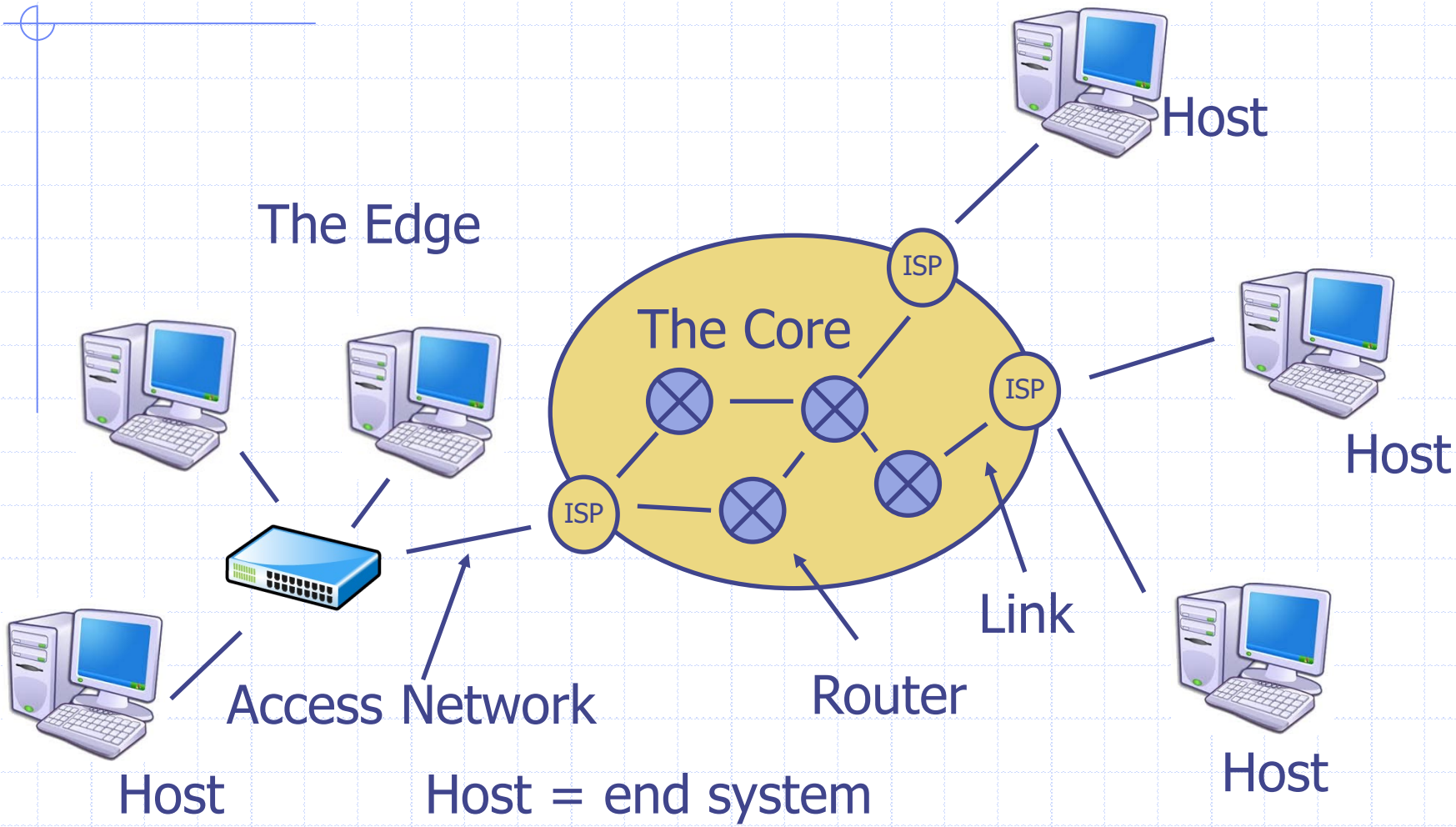
COS
Computersystemer

Lektion #8
Computer netværk, fortsat.

Morten Hansen
moh@sdu.dk

The Internet



Morten Hansen
moh@sdu.dk

IP- adresser

- 32 bit
- 4 billion possible adresses.
- xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
- ddd.ddd.ddd.ddd
- f.eks.:
  - 192.168.1.12

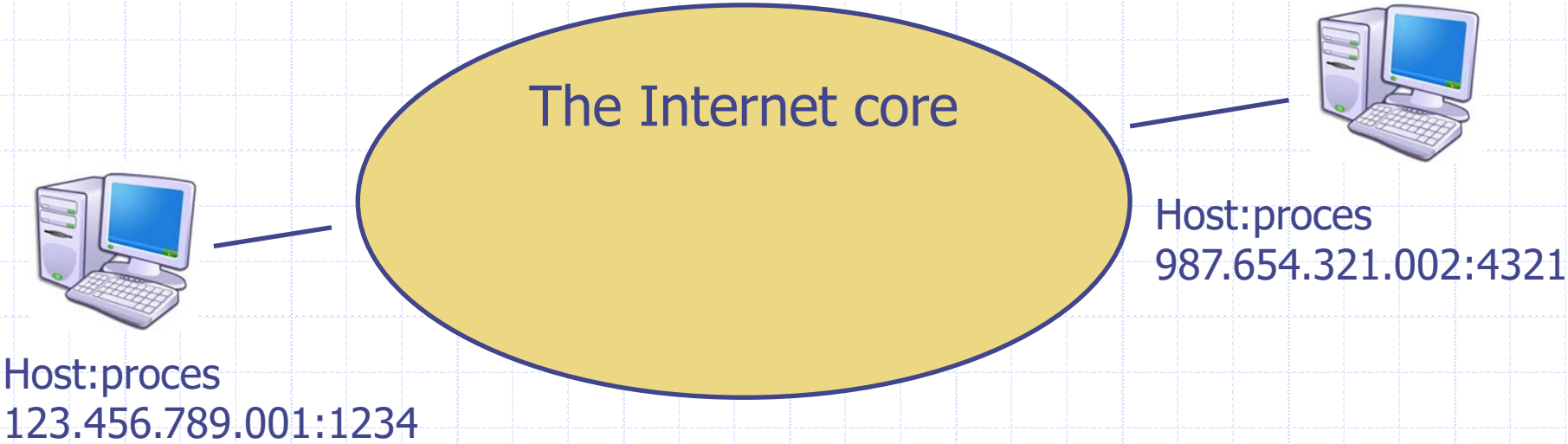
Morten Hansen
moh@sdu.dk

Internet Addressing

- IP address: pattern of 32 or 128 bits often represented in dotted decimal notation
- Mnemonic address:
  - Domain names
  - Top-Level Domains
- Domain name system (DNS)
  - Name servers
  - DNS lookup

