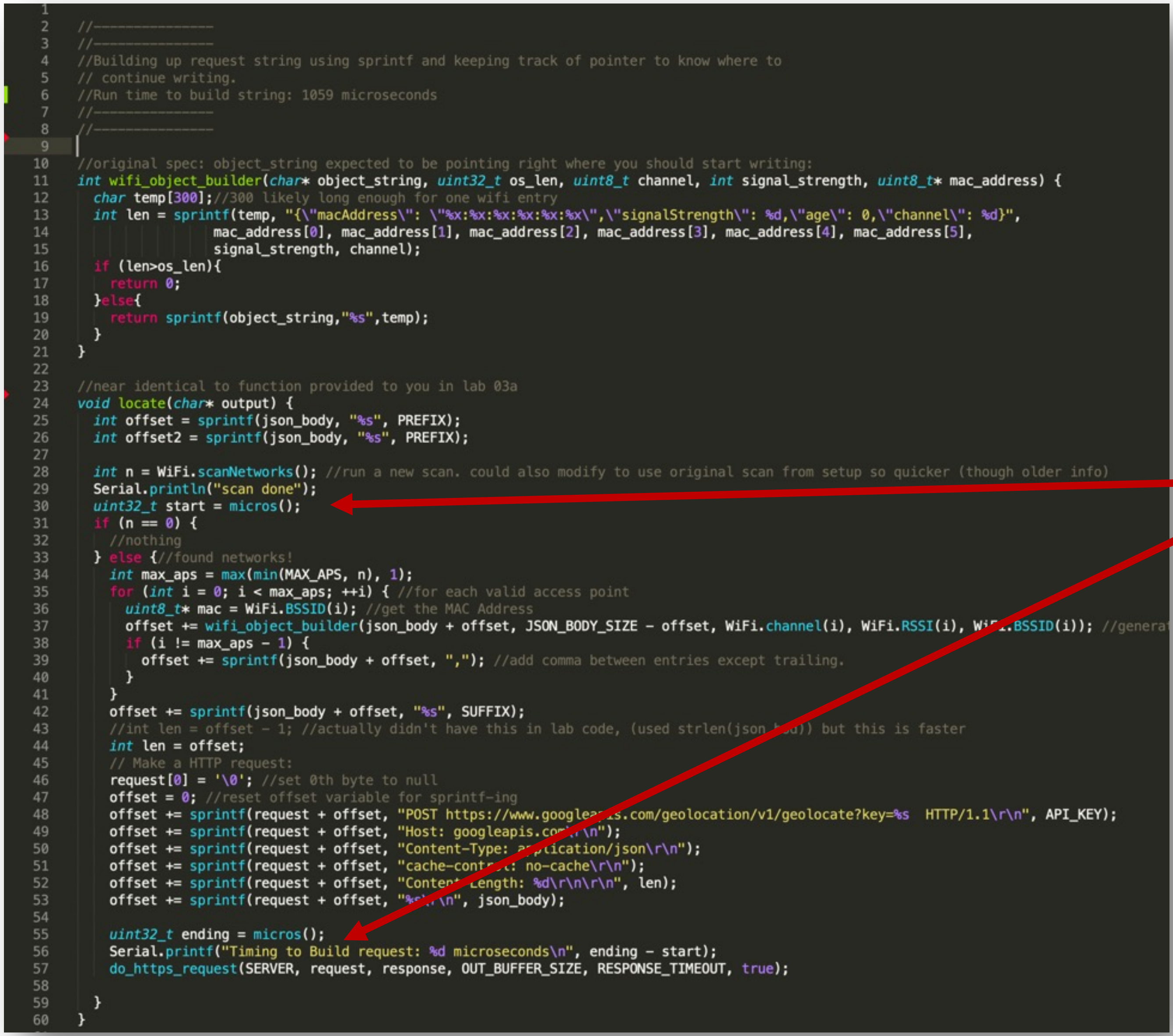
6.08

Lecture 05

sprintf vs. strcat in Lab04B?

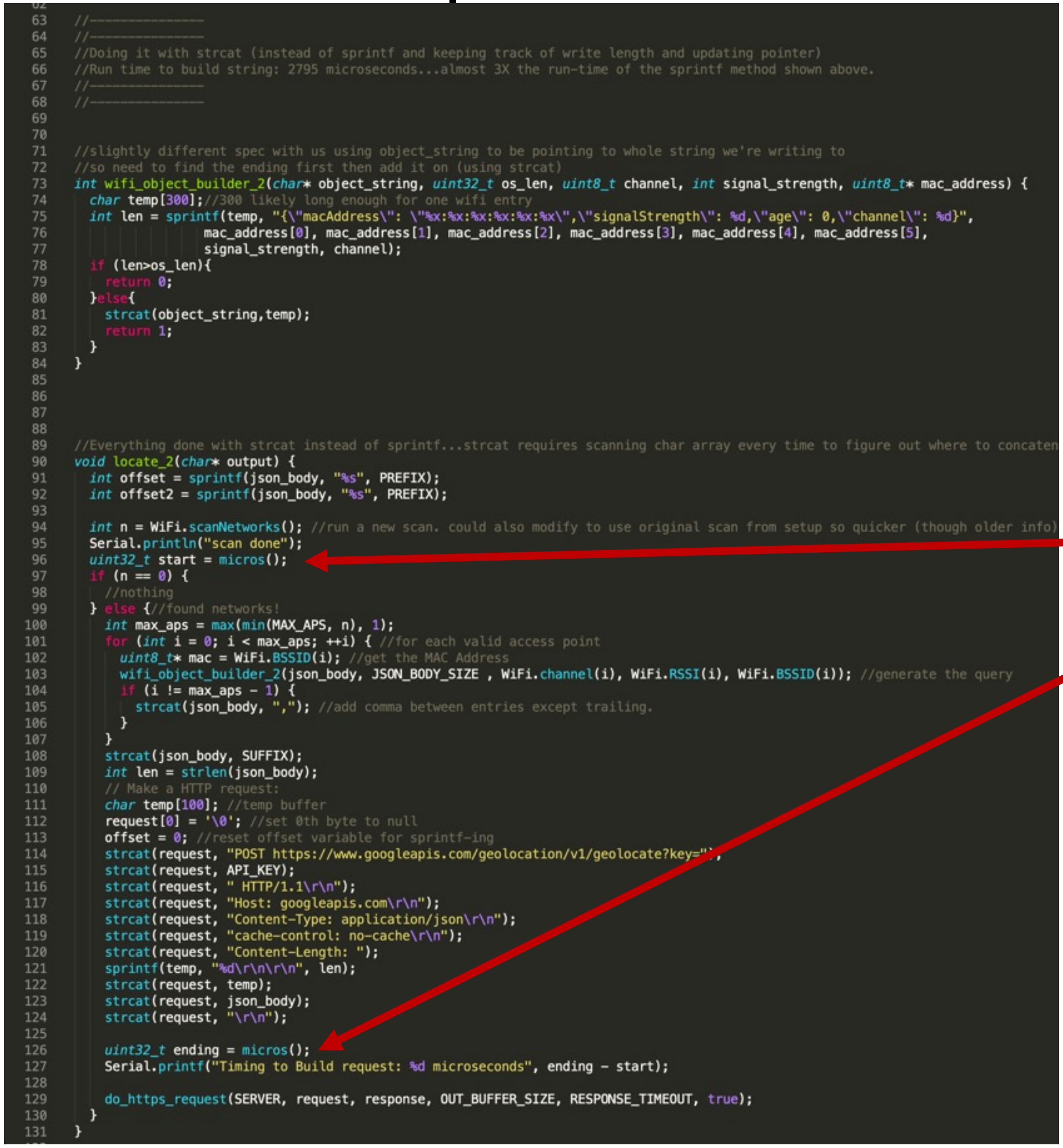
Sprintf solution from Lab04B



```
//---------------
//---------------
//Building up request string using sprintf and keeping track of pointer to know where to
// continue writing.
//Run time to build string: 1059 microseconds
//---------------
//---------------

//original spec: object_string expected to be pointing right where you should start writing:
int wifi_object_builder(char* object_string, uint32_t os_len, uint8_t channel, int signal_strength, uint8_t* mac_address) {
  char temp[300];//300 likely long enough for one wifi entry
  int len = sprintf(temp, "{\"macAddress\": \"%x:%x:%x:%x:%x:%x\",\"signalStrength\": %d,\"age\": 0,\"channel\": %d}",
                    mac_address[0], mac_address[1], mac_address[2], mac_address[3], mac_address[4], mac_address[5],
                    signal_strength, channel);
  if (len>os_len){
    return 0;
  }else{
    return sprintf(object_string,"%s",temp);
  }
}

//near identical to function provided to you in lab 03a
void locate(char* output) {
  int offset = sprintf(json_body, "%s", PREFIX);
  int offset2 = sprintf(json_body, "%s", PREFIX);

  int n = WiFi.scanNetworks(); //run a new scan. could also modify to use original scan from setup so quicker (though older info)
  Serial.println("scan done");
  uint32_t start = micros();
  if (n == 0) {
    //nothing
  } else {//found networks!
    int max_aps = max(min(MAX_APS, n), 1);
    for (int i = 0; i < max_aps; ++i) { //for each valid access point
      uint8_t* mac = WiFi.BSSID(i); //get the MAC Address
      offset += wifi_object_builder(json_body + offset, JSON_BODY_SIZE - offset, WiFi.channel(i), WiFi.RSSI(i), WiFi.BSSID(i)); //generat
      if (i != max_aps - 1) {
        offset += sprintf(json_body + offset, ","); //add comma between entries except trailing.
      }
    }
    offset += sprintf(json_body + offset, "%s", SUFFIX);
    //int len = offset - 1; //actually didn't have this in lab code, (used strlen(json_bod)) but this is faster
    int len = offset;
    // Make a HTTP request:
    request[0] = '\0'; //set 0th byte to null
    offset = 0; //reset offset variable for sprintf-ing
    offset += sprintf(request + offset, "POST https://www.googleapis.com/geolocation/v1/geolocate?key=%s  HTTP/1.1\r\n", API_KEY);
    offset += sprintf(request + offset, "Host: googleapis.com\r\n");
    offset += sprintf(request + offset, "Content-Type: application/json\r\n");
    offset += sprintf(request + offset, "cache-control: no-cache\r\n");
    offset += sprintf(request + offset, "Content-Length: %d\r\n\r\n", len);
    offset += sprintf(request + offset, "%s\r\n", json_body);

    uint32_t ending = micros();
    Serial.printf("Timing to Build request: %d microseconds\n", ending - start);
    do_https_request(SERVER, request, response, OUT_BUFFER_SIZE, RESPONSE_TIMEOUT, true);

  }
}
```

Timing to build request:
1059 microseconds

Strcat implementation (mod Lab04B)



