Guy Ezer & Roee Wodislawski

Prof. Sivan Toledo

Advanced Computer Systems, TAU

April 2018

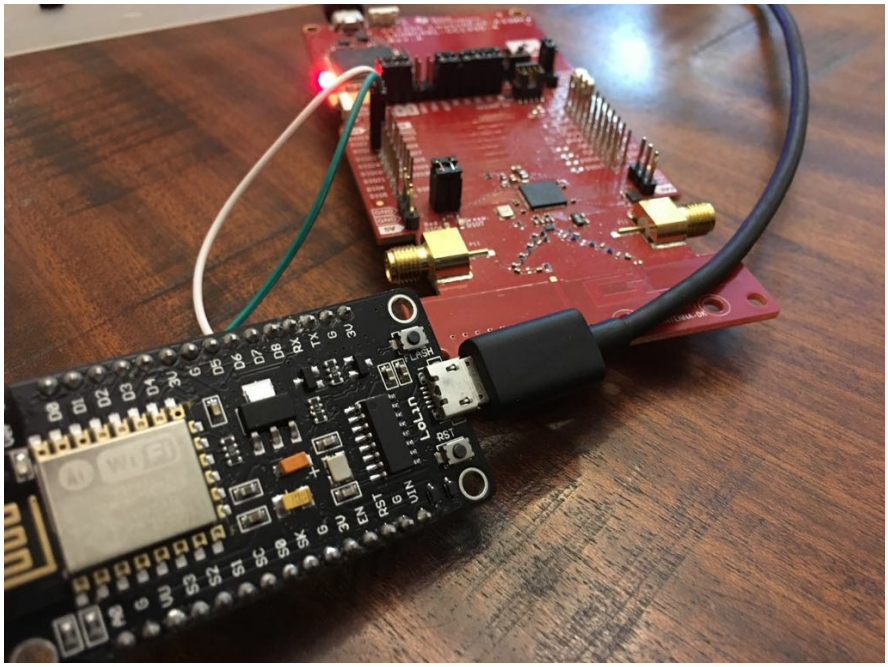
Scalable Sensors Network

Our project's main goal is to create an infrastructure for a "sensors network" in which multiple endpoints communicate with a central server, sending information about attached sensors.
The server role is to to gather information and manage the different endpoints by responding to their queries.
Examples for fitting applications are access control and motion detection.

The project is designed with scalability in mind. The communication protocols and software libraries were chosen to allow addition of features, new sensors and expanding the communication protocol with ease.

This document will describe the project's components - from the main board and the external WiFi chip to the software packages we used to reach our goals. It will also include details on how we combined these components together.

Link to Youtube video of project demonstration.



General structure

As mentioned, the network consists of multiple endpoints and a single server.
The main component in each endpoint is the CC1350 LaunchPad board, which we used in class.
The network relies on WiFi for communication between the endpoints and the server and since this board doesn't support WiFi, each endpoint also includes NodeMCU ESP8266 board for WiFi communication.
The server is written in python and can be run in any machine which holds the server static IP in the local network.

CC1350

This board gathers information from the different sensors attached to it, then creates queries with said information to send to the server. The board can also receive responses back, which will guide it in operating those sensors and in general activity.

Using the TI-RTOS kernel, each sensor can be represented with a task whose purpose is to create the queries and if needed wait for the server response and act upon it.

In order to communicate with the server, the board has a UART connection to the ESP8266 board, which acts as a tunnel between them.

When waking up, the board waits for a magic from the ESP8266 through the UART connection, registers as an endpoint and then starts execution of the sensors tasks.

