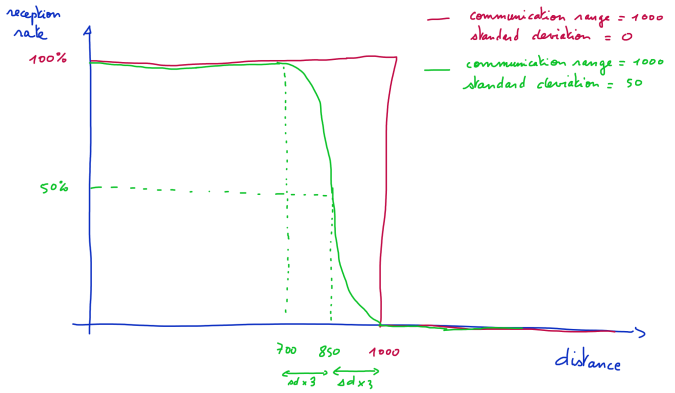


Figure 3: BitSimulator uses an improved disc model for propagation.

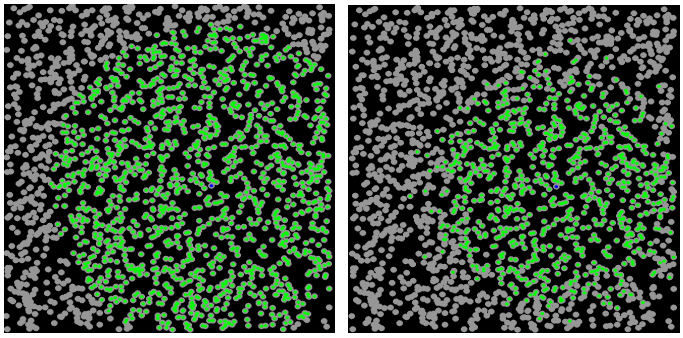

Figure 4: The classical disc model on the left, and BitSimulator model on the right; the blue node sent a packet, and the green nodes received a packet.

receives packets. Figure 4 clearly shows the difference compared to the disc model; it can also be noticed that some nodes *closer* to the central node do not receive the packet, while others are further and do receive it.

### 4.2.4 Neighbours' list computation

Given that links are wireless, messages are broadcasted to all the neighbours of sending node. The lists of all neighbours of each node can be computed in two ways:

- dynamically: the simulator (re)computes it each time a node sends a message.
- statically: the simulator computes each node's neighbourhood at the beginning and stores it in memory for the duration of the simulation; of course, this mode consumes a lot of memory, and cannot be used for mobile nodes

The best mode depends on the scenario, for instance in a broadcast (flooding) all the nodes send at least a message, hence static mode is identical or faster; in SLR point-to-point, numerous nodes do not send any message, hence the dynamic mode might be faster.

The default mode is dynamic. To use the static mode, use the `--neighboursList` command line option.

Note that the results of the two modes could be different, due to different sorting order of neighbour's list due to random number usage.

In the static mode, when executing several times a given simulation scenario, it would be faster to compute once for

all the neighbours' list and load it each time the simulator runs; this feature is not yet implemented.

#### 4.2.5 Packet collisions

In BitSimulator, the collision model has some real aspects, and some other statistic. It is real in the sense that the values of the bits are considered, and is statistic because two bits are considered as collision if the time of reception, whose length is $T_p$, intersect.

We define two bits arriving at the same time at a receiver as collided (or overlapped) bits. This receiving time depends on bit sending timing and receiver position. This overlapping is destructive for a receiving bit (changes its value, and makes the bit corrupted) only when this bit is 0, and at the same time it receives a 1 bit from another flow. For random bits for example, this means that only one quarter of overlappings lead to corruption. More information is given in [4].

In BitSimulator, collisions are computed on each node, on each bit of each frame. Whenever an `endReceiveEvent` is processed, the concerned packet is checked against all the other packets that are being received by the same node. It also runs on `endSendPacketEvent`, which checks against all the packets being received and mark them if there is collision. The collision checking does not generate any event, this process is integrated in the reception process.

In real networks, to find out whether a packet is altered or not, packets must include a checksum. On the contrary, in BitSimulator, if the number of corrupted bits exceeds a given threshold, the packet is dropped. This threshold is 0 by default, but can be set with `--ecc` command line option, which allows to test error correction codes for example.

