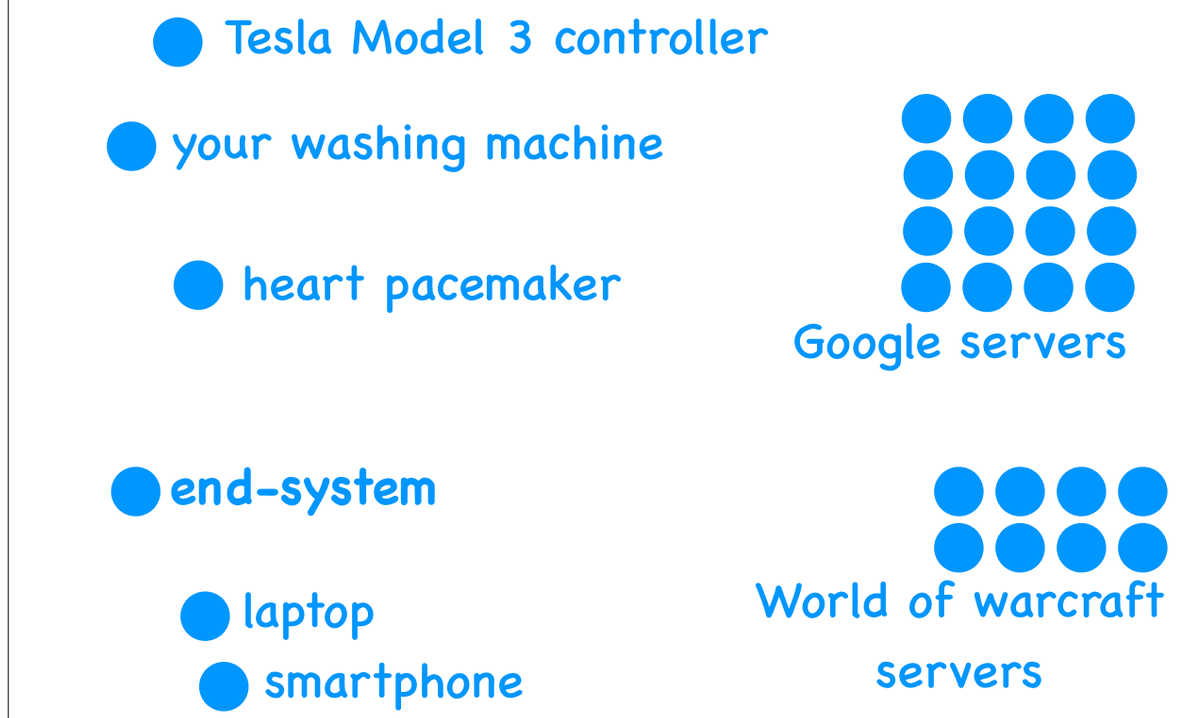
Lecture 3:

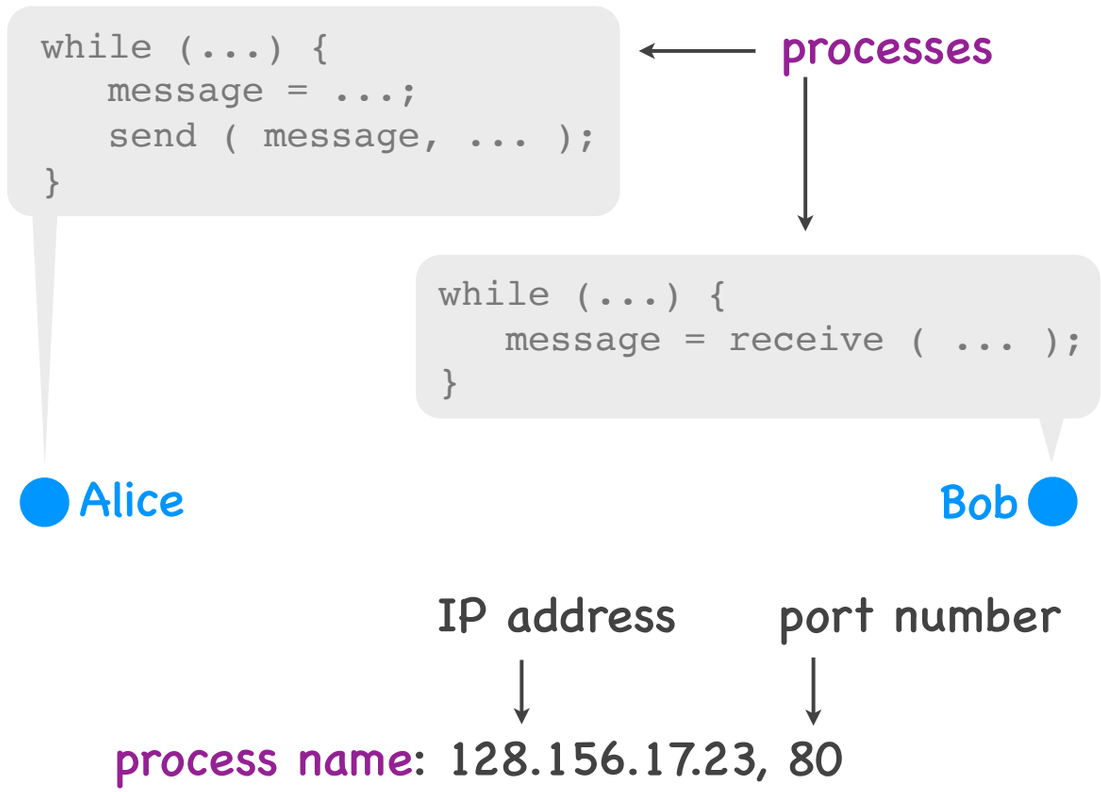
The Application Layer

Katerina Argyraki, EPFL

In the last two lectures, we did a summary of Internet operation.
Now it's time to explore the layers of the Internet architecture, starting from the Application layer.



We said that the goal of the Internet is to enable end-systems to run distributed applications like web browsing and online gaming.



We also said that a distributed application consists of pieces of code running on multiple end-systems at the same time,
and that these pieces of code communicate, exchange messages, using a well defined Application Programming Interface (API).

We will be talking a lot about “pieces of code running on an end-system”, so we need to give them a name. We call them “processes”.
Informally, a process is a piece of code that is running on an end-system and belongs to the application layer.

In this particular example, there is one process running on Alice’s computer,
and a second process running on Bob’s computer,
and the two processes are communicating.

To do that, they need a way to reach each other, and for that they use process names.

We already saw what a process name looks like in the first lab:

- There is a first part that identifies a network interface. This could be a DNS name or an IP address. However, when processes communicate with each other, they use IP addresses, not DNS names to identify network interfaces (DNS names are for humans).
- And there is a second part in a process name that identifies a particular process, behind the specified network interface. This is called a port number.

When you look at a process name like the one I am showing you here, you can think of the IP address as a street address and the port number as an apartment number.

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport-layer technology
  - what will the app expect from the transport layer?

Designing a distributed application involves at least the following tasks:
- designing the architecture of the application,
- designing the communication protocol,
- and choosing the transport-layer technology.

We will first discuss some of these problems in general,
and then we will look at specific application examples.

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport-layer technology
  - what will the app expect from the transport layer?

We said that, in a distributed application, we have multiple processes running on different end-systems.
So, the first question is

- how many processes will our application involve
- and what will each process do.

