Transport Layer Security (TLS)

Diana Gratiela Berbecaru
diana.berbecaru@polito.it

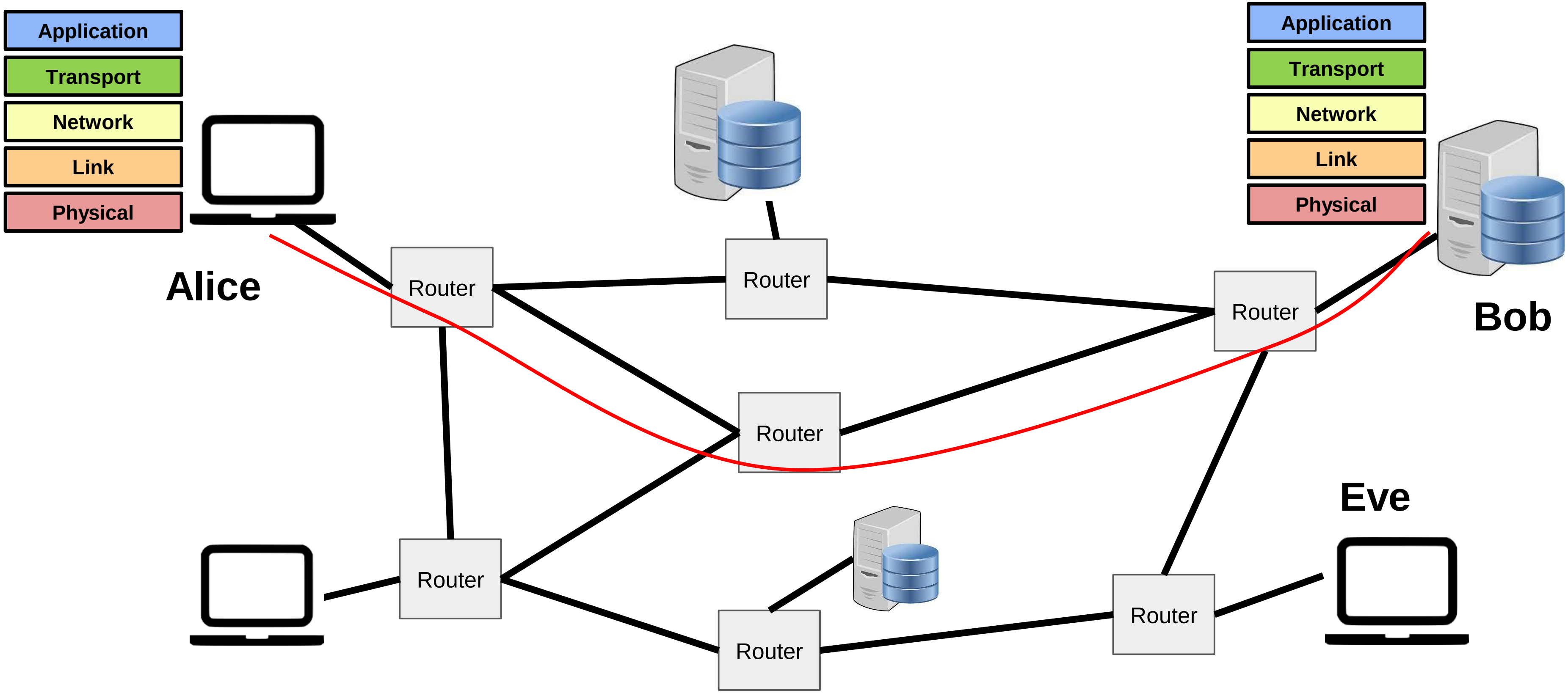
Politecnico di Torino
Dip. Automatica e Informatica

AY. 2023 - 2024

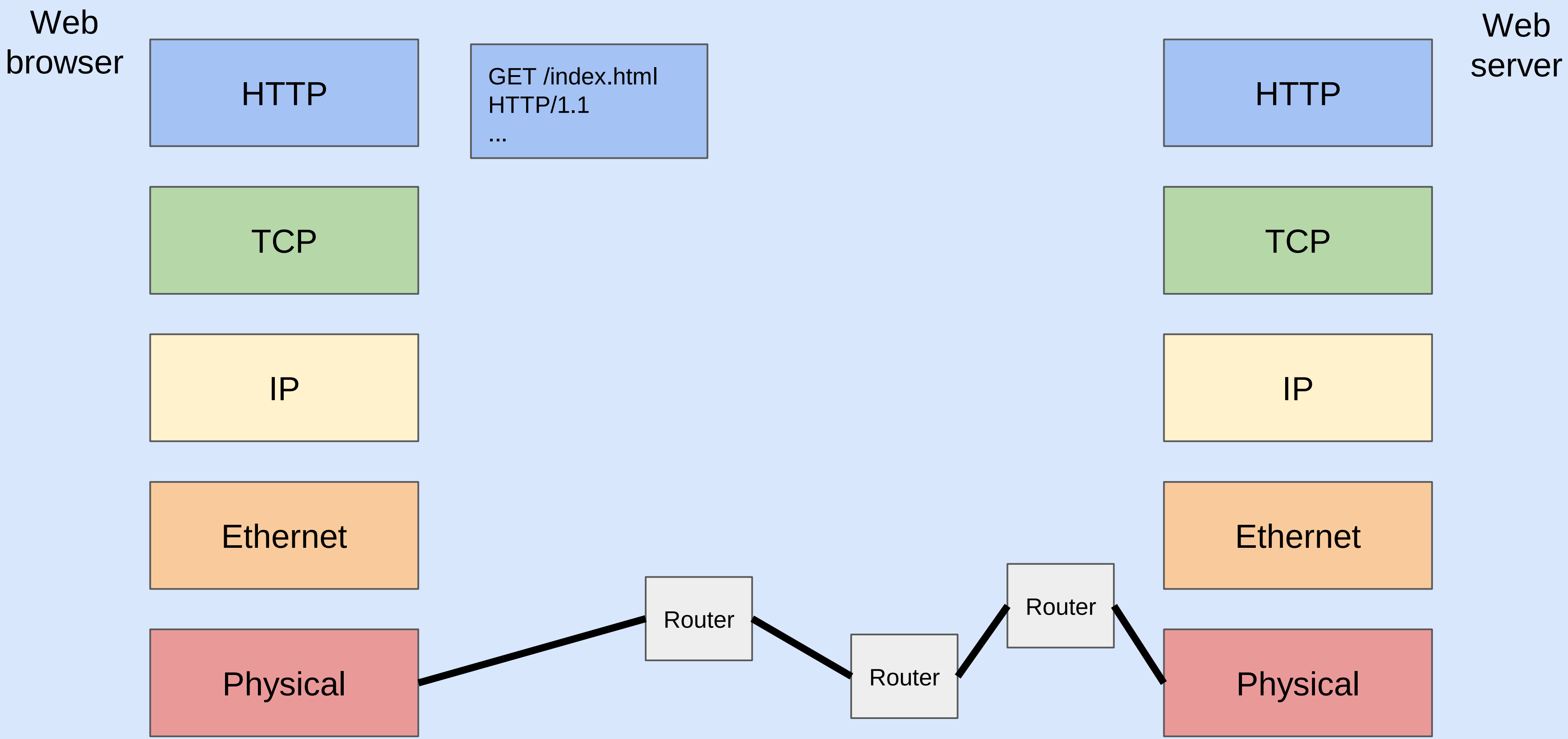
Acknowledgment

- **Some of the slides have been derived from the material created by Prof. Antonio Lioy for the course Information Systems Security (2008 - 2022)**

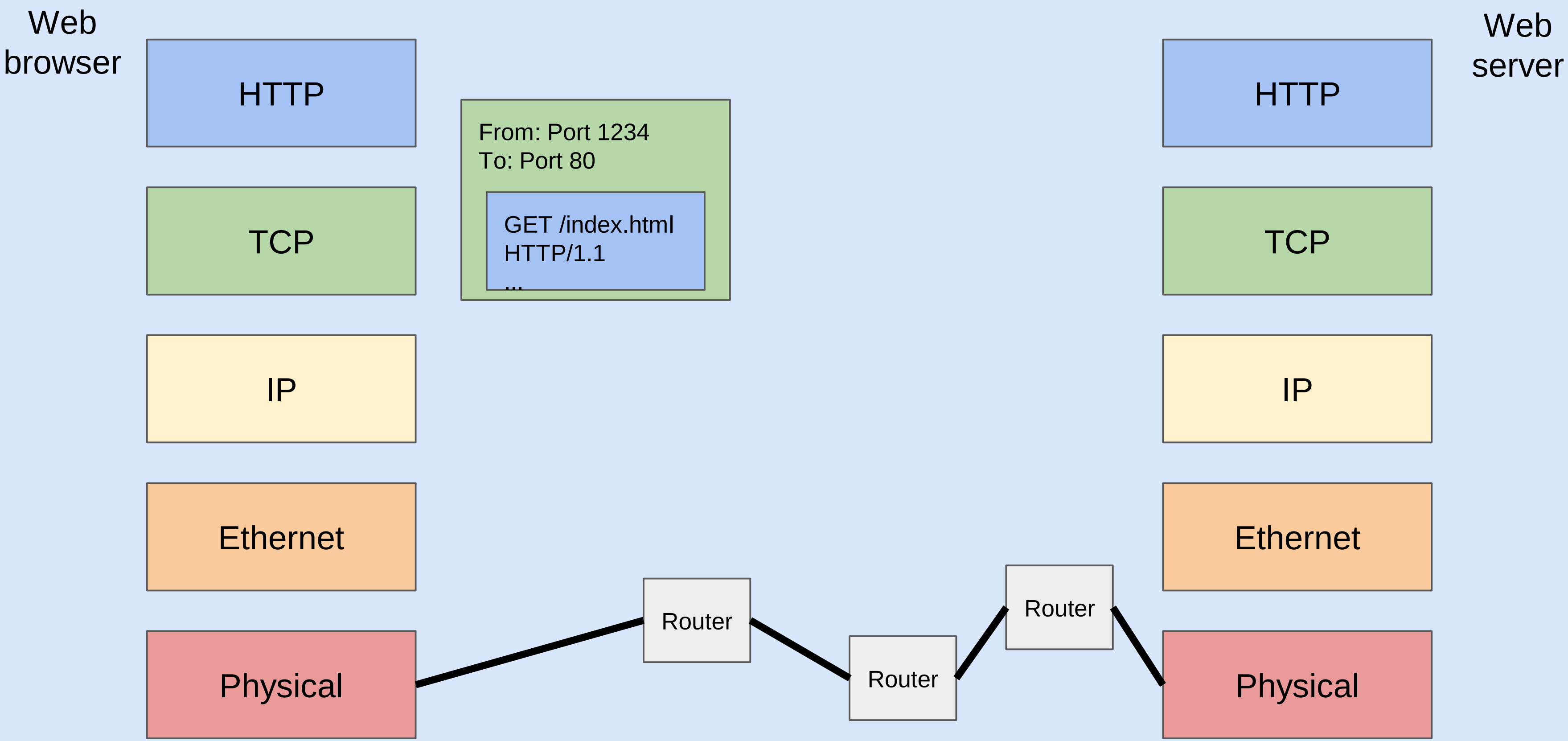
Why TLS has occurred?



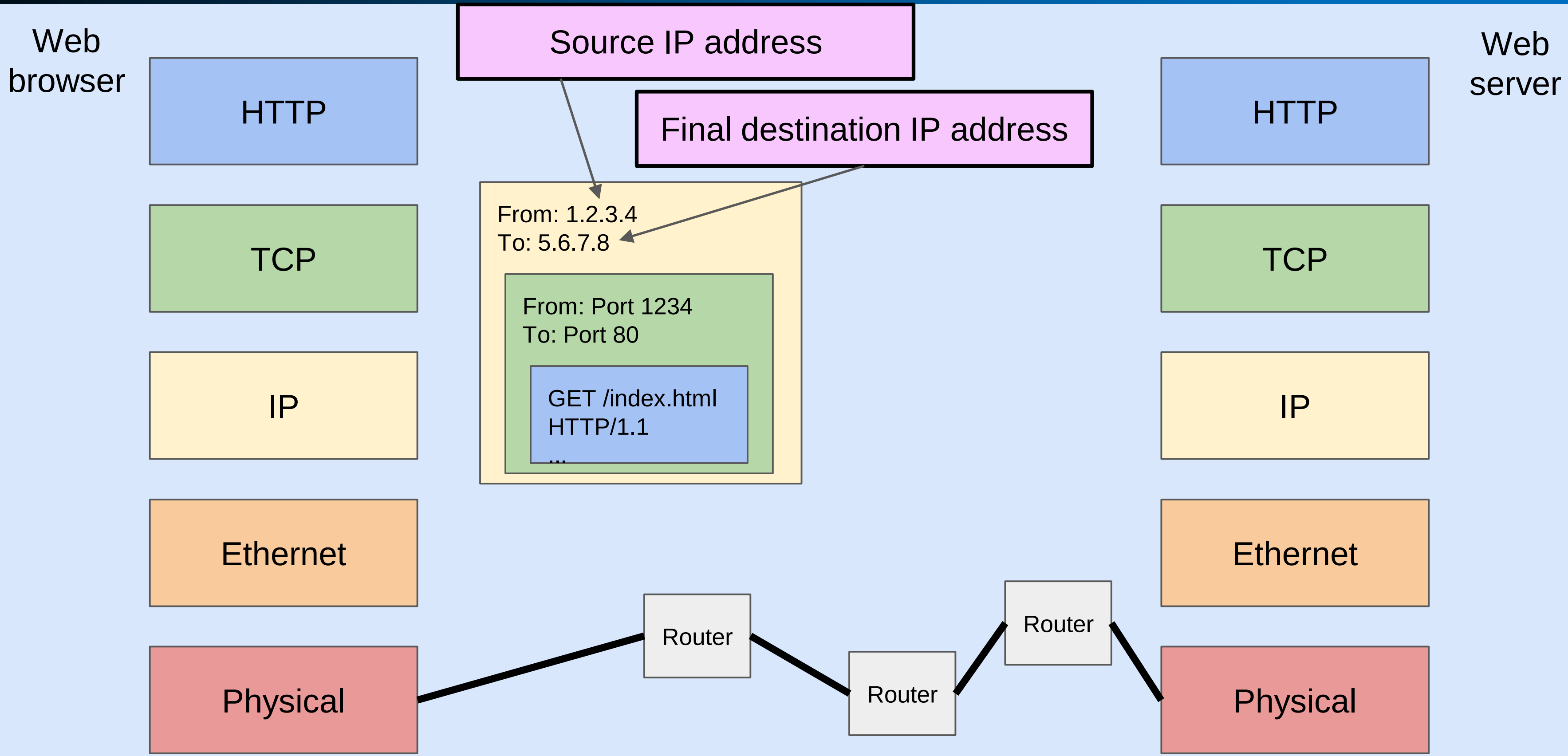
Data transfer over TCP/IP, Example: HTTP



