Sockets and `poll()`

- Plan for this slide deck:
  - Learn a little about sockets
  - Think about blocking calls
  - Explore options for parallelism

Sockets

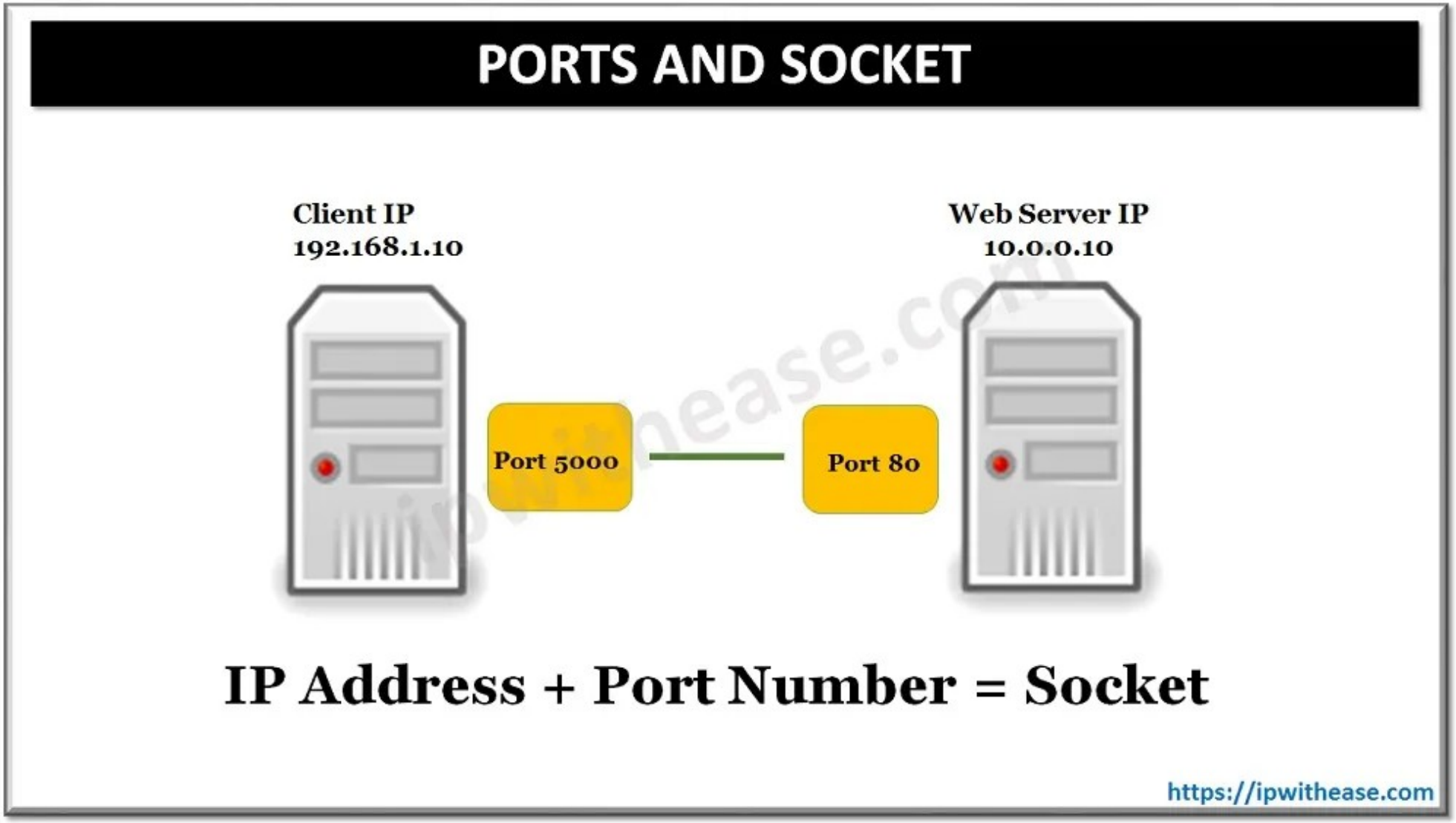
- What happens when we open a webpage?