One popular approach is the…

a process that is always running

reachable at a fixed,
known process name

answers service requests

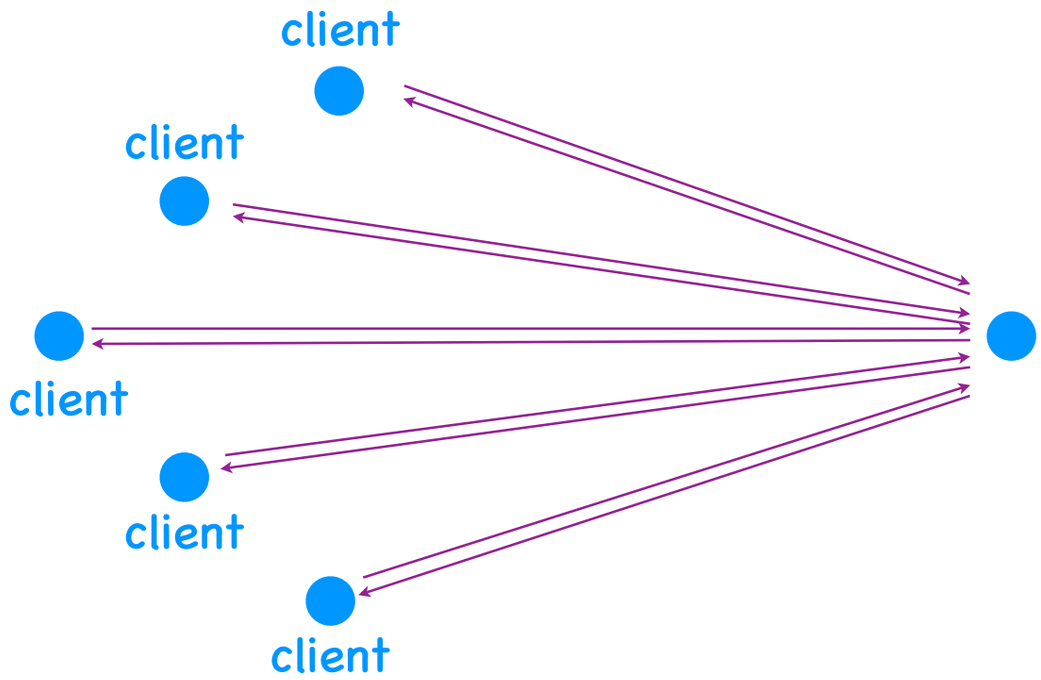
a process that generates
service requests

… client-server architecture, where we have client processes and server processes.
The client is a process that generates requests for service.
The server is a process that is

- always running on some computer,
- is reachable at a fixed, known process name,
- and answers service requests.

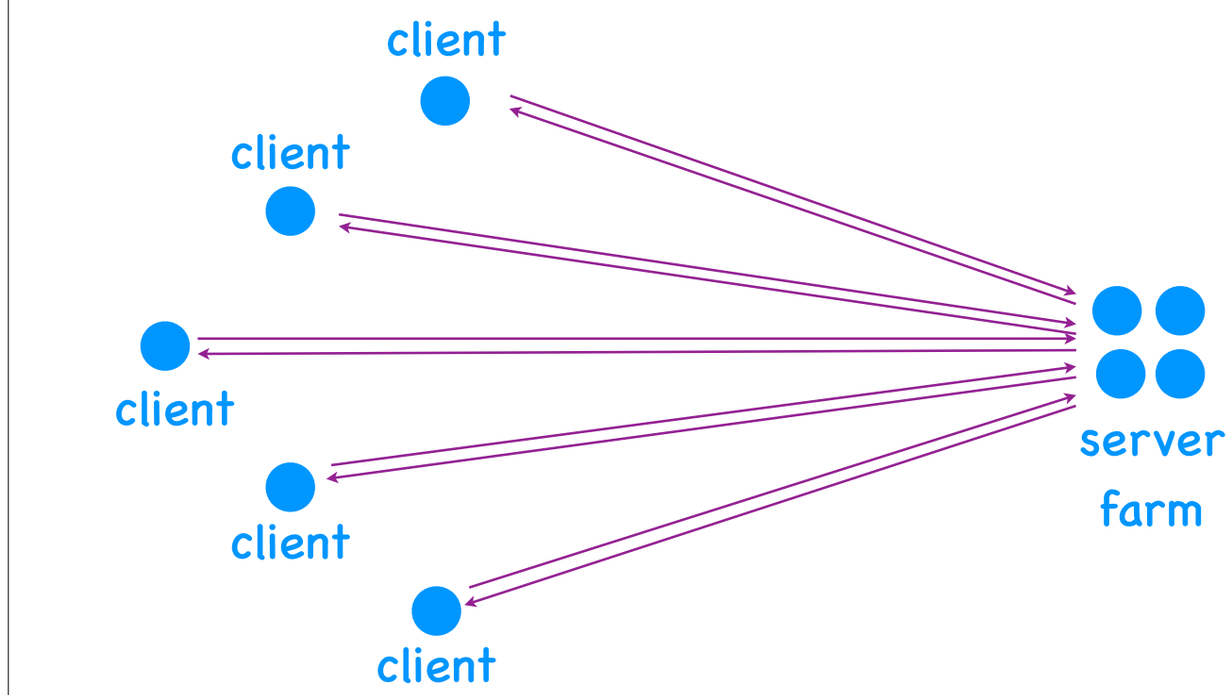
By the way, people often use the term “server” to refer not only to the server process, but to an entire computer that is running server processes in its application layer.

When an application uses the client/server architecture,
we usually end up with a picture like…



… this,
where multiple client processes are served by the same server process.
So, we typically have much fewer server processes than client processes running in the world.

Moreover, in the real world, the picture looks more like…



… this:
On the server side, we have multiple computers running the same process,
and they all answer at the same process name.

Client-server architecture

- Clear separation of roles
  - a client generates service requests
  - a server answers (or denies) the requests
- Server runs on dedicated infrastructure
  - could be one computer
  - or an entire data-center

So: In the client-server architecture, we have two key features: …

a process that may both
generate and answer requests

Another approach is the peer-to-peer architecture, where we have only peer processes.
A peer is a process that may both generate and answer requests.

Peer-to-peer architecture

- A peer may act as both server and client
  - generates service requests
  - answer (or deny) requests
- Runs on personally owned end-system
  - PC, laptop, smartphone
  - no dedicated infrastructure

So: In the peer-to-peer architecture, …

A special aspect of peer-to-peer applications is that there is typically no dedicated infrastructure.
The application is typically made up of (peer) processes running on people's laptops and smartphones around the world.

Client-server or peer-to-peer?

How should an application developer choose between these two architectures?

How did we say that we evaluate a network?
- we look at performance metrics, like delay and throughput
- and we also look at the kind of security that it offers.

I would like you to consider these two aspects and think: does the client/server or the P2P architecture offer better performance and better security?

Let's consider performance:
- Would you expect a client/server application or a peer-to-peer application to offer better performance?
The answer is not clear:
- In principle, peer-to-peer scales better, because all the processes that participate in the application contribute to serving each other's requests.
- In practice, client-server is more predictable, because the server process runs on dedicated infrastructure.

Consider a group of tourists wandering around in EPFL, trying to figure out where the Rolex center is.
- The equivalent of the client-server approach would be that there is a tourist office, in a well-known location, and all the tourists go there and ask.
- The equivalent of the peer-to-peer approach would be that the tourists exchange information with each other.
- Peer-to-peer scales better: even if we have 10000 tourists going around, when every one of them helps the others, most of them will eventually get the information they need.
- But it is less predictable: if you are a tourist, you may be unlucky and not run into other tourists; or you may run into tourists who do not have the correct information and send you in the wrong direction.

