Lab 6 - Distributed Sensor Network

Assistants

Cees Portegies
Frederick Kreuk
Johannes Blaser
Kyrian Maat
Niek van Noort
Lu Zhang
Zi Long Zhu

You can contact all TAs by emailing the following address: nns18-ta@list.uva.nl

Lab dates

Tuesday October 2nd 2018, Friday October 5th 2018 and Tuesday October 9th

Deadline

23:59 CEST Thursday October 11th, 2018

Total points

35

This lab may be done in pairs.

1. Preparation

For this assignment you must use Python 3.x
You can find examples on socket programming in Python in your textbook[1], section 2.7 or you can use Google. You can find the documentation for the socket module at https://docs.python.org/3/library/socket.html.

2. Submission

IMPORTANT: Your code has to meet the following requirements. Failing to meet this can result in the submission not being graded

- Your code must be PEP8 and PEP257 compliant. It should contain docstrings and pass a PEP8checker. See: http://pep8online.com
- Your code must be written in Python3. Python2 submissions will not be graded!
- Submit your working code in an archive called lab6-<groupnumber>.tar.gz. It should contain the following files and other files if needed (not including dot or Mac files):
- *lab6.py*
- *gui.py (provided)*
- *sensor.py (provided)*

Where <groupnumber> is your group number.
For this group, we expect you to sign up for groups in Canvas. One of the members of the group is assigned group leader in Canvas. We do not allow switching members of the group during the lab.

---

11. Textbook: Computer Networking: A Top-Down Approach International Sixth Edition.

Please make sure that the **group leader in Canvas submits** the work! We do not appreciate group members both submitting the work as this creates extra overhead.
You must also write your full name(s) and student number(s) at the top of the files (in comments).

3. 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 and lab6.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	Strength	Decay	Payload
int32	int32	(int32, int32)	(int32, int32)	int32	int32	float32	float32

- **Type:** The message type (ping, pong, echo, echo_reply or jam).
- **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).
- **Decay:** The initiator's decaying rate.
- **Strength:** The initiator's initial signal strength.
- **Payload:** Data related to the operation (task 3).

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. If you have trouble getting the GUI to work, do not wait until the last moment to let us know.
In the GUI you must support the following commands (which are all called as plain commands, i.e. no backslash or slash):

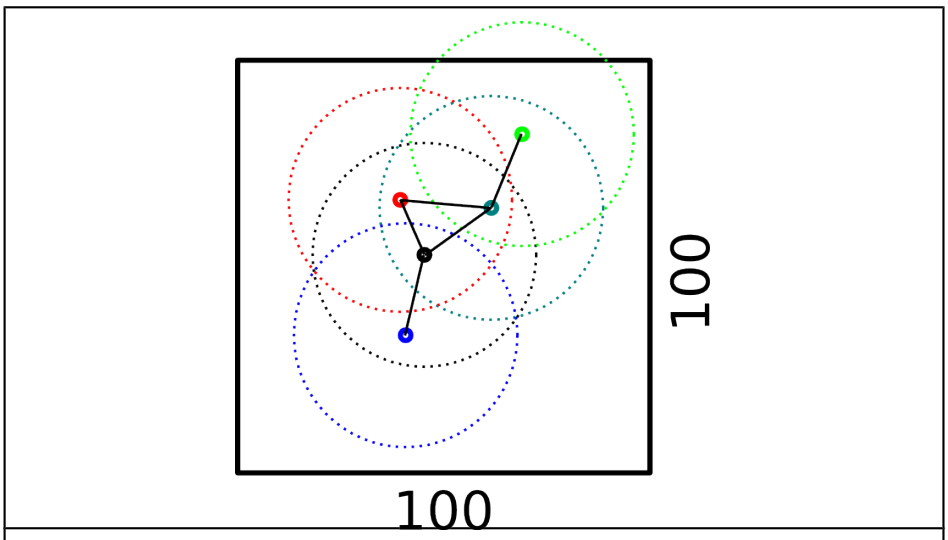
Command	Action	Task
ping	Sends a multicast ping message.	1

