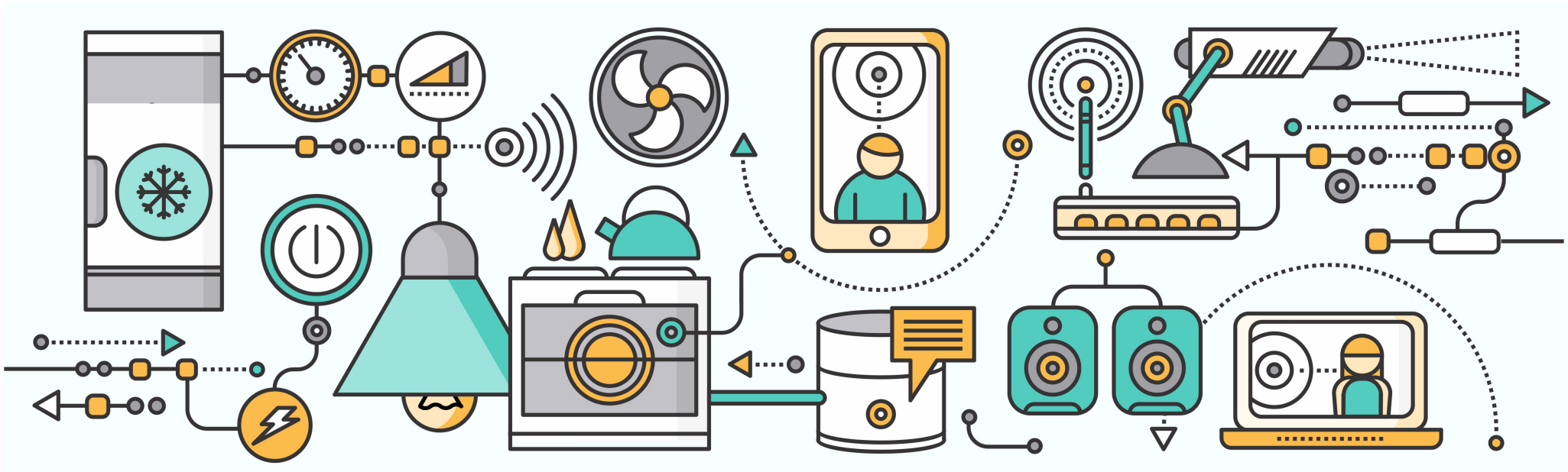
ENME 441
Mechatronics and the Internet of Things



TCP/IP Sockets and HTTP Requests

Sockets

Sockets offer a programming interface for inter-process communication based on a simple **client/server model** to send messages across a network:

- Sockets have long been central to the Internet (core API unchanged since 1971 with ARPANET).
- Python API is based on Internet sockets (BSD or Berkeley sockets).
- ENME 441 will focus on sockets via Transmission Control Protocol (TCP) to connect devices with IPv4-formatted addresses (TCP/IP), see:

Python `socket` Module

client

server

client

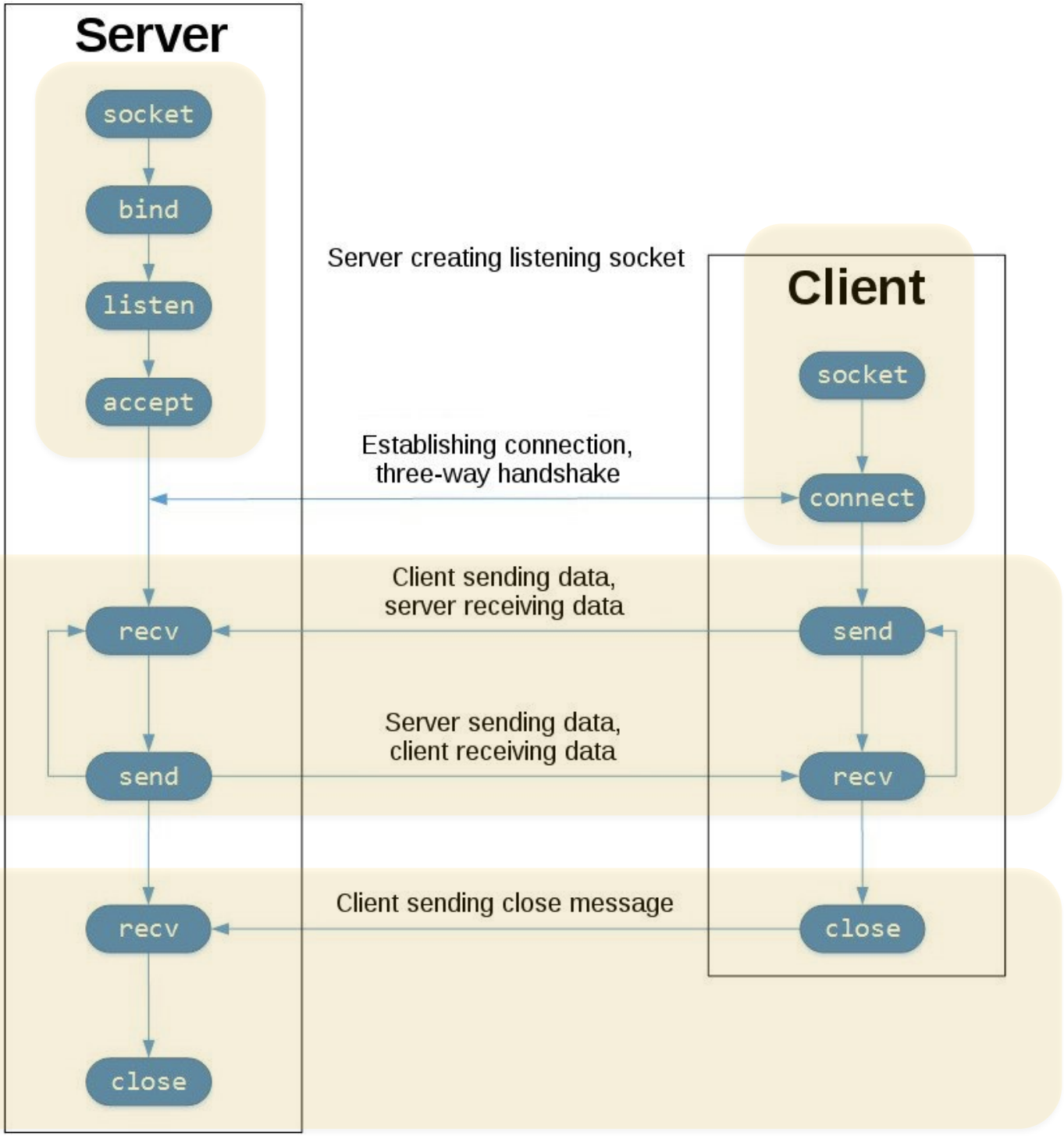
• `socket()`	create a socket
• `bind()`	bind to a socket address
• `listen()`	allow server to accept connections
• `accept()`	accept a connection (after listening)
• `send()`	send data to a socket
• `sendall()`	send all data to a socket
• `recv()`	receive data from a socket
• `close()`	close a socket
• `connect()`	connect to a server socket

This is a subset of methods available through the `socket` module (sufficient for all client/server communications in ENME441)

Create a TCP socket bound to the device IP address, listen for client connections, and accept the connection

Multiple rounds of data transfer after initial connection

Always close the connection to ensure both server and client know the current state



