Ministry of high Education and Scientific Research
Southern Technical University
Technological institute of Basra
Dep. Of Computer network and software technologies

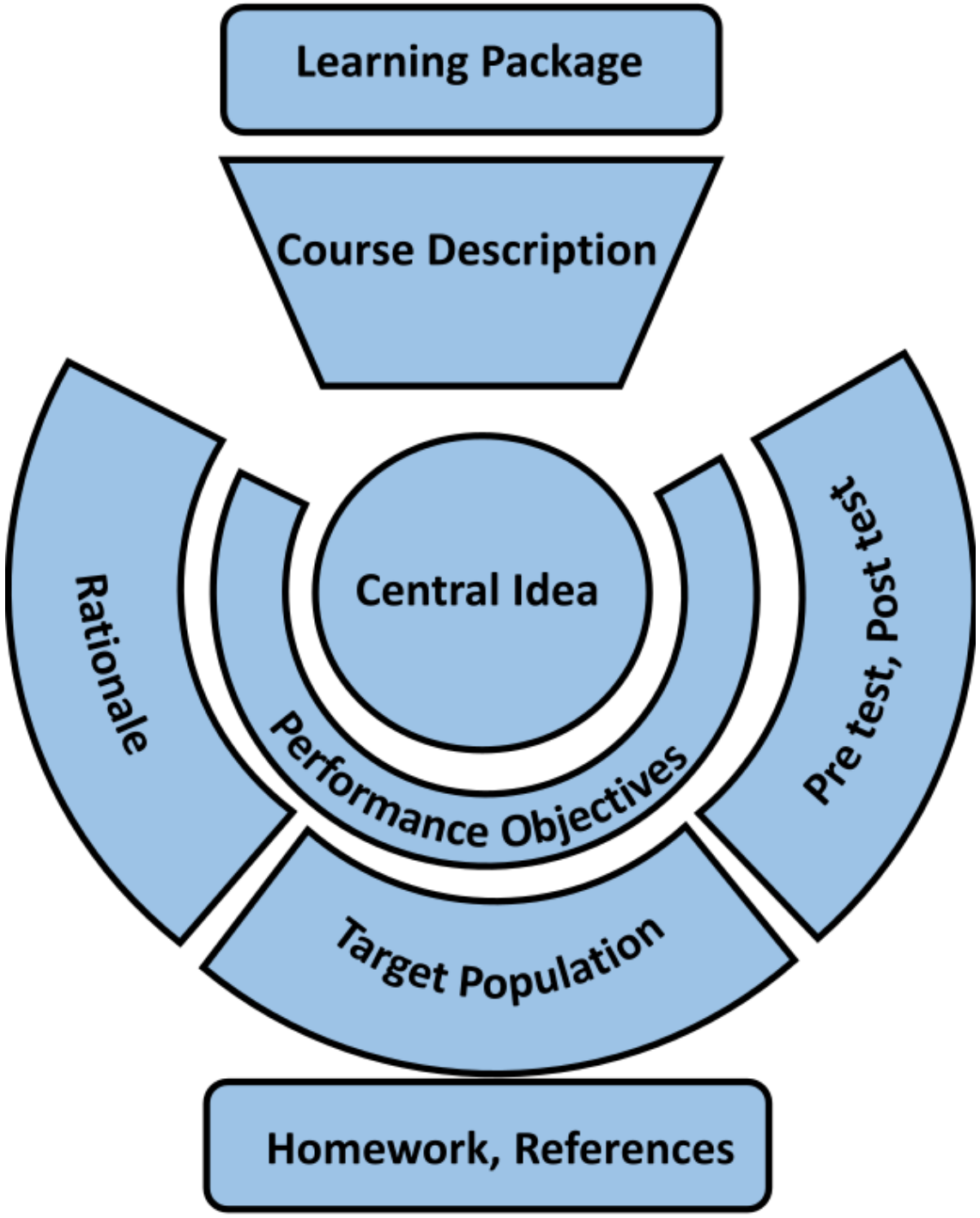
Learning package

Network Programming ()

For

Second year students

By
Dr. Hassan Almazini
Lecturer
Dep. Of Computer network and software technologies
2025



Course Description

Course Name:
Network Programming
Course Code:
Semester / Year:
Semester
Description Preparation Date:
24/ 06/ 2025
Available Attendance Forms:
Attendance only
Number of Credit Hours (Total) / Number of Units (Total)
60 hours/4 hour weekly/9 unit
Course administrator's name (mention all, if more than one name)
Name: Dr. Hassan Almazini Email: hassan.f.abbas@stu.edu.iq
Course Objectives

1. Understand the OSI and TCP/IP network models and apply them in programming. 2. Gain proficiency in network programming using Python and the socket interface. 3. Design and implement applications that use UDP and TCP protocols. 4. Develop client and server programs that communicate using HTTP protocols. 5. Manage messaging and message queues using technologies like RabbitMQ. 6. Utilize network management protocols such as SNMP. 7. Distinguish between switching and routing techniques and explain their importance in modern network environments.	• ..... • ..... • .....

8. Apply concurrent programming techniques and handle common issues such as deadlocks.

Teaching and Learning Strategies

1. Cooperative Concept Planning Strategy.
2. Brainstorming Teaching Strategy.
3. Note-taking Sequence Strategy.

Course Structure

Weeks	Hours	Required Learning Outcomes	Unit or subject name	Learning method	Evaluation method
1	4 hours (2 Theory + 2 La	Understand OSI and TCP/I models and basic networki devices.	Review of Networking Concept	Lecture & lab practice on netw models and devices.	Weekly quizzes, lab reports
2	4 hours (2 Theory + 2 La	Learn Python programmin basics.	Introduction to Python	Coding exercises and live demonstrations.	Coding assignments
3	4 hours (2 Theory + 2 La	Use sockets to program UI connections.	Python Sockets Programming - UDP	Lab implementation of UDP server/client.	Lab tests
4	4 hours (2 Theory + 2 La	Use sockets to program TC connections.	Python Sockets Programming - TCP	Lab implementation of TCP server/client.	Lab reports
5	4 hours (2 Theory + 2 La	Implement HTTP clients.	HTTP Clients	Develop Python scripts using HTTP libraries.	Weekly quizzes
6	4 hours (2 Theory + 2 La	Develop basic HTTP serve	HTTP Servers	Build simple web servers with Python.	Lab demonstration
7	4 hours (2 Theory + 2 La	Use Message Queues for async tasks.	Messaging and Message Queue	Implement tasks using RabbitM	Lab project

8	4 hours (2 Theory + 2 La	Understand basics of netw management protocols.	Network Management (SNMP)	Configure SNMP tools and practice queries.	SNMP lab test
9	4 hours (2 Theory + 2 La	Explain switching and routing concepts.	Switching and Routing Essentia	Simulation of switches/routers using software tools.	Weekly exam
10	4 hours (2 Theory + 2 La	Apply concurrency and handle deadlocks in netwo programs.	Concurrency and Deadlocks	Code examples with threading and deadlock scenarios.	Practical coding test
11	4 hours (2 Theory + 2 La	Develop a mini project integrating learned concep	Project Implementation (Part 1)	Guided lab sessions for project development.	Project progress rep
12	4 hours (2 Theory + 2 La	Complete and present the project.	Project Implementation (Part 2)	Finalize project and deliver presentation.	Project presentation
13	4 hours (2 Theory + 2 La	Review and prepare for fin exam.	Revision & Final Exam Preparation	Mock exams and Q&A session	Final written exam

Course Evaluation

- Weekly quizzes & lab reports.
- Practical coding tests.
- Final term project & presentation.
- Written final exam.

Learning and Teaching Resources

Data and Computer Communications	William Stallings
Essential SNMP	O'Reilly Media
Python Official Documentation	python.org/doc
RabbitMQ Official Documentation	rabbitmq.com

Network Programming

1) Review of Related Networking Concepts

Dr. Hassan Almazini

A – Target Population: For students of Second Year
Technological Institute of Basra
Dep. Of Computer network and software technologies
1 / B – Rationale: Understanding basic networking concepts and layered models is essential to grasp the foundations of how modern network applications operate and communicate.
1 / C – Central Idea:

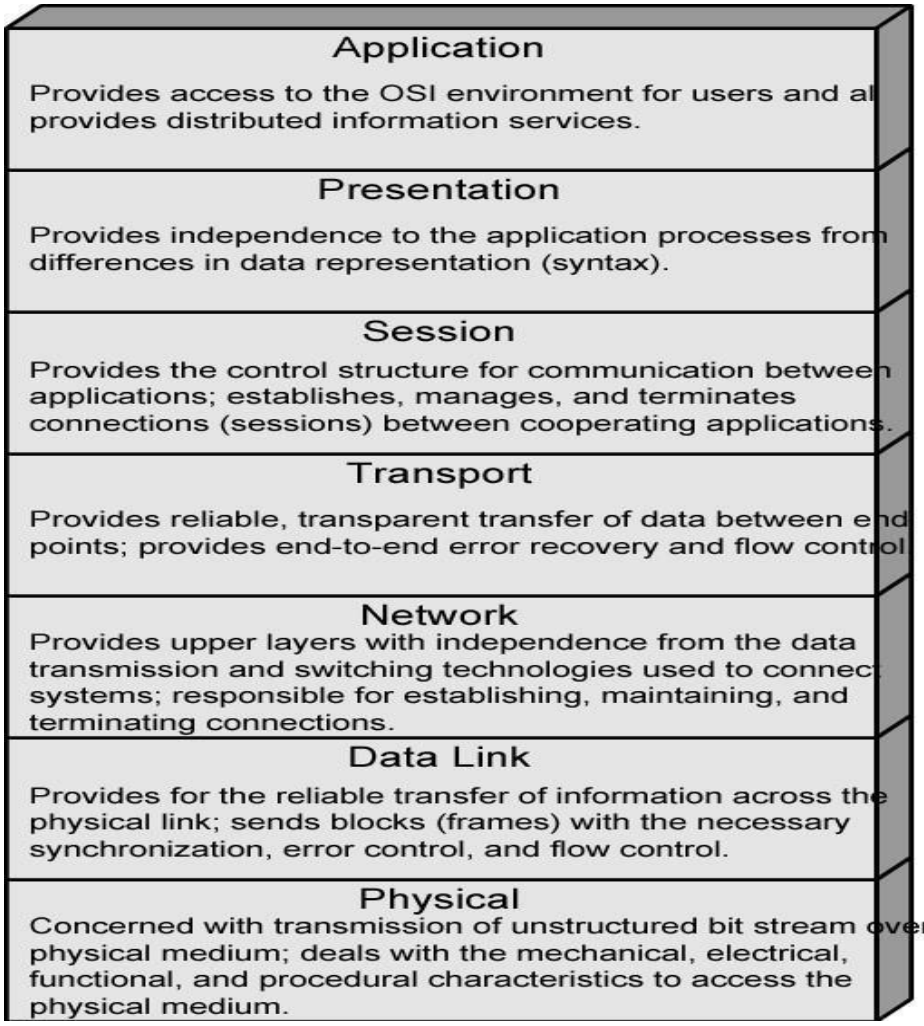
1. Overview of OSI vs TCP/IP models
2. Networking devices and data flow
3. Encapsulation and decapsulation
4. Client-server architecture

1 / D – Performance Objectives: After studying this unit, the student will be able to:

1. Differentiate between OSI and TCP/IP models.
2. Identify and describe the functions of networking devices.
3. Explain encapsulation and decapsulation processes.
4. Describe client-server architecture and communication.

Today, most programs are designed to work with networks!

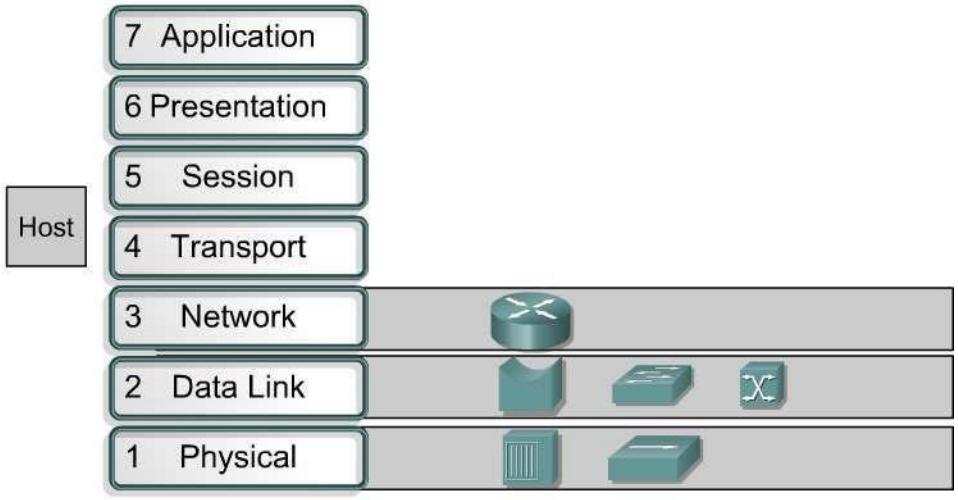
OSI Model



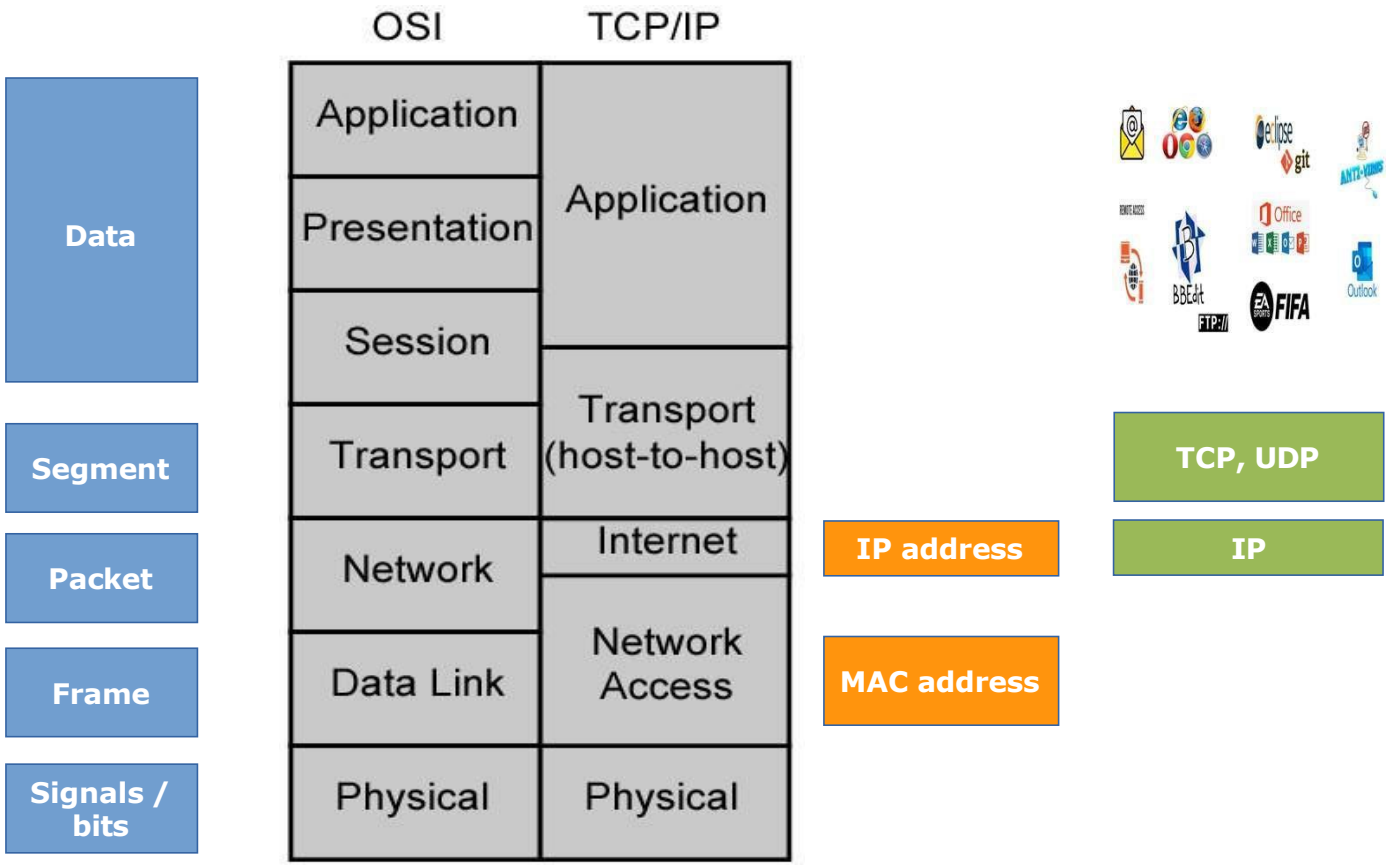
Why layered?

Source: Data and Computer Communications, 8th edition, By: William Stallings

Networking Devices



OSI Model vs. TCP/IP Model

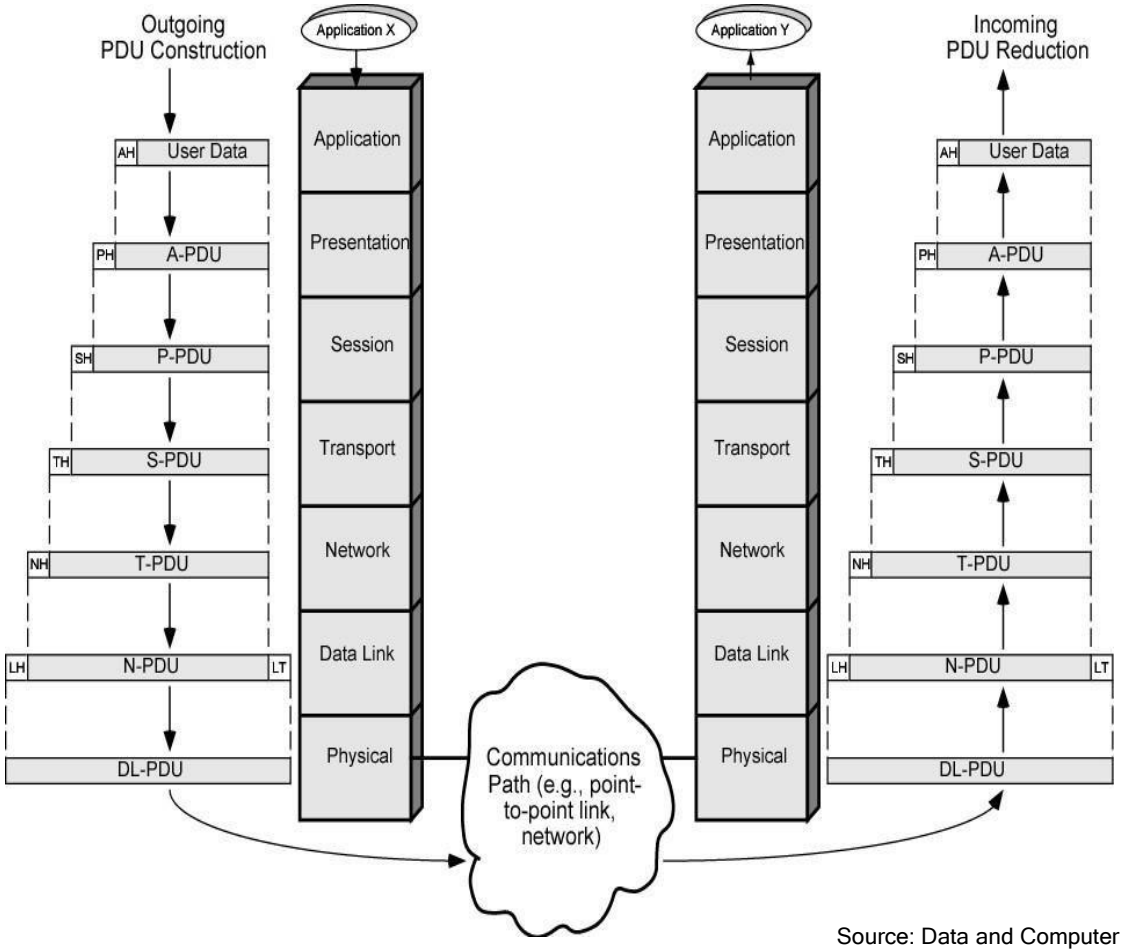


90% of the time your code will work in the application layer and only need to talk to the transport layer.

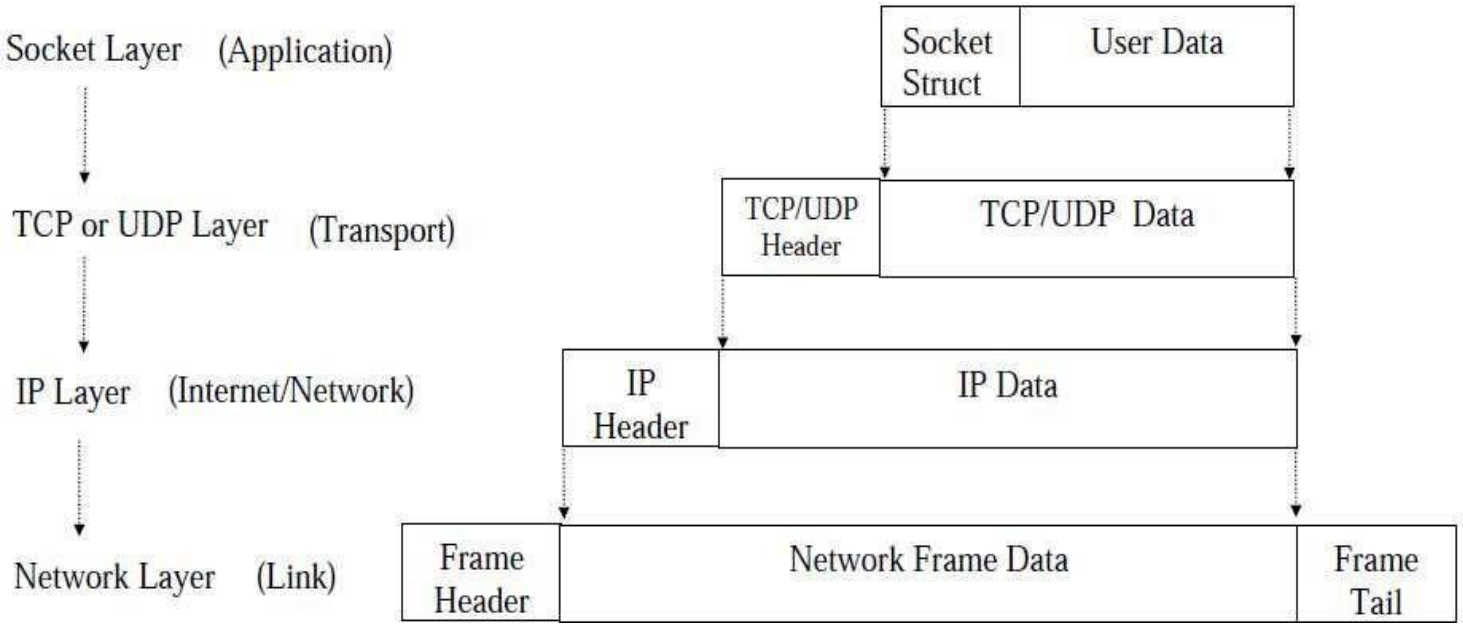
Source: Data and Computer Communications, 8th edition, By: William Stallings

Encapsulation / Decapsulation

Example

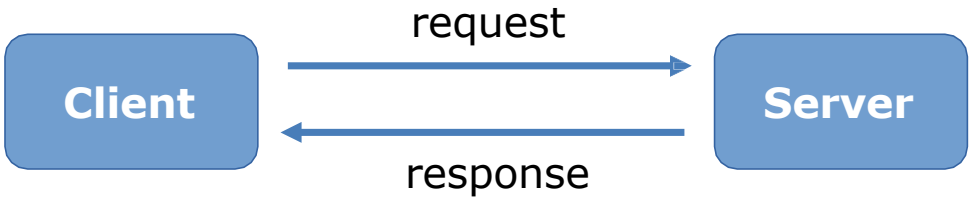


Source: Data and Computer Communications, 8th edition, By: William Stallings


Socket Layer (Application)
TCP or UDP Layer (Transport)
IP Layer (Internet/Network)
Network Layer (Link)
Socket Struct
User Data
TCP/UDP Header
TCP/UDP Data
IP Header
IP Data
Frame Header
Network Frame Data
Frame Tail

Most network application can be divided into two programs: client and server

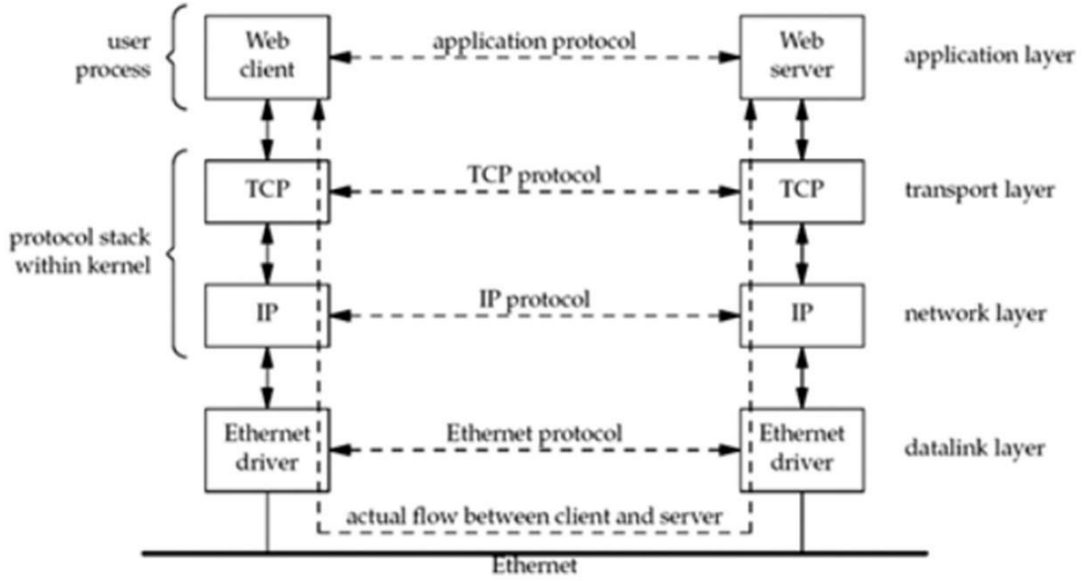
Examples?



Multiple layers of network protocols are typically involved in client-server communication

Client and server on the same Ethernet (using TCP)

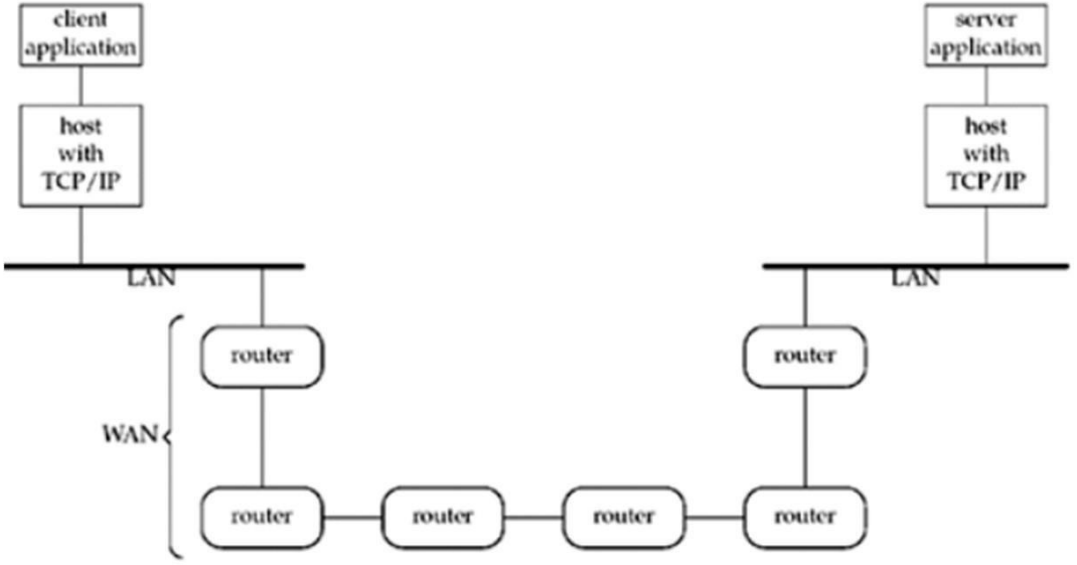
Protocol?



Source: Textbook 2

Multiple layers of network protocols are typically involved in client-server communication

Client and server on different LANs connecting through a WAN



Source: Textbook 2

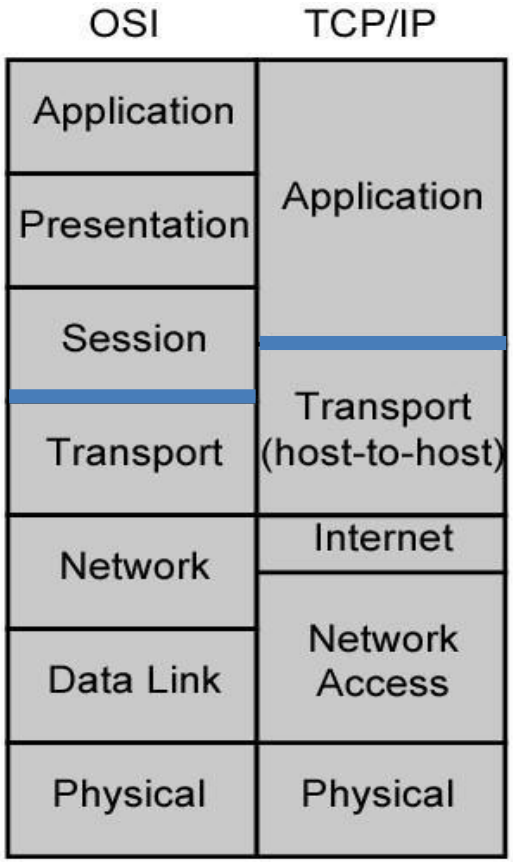
The sockets programming interfaces are interfaces from the upper layers (the “application”) into the transport layer

Handle all the details of the application, and know little about the communication details

User Process

Handle all the details of communication, and know little about the application

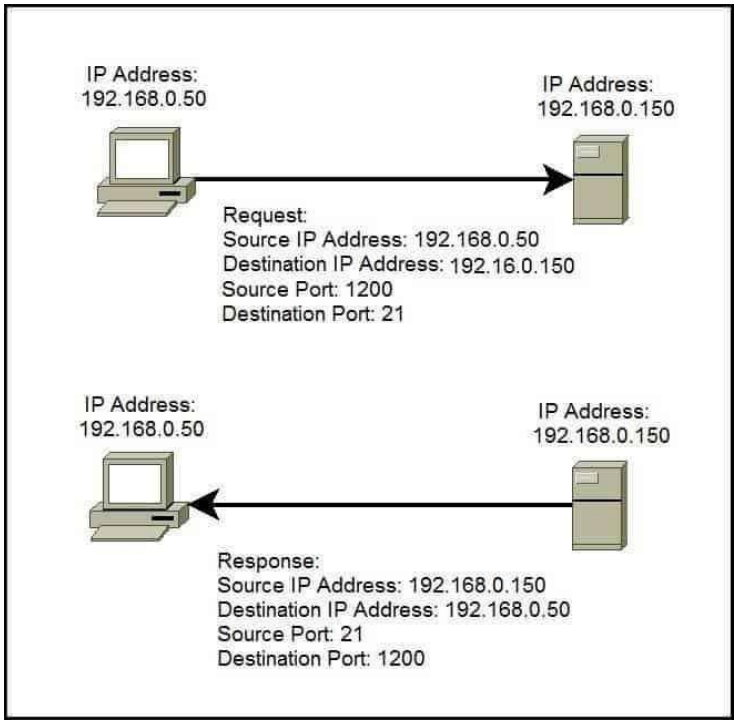
OS Kernel



Sockets

Why do sockets provide the interface from the upper three layers of the OSI model into the transport layer?

IP Address + Port Number = Socket



Port numbers are from 0 to 65535. The first 1024 ports are reserved for use by certain privileged services:

TCP		UDP	
FTP	20,21	DNS	53
SSH	22	BooTPS/DHCP	67
Telnet	23	TFTP	69
SMTP	25	NTP	123
DNS	53	SNMP	161
HTTP	80		
POP3	110		
IMAP4	143		
HTTPS	443		

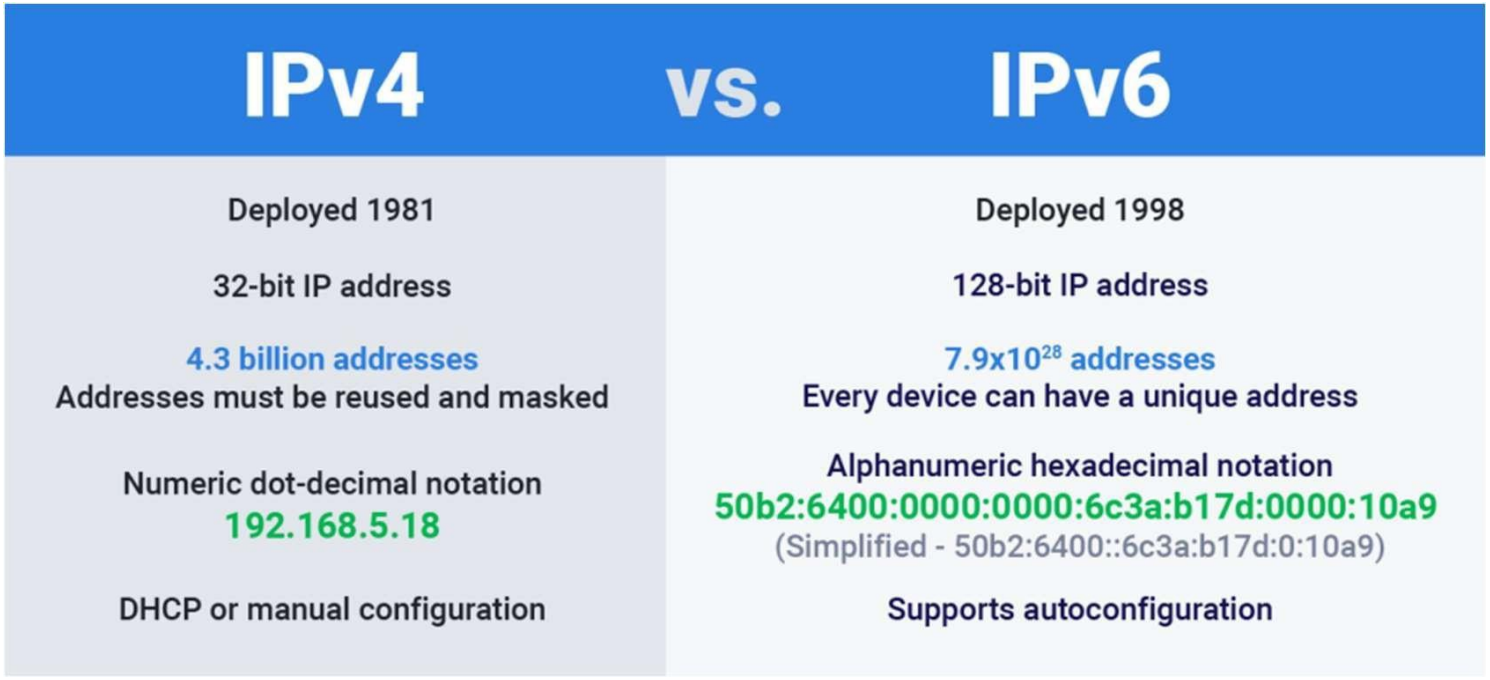
Source: https://study-ccna.com/ports-explained/

Five Classes of IPv4

Subnet?

