CS5283 – Week 4

More Socket Programming and Abstractions for Network Programming

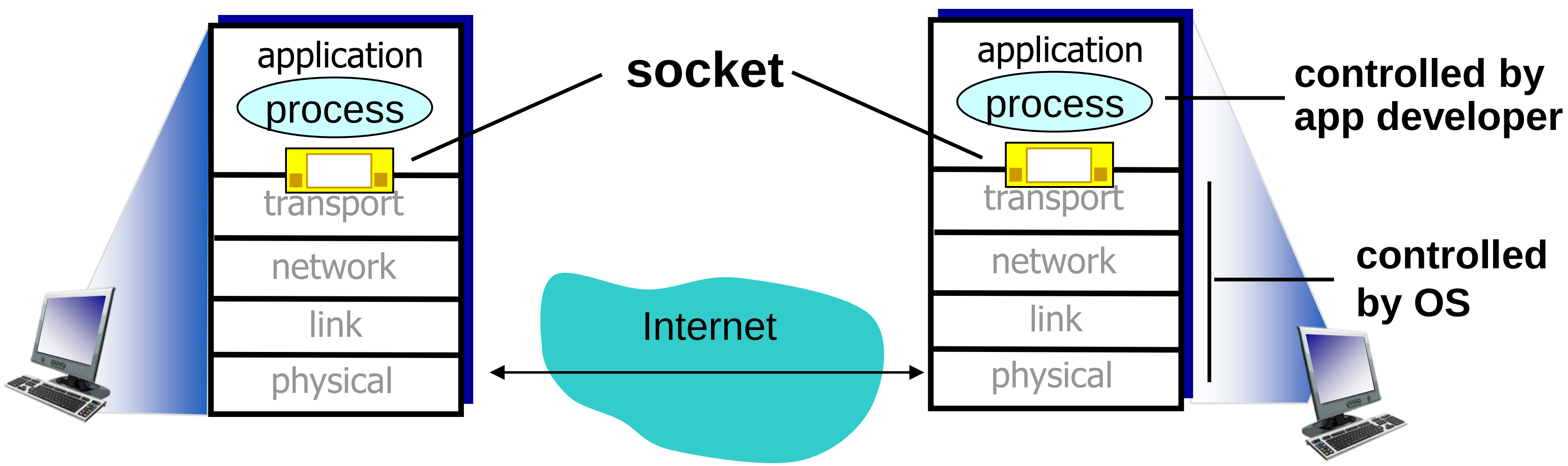
Outline

- Quiz 1 available, due 9/23 at class start
- Homework 2 available, due 9/30 at class start
- Q/A asynchronous content
  - Socket programming
    - Addresses/Ports
  - Abstractions for Network Programming
    - Extended state machines
- Breakout activity with interactive book content: https://gaia.cs.umass.edu/kurose_ross/interactive/
- Breakout activity with WireShark lab: http://gaia.cs.umass.edu/kurose_ross/wireshark.htm

More Socket Programming

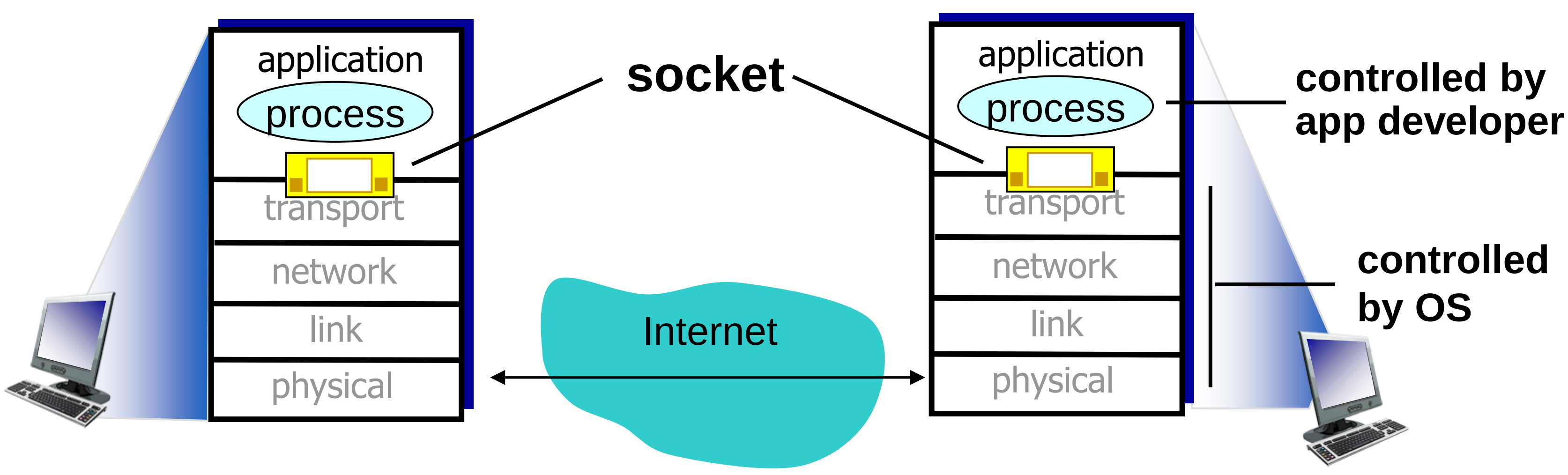
Sockets

- The process sends/receives messages to/from its **socket.**
- The socket is analogous to a door.
  - The sending process shoves the message out the door.
  - The sending process relies on the transport infrastructure on the other side of the door to deliver the message to the socket at the receiving process.



Socket Programming

- **Goal**: learn how to build client/server applications that communicate using sockets
- **Socket**: the door between the application process and end-end-transport protocol



Socket Programming

- ***Two socket types for two transport services:***
  1. **UDP:** unreliable datagram
  2. **TCP:** reliable, byte stream-oriented

Processes Communicating

Process: the program running within a host

- Within the same host, two processes communicate using **inter-process communication** (defined by the OS)
- Processes in different hosts communicate by exchanging **messages**

Clients, servers

- **Client process:** the process that initiates communication
- **Server process:** the process that waits to be contacted

- Aside: applications with P2P architectures have client processes and server processes

Addressing Processes

- To receive messages, the process must have an **identifier**
- The host device has a unique 32-bit IP address
- ***Q:*** Does the IP address of the host on which the process runs suffice for identifying the process?
  - ***A:*** No, *many* processes can be running on the same host
- The **identifier** includes both the **IP address** and **port numbers** associated with the process on the host
- Example port numbers:
  - HTTP server: 80
  - Mail server: 25
- To send an HTTP message to the gaia.cs.umass.edu web server:
  - **IP address:** 128.119.245.12
  - **Port number:** 80
- More shortly…

More Socket Programming

The End

Protocols

What Is a Protocol?

- A protocol is an **agreement on how to communicate**
- It includes:
  - **Syntax**: how a communication is specified and structured
    - Format of messages sent and received
    - Order of messages
  - **Semantics**: what a communication means
    - What actions are taken when transmitting, receiving, or when a timer expires

Organization of Air Travel

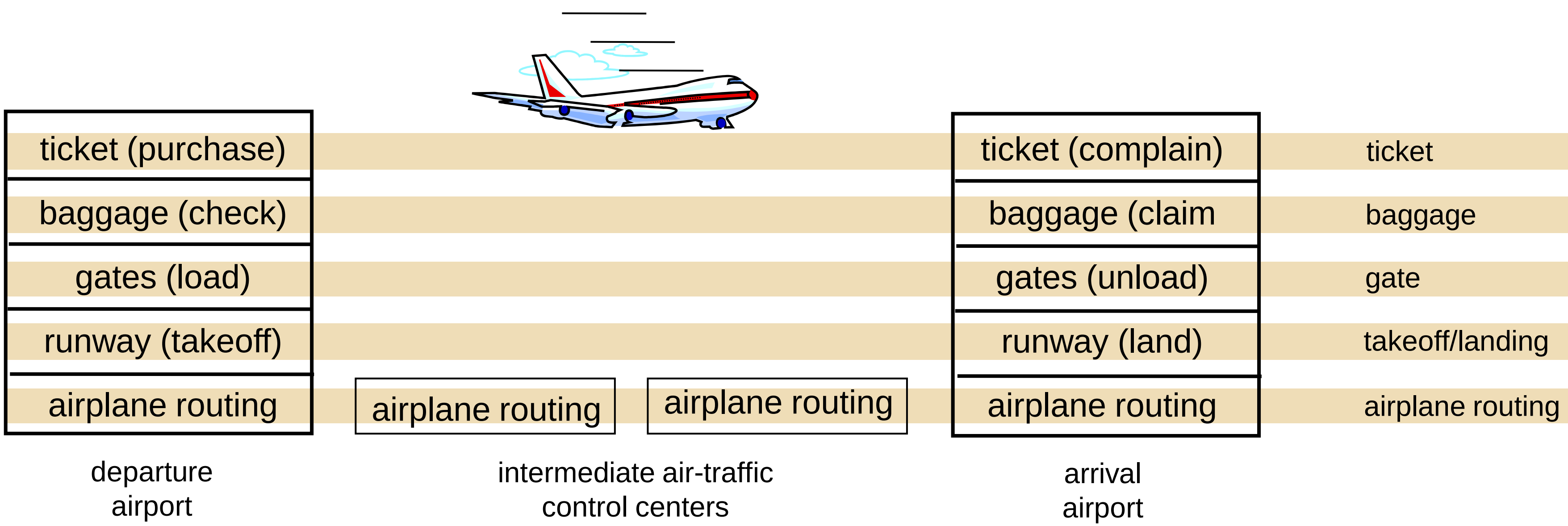
- A series of steps

ticket (purchase)	**ticket (complain)**
baggage (check)	**baggage (claim)**
gates (load)	**gates (unload)**
runway takeoff	**runway landing**
airplane routing	**airplane routing**