Now let's consider security:

- In a client/server architecture, each client needs to trust the server.
- In a peer-to-peer architecture, each peer needs to trust every peer from whom it receives data.
- Which one is better?
- If I want to get news on the war in Syria, is it better to trust cnn.com or bbc or aljazeera? Or is it better to discuss with random people whom I may or may not know personally and who are also interested in the topic?
- It is a matter of opinion, and it also depends very much on the particular context.

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport service
  - what will the app expect from the transport layer?

The second question in designing an application is the communication protocol:
what sequences of messages the client, server, or peer processes may exchange?

We will not discuss this question in general, we will answer it for specific applications, like the web or BitTorrent, later on.

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport-layer technology
  - what will the app expect from the transport layer?

The third question is the transport-layer technology:
when a process delivers a message to the transport layer (in other words, calls the “send” function that we talked about in the first lecture), what kind of service should it expect from the transport layer?

Here are a few options:

Reliable data delivery

- Deliver message to the destination process or signal failure
  - detect and recover from packet loss or corruption
  - loss-sensitive applications, e.g., web, file transfer, email, ...

The application may expect that, whenever it delivers a message to the transport layer, the transport layer will try very hard to deliver the message and, in any case, it will notify the application whether it succeeded or not.

This type of transport service makes sense for applications that are sensitive to loss.

For instance, when you send an email you expect that the email will be delivered;
in the rare occasions where it cannot be delivered, you receive a notification of the failure.

Guaranteed performance

- Minimum throughput
  - throughput-sensitive applications,
    e.g., video-conferencing
- Maximum end-to-end packet delay
  - delay-sensitive applications,
    e.g., emergency services, voice, gaming, …

An application could also expect guaranteed performance,
like minimum throughput or maximum end-to-end packet delay.

These make sense for applications that do not function well when the network is slow, e.g, …

Guaranteed security

- Confidentiality
  - message is revealed only to the destination
- Authenticity
  - message indeed came from claimed source
- Data integrity
  - message is not changed along the way

Finally, an application could expect security guarantees: …

Internet transport-layer protocols

- TCP: Transmission Control Protocol
  - reliable, in-order data delivery,
    flow control, congestion control
- UDP: User Datagram Protocol
  - detection of packet corruption
- No protocol offering guaranteed performance

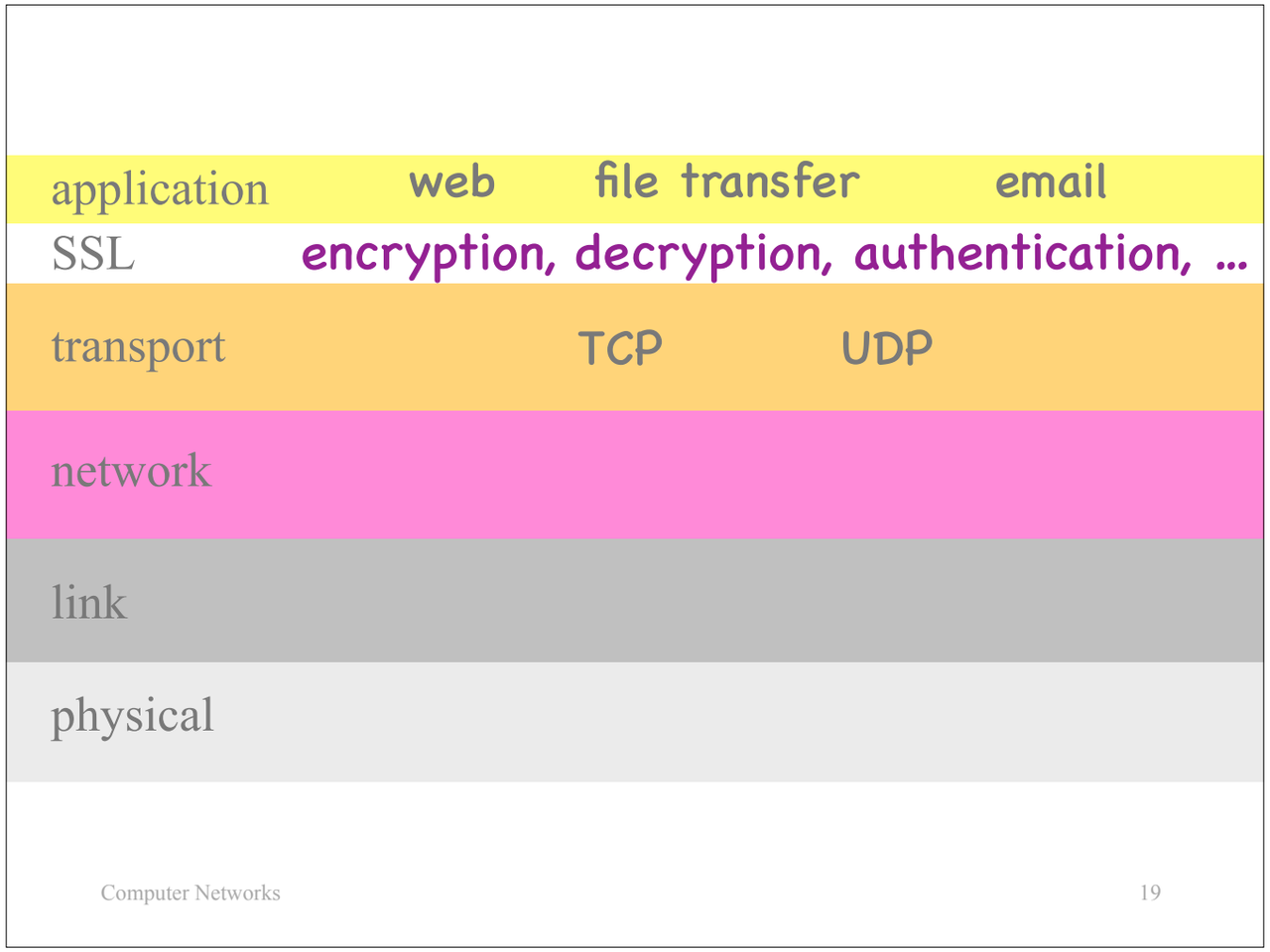
Which of these services does the Internet offer?

We said that there are two transport-layer technologies on the Internet: TCP and UDP.

- TCP offers reliable data delivery plus a few other services
- UDP offers a much simpler service (it only detects packet corruption).

No widely deployed protocol offers performance guarantees.

What about security?

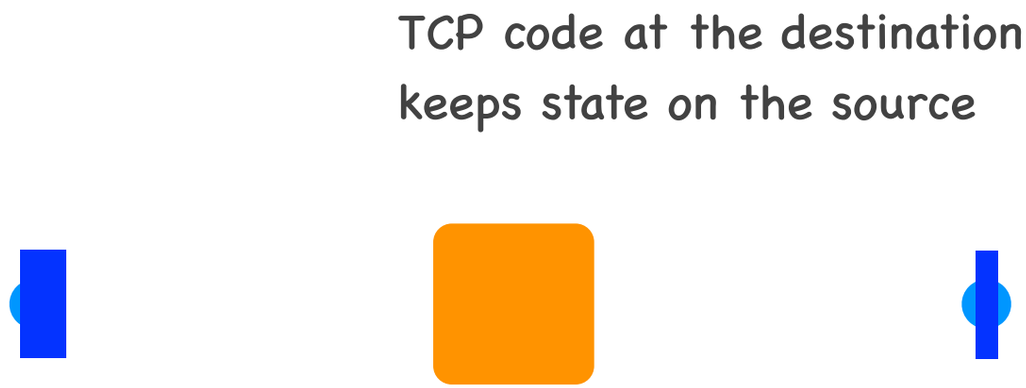


Here is the picture we saw in the first lecture, with all the layers of the Internet.
There is no official transport-layer technology that provides security guarantees.
However, because many applications do need that kind of service, they ended up implementing it themselves.
For example, an application can itself encrypt a message before passing it down, as opposed to expecting the transport layer to do it.
And because many applications used similar security-related functions, like encryption, a sub-layer emerged within the application layer, which is called the Secure Sockets Layer (SSL).
What does this really mean -- that we have a sub-layer within the application layer?
That we have a set of security-related functions in the form of a software library, and any application that wants secure transport can use these functions.

