# NS Lab 5 - Distributed Sensor Network

Junchao Wang (j.wang2@uva.nl)
Rex Valkering (r.a.valkering@uva.nl)
Daniël Louwrink (d.louwrink@uva.nl)
Daan Kruis (d.kruis@uva.nl)

Hand-in time (submit to blackboard) by Tuesday October 11 23:59
Total points: 40

**Abstract**

This assignment focuses on peer-to-peer distributed systems using UDP socket programming. You will learn how to create a network of communicating nodes by simulating a wireless sensor network.

This is a big assignment so you must form a group of two people. On the Blackboard wiki there is a page where groups must enter their names.

## Preparation

For this assignment you must use Python 2.x

You can find examples on socket programming in Python in your textbook, section 2.6.

Socket module documentation: http://docs.python.org/library/socket.html

Select module documentation: http://docs.python.org/library/select.html

## Submission

Submit your working code in an archive called lab5-<groupnumber>.zip (or 7z, or tar.gz, etc). It should contain the following files:

- Lab5-<groupnumber>.py
- gui.py (provided)
- sensor.py (provided)
- multi-<groupnumber>.py
- task5.png

Where <groupnumber> is your group number. Either group member can submit, it only has to be done once.

You must also write your full name(s) and student number(s) at the top of the files (in comments).

## Assignment

Wireless sensor networks consist of spatially distributed autonomous sensors that monitor physical conditions such as temperature, light intensity, etc. Your assignment is to program a peer-to-peer program that simulates a sensor node in a distributed system. Every node is identified by its position on a **100x100** grid. All communication is done using UDP sockets. Every node has a *sensor value*, which represents a monitored quantity.

Some skeleton code with essential definitions is given in **sensor.py**, **lab5-yourname.py and multi.py**. Since every running instance of your program simulates only a single node, you have to run multiple instances at the same time to test your code. There are some predefined command line arguments to set various variables to known values. That should make testing easier.

## Message Format

Nodes communicate by sending messages. The message format is predetermined and has the following fields:

| Type | Sequence | Initiator | Neighbor | Operation | Capability | Payload |
|---|---|---|---|---|---|---|
| int32 | int32 | (int32, int32) | (int32, int32) | int32 | int32 | float32 |

- **Type**: The message type (ping, pong, echo, echo_reply).
- **Sequence**: The initiator's echo wave sequence number (task 2).
- **Initiator**: The (x, y) position of the node that initiated a ping (task 1) or echo wave (task 2).
- **Neighbor**: The (x, y) position of the node that sent the message. This must always be filled in correctly.
- **Operation**: Echo messages can carry an operation for nodes to execute (task 3 + bonus).
- **Capability:** The initiator's range.
- **Payload**: Data related to the operation (task 3 + bonus).

The provided file **sensor.py** contains definitions for message length, message types, operation types, and two functions *message_encode*() and *message_decode*() to encode and decode messages to/from a binary format.

All tasks make use of this message format and you must make use of **sensor.py** in your code.

## GUI Interface

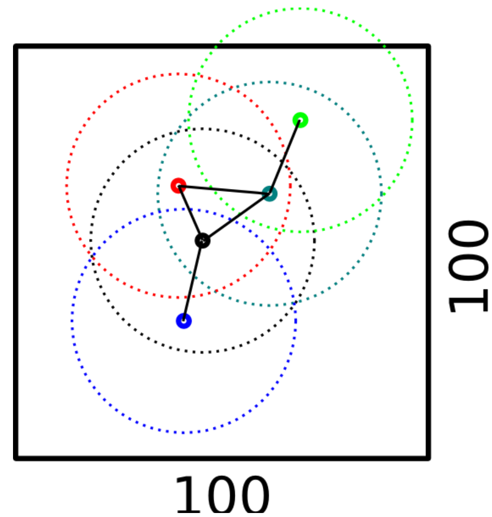
Due to the asynchronous nature of the assignment you must use the GUI interface provided by **gui.py**. <u>If you have trouble getting the GUI to work, do not wait until the last moment to let us know.</u>