airplane routing

Layering of Airline Functionality

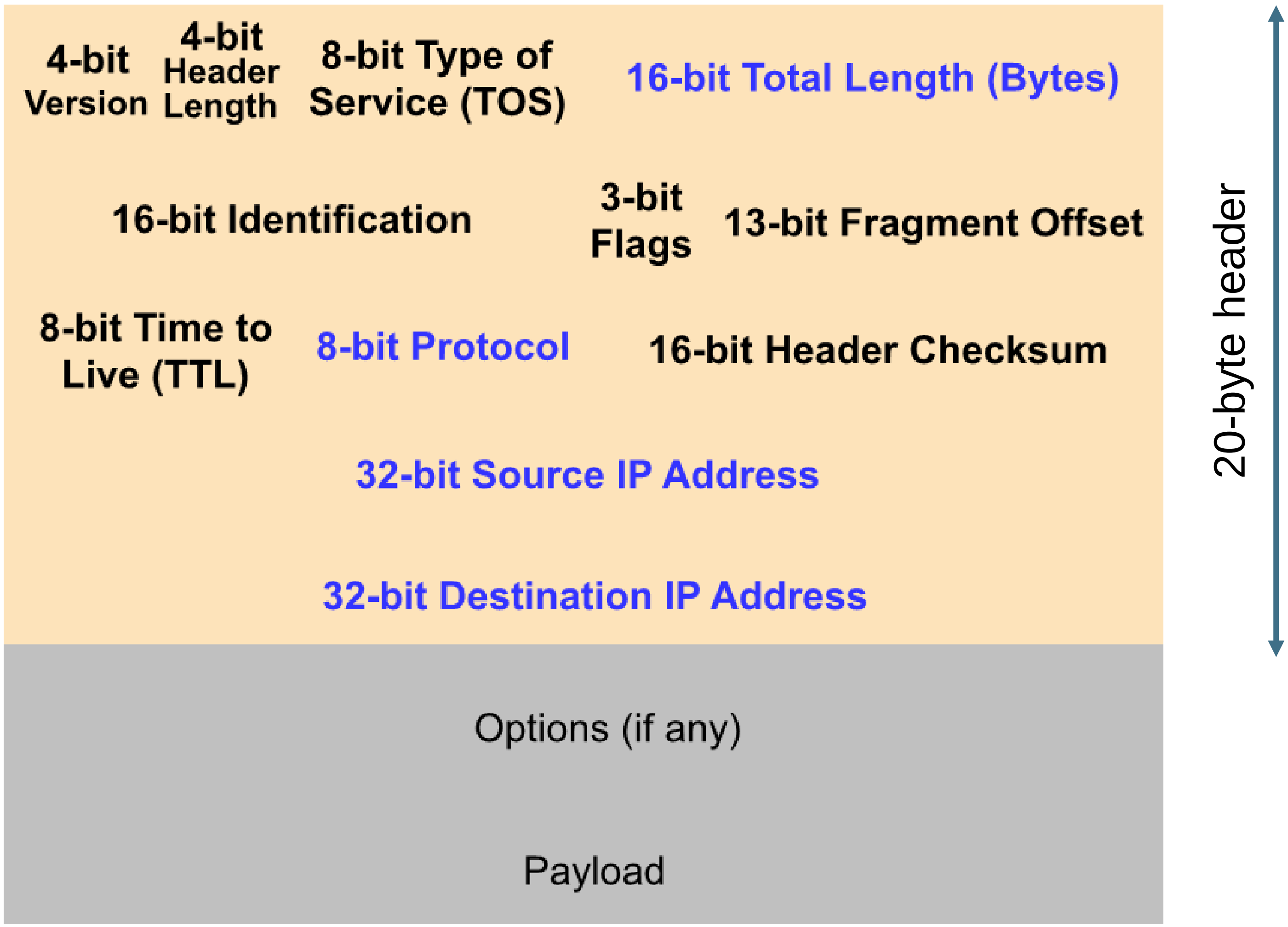
- **Layers:** each layer implements a service
  - Via its own internal-layer actions
  - Relying on the services provided by the layer below



Example: The Internet Protocol (IP)

- Problem
  - Many different network technologies
    - Examples: Ethernet, Wi-Fi, fiber, satellite, etc.
  - How can you hook them together?
    - n x n translations
- IP was designed to glue them together
  - n translations
  - Minimal requirements (datagram)
- The Internet is founded on IP
  - “IP over everything”

Example: IP Packet



IP: “Best-Effort” Packet Delivery

- Datagram packet switching
  - Sends data in packets
  - Header with source and destination addresses
- The service it provides
  - Packets may be **lost**
  - Packets may be **corrupted**
  - Packets may be **delivered out of order**
  - The same packet may be **received more than once**

Example: Transmission Control Protocol (TCP)

- Communication service
  - Ordered, reliable byte stream
  - Simultaneous transmission in both directions
- Key mechanisms at end **hosts**
  - Retransmission of lost and corrupted packets
  - Discard duplicates
  - Put packets in order
  - Flow control to avoid overloading the receiver buffer
  - Congestion control to adapt sending rate to network load

Protocol Standardization

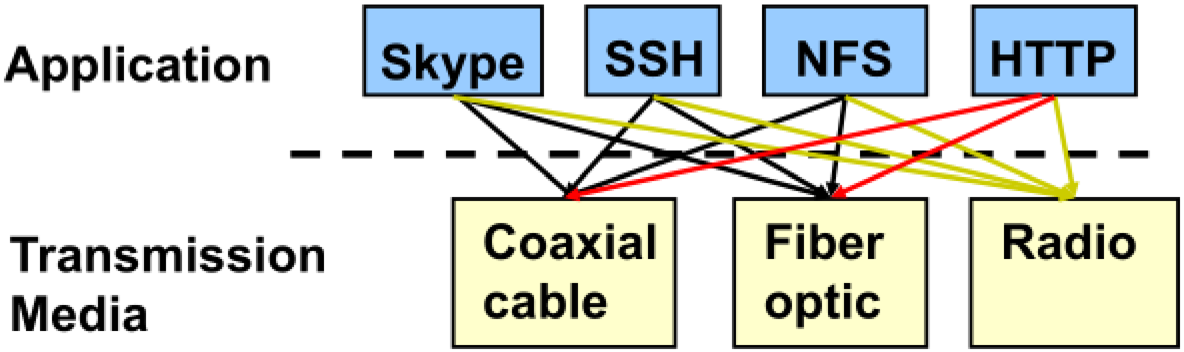
- Ensure communicating parties speak the **same language**
  - Standardization to **enable multiple implementations**
  - Or, the same folks have to write all the software
- Standardization: Internet Engineering Task Force
  - Based on working groups that focus on specific issues
  - Produces “Request for Comments” (RFC) documents
    - Promoted to standards via rough consensus and running code
  - The IETF website
  - RFCs are archived here
- De facto standards: the same folks writing the code
  - P2P file sharing, Skype, <your protocol here>

The Problem

- Many different applications
  - Email, web, video streaming, etc.
- Many different network styles and technologies
  - Circuit-switched vs. packet-switched, etc.
  - Wireless vs. wired vs. optical, etc.
- **How do we organize this?**

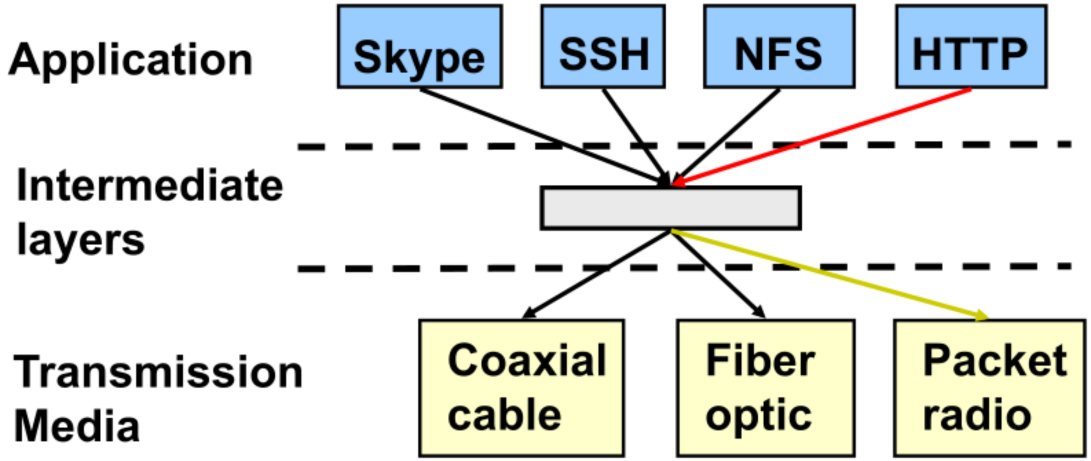
The Problem (cont.)

- Re-implement every application for every technology?
- No! But how does the Internet design avoid this?