### 4.3 Routing/network layer specification

Available protocols are found in `src/agents` directory. Only one protocol can be used in a simulation.

#### 4.3.1 Pure flooding

The routing agent is `PureFloodingRouting`. Note that each node stores the max id (and not the whole list of ids seen) seen to decide whether it forwards a packet or not. A backoff window is used for each forwarding, given by `defaultBackoffWindow` attribute, which also uses `backoffRNGSeed` attribute, and corresponding command line options.

#### 4.3.2 Probabilistic flooding

The routing agent is `ProbaFloodingRouting`. It is the same as pure flooding, except that it uses a probability to decide whether to forward a packet or not (attribute in XML and `--probability` parameter in command line).

#### 4.3.3 Backoff flooding

The routing agent is `BackoffFloodingRouting`. Backoff flooding greatly reduces the number of forwarders for flooding. It is described in [3].

Backoff flooding needs the number of neighbours to compute its backoff window. If DEDeN algorithm is enabled (section 4.4.2), nodes use the estimation it provides, otherwise they use the real number of neighbours.

The `--backoffRNGSeed` command line parameter (found in XML file too) sets the seed for the RNG used to set the sending time inside the waiting window, and `--backoffRedundancy` (defaults to 1) sets the redundancy.

#### 4.3.4 SLR addressing and routing protocol

The routing agent is `SLRRouting`. The simulator implements the original SLR [7], with its two phases: *setup* (also known as addressing or coordinate acquisition) and *routing*. It brings to it several improvements, described at the end of this section.

Anchors are specified in the XML file, as attributes of the agent (section 3.1). Up to three anchors can be specified.

If the `dstNode` of a flow is `-1`, then all the nodes forward the packets of this flow, acting like a flooding.

`slrPathWidth` attribute and `--slrPathWidth` command line option sets the width of the SLR path (must be strictly positive, default value is 1).

The SLR addressing phase starts at time $200 * 10^9$ fs in the simulator, as shown in figure 2; this time can be modified in the XML file (`firstBeaconStartTime_fs` attribute). At that time the anchor with the smallest `id` sends its beacon. To avoid beacons to meet each other, each of the other anchors sends its beacon after $.1 * 10^9 * \text{id}$ fs after having received the beacon from the first anchor, where id is the number of that anchor (in ascending order: 1, 2, etc.) This latter value can be modified in the XML file (`intervalBetweenBeaconStart_fs` attribute).

Note that BitSimulator assumes that the sending node knows not only the destination node id, but also its SLR coordinates. BitSimulator does not deal with acquiring this information. Dealing with this depends on the communication pattern. In many-to-one, the sink might flood once the network, so that all the nodes get its SLR coordinates. In many-to-many, a protocol to discover the SLR zone of the destination is needed. This can be done either using flooding, or using a register service like DNS, where all the nodes register at the beginning. This is to be done when needed (the node has a packet to send), and maybe only once, because the node will cache this information (depending on its available memory too).

The `--loadSLRPositionsFromFile` parameter skips SLR addressing phase and loads SLR coordinates from previously generated files (acting like an SLR cache). It tremendously speeds up very short simulations. But be very careful: No check is done whether the scenario has changed since then, so use it only during testing for a small period of time, and regenerate it often!

BitSimulator adds to the original SLR several improvements:

1. It adds a backoff when forwarding packets, as specified by `defaultBackoffWindow` attribute and `--defaultBackoffWindow` command line option (section 3.1).

2. It allows (through `commRangeSetup_nm` attribute of `SLRRouting` tag) to specify the communication range used during the setup phase, which can be smaller than the one used during routing (i.e. `defaultCommRange`). This allows to create SLR mini-zones, to ensure or at least to increase the chances that a node in a zone can reach at least a node in the next zone.

3. When sleep is used, *all* the nodes in the destination zone retransmit the packet (this is needed to allow the destination node to receive the packet when the packet arrived in the destination zone and by bad luck it was sleeping at that time).

4. When using `useCounterBasedForwarding` attribute, during the routing phase, only some of the nodes forward the packet, using a backoff (when forwarding packets) taken not from a fixed window, but from a dynamic one, and a counter, exactly like in backoff flooding (described in section 4.3.3).

