Lab 6 Networking and Internet-of-Things (IoT)

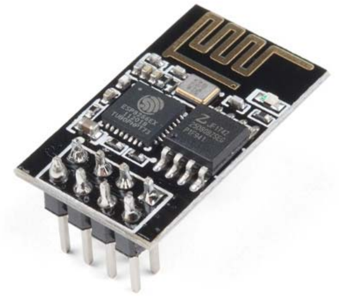
Goals
- Integrate ESP8266 Wifi module,
- Develop a layered communication system,
- Design and implement a hardware/software interconnect protocol,
- Implement remote terminal access to the interpreter over the network,
- Interface two Launchpads to realize remote procedure calls (RPC).

Review
- Textbook Sections 9.1 through 9.4,
- ESP8266 Wifi module documentation on the class website.

Starter files
- **`RTOS_Lab6_Networking`** (starter project for Lab 6)
  - **`can0.c`** and **`can0.h`** containing CAN0 driver (in **RTOS_Labs_common** folder), not used in this lab.
  - **`esp8266.c`** and **`esp8266.h`** containing ESP8266 Wifi module driver on UART2 (in **RTOS_Labs_common** folder).
  - **`WifiSettings.h`** containing Wifi access point settings for the ESP8266 module (in **RTOS_Labs_common** folder).
  - **`Lab6.c`** with the bare-bones OS and Testmains.

Background

The ultimate goal of this lab is to network your two Launchpads such that they can communicate with each other using standard internet (TCP/IP) protocols over Wifi connections. One Launchpad will be a client and the other a server, where the client will be able to make remote procedure calls (RPC) to call software functions on the server.



Launchpads will be connected to a Wifi network and the internet using **ESP8266** Wifi modules. Specifically, we will use ESP-01S Wifi modules made by Ai-Thinker, which incorporate a UART interface, radio frontend and antenna around an ESP8266 chip by Espressif. The ESP8266 chip implements a complete TCP/IP network stack (in firmware running on an embedded microcontroller) on top of a basic Wifi hardware interface. The Sensorboard has a socket that connects an ESP-01 module to UART2 on the TM4C, including a dedicated voltage regulator to drive ESP power with higher current needs. All communication with the ESP8266 from the TM4C will be via UART2 using the ESP8266's 'AT' command syntax. The **`esp8266.c`**/**`.h`** driver implements a basic set of commands and provides a high-level API to interact with the ESP module. However, you will need to integrate the **`esp8266.c`**/**`.h`** driver with your OS. Specifically, the driver internally uses three FIFOs for UART2 communication with the ESP. Similar to UART0 in Lab 2, busy-waiting FIFOs have to be replaced with FIFOs that use semaphores for synchronization between background interrupt handlers and foreground threads.

To connect the ESP8266 to a Wifi network, you need to provide the access point SSID and password/key in **`WifiSettings.h`**. The **`esp8266.c`**/**`.h`** driver provides APIs to connect the ESP to the access point, which will assign it an IP address (via DHCP). From there on, the driver provides routines for the ESP to act as a TCP client or server: an ESP can either initiate TCP client connections to servers on the internet using their IP address or name (the ESP will internally perform DNS lookups if necessary) and port, or the ESP can be switched into server mode where it will listen for incoming connections on a specific port. The starter project includes examples for the Launchpad to fetch weather information from the internet as a TCP client sending HTTP requests to a public web server (*Testmain2*) or to act as a simple web server itself that understands basic HTTP 'GET' requests and serves an HTML page with a form that users can submit via a web browser to output on the Launchpad's display (*Testmain3*).

In the first part of this Lab, you will add capabilities to your OS to access its command interpreter remotely through Wifi and the internet. You will need to add a remote terminal server thread that will listen for incoming connections on a chosen port (e.g., the default port 23 for remote terminal/telnet servers) and will launch an instance of your interpreter that communicates with a remote terminal via the ESP whenever a new TCP connection to the server is made. This interpreter instance should run concurrently to the default interpreter on UART0. In order to do so, you must have a version of the interpreter that is able to receive inputs and send outputs via the TCP connection instead of UART0. One option is to create a copy of your interpreter in which all UART I/O is replaced with corresponding **ESP8266_Send**/**ESP8266_Receive** calls. Alternatively, you can modify your interpreter such that it allows I/O redirection to either UART0 or ESP. There are different ways of doing this. You can overload all I/O functions in the interpreter with wrappers that call corresponding UART0 or ESP routines depending on a parameter passed to the interpreter, and then create two wrapper tasks that launch interpreter instances with different parameters. If you want to achieve the same without having to pass parameters into every interpreter function, I/O redirection can be done depending on the context, i.e. by which thread ID (**OS_Id()**) the **Interpreter()** routine is called. For any I/O routines used by the interpreter, overload them with a wrapper like this:

```
int TelnetServerID = -1;
void Interpreter_OutString(char *s) {
  if(OS_Id() == TelnetServerID) {
    if(!ESP8266_Send(s)) { // Error handling, close and kill }
  } else {
    UART_OutString(s);
  }
}
```