In the GUI you must support the following commands:

| Command | Action | Task |
|---|---|---|
| ping | Sends a multicast ping message. | 1 |
| list | Lists all known neighbors (those who responded to the ping), with (x,y) position and IP:port address. | 1 |
| move | Moves the node by choosing a new position randomly. | 1 |
| set | Set the radius of the node(Set default 50 in your program but the radius can be changed with the range from 20 to 70 with an increment of 10) | 1 |
| echo | Initiates an echo wave. All nodes must print to the GUI what messages they receive and the initiator must print if there is a decide event (including the payloads, that will help you with tasks 3 + bonus). | 2 |
| size | Computes the size of the network. | 3 |
| value | The node chooses a new random sensor value (also print it to the GUI). | 4 |
| sum | Computes the sum of all sensor values. | 4 |
| same | Computes the number of sensors with the same value. | 4 |
| min | Computes the minimum of all sensor values. | 4 |
| max | Computes the maximum of all sensor values. | 4 |

## Task 1 – Neighbor discovery (10 pts)

To communicate, a node must first discover its neighbors (other nodes that are in range). In this assignment we will achieve this using UDP multicasting. Every node chooses a random position on a **100x100** grid, and has a wireless radius of **N, denoting the capability of the sensor (say 50 in the following figure)**.




Example of a topology found by using multicast. Nodes in range of each other are neighbors, and if a node enters or leaves the topology changes. It is possible that the network becomes disconnected.

You will need two UDP sockets: one for receiving multicast messages (multicast listener socket), and one for sending multicast messages and sending and receiving unicast messages (peer socket). Setting a UDP socket up for multicast is somewhat obscure, so we provide some code for it in **lab5-yourname.py.**

The neighbor discovery algorithm works as follows:

1. The initiator clears its list of neighbors and sends a multicast PING message (which contains the initiator's position and the current range (default=50)).
2. When a non-initiator receives a PING message it compares its position with that of the initiator, and sends back a unicast PONG message (which contains the non-initiator's position in the *neighbor* field) to the initiator if they are within sensor range.
3. When the initiator receives a PONG message it adds the position and remote IP:port address to its list of neighbors.

All nodes must periodically resend a PING message to update its list of neighbors, in case some are added or removed. This way the topology updates automatically.

Notes and tips:

- Use select() to handle sockets, and set the timeout to zero so that the GUI remains responsive.
- UDP sockets use the *sendto*() and *recvfrom*() methods instead of *send*() and *recv*().

- Note that PING messages are the *only* messages in the whole assignment that are sent using multicast.
- When in the lab, every group must choose a different multicast port so that different groups do not receive each other's messages.

## Task 2 - Echo algorithm (10 pts)

The echo algorithm is a **centralized wave algorithm** where a message is sent from an initiator and forwarded to all nodes in a distributed system. We will use this algorithm to communicate and perform computations over the entire distributed system.

The echo algorithm works as follows:

1. The *initiator* sends a unicast ECHO message to each of its *neighbors*.
2. When a *non-initiator* receives an ECHO message for the first time, it makes the sender its *father* (for this particular wave). Then it sends an ECHO message to all neighbors except its father. This forwards the wave to the rest of the network.
3. When a non-initiator receives an ECHO message for the first time and has only one neighbor (necessarily the father), it immediately sends an ECHO_REPLY message to the father.
4. When a non-initiator receives an ECHO message from the same wave again, it immediately sends an ECHO_REPLY message to the sender. This way the wave is not propagated twice.
5. When a non-initiator has received an ECHO_REPLY message from all neighbors, it sends an ECHO_REPLY message to its father.
6. When the initiator has received an ECHO_REPLY message from all neighbors, it decides (the algorithm terminates).

The operation is OP_NOOP and the payload is 0 for all messages. The initiator and sequence number must be carried unchanged by all messages, because that is how a wave is uniquely identified.

Echo waves are identified by the tuple (initiator, sequence), which is enough information to handle multiple concurrent waves. Every time a node initiates an echo wave, it increments its sequence number by one.

