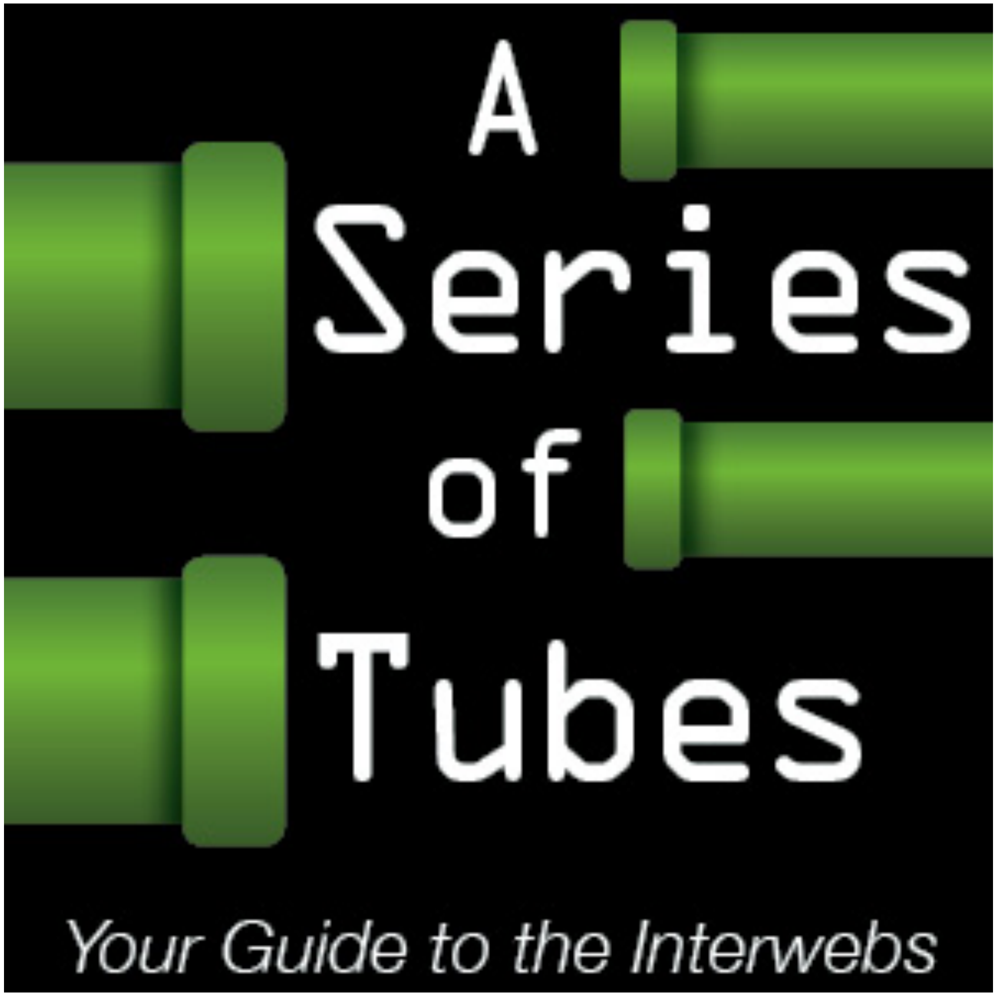
Web Basics

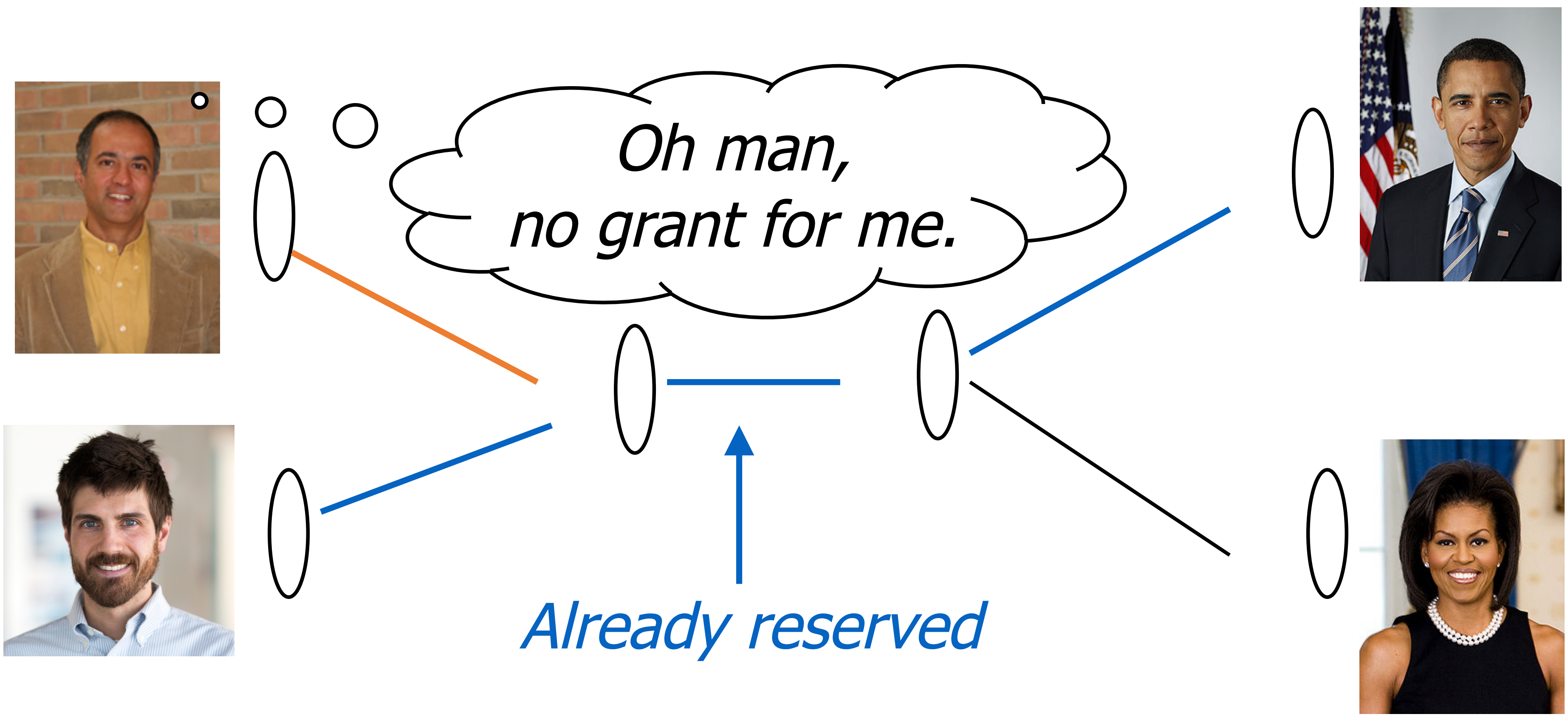


Web Essentials

- A brief history
- Transfer
  - Networking and OS basics
  - HTTP
- Content
  - HTML
  - Encoding
  - Dynamic pages

