Recursion and Networking
CS 118
Computer Network Fundamentals
Peter Reiher

Outline

- Preview and motivation
- What is recursion?
- The basic block concept
- Stacks, hourglasses, and DAGs

Preview and motivation

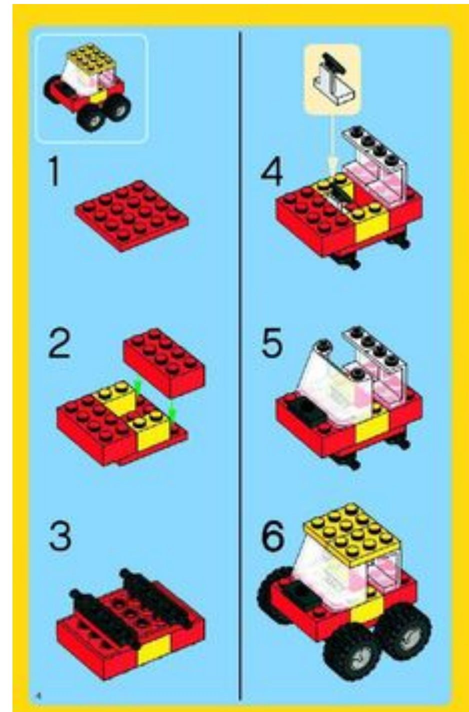
- What do we have so far?
- Putting the pieces together
- What’s missing?

What do we have so far?

- Communication
  - 2-party info. coordination over a direct link
  - Requires a protocol
- A layer
  - Homogenous indirect communication
  - Requires naming, relaying
- Stacked layers
  - Heterogeneous indirect communication
  - Requires resolution

Putting them together

- We have the pieces
  - Communication
  - Layers
  - Stacking
- Some assembly required
  - Is there just one way?



How do we know:

- Which layers *can* stack
  - Have resolution mechanisms
- Which layer you *should* use next
  - Does it help you move closer towards communicating?

What's missing?

- A map
  - To show layer relationships

- A way to use that map
  - Picking a trail
  - Following a trail
  - Some breadcrumbs to find our way home

Maps and map use

- We'll start with map use
  - That's where recursion comes in
- Then we'll look at the map
  - Hint: remember stacks and hourglasses?

Using recursion to describe network layering

- We will use the general idea of recursion to unify our understanding of network layering
- That's NOT how the code, hardware, and most architectures really work
  - You'd look in vain for obvious recursive steps
- But at a high level it's really what's going on
- REMEMBER – we're talking concepts, not implementations, here

What is recursion?

- Definition
- Properties
- Variants

Induction

- Base case:
  - Prove (or assert) a starting point
  - E.g., 0 is a natural number
- Inductive step:
  - Prove (or assert) a composite case *assuming* already proven cases
  - E.g., X+1 is a natural number if X is too

Induction proof

- Prove: $\sum_{i=0}^{N} i = \frac{N(N+1)}{2}$
- Base:
  - *Prove it is true for N=0*
  - When N=0, sum is correct: $\frac{0\,(0+1)}{2} = 0$
- Inductive step:
  - *If it is true for N, prove it is true for N+1*
  - For N, assume sum is: $\frac{N\,(N+1)}{2}$
  - For N+1, sum should be: $\frac{N\,(N+1)}{2} + (N+1)$
  - And it is: $\frac{N\,(N+1)}{2} + (N+1) = \frac{N\,(N+1)}{2} + \frac{2(N+1)}{2} = \frac{(N+1)\left((N+1)+1\right)}{2}$

Recursion: backwards induction

- Reductive step:
  - Rules that reduce a complex case into components, assuming the component cases work
- Base case:
  - Rules for at least one (irreducible) case

Recursion: example

- Reduction case:
  - $N! = N * (N - 1)!$

- Base case:
  - $0! = 1$

Recursion as code

- int factorial(int n)

```
{
    if (n < 0) {
        exit(-1); // ERROR
    }
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n-1);
    }
}
```

Fibonacci series

- Base:
  - Fib(0) = 0
  - Fib(1) = 1
- Reduction:
  - F(n) = F(n-1) + F(n-2)

Properties of recursion

- Base case
  – Just like induction
- Self-referential reduction case
  – Just like induction, but in reverse

Differences

- Induction
  - Starts with the base case
  - Uses finite steps
  - Extends to the infinite

- Recursion
  - Starts with a finite case (base or otherwise)
  - Uses finite steps
  - Reduces to the base case

Properties of recursion

- All cases are the same
  - Except the base case(s)
- Recursive step is self-referential
  - Import interface = export interface
  - “Provides what it expects”
  - E.g., C func: `vtype recfunc(vtype x)`

Variants of recursion

- Regular
- Tail

Regular recursion

- Reductive step is an arbitrary function
  - MUST include self-reference
  - Self-reference MUST be ‘simpler’

  - int fib(n) { return fib(n-1) + fib(n-2); }

Why simpler?

