olsonet Communications

The Netting Demo

Version 1.2
December 2015

Introduction

The praxis illustrates some of the networking capabilities of our nodes and the communication aspects of applications built with them. Note that it doesn't demonstrate everything, like, for example, all the low-level RF communication options (some of which may prove advantageous in specific situations), or even all the higher-level features that our production praxes can routinely rely on, e.g., remote node configurations. We have trimmed down the functionality to make things simple and relatively obvious, especially for a novice user.

The praxis user interface is very simple and consists of textual commands entered over the node's UART. Every node running the praxis can (but doesn't have to) be connected (via a USB dongle) to a COM port on a PC.

You have two options for playing with the praxis: virtual (i.e., via a VUE^2 model) and real. The first option is definitely simpler from the logistics point of view: there is no need to run around with the boxes, turn them on and off, connect (at least some of) them to a PC, physically adjust their placement, and so on. The model will let you accomplish all that by clicking and dragging the mouse. Moreover, it will let you test things that are impossible (or not feasible) to test in real life (deploying more nodes than you have physically available, redefining the properties of the wireless channel, and much more). From some important point of view, the two options are equivalent. With both of them you are running the same code and the same program. True, with the virtual option the environment of that program is artificial, but its level of sophistication (and reflection of reality) is sufficient to make a real program for a real node believe that it is running for real. You should keep in mind that there is no such thing as the single right target environment for running an ad-hoc wireless network. So there is no need to feel shortchanged.

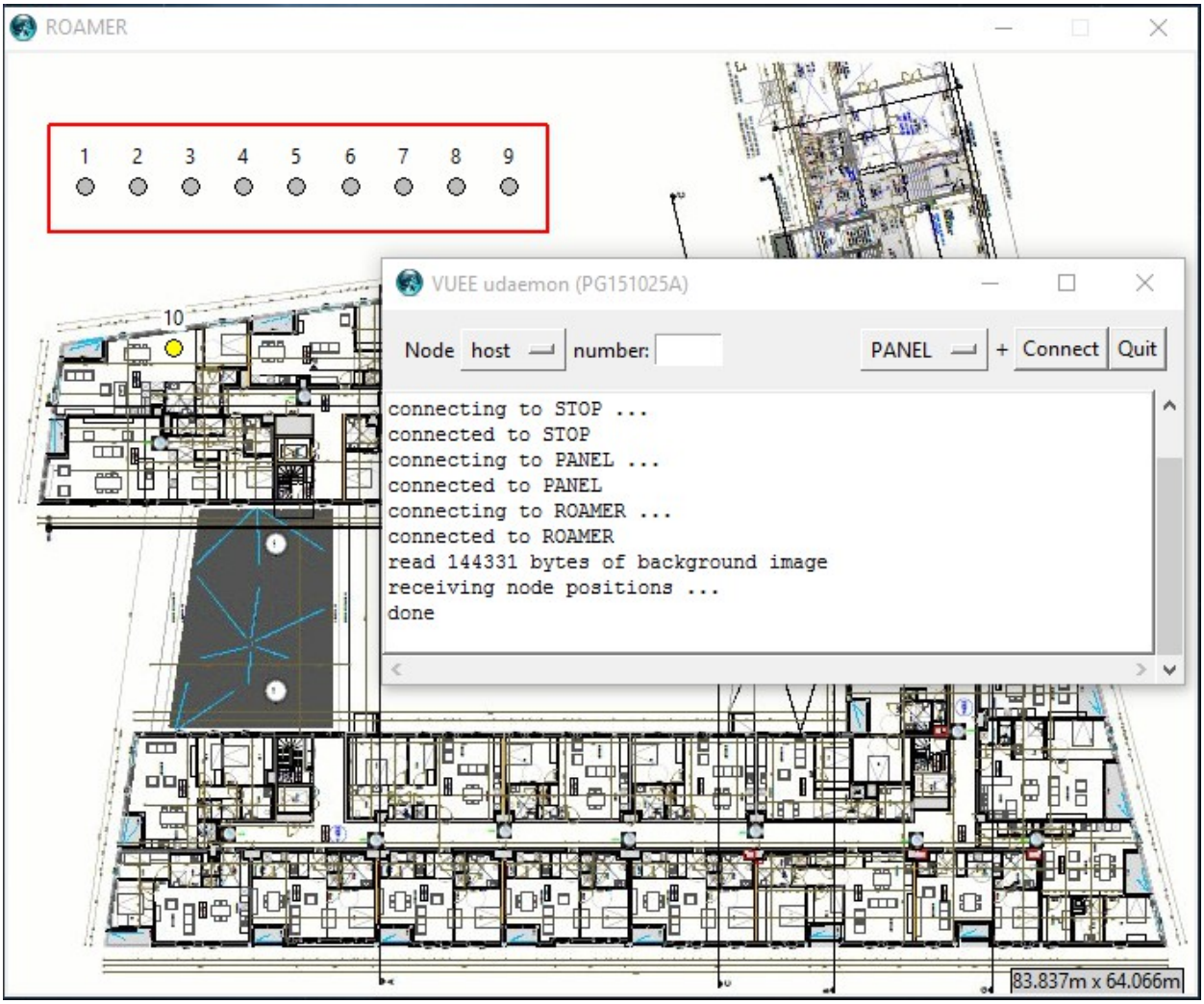
This point cannot be overstressed, so at the risk of repeating ourselves, let us rephrase it a bit to make it even more clear: the model is not some simplistic fake (as models often are). The application code executed by the virtual nodes is *exactly* the same as that loaded into the real nodes. You see exactly the same commands and the same kind of responses as you would (will) see in the real network. The virtual wireless channel exhibits qualitatively the same properties as any real RF propagation environment (packets are transmitted and received in finite time depending on their length and transmission rate, the propagation time between a pair of nodes is finite and depends on their distance, the bit error rate is a function of background noise and interference, and so on). Needless to say, your actual (real) wireless channel may exhibit quantitatively different properties, depending on your actual (physical) distribution of nodes, the obstacles, local sources of interference, etc. This may affect the quantitative statistics of packet delivery, but the essence of the system behavior will remain the same.