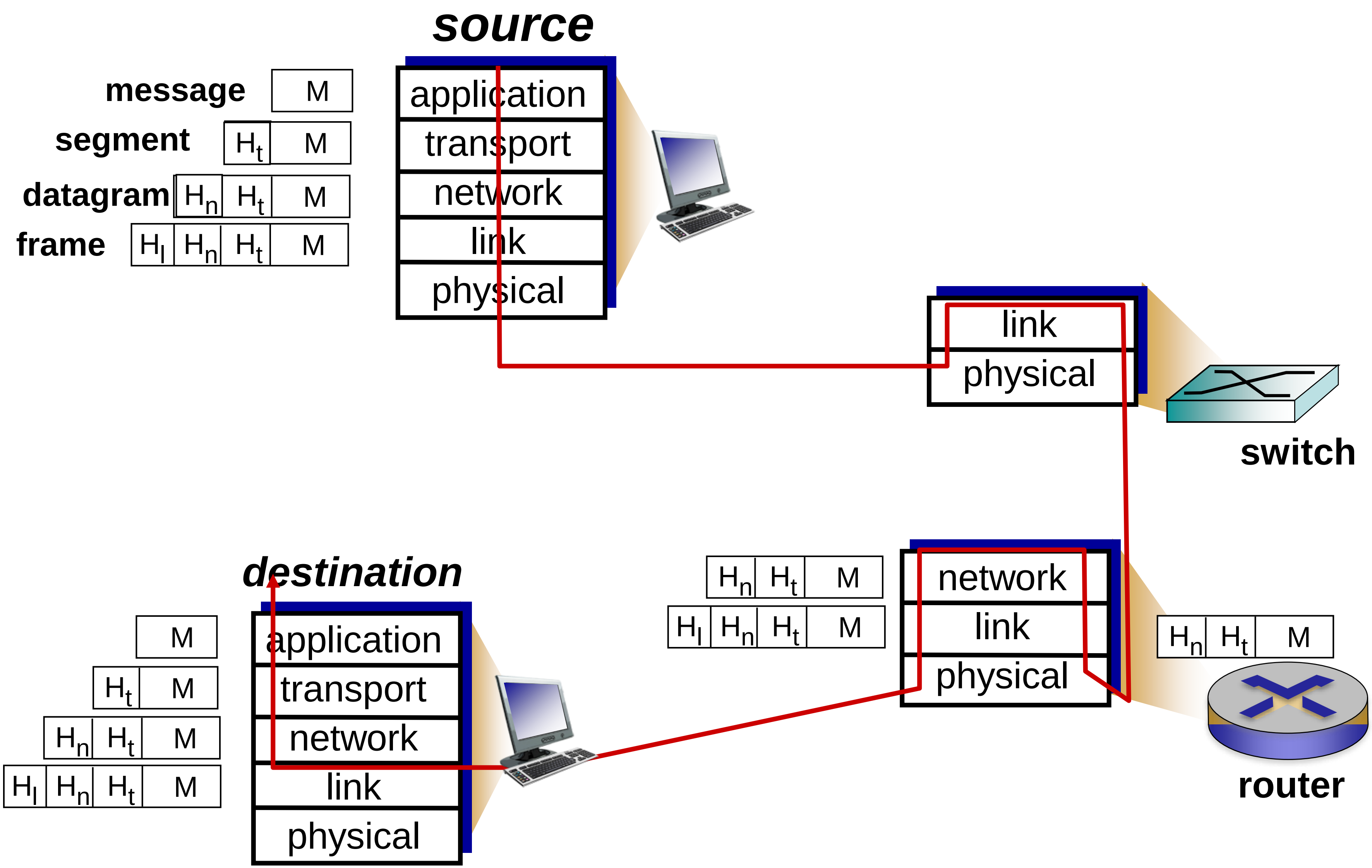


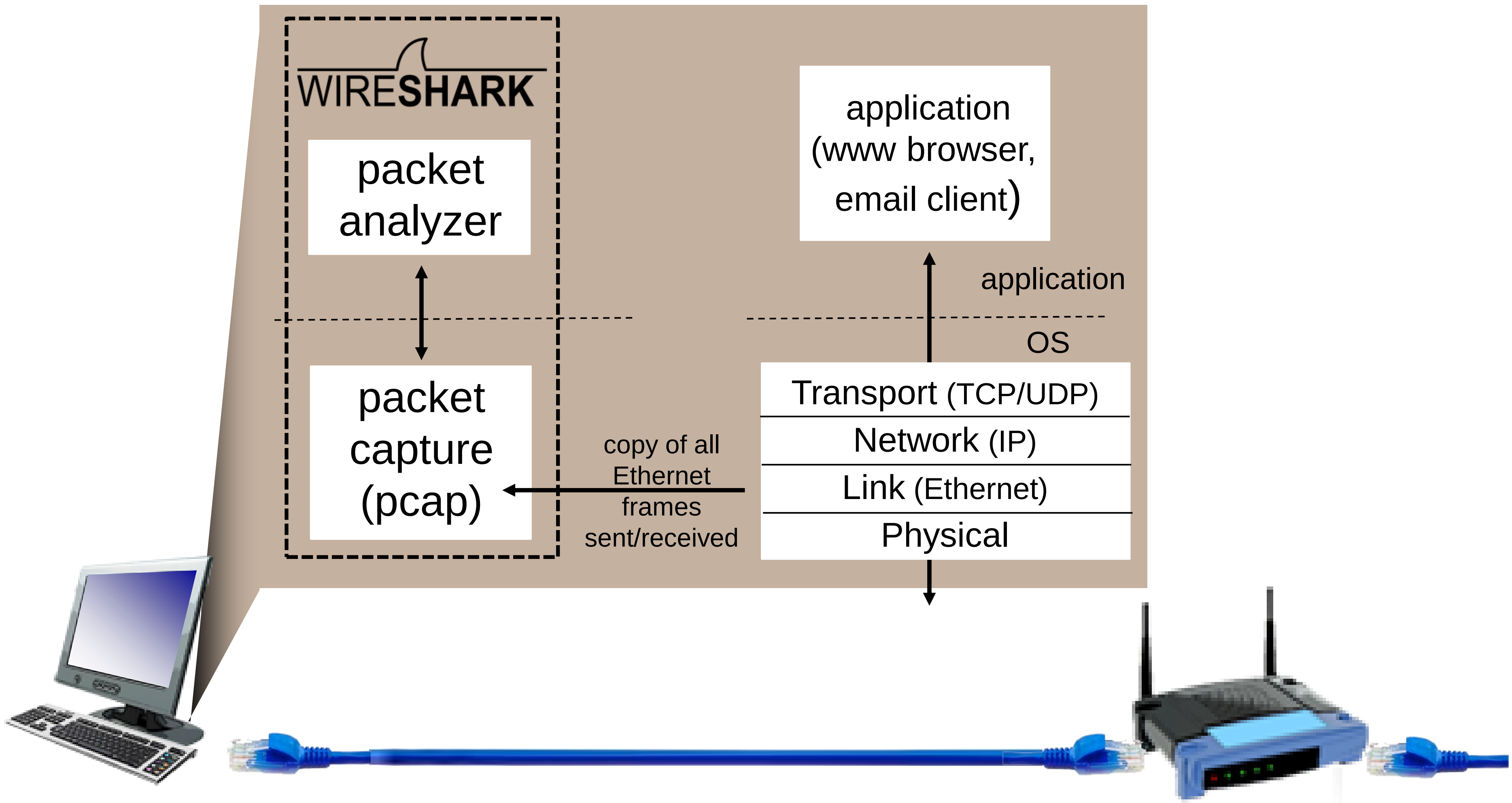
Solution: Intermediate Layers

- Introduce **intermediate layers** that provide a set of abstractions for various network functionality and technologies
  - A new app/media implemented only once
  - A variation on “add another level of indirection”



Encapsulation




WIRESHARK
packet analyzer
packet capture (pcap)
copy of all Ethernet frames sent/received
application (www browser, email client)
application
OS
Transport (TCP/UDP)
Network (IP)
Link (Ethernet)
Physical

Protocols

The End

HTTP

HTTP Request Message

- Two types of HTTP messages: **request, response**
- **HTTP request message:**
  - ASCII (human-readable format)

carriage return character

line-feed character

request line (GET, POST, HEAD commands)