- Reductive step must simplify
  - If it *ever* doesn't, recursion is infinite
  - If you don't change just once, you *never* will

Tail recursion

- Same rules as regular recursion
  PLUS
- Self-reference ONLY as the *sole* last step

  - ```
    int fib(int i) {
            return dofib(i, 0, 1);
    }
    ```
  - ```
    int dofib(int i, int x, int y) {
            if (i==0) { return x; }            // base case
            if (i==1) { return y; }            // base case
            return dofib(i-1, y, x+y);         // reduction step
    }
    ```

Why tail recursion?

- Replace self-reference with “goto”
  - Turns recursion into a *while* loop
  - int fib(int i) {
    ```
    int fib(int i) {
            return dofib(i, 0, 1);
    }
    ```
  - int dofib(int i, int x, int y) {
    ```
    int dofib(int i, int x, int y) {
            while (i > 0) {
                    tx = x; ty = y;                  // need for temp storage
                    i = i-1; x = ty; y = tx+ty;      // “recursive call”
            }
            return x;
    }
    ```

How is recursion related to networking?

- Base case: communication
  - Two parties already directly connected
- Reduction steps: networking
  - Stacked layering = regular recursion
  - Relaying = tail recursion

Stacked layering as recursion

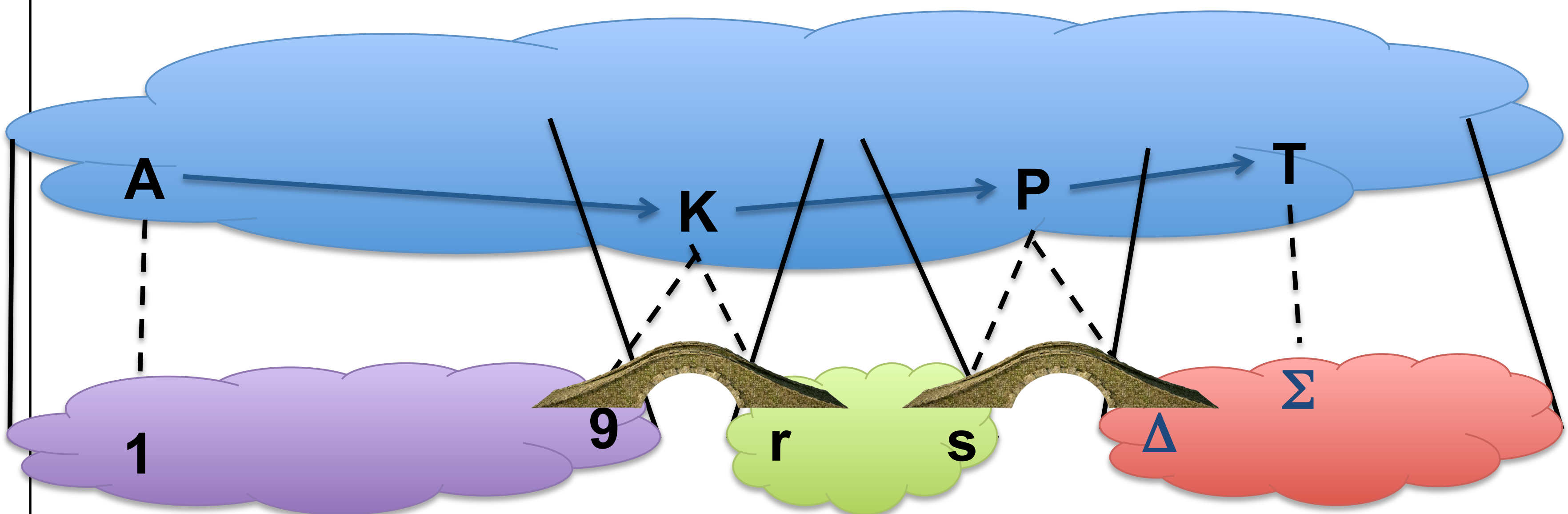
- P can reach Q
  - Assuming P translates to X,
  - Q translates to Y,
  - and X can reach Y
- Turns P-Q layer into X-Y layer
  - Using resolution
- Base case – some layer in the stack allows the source to reach the destination

Relaying as tail recursion

- A can reach C
  - Assuming A can reach B
  - and B can reach C
- How is this tail recursion?
  - We’ll get back to that …

Recall how stacked layering works

- Get to the layer you share with dest.
  - Go down and up to get where you need to go



Where's the elevator?

- Next layer down?
  - When do we do this?
    - When we don't share a layer with current destination
    - How do we know?
- What do we do if we can't go down?
    - We pop "up" instead
    - Then we need to pick another layer to go down
    - How do we know?