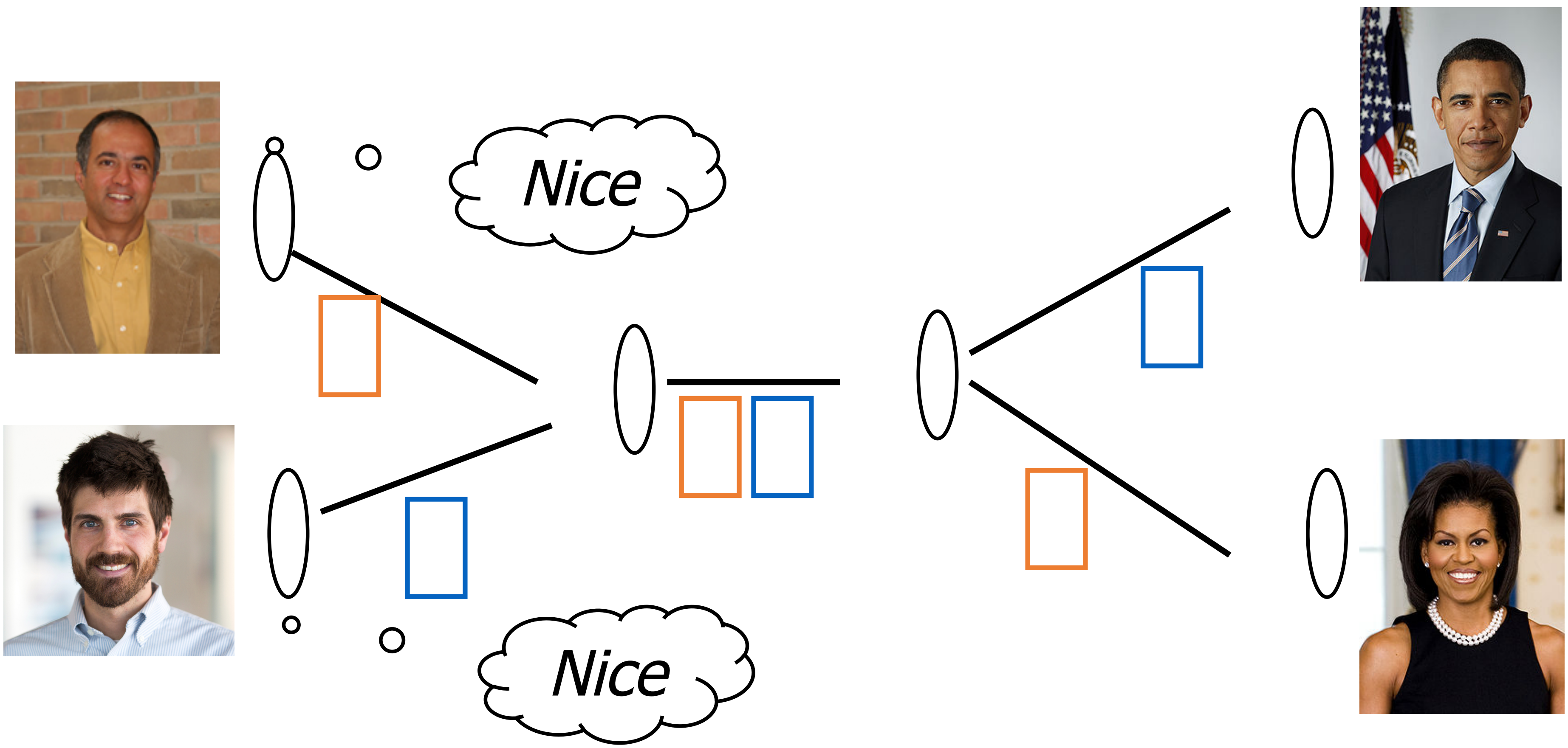
Networking Basics

- How to achieve reliable machine-to-machine data transport?
  - Circuit-switched
  - Packet-switched



Networking Basics

- How to achieve reliable machine-to-machine data transport?
  - Circuit-switched
  - Packet-switched



Discussion Questions

- What are advantages of circuit-switched?

- What are possible advantages of packet-switched approach?