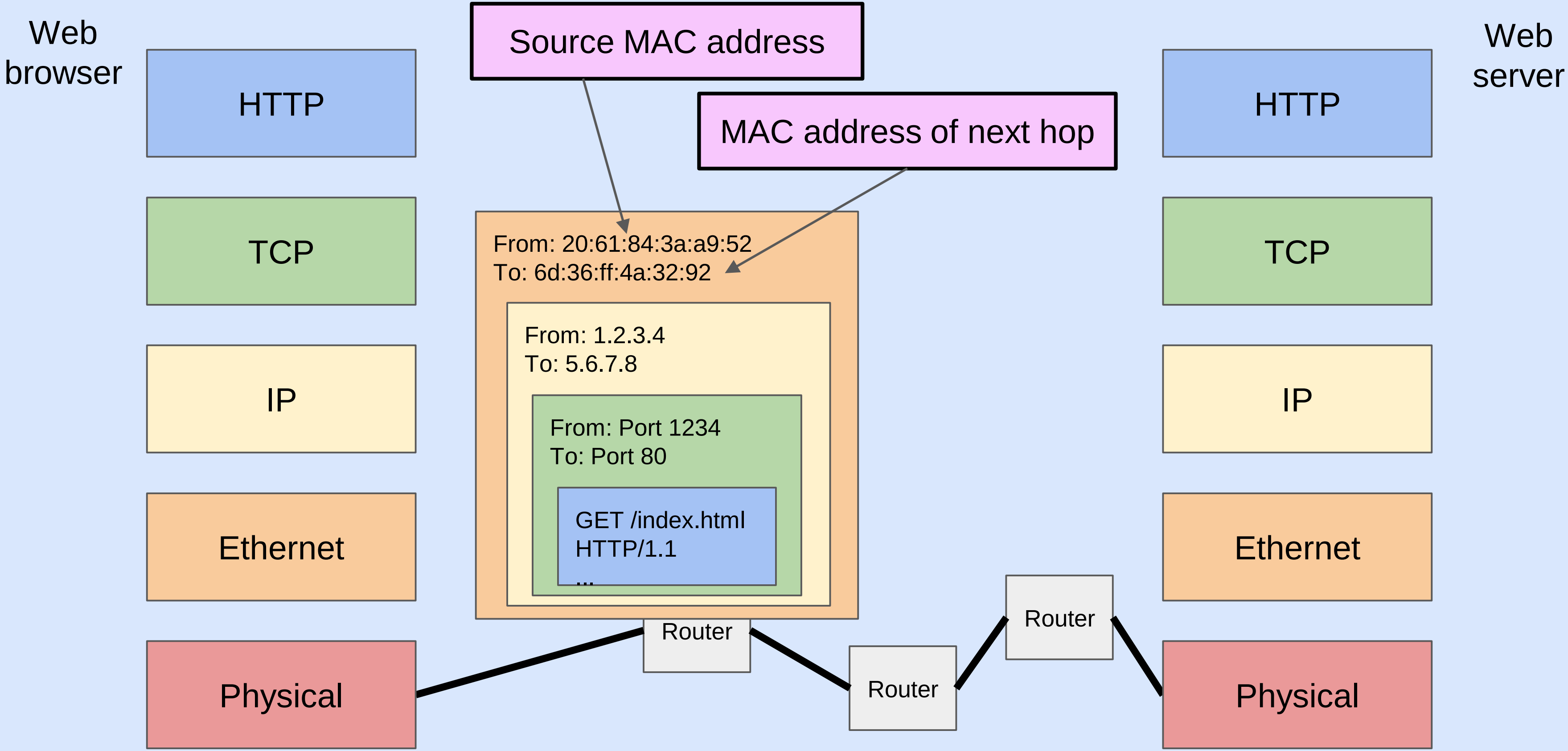
Example: HTTP data transfer



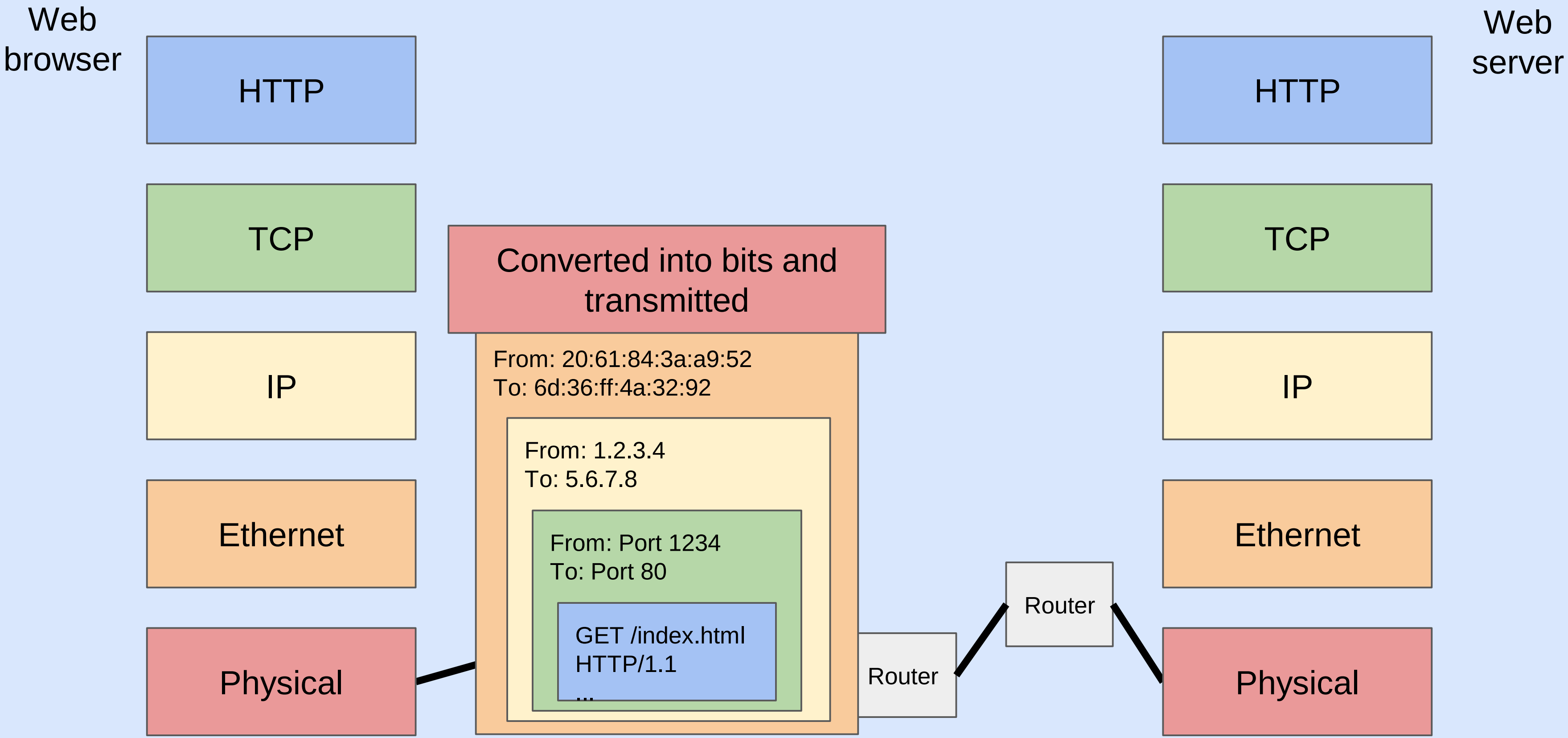
Example: HTTP data transfer



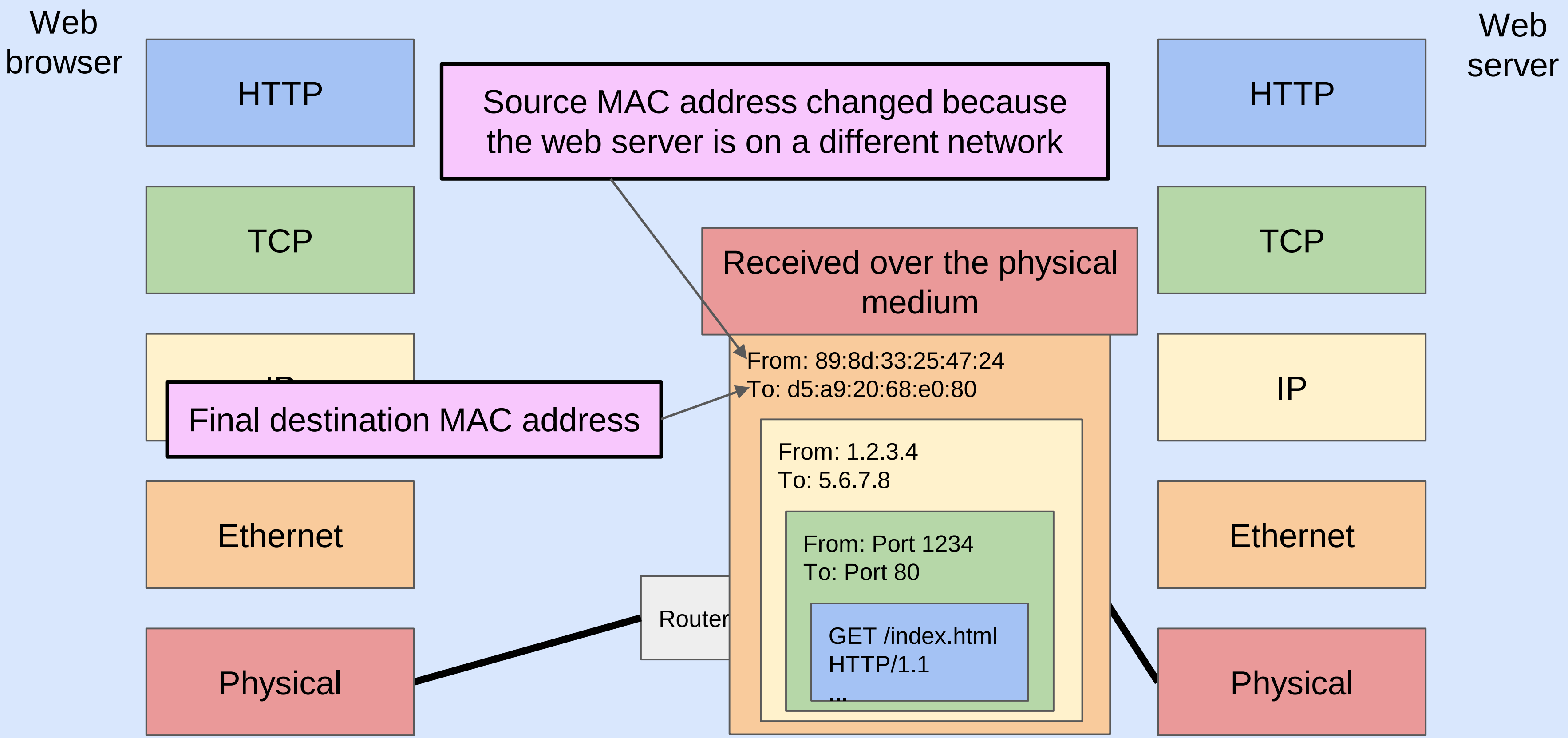
Example: HTTP data transfer



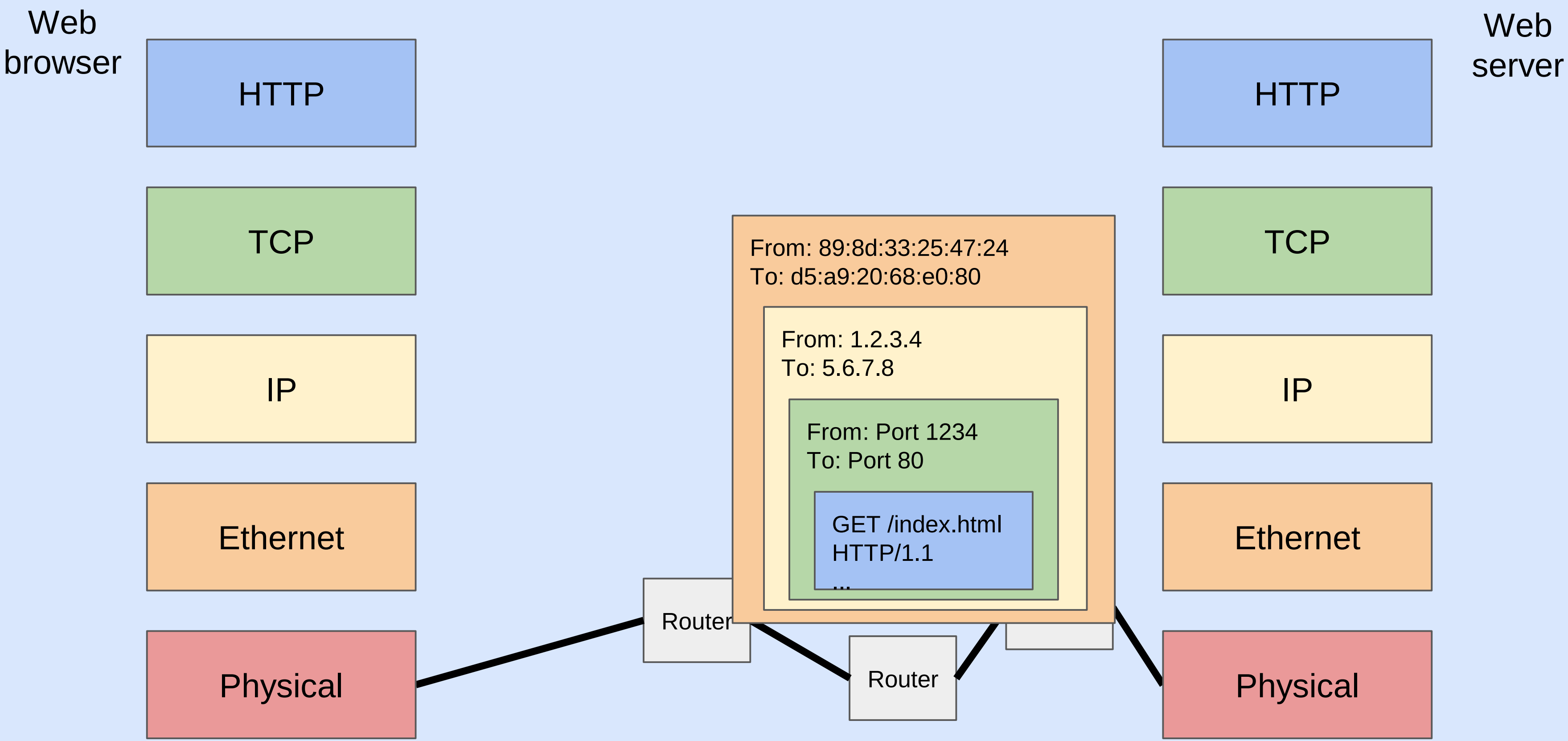
Example: HTTP data transfer



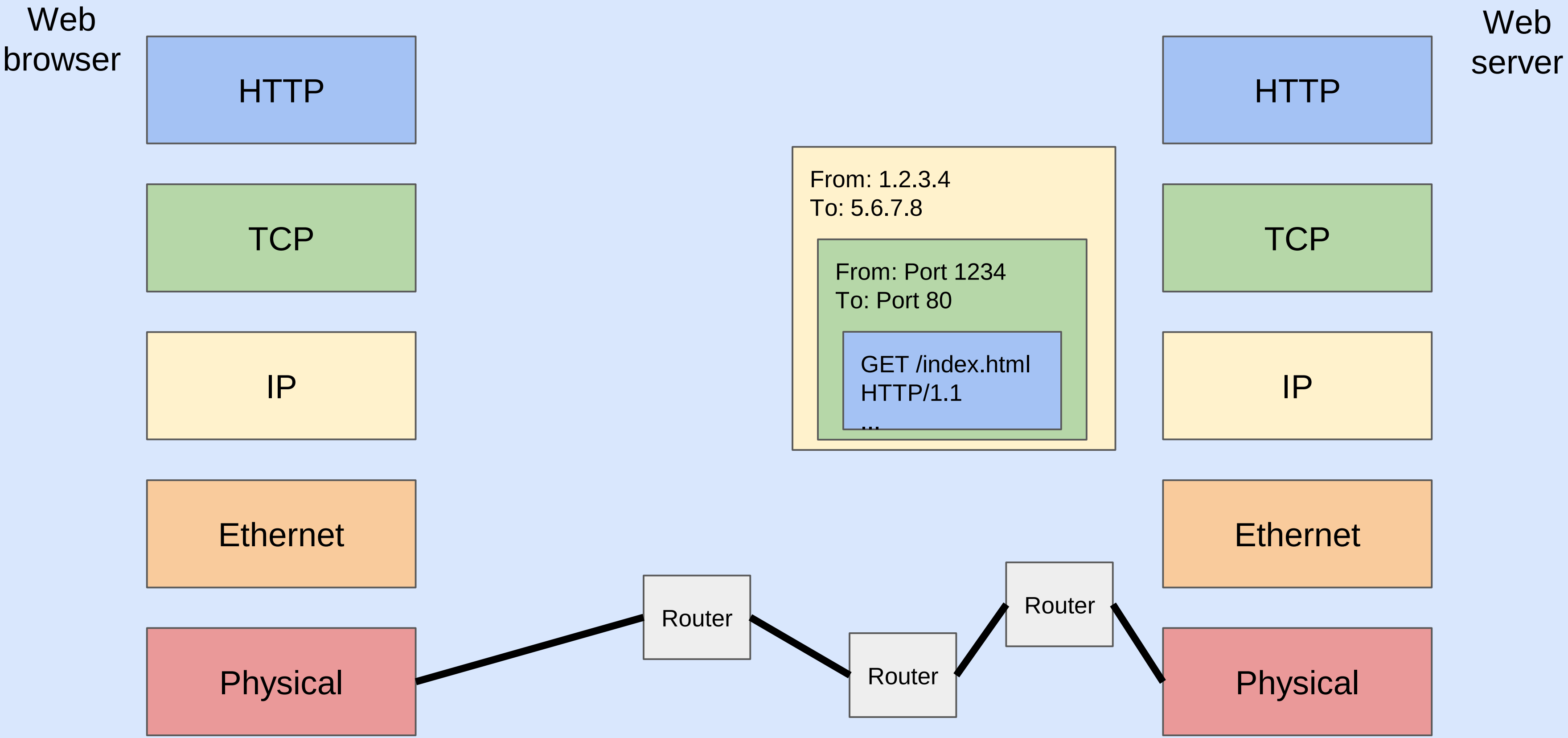
Example: HTTP data transfer



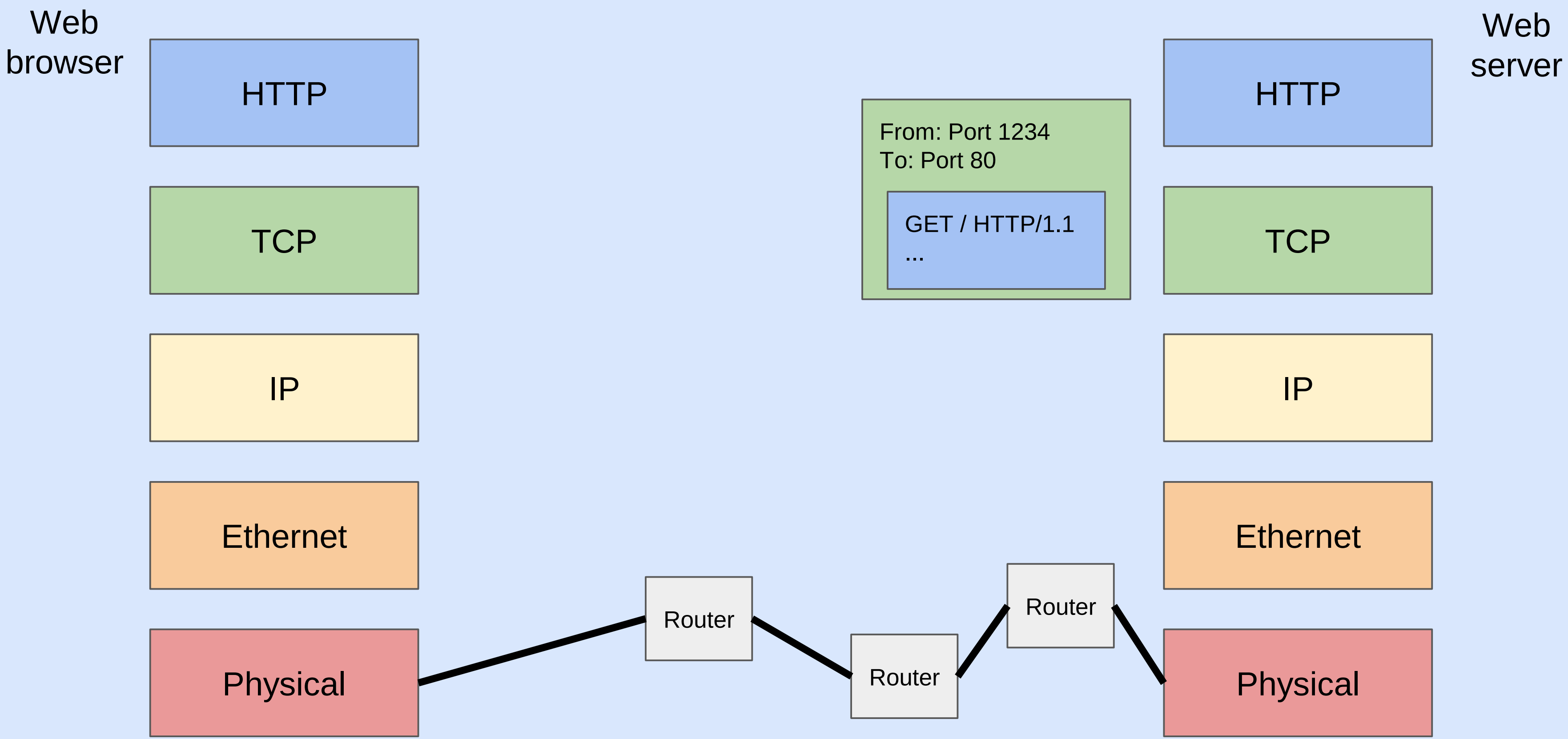
Example: HTTP data transfer



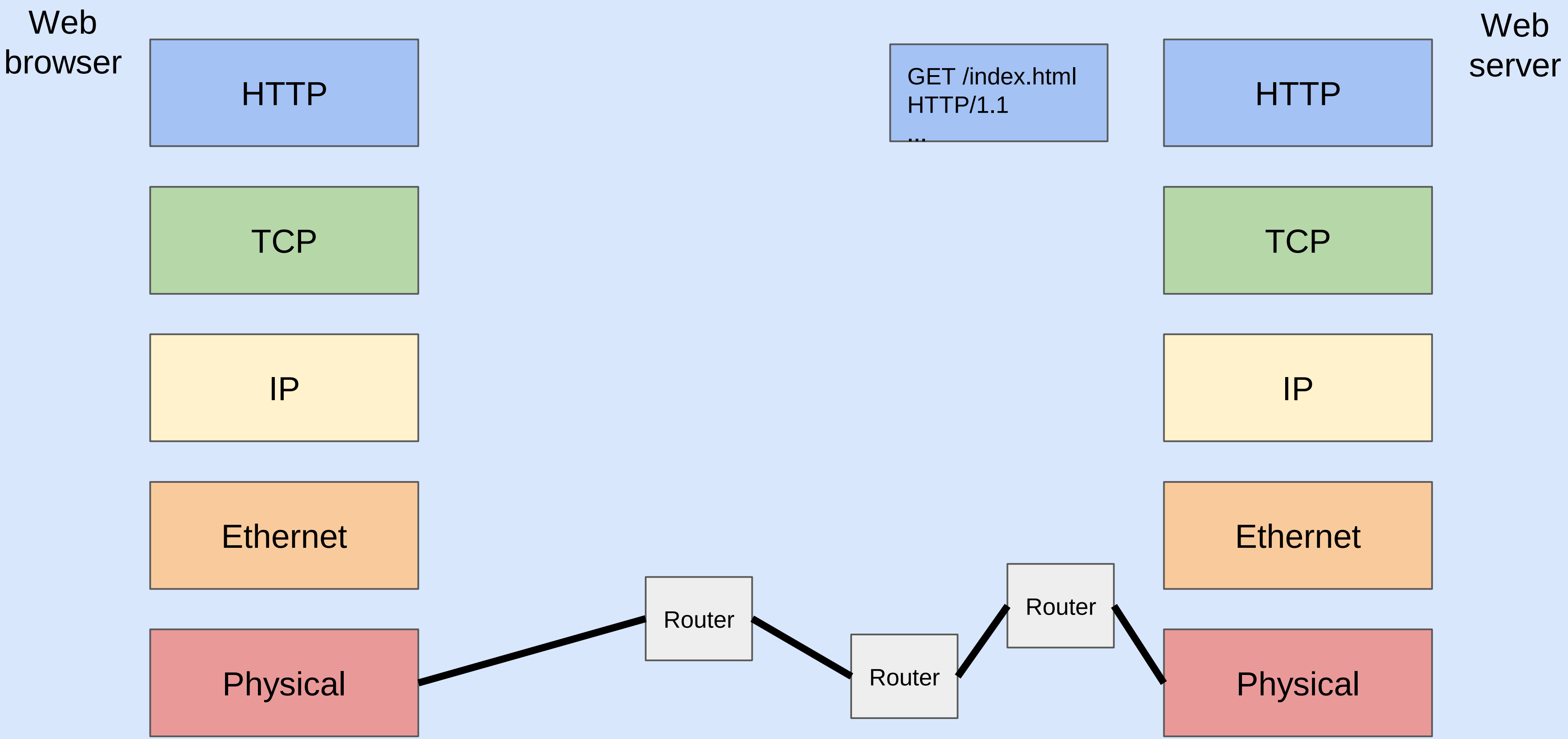
Example: HTTP data transfer



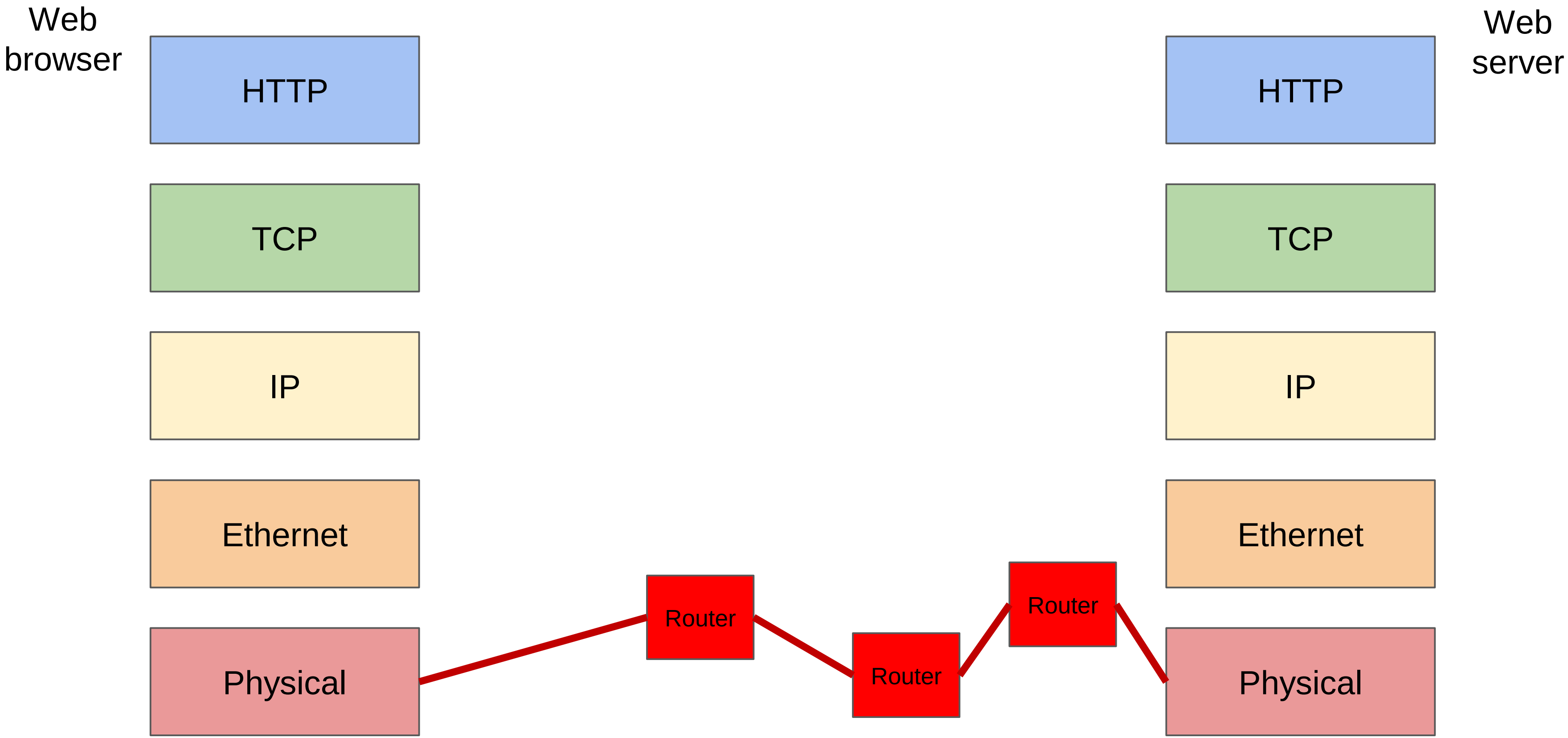
Example: HTTP data transfer



Example: HTTP data transfer

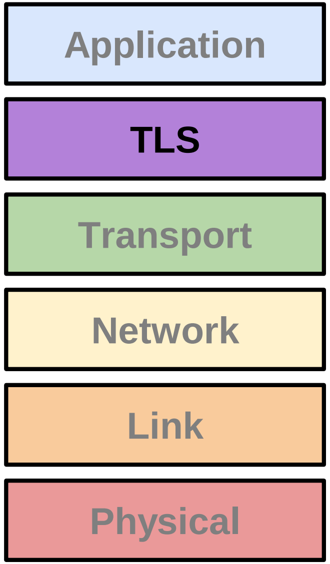


Where is the attacker?



What is TLS?

- **TLS (Transport Layer Security): protocol for creating a secure communication channel over the Internet**
  - replaces SSL (Secure Sockets Layer), which is the original (old) version of the protocol
- **TLS is built on top of TCP (transmission control protocol) running at transport level**
  - TCP provides a byte stream abstraction between the client and the server …… but TCP segments are not (cryptographically) protected
  - TLS provides byte stream abstraction between the client and the server … and provides security services to application data



SSL and TLS versions

- **SSL / TLS**
  - SSL: originally proposed by Netscape Communications (1994)
  - … then evolved and standardized by IETF
  - TLS - the most widely used security protocol nowadays !
- **SSL v2, SSL v3 (‘95-’96), then TLS**
  - … TLS v1.0 (‘99), TLS v1.1 (2006), TLS v1.2 (2008), TLS v1.3
- **SSL/TLS protocol versions below TLS v1.2 are insecure and deprecated**
- **nowadays, two versions in use:**
  - TLS 1.2 … but everyone currently migrating to TLS 1.3 (2018)

TLS security properties

- **provides a secure TCP transport channel from client to server:**
  - peer authentication of server, optionally also of client
    - allows client to verify that is connecting to the legitimate server
      - defend against an attacker impersonating the server
    - optionally, authenticate the client, when establishing the channel (!)
  - message (data) confidentiality
    - ensures that attackers cannot read the traffic exchanged
  - message (data) authentication and integrity
    - ensures attackers cannot tamper with/modify the traffic exchanged, or inject new data

TLS security properties (II)

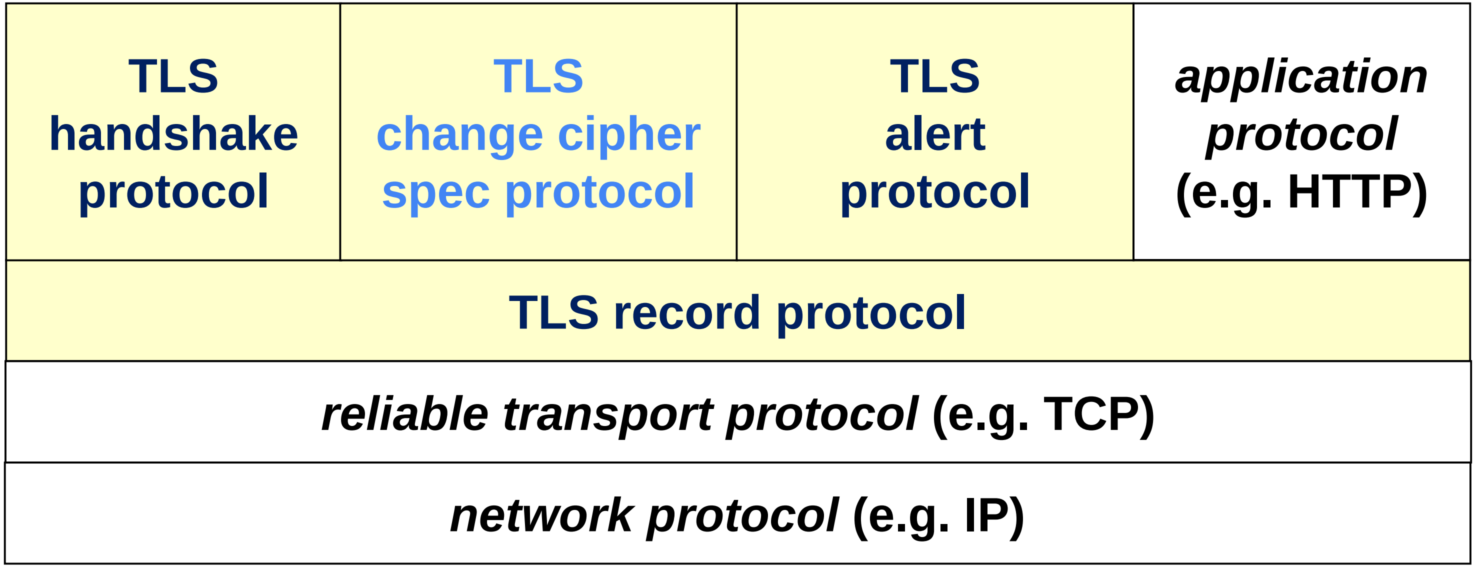
- **(cont.):**
  - protection against replay, reordering, and filtering/cancellation attacks
    - ensures attackers cannot replay, reorder, filter the traffic exchanged
- **easily applicable to all protocols based on TCP:**
  - HTTP, SMTP, NNTP, FTP, TELNET, ...
  - e.g. the famous secure HTTP (https://....) = 443/TCP

Official ports for TLS applications

nsiiops	261/tcp	# IIOP Name Service over TLS/SSL