Class	First Octet decimal (range)	First Octet binary (range)	IP range	Subnet Mask	Hosts per Network ID	# of networks
Class A	0 – 127	0XXXXXXX	0.0.0.0-127.255.255.255	255.0.0.0	$2^{24}-2$	2^{7}
Class B	128 – 191	10XXXXXX	128.0.0.0-191.255.255.255	255.255.0.0	$2^{16}-2$	2^{14}
Class C	192 – 223	110XXXXX	192.0.0.0-223.255.255.255	255.255.255.0	$2^{8}-2$	2^{21}
Class D (Multicast)	224 – 239	1110XXXX	224.0.0.0-239.255.255.255			
Class E (Experimental)	240 – 255	1111XXXX	240.0.0.0-255.255.255.255			

Source: https://medium.com/networks-security/tricks-to-remember-five-classes-of-ipv4-484c191678fb



Source: https://howtocheckversion.com/how-to-determine-if-you-are-on-ipv4-or-ipv6/

Homework 1

- What is the IETF? Where and when will the next IETF meeting take place?
- Before a new protocol can become an Internet standard, it will go through different steps. Briefly define, and then order the following types of document in the correct time line: "draft standard (RFC)", "Internet standard (RFC/STD)", "Internet draft (I-D)", "proposed standard (RFC)".
- What is the best place to search for an RFC? For an Internet draft?
- You are tasked to implement a server supporting RFC 1531. What kind of server you will implement?
- You are looking for RFCs with "BGP" in the title. (BGP – Border Gateway Protocol). How many RFCs do you find? Is anyone of them an Internet Standard (STD)?
- What IETF means by “an RFC obsoletes another RFC”. Give an example.
- The IETF standards are developed in different Working Groups. Find the home page for an IETF Working Group which deals with a topic you find interesting. Then perform the following tasks:
  - State which IETF Working Group you picked!
  - Pick one of the RFCs or IETF drafts published by this Working Group. State the title of the document.
  - RFCs and IETF drafts usually have an introduction section. Read it! Then write a short summary (a few lines) of the topic of this standards document.

Recommended Reading Material and Learning Resources

- Rick Graziani, slides of networking courses (Cabrillo College)
- Sam Hsu, slides of computer network programming course (Florida Atlantic University)
- Textbook 2: Chapter 1

Questions?

Network Programming

2) Introduction to Python

Dr. Hassan Almazini

A – Target Population: For students of Second Year
Technological Institute of Basra
Dep. Of Computer network and software technologies
2 / B – Rationale: Python is a powerful, versatile programming language commonly used in network programming. Its readability and extensive libraries make it ideal for developing and testing network applications.
2 / C – Central Idea:

1. Introduction to Python syntax and structure
2. Variables, types, and referencing
3. Sequence types: lists, tuples, strings
4. Functions, conditionals, loops

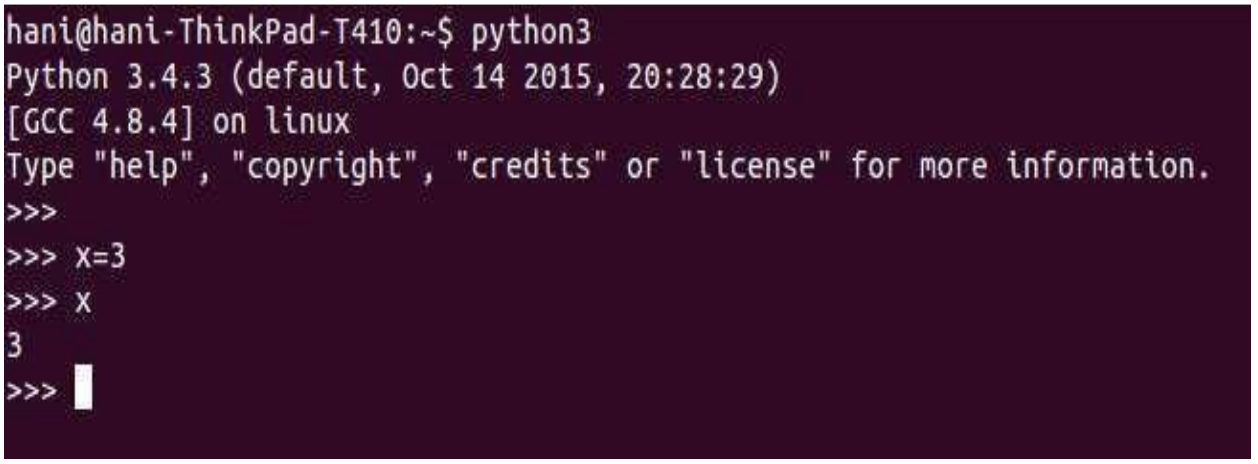
2 / D – Performance Objectives: After studying this unit, the student will be able to:

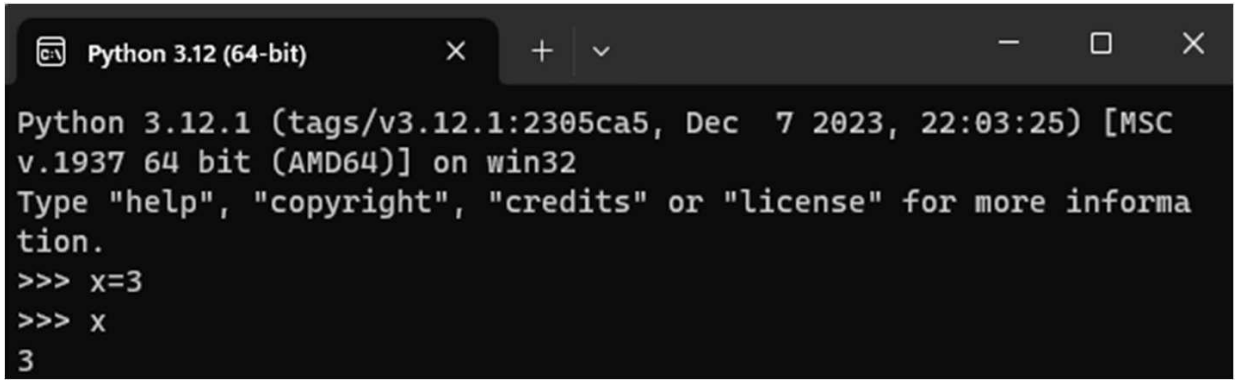
1. Write basic Python programs using correct syntax.
2. Use variables, lists, dictionaries, and functions appropriately.
3. Implement logical conditions and loops to control program flow.
4. Understand the difference between mutable and immutable types.

Python

- Open source
- General-purpose
- Object Oriented, Procedural, Functional
- Easy to interface with C, Java, Fortran (and C++)
- Versions: 2.5.x / 2.6.x / 3.x
- http://www.python.org
- http://www.python.org/doc/

Why Python?


hani@hani-ThinkPad-T410:~$ python3
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> x=3
>>> x
3
>>>


Python 3.12 (64-bit)
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC
v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more informa
tion.
>>> x=3
>>> x
3

HS1

test.py

```
1
2  x = 14 + 3  # A comment.
3
4  y = "Hi" # Another comment
5
6  z = 2.5
7
8  if z == 4.2 or y == "Hi":
9      x = x + 1
10     y = y + " World"
11
12 print x
13 print y
14
```

python test.py

- **Comments**
- **Operators**
- **Assignment (reference)**
- **Create variables (names)**
- **Variable types**
- **Print**
- **Blocks**
- **Garbage collection**
- **Access non-existing name**

Slide 4

HS1 Hassan Almazini;
10.02.2024

Naming

- Case-sensitive
- Letters, numbers, underscore
- Don't start with a number
- Don't use the **reserved words**

and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while

Referencing

integer, float, string, and tuple are "immutable"

```
>>> y = 2
```

- an integer 2 is created and stored in memory
- a name y is created
- a reference to the memory location storing 2 is assigned to y
- The value of y is 2 → y refers to the integer 2

```
>>> x = y
```

makes x reference the object y references

```
>>> y = y + 2
```

```
>>> x
2
```

```
>>> y
4
```

Referencing (Cont.)

```
>>> x = [1, 2, 3]   # list

>>> y = x

>>> x.append(4)

>>> x
[1, 2, 3, 4]

>>> y
[1, 2, 3, 4]

>>> x.insert(2,'i')
>>> x
[1, 2, 'i', 3, 4]
```

lists, dictionaries, user-defined types are “mutable”

Sequence Types

Tuples

```
>>> T = ('a', 4, 5.6)
>>> T [1]
4
>>> T [1] = 5
```

Lists

```
>>> L = ['a', 4, 5.6]
>>> L [1]
4
>>> L [1] = 2
>>> L                    # points to the same memory reference
['a', 2, 5.6]
```

Strings

```
>>> S1 = “hello!”
>>> S1 [1]
e
```

Sequence Types (Cont.)

```
>>> T = ('a', 4, 5.6, 8 , 9)
>>> T [-2]
8

>>> T [1 : 3]
4, 5.6

>>> T [1 : -1]
4, 5.6, 8

>>> T [ : -1]
'a', 4, 5.6, 8

>>> T [2: ]
5.6, 8, 9

>>> T [ : ]
'a', 4, 5.6, 8 , 9
```

Sequence Types (Cont.)

```
>>> list2 = list1
# 2 names refer to 1 ref
# Changing one affects both


>>> list2 = list1[:]
# 2 independent copies, 2 references
```

“in” Operator

```
>>> T = (1, 2, 3, 4)
>>> S = “hello”

>>> 1 in T
True

>>> 5 in T
False

>>> 'e' in S
True

>>> 'el' in S
True

>>> 'eo' in S
False
```

Concatenation

```
>>> L1 = [1, 2, 3, 4]

>>> S1 = “hello”

>>> S2 = “world”


>>> L1 + [4, 5, 6, 7]
[1, 2, 3, 4, 4, 5, 6, 7]


>>> S1 + “ ” + S2 + “!”
“hello world!”
```

“*” Operator

```
>>> 3 * 2
6


>>> [4, 5, 6, 7] * 3
[4, 5, 6, 7, 4, 5, 6, 7, 4, 5, 6, 7]


>>> (5, 6, 7) * 3
(5, 6, 7, 5, 6, 7, 5, 6, 7)


>>> S2 = “world”
>>> S2 * 2
“worldworld”
```

Other Operators

```
>>> L = [1, 2, 3, 3, 3]
>>> T = (1, 2, 3, 3, 3)


>>> T.index(3)                  # same for L
2


>>> T.count(3)                  # same for L
3


>>> L.remove(3)
>>> L
[1, 2, 3, 3]

>>> T.remove(3)
```

Other Operators

```
>>> L = [1, 2, 3, 3, 3]
>>> T = (1, 2, 3, 3, 3)


>>> L.reverse()
>>> L
[3, 3, 3, 2, 1]

>>> L.sort()
>>> L
[1, 2, 3, 3, 3]


>>> T.reverse()
>>> T.sort()
```

Lists & Tuples

- Tuples are immutable, faster and less powerful than lists
- **Conversion**

```
>>> L = list(T)

>>> T = tuple(L)
```

Dictionaries

- Store a mapping between a set of **keys** and a set of **values**
  - **Keys:** any immutable type
  - **Values:** any type
- A single dictionary can hold values of different types
- **Operations:** define, modify, view, lookup, and delete the key-value pairs in the dictionary

Dictionaries (Cont.)

```
>>> d = {'name' : 'Ahmad', 'password' : 'abcd'}

>>> d['password']
'abcd'

>>> d['Ahmad']
```

```
>>> d['password'] = 'xyz'          # mutable
>>> d
{'name' : 'Ahmad', 'password' : 'xyz'}


>>> d['id'] = 55
>>> d
{'name' : 'Ahmad', 'password' : 'xyz', 'id' : 55}
```

Dictionaries (Cont.)

```
>>> d
{'name' : 'Ahmad', 'password' : 'xyz', 'id' : 55}

>>> d.keys()
['name' , 'password' , 'id']          # list of keys

>>> d.values()
['Ahmad' , 'xyz' , 55] # list of values

>>> d.items()
[ ('name' , 'Ahmad') , ('password' , 'xyz' ) , ('id' , 55) ]

>>> del d['id']
>>> d
{'name' : 'Ahmad' , 'password' : 'xyz'}

>>> d.clear()
>>> d
{}
```

Functions

```
def add (a, b):
    return a + b


def change (a, b):
    a = 2
    b[1] = 3
```

Functions (Cont.)

```
def myfunc (a, b, c = 10, d = 20):
   print a , print b, print c, print d
```

```
>>> myfunc (3, 4)
3  4  10  20
```

```
>>> myfunc (1, 2, 3, 4)
1, 2,  3,  4
```

Functions (Cont.)

- No function overloading
  - Two different functions can’t have the same name, even if they have different arguments.

- Functions can:
  - be arguments to function
  - return values of functions
  - be assigned to variables
  - be parts of tuples, lists, etc.

“if” Statement

```
if a == 4:
  print “a equals 4”
elif a == 3:
   print “a equals 3”
else:
   print “neither 4 nor 3!”
print “this is not part of if”
```

“for” Statement

```
for a in range (5):                    # print 0 to 4
  print a
print “this is not part of the loop”
```

```
for a in range (1, 5):                 # print 1 to 4
  print a
```

```
for a in range (1, 10):        # print 1, 2, 3, 5, 6, 7
  if x == 4:
     continue
  if x == 8:
     break
  print a
```

“while” Statement

```
a = 5

while a < 10:
  print (a)
  a+=2
```

Recommended Reading Material and Learning Resources

http://tdc-www.harvard.edu/Python.pdf

Questions?

Network Programming

3) Python Sockets Programming – UDP

Dr. Hassan Almazini

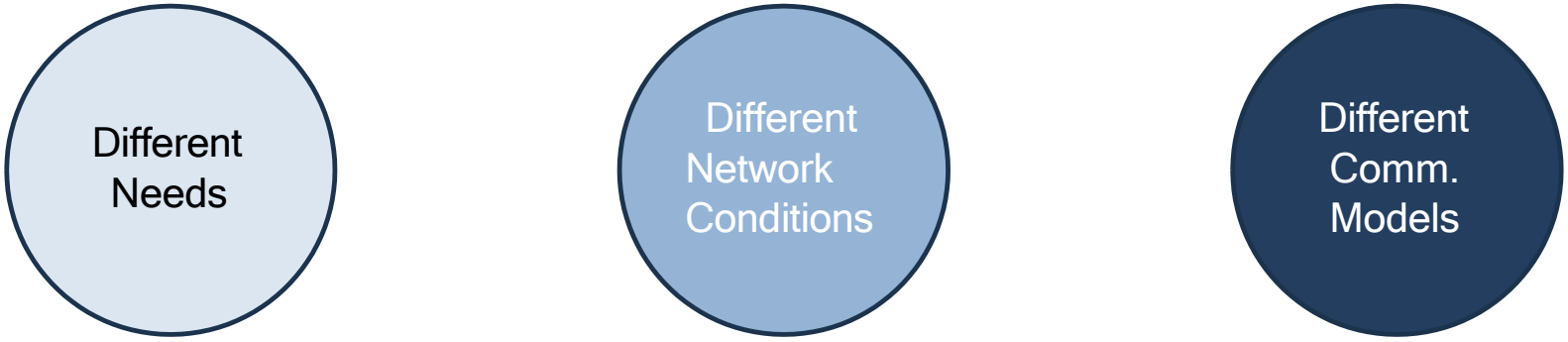
A – Target Population: For students of Second Year Technological Institute of Basra
Dep. Of Computer network and software technologies
3 / B – Rationale: Understanding how to send and receive data using the UDP protocol in Python is essential for building efficient, low-latency applications like streaming, gaming, or real-time communication systems.
3 / C – Central Idea:

1. Introduction to UDP protocol
2. Python socket programming basics
3. Writing simple UDP server and client
4. Security issues and mitigations

3 / D – Performance Objectives: After studying this unit, the student will be able to:

1. Describe the characteristics and behavior of the UDP protocol.
2. Create a simple UDP server and client in Python.
3. Implement message exchange using socket APIs.
4. Identify potential security vulnerabilities in UDP communication.

The existence of both UDP and TCP sockets allows to choose the appropriate protocol based on the specific needs of the applications



User Datagram Protocol (UDP) is a connectionless transport layer protocol in the Internet Protocol Suite

- Provides a lightweight, best-effort delivery mechanism
  - UDP has minimal overhead
  - No handshaking, congestion control, acknowledgment mechanisms, error checking or retransmission
  - (Ordered) delivery is not guaranteed
- Operates by sending datagrams between devices without establishing a connection (i.e., **connectionless**)
  - Faster but less reliable than TCP
- Often used for time-sensitive applications where
  - Low latency and speed are prioritized over guaranteed delivery
  - Occasional packet loss is acceptable

The checksum in UDP helps to detect errors, but doesn't provide mechanisms for recovering lost packets or ensuring their correct order

Source: https://voip-sip-sdk.com/p_7231-user-datagram-protocol.html

UDP server that listens on port 9000 for incoming messages from clients

Python socket module provides access to the socket interface

Max number of bytes the server can receive in one datagram

&

the port number on which the server will listen for incoming datagrams

```python
import socket

MAX_BYTES = 65535
port = 9000

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('127.0.0.1', port))
print('Listening at {}'.format(sock.getsockname()))
while True:
    data, address = sock.recvfrom(MAX_BYTES)
    text = data.decode('ascii')
    print('The client at {} says {!r}'.format(address, text))
    text = 'Your data was {} bytes long'.format(len(data))
    data = text.encode('ascii')
    sock.sendto(data, address)
```

Creates a socket object using IPv4 (AF_INET) and UDP (SOCK_DGRAM) protocols

Binds the socket to the localhost (127.0.0.1) and the specified port number (9000)

Receives data from a client and stores it in the data variable & retrieves the address of the client that sent the data

Sends the response message back to the client at the address from which the data was received

UDP client that sends a message containing the current time to a server listening on port 9000 at localhost, then receives a response & prints it

Imports the datetime class from the datetime module

```
1  import socket
2  from datetime import datetime
3
4  MAX_BYTES = 65535
5  port = 9000
6
7  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
8  text = 'The time is {}'.format(datetime.now())
9  data = text.encode('ascii')
10 sock.sendto(data, ('127.0.0.1', port))
11 print('The OS assigned me the address {}'.format(sock.getsockname()))
12 data, address = sock.recvfrom(MAX_BYTES)
13 text = data.decode('ascii')
14 print('The server {} replied {!r}'.format(address, text))
```

Run the server, follow with the client (multiple times), and then the server again

```
Listening at ('127.0.0.1', 9000)
The client at ('127.0.0.1', 64784) says 'The time is 2024-02-19 03:50:06.673984'
The client at ('127.0.0.1', 64785) says 'The time is 2024-02-19 03:50:10.972419'
The client at ('127.0.0.1', 64786) says 'The time is 2024-02-19 03:50:12.241157'
The client at ('127.0.0.1', 64787) says 'The time is 2024-02-19 03:50:13.097184'
```

```
The OS assigned me the address ('0.0.0.0', 51440)
The server ('127.0.0.1', 9000) replied 'Your data was 38 bytes long'
```

Run another copy of the server

```
Traceback (most recent call last):
  File "ch2_1_udp_server.py", line 9, in <module>
    sock.bind(('127.0.0.1', port))
  File "/usr/lib/python2.7/socket.py", line 224, in meth
    return getattr(self._sock,name)(*args)
socket.error: [Errno 98] Address already in use
```

Solution: Run the script with different port numbers

python server2.py 8000

python server2.py 8001

...

```python
import socket
import sys

from datetime import datetime

MAX_BYTES = 65535

if len(sys.argv) != 2:
    print("Usage: python simple_server2.py <port>")
    exit(1)

try:
    port = int(sys.argv[1])
except ValueError:
    print("Invalid port number")
    exit(1)

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('127.0.0.1', port))
print('Listening at {}'.format(sock.getsockname()))

while True:
    data, address = sock.recvfrom(MAX_BYTES)
    text = data.decode('ascii')
    print('The client at {} says {!r}'.format(address, text))
    text = 'Your data was {} bytes long'.format(len(data))
    data = text.encode('ascii')
    sock.sendto(data, address)
```

Three Binding Options

```
sock.bind (('127.0.0.1', port))
```

- **'127.0.0.1':** the server will listen to packets from other programs running on the same machine only
- **' ' :** the server will listen to packets arriving via any of its network interfaces
- **Some IP (e.g. '193.10.0.22'):**
  - One of the machine's IP interfaces
  - The server will listen only for packets destined for the specified IP

The client code doesn't check the source address of the datagram it receives to verify that it is actually a reply from the server!

```
1  import socket
2  from datetime import datetime
3
4  MAX_BYTES = 65535
5  port = 9000
6
7  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
8  text = 'The time is {}'.format(datetime.now())
9  data = text.encode('ascii')
10 sock.sendto(data, ('127.0.0.1', port))
11 print('The OS assigned me the address {}'.format(sock.getsockname()))
12 data, address = sock.recvfrom(MAX_BYTES)
13 text = data.decode('ascii')
14 print('The server {} replied {!r}'.format(address, text))
```

The client is vulnerable to anyone who can address a UDP packet to it

- Freeze the server (ctrl + z) (note: type “fg” to unfreeze)
- Run the client

```
The OS assigned me the address ('0.0.0.0', 34641)
```

- Fake Server:

```
>>> import socket
>>> sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
>>> sock.sendto('fake', ('127.0.0.1', 34641))
```

- The client output now:

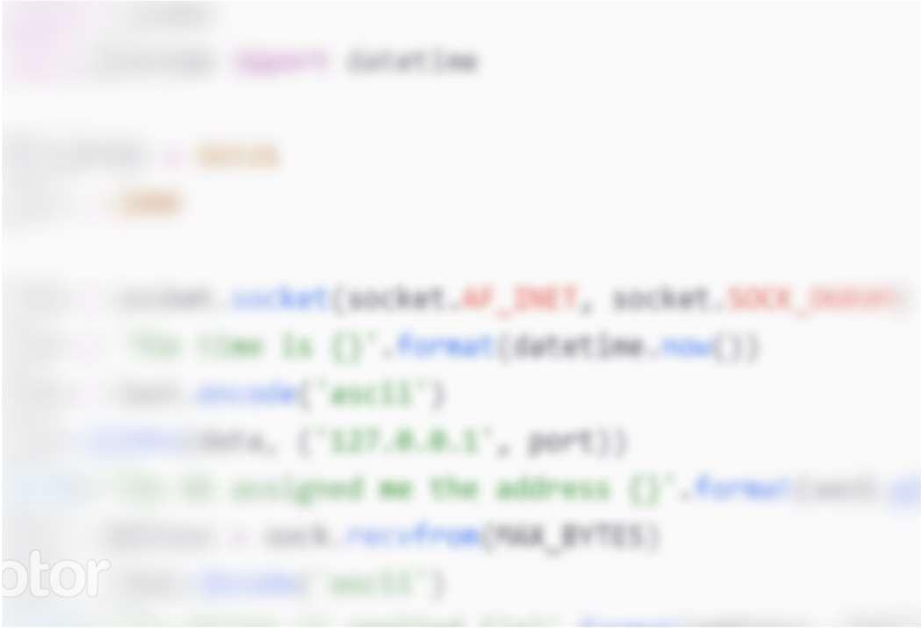
```
The OS assigned me the address ('0.0.0.0', 34641)
The server ('127.0.0.1', 59166) replied 'fake'
```

Solution 1: Add a check to ensure that the response comes from a trusted source

```python
import socket
from datetime import datetime

MAX_BYTES = 65535
port = 9000
trusted_server_address = '127.0.0.1' # Example trusted server address

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
text = 'The time is {}'.format(datetime.now())
data = text.encode('ascii')
sock.sendto(data, (trusted_server_address, port))
print('The OS assigned me the address {}'.format(sock.getsockname()))
data, address = sock.recvfrom(MAX_BYTES)

# Check if the response comes from the trusted server
if address[0] == trusted_server_address:
    text = data.decode('ascii')
    print('The server {} replied {!r}'.format(address, text))
else:
    print('Received a response from an untrusted source:', address)
```

Solution 2: Use “connect()” to establish a connection with the server, “send()” to send data and “recv()” to receive responses

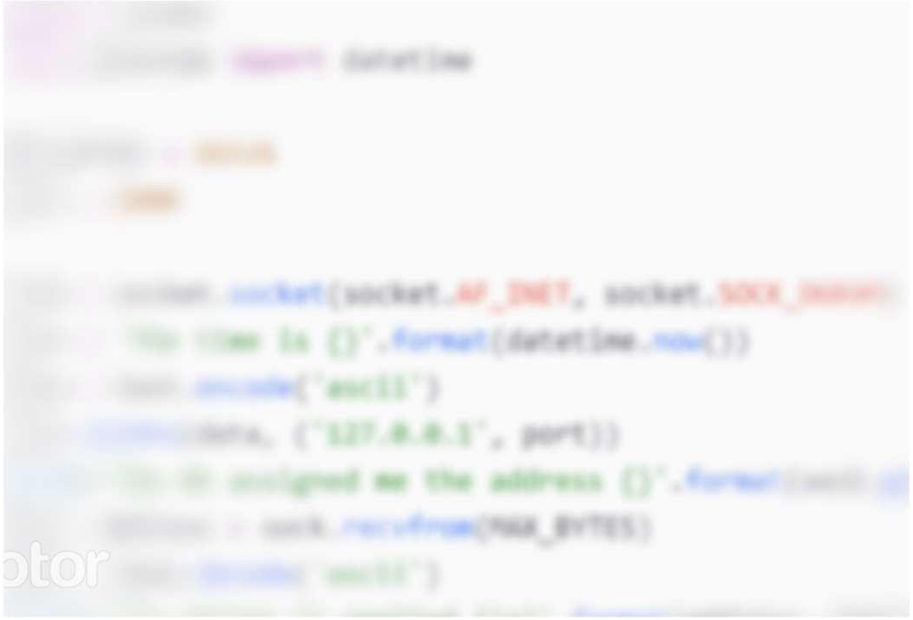
- When **connect()** is used, the socket is associated with a specific remote address and port
  - Only packets from that address are accepted
  - Implicitly filtering out responses from unauthorized sources
- Simple and easy to use
- It may be not flexible
  - If the client needs to communicate with multiple servers
  - If the server’s address is dynamic



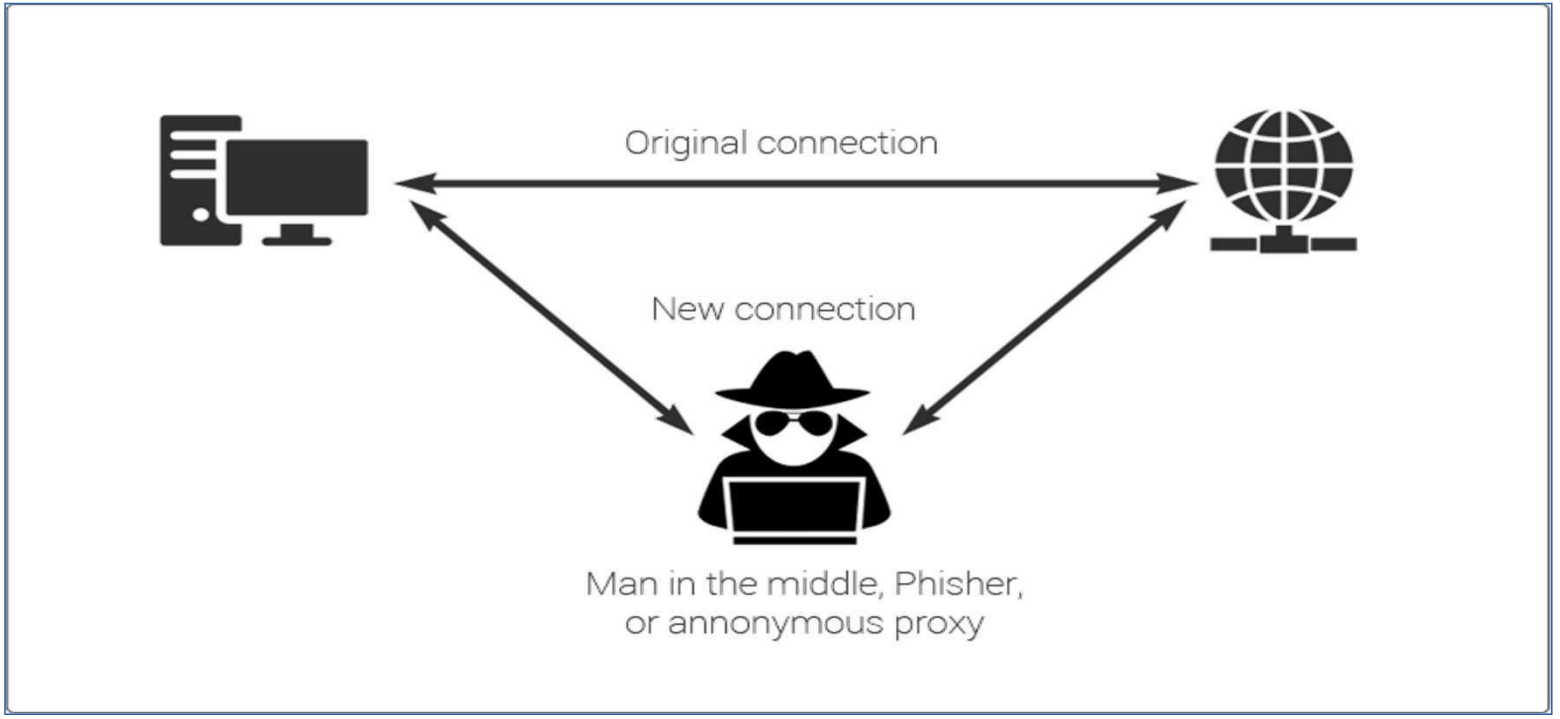
Solution 3: Include a unique ID in the request that gets repeated in the reply, and accept the reply if it contains the same ID

- Useful only when the attacker cannot see the requests!
- Ensure that the IDs are truly **random** and **unpredictable**
- Avoid using **predictable** sequences or algorithms that can be easily **guessed**
- **Longer IDs**, e.g., 128 bits, generally provide higher security against **brute-force attacks**
- Ensure that the IDs are **unique** (**collisions** could lead to confusion or security vulnerabilities)
- Consider including a **validity period** or **expiration time** with the IDs
- Consider **hashing the IDs** before sending them, and incorporating **authentication mechanisms**

Solution 3: Include a unique ID in the request that gets repeated in the reply, and accept the reply if it contains the same ID



Man-In-The-Middle Attack (MITM)



Source: https://securebox.comodo.com/ssl-sniffing/man-in-the-middle-attack

UDP is unreliable:
Packets might be lost; server might stop

```python
import random, socket

MAX_BYTES = 65535
port = 9000

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('127.0.0.1', port))
print('Listening at {}'.format(sock.getsockname()))

while True:
    data, address = sock.recvfrom(MAX_BYTES)
    if random.random() < 0.5:
        print('Pretending to drop packet from {}'.format(address))
        continue
    text = data.decode('ascii')
    print('The client at {} says {!r}'.format(address, text))
    message = 'Your data was {} bytes long'.format(len(data))
    sock.sendto(message.encode('ascii'), address)
```

- **Lucky Client:**

```
The OS assigned me the address ('0.0.0.0', 60161)
The server ('127.0.0.1', 9000) replied 'Your data was 38 bytes long'
```

- **Unlucky (Blocking) Client:**

```
The OS assigned me the address ('0.0.0.0', 65390)
```

- **The Server:**

```
Listening at ('127.0.0.1', 9000)
Pretending to drop packet from ('127.0.0.1', 64322)
Pretending to drop packet from ('127.0.0.1', 65390)
The client at ('127.0.0.1', 60161) says 'The time is 2024-02-18 20:00:02.411911'
```

Applications that use UDP typically handle reliability and ordering at a higher level, by implementing their own protocols on top of UDP

A “real” UDP client that can deal with the fact that packets might be lost

- The client has to perform its request inside a loop
- **The client doesn’t know the reason:**
  - The reply is taking a long time to come back, but it will soon arrive
  - The reply will never arrive because it, or the request, was lost
  - The server is down
- The client has to choose a **schedule** on which it will send duplicate requests if it waits a reasonable period without getting a response
- **Solution:** Exponential Backoff

```python
import socket
from datetime import datetime

MAX_BYTES = 65535
port = 9000

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
hostname = '127.0.0.1'

sock.connect((hostname, port))
print('Client socket name is {}'.format(sock.getsockname()))
delay = 0.1 # seconds
text = 'This is another message'
data = text.encode('ascii')
while True:
    sock.send(data)
    print('Waiting up to {} seconds for a reply'.format(delay))
    sock.settimeout(delay)
    try:
        data = sock.recv(MAX_BYTES)
    except socket.timeout:
        delay *= 2 # wait even longer for the next request
        if delay > 2.0:
            raise RuntimeError('I think the server is down')
    else:
        break # we are done, and can stop looping
```

```
Waiting up to 0.1 seconds for a reply
Waiting up to 0.2 seconds for a reply
Waiting up to 0.4 seconds for a reply
```

```
RuntimeError: I think the server is down
```

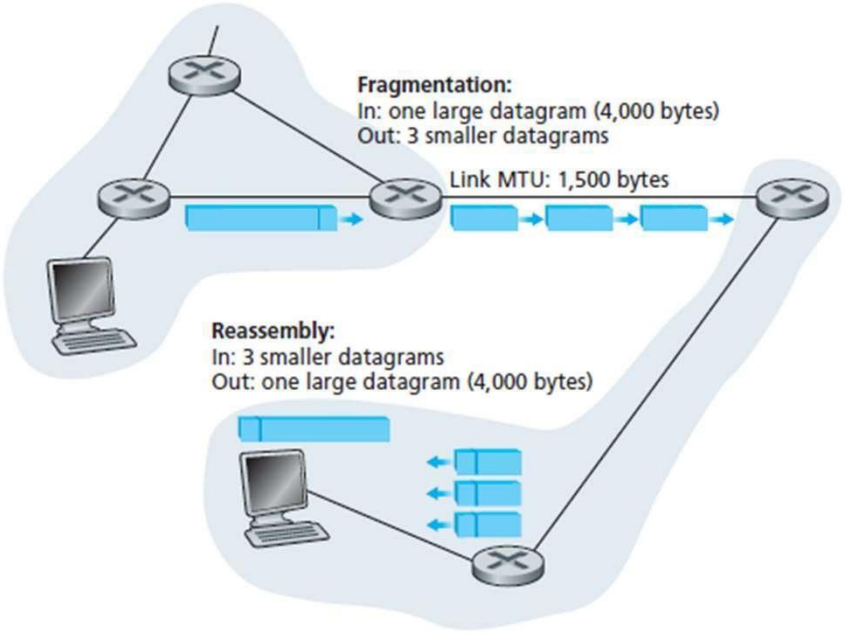
Broadcasting: Sending datagrams to an entire subnet to which your machine is attached

```python
import socket

port = 9000
network = '192.168.1.255'

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
text = 'Broadcast datagram!'
sock.sendto(text.encode('ascii'), (network, port))
```

```
hani@hani-ThinkPad-T410:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr f0:de:f1:46:90:1a
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:20 Memory:f8500000-f8520000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:6812 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6812 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:2017168 (2.0 MB)  TX bytes:2017168 (2.0 MB)

wlan0     Link encap:Ethernet  HWaddr 00:24:d7:9b:81:94
          inet addr:192.168.1.103  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::224:d7ff:fe9b:8194/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:170347 errors:0 dropped:0 overruns:0 frame:0
          TX packets:129906 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:193096013 (193.0 MB)  TX bytes:19541114 (19.5 MB)
```

Fragmentation refers to the process of breaking down large packets of data into smaller fragments

- This is necessary when the size of the data packet exceeds the maximum size allowed by a network's **Maximum Transmission Unit (MTU)**
- MTU is the size of the largest PDU that all the devices between two hosts support
- **Examples:**
  - MTU = 1500 bytes in Ethernet v2
  - IEEE 802.11 (Wi-Fi): Typically >= 2304 bytes
- Fragmentation can introduce overhead and potentially impact network performance, so it's generally more efficient to avoid it when possible



Source: https://electronicspost.com/ip-datagram-fragmentation-with-example/

Fragmentation refers to the process of breaking down large packets of data into smaller fragments

- In the provided scripts, the MAX_BYTES constant is set to 65535, which represents the maximum size of data that the client can receive in a single UDP datagram
  - This value is the theoretical maximum size for UDP datagrams

```
MAX_BYTES = 65535
```

- In practice, the actual size may be limited by the MTU of the network
- It's essential to consider the MTU of the network when determining the appropriate maximum size for UDP datagrams in real-world applications
- If a UDP datagram exceeds the MTU size of the network, it will be fragmented into smaller packets during transmission, potentially leading to inefficiencies and performance issues
- What if you don't have prior knowledge of the networks and paths across which packets will travel?