Let’s make a parenthesis here to gain a bit more insight into the transport layer.
We are not studying the transport layer yet, but the application layer uses the transport layer, so, it’s good to start gaining a bit of insight into how it works.

Let’s say that we have an application that uses TCP.
What does it mean that TCP provides “reliable data delivery”?

Suppose a process (of this application) sends a packet to another process.
If the packet makes it to the destination end-system, great.



But what if the packet gets dropped along the way, e.g., because some queue inside a packet switch is full?

We say that TCP provides “reliable data delivery” because the TCP code running on the source end-system will keep retransmitting the packet until it has done its best to get it to the destination.

The process that uses TCP does not have to deal with any of this.
It simply uses the proper API to pass down to TCP the data that it wants to send.
It is TCP’s job to retransmit the data enough times to get it to the destination or signal to the application that it has failed.

How can the TCP code running on the source end-system know whether the packet made it to the destination?
The TCP code running on the destination end-system sends feedback in the form of small packets, acknowledging to the source what data it has received.

So, the TCP code running at the destination keeps a record with information about the source, while the TCP code running at the source keeps a record with information about the destination.

This is why we say that TCP is a “connection-oriented” or “stateful” protocol, because the TCP code keeps track of things at the other end.

Connection = memory

- TCP is "connection-oriented" or "stateful"
  = maintains state on all the local/remote process pairs that use TCP
- UDP is "connection-less" or "stateless"
  = does not maintain state on remote processes

You will often hear people talk about “connections” in networking.
A “connection” can mean different things in different contexts.
But usually “connection” is related to memory.

I have a connection with you means that every time we meet, you remember the things I told you in the past lecture, and I remember the questions you asked me in the past lecture, and that memory, that knowledge, plays a role in our communication.

A source has a TCP connection with a destination means that the two of them keep a record about each other, exchange information/feedback, and adjust their behaviour accordingly.

This is why we say that TCP is a “connection-oriented” or “stateful” protocol, which means …
In contrast, UDP is a “connection-less” or “stateless” protocol, which means …

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport-layer technology
  - how will the app interact with the transport layer?

This completes our general discussion on application design, and we are ready to move on to concrete examples.

Example 1: the web

The first example application that we will look at is the web.

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport service
  - what will the app expect from the transport layer?

First question: what architecture does the web have? client/server or P2P?
Clearly…

a process that is always running

reachable at a fixed,
known process name

answers web requests

web client = web browser

web server

a process that generates
web requests

… client server.

A web client, also called a web browser, is a process that generates web requests.
A web server is a process that is

- always running on some end-system,
- is reachable at a fixed, known process name,
- and generates web responses.

Web requests and responses concern

Web page

- Base file + referenced files
  - base file specifies structure and potentially content
  - referenced files can be images, video, scripts, ...
- Each file has its own URL
  - URL = address for Internet resources
  - e.g., http://www.epfl.ch/index.html

web pages.

A web page consists of a base file + all the files that the base file references.
The base file specifies the skeleton, the main structure of the web page and sometimes also the content.
The referenced files can be images, videos, scripts, in general, any type of content that may be included in a web page.

Each file, whether it's the base file or a reference file, has its own URL, which stands for Uniform Resource Locator.
A URL is a name, an address for an Internet resource.
Anything that you type into the top window of your web browser, e.g., www.epfl.ch, is a URL.

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport-layer technology
  - what will the app expect from the transport layer?

Second question: what communication protocol does the web use?
It uses HTTP, which stands for Hypertext Transfer Protocol.

HTTP consists of twi general kinds of messages: requests and responses.

HTTP request types

- GET: client requests to download a file
- POST: client provides information
- HEAD: client requests file metadata
- PUT: client requests to upload a file
- …

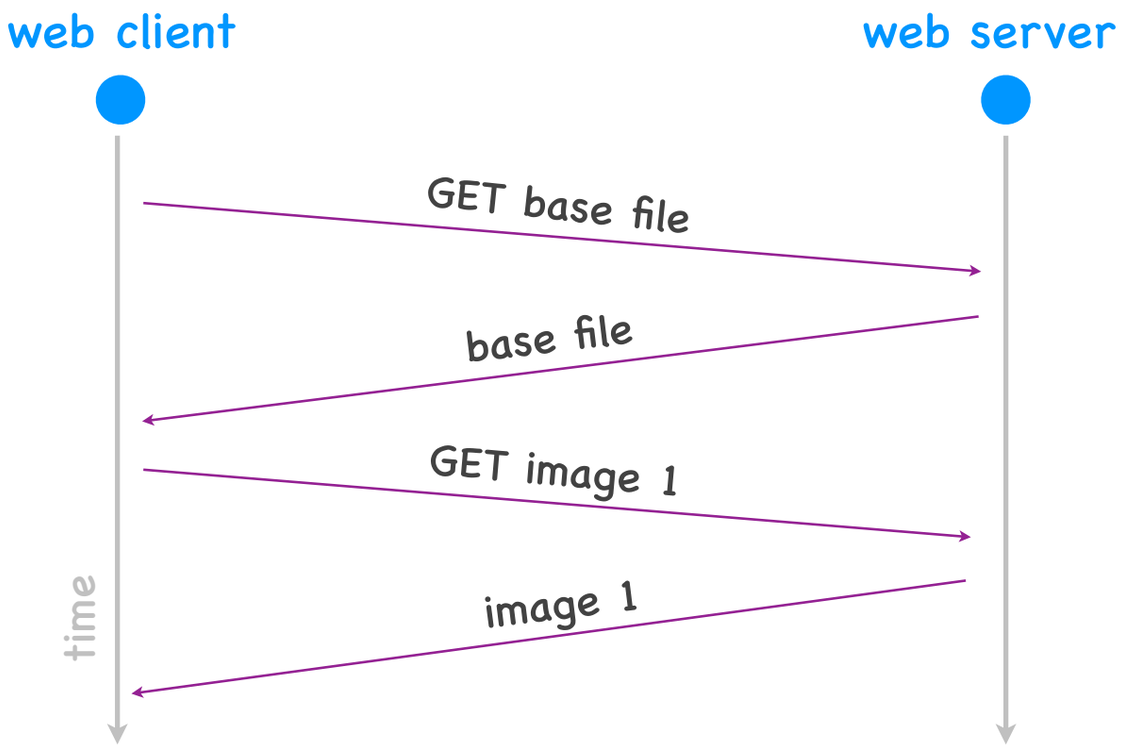
Here are some examples of HTTP request types.
The ones you use most often are:

- GET, which is what your web browser sends to a web server every time you access a web page, and
- POST, which is what your web browser sends to a web server every time you use a web form to provide information.