Basic Socket Methods

```
# Server side:
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # create a socket
# AF_INET --> IPv4 socket
# SOCK_STREAM --> use TCP as the message transport protocol

s.bind((HOST, PORT)) # Bind HOST IP address through the given PORT
# HOST = specific IP address, the loopback address (127.0.0.1),
#   or an empty string (meaning any connection will be allowed).
# PORT = privileged port (e.g. 80 for HTTP, or a custom port >1023.

s.listen(n)                  # Listen for up to n queued connections
conn, addr = s.accept()      # Accept client connection (blocking call)
                             # On connection, create a new socket
data = conn.recv(N)          # Receive up to N bytes of data
conn.send(data)              # Send a block of data
conn.sendall(data)           # Send all remaining data
conn.close()                 # Close the connection

# Client side:
c = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
c.connect((HOST, PORT))      # Connect a client socket to a server
c.sendall(data)              # Send all remaining data
recv_data = c.recv(bytes)    # Receive data (up to given bytes)
```

Echo Server

- Create an "echo server": a server that echoes back messages received from any clients:

Server side:

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((HOST, PORT))        # bind the port
server.listen(3)                 # listen for clients
conn, addr = server.accept()     # accept client connection
data = conn.recv(1024)           # receive data from client
conn.sendall(data)               # send data back to client
```

Client side:

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((HOST, PORT))     # connect to the server
client.sendall(send_data)        # send data (byte format) to server
recv_data = client.recv(1024)    # receive server response
```

echo_server.py

(reminder: `git pull` to update the repository on your Pi)

HTTP and HTML

- HTTP = HyperText Transfer Protocol
  - Client/server communication protocol designed for the Web
  - A client (e.g. Web browser) sends a specially-formatted HTTP request to a Web server
  - The server receives the request and generates an HTTP response
  - The client receives the response and acts on it (often by displaying a new web page)

- HTML = HyperText Markup Language
  - HTML is a language that can be interpreted by a Web client to display a web page (and much more).
  - See https://www.w3schools.com/html/ for an excellent HTML reference *(w3schools also has a very good Python tutorial).*

HTTP Request Format

https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages#http_requests

The client generates the HTTP request – we must understand the request format to interpret the client's message

- Start line:
  - HTTP method (e.g. `GET`, `POST`)
  - Target (path or URL, may include GET/POST information)
  - HTTP version (determines structure of the remainder of the request)

- Headers:
  - Multiple lines containing *header* : *value* pairs
  - Many possible messages (e.g. content length, text encoding, etc.)
  - Headers end with double-newline: `\r\n\r\n`

- Body:
  - Optional for `GET` requests
  - Required to send data for `POST` requests

HTTP Response Format

https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages#http_responses

The server generates an HTTP response (following a client request) with a defined format:

- Status line:
  - HTTP protocol version (e.g. `HTTP/1.1`)
  - Status code (e.g. ok = `200`, not found = `404`)
  - Status text (optional description of message, e.g. `Not Found`)

- Headers:
  - Multiple lines containing *header* : *value* pairs
  - Many possible messages (e.g. date, content type, keep-alive time, etc.)
  - End headers with double-newline: `\r\n\r\n`

- Body:
  - Optional (not needed if the status line gives all required information)
  - Often contains HTML code to display a new web page

HTTP Response Example: Sockets-based web server

To serve a web page, use a header to tell the client to interpret the message body as HTML, then send the HTML code in the body:

```
HTTP/1.0 200 OK
Content-type: text/html
<html>
<body>
This is a web page
</body>
</html>
```

Status line

Headers

Body (HTML code)

Python Web Server

```
import socket
addr = socket.getaddrinfo('', 80)[-1][-1][0]
s = socket.socket()
s.bind(addr)
s.listen(1)
conn, addr = s.accept()

# status line:
conn.send(b'HTTP/1.0 200 OK\r\n')
# headers:
conn.send(b'Content-type: text/html\r\n\r\n')
# body:
conn.sendall(b'<html><body>hello there</body></html>')

conn.close()
```

See example with GPIO state display in webserver.py

Python Web Server (Threaded)

- Since the server requires a blocking call, it must be executed in a separate thread or process to allow other code to run while the server is waiting for a client connection

See example: webserver_threaded.py

Passing Data to the Web Server

- The server in `webserver.py` responds to any client connection by sending a web page with GPIO pin status values.

- We can make the server response (and resulting web page) change dynamically based on user input through HTTP REST API requests.

