Lecture 16: Networks, HTTP, Proxies and MapReduce

Principles of Computer Systems

Autumn 2019

Stanford University

Computer Science Department

Instructors: Chris Gregg

Philip Levis

PDF of this presentation

Computer Networks

- Computer networks are a type of computer system
- Most exciting systems and applications today are networked
- Today we'll look under the covers a bit about how networks are designed and work
    - Layering and encapsulation
    - The 7 OSI layers
- This lecture will just skim the surface: if you want to learn more, take CS144

The Basic Benefit and Two Abstractions

- Computer networks mean your application is no longer limited to the data that it has locally on your own machine
  - It can receive data from and send data to other machines
  - This data can be interactive (people), not just documents
- Today, when we talk about computer networks we generally mean the Internet
  - There were many networking technologies before and concurrent with the Internet
  - The Internet won! Thankfully. Ever heard of IPX, SNA, AppleTalk?
- Operating systems provide two basic abstractions to the Internet
  - A reliable, in-order byte stream to an application running on another computer
    - Provided by TCP (Transmission Control Protocol)
    - Used for: ssh, http, email, NFS, etc.
  - An unreliable message (finite datagrams) service to an application running on another computer
    - Provided by UDP (User Datagram Protocol)
    - Used for DNS, some latency-sensitive applications (voice, video, gaming)

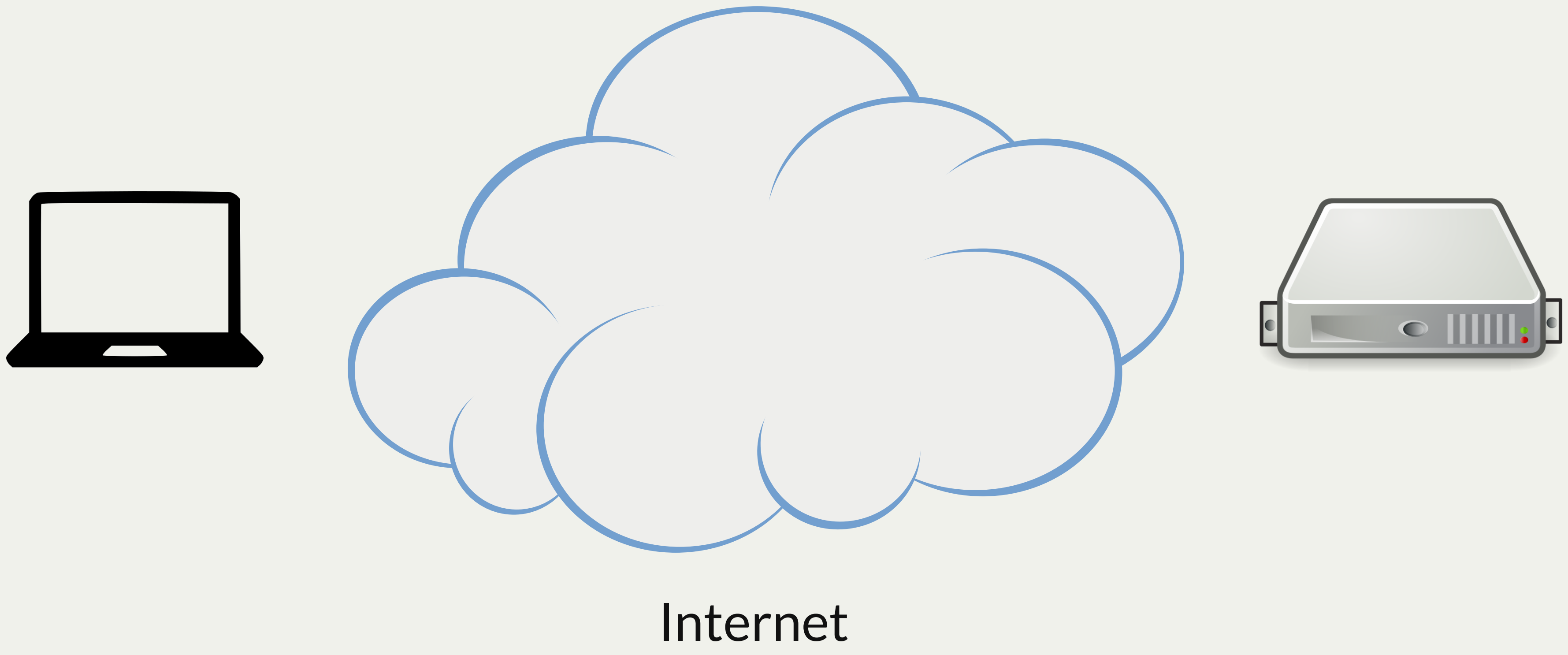
The Basic Benefit and Two Abstractions

- Computer networks mean your application is no longer limited to the data that it has locally on your own machine
  - It can receive data from and send data to other machines
  - This data can be interactive (people), not just documents
- Today, when we talk about computer networks we generally mean the Internet
  - There were many networking technologies before and concurrent with the Internet
  - The Internet won! Thankfully. Ever heard of IPX, SNA, AppleTalk?
- Operating systems provide two basic abstractions to the Internet
  - **A reliable, in-order byte stream to an application running on another computer**
    - **Provided by TCP (Transmission Control Protocol)**
    - **Used for: ssh, http, email, NFS, etc.**
  - An unreliable message (finite datagrams) service to an application running on another computer
    - Provided by UDP (User Datagram Protocol)
    - Used for DNS, some latency-sensitive applications (voice, video, gaming)

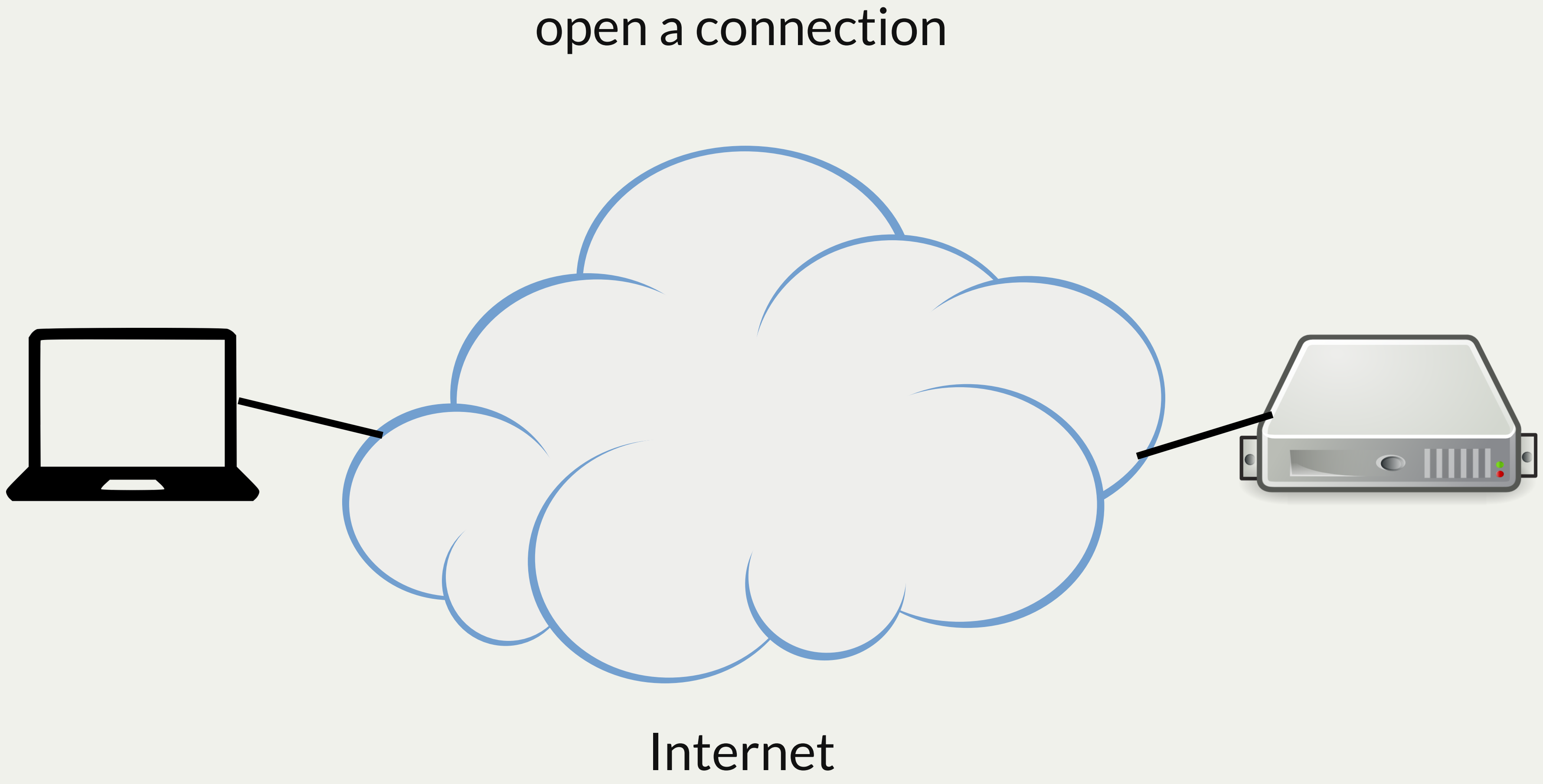
Reliable, In-Order Byte Stream

- A TCP connection behaves much like a bidirectional pipe
  - Both sides can read, both sides can write
  - Establishing a connection has nothing to do with forking: two programs can be on different computers across the globe
  - Usually, one computer (a server) waits for connection requests from clients
    - You've seen this with connect and accept
    - Other approaches are for edge cases and we won't worry about them
- Making TCP work as well as it does has taken 30 years of continual research and engineering: people still publish papers on it (e.g., Keith Winstein, Balaji Prabhakar)
  - One example: TCP was originally designed for wide area networks, where nodes are milliseconds apart; making it work well in the datacenter, where nodes are microseconds apart, required changing a few of its algorithms (Balaji).
  - Another example: TCP is designed to be extremely robust and work in all kinds of network conditions. Can you tune TCP to perform better in certain network settings (e.g., LTE vs. WiFi vs. wired) and how much improvement will you see (Keith)?