Let's start with the elevator itself

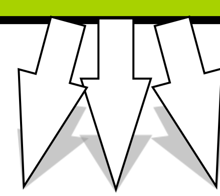
The basic block

- The block
- Interfaces
- Internal functions
- The role of naming and routing

The block

- The elevator:

```
LAYER(DATA, SRC, DST)
    Process DATA, SRC, DST into MSG
    WHILE (Here <> DST)
        IF (exists(lower layer))
            Select a lower layer
            Resolve SRC/DST to next layer S',D'
            LAYER(MSG, S', D')
        ELSE
            FAIL /* can't find destination */
        ENDIF
    ENDWHILE
    /* message arrives here */
    RETURN {up the current stack}
```



Next Layer

What's happening inside…

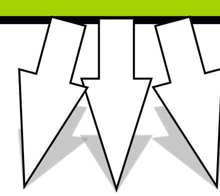
- A layer is…

```
LAYER(DATA, SRC, DST)
    Process DATA, SRC, DST into MSG
    WHILE (Here <> DST)
        IF (exists(lower layer))
            Select a lower layer
            Resolve SRC/DST to next layer S',D'
            LAYER(MSG, S', D')
        ELSE
            FAIL /* can't find destination */
        ENDIF
    ENDWHILE
    /* message arrives here */
    RETURN {up the current stack}
```

What's happening inside…

- A layer is:
  - Prepare msg for communication

```
LAYER(DATA, SRC, DST)
    Process DATA, SRC, DST into MSG
    WHILE (Here <> DST)
        IF (exists(lower layer))
            Select a lower layer
            Resolve SRC/DST to next layer S',D'
            LAYER(MSG, S', D')
        ELSE
            FAIL /* can't find destination */
        ENDIF
    ENDWHILE
    /* message arrives here */
    RETURN {up the current stack}
```



Next Layer

What’s happening inside…

- A layer is:
  - Is it for you?

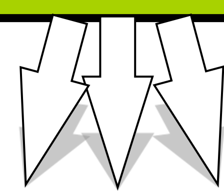
```
LAYER(DATA, SRC, DST)
  Process DATA, SRC, DST into MSG
  WHILE (Here <> DST)
    IF (exists(lower layer))
      Select a lower layer
      Resolve SRC/DST to next layer S’,D’
      LAYER(MSG, S’, D’)
    ELSE
      FAIL /* can’t find destination */
    ENDIF
  ENDWHILE
  /* message arrives here */
  RETURN {up the current stack}
```

Next Layer

What's happening inside…

- A layer is:
  - Is it for you?
    - Yes – done
- Well, except you need to go back up the stack

```
LAYER(DATA, SRC, DST)
    Process DATA, SRC, DST into MSG
    WHILE (Here <> DST)
        IF (exists(lower layer))
            Select a lower layer
            Resolve SRC/DST to next layer S',D'
            LAYER(MSG, S', D')
        ELSE
            FAIL /* can't find destination */
        ENDIF
    ENDWHILE
    /* message arrives here */
    RETURN {up the current stack}
```

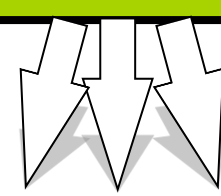


Next Layer

What's happening inside…

- A layer is:
  - Is it for you?
    - No:
      - Find help

```
LAYER(DATA, SRC, DST)
    Process DATA, SRC, DST into MSG
    WHILE (Here <> DST)
        IF (exists(lower layer))
            Select a lower layer
            Resolve SRC/DST to next layer S',D'
            LAYER(MSG, S', D')
        ELSE
            FAIL /* can't find destination */
        ENDIF
    ENDWHILE
    /* message arrives here */
    RETURN {up the current stack}
```

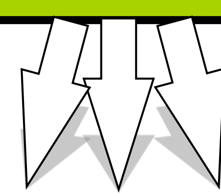


Next Layer

What's happening inside…

- A layer is:
  - Is it for you?
    - No:
      - Find help
      - Translate ID

```
LAYER(DATA, SRC, DST)
    Process DATA, SRC, DST into MSG
    WHILE (Here <> DST)
        IF (exists(lower layer))
            Select a lower layer
            Resolve SRC/DST to next layer S',D'
            LAYER(MSG, S', D')
        ELSE
            FAIL /* can't find destination */
        ENDIF
   ENDWHILE
   /* message arrives here */
   RETURN {up the current stack}
```

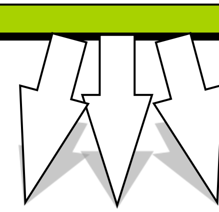


Next Layer

What’s happening inside…

- A layer is:
  - Is it for you?
    - Yes – done
    - No:
      - Find help
      - Translate ID
      - Send it there

```
LAYER(DATA, SRC, DST)
    Process DATA, SRC, DST into MSG
    WHILE (Here <> DST)
        IF (exists(lower layer))
            Select a lower layer
            Resolve SRC/DST to next layer S’,D’
            LAYER(MSG, S’, D’)
        ELSE
            FAIL /* can’t find destination */
        ENDIF
    ENDWHILE
    /* message arrives here */
    RETURN {up the current stack}
```



Next Layer

Deeper look at the steps

- Prepare message for communication
  - Take what you get (from the user/FSM)
  - Add whatever you need for *your* state sharing
  - Run the protocol at this layer
- Then check to see where it goes

Why prepare *then* send?