Code snippet for encoding a node message (query) in the board:

```
NodeMessage nodeMessage = NodeMessage_init_zero;
ButtonPressQueryMessage buttonPressQueryMessage = ButtonPressQueryMessage_init_zero;
pb_extension_t nodeMessageExt;

// Create button press query
nodeMessage.type = NodeMessage_Type_BUTTON_PRESS_QUERY;
buttonPressQueryMessage.nodeId = nodeId;
nodeMessage.extensions = &nodeMessageExt;
nodeMessageExt.type = &ButtonPressQueryMessage_key;
nodeMessageExt.dest = &buttonPressQueryMessage;
nodeMessageExt.next = NULL;

// Encode button press query
oStream = pb_ostream_from_buffer(buffer, BUFFER_SIZE);
if (!pb_encode(&oStream, NodeMessage_fields, &nodeMessage))
{
    while (1);
}
data_len = oStream.bytes_written;
```

ESP8266

We use an ESP8266 in order to communicate with the server. The ESP8266 serves as a 'tunnel' between the CC1350 and the server - it is wired to the CC1350 UART pins, and forwards tcp/uart messages between the two.

We are using a NodeMCUv3 ESP8266 model.
A few bash scripts were written in order to automate the flashing procedure using esptool. See the installation doc on the documentation dir for more information.

The ESP8266 runs the Arduino Core.
We discuss our choice of Arduino Core and other alternatives in the following sections.

Arduino port

At first we used MicroPython, but after encountering multiple UART problems we moved to the Arduino port which provides:

- Easy Arduino-like API
- Debug-by-printf capabilities
- Large development community

  In the newer IDE versions, we had to change to flashing toolchain inside the Arduino IDE in order for the flashing to work, which took a lot of work. Other than that we had a good experience with using the Arduino port.

Micropython and other alternative

Originally, our initial choice was Micropython.
Micropython (uPython) is a light implementation of the Python3 programming language that is optimized to run on MCUs.

Pros:

- Relative ease of developing once uPython is up and running on the MCU
- Filesystem support - configuration can be done easily
- Web interface (webrepl) which makes remote maintenance possible
- Large community - which results in stable images, documentation and online help

Cons:

- Threads are not supported on the ESP8266 port (but asynchronous io is supported)
- Not all UART ports are accessible via the API, in fact, only one UART port is accessible
- Debugging is hard - because of the two cons above, in addition for no PDB (python debugger) support

The UART issues took too much time to handle so we moved to the Arduino port.

Other alternatives we looked over

1. **Using a ready AT-CMD firmware**
    - Relatively easy 'out of the box' solution
    - Configurability is a problem
    - Can not modify the firmware
    - No ability to store logs
2. **Using the SDK**
    - Most control we can get over the MCU
    - no 'out of the box' support for NodeMCUv3 - we've configured it overselves
    - API was inconvenient

UART Issues

The ESP8266 we were using has three UART ports. Of those three, UART1 only has a TX line and UART0 is not accesible by uPython. uPython API also maps, wrongly, UART0 to UART2 and UART2 to UART0. That took us a bit of time to understand. In addition, uPython maps REPL's (read eval print loop) standard input and output to UART2 - meaning that any use of the REPL - including the webrepl - can not be done when using UART2. That made debugging much harder.

Unlike MicroPython, the Arduino core provides choise of multiple IO pins to function as UARTs - so the UART configuration, alongside debugging was much easier.

Server & Protocol

We wrote a simple TCP server to handle communication with the different endpoints.
The server is written in Python3 and is based on the asyncio module, which allows it to handle connections asynchronously.

As taught in class, the asynchronous model is better for servers, since it requires smaller memory footprint than threads (no stack is needed) and easier client, traffic & resource management.

After creating the TCP connection, each endpoint registers and receives a node id. After, this endpoint will use this node id to identify when further querying and receiving information back from the server. The query format allows the endpoint to specify type (according to the specific related sensor) and add the relevant info. The mechanism that enables the server and the endpoint to communicate using the defined protocol will be described in the following section.

To allow this flow of operation, the server includes components for registering nodes and handling incoming messages. According to the message type, the handler will forward it to the relevant component (e.g. LedManager) and will send back its response.

