Putting It All Together
CS 118
Computer Network Fundamentals
Peter Reiher

What have we learned?

- We can communicate with another party over a channel
- Channels have limited capacity to carry information
- Many channels experience noise that alters some signals
  - Which reduces their capacity to carry information
- We can use encodings to overcome noise
  - But there is a computable limit to that

More things we've learned

- Some channels can be shared by more than two parties
- Which introduces a need for naming of the parties
- We need methods to share such channels
  - Either under the control of some party
  - Or by some set of rules everyone follows

More things we’ve learned

- Only a limited number of parties can share realistic channels
  - Not an approach that scales up too high
- To reach higher scale in participating parties, we need to introduce relaying
  - Having some parties pass on messages to others
- Relaying increases the need for naming
- And introduces many new issues

Things we’ve learned about relaying

- Relaying requires routing
  - Figuring out how to pass on particular messages
- Relaying requires congestion control
  - We’re sharing channels we don’t directly observe
  - They have limited capacity
  - We must make sure they aren’t overloaded
- Relaying magnifies security and privacy problems
  - Parties we don’t control handle our messages

Protocols

- Both direct communications and networking require parties to follow rules
  - Describing what each one does when
- Those rules are called protocols
- Usually, all parties must agree on the protocol rules
- An individual party’s progress in a protocol can be encoded as a finite state machine

Protocols and layers

- No single protocol will work well in all situations
- But we need to share a protocol to communicate to someone else
  - Who might be in a different situation
- Solve the problem with layered protocols
  - A common high level protocol is shared
  - Specialized lower level protocols handle different situations

Layering protocols

- We need some way to compose layers of protocols
  - Rules for putting them together
- Recursion describes the basic method of composing protocols
- We recurse down a protocol stack
- And back up again on the other side

Organizing layered protocols

- In real use, our protocols follow an hourglass shape hourglass shape
  - Many choices at the top
  - Few (one) choices in the middle
  - Many choices at the bottom
- Thinking in terms of a DAG helps clarify the choices and their implications

Layers and optimization

- One job of a protocol layer is to improve an underlying deficiency
  - To change the network we have into the network we want
- Deficiencies include:
  - Capacity
  - Latency
  - Reliability
  - Security and privacy
- Optimizations are made at many layers

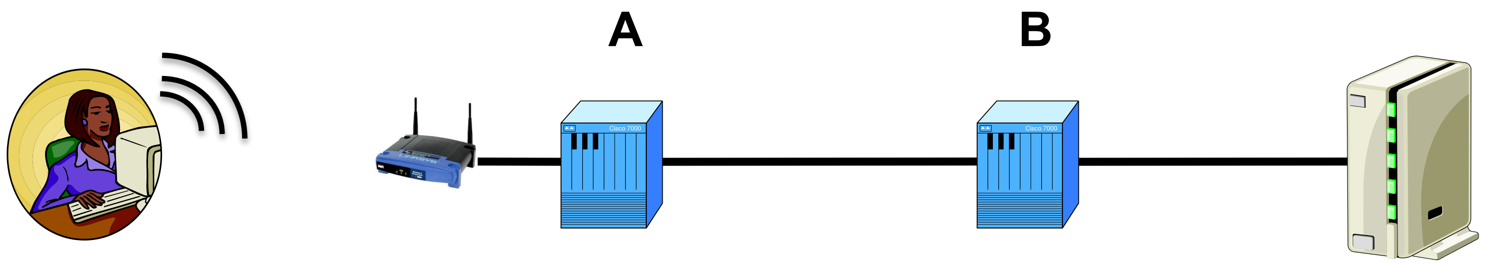
Putting it together

- Let's go through an example
  - An HTTP request
- Looking at the layers involved
- And their optimizations and behaviors