Web: HTTP over Reliable, In-Order Byte Stream

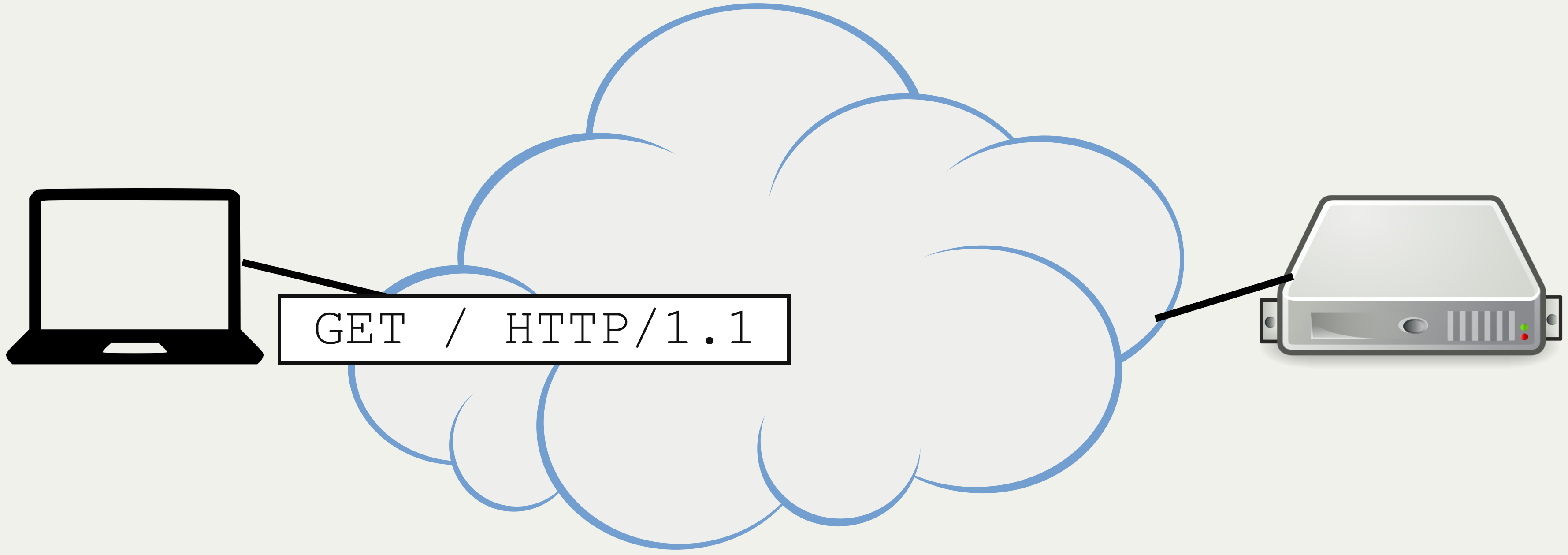


Web: HTTP over Reliable, In-Order Byte Stream



Web: HTTP over Reliable, In-Order Byte Stream

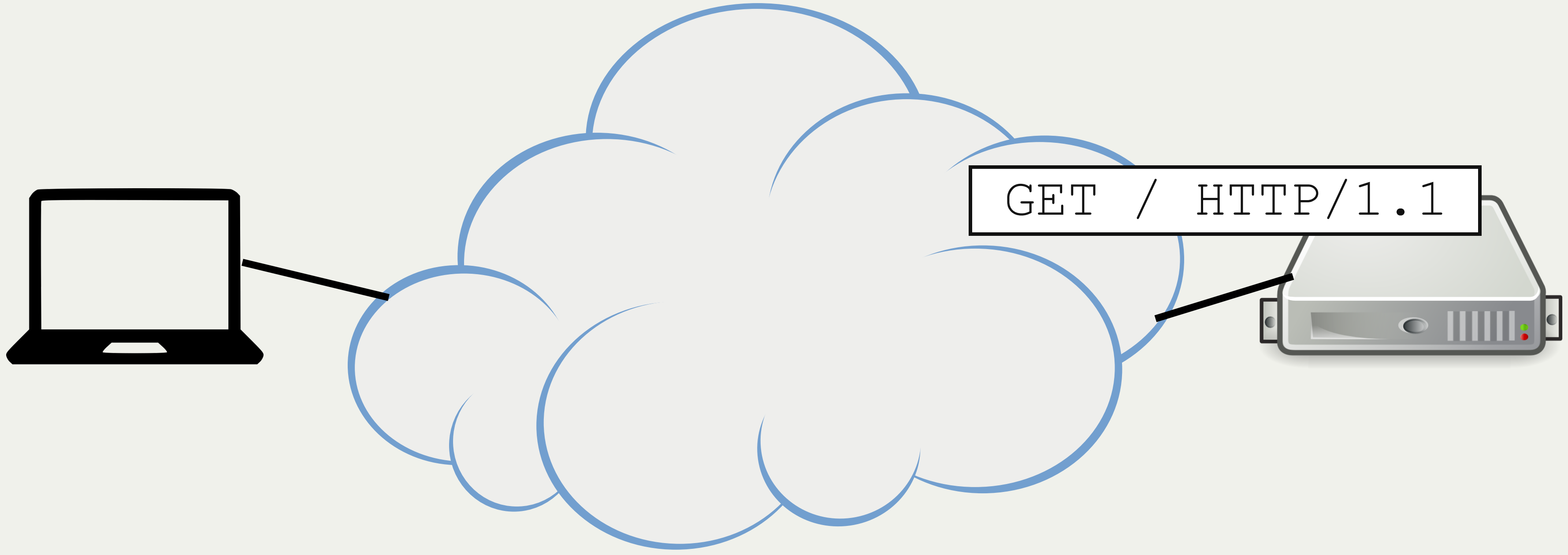
write a request



Internet

Web: HTTP over Reliable, In-Order Byte Stream

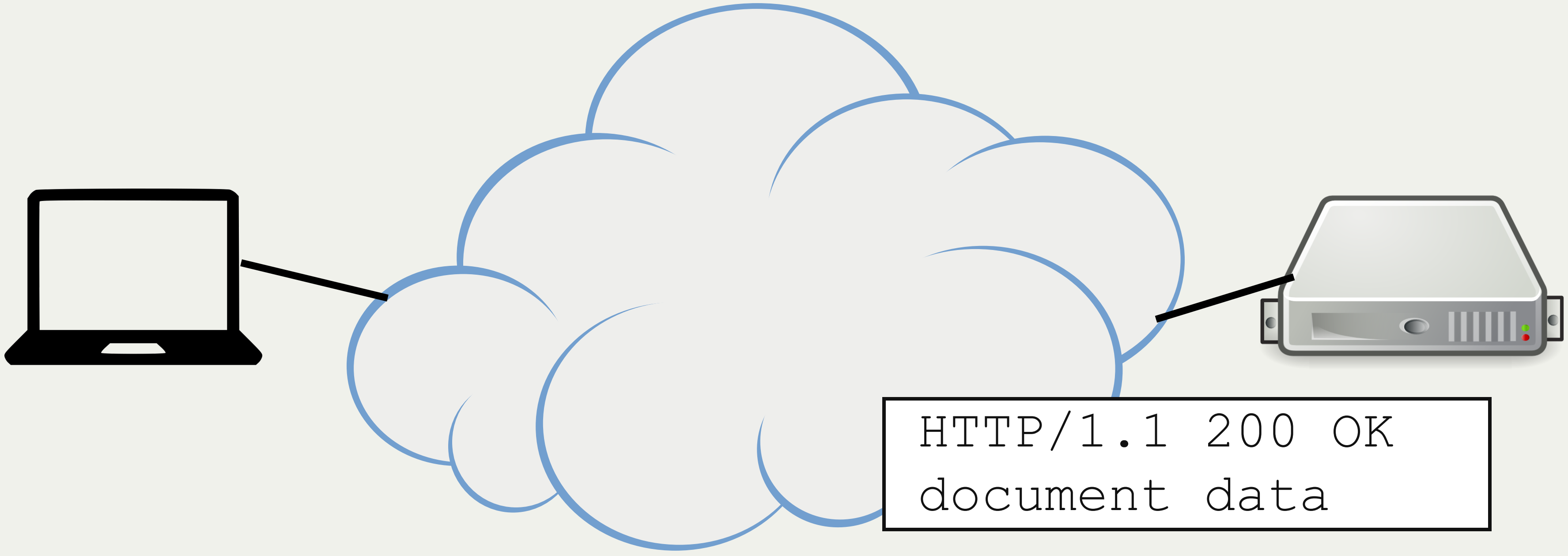
read the request



Internet

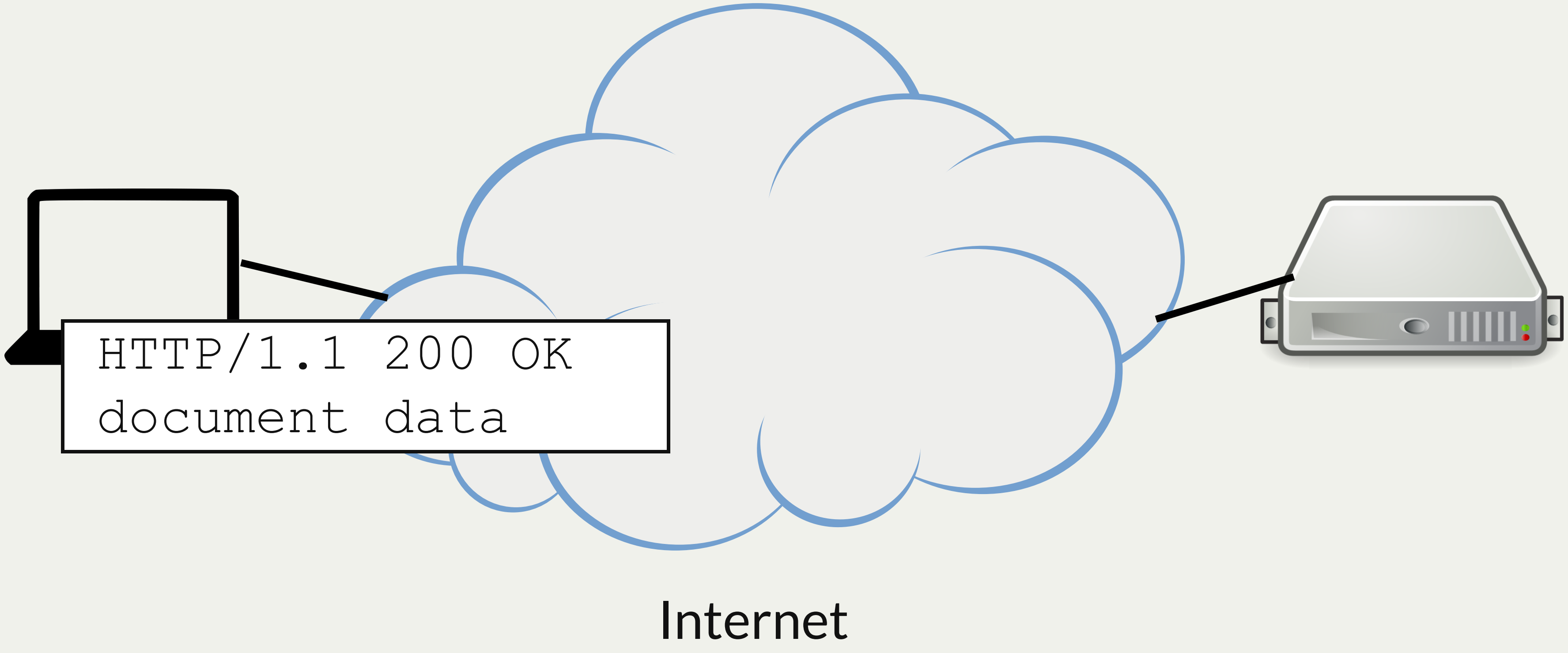
Web: HTTP over Reliable, In-Order Byte Stream

write the response



```
HTTP/1.1 200 OK
document data
```

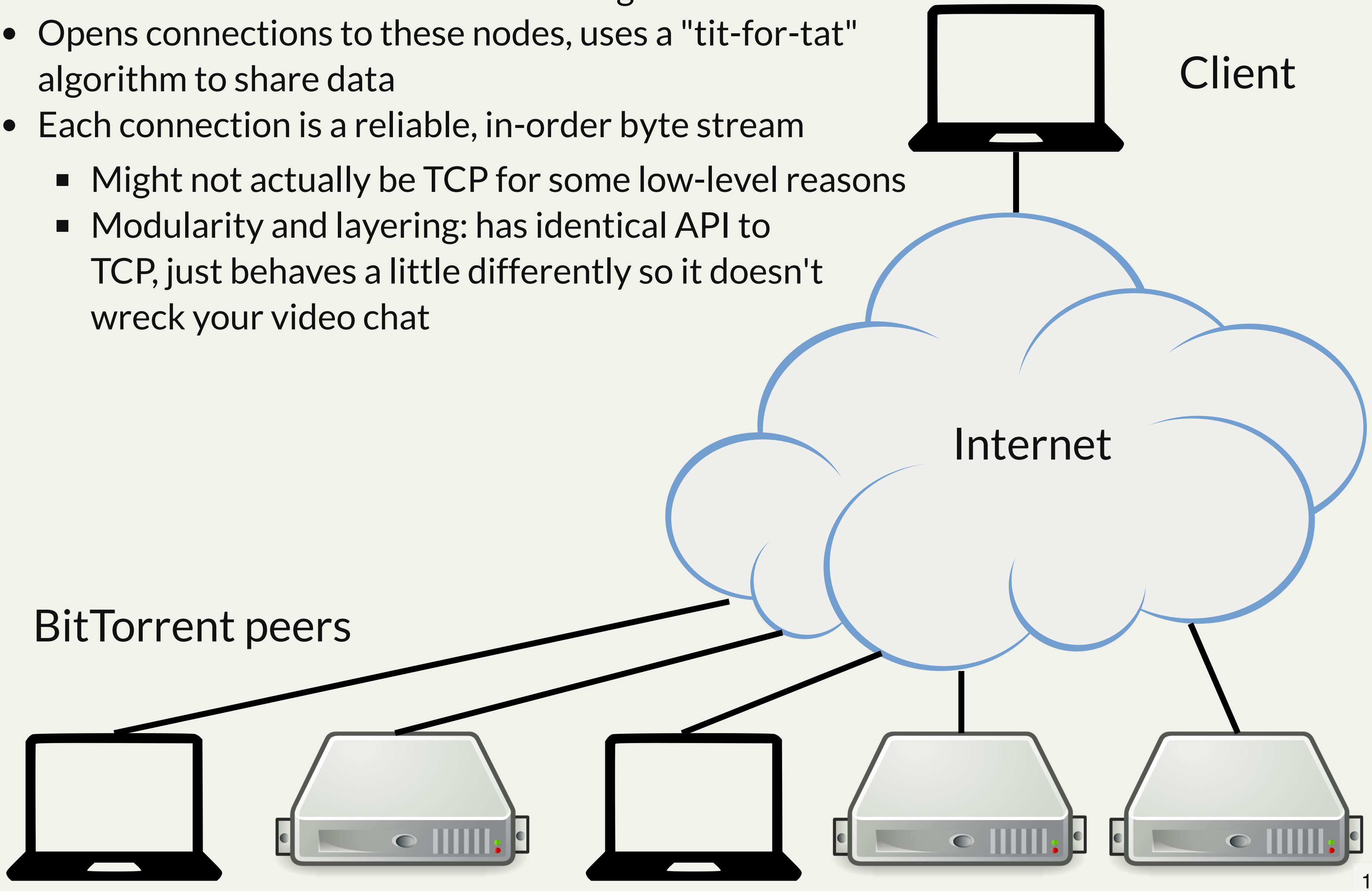
Web: HTTP over Reliable, In-Order Byte Stream



```
HTTP/1.1 200 OK
document data
```

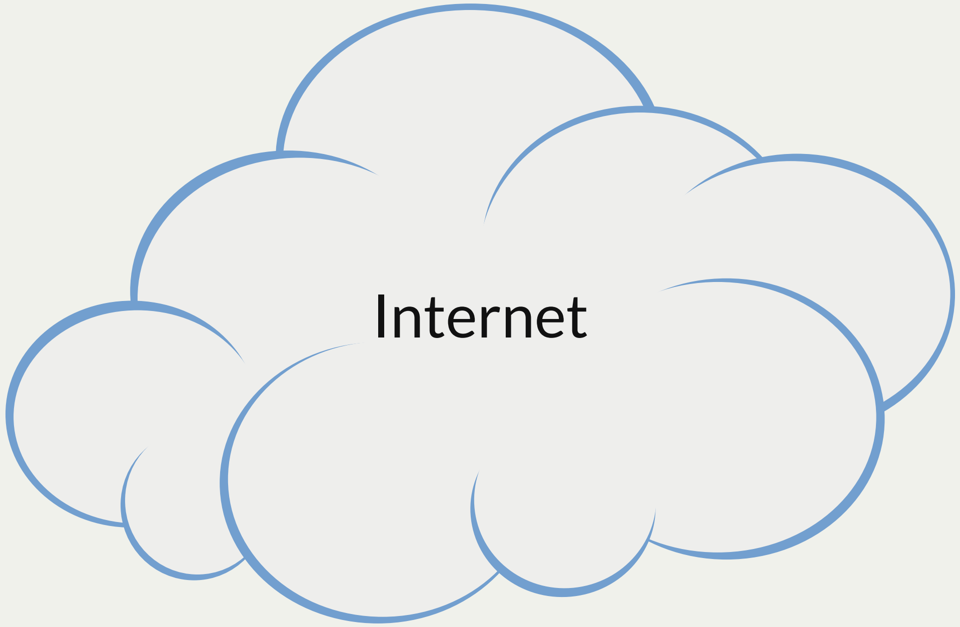
BitTorrent: Organizing a Swarm

- Client learns locations of nodes storing "chunks" of file
- Opens connections to these nodes, uses a "tit-for-tat" algorithm to share data
- Each connection is a reliable, in-order byte stream
  - Might not actually be TCP for some low-level reasons
  - Modularity and layering: has identical API to TCP, just behaves a little differently so it doesn't wreck your video chat



