Network Programming 101 & node.js.

- Sockets,
- servers, clients
- ~~event loop~~
- event-driven programming
- ~~event-driven~~ network printing.

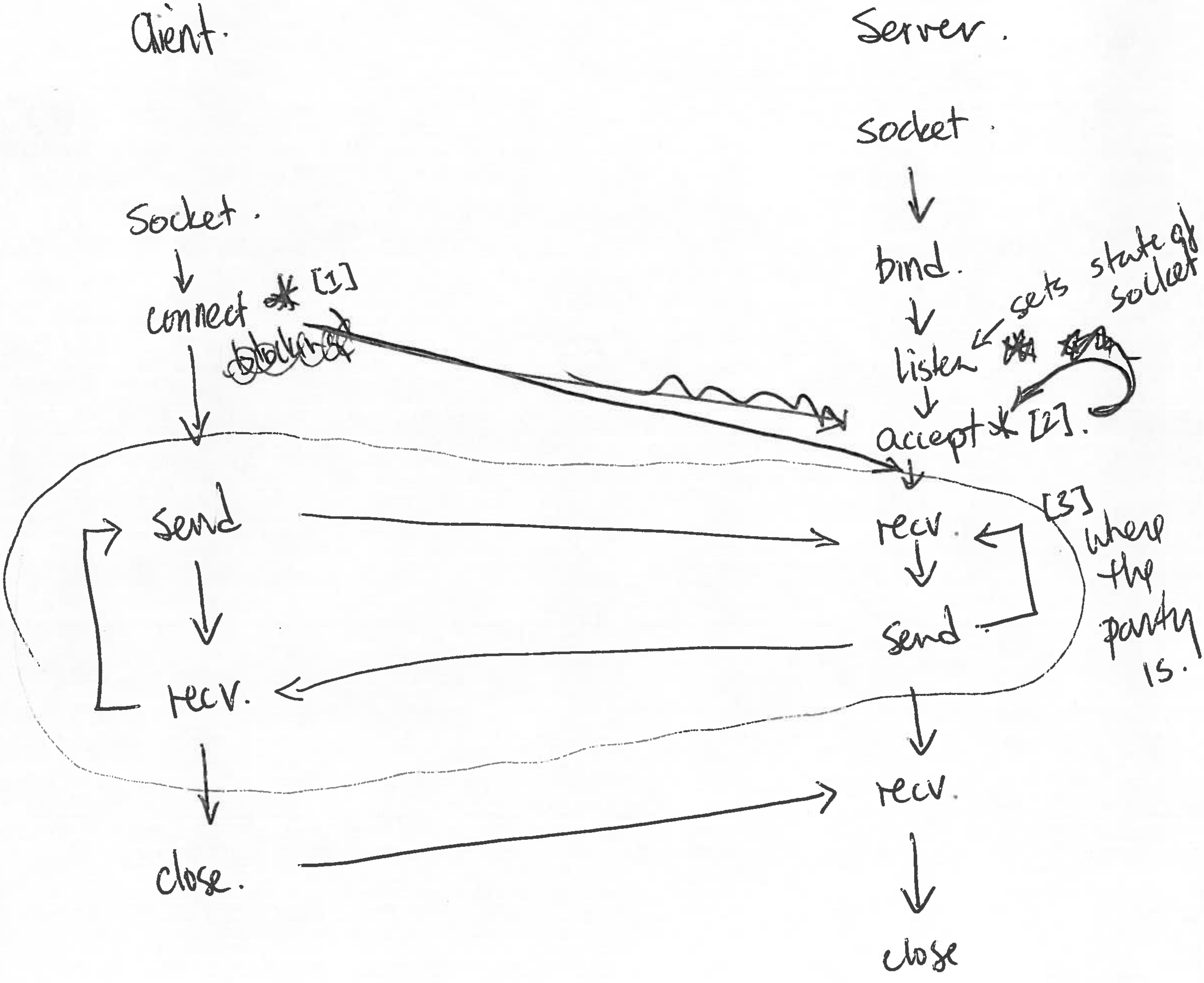
(tight timing overall)

Sockets API:

- unified concept for dealing with connections at the Internet layer.
- originated from BSD in late 80s, became the defacto standard, and has evolved into the POSIX spec.
- mostly similar across languages (C, Java, C#, Python, etc.)

- Socket:
  - endpoint for communicating to another process. (which exposes another endpoint)
  - process can be on another machine.
  - you can read & write to it, like a file.
  - usually, a socket connection is ~~spe~~ known by two (address, port) pairs
  - netstat lets you see the connections currently to your own machine!
    - -b ~~for~~ bytes.
    - -a for server
    - -f inet

TCP Socket Flow Diagram.



[1] blocks until the connection is established

[2] blocks until a new connection comes in
- creates a new socket paired w client just to communicate w client.
- at this point, you can spawn a new thread to have it take care of the client connection.

[3] the party is here

Server Pseudo code.

```
s = new socket (internet, streaming)
s.bind (IP, port)
s.listen()
while true:
    clientSocket = s.accept()
    while true:
        input = clientSocket.recv(1024)
        if input is emty
            break.
        clientSocket.send(input)
    clientSocket.close()
```

leave gap.