- You can't reverse order
  - You need your message in order to talk
  - One request might turn into multiple messages

- It might be for you
  - A nice degenerate case
  - "Dancing with yourself"

Why does this work?

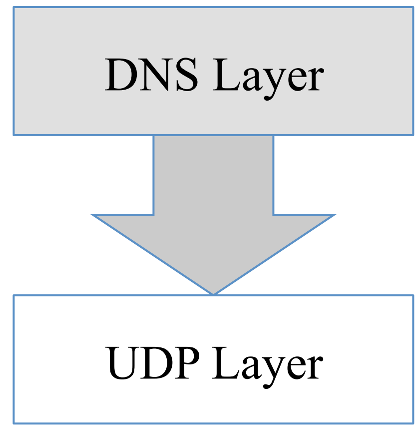
- Recursion
  - Base case: direct connection
  - Recursive steps:
    - Layering
    - Relaying

An example: DNS request

- User requests `gethostbyname()` to the OS
  - Prepares the DNS query message to the default server (random root or local)
  - Is it for me?
    - No:
      - Find a way to get to the server
      - Translate this layer's names ("YOU", "servername") into the next layer's names
      - RECURSE

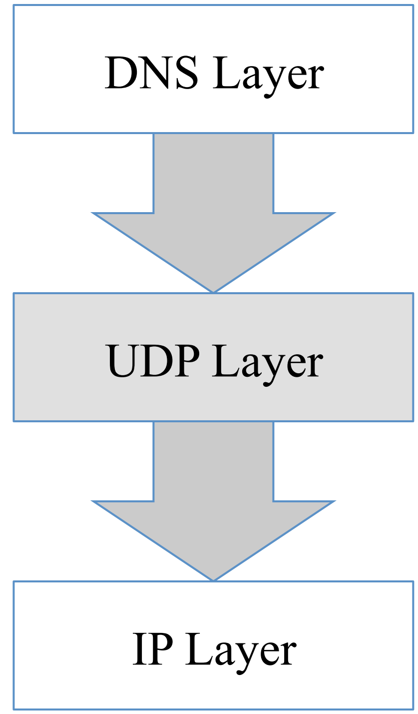
Recursion steps

- User calls `gethostbyname()` to OS
  - Make DNS query "me"->dns
  - For "dns" use UDP
  - Translate me to bob.com: 61240, dns to ns.com:53
  - Call UDP



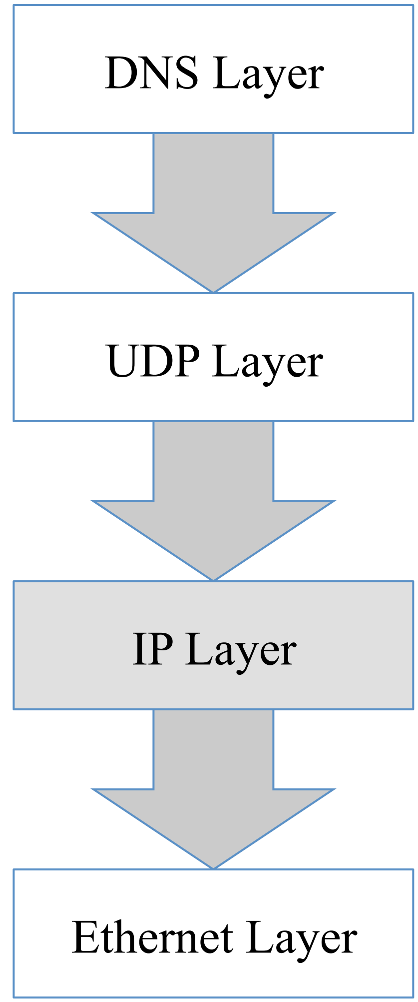
Recursion steps

- User calls `gethostbyname()` to OS
  - …
  - Call UDP
    - Make UDP message 61240->53
    - For "UDP" use IP
    - Translate bob.com to 52.3.5.3, ns.com to 2.43.14.123
    - Call IP



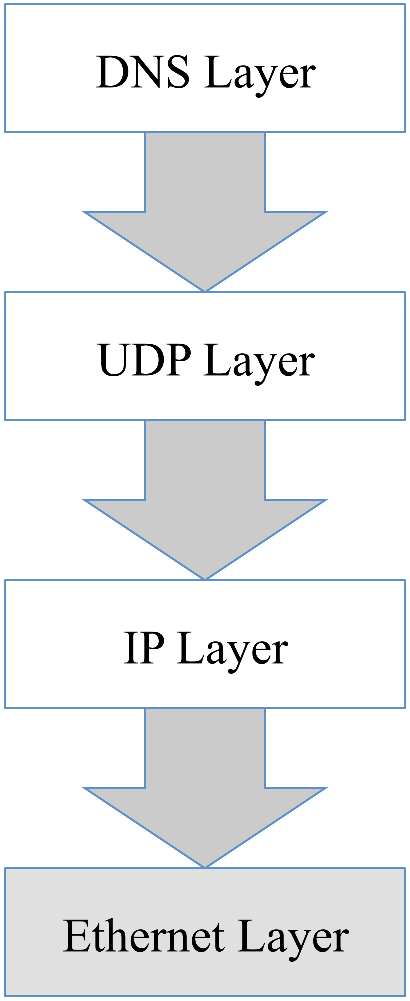
Recursion steps

- User calls `gethostbyname()` to OS
  - …
  - Call UDP
    - …
    - Call IP
      - Make IP message 52.3.5.3 ->2.43.14.123
      - For IP use ethernet
      - Translate 52.3.5.3, 2.43.14.123 to ethA, ethB
      - Call Ethernet