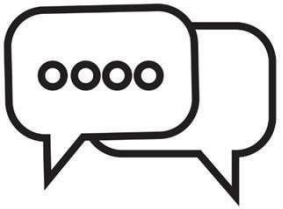
Recap

- UDP lets apps send individual packets across an IP network
  - A **client** program sends a packet to a server
  - The **server** then replies using the return address built into every UDP packet
- The network stack gives access to UDP through a **socket**, which is a communications endpoint that can sit at an IP address and UDP port number (IP address + port number = socket address)
- Python offers these primitive network operations through the built-in **socket** module
- The server needs to **bind()** to an address and port before it can receive incoming packets
- Client UDP programs can just start sending, and the OS will give them a port number automatically

Recap (Cont.)

- UDP is unreliable
  - Packets can be dropped, e.g., because of a network glitch or busy network segment
  - Clients have to compensate for this by **retransmitting** a request until they receive a reply
  - To prevent making a busy network even worse, clients should use **exponential backoff**, and they should make their initial wait time longer if they find that round-trips to the server are taking long
- **Request IDs** are crucial to combat the problem of reply **duplication**
  - If randomly chosen, request IDs can also help protect against naive spoofing attacks
- **connect()** limits replies received so that they can come only from the specified server
- The **broadcast** option lets you send packets to every host on the subnet

Experiment with crafting more advanced applications using the concepts we have covered!



- Develop a simple command-line chat application over UDP
- Each user's application will act both as a client and a server
- Each user have a command-line interface where he can type messages to send to other users (individual and broadcast) and view incoming messages
- Include features such a:
  - User authentication
  - Message encryption
  - Message history

"By the time you have a UDP protocol kind of working for your application, you have probably just reinvented TCP badly!"

Source

CHAPTER 2

UDP

Questions?

Network Programming

4) Python Sockets Programming – TCP

Dr. Hassan Almazini

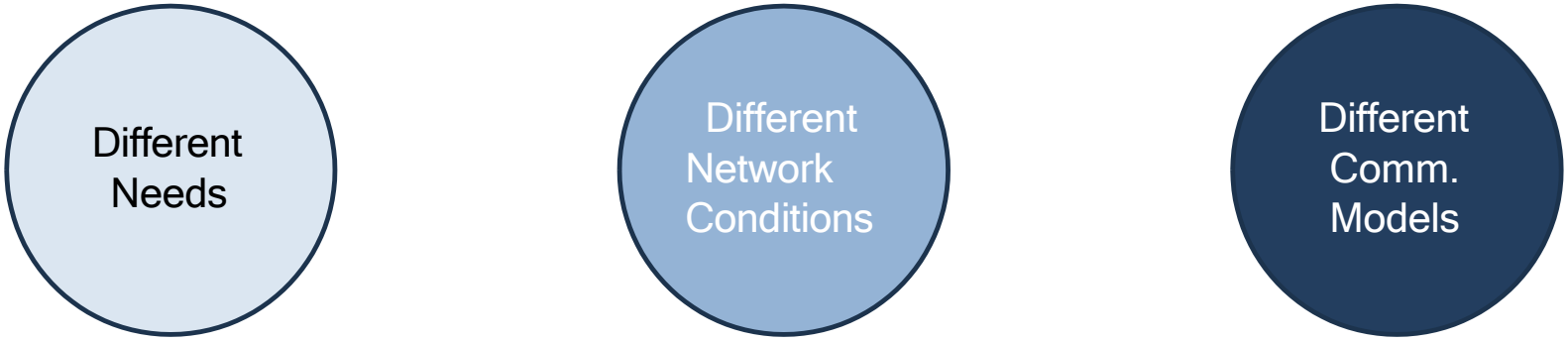
A – Target Population: For students of Second Year Technological Institute of Basra
Dep. Of Computer network and software technologies
4 / B – Rationale: TCP provides reliable, ordered, and error-checked delivery of a stream of bytes between applications. This unit teaches students how to implement TCP-based applications using Python sockets.
4 / C – Central Idea:

1. TCP fundamentals and handshake
2. Writing TCP server and client
3. Handling send and receive operations
4. Understanding deadlock and stream handling

4 / D – Performance Objectives: After studying this unit, the student will be able to:

1. Understand TCP connection setup, data flow, and teardown.
2. Build reliable TCP server-client applications using Python.
3. Use `send()` and `recv()` correctly for robust data exchange.
4. Prevent deadlocks and implement proper stream termination.

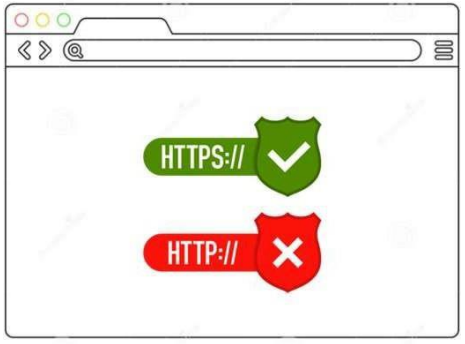
The existence of both UDP and TCP sockets allows to choose the appropriate protocol based on the specific needs of the applications



Transmission Control Protocol (TCP) is a connection-oriented transport layer protocol in the Internet Protocol Suite

- Provides a reliable, ordered delivery mechanism
- TCP has more overhead compared to UDP due to its extensive features for reliable data delivery
  - Handshaking
  - Congestion control
  - Acknowledgment mechanisms
  - Error checking
  - Retransmission
  - Ordered delivery
- Operates by establishing a connection before data exchange (i.e., **connection-oriented**)
- Slower but more reliable than UDP

TCP is preferred for applications where data integrity and reliability are crucial (incl. protocols that carry documents and files)



TCP Basics (RFC 793 from 1981)

- Every TCP packet is given a **sequence number**
  - Ordering & noticing missing packets (will be retransmitted)
  - It is more secure when the initial sequence number is unpredictable (e.g. chosen randomly)

- TCP uses a **counter** that counts the number of bytes transmitted
  - E.g., a packet of size 2048 bytes and sequence number 6000 will be followed by a packet with a sequence number 8048

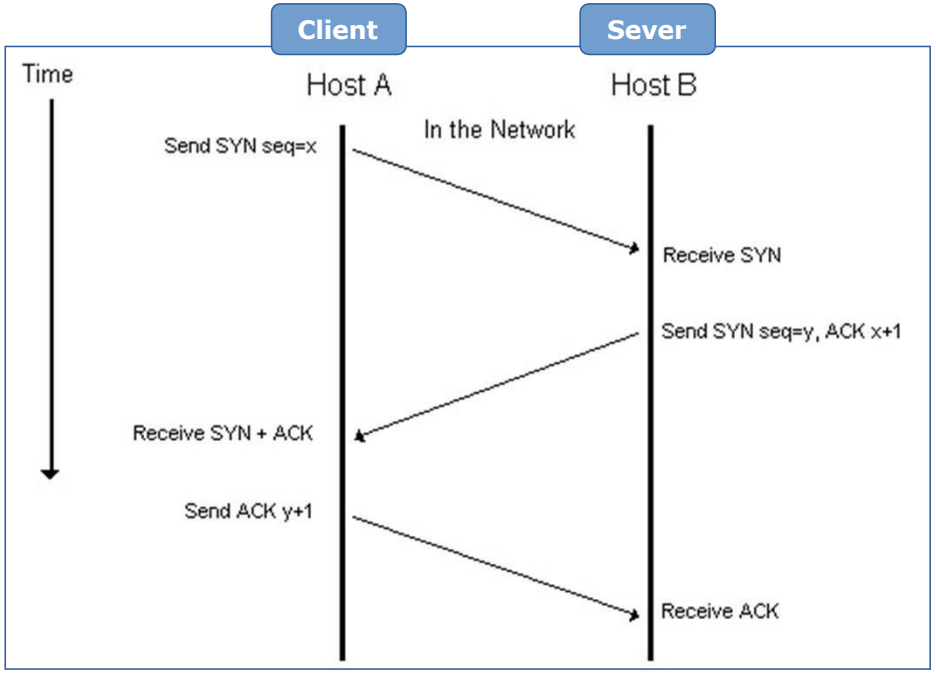
- TCP sends whole bursts of packets at a time before expecting a response
  - **TCP Window** is the amount of data that a sender is willing to send at once
  - **Flow Control:** The receiver (e.g., when its buffer is full) can regulate the window size (of the sender) thus slow or pause the connection
  - If TCP believes that packets are being dropped, it assumes that the network is becoming congested and reduces how much data it sends every second

Establishing a TCP connection costs three packets (3-way handshake)

SYN: "I want to talk; here is the packet sequence number I will be starting with."

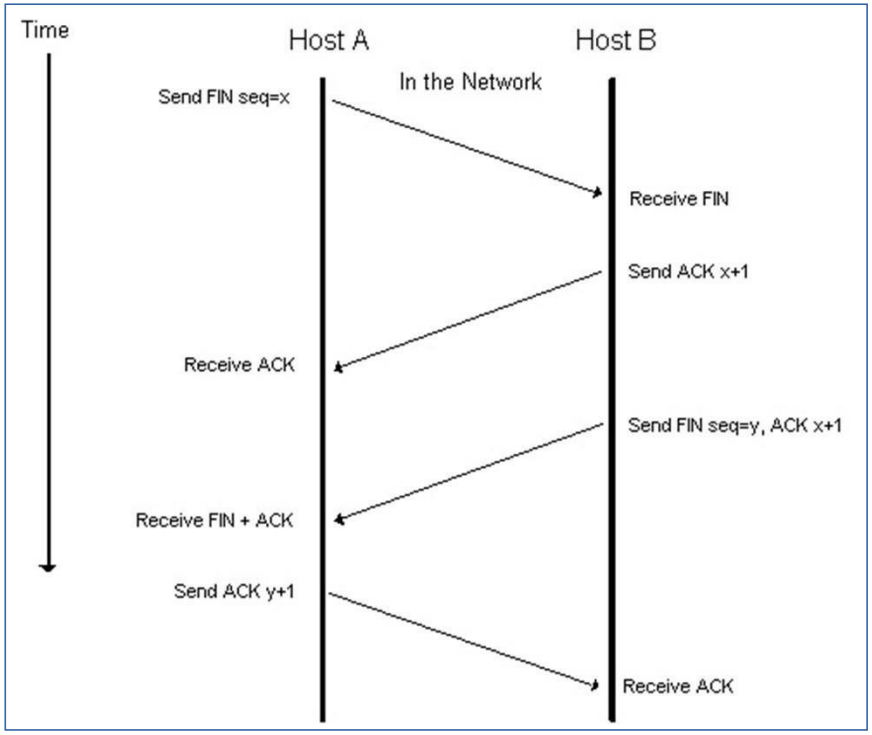
SYN-ACK: "Okay, here is the initial sequence number I will be using in my direction."

ACK: "Okay!"



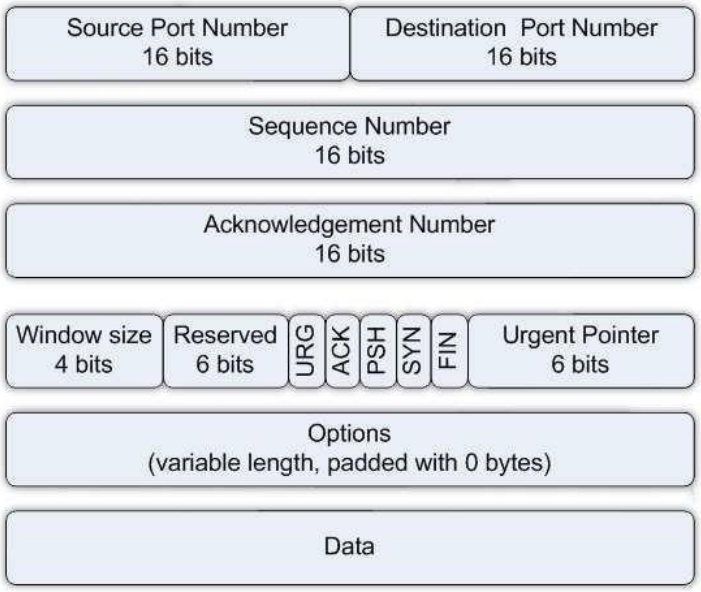
Source: https://condor.depaul.edu/jkristof/technotes/tcp.html

Shutting a TCP connection down costs three to four packets



Source: https://condor.depaul.edu/jkristof/technotes/tcp.html

The TCP header ensures reliability by including essential control information for managing data transmission



Source: https://voip-sip-sdk.com/p_7231-user-datagram-protocol.html

Simple TCP Server

```python
import socket

interface = 'localhost'
port = 9000

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((interface, port))
sock.listen(1)
print('Listening at', sock.getsockname())

while True:
  sc, sockname = sock.accept()
  print('We have accepted a connection from', sockname)
  print(' Socket name:', sc.getsockname())
  print(' Socket peer:', sc.getpeername())
  message = sc.recv(16)
  print(' Incoming sixteen-octet message:', repr(message))
  sc.sendall(b'Farewell, client')
  sc.close()
  print(' Reply sent, socket closed')
```

- **Passive (aka "listening") sockets** maintain the socket name at which the server is ready to receive data
  - No data send / receive
  - (IP, port)
- **Active (aka "connected") sockets** bound to one remote conversation partner
  - Send / receive
  - (local_IP, local_port, remote_IP, remote_port)

Simple TCP Client

- “connect()” is essential in TCP clients!
  - TCP connect() starts the 3-way handshake protocol
  - It (unlike UDP connect()) may fail! (e.g., the remote host might not answer or refuse the connection)

```
1  import socket
2
3  host = 'localhost'
4  port = 9000
5
6  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
7  sock.connect((host, port))
8  print('Client has been assigned socket name', sock.getsockname())
9  sock.sendall(b'Hi there, server')
10 reply = sock.recv(16)
11 print('The server said', repr(reply))
12 sock.close()
```

Server

```
Listening at ('127.0.0.1', 9000)
We have accepted a connection from ('127.0.0.1', 55735)
 Socket name: ('127.0.0.1', 9000)
 Socket peer: ('127.0.0.1', 55735)
 Incoming sixteen-octet message: b'Hi there, server'
 Reply sent, socket closed
We have accepted a connection from ('127.0.0.1', 55736)
 Socket name: ('127.0.0.1', 9000)
 Socket peer: ('127.0.0.1', 55736)
 Incoming sixteen-octet message: b'Hi there, server'
 Reply sent, socket closed
We have accepted a connection from ('127.0.0.1', 55737)
 Socket name: ('127.0.0.1', 9000)
 Socket peer: ('127.0.0.1', 55737)
 Incoming sixteen-octet message: b'Hi there, server'
 Reply sent, socket closed
```

Clients

```
Client has been assigned socket name ('127.0.0.1', 55735)
The server said b'Farewell, client'
```

```
Client has been assigned socket name ('127.0.0.1', 55736)
The server said b'Farewell, client'
```

```
Client has been assigned socket name ('127.0.0.1', 55737)
The server said b'Farewell, client'
```

When performing a TCP send(), there are three scenarios

- **The data is immediately accepted by the local system**
  - The network card is free
  - Or the system has room to copy the data to a temporary outgoing buffer so that send() immediately returns the length of data
  - The program can continue running
- **Block, pause the program until the data can be accepted**
  - The network card is busy and the outgoing data buffer is full
- **Part of the data can be immediately queued and the rest waits**
  - The outgoing buffers are not completely full
  - Part of the data can be immediately queued
  - send() completes immediately, returns the number of bytes accepted, and leaves the rest unprocessed

Put a send() call inside a loop that will keep trying to send the remaining data:

```
bytes_sent = 0
while bytes_sent < len(message):
  message_remaining = message[bytes_sent:]
  bytes_sent += s.send(message_remaining
```

OR

use “**sendall()**”

When performing a TCP recv(), there are three scenarios

- **No data is available**
  - recv() blocks, and the program pauses until data arrives

- **Plenty of data is available in the incoming buffer**
  - The app receives as many bytes as recv() is given permission to deliver

- **The buffer contains some waiting data less than what recv() is given permission to deliver**
  - Immediately returns the available of data

There is no built-in: recvall()

 Make your own!

Modified TCP Server

```python
import socket

interface = 'localhost'
port = 9000

######################

def recvall(sock, length):
  data = b''
  while len(data) < length:
    more = sock.recv(length - len(data))
    if not more:
       raise EOFError('was expecting %d bytes but only received'
                      ' %d bytes before the socket closed'
                      % (length, len(data)))
    data += more
  return data

######################

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((interface, port))
sock.listen(1)
print('Listening at', sock.getsockname())

while True:
  sc, sockname = sock.accept()
  print('We have accepted a connection from', sockname)
  print(' Socket name:', sc.getsockname())
  print(' Socket peer:', sc.getpeername())
  message = recvall(sc, 16)
  print(' Incoming sixteen-octet message:', repr(message))
  sc.sendall(b'Farewell, client')
  sc.close()
  print(' Reply sent, socket closed')
```

Modified TCP Client

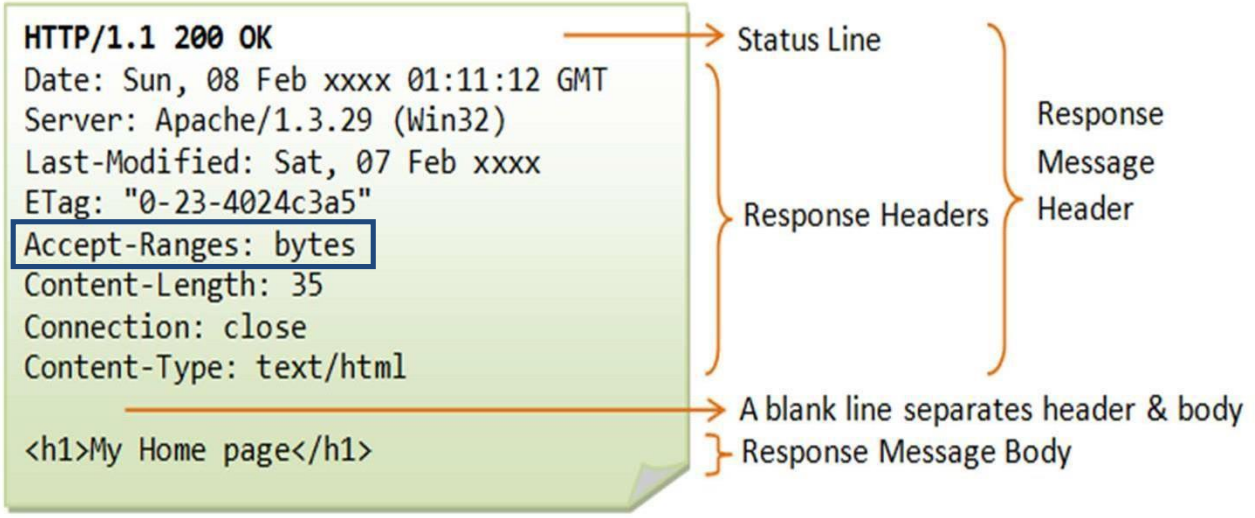
```python
import socket

host = 'localhost'
port = 9000

######################

def recvall(sock, length):
  data = b''
  while len(data) < length:
    more = sock.recv(length - len(data))
    if not more:
       raise EOFError('was expecting %d bytes but only received'
                      ' %d bytes before the socket closed'
                      % (length, len(data)))
    data += more
  return data

######################

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
print('Client has been assigned socket name', sock.getsockname())
sock.sendall(b'Hi there, server')
reply = recvall(sock, 16)
print('The server said', repr(reply))
sock.close()
```

A more complicated “recvall()”

- Most real-world programs have to read (or process) part of the message before it can guess how much more is coming!

- **Example:** HTTP Response

Try it!

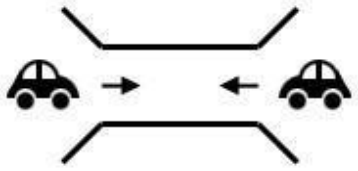


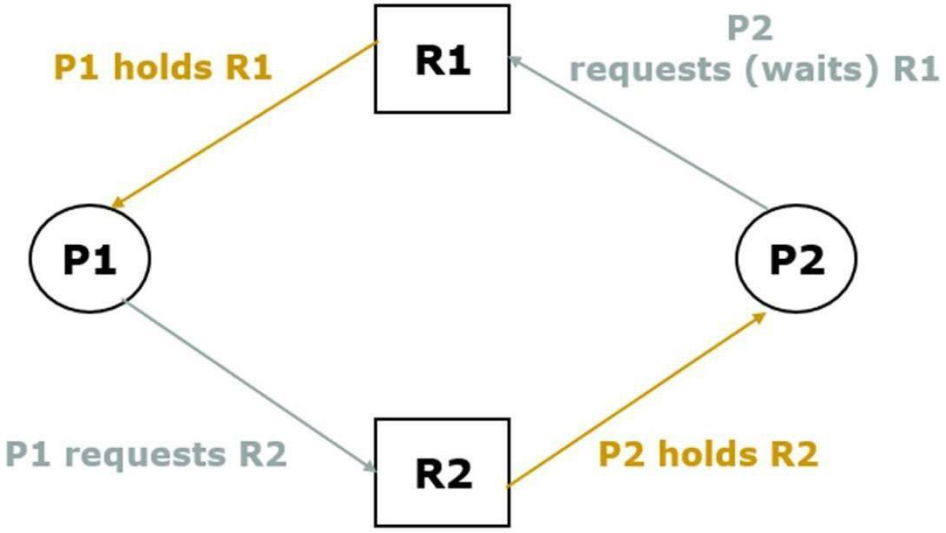
recv()

- TCP is a streaming protocol
  - A message has no boundaries
- recv() will return 0 only when the sender has closed the socket or explicitly called shutdown (SHUT_WR)
- Otherwise, it can return >= 1 (bytes)
  - It will block until it has at least one byte to return
- Three ways to determine when a complete message is received:
  - Use fixed length messages
  - Send a fixed-length message containing the length, followed by a variable-length message
  - After each message, send a unique / special termination message

Deadlock

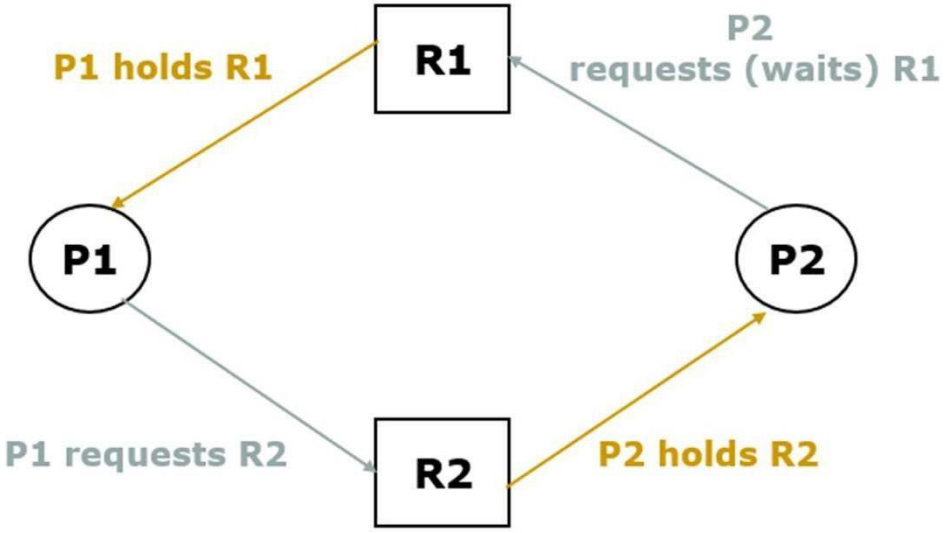
- Two or more processes, sharing limited resources, waiting each other forever due to poor planning
- In other words, a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set





Deadlock may occur when using TCP!

- **Resource:** Buffers
  - Store incoming packet until an app is ready to read it
  - Store outgoing packet till the network hardware is ready to transmit it
  - Limited in size!
- No problem if each app (client or server) reads the other app's complete message before sending data in the other direction
- When it becomes problematic?



Modified TCP Server

```python
import socket, sys

interface = 'localhost'
port = 9000

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((interface, port))
sock.listen(1)
print('Listening at', sock.getsockname())

while True:
  sc, sockname = sock.accept()
  print('Processing up to 1024 bytes at a time from', sockname)
  n = 0
  while True:
    data = sc.recv(1024)
    if not data:
       break
    output = data.decode('ascii').upper().encode('ascii')
    sc.sendall(output) # send it back uppercase
    n += len(data)
    print('\r %d bytes processed so far' % (n,), end=' ')
    sys.stdout.flush()
  print()
  sc.close()
  print(' Socket closed')
```

Modified TCP Client

```python
import socket, sys

host = 'localhost'
port = 9000
bytecount = 32

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
bytecount = (bytecount + 15) // 16 * 16 # round up to a multiple of 16
message = b'capitalize this!' # 16-byte message to repeat over and over
print('Sending', bytecount, 'bytes of data, in chunks of 16 bytes')
sock.connect((host, port))

sent = 0

while sent < bytecount:
  sock.sendall(message)
  sent += len(message)
  print('\r %d bytes sent' % (sent,), end=' ')
  sys.stdout.flush()

print()
sock.shutdown(socket.SHUT_WR)
print('Receiving all the data the server sends back')

received = 0
while True:
  data = sock.recv(42)
  if not received:
     print(' The first data received says', repr(data))
  if not data:
     break
  received += len(data)
  print('\r %d bytes received' % (received,), end=' ')

print()
sock.close()
```

Server

```python
import socket, sys

interface = 'localhost'
port = 9000

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((interface, port))
sock.listen(1)
print('Listening at', sock.getsockname())

while True:
  sc, sockname = sock.accept()
  print('Processing up to 1024 bytes at a time from', sockname)
  n = 0
  while True:
    data = sc.recv(1024)
    if not data:
       break
    output = data.decode('ascii').upper().encode('ascii')
    sc.sendall(output) # send it back uppercase
    n += len(data)
    print('\r %d bytes processed so far' % (n,), end=' ')
    sys.stdout.flush()
  print()
  sc.close()
  print(' Socket closed')
```

Client

```python
import socket, sys

host = 'localhost'
port = 9000
bytecount = 32

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
bytecount = (bytecount + 15) // 16 * 16 # round up to a multiple of 16
message = b'capitalize this!' # 16-byte message to repeat over and over
print('Sending', bytecount, 'bytes of data, in chunks of 16 bytes')
sock.connect((host, port))

sent = 0

while sent < bytecount:
  sock.sendall(message)
  sent += len(message)
  print('\r %d bytes sent' % (sent,), end=' ')
  sys.stdout.flush()

print()
sock.shutdown(socket.SHUT_WR)
print('Receiving all the data the server sends back')

received = 0
while True:
  data = sock.recv(42)
  if not received:
     print(' The first data received says', repr(data))
  if not data:
     break
  received += len(data)
  print('\r %d bytes received' % (received,), end=' ')

print()
sock.close()
```

Server

```
Listening at ('127.0.0.1', 9000)
Processing up to 1024 bytes at a time from ('127.0.0.1', 63198)
 32 bytes processed so far
 Socket closed
Processing up to 1024 bytes at a time from ('127.0.0.1', 63199)
 32 bytes processed so far
 Socket closed
Processing up to 1024 bytes at a time from ('127.0.0.1', 63200)
 32 bytes processed so far
 Socket closed
```

Clients

```
Sending 32 bytes of data, in chunks of 16 bytes
 32 bytes sent
Receiving all the data the server sends back
 The first data received says b'CAPITALIZE THIS!CAPITALIZE THIS!'
 32 bytes received
```

```
Sending 32 bytes of data, in chunks of 16 bytes
 32 bytes sent
Receiving all the data the server sends back
 The first data received says b'CAPITALIZE THIS!CAPITALIZE THIS!'
 32 bytes received
```

```
Sending 32 bytes of data, in chunks of 16 bytes
 32 bytes sent
Receiving all the data the server sends back
 The first data received says b'CAPITALIZE THIS!CAPITALIZE THIS!'
 32 bytes received
```

Server

```
Listening at ('192.168.1.103', 9000)
Processing up to 1024 bytes at a time from ('192.168.1.103', 49778)
  128 bytes processed so far
  Socket closed
```

Server

```
Processing up to 1024 bytes at a time from ('192.168.1.103', 49782)
  6879248 bytes processed so far
```

Clients

```
Sending 128 bytes of data, in chunks of 16 bytes
  128 bytes sent
Receiving all the data the server sends back
  The first data received says b'CAPITALIZE THIS!CAPITALIZE THIS!CAPITALIZE'
  128 bytes received
```

Clients

```
Sending 1288729812752 bytes of data, in chunks of 16 bytes
  12598640 bytes sent
```

What happened?

- **Why have both client and server been frozen?**
  - The server's output buffer and the client's input buffer became full
  - TCP has used its window adjustment protocol to signal this fact and stop the socket from sending additional data that would have to be discarded and later resent

- **Why has this resulted in deadlock?**
  - The client sends each block with sendall()
  - Then the server accepts it with recv(), processes it, and transmits its capitalized version back out with another sendall() call
  - And then what? Nothing!
  - The client is never running any recv() calls (not while it still has data to send) so more and more data backs up until the operating system buffers are not willing to accept any more

How are clients and servers supposed to process large amounts of data without entering a deadlock?

- They can use socket options to turn off blocking so that calls like send() and recv() return immediately if they find that they cannot send / receive any data yet

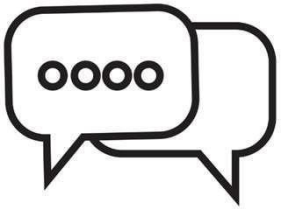
- Programs can use one of several techniques to process data from several inputs at a time, either by splitting into separate threads or processes (one tasked with sending data into a socket, perhaps, and another tasked with reading data back out)

- Or by running operating system calls such as select() or poll() that let them wait on busy outgoing and incoming sockets at the same time and respond to whichever is ready

Recap

- The TCP-powered “stream” socket involves retransmitting lost packets, reordering the ones that arrive out of sequence, and splitting large data streams into optimally sized packets
- A program that wants to accept incoming TCP connections needs to **bind()** to a port, run **listen()** on the socket, and then go into a loop that runs **accept()** to receive a new socket for each incoming connection
- To connect to existing server ports, clients need only create a socket and **connect()** to an address
- Data is actually sent and received with **send()** and **recv()**
- Deadlock can occur if two peers are written such that the socket fills with more and more data that never gets read
  - Eventually, one direction will no longer be able to send() and might hang forever waiting

Experiment with crafting more advanced applications using the concepts we have covered!



- Develop a simple command-line chat application over TCP
- Each user's application will act both as a client and a server
- Each user have a command-line interface where he can type messages to send to other users (individual and broadcast) and view incoming messages
- Include features such as:
  - User authentication
  - Message encryption
  - Message history
  - Online User List
  - Offline Messaging
  - File Transfer

Source

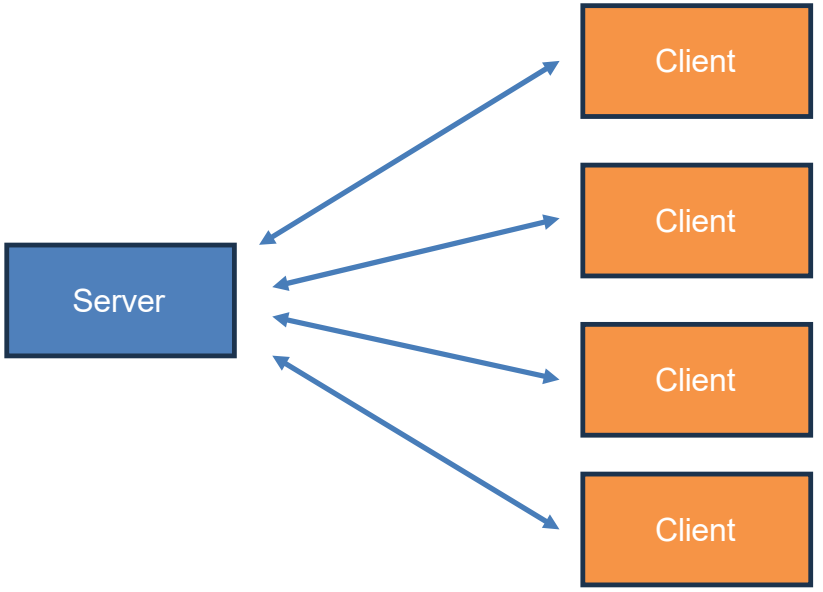
CHAPTER 3

TCP

Questions?

Concurrency

- Servers usually need to handle multiple clients
- The server must always be ready to accept new connections
  - A new connection makes a new socket
- Each client gets its own socket on server
  - Each may be performing different tasks



One solution is to handle each client by a separate thread

Single-threaded

```
2  import socket
3
4  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5  s.bind(("",9000))
6  s.listen(5)
7
8  while True:
9    c,a = s.accept()
10   print "Received connection from", a
11   c.recv(1000)
12   c.send("Hello " + a[0])
13   c.close()
14
```

Multi-threaded

```
2  import threading
3  import socket
4
5  def handle_client(sock,addr):
6    print "Received connection from", addr
7    sock.recv(1000)
8    sock.send("Hello " + addr[0])
9    sock.close()
10   return
11
12 s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
13 s.bind(("",9000))
14 s.listen(5)
15
16 while True:
17   c,a = s.accept()
18   threading.Thread(handle_client(c,a))
19
```

Network Programming

5) HTTP Clients

Dr. Hassan Almazini

A – Target Population: For students of Second Year Technological Institute of Basra
Dep. Of Computer network and software technologies
5 / B – Rationale: HTTP is the foundation of data communication for the World Wide Web. Understanding how HTTP clients work is essential for building web-enabled applications and services.
5 / C – Central Idea:

1. HTTP request-response model
2. HTTP methods and status codes
3. Building HTTP clients in Python
4. Caching, cookies, and authentication

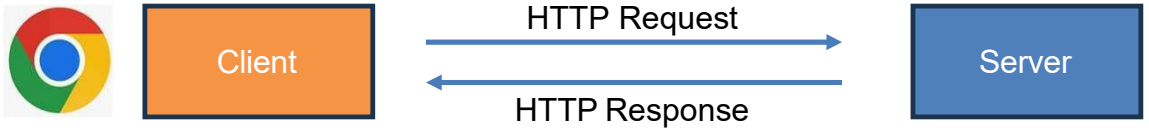
5 / D – Performance Objectives: After studying this unit, the student will be able to:

1. Explain the HTTP client-server communication model.
2. Use HTTP request methods such as GET, POST, and PUT.
3. Interpret HTTP status codes and headers.
4. Build Python-based HTTP clients using urllib and Requests libraries.