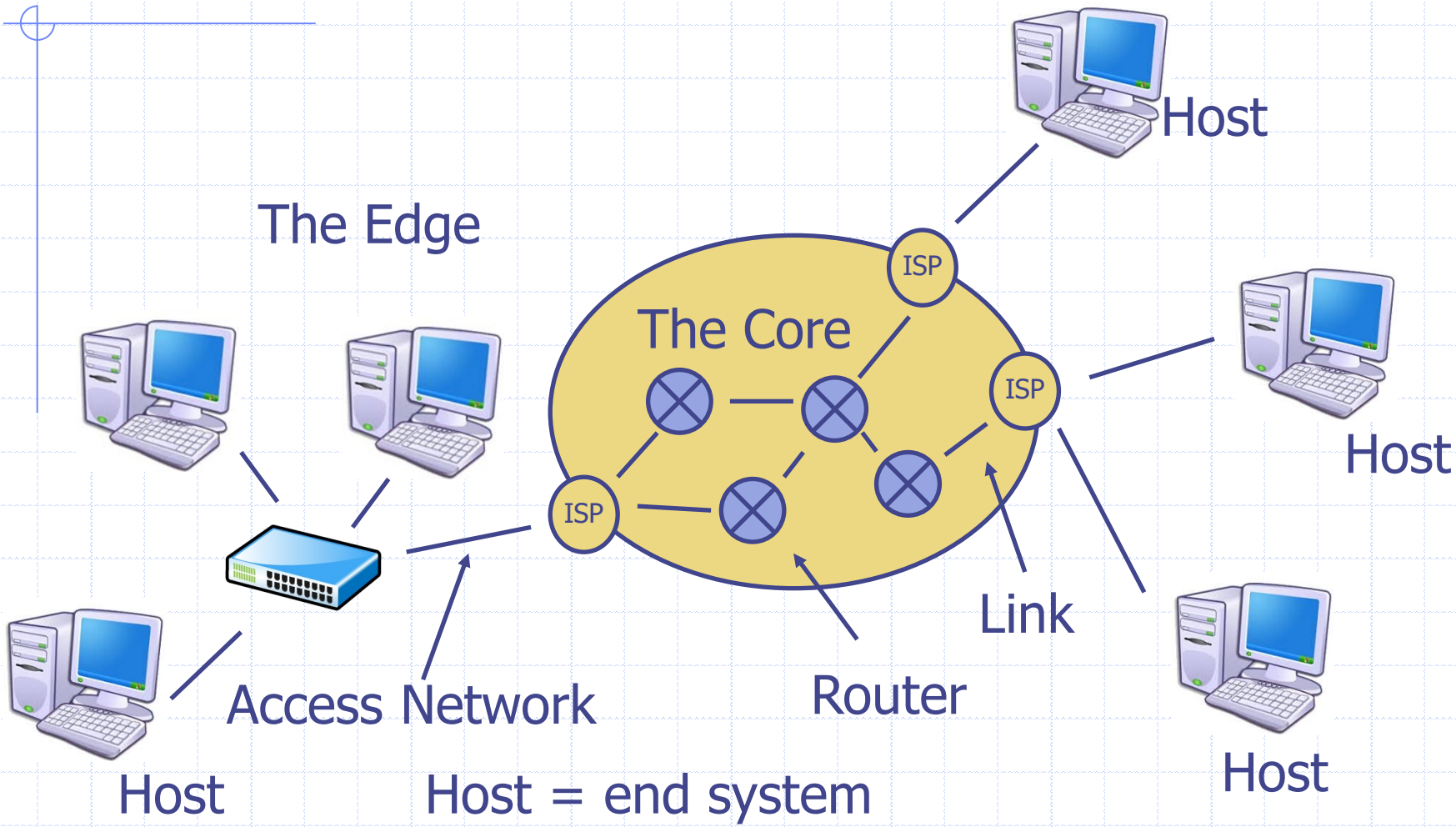
Morten Hansen
moh@sdu.dk

I virkeligheden vil vi bare kommunikere fra en proces til en anden proces. Nettet er transparent.



Morten Hansen
moh@sdu.dk

The Internet



Morten Hansen
moh@sdu.dk

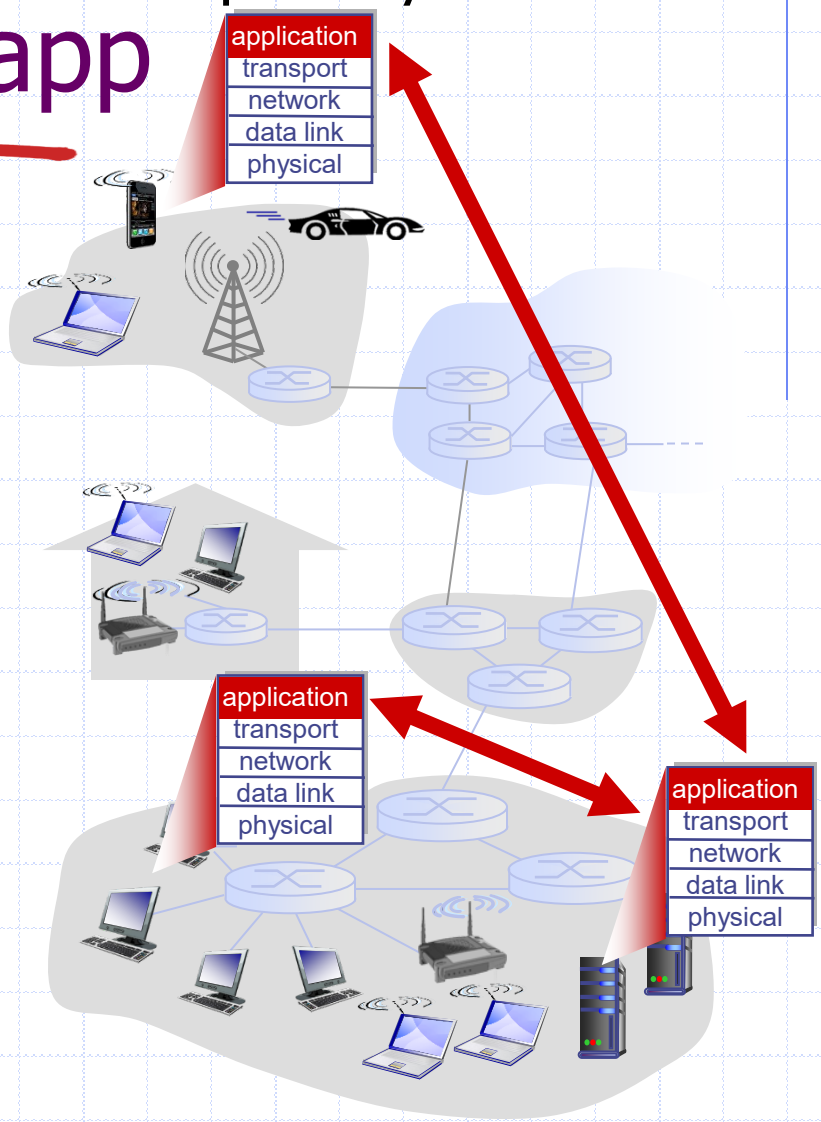
Creating a network app

write programs that:

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

no need to write software for network-core devices

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation

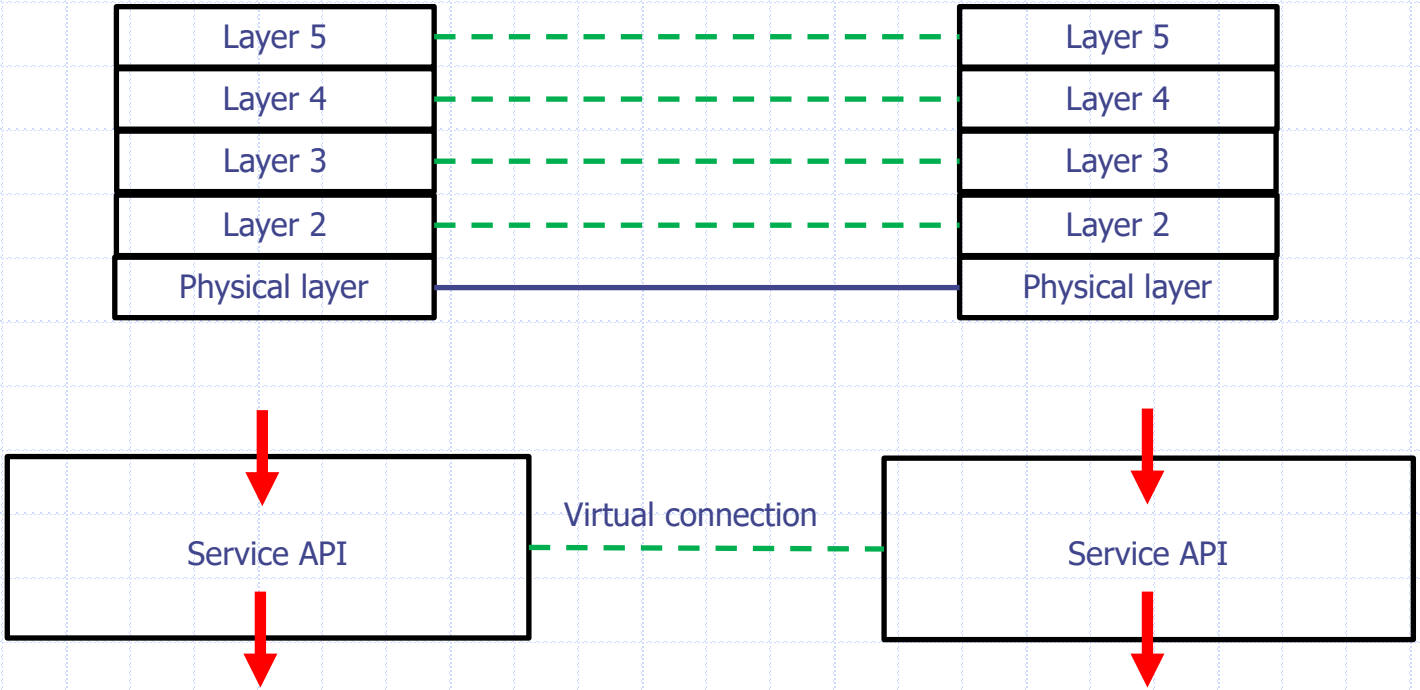


Morten Hansen
moh@sdu.dk

Lagdeling.

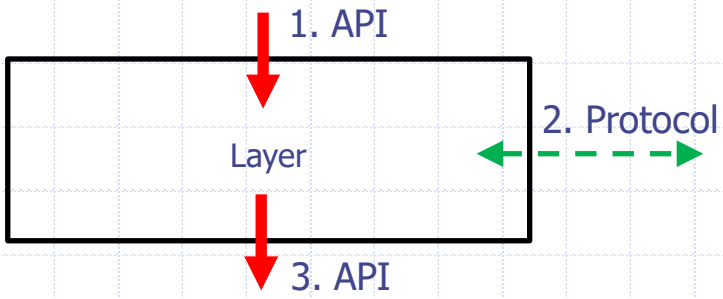
Morten Hansen
moh@sdu.dk

Layered protocol stack



Morten Hansen
moh@sdu.dk

Every layer must…

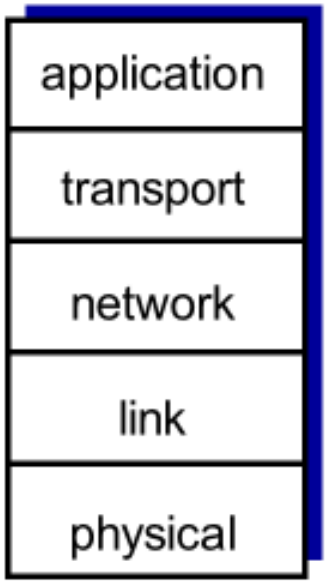


1. offer services to an upper layer.
2. comply with agreed protocols.
3. utilize services from the underlying layer.

Morten Hansen
moh@sdu.dk

Internet protocol stack

- *application:* supporting network applications
  - FTP, SMTP, HTTP
- *transport:* process-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- *physical:* bits "on the wire"



Introduction 1-60

Morten Hansen
moh@sdu.dk

Lagdelte netværksmodeller.

TCP/IP 5-lag	Bogens 4-lag	OSI 7-lag
Applikationslag	Applikationslag	Applikationslag
		Præsentationslag
		Sessionslag
Transportlag	Transportlag	Transportlag
Netværkslag	Netværkslag	Netværkslag
Linklag	Linklag	Linklag
Fysisk lag		Fysisk lag

Morten Hansen
moh@sdu.dk

TCP 5-lagsmodellen

TCP/IP 5-lag		
Applikationslag		Her kører applikationens processer
Transportlag		..er ansvarlig for at transmitere data fra proces til proces.
Netværkslag		..er ansvarlig for at transmittere data fra host til host.
Linklag		..er ansvarlig for at transmittere data via en link fra node til node.
Fysisk lag		Overfører de fysiske bit via et transmissionsmedie.

Morten Hansen
moh@sdu.dk

Protocol "layers"

Networks are complex, with many "pieces":

- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

Question:

is there any hope of *organizing* structure of network?

…. or at least our discussion of networks?

Morten Hansen
moh@sdu.dk

Why layering?