  https://www.example.com

- There's a thousand details
  - HTML syntax
  - HTTP protocol
  - Encryption (SSL/TLS)
  - Sockets

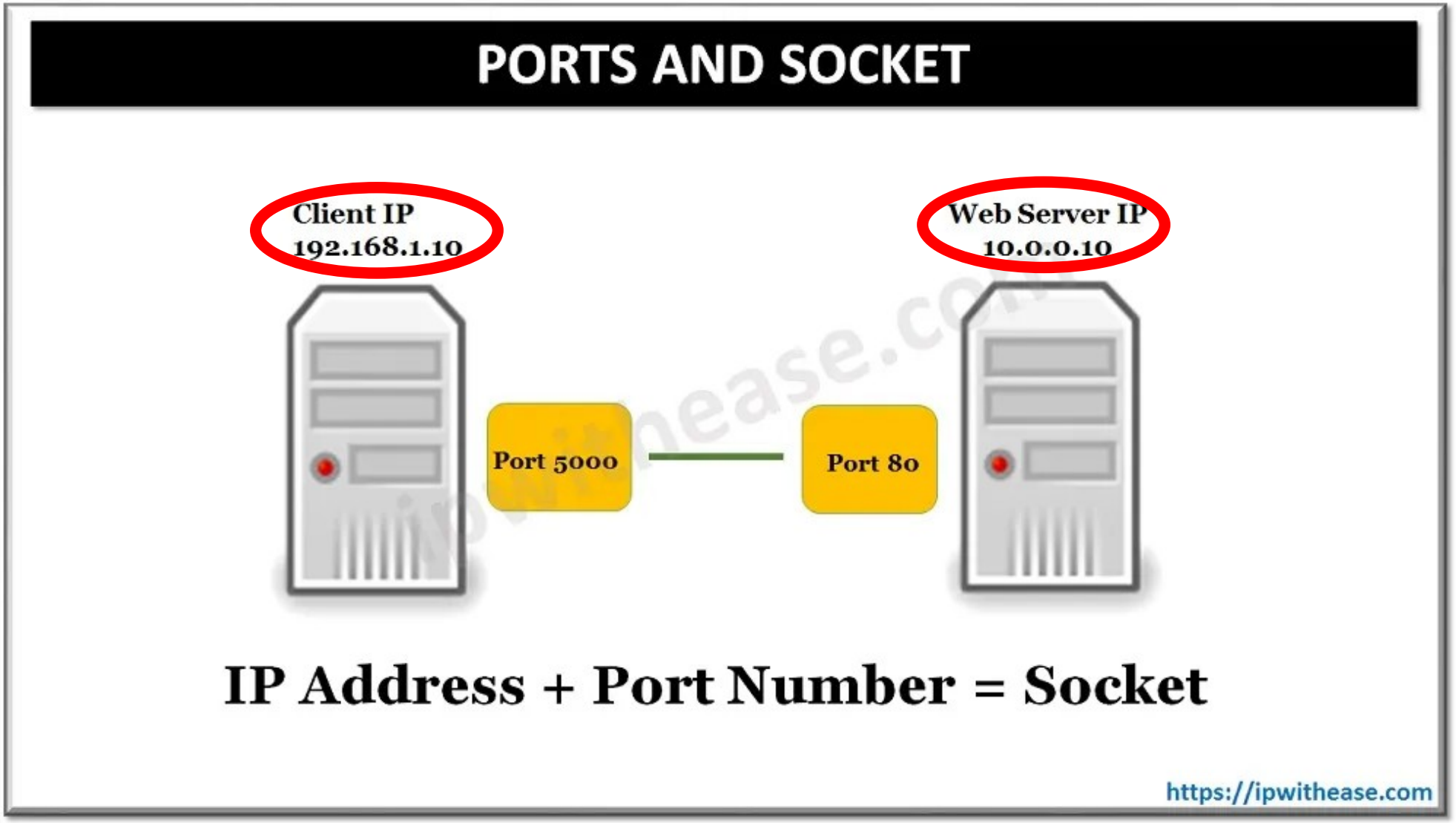
Sockets

- A **socket** is a bi-directional channel for sending data from one computer to another
  - Occasionally, between processes on a single computer
- Most sockets use TCP
  - In-order data
  - Automatic re-transmit of lost packets
- Outside the scope of today: UDP, UNIX sockets, etc.