The default interpreter instance on UART0 can then be launched as before. For remote terminals, create a *TelnetTask* that is launched on every incoming server connection, sets the server ID for redirection and launches an instance of the interpreter with I/O redirected:

```
void TelnetTask(void) {
  TelnetServerID = OS_Id();
  Interpreter();
  ESP8266_CloseTCPConnection();
  OS_Kill();
}
```

In all cases, you should add a command that allows exiting the interpreter, which will then close the TCP connection. For the default interpreter on UART0, exiting can be ignored or the **Interpreter()** can simply be respawned in an endless loop.

The main part of the lab will then be to connect two Launchpads and develop a remote procedure call (RPC) protocol that will allow a client Launchpad to call and execute functions remotely on the server Launchpad. This can simply be done by letting the client Launchpad send interpreter commands to a remote terminal server running on the other Launchpad. However, your approach will need to support passing parameters and sending return values back. The interpreter is primarily designed for interactions of a human with the Launchpad. Rather than emulating remote calls by constructing appropriate interpreter command sequences and parsing of the human-readable output produced by the interpreter, you may want to develop a simple custom RPC protocol optimized for machine-to-machine communication instead. Your RPC setup should at minimum support the following remote function calls:

- **ADC_In** – Get a remote ADC sample
- **OS_Time** – Get the time on the remote Launchpad
- **ST7735_Message** – Output a message on the remote display
- **LED_xxx** – Allow remote LEDs to be toggled or turned on/off
- **eFile_xxx** – Remote file system access
- **exec_elf** – Remotely execute a given program on the disk

You will need to add interpreter commands on the client side to trigger such remote procedure calls. In addition, the client Launchpad should cycle through triggering the red, green and blue LEDs on the server Launchpad when pressing the SW1 button on the client.

If you want to connect two Launchpads that are not on the same local network, you may run into the issue that the server will not be reachable from the client, e.g. if the server Launchpad is behind a firewall or on a local home network that assigns only a private IP address not visible on the public internet. This would normally require enabling port forwarding in the home/network router and firewall on the server side. If you are not comfortable with or don't know how to do that for your router, an alternative is to use SSH port forwarding and tunneling via a local laptop/PC and an ECE LRC machine. See the Hints at the end of the document for how to set this up.

Preparation

1) Make sure that you have SSH (if you need to use SSH tunneling and port forwarding) and either 'telnet' or netcat/'nc' (for testing of TCP connections) installed on your laptop. For Windows, you can either use Putty or the command-line OpenSSH and Telnet clients that come with Windows. The OpenSSH client is part of the "Optional Features" in Windows 10 (under Windows Settings). The Telnet client can enabled by going to "Turn Windows features on or off" under "Programs and Features" in the Windows Control Panel (not Windows Settings).

2) If you want to use SSH tunneling and port forwarding, make sure that your ECE LRC account is active, that you have support for the UT VPN installed and that you can SSH into one of the LRC machines using the UT VPN (SSH connections are otherwise blocked from outside).

3) Integrate the **`esp8266.c`**/**`.h`** driver into your OS. In particular, replace the UART2 FIFOs with a semaphore-based implementation.

4) Make sure to adjust **`WifiSettings.h`** and set SSID_NAME and PASSKEY to the Wifi network name and password/key of the desired access point.

5) Run the *Testmain2* and *Testmain3* from the starter project. Study their code to make sure you understand their operation and use of the ESP8266 driver API. Debug output including and echoing of 'AT' commands exchanged with the ESP is shown in the Launchpad's UART0 terminal. In particular, write down the IP addressed assigned to your Launchpad (output of the 'AT+CIFSR' command). For Testmain3, you can interact with the web server on the Launchpad by opening the address http://<*Launchpad_IP*> in any web browser (tested with Firefox and Chrome).

6) Design your protocol for remote procedure calls (RPC) between Launchpad. In particular, define and document the syntax and semantics of commands and responses exchanged between client and server.

Procedure

1) Implement a remote terminal (telnet) server for your Launchpad. Test the server by opening a TCP connection to the Launchpad's IP on the chosen port from a laptop or PC on the same local network, and then issuing commands to the remote interpreter:

`telnet` <*Launchpad_IP*> <*port*> or **`nc`** <*Launchpad_IP*> <*port*> (or using Putty)

2) If client and server are on different (home) networks, make sure that the server is reachable from the client's local network, if necessary using port forwarding on the router or via SSH tunnels (as described under Hints below). Test the remote connection from a laptop/PC on the same local network as the client by opening a TCP connection to the remote Launchpad and issuing remote interpreter commands.

