Course Project:
Event-Triggered Control

Low-power Wireless Networking for the Internet of Things
2021–2022

In this project, you will implement a low-power wireless networking protocol for event-triggered control, capable of reacting to unpredictable events by rapidly and reliably collecting sensor readings at one or multiple entities (typically referred to as sinks or controllers in this context) and distribute actuation commands over a multi-hop wireless network.

You are provided with an application template that contains the basic structure of the requested implementation. You will run both simulations and testbed experiments to assess the performance of your implementation. You will follow a research methodology similar to the one commonly adopted by the low-power wireless research community.

System structure and node roles. Similar to a real scenario, nodes in your wireless network will be organized in a control loop, holding a predefined role:

1. *Sensors* monitor the process under study by periodically sensing the environment and evaluating a local triggering condition (one for each sensor node). Upon detecting a violation of their triggering condition, sensor nodes instruct all the other sensors in the network to communicate their updated readings to the controller, effectively providing this node with an updated snapshot of the system status.

2. The *controller*—one per control loop—is a special entity, where the control logic resides. It is responsible of *i)* collecting and processing sensor readings from all the sensor nodes in the network, *ii)* generating new actuation commands, in order to bring the system back to steady state, and *iii)* communicate such actuation commands to all the actuators.

3. *Actuators* wait for commands from the controller and execute them to change the state of the system. In this project, to limit the amount of nodes to emulate in Cooja, sensor nodes coincide with actuators.

4. *Forwarders* are nodes whose only role is to route traffic between controllers, sensors, and actuators, extending the physical coverage of the system.

For simplicity, in this project a *single* control loop is considered. In real systems (e.g., industrial plants), multiple control loops can share the same wireless network, requesting for strategies to minimise, if not totally avoid, mutual interference.

Message types and communication patterns. A centralized control loop includes several phases. Upon acquiring a measurement that requires controller intervention, a sensor node generates and distributes an *event message*, which must be reliably delivered to all the sensors. Upon detecting the event, the sensors start transmitting their most recent sensed data towards the controller, so that it can exploit the full view of the system status to compute new actuation commands. Finally, if and when needed, the controller communicates such updated actuation commands to some (or all) of the actuators.
To achieve this behaviour, your protocol should combine different message types and communication patterns:

- To enable data collection, the protocol must build and maintain a tree rooted at the controller by means of `TREE` beacon messages.

- In the *event phase*, when an event is detected, one (or multiple) `EVENT` message should be propagated network-wide, to reach all sensors potentially across multiple hops.

- In the *collection phase*, nodes should forward sensor readings towards the controller. Previously learned topology information could be exploited to let `COLLECT` messages reach the controller.

- Finally, in the *actuation phase*, `ACTUATION` commands are sent downwards from the controller to the designated sensor-actuator node(s). To accomplish this task, the same routing tree used in collection can be exploited, but downwards, as detailed in the dedicated section.

To efficiently support the various protocol phases above, different traffic patterns can be exploited and multiple design decisions are possible. It is your responsibility to identify the ones providing the best trade-off in terms of reliability and energy efficiency to your system, and present them in the final report.

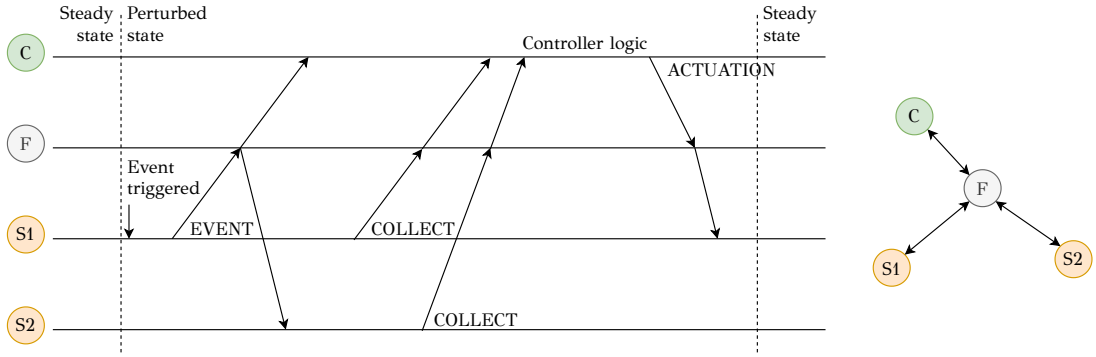

Figure 1: Example of event, collection, and actuation with one controller, one forwarder, and two sensors. The actuation command is sent only towards node S1 in this case.

