End-to-End

Spring 2024
cs168.io

Rob Shakir

Thanks to Murphy McCauley for some of the material!

Today

- Putting everything together…

- But let’s start with a gap we left – which is important for end-to-end operation – why do hosts sometimes get *private* addresses assigned when they need to access the Internet?

- Then - step through end-to-end operation!

NAT - network address translation.

Recall: IPv4 addressing.

- We have 2^{32} IPv4 addresses.
- Each *host* needs a unique IP address.
- We have a **lot** more than 2^{32} devices that need an IP address.
- You probably have at least two devices with you right now!

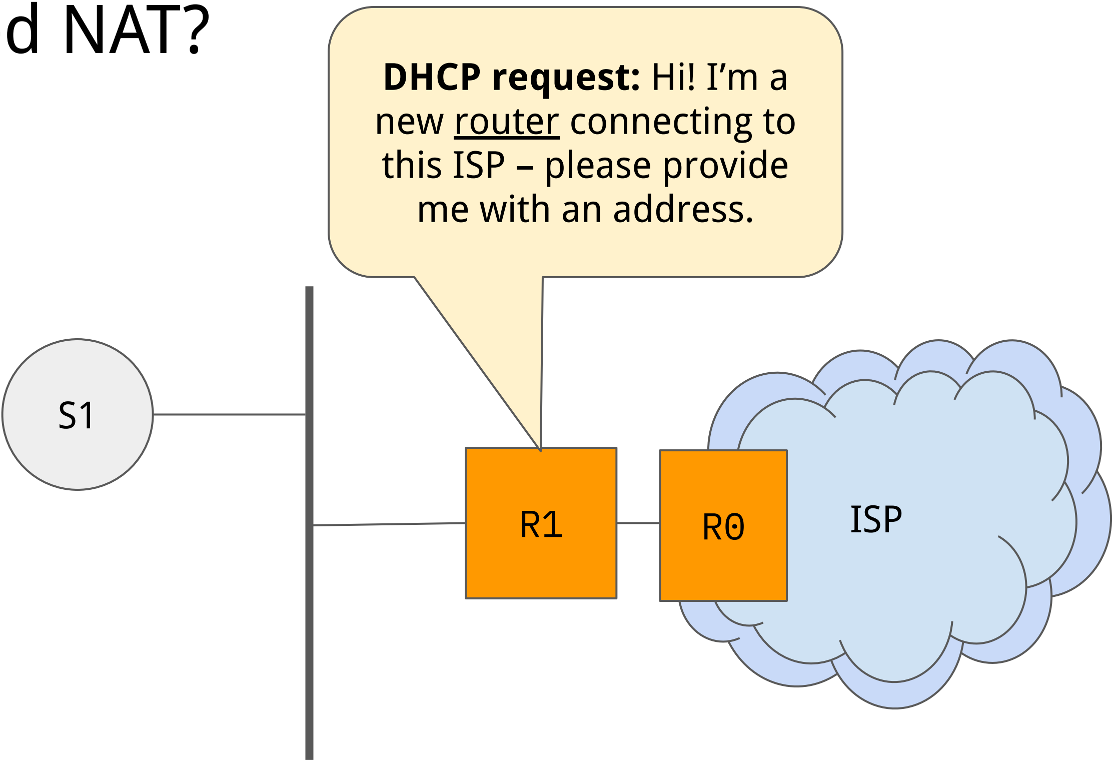
Recall: Private IP addresses.

- IANA allocated ranges for use in networks that don’t require Internet access.
  - 192.168.0.0/16
  - 10.0.0.0/8
  - 172.16.0.0/12

- You’ll see these addresses in use within your home network, and most networks that you connect to.

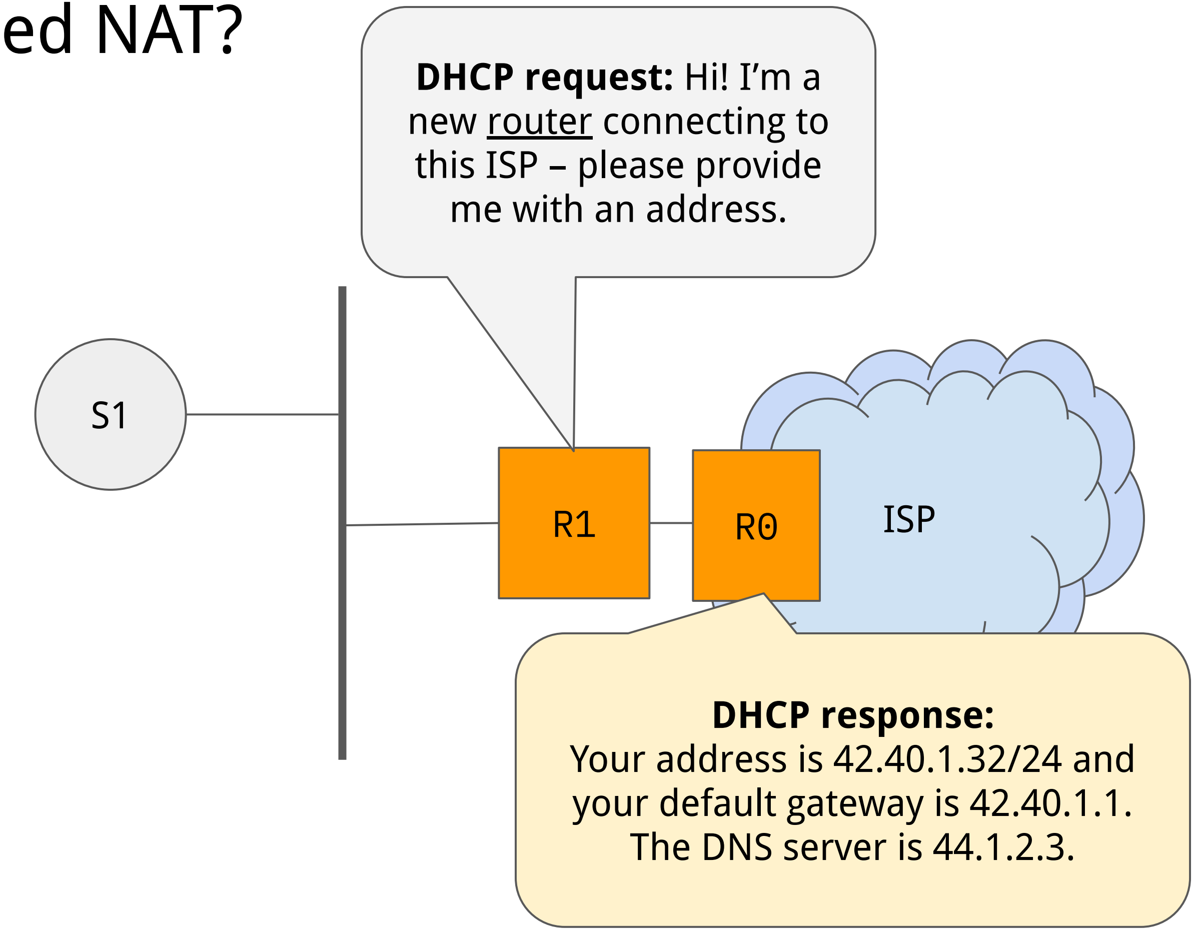
- But… you do need Internet access!

Why do we need NAT?



DHCP request: Hi! I’m a new router connecting to this ISP – please provide me with an address.

Why do we need NAT?

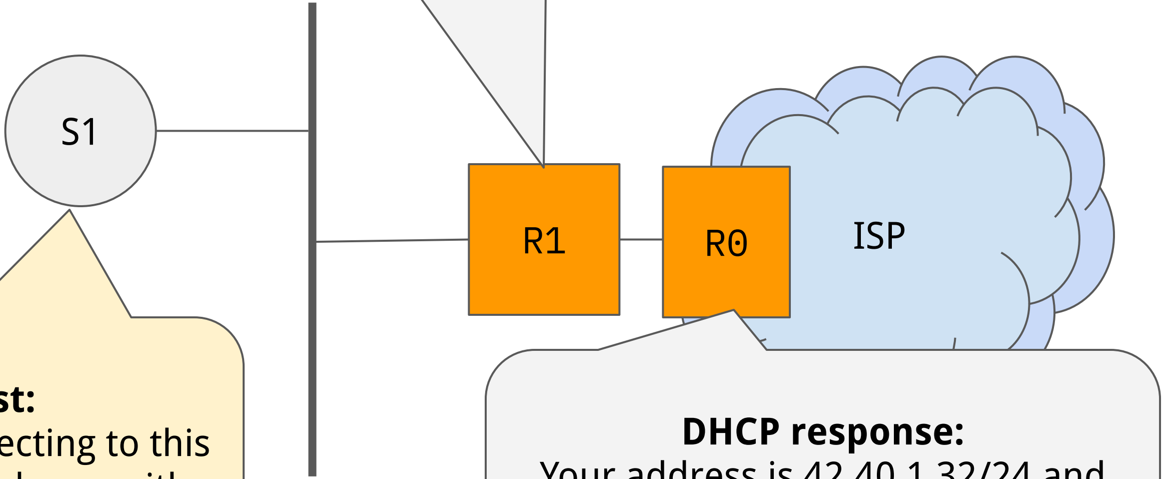


DHCP request: Hi! I’m a new router connecting to this ISP – please provide me with an address.

DHCP response:
Your address is 42.40.1.32/24 and your default gateway is 42.40.1.1.
The DNS server is 44.1.2.3.

Why do we need NAT?

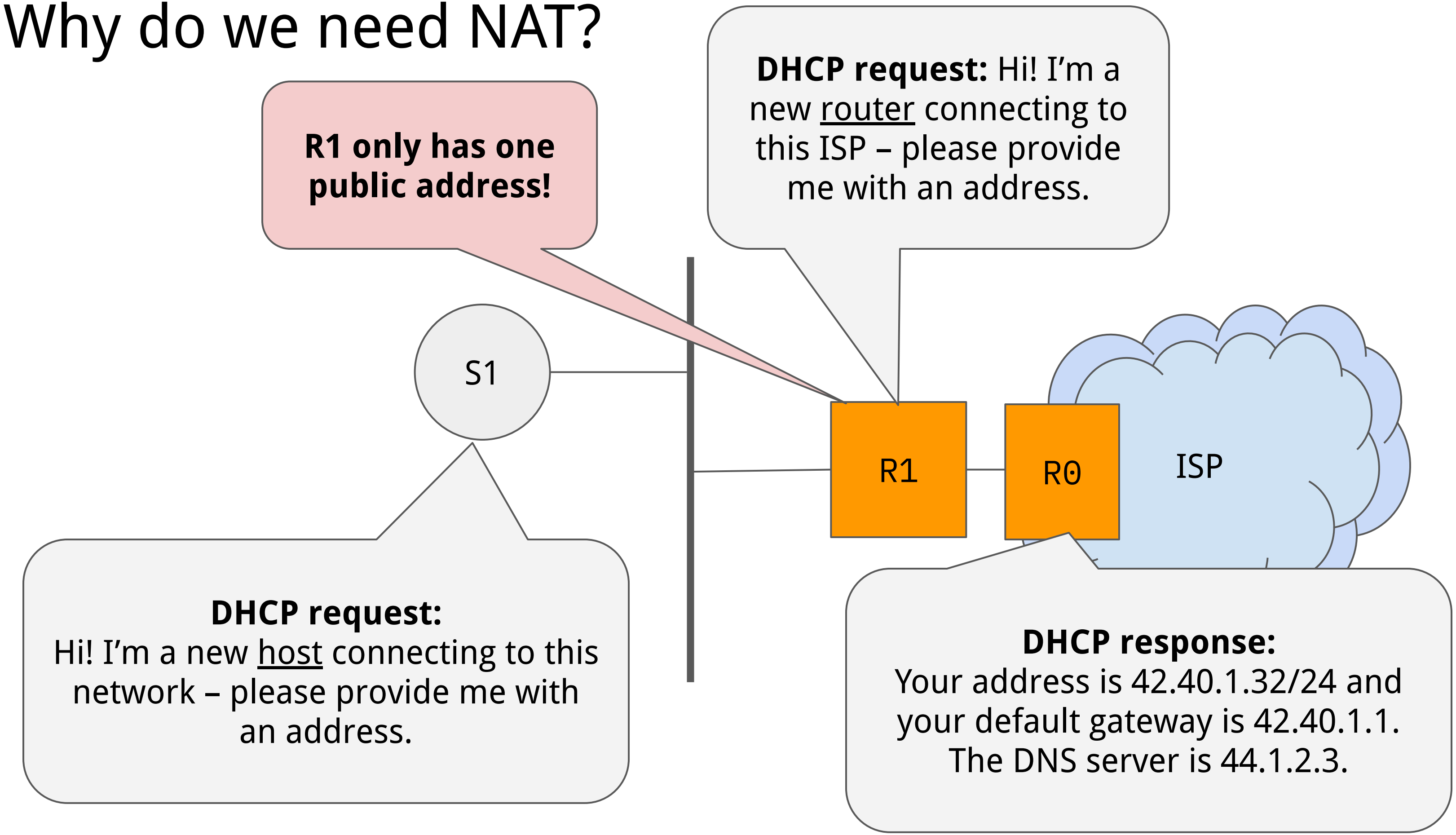
DHCP request: Hi! I'm a new router connecting to this ISP – please provide me with an address.



DHCP request:
Hi! I'm a new host connecting to this network – please provide me with an address.

DHCP response:
Your address is 42.40.1.32/24 and your default gateway is 42.40.1.1. The DNS server is 44.1.2.3.

Why do we need NAT?



R1 only has one public address!

DHCP request: Hi! I’m a new router connecting to this ISP – please provide me with an address.

DHCP request:
Hi! I’m a new host connecting to this network – please provide me with an address.

DHCP response:
Your address is 42.40.1.32/24 and your default gateway is 42.40.1.1. The DNS server is 44.1.2.3.

Sharing an External IP Address

- **Idea**: can we have a way to be able to use private IP addresses within a network – but share a single Internet-accessible IP address?

- Network Address Translation (NAT) – Port Address Translation.
  - Provides a way to have many hosts share a single public address.
  - Requires us to introduce Layer 4 awareness on routers that are "hiding" multiple hosts behind them.

- Different modes of NAT:
  - Sharing IP addresses requires port address translation (PAT).
  - Simpler modes of NAT that allow 1:1 address translation – e.g., `10.0.0.1` → `42.0.2.1` and `10.0.0.2` → `42.0.2.2`.
  - PAT is the most complex, and widely used NAT mode.

Questions?

How does NAT work?

