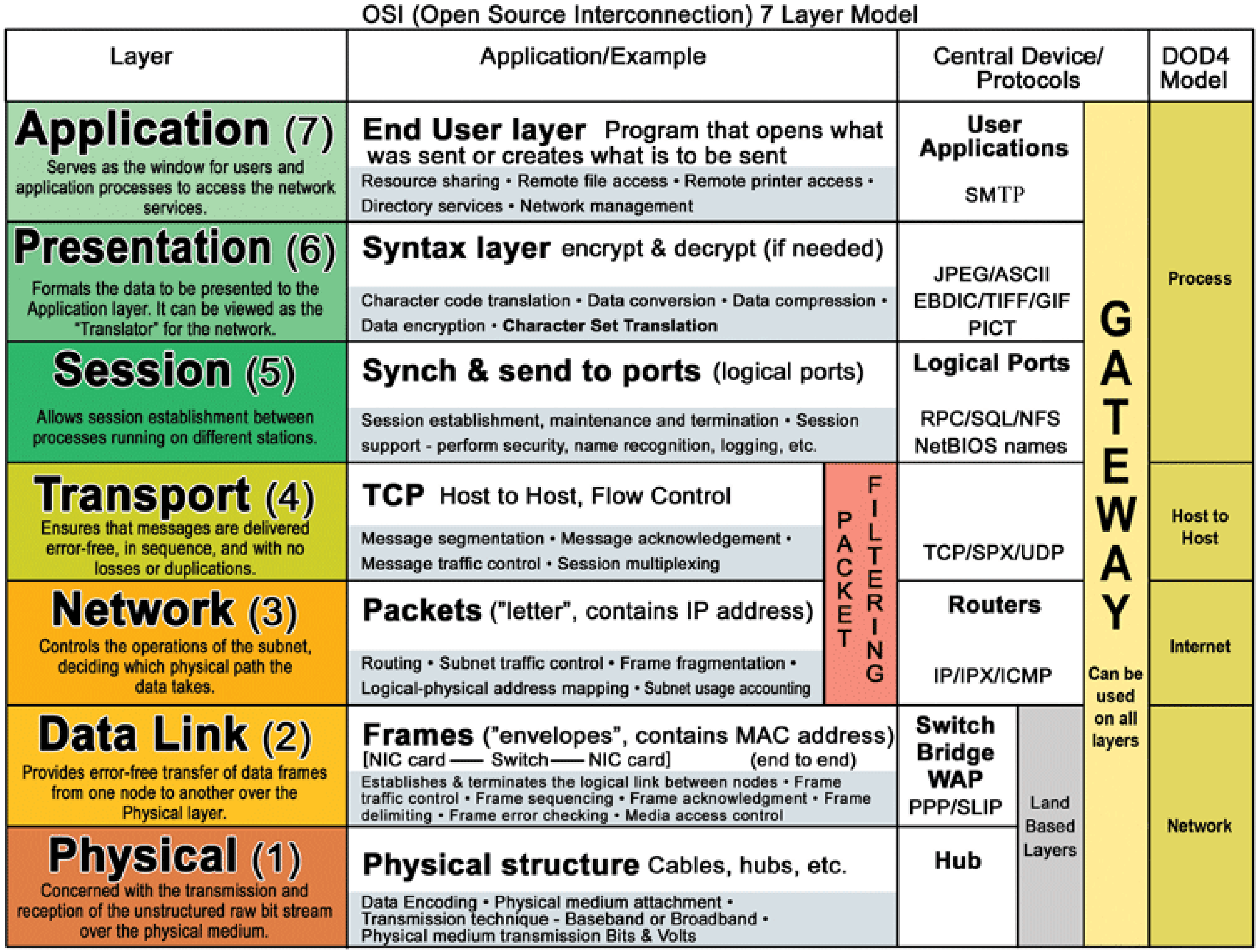
Drilling Down the OSI Stack

Preston Maness and Shams Mansoor

What is the OSI Stack?