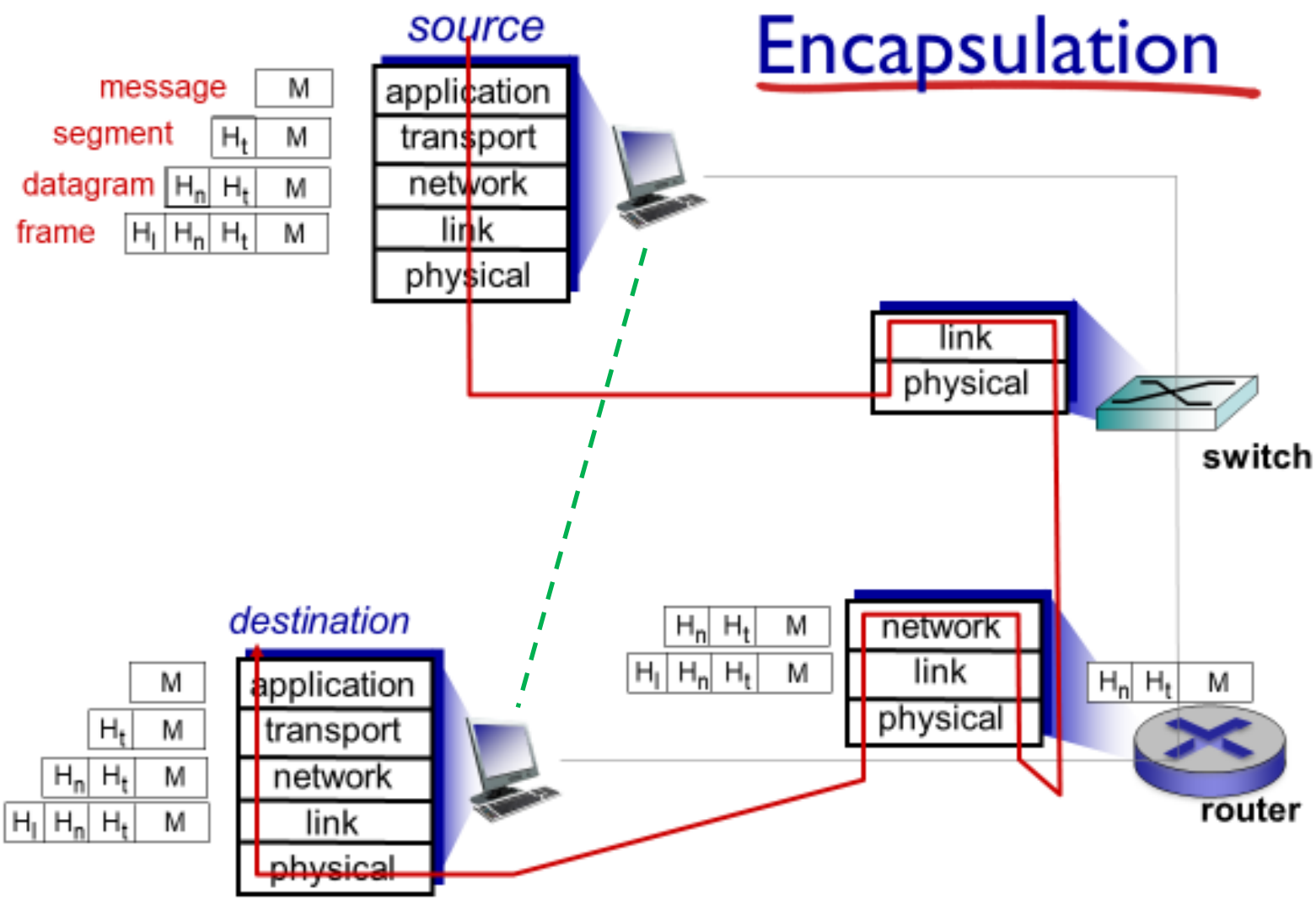
dealing with complex systems:

- explicit structure allows identification, relationship of complex system's pieces
  - layered *reference model* for discussion
- modularization eases maintenance, updating of system
  - change of implementation of layer's service transparent to rest of system
  - e.g., change in gate procedure doesn't affect rest of system
- layering considered harmful?

Introduction 1-59

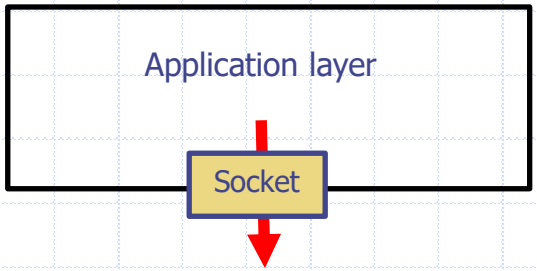
Morten Hansen
moh@sdu.dk

Encapsulation



ansen
mon@sdu.dk

The application layer...

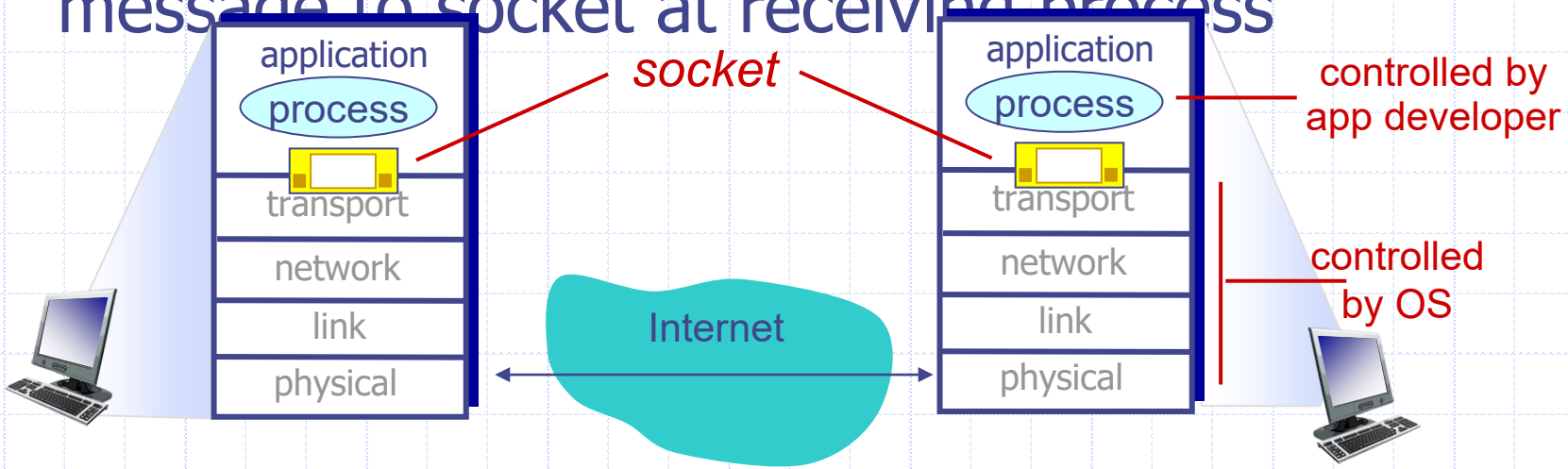


- has no upper layer.
- hosts an application.
- utilizes services from the transport layer through sockets.

Morten Hansen
moh@sdu.dk

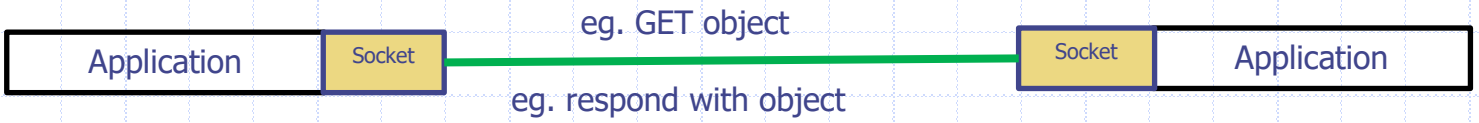
Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process

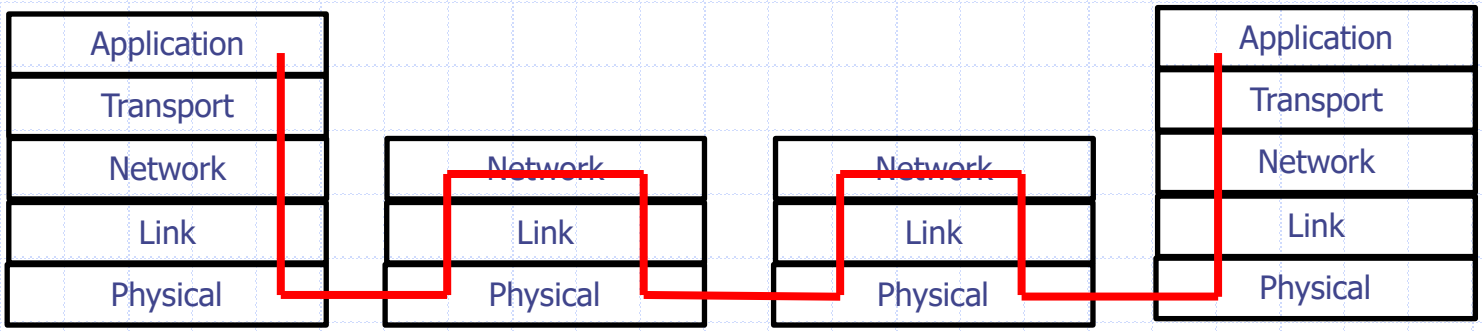


Morten Hansen
moh@sdu.dk

Application view



Real view



Morten Hansen
moh@sdu.dk

Some network apps

- e-mail
- web
- text messaging
- remote login
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Hulu, Netflix)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- search
- …
- …
- …
- Custom/proprietary applications.