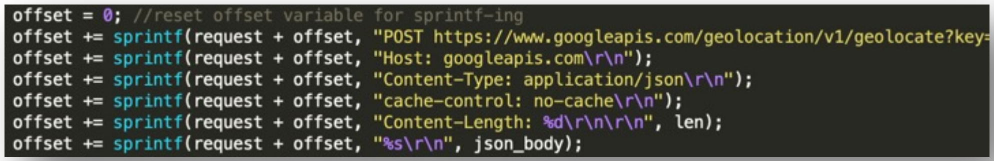
```c
//---------------
//---------------
//Doing it with strcat (instead of sprintf and keeping track of write length and updating pointer)
//Run time to build string: 2795 microseconds...almost 3X the run-time of the sprintf method shown above.
//---------------
//---------------


//slightly different spec with us using object_string to be pointing to whole string we're writing to
//so need to find the ending first then add it on (using strcat)
int wifi_object_builder_2(char* object_string, uint32_t os_len, uint8_t channel, int signal_strength, uint8_t* mac_address) {
  char temp[300];//300 likely long enough for one wifi entry
  int len = sprintf(temp, "{\"macAddress\": \"%x:%x:%x:%x:%x:%x\",\"signalStrength\": %d,\"age\": 0,\"channel\": %d}",
                    mac_address[0], mac_address[1], mac_address[2], mac_address[3], mac_address[4], mac_address[5],
                    signal_strength, channel);
  if (len>os_len){
    return 0;
  }else{
    strcat(object_string,temp);
    return 1;
  }
}




//Everything done with strcat instead of sprintf...strcat requires scanning char array every time to figure out where to concaten
void locate_2(char* output) {
  int offset = sprintf(json_body, "%s", PREFIX);
  int offset2 = sprintf(json_body, "%s", PREFIX);

  int n = WiFi.scanNetworks(); //run a new scan. could also modify to use original scan from setup so quicker (though older info)
  Serial.println("scan done");
  uint32_t start = micros();
  if (n == 0) {
    //nothing
  } else {//found networks!
    int max_aps = max(min(MAX_APS, n), 1);
    for (int i = 0; i < max_aps; ++i) { //for each valid access point
      uint8_t* mac = WiFi.BSSID(i); //get the MAC Address
      wifi_object_builder_2(json_body, JSON_BODY_SIZE , WiFi.channel(i), WiFi.RSSI(i), WiFi.BSSID(i)); //generate the query
      if (i != max_aps - 1) {
        strcat(json_body, ","); //add comma between entries except trailing.
      }
    }
    strcat(json_body, SUFFIX);
    int len = strlen(json_body);
    // Make a HTTP request:
    char temp[100]; //temp buffer
    request[0] = '\0'; //set 0th byte to null
    offset = 0; //reset offset variable for sprintf-ing
    strcat(request, "POST https://www.googleapis.com/geolocation/v1/geolocate?key=");
    strcat(request, API_KEY);
    strcat(request, " HTTP/1.1\r\n");
    strcat(request, "Host: googleapis.com\r\n");
    strcat(request, "Content-Type: application/json\r\n");
    strcat(request, "cache-control: no-cache\r\n");
    strcat(request, "Content-Length: ");
    sprintf(temp, "%d\r\n\r\n", len);
    strcat(request, temp);
    strcat(request, json_body);
    strcat(request, "\r\n");

    uint32_t ending = micros();
    Serial.printf("Timing to Build request: %d microseconds", ending - start);

    do_https_request(SERVER, request, response, OUT_BUFFER_SIZE, RESPONSE_TIMEOUT, true);
  }
}
```

Timing to build request:
2795 microseconds

Almost three-fold slower :/

WHY????

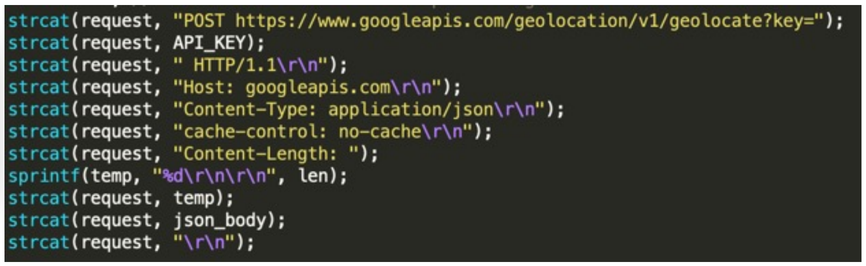
sprintf: 1059 microseconds



```
offset = 0; //reset offset variable for sprintf-ing
offset += sprintf(request + offset, "POST https://www.googleapis.com/geolocation/v1/geolocate?key=
offset += sprintf(request + offset, "Host: googleapis.com\r\n");
offset += sprintf(request + offset, "Content-Type: application/json\r\n");
offset += sprintf(request + offset, "cache-control: no-cache\r\n");
offset += sprintf(request + offset, "Content-Length: %d\r\n\r\n", len);
offset += sprintf(request + offset, "%s\r\n", json_body);
```

- sprintf starts writing where you tell it to.
- Sprintf also returns the number of characters written
- Keeping track of this let's us just keep appending by modifying the pointer with the total written.

strcat: 2795 microsecond



```
strcat(request, "POST https://www.googleapis.com/geolocation/v1/geolocate?key=");
strcat(request, API_KEY);
strcat(request, " HTTP/1.1\r\n");
strcat(request, "Host: googleapis.com\r\n");
strcat(request, "Content-Type: application/json\r\n");
strcat(request, "cache-control: no-cache\r\n");
strcat(request, "Content-Length: ");
sprintf(temp, "%d\r\n\r\n", len);
strcat(request, temp);
strcat(request, json_body);
strcat(request, "\r\n");
```

- strcat works by automatically appending to the string you point to
- But how do you actually find the end?
- Strcat has to scan the string starting at the beginning and find the first NULL. Then it writes after that
- As string gets longer and longer, each call takes longer to run since the scan has to run for longer
- Same issue with strlen()…this function runs faster on short strings than longer strings

Slow, Inefficient Code Wastes resources

- In many of our ESP32 applications (or server applications) even poorly written code will ”work”,
- But at every level doing things more efficiently (with software and hardware) can be meaningful
- 1 ms vs 3 ms seems trivial, but what if you needed to process 1 million requests? This means 1000 seconds vs 3000 seconds.
- That means:
    - 2000 fewer seconds for doing other tasks (time is $)
    - That means running your processor for 2000 more seconds (costs power and $, and of course C.R.E.A.M.)

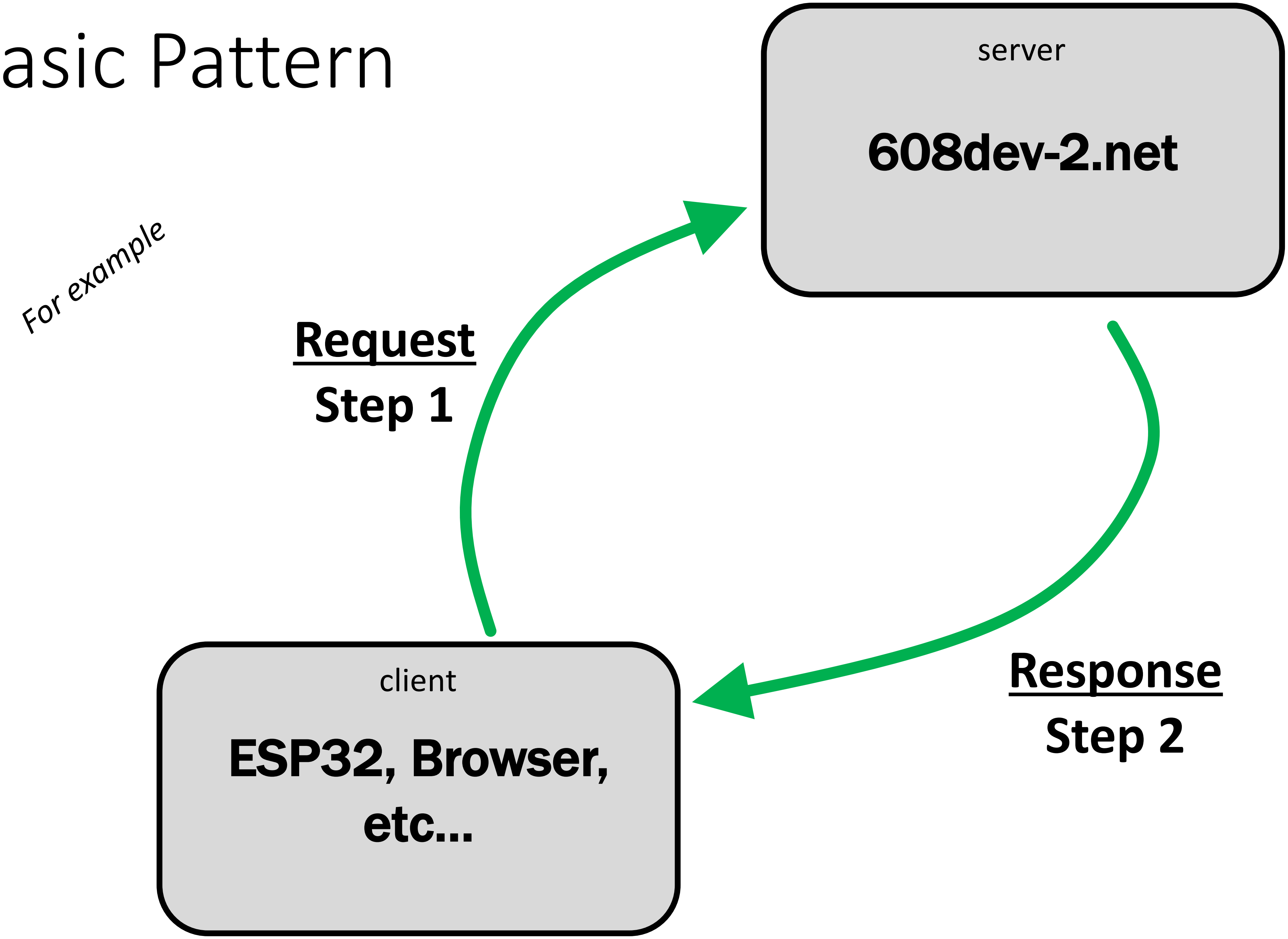
HTTP Requests

Sort of a Review

HTTP

- HyperText Transfer Protocol (HTTP)
- Request-Response Protocol
- Client and Server:
  - Client makes a request
  - Server provides a response
- Both sides of this exchange have very strict requirements on their formatting!