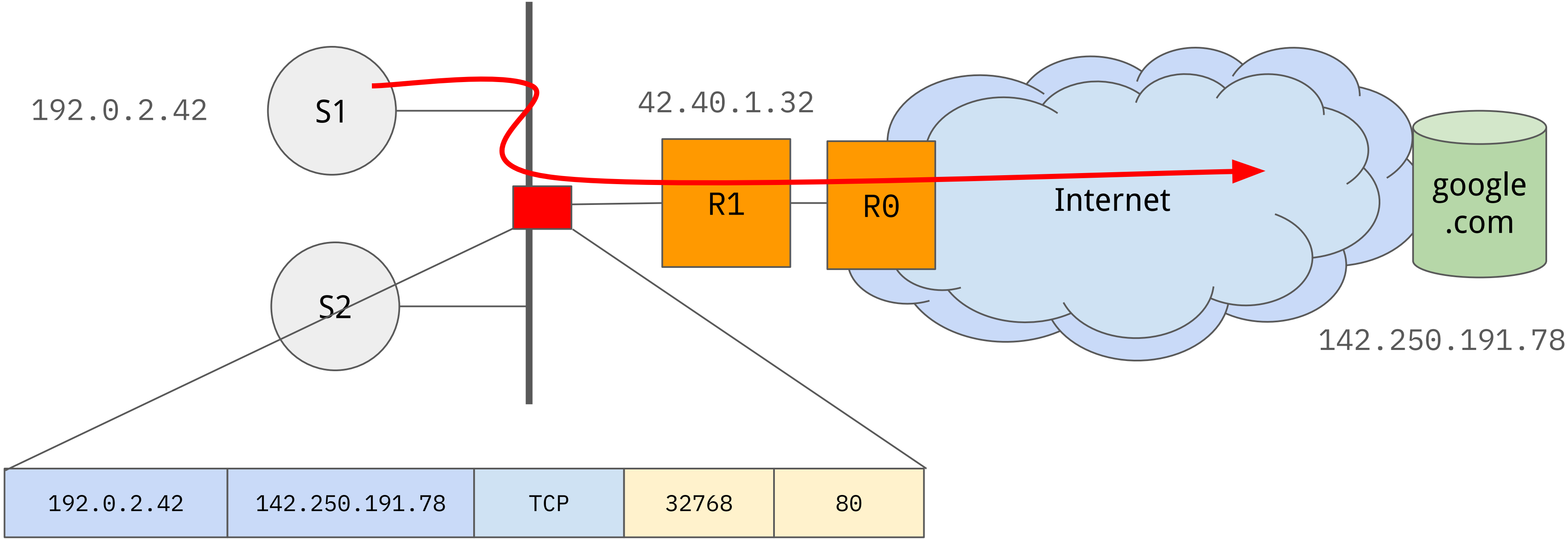
- Host A sends a packet to a destination address on the Internet, which has a specific Layer 4 protocol with source and destination port.

- The router performing NAT (doing translation) keeps state of the L4 (TCP, UDP) connections that are in flight.

- The combination of `<IP src, IP dst, protocol, TCP/UDP src port, TCP/UDP dst port>` uniquely identifies a connection.

Operation of NAT

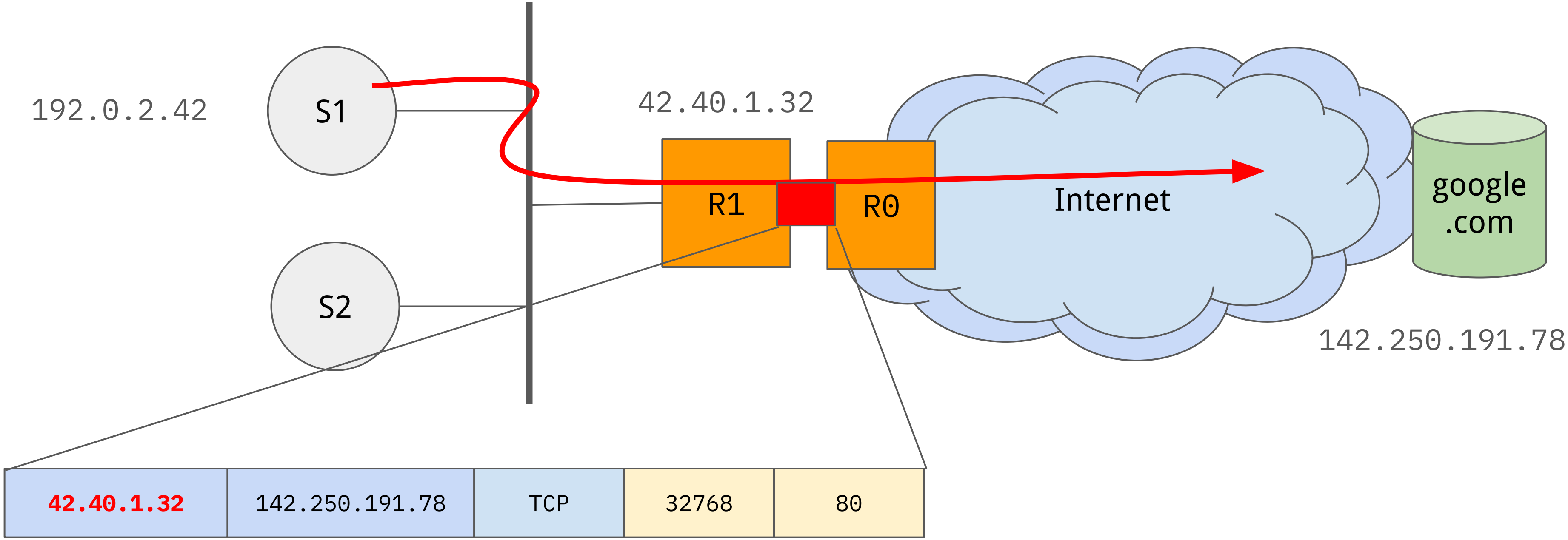
- Source host is unaltered – builds a packet as usual.



192.0.2.42	142.250.191.78	TCP	32768	80

Operation of NAT

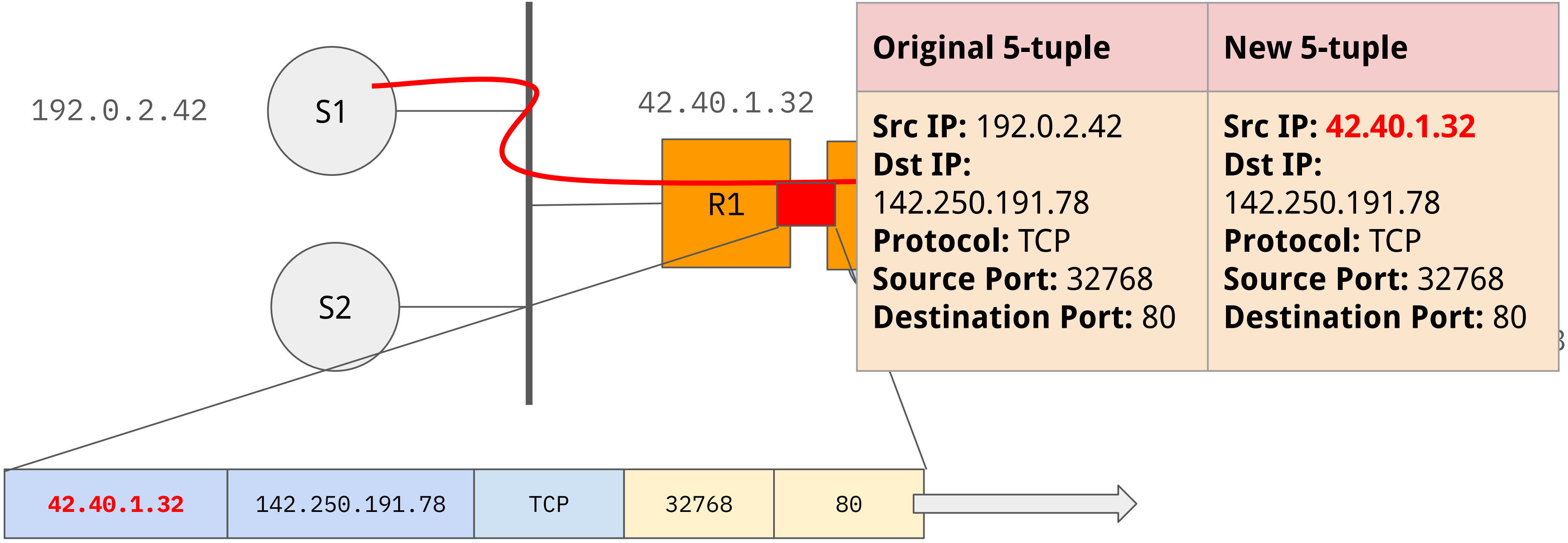
- Router performing NAT changes the source IP address to the public address.



42.40.1.32	142.250.191.78	TCP	32768	80

Operation of NAT

- Router performing NAT stores <u>state</u> for the connection.

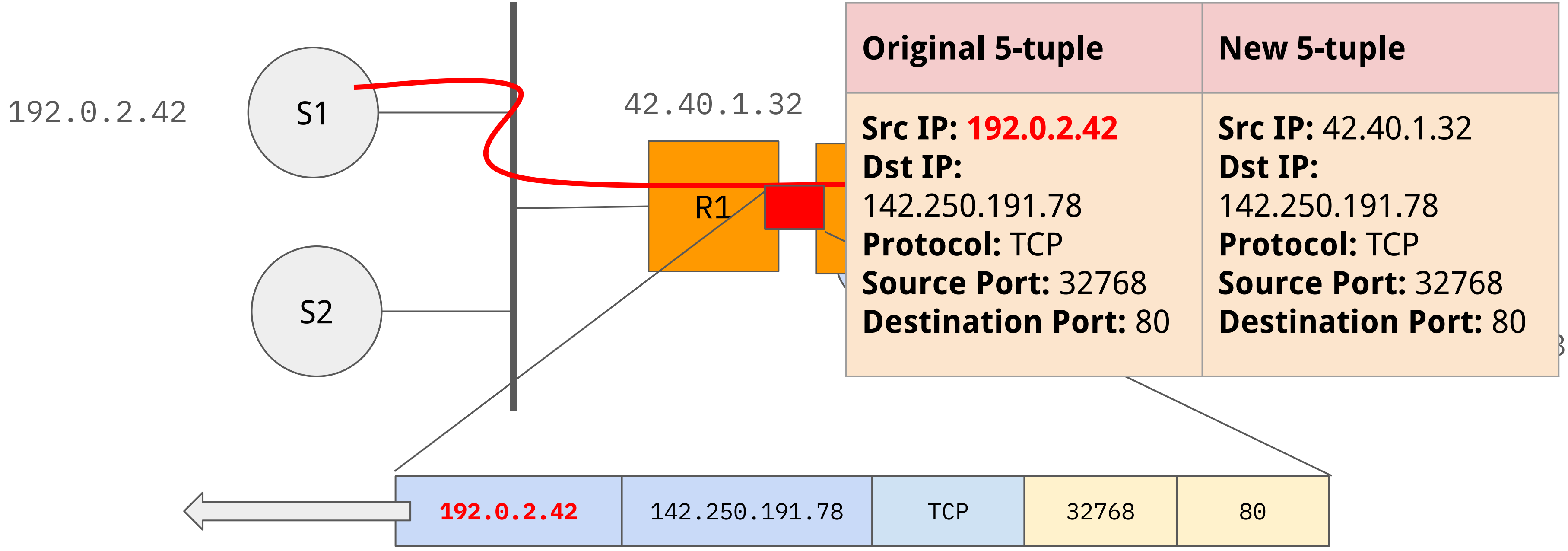


Original 5-tuple	New 5-tuple
Src IP: 192.0.2.42 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 32768 **Destination Port:** 80	**Src IP:** 42.40.1.32 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 32768 **Destination Port:** 80

42.40.1.32	142.250.191.78	TCP	32768	80

Operation of NAT

- Return packets are mapped back using the NAT table.

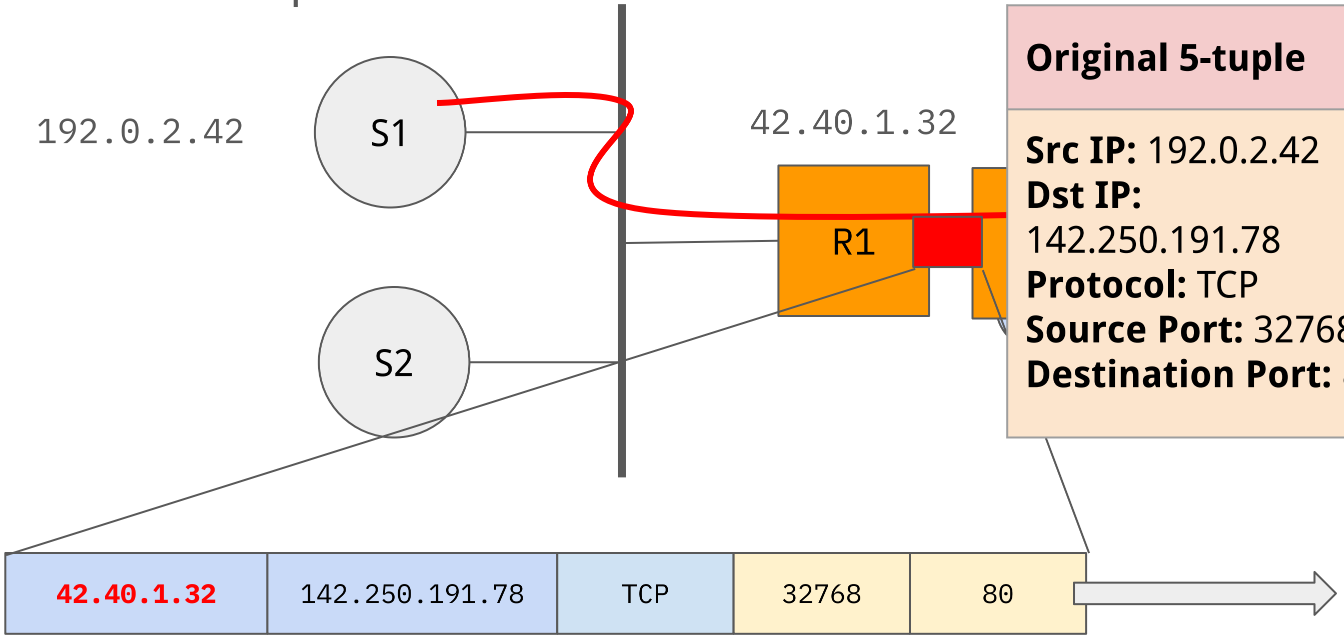


Original 5-tuple	New 5-tuple
Src IP: 192.0.2.42 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 32768 **Destination Port:** 80	**Src IP:** 42.40.1.32 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 32768 **Destination Port:** 80

192.0.2.42	142.250.191.78	TCP	32768	80

Modifying TCP source port…

- What happens if we have two flows to the same destination with the same source port?

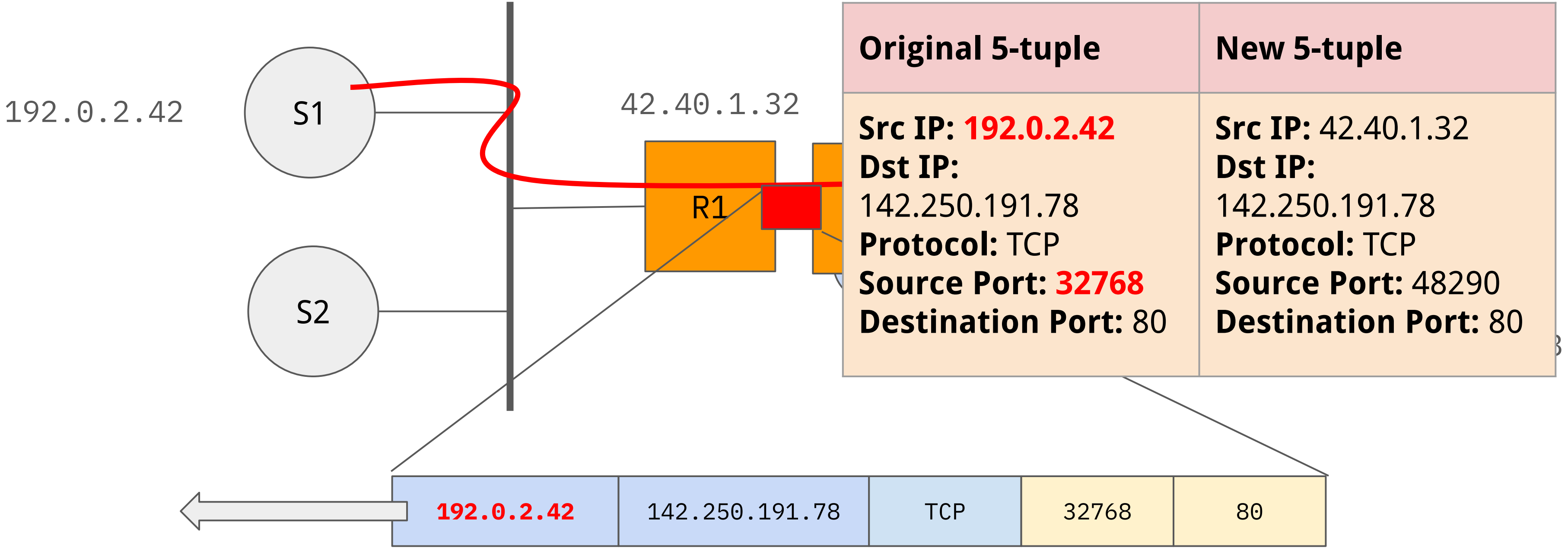


42.40.1.32	142.250.191.78	TCP	32768	80

Original 5-tuple	New 5-tuple
Src IP: 192.0.2.42 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 32768 **Destination Port:** 80	**Src IP:** 42.40.1.32 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 48290 **Destination Port:** 80

Operation of NAT

- Return packets are mapped back using the NAT table.