vvv it's this thing vvv



OSI (Open Source Interconnection) 7 Layer Model

Layer	Application/Example		Central Device/ Protocols			DOD4 Model
Application (7) Serves as the window for users and application processes to access the network services.	**End User layer** Program that opens what was sent or creates what is to be sent Resource sharing • Remote file access • Remote printer access • Directory services • Network management		**User Applications** SMTP		**GATEWAY** Can be used on all layers	Process
Presentation (6) Formats the data to be presented to the Application layer. It can be viewed as the "Translator" for the network.	**Syntax layer** encrypt & decrypt (if needed) Character code translation • Data conversion • Data compression • Data encryption • **Character Set Translation**		JPEG/ASCII EBDIC/TIFF/GIF PICT			
Session (5) Allows session establishment between processes running on different stations.	**Synch & send to ports** (logical ports) Session establishment, maintenance and termination • Session support - perform security, name recognition, logging, etc.		**Logical Ports** RPC/SQL/NFS NetBIOS names			
Transport (4) Ensures that messages are delivered error-free, in sequence, and with no losses or duplications.	**TCP** Host to Host, Flow Control Message segmentation • Message acknowledgement • Message traffic control • Session multiplexing	PACKET FILTERING	TCP/SPX/UDP			Host to Host
Network (3) Controls the operations of the subnet, deciding which physical path the data takes.	**Packets** ("letter", contains IP address) Routing • Subnet traffic control • Frame fragmentation • Logical-physical address mapping • Subnet usage accounting		**Routers** IP/IPX/ICMP			Internet
Data Link (2) Provides error-free transfer of data frames from one node to another over the Physical layer.	**Frames** ("envelopes", contains MAC address) [NIC card —— Switch —— NIC card] (end to end) Establishes & terminates the logical link between nodes • Frame traffic control • Frame sequencing • Frame acknowledgment • Frame delimiting • Frame error checking • Media access control		**Switch** **Bridge** **WAP** PPP/SLIP	Land Based Layers		Network
Physical (1) Concerned with the transmission and reception of the unstructured raw bit stream over the physical medium.	**Physical structure** Cables, hubs, etc. Data Encoding • Physical medium attachment • Transmission technique - Baseband or Broadband • Physical medium transmission Bits & Volts		**Hub**			

An Example Use-Case

- It's 1630. In half an hour you'll be home free. But in the meantime, you're chained to a corporate culture that insists you keep tapping your keyboard until precisely 1700.
- You would jam out to something on Spotify to go along with the solo drum routine that you're subjecting your keyboard-turned-drum-machine to.
- Too bad IT won't let you install software on the machine.
- Your iPhone 5 S comes to the rescue. Time to jam out.

The Problem

If you could keep typing that would be ggrrrrreat

The Solution

Layer 7 - Application

- Spotify develops an iOS app
- Spotify submits it to the ~~Apple Overlords~~ App Store QA Team
- You download the app
- You surreptitiously activate the program when Lumbergh isn't looking
- Earbuds to the rescue!

- You select your favorite playlist, chock full of this year's top 40 junk and assorted EDM
- (Don't worry. I won't judge. Too harshly.)
- You hit play

Layer 7 - Application

```
03:39:57.598 I [ap_connection_impl.cpp:898      ] Connecting to AP ash2-accesspoint-
a13.ap.spotify.com:4070

03:39:58.244 I [upnp.cpp:513                    ] 192.168.0.1: got external ip 0x[REDACTED]

03:39:58.310 I [upnp.cpp:461                    ] 192.168.0.1: mapping add ok
03:39:58.316 I [upnp.cpp:487                    ] 192.168.0.1: Port 27020 mapped OK

04:44:56.119 I [file_streamer.cpp:1816          ] Getting CDN url:
http://audio2.spotify.com/audio/[LONG-HASH-REDACTED]
04:44:56.467 I [http.cpp:887                    ] Result 206 Partial Content

==================================================================

$ netstat -pnt | grep spotify | grep ':27020' | wc -l
39
$ netstat -pnt | grep spotify | grep ':27020' | head -n 1
tcp        0      0 192.168.0.133:27020     75.149.125.217:3199     ESTABLISHED
16956/spotify
```

Layer 7 - Application

- Wow! Spotify is quite a busy bee:
  - UPNP
  - 39 TCP connections to various IPs
  - **HTTP to CDNs (and ad networks)**
- And all of this to get some mp3 files, which tosses us to Layer 6!