https	443/tcp	# http protocol over TLS/SSL
smtps	465/tcp	# smtp protocol over TLS/SSL (was ssmtp)
nntps	563/tcp	# nntp protocol over TLS/SSL (was snntp)
imap4-ssl	585/tcp	# IMAP4+SSL (use 993 instead)
sshell	614/tcp	# SSLshell
ldaps	636/tcp	# ldap protocol over TLS/SSL (was sldap)
ftps-data	989/tcp	# ftp protocol, data, over TLS/SSL
ftps	990/tcp	# ftp protocol, control, over TLS/SSL
telnets	992/tcp	# telnet protocol over TLS/SSL
imaps	993/tcp	# imap4 protocol over TLS/SSL
ircs	994/tcp	# irc protocol over TLS/SSL
pop3s	995/tcp	# pop3 protocol over TLS/SSL (was spop3)
msft-gc-ssl	3269/tcp	# MS Global Catalog with LDAP/SSL

TLS architecture

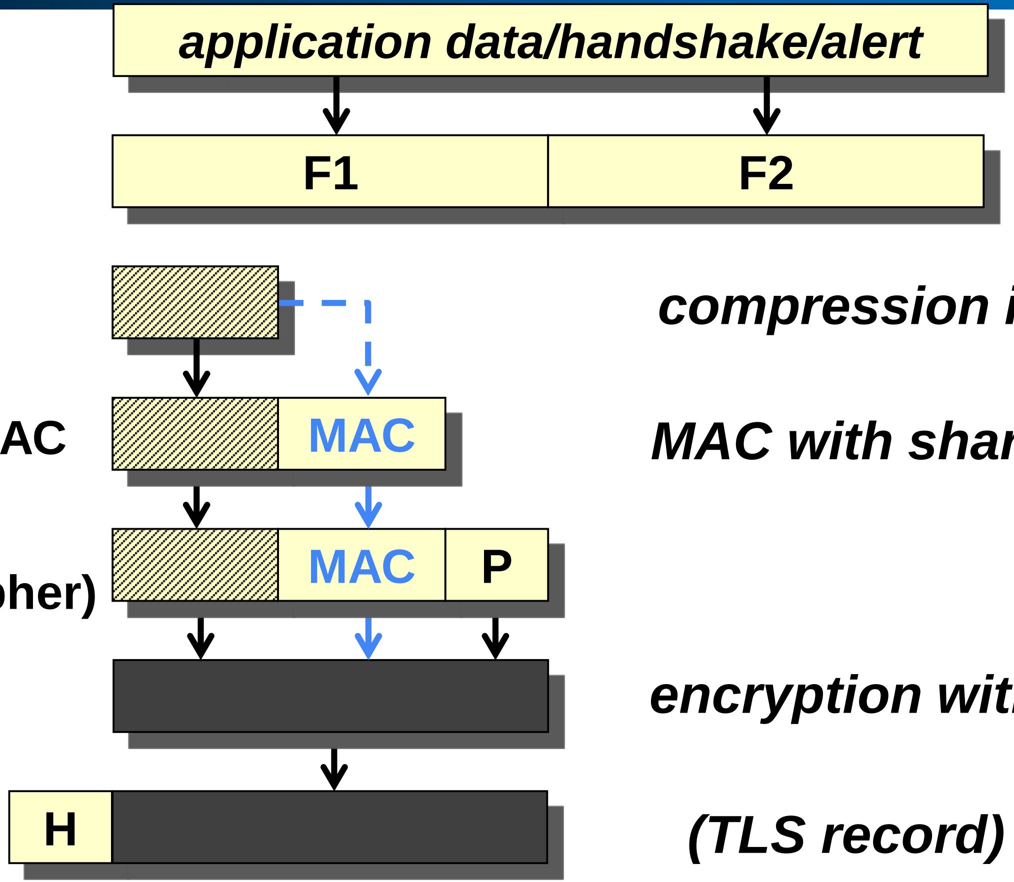
TLS architecture



TLS architecture (protocols)

- **TLS handshake protocol**
  - specific handshake messages used in TLS handshake
- **change cipher spec protocol**
  - used to trigged the change of algorithms (and keys) to be used for message protection
  - some analysis suggest it could be eliminated (absent in TLS 1.3)
- **(application) data protocol, e.g. HTTP, FTP, SMTP, POP3, …**
- **TLS alert (teardown) protocol**
  - specific TLS alert messages to close the channel or indicate errors (!)
- **TLS record protocol**
  - specifies the format to cryptographycally protect the application data, handshake, alert, or change cipher spec messages

TLS record protocol (authenticate-then-encrypt)



fragmentation — ***fragment is at most 16KB***

[compression] — ***compression is optional***

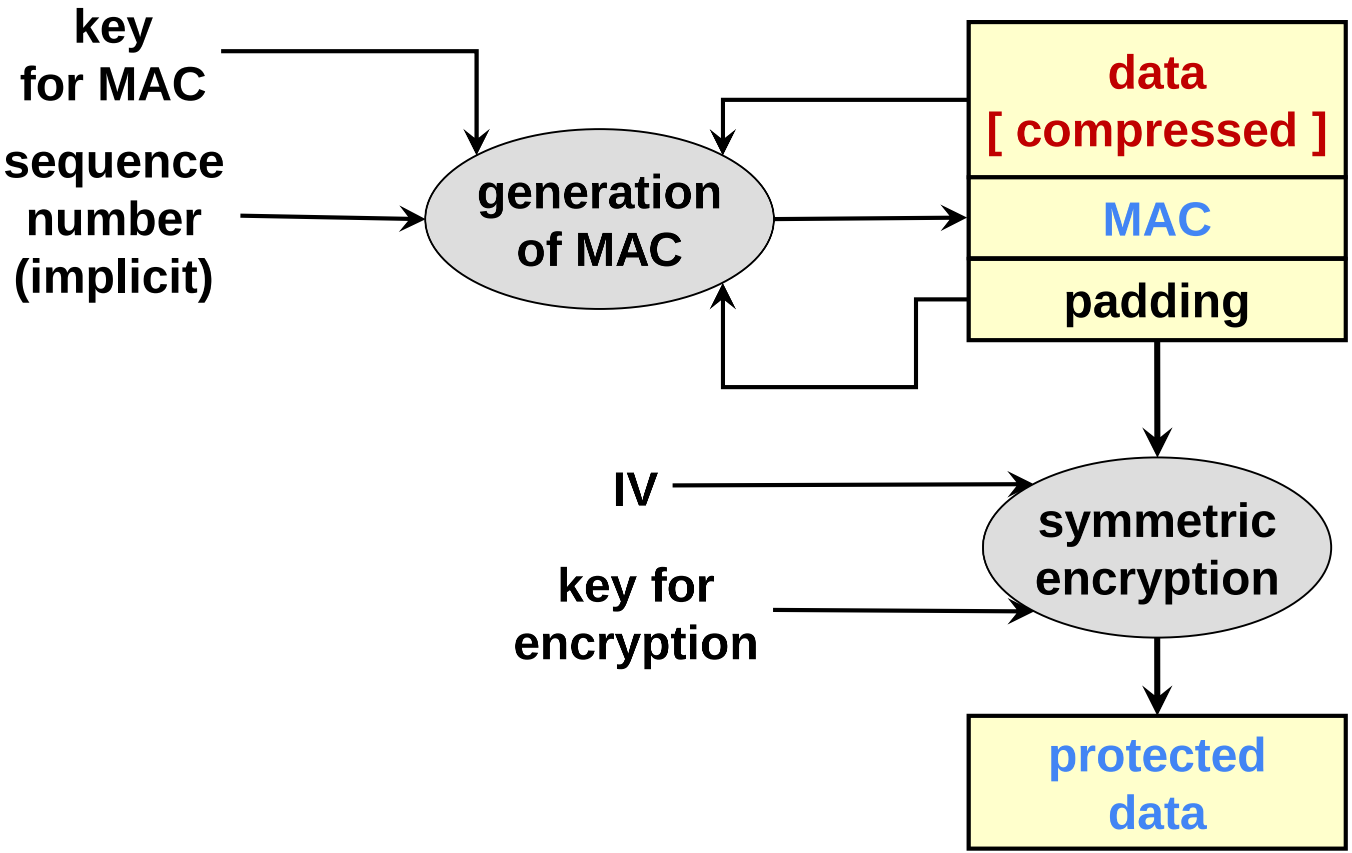
computation of MAC — ***MAC with shared key***

padding (if using block cipher)

encryption — ***encryption with shared key***

header — ***(TLS record)***

TLS 1.2 data protection (authenticate-then-encrypt)



TLS – computation of MAC

```
MAC = message_digest ( key, seq_number || type
|| version || length || fragment )
```

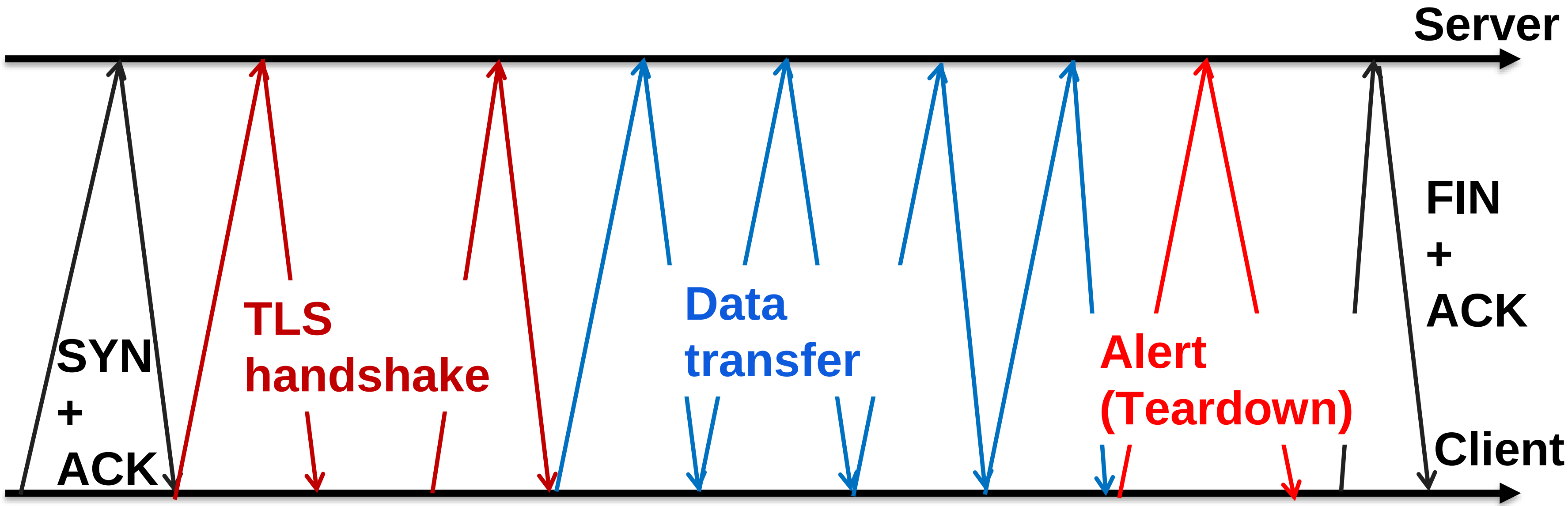
- `message_digest` – typically, HMAC-SHA256 or better
- `key` – client_write_MAC_key or server_write_MAC_key
- `seq_number` – 64-bit integer
  - starts from 0 for a new connection, cannot exceed 2**64-1
  - never transmitted but computed implicitly by client and server
- `type` – application data, handshake, alert, change_cipher_spec
- `version` – protocol version
- `length` – fragment length

TLS operation phases

TLS operation phases (high level)

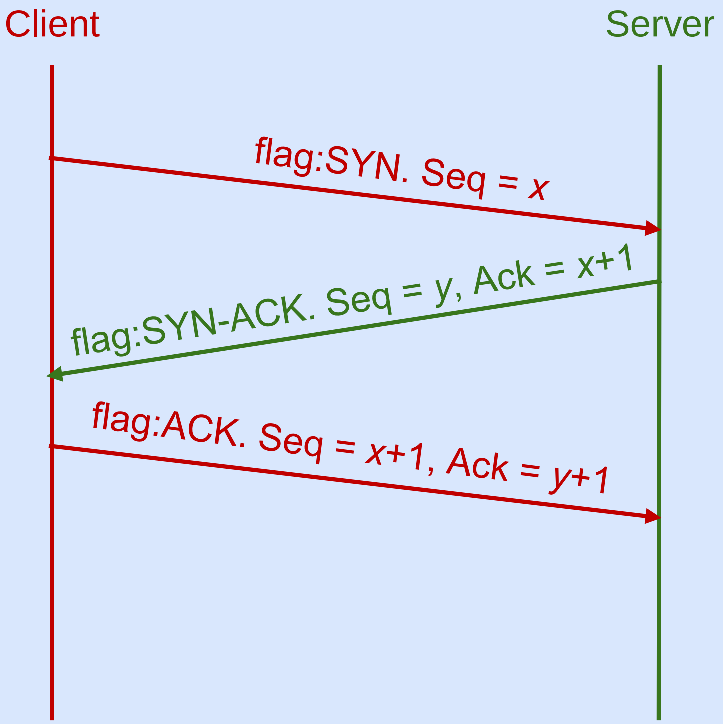
- **TCP connection (3-way handshake)**
- **TLS handshake**
  - authenticate server and optionally the client
  - negotiate (agree on) the cryptographic algorithms used for key exchange/agreement and data protection (MAC & encryption)
  - establish keys
- **(application) data transfer**
- **TLS teardown**
  - invoked by application closing connection
  - or due to error (during TLS handshake or data transfer)