Recursion steps

- User calls `gethostbyname()` to OS
  - …
  - Call UDP
    - …
    - Call IP
      - …
      - Call Ethernet
        - » Make ethernet message ethA->ethB
        - » For ethB, use em0 directly
        - » BASE CASE – send it!



What about at the receiver?

- Message comes in at some base protocol
  - E.g., the Ethernet on the receiving node
- It's to be handled by a higher level protocol
  - E.g., DNS
- How do we get up to that layer?
- Recursion in the opposite direction
- Call up the stack, instead of down

Recursion block at receiver

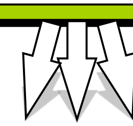
- Now you pop back up the stack
- You're at the destination, but not at the right layer
- It's recursive calls again
- But in the opposite direction

```
LAYER(DATA, SRC, DST)
   Process DATA, SRC, DST into MSG
   WHILE (Here <> DST)
      IF (exists(lower layer))
         Select a lower layer
         Resolve SRC/DST to next layer S',D'
         LAYER(MSG, S', D')
      ELSE
         FAIL /* can't find destination */
      ENDIF
   ENDWHILE
   /* message arrives here */
   RETURN {up the current stack}
```

Interfaces

- What does the block input?
  - Source name
  - Destination name
  - Message
  - <u>*In the layer of the block*</u>
- What does the block output?
  - Recursive step: same thing! (it has to)
  - Base case: physical signal <u>*with same effect*</u>

```
LAYER(DATA, SRC, DST)
  Process DATA, SRC, DST into MSG
  WHILE (Here <> DST)
    IF (exists(lower layer))
      Select a lower layer
      Resolve SRC/DST to next layer S',D'
      LAYER(MSG, S', D')
    ELSE
      FAIL /* can't find destination */
    ENDIF
  ENDWHILE
  /* message arrives here */
  RETURN {up the current stack}
```



Process the message

- This is the protocol FSM
  - Starts in default state (non-persistent) or last state (persistent)
  - Tape-in is the "input" message to be shared
  - Tape-out is the "output" message(s) to share with the corresponding FSM at the destination

The role of naming and routing

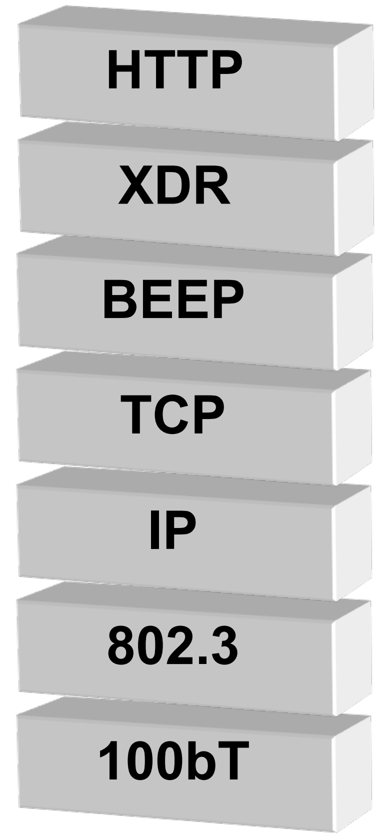
- Resolution tables
  – Indicate whether you can get somewhere
  – Translate names from one layer to next
- I.e., resolution tables are BOTH
  – Name translation
  – Routing

Stacks, hourglasses, and DAGs

- Recursion: the engine that gets you there
  - But it needs a map to follow

Stacks

- A linear chain of layers
  - "Next layer" is fixed
  - Describes a path *taken* by the recursive steps
  - But not all possible paths that *could* be taken