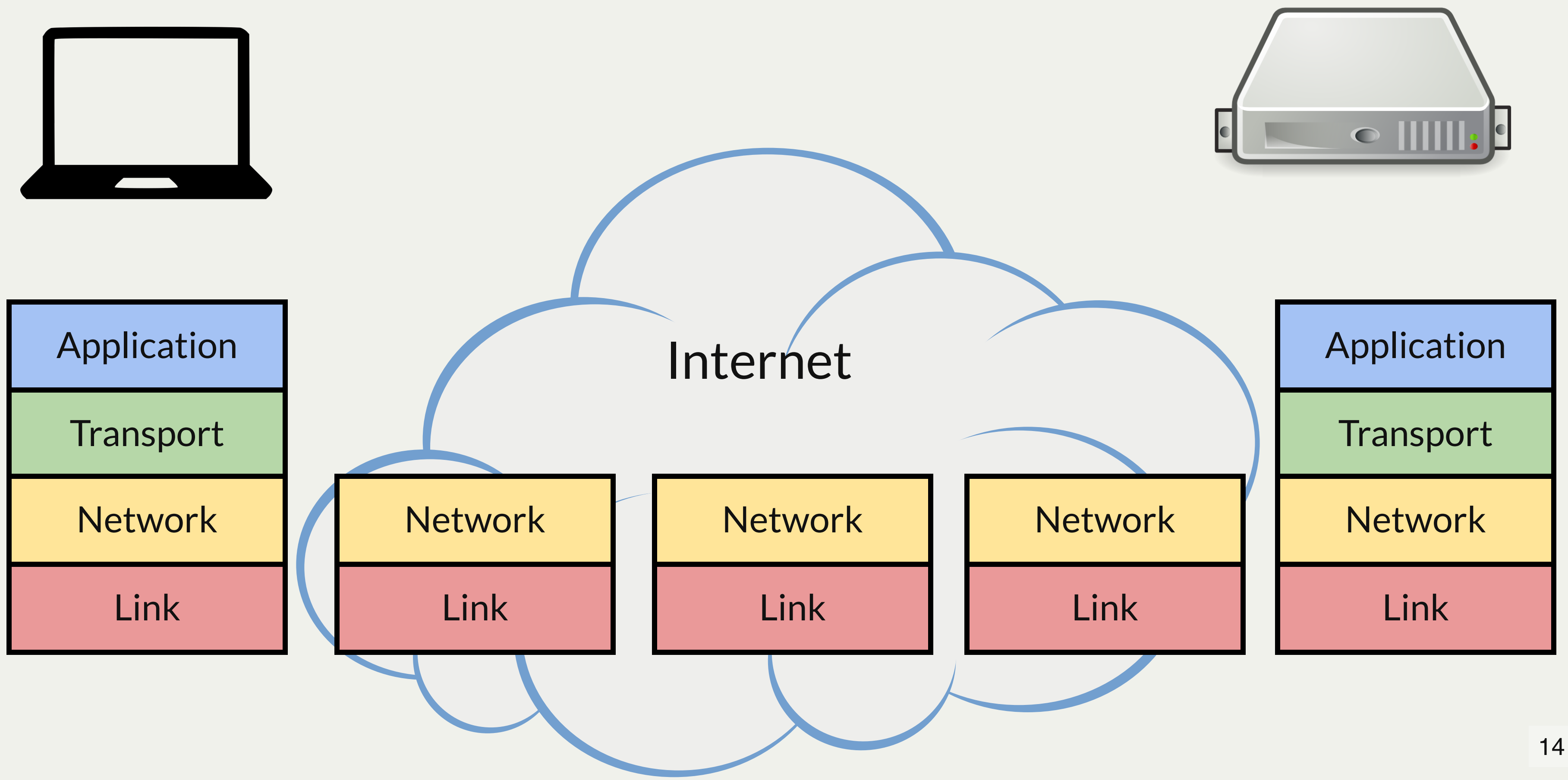
Internet: Under the Covers

- It's not a magic cloud (although it sometimes seems to be)
- When you make a web request, there are four layers of name in play
    - Application layer: name of document (e.g., index.html)
    - Transport layer: port identifying what service (e.g., HTTP on port 80)
    - Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
    - Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



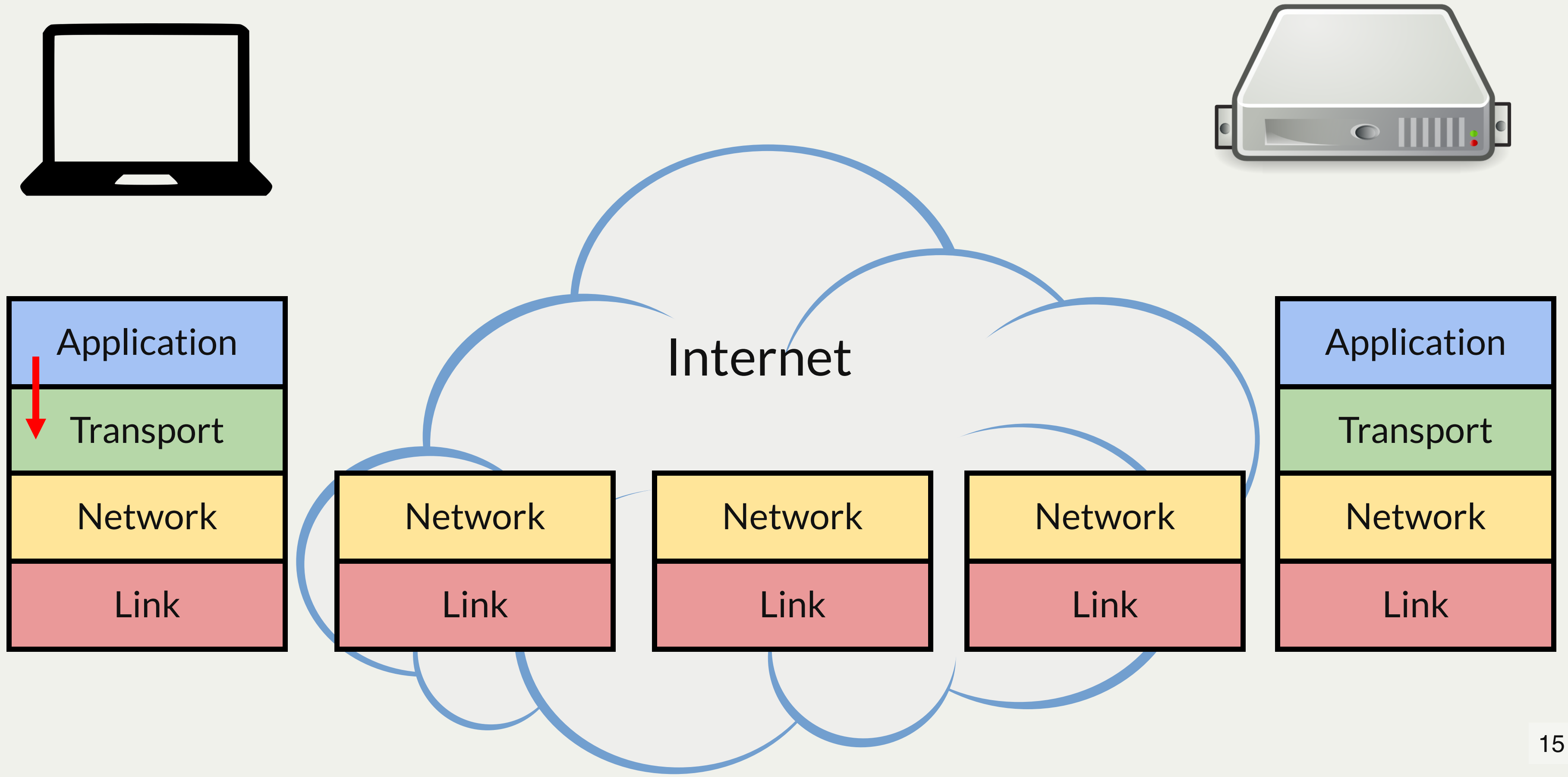
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



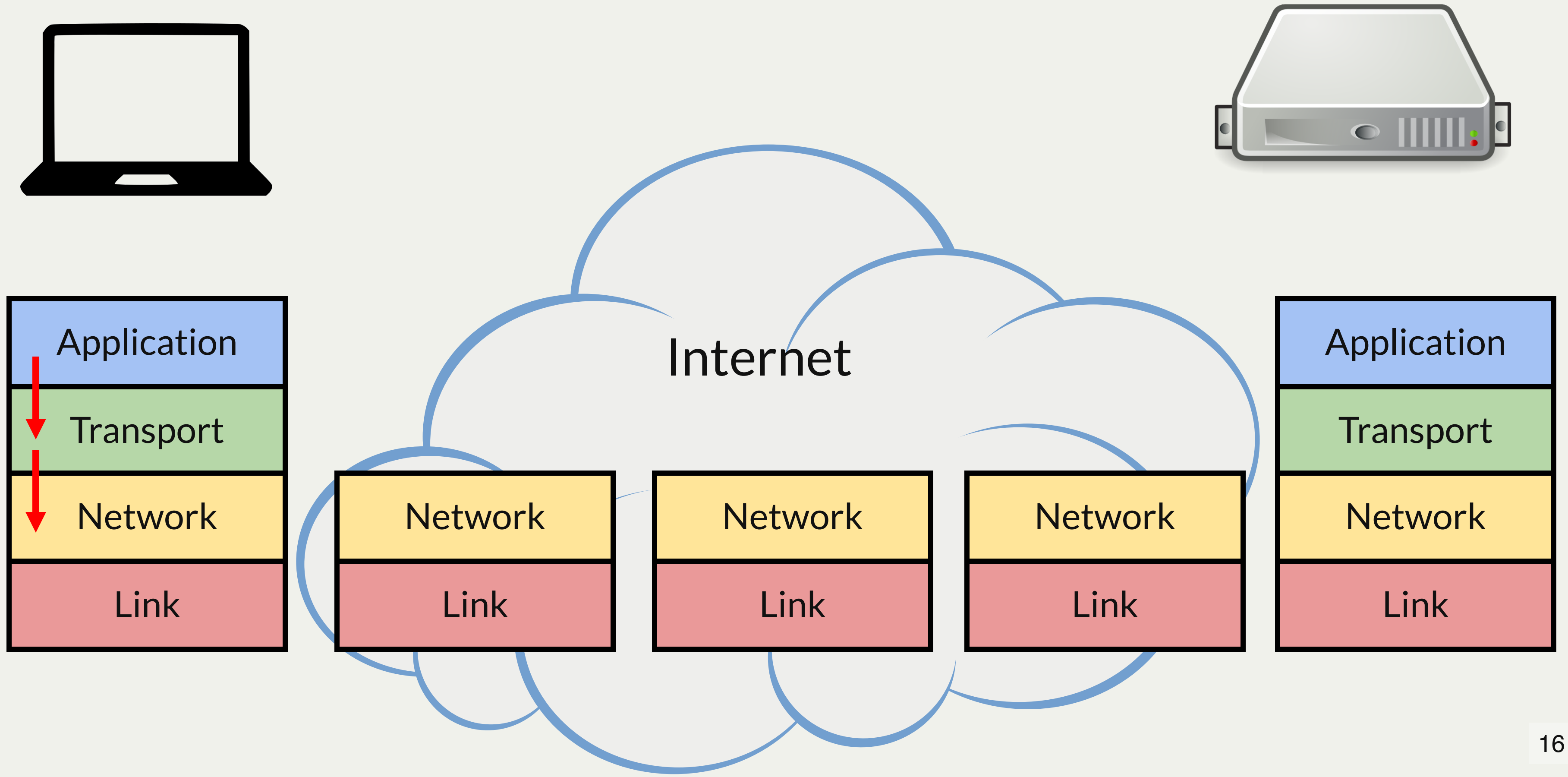
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



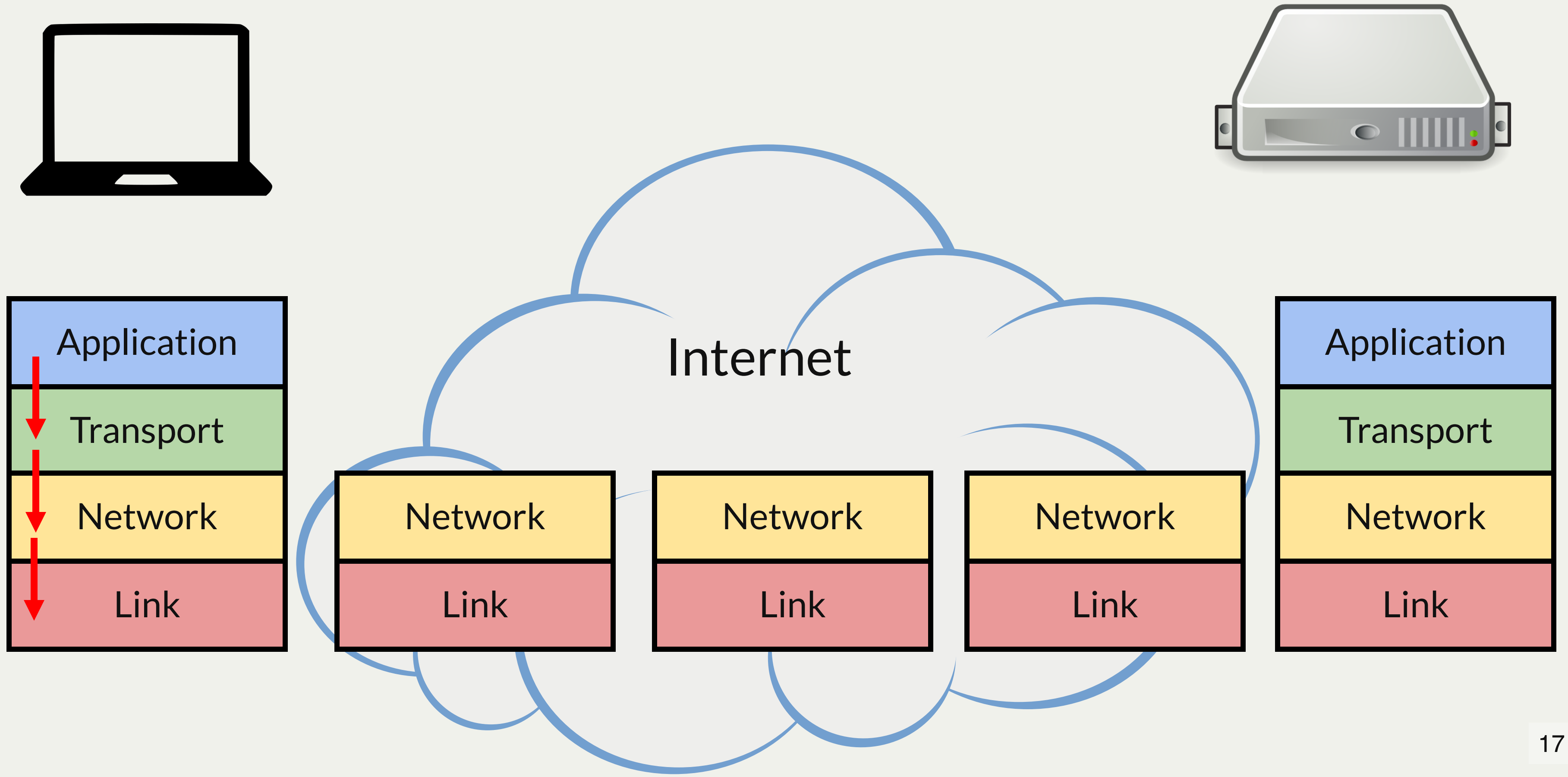
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