Our example

Carol wants a web page from www.funsite.com

Here's the server for www.funsite.com



To get there, she'll need to use her wireless LAN

Then go through two routers

Then go up through the server's protocol stack

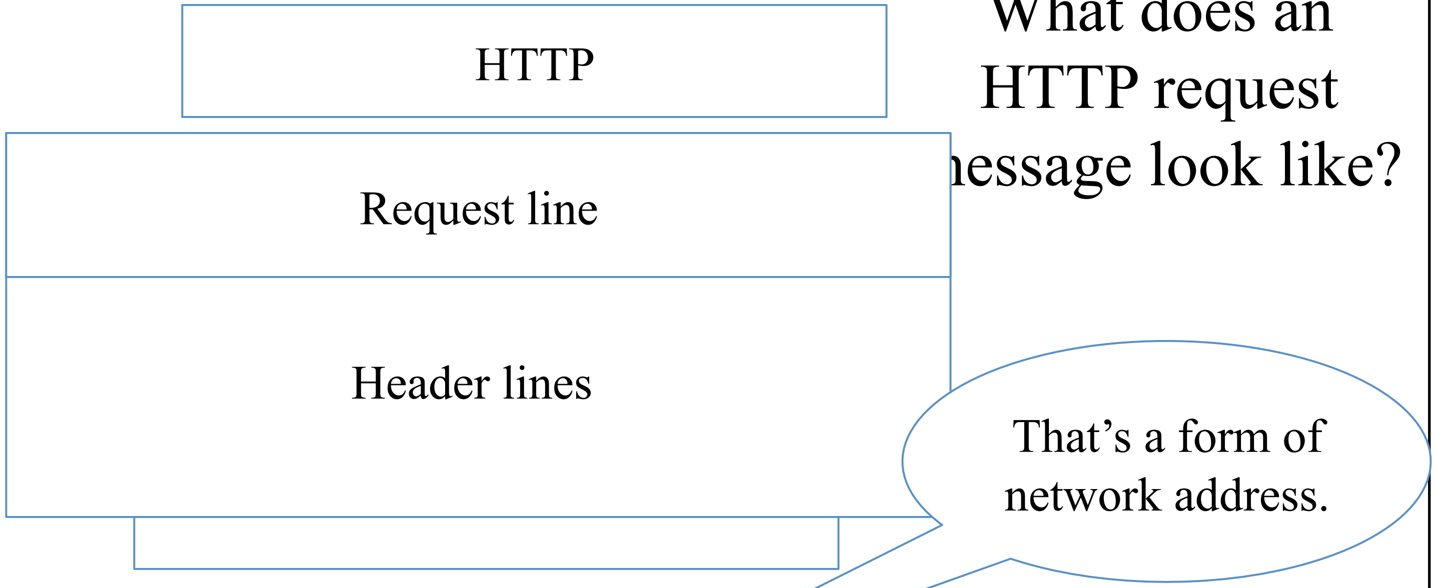
Getting started

- Carol is running a web browser
- It generates a system call requesting a web page
- That system call is translated by the OS to require network activity
- Now we enter the realm of networking

Starting at the top

Carol wants a web page from www.funsite.com

Her browser asks the OS to send an HTTP request message



What does an HTTP request message look like?

Request line indicates that it's a GET request

And includes the URL

Now what?

- From a protocol stack perspective, we need to determine if this layer can deliver it
- Well, can it?
- The local node isn't the destination
- The HTTP layer doesn't know how to forward itself
- So, no, it can't deliver it
- Better go down the stack

Moving down

So we've got an HTTP message to send, and this layer can't handle it

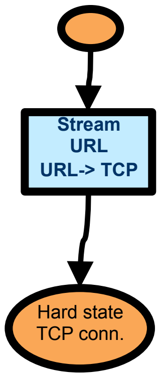
Find a lower layer that can help

HTTP

OK, TCP can help

TCP is connection oriented

Let's assume we already have our TCP connection established



Getting the TCP address info

- We’ve discussed this before
- The IP address is obtained by dynamic naming
  - Translating the DNS name to an address
  - Probably via either cache or a DNS request
- The port number is based on static naming
  - HTTP is pre-defined to go to port 80
- We’ve got the name, so now we can build a TCP message

Now something a bit funky

- Remember the hourglass with the narrow waist?
- That waist is the IP protocol
- Things right above it tend to use IP by default
- In particular, TCP does
- So if the IP address will appear in the IP header, why include it in the TCP header?
- Well, let's not (and we don't)
  - We'll worry about the IP address later

A cross-layer optimization

Other TCP operations

- We do whatever our TCP FSM tells us to do, now
- We consider the send window
- And the congestion window

Space-related optimizations

- If either tells us not to send, we wait
  - Saving this message
- If everything is OK to send more, we move down the stack

Now what?