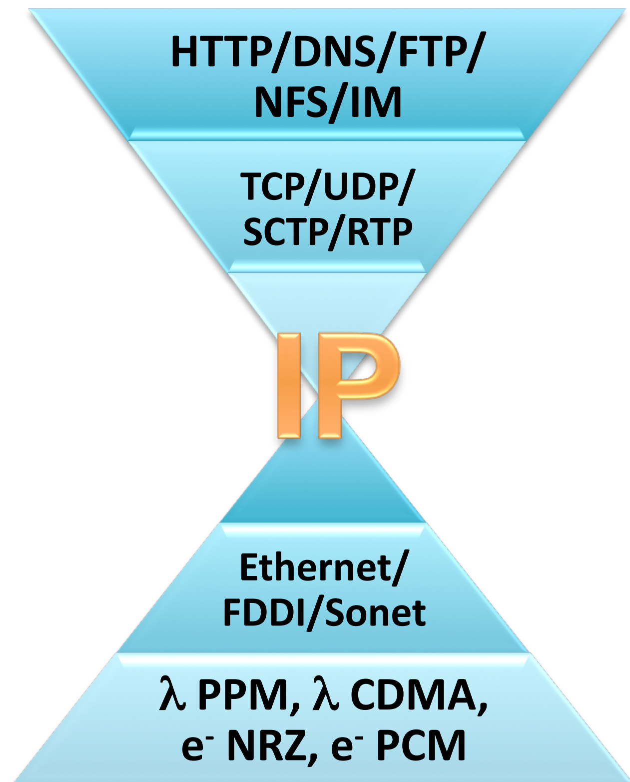


The Hourglass

- A bigger picture
  - Many possible paths
- Top half describes reuse
  - Many different layers share ways to "get there"
- Bottom half describes choices
  - One layer has many ways to "get there"

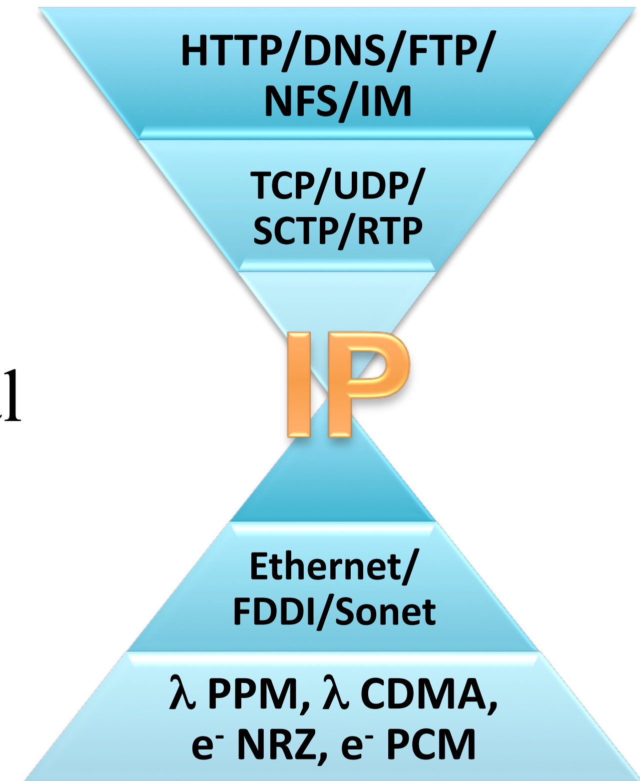
Top half

- HTTP, DNS, FTP
  - All use TCP
- TCP, UDP, SCTP
  - All use IP
- Sharing to reuse mechanism



Bottom half

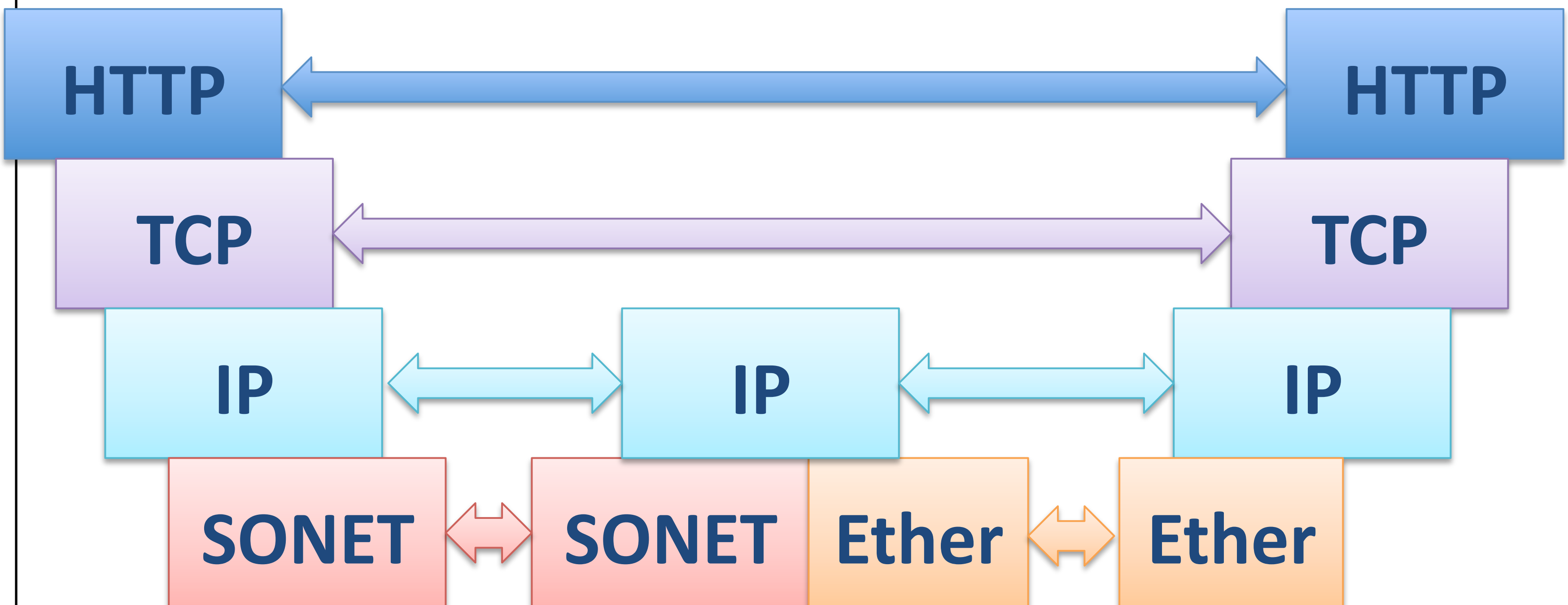
- IP
  - Can use ethernet, sonet
- Ethernet
  - Can use optical, electrical
- Choice to allow diversity and optimization



