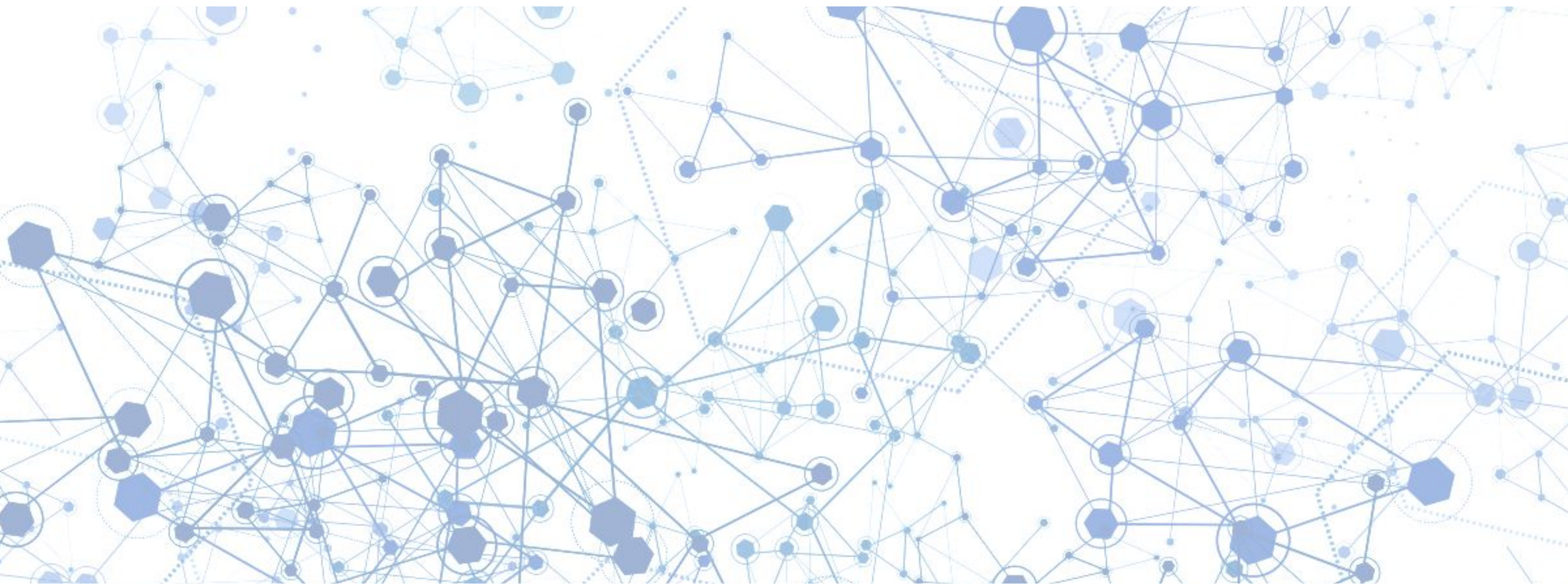
Project Report

Ad Hoc On-Demand Distance Vector routing protocol

IoT - 2017/2018

POLITECNICO
MILANO 1863



NICOLA CASTALDO - 875019

The protocol

In this project the task was to design and implement a **protocol** similar to AODV, in which each node updates its **dynamic Routing Table**; thus, the last one, specifies the **next hop** of an outgoing packet for each possible destination. When the route to a given node is not present, the node willing to send has to broadcast a **ROUTE_REQ** including the destination node it wants to reach.
This request is then broadcasted by the other nodes until it reaches the wanted one, which in turn must reply with a **ROUTE_REPLY** to each node from which the broadcast was received. This ROUTE_REPLY must be re-sent back to the origin, being modified at each node with an addition of one "**HOP**". When it finally reaches the source of the request, this last node can update its Routing Table if the ROUTE_REPLY shows a better path to the destination. After 1 second it can transmit to the wanted destination its **DATA** message with a random payload, checking on the Routing Table the next node of the communication. ROUTE_REPLY messages received after 1 second must be discarded, as well as duplicated ROUTE_REQ; moreover each Routing Table entry is valid for just 90 seconds.

The choice of the OS

I chose to implement the protocol with **Contiki-2.7** and I tried to get advantage of the possibility to use "**protothreads**" to accomplish the **concurrency** and the **event-driven** sequence of actions of the nodes.

My approach

First of all I **studied** the AODV protocol (https://tools.ietf.org/html/rfc3561) and I reviewed the slides of this course about ZigBee Routing, in order to achieve a good understanding of how the protocol is implemented and works.