Basic Pattern



Basic Pattern

- The client sends a request to the server
- The server parses it, carries out the specified actions (as dictated by internal code), and then returns a response
- There are two major verbs for requests that we use (though there are many others):
    - GET
    - POST

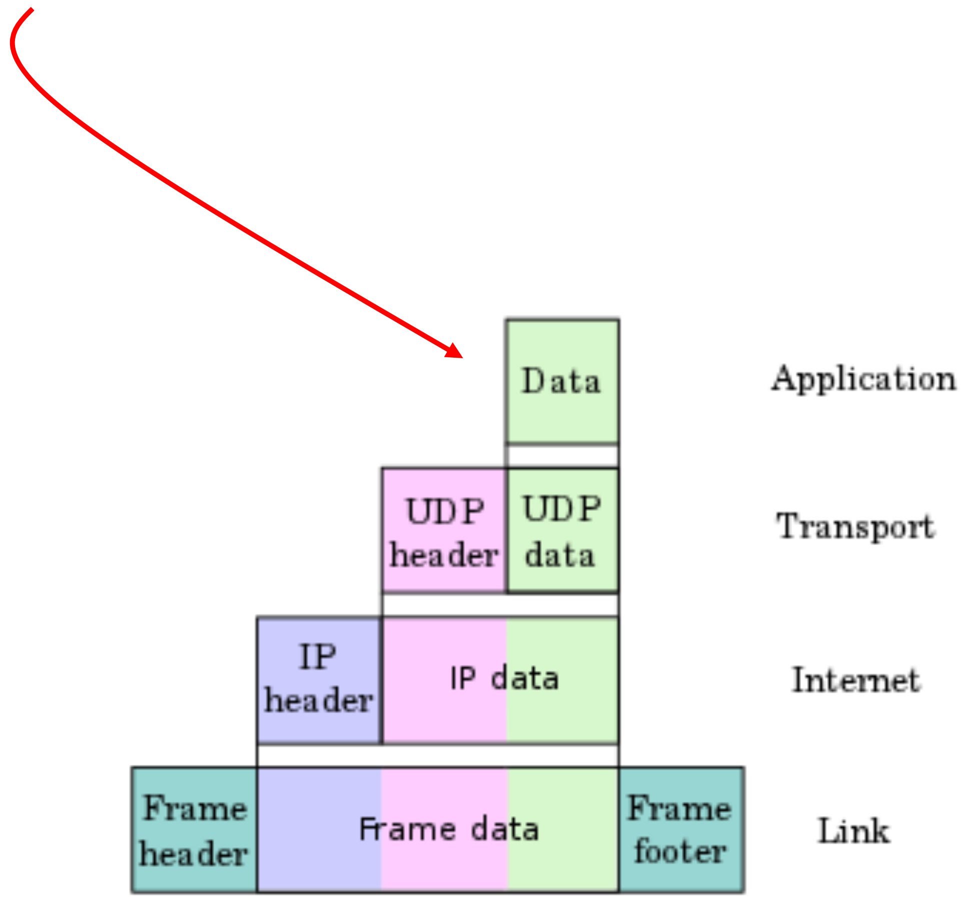
The client sends the request

- Why does the client have to initiate contact?
- Sub-question: How is the client able to direct its message to a server that may be sitting all the way in New Jersey (608dev.net), Toronto (608dev-2.net), etc…?

HTTP is one small piece

- HTTP is built on top of a number of layers in the TCP/IP
- Transmission Control Protocol (TCP)
- Internet Protocol (IP)

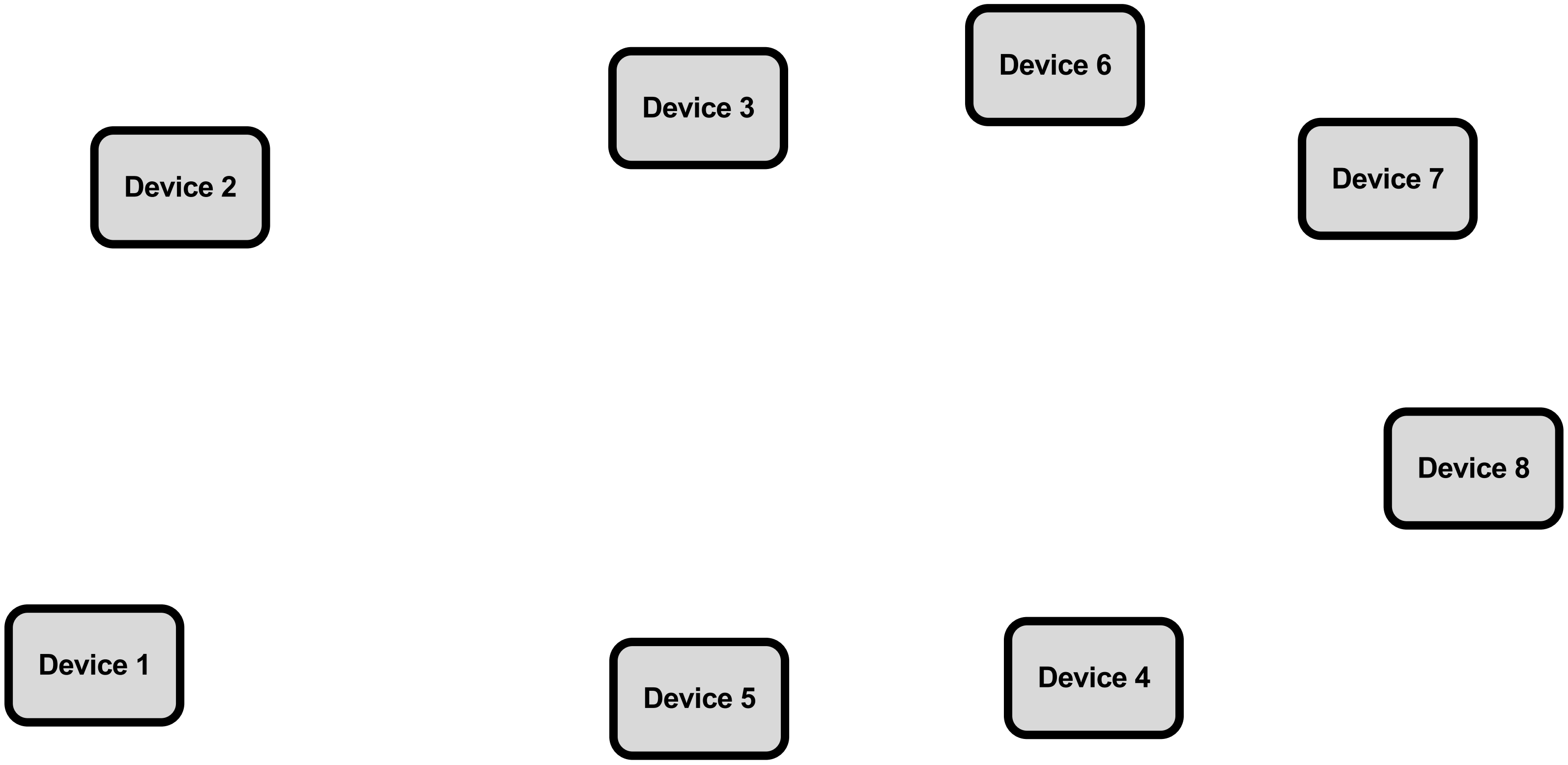
HTTP is an example of an application layer



https://en.wikipedia.org/wiki/Internet_protocol_suite

How to Allow Many Things to Communicate with Many Things?

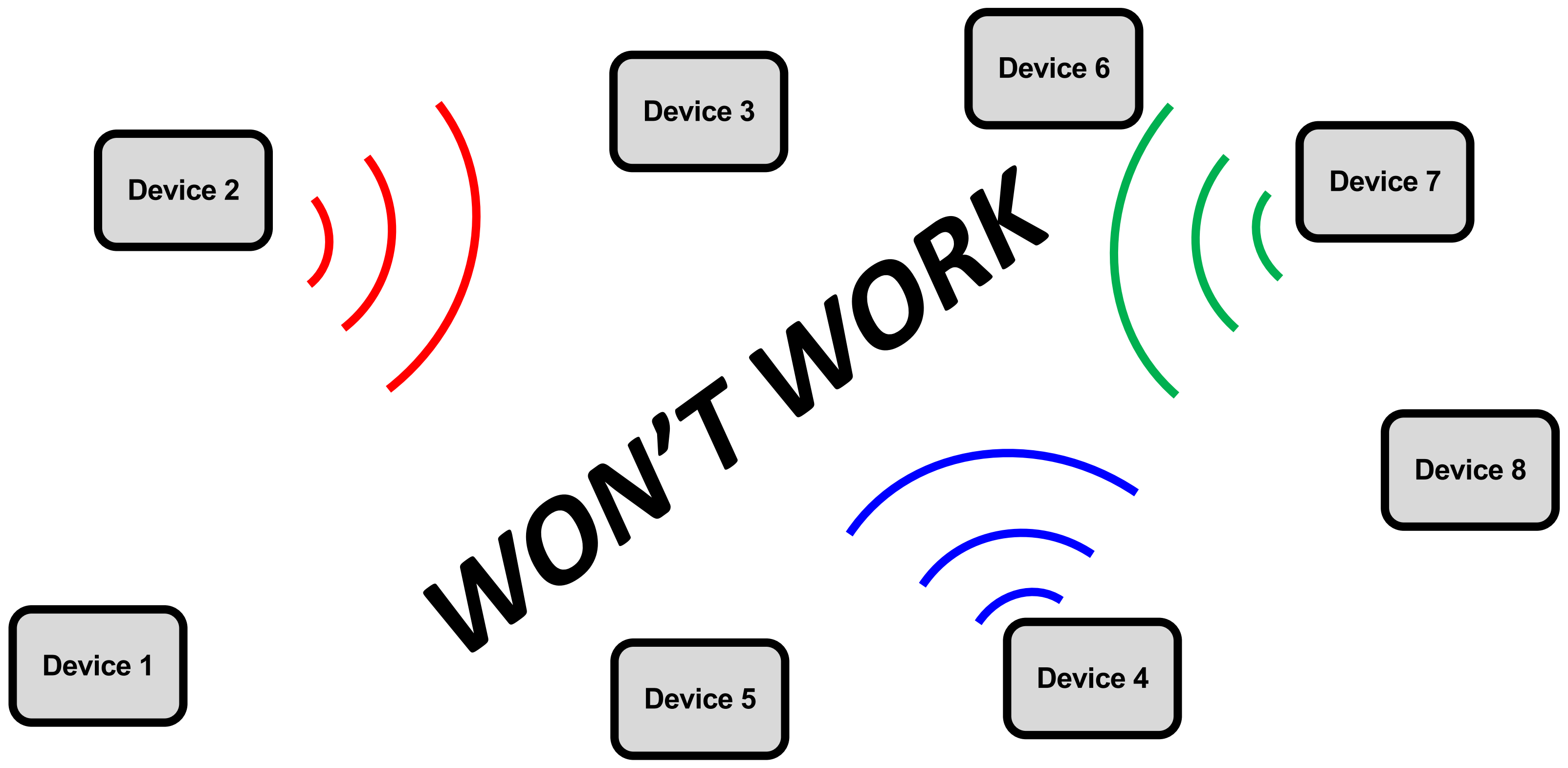
- ????



How to Allow Many Things to Communicate with Many Things?

- Solution 0: Broadcast???
  - *Have Every Device Listen to Every Device*

Want Device 6 to Talk with Device 2? Shout!