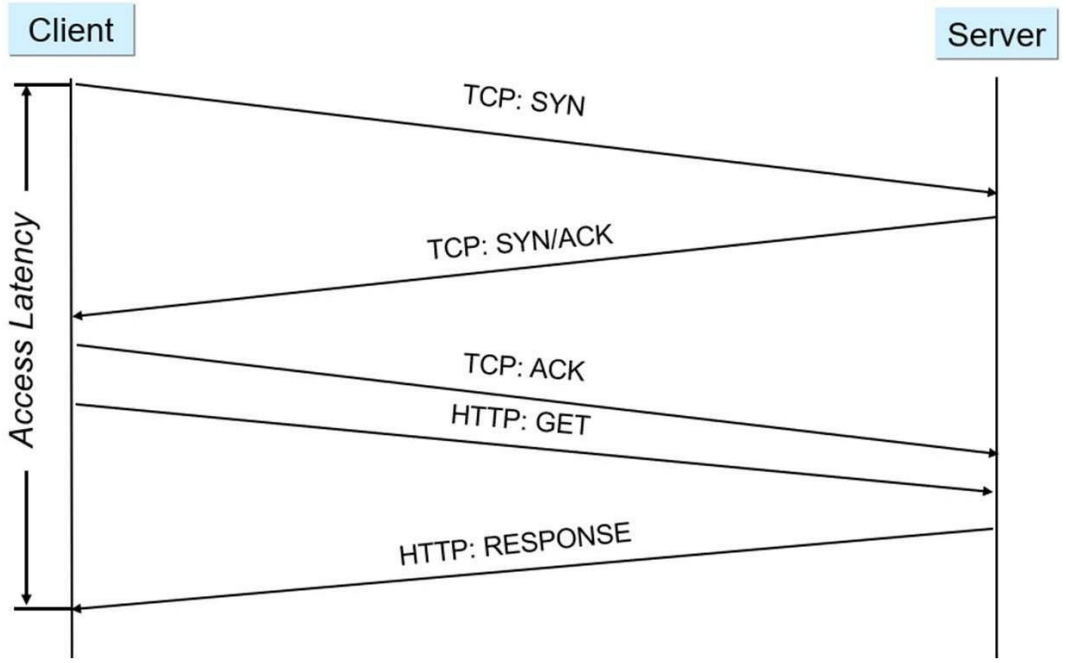
HTTP (Hypertext Transfer Protocol) is an application protocol for distributed, collaborative, hypermedia information systems

- HTTP is the foundation of web communication, and knowledge of it is crucial for understanding how web servers and clients interact

- HTTP operates as a request-response protocol in the client-server computing model
  - Clients, typically web browsers, initiate requests for resources
  - Servers, like web servers, process these requests and deliver the requested resources



While HTTP itself doesn't have a strict dependency on a specific transport protocol, it traditionally and predominantly operates over TCP



HTTP programming opens up various possibilities for building web applications and services

- **Web Scraping Tools:** Build tools to extract data from websites for analysis or storage
  - Enable tasks like monitoring competitor prices, gathering research data, or aggregating information from multiple sources
- **Web Servers:** Develop custom web servers for hosting websites or web applications
- **IoT Applications:** Create web interfaces to interact with IoT devices or sensors
  - Control, monitor, and remotely manage IoT devices
  - Process sensor data, send commands to devices, and automate tasks based on predefined conditions (e.g., adjusting temperature based on occupancy)
  - Enhance security by integrating authentication mechanisms and encrypted channels
- **Webhooks:** Implement endpoints to receive and process HTTP requests triggered by external events

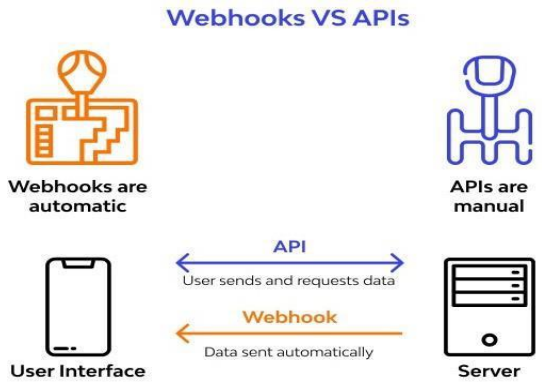
A webhook is a mechanism for triggering actions in one system based on events that occur in another system using HTTP requests

- It allows real-time communication between different applications or services over the web

- **Use Cases:**
  - **Real-time Notifications:** Deliver real-time notifications to users or systems

  - **Data Synchronization:** For instance, an e-commerce platform can use webhooks to notify inventory management systems about product purchases or stock updates

  - **Integration:** For instance, a payment gateway can use webhooks to notify an e-commerce platform about payments



https://www.wallarm.com/what/a-simple-explanation-of-what-a-webhook-is

https://botpenguin.com/glossary/webhooks

HTTP request methods define the action to be performed on a resource

- **GET:** Retrieves data from a server (e.g., fetching web pages, loading images)
  - Parameters attached to the ULR; not to be used when dealing with sensitive data
  - Requests can be cached

google.com/search?q=http+client+programming+in+python

  - **HEAD:** Same as GET but returns only HTTP headers (no body)
- **POST:** Submits data to a server (e.g., creating new posts, submitting login forms)
  - The results of a POST are never cached
- **PUT:** Updates data on a server (e.g., updating user profiles, modifying product information)
- **DELETE:** Deletes data from a server (e.g., removing user accounts, deleting tasks)
- **PATCH:** Partially updates data on a server (e.g., updating specific fields in a user profile)

The “urllib” module allows clients to connect to HTTP(s) and FTP

- Returns a file-like object
- Read from it to get downloaded data

```
>>> import urllib
>>> import urllib.request
>>> result = urllib.request.urlopen("https://ppu.edu/p/en")
>>> data = result.read()
>>> print (data)
```

```
b'<!DOCTYPE html>\n<html lang="en" dir="ltr" prefix="content: http://purl.org/rss/1.0/modules/content/ dc: http://purl.org/dc/terms/ foaf: http://xmlns.com/foaf/0.1/ og: http://ogp.me/ns# rdfs: http://www.w3.o
rg/2000/01/rdf-schema# sioc: http://rdfs.org/sioc/ns# sioct: http://rdfs.org/sioc/types# skos: http://www.w3.org/2004/02/skos/core# xsd: http://www.w3.org/2001/XMLSchema#" class="ltr"\n>\n<head>\n  <link rel="
profile" href="http://www.w3.org/1999/xhtml/vocab" />\n  <meta name="viewport" content="width=device-width, initial-scale=1.0">\n  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />\n  <titl
e>Palestine Polytechnic University | Towards Science,Technology,and Innovation Entrepreneurial University</title>\n  <style>\n@import url("https://ppu.edu/p/modules/system/system.base.css?s9rxjl");\n</style>\n
<style media="screen">\n@import url("https://ppu.edu/p/sites/all/libraries/shadowbox/shadowbox.css?s9rxjl");\n</style>\n<style media="print">#sb-container{position:relative;}#sb-overlay{display:none;}#sb-wrapp
er{position:relative;top:0;left:0;}#sb-loading{display:none;}\n</style>\n<style>\n@import url("https://ppu.edu/p/sites/all/modules/views_slideshow/views_slideshow.css?s9rxjl");\n</style>\n<style>\n@import url(
"https://ppu.edu/p/sites/all/modules/calendar/css/calendar_multiday.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/calendar_tooltips/calendar_tooltips.css?s9rxjl");\n@import url("https://ppu.e
du/p/sites/all/modules/date/date_api/date.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/date/date_popup/themes/datepicker.1.7.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/d
ate/date_repeat_field/date_repeat_field.css?s9rxjl");\n@import url("https://ppu.edu/p/modules/field/theme/field.css?s9rxjl");\n@import url("https://ppu.edu/p/modules/node/node.css?s9rxjl");\n@import url("https
://ppu.edu/p/sites/all/modules/views/css/views.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/back_to_top/css/back_to_top.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/ckedit
or/css/ckeditor.css?s9rxjl");\n</style>\n<style>\n@import url("https://ppu.edu/p/sites/all/modules/ctools/css/ctools.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/lightbox2/css/lightbox.css?s
9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/shorten/shortener/shortener.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/libraries/superfish/css/superfish.css?s9rxjl");\n@import url("http
s://ppu.edu/p/sites/all/libraries/superfish/css/superfish-vertical.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/libraries/superfish/css/superfish-navbar.css?s9rxjl");\n@import url("https://ppu.edu/p
/modules/locale/locale.css?s9rxjl");\n@import url("https://ppu.edu/p/sites/all/modules/views_slideshow/contrib/views_slideshow_cycle/views_slideshow_cycle.css?s9rxjl");\n</style>\n<style>\n@import url("https:/
/ppu.edu/p/sites/all/themes/ppu2018/css/style.css?s9rxjl");\n</style>\n  <!-- HTML5 element support for IE6-8 -->\n  <!--[if lt IE 9]>\n    <script src="https://cdn.jsdelivr.net/html5shiv/3.7.3/html5shiv-print
```

HTTP status codes indicate the outcome of a request

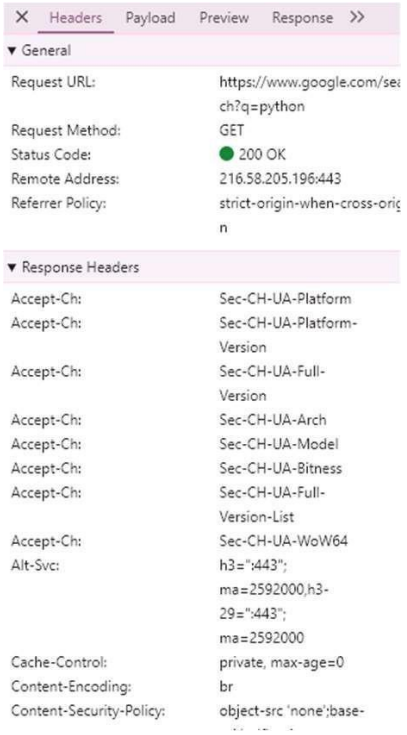
- 200 OK
- 301 Moved Permanently
- 304 Not Modified
- 307 Temporary Redirect
- 400 Bad Request
- 401 Unauthorized
- 403 Forbidden
- 404 Not Found
- 405 Method Not Allowed
- 500 Server Error

```
>>> print (result.code)
200
```

```
>>> if result.code == 200:
...     print("success")
... elif result.code == 404:
...     print("page not found")
... else: print("another code")
...
success
```

https://www.google.com/search?q=python

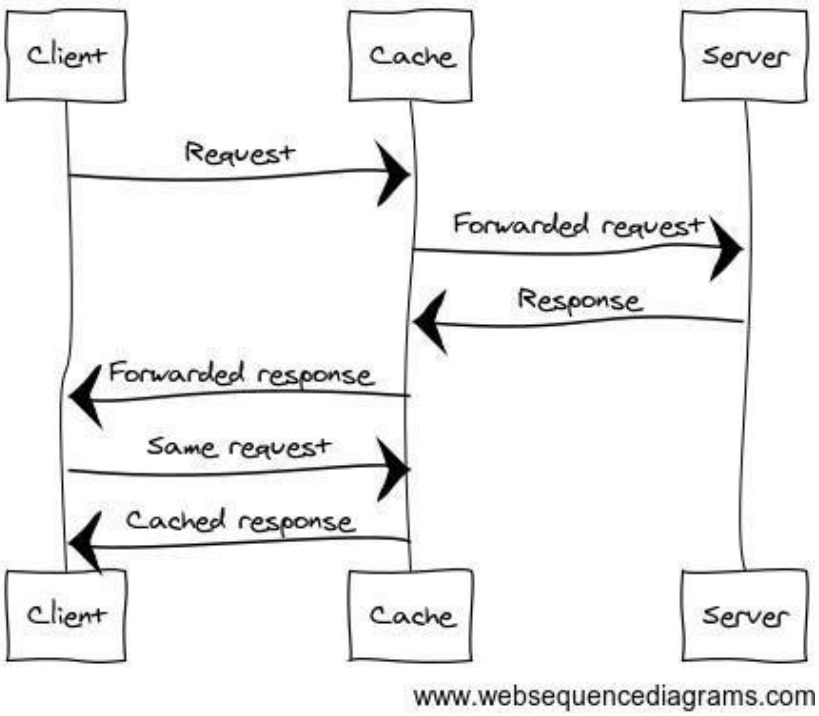

```
>>> import urllib
>>> import urllib.request
>>> import urllib.parse
>>> base_url = "https://www.google.com/search"
>>> params = {'q': 'python'}
>>> encoded_params = urllib.parse.urlencode(params)
>>> full_url = f"{base_url}?{encoded_params}"
>>> request = urllib.request.Request(full_url, headers={'User-Agent': 'Mozilla/5.0'})
>>> response = urllib.request.urlopen(request)
>>> html = response.read()
>>> print(html)
```



HTTP Response Headers

```
>>> headers = response.info()
>>> headers.keys()
['Content-Type', 'Date', 'Expires', 'Cache-Control', 'Content-Security-Policy', 'P3
P', 'Server', 'X-XSS-Protection', 'X-Frame-Options', 'Set-Cookie', 'Set-Cookie', 'S
et-Cookie', 'Alt-Svc', 'Accept-Ranges', 'Vary', 'Connection', 'Transfer-Encoding']
>>> headers['connection']
'close'
>>> print(headers)
Content-Type: text/html; charset=UTF-8
Date: Sun, 03 Mar 2024 19:23:50 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Security-Policy: object-src 'none';base-uri 'self';script-src 'nonce-CI4KmN
CPaYpUqTKEDkGnSQ' 'strict-dynamic' 'report-sample' 'unsafe-eval' 'unsafe-inline' ht
tps: http:;report-uri https://csp.withgoogle.com/csp/gws/xsrp
P3P: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
X-XSS-Protection: 0
X-Frame-Options: SAMEORIGIN
Set-Cookie: 1P_JAR=2024-03-03-19; expires=Tue, 02-Apr-2024 19:23:50 GMT; path=/; do
main=.google.com; Secure
Set-Cookie: AEC=Ae3NU9NPik49fBifw9hlqhxB94elfEbBCOhYSruoxhy30XI2gNwisjICA5E; expire
s=Fri, 30-Aug-2024 19:23:50 GMT; path=/; domain=.google.com; Secure; HttpOnly; Same
Site=lax
Set-Cookie: NID=512=G3QAiGpQshJx5YeoCWkrvfD-gfPedh70iK0-NKfSHl-beDg9E79Cn4q0etvq0E1
prn3mN6t4Qa1HwY8qA8wFXu3I8DwpXhZswJSlq7NOQEu8bFzLBcUxEfpQq0EY8F1Fh_L8hZ_63x4QFndCyd
Tg8mOOu2Oo5YR-hWLLIzoPa6I; expires=Mon, 02-Sep-2024 19:23:50 GMT; path=/; domain=.g
oogle.com; HttpOnly
Alt-Svc: h3=":443"; ma=2592000,h3-29=":443"; ma=2592000
Accept-Ranges: none
Vary: Accept-Encoding
Connection: close
Transfer-Encoding: chunked
```

Improve Performance With Caching



Source: https://gnugat.github.io/2015/11/26/http-cache.html

How to make sure that the cached version of a web object is up-to-date?

One solution: Cache uses **Conditional GET** to check whether a cached object is stale

- First request from cache to origin server (due to a cache miss)

```
GET index.html HTTP/1.0
User-Agent: Mozilla/4.0
Accept: text/html image/gif, image/jpeg
```

- Response from origin server to cache

```
HTTP/1.0 200 OK
Date: … 04:29:01 GMT
Server: Apache/1.3.12 (Unix)
Last-Modified: 28 Feb 2024 01:10:42 GMT
Content-Type: text/html <CR/LF>
<data>
```

Conditional Get (Cont.)

- Conditional GET from cache to origin server (due to a cache hit)

```
GET index.html HTTP/1.0
User-Agent: Mozilla/4.0
Accept: text/html image/gif, image/jpeg
If-modified-since: 16 Feb 2024 01:10:42
```

- Response from origin server to cache

```
HTTP/1.0 304 Not Modified
Date: …
Server: Apache/1.3.12 (Unix) <CR/LF>
(empty entity body)
```

Setting expiration time is another solution

- Strong consistency and a freshness guaranty can be provided by using a conditional GET for every cache hit
- However, this results in unnecessary delay and bandwidth consumption for requests yield not modified
- Origin server can set an expiration time for web pages, which will be included in the entity header of HTTP responses:
  - A web page will not be modified before it expires
  - There is no need for consistency checks (e.g. using conditional GET) ahead the expiration time

HTTP-Specific Considerations

- HTTP embeds cache-ability information in the HTTP header
  - Cache control directives (e.g. Cache-Control: no-cache)
- HTTP defines rules indicating which HTTP responses are cacheable
- A response is cacheable only if request method and headers, and response status and headers all indicate so
  - E.g., responses to PUT, DELETE, OPTIONS are not cacheable
  - E.g., response to POST is not cacheable unless indicated otherwise
- Some responses include object-specific information from the origin server that may preclude caching of the message
- A well-behaved (but not all) Web cache must abide by the constraints imposed by HTTP!

Cache Control Directives in HTTP/1.1

- Cache-ability of an object can be controlled using the Cache-Control header in an HTTP request or response

- Some of the directives in requests:
  - **no-cache:** client forces freshness check with origin server
  - **no-store:** do not store any portion of this request or response
  - **max-age=n:** client is prepared to accept object without freshness check if age is < n seconds

- Some of the directives in responses:
  - **public:** Object is cacheable by client and network caches
  - **private:** Object can be cached only by the client
  - **no-cache:** Object must not be cached
  - **max-age=n:** Server imposes freshness check after n seconds.

Cache Control Directives: Examples

- **HTTP Request:**

```
GET index.html HTTP/1.0
User-Agent: Mozilla/4.0
Cache-Control: no-cache
Accept: text/html image/gif, image/jpeg
```

- **HTTP Response:**

```
HTTP/1.0 200 OK
Date: 16 Feb 2024 04:29:01 GMT
Server: Apache/1.3.12 (Unix)
Last-Modified: 16 Feb 2024 01:10:42 GMT
Cache-Control: private
Content-Type: text/html <CR/LF>
<data>
```

Authentication

- Determining whether a request really comes from someone authorized to make it
- HTTP request often needs to carry built-in proof as to the identity of the machine or person making it
- The error code 401 Not Authorized is used by servers that want to signal formally, through HTTP, either that:
  - They cannot authenticate your identity
  - Or that the identity is fine but is not one authorized to view this particular resource.

```
>>> r = requests.get('http://example.com/api/', auth=('myname','mypass'))
```

Or prepare a **Requests Session** for authentication to avoid having to repeat it with every request

```
>>> s = requests.Session()
>>> s.auth = 'myname', 'mypass'
>>> s.get('http://.......')
```

Cookies

- Cookie is a message given (as part of a successful HTTP response) by a Web server to a Web client (i.e. browser)

- **Purposes:**
  - Identify users
  - Prepare customized Web pages
  - Save login information
  - …

- **Types:** session & persistent

- **Parameters:** name, value, expiration date, domain (which the cookie is valid for), need for a secure connection

Cookies

- Access a cookie contained in a response

```
>>> r = requests.get('http://www.facebook.com')
>>> print(r.cookies)
<<class 'requests.cookies.RequestsCookieJar'>[<Cookie fr=0MQt1RCPZa4v27HCJ..BX6Y3i.pb.AAA.0.0.BX6Y3i.AWWqlv3p for .face
book.com/>]>
>>> print(r.cookies['fr'])
0MQt1RCPZa4v27HCJ..BX6Y3i.pb.AAA.0.0.BX6Y3i.AWWqlv3p
>>>
```

- Send a cookie in a HTTP request

```
>>> my_cookie = dict(fr='0MQt1RCPZa4v27HCJ..BX6Y3i.pb.AAA.0.0.BX6Y3i.AWWqlv3p')
>>> r = requests.get('http://www.facebook.com', cookies = my_cookie)
```

Experiment with crafting more advanced applications using the concepts we have covered!

Develop an HTTP client in Python using libraries like Requests or urllib

- Send HTTP requests to web pages
- Extract data (text, links, or images) from the web pages using libraries like BeautifulSoup or Scrapy
- Create a simple interface, either command line or GUI, where users can input the URL of the page they want to scrape
- Store the data in computer's memory, a file, or a database

Source

CHAPTER 9

HTTP Clients

Questions?

Network Programming

6) HTTP Servers

Dr. Hassan Almazini

A – Target Population: For students of Second Year Technological Institute of Basra
Dep. Of Computer network and software technologies
6 / B – Rationale: Building a server-side application is essential for understanding how web systems handle incoming client requests and generate responses. This unit introduces lightweight HTTP server development using Python.
6 / C – Central Idea:

1. Introduction to HTTP servers
2. Static and dynamic content
3. CGI scripting in Python
4. Building and testing HTTP servers

6 / D – Performance Objectives: After studying this unit, the student will be able to:

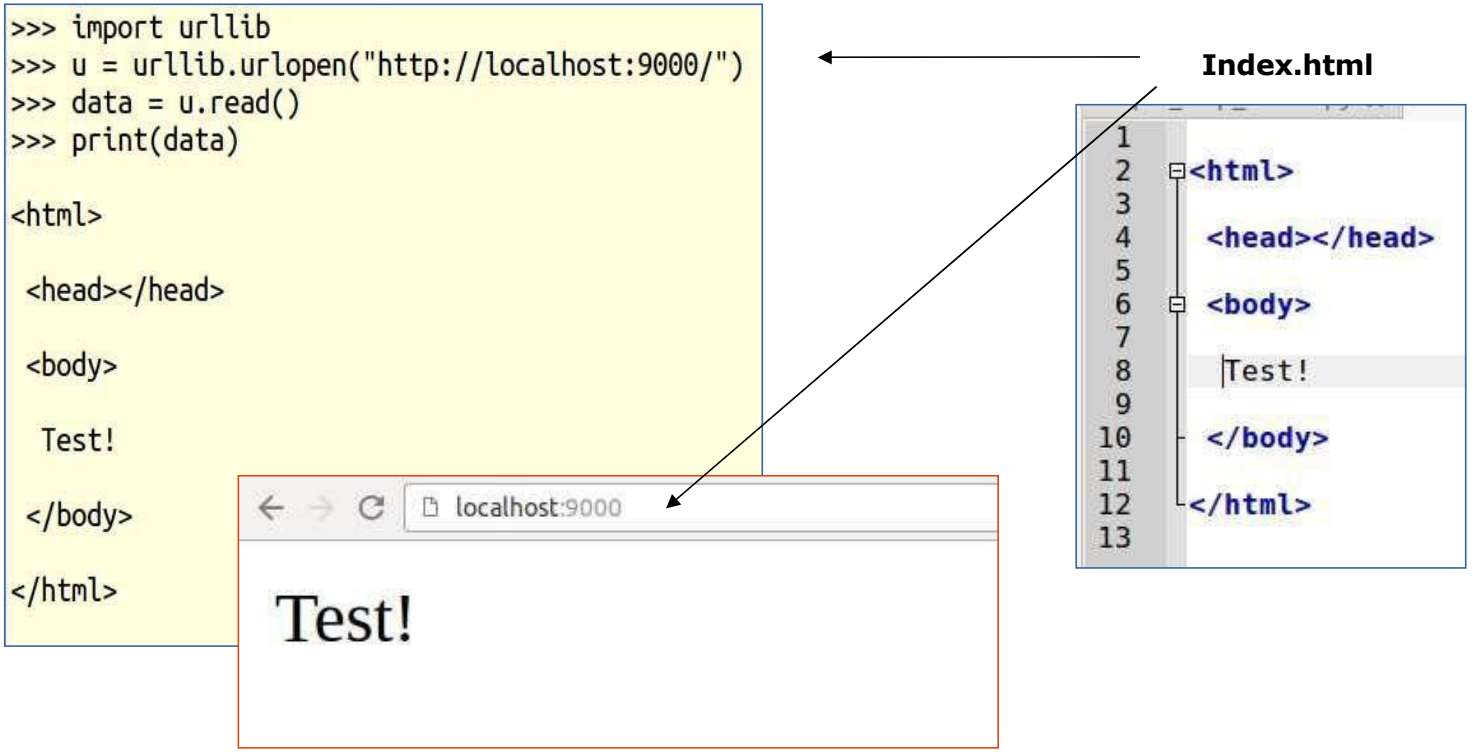
1. Understand the purpose and functionality of HTTP servers.
2. Serve static and dynamic content using Python.
3. Use Common Gateway Interface (CGI) for server-side scripting.
4. Develop, deploy, and test simple HTTP servers.

Python scripts used on the server to generate content (static & dynamic), manage content, and send content back to clients

- Python has libraries that implement simple self-contained web servers
  - Useful mainly for testing or where you don't want to install a “large” web server (e.g. Apache)
  - Not high performance in general (but might be "good enough")

```
1
2  from BaseHTTPServer import HTTPServer
3  from SimpleHTTPServer import SimpleHTTPRequestHandler
4  import os
5
6  os.chdir("/home/hani/html")
7  serv = HTTPServer(("",9000),SimpleHTTPRequestHandler)
8  serv.serve_forever()
```

Run a HTTP client against the server

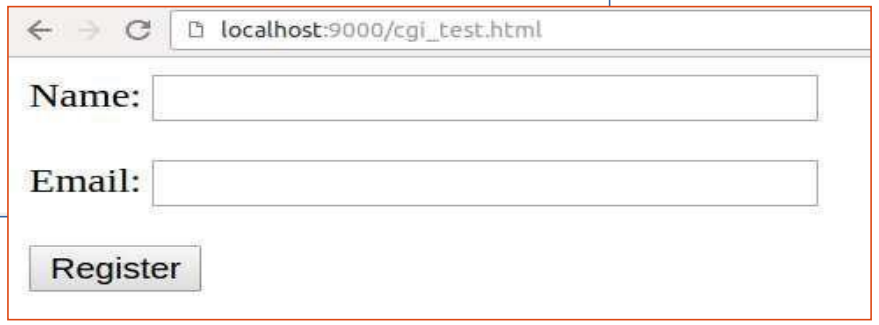


Common Gateway Interface (CGI) is a common protocol used by web servers to run server-side scripts to create dynamic content

```
1
2   from BaseHTTPServer import HTTPServer
3   from CGIHTTPServer import CGIHTTPRequestHandler
4   import os
5
6   os.chdir("/home/hani/html")
7   serv = HTTPServer(("",9000),CGIHTTPRequestHandler)
8   serv.serve_forever()
9
```

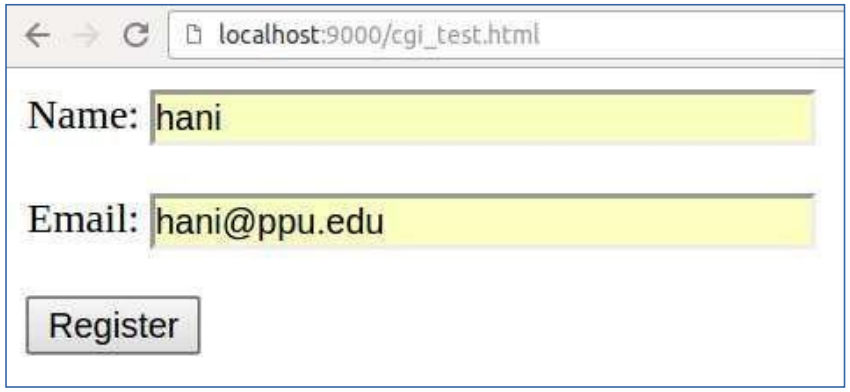
cgi_test.html

```
1
2  <html>
3
4   <head></head>
5
6  <body>
7
8  <form action="cgi-bin/register.py" method="POST">
9   <br>
10  Name: <input type="text" name="name" size="20">
11  <br><br>
12  Email: <input type="text" name="email" size="20">
13  <br><br>
14  <input type="submit" name="submit-button" value="Register">
15
16 </form>
17
18 </body>
19
20 </html>
21
```



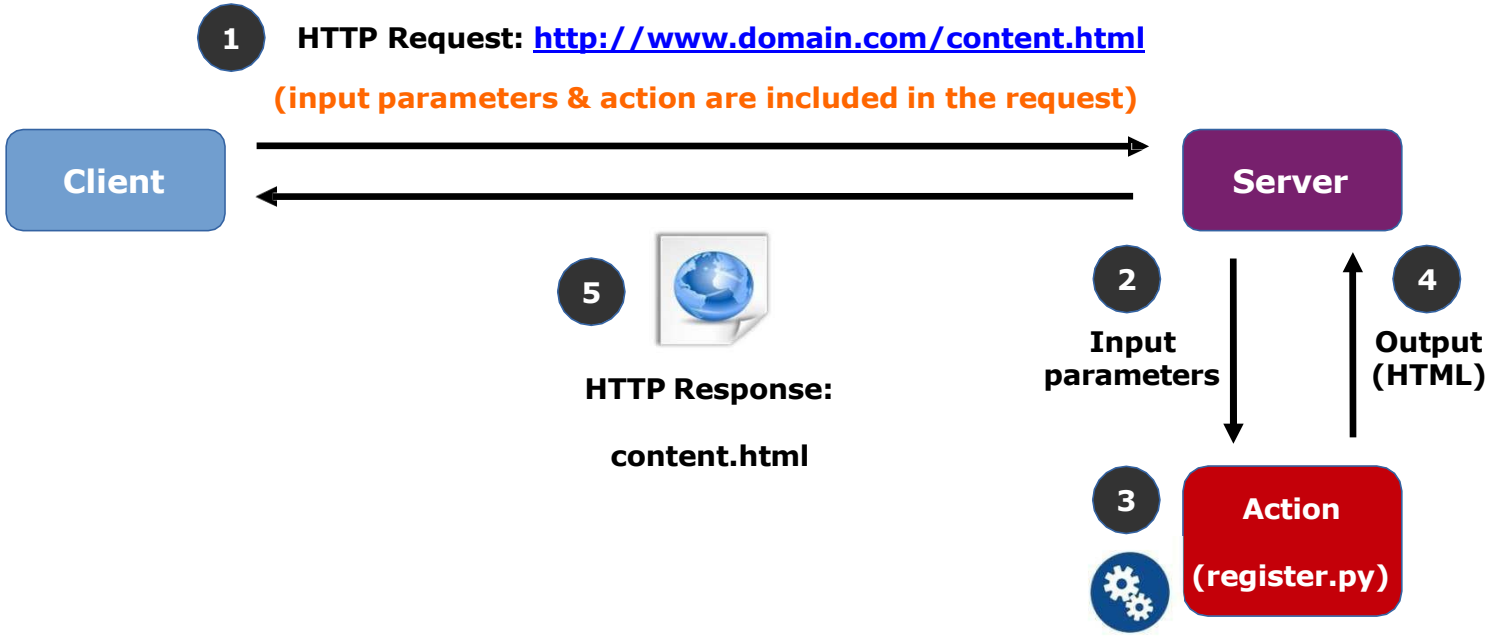
register.py

```
1
2  #!/usr/bin/python
3
4  import cgi, sys
5
6  sys.stderr = sys.stdout
7
8  # Read field values
9  form = cgi.FieldStorage()
10 name = form.getvalue('name')
11 email = form.getvalue('email')
12
13 # Output a response!
14 print "Content-type: text/html"
15 print
16 print """
17 <html>
18 <head><title>Test!!!</title></head>
19 <body>
20 <br>
21 Your name: %s
22 <p>
23 Your email: %s
24 </body>
25 </html>
26 """ %(name, email)
```


localhost:9000/cgi_test.html
Name: hani
Email: hani@ppu.edu
Register


localhost:9000/cgi-bin/register.py
Your name: hani
Your email: hani@ppu.edu

Put it all together



Source

CHAPTER 10

HTTP Servers

Recommended Self-Reading

CHAPTER 11

The World Wide Web

Questions?

Network Programming

7) Messaging and Message Queues

Dr. Hassan Almazini

A – Target Population: For students of Second Year Technological Institute of Basra
Dep. Of Computer network and software technologies
7 / **B – Rationale:** Scalability and decoupling are critical for large distributed systems. Message queues help achieve these goals by enabling asynchronous communication between services.
7 / **C – Central Idea:**

1. Asynchronous communication models
2. Message brokers and queues
3. RabbitMQ and AMQP
4. Fanout, direct, and topic exchanges

7 / **D – Performance Objectives:** After studying this unit, the student will be able to:

1. Understand the concept of asynchronous task execution.
2. Use message queues to decouple services.
3. Implement task distribution using RabbitMQ.
4. Apply different exchange types for message routing.

Preface

- What would happen when many users are browsing a website / using an app?
  - Load on the networks (traffic) and the provider

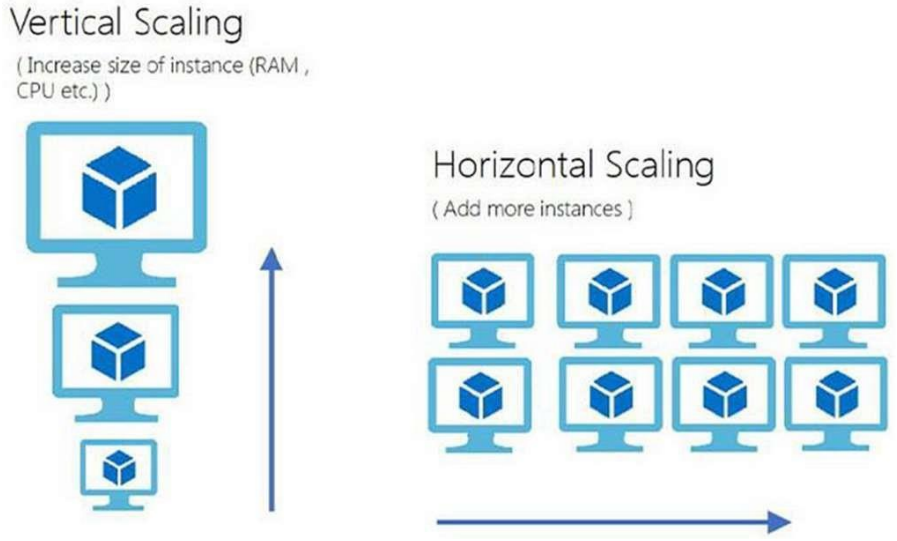
- **Can be complex**
  - Geographically distributed clients
  - Different types of clients
  - Different types of requests (e.g. data retrieval, data submission, data editing, …)
  - Different types of data (e.g. relational data, text, images, videos, …)

- **How to keep**
  - the service available?
  - high performance?
  - operating the service in a cost-effective way?