Original 5-tuple	New 5-tuple
Src IP: 192.0.2.42 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 32768 **Destination Port:** 80	**Src IP:** 42.40.1.32 **Dst IP:** 142.250.191.78 **Protocol:** TCP **Source Port:** 48290 **Destination Port:** 80

Questions?

NAT: Expectation of Routers

- Ability to modify packets as they are forwarded:
    - IP source address – hides the private addressing in the network.
    - TCP source port – handles the case where two different hosts are contacting the same remote address (e.g., `google.com:80`).

- A connection state table – means the router performing NAT must be aware of each TCP and UDP flow that are traversing the router.

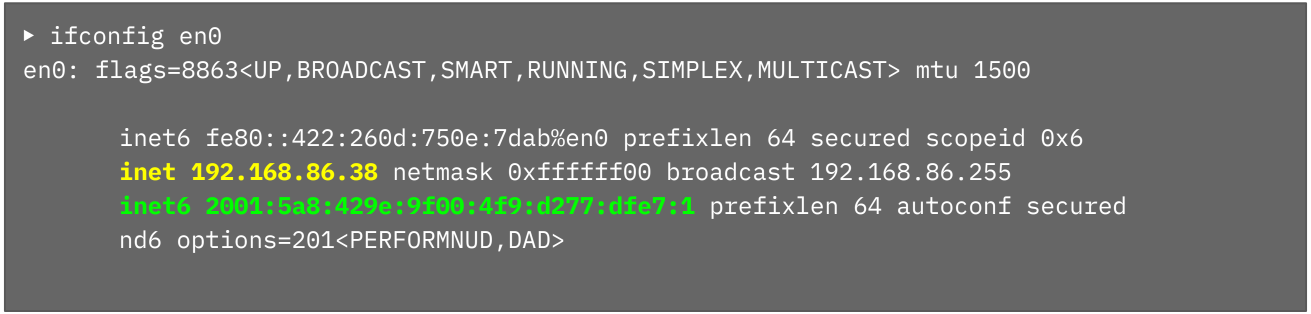
- This increases the complexity of packet forwarding – needs more cycles, and memory for each flow.
    - NAT is therefore performed as close to the edge of the network as possible (at your home router!).

Where is NAT used?

- Small scale NAT used in almost every network for IPv4.

- As IPv4 addresses ran out – ISPs did not have enough IPv4 addresses for each customer → Carrier Grade NAT (CGNAT).
    - More complex – many more connections to maintain state for.

- NAT is generally not used for IPv6 – there are enough addresses!



```
▸ ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500

        inet6 fe80::422:260d:750e:7dab%en0 prefixlen 64 secured scopeid 0x6
        inet 192.168.86.38 netmask 0xffffff00 broadcast 192.168.86.255
        inet6 2001:5a8:429e:9f00:4f9:d277:dfe7:1 prefixlen 64 autoconf secured
        nd6 options=201<PERFORMNUD,DAD>
```

NAT: Inbound Connections

- When a router performing NAT gets an inbound connection – where does it send it?
  - Who is a connection to public address `42.40.1.32` port `80` destined to?

- Breaks the end-to-end principle!

- In order to allow inbound connections, routers performing NAT need port mapping tables.
  - May be statically specified.
  - Dynamic protocols such as UPnP (Universal Plug-n-Play) and NAT-PMP (NAT Port Mapping Protocol) allow dynamic configuration of open ports.

Interesting Implications of NAT

- Since NAT breaks the end-to-end principle it is sometimes thought of as security.
  - Essentially a firewall that by default drops all inbound connections.
  - This is a side effect – and really doesn't implement a principled security policy.

- Where we don't use NAT – client privacy might be impacted.
  - Always see the end machine's IP address at a remote server.
  - Particularly where it is dynamically configured based on the MAC address (IPv6 EUI64) this is identifiable down to a particular computer.
  - NAT would hide this identity.
  - Alternate solutions like IPv6 temporary/privacy addresses.

Questions?

Putting it all together – End-to-End.

End-to-End Connectivity

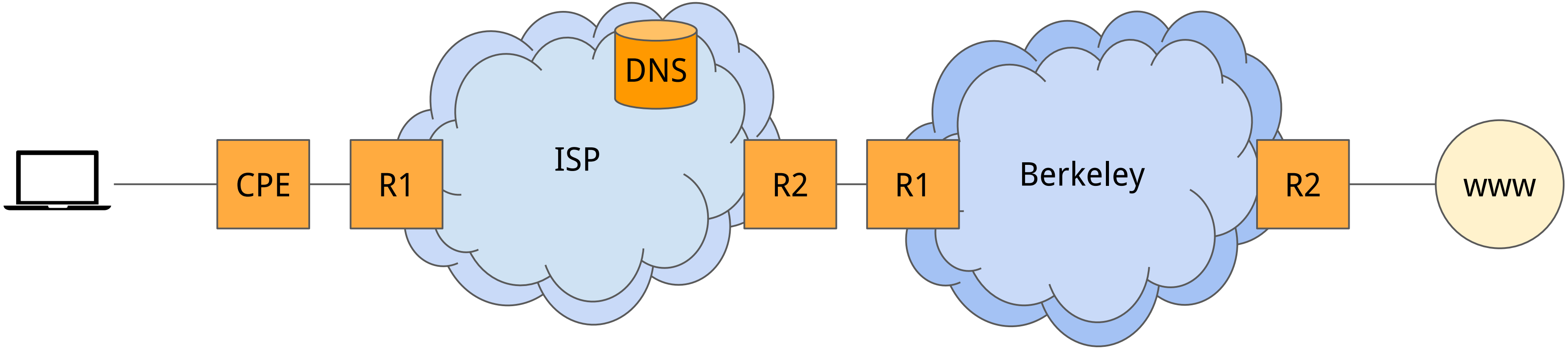
- We’ve been working towards this all semester!

- Let’s think about what happens when we:
  - Turn on our computer and plug it into an Ethernet network
  - Type `berkeley.edu⏎` into our browser .

- We’ll assume that we don’t need to turn the Internet on from scratch!

Scenario

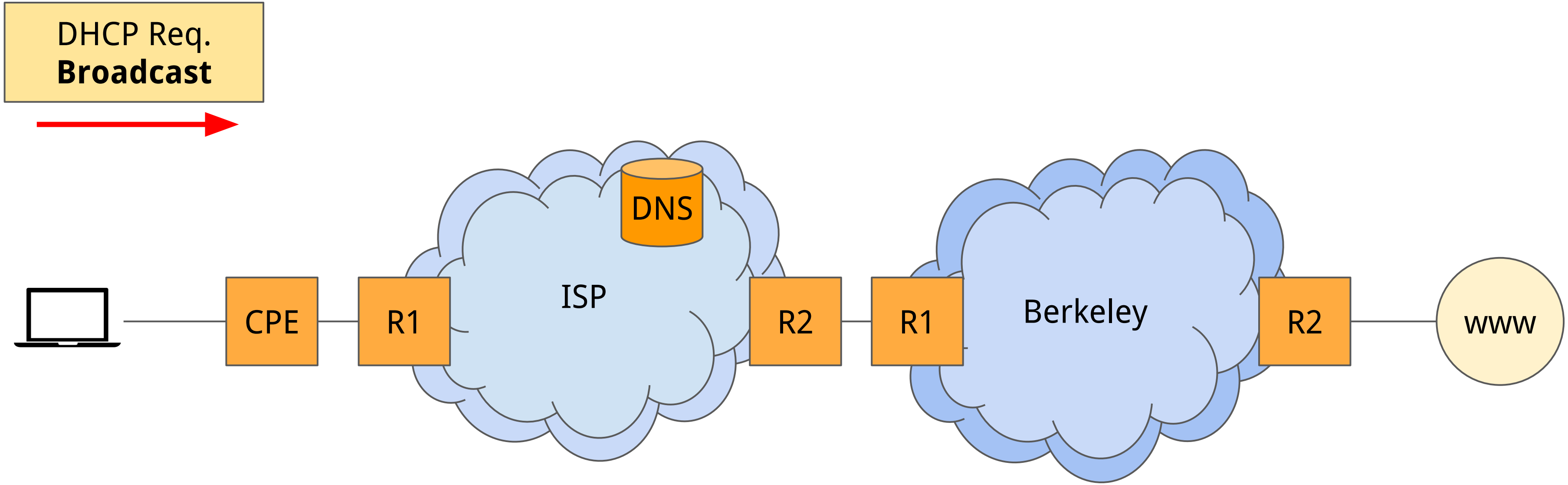
- Let’s think about the following end-to-end network.
- Our host (laptop) connected to an ISP network, that is accessing www.berkeley.edu.



Please interrupt me if you have questions as we step through!

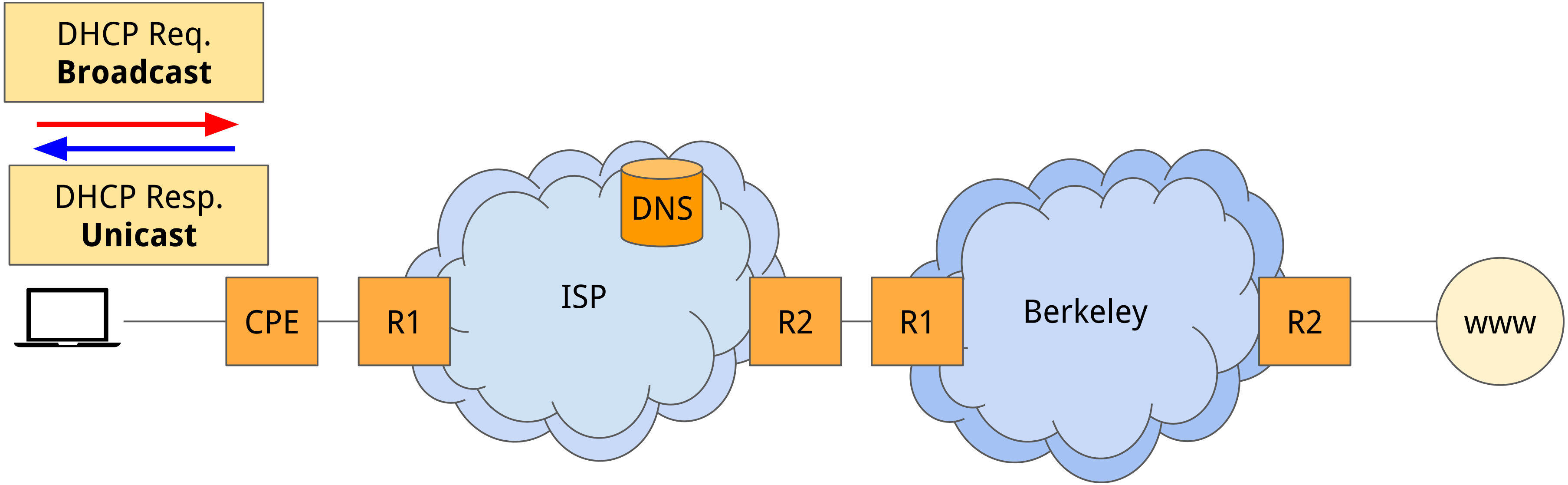
DHCP

- We connect to an Ethernet network and make a DHCP request.
- The DHCP server responds with an IP address, subnet mask, default gateway and DNS server.



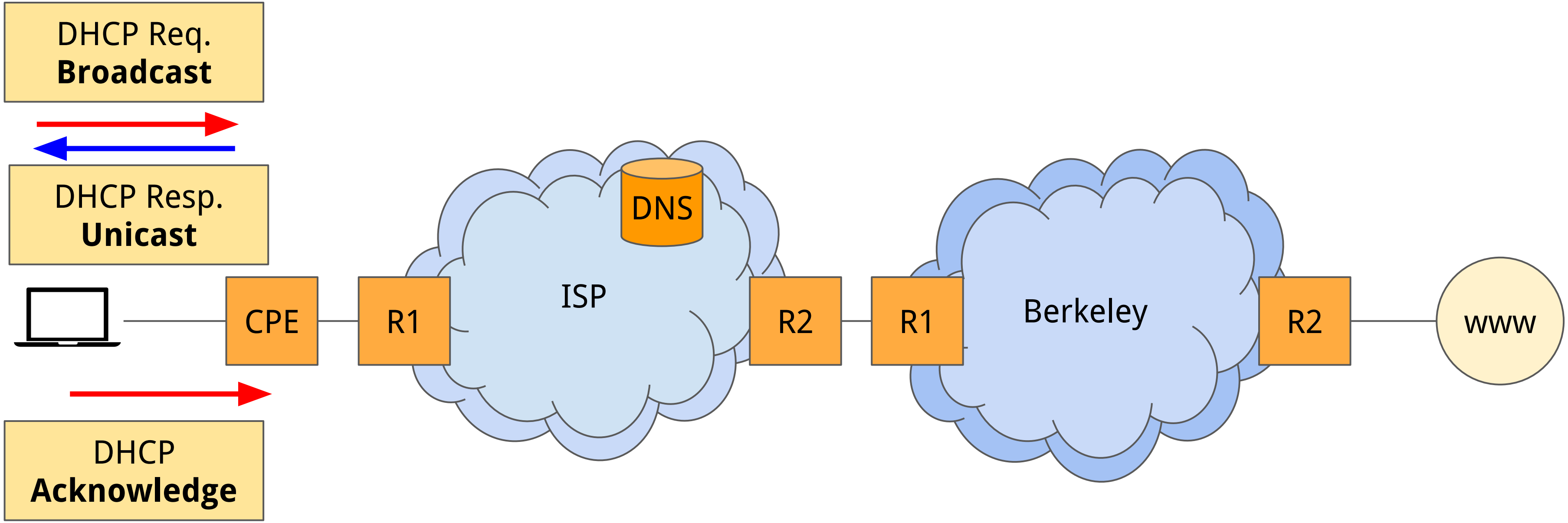
DHCP

- We connect to an Ethernet network and make a DHCP request.
- The DHCP server responds with an IP address, subnet mask, default gateway and DNS server.