- REST = REpresentational State Transfer, a “request-response” model for HTTP communications.

GET and POST Requests

- GET and POST are part of the REST API, and can be used to transfer information from a client to a server.

- GET vs. POST:
  - GET is used when server resources <u>will not</u> be modified due to the request
  - GET requests appear in the <u>message start line</u> (following "`?`" in the target URL)
  - POST is used when a server resource <u>will</u> be modified based on the request
  - POST requests appear in the <u>message body</u>

- Requests are structured as `key=value` pairs, with multiple pairs combined with "`&`", for example:

  `gpio1=on&gpio2=off&gpio3=off`

- GET and POST send data as plain text, so neither is secure
  - Use HTTPS / SSL (secure socket layer) to encrypt GET or POST data before sending

GET and POST Examples

GET (request data in start line)

GET /?val1=-2.2&val2=6&pin2=on HTTP/1.1	**Start line**
Host : 192.168.0.133 Accept: text/html,application/xhtml+xml,application/xml Accept-Encoding: gzip, deflate	**Headers**
	Body (empty)

POST (request data in body)

POST HTTP/1.1	**Start line**
Origin: 192.168.0.133 Content-Type: application/x-www-form-urlencoded Accept: text/html,application/xhtml+xml,application/xml Accept-Encoding: gzip, deflate	**Headers**
val1=-2.2&val2=6&pin2=on	**Body**

Example 1: Modify GPIO state with GET

- *NOTE: GET is not the best HTTP method to use here since a server resource (GPIO pin state) will be modified – POST should be used instead (see next example).*

- Similar to `webserver.py`, but now the client can request different information from the server (the HTML was immutable in the simple webserver case).

- The GET request could be generated by manually constructing the full URL + linking the button to the URL using the `<a>` tag, but using an HTML form is more robust.

- For details on HTML forms see: https://www.w3schools.com/html/html_forms.asp

web_gpio_GET.py

Example 2: Modify a GPIO state with POST

- POST is better than GET for this application since a server resource (GPIO state) will be modified by the request.
  - Both POST and GET will work, since we have full control over how our simple web server parses and responds to the request.
  - However, using POST makes it clear (in the code) that a server resource will be modified through the request.

- Generate the POST request using an HTML form.
  - We cannot embed the requested data in a URL since POST data resides in the *message body*

web_gpio_POST.py

Helper Function: parsePOSTdata()

- When there are multiple values in the message, we need a more efficient extraction method – here is a function that will return a dictionary of key:value pairs from the message:

```
# Helper function to extract key,value pairs of POST data

def parsePOSTdata(data):
    data_dict = {}
    idx = data.find('\r\n\r\n')+4
    data = data[idx:]
    data_pairs = data.split('&')
    for pair in data_pairs:
        key_val = pair.split('=')
        if len(key_val) == 2:
            data_dict[key_val[0]] = key_val[1]
    return data_dict
```

parsePOSTdata.py

HTML Forms

- Design your form using an HTML emulator, then copy & paste to your MicroPython code:

  https://repl.it
  or
  https://www.w3schools.com/html/tryit.asp?filename=tryhtml_form_submit

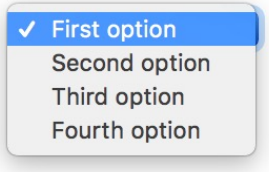
- The default form action is GET – add `action="POST"` if a POST request is intended.

- We will always target the root path – remove any `target="..."` section from the form definition line.