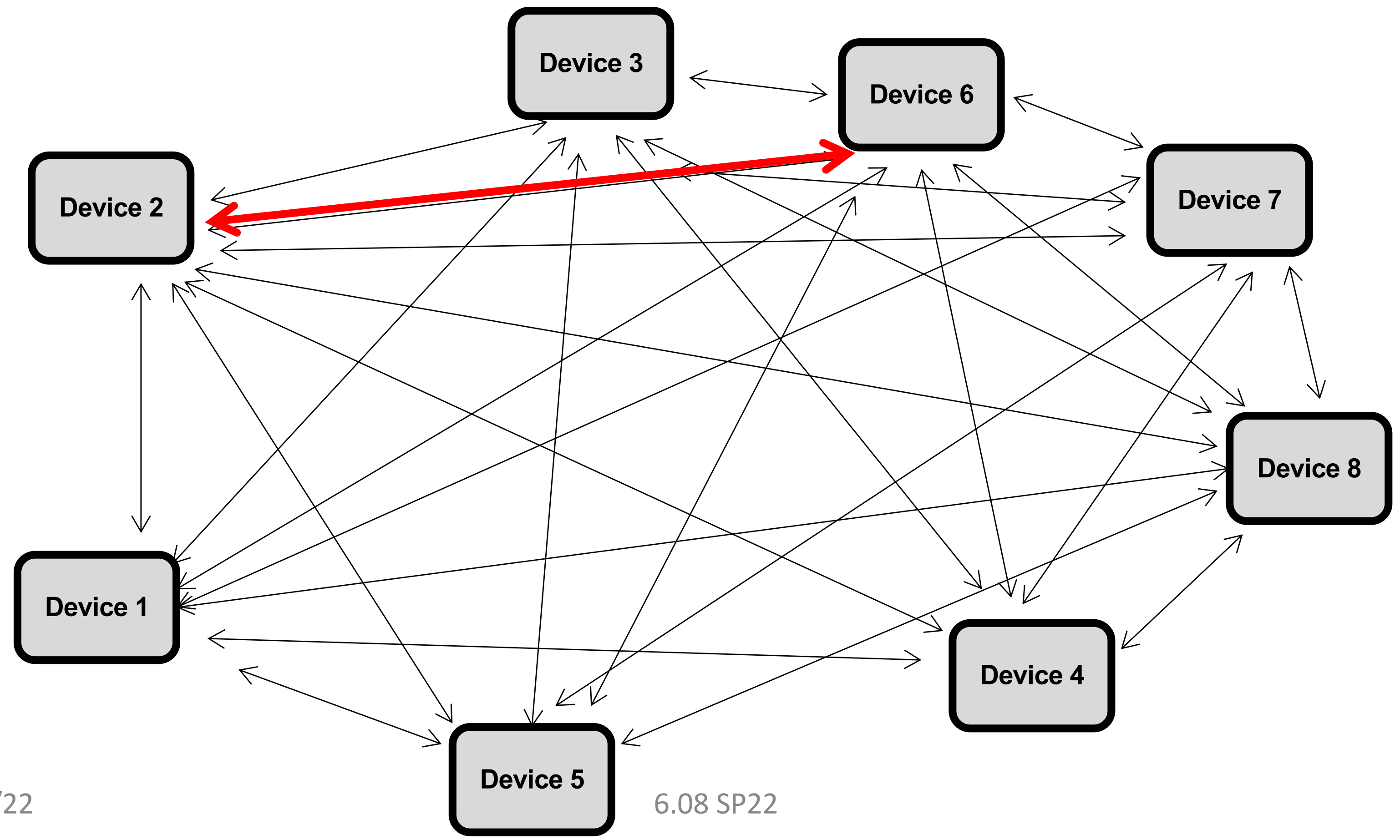
Need Dedicated Communication Channels

- A way to send information from one device to another device ONLY!
- This could be dedicated wires to send the 1's and 0's of our digital existence
- Could be particular radio frequency that sends data
- Could be sound at a particular frequency
- Many options, but need dedicated channels!

How to Allow Many Things to Communicate with Many Things?

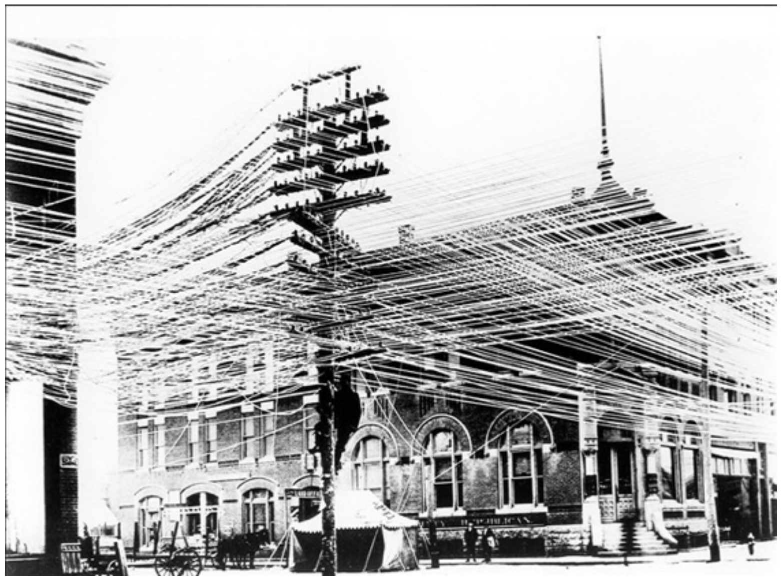
- Solution 1: Direct P2P
  - *Have every device wired to every other device*

Want Device 6 to Talk with Device 2? Activate their Dedicated Wires!



Absurd But Was Done

- Early Telegraph/ Telephone, this is how it was done originally
- Obviously didn't scale so usually one device couldn't necessarily connect to every other device.
- But point-to-point wiring was very common

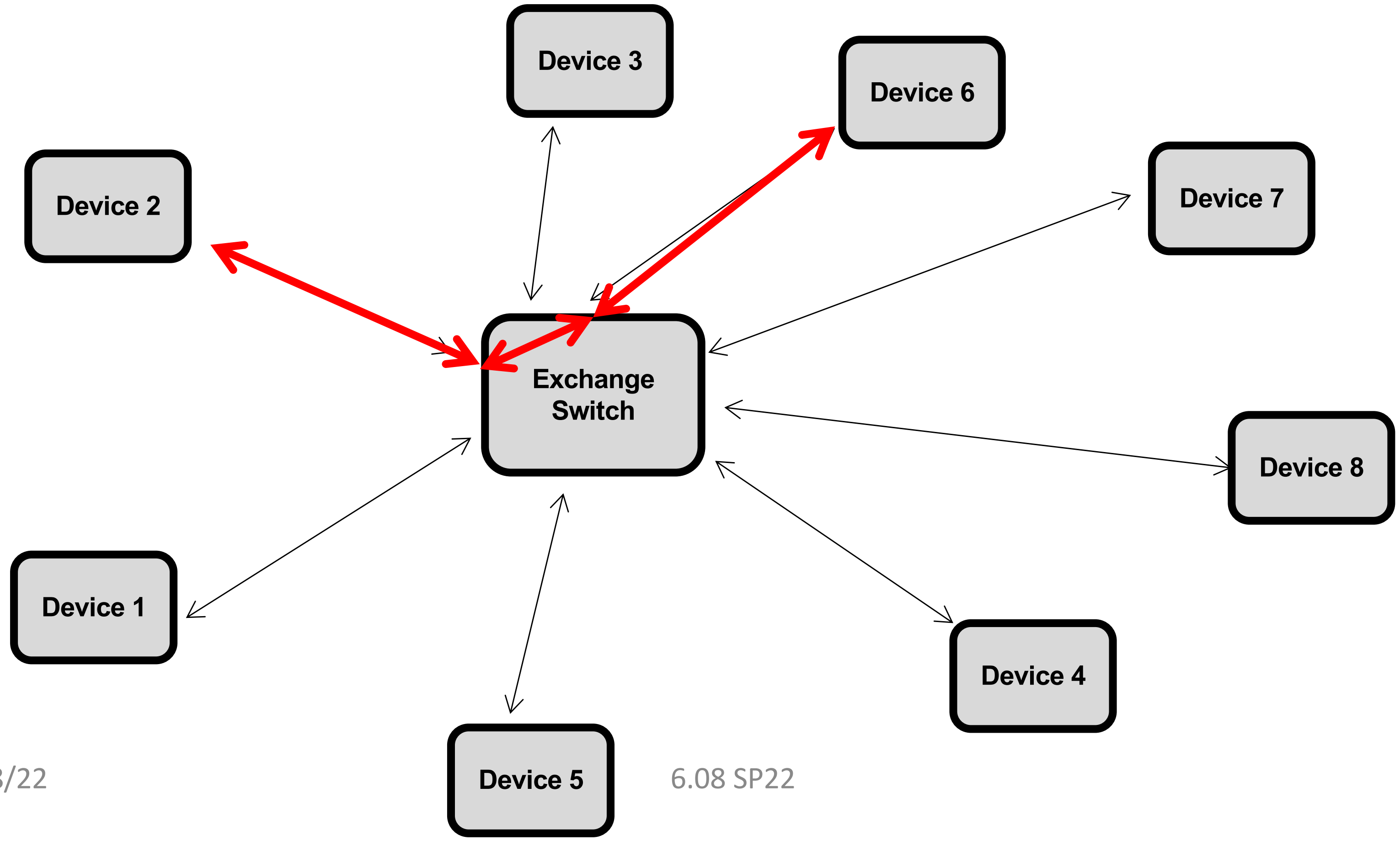


Telegraph wires, Kansas, 1880s

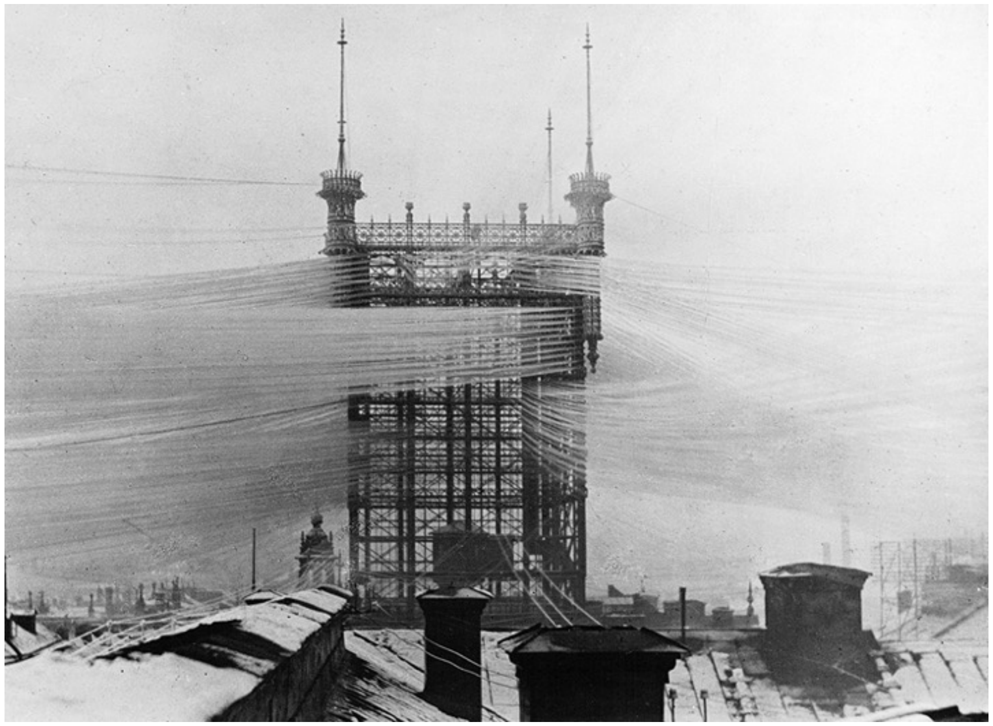
How to Allow Many Things to Communicate with Many Things?