HTTP response types

- OK
- Not found
- Moved permanently
- Bad request
- …

And here are some examples of response types that a web server may send in response to a GET request.
The one that you encounter most often is OK, meaning that the web server is sending the content that the web browser requested.

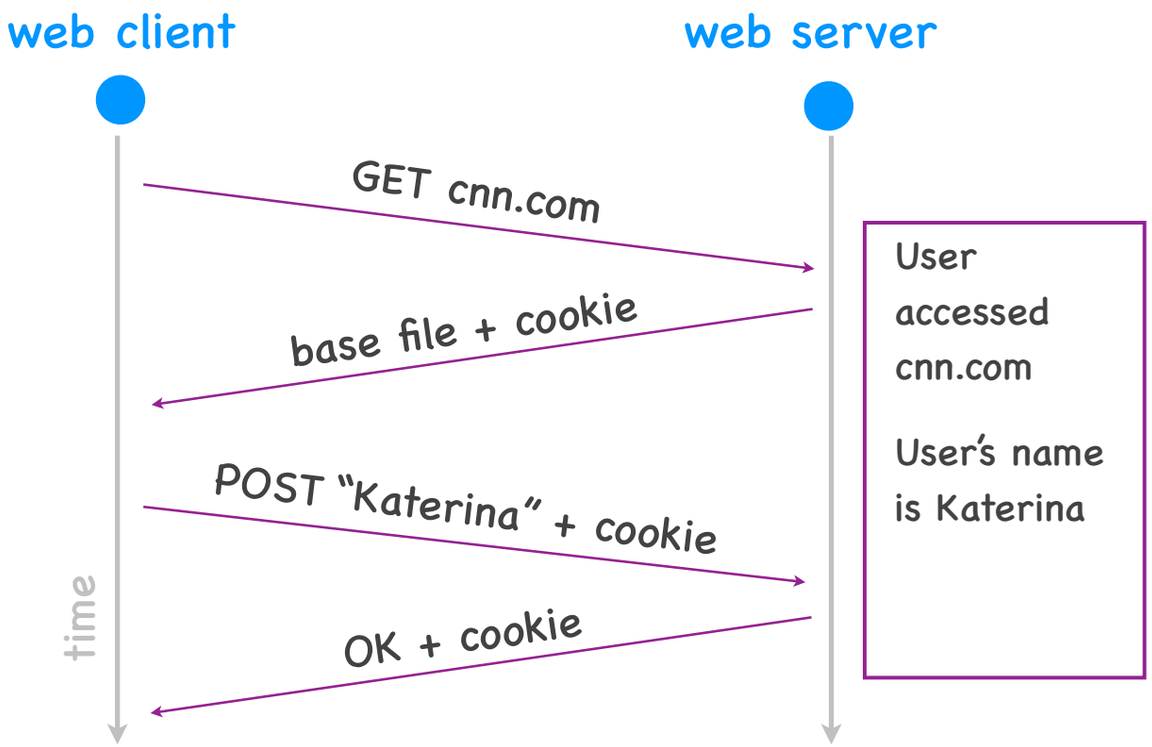


The most common message exchange is this:

- The web client requests the base file of a web page.
- The web server sends it.
- The web client reads the base file, identifies all the objects that it needs in order to present this web page to the human user, and requests each of these objects one by one.

So, when you look at a web page, under the covers, your browser has made multiple requests to the corresponding web server in order to collect all the objects -- pictures, videos, scripts, etc.

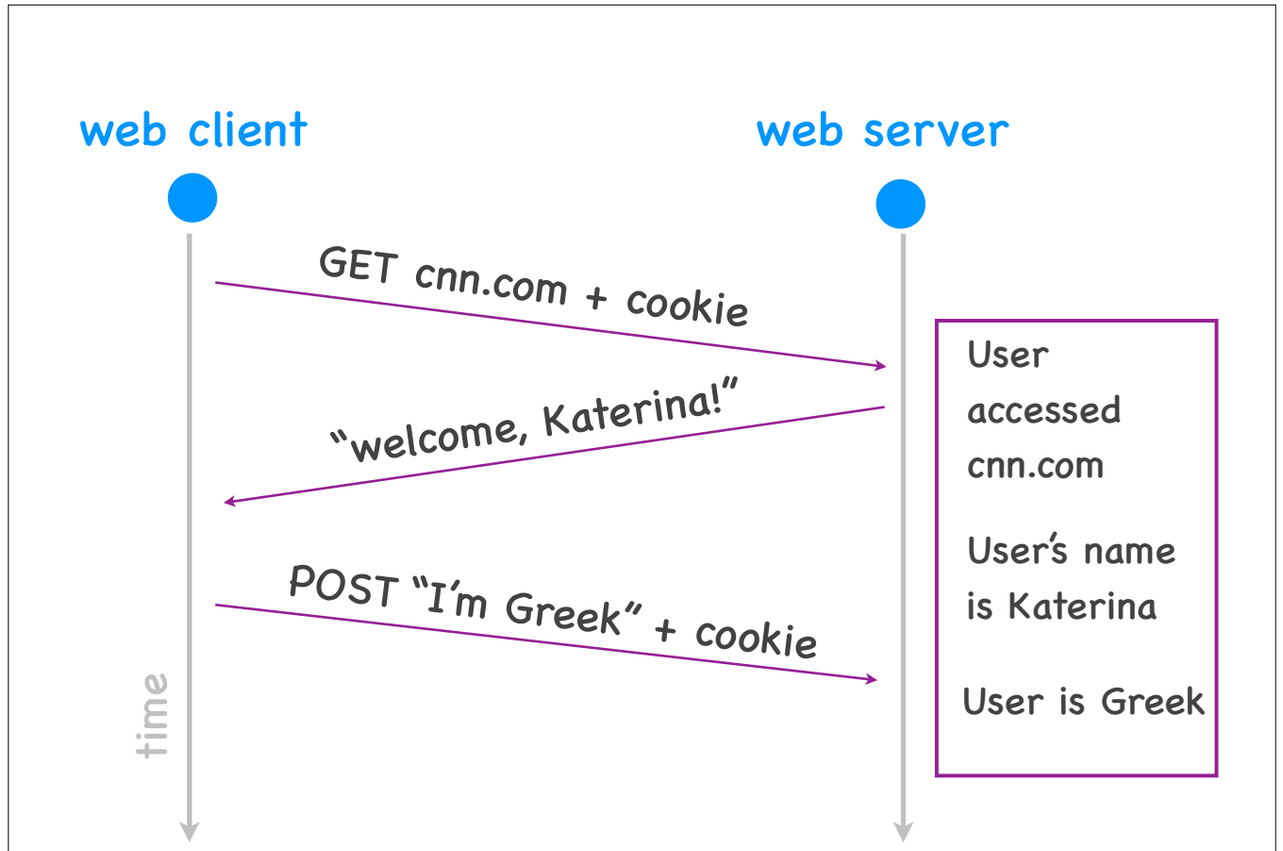
—> Is HTTP a connection-oriented protocol or not?
Does the server keep state on the client?
It used not to. HTTP was designed to be a connectionless protocol.
Then how does Facebook or Amazon know who you are whenever you return to the Facebook or Amazon home page? You must have noticed that they know who you are even before you login. How do they do they do that if their web servers are not keeping any state on the web clients that communicate with them?