Layer 6 - Presentation

Layer 6 - Presentation

- Technically, the audio codec that Spotify sends varies depending on the platform
- Regardless, iOS probably exposes an API –or there exists some other, external library– that the Spotify application can utilize to process these audio codecs; this library is the “layer 6” interface that feeds structured data to the application in “layer 7”
- Disclaimer: In practice, these layers often overlap and bunny-hop over each-other.

Layer 5 - Session

- Oh goody! This is the part where the kernel really starts to get involved!
- Remember all those TCP connections that Spotify opened?
- Yeah. The kernel does a lot of the heavy lifting.
- Granted, the application might have its own session handlers too.
  - Which is probably a separate process or thread that shimmies data between the kernel and the main Spotify application.

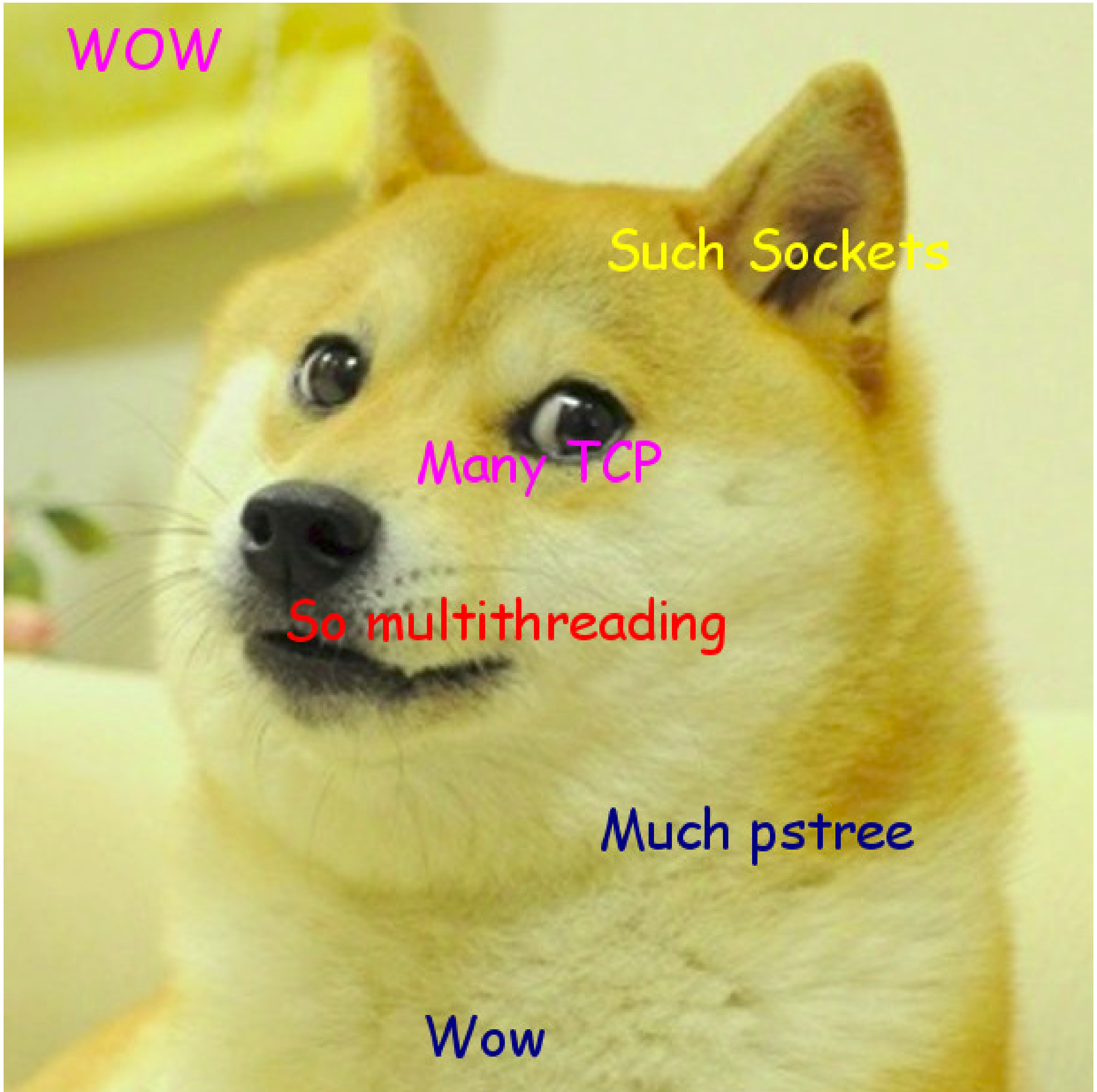
Layer 5 - Session

```
$ netstat -apn | grep spotify | wc -l
95

$ netstat -apn | grep spotify

[A small sample of the output]

tcp        0      0 192.168.0.133:27020     67.84.131.211:57746     ESTABLISHED 16956/spotify
tcp        0      0 192.168.0.133:27020     75.74.194.1:59385       ESTABLISHED 16956/spotify
tcp        0      0 192.168.0.133:27020     208.105.72.142:51236    ESTABLISHED 16956/spotify
tcp        0      0 192.168.0.133:27020     72.28.38.93:54983       ESTABLISHED 16956/spotify
udp        0      0 192.168.0.133:1900      0.0.0.0:*                           16956/spotify
udp        0      0 0.0.0.0:57621           0.0.0.0:*                           16956/spotify
udp        0      0 192.168.0.133:21328     0.0.0.0:*                           16956/spotify
unix  3      [ ]         STREAM     CONNECTED     1310753  16956/spotify
unix  3      [ ]         SEQPACKET  CONNECTED     1310737  16956/spotify
unix  3      [ ]         STREAM     CONNECTED     1311180  16956/spotify
unix  3      [ ]         STREAM     CONNECTED     1311007  16956/spotify
unix  3      [ ]         SEQPACKET  CONNECTED     1310736  16956/spotify
```