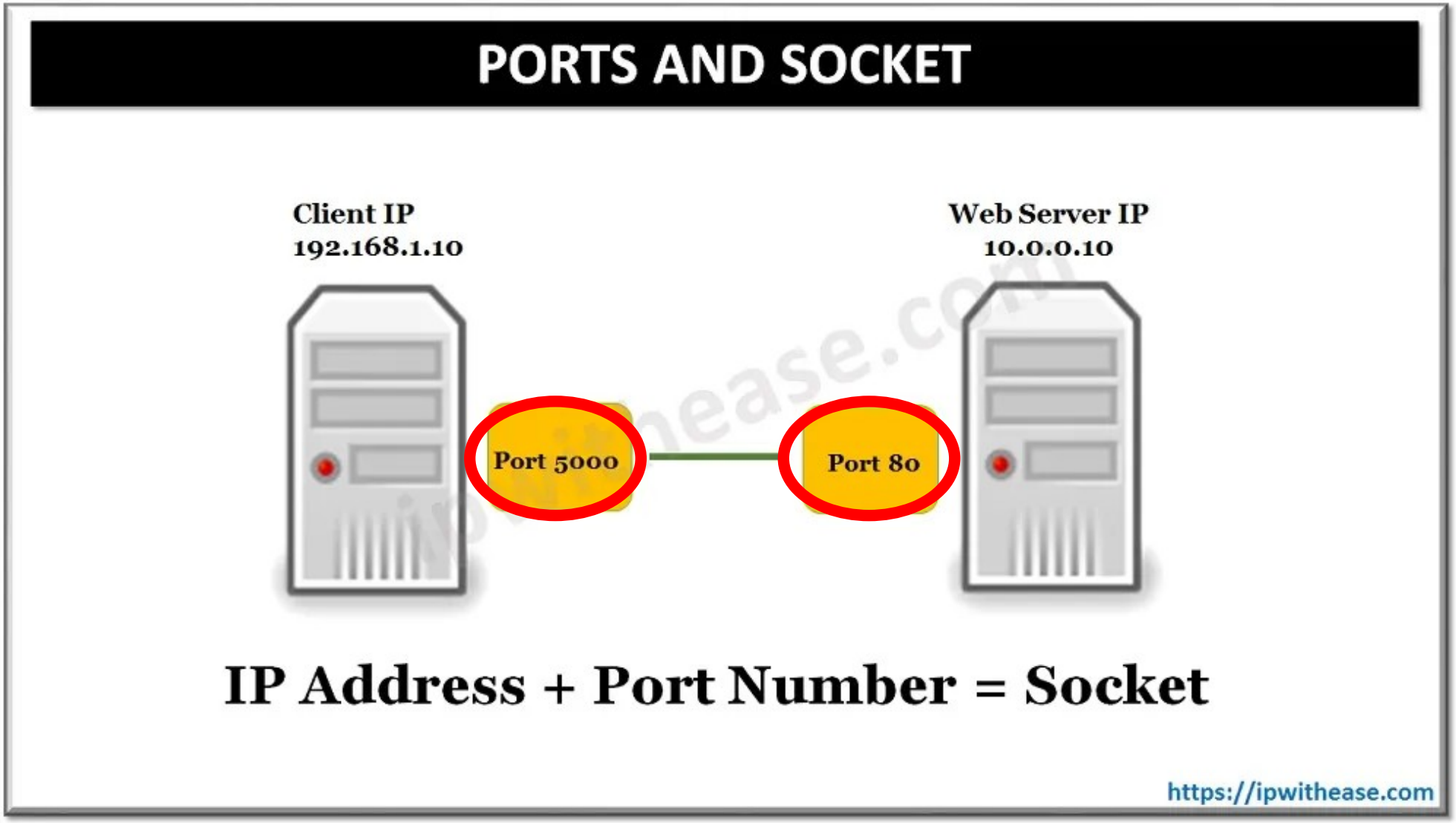
- All TCP connections have a **client** (which asks for the connection) and a **server** (which listens for connection requests)