Hereafter, we are offering some basic suggestions on how you could implement the different phases of your protocol. Nonetheless, we encourage you to explore additional strategies to enhance the performance of your solution, devoting particular attention to the fact that the network topology could suddenly change due to node failures.

Sensor readings, steady state, perturbed state, and controller logic. In this project, sensor readings are simulated in software and provided as part of the project template. A dedicated function is called periodically to update the sensor readings by a random amount and yield the updated reading to the application. You can expect sensor readings to grow monotonically over time.

When the system is considered stable, the maximum difference between any two sensor readings in the network is lower than a predefined threshold T: only TREE messages should be transmitted, to update the routing tree. On the contrary, as soon as such difference grows too large, the system becomes unstable and is in need of controller intervention. The triggering condition for sensors is therefore defined based on T: when the reading of a sensor node is greater than T, an event is generated and a collection phase starts. Indeed, another sensor in the network could have a sensor reading value equal to 0, invalidating the constraint on the maximum difference discussed above. To find out whether or not the system is stable, the controller needs to collect all readings. Upon receiving a COLLECT message from all the sensors, the controller (i) computes new control actions, which in this simple scenario translate to either resetting the sensed value to 0 or assigning some sensors a new triggering threshold greater than T; (ii) decides which actuators it needs to communicate with (e.g., those whose sensor readings violate the constraint, and those whose readings violate constraints because of other resets), and (iii) transmits the actuation commands (ACTUATION messages) to such subset of nodes.

Tree construction and routing topology. The controller is in charge of initiating floods of TREE messages to (re)construct the collection tree. To propagate the flood while reducing the chance of collisions, nodes should retransmit the received TREE beacon message after a small, random delay, like in Labs 6 and 7. The hop count is the primary metric we suggest you to follow to select nodes parent(s).

Event-triggered data collection. The sensor code we provide implements a special logic that notifies sensor nodes when they are supposed to generate an EVENT message. Whenever a sensor node detects an event, a dedicated EVENT message is disseminated to inform all sensors that their updated readings (sensor value and threshold) need to be timely collected at the controller.

After sending the EVENT, or after receiving and disseminating it, sensors should transmit their readings towards the sink of the collection tree (the controller). Forwarders relay the received COLLECT message upwards. Unicast COLLECT messages directed towards the node's selected parent(s) could be exploited to ensure forwarding progress during collection. COLLECT messages should include the address of the source of the reading, as discuss next.

Downward forwarding for actuation commands. Differently from Labs 6 and 7, the data collection phase could be also used by nodes to populate downward routing tables, stored locally. Whenever a node receives a COLLECT message, it associates the source of the sensor reading (found in the payload) with the sender of the received message. Nodes should then leverage such information when forwarding ACTUATION commands:

the sender of the COLLECT message becomes the next forwarder in the downward path from the controller to the specific source. Your implementation should ensure that no routing loop is created.

Transmission failures and node failures. Your implementation should include mechanisms to reduce the chance of collisions and to improve the overall reliability of your control loop. These mechanisms should not only counteract packet losses due to collisions, but also ensure that the system maintains reasonable performance if nodes become unavailable. Your protocol should timely react to node failures, e.g., re-transmitting packets along new routes.

By simulating a node failure in Cooja you can easily assess the robustness of your solution when a node suddenly stops working. In the presentation for this project, you can find additional details on how to simulate node failures by leveraging Cooja button sensor events.

As a simplifying assumption, you can expect *only forwarders* to experience malfunctions, and at most one node failure per experiment.

Testbed experiments. Simulations are a very useful tool for protocol design. Nonetheless, testbed experiments are key to understand the effect of the environment on our solutions. Once your implementation achieves satisfactory results in Cooja, you can run the same tests in the testbed, using Zolertia Firefly nodes. You may need to make small adaptations of your implementation for the testbed.

Testbed experiments are optional. However, the maximum mark for a project submitted with no testbed experiments is 27/30.

Protocol evaluation. You should analyze the performance of your protocol in terms of *i)* packet delivery rate (PDR), focusing on the reliability of sensor readings collection and commands dissemination; and *ii)* duty cycle (DC). Towards this end, you may use `parse-stats.py`, a Python script that parses your Cooja or testbed `.log` files and computes the above mentioned metrics.

We encourage you to study the impact of the various design decisions you have made, and discuss how you balance reliability and energy efficiency in your system.

The core of your analysis should focus on Cooja experiments with no simulated node failures, which you can run without limitations in a controlled setting ensuring repeatability.

A few testbed experiments would be enough to showcase the performance of your solution in a real environment. No node failure needs to be emulated in testbed experiments.