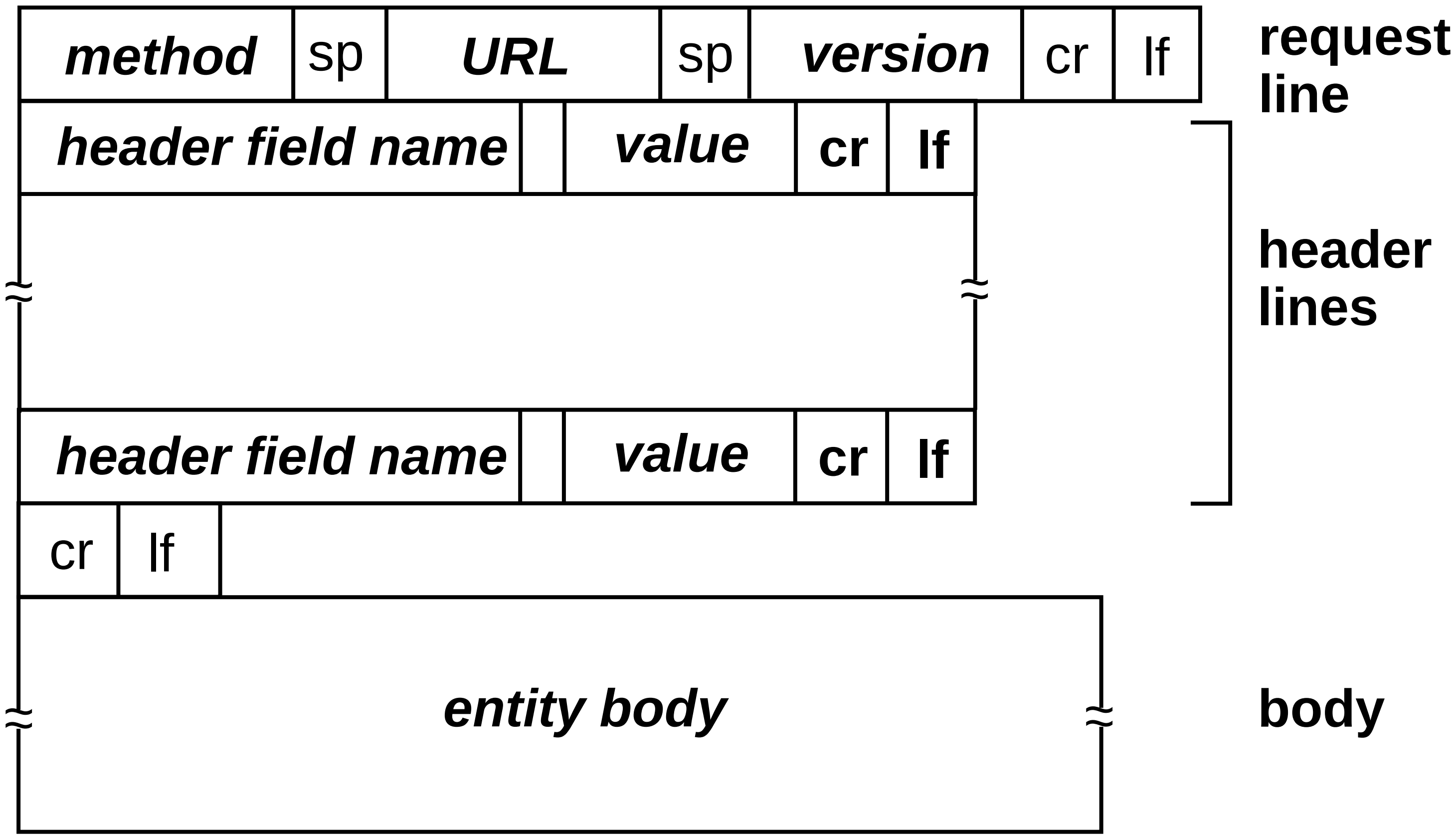
header lines

carriage return, line feed at start of line indicates end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP Request Message: General Format



HTTP Response Message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP Response Status Codes

- The status code appears in the first line in a server-to-client response message
- Some sample codes:

`200 OK`

- Request succeeded, requested object later in this message

`301 Moved Permanently`

- Requested object moved, new location specified later in this message (Location:)

`400 Bad Request`

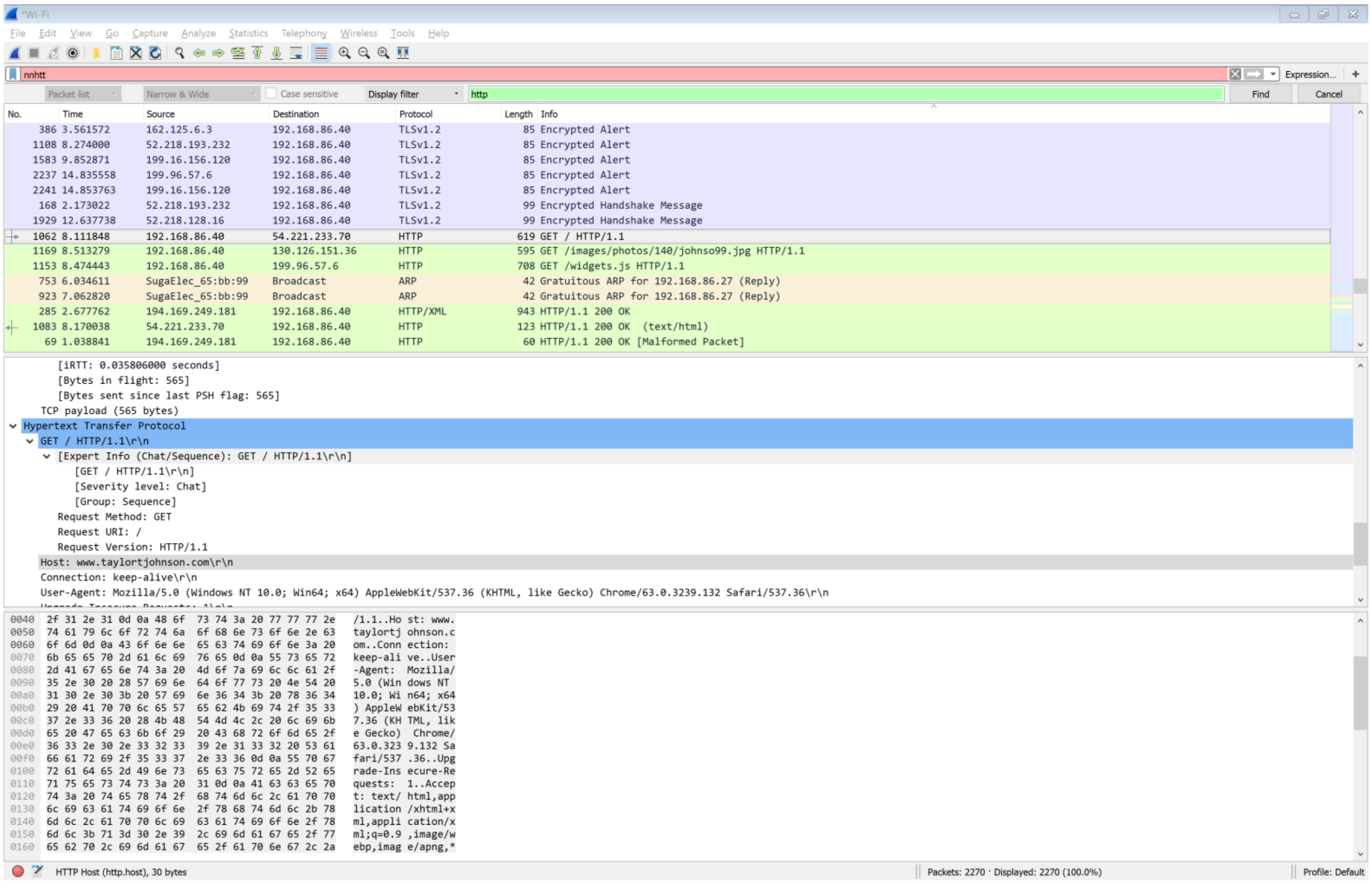
- Request message not understood by server

`404 Not Found`

- Requested document not found on this server

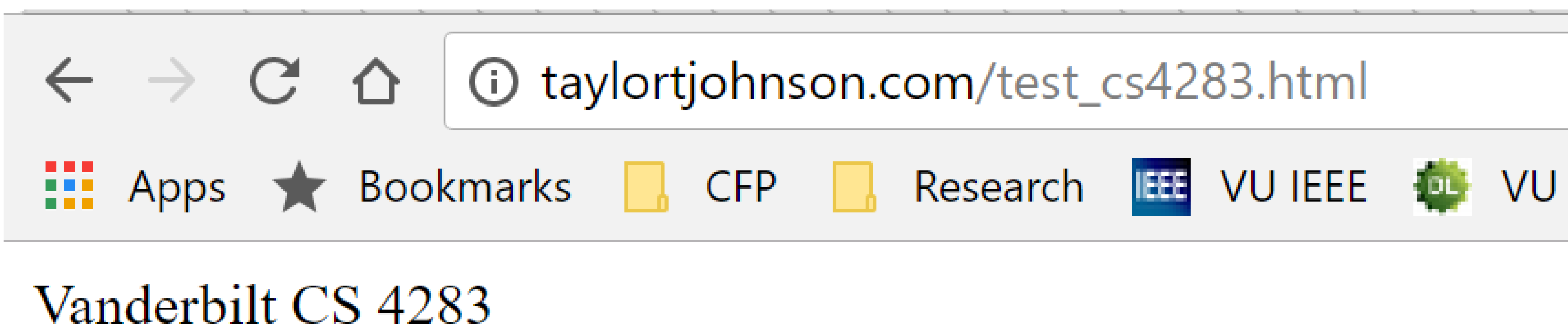
`505 HTTP Version Not Supported`

Protocol Example: HTTP



Protocol Example: HTTP

- HTTP protocol overview
  - GET request with hostname and filename
  - RESPONSE



Protocol Example: HTTP

- http://www.taylortjohnson.com/test_cs4283.html

```
    Request Method: GET
    Request URI: /test_cs4283.html
    Request Version: HTTP/1.1
Host: taylortjohnson.com\r\n
Connection: keep-alive\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.3
Upgrade-Insecure-Requests: 1\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/web
DNT: 1\r\n

0030  01 05 aa 96 00 00 47 45  54 20 2f 74 65 73 74 5f   ......GE T /test_
0040  63 73 34 32 38 33 2e 68  74 6d 6c 20 48 54 54 50   cs4283.h tml HTTP
```

GET in ASCII hex:

47 45 54

/test_cs4283.html in ASCII hex:

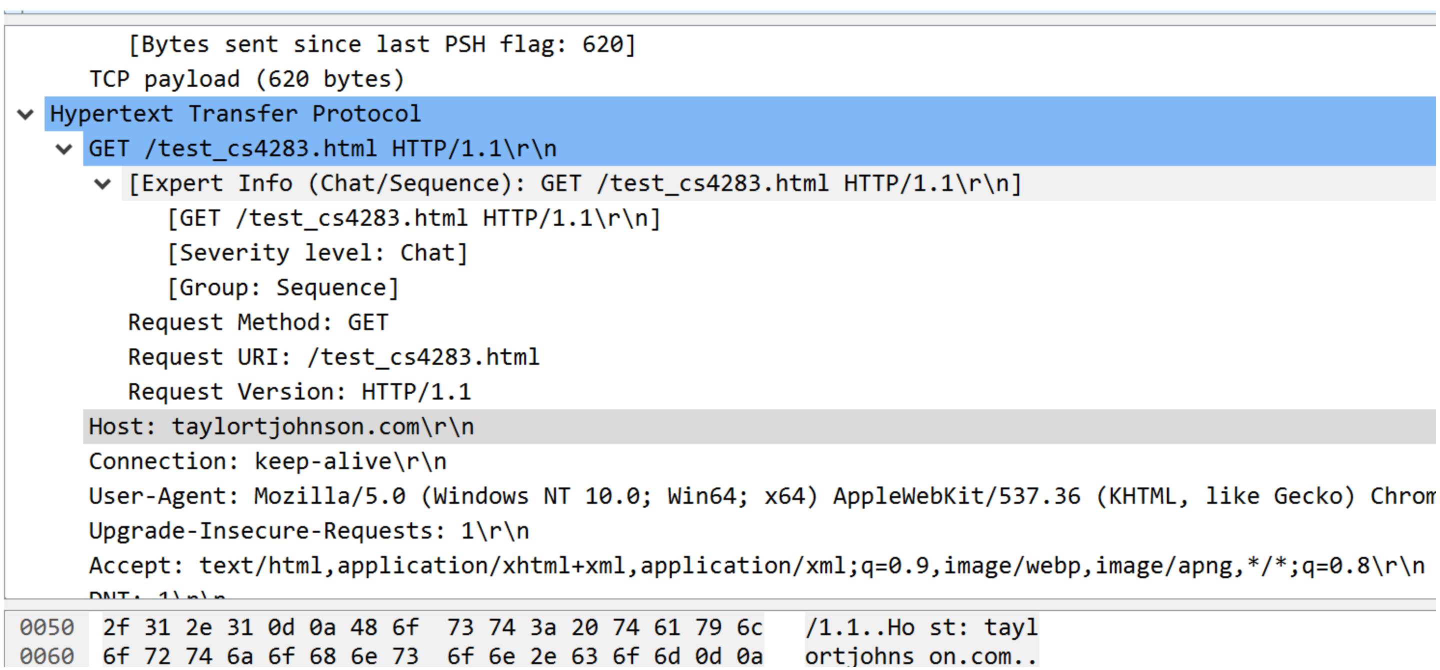
2f 74 65 73 74 5f 63 73 34 32 38 33 2e 68 74 6d 6c

```
    Request Method: GET
    Request URI: /test_cs4283.html
    Request Version: HTTP/1.1
Host: taylortjohnson.com\r\n
Connection: keep-alive\r\n
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.3
Upgrade-Insecure-Requests: 1\r\n
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/web
DNT: 1\r\n

0030  01 05 aa 96 00 00 47 45  54 20 2f 74 65 73 74 5f   ......GE T /test_
0040  63 73 34 32 38 33 2e 68  74 6d 6c 20 48 54 54 50   cs4283.h tml HTTP
0050  2f 31 2e 31 0d 0a 48 6f  73 74 3a 20 74 61 79 6c   /1.1..Ho st: tayl
0060  6f 72 74 6a 6f 68 6e 73  6f 6e 2e 63 6f 6d 0d 0a   ortjohns on.com..
0070  43 6f 6e 6e 65 63 74 69  6f 6e 3a 20 6b 65 65 70   Connecti on: keep
```

Protocol Example: HTTP

- http://www.taylortjohnson.com/test_cs4283.html

```
HOST taylortjohnson.com in ASCII hex:
74 61 79 6c 6f 72 74 6a 6f 68 6e 73 6f 6e 2e 63 6f 6d 0d 0a
```



Protocol Example: HTTP

- http://www.taylortjohnson.com/test_cs4283.html

RESPONSE

```
Line-based text data: text/html
    Vanderbilt CS 4283\n

0030  01 dc 9b fb 00 00 48 54  54 50 2f 31 2e 31 20 32   ......HT TP/1.1 2
0040  30 30 20 4f 4b 0d 0a 44  61 74 65 3a 20 54 75 65   00 OK..D ate: Tue
0050  2c 20 31 36 20 4a 61 6e  20 32 30 31 38 20 31 34   , 16 Jan  2018 14
0060  3a 31 31 3a 33 34 20 47  4d 54 0d 0a 53 65 72 76   :11:34 G MT..Serv
0070  65 72 3a 20 41 70 61 63  68 65 2f 32 2e 32 2e 33   er: Apac he/2.2.3
0080  34 20 28 41 6d 61 7a 6f  6e 29 0d 0a 4c 61 73 74   4 (Amazo n)..Last
0090  2d 4d 6f 64 69 66 69 65  64 3a 20 54 75 65 2c 20   -Modifie d: Tue,
00a0  31 36 20 4a 61 6e 20 32  30 31 38 20 31 34 3a 30   16 Jan 2 018 14:0
00b0  37 3a 33 35 20 47 4d 54  0d 0a 45 54 61 67 3a 20   7:35 GMT ..ETag:
00c0  22 34 30 31 39 64 2d 31  33 2d 35 36 32 65 35 34   "4019d-1 3-562e54
00d0  30 61 36 65 34 30 64 22  0d 0a 41 63 63 65 70 74   0a6e40d" ..Accept
00e0  2d 52 61 6e 67 65 73 3a  20 62 79 74 65 73 0d 0a   -Ranges:  bytes..
00f0  43 6f 6e 74 65 6e 74 2d  4c 65 6e 67 74 68 3a 20   Content- Length:
0100  31 39 0d 0a 43 6f 6e 6e  65 63 74 69 6f 6e 3a 20   19..Conn ection:
0110  63 6c 6f 73 65 0d 0a 43  6f 6e 74 65 6e 74 2d 54   close..C ontent-T
0120  79 70 65 3a 20 74 65 78  74 2f 68 74 6d 6c 3b 20   ype: tex t/html;
0130  63 68 61 72 73 65 74 3d  55 54 46 2d 38 0d 0a 0d   charset= UTF-8...
0140  0a 56 61 6e 64 65 72 62  69 6c 74 20 43 53 20 34   .Vanderb ilt CS 4
0150  32 38 33 0a                                        283.

Text item (text), 19 bytes
```

Method Types

HTTP/1.0:

- GET
- POST
- HEAD
  - Asks the server to leave the requested object out of the response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - Uploads the file in the entity body to the path specified in the URL field
- DELETE
  - Deletes the file specified in the URL field

Uploading Form Input

- **POST method:**
  - The web page often includes form input
  - Input is uploaded to the server in the entity body
- **URL method:**
  - Uses the GET method
  - Input is uploaded in the URL field of the request line:
    www.somesite.com/animalsearch?monkeys&banana

Trying Out
HTTP (Client Side) for Yourself

1. Telnet to your favorite web server:

```
telnet gaia.cs.umass.edu 80
```

Opens the TCP connection to port 80 (default HTTP server port) at gaia.cs.umass. edu.
Anything typed in will be sent to port 80 at gaia.cs.umass.edu.

2. Type in a GET HTTP request:

```
GET /kurose_ross/interactive/index.php HTTP/1.1
Host: gaia.cs.umass.edu
```

By typing this in (hit carriage return twice), you send this minimal (but complete) GET request to the HTTP server.

3. Look at the response message sent by the HTTP server! (Or use Wireshark to look at the captured HTTP request/response)

HTTP

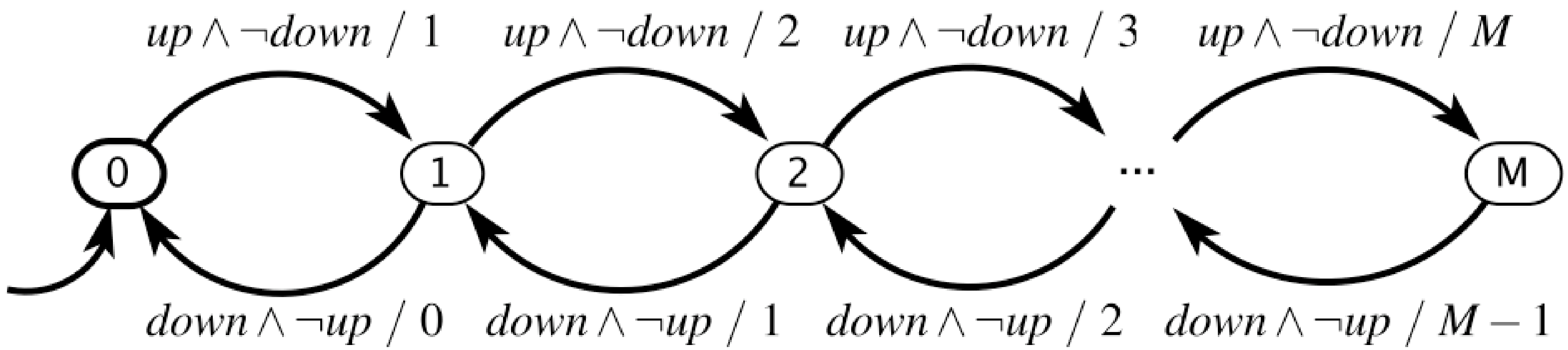
The End

Extended State Machines

Extended State Machines Overview

- Models that are abstractions of **system dynamics** (how states change over time)
- Examples:
  - Modeling physical phenomena: ODEs
  - Feedback control systems: time-domain modeling
  - Modeling modal behavior: FSMs, hybrid automata
  - Modeling sensors and actuators: calibration, noise
  - Modeling software: concurrency, real-time models
  - Modeling networks: latencies, error rates, packet losses

Finite State Machine as a Graph



Formally: $(States, Inputs, Outputs, update, initialState)$, where

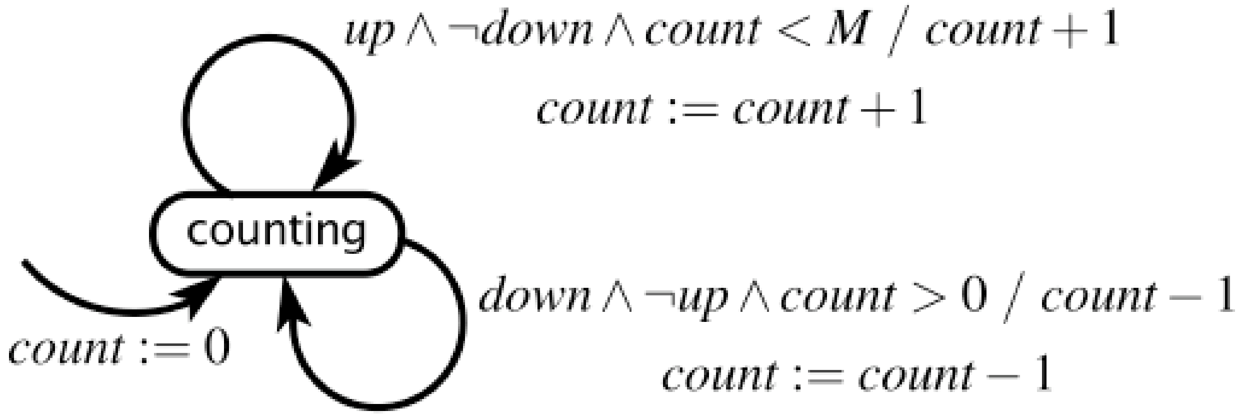
- $States = \{0, 1, \cdots, M\}$
- $Inputs$ is a set of input valuations
- $Outputs$ is a set of output valuations
- $update : States \times Inputs \to States \times Outputs$
- $initialState = 0$

The picture above defines the update function.