Then, of course, you can always rerun your tests with the actual physical nodes and compare them to the ones obtained from the virtual sessions. All the exercises described here will work in both cases. Their outcomes may differ slightly (in quantitative terms), to the extent they would differ in different real-life deployments. This is OK.

Running the model

For convenience (and abstraction from any particular terminal emulator), the introductory examples will be presented as run in the model. To run the model yourself, do this:

1. Unzip the package (its file name probably starts with *VUEEPACK*, and the extension is .zip or .7z). Open the resulting folder and double click on the file *runmodel*).
2. In the window that opens, select the data file (the button menu close to the lower left corner). The discussion in the next section is based on file *example.xml*. This file should appear in the menu button being selected by default.
3. Click *Run* (the lower right corner) to start the model. After about two seconds, the text area of the main window will fill with some output (this is the, generally uninteresting, textual output produced by the emulator) and two windows will pop up, looking (more or less) like this:



The smaller window (titled *VUEE udaemon*) is the emulator's console. Strictly speaking, the windows are presented by a separate program (called *udaemon*) connecting to the emulator and acting as its GUI proxy.[1] The larger window (dubbed *ROAMER*) shows the deployment of our network. To make it less artificial, we put it against the layout of some building. For example, imagine that it is some facility (e.g., an independent-living home) where sensors and panic

[1]For the technical details (which we will not concern ourselves with in this document) refer to the VUE[2] manual.

buttons are going to be deployed, so the purpose of our experiments is to test the communication within a network of such devices.

Hint: after a while, you will probably become annoyed by the yellow help balloons popping-up whenever you hover the mouse over some widgets. To switch them off, click the + sign in the console window of *udaemon* (the one between the menu and *Connect* buttons).

For this experiment, we have 10 nodes to play with. One of them, node 10, is already "deployed" in a room within the building. The remaining nodes are "stored in a box" and switched off. In contrast to Node 10, they are painted gray. You can see more clearly that they are switched off when you open the *PANEL* window which will allow you to power the nodes up or down and also reset them. Thus, select *PANEL* in the console window, in the menu to the left of the *Connect* button, and click the *Connect* button. You should see this window:

Only Node 10 is turned on,[2] the remaining nodes are off. You can switch any node on and off at any time (corresponding to inserting/removing batteries in real life) from the panel. Resetting a node amounts to switching it off and back on.

Talking to the nodes

Each node in our network is equipped with a UART module. Let's try to connect to the UART of Node 10. You can do it by right-clicking on the node's circle in the *ROAMER* window and selecting *UART (ascii)* from the menu. You should see a window titled *UART at node 10* looking like a terminal emulator with an input area at the bottom. Don't worry about the widgets to the right of the entry area: they are irrelevant for our exercise. Enter *h* (or *help*) in the input area and press the *Return* key. The node will respond with the list of available commands, something like this:

```
Netting commands:
```

[2]For node numbers (addresses) look at the pinkish squares (second from the left), the gray (leftmost) ones show some internal numbers which we don't care about.

```
s (set / show) disp, odr, trace, beacon, TARP:
sd [ref# dst ack <string>]
so [ref# ack hop1 ... hop9]
st [ref# [dst(0) [ dir(3) [ hops(0) ]]]]
sb [ (d,o,t) freq (<64) volume (>0) ]
sT *CAREFUL* [plev rx fwd slack rte tlev]
s (show node status)

Beacon operations:
ba (activate)
bd (deactivate)

Transmit:
x(d,o,t)

FIM operations:
Fw(rite)
Fe(rase)
F (read)

m(aster) [1/0]
t(ransient) [1/0]
f(ormat) [1/0]
h(elp)
q(uit)
```

By the way, if you open the UART window of some other node, e.g., Node 1, its display area will be gray. This is an indication that the node is switched off. Needless to say, such a node will not respond to your commands.

Note: another way to open a node's UART window is to enter the node number into the *number* field in *udaemon's* console, select *UART (ascii)* from the console's menu (the same you used to open the *PANEL* window), and click *Connect*. This (more complicated) variant may be handy, e.g., when the node is mobile and keeps running away from the mouse.

Messages

Netting's mode of operation is “set & go”. By this we mean that you first describe the parameters of a message that you may want to transmit (i.e., preconfigure the message), and then use a separate (and trivially simple) command to send that message out as many times as you please. This way, we minimize the number of keystrokes needed to quickly run many potentially interesting test cases. There exist three message types which can be defined and referenced independently. Only one message of a given type can be defined at any given time, but up to three messages (of different types) can coexist. The message types are:

display A simple message whose role is to be sent to a specific destination (including all destinations, i.e., broadcast, as a special case). Such a message may carry in its payload a string of characters. The way the message is delivered is up to the network. The multi-hop delivery mechanism is based on our proprietary TARP scheme.

odr A message that is supposed to travel a specific path prescribed by the operator

(as in Operator Directed Routing). Note that normally our applications do not use this form of communication; it has been implemented for illustration, comparisons, and various networking experiments and demonstrations.

trace A message to be delivered via TARP, whose actual route is to be tracked and reported. The report can be produced at the destination or/and at the source (based on information carried in a reply message sent by the destination back to the source).