Report. After the performance evaluation, you should write a report with a brief description of the design decisions behind your protocol logic, your implementation details, the results of your performance evaluation, and a concise summary of the findings. Your report should clearly describe your experimental methodology (i.e., how you run your experiments) and how the metrics are computed. To show your results, we encourage you to add tables or make meaningful plots, e.g., showing the node DC and PDR.

Implementation Notes.

- **Rime, RDC and MAC layers.** The system should be implemented at the Rime layer. The MAC layer should be CSMA, while RDC should be ContikiMAC. You may configure these layers through the available settings to improve reliability and duty cycle.
- **Application Interface.** Your protocol implementation must provide: *i)* `etc_open()` function to open all needed connections and start the protocol, *ii)* `etc_close()` function used to simulate node failure, and *iii)* callback functions for the application. The project template includes the declaration of these functions in the file `etc.h`. Any change to the return values and arguments of these functions should be described in the report.
- **Simulating node failure.** In this project, node failures should be simulated in firmware. This can be accomplished by simply closing all open connections, ignoring any incoming packet, and ending all Contiki processes that use those connections to transmit. Failure simulation is triggered with the user button of the node, that can be "pressed" in Cooja.
- **Logging.** The provided application already includes several `printf` functions to log events, collected data and actuation commands. You should not modify these output strings, as they are used to automatically evaluate the performance of your protocol.

Code Template. To implement the project, we provide several files to simplify your development.

- **app.c** a simple application that starts the protocol and implements the callback functions. The application uses the API your protocol should provide.

- **etc.h** the header file of your Rime layer protocol with the minimum API you should provide. This API is used by the top-level application (`app.c`). You may extend the header file with other functions or data structures as required.
- **etc.c** the source file of your protocol with stub functions. You should build your logic in this file, although you may use additional source files as you may need.
- **tools** we provide tools to enable duty cycle estimation in both simulated Tmote Sky nodes in Cooja and Zolertia Firefly nodes in the testbed. The files within the `tools` folder MUST NOT be changed.
- **testbed** we provide some bash scripts and a `experiment.json` file to facilitate the evaluation of your protocol in the Firefly testbed.
- **Makefile** to compile your code. You may add new source code files to the Makefile.
- **project-conf.h** to configure your application and protocol.
- **Cooja simulation files and node roles** to test and evaluate your protocol. You are provided with different Cooja topologies (`scenario1`, `scenario2`) to test your implementation. For each Cooja configuration, and for testbed experiments as well, topologies come with pre-assigned node roles (see `app.c`). Results for all topologies should be included in the report. You may also define your own simulation files in addition to those provided.

Rules of the game: How to (and how not to) work on the project

- The project is individual. Due to the structure of the course, students are strongly encouraged to deliver it before the beginning of the second semester. However, students that are unable to meet this deadline are encouraged to contact the instructor well in advance, especially if they have already taken the written exam successfully.
- You should submit *(i)* the Contiki source code and *(ii)* a brief report with the description of your solution, your evaluation results, and your conclusions. The report must be in English.
- You should demonstrate that the project works as expected using the Cooja simulator and/or the testbed in front of the teaching assistants and/or the instructor.
- In the view of the current pandemic, you may be required to present your project via Zoom instead of discussing it in presence with the instructors. These logistics aspects will be clarified when setting up the date and time of your presentation.
- The code **must** be properly formatted. Follow style guidelines (e.g., Contiki code style).
- You **must** contact through e-mail the instructor (gianpietro.picco@unitn.it) **and** the teaching assistants (davide.vecchia@unitn.it, matteo.trobinger@unitn.it) well in advance, i.e., at least a couple of weeks before the presentation. If you have time constraints, it is up to you to make them known to the instructors in due time. Remember that the instructors have their own availability constraints and may not be able to accommodate exactly the date/time you need.
- Both the code and the documentation must be submitted in electronic format via email at least three days before the meeting. The documentation must be a single self-contained PDF. All code must be sent in a single tarball consisting of a single folder (named after your surname) containing all your source files and analysis scripts.
- The code **should not** be published in GitHub or any other online service/tool. While we encourage students to become part of the open source community, we believe sharing your project code can help other students develop their implementation, which can be unfair to fellow students and result in plagiarism (see below).
- The project will be evaluated based on the technical implementation (correctness and efficiency) and the report quality, which should demonstrate the student understanding of the problem at hand.

Plagiarism is not tolerated. Students whose project is partially copied/adapted from other students' projects will undergo disciplinary measures. Depending on the gravity of the plagiarism, these may involve reporting the student to the highest disciplinary bodies of the university, therefore possibly jeopardizing the offending student's academic career. If you are afraid you may not complete your project, get in touch with the instructors. Remember: an incomplete project will be considered more positively than one we discover to have been partially or totally copied from someone else's project.