5. When using `useDeviation` attribute, during the routing phase, the SLR width, initially 1 (or as given by `slrPathWidth` value above), is stored in packets and is increased by nodes in case of congestion of concurrent flows. Additionally, `deviateThresh` and `convergeThresh` parameters in the agent `.cpp` file change the thresholds. Details are given in [1].

#### 4.3.5 Ring protocols

#### 4.3.6 Implementation details

**RoutingAgent** The `RoutingAgent` class handles the reception and the transmission at the network level. Any routing agent inherits from the `RoutingAgent` class. Only one routing agent can be used in a simulation. A routing agent runs two main functions: `receivePacketFromApplication`, which handles packets coming from the application (upper layer) and is called only on the source node of the packet, and `receivePacketFromNetwork`, which handles the reception of packets coming from the network (lower layer) and is called on each hop of the packet (but not on the source node of the packet). Section 5.1 describes how to create new routing agents.

### 4.4 Application layer specification

#### 4.4.1 CBR sources and sinks

By default, all the flows in CBR use the default beta (from `ts-ook` element). It is possible to specify a different beta for a flow, by using `beta` attribute.

Use positive ($\geq 0$) values for `flowID`, so that it does not conflict with other protocols (e.g. DEDeN, SLR setup) which use `-1`.

The `startTime` attribute specifies when the CBR flow starts sending data ($1000 * 10^9$ fs in our example in section 3.1). Note that it usually starts after the other algorithms, as shown in figure 2, be careful that it does not overlap with SLR setup phase for example.

If the `dstId` attribute is `-1`, then all the nodes are destination of the CBR packets. Note that there is a difference between flooding with a given destination, and with `-1`: in both cases all the nodes forward the packet, but whereas in the first case only the destination node upwards the packet to application level, in the second case all the nodes do that. This is similar to sending a packet in Wi-Fi to a given IPv4 address, or to 255.255.255.255: in both cases all the nodes receive the packet, but it is processed only by the destination node in the first case, or by all nodes in the second case. TODO write/move this info to pure-flooding?

`repetitions` attribute sets the number of packets generated. If greater than 1, `interval_ns` attribute sets the time between the sending of the *first* bit of two consecutive packets. A value of 0 means to send all the packets at the same time; note however that a node in BitSimulator cannot send several packets at the same time, but sequentially.

The payload is binary and random. It uses its own RNG, whose initial seed can be set using `--payloadRNGSeed` parameter in command line.

#### 4.4.2 DEDeN

Density Estimation of Dense Networks is explained in [2]. It is enabled with `--deden` command line or with `<DEDeN enabled="true"/>` in the XML file. It has two phases: *init*, where a packet is broadcasted to whole network to initiate the algorithm, and *probe*, where nodes send packets to get their estimation of neighbour density.

Enabling DEDeN creates the two-phase packet exchanges above to estimate the node densities, and makes protocols using density information, such as those based on backoff flooding (see their description), use that estimation instead of the *real* number of neighbours.

By default, DEDeN init phase starts at time 0 in the simulator, and probe phase a bit later???, as shown in figure 2. `startTime_fs` attribute can be used to modify the starting time.

The `--dedenLoadEstimationFromFile` parameter skips DEDeN phase and loads DEDeN information from previously generated files (acting like a DEDeN cache). It tremendously speeds up very short simulations. But be very careful: No check is done whether the scenario has changed since then, so use it only during testing for a small period of time, and regenerate it often!

#### 4.4.3 Implementation details

**ApplicationAgent, ServerApplicationAgent, DataSinkApplicationAgent** An `ApplicationAgent` class can be attached to nodes, either by instantiation or by inheritance. It handles packet transmission at application layer. For instance, the CBR application generates packets with a fixed interval between consecutive packets. `ServerApplicationAgent` is a specific `ApplicationAgent` able to handle transmission and reception. `DataSinkApplicationAgent` is a basic server, which just receives packets. Applications have ports attached, like in TCP/UDP, so that several applications can run on a node.

### 4.5 Log system

The log system writes two main files:

- `events.log`: the main log file, which contains all the tracked events during the simulation, such as packet receptions and emissions
- `positions.log`: contains the geographical positions of nodes.