Any message, regardless of its type, can be turned into a *beacon*, i.e., be automatically sent at prescribed intervals a specified number of times.

Some experiments

The first thing to note is that you can place nodes by dragging them with the mouse. Let us start by placing Node 5 in some distant location, e.g., in the lower right corner of the floor and turning it on. Open the UART window for Node 5. Issue some simple command, e.g., *h* or *q*. Although the apparent outcome of both commands is the same, the latter resets the node and brings it back to the initial state. The help menu is also shown whenever the node starts up.

Try *s* to see the node status. In response, you will see a line similar to this one:

```
5 at 34: M 0 at 0 mem 4292.4052.256 bat 2600 trans 0 ofmt 0
```

which says that you are at node number 5 which has been running for 34 seconds (from last power up or reset). The node knows of no master node (this is the *M 0* part, more about it later). The amount of free RAM currently available to the node is 4292 words (translating into twice as many bytes) with the minimum that was ever noticed by the node equal to 4052 words. The third number in that sequence (256) shows the minimum number of stack locations (words) noted so far. The three numbers are indications of the node's health when it comes to memory requirements.[3] The third last number (2600) tells the battery status (it is the output of a voltage sensor). The battery should be replaced when the value drops to around 2200. The next number is the binary *transient* flag, driving the display of *ODR* messages on intermediate nodes. The last number indicates the output format variant applicable to some messages.[4]

Display messages

We shall try to send a simple message from node 5 to node 10. For that, we have to define the message first. It will be a simple message of type *display* (see above). Enter this command in the UART window of node 5:

```
sd 88 10 0 hello_node_ten
```

The command (**s**et **d**isplay) presets a message of type *display*, or should we say *the* message of type *display*, because there can only be at most one message of that type preset at any given time. The arguments are, in this order:

1. Some number to be sent inside the message (which you can view as a numerical identifier); the size of that number is 4 bytes (large enough to encode millisecond timestamps, for example).
2. The destination node ID; in our case it is 10.

[3]Our operating system run by the nodes (PicOS) employs dynamic memory allocation, e.g., for buffering packets.
[4]It can be reset to 1 for terse format which we ignore in this exercise).

3. A binary flag (0 or 1) telling whether we want to receive in return an acknowledgment from the recipient. In our case it is 0 meaning "no acknowledgment".
4. A piece of text to be sent in the message along with the numerical identifier. This text cannot contain a space (or rather a space terminates it in the same way as the end of line) and must be shorter than 40 characters.

Whenever you issue *sd* without arguments, you will see the currently preset display message. For example, following the above definition, an argument-less *sd* will bring you this:

```
Disp #88 to 10 ack 0 len 14 <hello_node_ten>
```

Make sure that the UART window for node 10 is opened, so you can witness the act of reception. To dispatch the message, enter this command in the UART window of Node 5:

```
xd
```

However, nothing happens when you do that. Node 10 does not receive the message. This is because the nodes are too far apart.

To see the (straight line) distance between the nodes do this. First click inside one of the nodes (say Node 10) using the normal (left click). Then move the mouse to the other node and click inside it in the same way. For a couple of seconds you should see a dotted line connecting the two nodes with a number somewhere in the middle. If you have placed Node 5 in the lower right corner of the floor, that distance should be about 84 meters.

Technically, the nodes can communicate over such a distance. We have intentionally set the (default) transmission power at all nodes to a level making communication within the building a bit more challenging. You can try to move Node 5 a little closer to Node 10 and run the command (*xd*) again. You should begin to see reception at a distance of 35-38 meters. When that happens, a message looking like this will show up in the UART window of Node 10:

```
2531: disp #88 fr 5.1 ack 0 14<hello_node_ten>
```