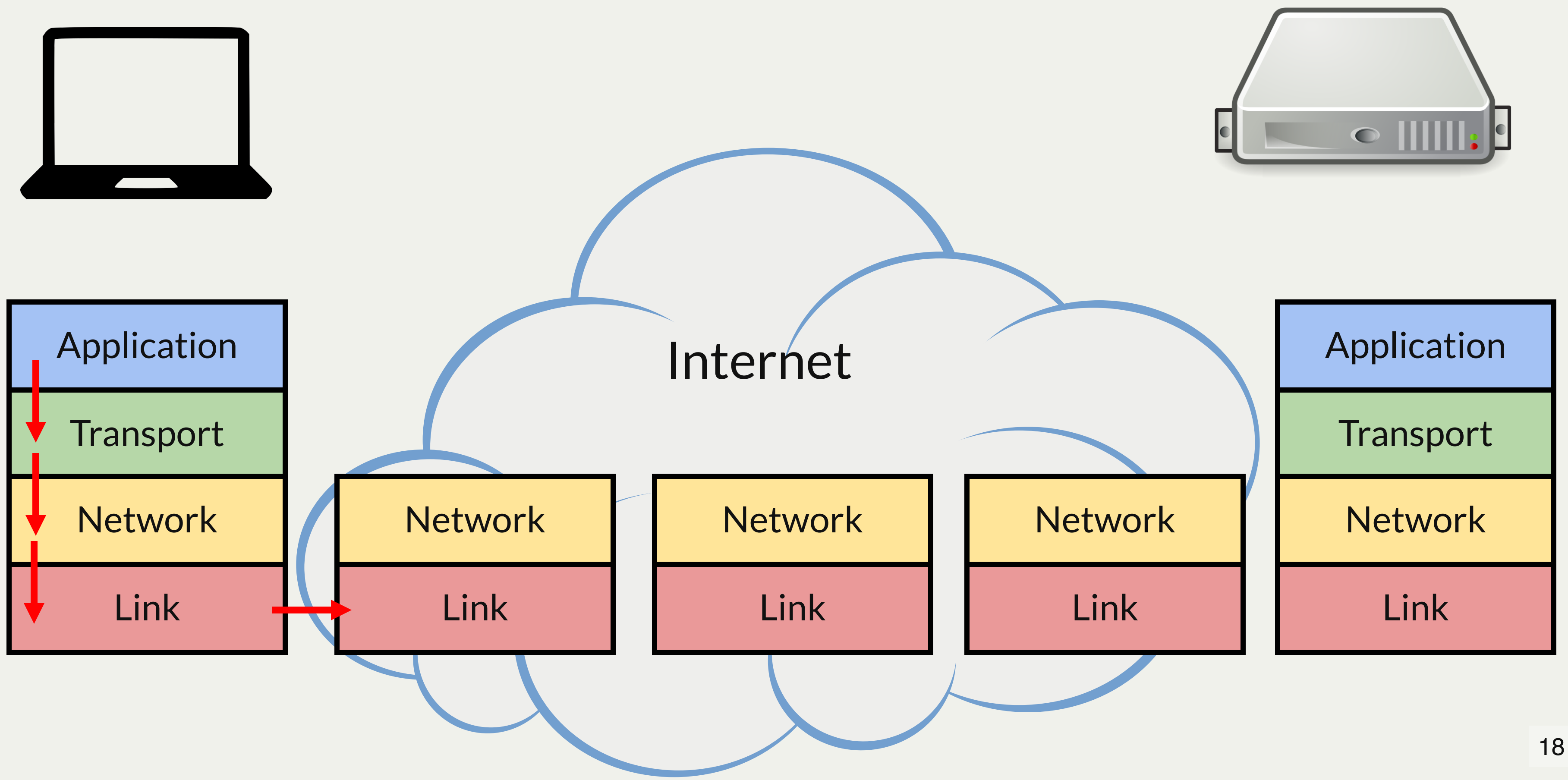
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



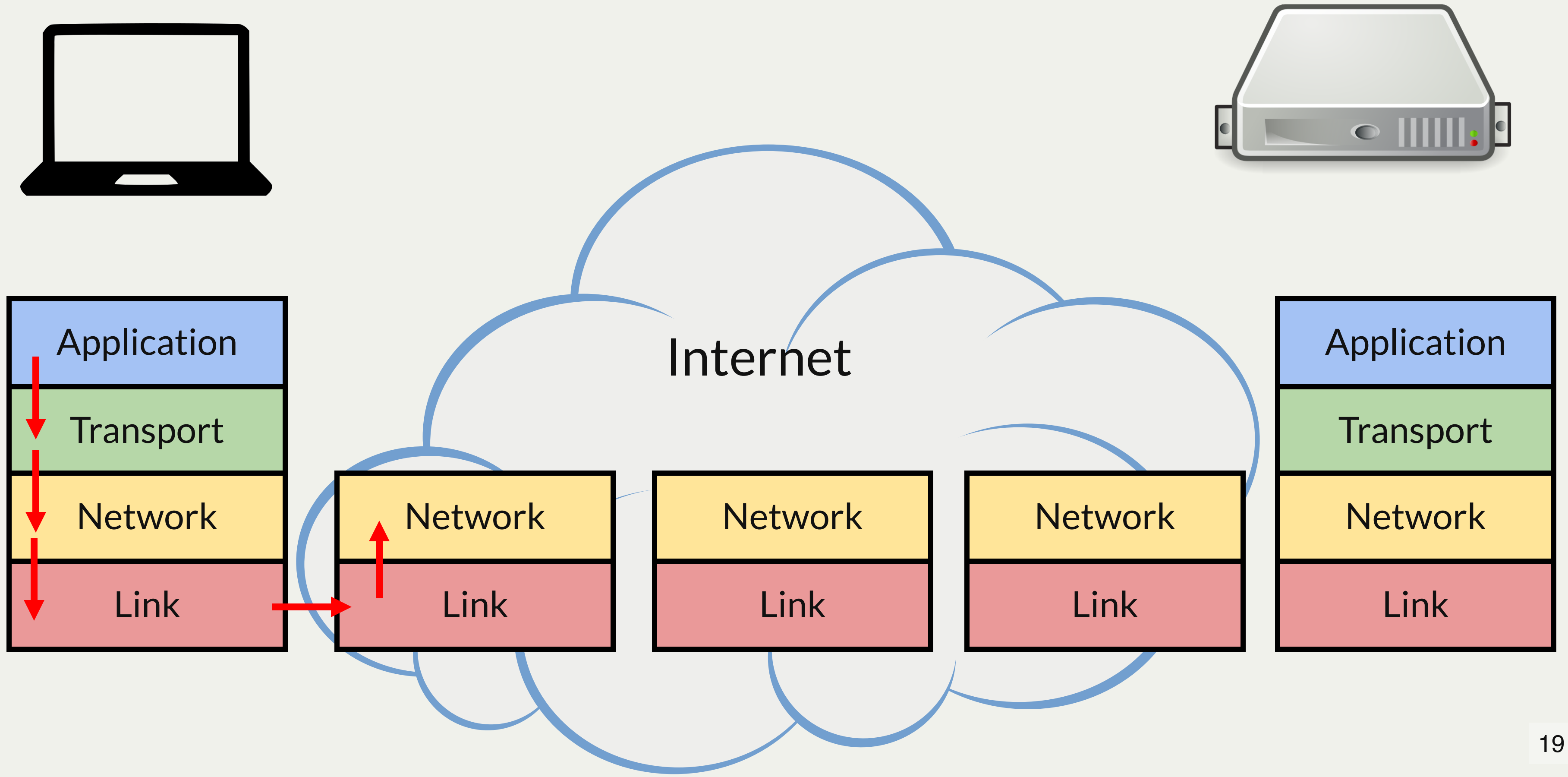
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



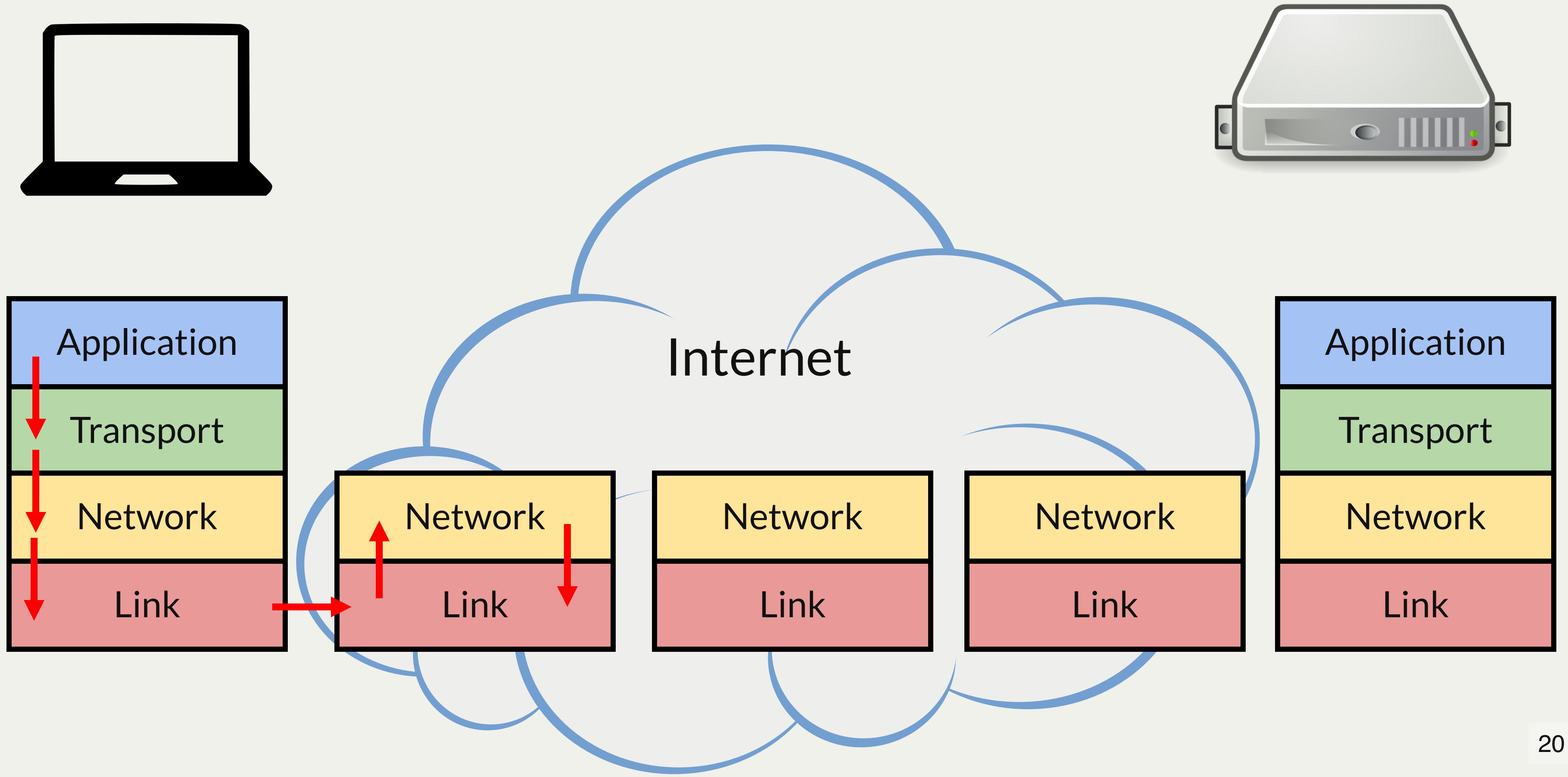
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



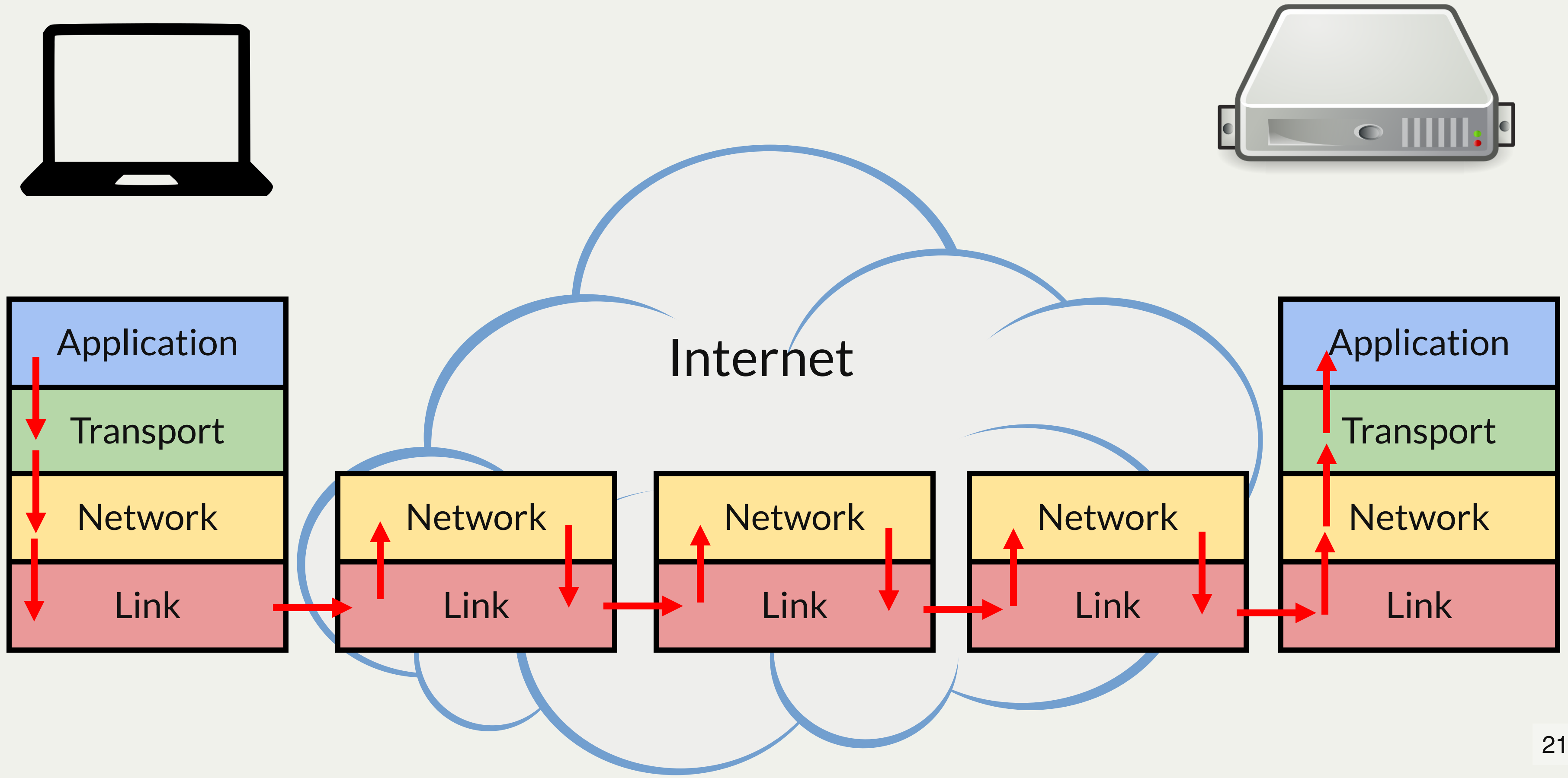
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