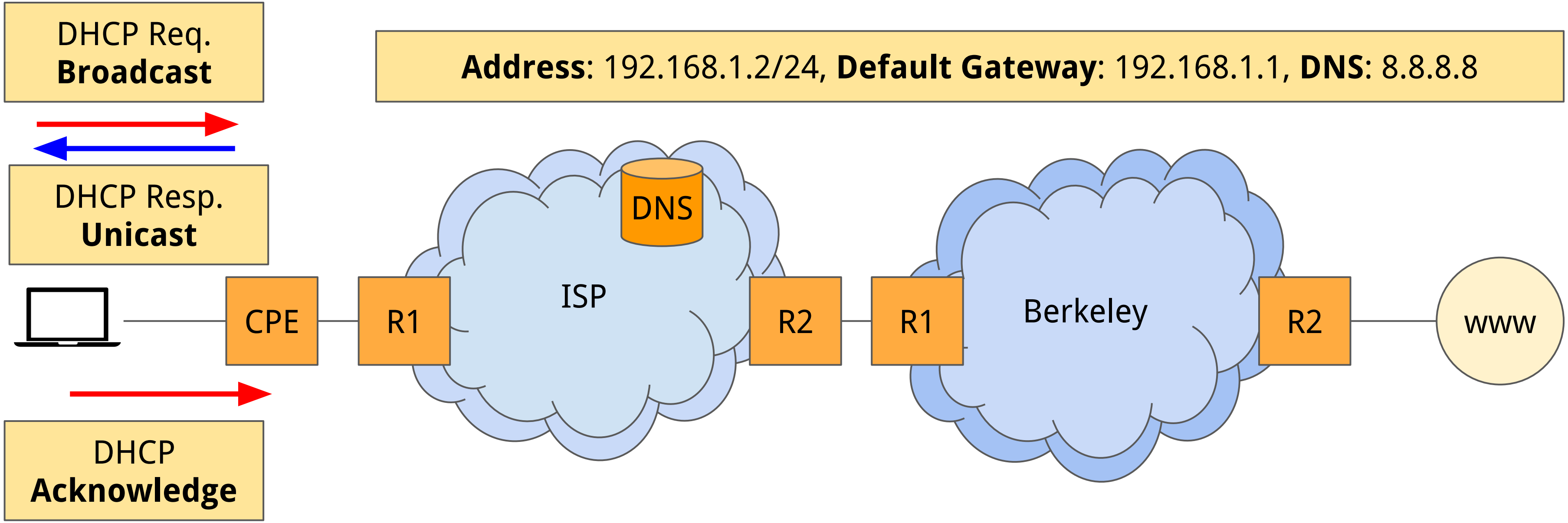
DHCP

- We connect to an Ethernet network and make a DHCP request.
- The DHCP server responds with an IP address, subnet mask, default gateway and DNS server.



DHCP

- We connect to an Ethernet network and make a DHCP request.
- The DHCP server responds with an IP address, subnet mask, default gateway and DNS server.



DHCP Req. **Broadcast**

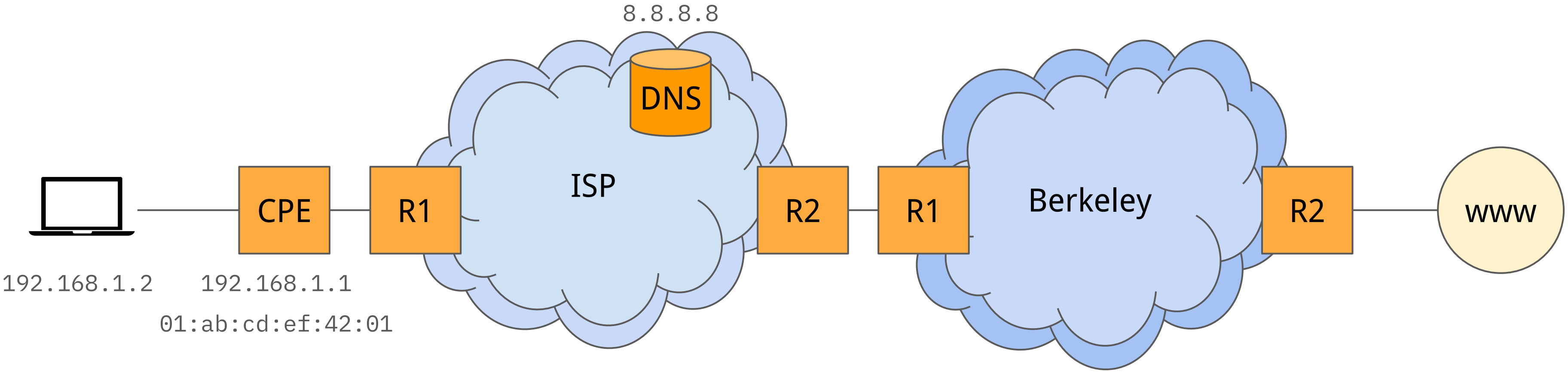
DHCP Resp. **Unicast**

DHCP **Acknowledge**

Address: 192.168.1.2/24, **Default Gateway**: 192.168.1.1, **DNS**: 8.8.8.8

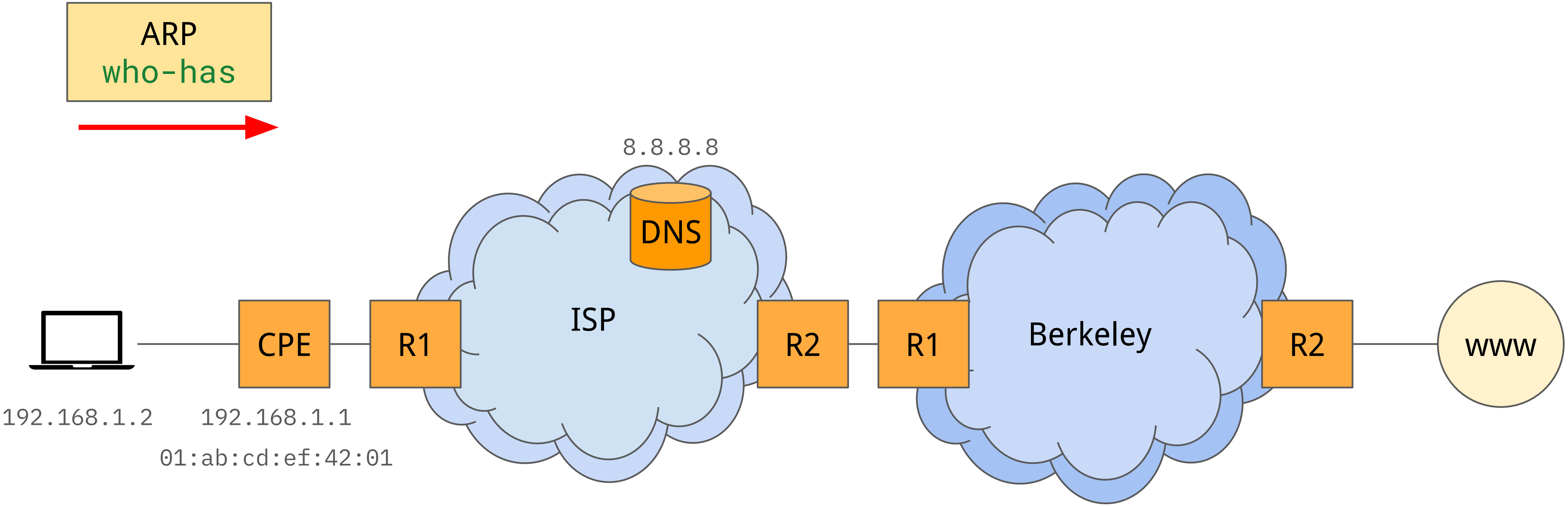
Starting Our Connection

- Now – we want to look up www.berkeley.edu.
  - DNS: Browser makes `getaddrinfo` call to resolve.
- Kernel: Our DNS server is `8.8.8.8` - which is not local!
  - Host routing table says `0.0.0.0/0` → `192.168.1.1`.



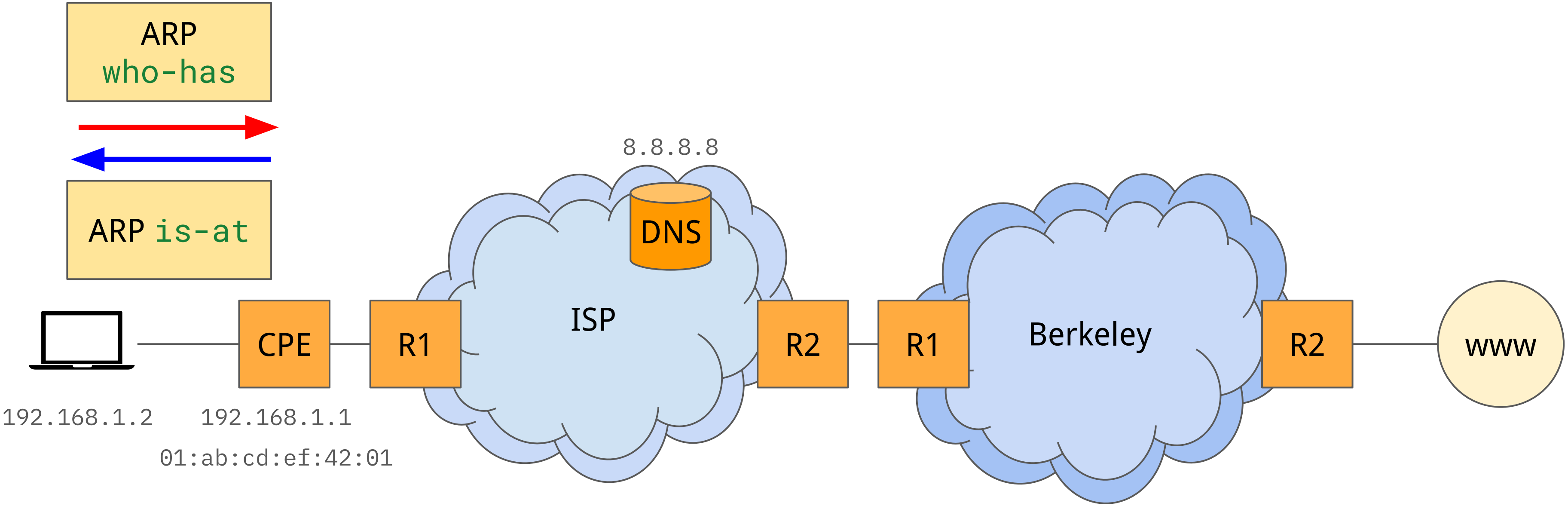
ARP

- We need to know where 192.168.1.1 is on our local network – ARP!
- Again, in the kernel – browser does not need to know about this layer.



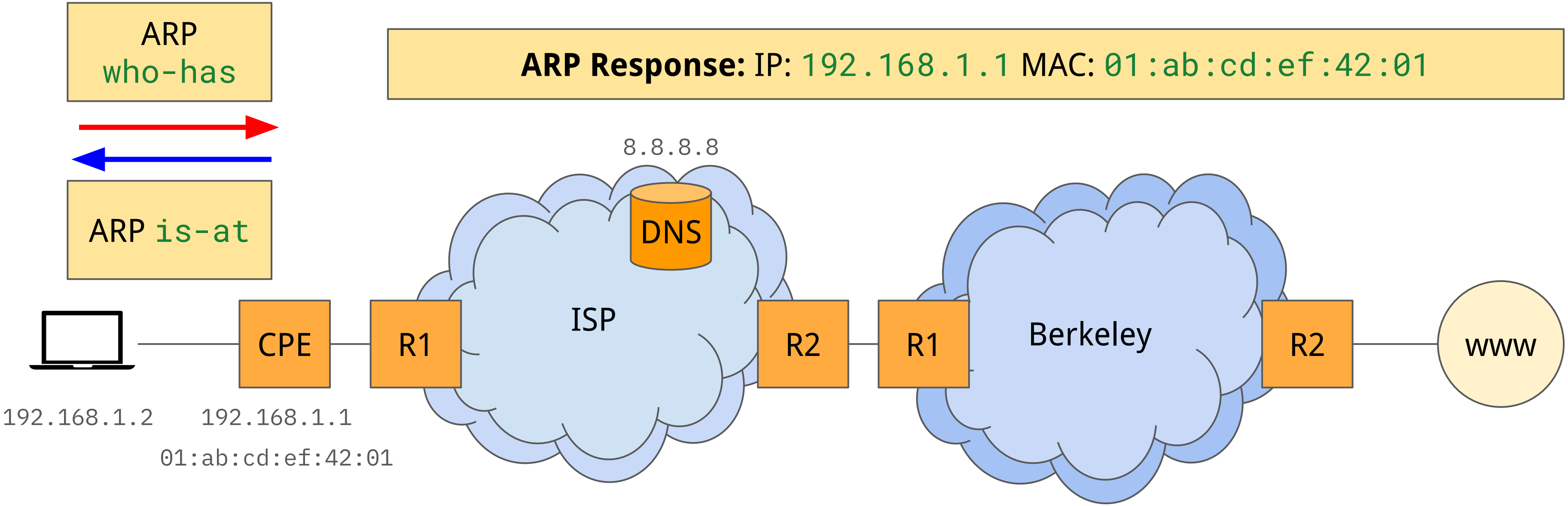
ARP

- We need to know where 192.168.1.1 is on our local network – ARP!
- Again, in the kernel – browser does not need to know about this layer.



ARP

- We need to know where 192.168.1.1 is on our local network – ARP!
- Again, in the kernel – browser does not need to know about this layer.

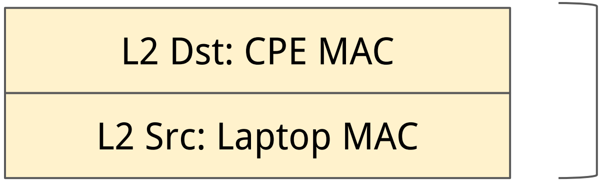


ARP who-has

ARP is-at

ARP Response: IP: 192.168.1.1 MAC: 01:ab:cd:ef:42:01

Building Up Packets: L2

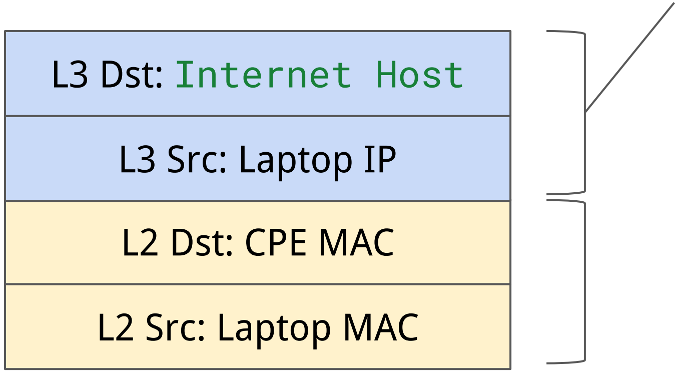


L2 Dst: CPE MAC

L2 Src: Laptop MAC

All subsequent packets that are being sent by the host towards the Internet use the same Layer 2 source and destination values.

Building Up Packets: L3

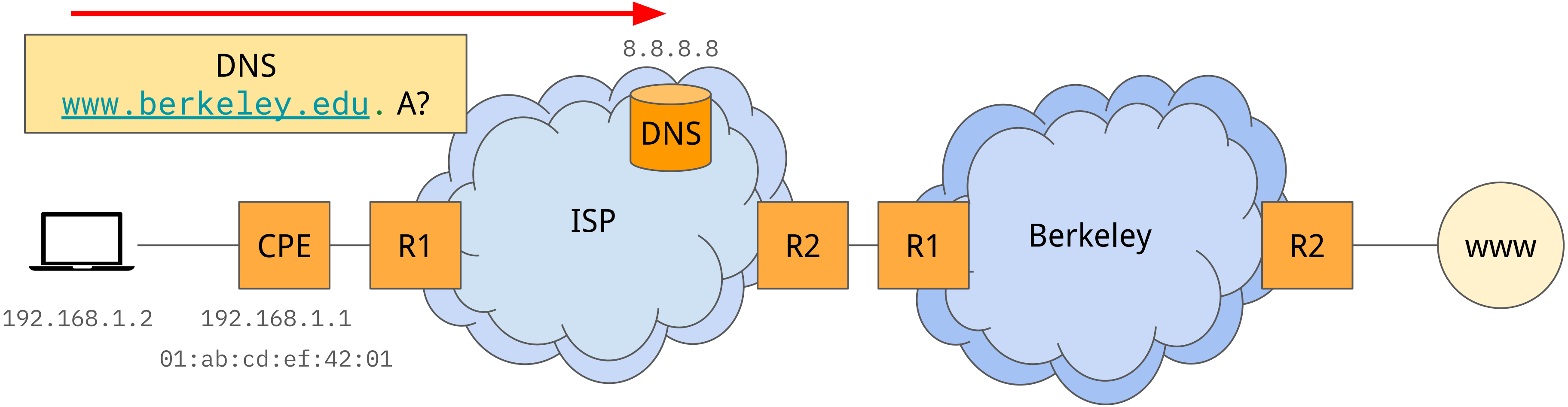


- Layer 3 (IPv4) source uses the address assigned by DHCP.
- Layer 3 (IPv4) destination is either known statically or resolved via DNS.
- NAT may rewrite the L3 headers in flight – but the host never sees this.