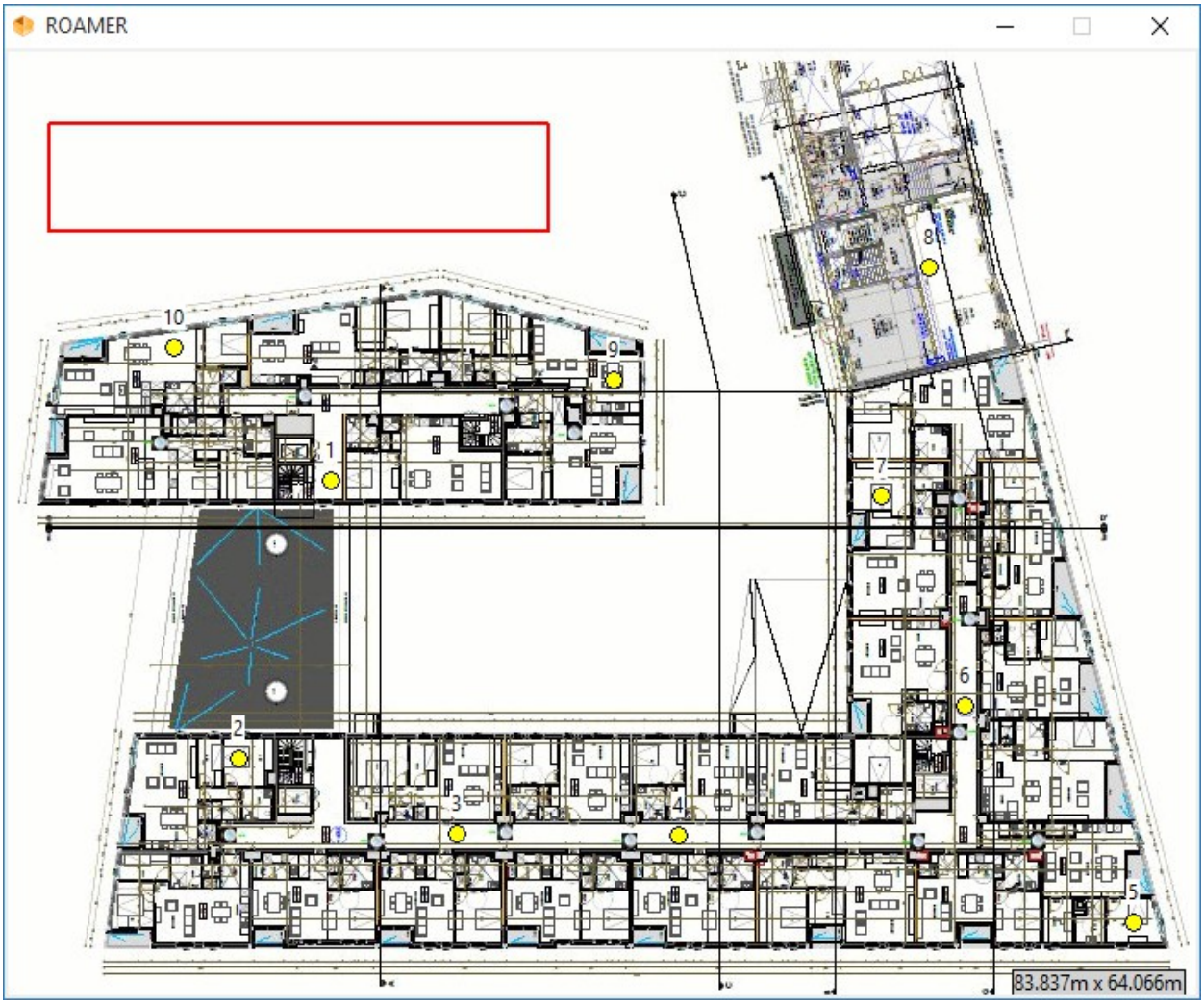
The line says that the node has received a display message with the numerical identifier 88 from Node 5. The message has traveled a single hop (the 1 following the dot), it doesn't ask to be acknowledged, and it carries the 14-character string *"hello_node_ten"*. The first number (2531) provides the time stamp of the reception. This is the recipient's time in seconds counted from the moment of its last power up or reset.

Using 25 meters as the basic, reasonably safe, single-hop distance, we can think of deploying the remaining nodes into a network that covers the floor.

Hint: once you have moved the nodes to their positions, you can save those positions, such that the next time you invoke the emulator, you won't have to deploy them again. To do that, click on the grayed dimension text in the lower right corner of the *ROAMER* window. You will see a dialog prompting you for the file to save the location information. Click on the *DATAFILES* folder and choose the (already existing) file *locations.xml*. The original version of that file contains the initial locations of the nodes. There is no completely trivial way to save the *on* status of the (all or selected) nodes. If you want to avoid powering them up after every start of the emulator, edit the data set (*example.xml*) and replace *start="off"* with *start="on"* for those nodes that you want to start powered up. There is a variant of the input file named *example_power_up.xml* which you can select (in the startup window) before running the model That file is identical to the original file, except that all the nodes are initially powered up. While

we are at it, one more input file that may be of some interest is *example_roaming.xml* with all the nodes already deployed and powered up. That data set includes two extra mobile nodes numbered 101 and 102 following some simple mobility patters.

For example, we can arrange the nodes like this:



For an easy check if they can communicate, we can try to reach them all from Node 10 with a single broadcast message. To define such a message, issue this command in the UART window of Node 10:

```
sd 12 0 0 broadcast_message_from_10
```

Note the destination address of 0 which means “broadcast”. When you dispatch the message (with *xd*) you can see whether it is being received by all nodes. No need to open a UART window at every node. When a node receives a message, it briefly changes its color to a shade of purple. By looking at the received message (in a UART window) you can tell how many hops it has made. For example, this is what you may (but not have to) see at Node 8:

```
611: disp #12 fr 10.2 ack 0 25<broadcast_message_from_10>
```

The message has reached Node 8 in just two hops. The output is still a bit short of telling the whole story, because, for example, it is not clear which way the message went. For that, we can resort to *trace* messages.

Trace messages

The role of trace messages is to track TARP routes. They travel to specific destinations, possibly over multiple hops, being routed by TARP rules, and report information about the hops taken along the way. Here is a way to set a trace message (we do this at Node 10):

```
st 17 8 1 0
```

Again, the first argument is a numerical ID of the message. The remaining ones say that the message is addressed to Node 8, the report should be made at the destination, and there is no limit on how many hops the message is allowed to travel. There is no textual content: the payload will consist of the route information automatically collected on every hop.

To see the report, open the UART window at Node 8 and dispatch the message (with *xt*) at Node 10. You should see at Node 8 something like this:

```
1321: tr(10) #17 5 189 2:
 9 90
 8 104
```

The output says that the message has been forwarded by Node 9 (which makes perfect sense when you look at the node placement). The first number in the first line is (as usual) the time stamp of the event (the number of seconds at the destination). The remaining numbers are not very interesting. You can spot the numerical ID (17) assigned to the message. The last number gives the total hop count. Then there is a line for every hop, each line identifying the receiving node. The second value is the RSS (Received Signal Strength) indication at the recipient. It is always an unsigned, single-byte value between something like 40 and 250 (higher values indicate a stronger signal).