Note that if even *one* node does not participate in the wave, there will be no decide event (i.e. the algorithm never terminates)! But due to the way you will implement the algorithm, this does not matter: If there is no decide event a node can simply initiate another wave.

## Task 3 - Determining the size of the network (5 pts)

The next task is to discover the size of the network using the echo algorithm you have implemented in the previous task. For this you need the message fields **operation** and **payload**. Operation tells nodes which computation to perform, and payload carries data related to that computation. In this task, operation=OP_SIZE (defined in **sensor.py**).

To compute the size of the network using the echo algorithm, every node computes the partial sum of the replies it has received from its neighbors, and sends a message to its parent with payload=1 + sum(payloads received from its neighbors). Eventually the initiator receives the sum of the entire network.

In more detail, the echo algorithm is adapted as follows:

1. The initiator sends an echo with operation=OP_SIZE.
2. When a non-initiator receives a message for the first time, it makes the sender its father for this wave. Then it sends a reply to all neighbors except its father. (this step is unchanged).
3. When a non-initiator receives a message with operation=OP_SIZE for the first time and it has only one neighbor (the father), it immediately sends a reply with operation=OP_SIZE and payload=1.
4. When a non-initiator receives the same message again, it replies with operation=OP_SIZE and payload=0. Even though the non-initiator had already sent a reply before, it must do so again or the algorithm never terminates. Since the payload is 0 the final sum does not change.
5. When a non-initiator has a received a reply from all neighbors, it sends a reply to its father with operation=OP_SIZE and payload = 1 + sum(payloads received from neighbors).
6. When the initiator has received a reply from all its neighbors, it computes the size of the network = 1 + sum(payloads received from all neighbors) and decides.

## Task 4 - Computations on sensor values (5 pts)

The next task is to perform computations on the sensor values of each node, in the same way the size of the network is computed in Task 3. Each node randomly generates a sensor value, which is used in the following operations:

- Sum (OP_SUM) - Compute the sum of all sensor values.
- Same(OP_SAME) – Compute the number of the sensors with the same value of the node.
- Minimum (OP_MIN) - Compute the minimum of all sensor values.
- Maximum (OP_MAX) - Compute the maximum of all sensor values.

## Task 5 – Spawning and communicating with multiple sensors (10 pts)

You now have created a model of a wireless sensor. In science, models are often used to experiment with setups and measure their characteristics, which is easier than testing the setup in a lab. In this final task you will create a python file that builds a sensor network and performs measurements.

In the template there is a file called *multi-yourname.py*. There are two ways to invoke *multi-yourname.py*:

1. `python multi-yourname.py --number` *n* `--range` *s*

With *n* and *s* being positive integers, this program creates *n* sensors with random position (x, y) and sensor range *s*. After doing this, it outputs whether all nodes in the network are connected (e.g. the size of the network equals *n*). If so, it prints the sum, min and max of the network.

2. `python multi-yourname.py –-steps –-number n –-range s`

With *n* and *s* being positive integers, this program keeps increasing the number of sensors until there are *n* sensors, and for every step prints whether all nodes in the network are connected.

**Sensor spawning and communication**

One of the main challenges of this assignment is to a) be able to spawn multiple sensors, and b) to be able to communicate with then through your multi-yourname.py file. There are multiple ways to solve this, for example:

- You could create a file that has the same functions as the MainWindow in gui.py. This file would then not create a GUI, but send the information to your *multi-yourname.py* process using a pipe or a socket. If you choose this method, don't forget to submit this new file!
- Another way is to change your *lab5-yourname.py* such that it can also take commands through a pipe or socket, and also returns the results of these commands through a pipe or socket.

The template *multi.py* file will contain some example code which shows you how two processes can communicate over a pipe. You should know how to work with sockets by now.

**Performing a small scientific test**

Finally, you will use your program to acquire measurements for visualizing **one** of the following:

- The relation between the sensor range and the chance of all nodes in a network being connected.
- The relation between the number of nodes in a network and the chance of all nodes in a network being connected.

Visualize the above relation using the data from your measurements and save your visualization as **task5.png**. Make sure you perform enough tests to minimize randomness. Don't forget to mention the relevant variables.