Image: https://ipwithease.com/ports-and-socket-explanation/



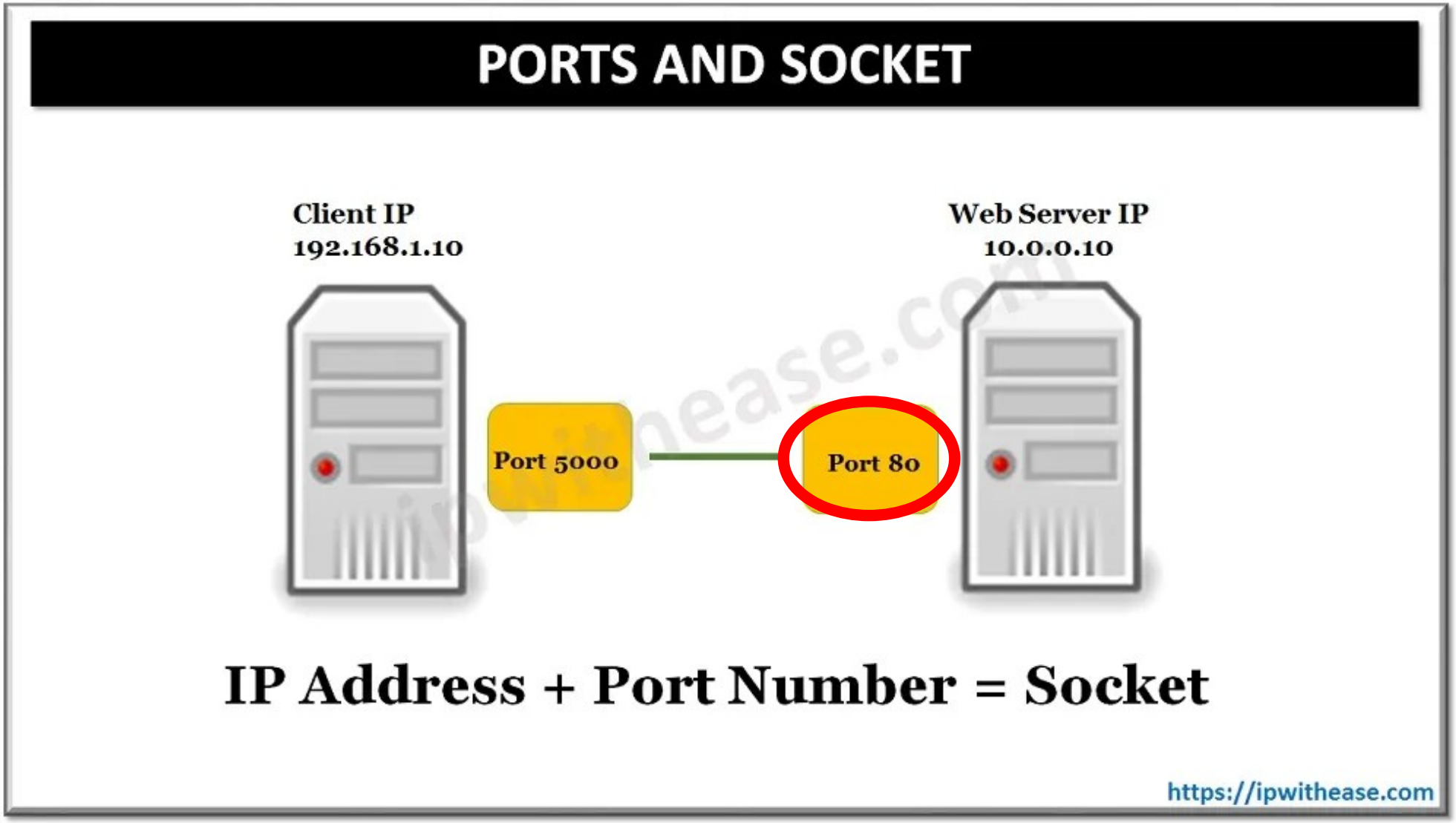
- IP addresses identify computers
  - Multiple IPs per computer