The third (direction) argument of *st* can take these values:

0. Back trace reported at the sender only. It reports the route back from the recipient to the sender.
1. Forward trace reported at the destination. It reports the route forward from the sender to the recipient.
2. Forward trace reported at the sender. A special message containing the report is sent back from the recipient to the sender.
3. Bidirectional trace reported at the sender. The trace shows the full circle made by the original message + the reply.

Let us try a full report, say a route to Node 6 (still from Node 10). The trace message (defined at Node 10) may look as follows:

```
st 18 6 3 0
```

Here is one report produced at Node 10:

```
2805: tr(6) #18 6 3 5:
 2 93
 4 87
```


```
6 107
5 113
4 85
3 121
2 122
10 94
```

It says that the message went to Node 2, then to Node 4, then to Node 6 (the destination). On its way back, it went via Node 5 before making it to Node 4, and made an intermediate hop via Node 3 before arriving back at Node 2, from where it was finally delivered to the original sender. This illustrates that routes do not have to look identical in both directions. With TARP, they don't even have to look identical in the same direction, on different turns. Here is another report, for the same trace message, done just a few (16 to be exact) seconds after the first one:

```
Trace #18 to 6 dir 3 hco 0
2821: tr(6) #18 6 2 4:
 9 89
 6 83
 4 109
 3 120
 1 98
 10 123
```

This time the forward message went via Node 9 (happily reaching the destination in just two hops). On its way back, the message managed to bypass Node 5 (relative to the previous trace), making it to Node 4 in a single hop, from where it went to Node 3 (as before), but then it traveled to the source via Node 1, rather than Node 2.

Note that if the report is produced at the source, it starts with an extra header (the *Trace* line above). That header appears immediately, i.e., as soon as the message is dispatched. The remaining part of the report will only materialize when (and if) the response makes it back from the destination. The last two values in the (proper) report header indicate the two hop counts, i.e., forward and back.

Here is one more report (for the same *trace* message):

```
Trace #18 to 6 dir 3 hco 0
2943: tr(6) #18 6 3 2:
 2 92
 4 88
 6 109
 9 82
 10 92
```

illustrating that the number of hops can be larger in the forward direction.

ODR messages

This message type is intended to send data along specific, hard, point-to-point paths. Messages of this type can be used to test point-by-point routes (employed by traditional forwarding schemes that separate path construction from the actual forwarding). We don't forward messages this way, so we do not provide any specific path construction methods (akin to DSDV, AODV, and so on). If you want your message to go some hard p-p route, you have to specify the full path at the source.

Try this (assuming we still are at Node 10):

```
so 99 1 1 2 3 4 5 6 7 8
```

The command defines an ODR message whose numerical ID is 99 and whose route goes through nodes 1, 2, 3, 4, 5, 6, 7 to Node 8, with the report collected at the destination, i.e., Node 8, and sent back to the source, i.e., Node 10 (the report type is described by the second argument). When you open the UART window at Node 8 and issue the message (*xo*) at Node 10, you will see (at Node 8) something like this:

```
5123: odr #99 [1.8.8]:
 1:  (10 0 0)
 2:  (1 124 0)
 3:  (2 113 0)
 4:  (3 120 0)
 5:  (4 122 0)
 6:  (5 84 0)
 7:  (6 115 0)
 8:  (7 122 0)
 9: >(8 118 0)
```

The report presented by Node 10 looks similar:

```
Odr(8) #99 ack 1 [1 2 3 4 5 6 7 8 0]
5124: odr #99 [1.0.8]:
 1:  (10 0 124)<
 2:  (1 124 113)
 3:  (2 113 120)
 4:  (3 120 120)
 5:  (4 122 85)
 6:  (5 84 114)
 7:  (6 115 121)
 8:  (7 122 117)
 9:  (8 118 0)
```

The markers (> <) indicate the current node, i.e., the one presenting the report. The numbers in parentheses, in addition to reporting the node ID, indicate the RSS of the received packet on the forward path (the first value) and the RSS of the report packet on its way back (note that the destination cannot know those values, so they are shown as zeros).

By default, ODR messages show no visible trace at intermediate nodes. By executing:

```
t 1
```

on a node, you set a flag that will make the node report all ODR messages that it forwards. For illustration, this is how an instance of the above message has been reported at Node 3:

```
6342: odr #99 [1.3.8]:
 1:  (10 0 0)
 2:  (1 124 0)
 3:  (2 113 0)
 4:  (3 119 121)<
 5:  (4 120 86)
 6:  (5 87 113)
 7:  (6 114 122)
 8:  (7 121 118)
 9:  (8 117 0)
```

Note that the marker now indicates the position of Node 3 in the list of forwarders. Use:

```
t 0
```


to switch transient reports back off.