The server design is flexible, in order to ease the addition of new sensors with appropriate handlers or making changes in the protocol or flow of operations.

Code snippet for handling a node message (query) in the server:

```
65      def handle(self, message):
66          self._log("trying to parse node messsage...")
67          nodeMessage = NodeMessage()
68
69          try:
70              nodeMessage.ParseFromString(message)
71          except:
72              self._log("failed to parse node message")
73              return None
74
75          self._log("parsed node messsage, handling...")
76
77          try:
78              node_message_key_and_handler = self._node_messages_key_and_handler_map[nodeMessage.type]
79              node_message_key = node_message_key_and_handler[0]
80              node_message_handler = node_message_key_and_handler[1]
81              node_message_extension = nodeMessage.Extensions[node_message_key]
82              result = node_message_handler(node_message_extension)
83              self._log("handled node message")
84              return result
85          except:
86              self._log("failed to handle node message")
87              return None
```

Protobuf

Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data. Google developed Protocol Buffers for use internally, with the design goals of emphasized simplicity and performance.

Protobuf is useful in developing programs to communicate with each other over a wire or for storing data - after defining once how the data should be structured once, you can use generated source code to write and read your structured data, using a variety of languages.

Numbered fields in proto definitions obviate the need for version checks - you can update your

data structure without breaking deployed programs.
New fields could be easily introduced without the server & client needing to know about all the fields.

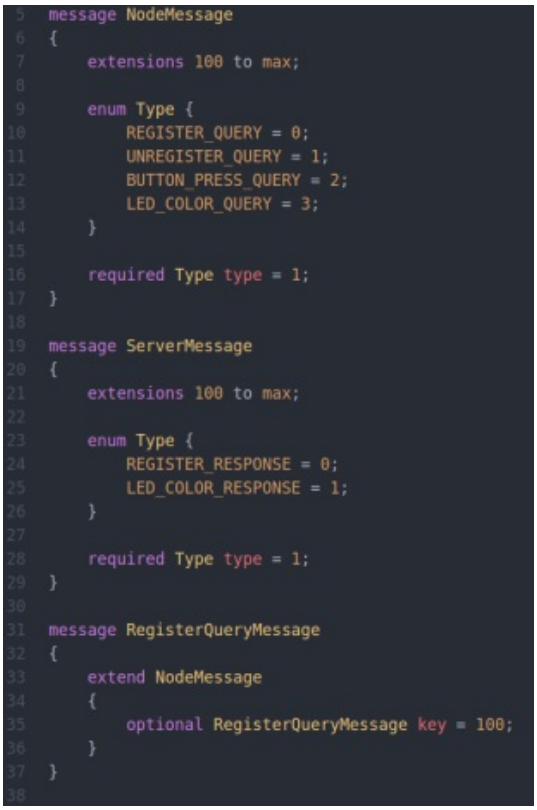
Comparing to other popular serializing methods (XML, JSON), Protobuf is considered to be simpler, less ambiguous, easier to use programmatically and generates smaller data.

Protocol buffers currently officially supports generated code in Java, Python, Objective-C, C++, C# and more.

User of Protobuf should create a file with .proto extension, where Protobuf syntax is used to define the required protocol rules. After creating such files, source code file in the supported programming languages can generated using protoc, a generator which can be downloaded or built from source.

In our project, we needed a way to define a protocol to serialize data which both client and server will be able to parse, even though they will be written in different languages and run in different environments.
On top of that, one of our main design goals is scalability which is a main feature of Protobuf. These requirements together with the general advantages mentioned before make Protobuf perfect for our needs.