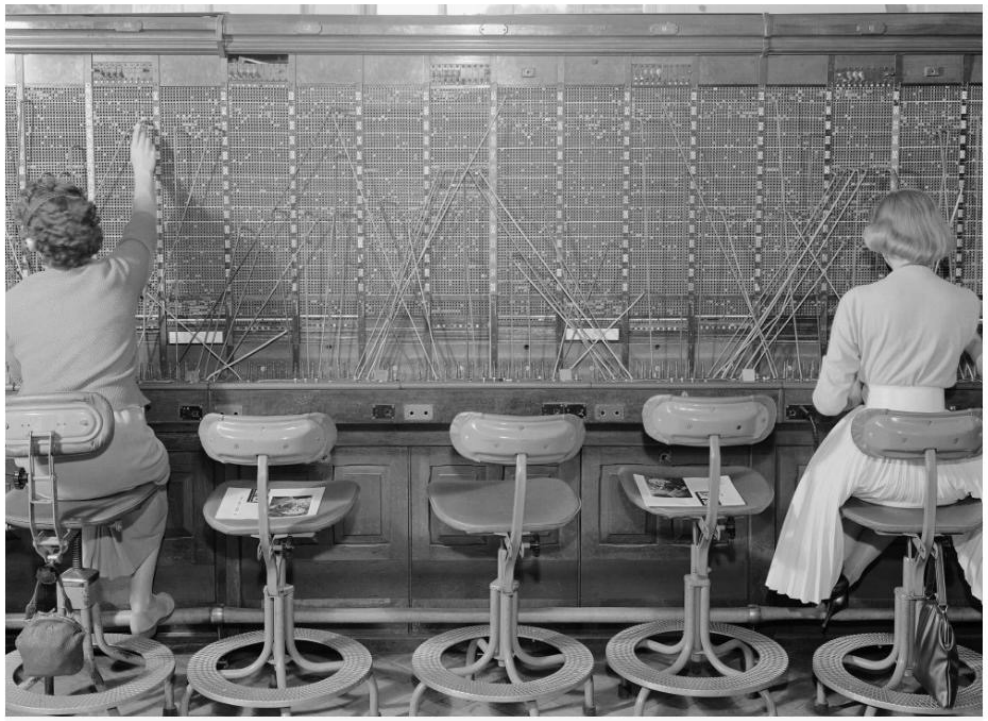
- Solution 2: Exchange
  - *Have Every Device wired to a central exchange*

Want Device 6 to Talk with Device 2? Connect them!



How Would Work?


Central Telephone Tower in Stockholm Sweden ~1900


Switchboard operators at Enfield telephone exchange, 1960.
Science Museum Group collection

Switching was usually done by humans at first

Later on was done by machines/automated telephone switching equip

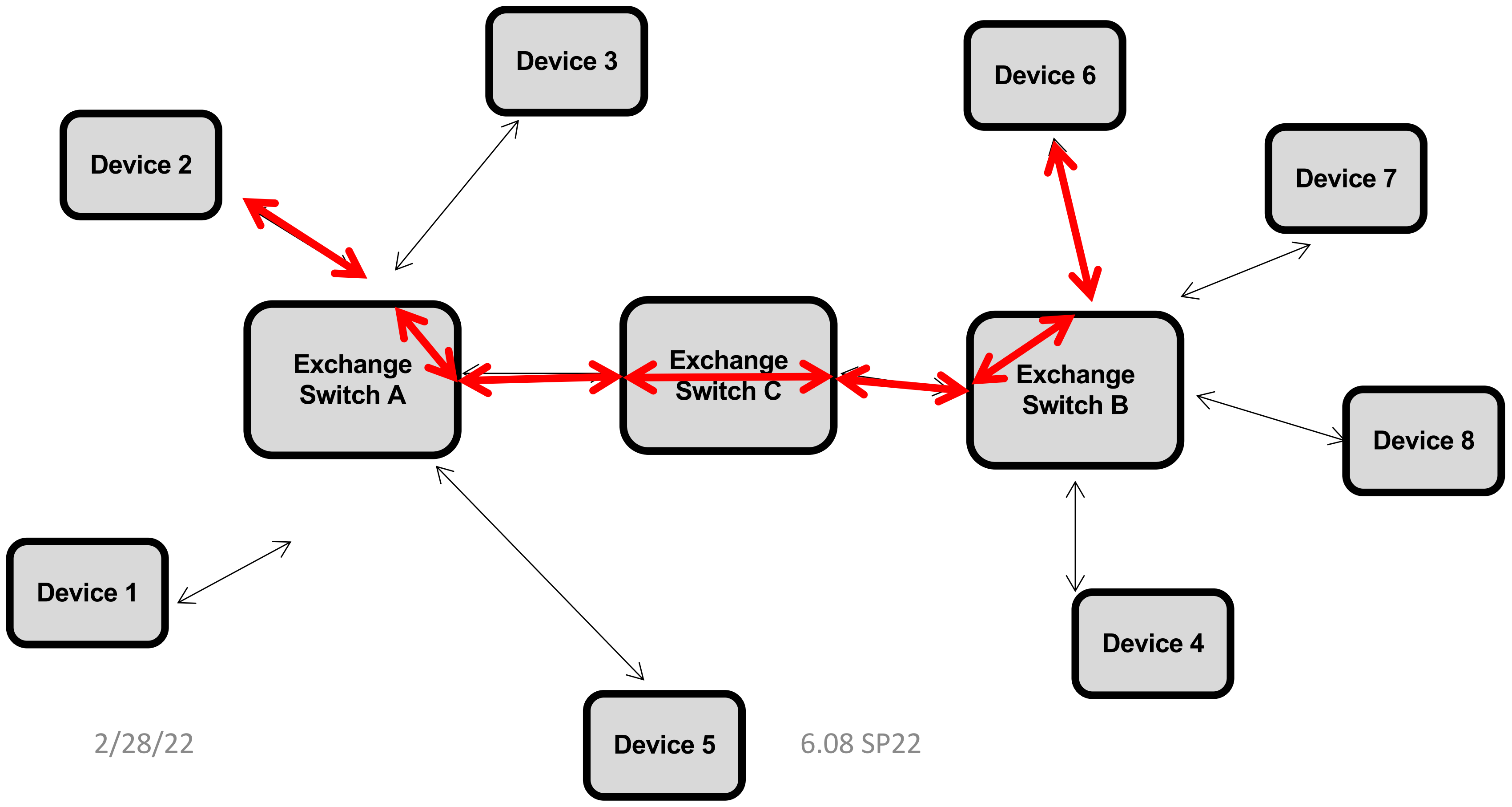
- Each device would need an address of some form (owner of house, phone number, etc…) for the exchange to know how to do the interconnect

How to Allow Many Things to Communicate with Many Things?

- Solution 2b: Nested Exchange
  - Have regional exchanges
  - *Layer these*

Very simple cartoon. Modern systems have many more layers

Want Device 6 to Talk with Device 2? Connect them!



Large Nested Exchange System

- Scales Much More easily:
  - Every device does not need dedicated communication channel with central exchange (saves copper or global radio bandwidth)
- Addressing scheme will get more complicated. Need to build up an addressing scheme that works with layers of local exchanges (routers/network switches)
- This is the TCP/IP stack!

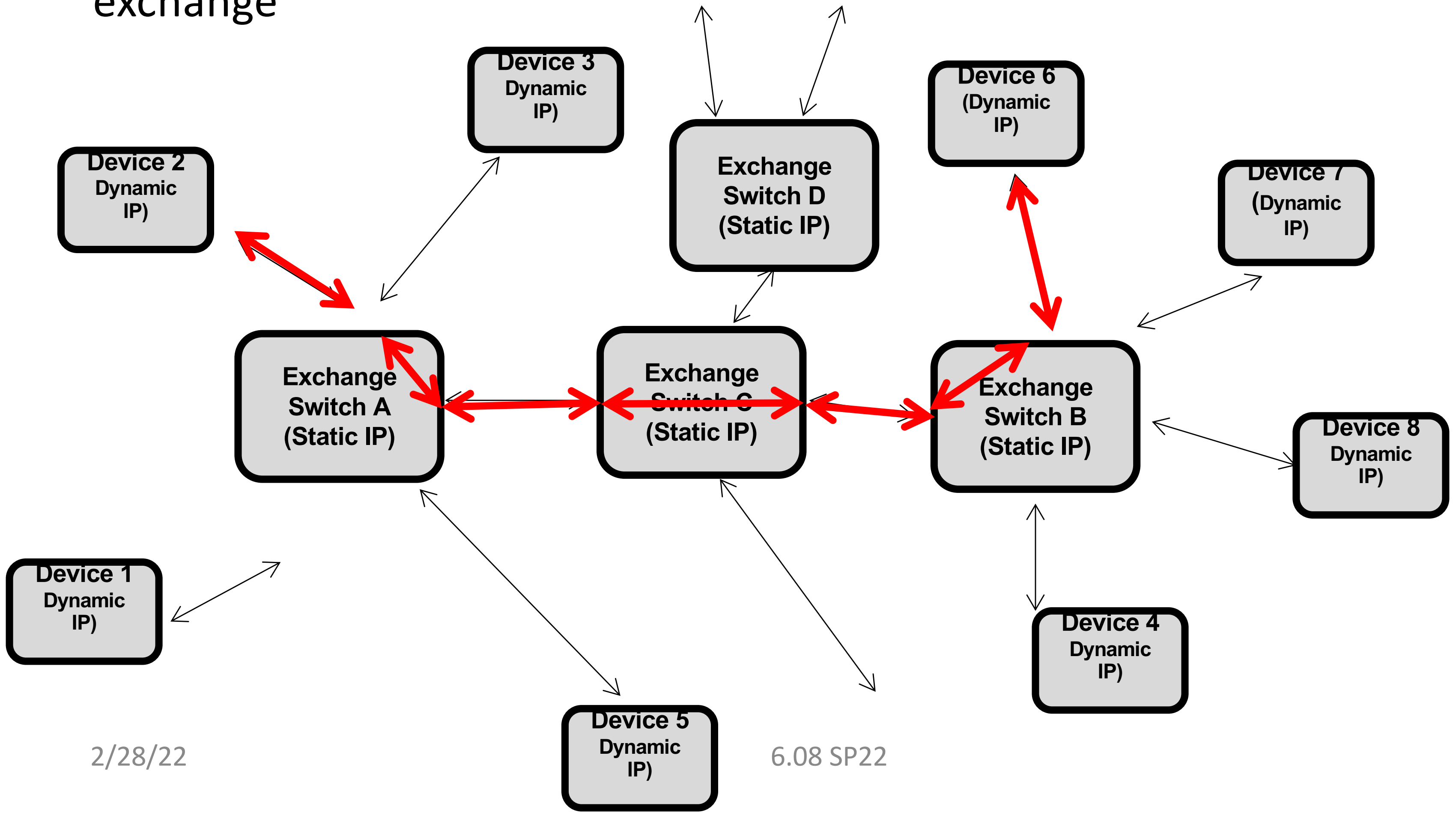
IP (Internet Protocol) Address

- The internet as a system is set up to direct messages based on an addressing system (each device has number)
- Every device should be uniquely identifiable but it doesn’t mean each device needs to have a globally unique address nor does it mean that every device has a constant (forever address*)
- Clients which may connect from many locations usually have a **dynamic IP address** assigned by the local network they’re interfacing through
- Many servers/heavily-used resources will have **static IP addresses** which are pretty fixed in time.