In some cases, it writes other files as well:

- `SLRPositions.log`: contains the SLR coordinates of nodes, when using an SLR-based routing

- `neighboursPositions.log`: contains the neighbours list for all nodes, when `--neighboursList` is passed in the command line

- `densityError.log`: contains the neighbours estimation (node density) for all nodes, when DEDeN is enabled.

When running several simulations with the same scenario, it becomes useful to keep their log files (for analysis purposes) instead of overwriting them each time. This can be achieved with `prefix` attribute in xml file or in command line: all the log files generated will have this prefix.

#### 4.5.1 events.log file

`events.log` describes the simulations events and embeds its own dictionary. Each event is described by a type of line, and each time a new type of line appears in the log file, an XML description of this line is given right before it. For example:

```
#<lineFormat id="0" key="s" description="packet sent">
#  <item type="Integer 64" key="time">simulation time</item>
#  <item type="Integer 32" key="nodeID">node ID</item>
#  <item type="Integer 32" key="transmitterID">node ID</item>
#  <item type="Integer 32" key="beta">beta</item>
#  <item type="Integer 32" key="size">packet size</item>
#  <item type="Integer 32" key="type">packet type</item>
#  <item type="Integer 32" key="flow">flow id</item>
#  <item type="Integer 32" key="seq">pkt sequence number</item>
#</lineFormat>
0 3900234 0 0 1000 40 3 1 0
```

The XML stanza (lines starting with `#`) is the dictionary, and is put right before each new line type. In this example, the last line contains the "communication" data, and has the following fields, in order:

1. event type, 0 stands for "packet sent", as given in the dictionary; common event types are: sent, received, collision, ignore (when the packet arrives but the buffer contains already MCR packets, hence packet is dropped; note that a packet which is both collided and ignored is counted as ignore only)

2. date of the event in femtoseconds

3. node on which the event happened

4. node that transmitted the packet; here it is the same as the 3rd field, since the event is a "sent" packet; in a reception event these two fields would be different

5. beta

6. packet size in bits

7. packet type (not to be confounded with the *event* type); packet types are specified in `packet.h` file:

```
enum class PacketType {
                        DATA,                   //0
                        SLR_BEACON,             //1
                        DENSITY_PROBE,          //2
                        D1_DENSITY_INIT,        //3
                        D1_DENSITY_PROBE,       //4
...
```

```
+-------------------------------------------------> Time
^ startSendEvent (sender starts sending the first bit)
|traveltime|
Bits at    +___+        +___+        +___+
receiver:  ^ startReceiveEvent           ^ endReceiveEvent
```

Figure 5: Times when events related to packet transmission occur. TO BE CHECKED

For the particular case of SLR beacons and DEDeN probes, the destination id is set to `-1`. (This information does not appear in the case above because the destination id is simply not added to the above particular log.)

Section 5.4 explains how to create log lines in `events.log` file.

### 4.6 Reproducibility information

BitSimulator gives reproducible results.

It eases the reproducibility by using several random number generator (RNG) instances, each with its own seed, e.g. `backoffRNGSeed`, `dedenRNGSeed`. Thus, it allows to see how results are affected by the random numbers used for backoffs (and not by node position for example). The various RNGs are mentioned in the appropriate sections.

Simulations done with BitSimulator are reproducible, with two exceptions. `gcc` compiler version 11 changed uniform_int_distribution, which affects results. Also, the same scenario executed on GNU/Linux and macOS give different results; this is because the RNG used in the two systems are different.

### 4.7 Other implementation details

**Scheduler** BitSimulator is an event-driven simulator; the scheduler is the core of the simulator. It pulls chronologically the events from a (time,event) map and consumes them.

**Event** Events can be generated at any moment by any class of the simulator. They are used to schedule most of things in the simulation. They are automatically consumed by the scheduler.

For example when a CBR application sends a message, a first CBR event is created. All the subsequent events are automatically managed by the simulator. This event is consumed (by the scheduler) and generates two events: it makes the sending node emit the packet (through a `startSendEvent` event) and it schedules another CBR event. Later, when the `startSendEvent` is consumed, it creates (schedules) one event (`startReceiveEvent`) for each of the nodes receiving the packet, and an `endSendEvent` for itself. Each `startReceiveEvent` creates an `endReceiveEvent`, which calls the collision code to decide if a packet is correctly received or not. Collision checking (described in section 4.2.5) does not generate any event.