Scalability is the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending the system's capacity

- **Vertical (scale up)**
  - Add resources to a single node in a system
  - E.g. CPU, RAM, disk space, etc.
  - Quickly reach a limit (technical or financial)

- **Horizontal (scale out)**
  - Add more nodes to a system
  - Allow distributing workload easily
  - Need to manage large number of nodes



https://medium.com/design-microservices-architecture-with-patterns/

- **Keys to build scalable systems:**
  - **Horizontal scaling:** allow scaling to be done quickly
  - **Decoupling:** let modules interact through simple and well defined interfaces

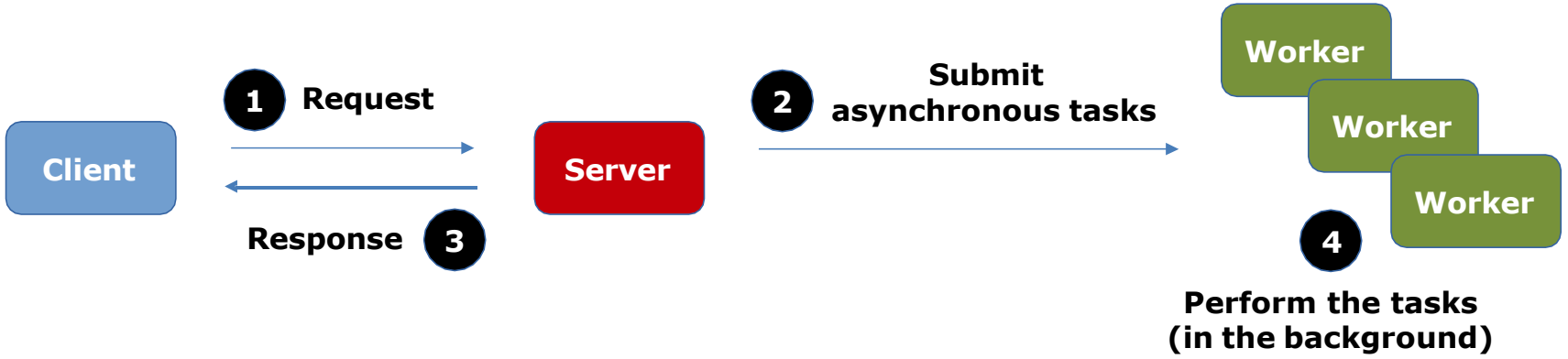
Basic Client-Server Model

- The server only performs work whenever there is a request
- The client waits for the response

- Long waiting is not expected
- Not all tasks can be completed quickly

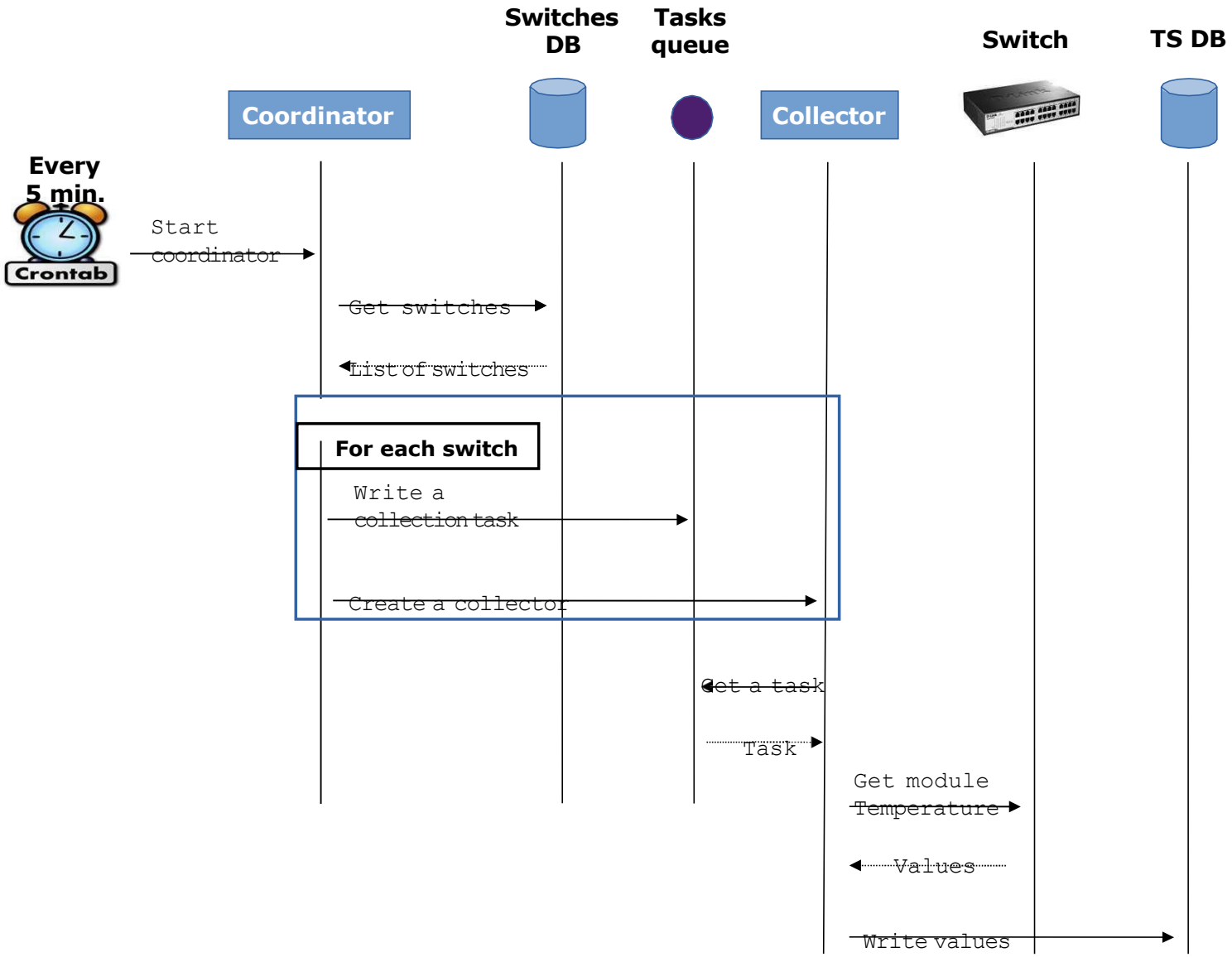
In many cases, it is necessary (or possible) to perform some tasks in the background

- **Asynchronous** (or non-blocking) tasks
- Achieve **decoupling** by separating the server and other services



Examples of Asynchronous Tasks

- An app in which notifications will be sent to the user's friends after she/he uploads a file
- A real-time feed of constantly updating information
- An app that deliver messages when the destination comes online
- A social online network in which a user needs to quickly retrieve the list of a friend's mutual friends

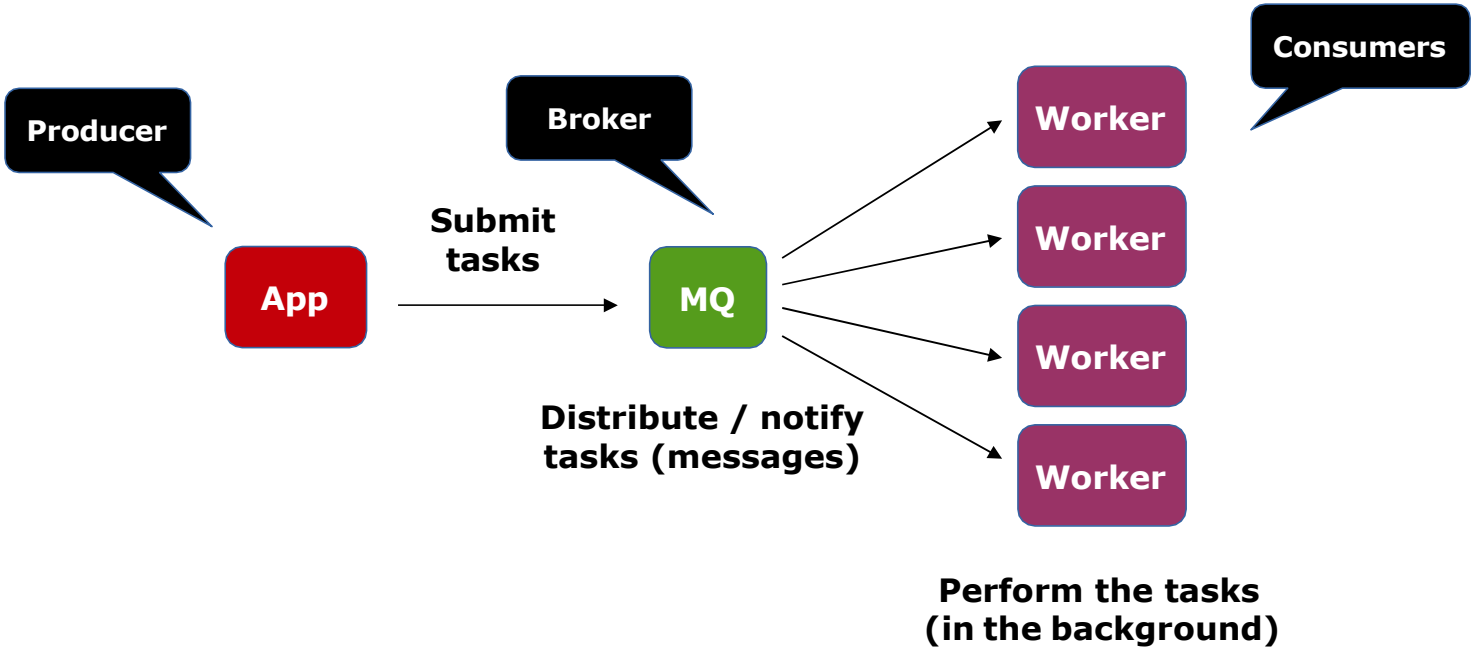

Switches DB
Tasks queue
Switch
TS DB
Coordinator
Collector
Every 5 min.
Crontab
Start coordinator
Get switches
List of switches
For each switch
Write a collection task
Create a collector
Get a task
Task
Get module Temperature
Values
Write values

Asynchronous Tasks

- May have to be executed sequentially or concurrently
- May have different priorities
- May require different amount of resources

 Some **manager** is needed to manage the tasks / messages between the server and the workers

Message Queues



Message queues make systems more robust

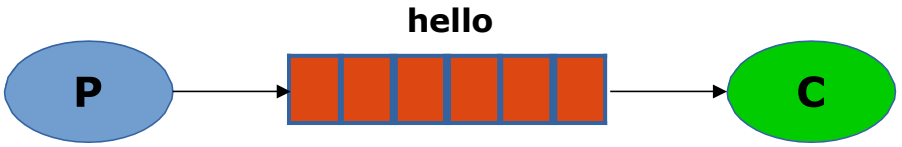
- Free the request – response circle from heavy tasks
- Clients are shielded from failures of background tasks
- The broker resubmit tasks in case of failures

RabbitMQ is a commonly used application-layer message broker software

- Speaks AMQP (Advanced Message Queuing Protocol)
- Accepts, stores, and forwards messages
- Defines how messages are routed and stored
- Defines how communications are done between clients and server
- Open source
- Easy to use
- Runs on all major operating systems & supported in Java, .NET, Ruby, Python, PHP, C, C++, Erlang, ...

Terminology

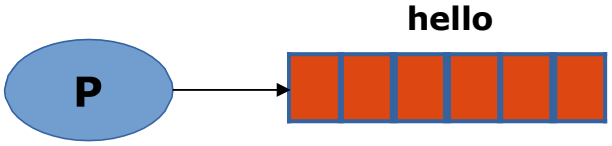
- **Producer:** app that sends a message
- **Queue:** a named messages store (infinite buffer)
- **Consumer:** app that waits to receive messages



Example 1: Send and receive messages from a named queue

send.py

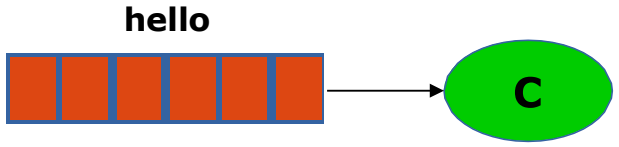
```
1
2   import pika
3
4   # establish a connection with a RabbitMQ server
5   connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
6   channel = connection.channel()
7
8   # create a queue to which the message will be delivered
9   channel.queue_declare(queue='hello')
10
11  channel.basic_publish(exchange='',
12                        routing_key='hello',
13                        body='Hello World!')
14
15  print(" [x] Sent 'Hello World!'")
16
17  connection.close()
18
```



Example 1: Send and receive messages from a named queue

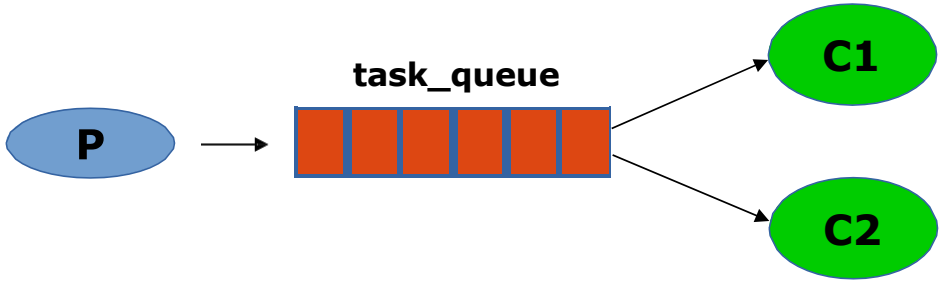
recv.py

```
1
2   import pika
3
4   # establish a connection with a RabbitMQ server
5   connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
6   channel = connection.channel()
7
8   # create a queue to which the message will be delivered
9   # created only once!!
10  channel.queue_declare(queue='hello')
11
12  # receive: subscribe a callback function to a queue
13  def callback(ch, method, properties, body):
14      print(" [x] Received %r" % body)
15
16
17  # tell RabbitMQ that the above callback function should receive messages from our hello queue
18  channel.basic_consume(callback, queue='hello', no_ack=True)
19
20  # wait for messages
21  print(' [*] Waiting for messages. To exit press CTRL+C')
22  channel.start_consuming()
23
```



Example 2: Distribute time-consuming tasks among multiple workers

- Create more workers and make them consume from the same queue
- Distribute messages to workers in a round-robin fashion
- **Idea:** avoid doing a resource-intensive task immediately and having to wait for it to complete (instead: schedule the task to be done later)
- A worker process running in the background will pop the tasks and eventually execute the job.



Example 2: Distribute time-consuming tasks among multiple workers

new_task.py

```
1
2  import pika
3  import sys
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
6  channel = connection.channel()
7
8  channel.queue_declare(queue='task_queue', durable=True)
9
10 message = ' '.join(sys.argv[1:]) or "Hello World!"
11 channel.basic_publish(exchange='',
12                        routing_key='task_queue',
13                        body=message,
14                        properties=pika.BasicProperties(
15                          delivery_mode = 2, # make message persistent
16                        ))
17
18 print(" [x] Sent %r" % message)
19 connection.close()
```

@ the consumer side as well

1 = non-persist

2 = persistent

Example 2: Distribute time-consuming tasks among multiple workers

worker.py

```
1
2  import pika
3  import time
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6          host='localhost'))
7  channel = connection.channel()
8
9  channel.queue_declare(queue='task_queue', durable=True)
10 print(' [*] Waiting for messages. To exit press CTRL+C')
11
12 def callback(ch, method, properties, body):
13     print(" [x] Received %r" % body)
14
15     # simulate (fake) a second of work for every dot in the message body
16     time.sleep(body.count(b'.'))
17
18     print(" [x] Done")
19     ch.basic_ack(delivery_tag = method.delivery_tag)
20
21 channel.basic_qos(prefetch_count=1)
22 channel.basic_consume(callback,
23                       queue='task_queue')
24
25 channel.start_consuming()
26
```

Example 2: Distribute time-consuming tasks among multiple workers

```
$ python new_task.py first
[x] Sent ‘first’

$ python new_task.py second
[x] Sent ‘second’

$ python new_task.py third
[x] Sent ‘third’

$ python new_task.py fourth
[x] Sent ‘fourth’

$ python new_task.py fifth
[x] Sent ‘fifth’
```

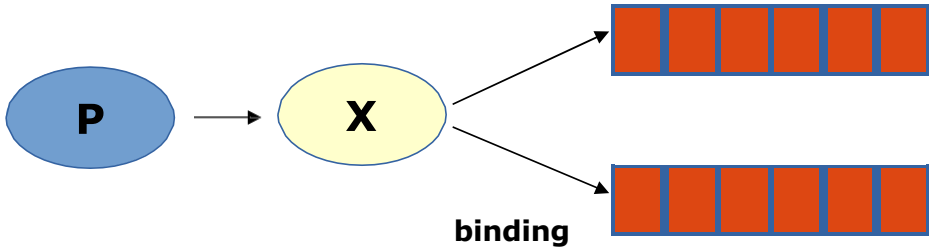
```
$ python worker.py
```

```
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'second'
[x] Done
[x] Received 'fourth'
[x] Done
```

```
$ python worker.py
```

```
[*] Waiting for messages. To exit press CTRL+C
[x] Received 'first'
[x] Done
[x] Received 'third'
[x] Done
[x] Received 'fifth'
[x] Done
```

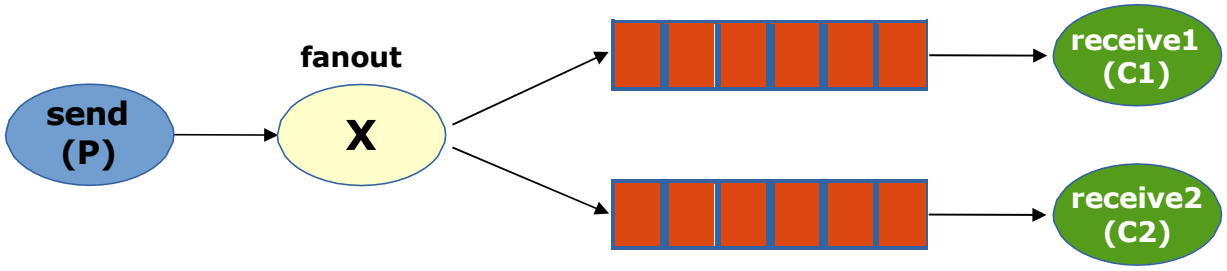
PubSub



- Deliver a message to multiple consumers
- Producer sends messages to an exchange (not directly to a queue, i.e. not through a default / nameless exchange: “”)
- **Exchange**
  - Receives messages from producers
  - & pushes them to (no, one / particular, or multiple) queues
- **Exchange Types:**
  - Fanout – broadcasts the messages it receives to all queues it knows
  - Direct
  - Topic

Fanout Exchange Example: A Simple Logging System

- Every receiver gets a copy of each queued message
- Consumer 1: directs messages to a log file
- Consumer 2: shows them on a display
- Consumer N: ….



Fanout Exchange Example: A Simple Logging System

send.py

```
1
2  import pika
3  import sys
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6          host='localhost'))
7  channel = connection.channel()
8
9  channel.exchange_declare(exchange='logs', type='fanout')
10
11 message = ' '.join(sys.argv[1:]) or "info: Hello World!"
12 channel.basic_publish(exchange='logs',
13                        routing_key='',
14                        body=message)
15 print(" [x] Sent %r" % message)
16 connection.close()
17
```

Fanout Exchange Example: A Simple Logging System

recv.py

```
1
2   import pika
3
4   connection = pika.BlockingConnection(pika.ConnectionParameters(
5           host='localhost'))
6   channel = connection.channel()
7
8   channel.exchange_declare(exchange='logs', type='fanout')
9
10  result = channel.queue_declare(exclusive=True)
11  queue_name = result.method.queue
12
13  # delete the queue once we disconnect the consumer
14  channel.queue_bind(exchange='logs', queue=queue_name)
15
16  print(' [*] Waiting for logs. To exit press CTRL+C')
17
18  def callback(ch, method, properties, body):
19      print(" [x] %r" % body)
20
21  channel.basic_consume(callback,
22                        queue=queue_name,
23                        no_ack=True)
24
25  channel.start_consuming()
26
```

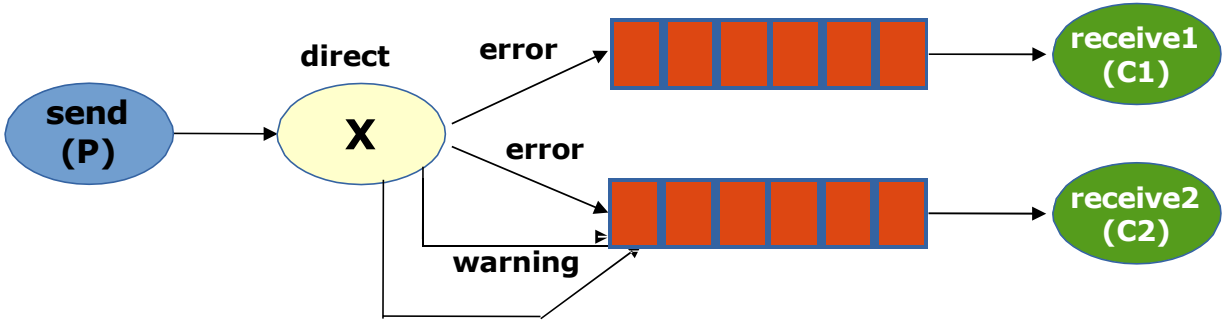
```
$ python recv.py

$ python recv.py > test.log
```

```
$ python send.py
$ python send.py
$ python send.py
```

Routing allows to subscribe only to a subset of the messages

- Uses direct exchange
  - Message goes to the queues whose binding key exactly matches the routing key of the message

- **Modified Logging System:**
  - Direct only critical error messages to Consumer 1 (to be saved in a log file)
  - Direct all messages to Consumer 2 (to be displayed on the monitor)



Routing Example:
Modified Logging System

send.py

```
1
2  import pika
3  import sys
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6          host='localhost'))
7  channel = connection.channel()
8
9  channel.exchange_declare(exchange='direct_logs',
10                           type='direct')
11
12 severity = sys.argv[1] if len(sys.argv) > 1 else 'info'
13 message = ' '.join(sys.argv[2:]) or 'Hello World!'
14 channel.basic_publish(exchange='direct_logs',
15                         routing_key=severity,
16                         body=message)
17 print(" [x] Sent %r:%r" % (severity, message))
18 connection.close()
19
```

Routing Example:
Modified Logging System

recv.py

```
2  import pika
3  import sys
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6          host='localhost'))
7  channel = connection.channel()
8
9  channel.exchange_declare(exchange='direct_logs',
10                           type='direct')
11
12 result = channel.queue_declare(exclusive=True)
13 queue_name = result.method.queue
14
15 severities = sys.argv[1:]
16 if not severities:
17     sys.stderr.write("Usage: %s [info] [warning] [error]\n" % sys.argv[0])
18     sys.exit(1)
19
20 for severity in severities:
21     channel.queue_bind(exchange='direct_logs',
22                        queue=queue_name,
23                        routing_key=severity)
24
25 print(' [*] Waiting for logs. To exit press CTRL+C')
26
27 def callback(ch, method, properties, body):
28     print(" [x] %r:%r" % (method.routing_key, body))
29
30 channel.basic_consume(callback,
31                       queue=queue_name,
32                       no_ack=True)
33
34 channel.start_consuming()
```

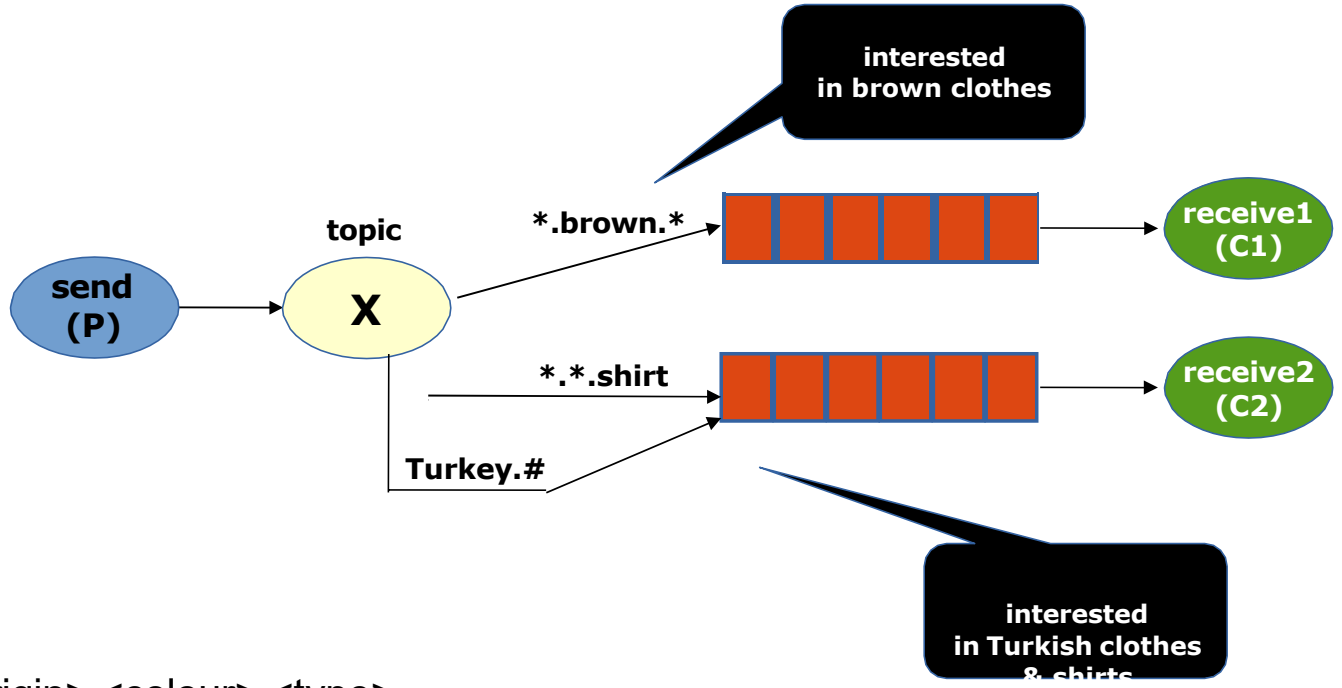
```
$ python recv.py error > test2.log
$ python recv.py error warning info
```

```
$ python send.py error 'test error!'
$ python send.py info 'test info!'
$ python send.py warning 'test warn!'
```

Topics Exchange

- Direct exchanges cannot route based on multiple criteria
- What if an app (i.e. consumer) needs to subscribe not only based on content but also based on the sender?
- **Solution:** Topic Exchange
  - Listen to messages based on a pattern
  - **Routing_key:** a list of words, delimited by dots (up to 255 bytes)
  - **Binding key:** a list of words, delimited by dots (up to 255 bytes)
    - * substitutes one word
    - # substitutes zero or more words

Topics Exchange Example



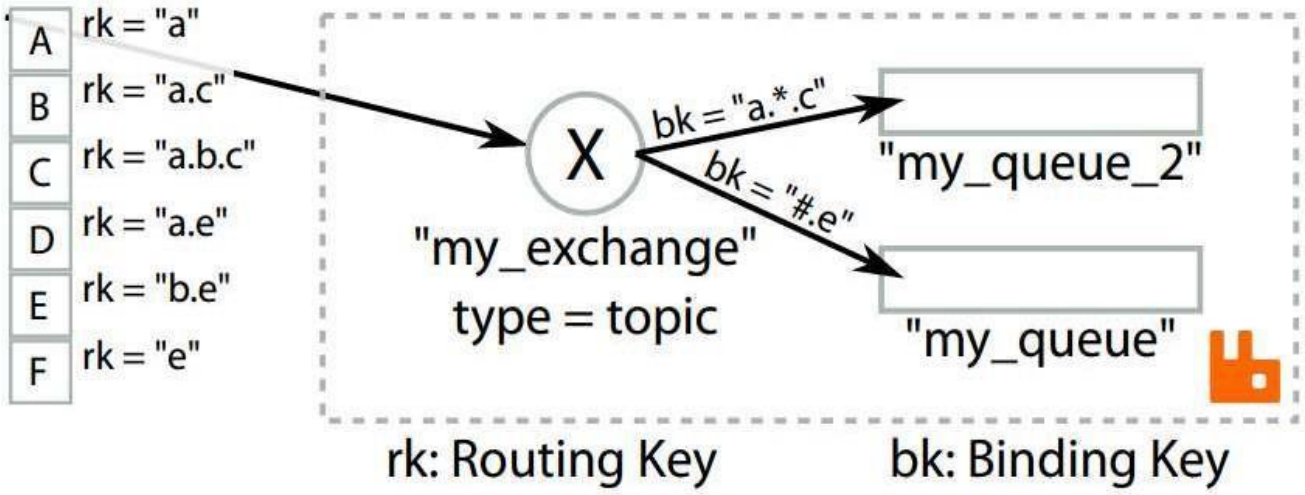
- **Routing key:** <origin>.<colour>.<type>

- **Special binding keys:**
  - # = fanout (receives all messages regardless of the routing key)

Topics Exchange Example

- no # nor * = direct

Topics Exchange Example



Topics Exchange Example

send.py

```
1
2  import pika
3  import sys
4
5  connection = pika.BlockingConnection(pika.ConnectionParameters(
6          host='localhost'))
7  channel = connection.channel()
8
9  channel.exchange_declare(exchange='topic_logs',
10                           type='topic')
11
12 routing_key = sys.argv[1] if len(sys.argv) > 1 else 'anonymous.info'
13 message = ' '.join(sys.argv[2:]) or 'Hello World!'
14 channel.basic_publish(exchange='topic_logs',
15                         routing_key=routing_key,
16                         body=message)
17 print(" [x] Sent %r:%r" % (routing_key, message))
18 connection.close()
19
```

Topics Exchange Example

recv.py

```
 2  import pika
 3  import sys
 4
 5  connection = pika.BlockingConnection(pika.ConnectionParameters(
 6          host='localhost'))
 7  channel = connection.channel()
 8
 9  channel.exchange_declare(exchange='topic_logs',
10                           type='topic')
11
12  result = channel.queue_declare(exclusive=True)
13  queue_name = result.method.queue
14
15  binding_keys = sys.argv[1:]
16  if not binding_keys:
17      sys.stderr.write("Usage: %s [binding_key]...\n" % sys.argv[0])
18      sys.exit(1)
19
20  for binding_key in binding_keys:
21      channel.queue_bind(exchange='topic_logs',
22                         queue=queue_name,
23                         routing_key=binding_key)
24
25  print(' [*] Waiting for logs. To exit press CTRL+C')
26
27  def callback(ch, method, properties, body):
28      print(" [x] %r:%r" % (method.routing_key, body))
29
30  channel.basic_consume(callback,
31                        queue=queue_name,
32                        no_ack=True)
33
34  channel.start_consuming()
```

```
$ pyth     recv.py “#”
$ pyth     recv.py “kern.*“
$ pyth     recv.py "*.critical"
$ python recv.py "kern.*" "*.critical"
```

```
$ python send3.py "kern.critical" "A critical kernel error"
```

Popular RabbitMQ Commands

```
$ sudo rabbitmqctl status

$ sudo rabbitmqctl list_queues

$ sudo rabbitmqctl list_exchanges

$ sudo rabbitmqctl list_bindings
```

Use Cases

Sources

- Slides (IERG 4080 Building Scalable Internet-based Services, Lecture 7)
- https://www.rabbitmq.com/

Questions?

Network Programming

8) Network Management

Dr. Hassan Almazini

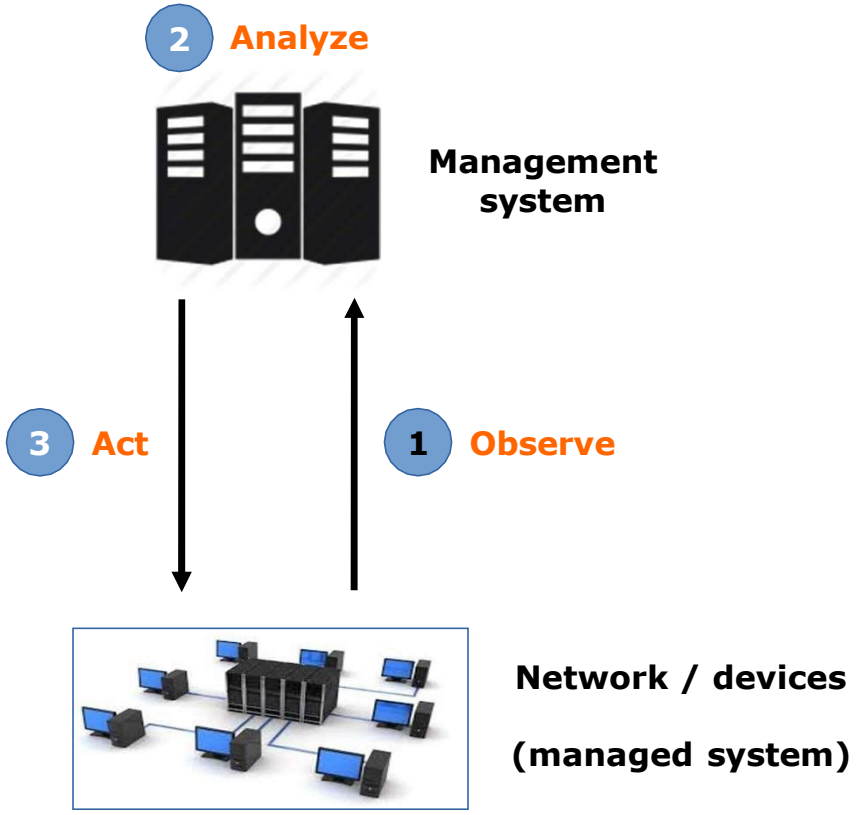
A – Target Population: For students of Second Year
Technological Institute of Basra
Dep. Of Computer network and software technologies
8 / **B – Rationale:** Efficient management of computer networks is essential for ensuring reliability, performance, and security. This unit introduces the foundational concepts and protocols of network management.
8 / **C – Central Idea:**

1. Overview of network management functions
2. FCAPS model
3. SNMP protocol and versions
4. Network monitoring and control

8 / **D – Performance Objectives:** After studying this unit, the student will be able to:

1. Explain the purpose and scope of network management.
2. Apply the FCAPS model (Fault, Configuration, Accounting, Performance, Security).
3. Understand SNMP operations and architecture.
4. Describe real-time network monitoring practices.

Introduction to Network Management


2 Analyze
Management system
3 Act
1 Observe
Network / devices
(managed system)

Network management Refers to the activities and tools that pertain to the operation and administration of networked systems

- A general concept that employs the use of various tools, techniques, and systems to aid human beings in managing various devices, systems, or networks

- **Involves five tasks (FCAPS):**
  - Fault management
  - Configuration management
  - Accounting management & user administration
  - Performance management
  - Security management

Fault Management

- **Goal:** detect, log, and notify users of systems or networks of problems
  - In many environments, downtime of any kind is not acceptable

- **Fault resolution steps:**
  - Isolate the problem by using tools to determine symptoms
  - Resolve the problem
  - Record the process that was used to detect and resolve the problem

Configuration Management

- **Goal:** monitor network and system configuration information so that the effects on network operation of various versions of hardware and software elements can be tracked and managed

- **Configuration parameters include:**
  - Version of operating system, firmware, etc.
  - Number of network interfaces and speeds, etc.
  - Number of hard disks
  - Number of CPUs
  - Amount of RAM
  - ...