Image: https://ipwithease.com/ports-and-socket-explanation/



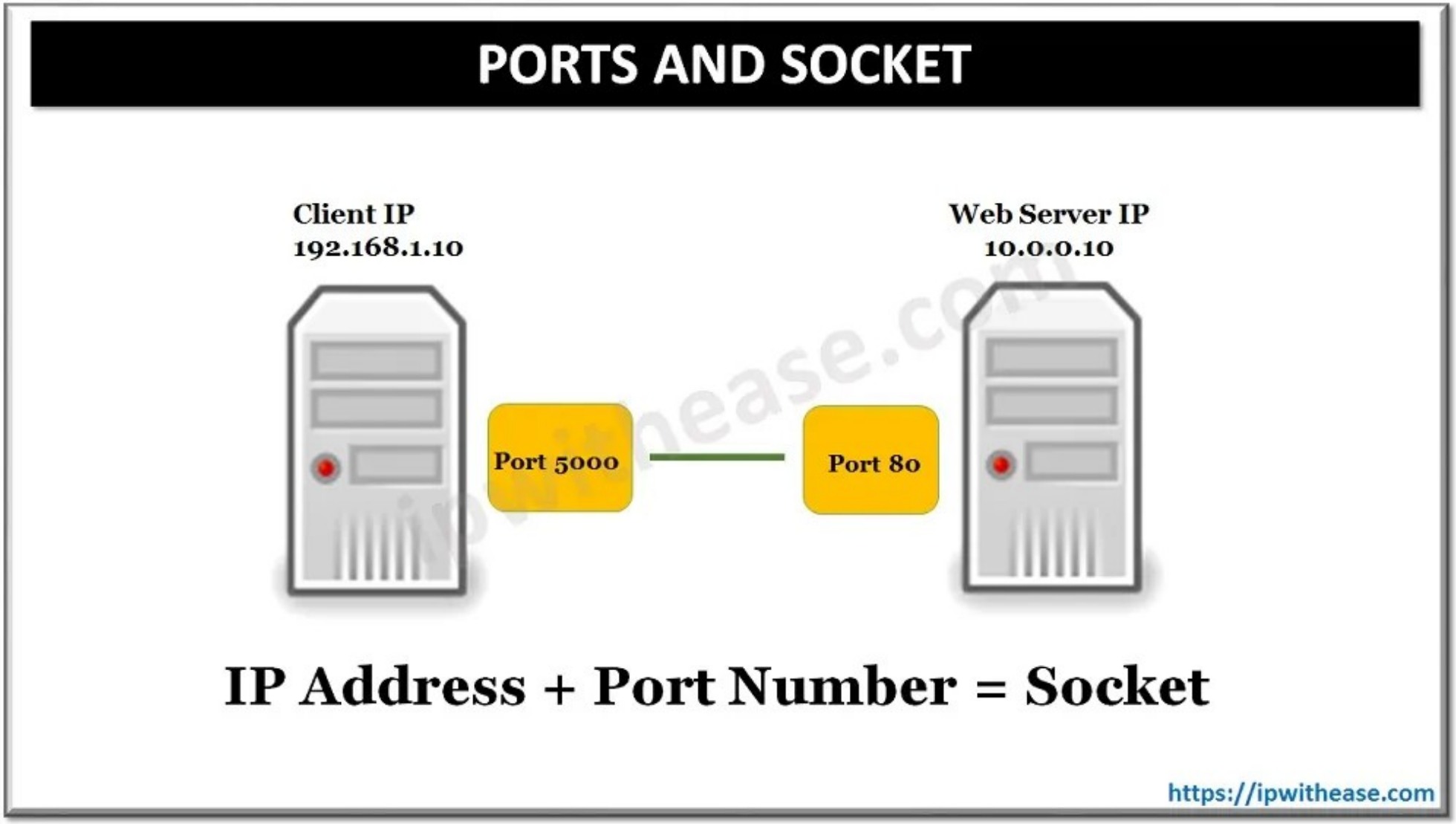
- Port numbers allow us to have multiple connected sockets

Image: https://ipwithease.com/ports-and-socket-explanation/



- On the server, the port number is fixed and well-known.
- On the client, it is assigned at random.