Beacons

A message (of any type) can be turned into a beacon, which means that you can send it automatically a certain number of times at certain, regular intervals. First, let us define a display message at node 10:

```
sd 12 8 0 beacon_from_10_to_8
```

The message is addressed to node 8 and doesn't ask to be acknowledged. To turn it into a beacon, enter this (at Node 10):

```
sb d 4 1000
```

The command sets up the beacon as consisting of the currently defined display message (the *d* argument) which will be sent 1000 times at 4-second intervals. Note that this command only defines the beacon. To actually start sending it, execute:

```
ba
```

which stands for **b**eacon **a**ctivate. Open the UART window at Node 8 to see the messages arrive at their destination. You will see that the number of hops traveled by the beacon varies slightly on different occasions, e.g., here is a sample output:

```
6336: disp #13 fr 10.2 ack 0 19<beacon_from_10_to_8>
6340: disp #14 fr 10.2 ack 0 19<beacon_from_10_to_8>
6344: disp #15 fr 10.2 ack 0 19<beacon_from_10_to_8>
6348: disp #16 fr 10.2 ack 0 19<beacon_from_10_to_8>
6353: disp #17 fr 10.3 ack 0 19<beacon_from_10_to_8>
6356: disp #18 fr 10.2 ack 0 19<beacon_from_10_to_8>
6361: disp #19 fr 10.2 ack 0 19<beacon_from_10_to_8>
6367: disp #20 fr 10.4 ack 0 19<beacon_from_10_to_8>
…
```

The number of hops is the value after the dot. Note that the numerical ID of a beacon message (call it the *reference number*) is increased by one each time the message is sent by its source. This is handy for cross-referencing sends and receives for a large quantity of interleaving messages. Furthermore, if you use 0 for the ID, then it will be replaced on each send with the time-stamp of the beacon's departure.

The beacon can be stopped (before its count runs down) with this command:

```
bd
```

which stands for **b**eacon **d**eactivate.

In practical cases, beacons are usually broadcast. Their periodic transmissions compensate for the poor reliability of any single case of a broadcast delivery. For illustration try this (still at Node 10):

```
sd 127 0 0 beacon_for_all
ba
```

Now you will see the beacon received by (mostly) all nodes.

Note that you do not have to redefine the beacon (the previous setting still holds), only change the display message that the beacon has been mapped to. However, you cannot modify the beacon message while the beacon is active. If you try it, you will get:

```
Locked
```

in response, and the command will be ignored.

Different combinations are possible, e.g., here is an acknowledged broadcast beacon:

```
bd
sd 126 0 1 acked_beacon
ba
```

which probably never makes a lot of sense, except for tests. To stress the network even more, you can make the interval short and the message long, e.g.,

```
bd
sd 125 0 1 this_is_a_longish_beacon_message
sb d 1 10000
ba
```

If you let the beacon run for a while in this fashion and then deactivate it (*bd*), you will see that it takes some time for the output (the acknowledgments) at Node 10 to die out. This lets us see the impact of buffering: the beacon has stopped at the source, but the network still has to deal with its legacy for a while.

The master node and its beacon

We have been using Node 10 as the primary source (and sink) of all traffic in the network. If such a node naturally occurs in an application, we call it the *master* node. You can make Node 10 the formal master by executing:

```
m 1
```

i.e., setting the master attribute of the node. You will see the node's colour in the *ROAMER* window change to blue. The node will be sending periodically and indefinitely, at 30-second intervals, a standard (broadcast) beacon telling all nodes its identity as the master. This way a node willing to send a report message to the network's sink will know which node to address. Additionally, the master beacons help TARP to organize forwarding in such a way that the paths leading to the master are (in some sense) preferred. Informally speaking, this is accomplished by making the distributed knowledge about those paths more difficult to forget.

The standard beacon messages are not reported by the nodes. In the model, you can see their traces in the *ROAMER* window, but they are completely hidden from view in the real network.

You can revoke the master status of a node with:

```
m 0
```

Also, when another node becomes the master (and sends its master beacon), the previous master node abdicates politely (as soon as it receives the beacon).

Communication parameters

The global communication parameters of the node, known as the TARP parameters, can be changed with the *sT* (**s**et **T**ARP) command. You probably don't want to change them unless you understand what you are doing. Note that you can render the node inoperable with a broken setting.

The command accepts six parameters with the following meaning (in this order):

plev The power level for transmission. The range is 0 (the minimum power) through 7

(full power). The default power level is 1.

rx This is a binary flag telling whether the receiver should be on (1) or off (0). The default setting is 1 (on). Note that by switching the receiver off you basically incapacitate the node. While it can still be used to transmit (e.g., beacons), it won't be able to receive or forward anything.

fwd This is a binary flag telling whether the node should act as a forwarder (1) or not (0). The default setting is 1. When you disable forwarding, the node will be able to transmit and receive packets, but it will not be retransmitting received packets destined to other nodes.

slack This parameter determines the degree of fuzziness (or redundancy) of TARP paths. The permissible values are 0 (no fuzziness), 1, and 2. The default setting is 1.

rte This is a number between 0 and 3 indicating how eagerly TARP should explore alternative routing opportunities. 1 means most eager, 3 means least eager, and 0 means "switched off", i.e., no exploration at all. The default setting is 2.

tlev This parameter can be 0, 1, or 2 and can be viewed as a TARP activation switch. The interpretation is: 0 – no TARP, pure flooding only restricted by the hop count limit of 10; 1 – flooding (as for 0) additionally restricted by the elimination of duplicate packets, 2 – TARP. The default setting is 2.