The exact times when these events occur are shown in figure 5. Thus, to have two packets exactly one after the other for instance, the delay to add at `endReceiveEvents` to schedule the second packet is $T_s - T_p$.

**Packet** The `Packet` class represents messages exchanged between nodes, no matter the layer (application, network

etc.) Header fields are specified as members of the `Packet` class. The `payload` array represents the actual data; currently, it is random (contains as many 1 as 0); by changing the distribution of 1 and 0 it is possible to simulate different types of coding.

# 5 Tutorials

## 5.1 Creating a new routing agent

This tutorial presents how to modify BitSimulator in order to add a new routing agent. This example shows a simple routing agent that forwards packets according to the packet sequence number. The idea is to make only half of the nodes forward a message in order to make a somewhat optimised broadcast from a node to the whole network.

All new agents are derived from the `RoutingAgent` class. All the routing agents are found in the `src/agents` directory. You can inspire from any file from this directory, such as `pure-flooding-routing-agent.cpp`.

Create the file `src/agents/new-routing-agent.cpp` with the following content:

```
#include "new-routing-agent.h"

NewRoutingAgent::NewRoutingAgent(Node *_hostNode) : RoutingAgent(_hostNode) {
}

void NewRoutingAgent::receivePacketFromNetwork(PacketPtr _packet) {
}
```

and the accompanying file `src/agents/new-routing-agent.h` with the following content:

```
#include "routing-agent.h"

class NewRoutingAgent : public RoutingAgent {
public:
  NewRoutingAgent(Node *_hostNode);
  void receivePacketFromNetwork(PacketPtr _packet);
};
```

Add these files to `Makefile.am` file, and run `./configure && make`. There should be no error.

The next step is to attach it to nodes. In the `src/node.cpp` file, `Node::startupCode` function the current code is:

```
// attach the routing agent
string agent = ScenarioParameters::getRoutingAgentName();
if (agent.compare("PureFloodingRouting") == 0)
  attachRoutingAgent(new PureFloodingRoutingAgent(this));
else if (agent.compare("NoRouting") == 0)
  attachRoutingAgent(new NoRoutingAgent(this));
[...]
else if (agent.compare("ProbaFloodingRingRouting") == 0)
  attachRoutingAgent(new ProbaFloodingRingRoutingAgent(this));
else
  assert (false);  // no valid routing agent specified
```

Here you have to add your own `else if` line and compare the `agent` value with the name you give to your routing agent. The last lines of the code above become:

```
else if (agent.compare("ProbaFloodingRingRouting") == 0)
  attachRoutingAgent(new ProbaFloodingRingRoutingAgent(this));
else if (agent.compare("NewRouting") == 0)
  attachRoutingAgent(new NewRoutingAgent(this));
else
  assert (false);  // no valid routing agent specified
```

Note that the "string name" for comparison and the actual class name can differ.

You also need to include the header file after the list of the other header files:

```
[...]
#include "agents/proba-flooding-ring-routing-agent.h"
#include "agents/slr-routing-agent.h"
#include "agents/slr-ring-routing-agent.h"
#include "agents/new-routing-agent.h"
```

Recompile with `make`, there should be no error.

The last step is to specify it in the `scenario.xml` file. Replace the already mentioned routing agent with your own:

```
<routing>
  <NewRouting/>
</routing>
```

The `NewRoutingAgent` is now available in the simulator.

Now, let us make the agent do the forwarding we want, as specified in the beginning of the section. For that, we modify the `NewRoutingAgent` constructor to add the new specificities: we add a "parity" variable in `NewRoutingAgent` attributes, and randomly set it to 0 or 1.

The `initializeAgent` method, called *before* the agent's constructor by `World::initAgents()`, can be used to set things which cannot be set in the constructor, such as log files or a specific random generator.