TLS Operation Phases



Review: TCP 3-Way Handshake

1. Client chooses an initial sequence number x its bytes and sends a SYN (synchronize) packet to the server
2. Server chooses an initial sequence number *y* for its bytes and responds with a SYN-ACK packet
3. Client then returns with an ACK packet
4. Once both hosts have synchronized TCP sequence numbers, the TCP connection is “established”



Review: Transmission Control Protocol (TCP)

- **provides reliability**
  - the destination sends **acknowledgements** (ACKs) for each sequence number received
  - if the source doesn’t receive the ACK, the source sends the packet again
- **provides ports**
  - multiple services (on a server) can share the same IP address by using different ports
- **does not provide security (TCP segments are not cryptographically protected in any way)**

Review: Ports

- **ports help us distinguish between different applications on the same computer or server**
  - on clients, port numbers (chosen for communication with servers) are typically random
  - on (public) servers, port numbers are typically constant and well-known (range 1-1024) so clients can access the right port for each service
- **remember: TCP is built on top of IP, so the IP header (and therefore the IP address) is still present**

IP Header: send to: 130.197.15.69

TCP Header: send to: port 80

Let’s meet at 17:00 at Room 2

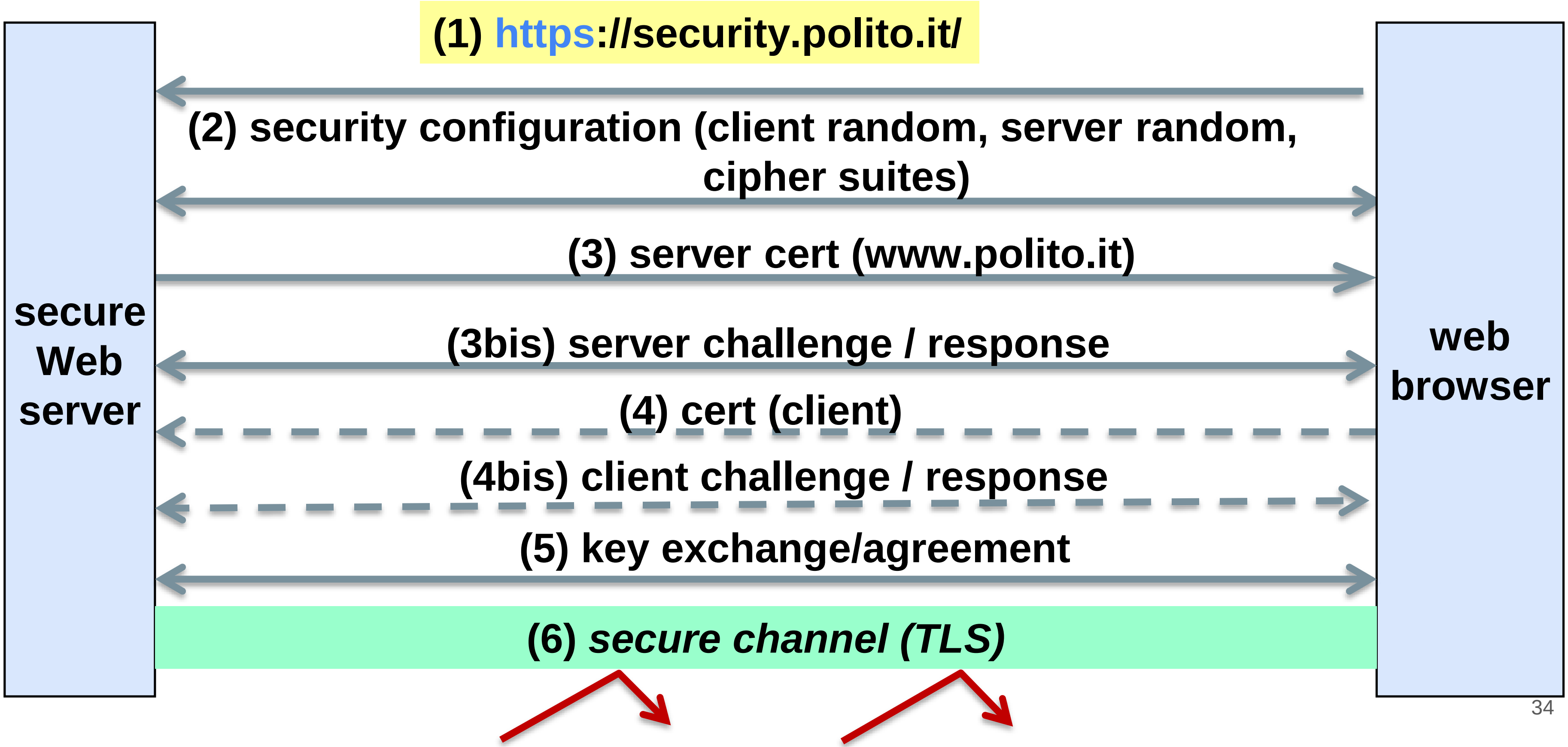
Review: TCP - Ending/Aborting a TCP Connection

- **to end a TCP connection, one side sends a packet with the FIN (finish) flag set, which should then be acknowledged**
  - this means "I will no longer be sending any more packets, but I will continue to receive packets"
  - once the other side is no longer sending packets, it sends a packet with the FIN flag set
- **to abort a TCP connection, one side sends a packet with the RST (reset) flag set**
  - this means "I will no longer be sending nor receiving packets on this connection"
  - RST packets are not acknowledged since they usually mean that <u>something went wrong</u>

Review: TCP Flags (I)

- **SYN - indicator of the beginning of the TCP connection**
- **ACK - indicates that the receiver is acknowledging the receipt of something (in the ack number)**
  - typically always set except the very first TCP segment sent
- **FIN - one way to end the TCP connection (e.g. after finishing data transfer)**
  - requires an acknowledgement
  - no longer sending packets, but will continue to receive packets
- **RST - abort a connection**
  - does not require an acknowledgement
  - no longer sending or receiving packets

TLS handshake (high level view): example for https



TLS handshake protocol - summary

- **exchange random numbers between the client and the server to be used for the subsequent generation of the keys**
  - client random, server random
  - generated for each connection
- **negotiate the session-id (< TLS 1.2)**
  - 32 bytes selected by the server
- **agree on a set of algorithms for protecting messages and key exchange (confidentiality, data authentication & integrity) – cipher suites**
- **exchange the necessary X.509 certificates (server – mandatory, client - optional) + peer authentication**

TLS handshake protocol - summary (II)

- **perform key exchange/agreement: establish a SECRET (called master secret) derived from another SECRET (called pre-master secret) exchanged by means of public key operations (DH, RSA)**
- **from the master secret (along with the client random and the server random) the client and the server will derive via a PRF the cryptographic symmetric keys (keys for MAC, keys for encryption) and the initialization vectors for data protection**
  - different for each connection
  - distinct encryption and authentication (MAC) keys
  - distinct (encryption and MAC calculation) keys – messages from client to server and from server to client

TLS session-id (< TLS 1.2)

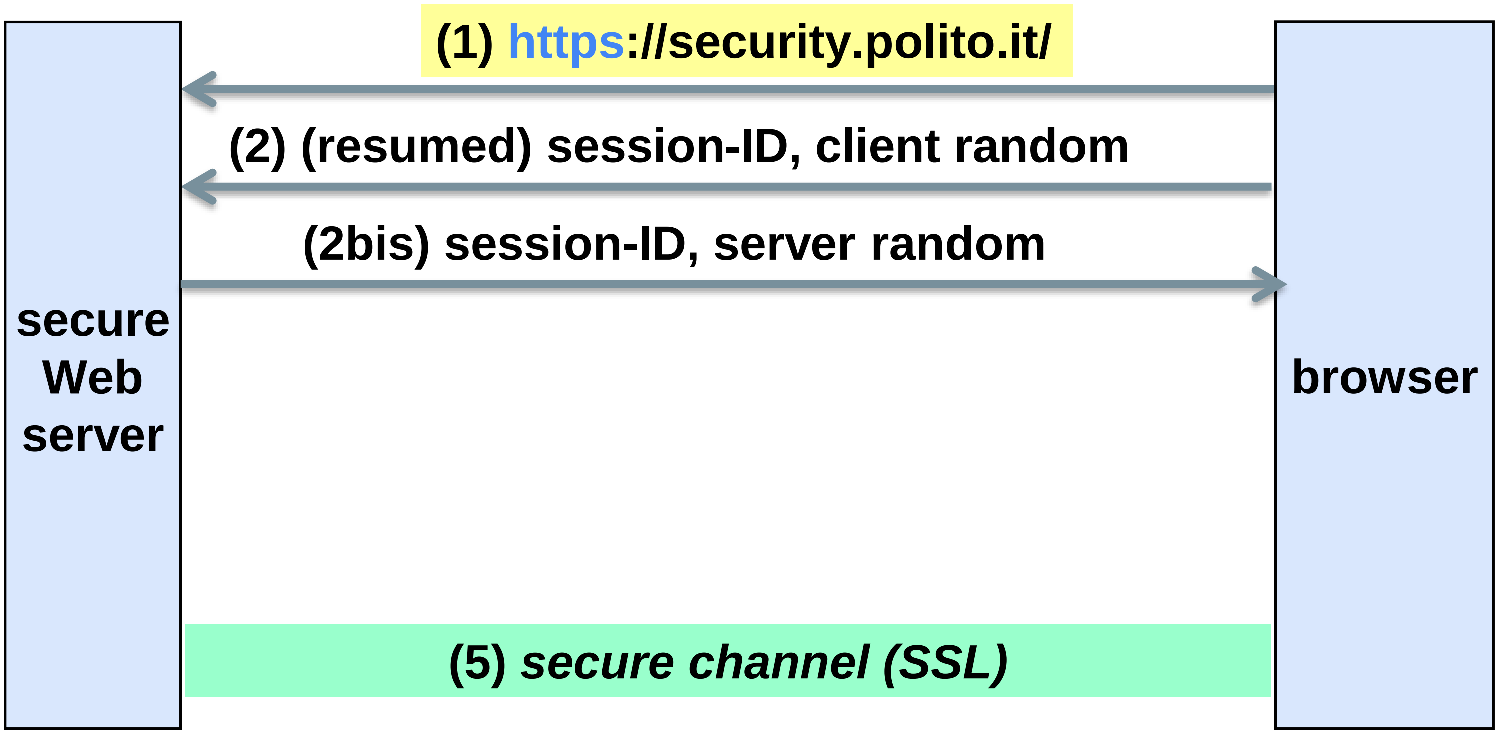
- **Typical web transaction:**
  - 1. open, 2. GET page.htm, 3. page.htm, 4. close
  - 1. open, 2. GET home.gif, 3. home.gif, 4. close
  - 1. open, 2. GET logo.gif, 3. logo.gif, 4. close
  - 1. open, 2. GET back.jpg, 3. back.jpg, 4. close
  - 1. open, 2. GET music.mid, 3. music.mid, 4. close

If the TLS cryptographic parameters must be negotiated every time, then the computational load becomes high.

TLS session-id (< TLS 1.2)

- **in order to avoid re-negotiation of the cryptographic parameters for each TLS connection, the TLS server can send a session identifier (that is, more connections can be part of the same logical session)**
- **if the client, when opening the TLS connection, sends a valid *session-id* then the peer authentication and key exchange/agreement part are skipped, keys are derived based on a previously negotiated master-secret and the (new random numbers)**
  - data immediately exchanged over the secure channel
- **the server can reject the use of session-id (always or after a time passed after its issuance)**

TLS handshake with session-ID (< TLS 1.2)

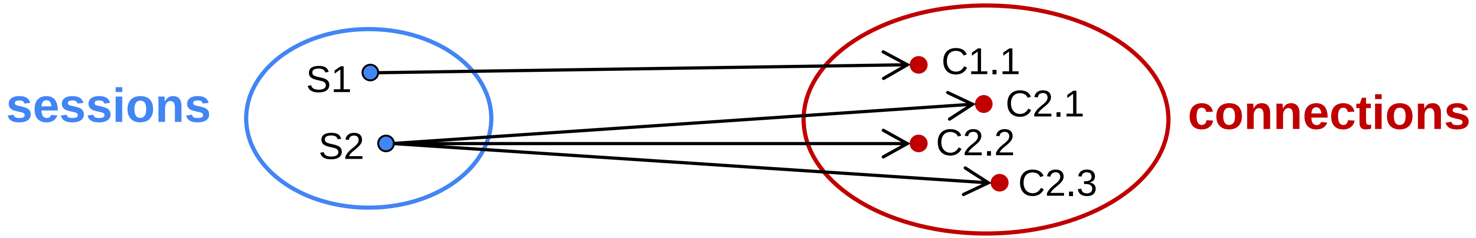


TLS sessions and connections

- **TLS session**
  - a logical association between client and server
  - may span multiple TLS connections for efficiency (1:N)
  - created by the TLS handshake protocol
  - defines a set of cryptographic parameters common for several connections
    - X.509v3 peer certificate
    - cipher spec (algorithms) – encryption, MAC
    - master secret – 48 bytes, known to both

TLS sessions and connections

- **TLS connection**
  - a transient TLS channel between client and server
  - associated to one specific TLS session (1:1)
  - defines crypto parameters for <u>each</u> TLS connection (server and client sequence numbers, server and client random, cryptographic keys and IVs for data protection)



TLS security services

TLS services

- **server authentication (mandatory)**
  - using public-key certificates and asymmetric challenge/response
- **client authentication (optional – if required by server)**
  - using public key certificates and asymmetric challenge/response
- **data protection**
  - confidentiality
  - message (data) integrity and authentication
  - reliability: protection against re-ordering, replay, cancellation of messages
- **efficiency: allow resumption of TLS session in new connection (no need to re-do TLS handshake)**

TLS – authentication and integrity

- **peer authentication (at channel setup):**
  - the server authenticates itself by sending its public key (X.509v3 certificate) and by responding to an implicit asymmetric challenge/response
  - the client authentication (with public key, X.509v3 certificate, and explicit asymmetric challenge/response
- **for (data) authentication and integrity of the data exchanged over the channel the TLS record protocol uses:**
  - MAC = keyed digest (HMAC-SHA256 or better)
  - an MID (seq_number) - to avoid *replay* and cancellation
  - algorithms and symmetric keys for HMAC calculation are negotiated in the TLS handshake

TLS - confidentiality

- **client and server negotiate symmetric algorithm to be used for data encryption**
  - block algorithms, e.g., AES, 3DES
  - in TLS v1.2, AEAD ciphers (AES in GCM or CCM mode) are also available, besides AES, 3DES (CBC mode)
  - starting with TLS v1.3 – only AEAD ciphers
  - stream algorithms (in TLS v1.2, RC4 still possible!)
- **client and server perform key exchange/agreement to derive the symmetric keys used for encryption of the data**
  - performed via asymmetric cryptography
    - Diffie-Hellman
    - RSA (no longer supported for key exchange in TLS v1.3)

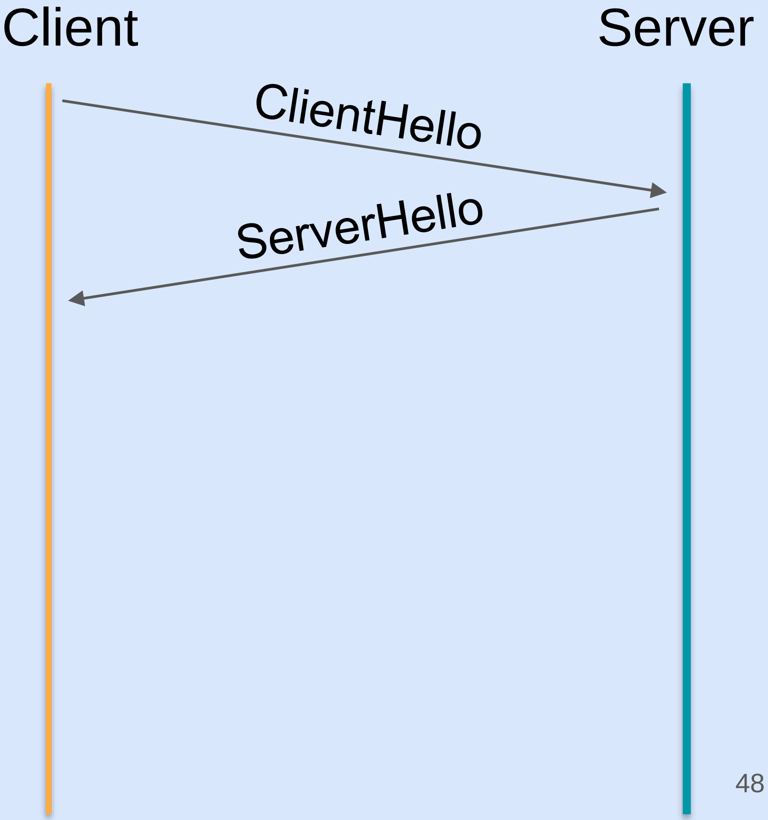
TLS ciphersuites

- **string specifying:**
  - key exchange algorithm (e.g. RSA, DH, in versions <TLS 1.2)
  - symmetric encryption algorithm (e.g. AES)
  - hash algorithm (for MAC)
- **examples:**
    - SSL_NULL_WITH_NULL_NULL
    - SSL_RSA_WITH_NULL_SHA
    - SSL_RSA_WITH_3DES_EDE_CBC_SHA
    - TLS_RSA_WITH_AES_128_CBC_SHA
    - TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
    - TLS_AES_128_GCM_SHA256

TLS Handshake (v1.2), server authentication only: steps and messages (high-level)

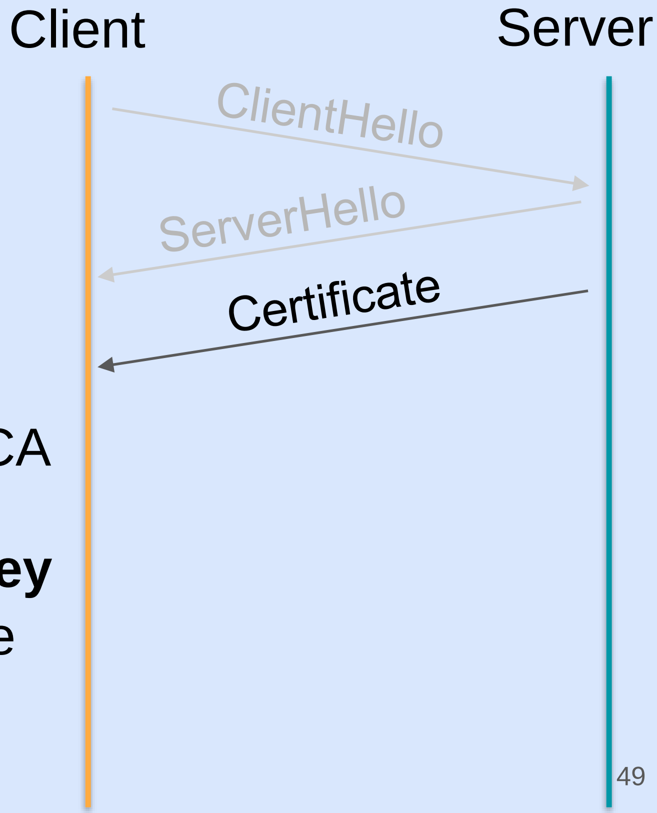
TLS Handshake Step 1: Exchange Hellos

- **the client sends ClientHello with**
  - a 256-bit random number *Rc* ("client random")
  - a list of supported cryptographic algorithms – cipher suites
- **the server sends ServerHello with**
  - a 256-bit random number *Rs* ("server random")
  - the algorithms to use (chosen from the client's list) – selected cipher suite
- ***Rc* and *Rs* prevent replay attacks**
  - *Rc* and *Rs* are randomly chosen for every TLS session