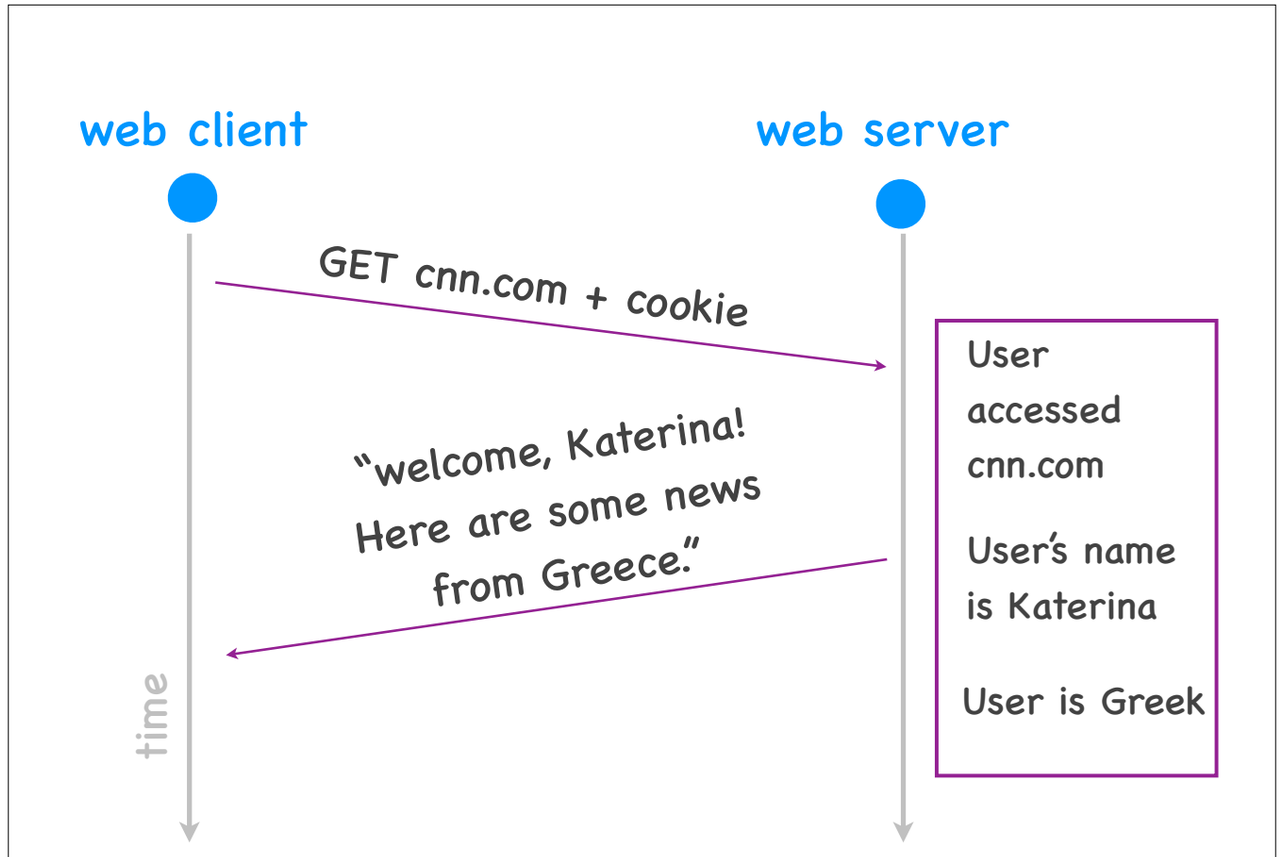
The answer is cookies.

Suppose I open my web browser, and I type in cnn.com.

In response, my web browser creates an HTTP GET request and sends it to a web server that serves the cnn.com website.
At that point, the web server can create a cookie for me, which is state about me, e.g., what webpage I requested.
The web server responds to my web browser's GET request with an appropriate HTTP response, which includes the cookie.
As part of that response, the web server can include a web form that says "Hey, we are making a survey on first names, so would you mind telling us your first name?"
If I agree, I provide my first name, which means that my web browser now creates an HTTP POST request and sends it to the web server.
Crucially, my web browser includes in this request the cookie that the web server previously created about me.
So, now, the web server can add my name to the cookie and, again, include the cookie with the HTTP response it sends back to my web browser.



The next day, when I visit the cnn web page again, my web browser again includes the cookie with every HTTP request that it sends to the web server.
So, the web server can read the cookie and create a personalised HTTP response, e.g., give me a webpage that includes my name.
Again, as part of its response, the web server can include a web form that says "we are doing a survey on nationalities, would you mind telling is where you are from?"
If I agree, I provide my nationality, which means that my web browser now creates an HTTP POST request and sends it to the web server, together with the cookie.
So, now, the web server can add my nationality to the cookie (and always send the cookie back to my web browser as part of its HTTP response).



The next day, when I visit the cnn web page again, my web browser again includes the cookie with every HTTP request that it sends to the web server. So, the web server can read the cookie and create another personalised HTTP response for me, e.g., focus on news from Greece.

This is the idea behind cookies: to enable web servers to collect information about their users.

Please be careful, I am not saying that websites follow exactly this scenario, with web forms and explicit questions. This is only one possible scenario.

Cookies

- Cookie = state created by the web server, stored at the web client
- It links subsequent HTTP requests to the same web client
- HTTP remains a stateless protocol

(Pushing state to the client)

- Classic systems trick to keep servers simple
- The server can do stateful processing of client requests
- Without maintaining per-client state

Anything wrong with cookies?

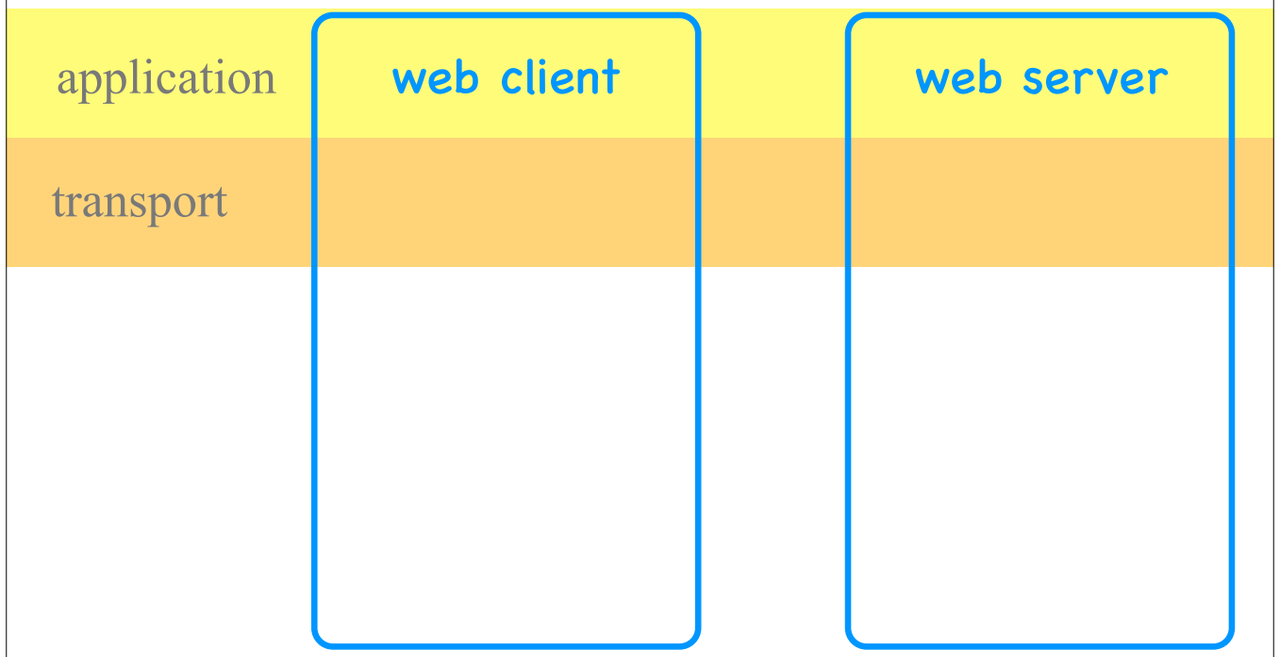
The question is: is anything wrong with cookies?