Client Pseudo Code.

```
s = new socket (internet, streaming)
s.connect (IP, port)
while true:
    string = readLine()
    if string = close, break.
    s.send (string)
    reply = s.recv(1024)
s.close()
```

Multithreaded server

- common pattern: one thread listens for connects
  new threads are spawned to actually deal
  to each connection

```
t = new thread(~~.~~ client Socket ) {
    ~~run(client~~
    while true

} .run().
```

~~Client, is often~~

- Interactive clients are also often threaded, too.
- reason: messages might come any time, and not just as responses to some client → server communication. (e.g. IM server may send "Joe is online" at any time!)

EVENT ~~QUEUE~~ LOOP / Event-Driven Programming

- a recent phenomenon has been to simply network programming.
    - challenge: multi-threading / deadlocks
    - challenge: interaction to UI.
    - challenge: blocking operations can be very expensive
- insight: actually, UIs have some nice strategies to deal with this kind of stuff.

Check this example

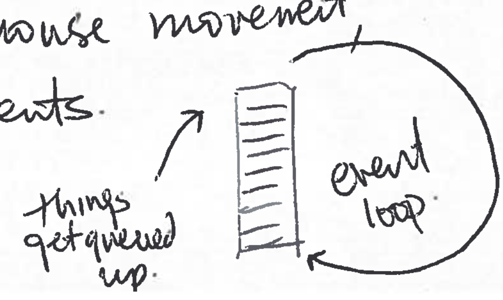
C# ProgressBar & BackgroundWorker

```
pBar = new ProgressBar
pBar.show

for (var file in files) {

    file.read()
    pBar.performStep()

}

//doesn't work!
```

```
↳ RunWorkerAsync() {
    ProgressChanged - %
}

bgw.ProgressChanged +=
~~new ProgressChanged~~
(o, e) => {
pBar.performStep
}
```

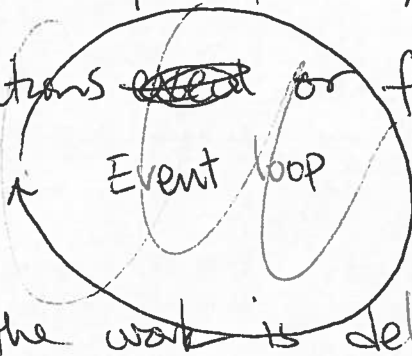
*screwed

- with UIs, usually, they are working on an event loop, responding to events like mouse movement, clicks & keyboard or touch events.