Layer 5 - Session

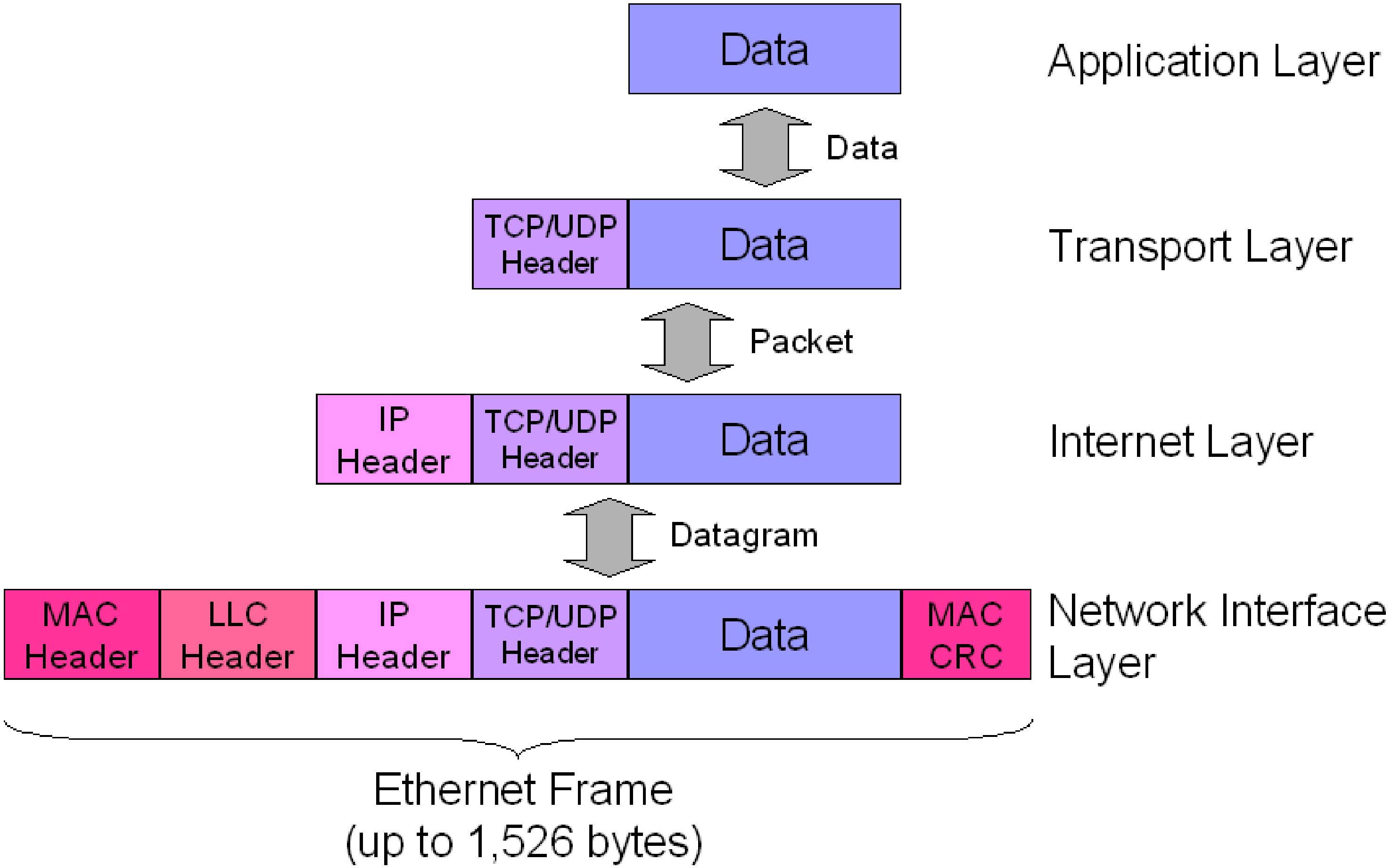
```
$ pstree 16956
spotify─┬─SpotifyHelper───7*[SpotifyHelper─┬─{Chrome_ChildIOT}]
        │                                  ├─{SignalSender}]
        │                                  ├─{VC manager}]
        │                                  └─2*[{v8:SweeperThrea}]]
        ├─SpotifyHelper─┬─{Chrome_ChildIOT}
        │               ├─{SGI_video_sync}
        │               └─{Watchdog}
        ├─spotify
        ├─{AsyncKeyValueSt}
        ├─{AudioThread}
        ├─2*[{BrowserBlocking}]
        ├─{Chrome_CacheThr}
        ├─{Chrome_DBThread}
        ├─{Chrome_DevTools}
        ├─{Chrome_FileThre}
        ├─{Chrome_FileUser}
        ├─{Chrome_IOThread}
        ├─{Chrome_ProcessL}
        ├─{Chrome_WebKitTh}
        ├─{Dns Thread}
        ├─{MediaStreamDevi}
        ├─{NSS SSL ThreadW}
        ├─{Network Thread}
        ├─{NetworkChangeNo}
        ├─{Proxy resolver}
        ├─{QThread}
        ├─{WorkerPool/1886}
        ├─2*[{dispatcher}]
        ├─{inotify_reader}
        ├─{spotify}
        ├─2*[{threaded-ml}]
        └─2*[{}]
```

Layer 5 - Session



A Useful Aside

The kernel handles lots of the gory details



Layer 4 - Transport

- We've already seen that Spotify is busy crafting lots of HTTP requests that get stuffed into TCP datagrams
- At layer 4, Spotify –or its helper threads or processes– effectively hands off the TCP datagrams to the kernel, and the kernel passes received TCP datagrams back to Spotify
- Kernel tcp modules handle
  - network scheduling
  - Multiplexing
  - Data ordering
- Kernel passes off the TCP datagrams to layer three to be packetized with IP address and routed

Layer 3 - Network

- BUT WHERE DID IT GET THE IP ADDRESSES?
  - TCP requests are bound to unix sockets (remember the netstat output?) ; sockets contain the source and destination IP address info.
  - Or Magic.
- Kernel constructs IP packet with source/destination IP info and TCP (or UDP) datagram along with routing info
- Passes off packets to Link Layer

Layer 2 - Link

- Kernel takes packet and makes full ethernet frame. Ships off to NIC firmware via the NICs drivers
- In this case, the “NIC” is the 3G/4G modem
- Pray that the drivers and firmware are solid