Who talks to whom?

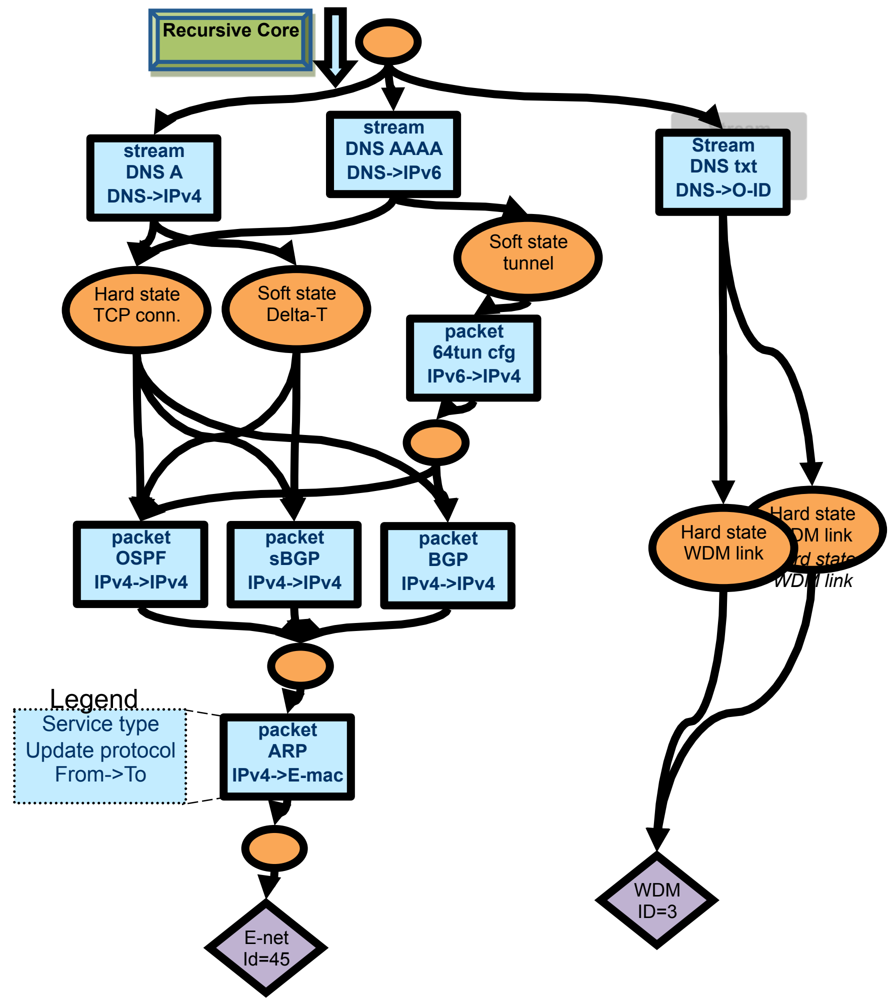
- Every communicating pair
  - Is at the same layer
  - MAY have different lower layers (recursive next steps)
  - CANNOT have different upper layers (share a common previous recursive steps)

Who talks to whom



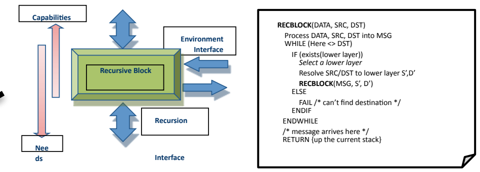
The DAG

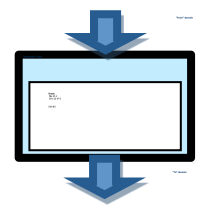
- Structure of tables
  - Directed
  - Acyclic
  - Graph

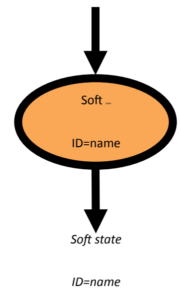


DAG Components

- Components
  - Recursive block (RB)
  - Translation table (TT)
  - State instance (SI)







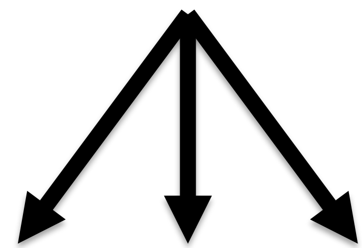
- Structure
  - Directed acyclic graph
    - TT as primary nodes, connected on matching entries
    - SI as intermediate nodes on all arcs connecting TTs
  - Recursive block – traverses the graph

What does the DAG indicate?