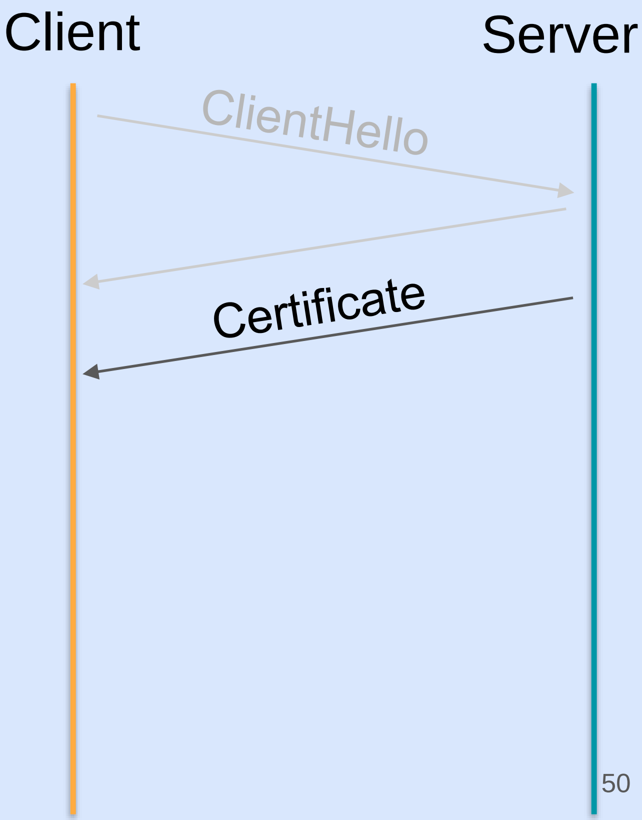
TLS Handshake Step 2: X.509v3 certificate

- **the server sends its X.509v3certificate**
  - server certificate: the server's identity and public key, signed by a trusted CA
  - … typically the whole cert. chain is sent
- **the client validates the certificate**
  - verifies the signature in the certificate, revocation, expiration, …
  - note: client needs to have the root/trusted CA already(!)
- **the client now knows the server's public key**
  - the client is not yet sure that is talking to the legitimate server (not a fake one)
  - remind: certificates are public. Anyone can provide a certificate for anybody



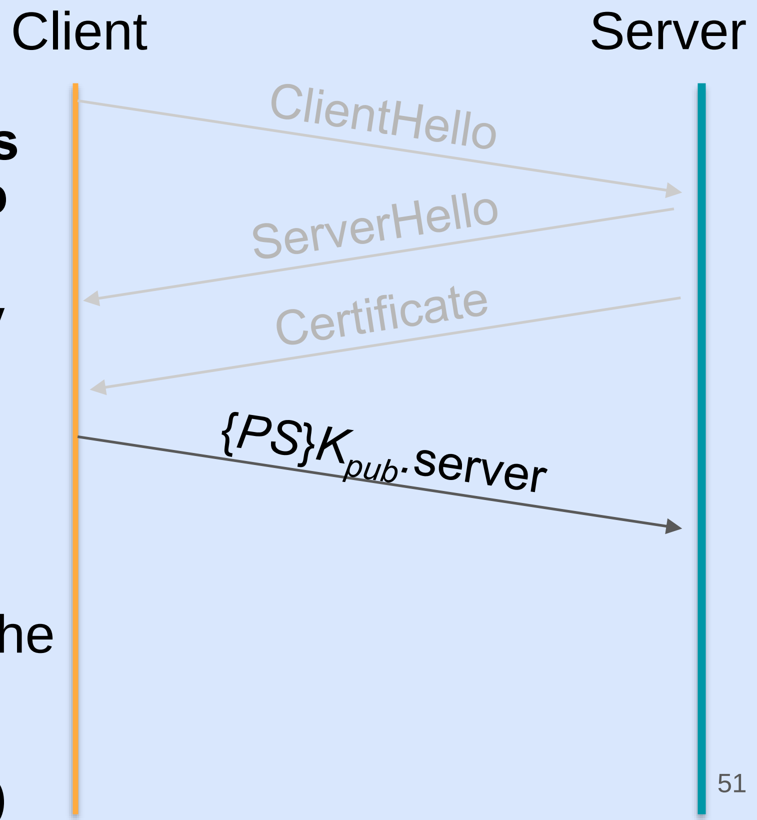
TLS Handshake Step 3: Premaster Secret

- **this step has two main purposes:**
  - makes sure the client is talking to the legitimate server (not a fake one)
    - the server must prove that it owns the K_{priv}.*server* corresponding to the K_{pub}.*server* in the certificate
  - allows the client and server exchange/agree on a **shared (premaster) secret**
    - an attacker should not be able to learn the (premaster) secret
    - this will help the client and the server secure messages later
- **two approaches to exchange a premaster secret: RSA or Diffie-Hellman**



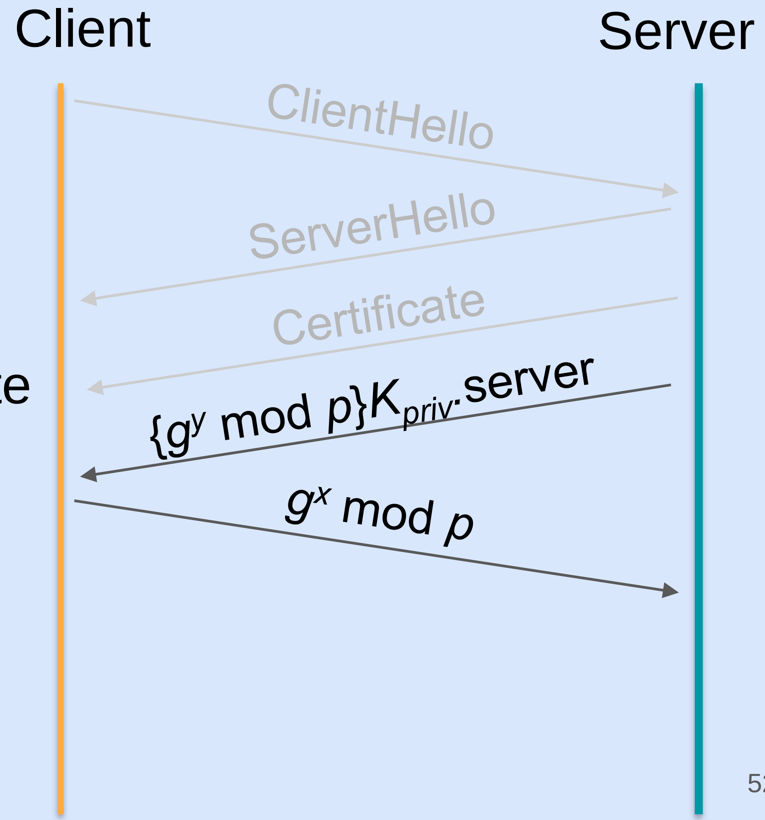
TLS Handshake Step 3: Premaster Secret (RSA)

- **the client randomly generates a premaster secret (PS)**
- **the client encrypts PS with the server's public key (K_{pub}.server) and sends it to the server**
  - the client knows the server's public key from the certificate
- **the server decrypts the PS with his private key (K_{priv}.server)**
- **client and server now share the PS**
  - only the legitimate server can decrypt the PS
  - proves that the server owns the private key (otherwise, it could not decrypt PS)



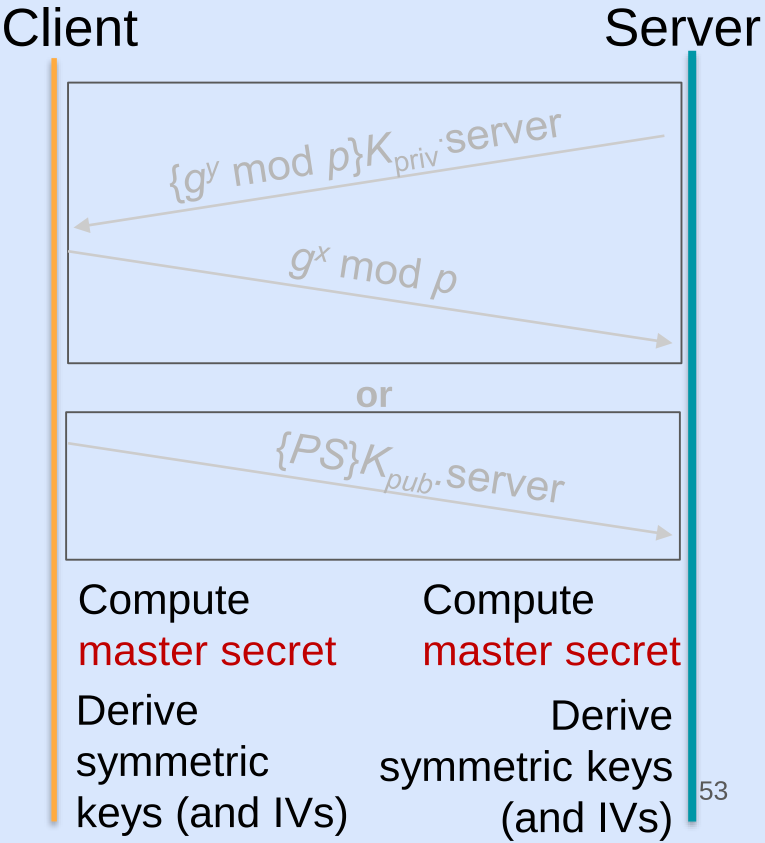
TLS Handshake Step 3: Premaster Secret (DH)

- **the server generates a (DH) secret y and computes g^y mod p**
- **the server signs g^y mod p with its private key and sends the message and signature**
- **the client verifies the signature**
  - proves that the server owns the private key
- **the client generates a secret x and computes g^x mod p**
- **the client and server now share a premaster secret: g^{xy} mod p**
  - recall Diffie-Hellman: an attacker cannot compute g^{xy} mod p

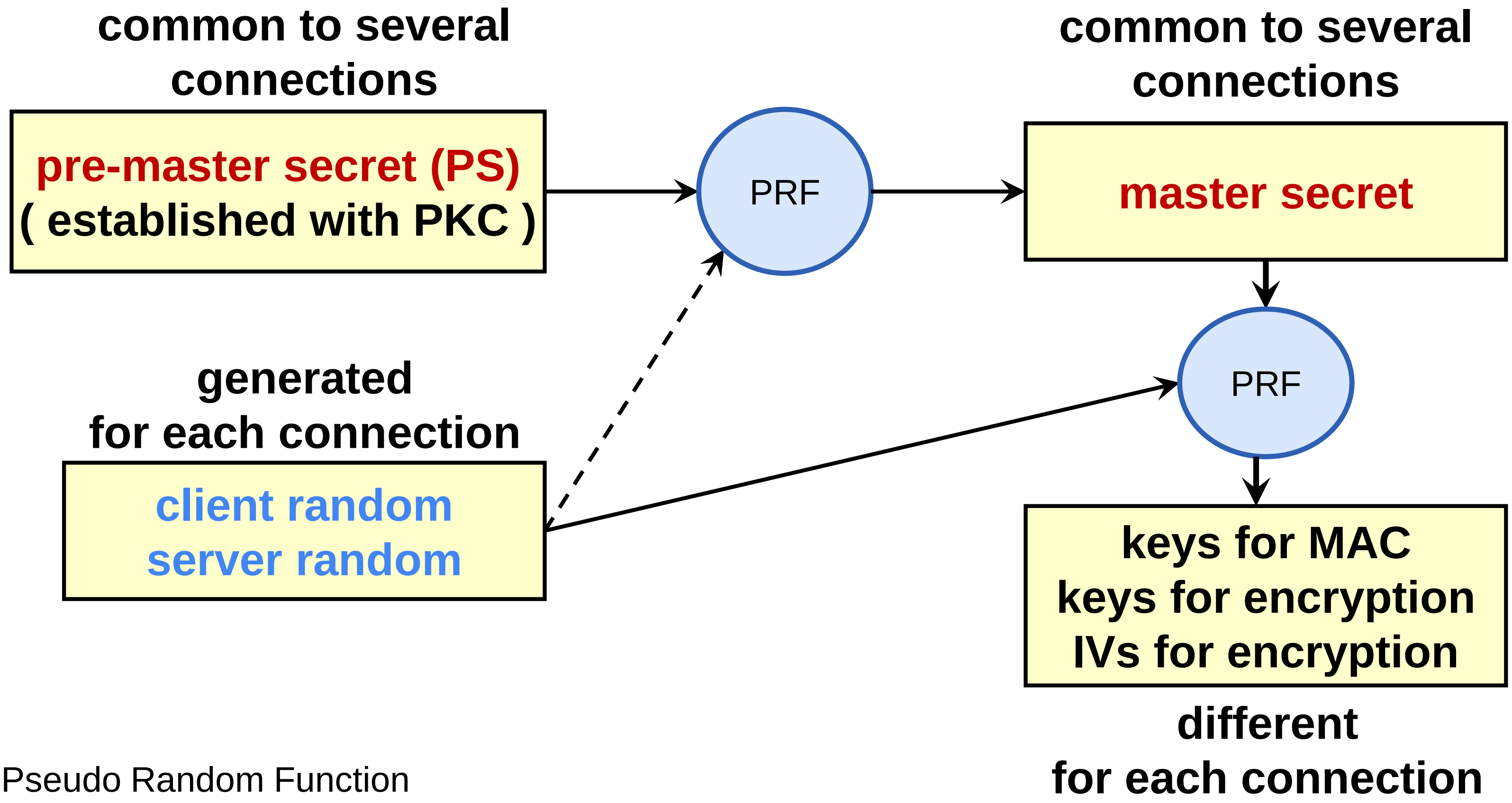


TLS Handshake Step 4: Derive Symmetric Keys

- **server and client each derive a master secret (via PRF) from Rc, Rs, and PS**
  - changing any of the values results in different symmetric keys
- **from master secret, Rc, and Rs, client and server derive four symmetric keys**
  - *two keys ($K_{enc}c$ and $K_{enc}s$)* - used for encrypting client-to-server and server-to-client messages
  - *two keys ($K_{IA}c$ and $K_{IA}s$)* - used for calculating MAC for the client-to-server and server-to-client messages
  - … plus two IVs (for encryption), one for each side
- **both sides know all four keys (and IVs)**



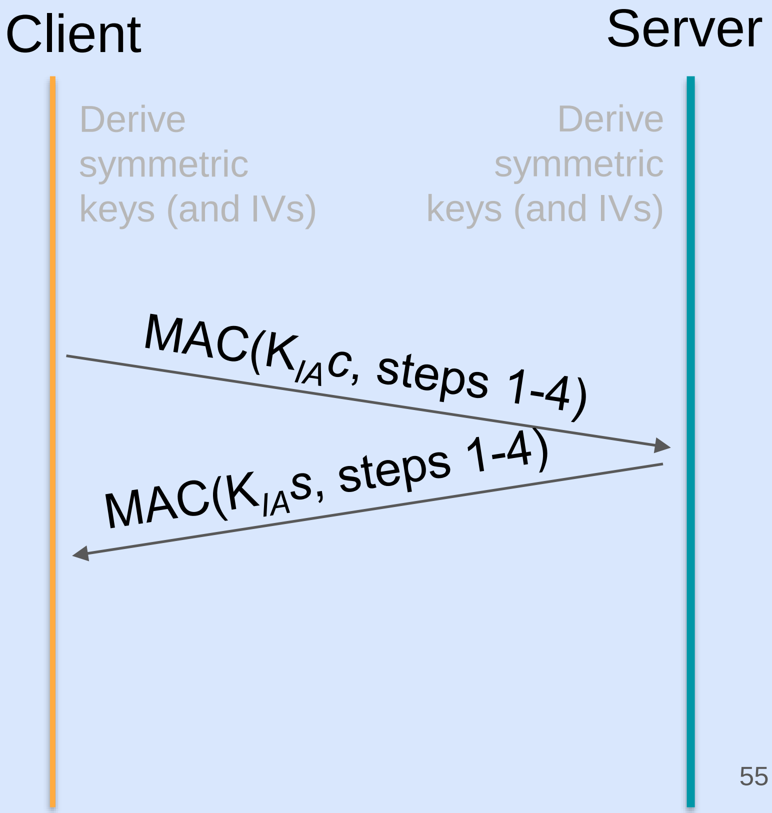
TLS Handshake: deriving symmetric keys (& IVs)



PRF = Pseudo Random Function

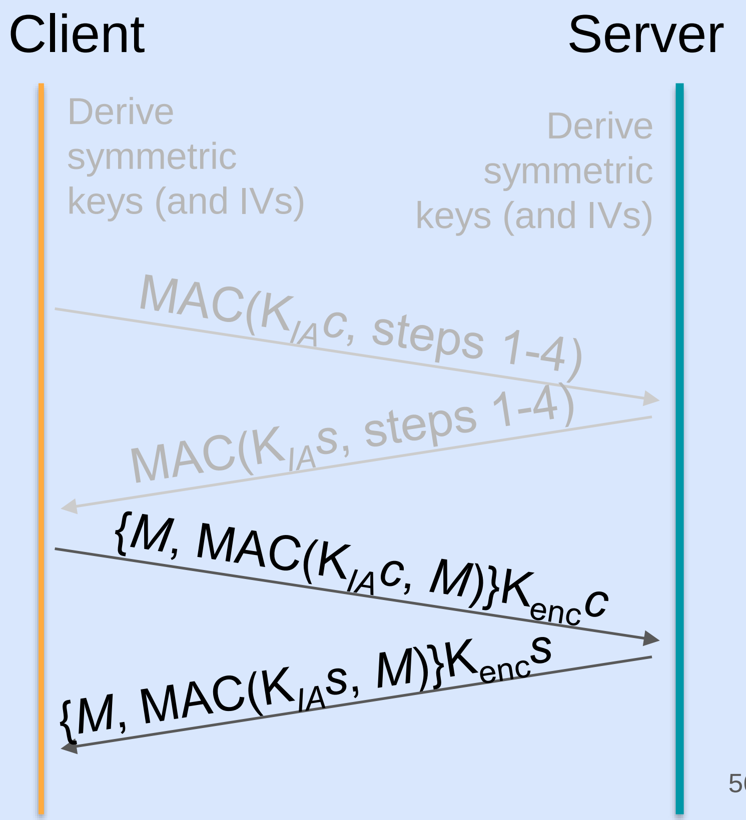
TLS Handshake Step 5: Exchange MACs

- **the server and client exchange MACs on all the messages of the TLS handshake so far**
- **very important step:**
  - protects the TLS handshake: any tampering on (any of the) the TLS handshake messages will be detected
    - in case of error, the connection is immediately aborted (no key saved)
  - if the server was an impostor (fake one), he could not derive the right key (K_{IA}s) used to calculate the MAC
    - so, this step serves for server authentication as well



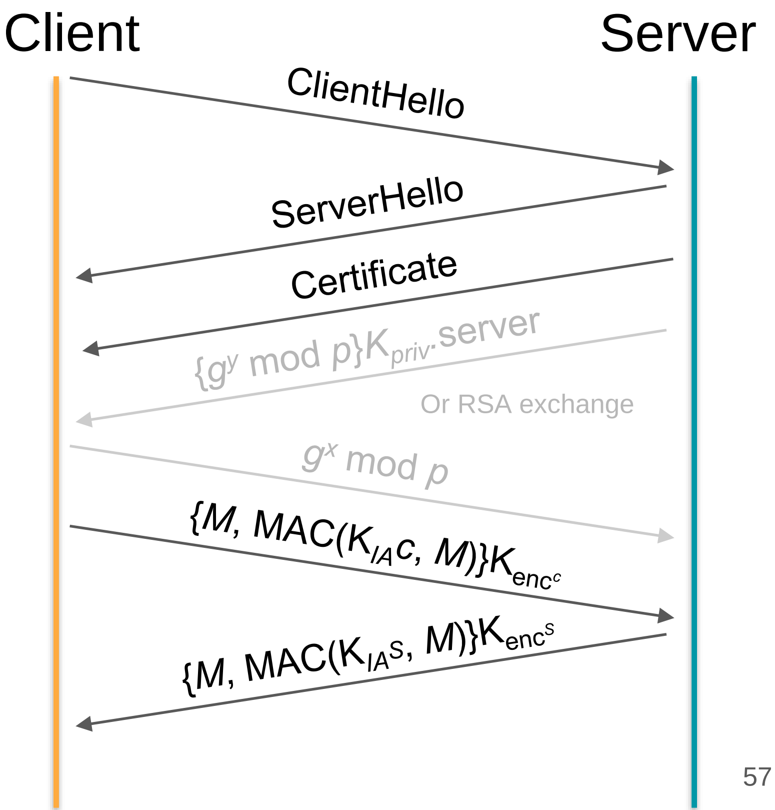
TLS Handshake Step 6: Send Messages

- **messages (application data) can now be sent securely**
  - AtE: first calculate the MAC, then encrypt
  - note (TLS 1.2): TLS uses MAC-then-encrypt, even though encrypt-then-MAC is generally considered better