- Fixed L2 headers

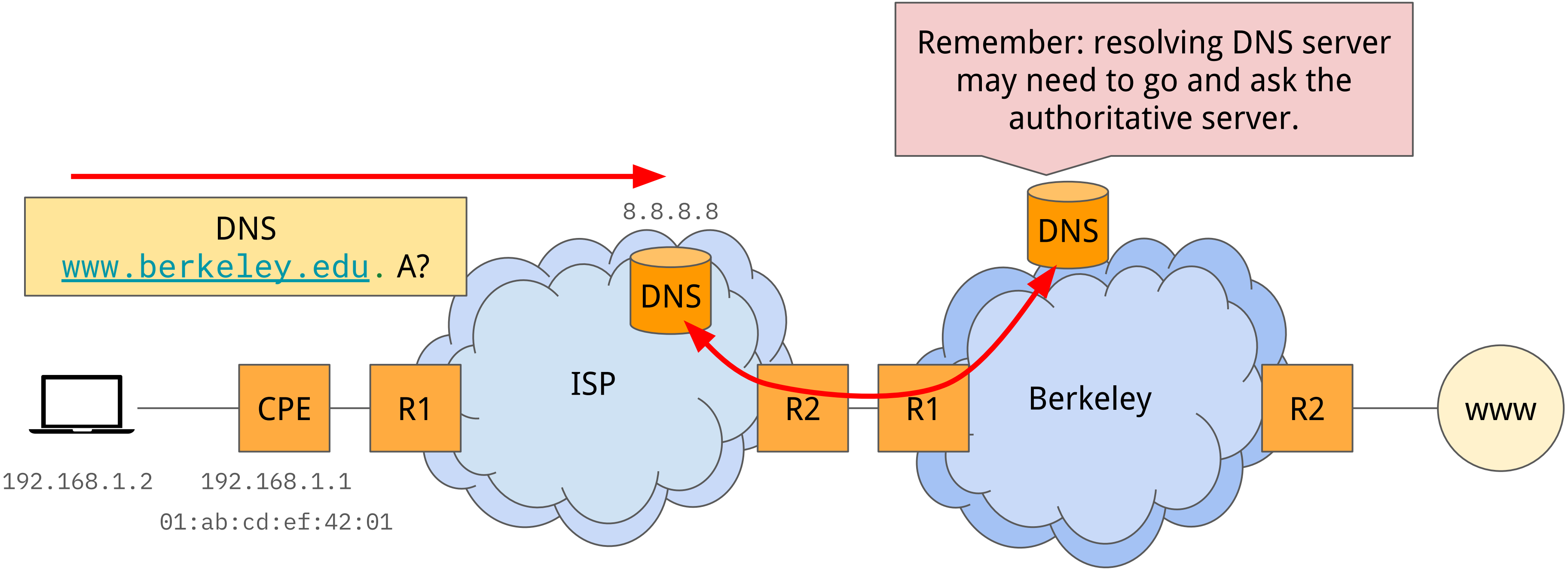
DNS

- We are now able to make a DNS request – sending a packet to 8.8.8.8.



DNS

- We are now able to make a DNS request – sending a packet to `8.8.8.8`.

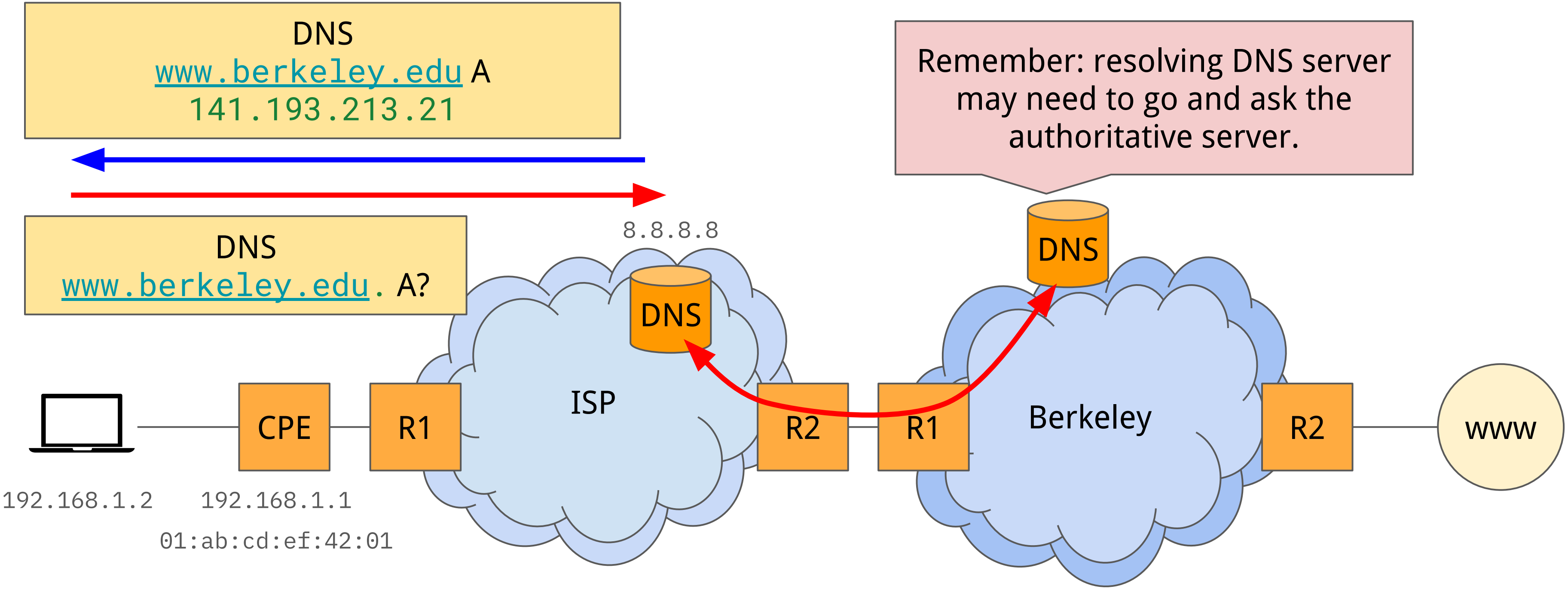


Remember: resolving DNS server may need to go and ask the authoritative server.

DNS
www.berkeley.edu. A?

DNS

- We are now able to make a DNS request – sending a packet to 8.8.8.8.



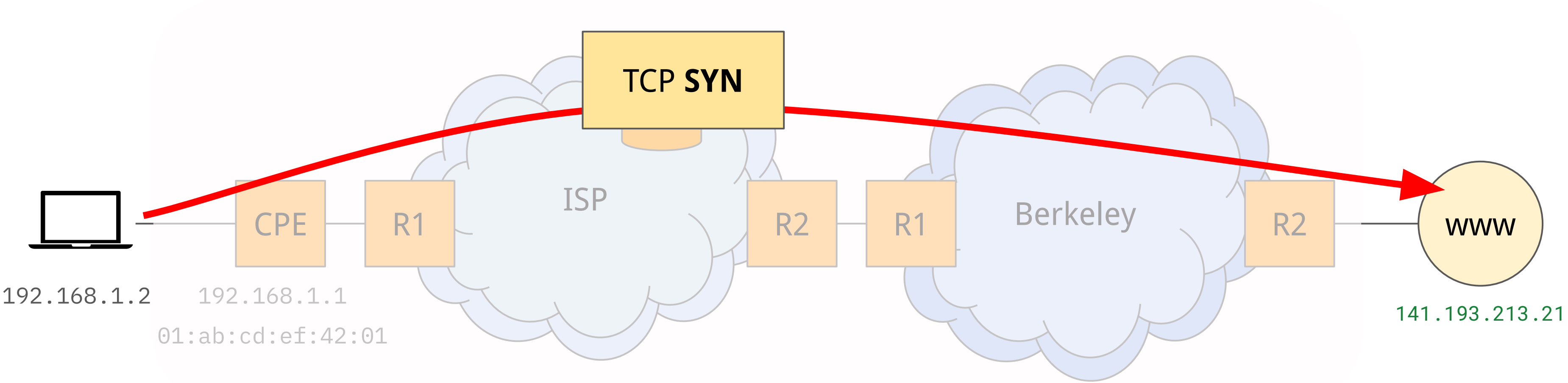
DNS
www.berkeley.edu A
141.193.213.21

DNS
www.berkeley.edu. A?

Remember: resolving DNS server may need to go and ask the authoritative server.

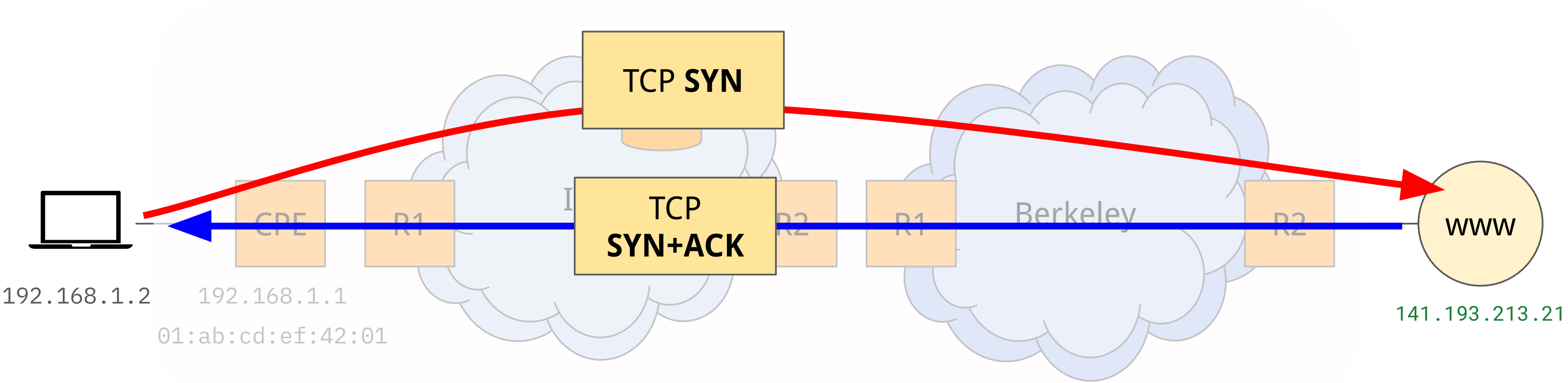
TCP

- TCP three-way handshake required to initiate a TCP connection to www.berkeley.com:80 (well known HTTP port).
- Established by the browser by calling connect on a particular socket to the address returned by getaddrinfo.



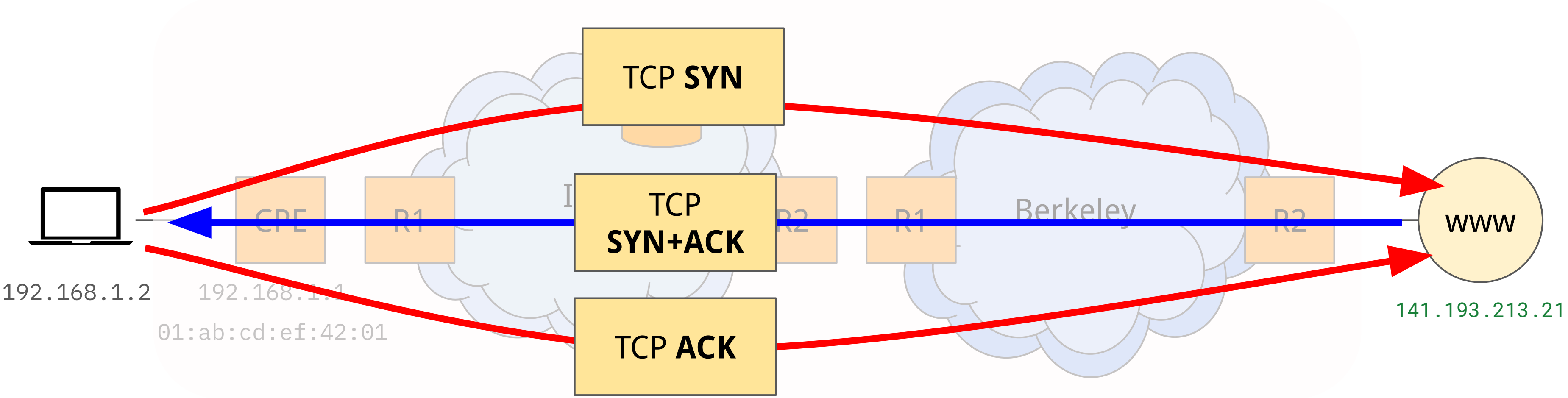
TCP

- TCP three-way handshake required to initiate a TCP connection to www.berkeley.com:80 (well known HTTP port).
- Established by the browser by calling connect on a particular socket to the address returned by getaddrinfo.



TCP

- TCP three-way handshake required to initiate a TCP connection to `www.berkeley.com:80` (well known HTTP port).
- Established by the browser by calling `connect` on a particular `socket` to the address returned by `getaddrinfo`.



A note on TCP piggybacking

- We assume there is no TCP piggybacking – i.e., no data being included when an `ACK` is returned.

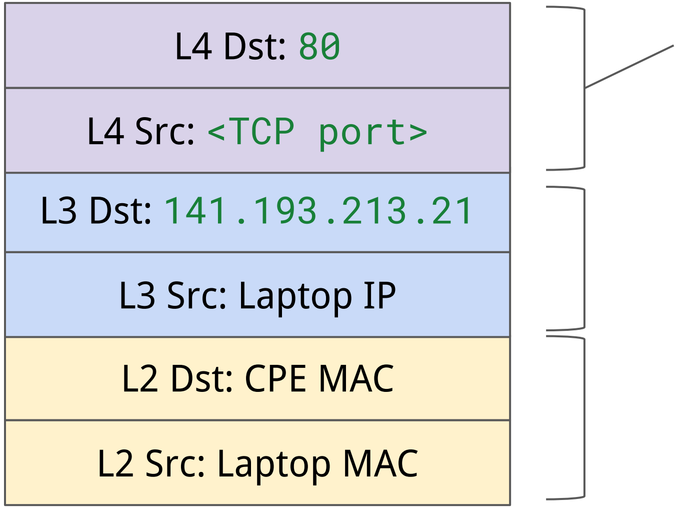
- TCP implementation is typically in the kernel.
  - `ACK`s are generated in the kernel.

- Applications (HTTP etc.) are in userspace.
  - Application responses are generated in userspace.

- `ACK`s are often generated before the application has a chance to respond.
  - Kernel creates `ACK` and *schedules* the application to run.

- `SYN-ACK` is the only case where we have information “piggybacked” on `ACK`s.
  - Because this is done in the kernel.