*that should be the job of the MAC address, but even that can change

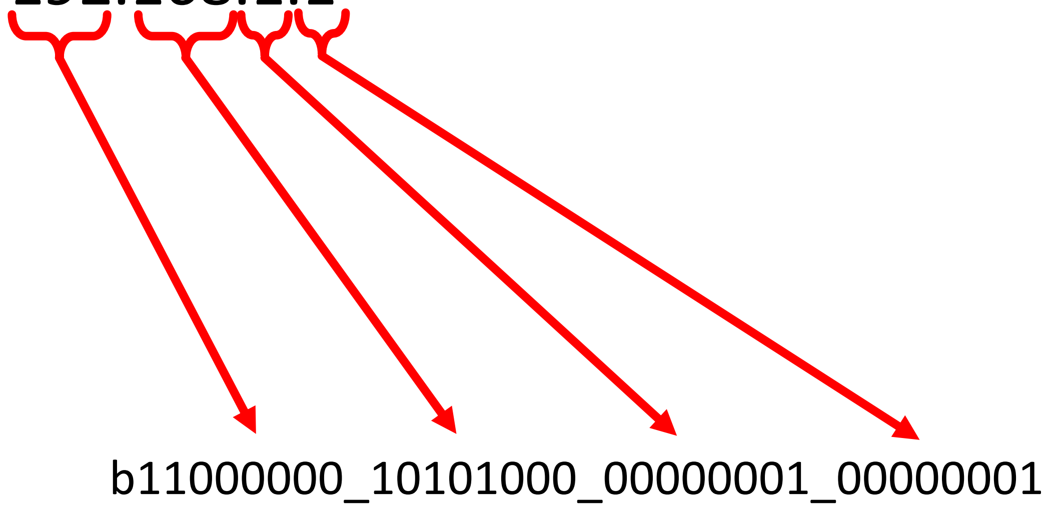
Dynamic and Static IP Addresses

- Some things will have **globally unique static IP addresses**...it is the job of regional exchanges to assign and keep track of local dynamic IP addresses
- Device 3 and Device 7 could have same IP address actually, but that's ok because they live underneath two exchanges with globally unique IP addresses and can still be known due to information tracked in each exchange



IP (Internet Protocol) Address

- Unlike MAC addresses, IP addresses are usually expressed in base 10 numbers.
- The original IP addresses were four bytes, and we write them as each byte (in base 10) separated by periods:
    - 192.168.1.1



b11000000_10101000_00000001_00000001

Having a Static IP is Good if You’re an Online Resource

- If your address is fixed/constant/known it is easy for many devices to connect to you.
- 608dev.net has address: 50.116.55.137
- iesc.io has address: 172.104.28.81
- 608dev-2.net has address: 172.105.8.169
- Static IP addresses are **not** a moving target and random devices always know where you’re at.
- This is why clients start the conversation with a request…they know where to send it. The server may not know where the client is “in” the internet

IPv4 The Original

- Around when the internet was created, they decided that we as a world would need one 32 bit number to describe/uniquely label all resources on the internet
- 2**32 ~ 4.29 billion (from previous lectures)
- Is this enough?
- No…called “IPv4 Exhaustion”…happened on January 31, 2011….though there’s still unused IPv4 addresses
- they’re not all actively used, but all are “owned”…going rate of about 10 to 20 bucks a piece

MIT and IP Addresses

- MIT was a major contributor to the early internet
- Back when they were worth nothing, MIT claimed a 24 bit block (about 1/256th) of the internet as their own (4.29 billion/256 is about 16.5 million addresses)
- Times have changed. Those worthless bits are valuable now
- MIT "returned" 8 million of them in 2017/18/19 for an undisclosed sum (to Amazon)
- The Institute still has about 6 million of its remaining 8 million IPv4 addresses unused/unallocated

Solution to IP Exhaustion is IPv6

- Internet is migrating to newer standard…IPv6.
- Now use 128 bit address rather than 32 bit:
  - 3.4×10^{38}addresses
- Should be good for a while
- There are reverse compatibility issues and various other things in place, so IPv4 addresses are still valulable
- MIT owns about 2.0×10^{31} of these new IPv6 addresses by the way

The internet is far more complicated than this

- 6.033 goes into this a lot:
  - How do dynamic IP addresses get assigned
  - How does “608dev.net” get mapped to 50.116.55.137
  - How does a packet from a dynamic IP get sent to a static IP and then back again while lots of other ones are going around as well.
  - Maybe IPv4 vs. IPv6

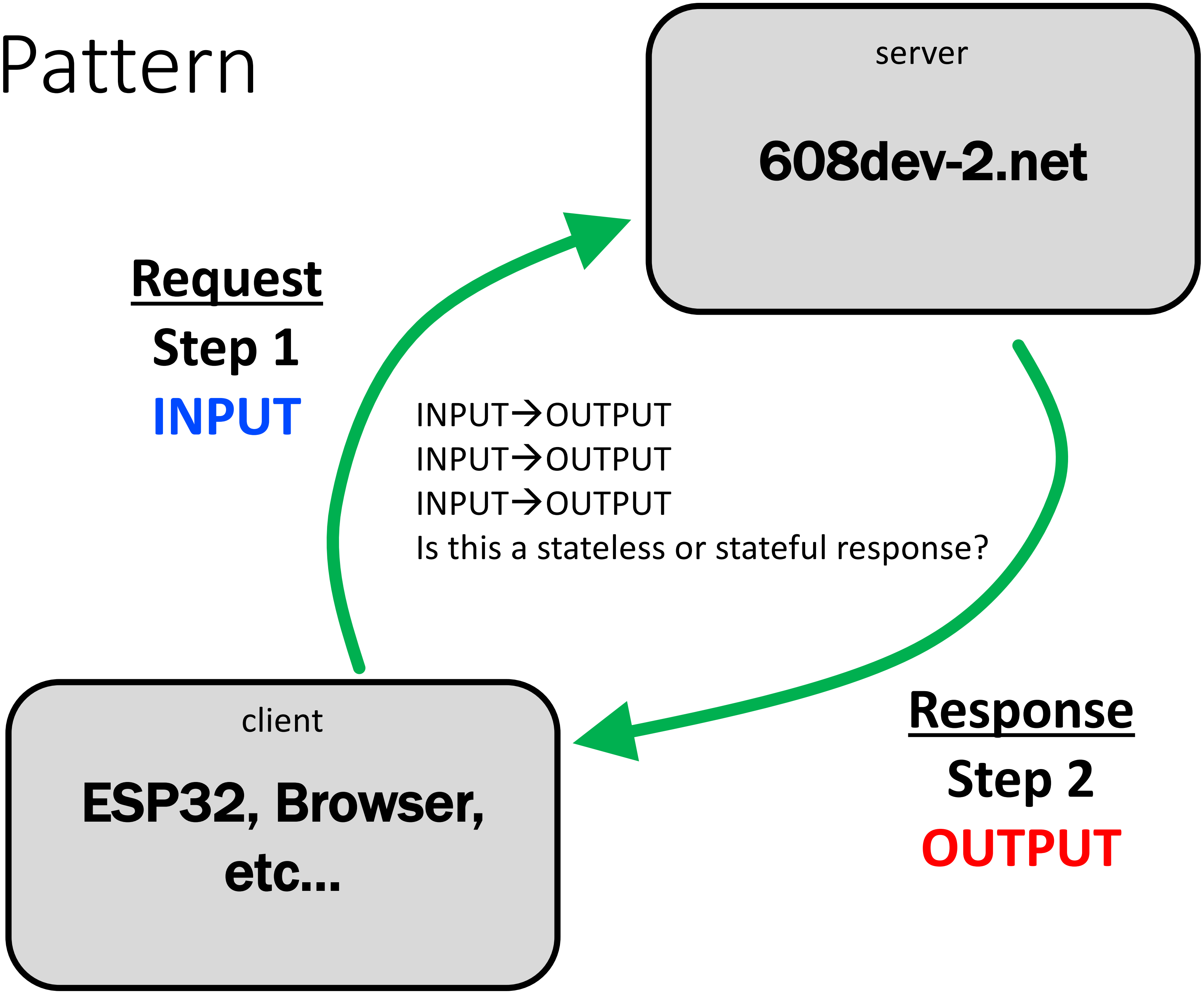
6.033
Computer Systems
Engineering

Back to HTTP: Basic Pattern

- The client sends a request to the server
- The server parses it, carries out the specified actions (as dictated by internal code), and then returns a response
- **There are two major verbs in HTTP requests that we use (though there are many others):**
  - **GET**
  - **POST**

*other HTTP verbs include: HEAD, PUT, DELETE,OPTIONS,TRACE,CONNECT

Basic Pattern

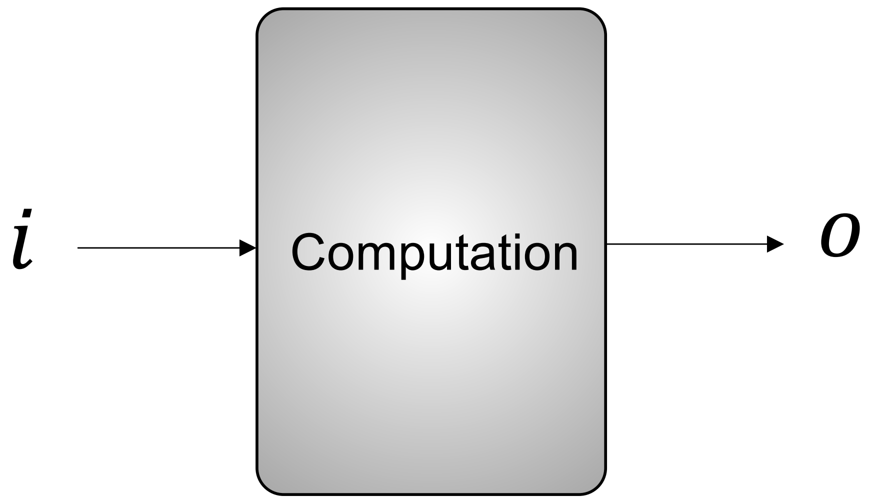


Systems on the Internet are the same as Systems not on the Internet

- They're still just digital computers that get inputs and produce outputs

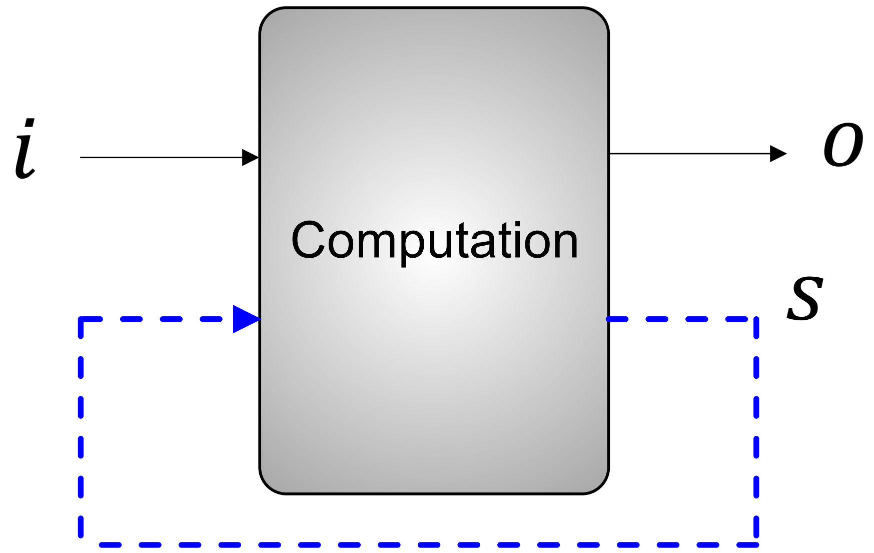
<u>STATELESS:</u>

$$o = f(i)$$



<u>STATEFUL:</u>

$$o_{n+1}, s_{n+1} = f(i_n, s_n)$$



State on the Server

- Servers are computers
- They're running most of the time
- They can read and write files and those files can store information over time

- One general way to read/write files efficiently on a server is with a **database**, which we'll start working with this week.
- If a server can store and access information based on previous interactions in its database, it has the ability to provide **stateful** behavior (response based on query and past information)

What makes a Database a Database?