Morten Hansen
moh@sdu.dk

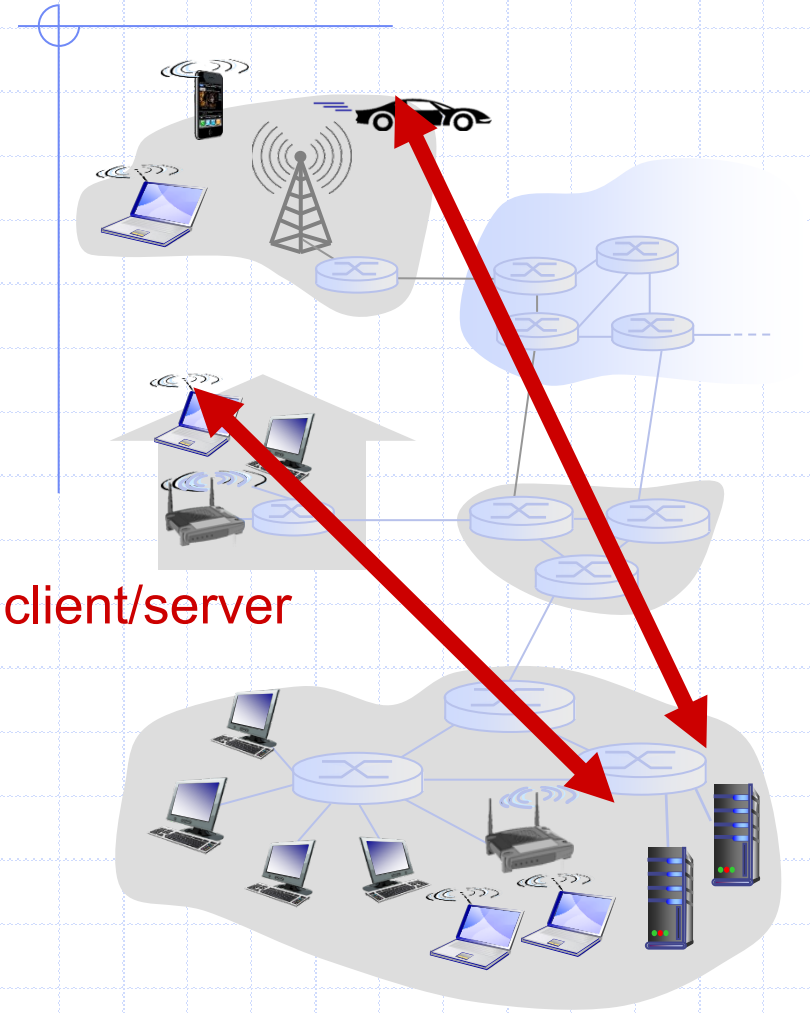
Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)
- Publish / subscribe

Morten Hansen
moh@sdu.dk

Client-server architecture



server:

- always-on host
- permanent IP address
- data centers for scaling

clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

Morten Hansen
moh@sdu.dk

Processes communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

clients, servers

client process: process that initiates communication

server process: process that waits to be contacted

Morten Hansen
moh@sdu.dk

Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *identifier* includes both IP address and port numbers associated with process on host.

- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

Morten Hansen
moh@sdu.dk

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

- open protocols:
  - defined in RFCs
  - allows for interoperability
  - e.g., HTTP, SMTP
- proprietary protocols:
  - e.g., Skype

Morten Hansen
moh@sdu.dk

What transport service does an app need?

data integrity

some apps (e.g., file transfer, web transactions) require 100% reliable data transfer

other apps (e.g., audio) can tolerate some loss

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

security

- encryption, data integrity, …

Morten Hansen
moh@sdu.dk

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup,

Morten Hansen
moh@sdu.dk

Web and HTTP

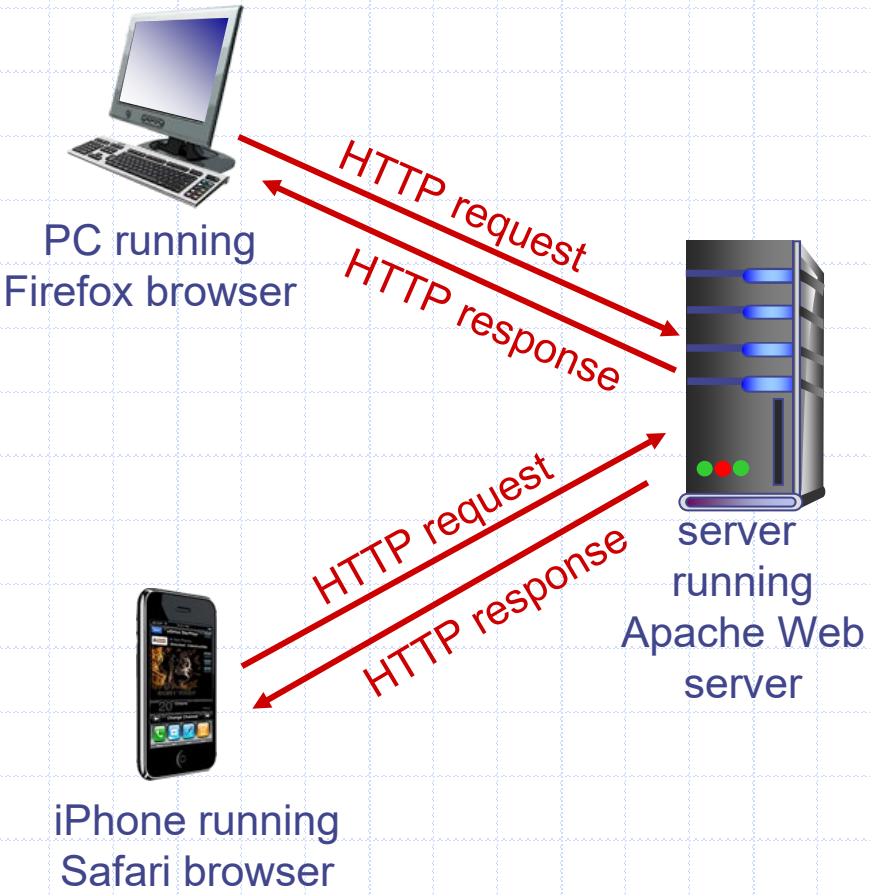
- *web page* consists of *objects*
- object can be HTML file, JPEG image, Java applet, audio file,…
- web page consists of *base HTML-file* which includes *several referenced objects*
- each object is addressable by a *URL,* e.g.,

$$\underbrace{\texttt{www.someschool.edu}}_{\text{host name}}\underbrace{\texttt{/someDept/pic.gif}}_{\text{path name}}$$

HTTP overview

HTTP:
hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client:* browser that requests, receives, (using HTTP protocol) and "displays" Web objects
  - *server:* Web server sends (using HTTP protocol) objects in response to requests



Morten Hansen
moh@sdu.dk

HTTP overview (continued)

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

Morten Hansen
moh@sdu.dk

Non-persistent HTTP

suppose user enters URL:

`www.someSchool.edu/someDepartment/home.index`

(contains text, references to 10 jpeg images)

1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on port 80

1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. "accepts" connection, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object someDepartment/home.index

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

time

Morten Hansen
moh@sdu.dk

HTTP sekvens (cont.)