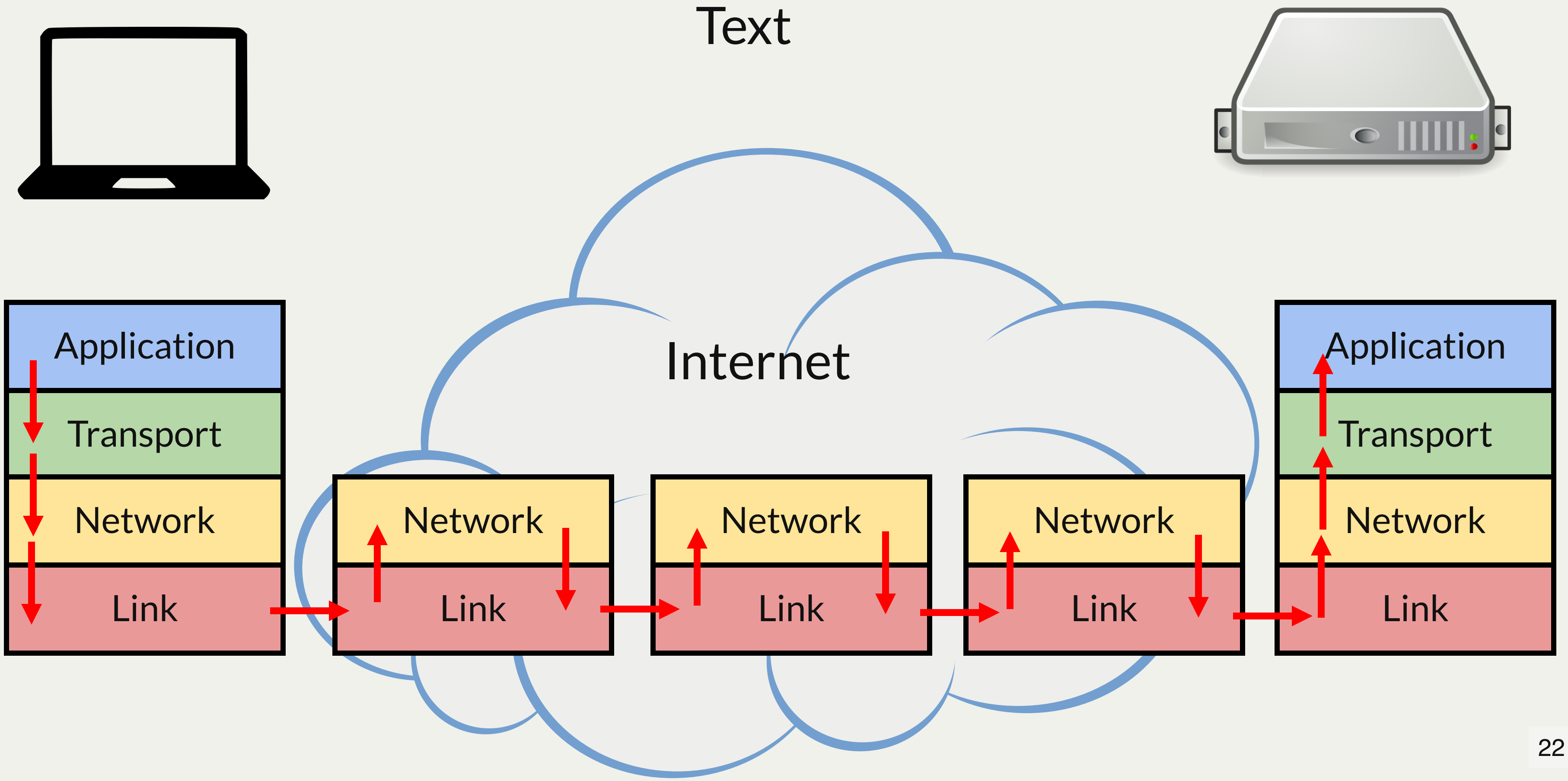
Internet in 4 Layers

- Application layer: name of document (e.g., index.html)
- Transport layer: port identifying what service (e.g., HTTP on port 80)
- Network layer: global IP address identifying which computer (e.g., 171.67.76.12)
- Link layer: MAC address specifying which device on a link (e.g., 00:25:90:e7:10:48)



Hop by Hop

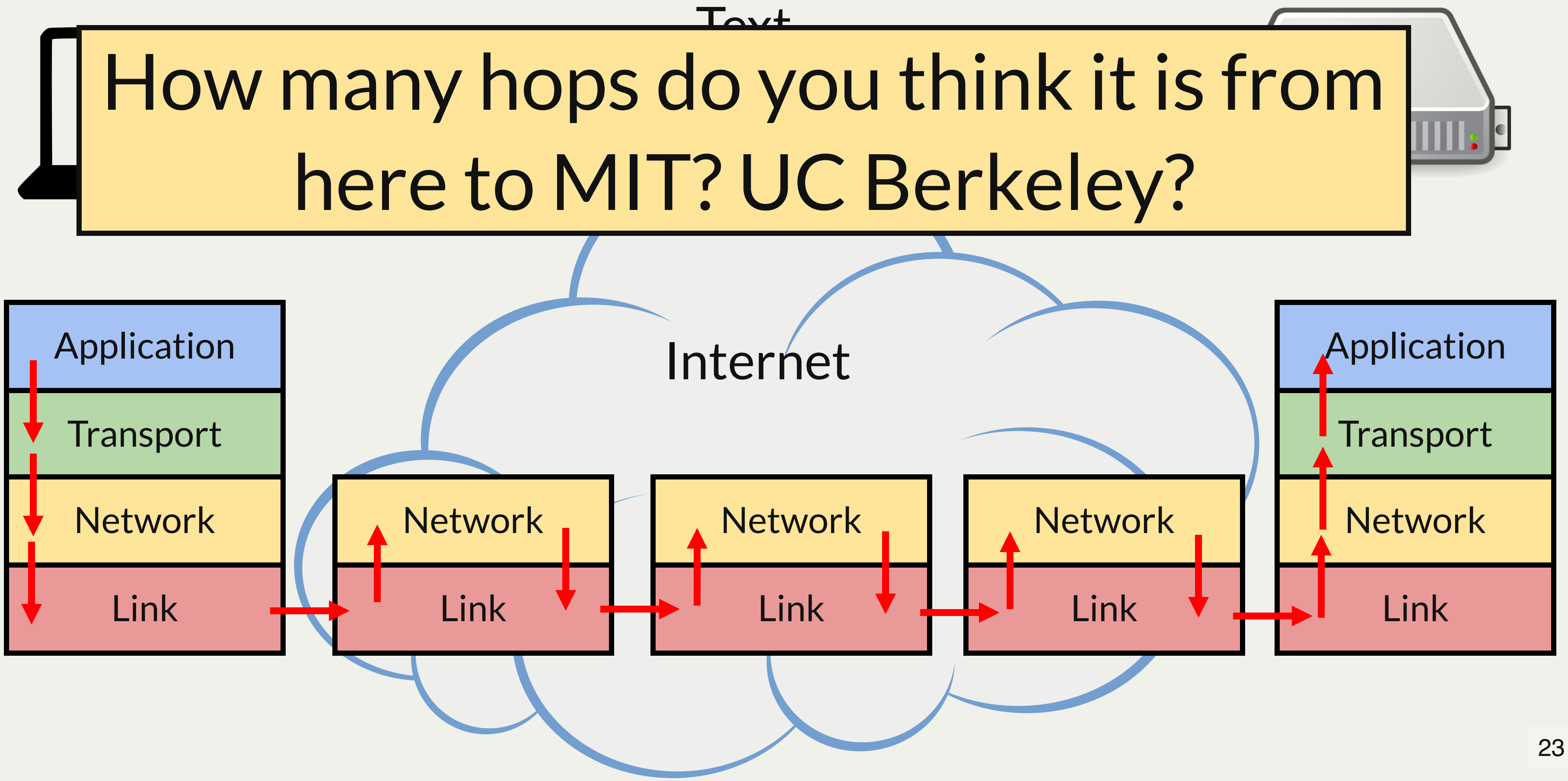
- A packet from one node to another takes many hops to get to its destination
- At each hop, the packet is sent to the link layer address of the next hop
- Each hop checks the IP (network) address to see if it's the final destination
    - If no, forward to next hop using a new link layer destination
    - If yes, bubble up to transport and potentially application layers



Hop by Hop

- A packet from one node to another takes many hops to get to its destination
- At each hop, the packet is sent to the link layer address of the next hop
- Each hop checks the IP (network) address to see if it's the final destination
  - If no, forward to next hop using a new link layer destination
  - If yes, bubble up to transport and potentially application layers

How many hops do you think it is from here to MIT? UC Berkeley?



7 Layer Model of Networks

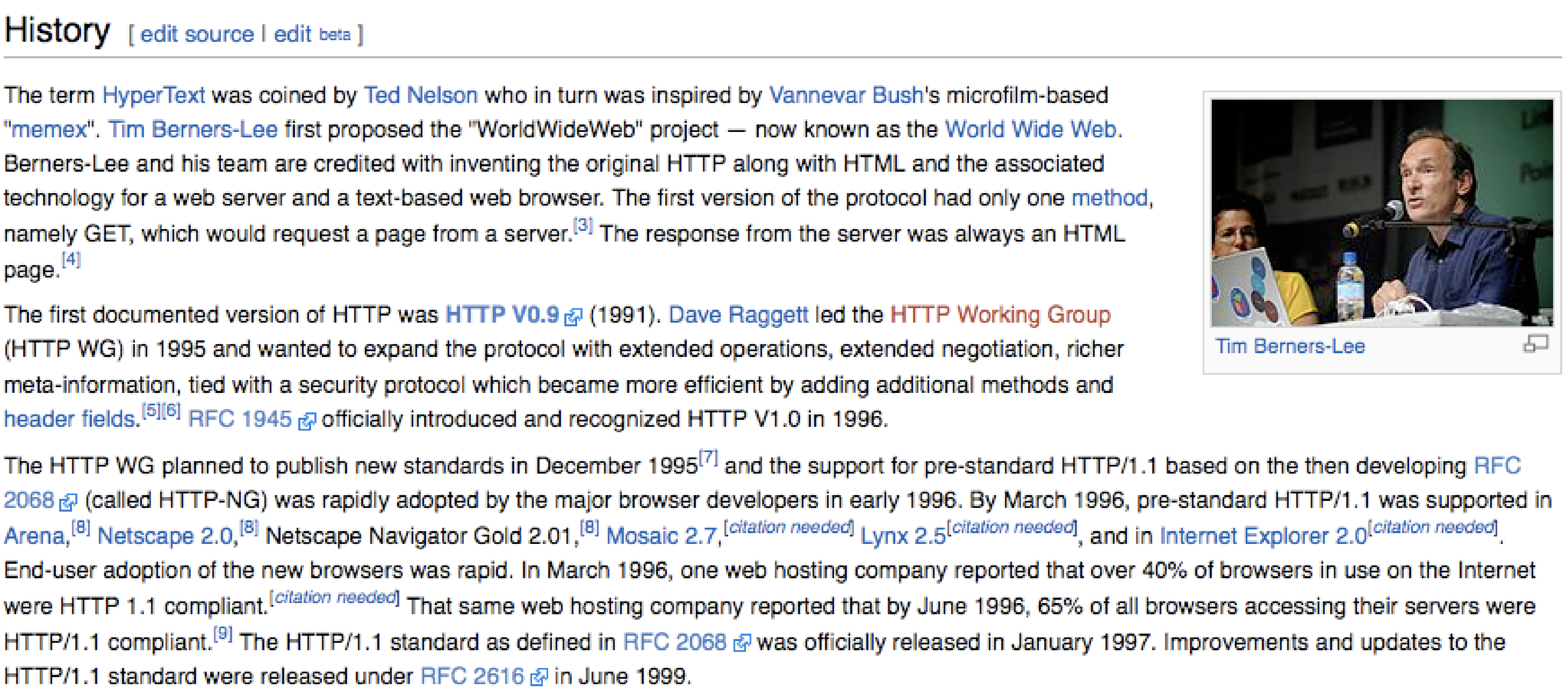
Application	Application level data: HTTP
Presentation	Let's not worry about this one
Session	A communication session: TLS
Transport	Application: TCP, UDP ports
Network	Node: IP address
Link	A network interface: MAC
Physical	How you encode bits

Layering and Modularity

- One of the seven principles of system design
- Layering: decomposing a system into a well defined set of services, each with a clear interface it provides above and an interface it uses below
  - Internet and network protocols are the canonical example
- Transmission Control Protocol (TCP)
  - Provides a bidirectional, reliable, in-order byte stream above
  - Uses an unreliable datagram protocol below
- If layers meet these interfaces very precisely, they are *modular*: they can be interchanged
  - TCP has many variants: NewReno, CUBIC, Compound, DCTCP
  - Your application can use any of them, they all provide the same interface
  - They all use the Internet Protocol: IP
  - They are interchangeable and so are modular
  - TLS (Transport Layer Security) is a session layer protocol that secures TCP: it looks likes a bidrectional, reliable, in-order byte stream, so it can often be swapped in for TCP (e.g., https: instead of http:)

AMA about networks

Example Layer 7 Protocol: HyperText Transfer Protocol (HTTP)



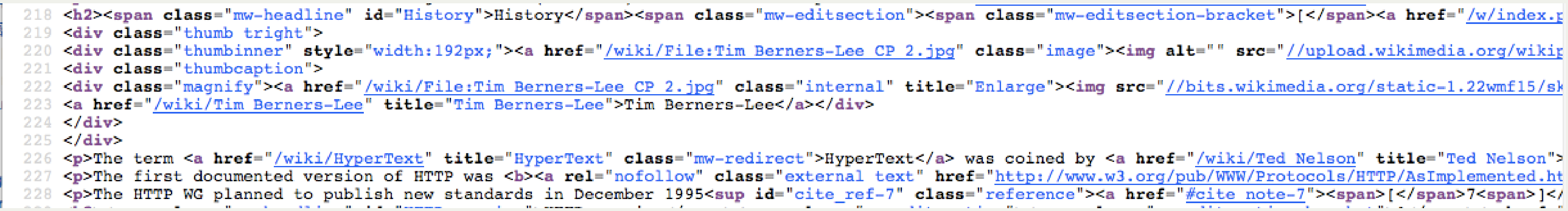
History [edit source | edit beta]

The term HyperText was coined by Ted Nelson who in turn was inspired by Vannevar Bush's microfilm-based "memex". Tim Berners-Lee first proposed the "WorldWideWeb" project — now known as the World Wide Web. Berners-Lee and his team are credited with inventing the original HTTP along with HTML and the associated technology for a web server and a text-based web browser. The first version of the protocol had only one method, namely GET, which would request a page from a server.[3] The response from the server was always an HTML page.[4]

Tim Berners-Lee

The first documented version of HTTP was **HTTP V0.9** (1991). Dave Raggett led the HTTP Working Group (HTTP WG) in 1995 and wanted to expand the protocol with extended operations, extended negotiation, richer meta-information, tied with a security protocol which became more efficient by adding additional methods and header fields.[5][6] RFC 1945 officially introduced and recognized HTTP V1.0 in 1996.

The HTTP WG planned to publish new standards in December 1995[7] and the support for pre-standard HTTP/1.1 based on the then developing RFC 2068 (called HTTP-NG) was rapidly adopted by the major browser developers in early 1996. By March 1996, pre-standard HTTP/1.1 was supported in Arena,[8] Netscape 2.0,[8] Netscape Navigator Gold 2.01,[8] Mosaic 2.7,[*citation needed*] Lynx 2.5[*citation needed*], and in Internet Explorer 2.0[*citation needed*]. End-user adoption of the new browsers was rapid. In March 1996, one web hosting company reported that over 40% of browsers in use on the Internet were HTTP 1.1 compliant.[*citation needed*] That same web hosting company reported that by June 1996, 65% of all browsers accessing their servers were HTTP/1.1 compliant.[9] The HTTP/1.1 standard as defined in RFC 2068 was officially released in January 1997. Improvements and updates to the HTTP/1.1 standard were released under RFC 2616 in June 1999.