- From a protocol stack perspective, we need to determine if this layer can deliver it
- Well, can it?
- The local node isn't the destination
- The TCP layer doesn't know how to forward itself
- So, no, it can't deliver it
- Better go down the stack

Moving down

So we've got a TCP message to send, and this layer can't handle it

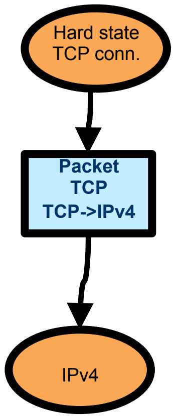
Find a lower layer that can help

TCP

TCP likes to work with IP

So let's drop the TCP message to the IP level

In particular, let's use IPv4



Doing IP’s work

- IP is a packet protocol
  – Not a stream protocol
- It handles each packet independently
- So we don’t need any preparation to send out this packet
- But . . .
- IP doesn’t itself forward data

Adding the IP header

- The IP header will be used for network transit
- It must specify what's needed to get from one side of the network to the other
- In particular, a source and destination address
- We've got the source address by default
- We need a destination address
  - We've got one, but it's www.funsite.com
  - We need an IP address

Getting the IP address

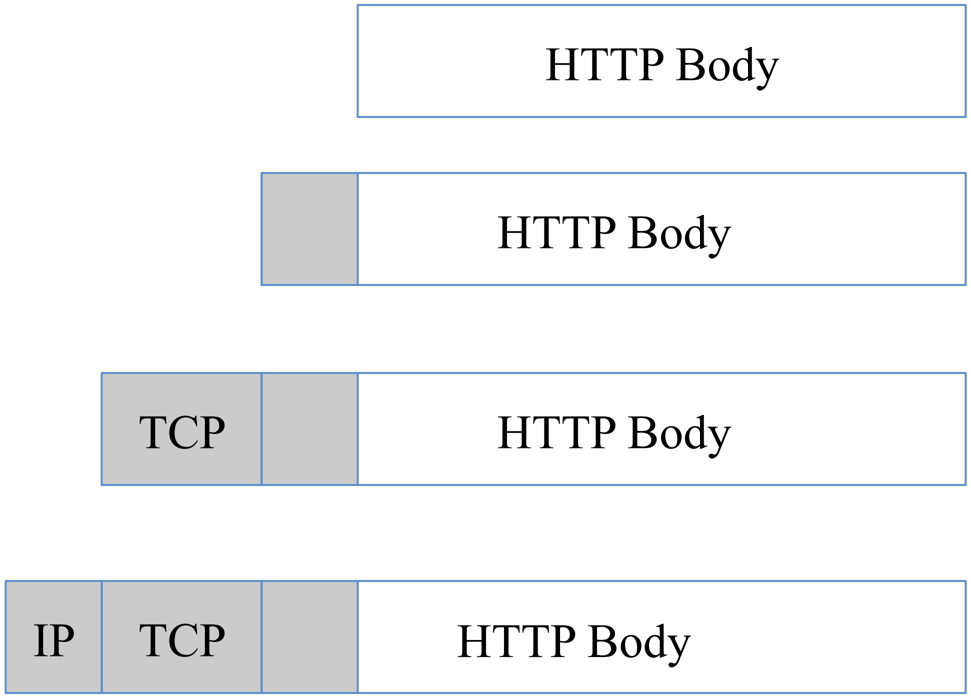
- We've got www.funsite.com and we want something like 131.179.192.47
- We may have that information in a local file
  - Like /etc/hosts
- Or we may have cached it
  - Having done the translation previously
- If not, we need to use DNS to look it up
  - We won't go into the details here

Another space-related optimization

More IP work

- Fill in the rest of the header
  - Time to live
  - Flags
  - Length
  - Checksum
  - Etc.
- No flow control, no congestion control
  - IP doesn’t do those
- Just get on with sending it

So what have we got so far?