Code snippet from the definition of our protobuf protocol:



```
message NodeMessage
{
    extensions 100 to max;

    enum Type {
        REGISTER_QUERY = 0;
        UNREGISTER_QUERY = 1;
        BUTTON_PRESS_QUERY = 2;
        LED_COLOR_QUERY = 3;
    }

    required Type type = 1;
}

message ServerMessage
{
    extensions 100 to max;

    enum Type {
        REGISTER_RESPONSE = 0;
        LED_COLOR_RESPONSE = 1;
    }

    required Type type = 1;
}

message RegisterQueryMessage
{
    extend NodeMessage
    {
        optional RegisterQueryMessage key = 100;
    }
}
```

When we wanted to design our protocol with polymorphism hierarchies (messages based on another messages), we stumbled upon the article Protocol Buffer Polymorphism.
The author explores different option to implement polymorphism mechanism using Protobuf syntax, providing interesting influences of each. Our protobuf code was written based on the suggested mechanism.

More information and documentation of Protobuf can be found in Protocol Buffers developers site.

Nanopb

Nanopb is a plain-C, small code-size implementation of Protobuf. It is suitable for use in microcontrollers, designed to fit other embedded, memory restricted systems.
All code include needed to use is pure C runtime, with small code size (2–10 kB depending on processor) and small ram usage (typically ~300 bytes).
It does not depends on dynamic allocation - everything can be allocated statically or on the stack.
It is structured in a way which allows usage of either encoder or decoder alone in order to cut the code size in half.
Includes support for most Protobuf features and an extensive set of tests.

There are some limitations, as some speed has been sacrificed for code size
and reflection (runtime introspection) is not supported - you can't request a field by giving its name in a string.

Since our development board does not allow usage of the officially Protobuf supported programming languages,
Nanopb allows us to enjoy the benefits of Protobuf while meeting our environment requirements.

In order to use Nanopb, some static (not generated) c headers and implementation files are needed to be included in the project.
Nanopb also provides an extension to protoc (Protobuf code generator, mentioned above) which is responsible for the generation the c source files.

More information about Nanopb can be found in it's Homepage.

Expanding the project

Here we detail some ideas and thoughts which we had while working on the project. These would bring features, abilities or more robustness to the infrastructure, but due to lack of time those were not implemented:

TinyFrame

Since the UART communication between the cc1350 and the esp8266 is unreliable some sort of data-link protocol is needed.
We looked over some data-link protocols such as PPP (point to point protocol; has a large overhead) and SLIP (has a smaller overhead but is too bounded to TCP/IP).

An interesting alternative is the TinyFrame library.
TinyFrame is a library for parsing data frames to be sent over a serial interface. The library provides a high level interface for passing messages between two peers: Multi-message sessions, response listeners, checksums and timeouts are all handled by the library.

Using TinyFrame will make the sensor management easier because its support of multi-message sessions.
TinyFrame has a relative small memory footprint as well.

TinyFrame also provides a python port, meaning it can be ported to the esp8266 (using MicroPython), but the python port is not mature yet.

Since we lacked time to stabilize the Python port, we did not use TinyFrame. We still included this section because we think it would be a great addition.

Robustness

There are more places in which more robust mechanisms can take place, from the server being able to manage the endpoints better (which come and go) to the main board that is currently written with synchronous UART API, which we now find to be sensitive and less flexible than its asynchronous counterpart.

Also, the current access to the server is through static IP address, it may be more robust to use DNS but it would require the relevant setup.

Sensors implementation

Originally, we wanted to include implementation of sensors to showcase the infrastructure abilities. The candidates were PIR sensors for motion detection or RFID sensors for access control.
The PIR sensors we tried to communicate with (P/N 28032-ND) did not work well and returned many false-positives so we couldn't make a functioning tracking system.
As for RFID, we obtained a RFID-RC522 module but because of its complex API (with no relevant, existing code to use) we decided it would be out of the project scope to implement usage.

We also thought about adding support for TCP/IP based components, such as cameras, via the ESP8266 to be accessible to the server.

Currently, we use the board's built-in button as a sensor and light the built-in leds according to the server responses.

Accessibility

Improving general logging capabilities of the boards and server, with optional web interface for detailed information from endpoints or exporting an API for the sensors network in order to allow applications based on this infrastructure to be developed.