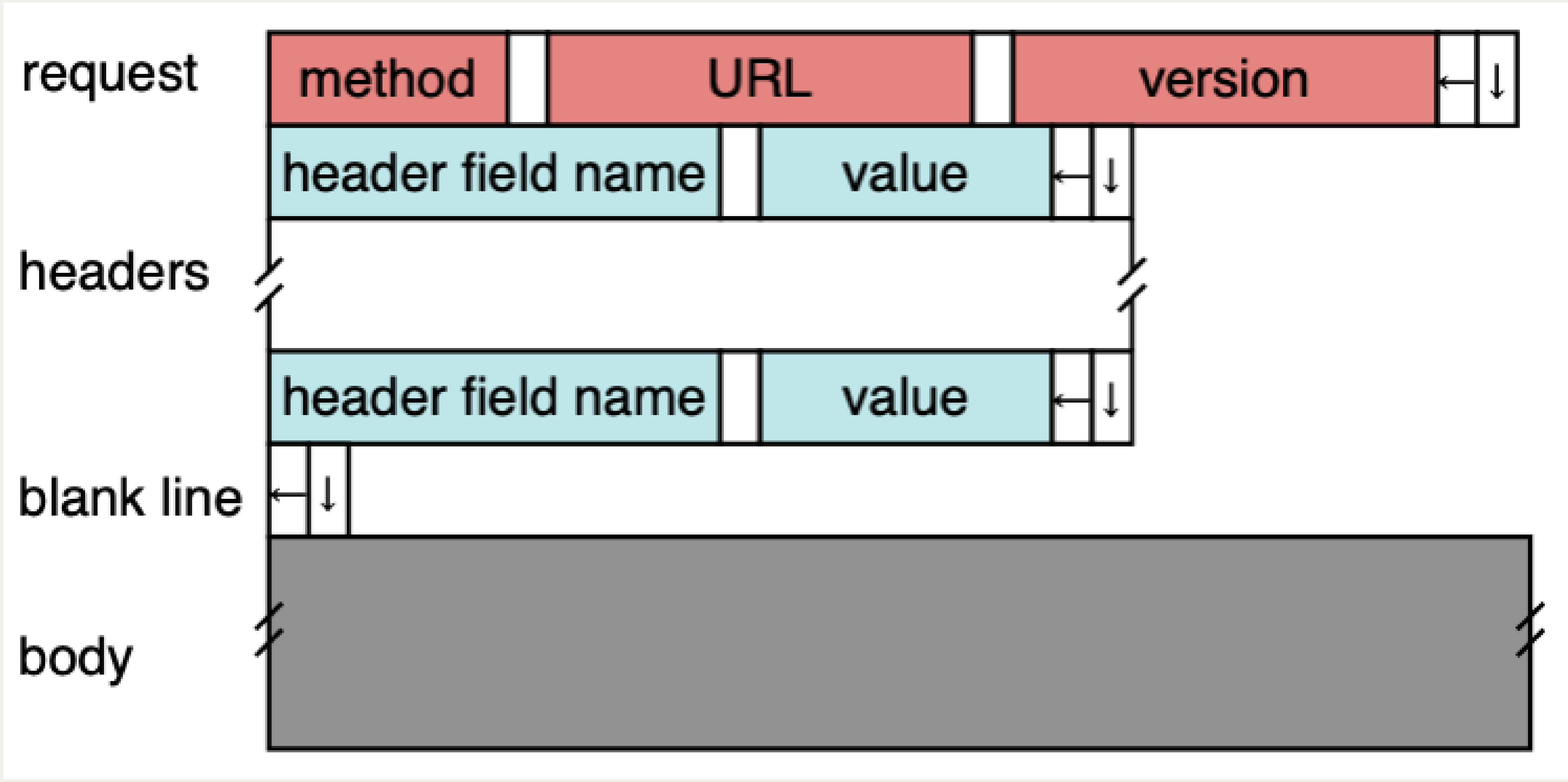


```
<h2><span class="mw-headline" id="History">History</span><span class="mw-editsection"><span class="mw-editsection-bracket">[</span><a href="/w/index.p
<div class="thumb tright">
<div class="thumbinner" style="width:192px;"><a href="/wiki/File:Tim Berners-Lee CP 2.jpg" class="image"><img alt="" src="//upload.wikimedia.org/wikip
<div class="thumbcaption">
<div class="magnify"><a href="/wiki/File:Tim Berners-Lee CP 2.jpg" class="internal" title="Enlarge"><img src="//bits.wikimedia.org/static-1.22wmf15/sk
<a href="/wiki/Tim Berners-Lee" title="Tim Berners-Lee">Tim Berners-Lee</a></div>
</div>
</div>
<p>The term <a href="/wiki/HyperText" title="HyperText" class="mw-redirect">HyperText</a> was coined by <a href="/wiki/Ted Nelson" title="Ted Nelson">
<p>The first documented version of HTTP was <b><a rel="nofollow" class="external text" href="http://www.w3.org/pub/WWW/Protocols/HTTP/AsImplemented.ht
<p>The HTTP WG planned to publish new standards in December 1995<sup id="cite_ref-7" class="reference"><a href="#cite_note-7"><span>[</span>7<span>]</
```

HTTP/1.x Properties

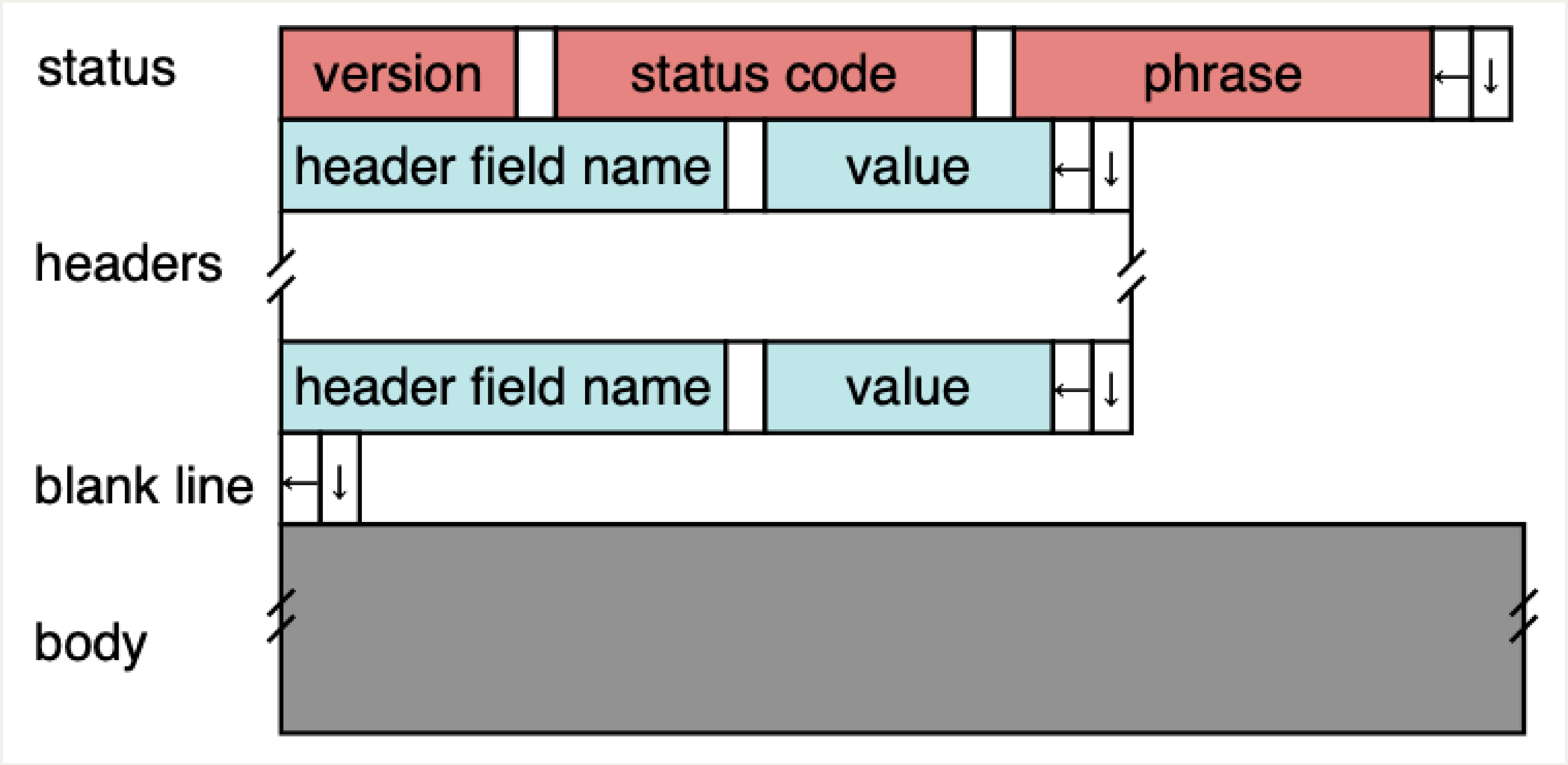
- HTTP is an ASCII protocol: protocol is in text, not binary values
  - In contrast to IP and TCP, which use binary: a 32-bit IP address is a 32-bit number
  - Makes everything much simpler and easier to debug!
  - ASCII is expensive, but if you're sending large files it's a small overhead
  - HTTP/2.0 is binary, trying to optimize things for mobile devices
- Basic model: request, response

HTTP/1.x Request



- Three principal methods
  - GET: fetch a resource (page, image, etc.)
  - HEAD: fetch only the headers (useful to see if the resource has changed)
  - POST: send data

HTTP/1.x Response

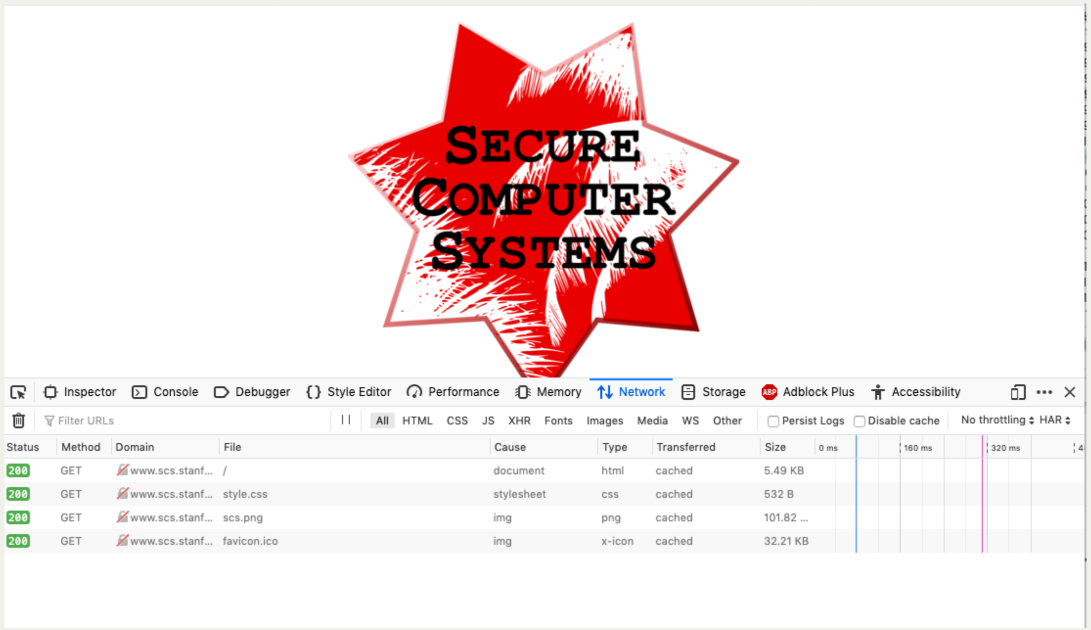


- Five classes of status codes
  - 1xx: hold on
  - 2xx: here you go
  - 3xx: go away
  - 4xx: you screwed up
  - 5xx: I screwed up

Let's Look at a Request/Reponse in telnet and Firefox

```
partysaur:~ pal$ telnet www.scs.stanford.edu 80
Trying 2001:470:806d:1::9...
Connected to www.scs.stanford.edu.
Escape character is '^]'.
GET / HTTP/1.1
Host: www.scs.stanford.edu

HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 5624
```

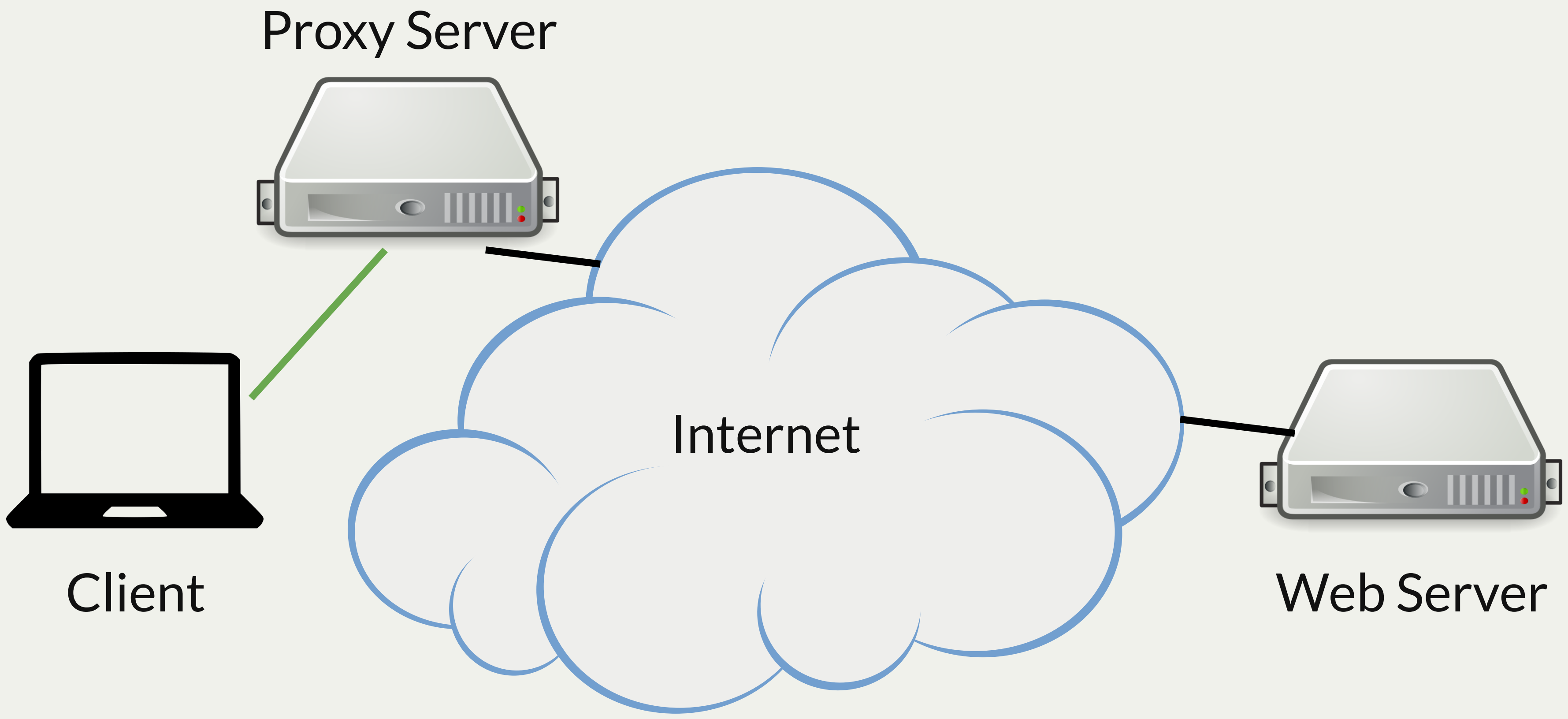


Status	Method	Domain	File	Cause	Type	Transferred	Size
200	GET	www.scs.stanf...	/	document	html	cached	5.49 KB
200	GET	www.scs.stanf...	style.css	stylesheet	css	cached	532 B
200	GET	www.scs.stanf...	scs.png	img	png	cached	101.82 ...
200	GET	www.scs.stanf...	favicon.ico	img	x-icon	cached	32.21 KB