Discussion Questions

- What are advantages of circuit-switched?
  - Good for real-time things like voice and teleconferences.
- What are possible advantages of packet-switched approach?
  - Makes better use of resources
  - Can survive destruction of a node in the network

Network Protocol Stack Model

Application	User interaction	HTTP, FTP, SMTP
Presentation	Data representation	XML, cryptography
Session	Dialogue mangement	???
Transport	Reliable end-to-end link	TCP
Network	Routing via multiple nodes	IP
Data Link	Physical addressing	Ethernet
Physical	Metal or RF representation	802.11, Bluetooth

Open System Interconnection (OSI) Reference Model

Network Protocol Stack Model

Web

Application	User interaction	HTTP, FTP, SMTP
Presentation	Data representation	XML, cryptography
Session	Dialogue mangement	???
Transport	Reliable end-to-end link	TCP
Network	Routing via multiple nodes	IP
Data Link	Physical addressing	Ethernet
Physical	Metal or RF representation	802.11, Bluetooth

Internet

- IP is best-effort. Packets may get dropped or delayed.
- TCP is reliable. Guarantees data will get there in-order.

Network Protocol Stack Model

- A full computer needs to implement "the entire stack"
- What about routers and switches? What do they speak?

- What about your WiFi router?

What is the Web, exactly?

- Network Transfer
  - Hypertext Transfer Protocol (HTTP)
- Content Encoding
  - Hypertext Markup Language (HTML)
  - Look at some with the "view source" command on your browser
- At the command line:
  - `wget google.com`
  - then see downloaded `index.html`

  - `curl google.com`
  - then see stdout

HTTP

- Hypertext Transfer Protocol
- Request/response protocol
  - Client (your browser) opens TCP connection to server and writes a request
  - Server responds appropriately
  - Connection is closed
  - That's it
- HTTP is dead simple
  - Server can't open connection to client
  - Completely stateless
    - Each request is treated as brand new
    - No state => no history

HTTP

- Stateless means that the server and client don’t need (and cannot) remember anything between the request/response.
- TCP is not stateless; the endpoints must remember which IP packets have been transmitted, so that they can be reconstructed and put back in the correct order.

HTTP Structure

- Each request/response has a header plus (optional) content.
- Protocol specifies header.
- Client requests have one of several possible forms:
  - GET, POST, PUT, DELETE, HEAD, TRACE, CONNECT, OPTIONS
  - Each one has associated parameters. E.g., `GET /foo.html HTTP/1.1`
  - Plus content for POST, …
- Server responds with error code
  - ("200 OK" or "404 Not Found") + content

Header Details

- GET is the most common request
  - Limited, for efficiency. 1024 bytes, characters in ISO-8859-1
- POST writes data
  - Client can write an additional payload of arbitrary data for the server
  - Used in HTML form postings
- HEAD is similar to GET, but expects no content in the response, only the
  - Good for things like last-modified timestamps

HTTP Example

- Try this:
  - `telnet google.com 80`
  - `GET /index.html HTTP/1.1`
  - `<blank line>`
- You should see the HTML for the front page of Google

Implementing HTTP

- At the heart of every browser is code that fires off lots of HTTP requests
  - Even a single page can consist of dozens
  - Desktop browsers are hugely complicated, but that is because they have tons of add-on features
  - The core functionality in a minimal browser is little more than the telnet/get example above, and an HTML renderer.

Another way to watch HTTP

- Here's another way to watch HTTP in action from the command line
  `curl --verbose https://www.google.com/ -o index.html`
- More detail
  `curl --trace-ascii log.txt https://www.google.com/`
- See the timing
  `curl --trace-ascii log.txt --trace-time https://www.google.com/`
- `curl` is great for scraping!

HTTP Client Algorithm

- Wait for user to type into browser bar
  http:///www.google.com/index.html
- Break the URL into hostname and path
- Contact host at port 80, send
  `GET <path> HTTP/1.1`
- Download result code and bytes
- Send content bytes to HTML renderer for drawing onscreen

Implementing HTTP

- Servers are architecturally unusual
  - Simply wait around for requests to arrive
  - What is the best way to design an HTTP server?

HTTP Server Design

- Approach #1
  - wait till an HTTP request arrives
  - then start server
  - serve request
  - and kill server
- Approach #2
  - Sit in a loop, waiting for requests
- Approach #3
  - Large set of processes hanging around
- Approach #4
  - Processes with threadpools

Process/Thread Background

- A process is a unit of parallel execution in an operating system.
- OS manages resources and distributes to processes, including memory space.
- Processes are “heavy-weight”: Need much book-keeping and strong walls.
- Threads are lighter structures to achieve parallelism with less protection.