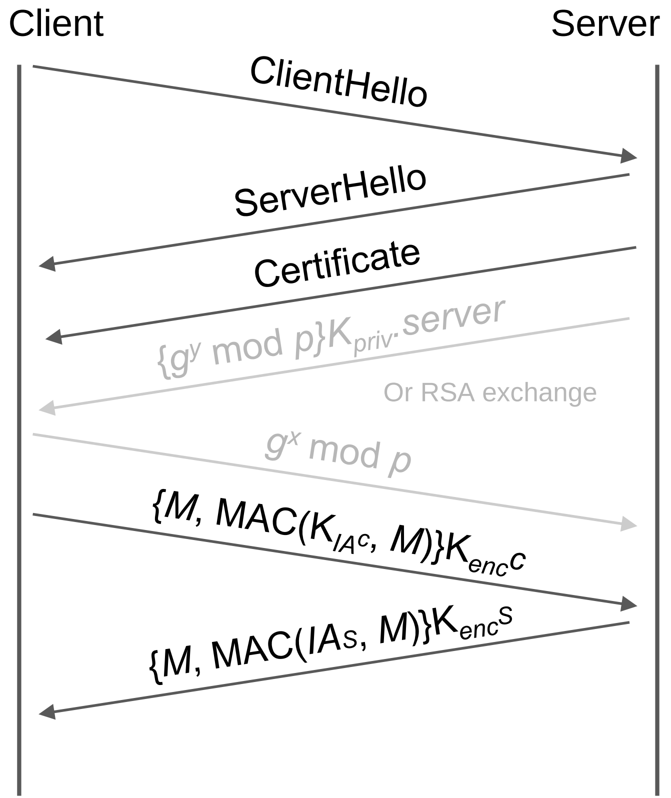
TLS: Talking to the Legitimate Server

- **how can the client be sure is talking to the legitimate server?**
  - the server sent its certificate, so the client knows the server's public key
  - the server proved that it owns the corresponding private key
    - RSA: the server decrypted the PS
    - DH: the server signed its half of the DH exchange
    - …and calculated the correct MAC (with derived key) in step 5
- **an attacker would not have the server's private key (assuming he has not compromised the server)**



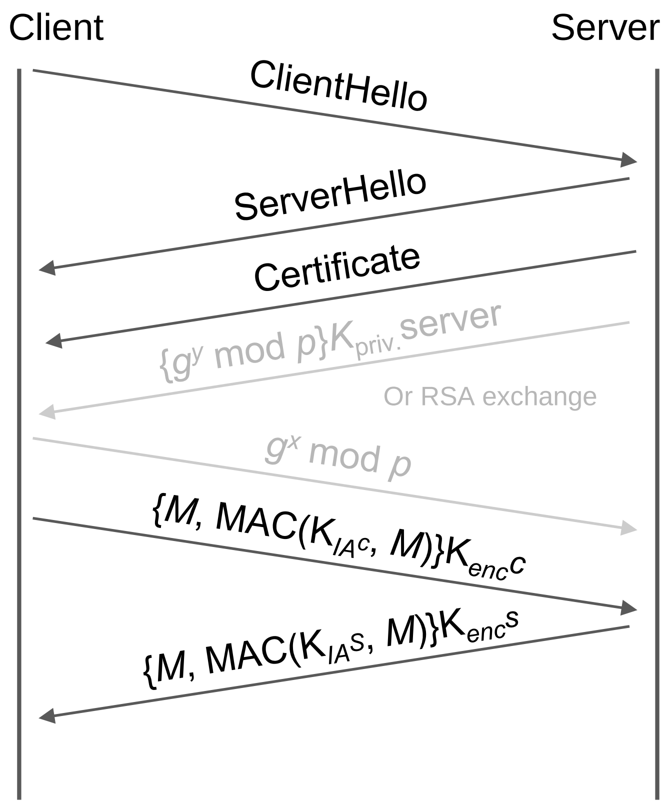
TLS: Secure messages

- **how can we be sure that attackers can't read or tamper with messages exchanged between client and server?**
- **encryption and MACs in TLS records provide confidentiality, data authentication and integrity for messages M**
  - application data or
  - TLS specific messages (handshake, alert, change cipher spec)



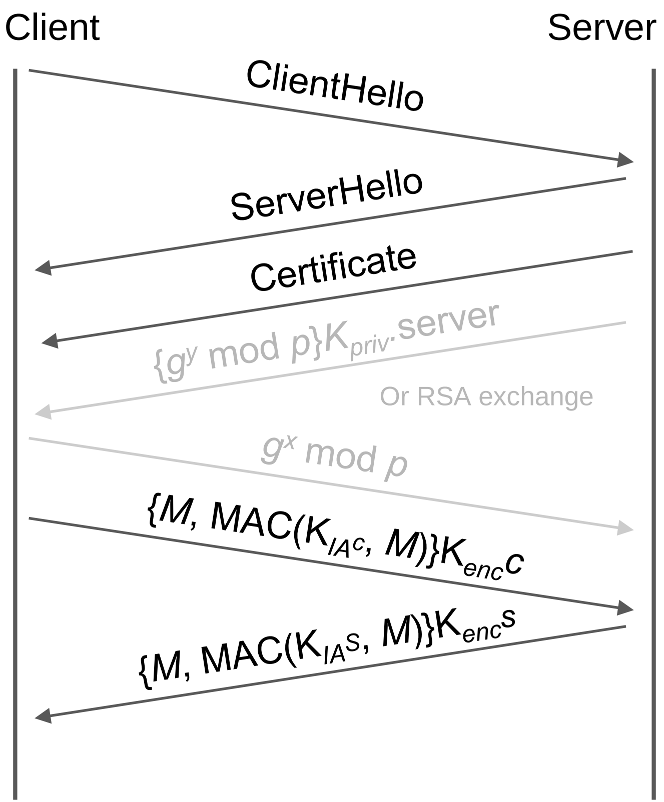
TLS: Secure Messages (II)

- **the attacker doesn't know PS**
  - RSA: PS was encrypted with the server's public key
  - DH: an attacker cannot learn the (Diffie-Hellman) secret
  - … so he cannot know the master secret
- **the symmetric keys are derived individually from master secret (and Rc, Rs)**
  - the attacker doesn't know the symmetric keys used to encrypt and calculate MAC on messages



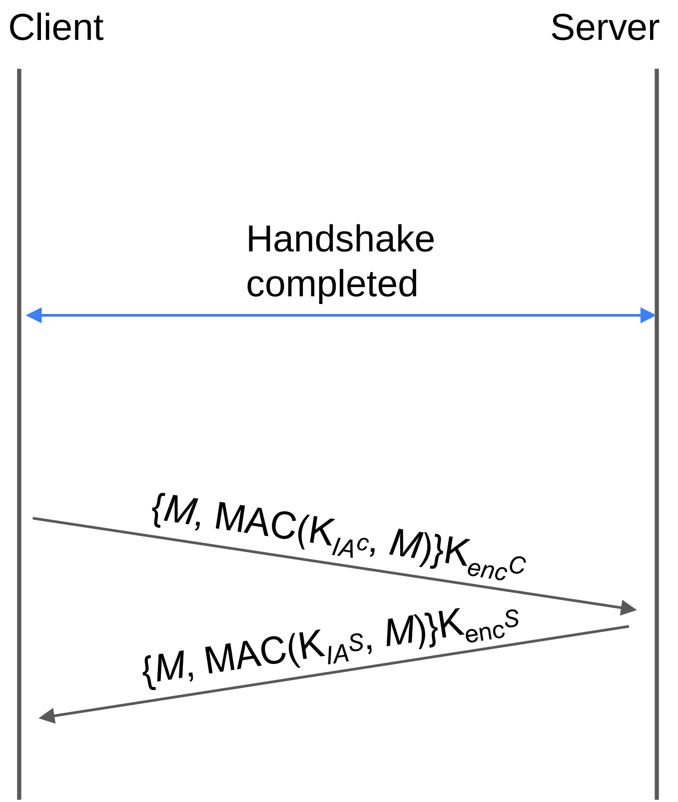
TLS: Replay Attacks

- **how can we be sure that the attacker hasn't replayed old records from a past TLS connection?**
- **every handshake uses a different Rc and Rs**
- **the symmetric keys are derived from Rc and Rs (and master secret)**
  - the symmetric keys are different for every TLS connection



TLS: Replay/Filtering Attacks

- **how can we be sure that the attacker hasn't replayed old records from the current TLS connection or filtered part of them?**
- **add TLS sequence numbers in the encrypted TLS record**
  - every TLS record uses a unique sequence number (seq_number)
  - if the attacker replays/filters/reorders a record, the TLS seq_number (at the server) will be wrong, thus when MAC is checked will generate error



TLS sequence numbers and TCP sequence numbers

- **TLS sequence numbers are not TCP sequence numbers**
  - TLS sequence numbers are encrypted and used for security
  - TCP sequence numbers are unencrypted and used for reliability in TCP

Source Port (16 bits)		**Destination Port (16 bits)**
Sequence Number (32 bits)		
Acknowledgement Number (32 bits)		
Data Offset (4 bits)	**Flags (12 bits)**	**Window Size (16 bits)**
Checksum (16 bits)		**Urgent Pointer (16 bits)**
Options (variable length)		
Data (variable length)		

TCP segment format

type	
major	**minor**
length	
. . . fragment [length] . . .	

TLS record format

Forward Secrecy in TLS

Forward Secrecy : premises

- **If a server has an X.509 certificate valid for both signature and encryption**
- **… then it can be used both for (peer) authentication (via a signature) and key exchange (asymmetric encryption of the session key)**
- **… but if**
  - an attacker intercepts and copies all the encrypted traffic
  - and later discovers the server’s (long term) private key
- **… then the attacker can decrypt all the traffic, past, present, and future**

perchè all'inizio della comunicazione le informazioni passano in chiaro

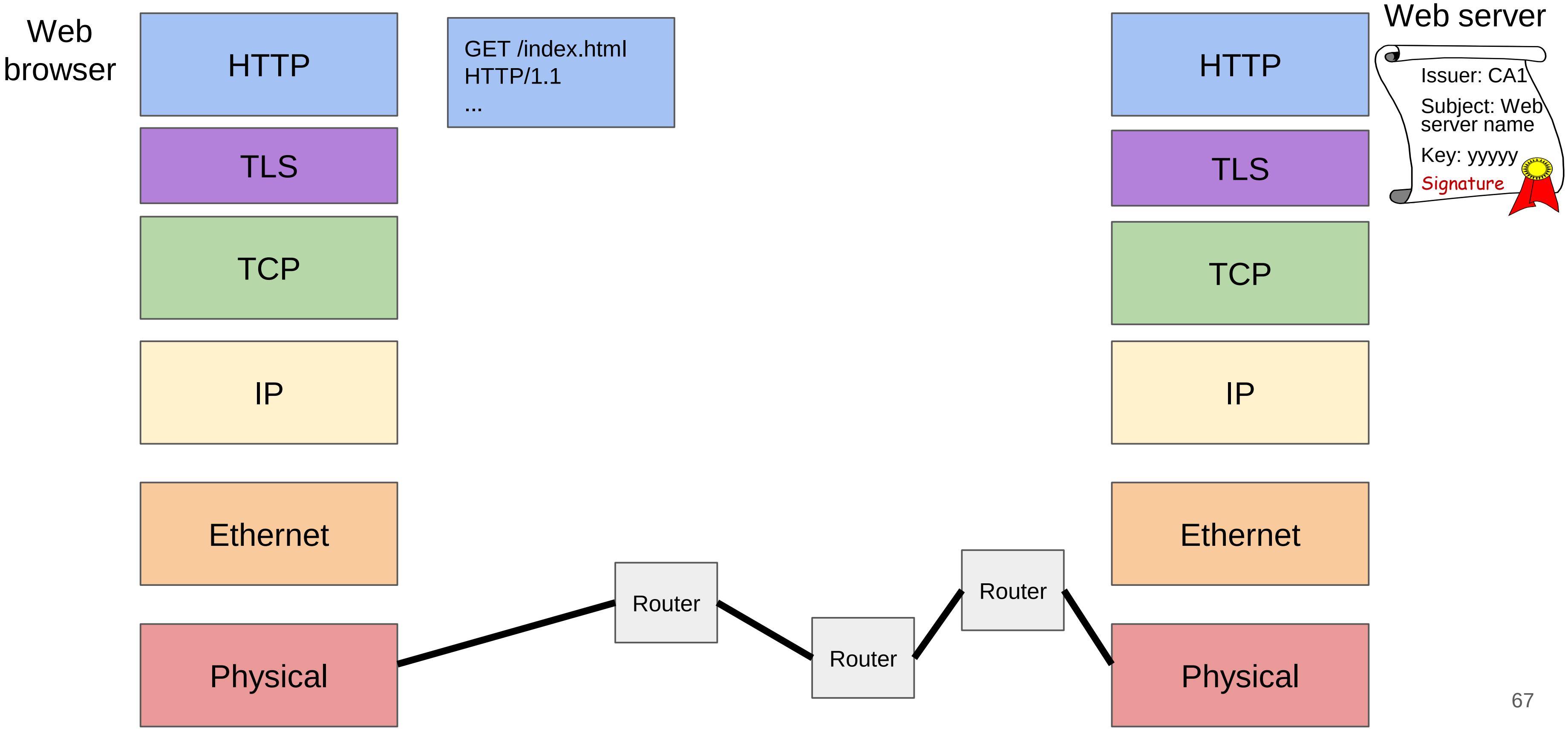
Perfect forward secrecy

- **What is perfect forward secrecy?**
  - the compromise of a server’s (asymmetric) private key compromises only the current (and eventually future) traffic but not the traffic that has been exchanged in the past
- **Premaster secret exchanged with RSA: No forward secrecy is achieved**
  - the adversary can record client and server random (Rc, Rs) and the encrypted PS
  - if the adversary later compromises the server’s private key, it can decrypt PS and derive the keys (for encryption and MACs) protecting the TLS records!

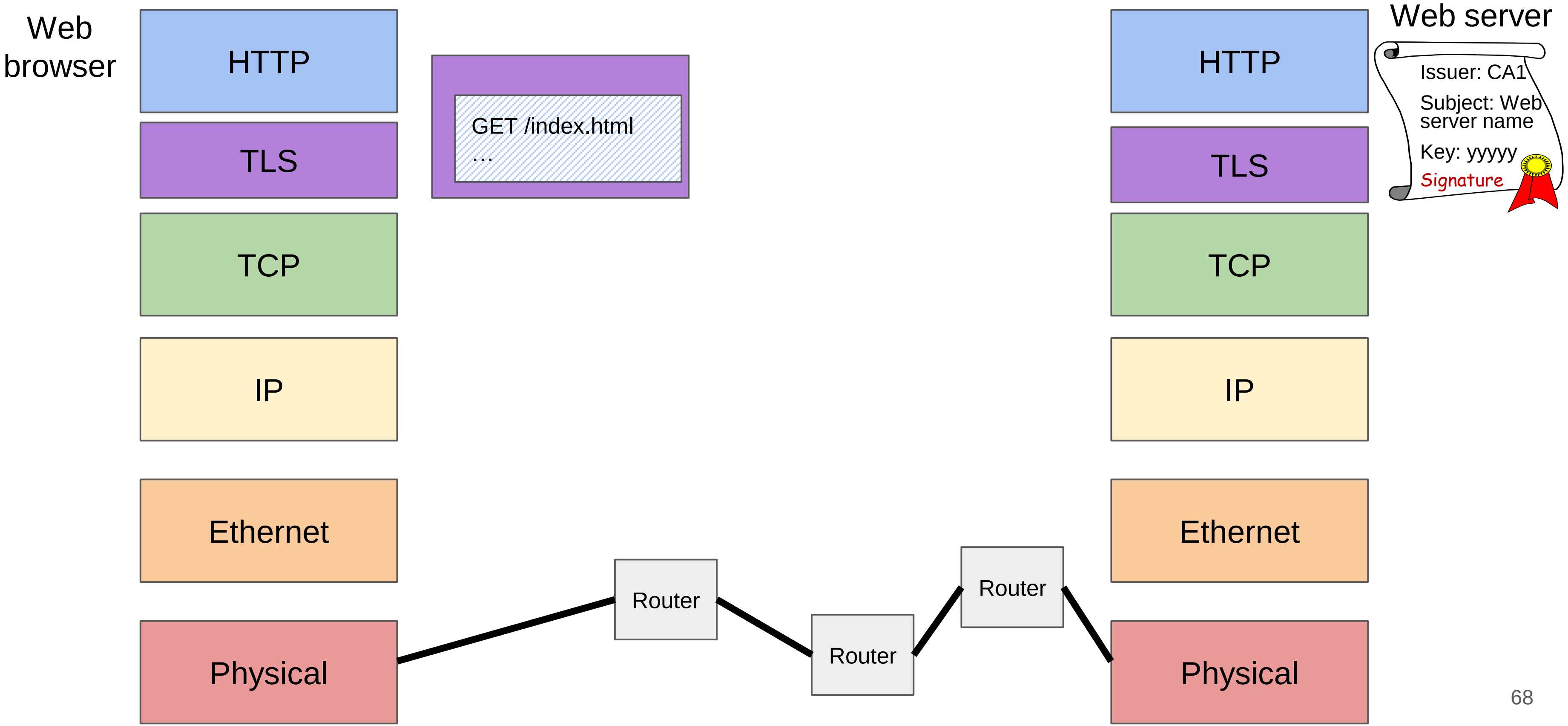
Forward secrecy: with 'ephemeral' mechanisms

- **Premaster secret agreed with Ephemeral DH (DHE): forward secrecy is achieved**
  - a DH one-time key is generated on the fly – session key
    - for authenticity, the server's DH public exponent must be signed
      - the server's long-term private key is used (only) for signing
  - thus we obtain perfect forward secrecy:
    - if the (temporary or short-lived) server's private key is compromised then the attacker can decrypt only the related traffic for that session
    - … while compromise of the long-term server's private key is an issue for authentication but not for confidentiality
  - examples: ECDHE

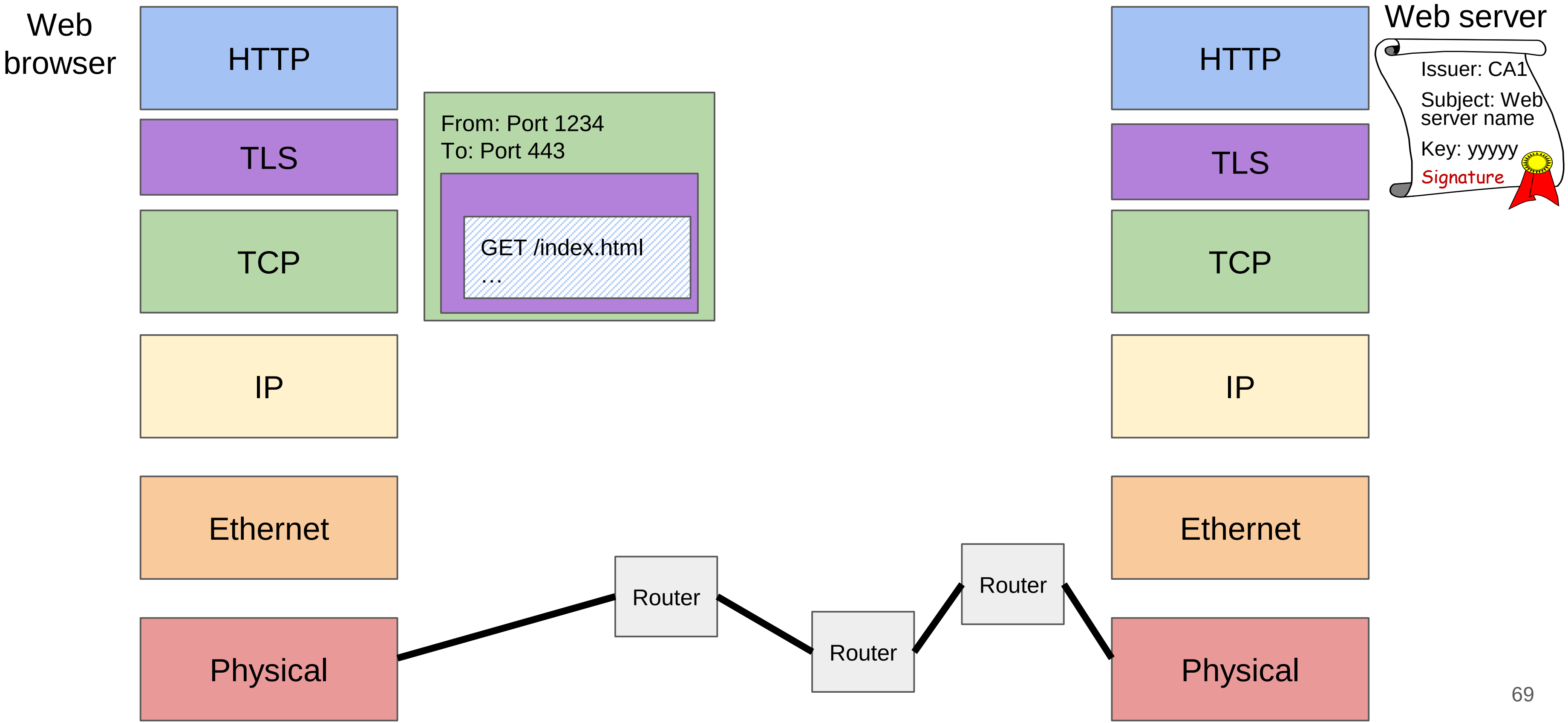
Data transfer with HTTP over TLS (HTTPS)