Pray to Firmware Gods

Layer 1 - Physical

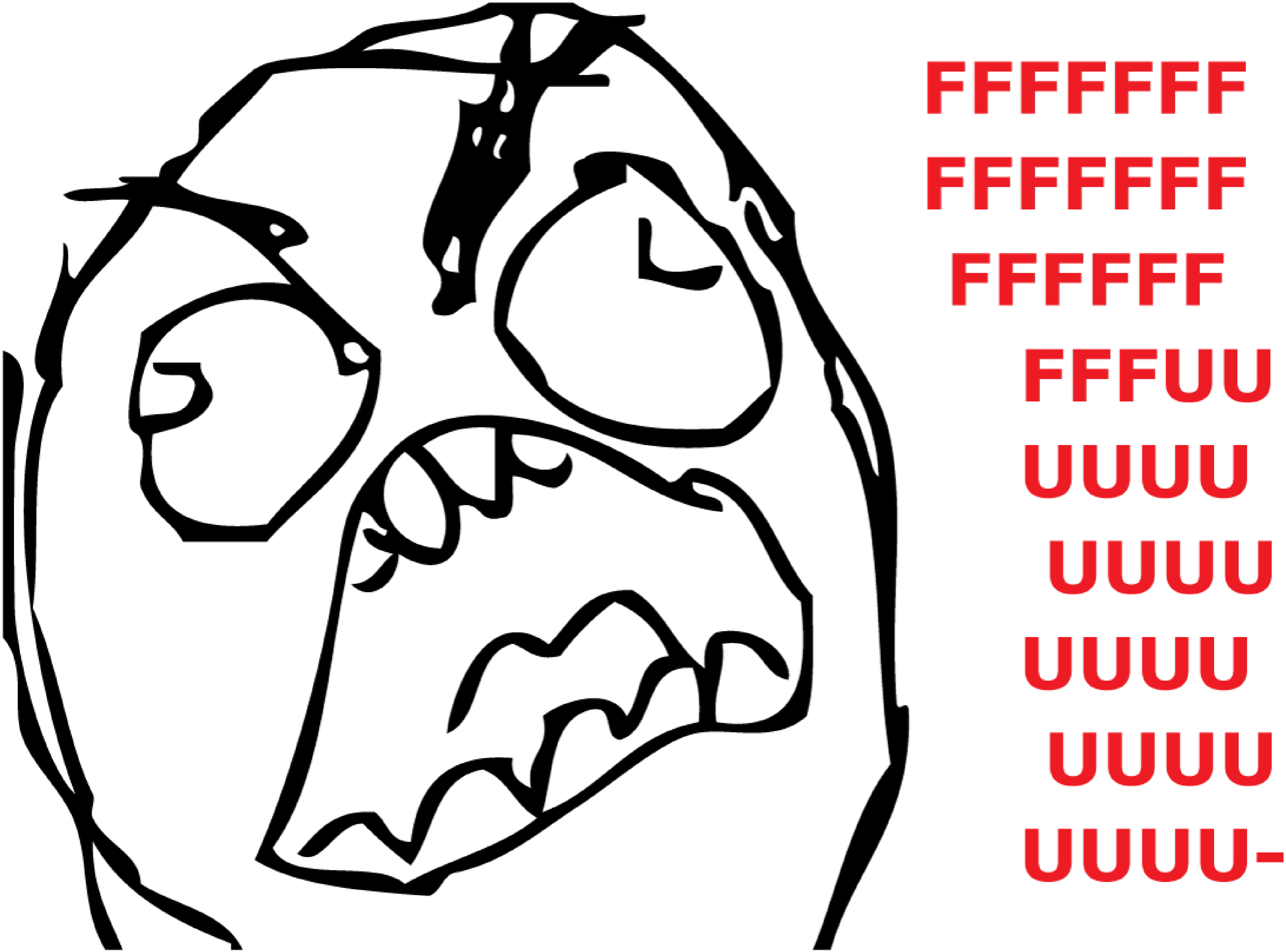
- And this is the part where the NIC's firmware actually takes the ethernet frame and does its thing, translating the frames into meaningful EM signals the tower can read
- And translating received signals back to ethernet frames and sending it all back up the stack

OPERATOR: WE GET SIGNAL.

CAPTAIN: WHAT !

OH OH I ALMOST FORGOT

I'm gonna need you to come in on Sunday too


FFFFFFF
FFFFFFF
FFFFFF
FFFUU
UUUU
UUUU
UUUU
UUUU
UUUU-

And that's how your iPhone works.