Prelab: Contiki-NG and Cooja

General Instructions

In this series of labs, we will look at advanced concepts of communication in the context of the Internet of Things. We will focus on short-range wireless mesh networks for small, embedded platforms. We will target resource-constrained hardware, in our case featuring a 4 MHz CPU and 10 kB of RAM. All implementations will be done in C.

We divided this course into three main blocks: a mandatory prelab, three labs during which you will implement wireless protocols for IoT devices, and a project during which you will be given more freedom. Please check the lab instructions slides for a complete view on how the labs will be performed. This prelab is **mandatory**, and is a **Pass or Fail** examination. You have one week to submit your written answers and screenshots as a PDF document. Your document **must be submitted on iLearn.** You cannot take part of the final exam if you did not deliver your prelab on time. In the document, all exercises in the instructions must be answered unless stated otherwise. Your answers should be concise; you don't need more than a few lines to answer each question. Questions that needs to be answered are within boxes.

You should complete the labs in groups of two persons — use the group you've created in iLearn!

Internet of Things

The Internet of Things is a new paradigm having gained a spectacular momentum since the past decade. The Internet of Things, also shortened as IoT, is an umbrella term covering many applications:

- Smart homes: window shutters, thermostat and heaters, and door locks are connected and controllable from your smartphone, anywhere on the planet;
- Smart factories: industrial robots are equipped with sensors and actuators, allowing the factory to be fully automated and quickly adapt the supply chain to new requirements in a matter of minutes;
- Smart cities: connected meters allow electricity and water providers to adapt energy production to the actual demand of the population, your car can communicate with red lights to determine the most efficient path to your destination;
- Smart textiles: Clothing items incorporate sensors to monitor your physical activity.

When we refer to IoT devices, we often speak of small, resource-constrained devices. These platforms often operate with limited energy, and are usually powered by batteries or small solar panels. Computing power is limited, with microcontrollers often integrating single

core CPU running at a few MHz. Memory is also scarce, with flash and RAM capped to a few MB.

With such limited hardware, most applications must run bare-metal (without an underlying operating system), or must use specific OSes with limited functionalities. Thus, programmers have to use low-level languages to take full control of the platform.

Part 1 - Advanced C Programming

This course assumes that you have an advanced understanding of programming. All implementations will be done in C. We require you to have taken a course featuring C programming in your previous studies, or that you've been using the C language in past personal projects. Additionally, you must have taken a course on operating systems, to fully grasp memory management.

In this first part of the prelab, we test your knowledge of C.

Question 1. Answer the first 10 questions of this online test: `https://www.geeksforgeeks.org/c-language-2-gq/pointers-gq/`
Report your score in your report.

If you scored less than 60%, we highly recommend you to take a C course (or check the videos linked on iLearn). Programming embedded programs requires a deep understanding of memory management and computation complexity. In the following labs, you will have to constantly deal with pointer operations, C structures, integer overflows, and value padding and packing.

Question 2. What is the stack? The heap? What happens when the stack overflows?
Question 3. What does the **static** keyword do in C?
Question 4. Assume a 16-bit unsigned integer K. Using only binary operators (>>, <<, &, |), how can you write the integer operation $K/16$? How can you write the operation $K\%16$?

Part 2 - Installing Contiki-NG with Docker

During the labs, we will use a special operating system called Contiki-NG (`https://github.com/contiki-ng/contiki-ng/wiki`). Contiki-NG is an open-source OS specially designed for resource-constrained devices, typically used in the IoT ecosystem.

Since IoT devices and your personal desktop or laptop do not share the same architecture, we will need to install some tools. Compiling code for different hardware (also known as cross-compiling) requires additional software, commonly referred as toolchain. Contiki-NG

provides a Docker image comprising the entire toolchain.

Note: While there are some other alternatives to install the toolchain (native installation, Vagrant image), we recommend to use Docker for its stability and simple usage.

Instructions. Follow this tutorial to install the required toolchain: `https://github.com/contiki-ng/contiki-ng/wiki/Docker`

For MacOS users, additional steps are required. First, you need to allow connections from network clients in the settings. Then, you must use the command **xhost +127.0.0.1** before starting XQuartz and the Contiki-NG docker image (see `https://fredrikaverpil.github.io/2016/07/31/docker-for-mac-and-gui-applications/` for more information).