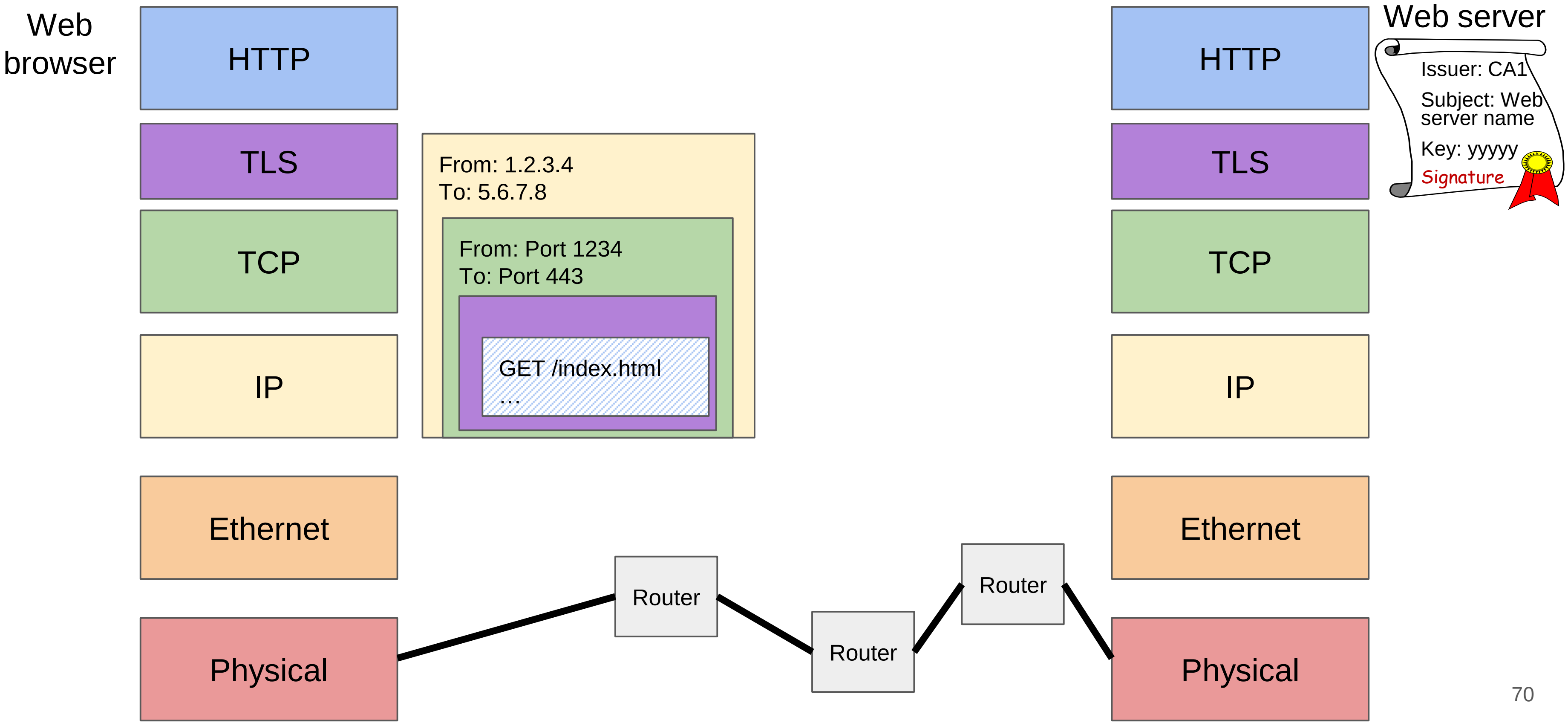
Data transfer with HTTP over TLS (HTTPS)



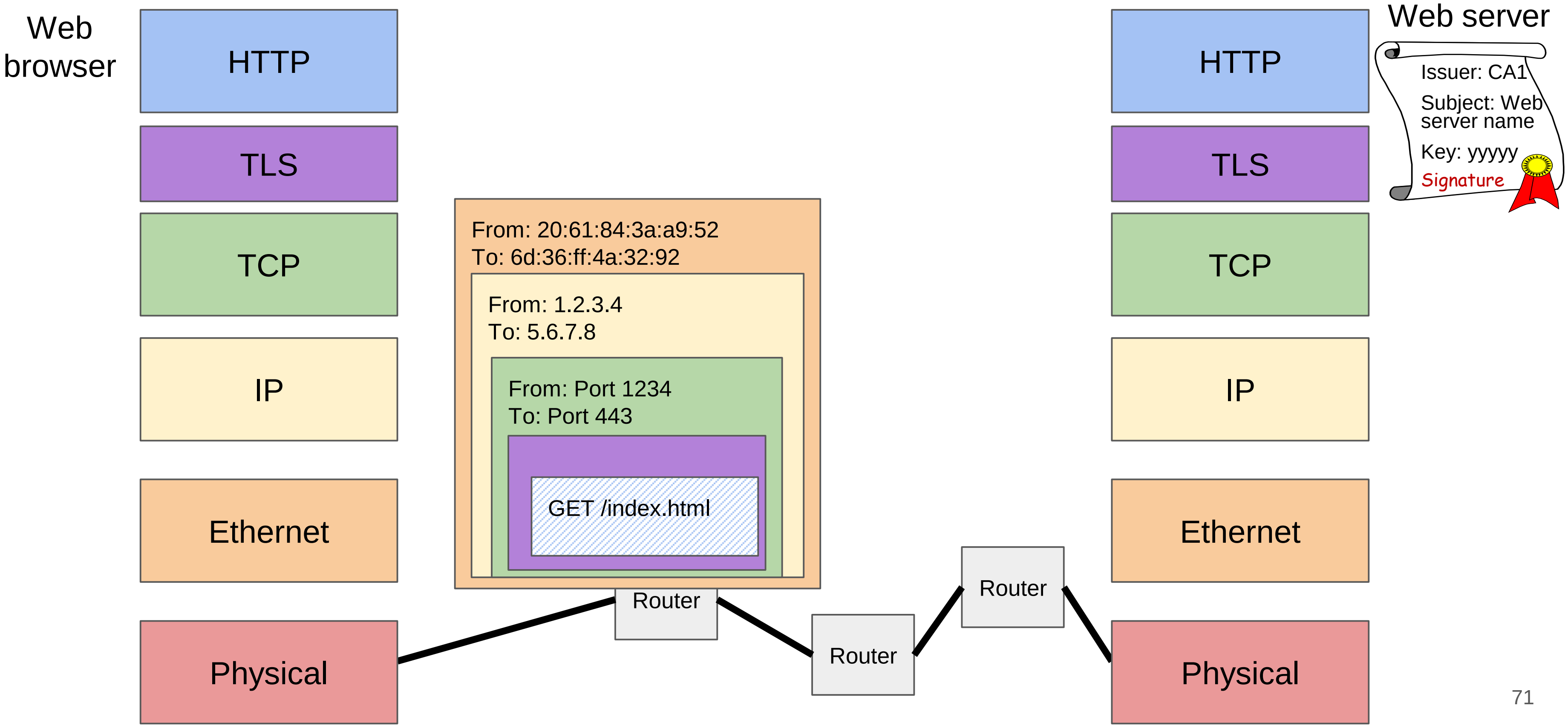
Data transfer with HTTP over TLS (HTTPS)



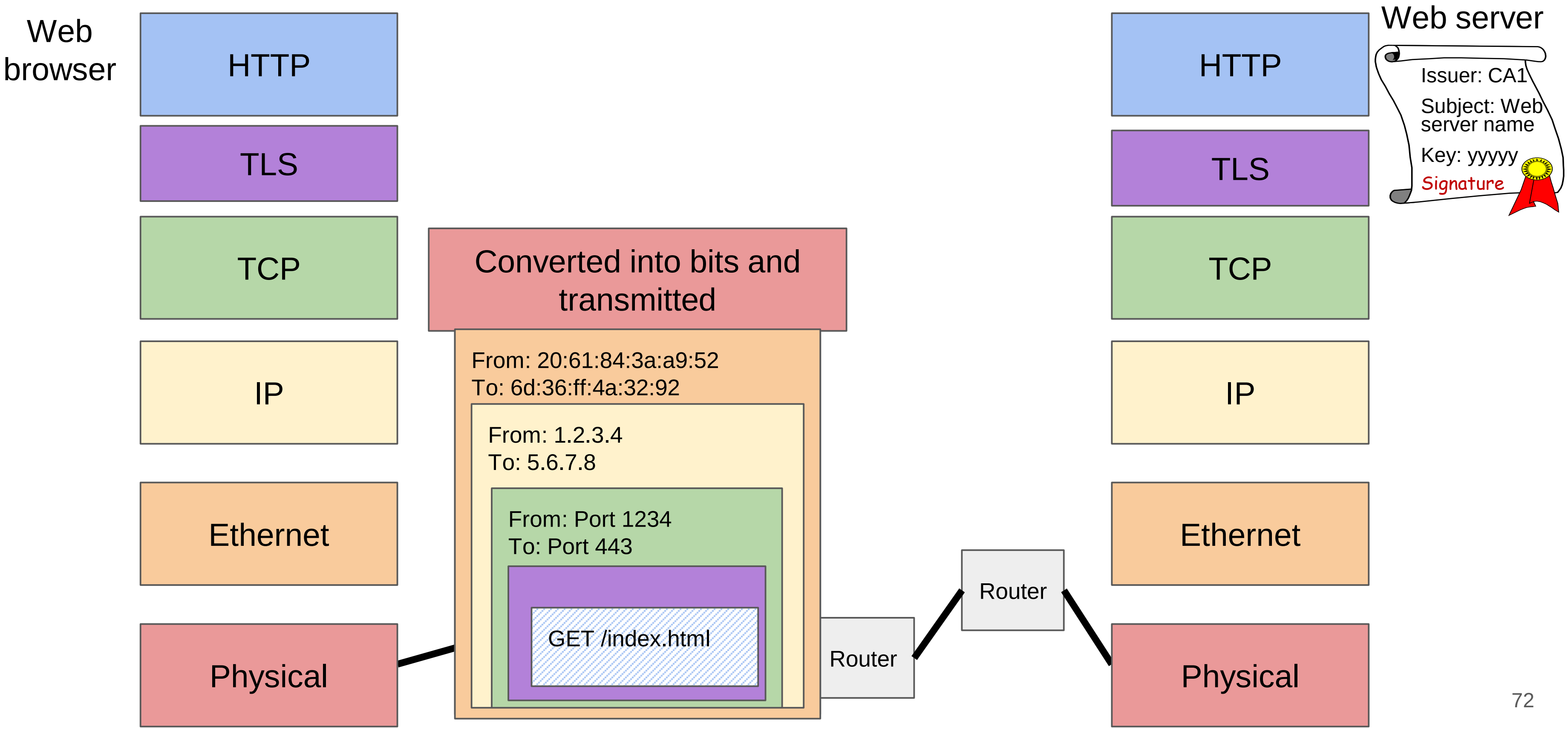
Data transfer with HTTP over TLS (HTTPS)



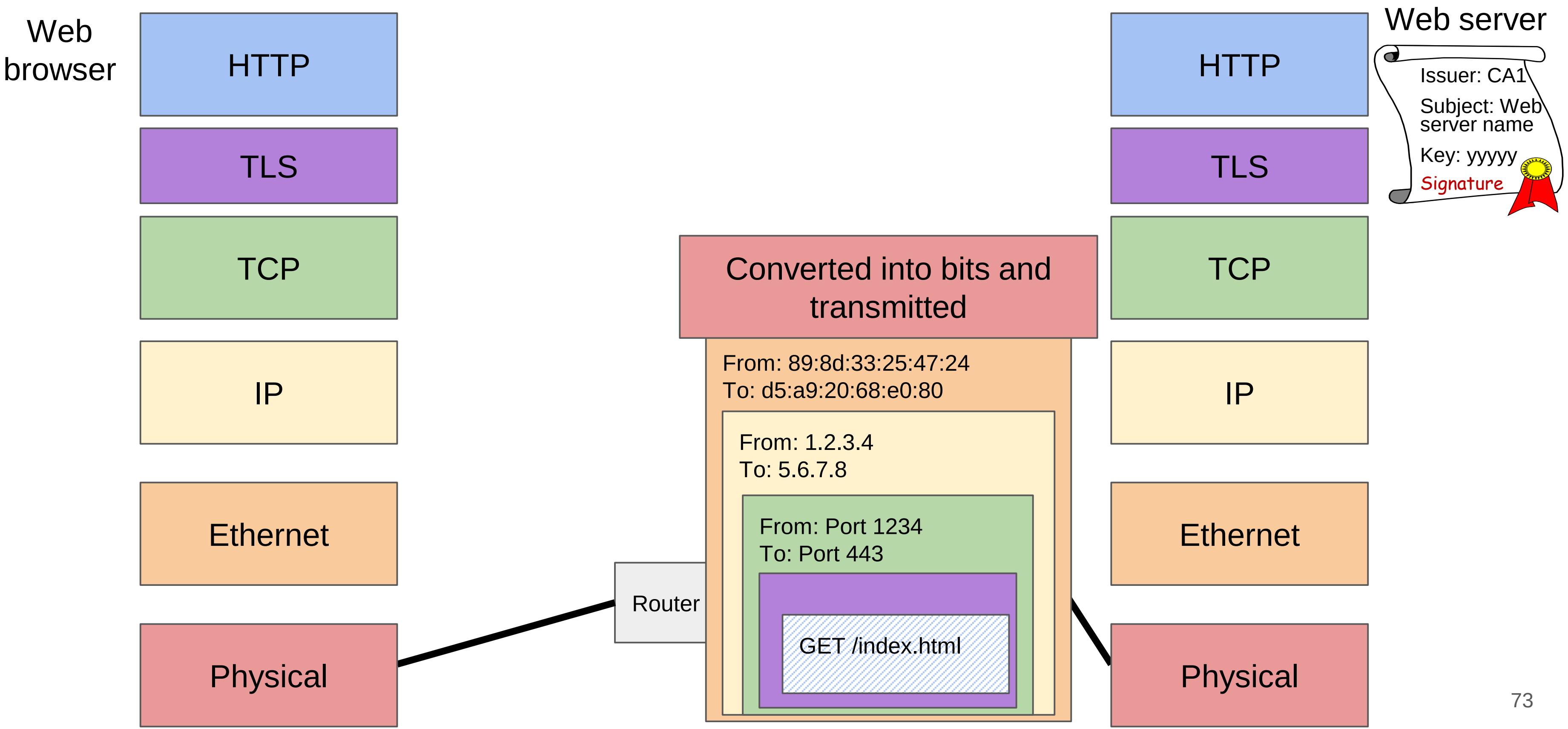
Data transfer with HTTP over TLS (HTTPS)



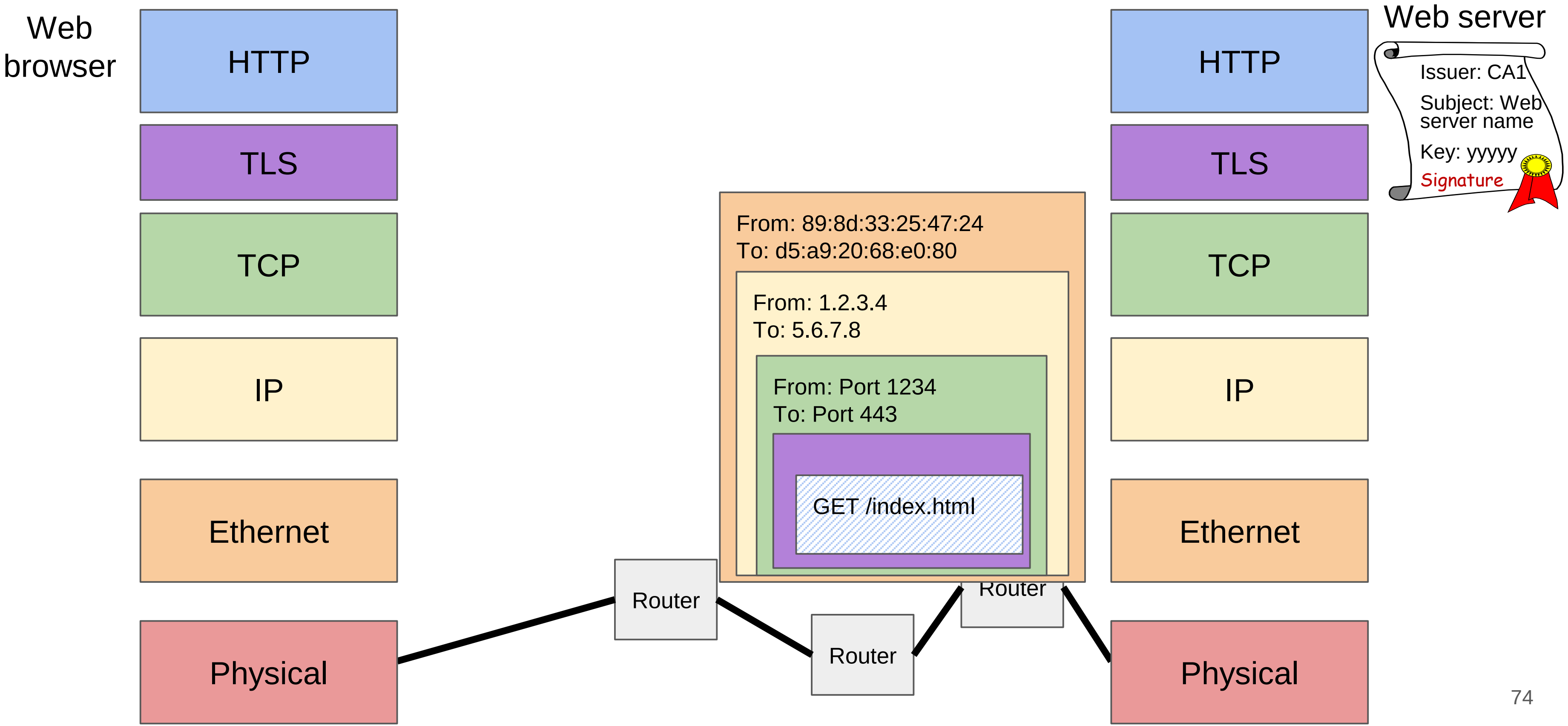
Data transfer with HTTP over TLS (HTTPS)



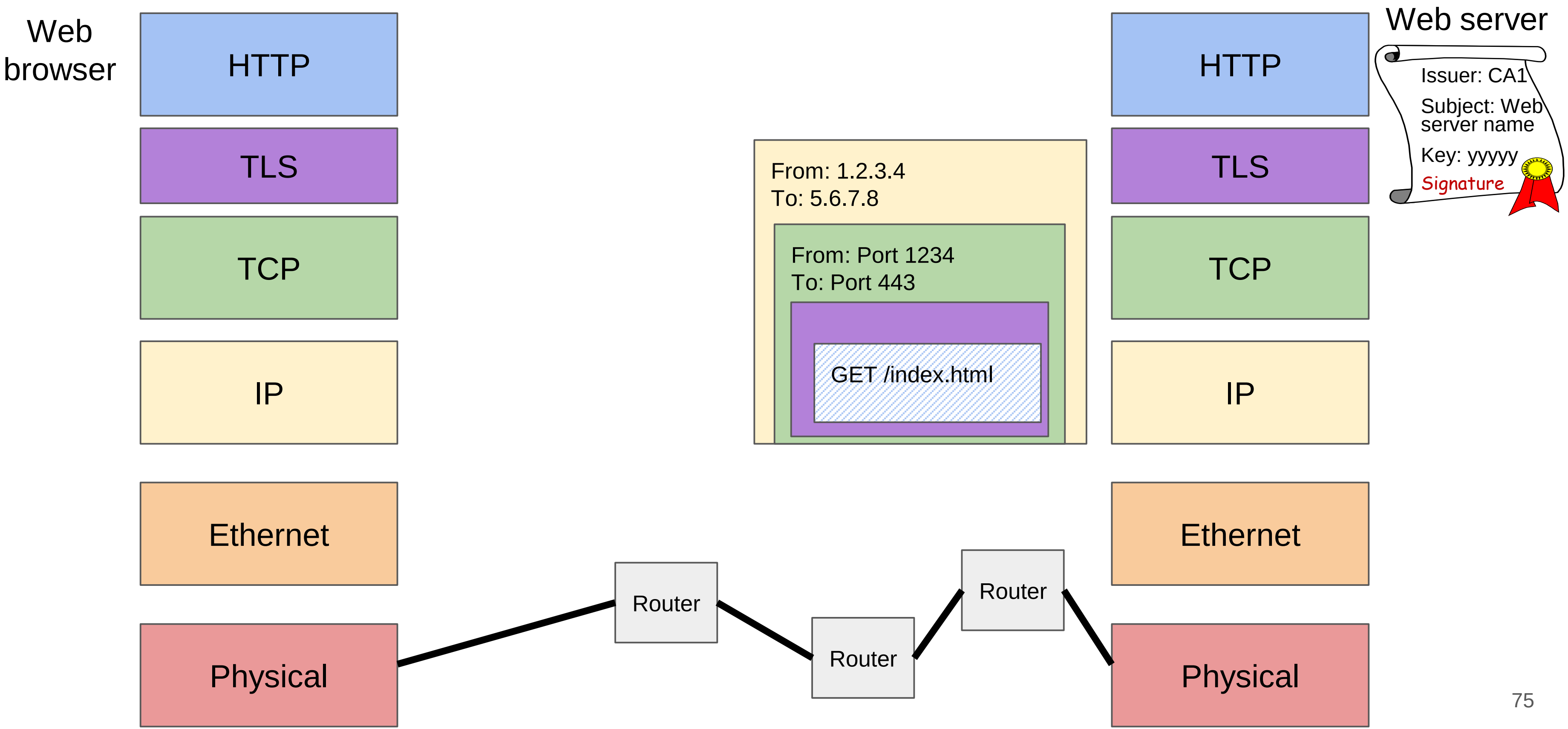
Data transfer with HTTP over TLS (HTTPS)