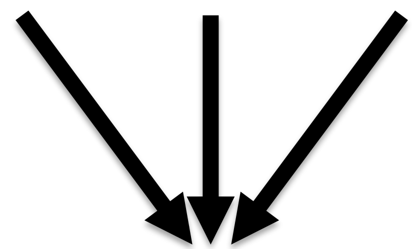
- Recursive steps
- FSM rules and state

Recursive steps

- Fan-out
  - Alternate (equivalent) next step

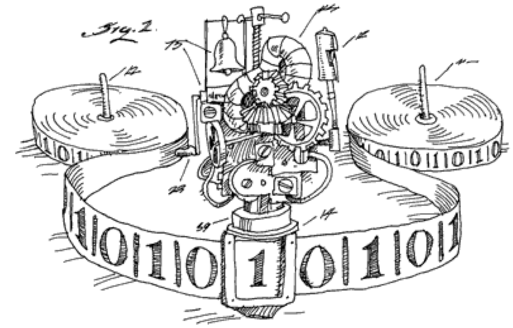


- Fan-in
  - Protocol reuse/sharing (NOT interoperation)



FSM rules and state

- A place to “wait” until there’s more tape-in
  - State needs a place to wait
  - FSM rules need a place too
  - I.e.,a paused FSM



- i.e., the “breadcrumbs”

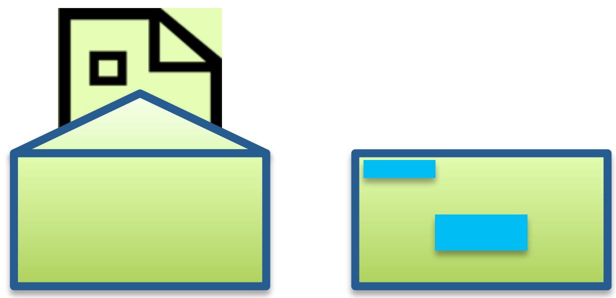
Follow the yellow brick road

(overlapping euphemism alert)

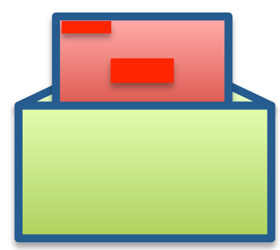
- Picking a trail
  - Use the map; search all "next step" options
  - Find a choice with a translation entry
- Follow the trail
  - Use the "breadcrumbs" (state) left by previous msg
- Find your way home
  - Use the "breadcrumbs" inside the message

Breadcrumbs inside the message?

- Remember the message in the envelope?



- Envelope inside an envelope
  - Inner envelope is the “breadcrumbs”
  - Encodes path UP at receiver



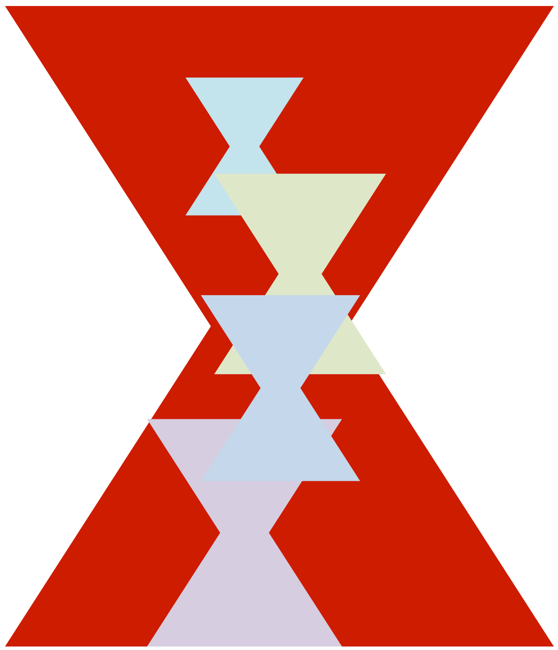
The DAG looks complicated

It is because it supports:

- More than one hourglass
- Dynamic path selection

More accurate than ONE hourglass

- Describes many overlapping hourglasses



Dynamic graph path selection

- Internet “stacks” graph
  - Static
  - Only ever picks one choice:
    it never tries another on failure
- Other variants allow dynamic choice
  - Research projects
  - Datacenter optimizations

Summary

- Networking traverses layers via recursion
- That recursion needs a map
- The map governs recursive step choice *and* manages FSM (protocol) state