One potentially useful parameter to experiment with is *plev*. Reducing/increasing the transmission power may have a similar effect to increasing/reducing distances between nodes.[5]

If you ever decide to change some of the communication parameters, then you may want to make the change non-volatile across resets (power-down/power-up). This can be accomplished by writing the new set of parameters to the so-called FIM (Flash Information Memory) from where they will be reloaded on every reset. Consider this sequence of commands:

```
sT 7
Fw
```

The first one changes the first (i.e., *plev*) communication parameter to 7, i.e., selects the highest transmission power. The second command makes the change non-volatile by writing the new set of values to FIM. Now, when the node reboots, it will see that FIM contains something that looks like a correct setting of the communication parameters and it will use that setting instead of the factory defaults.[6]

You can see the current configuration of the communication parameters by issuing *sT* without arguments. Here are two more operations on FIM:

```
F
```

Lists the configuration of parameters stored in FIM, and:

[5]Generally, when some communication parameters are changed, it makes sense to change them globally for all nodes in the network. Production networks have the capacity to carry out such global changes consistently upon an OSS request passed to the network through the master node.

[6]In the model, the contents of flash memory do survive node resets, but they do not survive the termination of the model. In other words, whenever the model starts up, all flash memories at all nodes are initialized to clean.

```
Fe
```

Erases the configuration stored in FIM, thus forcing a fallback to factory defaults on the nearest reset. The factory defaults are:

```
FIM(128): plev 1 rx 1 fwd 1 slack 1 rte 2 tlev 2
```

The above line has been produced by *F* on a virgin node.

Experimenting with real nodes

This section assumes that you have received a bunch of nodes. Those nodes are labeled from 1 to 10 (there may be more). The network basically matches the model (described above), except that your deployment area will most likely differ.

You need two additional requisites to run the network:

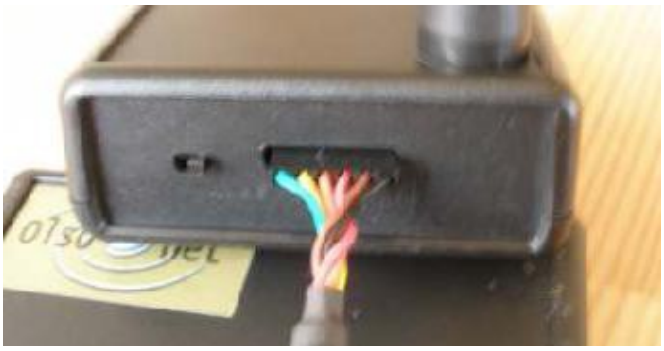
1. A few (at least one, preferably two) USB cables to connect (selected) nodes to a PC. These are FTDI FT232R 3V cables that should arrive with the boxes.
2. A terminal emulator capable of connecting to the COM ports created by the USB dongles.

Note that starting from Windows 7, the drivers for FTDI chips are located and configured automatically. The drivers for Windows XP can be obtained from the manufacturer.[7]

Regarding the terminal emulator, any reasonable program should do. The *VUEEPACK* directory includes our terminal emulator, dubbed *piter*, which can be conveniently used for this purpose.

Quick Start

Connect two nodes, say #4 and #5 to USB ports (the black wire should be on the antenna's side):



The serial parameters of the interface are 9600 bps, 8 bits, no parity, and 1 stop bit, with CR/LF for line termination. Note the power switch in “USB position” (toward the antenna). If the node is supplied from batteries, the switch should be in “battery position” away from the UART pins and the antenna. As soon as the node is powered up on default settings, the yellow LED goes ON. The other two LEDs, barely visible under the continuous yellow, signal the events of RF transmission (green) and reception (red).

[7]See: http://www.ftdichip.com/

If a terminal emulator is connected to the corresponding COM port of the PC prior to connecting the node,[8] the startup and welcome sequence should appear in the terminal window (also after every reset/power-up), looking like this:

```
PicOS v3.6/PG151104A-WARSAW, (C) Olsonet Communications, 2002-2015
Leftover RAM: 9222 bytes
CC1100: 1, 842.3MHz, 0/200kHz=842.3MHz
Netting commands:

s (set / show) disp, odr, trace, beacon, TARP:
sd [ref# dst ack <string>]
so [ref# ack hop1 ... hop9]
st [ref# dst [ dir(3) [ hops(0) ]]]
sb [ (d,o,t) freq (<64) volume (>0) ]
sT *CAREFUL* [plev rx fwd slack rte tlev]
s (show node status)

Beacon operations:
ba (activate)
bd (deactivate)

Transmit:
x(d,o,t)

FIM operations:
Fw(rite)
Fe(rase)
F (read)

m(aster) [1/0]
t(ransient) [1/0]
f(ormat) [1/0]
h(elp)
q(uit)
```

Note that, in the above output, the RF module is configured for the European ISM band. It may differ in your case. You may now begin exploring Netting, e.g., starting with the simple experiments described above.

Programming the nodes

If you have received flashable images for the nodes, and ever want to reflash a node, you have to connect to it the FET programmer (typically MSP430 FET UIF from Texas Instruments).[9] On the node's side the cable should be plugged in such that the red stripe is on the east side (assuming that the JTAG socket is facing north). Also, if you are using Windows, you have to install a programming utility, e.g., the free flash loader from Elprotronic.[10]

Imagination is the limit

As already mentioned, the message beacon allows unattended (logged) operations and diverse stress-tests. Everything will become clear with a bit of practice; here we'd like to illustrate the mechanism of morphing the (special) message reference number 0 to time stamps on the beacon.

[8]On a Unix (Linux) system the device is usually named /dev/ttyUSBx.
[9]Go to https://store.ti.com/ to look it up, if you haven't received it with the package.
[10]Go to https://www.elprotronic.com/ and click Products, then select FET-Pro-430 Lite.