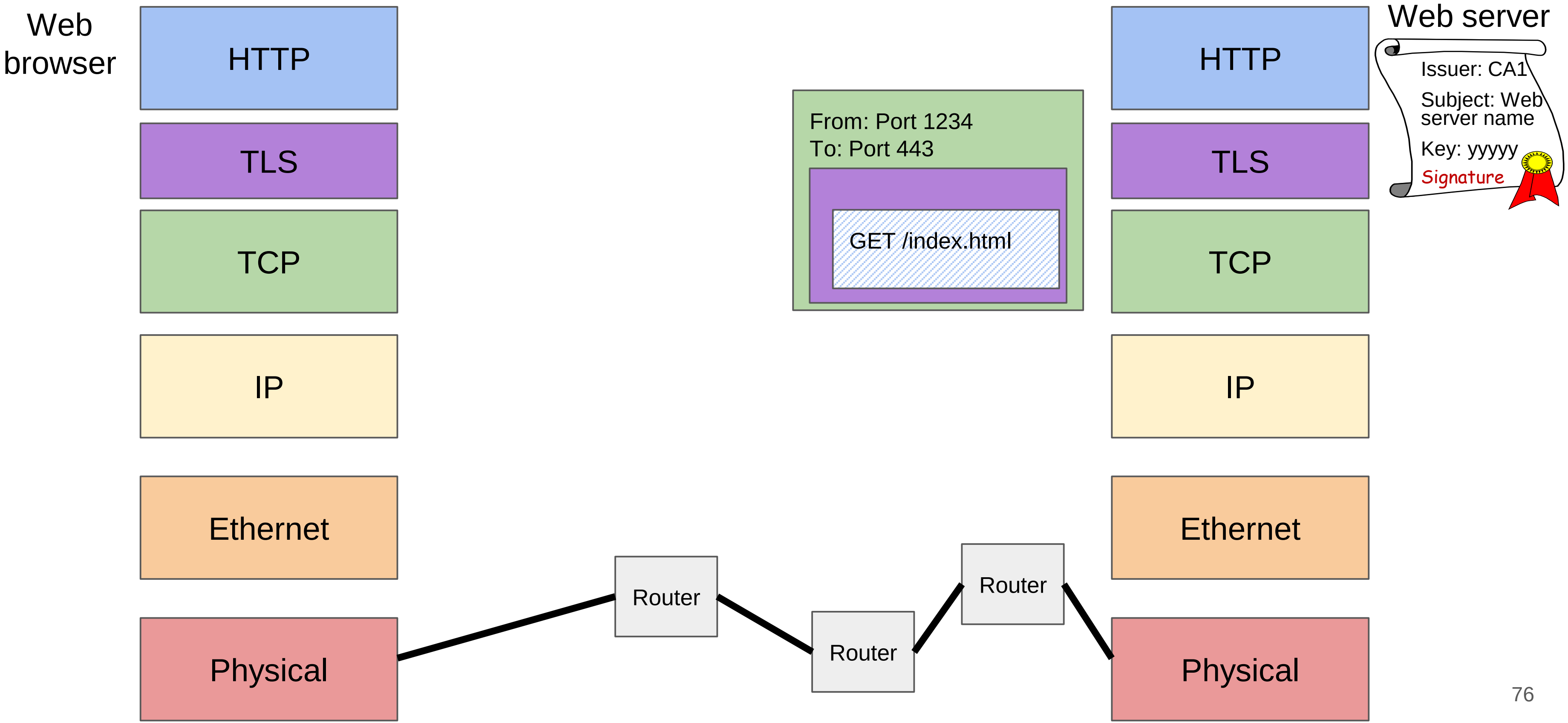
Data transfer with HTTP over TLS (HTTPS)



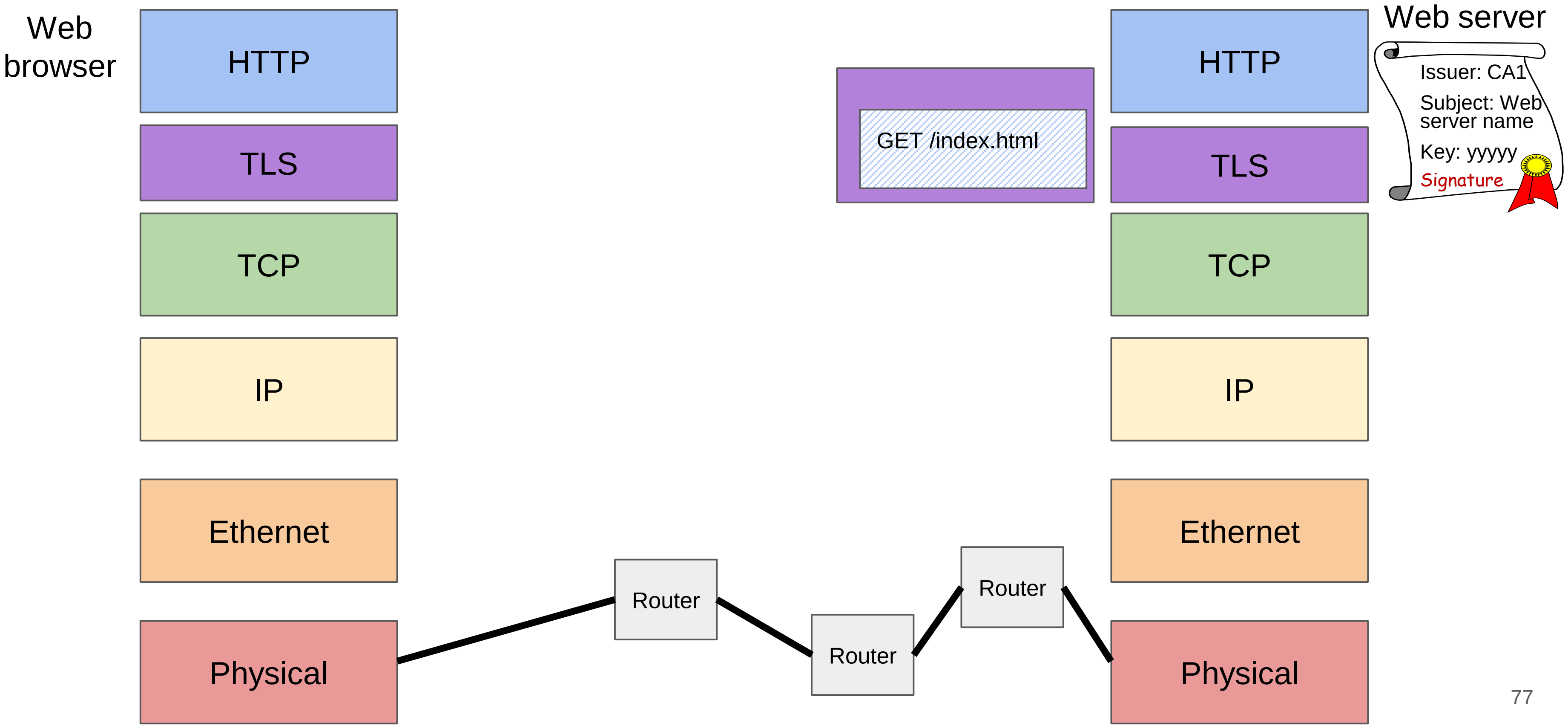
Data transfer with HTTP over TLS (HTTPS)



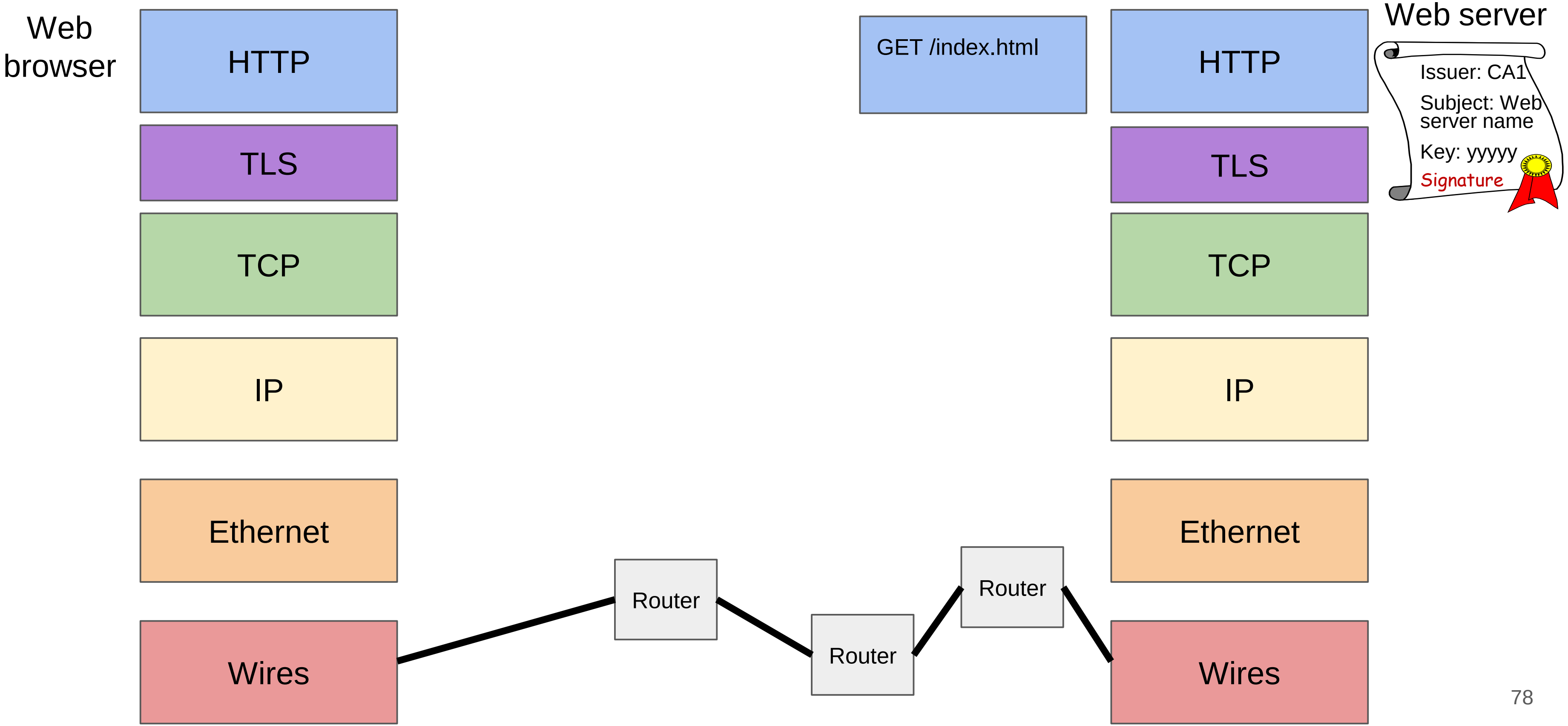
Data transfer with HTTP over TLS (HTTPS)



Data transfer with HTTP over TLS (HTTPS)



Data transfer with HTTP over TLS (HTTPS)



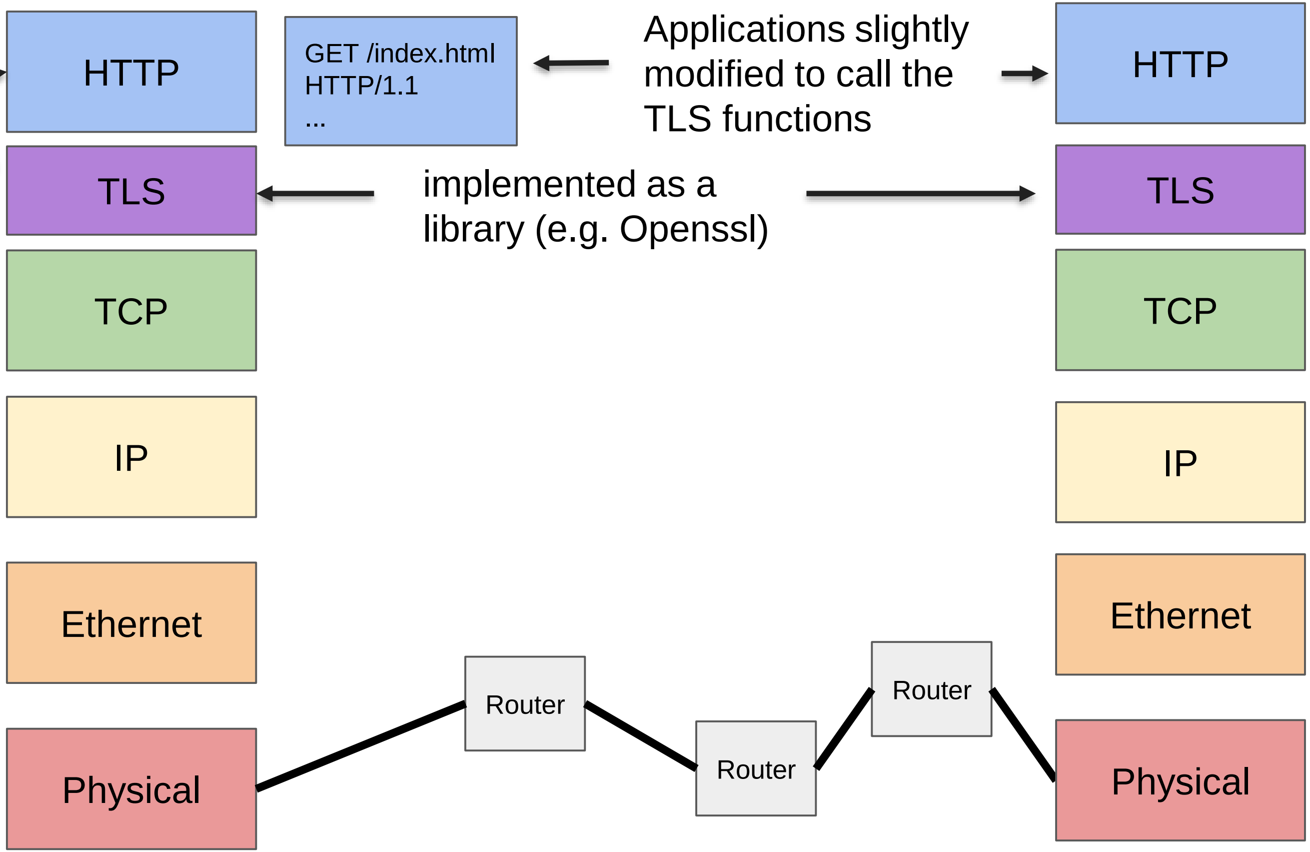
HTTP over TLS (HTTPS): Implementation Notes

Web browser

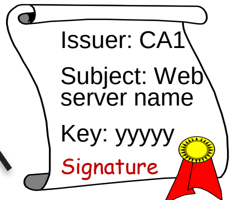
Lock icon shown in the web page

Trusted CA certificates pre-loaded in the browser

browser checks the domain name in the URL **matches** the domain name in the server certificate



Web server

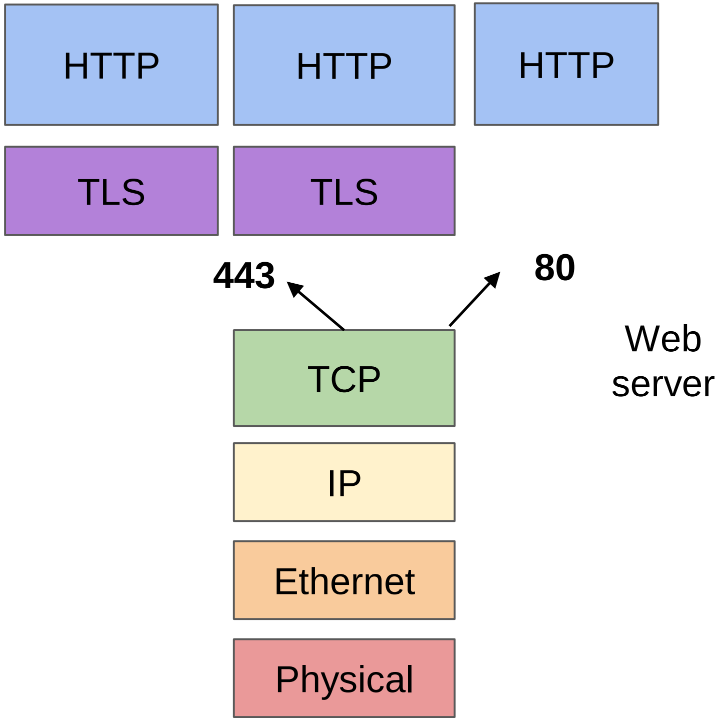


Server sends the whole certificate chain (except the trusted CA certificate !)

can use HSM to protect the private key and speed up operations

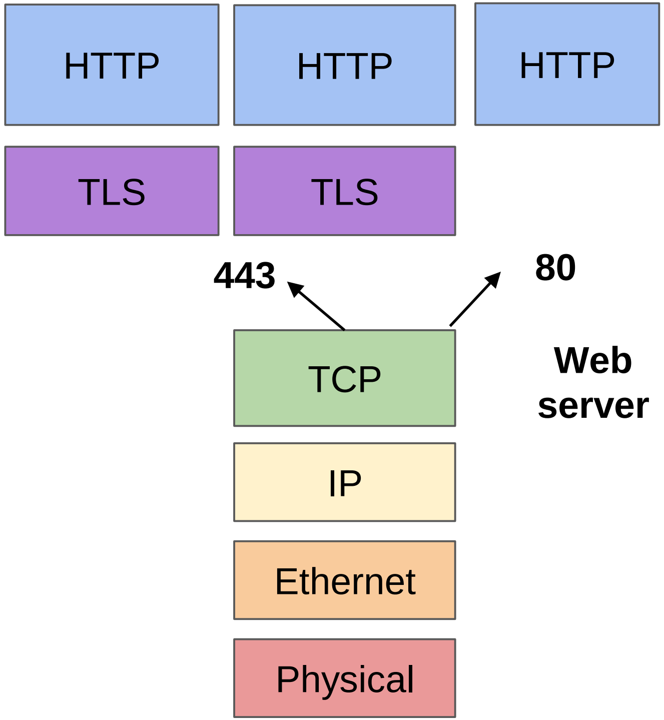
Web server notes

- **Web server uses different ports for HTTP and HTTPS**
  - 80 for HTTP, 443 for HTTPS
- **Web server may host different services = virtual server (frequent case with web hosting)**
  - different logical names associated to the same IP address
  - e.g. home.myweb.it=5.6.7.8, food.myweb.it=5.6.7.8



TLS and virtual server: the problem

- **how to indicate to which (virtual) server the client wants to connect to?**
- **easy in HTTP/1/1**
  - the client uses the Host header to specify the server it wants to connect to
- **… but difficult in HTTPS**
  - because TLS is activated before HTTP
  - which X.509v3 certificate should the server provide? (must contain the server's name)



TLS and virtual servers: solutions

- **collective (wildcard) server certificate**
  - e.g. CN=*.myweb.it
  - private key shared by all servers
  - different treatment by different browsers
- **server certificate with a list of (virtual) server names in subjectAltName extension**
  - private key shared by all servers
  - need to re-issue the certificate at any addition or cancellation of a (virtual) server
- **use the SNI (Server Name Indication) extension**
  - in ClientHello (permitted by RFC-4366)

When is TLS effective?

- **TLS provides end-to-end security: Secure communication between the two endpoints**
  - authentication (single or mutual), data authentication, integrity and confidentiality only during the transit inside the TLS communication channel
  - no non repudiation
- **No need to trust intermediaries (routers, communication lines,..)**
  - even if everybody between the client and the server is malicious, TLS provides a secure communication channel
    - example: a local network attacker tries to make MITM with ARP poisoning, but can’t read TLS messages and can’t modify/delete TLS messages because MAC won’t be correct at destination

When is TLS effective? (cont.)

    - example: a MITM in the router (or on backbone) tries to inject TCP packets, but packets will be rejected because MAC won’t be correct

- **using TLS defends against most lower-level network attacks**
- **BUT: NOTE(!) end-to-end security does not help if one of the endpoints is malicious, e.g.**
  - client communicating with a fake/shadow server presenting a `valid’ certificate, such as one issued by a compromised CA
  - or one of the parties (client/server) has been infected by malware