Command	Action	Task
list	Lists all known neighbors (those who responded to the ping) in descending order according to their distance. Print (x,y) position and IP:port address.	1
move	Moves the node by choosing a new position randomly.	1
decay <float_value>	Set the decay rate of the strength of the sensor. (Set default decay rate as 1 but the decay rate can be changed from 1 to 2, with all decimal values in between)	1
strength <int_value>	Set the strength of the sensor. (Set default strength as 50 but the strength can be changed from 10 to 100 with an increment of 1)	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
route_decay <(int_value, int_value)>	Computes the route from the sensor to the destination to minimize the signal decay. (e.g. route (10,10))	4
route_strength <(int_value, int_value)>	Computes the route from the sensor to the destination to maximize the signal strength. (e.g. route (10,10))	4
jam	Jams (Kills) the most central node if the node that issued the command is a neighbour.	5

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 an original signal strength and a decaying rate, making it possible for nodes to calculate the range covered by the initiator.

Usually the sensors have signal decay in the wireless environment. In this experiment, we use a simple exponential decaying model to simulate the signal decay of sensors. For instance, the original signal strength in position *A* is *v* and the decaying rate is *x*. The distance from *A* to *B* is *d*. The signal strength at *B* is *v-x^(d) (rounded down)*. Note that the signal strength should be no less than 0.



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 **lab6.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 initial signal strength and decay).
2. When a non-initiator receives a PING message it compares its position and range 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's range.
3. When the initiator receives a PONG message, it compares its position and range with that of the neighbor. If the initiator is in its neighbor's range, 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.
- The sensor value should at least be 0, but without negative numbers.

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 – Sensor Network Routing (5pts)

This task is to simulate a routing algorithm. After data communication, a sensor *A* can have an overview of all the sensors that can access it. To do this, you will have to maintain the position, strength and decay of every node in the network (even if they are not your neighbor). This information can be retrieved from the multicast ping messages send periodically by every node.
The final two commands, route_decay and route_strength, will be used to calculate the shortest path from sensor A to a sensor B. The metric that defines "shortest" depends on the command used.
When trying to execute one of the routing commands you will have to specify a destination sensor *B* with coordinates, formatted as (x_B, y_B). The steps you will have to implement are:

- Verify that B exists, i,e that at (x_B, y_B) there is a sensor; if not print " No sensor at (x_B, y_B)".
- If there is a sensor B at (x_B, y_B) and your command is route_decay, print the shortest path from A(x_A, y_A) to B(x_B, y_B) where the cost is calculated summing up the signal decay in the path. The shortest path is the path with the smallest signal decay.
- If there is a sensor B at (x_B, y_B) and your command is route_strength, print the shortest path from A(x_A, y_A) to B(x_B, y_B) where the cost is calculated summing up the strength of the signal at the end of each hop in the path. The shortest path is the path with the maximum signal strength.
- If there is no path from A to B, print "There is no path between A(x_A, y_A) to B(x_B, y_B) in the sensor network" in the GUI.

Task 5 – Disruption (5pts)

As a malicious attacker, the best way to disrupt a network is to attack the core of the network topology. We have learned in the previous assignment to compute the betweenness centrality and we are going to use that information to jam the central node to cripple a network. Use the shortest path algorithm (based on decay) implemented in Task 4 to determine the betweenness centrality of the nodes in the network and determine the most central node of the network. When a user issues the "jam" command, the node should check if they are neighbouring the central node. If they are a neighbour, a message containing the jam type should be sent to the node. The node receiving this message should disconnect from the network. Only a neighbour of the central node can send a jam message to the central node and the command should only aim to disable the central node! The command is thus ineffective on any other node. We want our attack to be quick and effective.

A few notes about this task:

- Because nodes can move and join at any time, the network could potentially be changing will executing the jam command. We will only test with frozen networks, as performing an attack on a changing network might prove difficult for our apprentice attacker.
- Additionally, we will also not be testing situations with multiple nodes having the highest betweenness centrality in the **same network topology.** However, disconnected networks existing in the 100x100 grid should be handled appropriately.