4. HTTP server closes TCP connection.

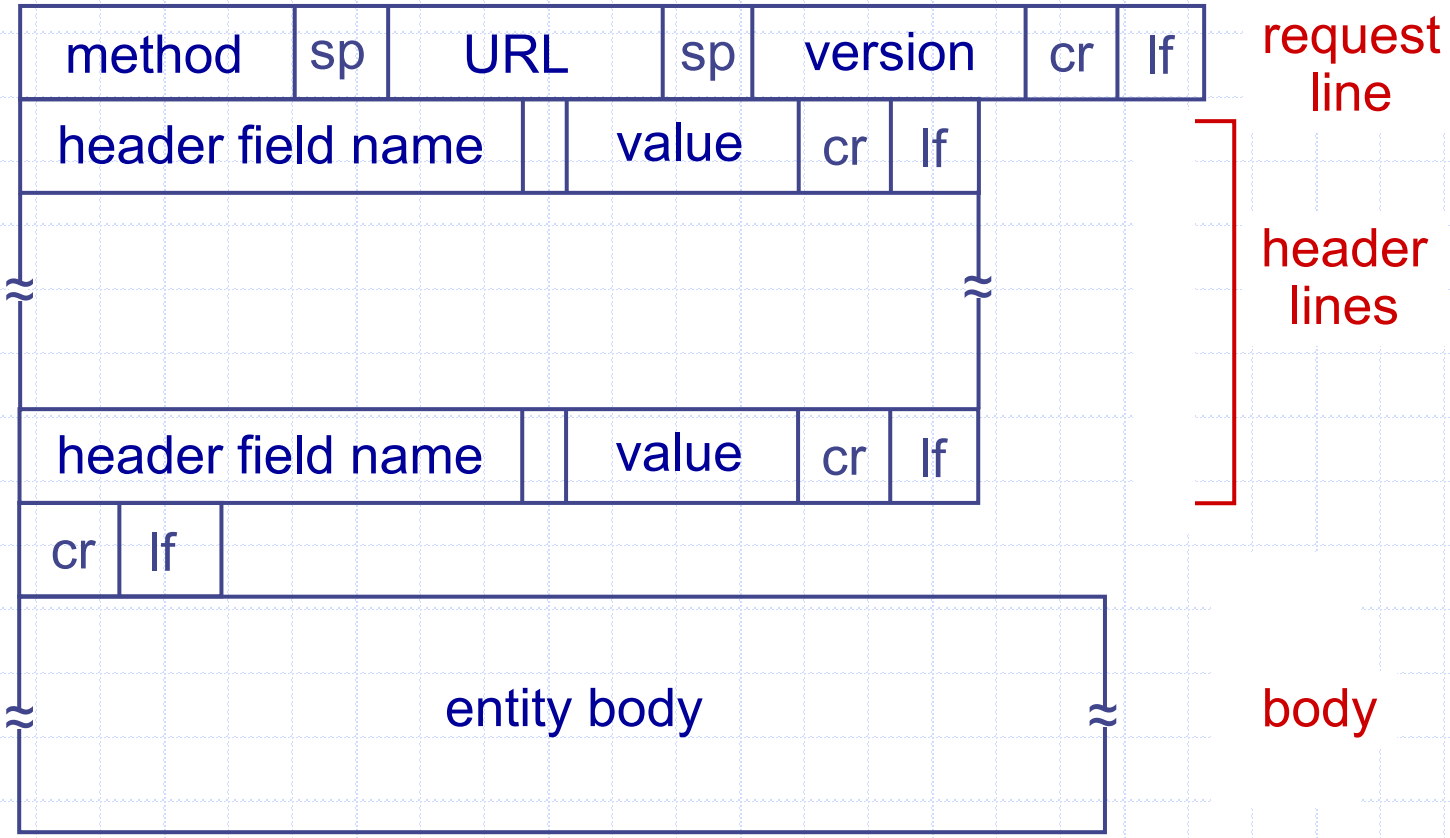
5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

time

6. Steps 1-5 repeated for each of 10 jpeg objects

Morten Hansen
moh@sdu.dk

HTTP request message: general format



Morten Hansen
moh@sdu.dk

HTTP request message

- two types of HTTP messages: *request*, *response*
- HTTP request message:
  - ASCII (human-readable format)

carriage return character

line-feed character

request line (GET, POST, HEAD commands)

header lines

carriage return, line feed at start of line indicates end of header lines

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```


Morten Hansen
moh@sdu.dk

ASCII codes.

´\n´ → 10
´\r´ → 13

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
0	0	NUL	16	10	DLE	32	20	(space)	48	30	0
1	1	SOH	17	11	DC1	33	21	!	49	31	1
2	2	STX	18	12	DC2	34	22	"	50	32	2
3	3	ETX	19	13	DC3	35	23	#	51	33	3
4	4	EOT	20	14	DC4	36	24	$	52	34	4
5	5	ENQ	21	15	NAK	37	25	%	53	35	5
6	6	ACK	22	16	SYN	38	26	&	54	36	6
7	7	BEL	23	17	ETB	39	27	'	55	37	7
8	8	BS	24	18	CAN	40	28	(	56	38	8
9	9	TAB	25	19	EM	41	29	)	57	39	9
10	A	LF	26	1A	SUB	42	2A	*	58	3A	:
11	B	VT	27	1B	ESC	43	2B	+	59	3B	;
12	C	FF	28	1C	FS	44	2C	,	60	3C	<
13	D	CR	29	1D	GS	45	2D	-	61	3D	=
14	E	SO	30	1E	RS	46	2E	.	62	3E	>
15	F	SI	31	1F	US	47	2F	/	63	3F	?

ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol	ASCII	Hex	Symbol
64	40	@	80	50	P	96	60	`	112	70	p
65	41	A	81	51	Q	97	61	a	113	71	q
66	42	B	82	52	R	98	62	b	114	72	r
67	43	C	83	53	S	99	63	c	115	73	s
68	44	D	84	54	T	100	64	d	116	74	t
69	45	E	85	55	U	101	65	e	117	75	u
70	46	F	86	56	V	102	66	f	118	76	v
71	47	G	87	57	W	103	67	g	119	77	w
72	48	H	88	58	X	104	68	h	120	78	x
73	49	I	89	59	Y	105	69	i	121	79	y
74	4A	J	90	5A	Z	106	6A	j	122	7A	z
75	4B	K	91	5B	[	107	6B	k	123	7B	{
76	4C	L	92	5C	\	108	6C	l	124	7C	|
77	4D	M	93	5D	]	109	6D	m	125	7D	}
78	4E	N	94	5E	^	110	6E	n	126	7E	~
79	4F	O	95	5F	_	111	6F	o	127	7F	

Method types

HTTP/1.0:

- GET
- POST
- HEAD
  - asks server to leave requested object out of response

HTTP/1.1:

- GET, POST, HEAD
- PUT
  - uploads file in entity body to path specified in URL field
- DELETE
  - deletes file specified in the URL field

Morten Hansen
moh@sdu.dk

HTTP response message

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
data data data data data ...
```

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

HTTP response status codes

- status code appears in 1st line in server-to-client response message.
  - some sample codes:

200 OK

- request succeeded, requested object later in this msg

301 Moved Permanently

- requested object moved, new location specified later in this msg (Location:)

400 Bad Request

- request msg not understood by server

404 Not Found

- requested document not found on this server

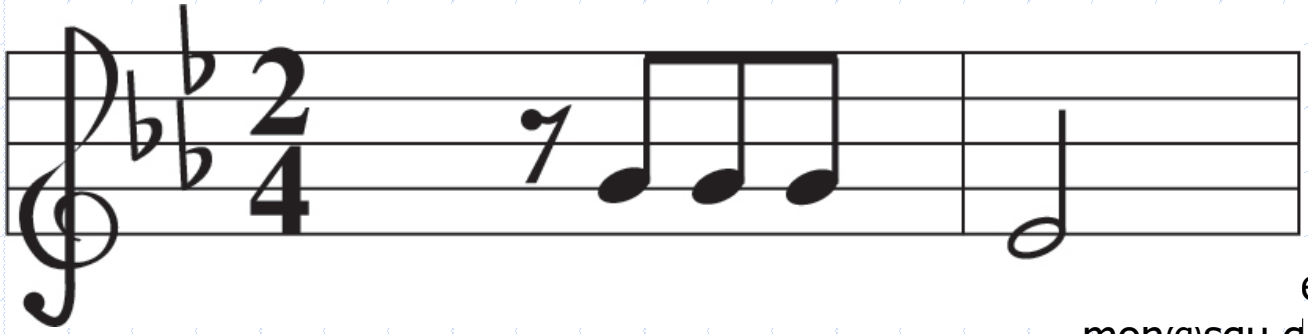
505 HTTP Version Not Supported

Morten Hansen
moh@sdu.dk

Extensible Markup Language (XML)

- XML: A language for constructing markup languages
  - A descendant of the Standard Generalized Markup Language
  - Opens door to a Worldwide Semantic Web (En af mange måder).
- Example:

```
<staff clef = "treble">
     <key>C minor</key>
     <time> 2/4 </time>
     <measure>
        < rest> egth </rest>
        <notes> egth G, egth G, egth G  </notes>
     </measure>
     <measure>
        <notes> hlf E </notes>
     </measure>
</staff>
```



Hypertext Markup Language (HTML)

- Encoded as text file
- Contains tags to communicate with browser
  - Appearance
    - <h1> to start a level one heading
    - <p> to start a new paragraph
  - Links to other documents and content
    - <a href = . . . >
  - Insert images
    - <img src = . . . >

Morten Hansen
moh@sdu.dk

A simple webpage

a. The page encoded using HTML.

Tag indicating beginning of document	`<html>`

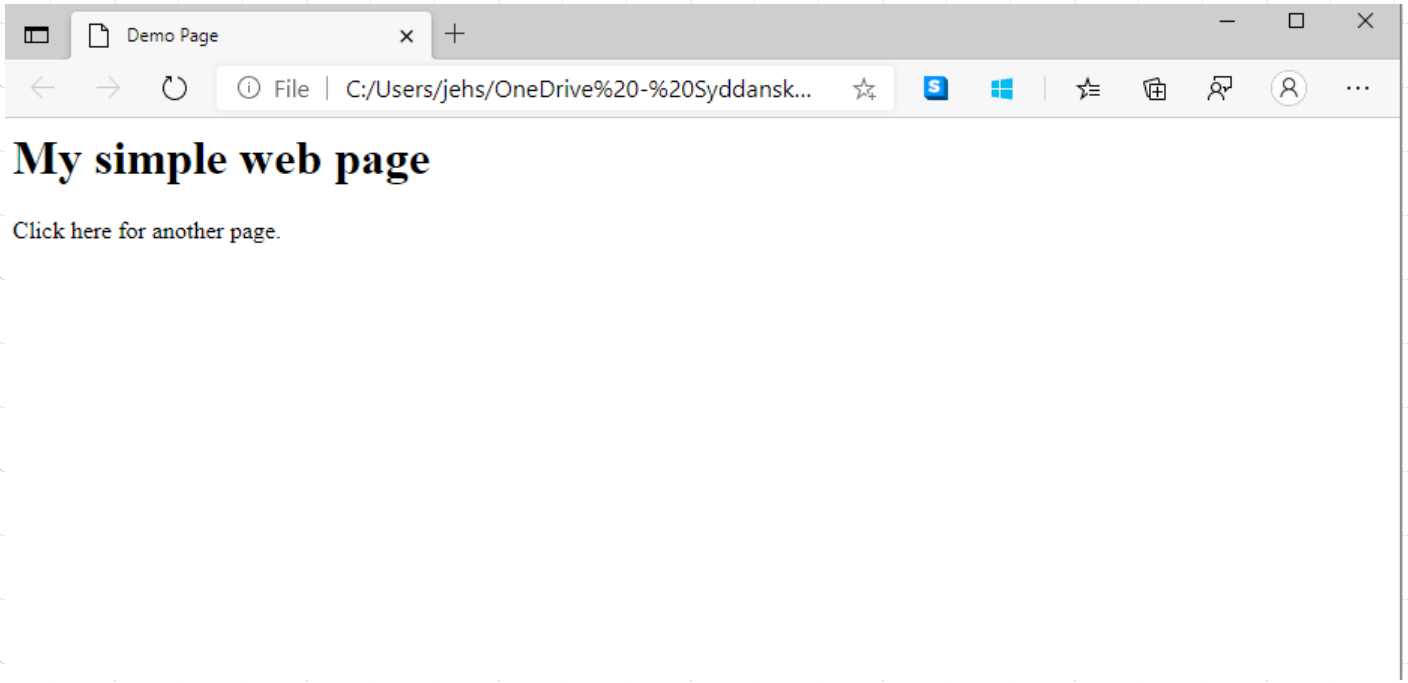
Preliminaries	`<head>`
	`<title>demonstration page</title>`
	`</head>`

The part of the document that will be displayed by a browser	`<body>`
	`<h1>My Web Page</h1>`
	`<p>Click here for another page.</p>`
	`</body>`

Tag indicating end of document	`</html>`

Morten Hansen
moh@sdu.dk

A simple webpage



Morten Hansen
moh@sdu.dk

An enhanced simple webpage – hyperlink

a. The page encoded using HTML.

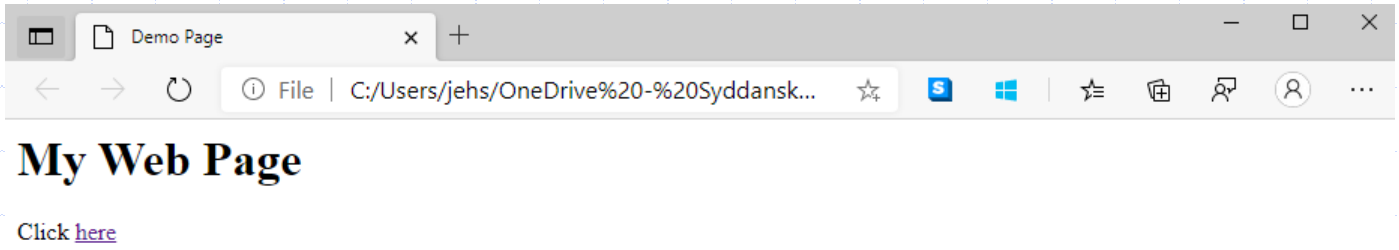
```
<html>
<head>
<title>demonstration page</title>
</head>
<body>
<h1>My Web Page</h1>
<p>Click
    <a href="http://crafty.com/demo.html">
    here
    </a>
    for another page.</p>
</body>
</html>
```

Anchor tag containing parameter

Closing anchor tag

Morten Hansen
moh@sdu.dk

An enhanced simple webpage – hyperlink



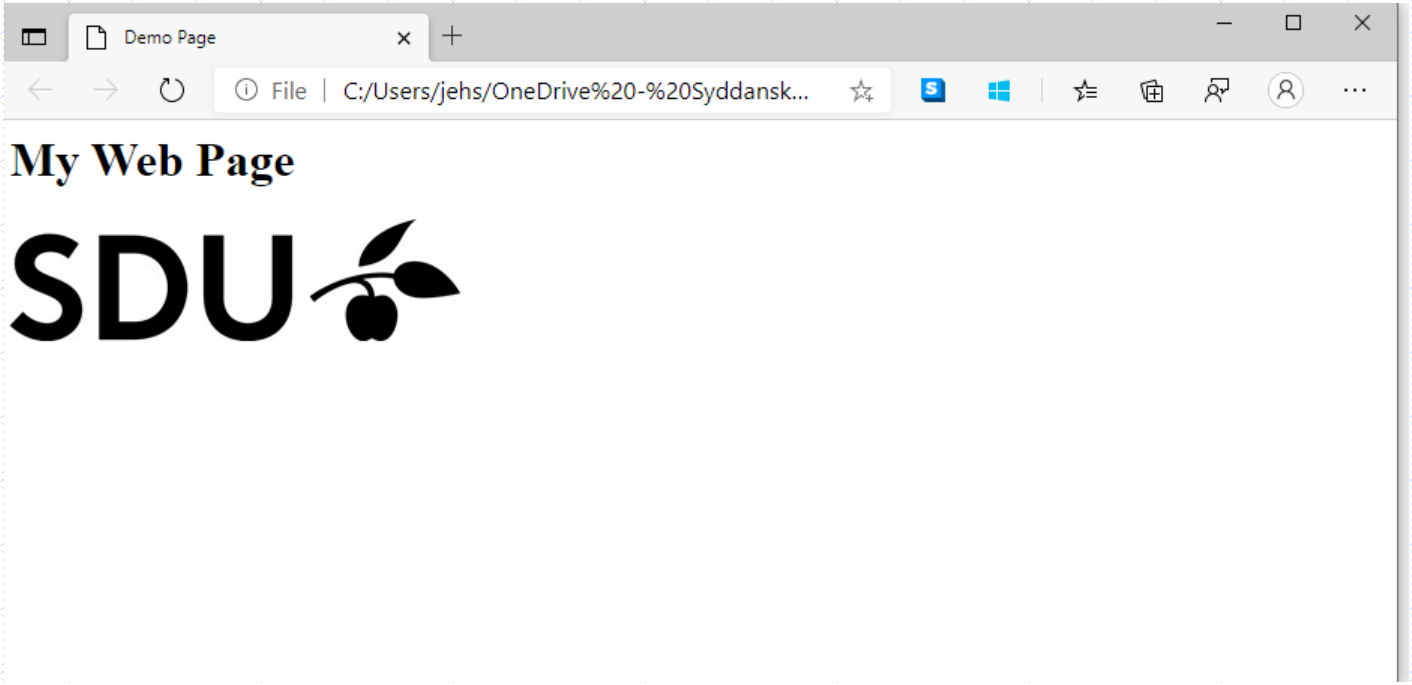
Morten Hansen
moh@sdu.dk

An enhanced simple webpage - Image