Whenever you want to run or compile your code, you first need to start the Docker image with **contiker bash** or **xhost +127.0.0.1;contiker bash** for MacOS. You can also use **contiker cooja** if you wish to open the simulator straight away, as we will see in the next section.

Part 3 - Running Hello World in Cooja

At this point, you should have Docker and the Contiki-NG docker image installed on your machine. Using **contiker bash**, you should be able to open a terminal featuring the compiling toolchain, as well as an access to your Contiki-NG repository. During the labs, we will use a simulator to run our IoT applications. Contiki-NG provides its own simulation environment called *Cooja*. Cooja is developped in Java and is also open-source. Follow these instructions to run your first application in Cooja.

Instructions. Run an application in Cooja: `https://github.com/contiki-ng/contiki-ng/wiki/Tutorial:-Running-Contiki%E2%80%90NG-in-Cooja`
Please select the **sky mote** during the mote selection.
Note: For MacOS users, resizing the Cooja window can help with visual bugs. It is also possible to run Cooja natively if you have Java 1.8 or older installed.

At this point, you should see a node running in the simulator, and running the application code.

Check the application code **examples/hello-world/hello-world.c**.
Question 5. Modify the printf() call to print your name in addition to the Hello World text. Run the simulation for a minute, and take a screenshot of Cooja. The printed text should be visible in your screenshot. Include it in your report.

Part 4 - Timers

Many IoT devices are powered with batteries. Sensors deployed in the Amazonian forest to monitor wildlife or sensors deployed across the Alps to detect avalanches are usually placed in remote locations, where maintenance is often tedious or even impossible. As such, IoT applications must consume as little energy as possible in order to perform for long periods of time, sometimes up to a decade.

One way to save energy is to limit code execution to periodic events, and put the device to sleep between executions. Contiki-NG provides several ways to enable periodic executions using timers.

Read the documentation on timers in Contiki-NG: `https://github.com/contiki-ng/contiki-ng/wiki/Documentation:-Timers`
Question 6. When should you use etimers? When should you use rtimers?
Check the application code **examples/hello-world/hello-world.c**.
Question 7. Modify the printf() call to print the system time and your name in addition to the Hello World text. The printf should be called first after 2 sec, then 4 sec, then 8 sec, etc. Run the simulation for a minute, and take a screenshot of Cooja. The printed text should be visible in your screenshot. Include it in your report.

Part 5 - Basic Networking

As you might recall from the lectures, an IoT device is defined as a resource-constrained device, augmented with computing and communication capabilities, sometimes able to sense or interact with its environment. During these labs, we will use the IEEE 802.15.4 standard for industrial short-range communication. As example, Zigbee devices, including the smart home appliances from companies like Ikea, Philips HUE, and many others, use the physical and MAC layer of IEEE 802.15.4.

More specifically, we will use the CSMA-based MAC layer of the wireless standard. Contiki-NG provides a simplified API to send and receive messages, through a network layer called *Nullnet*. As the name indicates, Nullnet does very little. You will have a direct access to the MAC layer, and your goals during the following labs will be to create low-latency and reliable communication over large (multi-hop) networks.

Read the documentation on communication using Nullnet in Contiki-NG: `https://github.com/contiki-ng/contiki-ng/wiki/Documentation:-NullNet`
Look into the example available in **examples/nullnet/nullnet-broadcast.c**.
Question 8. How do you receive messages in Nullnet? How do you send packets?
Question 9. What is the difference between broadcast and unicast?

Optional: To Go Further

Important: This part is not mandatory to pass the prelab. This is the kind of question we will ask you in the following labs. You can take a look and see if you believe this course is for you.

It is now time to design your first, simple wireless communication.

We will start with a simple networks composed of **5 (five) Tmote Sky motes**, placed close to each other. A company wants you to design a decentralized, global counter, to count the number of people entering a shopping mall. Each node must have a 16-bit unsigned integer counter (tip: set the counter variable as static). Every **10 seconds**, a node increments its counter by one and broadcasts its value. When a node receives a counter value, it selects the **maximum** between the received and local counter value, and sets its local value accordingly.
Optional question. From the **examples/nullnet/nullnet-broadcast.c** code, imagine how you could implement the simple protocol presented above. **You do not need to implement and run the protocol!** Simply, explain in your report how this protocol could be implemented (list the functions you would need to use, and the important part your code would require). What problems could arise? Will your counter be consistent everywhere?

During the following labs, you will need to design, justify, implement and evaluate such protocols in Cooja.