Next, two functions have to be implemented:

- `receivePacketFromApplication`, which handles packets coming from the application (upper layer)
- `receivePacketFromNetwork`, which handles the reception of packets coming from the network (lower layer)

In our case, `receivePacketFromApplication` is very simple and just sends the packet to the lower layer:

```
void NewRoutingAgent::receivePacketFromApplication (PacketPtr packet) {
  // keep track in the packet of how many packets have been
  //   sent before from the same source
  packet->setSrcSequenceNumber (hostNode->getNextSrcSequenceNumber());

  // add packet to the sending queue of the node
  // once in this sending queue, packets are sent automatically
  hostNode->enqueueOutgoingPacket (packet);
}
```

The `receivePacketFromNetwork` function is the following:

```
void NewRoutingAgent::receivePacketFromNetwork (PacketPtr packet) {
  // the log file system is explained in a dedicated tutorial
  LogSystem::EventsLogOutput.log (LogSystem::routingRCV,
    Scheduler::now(), hostNode->getId(), (int) packet->type,
    packet->flowId, packet->flowSequenceNumber,
    packet->modifiedBitsPositions.size());

  // if the packet arrived to the destination or is a
  //   broadcast packet (its dstId is -1), then send it
  //   to the coresponding ServerApplicationAgent hosted by the node
  if (packet->dstId == hostNode->getId() || packet->dstId == -1) {
    hostNode->dispatchPacketToApplication (packet);
    LogSystem::EventsLogOutput.log (LogSystem::dstReach,
      Scheduler::now(), hostNode->getId(), (int)packet->type,
      packet->flowId, packet->flowSequenceNumber);
  }

  // test if this packet has to be forwarded by comparing
  //   the parity variable with the packet sequence number
  // thus, packets are alternatively forwarded by half of the nodes
  if (packet->flowSequenceNumber == parity%2)
    hostNode->enqueueOutgoingPacket (packet);
}
```

The simulation is now ready to run with the brand new routing agent.

## 5.2 Creating, scheduling, and processing an event

This tutorial shows how to schedule an event and how to specify the code to execute when that event is triggered. Specifically, it shows how to send a packet after some delay (similar to a backoff or waiting time).

Let `NewRoutingAgent` be the class dealing with the event. Here is an example of creating, scheduling, and processing an event:

```
PacketPtr packetClone (packet->clone());
auto sendEvent = new NewSendEvent (Scheduler::now()+10000,
                                   this, packetClone);
Scheduler::getScheduler().schedule (sendEvent);

void NewRoutingAgent::processSendEvent (PacketPtr packet){
  hostNode->enqueueOutgoingPacket (packet);
}
```

In the header file of the class we need to add:

```
virtual void processSendEvent (PacketPtr packet);
using NewSendEvent = CallMethodArgEvent<NewRoutingAgent,
                          PacketPtr,
                          &NewRoutingAgent::processSendEvent,
                          EventType::GENERIC>;
```

If you need to create your own event type, replace `EventType::GENERIC` above with `EventType::NEW_SEND` for example, and add `NEW_SEND` in `enum class EventType` declaration in `src/eventtypes.h` file.

Commit `b40239` contains a real case of event creation and processing.

## 5.3 Creating a new command line option (parameter) with TCLAP library

This tutorial shows how to add options to command line when starting BitSimulator. This specific example adds a new RNG seed in the simulator, usable for some random phenomenon.

BitSimulator uses the TCLAP libary to handle the command line options. As such, this tutorial simply shows how to use TCLAP library to add a command line option (BitSimulator does not introduce any particularity in this respect) and how to integrate the changes in the code.

The first step is to declare the parameter in `utils.h` file, after de `##COMMANDLINE` tag in the `ScenarioParameters` class description. Two declarations are needed:

- the TCLAP variable that will receive the parameter
- a "standard" variable to access this parameter anywhere in the simulator.

Example :

```
int myRandomSeed;
TCLAP::ValueArg<int> *myRandomSeedParam;
```

The options can be of any type, however the template of the `TCLAP::ValueArg` has to match to type of the variable. Note that there is a special case for boolean.