Process/Thread Background

- See all running processes
  `ps -ax`
- See all processes and their threads
  `ps –axM`
- Monitor processes in real time
  `top`
- Monitor threads in real time (may not work on Darwin AKA OS X)
  `top -H`

HTTP Server Design

- Approach #1
  - wait till an HTTP request arrives
  - then start server
  - serve request
  - and kill server

- High startup and shut down time

HTTP Server Design

- Approach #2
  - Sit in a loop, waiting for requests

- Works fine ...
- ... if you never get more than one request at a time

HTTP Server Design

- Approach #3
  - Large set of processes hanging around

- Works
- High memory overhead

HTTP Server Design

- Approach #4
  - Processes with threadpools

- Best approach
- Advantages of #3
- But you lose some memory protection

- Threads or processes for parallelism
- Balance protection, startup costs, memory footprint, responsiveness

HTTP Server Algorithm

- HTTP server process (or thread) waits for connection from client
- Receives a `GET /index.html` request
- Looks in content directory, computes name
  `/content/index.html`
- Loads file from disk
- Write response to client:
  200 OK, followed by bytes for `/content/index.html`

URL Encoding

`protocol://server:port/path#fragment?search`

- URLs have several parts
  - Protocol
  - Server
  - Port
  - Path
  - Fragment
  - Search

URL Encoding

`protocol``://server:port/path#fragment?search`

- `protocol` tells the server what protocol to use. In other words, what "language" to speak
  - Examples: `http, https`

URL Encoding

`protocol://`**`server:port`**`/path#fragment?search`

- `server` helps locate the machine we want to talk to
  - Example: `www.umich.edu` → `135.22.87.1`
  - DNS lookup translates server name into an IP address
  - Try this:
    $ `host` [`www.umich.edu`](www.umich.edu)
- `port` is used to identify a specific service
  - Ports allow multiplexing (one server runs multiple services)
  - Example: 80 is typically HTTP

URL Encoding

`protocol://server:port`**`/path`**`#fragment?search`

- `path` is a file name relative to the server root
- If directory name, default action (depending on server configuration), e.g.:
  - Find `index.htm` or `index.html`
  - Show directory listing

URL Encoding

`protocol://server:port/path`**`#fragment`**`?search`

- `fragment` is identified at the client, ignored by server
- Example: navigate directly to the section labeled "Linking"
  `http://en.wikipedia.org/wiki/World_Wide_Web`**`#Linking`**

URL Encoding

`protocol://server:port/path#fragment`**`?search`**

- `search` string is a general-purpose (set of) parameter(s) that the server (or specified resource on server) can use as it pleases
- Example from project 1:
- `http://localhost:3000/albums`**`?username=spacejunkie`**
- The function controlling this URL will receive the (key, value) pair (username, spacejunkie)

Web Philosophy

- Build the simplest, fastest system that will do the most important things.
  - Build layers on top for additional required functionality.

- Expect diversity.
  - Do something sensible even if given a bad request.

- There is a great deal of unverified information from an unknown party. Know what must be verified and what can be trusted.

Bad Requests

- Traditional programming requires correct specification.
  - Even a small error may keep a program from compiling
- On the web, a browser *expects* to see tags and commands it does not understand.
  - It must not crash.
  - It must try not to complain.
  - Rather it must do the best it can, usually by ignoring what it does not understand.

Diversity

- In character encoding
- In scripting languages
- In versions of languages and protocols
- In screen formats for display
- …

- The world has more variety out there than you can imagine.
- You must embrace and manage this.

Unverified Information

- Sender field in email
- Referer field in HTTP
  - Referer: http://www.google.com/search?q=web+field
- Meta tag in HTML
- Ack from TCP receiver
- HTML Character-set encoding
- …