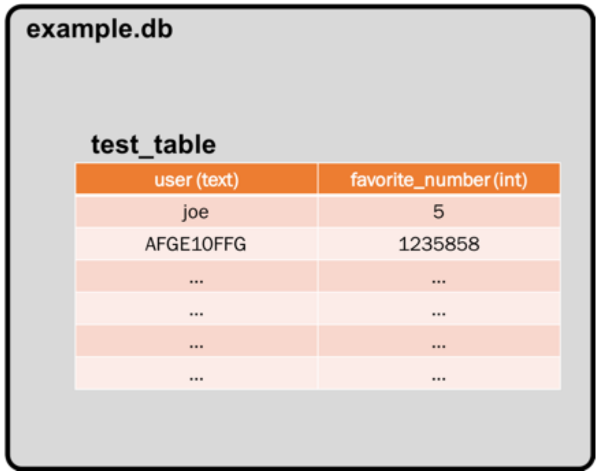
- Any persistent file-storing thing that allows quick storing and looking up of information
- A lot of design has to go into building a system that can quickly store and look up particular things!
- Deal with simultaneous connections!
- Deal with data stored across multiple machines!
- Lots to consider as things get very large!

6.814
Database Systems

Two Broad Classes

SQL

- Table-like data structure



example.db

test_table

user (text)	favorite_number (int)
joe	5
AFGE10FFG	1235858
...	...
...	...
...	...
...	...

NOSQL

- Tree-like or “dictionary” data structure

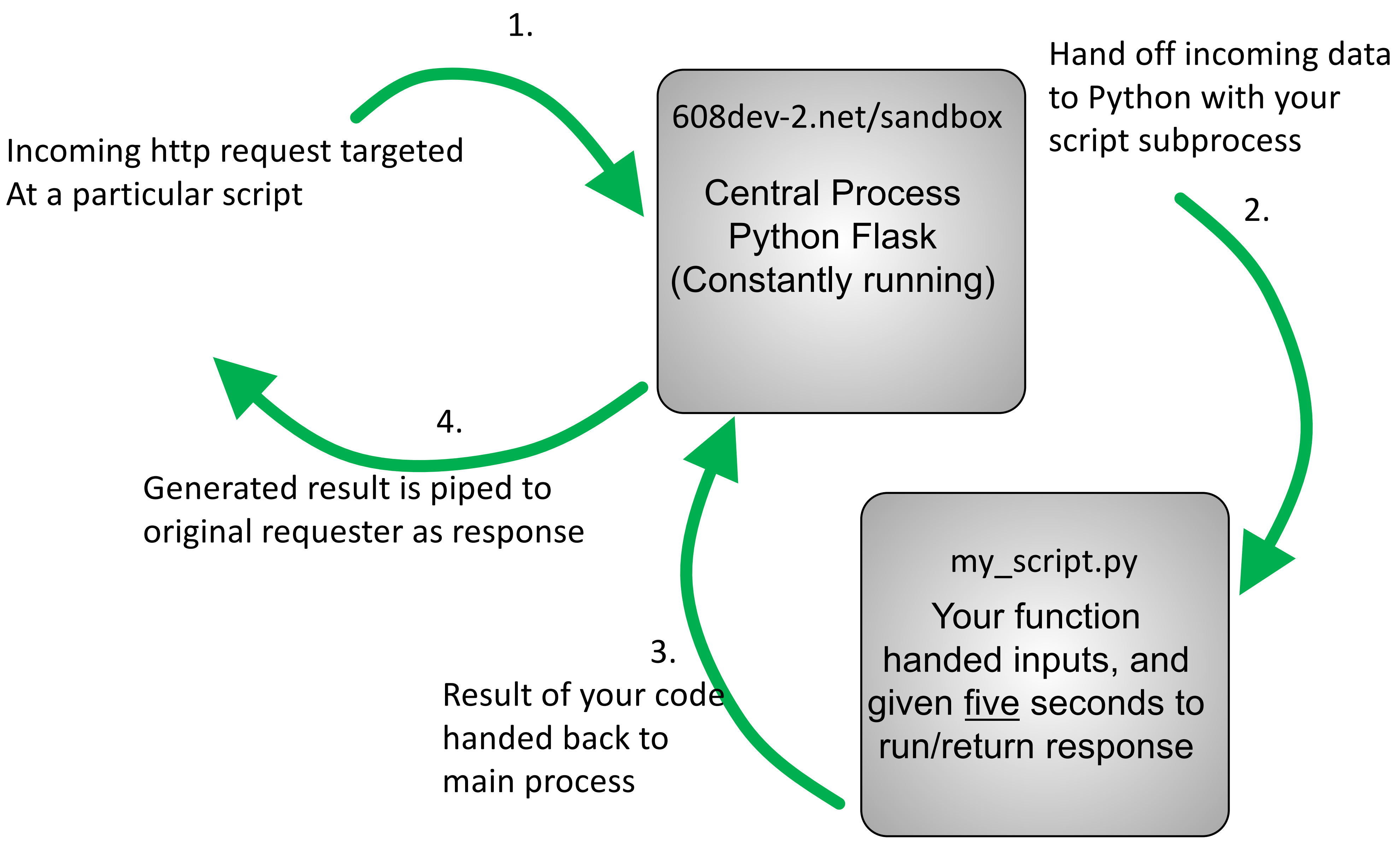
```
{'values': {}, 'data':
'{“cat”:”brown”,”dog”:”bl
ue”,"favorite_numbers":[1
,4,11]}', 'method':
'POST', 'args': [],
'is_json': True}
```

Both have strengths, both have weaknesses

We use SQLite in 6.08

- Arguably the most widely used database software on Earth
- Uses SQL dialect, which is good to get some experience with (Lab05B and beyond)
- Databases are files (easy to backup, move around)
- Lightweight and easy to set up (doesn't need a DB server running)

608dev-2.net Server



Must use Database to Store State!

- From request to request, your python code/functions need the ability to refer to history
- Need a database to act as global variables that live out of scope
- You cannot have a continuously running Python process on our server

The Meaning of GETs and POSTs and <u>State</u>

- GETs are meant to request a resource from a server. In general, a server should **not update its state** (databases) from a GET

- POSTs are meant to report/provide information to a server with the intent for it to be stored/logged. This is usually meant to be a **state-changing action**

Reality*

- You can do whatever with GETs and POSTs (put stuff in database in response to both of them if you want), but it is a good convention to use with when creating an API or interfacing with one

Do you think Facebook doesn’t update its “state” when you do a simple GET request to view messages? You are mistaken

GET

url has all key-value query arguments

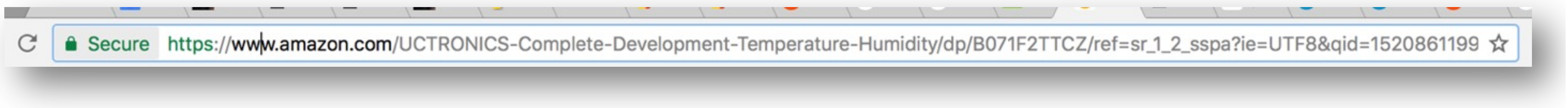
```
GET /sandbox/sc/jodalyst/special.py?cat=brown&dog=blue HTTP/1.1
Host: 608dev-2.net
```

On the Python side **of our web framework** the request dictionary looks like the following:

```
{'args': ['dog', 'cat'], 'method': 'GET', 'values': {'cat': 'brown', 'dog': 'blue'}}
```

Query arguments can act as inputs to our server side scripts!

GET is what you do in a web browser



Secure https://www.amazon.com/UCTRONICS-Complete-Development-Temperature-Humidity/dp/B071F2TTCZ/ref=sr_1_2_sspa?ie=UTF8&qid=1520861199

https://www.amazon.com/UCTRONICS-Complete-Development-Temperature-Humidity/dp/B071F2TTCZ/ref=sr_1_2_sspa?**ie=UTF8&qid=1520861199&sr=8-2-spons&keywords=esp32&psc=1**

- ie: UTF8 (text to render)
- qid: 1520861199 (query id…for logging/remembering actions)
- sr=8-2-spons (no idea)
- keywords=esp32 (my search query)
- psc=1 (no idea)

POST

url same as before

```
POST /sandbox/sc/jodalyst/special.py HTTP/1.1
Host: 608dev-2.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 18

cat=brown&dog=blue
```

How long is body

Specify how data in body is

body

On the Python side **of our system** the request dictionary looks like the following:

```
{'method': 'POST', 'form': {'cat':'brown', 'dog': 'blue'},
'is_json': False, 'values': {}, 'args': []}
```

POST Body Format

- POST requests have a Body (unlike GET). The body has much flexibility in the size and "shape" of data that it includes
- Two big ones we'll use in 6.08:
- `application/x-www-form-urlencoded:`
  - e.g. `cat=brown&dog=blue`
  - Key-value
- `application/json:`
  - e.g. `{“cat”:”brown”,”dog”:”blue”}`
  - Much more flexible and nestable, though complicated format (like a Python dictionary kinda)

POST Body Format

- `Content-Length:`
  - Sometimes not needed, but a good thing to include
  - Recommend you put in no matter what

POST

Json provides more flexibility in the structure of the body

```
POST /sandbox/sc/jodalyst/special.py HTTP/1.1
Host: 608dev-2.net
Content-Type: application/json
Content-Length: 56

{“cat”:”brown”,”dog”:”blue”,"favorite_numbers":[1,4,11]}
```

On the Python side **of our system** the request dictionary looks like the following:

```
{'values': {}, 'data':
'{“cat”:”brown”,”dog”:”blue”,"favorite_numbers":[1,4,11]}', 'method':
'POST', 'args': [], 'is_json': True}
```

POST

If you mix up the encoding/content-type some systems will throw errors, and some won't…our's will just shove it into 'data' field, so you might need to do some checking on that to see if it is indeed json

```
POST /sandbox/sc/jodalyst/special.py HTTP/1.1
Host: 608dev-2.net
Content-Type: application/json
Content-Length: 18

cat=brown&dog=blue
```

Wrong content-type for body

On the Python side **of our system** the request dictionary looks like the following:

{'data': 'cat=brown&dog=blue', 'method': 'POST', 'args': [], 'values': {}, 'is_json': True}

“Pros”/”Cons”?