We started with an HTTP request

We added an HTTP header

We added a TCP header

We added an IP header

Now what?

- From a protocol stack perspective, we need to determine if this layer can deliver it
- Well, can it?
- The local node isn't the destination
- The IP layer doesn't know how to relay itself
  - Not mechanically, at least
- So, no, it can't deliver it
- Better go down the stack

Moving down

So we've got a IP message to send, and this layer can't handle it

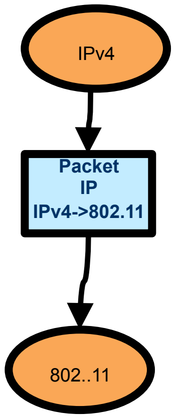
Find a lower layer that can help

IP

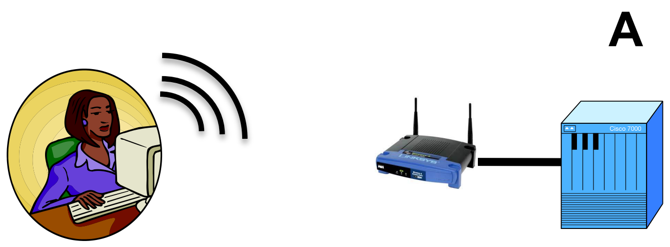
We know it has to go off node

What protocol layer can we go down to that might help?