Accounting Management

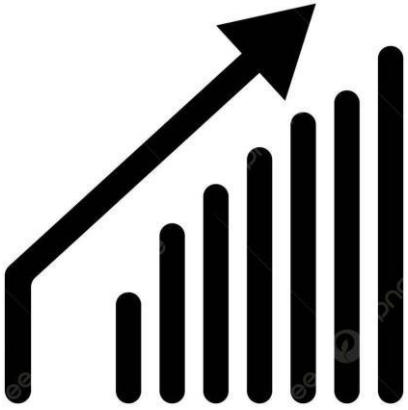
- **Goal:** ensure that computing and network resources are used fairly by all groups or individuals who access them

- Through this form of regulation, network problems can be minimized since resources are divided based on capacities

Performance Management

- **Goal:** measure and report on various aspects of network or system performance

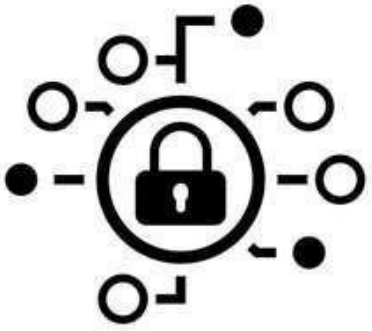
- **Steps:**
  - Gather performance data
  - Define baseline levels based on analysis of the data gathered
  - Define performance thresholds
    - When these thresholds are exceeded, it is indicative of a problem that requires attention



Security Management

- **Goals:**
  - Control access to network resources
  - Help to detect and prevent attacks

- **Tools:**
  - Firewalls
  - Intrusion Detection Systems (IDSs)
  - Intrusion Prevention Systems (IPSs)
  - Anti-virus systems
  - Policy management and enforcement systems



Simple Network Management Protocol (SNMP)

A set of operations that gives administrators the ability to monitor and change the state of some SNMP-enabled devices (and applications)






SNMP Versions

- **SNMPv1**
  - RFC 1157
  - Still the primary SNMP implementation that many vendors support
  - **Security:** based on communities (i.e. passwords)

- **SNMPv2**
  - RFC 3416, RFC 3417, and RFC 3418
  - Technically called SNMPv2c

- **SNMPv3**
  - RFC 3410, RFC 3411, RFC 3412, RFC 3413, RFC 3414, RFC 3415, RFC 3416, RFC 3417, and RFC 2576
  - Adds support for strong authentication and private communication between managed entities
  - Not widely supported

SNMP Managers

- Aka. **Network Management Stations (NMSs)**
- A server running software system that can handle management tasks for a network
- Responsible for polling and receiving **traps** from **agents** in the network
  - **Poll:** querying an agent for some piece of information
  - **Trap:** a way for the agent to tell the NMS that something has happened
    - sent asynchronously, not in response to queries from the NMS
- Also is responsible for performing an action based upon the information (i.e. traps) it receives from agents

SNMP Agents

- A piece of software that runs on the managed network devices.
  - It can be a separate program, or it can be incorporated into the OS
- Today, most IP devices come with a built-in SNMP agent
- The agent provides management information to the NMS by keeping track of various operational aspects of the device
- When the agent notices that something bad has happened, it can send a trap to the NMS
- Some devices will send a corresponding "**all clear**" trap when there is a transition from a bad state to a good state

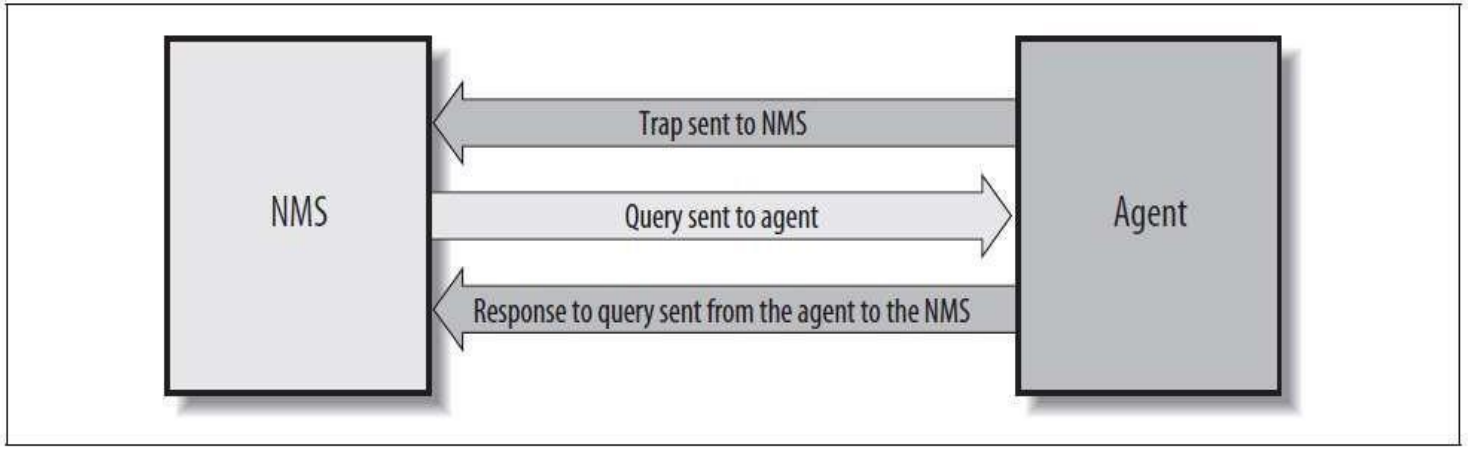


Figure 1-1. Relationship between an NMS and an agent

Structure of Management Information (SMI)

- A way to define managed objects and their behavior
- Each agent has a list of the objects that it tracks
  - This list collectively defines the information the NMS can use to determine the overall health of the device on which the agent resides

Management Information Base (MIB)

- A database of managed objects that the agent tracks
- Any sort of status or statistical information that can be accessed by the NMS is defined in a MIB
- The SMI provides a way to define managed objects while the MIB is the definition (using the SMI syntax) of the objects themselves
- An agent may implement many MIBs
- All agents implement a particular MIB called **MIB-II** (RFC 1213)
  - **Defines:**
    - interface statistics (speed, MTU, octets sent, octets received, etc.)
    - system info (location, contact, etc.)
    - other info (vendor-defined and/or administrator-defined)

Remote Monitoring (RMON)

- **RMONv1 (or RMON):** RFC 2819
  - A MIB provides the NMS with packet-level statistics about an entire LAN or WAN
- **RMONv2:** RFC 2021
  - Builds on RMONv1: provides network- and application-level statistics
- Place an RMON probe on every / some network segment(s)
- The RMON MIB allows RMON probes to run in an **off-line mode**
  - Gather statistics about the network without requiring an NMS to query it constantly
  - Later on, the NMS can query the probe for statistics
- Probes can set thresholds for various error conditions (when a threshold is crossed, alert the NMS with an SNMP trap)

SNMP (mostly v1) Details

SNMP is more than a protocol; it is a management framework, including architecture information model management operations

- In this part, we refer to SNMPv1, unless stated otherwise
- Virtually all networked devices support it
- SNMP uses UDP for transporting messages between NMSs and agents
  - Why UDP?
  - **Default ports:** 161 for receiving requests 162 for receiving traps
  - The NMS sends a UDP request to an agent and waits for a response
    - Configurable timeout ☐ retransmit if …
    - Number of retransmissions is also configurable

SNMP Communities

- Community string = password
- Used to establish trust between NMSs and agents
  - Manager needs to know community in order for access to succeed
- Agent is configured with three access modes
  - **Read-only** (default: "public")
  - **Read-write** (sometimes default is "private" or no default)
  - **Trap**
- **MIB View:** subset (or all) of objects in MIB
  - Possible that different communities may have different views
  - MIB object has ACCESS defined with it

Management Information

- **Management Information** is modeled as **managed objects (MOs)** and relationships among them
  - MOs: operational parameters of SNMP-capable devices
- A **Management Information Bases (MIB)** is a collection of objects, grouped for a specific management purpose
- All objects are organized in the global **MIB tree**
  - Each MIB represents a sub tree of this global MIB tree
  - The leaf objects of the tree contain object instances with the state and control variables of the managed system
  - **MIB-II** is the most popular MIB; it is implemented in most SNMP-managed devices
  - Device manufacturers often define their own device-specific MIBs

Structure of Information Management (SIM)

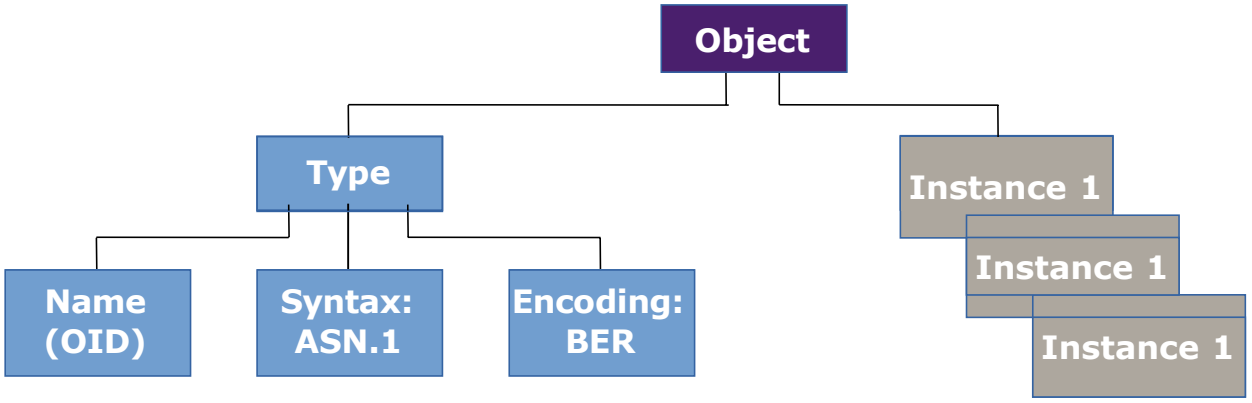
- **SMI** defines how **MOs** are named and specifies their associated data types
- **Two versions:** v1 (RFC 1155) & v2 (RFC 2578)
- These definitions are written in the language ASN.1 (Abstract Syntax Notation 1)
- **Has three attributes:**
  - **Name (or OID):** uniquely defines a MO
  - **Data type and syntax**
  - **Encoding:** a single instance of a managed object is encoded into a string of octets using the Basic Encoding Rules (BER)
    - BER defines how the objects are encoded and decoded

Data Types

- The SYNTAX attribute provides definitions of managed objects through a subset of **ASN.1**
- **SMIv1 Data Types:**
  - **Scalar Types:** INTEGER, OCTET, STRING, Counter, OBJECT IDENTIFIER, IpAddress, Gauge, TimeTicks, …
  - **Table:**
    - Table of scalar objects (seq of records)
    - E.g., ARP tables, routing tables, ...
    - Table has three parts:
      - Table name
      - Row name
      - Column object name

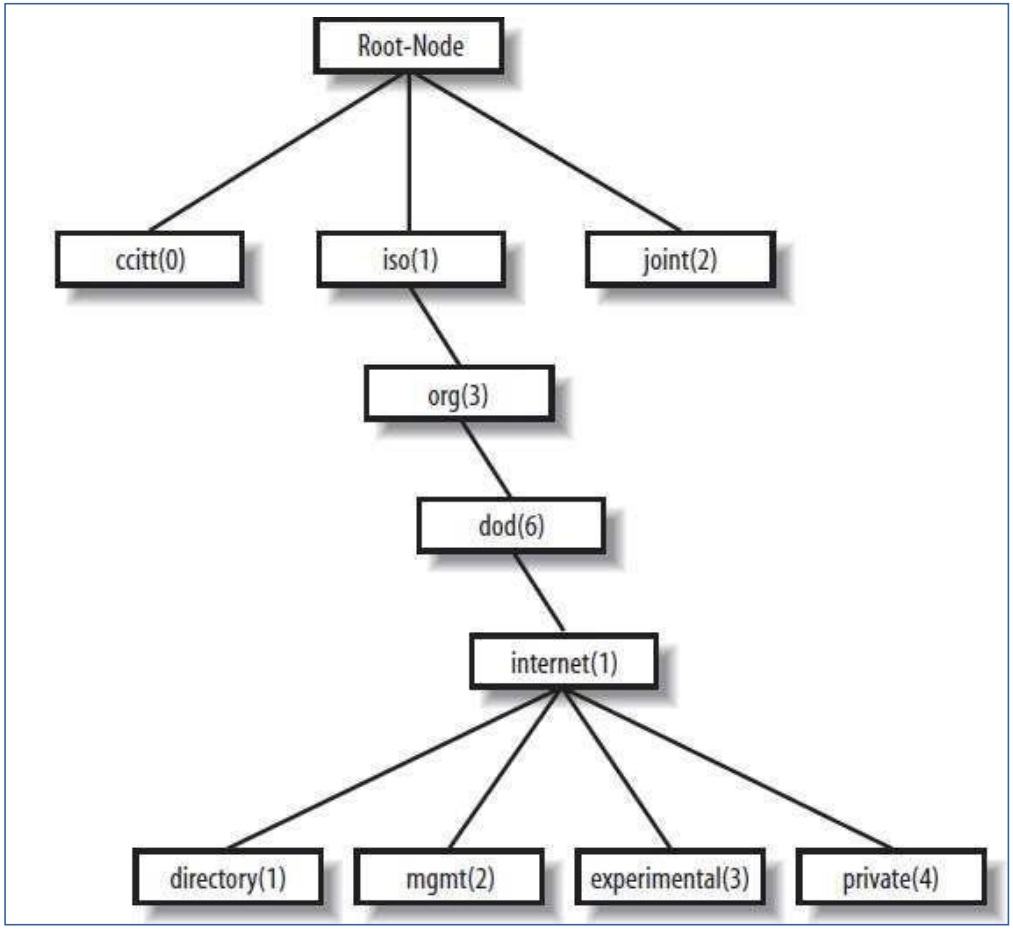
Multiple Instances of MOs

- **Example:** 3 switches of the same version have identical identifier (i.e. same object type)
- They differ in actual values stored: various instances; identified by different IP addresses



Naming OIDs

Upper part of the global MIB tree



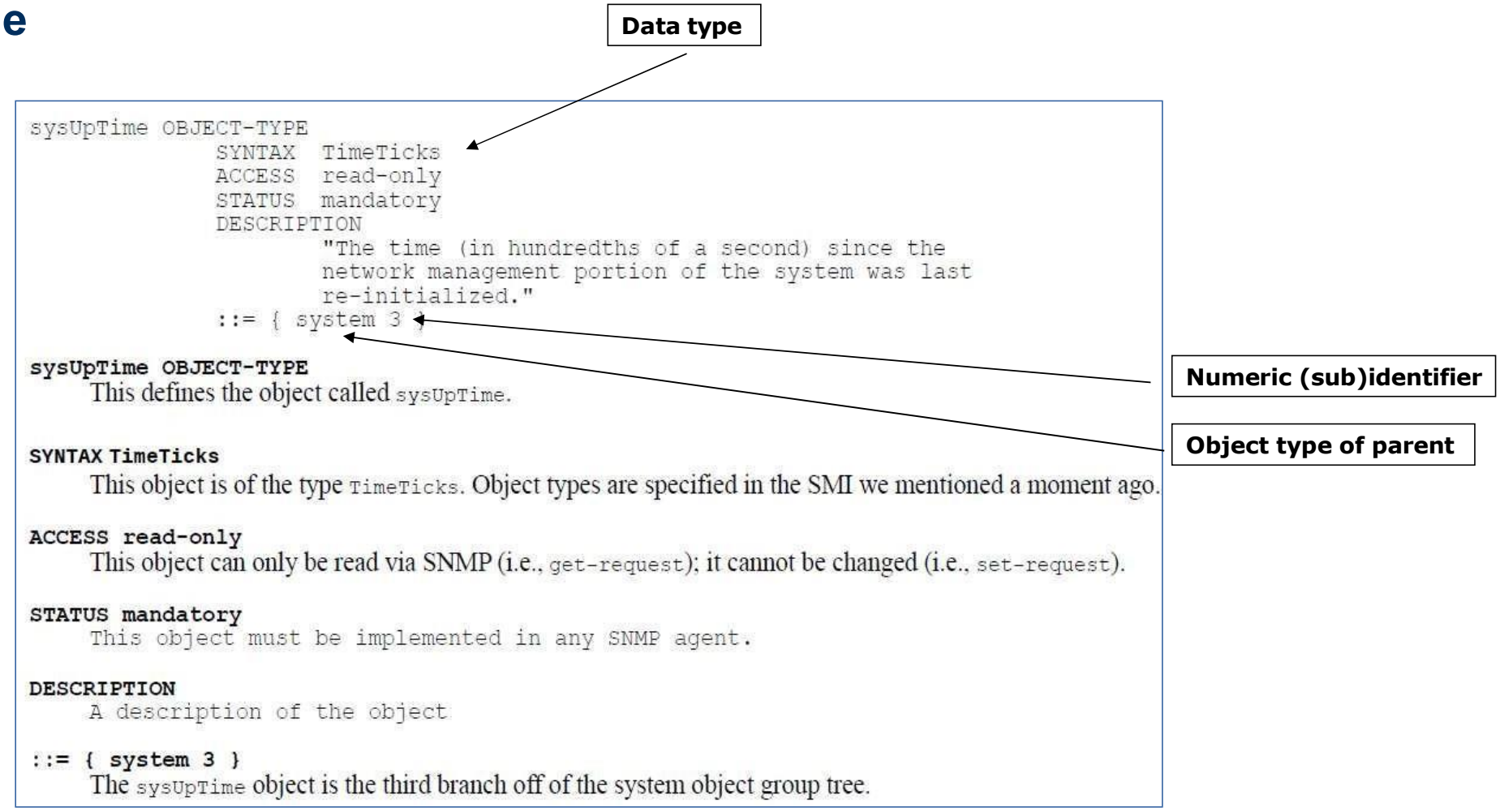
Naming OIDs

- IANA manages the **private** enterprise number assignments for individuals, institutions, organizations, companies, etc.
  - http://www.iana.org/assignments/enterprise-numbers

- **Example:** Cisco Systems private enterprise number is 9
  - The base OID for its private object space:
    - iso.org.dod.internet.private.enterprises.cisco, or
    - 1.3.6.1.4.1.9

- How do we convert from OIDs to Labels (and vice versa)?
  - Use MIBs files!

MIB Example

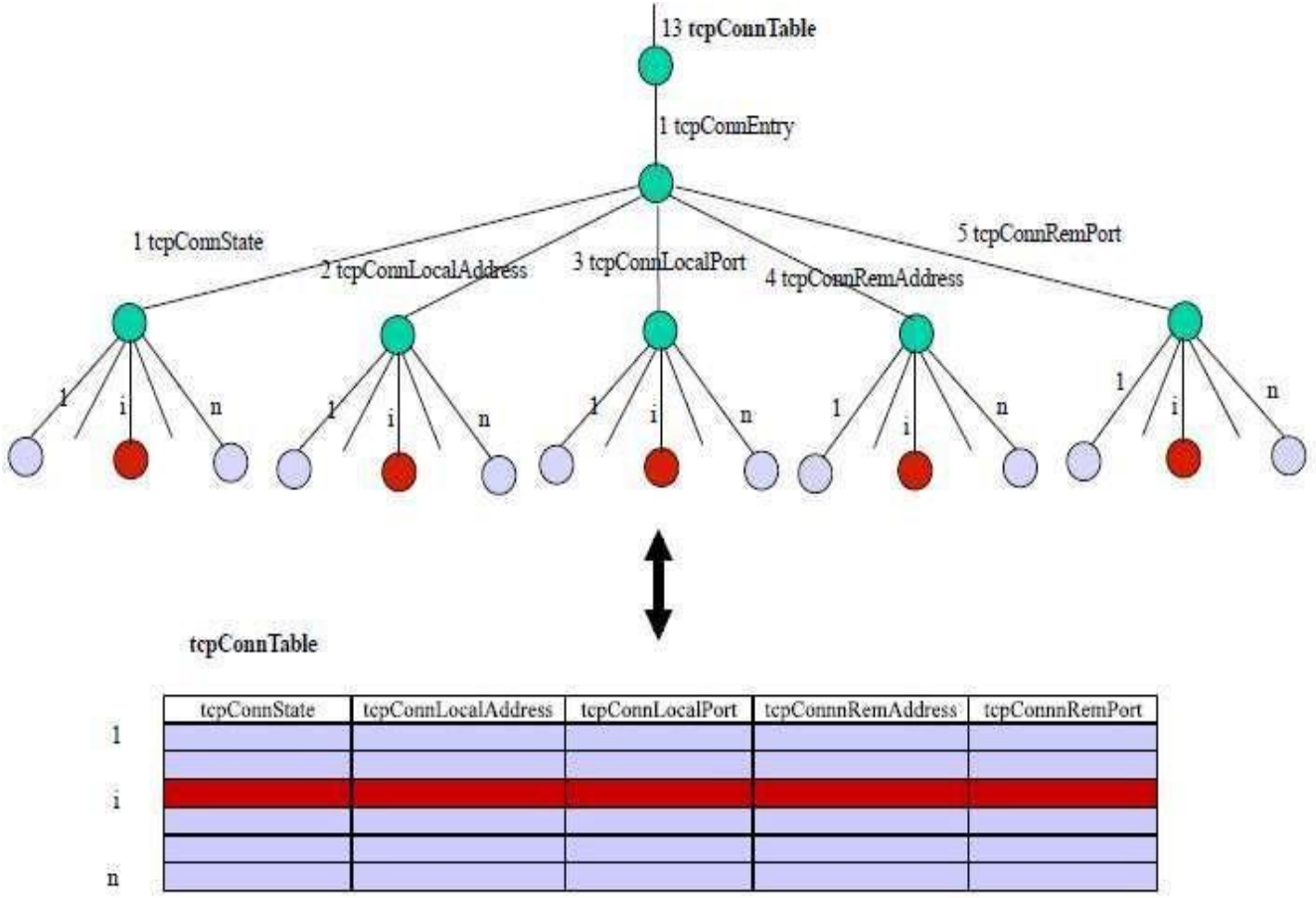

Data type
sysUpTime OBJECT-TYPE
SYNTAX TimeTicks
ACCESS read-only
STATUS mandatory
DESCRIPTION
"The time (in hundredths of a second) since the
network management portion of the system was last
re-initialized."
::= { system 3 }
Numeric (sub)identifier
Object type of parent
sysUpTime OBJECT-TYPE
This defines the object called sysUpTime.
SYNTAX TimeTicks
This object is of the type TimeTicks. Object types are specified in the SMI we mentioned a moment ago.
ACCESS read-only
This object can only be read via SNMP (i.e., get-request); it cannot be changed (i.e., set-request).
STATUS mandatory
This object must be implemented in any SNMP agent.
DESCRIPTION
A description of the object
::= { system 3 }
The sysUpTime object is the third branch off of the system object group tree.

Naming OIDs

- The OID of an instance of a scalar object type with OID X is denoted by **X.0**

- The OID of a table element in table X is denoted by **X.1.column.(i_1).(i_2).....(i_n)**, where:
  - **X** is the identifier of the table object type
  - **column** is the column number
  - **(i_1) … (i_n)** is the table index

Representing a Table Object on the MIB Tree



MIBs also …

- Make it possible to interpret a returned value from an agent
- For example, the status for a fan could be 1, 2, 3, 4, 5, 6
  - What does it mean?

MIB Example

```
CiscoEnvMonState ::= TEXTUAL-CONVENTION
        STATUS  current
        DESCRIPTION
                "Represents the state of a device being monitored.
                 Valid values are:

                 normal(1):          the environment is good, such as low
                                     temperature.

                 warning(2):         the environment is bad, such as temperature
                                     above normal operation range but not too
                                     high.

                 critical(3):        the environment is very bad, such as
                                     temperature much higher than normal
                                     operation limit.

                 shutdown(4):        the environment is the worst, the system
                                     should be shutdown immediately.

                 notPresent(5):      the environmental monitor is not present,
                                     such as temperature sensors do not exist.

                 notFunctioning(6): the environmental monitor does not
                                     function properly, such as a temperature
                                     sensor generates a abnormal data like
                                     1000 C.
                  "
```

SNMP Messages

- **GET X**
  - Query for a value of leaf object with OID X
  - Manager ☐ agent
- **GET-NEXT Y**
  - Get value of leaf object following Y
  - Allows to list the elements of a table or of the leaf objects of a MIB
  - Manager ☐ agent
- **RESPONSE**
  - Response to GET/SET, ACK to set, or error
  - Agent ☐ manager

SNMP Messages

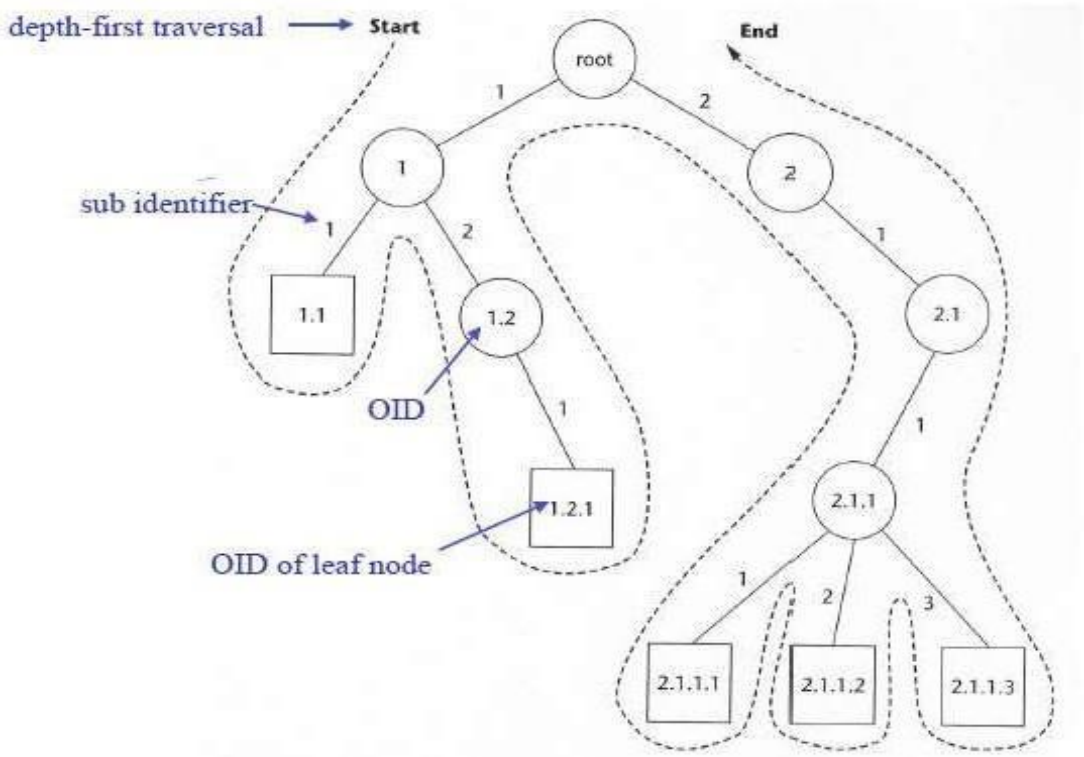
- **SET**
  - Set a value, or perform action
  - Manager ☐ agent
- **TRAP**
  - Alert from equipment (e.g. line down, temperature above threshold, ...)
  - Agent ☐ manager

Accessing MIB Elements

- **Random:** supply the exact OID, and get/set value
  - Tricky if table entry
  - Simple otherwise

- **Sequential:** done on basis of lexicographical ordering of OIDs
  - An OID is a series of small integers from left to right (x1, x2, ...)
  - Visit the root and then traverse subtrees from left to right
  - Depth-first search of the tree

Ordering MIB Elements



Commands for Querying Agents

- **snmpget**
- **snmpwalk**
- **snmpstatus**
- **snmptable**

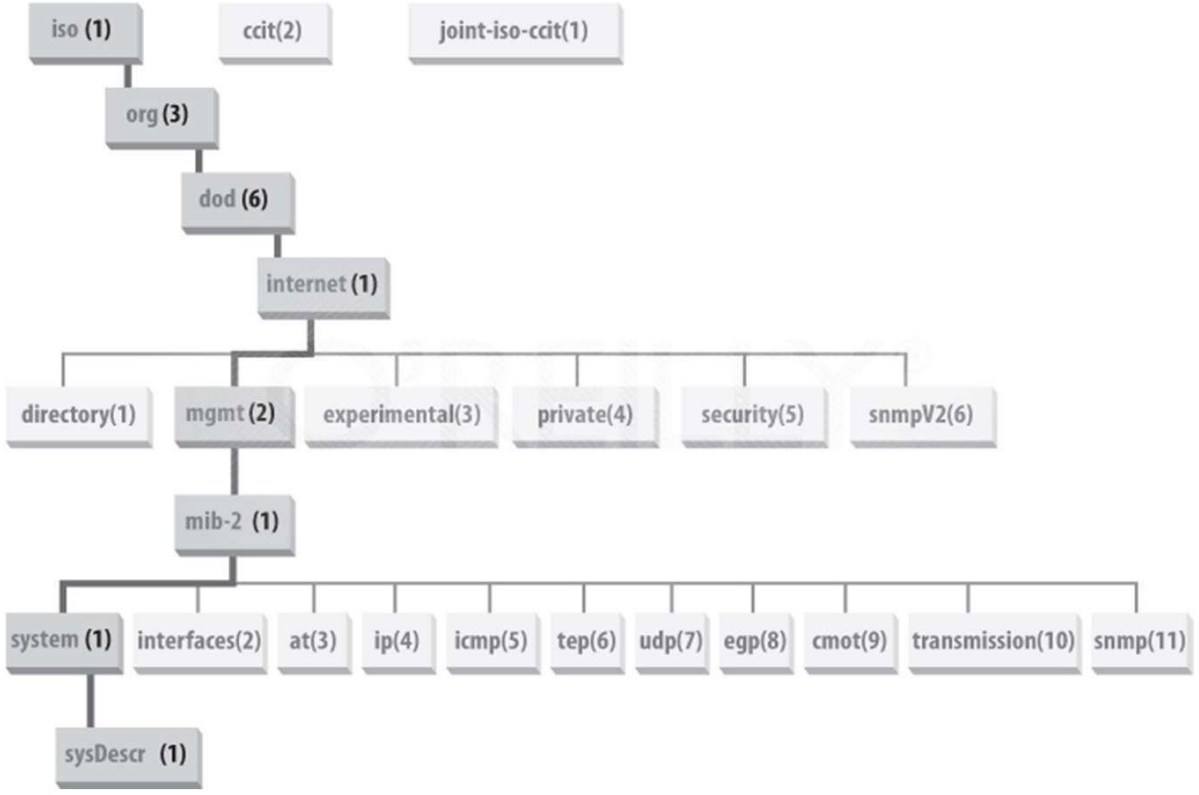
Syntax:

```
snmpXXX -c community -v1 host [oid]

snmpXXX -c community -v2c host [oid]
```

Examples

- snmpstatus -c NetManage -v2c 10.10.0.254
- snmpget -c NetManage -v2c 10.10.0.254 .iso.org.dod.internet.mgmt.mib-2.interfaces.ifNumber.0
- snmpwalk -c NetManage -v2c 10.10.0.254 ifDescr

```
snmpget -c public -v1 solarisbox .1.3.6.1.2.1.1.1.0
```




```
snmpget -c public -v1 solarisbox \
.iso.org.dod.internet.mgmt.mib-2.system.sysDescr.0
```

Traps

- Events / alerts asynchronously sent by agent to manager
  - Most important type is linkDown (interface crashed)

- **Types:**
  - coldStart(0) - unexpected restart or crash
  - warmStart(1) - soft reboot
  - linkDown(2)
  - linkUp(3)
  - authenticationFailure(4)
  - egpNeighborLoss(5)
  - enterpriseSpecific(6)

Strengths of SNMPv1

- **Simplicity**
  - Simple data model
  - Only four operations
  - Simple interaction model
  - Connectionless transport
- **Low complexity** on agent side
- **Ubiquity**
  - (Almost) every networked device has an SNMP agent
- **Well tested**

Limitations of SNMPv1

- **Limited Expressiveness**
  - Management commands have to be expressed as reading and writing single (scalar) object values
- **Limited Scalability:** In large networks, the polling model of interaction can lead to
  - High load on management station
  - High management traffic
  - Long execution time
- **Weak Security Model**
  - Authentication is based on unencrypted password (community string)
  - Consequently, SNMP is primarily used for monitoring, even today

SNMPv

- **Inform Request**
  - Multiple managers coordination Locking mechanisms prevent multiple managers from writing at the same time

- **Get-bulk X m**
  - Returns the OIDs and values of the m lexicographical (leaf) successors to the object with OID X
  - This allows, for instance, to read a table by sending a single request

- **Confirmation options for Traps**
  - Agents can ensure that trap was received correctly

SNMPv

- Security update of SNMPv2
- **Authentication:** Message authentication code with a shared secret key
- **Privacy:** Encryption using a shared secret key
- **Access Control:** Each manager can have a different set of read/write permission for various components of MIB

SNMP Commands & More

SNMP Daemon and Client

- **SNMP Manager**
  - sudo apt-get update
  - sudo apt-get install snmp
  - sudo apt-get install snmp-mibs-downloader (a package contains some proprietary information about standard MIBs that allow us to access most of the MIB tree by name)
  - sudo nano /etc/snmp/snmp.conf ☐ #mibs (to allow the manager to import the MIB files)

- **SNMP Agent** (another machine)
  - sudo apt-get update
  - sudo apt-get install snmpd
  - sudo nano /etc/snmp/snmpd.conf

```
#  Listen for connections from the local system only
#agentAddress  udp:127.0.0.1:161
#  Listen for connections on all interfaces (both IPv4 *and* IPv6)
agentAddress udp:161,udp6:[::1]:161
```

Naming OIDs

- The OID of an instance of a scalar object type with OID X is denoted by **X.0**

- The OID of a table element in table X is denoted by **X.1.column.(i_1).(i_2).....(i_n)**, where:
  - **X** is the identifier of the table object type
  - **column** is the column number
  - **(i_1) … (i_n)** is the table index

Example: sysName

- Under ”system” (OID: 1.3.6.1.2.1.1)
- SysName’s suffix: 5
- SysName's OID: 1.3.6.1.2.1.1.5.0
  - The final 0 represents a SNMP convention that indicates this is a scalar value and is not part of a table

```
$ snmpget -v1 -c public localhost 1.3.6.1.2.1.1.5.0
```

```
$ snmpget -v1 -c public localhost sysName.0
```

Siblings

- 1.3.6.1.2.1.1.1 – sysDescr
- 1.3.6.1.2.1.1.2 – sysObjectID
- 1.3.6.1.2.1.1.3 – sysUpTime
- 1.3.6.1.2.1.1.4 – sysContact
- 1.3.6.1.2.1.1.5 – sysName
- 1.3.6.1.2.1.1.6 – sysLocation
- 1.3.6.1.2.1.1.7 - sysServices

Example

- **IfTable**
- **OID:** 1.3.6.1.2.1.2.2
- **IfEntry prefix:** T = 1.3.6.1.2.1.2.2.1

OID	table column
{ ifEntry 1 } or **T**.1	The interface number, `ifIndex`
{ ifEntry 2 } or **T**.2	The interface name or description
{ ifEntry 4 } or **T**.4	The interface MTU
{ ifEntry 5 } or **T**.5	The interface bitrate
{ ifEntry 10 } or **T**.10	The number of inbound octets (bytes)
{ ifEntry 11 } or **T**.11	The number of inbound unicast packets

Example (cont.)

- **IfEntry prefix:** T = 1.3.6.1.2.1.2.2.1

OID	table column
{ ifEntry 1 } or **T**.1	The interface number, ifIndex
{ ifEntry 2 } or **T**.2	The interface name or description
{ ifEntry 4 } or **T**.4	The interface MTU
{ ifEntry 5 } or **T**.5	The interface bitrate
{ ifEntry 10 } or **T**.10	The number of inbound octets (bytes)
{ ifEntry 11 } or **T**.11	The number of inbound unicast packets

- The interface number is used as a single-level OID suffix

ifIndex	name	MTU	bitrate	inOctets	inPackets
1	lo	16436	10,000,000	171	3
2	eth0	1500	100,000,000	37155014	1833455677
3	eth1	1500	100,000,000	0	0
4	ppp0	1420	0	2906687015	2821825

- The inOctets value of Eth0?
  - **OID:** T.10.3 = 1.3.6.1.2.1.2.2.1.10.3

Example

- **ipForward**

- The **nextHop** column is assigned the number 7
  - The nextHop for 10.38.0.0 thus has the OID T.7.10.38.0.0

dest	mask	metric	next_hop	type
0.0.0.0	0.0.0.0	1	192.168.1.1	indirect(4)
10.0.0.0	255.255.255.0	0	0.0.0.0	direct(3)
10.38.0.0	255.255.0.0	1	192.168.4.1	indirect(4)

Example

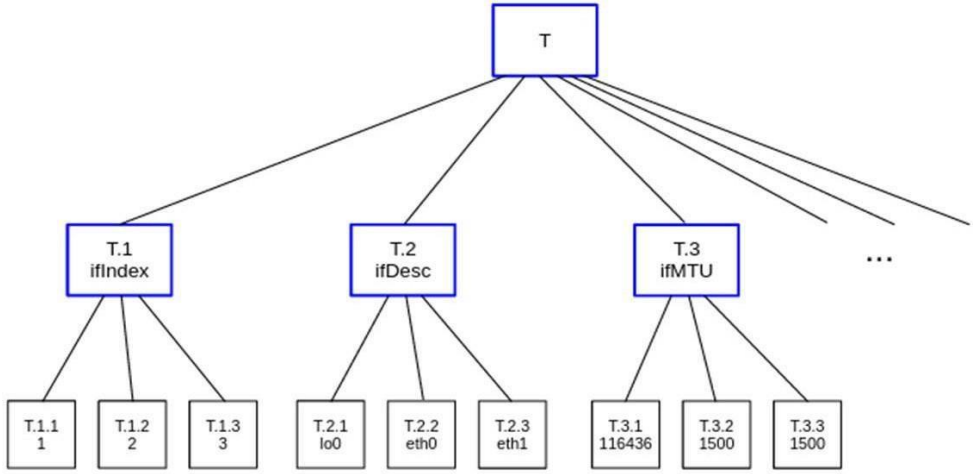
- **tcpConnTable**

localAddr	*localPort*	*remoteAddr*	*remotePort*	state
10.0.0.3	31895	147.126.1.209	993	established(5)
10.0.0.3	40113	74.125.225.98	80	timeWait(11)
10.0.0.3	20459	10.38.2.42	22	established(5)

- **The OID suffix for the state of the first connection:** .10.0.0.3.31895.147.126.1.209.993
- The state column is assigned the identifier 1, so this would all be appended to T.1

OID	table column
{ ifEntry 1 } or **T**.1	The interface number, `ifIndex`
{ ifEntry 2 } or **T**.2	The interface name or description
{ ifEntry 4 } or **T**.4	The interface MTU
{ ifEntry 5 } or **T**.5	The interface bitrate
{ ifEntry 10 } or **T**.10	The number of inbound octets (bytes)
{ ifEntry 11 } or **T**.11	The number of inbound unicast packets

ifIndex	name	MTU	bitrate	inOctets	inPackets
1	lo	16436	10,000,000	171	3
2	eth0	1500	100,000,000	37155014	1833455677
3	eth1	1500	100,000,000	0	0
4	ppp0	1420	0	2906687015	2821825



snmpgetnext()

- Allows a manager to walk through any subtree of the OID tree
- If the root of the subtree is prefix T, then the first call is **snmpgetnext(T)**
  - Returns ⟨oid1,value1⟩
- The next call is **snmpgetnext(oid1)**
  - Returns ⟨oid2,value2⟩
- The manager continues with the series of **snmpgetnext** calls until, finally, the subtree is exhausted
  - The agent returns either an error or else (more likely) an ⟨oidN,valueN⟩ for which oidN is no longer an extension of the original prefix

snmpgetnext()

- **Example:** start with the prefix 1.3.6.1.2.1.1 (the start of **system**)

  - 1.3.6.1.2.1.1.1 – sysDescr
  - 1.3.6.1.2.1.1.2 – sysObjectID
  - 1.3.6.1.2.1.1.3 – sysUpTime
  - 1.3.6.1.2.1.1.4 – sysContact
  - 1.3.6.1.2.1.1.5 – sysName
  - 1.3.6.1.2.1.1.6 – sysLocation
  - 1.3.6.1.2.1.1.7 - sysServices

- **snmpgetnext(1.3.6.1.2.1.1)** will return the pair: ⟨1.3.6.1.2.1.1.1.0, sysDescr_value⟩
  - The OID 1.3.6.1.2.1.1.1.0 is the first leaf node below the interior node 1.3.6.1.2.1.1.