- In a GET, all components are in the URL, including potentially things that matter. This is less secure since information is automatically stored in server logs
- In a POST you have a body (GET does not), if you are sending up potentially secure info, you should put it into the POST (and also encrypt..but we’ll worry about that later)
- POST is also far less limited in what you can put in body (images, audio, whatever) while GET is limited mostly to key-value pairs

Putting Query Arguments in POST?

- Yeah you can do it. It does sort of void the point and benefit of POST so try to avoid it unless you're using it to direct where data goes (not hard/fast rule)

```
POST /sandbox/sc/jodalyst/special.py?foo=bar&snow=no HTTP/1.1
Host: 608dev-2.net
Content-Type: application/json
Content-Length: 28

{“cat”:”brown”,”dog”:”blue”,”favorite_numbers”:[1,4,11]}
```

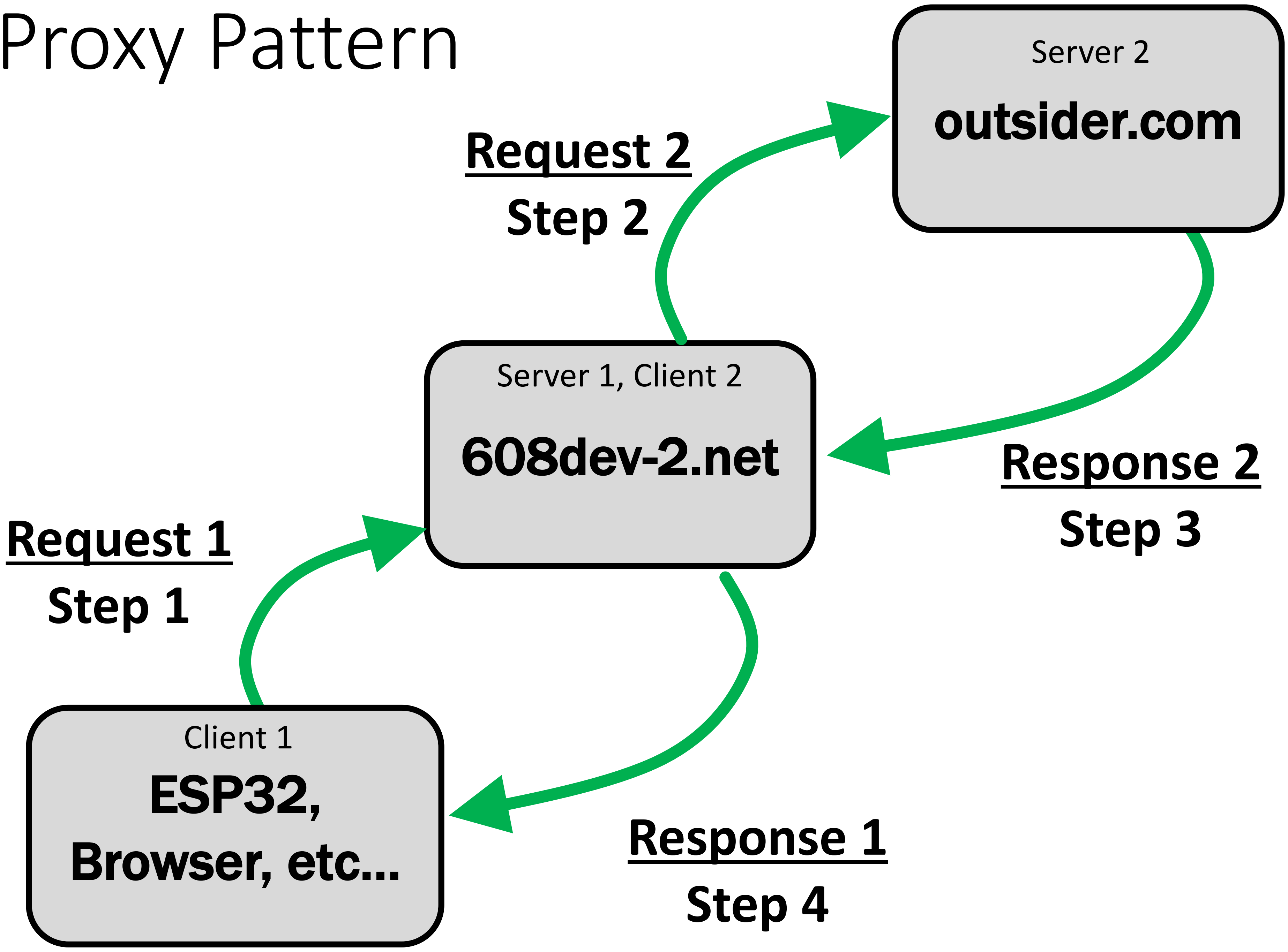
On the Python side **of our system** the request dictionary looks like the following:

```
{'args': ['snow', 'foo'], 'values': {'snow': 'no', 'foo': 
'bar'}, 'data': 
'{“cat”:”brown”,”dog”:”blue”,"favorite_numbers":[1,4,11]}', 
'method': 'POST', 'is_json': True}
```

Chained-Events

- Devices can act as both servers and clients in certain contexts.
- We will use this framework in the Wikipedia exercises (week 05), and a number of you are using this in current design exercises!
- This is an extremely common system in the world.
- Start building systems that exist across multiple platforms!

Proxy Pattern



Why not interface directly?

- Why not have your embedded device directly chat with some 3rd party server somewhere?
- Because if you do that then you're relying on other people,
- and other people will always let you down.

- If a server ***you*** control can interface between your deployed devices and outside resources you can:
  - Use server-side methods for processing/filtering data
  - Offload significant computation from deployed devices
  - More easily change server code in response to outside world changes (rather than recall your deployed devices)

Chain of Events

1. ESP (Client 1) sends request to 608dev-2.net (Server 1)
2. 608dev-2.net (Client 2) sends request to Wikipedia (Server 2)
3. Wikipedia (Server 2) provides response to 608dev-2.net (Client 2)
4. 608dev-2.net (Server 1) provides response to ESP (Client 1)

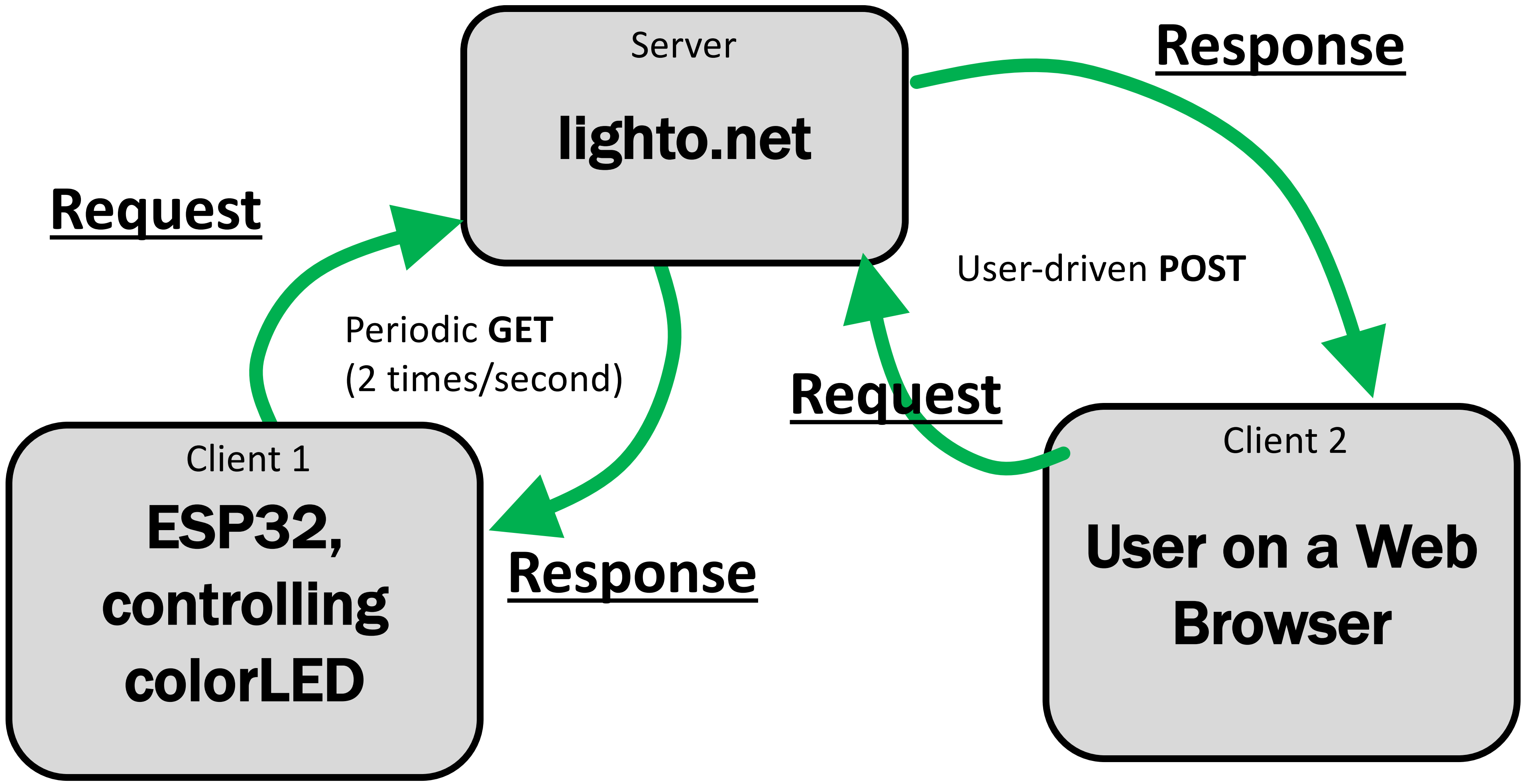
- The machine in the middle takes turns being both a server and a client (so the role can change)

Example/Design: Smart Lighting (Design Exercise)

hue
PERSONAL WIRELESS LIGHTING
Smart lighting, your way.
Get started by choosing a starter kit
Upgrade every bulb and fixture in your home
Easy to install
Screw in your Hue bulb
Plug in Hue Bridge and connect to Wi-Fi router.
Download Hue app and follow instructions to install.

hue
With Philips Hue, your voice is the key.
Voice control your lights.
LIGHTING
Easy to install
Screw in your Hue bulb.
Plug in Hue Bridge and connect to Wi-Fi router.
Download Hue app and follow instructions to install.

One way to do it



HTTP Requests

<u>2 times per second the smart light will:</u>

Request:

```
GET /light_control.py?lid=1989 HTTP/1.1
Host: lighto.net
```

Response:

```
HEADER stuff


level=42
```

Then make sure its lighting matches the set level

<u>Whenever User Wants:</u>

From a phone app or browser:

Request:

```
POST /light_control.py HTTP/1.1
Host: lighto.net
Content-Type: application/json
Content-Length: 56

{“light_id”:1989,”light_level”:42}}
```

Response:

```
HEADER stuff


change confirmed
```

**probably be some credentials in there too*

Just the Surface

- The “Internet” and the entire stack of software and hardware that it depends upon is extremely complex.
- Billions of lines of code in dozens of languages on millions of pieces of equipment all designed to work with one another efficiently.
- Is pretty amazing when it works.