Design an application =

- Design the architecture
  - which process does what?
- Design the communication protocol
  - what sequences of messages can be exchanged?
- Choose the transport-layer technology
  - what will the app expect from the transport layer?

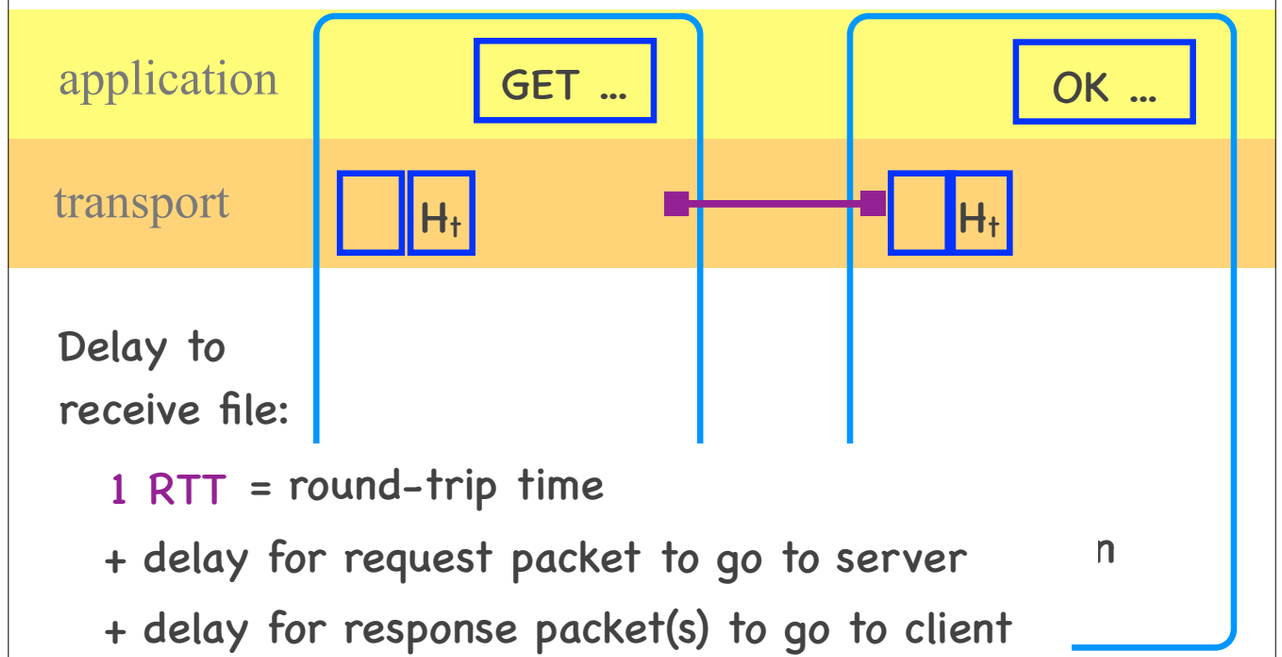
Third question: what transport-layer technology does the web use? TCP or UDP?
You know the answer from the first lab: it's TCP.

Let's dive in a little bit and see how HTTP (which is the application-layer protocol used by the web) interacts with TCP (which is the transport-layer protocol used by the web).



The two rectangles I am showing you are two computers,
one running a web client, the other one running a web server at the application layer.

Suppose the user behind this web client types in www.epfl.ch,
which happens to be served by this web server.



Under the covers, the web client process creates an HTTP GET request for the base file of the www.epfl.ch web page.

Then, it passes the request down to the transport layer and, in particular, the TCP code running at the transport layer.

We said that TCP is connection-oriented.
So, before it passes the request down to the network layer, it seeks to establish a TCP connection with the TCP code running on the server side.
So, it sends to it a small packet that carries a TCP connection setup request.
If the TCP code running on the server side agrees, it sends back a small packet that carries a response to the TCP connection setup request.

Once a TCP connection between the two computers has been established, the transport layer on the client side passes the HTTP GET request down to the network layer, which passes it down to the layers below, such that, eventually, the request reaches the web server process.

When the web server process receives the GET request, if it agrees, it sends an OK response that includes the requested base file.

Let's look at the time it takes from the moment the web client process passes the GET request down to the transport layer until it receives the response:

- First, it has to wait for one round-trip-time or RTT, which is the amount of time it takes for a small packet to go from one computer to the other and another small packet to return.
- Second, it has to wait for the actual GET request to go to the web server.
- And third, it has to wait for the OK response to come back.

The point to keep is that before the web client process can communicate with the web server process, they have to wait for the TCP connection to be established.
So, there is a cost to using TCP, it does not come for free.

Typical ways to use TCP

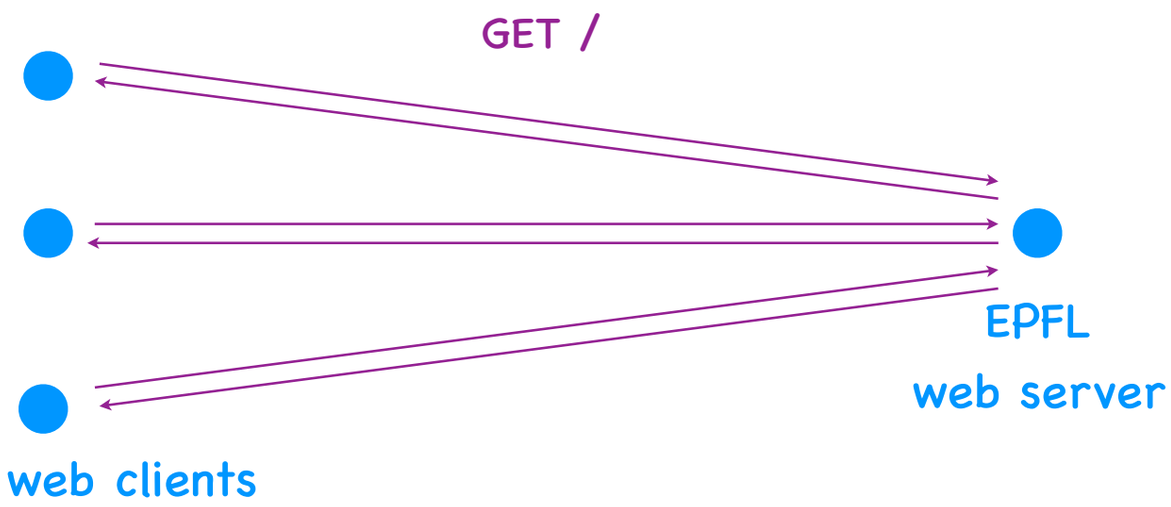
- Persistent TCP connections
  - reuse the same TCP connection
    for multiple HTTP requests and responses
- Parallel TCP connections
  - exchange multiple HTTP requests
    and responses in parallel

There are many different ways to use TCP, but most web clients and web servers today use *persistent* TCP connections, meaning that they use the same TCP connection to exchange multiple HTTP requests and responses.