```
<html>
<head>
<title>Demo page</title>
</head>
<body>
<h1>My Web Page</h1>
<img src="SDU_BLACK_RGB.jpg"/>
</body>
</html>
```


Morten Hansen
moh@sdu.dk

An enhanced simple webpage – Image



Morten Hansen
moh@sdu.dk

Sockets

Morten Hansen
moh@sdu.dk

Client Server Program using sockets

- Socket: an abstraction for processes at the application layer to connect to the network via the transport layer
  - Needs to know
    - UDP or TCP
    - Source Address name (localhost)
    - Source Port number(1023…65535)
    - Destination Address
    - Destination Port number

Morten Hansen
moh@sdu.dk

TCP Server Script in Python

```
from socket import *
serverPort = 12001
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
#Queue up as many as 1 connect requests before refusing outside
connections.
serverSocket.listen(1)
print ('The server is ready to receive')
while 1:
        #Wait for clients connecting
        connectionSocket, addr = serverSocket.accept()
        print('connection ' + str(addr))
        #Read message from client
        sentence = connectionSocket.recv(1024)
        print(sentence.decode())
        capitalizedSentence = sentence.upper()
        connectionSocket.send(capitalizedSentence) #Reply client
        #close connection
        connectionSocket.close()
```


Morten Hansen
moh@sdu.dk

TCP Client Script in Python

```
from socket import *
serverPort = 12001
serverName = 'localhost'
#Create a socket using TCP (SOCK_STREAM), AF_INET = IPv4
clientSocket = socket(AF_INET,SOCK_STREAM)

#Connect to server
clientSocket.connect((serverName, serverPort))

message = input('Input lowercase sentence:')
#Send messages to server
clientSocket.sendto(bytes(message,"utf-8"),(serverName, serverPort))

#Read message from server
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print (modifiedMessage.decode())
#close connection
clientSocket.close()
```


Morten Hansen
moh@sdu.dk

UDP Server Script in Python

```
from socket import *
serverPort = 12000
#Create a socket using UDP (SOCK_DGRAM)
serverSocket = socket(AF_INET, SOCK_DGRAM)

#bind the socket to localhost:12000
serverSocket.bind(('localhost', serverPort))
print ('The server is ready to receive’)

while 1:
        #Wait for messages
        message, clientAddress = serverSocket.recvfrom(2048)

        print('Received: ' + str(message.decode()))
        modifiedMessage = message.upper()
        #Send messages back to sender
        serverSocket.sendto(modifiedMessage, clientAddress)
```


Morten Hansen
moh@sdu.dk

UDP Client Script in Python

```
from socket import *
serverName = 'localhost'
serverPort = 12000
#Create a socket using UDP (SOCK_DGRAM), AF_INET = IPv4
clientSocket = socket(AF_INET, SOCK_DGRAM) #UDP

message = input('Input lowercase sentence:')
#Send messages to sender server
clientSocket.sendto(bytes(message,"utf-8"),(serverName,
serverPort))
#wait for server to reply
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print (modifiedMessage.decode())

clientSocket.close()
```


Morten Hansen
moh@sdu.dk

Spørgsmål?

Morten Hansen
moh@sdu.dk