HTTP Web Proxy

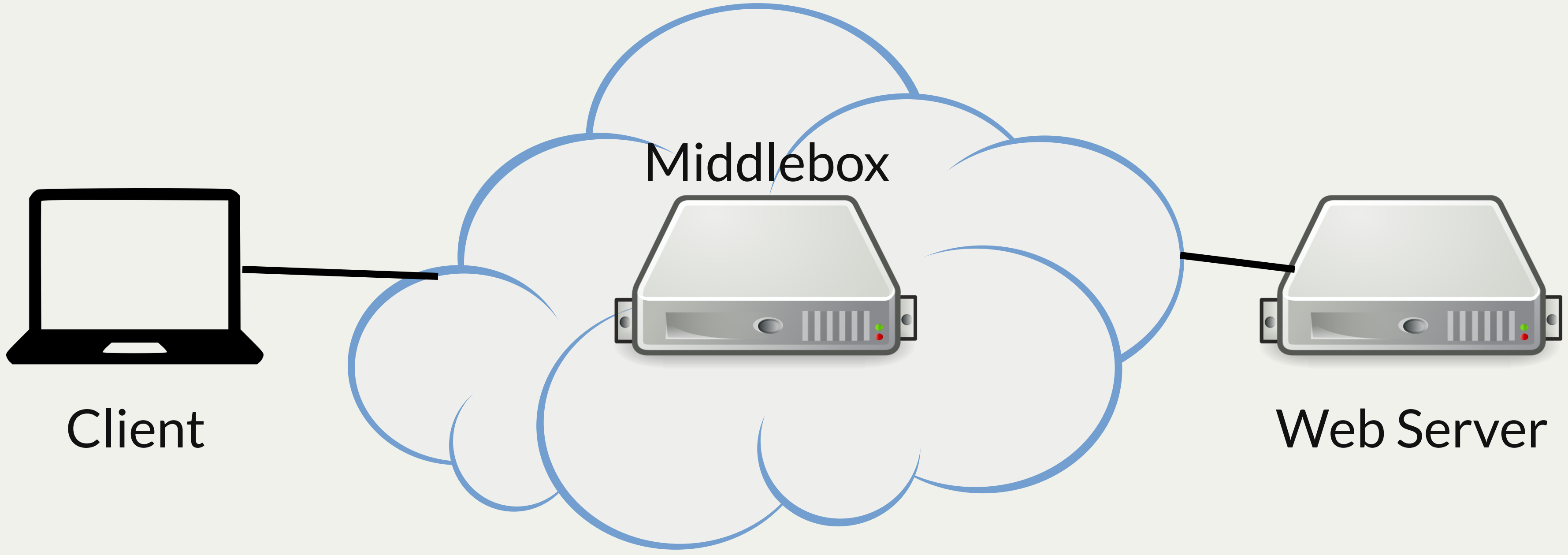
- A *proxy server* acts as a go-between a node and other nodes on the Internet
  - Clients opt-in to a proxy server (we'll see in a moment)
  - A *web proxy* handles client's HTTP requests and responds to them
- Proxy servers can be useful
  - Cache requests for static data (e.g., images) so it can later serve local copies rather than re-request from the web (happens a lot in mobile devices), welcome to 1999
  - Intercept image requests to scale them down/make them more efficient (e.g., Google Flywheel)
- If clients don't opt-in (it happens automatically), it's called a *middlebox*
  - Redirect to a paywall (e.g., what happens at airports)
  - Require agreeing to terms of service (e.g., Stanford Visitor)
  - Block access to certain websites (e.g., YouTube)
  - Block access to certain documents (big documents, .zip files, video streams.)

Web Proxy



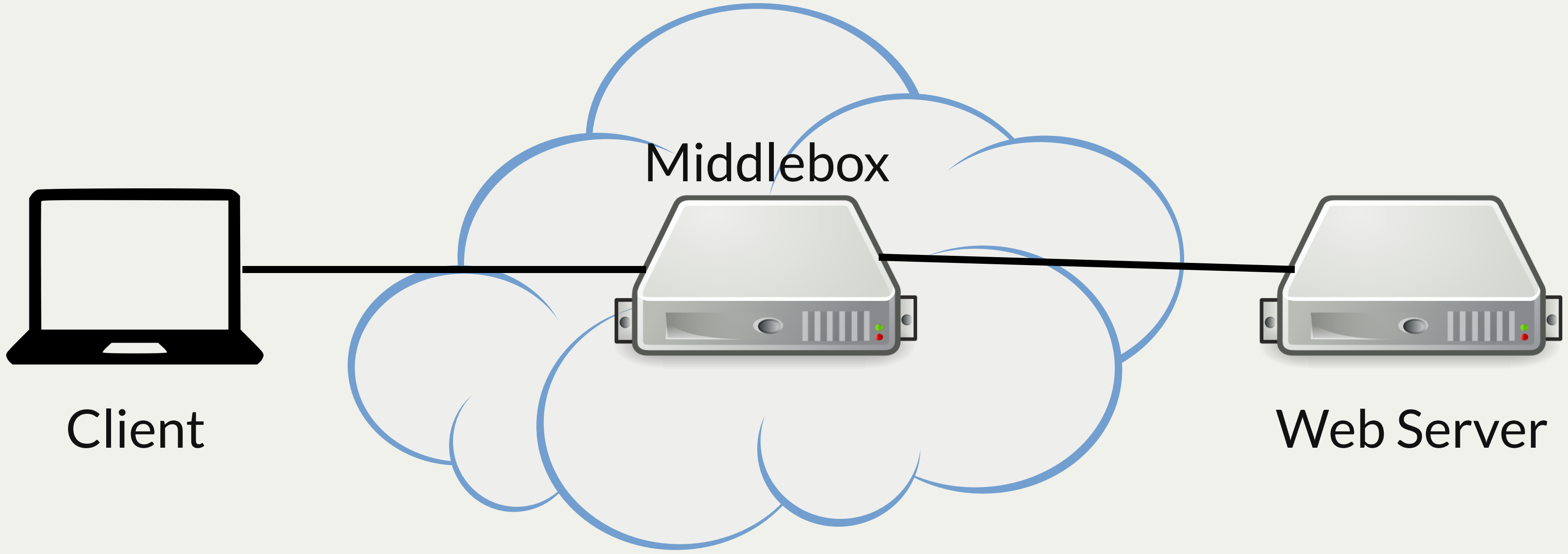
- Client explicitly asks proxy server to fetch web page for it
  - Proxy requests it from the server (or maybe another proxy server), maybe modifies it
  - Or, proxy gives a local cached copy

Middlebox



- Client thinks it's requesting from the web server
    - Middlebox interposes, responds to request looking like web server
- The difference between proxy and middlebox is the IP address to which the client sends its request. With a proxy, the client sends requests to the proxy. With a middlebox, it sends requests to the web server and the middlebox masquerades as the web server.

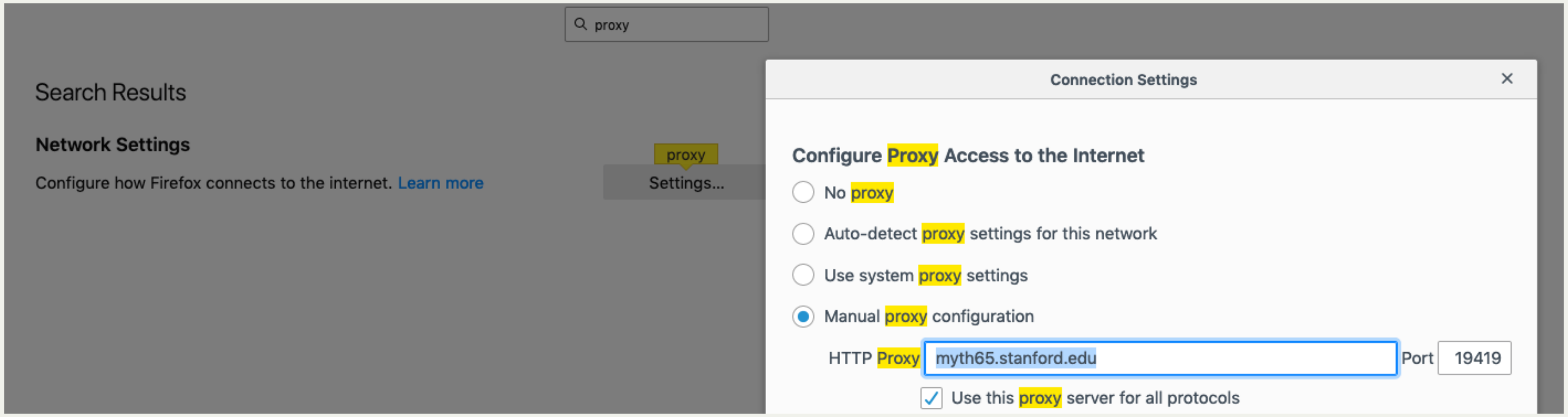
Middlebox



- Client thinks it's requesting from the web server
    - Middlebox interposes, responds to request looking like web server
- The difference between proxy and middlebox is the IP address to which the client sends its request. With a proxy, the client sends requests to the proxy. With a middlebox, it sends requests to the web server and the middlebox masquerades as the web server.

Next Assignment: HTTP Web Proxy

- We have built a very basic proxy for you, which you are going to turn into a full web proxy
- To test it, you need to set up Firefox (or another browser, but we suggest Firefox) to forward all web requests to the proxy:



Network Settings

Configure how Firefox connects to the internet. Learn more

Configure Proxy Access to the Internet

- ○ No proxy
- ○ Auto-detect proxy settings for this network
- ○ Use system proxy settings
- ● Manual proxy configuration

HTTP Proxy: myth65.stanford.edu Port: 19419

✓ Use this proxy server for all protocols

- To set up Firefox, go to Firefox->Preferences, then type "proxy" in the search box, then click on settings. You should have a window as above. Then, click on "Manual proxy configuration," and type in the myth machine and port number you get after starting your proxy:

```
$ ./proxy
Listening for all incoming traffic on port 19419.
```

- Make sure you also select the checkbox for "Use this proxy server for all protocols."

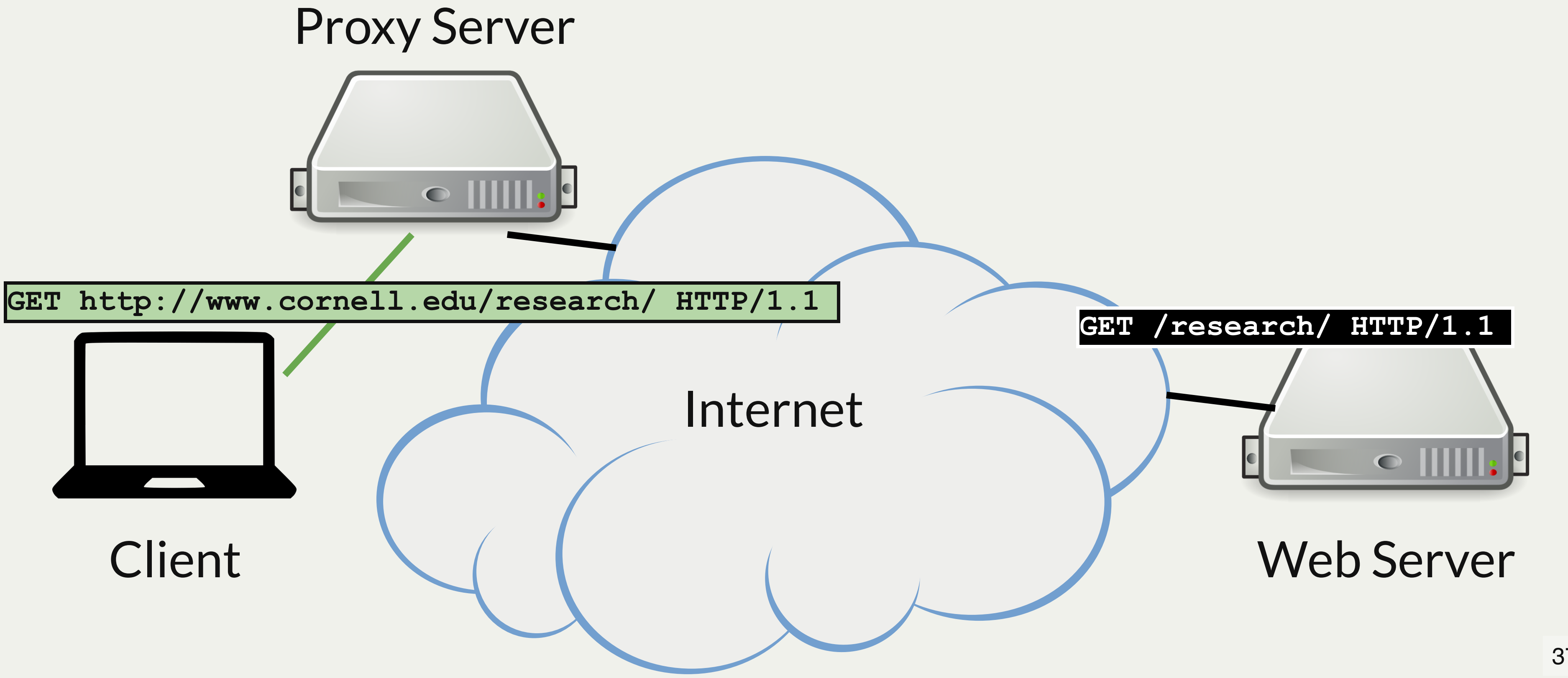
Web Proxy Requests

- Proxy requests use the full URL:

```
GET http://www.cornell.edu/research/ HTTP/1.1
```

- When a proxy server receives such as request, it forwards the request to www.cornell.edu, with the first line rewritten as follows:

```
GET /research/ HTTP/1.1
```