For example, consider this ODR beacon amounting to a 5-fold ping-pong exchange between nodes 1 and 10. The nodes are located next to each other, which you can deduce from the very high RSS indications (this output comes from real nodes). The terminal emulator was connected to node 1. Underlined sequences correspond to user input, the remaining ones are the node's responses. The listed reports cover the history of the first beacon message (of the 12 messages to be sent altogether).

```
so 0 1 10 1 10 1 10 1 10 1 10
Odr(9) #0 ack 1 [10 1 10 1 10 1 10 1 10]
sb o 5 12
Beac o(2) freq 5 0 of 12
t1
ba
Beac o(2) freq 5 0 of 12
597: odr #596 [0.2.9]:                                        *A
 0:  (1 0 0)
 1:  (10 235 0)
 2: >(1 236 0)
 3:  (10 0 0)
 4:  (1 0 0)
 5:  (10 0 0)
 6:  (1 0 0)
 7:  (10 0 0)
 8:  (1 0 0)
 9:  (10 0 0)
597: odr #596 [0.4.9]:
 0:  (1 0 0)
 1:  (10 235 0)
 2:  (1 236 0)
 3:  (10 235 0)
 4: >(1 236 0)
 5:  (10 0 0)
 6:  (1 0 0)
 7:  (10 0 0)
 8:  (1 0 0)
 9:  (10 0 0)
597: odr #596 [0.6.9]:
 0:  (1 0 0)
 1:  (10 235 0)
 2:  (1 236 0)
 3:  (10 235 0)
 4:  (1 236 0)
 5:  (10 235 0)
 6: >(1 235 0)
 7:  (10 0 0)
 8:  (1 0 0)
 9:  (10 0 0)
597: odr #596 [0.8.9]:
 0:  (1 0 0)
 1:  (10 235 0)
 2:  (1 236 0)
 3:  (10 235 0)
 4:  (1 236 0)
 5:  (10 235 0)
 6:  (1 235 0)
 7:  (10 235 0)
```


```
 8: >(1 236 0)
 9:  (10 0 0)
598: odr #596 [1.8.9]:
 0:  (1 0 0)
 1:  (10 235 0)
 2:  (1 236 0)
 3:  (10 235 0)
 4:  (1 236 0)
 5:  (10 235 0)
 6:  (1 235 0)
 7:  (10 235 0)
 8:  (1 236 235)<                                   *B
 9:  (10 235 0)
598: odr #596 [1.6.9]:
 0:  (1 0 0)
 1:  (10 235 0)
 2:  (1 236 0)
 3:  (10 235 0)
 4:  (1 236 0)
 5:  (10 235 0)
 6:  (1 235 236)<
 7:  (10 235 235)
 8:  (1 236 235)
 9:  (10 235 0)
598: odr #596 [1.4.9]:
 0:  (1 0 0)
 1:  (10 235 0)
 2:  (1 236 0)
 3:  (10 235 0)
 4:  (1 236 236)<
 5:  (10 235 235)
 6:  (1 235 236)
 7:  (10 235 235)
 8:  (1 236 235)
 9:  (10 235 0)
598: odr #596 [1.2.9]:
 0:  (1 0 0)
 1:  (10 235 0)
 2:  (1 236 236)<
 3:  (10 235 235)
 4:  (1 236 236)
 5:  (10 235 235)
 6:  (1 235 236)
 7:  (10 235 235)
 8:  (1 236 235)
 9:  (10 235 0)
599: odr #596 [1.0.9]:                              *C
 0:  (1 0 235)<
 1:  (10 235 235)
 2:  (1 236 236)
 3:  (10 235 235)
 4:  (1 236 236)
 5:  (10 235 235)
 6:  (1 235 236)
 7:  (10 235 235)
 8:  (1 236 235)
 9:  (10 235 0)
```


In the line marked **A* (on the right), we see the first turn of the beacon, i.e., the first arrival of the message back at Node 1. The first value in parentheses (the time stamp of the report) should be compared against the second value (representing the departure time of the beacon at the source). We see that the beacon's departure must have occurred close to the boundary of second 596, because the report's time stamp is 597 (the second count has advanced), while the entire round trip (all five turns) takes no more than 3 seconds (this is the difference between the last report time and the beacon's departure time at the source). Note that the report including the line marked with **B* was made at time 598, which is the first advancement over the 597 noted at the first turn (line **A*). This lets us claim that the 8 hops from **A* to **B* took very close to 1 second. Indeed, the next advance of the time stamp is noted at line **C*, again 8 hops later, which adds credibility to our 8-hops-per-second estimate.

We see the full ODR path, and the *sb* request asks for 12 of them (so there are 11 similar sequences to follow). If you do long hop chains, especially with round trips, choose a beacon frequency as to leave up to 2-3s per hop, to allow per hop retries. The above ODR beacon made 18 hops in 3s but with weaker RF transmit power, interference, or longer distances between nodes it can take longer.

Needless to say, you may want to explore other tasks with combinations of the above. Just to put things in some practical perspective:

Display: this kind of message for multi-hop delivery is typical in ad-hoc networking applications.

Beacon with requested ACK: well approximates the functionality of an audit process with end-to-end reliability.

Trace: for setup verification, site surveys, and some sophisticated host-based location algorithms. Usually, the application cares about message delivery, not about the path it took.

ODR: doesn't exist in “regular” applications. However, it can be used for:

- approximation of tube networking, although it is a bit misleading
- verification and visualization of distributed collaboration scenarios involving specific subsets of nodes
- troubleshooting, especially if the RF channel can be suspected of problems

ODR with delays, or per node triggered functionality, may serve as a distributed operator-prescribed audit

Note that the feature is generally useless with mobile nodes.