Actions

Then I defined the main **actions** that a node has to accomplish and how each **event** could be generated and handled.
Each node had to be able to:

- *Send* **new** DATA message every 90 seconds (*UNICAST*) to a **random** destination, with a **random** payload
- *Receive* DATA messages whether it is the destination or not; in the last case *resend* them
- *Send* **new** ROUTE_REQ messages (*BROADCAST*) if there is no route to a given destination
- *Keep track* of the ROUTE_REQ already sent (or re-sent)

- *Respond* to a ROUTE_REQ with a **new** ROUTE_REPLY message if it is the destination of the request
- *Receive* and accept the ROUTE_REPLY if it directed to it, *resend* the reply adding 1 hop if it is not, following the backward path
- *Update* the Routing Table for every ROUTE_REPLY received before 1 second
- *Keep track* of Routing Table entries deleting the ones created more than 90 seconds before

Packets

To tackle the problem one step at a time, I started to design the structure of the **packets** in terms of **header** and **payload**, considering the main aspects of each one (see *aodv.c* file). I also defined some helper functions in order to translate each message (*struct*) into a packet (*string*) and vice versa.

Tables

Then I defined as **structs** the entries of the Routing Table and the **Routing Discovery Table**. I decided to use the last one (as in the legacy protocol) in order to keep track of each ROUTE_REQ created and received and to update it for each *significant* ROUTE_REPLY.

Connections

Once messages and tables row were created, I studied from the examples on the Contiki-2.7 Documentation the correct approach to handle **connections** and their corresponding **callbacks**.

I decided to use different **channels** for each type of messages:

- 1 channel for the **broadcast** connection (ROUTE_REQ)
- 2 channels for the **unicast** connections (DATA and ROUTE_REPLY)

Processes

- I defined a **process** that could initiate the connections and I linked to them **3** different **callback** functions, one for each type of packet.
- **Another** process had to be responsible of creating the data for a random destination: it had to generate the data and check into the Routing Table whether there was or not a **next hop** for the selected destination; in case there wasn't it had to call another process to perform the broadcast. The *data_sender* process could also send DATA coming from other nodes: because of this I had to **distinguish** whether the *waking* **event** was coming from the internal timer (new DATA) or the *data_callback* (other nodes DATA)
- Thus, a **third** process had to handle the ROUTE_REQ message and save it into the Routing Discovery Table, then broadcast it. This process could also be called by the ROUTE_REQ **callback** function in the case that ROUTE_REQ was not present in

the Routing Discovery Table. In order to accomplish this, I used the "*process_post*" function offered by Contiki, which wakes a process in *waiting state*. As "*data"*, the parameters to pass to the process, i defined a **new struct** (*rreq_info*): that could carry the needed information about the ROUTE_REQ to broadcast.

- Moreover **another** process had the responsibility to decrease, at each second, the field "*EXP*" of each **active** entry in the Routing Table and **deactivate** them when the said field came to 0.

Testing and additional info

I tried to be as clear as possible in the **log** (*printf functions*), also because I had to test the project to assure its correct operating principle.
I decided to log only a few messages, corresponding to the main actions of each node: **creation** of DATA, **re-sending** of DATA, **reception** of DATA and **lack** of a possible **route** to perform the sending.
In order to get **more** information about the behaviour of each node, I defined an **additional process** that, whenever the *sky button* is clicked, toggle the **debug state** of the node. When the debug is active, it is possible to have much more info about the internal flow of the code.

The **results** showed that in case of a large number of nodes, in order to communicate to a "**distant**" node, each element could not receive any ROUTE_REPLY in just 1 second (modifying it to 3 or more seconds the results were better).
Moreover, better with a "never expiring" Routing Table but still depending on the topology, each node could also **fill** the **whole** Routing Table and never create ROUTE_REQ anymore.
Anyway, using the topology defined in the "*project proposal*", nodes **more connected** could be able to send/receive DATA messages (as well as ROUTE_REPLY and ROUTE_REQ) much more better than nodes on the edges and poorly connected.

I decided to add some **Leds** activation, in order (in Cooja) to see the different action that each node is performing.

In addition I included a **random** "*initial_delay*" to reduce conflicts; if not, each node would have started to communicate at the same time, leading to worse results.

Moreover there is the possibility to **change** the **number** of **nodes** expected, the **seconds between** a ROUTE_REQ and the ROUTE_REPLY, the **duration** of the Routing Table **entries** (also settable to "infinite", i.e. never expiring rows).