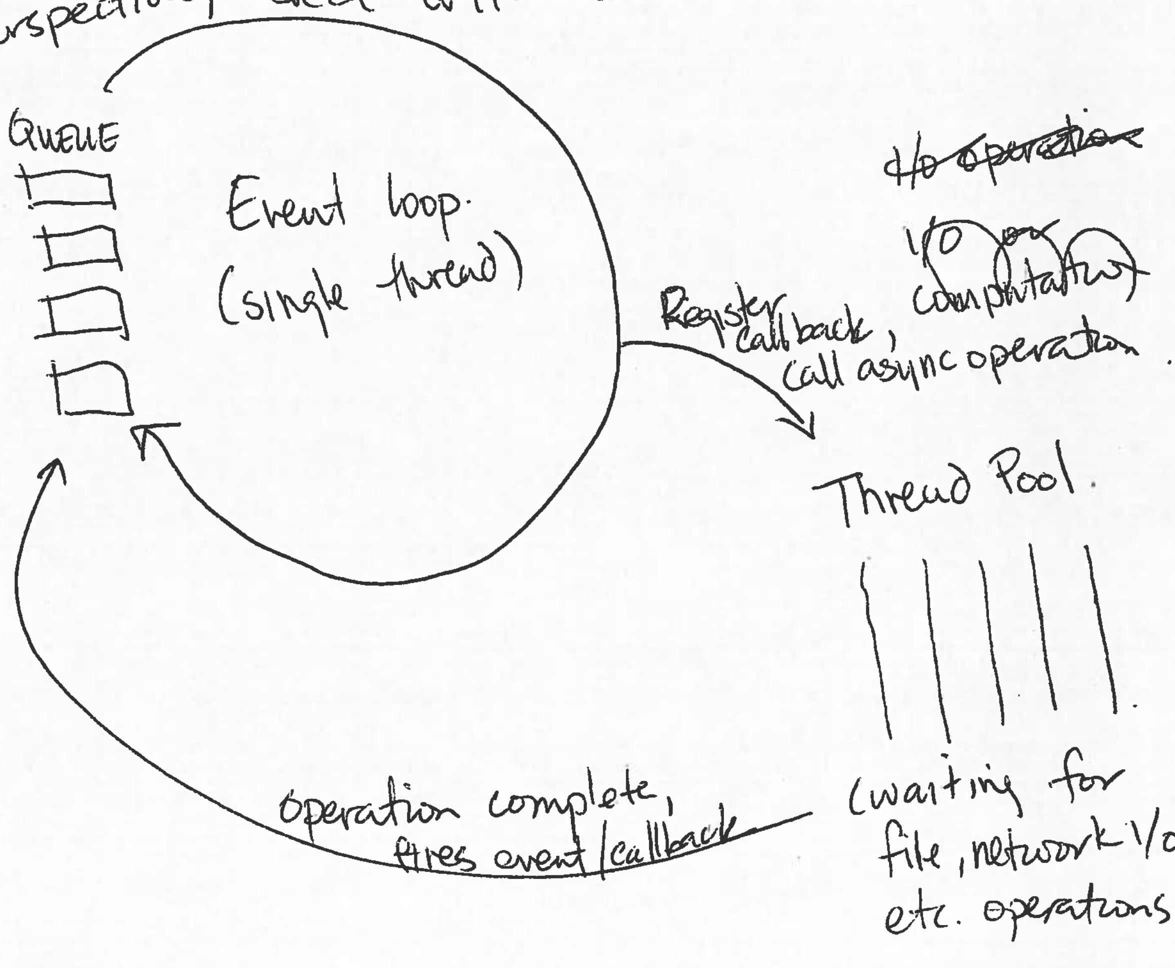


Event ~~Que~~ Loop in Network Programming

- node.js implements this style of programming natively.
- most network operations ~~exec~~ or file operations (read, write, etc.) are implemented as asynchronous operations



- under the hood, the work is delegated to the thread pool.
- from our perspective, deal with it when it comes back.



examples from node.js. 7/8

- http.createServer()

  http.on('request', ~~functio~~ requestFunc)

  ↑ register for the 'request' event, & execute requestFunc when it happens

```
function requestFunc(req, res){
    res.writeHead(200, ...)
    res.end("Hello world!")
}
```

Short hand:

```
http.on('request', function(req, res){
        :
        })

http.createServer(function(req, res){
        :
});
```

- ~~http~~

- fs.readFile(homeFile, func(err, data){

```
      console.log(data)
});
```

- socket.connect(port, [host], [connection listener])

  'data'

  'connect'

  :

8/8

* note: publish / subscribe pattern.

~~[illegible]~~ WPF(C#)	node.js
UIElement events:	socket events:
Mouse Click	'connect'
~~Mous~~	'data'
Touch Down	'end'
Touch Up	'timeout'
⋮	'error'
	'close'

- you can subscribe to any of these events, and ~~multiple~~ register multiple listeners if you want!

PROTOCOLS, MESSAGES & MESSAGE FORMATTING

- protocol: describes format of messages, expected ordering / style of communication b/n different processes.

- message formatting is important:
  - recipient needs to know how to decipher a message
  - * needs to know when the end of a message has been reached! *

- ~~code~~ encoding a message. can be done in a lot of ways: ASCII, binary, XML, JSON, ...

- many protocols use ASCII encoding:

```
HTTP :   GET /index.html  HTTP/1.0
SMTP :   HELO smtp.ucalgary.ca.
          MAIL from: tony@ucalgary.ca
          DATA
           Word!
```

commands ⏟ parameters

notice: easy to read & debug (line-~~terminated~~ oriented protocols)

Delimiting Messages. 2/5

- "stuffing": SMTP: every header is on a line of its own, multiline 'DATA' ends with "."
  - Client: can start sending w/o entire message constructed
  - server: need to slowly process by looking @ each line to see if it's the end
- "counting" HTTP messages indicate how many bytes it contains.
  - sender: need to know entire length before sending
  - recv: just get read length & go!
- "blasting" FTP: new socket per file, close when done.
  - send/recv: can be stupid & expensive if multiple files

- marshalling: gathering parameters & encoding them for transmission
- unmarshalling: unpacking for use by your system.

- for the most part, marshalling ⟺ serialization
  ↳ "making send"

~~most~~
- most complexity is ~~in~~ building a protocol that can deal with arbitrary objects.
- well-defined protocol ~~&~~ & known client/server allows us to take short cuts (i.e. not deal with it manually)

- most systems have a mechanism to help w/ serialization.
  - this process is <u>OPAQUE</u> (who knows what's going on)

Python
```
dict = {'foo': 'bar', 'tony', ... }
data = pickle.dumps(dict)
s.send(data)
```
———
```
data = s.recv(1024)
dict = pickle.loads(da)
```

~~Java~~ C#
```
[Serializable]
class Foo {
  ⋮
}
f = BinaryFormatter()
f.Serialize(stream, myObj
```

Javascript.
```
JSON.stringify(myObj)
var obj = JSON.parse(json)
```

4/5

Protocol Design

- request / response / notification
  - request → from client
  - response → in response
  - notification → from server asynchronously (think "Joe is online!")
- often, you will use sequencing number so you can pipeline, send multiple
- reply codes from servers can help indicate what happened

HTTP:
- 1XX informational
- 2XX success → 200 OK.
- 3XX redirection → 301 Moved Permanently
- 4XX client error → 404 Not found
- 5XX server error → 500 Internal server error.

... Fence Sketching (lo-fi prototypes)
- time is at the top moving down
- shows inter-actions b/w hosts.