The only one we have here is 802.11



Doing 802.11's work

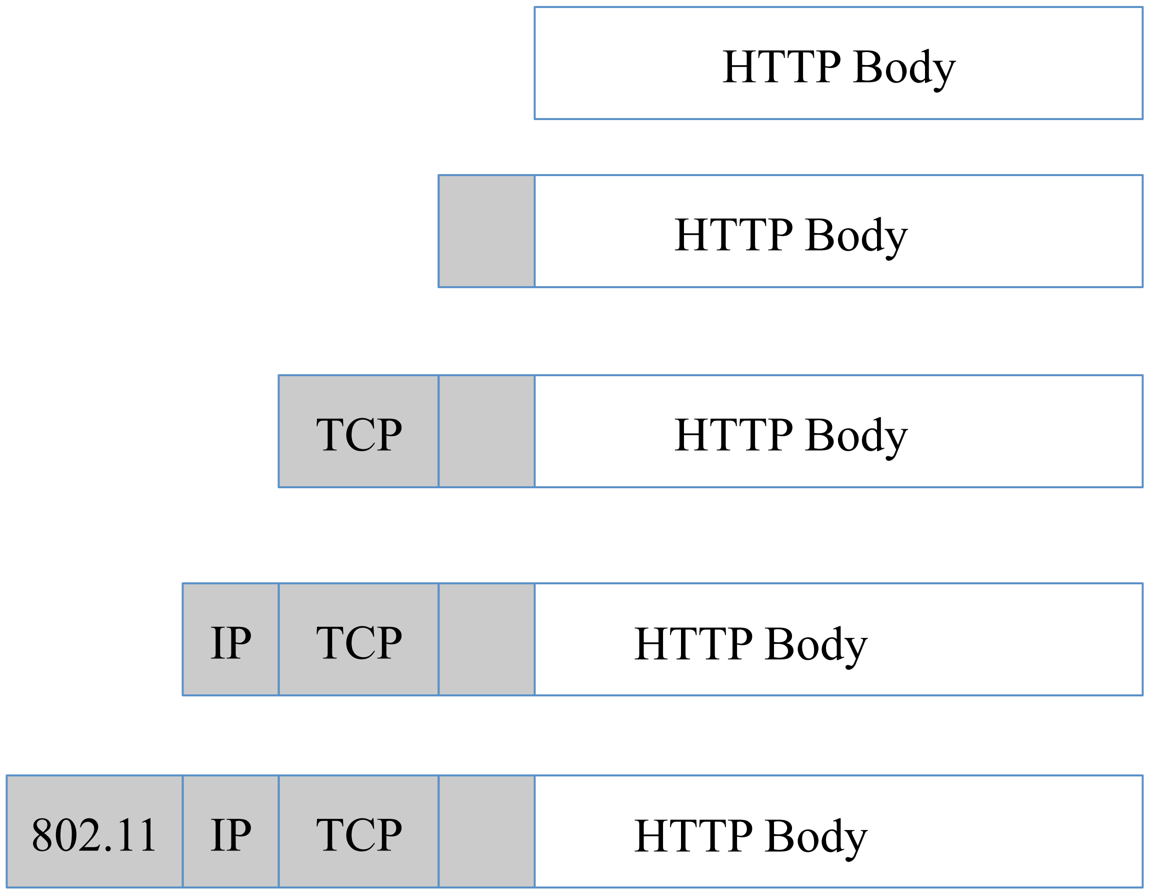


We need to translate the IP address into an 802.11 MAC address

We have a table that shows us that the MAC address of the access point is what we should use

Now we have to build an 802.11 packet

So what have we got so far?



We started with an HTTP request

We added an HTTP header

We added a TCP header

We added an IP header

Now we add an 802.11 header

Now what?

- From a protocol stack perspective, we need to determine if this layer can deliver it
- Well, can it?
- The local node isn't the destination
- The 802.11 link layer sort of knows how to deliver it
  - More precisely, it knows it can hand it to the wireless card and that will deliver it

Maybe not yet, though

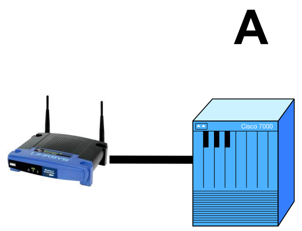
- Is the hardware ready?
- The wireless hardware might be busy already
- And any buffers it has might be full
- In which case, we can't pass the packet down yet
- But sooner or later, the hardware becomes free
- And we ship the packet off

One more wrinkle

Functionality specific to this link and this protocol layer

- The wireless medium is shared
- So we need to wait for the medium to be free
- Using (for 802.11) a CSMA/CA protocol
  - Listen for a little while
  - If no other transmission, then transmit the frame
- ACKs will tell us whether it got through
- Some (or all) of this link level protocol might be bundled into the hardware

The next step



Now we're at the wireless access point

Which, from a networking perspective, is a node

- So now we need to go up the stack at this node
  - Is this message for me?
  - Go up the stack to check
  - In particular, pop off the 802.11 header and check the IP header
  - OK, not for me

Now what?

- We must relay so down the stack again
- This time, we have a choice
  - Down to 802.11
  - Down to (say) DSL
- How do we choose?
- We have a table that tells us
- The IP address isn’t in the direction of the 802.11 link
- It’s in the direction of the DSL link

The DSL link

So we don’t need the functionality we required for the shared 802.11 channel.

- This is a point-to-point link
- Typically, a separate (frequency def d) channel in each direction
- So we’re treating this unidirectio al channel kind of like a Shannon channel
  - One sender, one receiver
  - Sender controls use of the channel
  - Receiver pulls out what sender puts in
  - Assuming no noise

Over the DSL link to router A

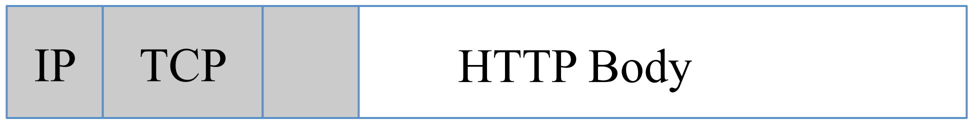
- Everything gets optimized at router A
  - We can't afford to go up and down stacks
- Instead, we simply strip off the DSL headers
- And treat what we've got as an IP packet
  - So it better be one
  - That's an implication of our narrow waist
- Consult a forwarding table and shoot it out the door to router B

Where did that table come from?