Image: https://ipwithease.com/ports-and-socket-explanation/



- On the server, we **listen** on our assigned port.
- On the client, we **connect** to a remote IP / port.

Image: https://ipwithease.com/ports-and-socket-explanation/

Sockets

- In UNIX, a socket is used to represent either:
  - A listening socket (1 per server)
  - A connected socket (many per server, 1 per client)

- All connected sockets have two parallel channels: `send() / recv()`
  - All data sent in order, on each channel
    - Analogous to our terminal devices from Phase 4

Sockets

- In this deck, we are using raw TCP sockets
- In the real world, you will generally use a library, which adds complexity on top!
  - Encryption
  - High-level protocol (HTTP, etc.)

Questions so far?

- Any questions so far?

- We’re about to look at code. I won’t explain it all. But ask whatever you want!

Sockets

See class website for files.

- Read the code in `socket_server.c`
  - How does it set up the server?
  - What happens when it accepts an incoming connection?
  - What does `worker()` do?
  - What happens if multiple clients connect?
- Feel free to run this program
  - Choose a port from 1025 ... 65535
  - Test with `telnet localhost PORT`

Sockets

- Read the code in `socket_client.c`
  - Compare to the server using `diff`
  - What is the same, what is different?

- Open up multiple windows to Lectura
  - Run one server
  - Connect 3-4 clients

Better Server?

- The server **blocks** while serving one client
  - Minor quirk if clients are fast
  - But clients are ***usually slow!***
    - Long, interactive sessions
    - Big downloads / uploads

- An simple (kludgy) solution: `fork()`

Sockets

- `socket_server_fork.c`
  - `diff` it against the original server
  - Creates a child process for every connection

- Pay attention to how `fork()` works
  - Not like `spork()` from Phase 1
  - Note that `exec()` is not required
  - Note how / when we close open sockets

Sockets

- `socket_server_pthreads.c`
  - `diff` it against the original server

- Creates a **thread** for every connection
  - Shared address space
    - Real world: ***threads share data***

So What's Wrong with Threads?

- `fork()` and `pthreads` are wonderful, but…
  - Slow. Why?

So What's Wrong with Threads?

- `fork()` and `pthreads` are wonderful, but…
  - Slow. Why?

- Context switches!
  - Imagine 100s of new clients per second
  - Or 100s of clients in parallel
    - Trickling data to you, a few packets at a time

`poll()` for Non-Blocking I/O

- `poll()` allows you to block, waiting for readiness on any of **many** files
- One thread can handle all of the I/O

```
fds[] = [list of files]
While (1) {
    poll(fds);
    foreach (file ready for I/O)
        do_the_IO();
}
```

`poll()` for Non-Blocking I/O

- `socket_server_poll.c`
  - How to set up the `fds[]` array?
  - How is a listening socket handled?
  - What happens when we accept a new connection?

- New server allows for multiple exchanges
  - Closes sock when client disconnects
  - `socket_client_looping.c` or `telnet`

Additional Thoughts

- What if the client needs to listen in both directions at once?
  - `poll()` isn't just for servers!

- Sockets can block while transmitting
  - Ex: Sending a huge file
  - Real systems use `poll()` to check for ***write*** as well as ***read***

Summary

- Sockets are the low-level way that computers connect
  - TCP: Bidirectional, file-like
  - Lots of layers above
- Need a way to listen to multiple sockets
  - `fork()` or `pthreads` are easy but beware performance
  - `poll()` more efficient