Extended State Machines

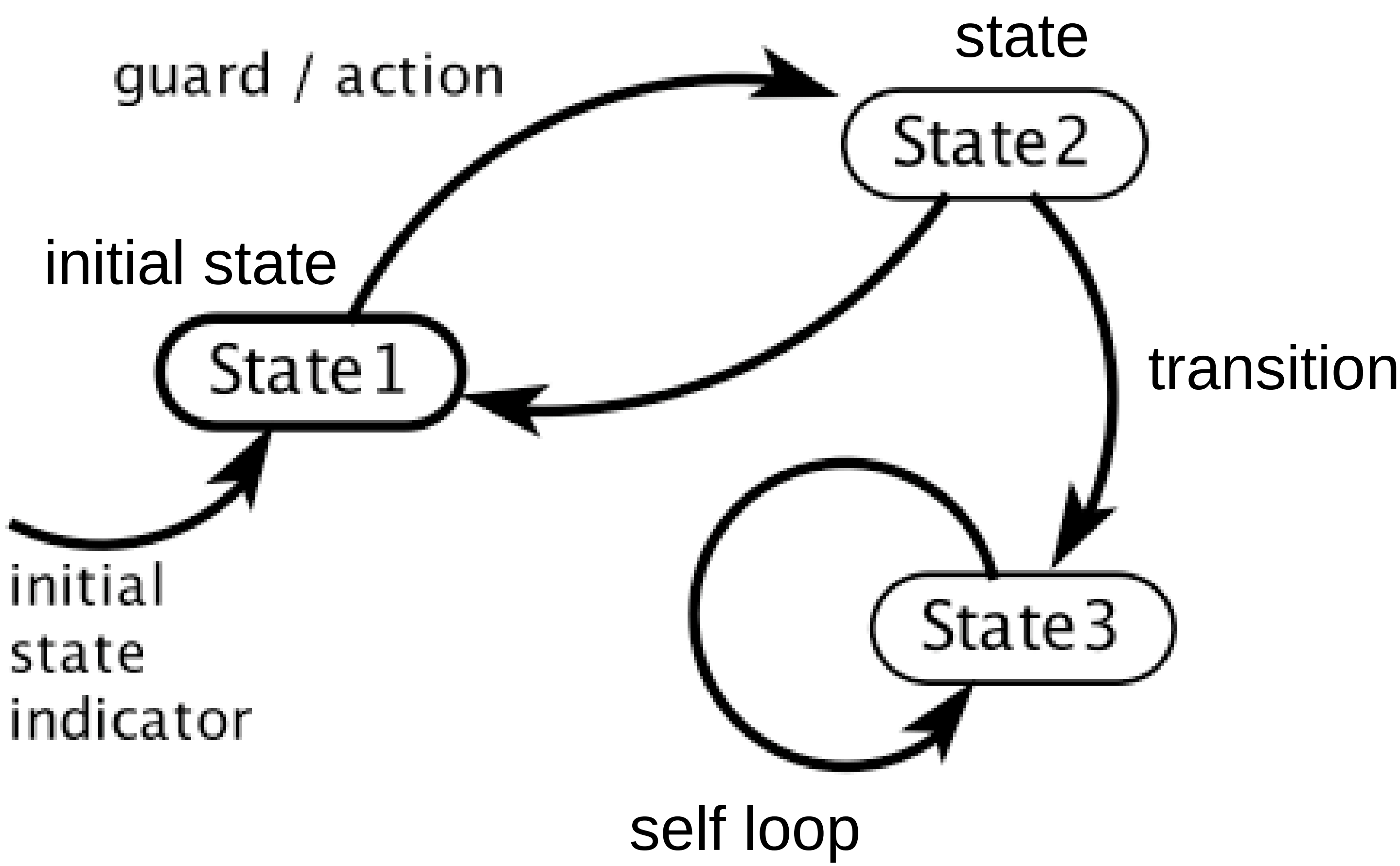
Extended state machines augment the FSM model with *variables* that may be read or written. Example:

variable: $count \in \{0, \cdots, M\}$
inputs: $up, down \in \{present, absent\}$
output $\in \{0, \cdots, M\}$



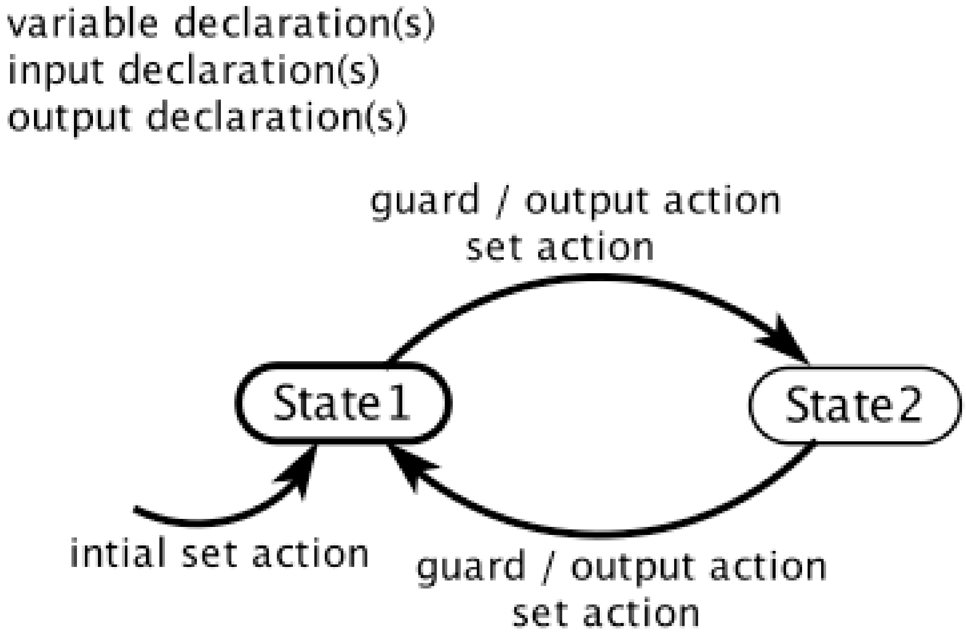
Question: What is the size of the state space?

FSM Notation



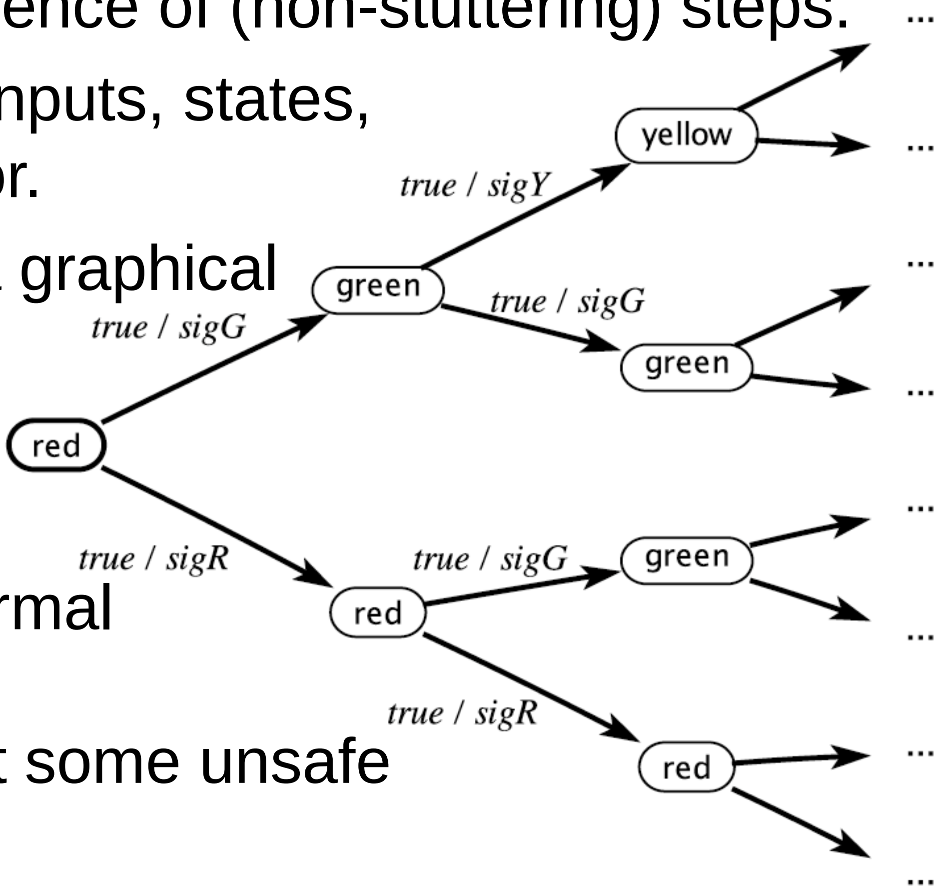
General Notation for Extended State Machines

We make explicit declarations of variables, inputs, and outputs to help distinguish the three.