- How did router A know to send the packet to router B?
- Rather than push it back out to the wireless access point?
- Or send it to some other link?
- Router A has a forwarding table
- Where did that come from?
- From a routing protocol (like BGP)

Which lives off to the side and has its own protocol stack.

What's being moved?



The routers are moving this packet

- Leaving aside link level headers that they add and remove

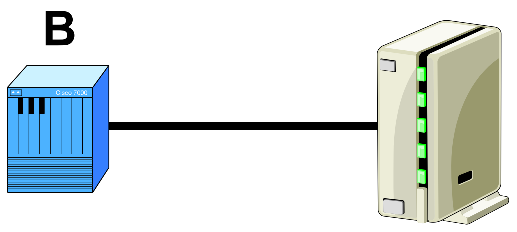
Usually moves through mostly unaltered

- Except for the TTL field in the IP header

Unless something special happens . . .

- Like IP fragmentation

Finishing the journey



The packet finally arrives at its destination

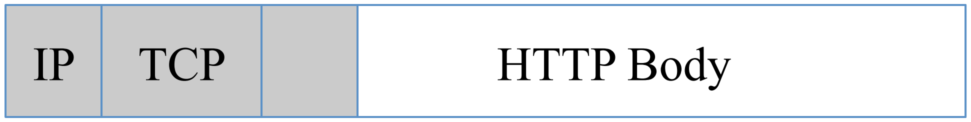
IP	TCP		HTTP Body

But we're not close to done

Up through the stack

- The destination examines the packet
- Is it for me?
- YES!
- OK, but there are multiple destinations within "me"
  - Remember internal addresses?
- Which one of those is it for?
- Well, better go up through the stack to see

Moving up



More layer-specific optimizations

We're done with the IP header, so get rid of it

OK, so it's a TCP packet

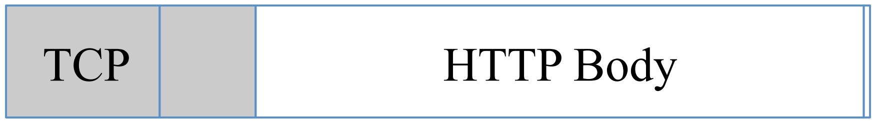
Now we do TCP processing (handle receive window, ACKs, etc.)

- Playing our part in TCP flow control, congestion control, and ordered delivery

What does the TCP header tell us to do next?

- Move up a protocol layer to HTTP

Continuing to move up



We're done with the TCP header, so get rid of it

Hand the message up to an HTTP protocol layer

    Probably implemented within a web server

That layer uses the HTTP header to handle networking issues

Then uses the body to deal with the actual business of providing a web page

And then?

- There are ACKs and responses
  - At various levels
  - Perhaps a link level ACK to the last router
  - Certainly a TCP ACK to the sender
    - No IP ACK – IP doesn't do ACKs
  - Almost certainly a message(s) containing the HTTP response
    - Or possibly an error response

The end-to-end principle is at work here

What else might be going on?

- The wireless channel might be used by other nodes
- Routers A and B might be handling other traffic
  - With different sources, destinations, or both
- Other sources might be sending messages to the destination
- The destination machine might be busy with other work
- The real world is very complicated

Are we happy with our own transmission?

- We got reliable, in-order delivery of packets representing HTTP requests
  - From one end of the network to the other
- We ensured flow control and handled congestion, if present
- What else *could* we want from our network?

Other things we might want

- We might want compression
- We might want encryption
- We might want quality of service guarantees
- We might want more control over routing
- And we might want many other things
- How might we get them?
- New protocol layers, mostly
- Composed with the existing ones

Summary

- Modern networks are complex
- Much of the field concerns managing that complexity
- Layering allows us to build protocols that compartmentalize complexity
- Recursive composition of layers allows us to handle heterogeneity in huge networks
- Layered protocols require many optimizations
- Which introduces yet more complexity