2) Implement the RPC server. Test the server both from a laptop/PC on the same local network and from a laptop/PC on the client's network (if different) by opening a TCP connection (see above) and manually sending RPC requests and observing the server responses.

4) Implement the RPC client. Add interpreter commands to issue RPC requests and output responses. Setup buttons on the client to send RPC requests that toggle and cycle through LEDs on the server.

4) Test the complete client-server system by issuing commands on the client interpreter or pressing client buttons to perform actions on or query the server.

Checkout (show this to the TA)

Demonstrate remote terminal access to a command line interpreter on each of your Launchpads from a local laptop or PC. Two copies of the interpreter should be accessible via the regular UART and a network connection at the same time. Further demonstrate remote procedure call functionality from the client to the server Launchpad. You should be able to demonstrate reading of ADC and time values, toggling of LEDs, displaying of messages and launching of user programs on the remote server Launchpad from the client.

Deliverables (exact components of the lab report and lab submission)

A) Objectives (1/2 page maximum)

B) Hardware Design (none)

C) Software Design (documentation and code)

1) Documentation of your RPC protocol
2) Remote terminal server
3) RPC client and server
4) High level software system (new interpreter commands)

D) Measurement Data

1) CPU utilization of the server OS when idle (not serving a request).

E) Analysis and Discussion (1 page maximum). In particular, answer these questions

1) Under what situations/conditions does the MAC protocol used by Ethernet and Wifi (CSMA/CD) work best/worst, and why?
2) How does your Launchpad get its IP address? And how does it obtain the IP address of the "api.opnweathermap.org" server in *Testmain2*?
3) What is the difference between UDP and TCP communication? More specifically when should we use UDP and when should we use TCP?
4) What limitations does your RPC protocol have, e.g. in terms of supported function call types? What is the overhead of your protocol?

Hints

1) The ESP8266 driver currently only supports one active client (outgoing) or server (incoming) connection at a time, i.e. no two simultaneous outgoing connections as a client, no outgoing client connection while some incoming server connection is active (in practice, one should be a client or server at any time, but can switch between at runtime), and no two incoming server connections at the same time (if a client connects to the server while another connection is already active, the second connection will simply be ignored and not served).

2) To connect two Launchpads that are on different local networks and are not directly reachable from each other, you will need to setup SSH tunneling by following the following steps. This works by setting up two tunnels, one each on the client and server side that route and connect everything through an ECE LRC machine. Here is what the final setup will look like:

```
               Laptop1 -- ssh -L --> LRC <-- ssh -R -- Laptop2
Launchpad1 --> Laptop1 == tunnel ==> LRC == tunnel ==> Laptop2 --> Launchpad2
(Client)                                                           (Server)
```

First, setup the server side forwarding:

1. On the server Launchpad:
   a) Start the server code and note the Launchpad's local <*server_IP*> and <*server_port*>
2. On a local laptop/PC that is on the same local network as the server Launchpad:
   a) Test the local connection to the server Launchpad:
      `telnet` <*server_IP*> <*server_port*> or **`nc`** <*server_IP*> <*server_port*> (or using Putty)
   b) If you are outside of the UT campus, start the UT VPN to be able to SSH into the ECE LRC machines.
   c) SSH into an LRC machine (e.g.' kamek') using remote port forwarding, i.e. setting up an <*LRC_port*> on the LRC machine that will listen for and forward any incoming connections to the Launchpad:
      `ssh -R` <*LRC_port*>**`:`**<*server_IP*>**`:`**<server*_port*> **`kamek.ece.utexas.edu`**
      Or, in Putty, setup an SSH connection to the LRC machine with SSH remote tunnel options as in shown in Figure 6.1(a) below. Pick a random <*LRC_port*> above 1024. If you get an error that the port is already in use (already "bound"), use a different port.
   d) From within the SSH session/terminal on the LRC machine, test the forwarding to the server Launchpad:
      `nc localhost` <*LRC_port*>
      This should open a session/connection to the server on your Launchpad.

Then setup the client side forwarding:

3. On a local laptop/PC that is on the same local network as the client Launchpad:
   a) If you are outside of the UT campus, start the UT VPN to be able to SSH into the ECE LRC machines.