Behaviors and Traces

- FSM **behavior** is a sequence of (non-stuttering) steps.
- A **trace** is the record of inputs, states, and outputs in a behavior.
- A **computation tree** is a graphical representation of all possible traces.



- FSMs are suitable for formal analysis; e.g., **safety** analysis might show that some unsafe state is not reachable.

Extended State Machines

The End

Parallel Compositions of State Machines

Modeling Concurrency

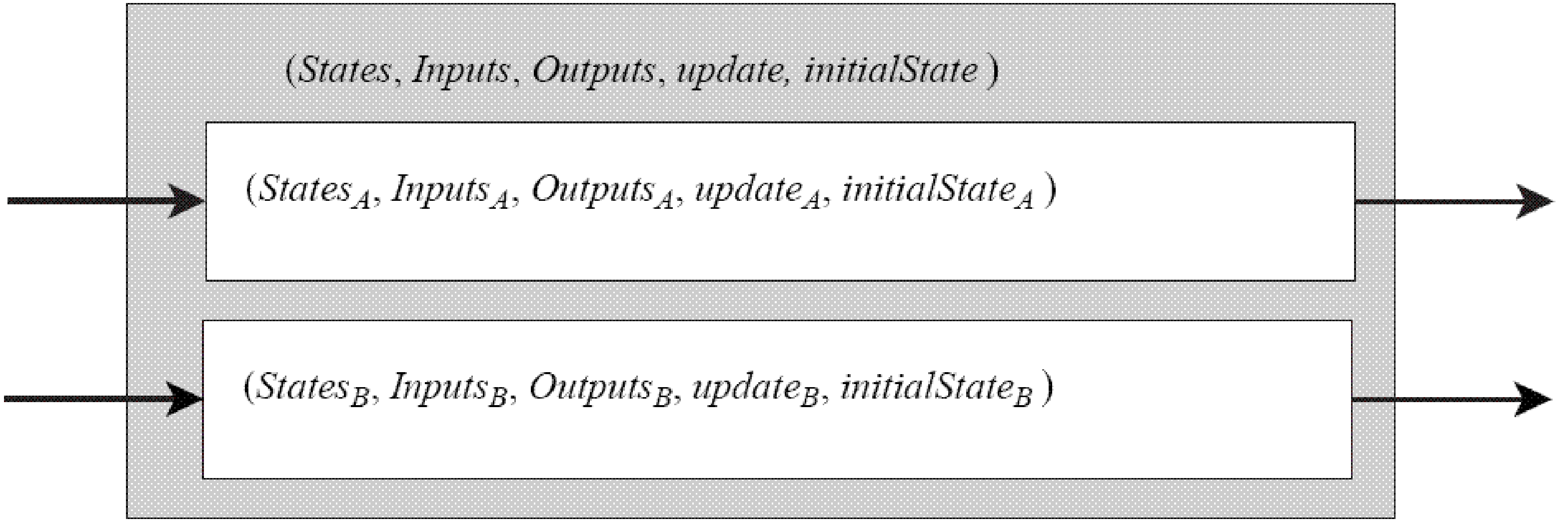
- State machines may be composed together
- Intuitively captures the **concurrent** operation of different systems (processes, tasks, threads, computers, servers, clients/servers, etc.)
- Will see examples of some primitive concurrent/**distributed algorithms** (mutual exclusion, etc.)
- Distributed algorithms are extremely important in computer networking, as different computers must operate together (e.g., clients/servers, groups of routers, etc.)

Side-by-Side (Parallel) Composition

A key question: When do these machines react?

Two possibilities:

1. Together (synchronous composition)
2. Independently (asynchronous composition)

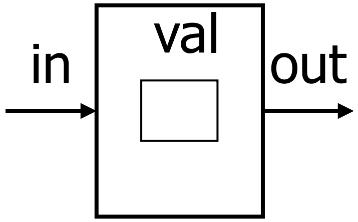


A 3-Bit Counter

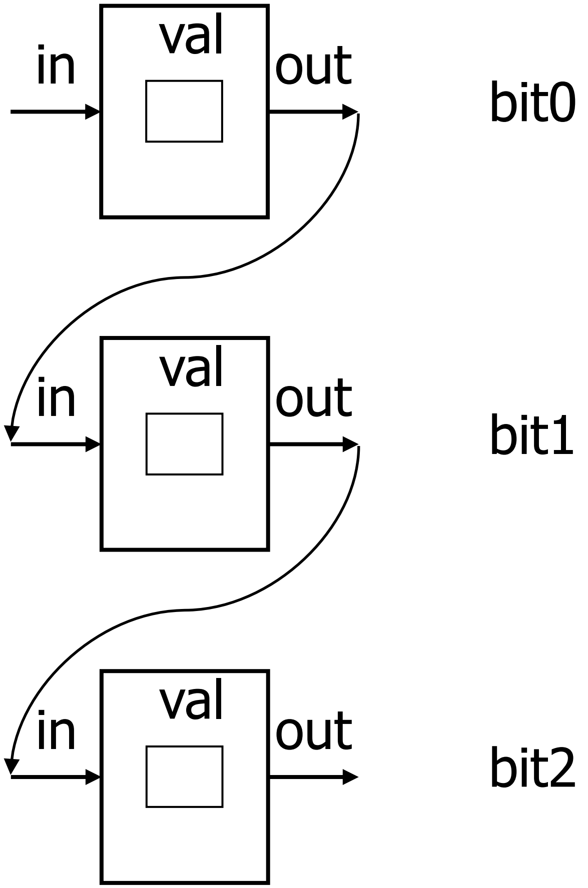
```
MODULE main
VAR
  bit0 : counter_cell(TRUE);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);

SPEC  AG AF bit2.carry_out

MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in;
```