- **snmpgetnext(1.3.6.1.2.1.1.1.0)** will return ⟨1.3.6.1.2.1.1.2.0, sysObjectID_value⟩

snmpwalk()

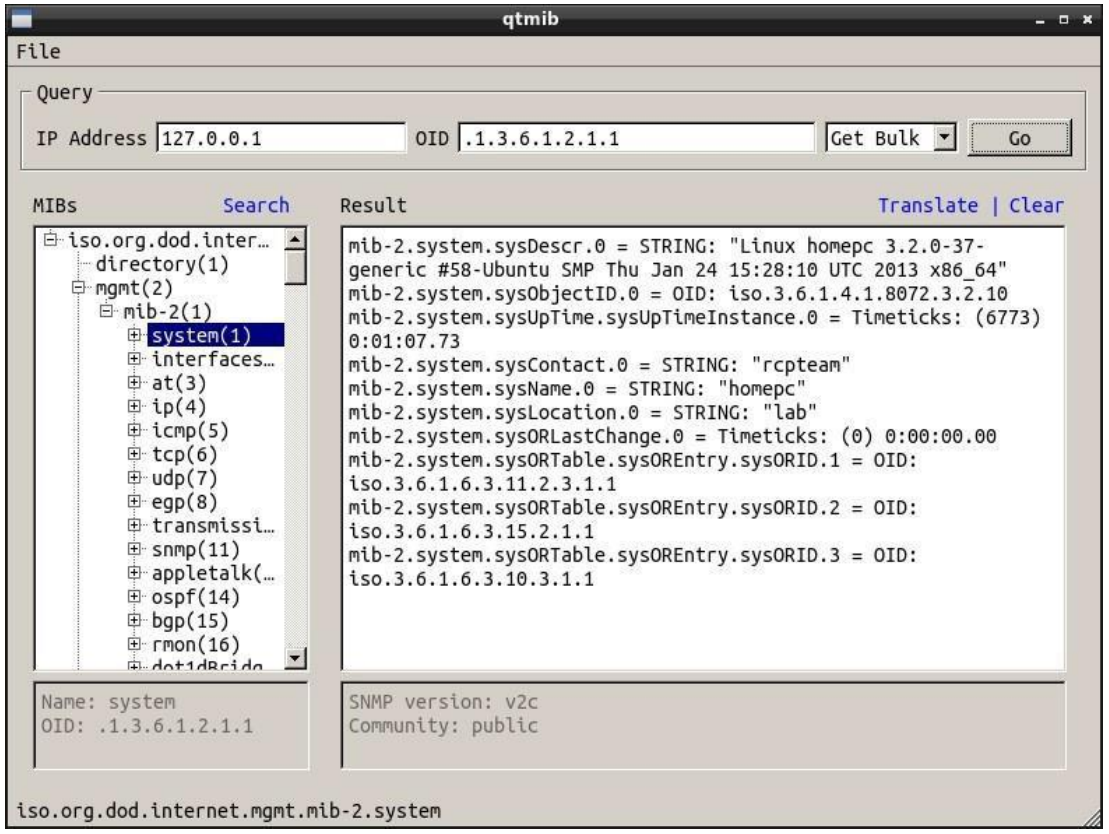
- Takes an OID representing the root of a subtree, and returns everything in that subtree.
- **Example:**
  - snmpwalk -v 1 -c tengwar localhost 1.3.6.1.2.1.2.2
  - Note: 1.3.6.1.2.1.2.2 = ifTable.

snmpbulkget()

- SNMPv2 command
- An extension of **snmpgetnext()**
- A manager includes an integer N in its request and the agent then iterates the action of **snmpgetnext()** N times
- **Example:** snmpbulkget -Cr2 -v2c -c tengwar localhost 1.3.6.1.2.1.2.2
- All N results (which can each represent an entire row) can then be returned in a single operation.

qtmib

- https://qtmib.sourceforge.net/
- SNMP MIB browser for Linux platforms
- Allows the user to query any SNMP-enabled device
- Implements SNMPv1 and SNMPv2c
- Supports a large number of MIBs
- Private MIBs can be installed



SNMP Simulator

- …

Homework #2

- …

Sources

- Essential SNMP: Chapters 1+2 + 3
- http://intronetworks.cs.luc.edu/current/html/netmgmt.html
- Slides of the Network Management Course (Prof. Rolf Stadler, KTH)

Questions?

Network Programming

9) Switching and Routing Essentials

Dr. Hassan Almazini

A – Target Population: For students of Second Year
Technological Institute of Basra
Dep. Of Computer network and software technologies

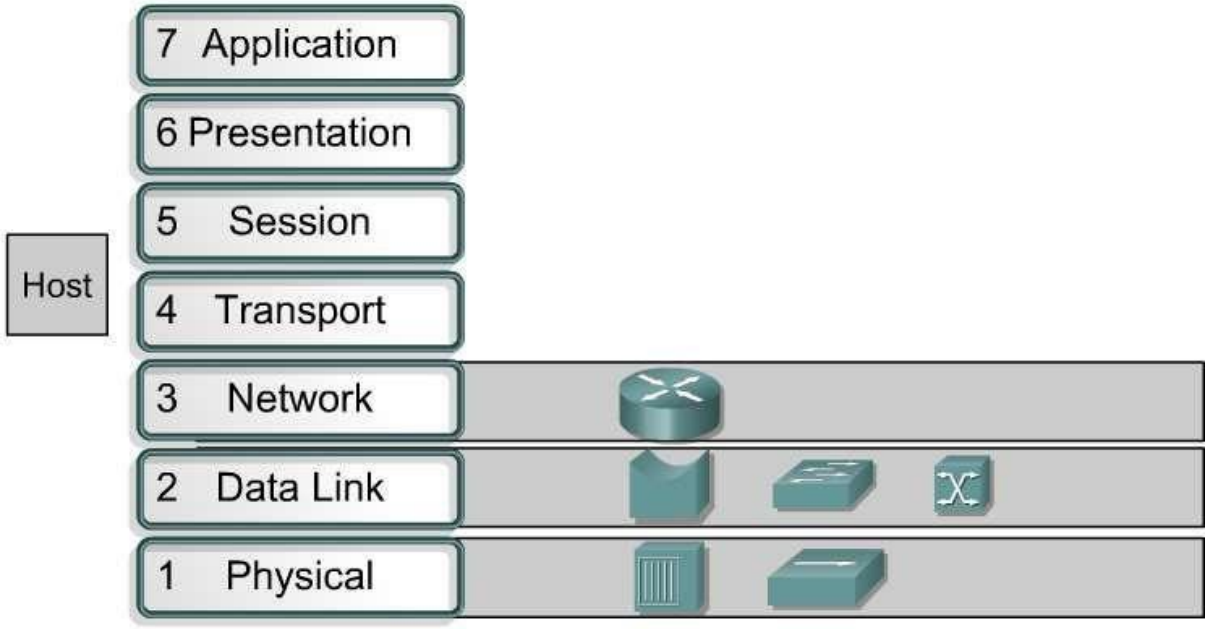
9 **/ B – Rationale:** Understanding how data is transferred within and between networks is fundamental to network programming. This unit provides foundational knowledge on switching and routing technologies that ensure efficient data delivery.

9 **/ C – Central Idea:**

1. Introduction to network switching
2. Types of switches and switching methods
3. Basics of routing and types of routers
4. Routing tables and protocols

9 **/ D – Performance Objectives:** After studying this unit, the student will be able to:

1. Define switching and explain its role in local area networks.
2. Differentiate between store-and-forward, cut-through, and fragment-free switching.
3. Explain how routers direct traffic between different networks.
4. Interpret routing tables and distinguish between static and dynamic routing protocols.
5. Describe real-time network monitoring practices.

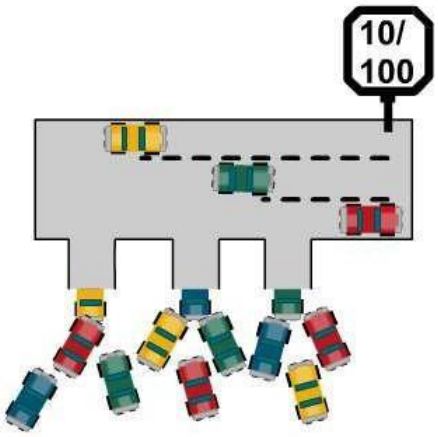


Routers

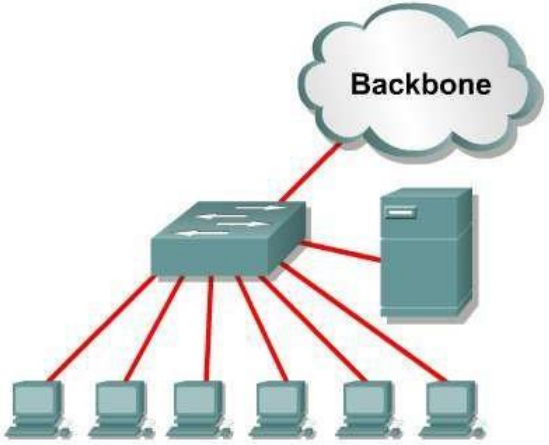
Switches, Bridges

Hub, Repeaters

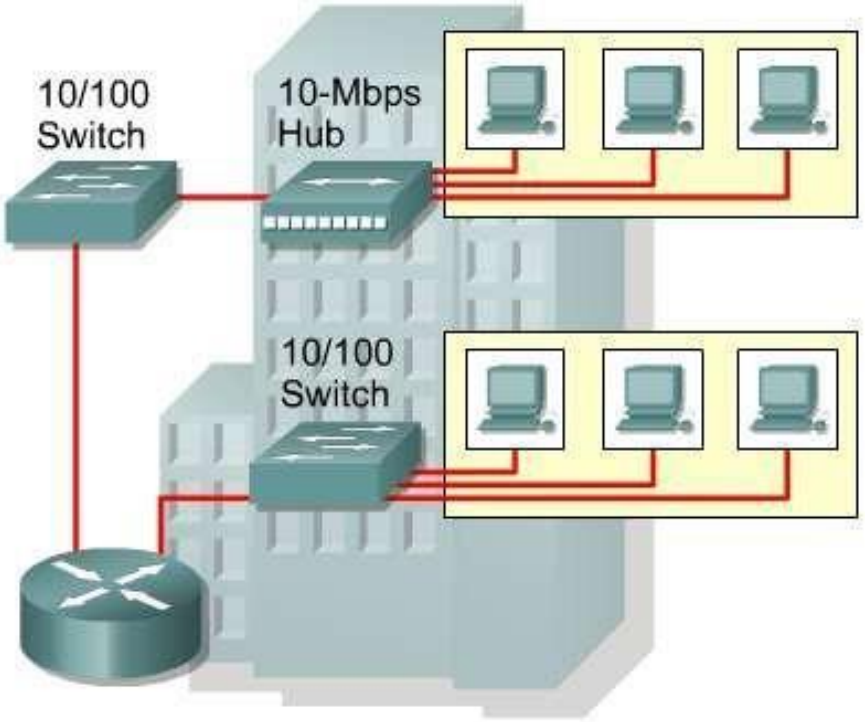
Switches



Multiple devices sending at the same time



Each node has 10/100 Mbps

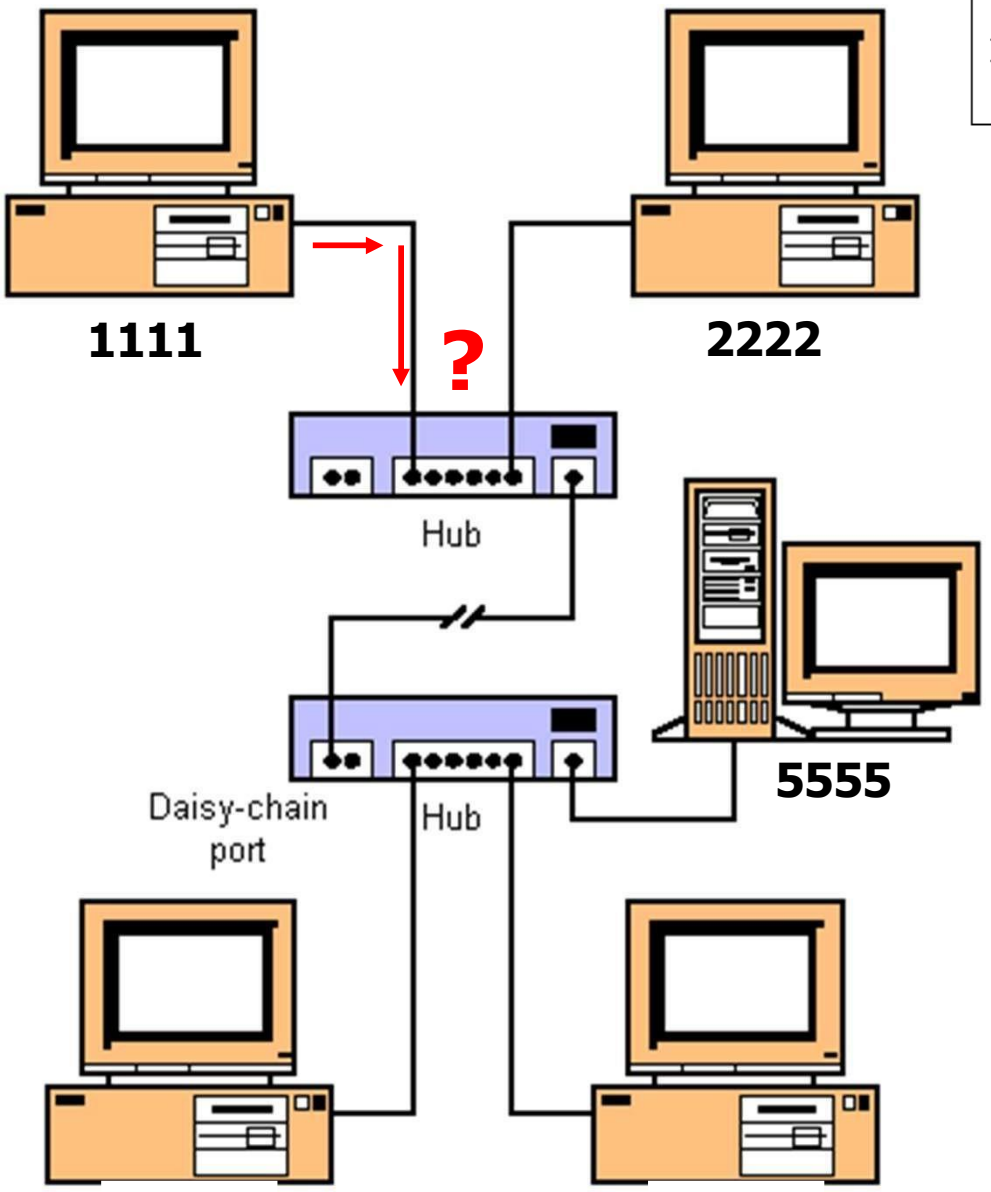


- Switches create a virtual circuit between two connected devices, establishing a dedicated communication path between two devices.
- Switches on the network provide **microsegmentation.**
- This allows maximum utilization of the available bandwidth.
- A switch is also able to facilitate multiple, simultaneous virtual circuit connections.
- Broadcast frames to all connected devices on the network.

Router

- A router is a Layer 3 device.
- Used to “route” traffic between two or more Layer 3 networks.
- Routers make decisions based on groups of network addresses, or classes, as opposed to individual Layer 2 MAC addresses.
- Routers use routing tables to record the Layer 3 addresses of the networks that are directly connected to the local interfaces and network paths learned from neighboring routers.
- Routers are **not** compelled to forward broadcasts.

Sending and receiving Ethernet frames via a hub



Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 1111

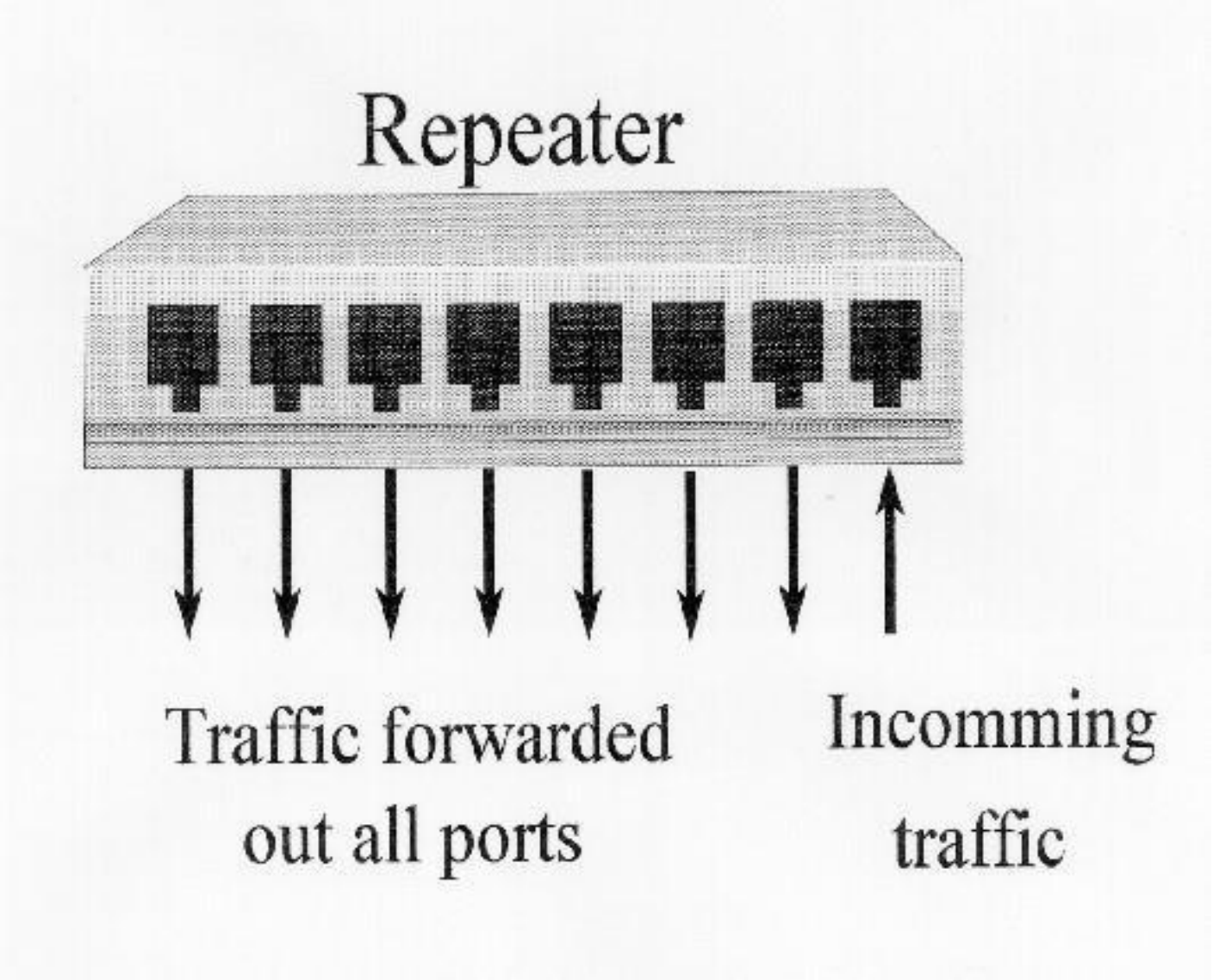
- So, what does a hub do when it receives information?
- Remember, a hub is nothing more than a multiport repeater.

Sending and receiving Ethernet frames via a hub

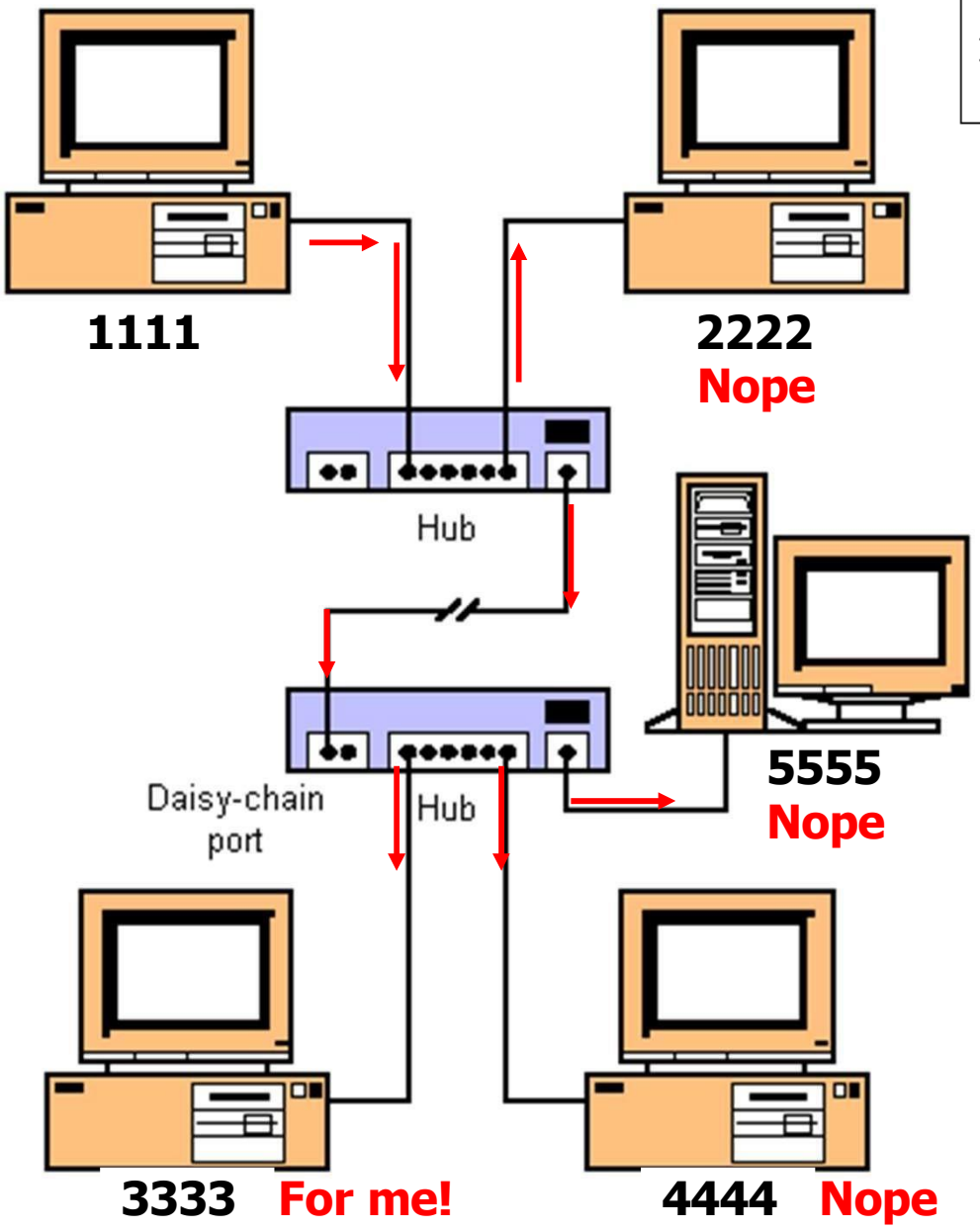
3333 **4444**

Sending and receiving Ethernet frames via a hub

Hub or



Sending and receiving Ethernet frames via a hub

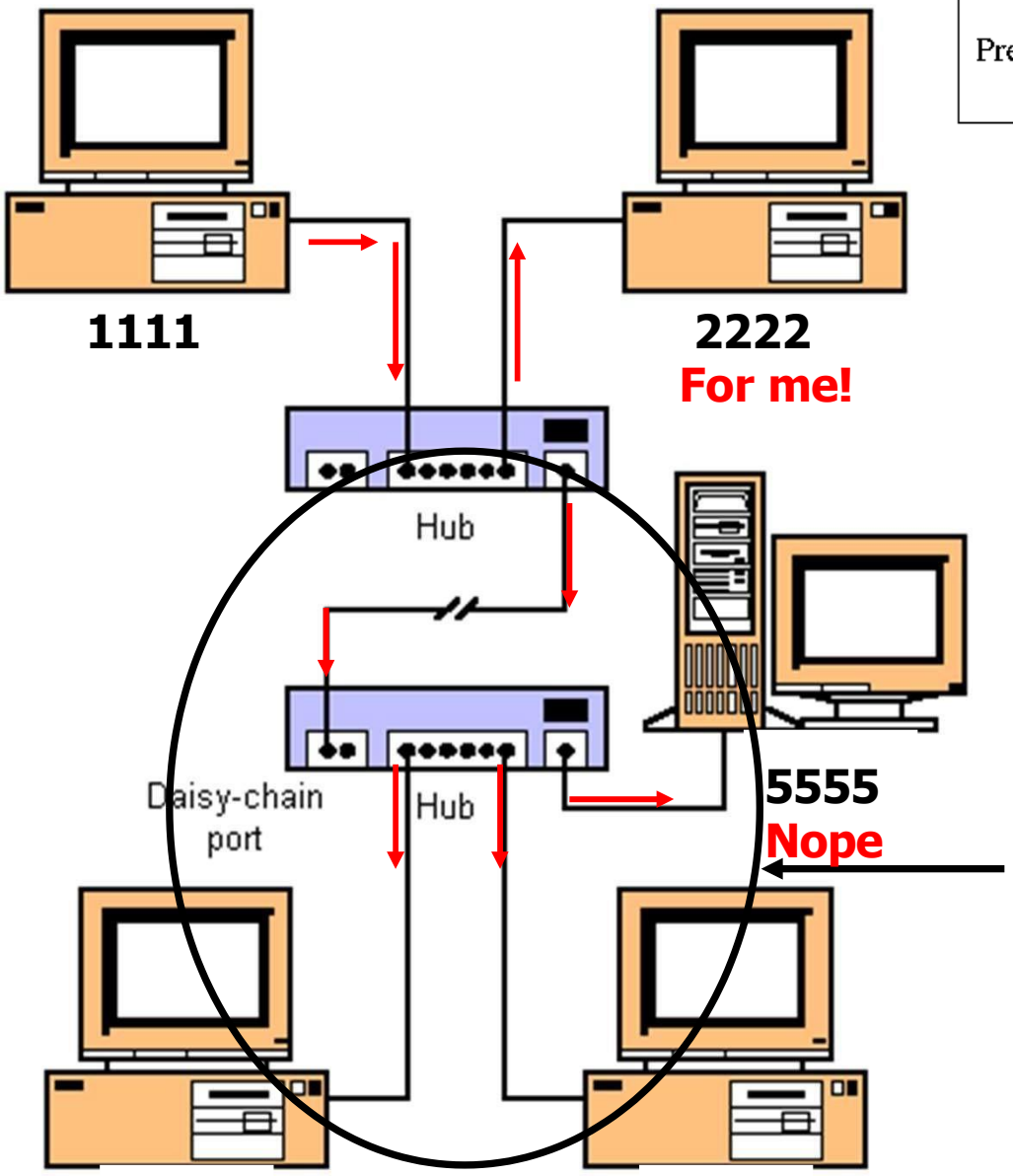


Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 1111

- The hub will **flood** it out all ports except for the incoming port.
- Hub is a layer 1 device.
- A hub does NOT look at layer 2 addresses, so it is fast in transmitting data.
- Disadvantage with hubs: A hub or series of hubs is a single **collision domain**.
- A collision will occur if any two or more devices transmit at the same time within the collision domain.

Sending and receiving Ethernet frames via a hub



Preamble	Destination Address	Source Address	Type	Data	Pad	CRC
	2222	1111				

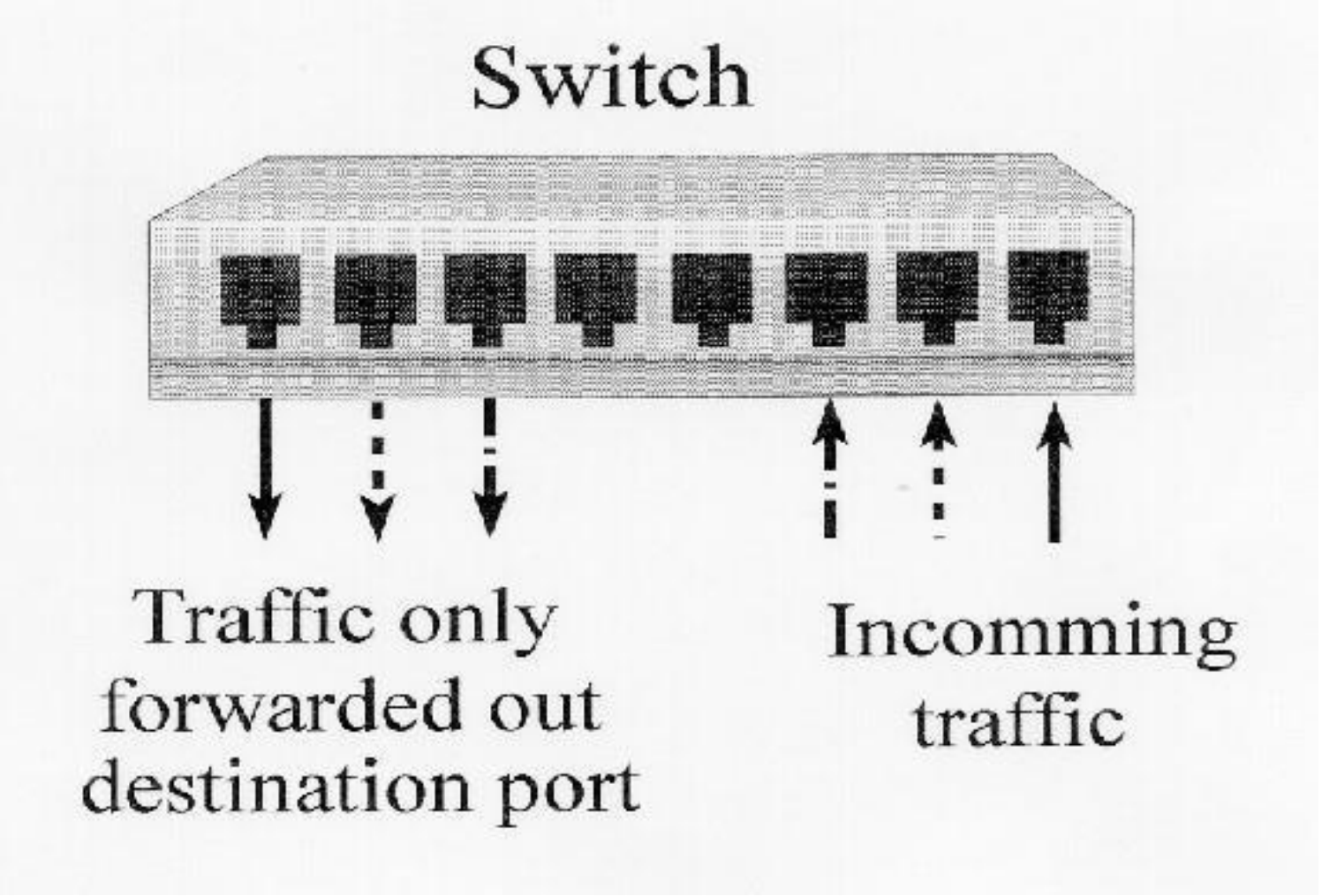
- Another disadvantage with hubs is that is take up unnecessary bandwidth on other links.

Wasted bandwidth

Sending and receiving Ethernet frames via a hub

3333 Nope 4444 Nope

Sending and receiving Ethernet frames via a switch



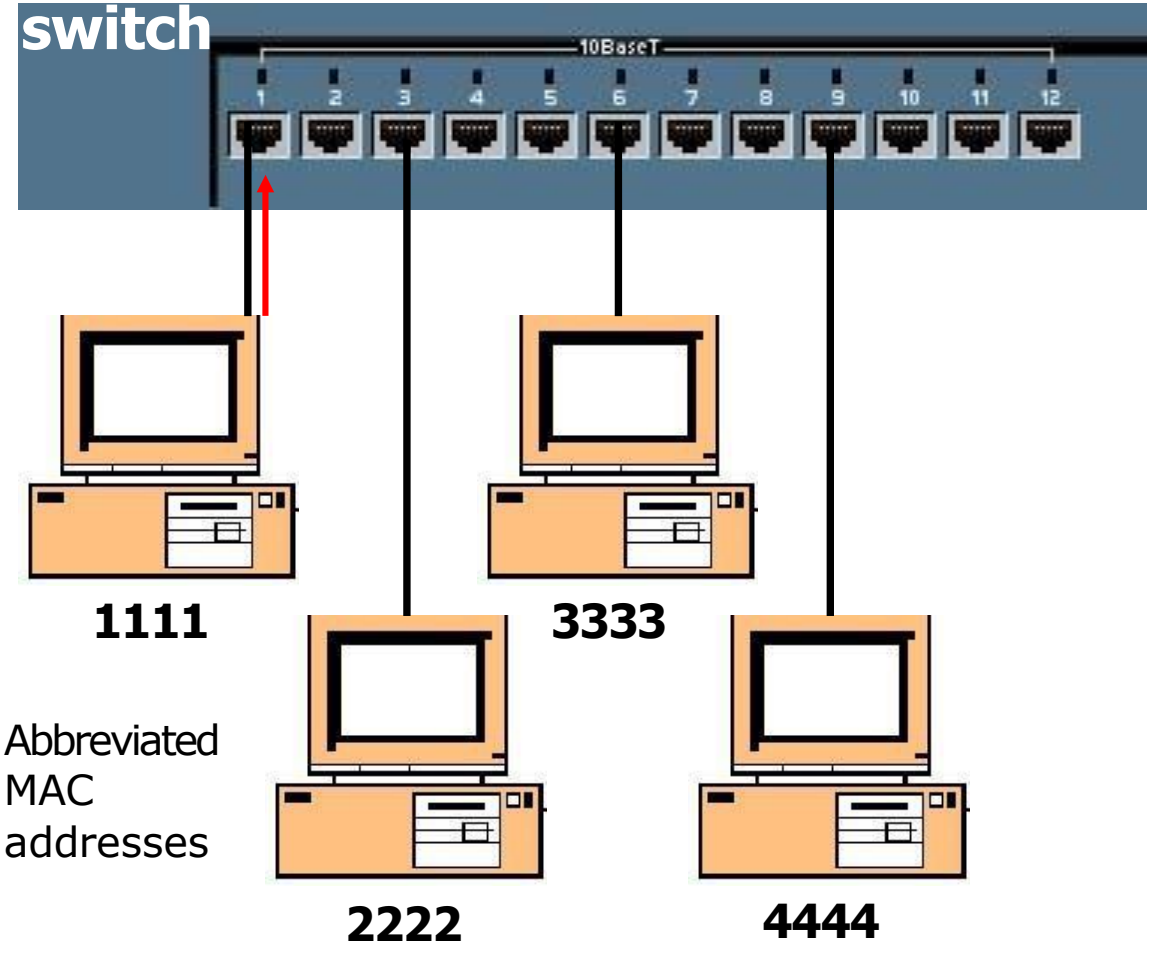
Sending and receiving Ethernet frames via a switch

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 1111



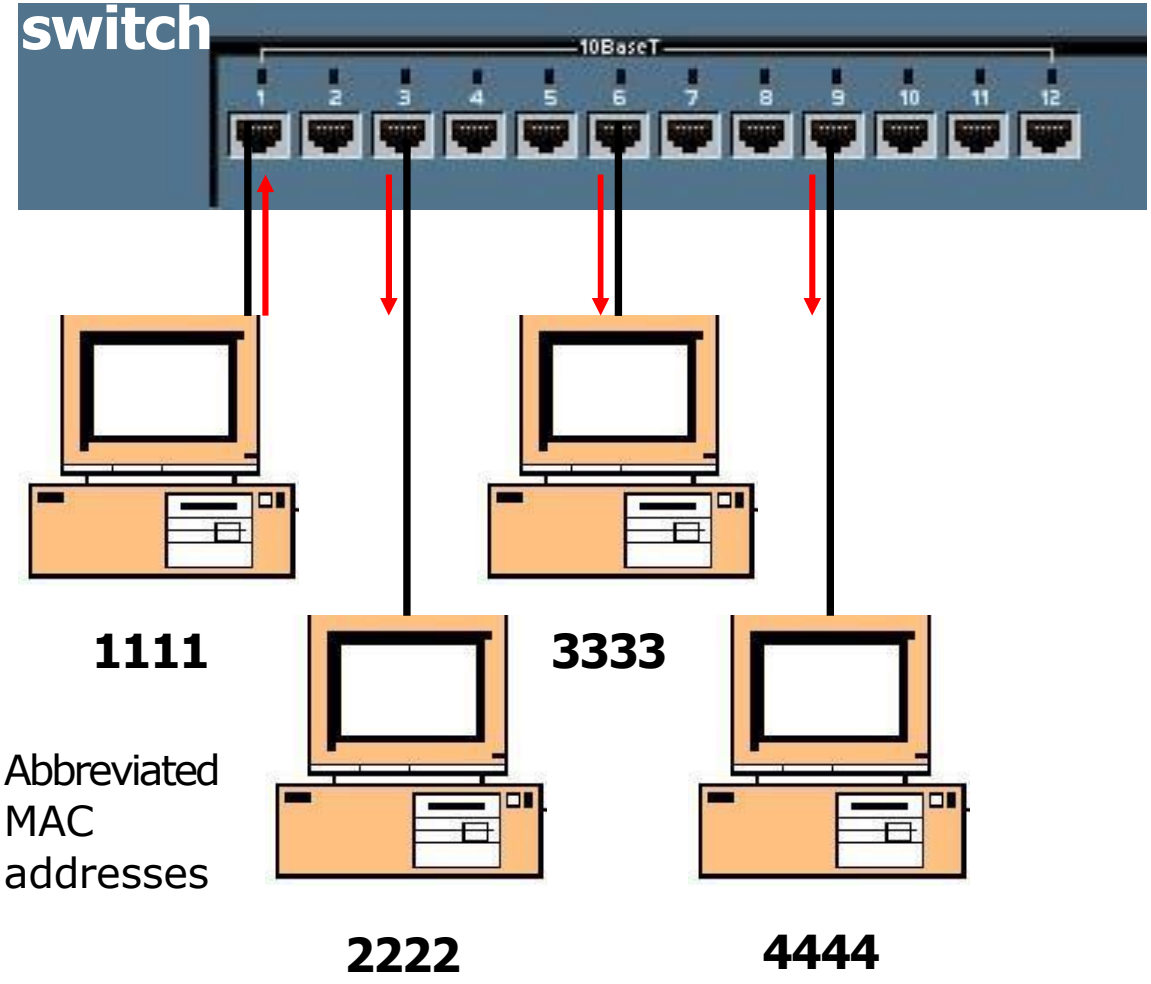
- Switches are also known as **learning bridges** or **learning switches**.
- A switch has a source address table in cache (RAM) where it stores source MAC address after it learns about them.
- A switch receives an Ethernet frame it searches the source address table for the Destination MAC address.
- If it finds a match, it **filters** the frame by only sending it out that port.
- If there is not a match if **floods** it out all ports.