HTML Form Elements

- Button: `<button type="submit">`

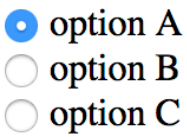
- Select menu: `<select>`



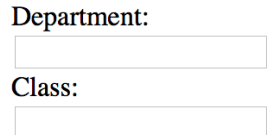
- Input submit `<input type="submit">`

Same as button but w/o content

- Radio buttons: `<input type="radio">`



- Text input: `<input type="text">`



- Range (slider): `<input type="range">`

For more elements & input types see:
https://www.w3schools.com/html/html_form_elements.asp
https://www.w3schools.com/html/html_form_input_types.asp

Hidden

- Use to pass fixed values that are not shown in the browser

HTML code:

```
<html>
<!-- save file as /var/www/html/hidden.html -->
<form action="/cgi-bin/hidden.py" method="POST">
    <input type="hidden" name="v1" value="enme441">
    <input type="hidden" name="v2" value="3.45">
    <input type="submit" value="Submit">
</form>
</html>
```

Submit element is needed to submit the form

Parsing the message:

```
data = parsePOSTdata(conn.recv[1024])
value1 = data['v1']
value1 = data['v2']
```

Button

Button

- Shared button name and different value fields to distinguish which button is selected:

HTML code:

```
<html>
<form action="/" method="POST">
   <button name="two_buttons" value="b1"> Click here </button>
   <button name="two_buttons" value="b2"> Or here </button>
</form>
</html>
```

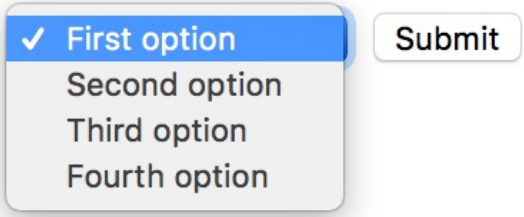
Same name

different values

Parsing the message:

```
data = parsePOSTdata(conn.recv[1024])
value = data['two_buttons']
```

Select



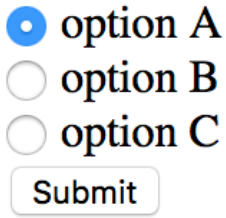
HTML code:

```
<html>
<form action="/cgi-bin/select.py" method="POST">
    <select name="menu_choice">
       <option value="option1"> First option</option>
       <option value="option2"> Second option</option>
       <option value="option3"> Third option</option>
       <option value="option4"> Fourth option</option>
     </select>
     <input type="submit" value="Submit">
</form>
</html>
```

Parsing the message:

```
data = parsePOSTdata(conn.recv[1024])
selection = data['menu_choice']
```

Radio



HTML code:

```
<html>
<form action="/cgi-bin/radio.py" method="POST">
    <input type="radio" name="option" value="a" checked> option A <br>
    <input type="radio" name="option" value="b"> option B <br>
    <input type="radio" name="option" value="c"> option C <br>
    <input type="submit" value="Submit">
</form>
</html>
```

Parsing the message:

```
data = parsePOSTdata(conn.recv[1024])
selection = data['option']
```

Text

Department:

Class:

Submit

HTML code:

```
<html>
<form action="/cgi-bin/text.py" method="POST">
    Department:<br>
    <input type="text" name="department"><br>
    Class:<br>
    <input type="text" name="class"><br>
    <input type="submit" value="Submit">
</form>
</html>
```

Parsing the message:

```
data = parsePOSTdata(conn.recv[1024])
dept = data['department']
class = data['class']
```

Range

HTML code:

```
<html>
<form action="/cgi-bin/range.py" method="POST">
    <input type="range" name="slider1" min ="0" max="1000"
           value ="500"/><br>
    <input type="range" name="slider2" min ="100" max="500"
           value ="300"/>
    <input type="submit" value="Submit">
</form>
</html>
```

Parsing the message:

```
data = parsePOSTdata(conn.recv[1024])
s1 = data['slider1']
s2 = data['slider2']
```

Webhooks

- Webhooks use HTTP POST requests to allow events on one site (the ESP32) to invoke behavior on another site (a web application or server with a defined webhook interface)
- Allowable POST data fields are defined by the server
- Request body content (`field:value` pairs) often uses JSON format
- Many servers provide both webhooks and custom APIs for data transfer:
  - Webhooks are one-way: a client (e.g. ESP32) can send information to a server (e.g. Discord, Google apps, etc).
  - Some APIs can provide two-way communication, but require custom code running on the client.

See discord_webhooks.py example