# Mobile Communication
# Summer term 2017

## Exercise Sheet # 3

Matthias Frank, Fabian Rump

- Release date: **June 15, 2017**
- The exercise sheet must be solved in the group registered for exercise sheet 1.
- Submission of solutions:
  - **No later than June 26, 2017 23:59 CEST.**
  - Submission is done online via the corresponding GitLab project. If you never used Git before, make sure to get some familiarity with Git before the deadline. Submissions using other media (e-mail, ...) **are not acceptable**.
  - In addition to submitting your source code to Gitlab, you solution **must contain a valid configuration to run your code on our Drone CI server**. Information about Drone CI will be provided in a separate PDF file on the MoCo website. Please make sure that your Drone CI configuration is working before the deadline.
  - A complete solution must contain:
    - **All scripts/source files** used for solving the programming exercises
    - **A single PDF file** containing your documentation/explanations for all exercise tasks.
  - You may update your solution using Git before the deadline as often as you want. Only the final submission made before the deadline will count.
- Presentation of solutions:
  - **On June 27, 2017 16:00-18:00 CEST.**
  - All group members must be present and able to explain your solution. If you are not present or fail to explain your solution properly, you will get 0 points.
  - In order to get admitted to the exam, you need to hand in and present solutions for at least 3 of the 4 exercise sheets and achieve at least 200 out of 400 total points (50%).

## Exercise 1: Proactive and reactive MANET routing protocols [30 Points]

Section 4 of the MoCo lecture introduces the topic of Mobile Ad-Hoc Networks. Specifically, we talked about different approaches of routing in MANETs.

1. What is the basic idea of proactive vs. reactive routing protocols?
2. What are the advantages and disadvantages of both ideas? Suggest exemplary scenarios in which using a proactive protocol is better than a reactive protocol and vice versa.

## Exercise 2: Implementing a Client in the MANET testbed [70 Points]

Executing real world MANET experiments requires a lot of preparation. Since this is not feasible for a MoCo exercise, we have created the "MANET testbed" for you. The MANET testbed is a TCP server which basically simulates that each connected TCP client is a MANET node. Each client is coupled to a virtual station entity and the testbed server simulates movement of these virtual nodes so that not all of the nodes are directly reachable.

Your job is to program a TCP client which connects to the MANET testbed and implements the simple MANET routing protocol below by "talking" to the other virtual nodes.

1. Carefully read all the instructions below. Plan ahead before you actually start coding.
2. Implement a client for the given MANET protocol. The server operates at:

   `hive-01.informatik.uni-bonn.de:8888`

   You can connect to the server from the computer science network. Your may also use the computer science department OpenVPN to work from home. Test your implementation thoroughly and make sure it conforms to the protocol specification.
3. Write a Drone CI configuration file which executes your client and runs for at least 5 minutes. Details about Drone CI have been published together with the second exercise sheet.

### General remarks

- All "messages" are sent as plain strings over the TCP socket. Messages from other nodes can be received by reading from the socket in your client application. A message to other nodes is sent by writing to the socket. All messages should conform to the JSON specification, see below for the exact message formats.
- Make sure to understand the syntax and the meaning of the different messages. If you are not sure, just connect to the TCP server and listen to the communication of other nodes to gain some examples.
- It may happen that other nodes contain errors or do not comply to the protocol specification. Messages may be malformed or totally missing. Make sure your implementation uses robust parsing of the received messages and is not affected by "rogue" nodes.
- The MANET testbed server "moves" the virtual nodes around, so at times it can happen that only few nodes or no nodes at all are in your vicinity. Let your client run for some minutes and wait for the topology to change.

### Your node id

When you initially connect to the testbed, the server will send you a message like this one:

```
{
  "type": "welcome",
  "id": 1234
}
```

The `id` value is very important. It denotes the address ID of your virtual nodes. You must save this ID and use it in all following messages where required (see below). When you disconnect, your ID invalidates and you have to use the ID provided with the next connection.

### Hello messages

To allow other virtual nodes in your vicinity to find you, you have to send **Hello messages** in regular intervals. Hello messages look like this:

```
{
  "type": "hello",
  "sender": <your-id>
}
```

Hello messages should be repeated every **2 seconds**. Obviously you will receive the Hello messages sent by other virtual nodes. Make sure to parse this messages accordingly and to keep track of the nodes in your neighborhood with a suitable data structure. When you receive no more Hello messages of one of your neighbors for **more than 10 seconds**, this node should be treated as "lost" and should be removed from your neighbor list.

### Topology messages

In order to distribute information about the network topology, each node must periodically send a **Topology message**. Topology messages contain information about the neighborhood of each node and must be flooded through the network.

```
{
  "type": "topology",
  "sender": <your-id>,
  "sequence": <sequence-number>,
  "neighbors": [<neighbor-id-1>, <neighbor-id-2>, ...]
}
```

To ensure that flooding does not continue endlessly, you **must include a sequence number** with each Topology message. The sequence number should be initialized at 1 and increased by 1 for each sent message, i.e. the first Topology message contains `"sequence":1`, the second Topology message `"sequence":2`, and so on.

When you **receive a Topology message**, you have to check:

1. Is this the first occurrence of this message? Due to flooding your will receive the same message multiple times. It is **critical** that you keep track of the **most recent sequence number used by each node**, i.e. you have to save the sequence number for each node id!
2. If (and only if!) the message is new, you should save the contained information and update the sequence number counter for this node.
3. If (and only if!) the message is new, you should re-send the message **without alteration** in order to make the flooding process work.

Every node should originate a new Topology message after **5 seconds**. When you receive no more Topology messages of some node for **more than 10 seconds**, this node should be treated as "lost" and should be removed from your topology cache.

### Routing table calculation

From the received Topology messages you know about the neighbors of each node. If we treat this information as the edges within a graph of the nodes, we can derive a graph of the complete network topology. For our example protocol, we want to do hop-by-hop routing. Thus you have to find out for each known node which one of your neighbors is the suitable next hop on the shortest route towards the destination node. There are common algorithms to do so (e.g. Dijkstra), but you can also derive your own algorithm.

Implement the routing table calculation and make sure that the routing information is refreshed in regular intervals to ensure it is up to date. Routing information will be needed for forwarding data messages, see the next step.

### Forwarding data messages

Finally, once the topology is discovered and the routing tables are computed, it is time to relay some **data messages**. Data messages will be sent with the following format:

```
{
  "type": "data",
  "sender": <sender-id>,
  "receiver": <receiver-id>,
  "next_hop": <next-hop-id>,
  "data": "<some-message>"
}
```

When you receive a data message and your ID matches the `receiver` value of the message, you are the intended receiver and don"t need to do anything (except for example printing the message to the console).

When you receive a data message and your ID matches the `next_hop`, you are responsible for forwarding the message. You need to query your routing table for the intended next hop for the `receiver` node and update the `next_hop` field accordingly. The message should then be re-sent.