Building Up Packets: L4



L4 Dst: 80

L4 Src: <TCP port>

L3 Dst: 141.193.213.21

L3 Src: Laptop IP

L2 Dst: CPE MAC

L2 Src: Laptop MAC

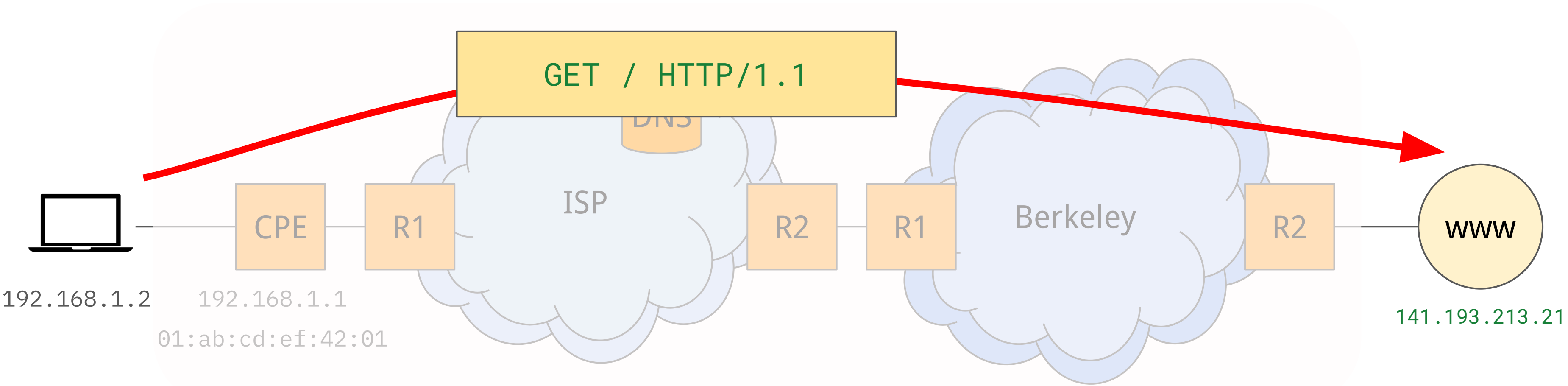
- L4 destination port is based on the application being used – 80 for HTTP,
- Source port is either developer specified or ephemerally allocated by the kernel (developer specifies :0).

- L3 headers to reach berkeley.edu

- Fixed L2 headers

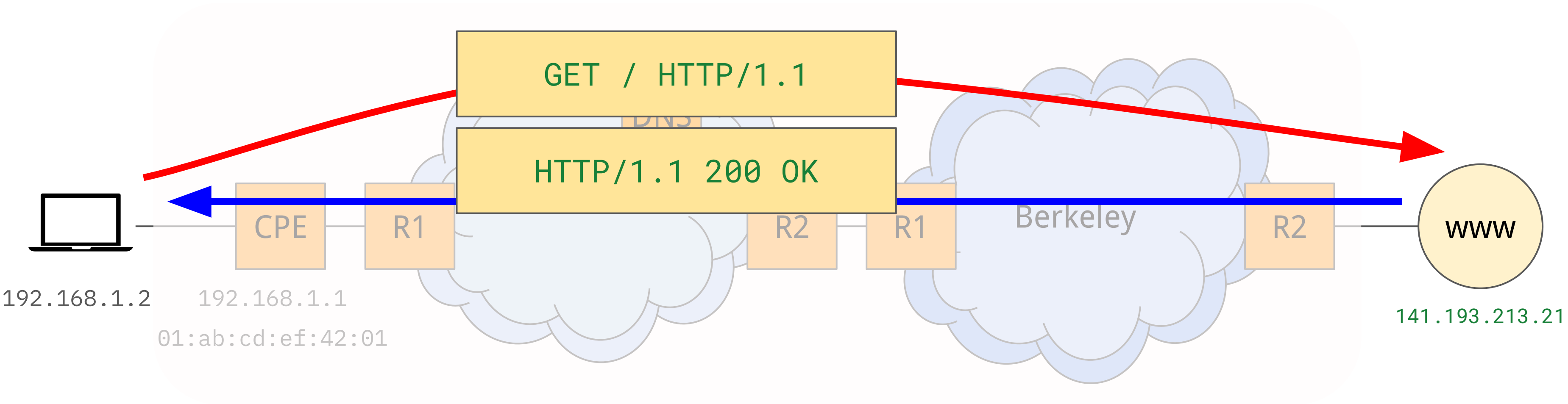
HTTP

- Default path / requested in HTTP returns an HTML page.
- This may have other content linked for it (e.g., `<img src=”/img.png”>`)
    - HTTP requests are pipelined across the same TCP connection.
- TCP connection will not immediately be closed.



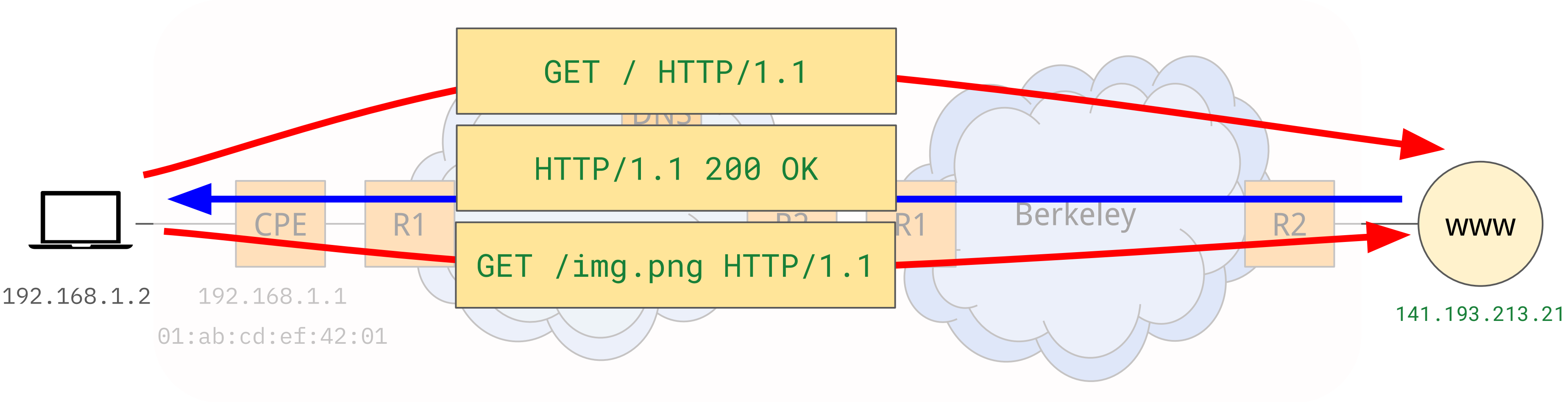
HTTP

- Default path `/` requested in HTTP returns an HTML page.
- This may have other content linked for it (e.g., `<img src=”/img.png”>`)
  - HTTP requests are <u>pipelined</u> across the same TCP connection.
- TCP connection will not immediately be closed.



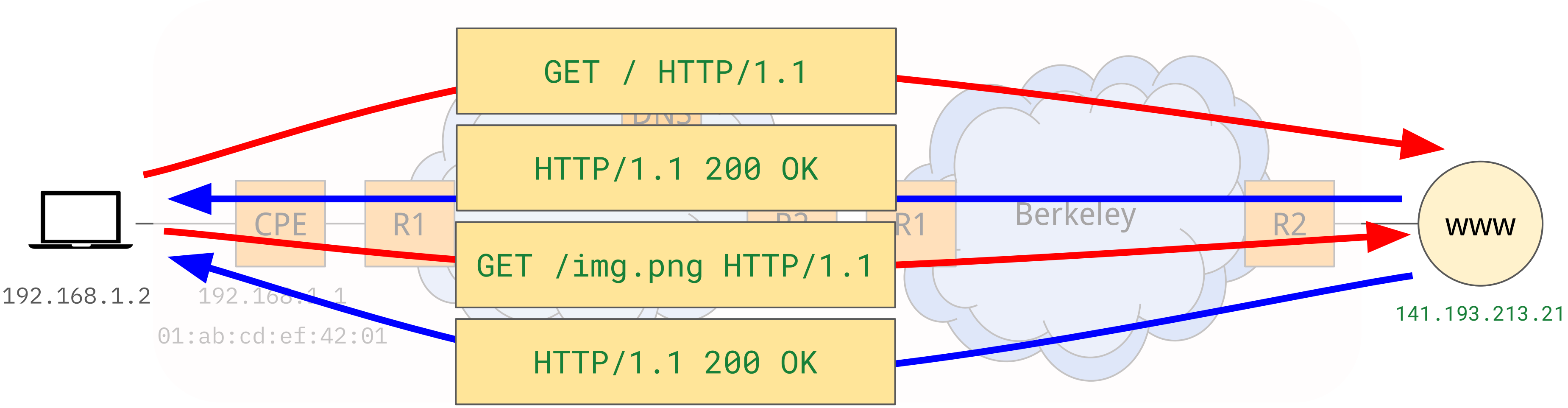
HTTP

- Default path / requested in HTTP returns an HTML page.
- This may have other content linked for it (e.g., <img src=”/img.png”>)
  - HTTP requests are pipelined across the same TCP connection.
- TCP connection will not immediately be closed.

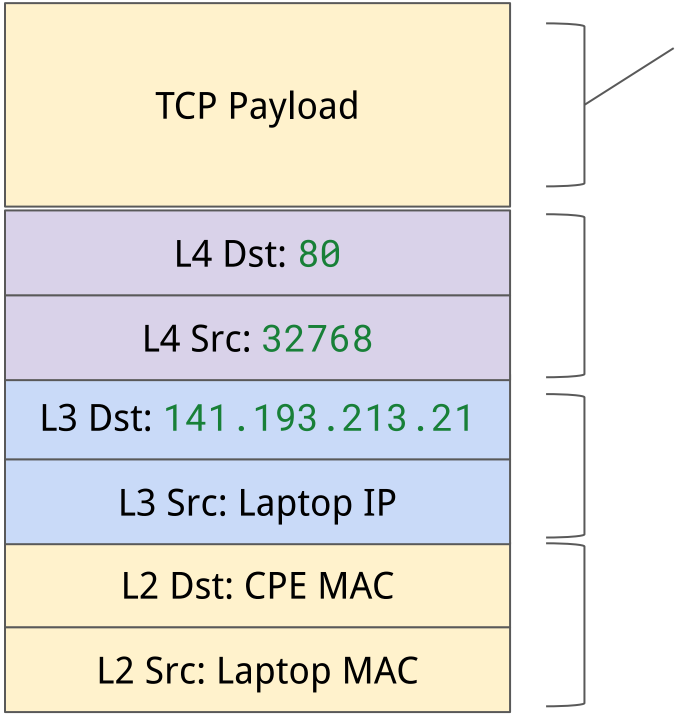


HTTP

- Default path / requested in HTTP returns an HTML page.
- This may have other content linked for it (e.g., <img src="/img.png">)
    - HTTP requests are pipelined across the same TCP connection.
- TCP connection will not immediately be closed.



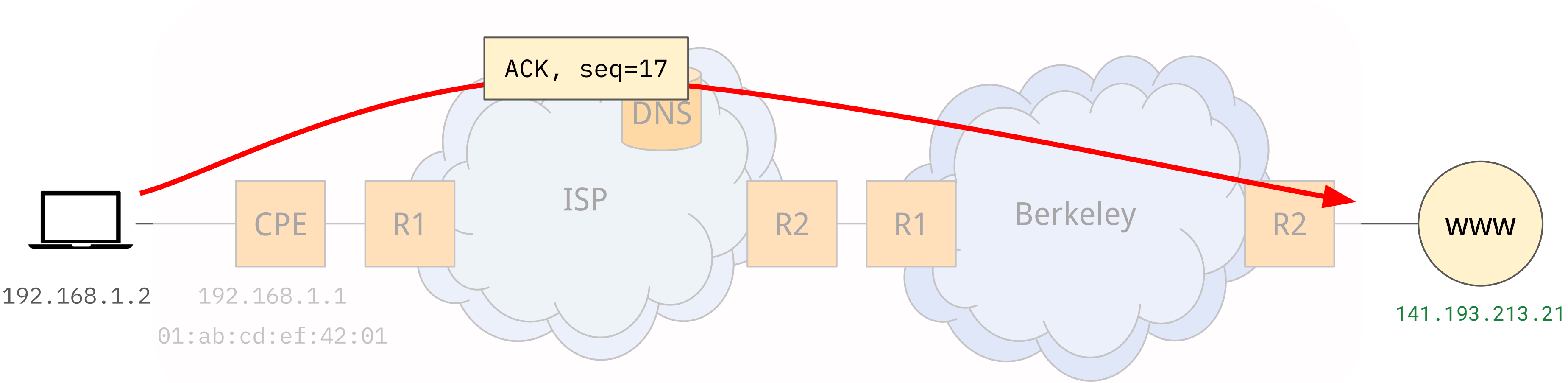
Building Up Packets: L7



- **Remember:** TCP provides a byte stream abstraction – so no 1:1 correlation of packets.
- Application calls `read` on the socket to receive the bytes.
- L4 headers are associated with the TCP connection.
- L3 headers to reach `berkeley.edu`
- Fixed L2 headers

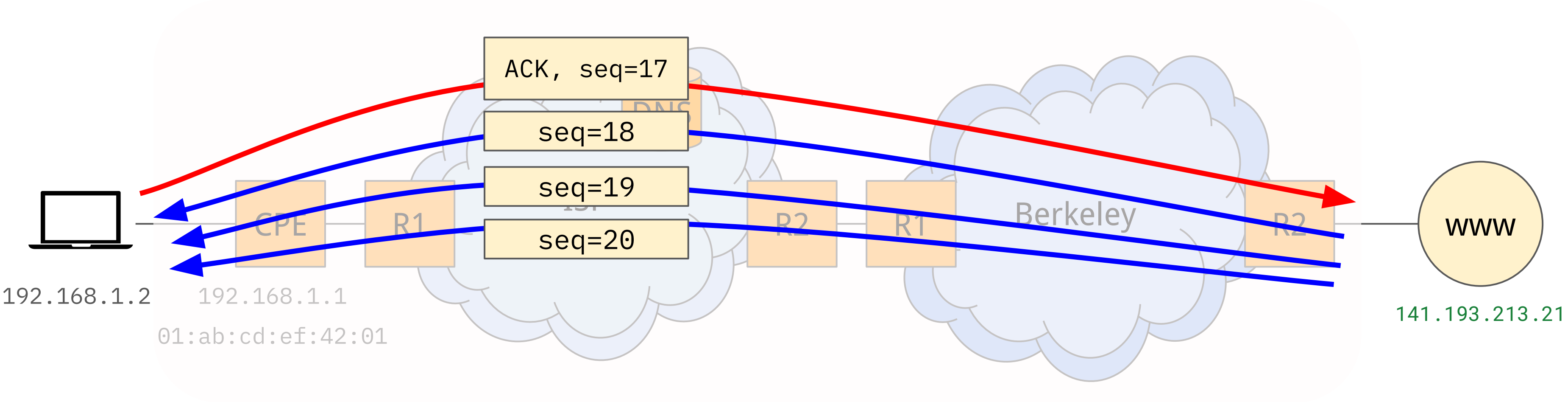
L4 view of our HTTP requests

- Multiple packets may be required to carry the HTTP requests and responses.
- Remember that there is not a 1:1 correspondence between packets in TCP – ACK the window size for the connection.