value + carry_in mod 2

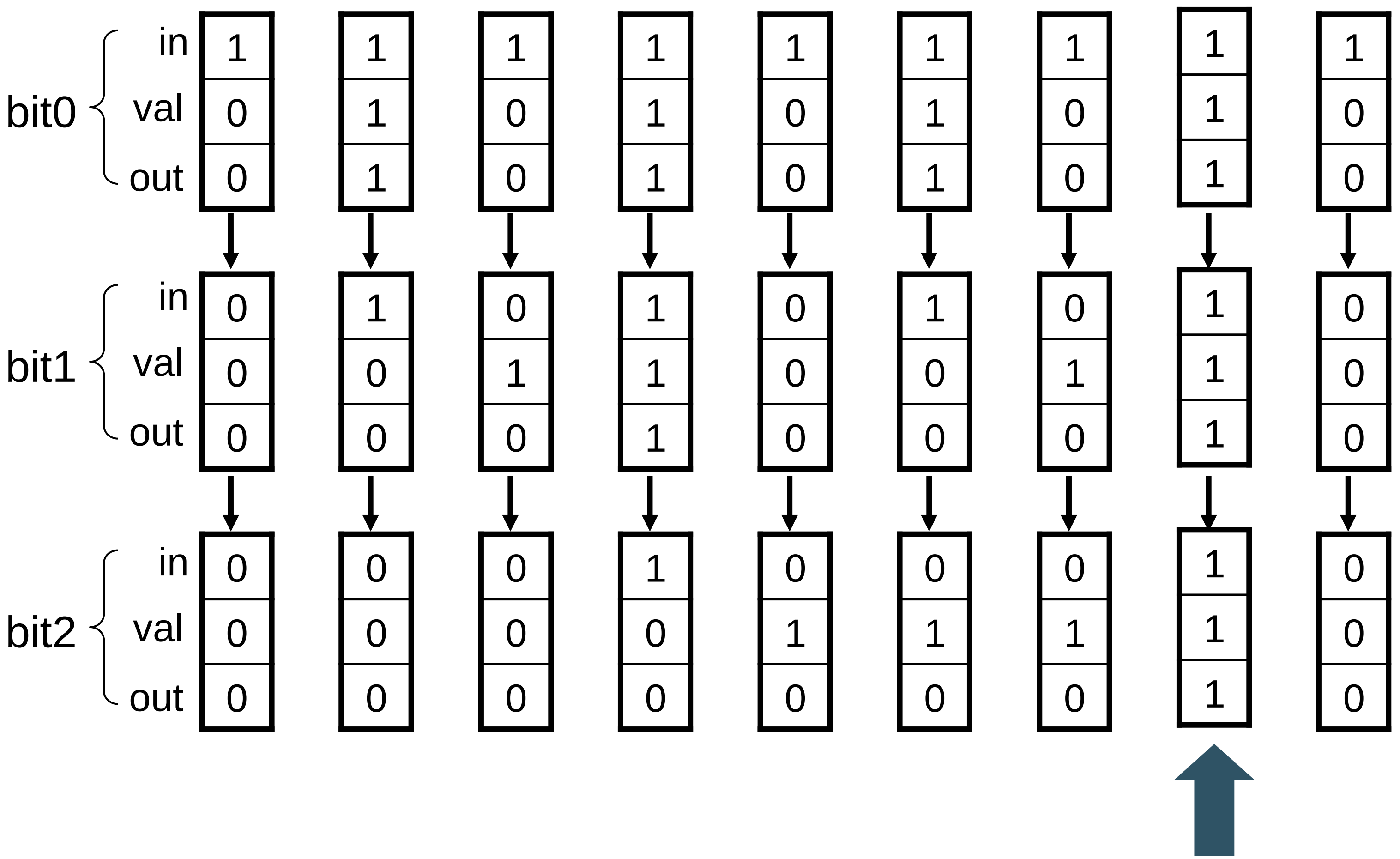
Module declaration



Module instantiations



`AG AF bit2.carry out` is true

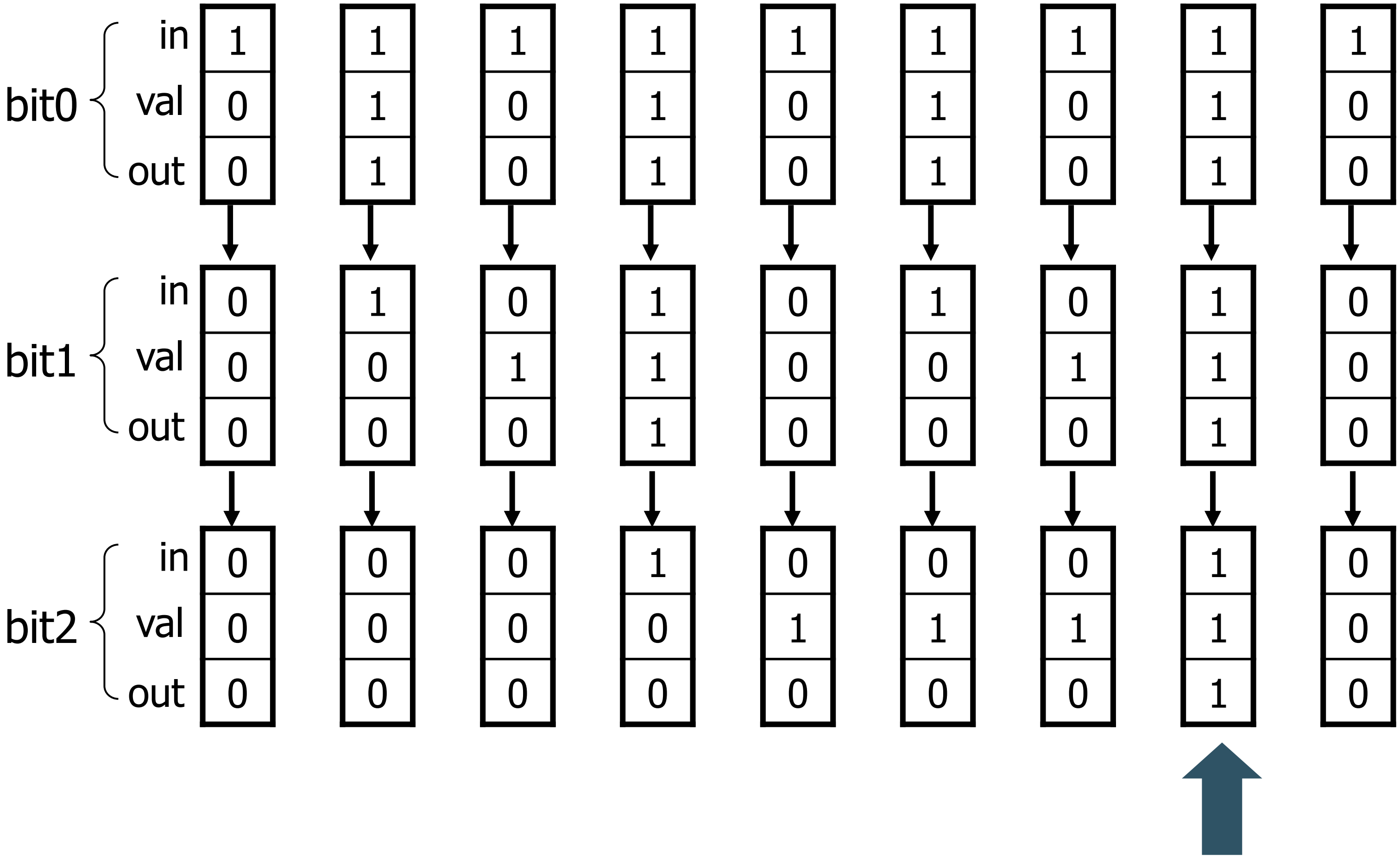


bit2.carry_out is true

A 3-Bit Counter

```
MODULE main
VAR
  bit0 : counter_cell(TRUE);
  bit1 : counter_cell(bit0.carry_out);
  bit2 : counter_cell(bit1.carry_out);

SPEC AG (!bit2.carry_out)

MODULE counter_cell(carry_in)
VAR
  value : boolean;
ASSIGN
  init(value) := FALSE;
  next(value) := value xor carry_in;
DEFINE
  carry_out := value & carry_in;
```

`AG (!bit2.carry_out)` is false



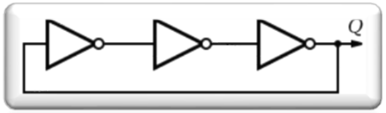
bit2.carry_out is true

Module Composition

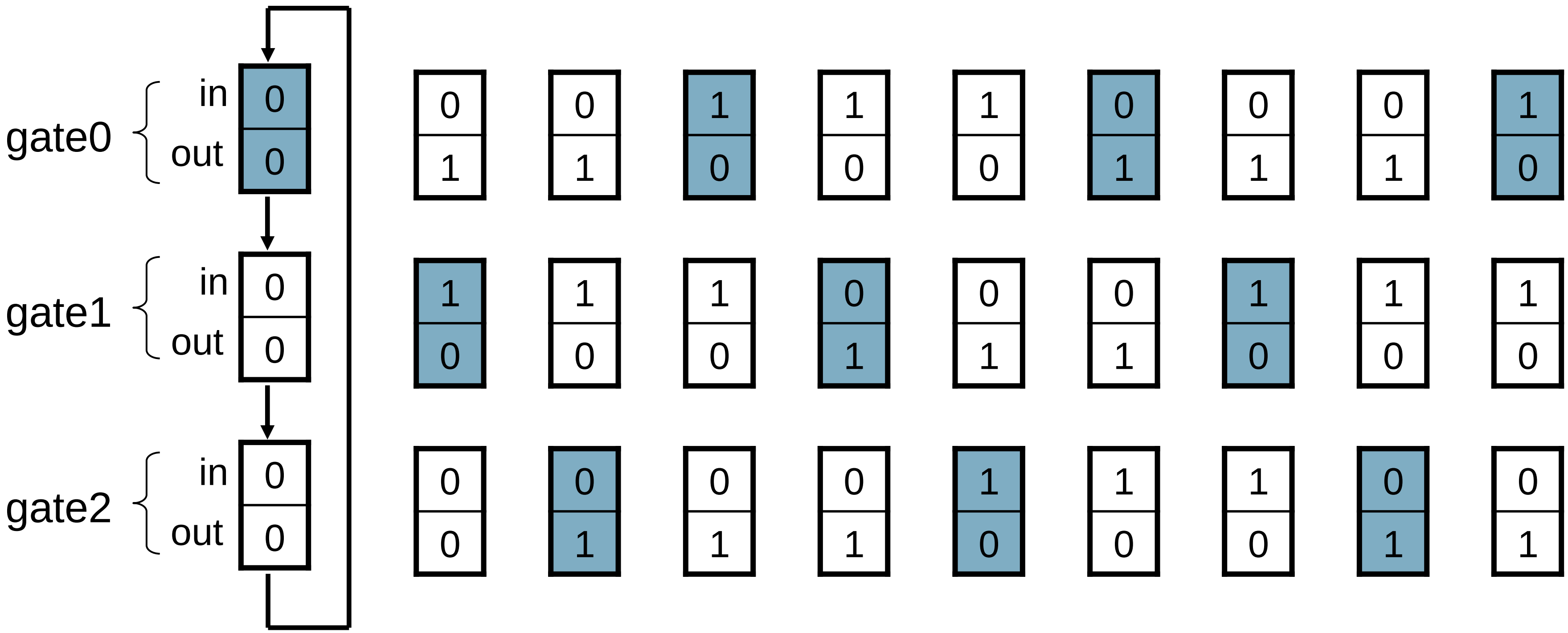
- **Synchronous** composition
  - All assignments are executed in parallel and synchronously
  - A single step of the resulting model corresponds to a step in each of the components
- **Asynchronous** composition
  - A step of the composition is a step by exactly one process
  - Variables not assigned in that process are left unchanged

Inverter Ring

```
MODULE main
VAR
  gate1 : process inverter(gate3.output);
  gate2 : process inverter(gate1.output);
  gate3 : process inverter(gate2.output);

SPEC (AG AF gate1.output) & (AG AF !gate1.output)
MODULE inverter(input)
VAR
  output : boolean;
ASSIGN
  init(output) := FALSE;
  next(output) := !input;

FAIRNESS
  running
```



In asynchronous composition, a step of the computation is a step by exactly one component. The process to execute is assumed to choose gate0, gate1, and gate2 repeatedly.



`(AG AF gate1.output) & (AG AF !gate1.output)` is true

nuXmv/NuSMV
Mutual Exclusion Example With LTL

```
MODULE main
    VAR
      semaphore : boolean;
      proc1 : process user(semaphore);
      proc2 : process user(semaphore);
    ASSIGN
      init(semaphore) := FALSE;

-- mutual exclusion: it is always the case
-- that there is at most one process in the
-- critical section
LTLSPEC G ! (proc1.state = critical &
proc2.state = critical)

-- liveness: it is always the case that, if
-- process 1 is in entering, then in the
-- future it will be in the critical
-- section
LTLSPEC G (proc1.state = entering -> F
proc1.state = critical)
```

```
MODULE user(semaphore)
  VAR
    state : idle, entering, critical, exiting;
  ASSIGN
    init(state) := idle;

    next(state) :=
    case
      state = idle : idle, entering;
      state = entering & !semaphore : critical;
      state = critical : critical, exiting;
      state = exiting : idle;
      TRUE : state;
     esac;

    next(semaphore) :=
    case
      state = entering : TRUE;
      state = exiting : FALSE;
      TRUE : semaphore;
    esac;
  FAIRNESS
    running
```

Parallel Compositions of State Machines

The End