In the following code, the `TCLAP::ValueArg<T>` becomes a `TCLAP::SwitchArg`:

```
bool myBoolean;
TCLAP::SwitchArg *myBooleanParam;
```

The second step is to add the new option to the command line. Add the description of your new parameter after the `##COMMANDLINE` tag in `utils.cpp` in the `ScenarioParameters` constructor, such as:

```
TCLAP::ValueArg<int> myRandomSeedParam ("w", "mySeed",
  "This is my brand new random seed", false, 0, "int", cmd);
```

The parameters of the `TCLAP::ValueArg<T>` constructor are, in order:

- The short option, invoked by using `bitsimulator -w 42`
- The long option, invoked by using `bitsimulator --mySeed 42`
- The option description, which appears at `bitsimulator --help`
- A boolean to indicate if this option is needed (`true`) or not (`false`)
- A default value, taken if it does not appear in the command line
- The type of the parameter
- The command line variable

The final step is to get the value of the parameter. After the line

```
cmd.parse( _argc, _argv );
```

which actually parses the command line, the value can be retrieved like this:

```
TCLAP::ValueArg<T> myRandomSeed = myRandomSeedParam->getValue();
```

It is a good practice to use a private variable in the `ScenarioParameters` class, available in the whole simulator, and to use a getter to access it, like this: `ScenarioParameters::getMyRandomSeed();`.

## 5.4 Creating new types of lines in the events.log file

This tutorial modifies BitSimulator in order to manipulate the log system. These lines appear in the `events.log` file and describe the simulation. This example adds a log line concerning `SLRBackoffFloodingAgent3`. In this routing protocol, some packets can be dropped if they are copies of another packets. We are going to track those drops.

The first step is to declare our new log line. In the `log.h` file, in the `LogSystem` class we add the new line format after the `#LogSystem` tag:

```
static OutputLineFormat SLRBackoffDrop;
```

Since the line format is static in the `LogSytem` class, we class the constructor outside the class, after the `#StaticLogSystem` tag in `utils.cpp`:

```
OutputLineFormat LogSystem::SLRBackoffDrop ("dr","SLR Backoff Drop");
```

The first parameter stands for the "key" and the second parameter is a brief description. Both appear in the log file dictionary.

The second step is to build the new log by adding some content in the line format. For that, in the `log.cpp` in the `initLogSystem()` function after the `#LogSystem` tag we add the specification of our new line:

```
SLRBackoffDrop.addItem (LogItem::INT64, "date", "date in fs");
SLRBackoffDrop.addItem (LogItem::INT32, "nID", "node ID");
SLRBackoffDrop.addItem (LogItem::INT32, "pkT", "packet Type");
SLRBackoffDrop.addItem (LogItem::INT32, "flow", "flow id");
SLRBackoffDrop.addItem (LogItem::INT32, "seq", "sequence number");
```

These 5 lines will create a log line of 6 elements:

- a line id automatically created, useful to recognize the new line in the log file
- the date of the drop coded on a 64 bit integer
- the id of the node that dropped the packet
- the packet type (packet type description is available in `packet.h`)
- the flow id of the packet
- the sequence number within the flow.

Some string items can also be added to the line (but not relevant in this example):

```
SLRBackoffDrop.addItem (LogItem::STRING, "mS", "My string");
```

The final step is to actually use the line just created. In the `slr-backoff-routing-agent3.cpp` file, the `SLRBackoffRoutingAgent3::receivePacketFromNetwork (PacketPtr _packet)` function handles packets coming from the network. This function decides whether an incoming packet is to be dropped or not. After the `// #Tuto` (which is where waiting packets are erased), we add our logging line to be notified each time a packet is dropped:

```
LogSystem::EventsLogOutput.log (LogSystem::SLRBackoffDrop,
  Scheduler::now(), hostNode->getId(), it->second.p->type,
  it->second.p->flowId, it->second.p->flowSequenceNumber);
```

Note that here `it->second.p` is a packet.

The `log` function is variadic, hence its parameters are not checked for correctness, so pay attention to them: their number and type depend on the `addItem` made in the `lineFormat`.

This log system can be called everywhere in the code as soon as the LogSystem has been initialised.

# 6 VisualTracer

VisualTracer is a *very powerful* tool to understand what happens in the network.