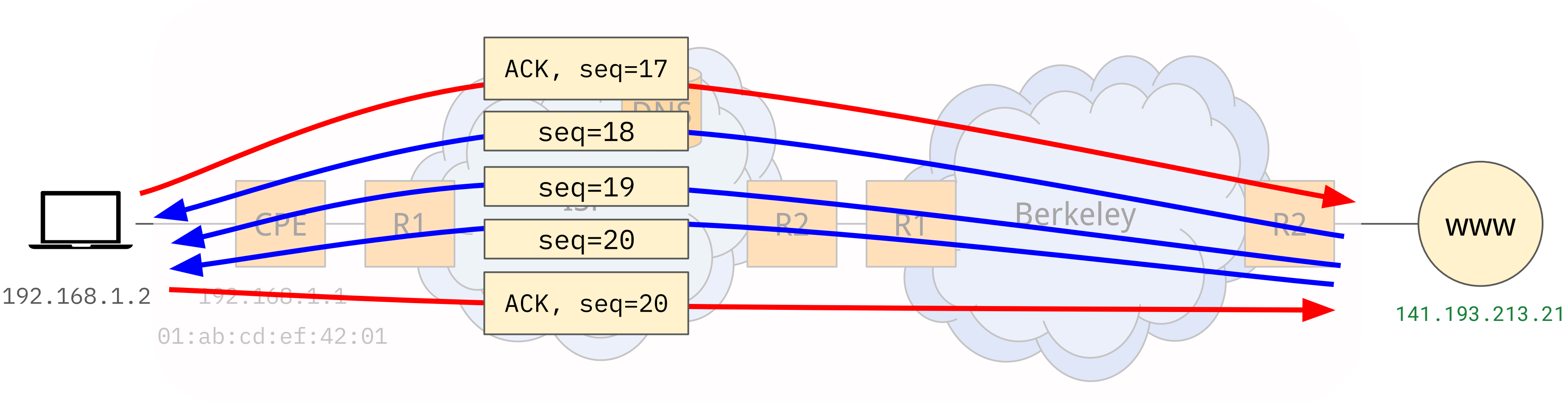
L4 view of our HTTP requests

- Multiple packets may be required to carry the HTTP requests and responses.
- Remember that there is not a 1:1 correspondence between packets in TCP – ACK the window size for the connection.



L4 view of our HTTP requests

- Multiple packets may be required to carry the HTTP requests and responses.
- Remember that there is not a 1:1 correspondence between packets in TCP – ACK the window size for the connection.



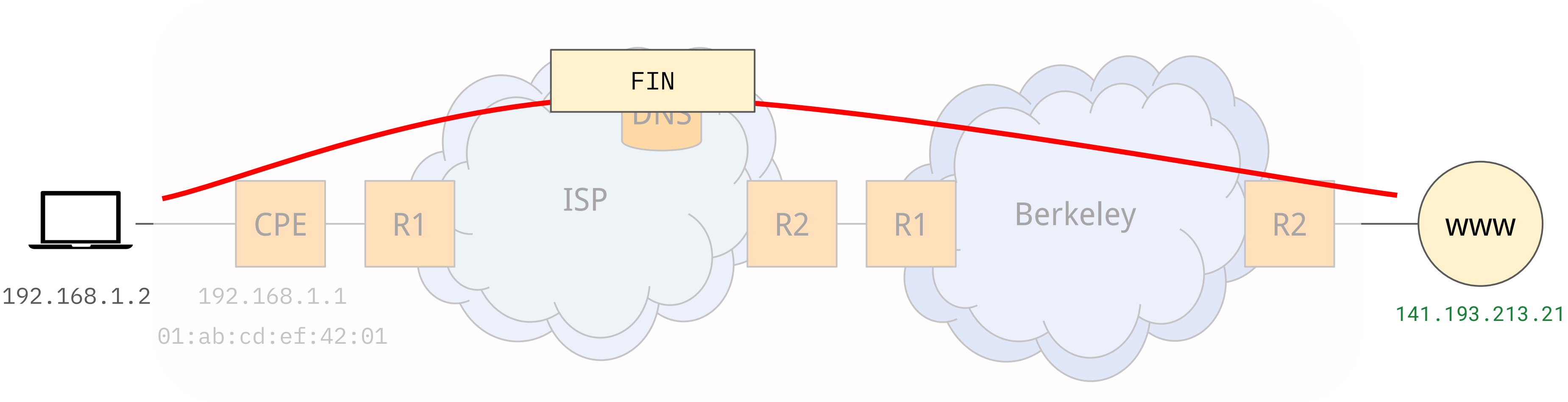
Dealing with the TCP byte stream

- From a programming perspective, socket operations are:
  - `socket` - establishes the socket (client or server)
  - `connect` - opens the TCP connection to the remote server
  - `write` - using the socket file descriptor, write contents to the socket.
  - `read` – read data from the TCP socket file descriptor, reads **N bytes**.

- We need the application to tell us something about when the request has ended.
  - Carriage return / line feed gives us a way to know when the request or response has ended.
  - We can use some headers to know how much memory to allocate (e.g., `Content-Length` tells us how many bytes to expect to read).

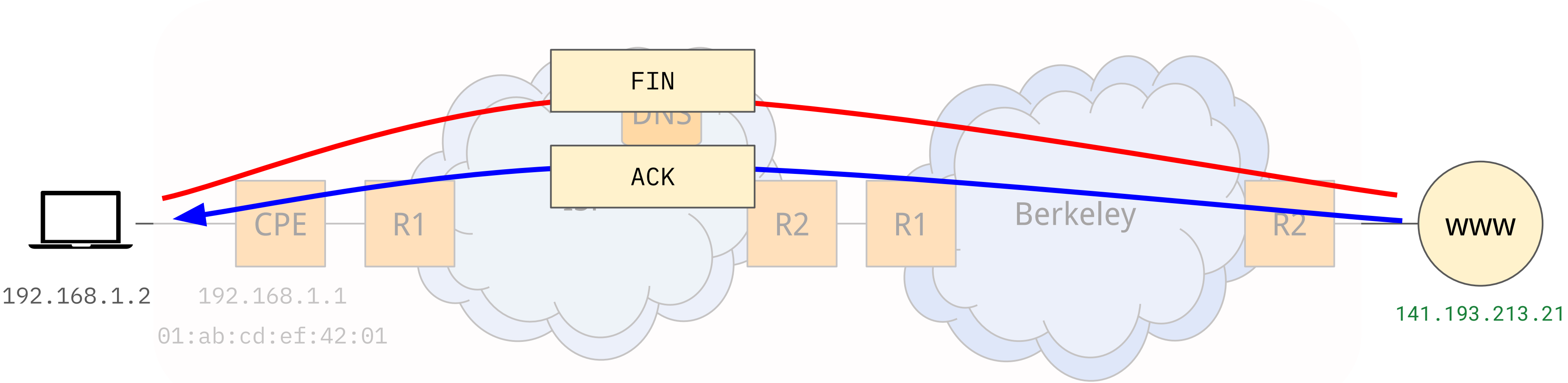
TCP - Close

- Persistent HTTP connections remain open until such time as the server or client chooses to close them.
  - Allows for pipelining of subsequent requests.
- Either the client or server can decide to close the connection.



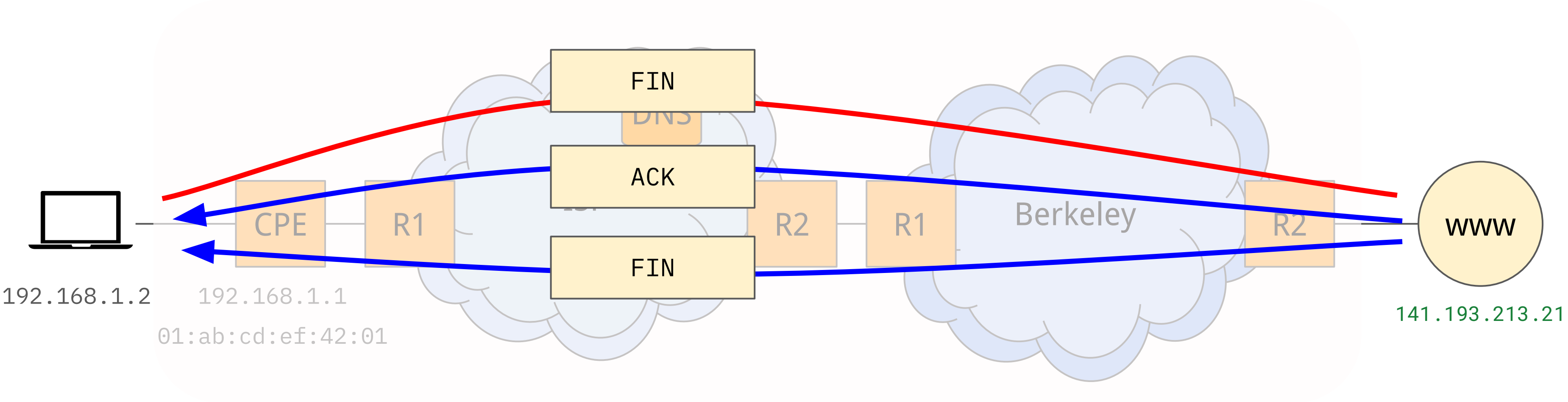
TCP - Close

- Persistent HTTP connections remain open until such time as the server or client chooses to close them.
  - Allows for pipelining of subsequent requests.
- Either the client or server can decide to close the connection.



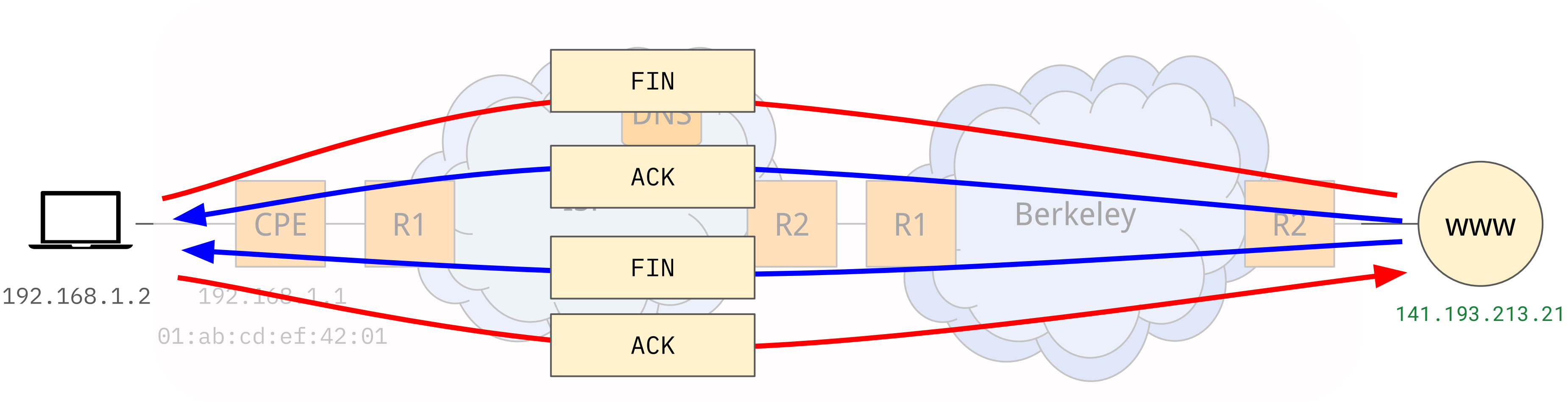
TCP - Close

- Persistent HTTP connections remain open until such time as the server or client chooses to close them.
    - Allows for pipelining of subsequent requests.
- Either the client or server can decide to close the connection.



TCP - Close

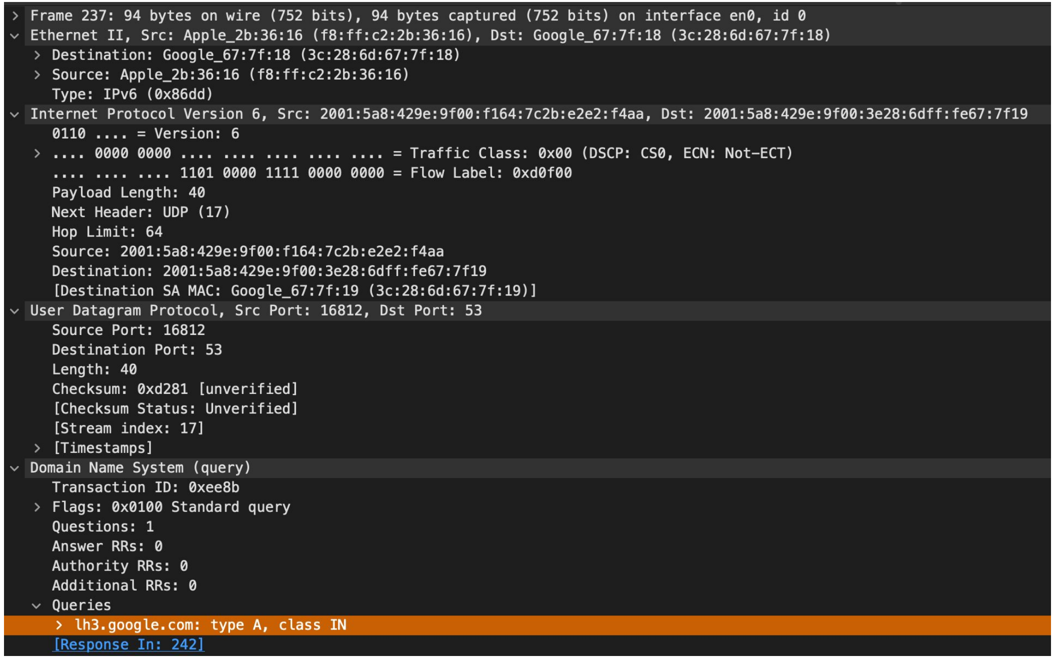
- Persistent HTTP connections remain open until such time as the server or client chooses to close them.
    - Allows for pipelining of subsequent requests.
- Either the client or server can decide to close the connection.



Questions?