To connect this to the example on the last slide,
once there is a TCP connection established between the two computers,
it is used to send not only the GET request for the base file,
but also the GET requests for all the referenced files.
That makes sense, so that the web client does not have wait for a new connection to be established in order to send every new GET request.

Moreover, most web clients today establish multiple parallel TCP connections to each web server, and multiplex their GET requests across these connections.
We will discuss why when we dive into TCP a couple lectures from now,
but I would like you to fork a thread in the background and start thinking about why it may make sense to use parallel TCP connections between the same two computers.

Silicon Valley

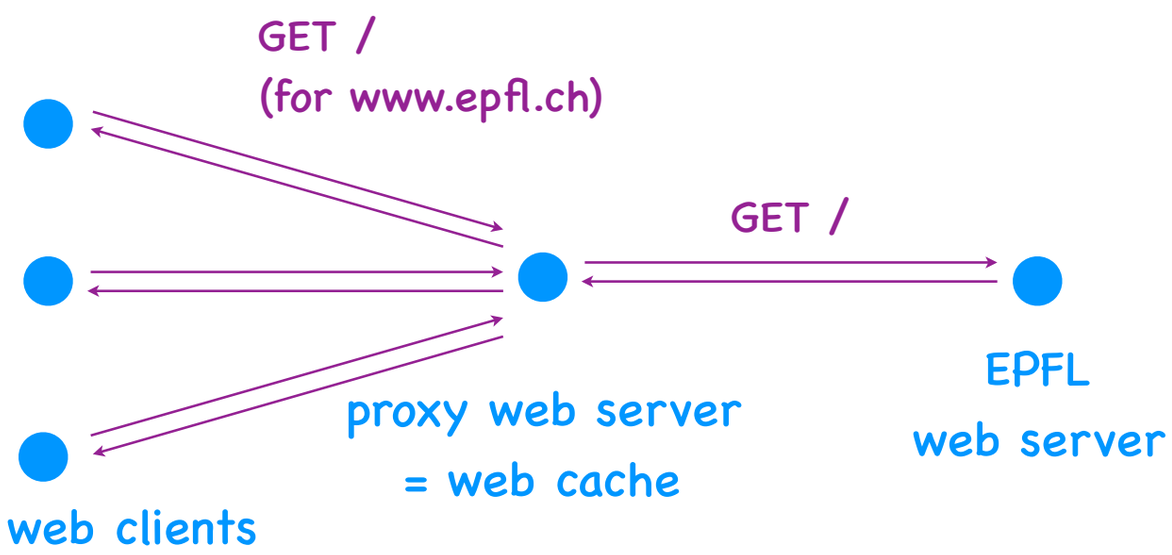


Consider the following scenario:
A group of you have gone to California to do an internship at Silicon Valley,
but you miss home, so every day you visit www.epfl.ch to see what's new.

All of you send the same sequence of HTTP requests:
a GET request for the base file of the www.epfl.ch web page,
and then GET requests for all the files referenced by the base file
(which I am not showing here).

Is there anything that can be done to reduce the delay experienced by each of you from the moment you type www.epfl.ch into your browser until you see the EPFL web page?

Silicon Valley



Caching.

On top of web clients and web servers, we will now introduce a third type of web-related process, called a proxy web server or proxy cache.

When sending an HTTP GET request for the base file (or any referenced file) of www.epfl.ch,
instead of sending this request to the EPFL web server,
a web client can send the request to the proxy web server.

When the proxy web server receives the request, it checks “do I have the requested file?”
If not, it sends the request to the EPFL web server, receives the file,
and forwards it to the requesting web client.

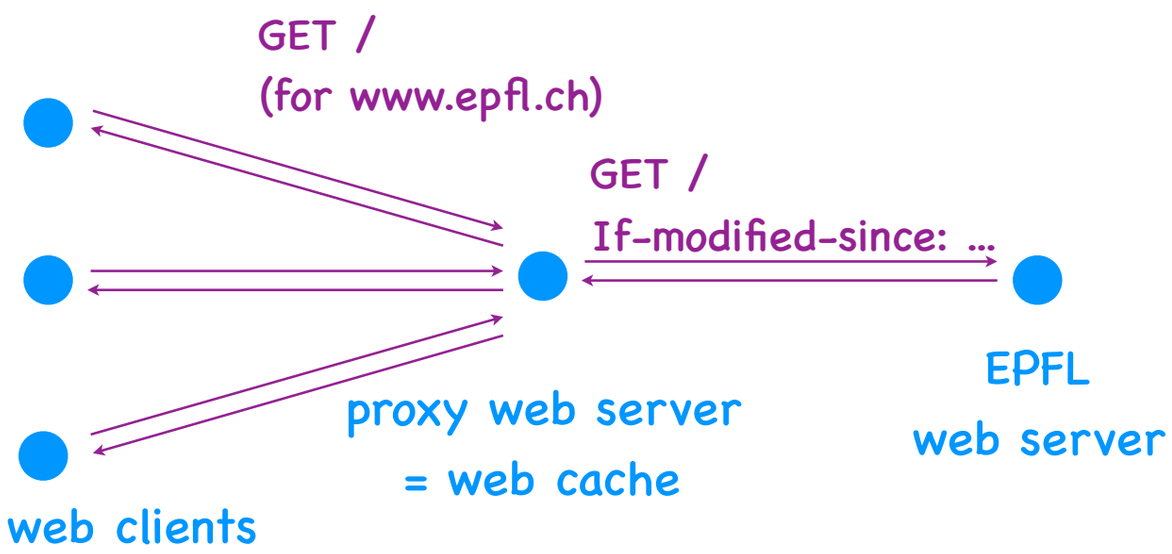
The second time the proxy web server gets a request for the same file,
it again checks “do I have this file”, and, this time, it does.
So, it sends it to the requesting web client immediately.

When the proxy web server receives a request for a file,
it does not only check whether it has this file cached.

It also checks *when* it obtained this file from the EPFL web server.

The proxy web server does not send a normal GET request for the file to the EPFL web server, it sends what we call a…

Silicon Valley



…*conditional* GET request, meaning, send me this file if it has not changed, it has not been modified since a given time.

If the file has not changed since the given time,
the EPFL web server does not send it to the proxy web server,
it simply tells the proxy “the file has not changed,”
in which case the proxy sends the cached file to the requesting web client.

Have we gained anything?
The proxy web server has to ask the EPFL web server anyway whether the file has been modified or not.
So, have we gained anything?
Will the second and third web clients download the EPFL web page any faster than the first web client?
What does the effectiveness of caching depend on?

I would like you to think about this and we will discuss it on Friday.
Keep in mind that you already know enough to give a very concrete answer.

Web caching

- Proxy web server or web cache
  - caches copies of other web-server files
  - acts as a web server to nearby web clients
- Reduces delay experienced by web clients
- Relies on conditional GET to ensure data freshness

Caching

- Universal technique for improving performance

- Challenge: stale data
  - option #1: dynamic check for staleness
  - may introduce significant delay

Caching is a universal technique in computer science.
Whenever we have a set of entities interested in the same data,
the moment one of these entities pays the cost to bring the data closer,
it is worth keeping the data closer -- so that the other entities do not have to pay the same cost.

The challenge is dealing with stale data, meaning data that has changed since it was cached.
One option is to use a dynamic staleness check, which is exactly what the conditional GET is.
This, however, may introduce significant delay.

I will close this lecture by asking you: how can caching deal with stale data *without* a dynamic check for staleness?