## 6.1 Command line

The command line options for VisualTracer are the following:

```
[--scenarioFile <string>] [-D <string>] [--prefix
<string>] [--3d] [--initialTimeSkip <long>] [-s <long>]
[--nodeZoom <int>] [--chrono <int>] [--cn <long>] ...
[--complementaryNodeFloatInfoFileName <string>] [--]
[--version] [-h] <string>
```

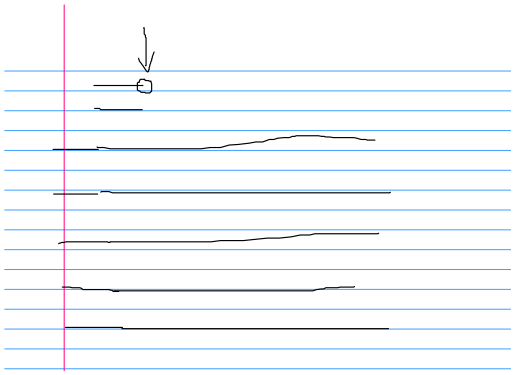

Figure 6: An ignored packet can be shown *before* the concurrent packets which made it ignored.

Execute `visualtracer -h` for more information.

Important parameters are `--chrono`, which shows the point of view of a node, and `--stepLength` (or `-s`), which sets the time window (step) length inside which the events are shown in the interface at a given moment.

## 6.2 Graphical interface

At any given time, VisualTracer shows the events in the network and node states during the entire step (time frame), i.e. all the nodes which send a packet somewhere during this step will be shown in blue etc. The length of a step in VisualTracer is given by `--stepLength` (or `-s`) parameter. Time windows shown in VisualTracer start at iTS + k*step, where iTS is 0 if not specified on the command line (`--initialTimeSkip`). Empty time windows are skipped (not shown). For ex. if events occur at time 0, 1, 11, 13, and if the step length is 2, then step 1 is [0,2), step 2 is [10,12), and step 3 is [12,14).

Note that, counterintuitively, an ignore event can be shown *before* the receiver event if a small packet ignored started being received *after* a big packet, cf. figure 6.

Histograms on the right side and on top show the number of *events* in the current time frame. Note that number of events can be higher than number of nodes, in case a node receives several packets in the same time frame.

In 3D mode, the histograms are not shown, and *inactive nodes* switch in the interface does not work.

## 6.3 Keys

You can interact with VisualTracer using the keyboard:

- Space bar or `n` to display the next step (go forward)
- Backspace key or `p` to go one step backwards
- Arrows to move the network view
- `z`/`a` to zoom in/out
- The keys specified at the top of the window (`s`, `r`, `c`, `i`, `t`, `m`) to show/hide specific states of nodes
- `q` or `ctrl-q` to quit

Additionally, in 3D the network view can be rotated with the mouse (click & move) or with the following keys: `h`, `k` (for left/right) and `u`, `j` (for up/down).

Clicking on a node shows information about it (id and coordinates) on terminal.

# References


[1] T. Arrabal, F. Büther, D. Dhoutaut, and E. Dedu. Congestion control by deviation routing in nanonetworks. In *6th ACM International Conference on Nanoscale Computing and Communication (NanoCom)*, pages 1–6, Dublin, Ireland, Sept. 2019. ACM/IEEE.

[2] T. Arrabal, D. Dhoutaut, and E. Dedu. Efficient density estimation algorithm for ultra dense wireless networks. In *27th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9, Hangzhou, China, July-Aug. 2018. IEEE.

[3] T. Arrabal, D. Dhoutaut, and E. Dedu. Efficient multi-hop broadcasting in dense nanonetworks. In *17th IEEE International Symposium on Network Computing and Applications (NCA)*, pages 385–393, Cambridge, MA, USA, Nov. 2018. IEEE.

[4] D. Dhoutaut, T. Arrabal, and E. Dedu. BitSimulator, an electromagnetic nanonetworks simulator. In *5th ACM/IEEE International Conference on Nanoscale Computing and Communication (NanoCom)*, pages 1–6, Reykjavik, Iceland, Sept. 2018. ACM/IEEE.

[5] J. M. Jornet and I. F. Akyildiz. Femtosecond-long pulse-based modulation for terahertz band communication in nanonetworks. *IEEE Transactions on Communications*, 62(5):1742–1753, May 2014.

[6] A. Medlej, K. Beydoun, E. Dedu, and D. Dhoutaut. Scaling up routing in nanonetworks with asynchronous node sleeping. In *28th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 177–182, Hvar, Croatia, Sept. 2020. IEEE.

[7] A. Tsioliaridou, C. Liaskos, E. Dedu, and S. Ioannidis. Packet routing in 3D nanonetworks: A lightweight, linear-path scheme. *Nano Communication Networks*, 12:63–71, June 2017.