HTTP Web Proxy Headers

- Properly proxying requires a couple of extra headers
- **x-forwarded-for**: allows the proxy to tell the server who is making the request
  - Should be filled in with the IP address of the requesting client
  - If x-forwarded-for is already present, extend its value into a comma-separated chain of IP addresses the request has passed through before arriving at your proxy. The IP address of the machine you're directly hearing *from* would be appended to the end.
    - E.g., if your proxy receives a request from 172.27.76.12 with

```
x-forwarded-for: 172.27.76.96
```

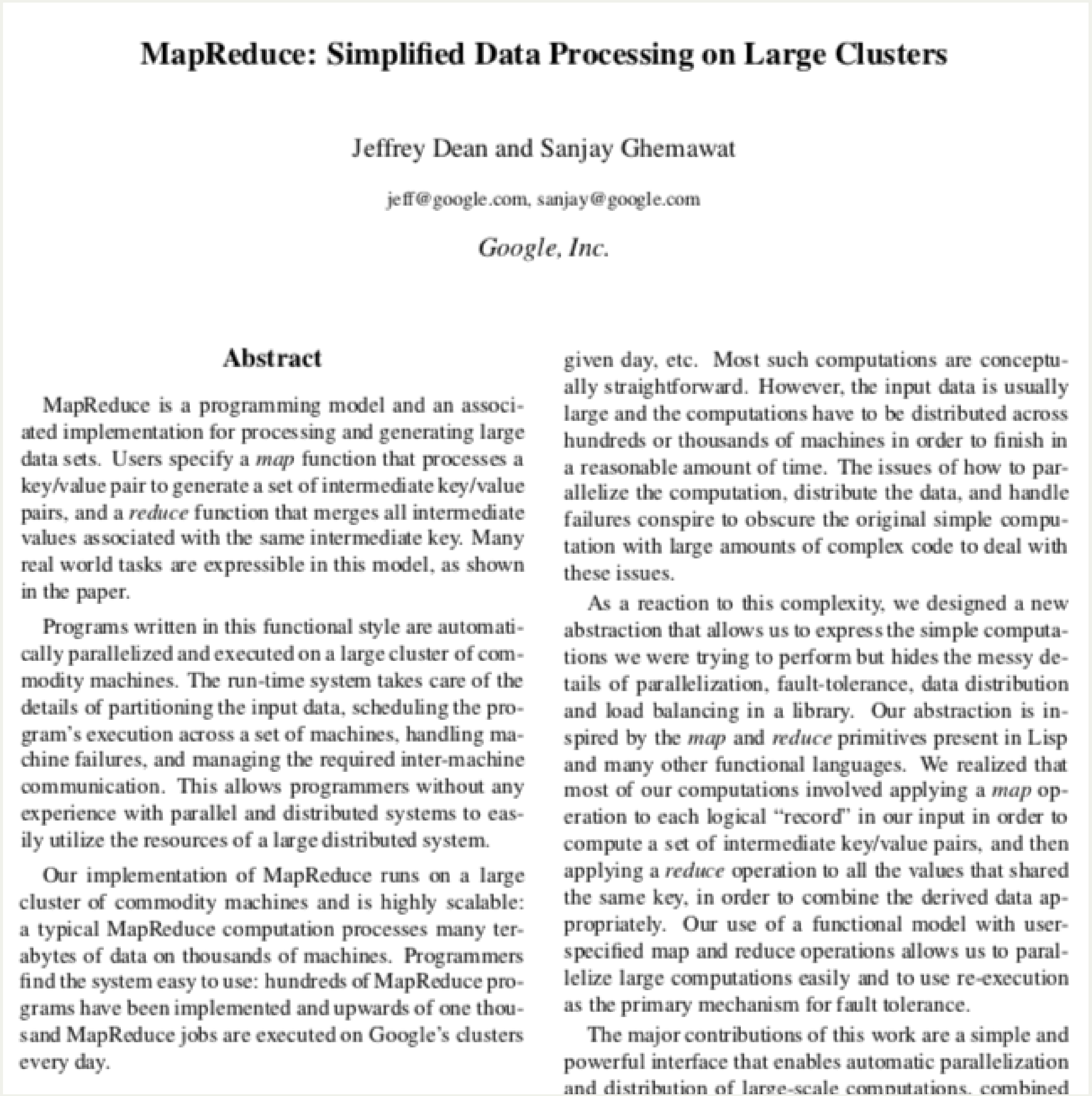
    - it will rewrite it to be

```
x-forwarded-for: 172.27.76.96, 172.27.76.12
```

- **x-forwarded-proto**: allows a server sitting behind a load-balancing proxy to be sure that the client request to the proxy was secure (https: instead of http:)
  - We're not going to worry about this use case, but we have to support it
  - Set its value to be **http**. If **x-forwarded-proto** is already included in the request header, then simply add it again.
  - Here's a nice explanation

AMA about HTTP and proxies

The Genesis of Datacenter Computing: MapReduce



MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined

The Genesis of Datacenter Computing: MapReduce

- Problem for large service providers such as Google: computation requires hundreds or thousands of computers
  - How do you distribute the computation?
  - Distributing the computation boils down to distributing data
  - Nodes fail, some nodes are slow, load balancing: difficult to make it work well
- System came from observation that many early Google systems had a common pattern
- Designing a computing system around this pattern
  - allows the system (written once) to do all of the hard systems work (fault tolerance, load balancing, parallelization)
  - allows tens of thousands of programmers to just write their computation
- Canonical example of how the right *abstraction* revolutionized computing
  - An open source version immediately appeared: Hadoop

Core Data Abstraction: Key/Value Pairs

- Take a huge amount of data (more than can fit in the memory of 1000 machines)
- Write two functions:

```
map(k1, v1) -> list(k2, v2)
reduce(k2, list(v2)) -> list(v2)
```

- Using these two functions, MapReduce parallelizes the computation across thousands of machines, automatically load balancing, recovering from failures, and producing the correct result.
- You can string together MapReduce programs: output of reduce becomes input to map.
- Simple example of word count (wc):

```
map(String key, String value):
  // key: document name
  // value: document contents
  for word w in value:
    EmitIntermediate(w,"1")

reduce(String key, List values):
  // key: a word
  // values: a list of counts
  int result = 0
  for v in values:
    result += ParseInt(v)
  Emit(AsString(result))
```

input "The number of partitions (R) and the partitioning function are specified by the user."

map output ("the", "1") , ("the", "1"), ("The", "1"), ("of", 1), (number, "1"), ...

reduce
"the", ("1", "1") -> "2"
"The", ("1") -> "1"
"of", ("1") -> "1"
"number", ("1") -> "1"

Key/Value Pairs: How and Where

- Keys allow MapReduce to distribute and parallelize load

```
map(k1, v1) -> list(k2, v2)
reduce(k2, list(v2)) -> list(v2)
```

- In the original paper...
  - Each *mapper* writes a local file for each key in k2, and reports its files to a *master* node
  - The *master* node tells the *reducer* for k2 where the all of the k2 files are
  - The reducer reads all of the k2 files from the nodes that ran the mappers and writes its own output locally, reporting this to the master node
  - There have been lots of optimizations since
- Core abstraction: data can be partitioned by key, there is no locality between keys
- One problem: how much do you parallelize load?
  - One map per input file, one reduce per key, is too fine grained (small input)
  - Split the key space into R partitions: partition = (hash(key) % R)
  - Jeff Dean rule of thumb: if you have N nodes, R >= 10N

MapReduce System Architecture

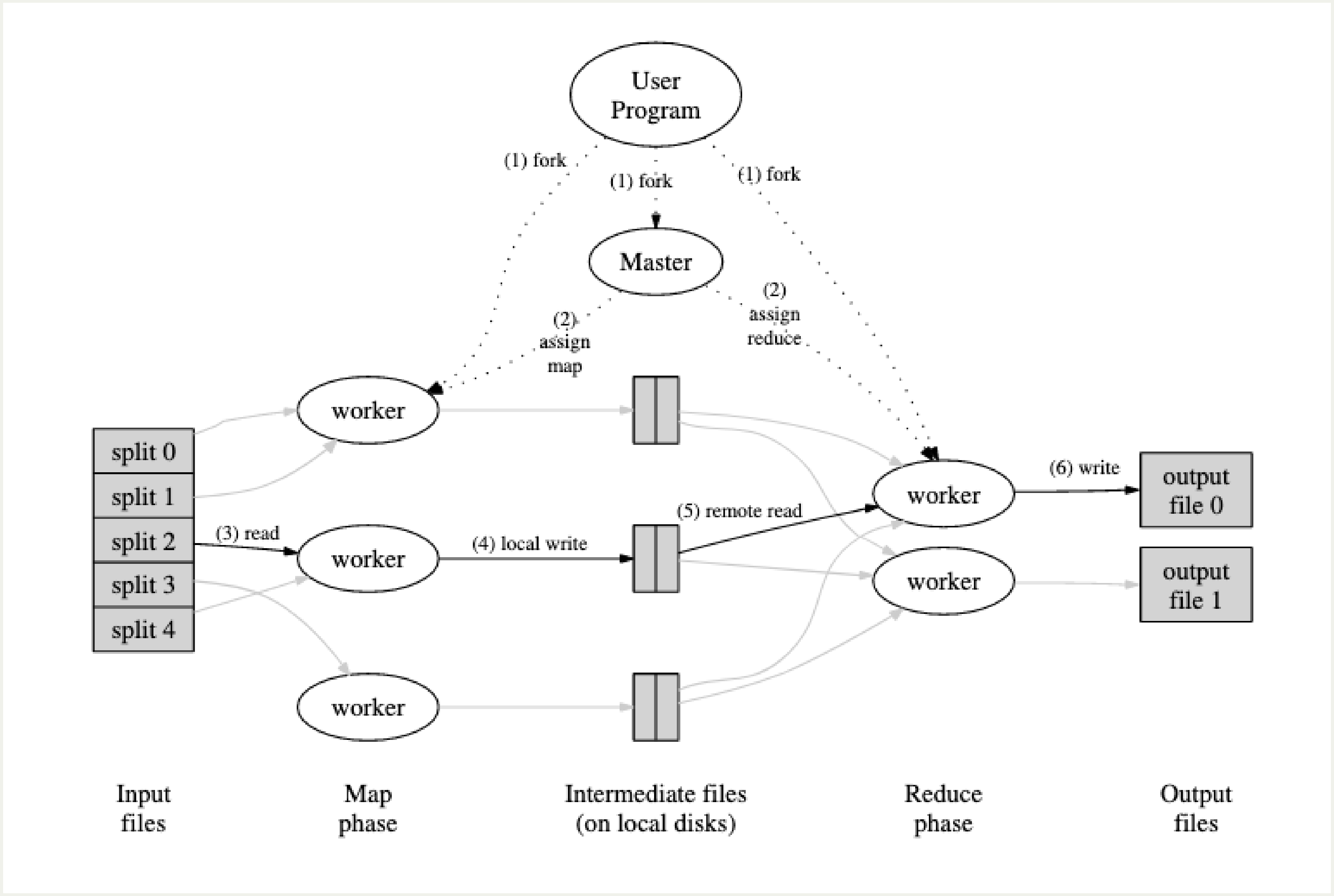
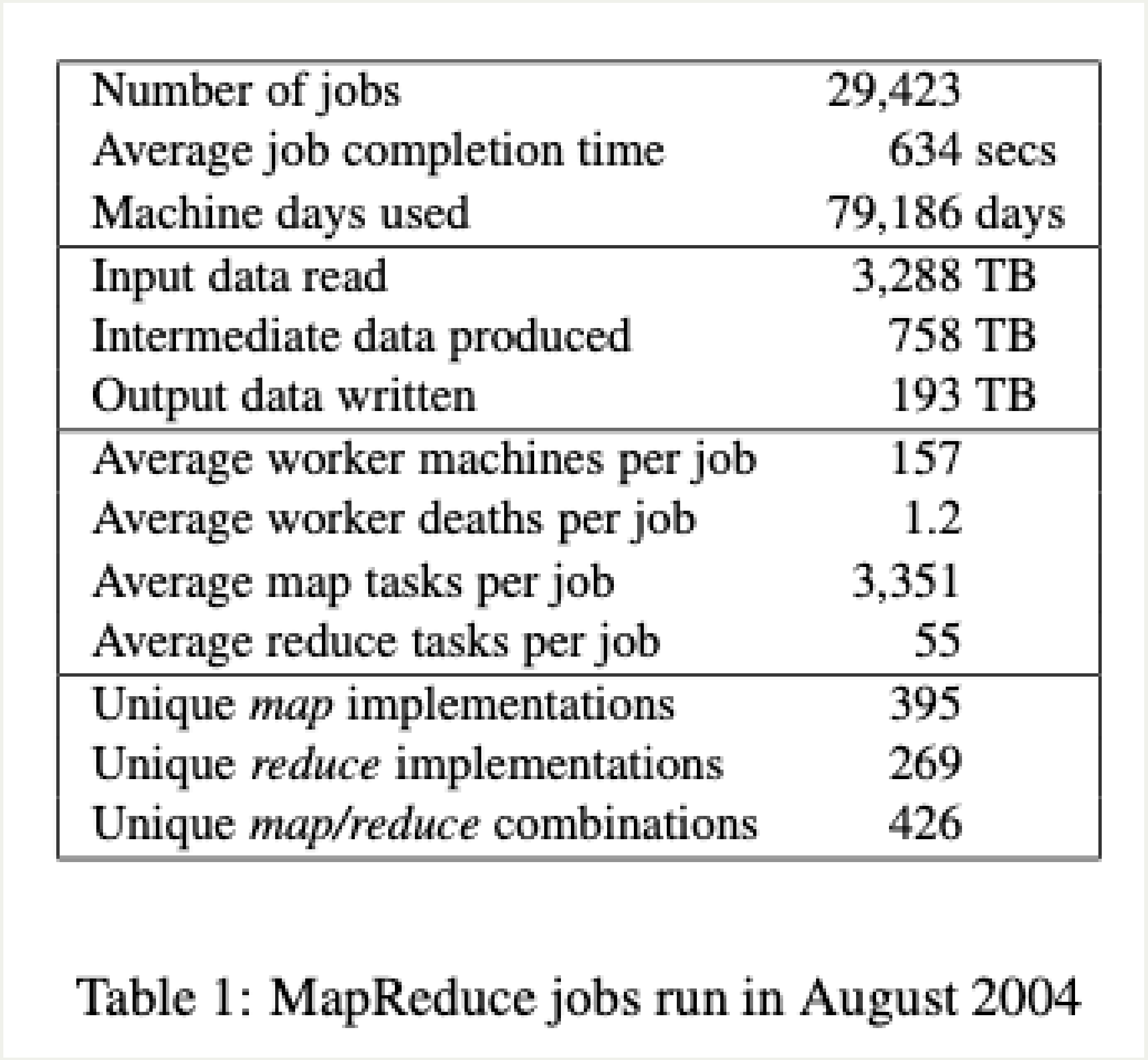


Table From Paper (August 2004)



Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

Google's IPO was September 2004

AMA about MapReduce