- b) SSH into the same LRC machine as above using local port forwarding, i.e. setting up a <*local_port*> on the laptop/PC that will be forwarding any incoming connection to the <*LRC_port*> on the LRC machine, which in turn will forward the connection to the server Launchpad through the other tunnel:

  ```
  ssh -L *:<local_port>:localhost:<LRC_port> kamek.ece.utexas.edu
  ```

  Or, in Putty, setup an SSH connection to the LRC machine with SSH local tunnel options as shown in Figure 6.1(b). <*local_port*> can be any port available on the laptop/PC, e.g. the same as <*server_port*>
- c) Make sure that the firewall on the laptop/PC allows incoming connections from the local network to <*local_port*>. For Windows, approve any such requests by the Windows firewall that pop up when starting the SSH session.
- d) Open a command prompt/window and get the laptops/PC's <*local_IP*> address on the local network:

  ```
  ipconfig
  ```
- e) Test the connection to the remote server Launchpad via the two tunnels:

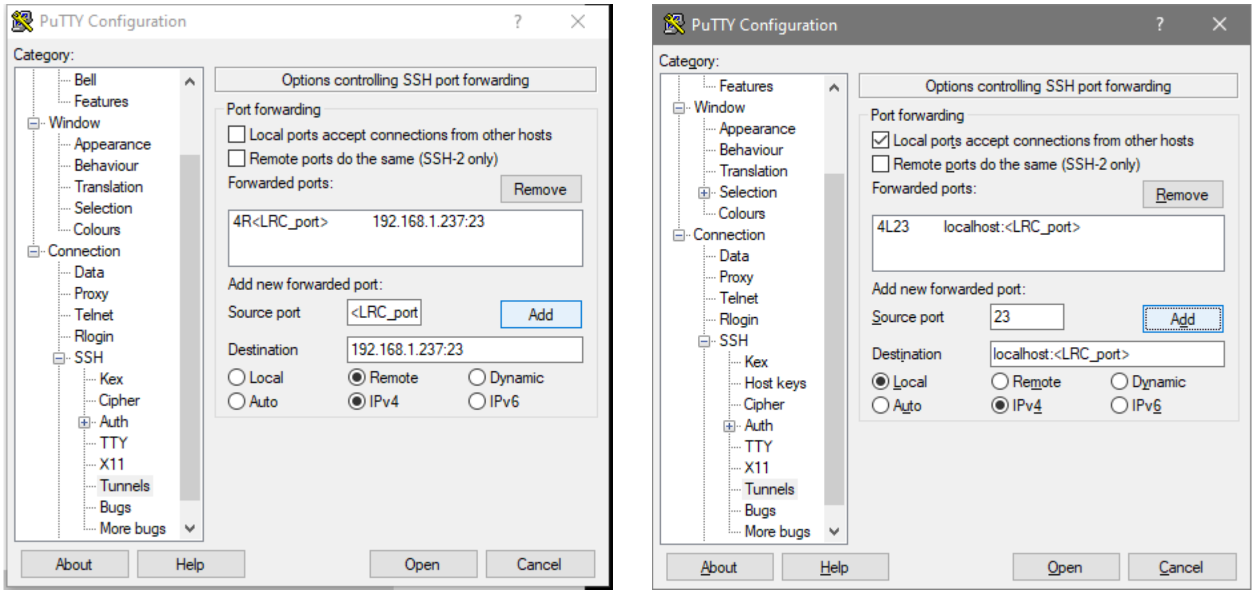
  `telnet localhost` <*local_port*> or `nc localhost` <*local_port*> (or using Putty)

  This should open a session/connection to the server on the remote Launchpad.

4. If you have a second local laptop/PC/RaspberryPi/etc that is on the same network as the client Launchpad:
   - a) Test the connection through the forwarding laptop/PC:

     `telnet` <*local_IP*> <*local_port*> or `nc` <*local_IP*> <*local_port*> (or using Putty)

     If this fails but the equivalent test above on the forwarding laptop/PC worked, then the firewall on that laptop/PC is interfering.
5. Finally, on the client Launchpad:
   - a) Setup the client to connect to <*local_IP*> and <*local_port*>



Options controlling SSH port forwarding

Port forwarding

☐ Local ports accept connections from other hosts

☐ Remote ports do the same (SSH-2 only)

Forwarded ports:

4R<LRC_port> 192.168.1.237:23

Add new forwarded port:

Source port <LRC_port

Destination 192.168.1.237:23

○ Local ● Remote ○ Dynamic

○ Auto ● IPv4 ○ IPv6

(a) Server side

Options controlling SSH port forwarding

Port forwarding

☑ Local ports accept connections from other hosts

☐ Remote ports do the same (SSH-2 only)

Forwarded ports:

4L23 localhost:<LRC_port>

Add new forwarded port:

Source port 23

Destination localhost:<LRC_port>

● Local ○ Remote ○ Dynamic

○ Auto ● IPv4 ○ IPv6

(b) Client side

Figure 6.1: Putty SSH tunnel configurations for port forwarding (using 192.168.1.237 as <server_IP> and port 23 as both <server_port> and <local_port> in this example).