Looking at packet flows

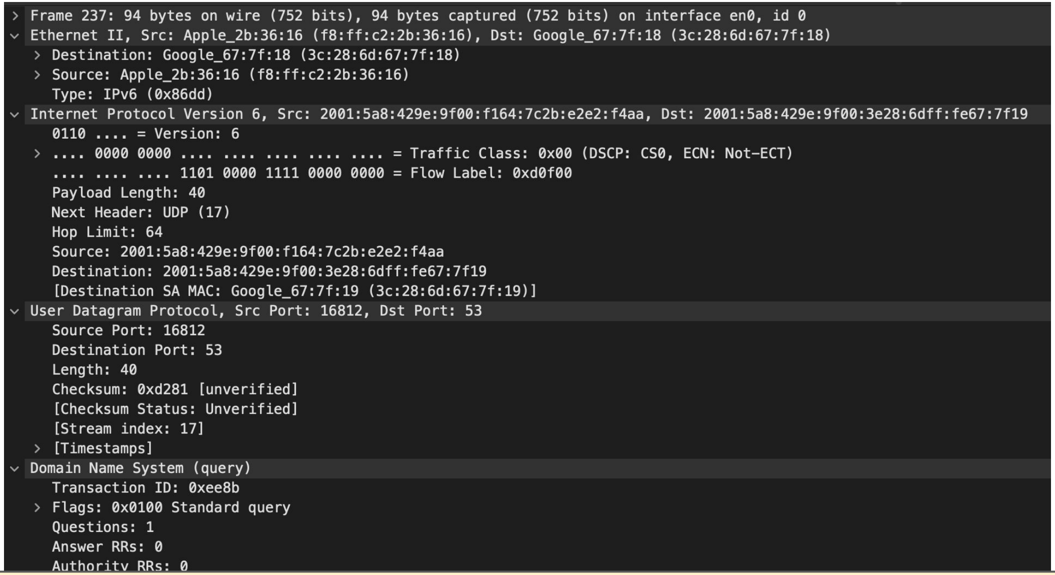
- You can use programs like `tshark` and `wireshark` to look at packets and their layers..



```
› Frame 237: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface en0, id 0
⌄ Ethernet II, Src: Apple_2b:36:16 (f8:ff:c2:2b:36:16), Dst: Google_67:7f:18 (3c:28:6d:67:7f:18)
  › Destination: Google_67:7f:18 (3c:28:6d:67:7f:18)
  › Source: Apple_2b:36:16 (f8:ff:c2:2b:36:16)
    Type: IPv6 (0x86dd)
⌄ Internet Protocol Version 6, Src: 2001:5a8:429e:9f00:f164:7c2b:e2e2:f4aa, Dst: 2001:5a8:429e:9f00:3e28:6dff:fe67:7f19
    0110 .... = Version: 6
  › .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 1101 0000 1111 0000 0000 = Flow Label: 0xd0f00
    Payload Length: 40
    Next Header: UDP (17)
    Hop Limit: 64
    Source: 2001:5a8:429e:9f00:f164:7c2b:e2e2:f4aa
    Destination: 2001:5a8:429e:9f00:3e28:6dff:fe67:7f19
    [Destination SA MAC: Google_67:7f:19 (3c:28:6d:67:7f:19)]
⌄ User Datagram Protocol, Src Port: 16812, Dst Port: 53
    Source Port: 16812
    Destination Port: 53
    Length: 40
    Checksum: 0xd281 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 17]
  › [Timestamps]
⌄ Domain Name System (query)
    Transaction ID: 0xee8b
  › Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  ⌄ Queries
    › lh3.google.com: type A, class IN
    [Response In: 242]
```

Looking at packet flows

- You can use programs like `tshark` and `wireshark` to look at packets and their layers..



```
> Frame 237: 94 bytes on wire (752 bits), 94 bytes captured (752 bits) on interface en0, id 0
⌄ Ethernet II, Src: Apple_2b:36:16 (f8:ff:c2:2b:36:16), Dst: Google_67:7f:18 (3c:28:6d:67:7f:18)
  > Destination: Google_67:7f:18 (3c:28:6d:67:7f:18)
  > Source: Apple_2b:36:16 (f8:ff:c2:2b:36:16)
    Type: IPv6 (0x86dd)
⌄ Internet Protocol Version 6, Src: 2001:5a8:429e:9f00:f164:7c2b:e2e2:f4aa, Dst: 2001:5a8:429e:9f00:3e28:6dff:fe67:7f19
    0110 .... = Version: 6
  > .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 1101 0000 1111 0000 0000 = Flow Label: 0xd0f00
    Payload Length: 40
    Next Header: UDP (17)
    Hop Limit: 64
    Source: 2001:5a8:429e:9f00:f164:7c2b:e2e2:f4aa
    Destination: 2001:5a8:429e:9f00:3e28:6dff:fe67:7f19
    [Destination SA MAC: Google_67:7f:19 (3c:28:6d:67:7f:19)]
⌄ User Datagram Protocol, Src Port: 16812, Dst Port: 53
    Source Port: 16812
    Destination Port: 53
    Length: 40
    Checksum: 0xd281 [unverified]
    [Checksum Status: Unverified]
    [Stream index: 17]
  > [Timestamps]
⌄ Domain Name System (query)
    Transaction ID: 0xee8b
  > Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
```

Often some “real world” complexities – e.g., secure/encrypted flows.

TLS

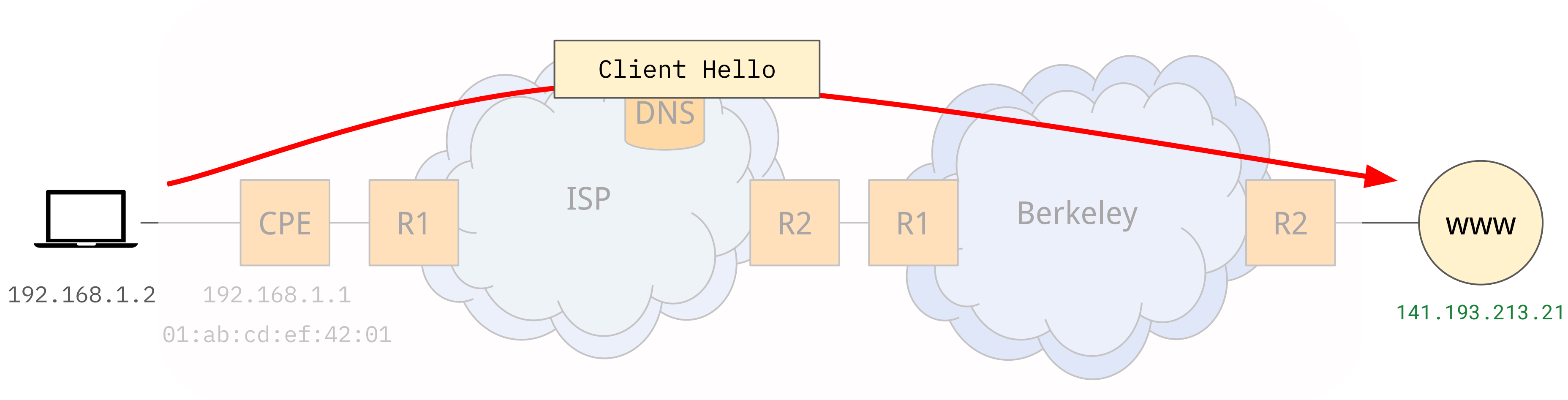
- We haven’t really mentioned encryption – but most HTTP traffic is securely transmitted as **HTTPS**.
  - Destination port is 443 not 80!

- Transport Layer Security (TLS) provides a way to encrypt traffic and authenticate remote hosts.
  - At the Transport Layer (i.e., L4) → TCP.

- Introduces new TLS handshake at the start of the TCP flow following the TCP three-way handshake.

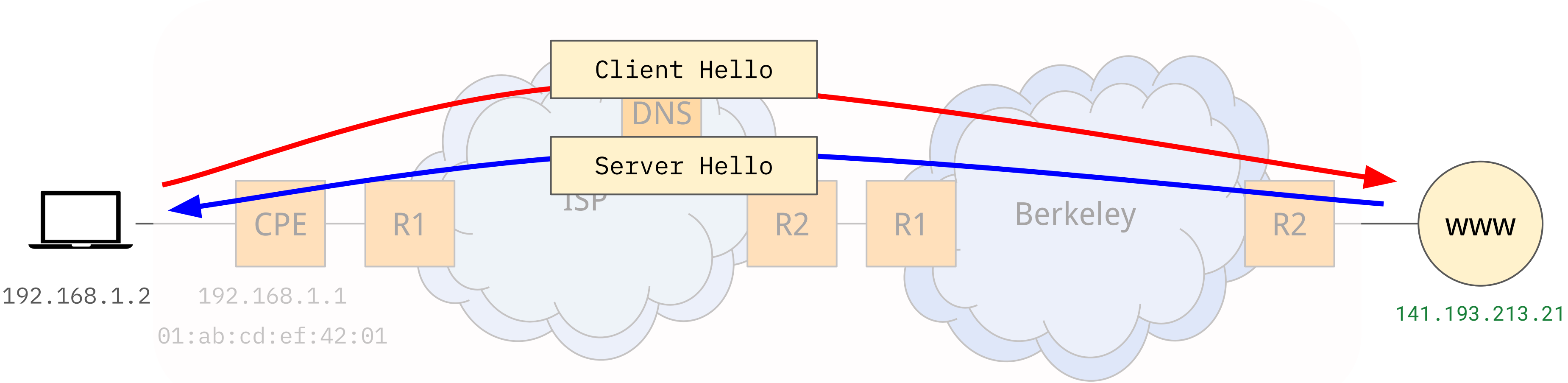
TLS Handshakes

- Handshake starts with a client sending a hello.
    - Specifies the version of TLS to be used.
    - Specifies on the cipher suites to be used to encrypt traffic.
    - A client-specific “random” secret.



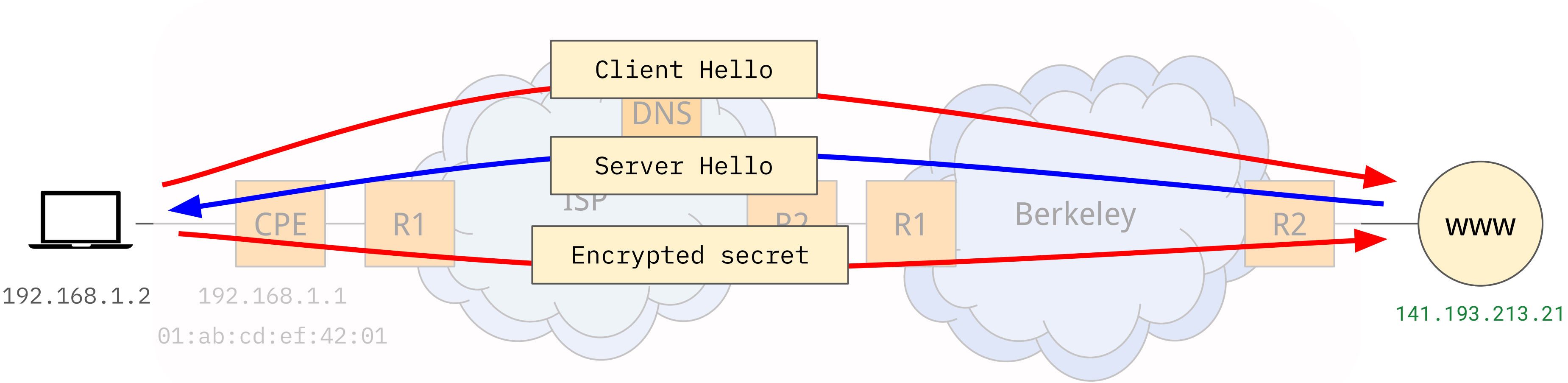
TLS Handshakes

- Server subsequently sends a hello message.
  - Provides a **certificate** that identifies the server.
  - Along with the selected cipher from the options the client provided.
  - A server-generated “random” secret.



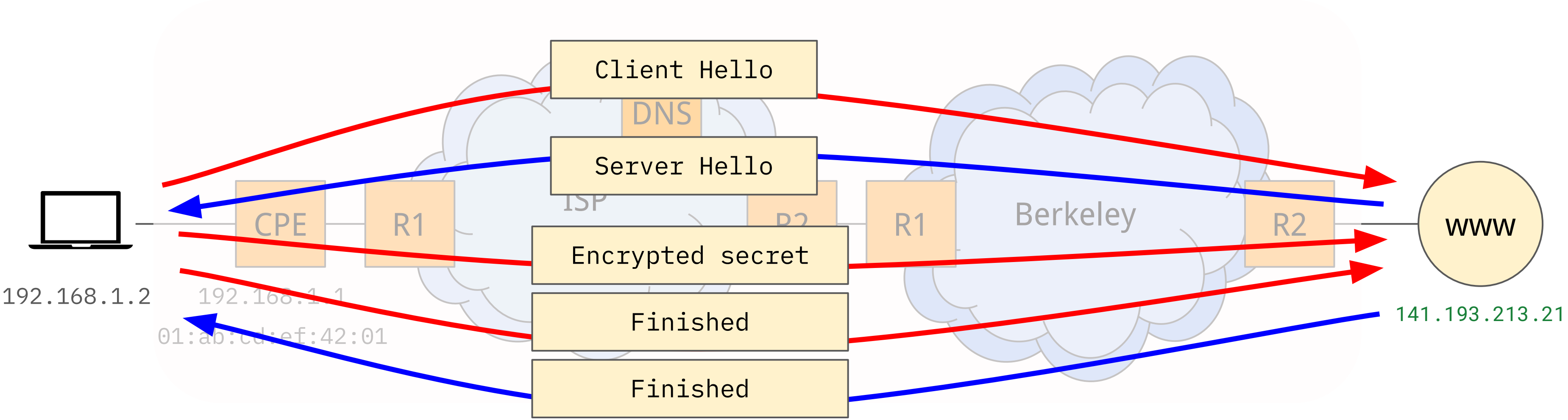
TLS Handshakes

- Client then provides an additional secret (“premaster”).
  - This is encrypted using the public key of the certificate that was provided by the server.
- The client and server calculate a key for the session:
  - Using the client random, server random, and the premaster secret.



TLS Handshakes

- Client and server then send each other an encrypted message to indicate the handshake is finished.
  - All subsequent messages sent in TCP can be encrypted using the session key negotiated.



Questions?

Layering

- Layering gives us a powerful way to solve specific problems without exposing everyone to the complexity of solving them.

- We haven't talked at all about the *electrical engineering* or *physics* of putting bits on the wire.
  - But we've relied on this working throughout our example.

- Fundamentally has let us continue to evolve networking for new applications.

Layering - building on top of L7 protocols.

- We looked at some common L7 protocols – particularly **HTTP** (1.1 & 2).
  - Provided us some common primitive operations.

- Often applications might have the same requirements.
  - e.g., multiplexing multiple data retrievals onto the same underlying HTTP connection.
  - e.g., bi-directionally streaming data between a client and a server.

- We therefore have common frameworks that enable us to implement these operations without starting from scratch.

- An example is a **Remote Procedure Call** (RPC) library.
  - Apache Thrift, gRPC.
  - Allow us to build networked applications without repeating ourselves.

Why layers? **Abstraction**.

- Let’s think about what a developer sees when they work with a higher-layer RPC framework.

- Making a simple request to a remote `Greeter` server:
  - Client says `Hello!`
  - Server returns a message (`Hello World`).

Why layers? **Abstraction**.

```
func main() {
	flag.Parse()
	// Set up a connection to the server.
	conn, err := grpc.Dial(*addr, grpc.WithTransportCredentials(insecure.NewCredentials()))
	if err != nil {
		log.Fatalf("did not connect: %v", err)
	}
	defer conn.Close()
	c := pb.NewGreeterClient(conn)

	// Contact the server and print out its response.
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	defer cancel()
	r, err := c.SayHello(ctx, &pb.HelloRequest{Name: *name})
	if err != nil {
		log.Fatalf("could not greet: %v", err)
	}
	log.Printf("Greeting: %s", r.GetMessage())
}
```

Why layers? **Abstraction**.

```
func main() {
	flag.Parse()
	// Set up a connection to the se
	conn, err := grpc.Dial(*addr, grpc.WithTransportCr
	if err != nil {
		log.Fatalf("did not connect: %v", err)
	}
	defer conn.Close()
	c := pb.NewGreeterClient(conn)

	// Contact the server and print out its response.
	ctx, cancel := context.WithTimeout(context.Backgro
	defer cancel()
	r, err := c.SayHello(ctx, &pb.HelloR
	if err != nil {
		log.Fatalf("could not greet: %v", err)
	}
	log.Printf("Greeting: %s", r.GetMessage())
}
```

Developer did not think about:

1) Machine address or DHCP
2) Any headers (IP, Ethernet, TCP)
3) DNS
4) TCP byte stream
5) HTTP/2
6) gRPC (specific use of HTTP/2)

Rather the developer could think at the **application layer** only – and implement their own logic.

What’s next?

- We’ve covered the end-to-end *typical* path – used on the Internet and assuming wired connections.

- We’ll explore some of the places where performance and new applications have driven new requirements – **how has inter-server networking evolved in the datacenter?** (Nandita)

- Look at some of the diversity that happens at different layers – particularly, our wired Ethernet clients can’t move – **how does wireless and cellular networking work?** (Sylvia)

- Hear from a guest lecturer about some of the networking challenges they have been facing!

Thanks for the engagement and questions over the semester.

I’ll be in lecture and have office hours until April 25th.