No Destination Address in table, Flood

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.
1	1111		

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC
	3333	1111				



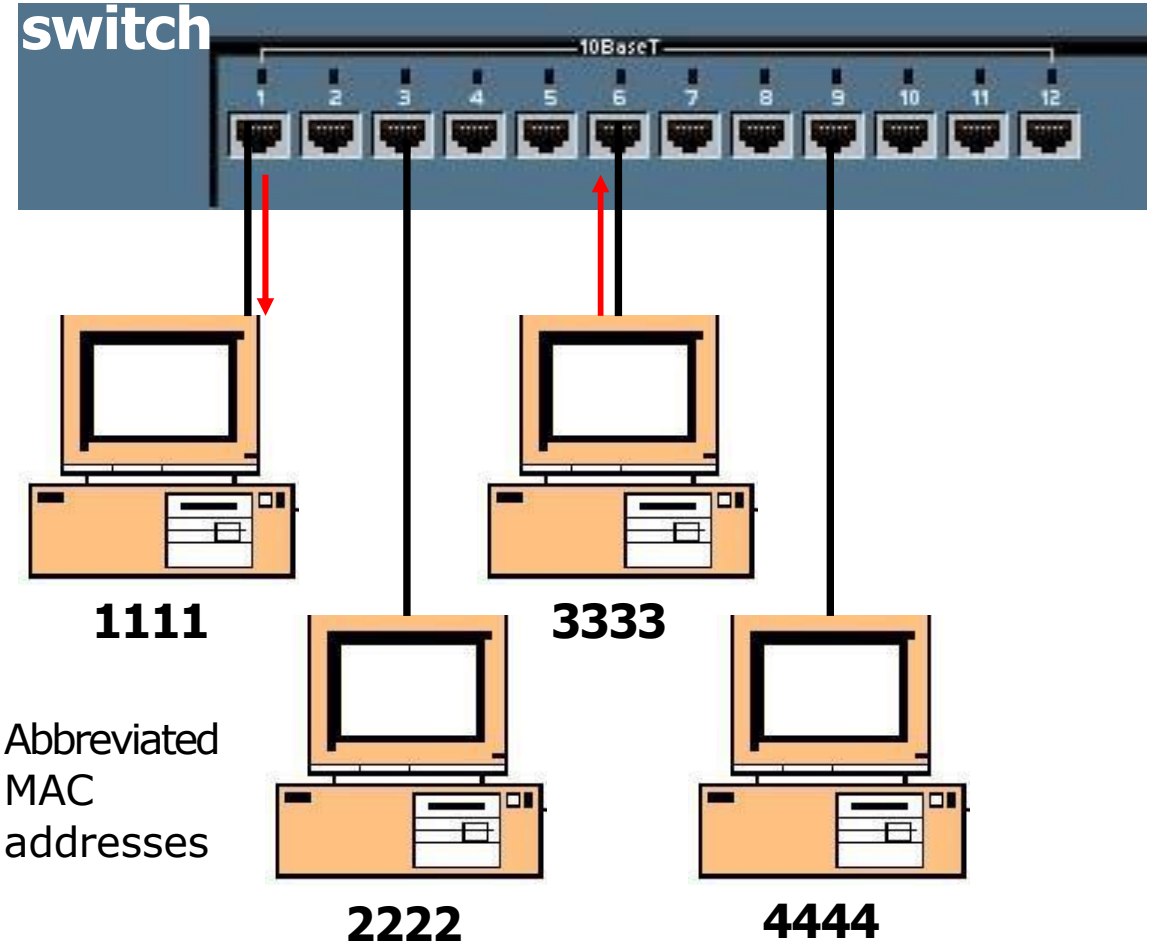
- How does it learn source MAC addresses?
- First, the switch will see if the SA (1111) is in it's table.
- If it is, it resets the timer (more in a moment).
- If it is NOT in the table it adds it, with the port number.
- Next, in our scenario, the switch will flood the frame out all other ports, because the DA is not in the source address table.

Destination Address in table, Filter

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.
1	1111	6	3333

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC
	1111	3333				



- Most communications involve some sort of client-server relationship or exchange of information. (You will understand this more as you learn about TCP/IP.)
- Now 3333 sends data back to 1111.
- The switch sees if it has the SA stored.
- It does NOT so it adds it. (This will help next time 1111 sends to 3333.)
- Next, it checks the DA and in our case it can filter the frame, by sending it only out port 1.

Destination Address in table, Filter

Source Address Table

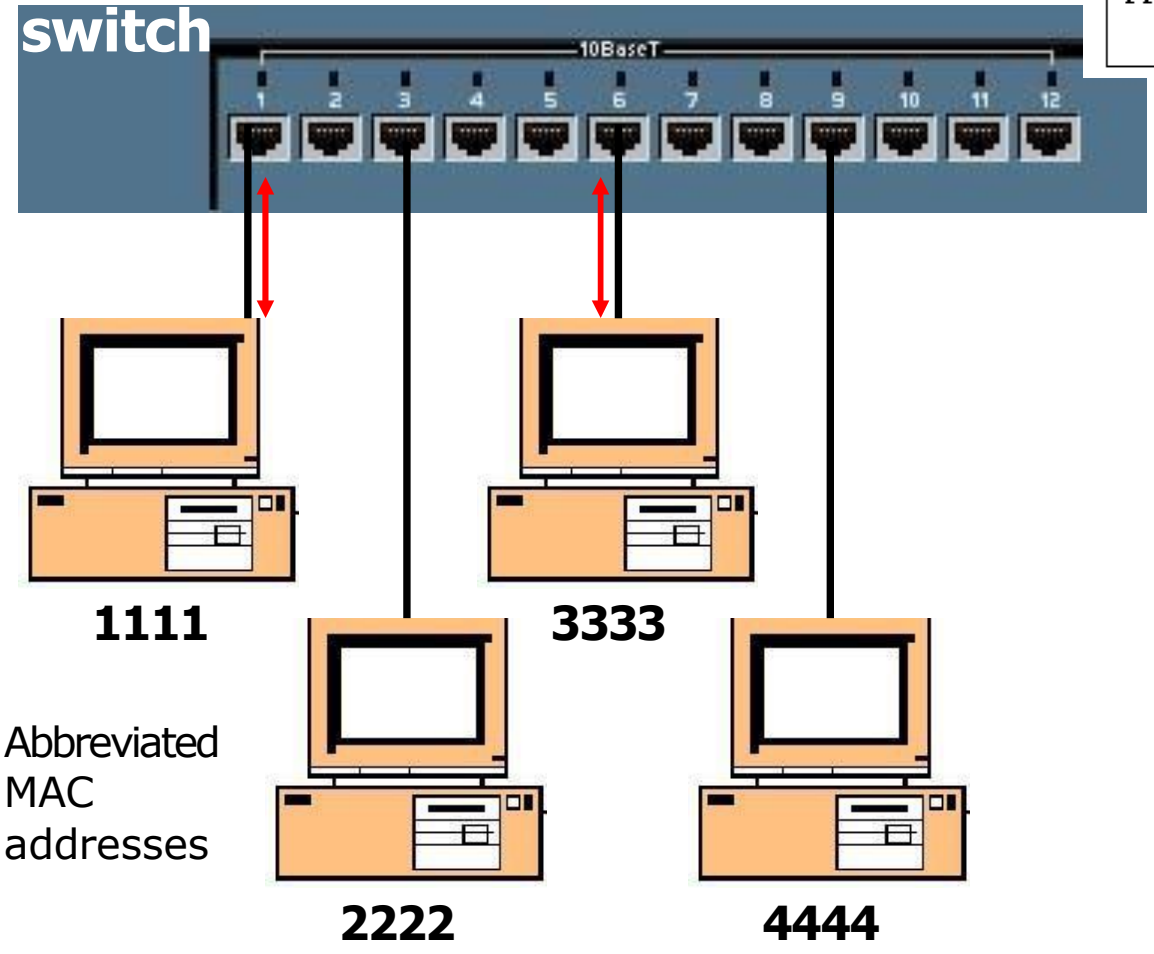
Port	Source MAC Add.	Port	Source MAC Add.
1	1111	6	3333

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 1111

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

1111 3333



Abbreviated MAC addresses

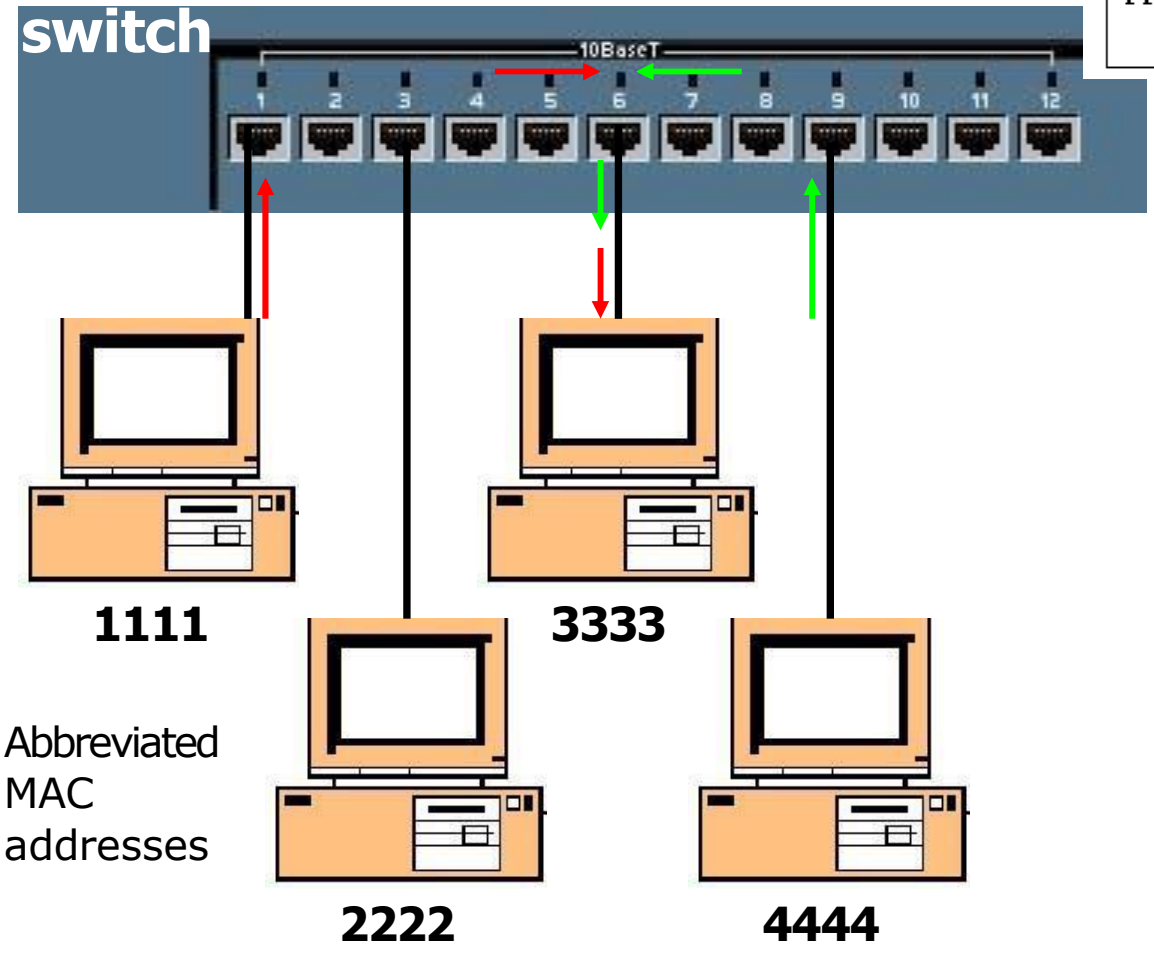
- Now, because both MAC addresses are in the switch's table, any information exchanged between 1111 and 3333 can be sent (filtered) out the appropriate port.

- **What happens when two devices send to same destination?**
- **What if this was a hub?**
- **Where is (are) the collision domain(s) in this example?**

No Collisions in Switch, Buffering

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.
1	1111	6	3333
9	4444		



Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 **1111**

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 **4444**

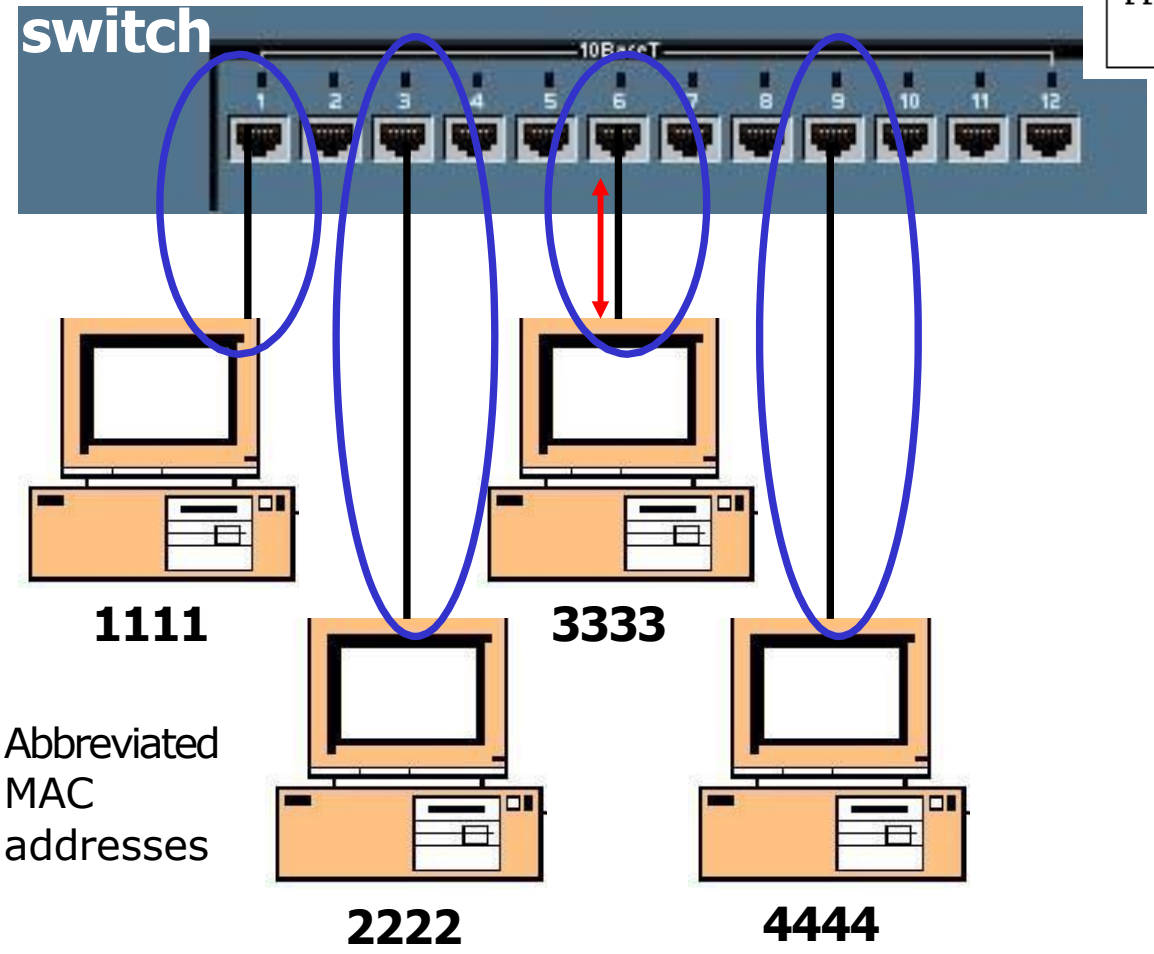
- Unlike a hub, a collision does NOT occur, which would cause the two PCs to have to retransmit the frames.
- Instead the switch buffers the frames and sends them out port #6 one at a time.
- The sending PCs have no idea that their was another PC wanting to send to the same destination.

Collision Domains

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.
1	1111	6	3333
9	4444		

Collision Domains



Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 **1111**

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

3333 **4444**

- When there is only one device on a switch port, the collision domain is only between the PC and the switch.
- With a **full-duplex** PC and switch port, there will be no collision, since the devices and the medium can send and receive at the same time.

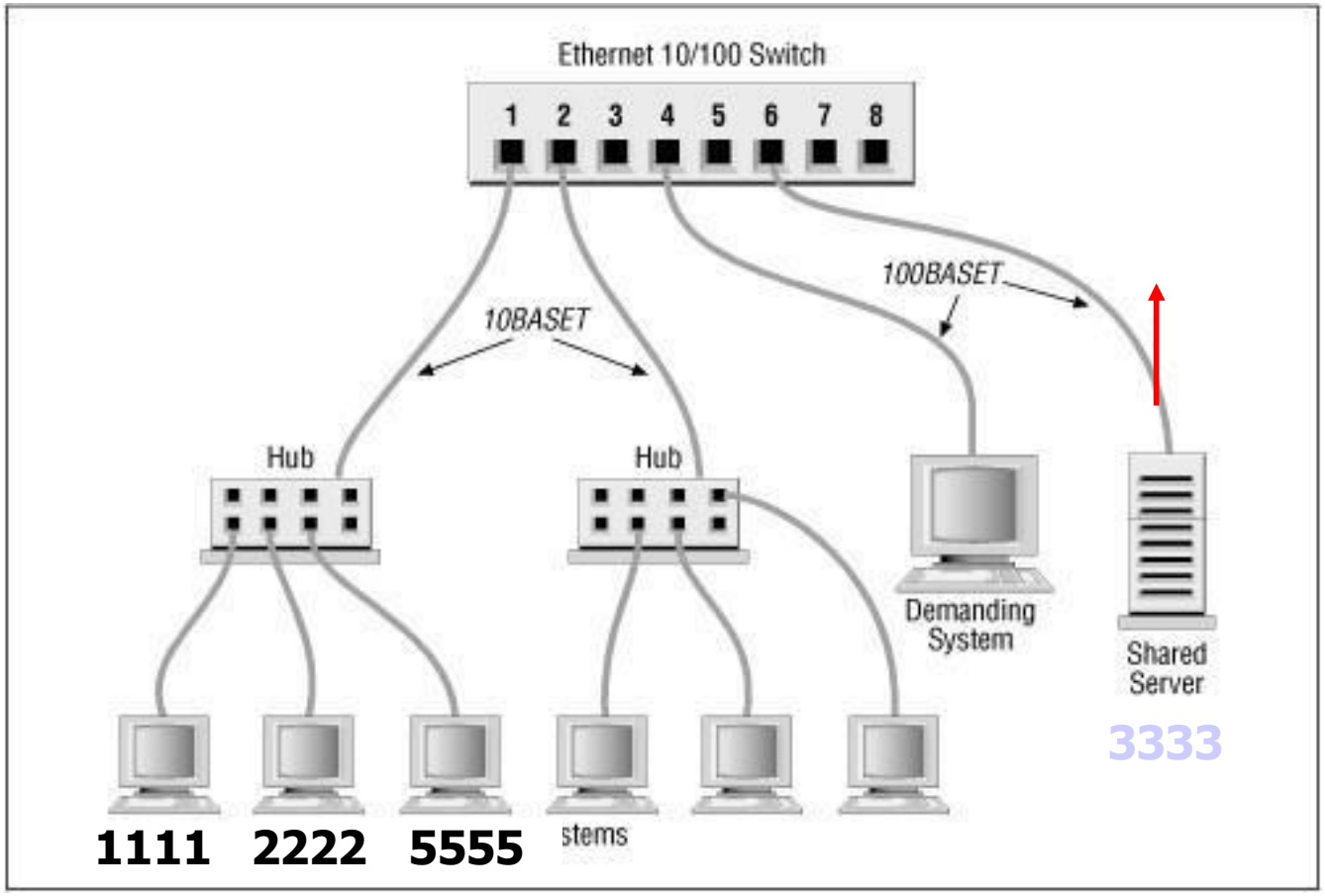
What happens here?

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.
1	**1111**	6	3333
1	**2222**	**1**	**3333**

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

1111 **3333**



- Notice the Source Address Table has multiple entries for port #1.

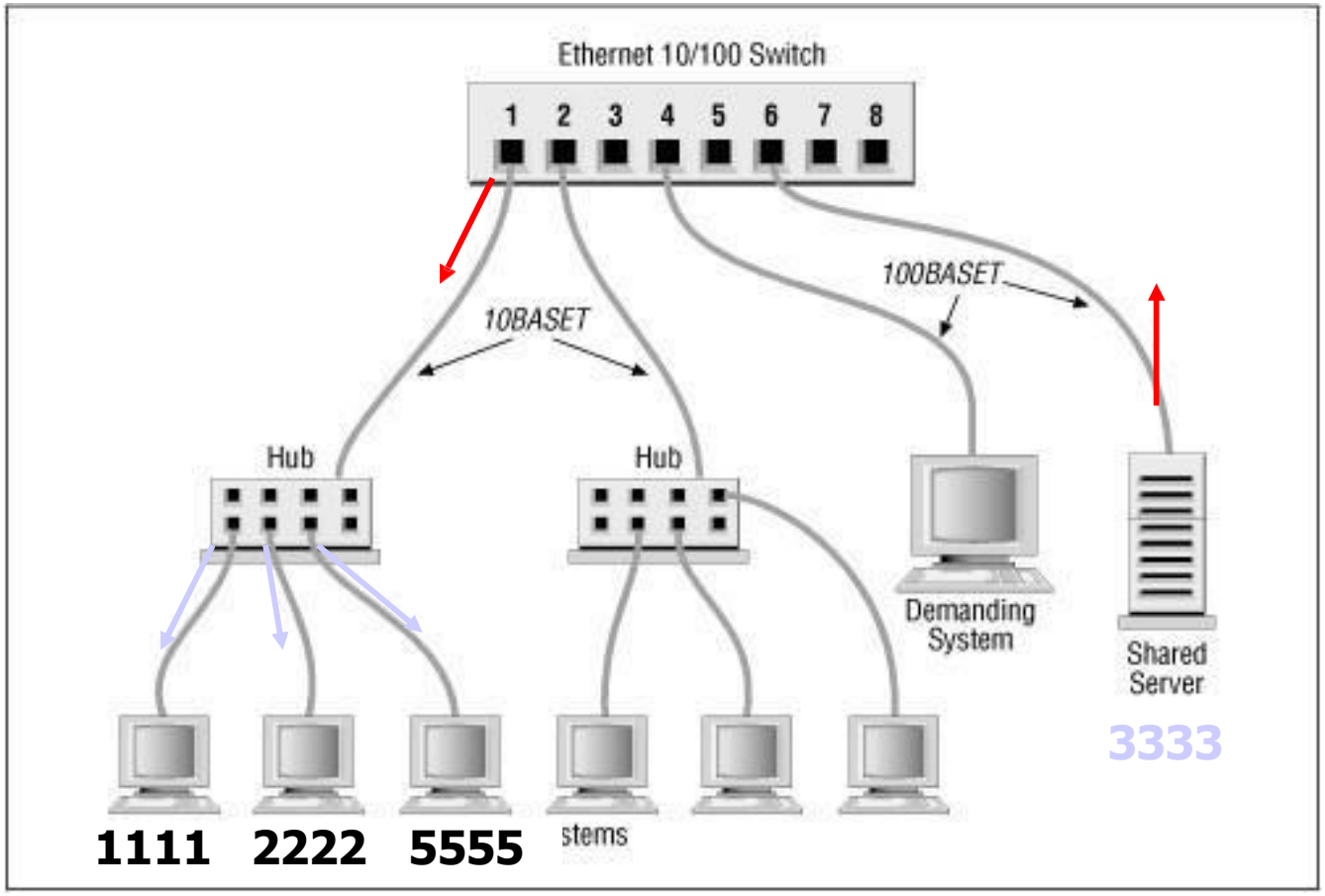
What happens here?

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.
1	**1111**	6	3333
1	**2222**	**1**	**5555**

Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

1111 **3333**



- The switch filters the frame out port #1.
- But the hub is only a layer 1 device, so it floods it out all ports.
- Where is the collision domain?

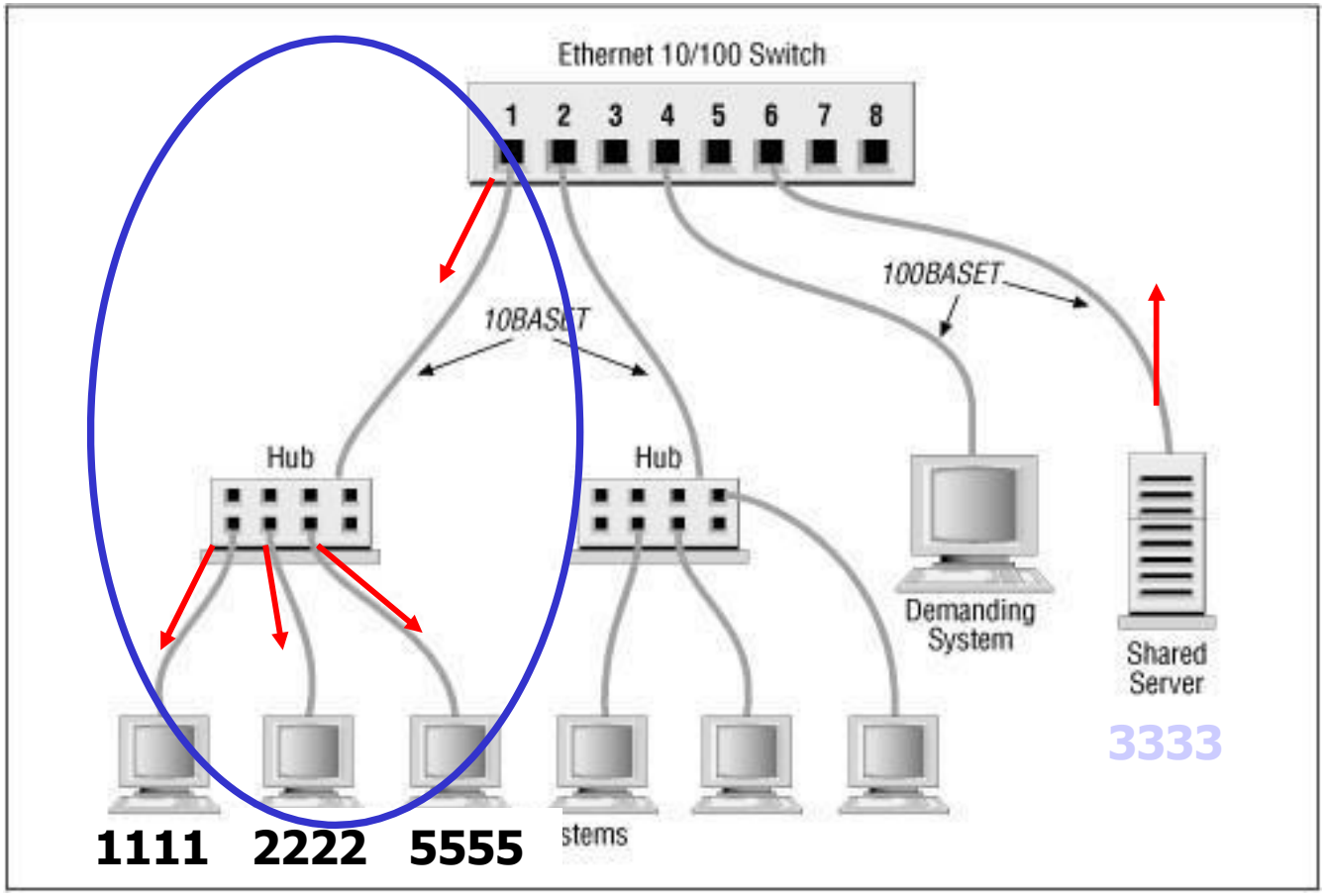
What happens here?

Source Address Table

Port	Source MAC Add.	Port	Source MAC Add.
1	**1111**	6	3333
1	**2222**	**1**	**5555**

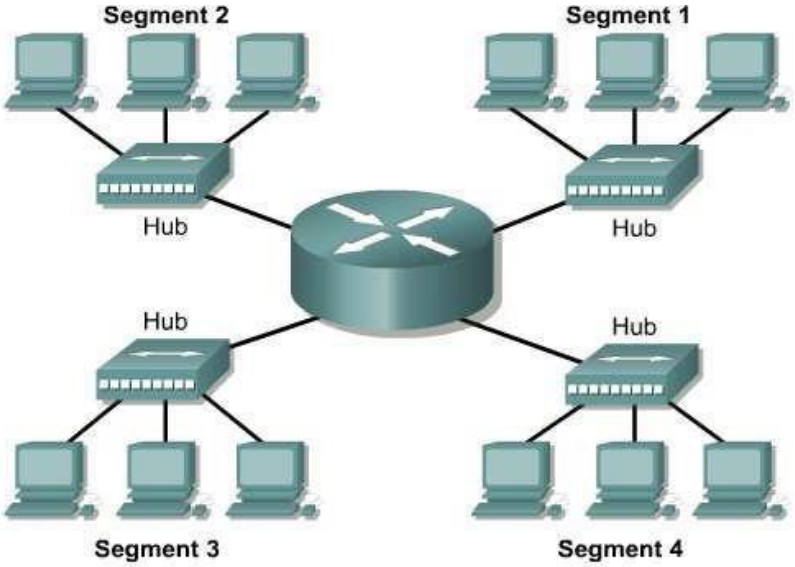
Preamble	Destination Address	Source Address	Type	Data	Pad	CRC

1111 **3333**



Collision Domain

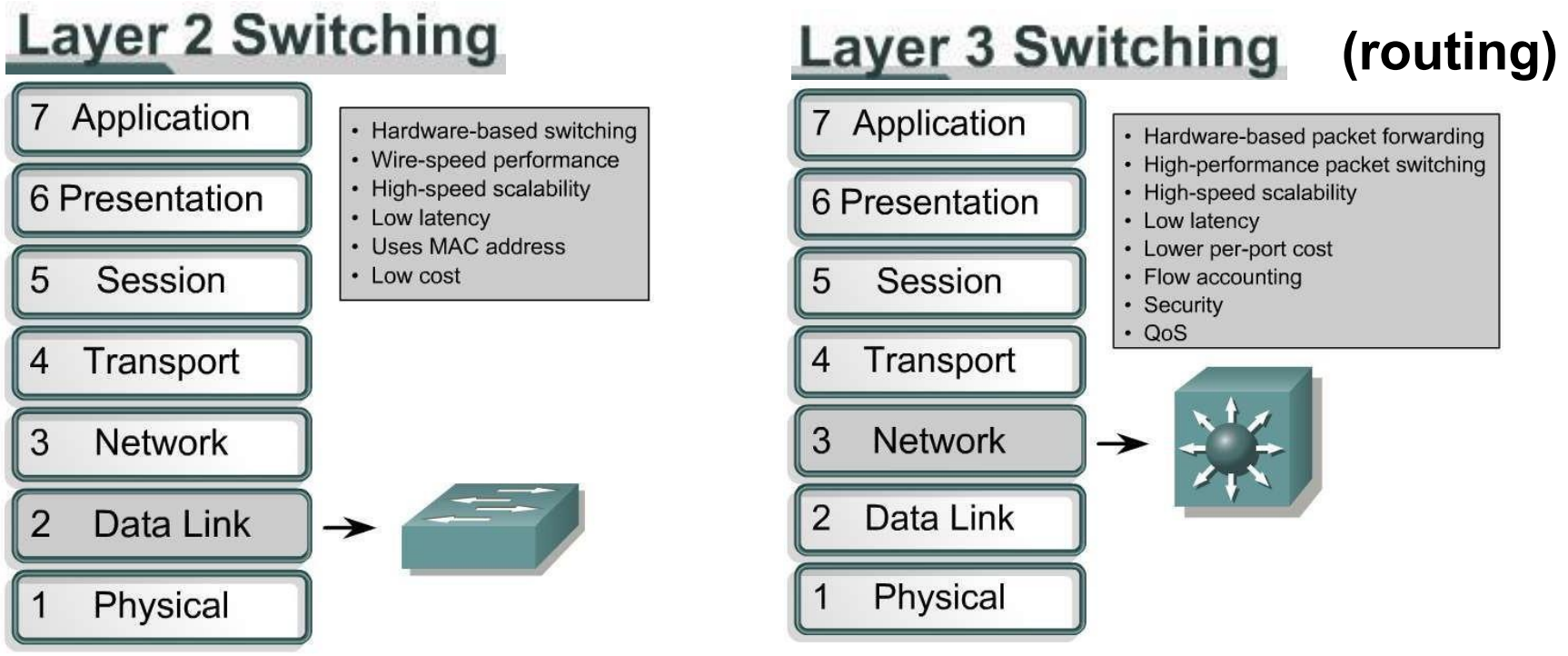
LAN segmentation with routers



- More manageable, greater functionality, multiple active paths
- Smaller broadcast domains
- Operates at Layer 3

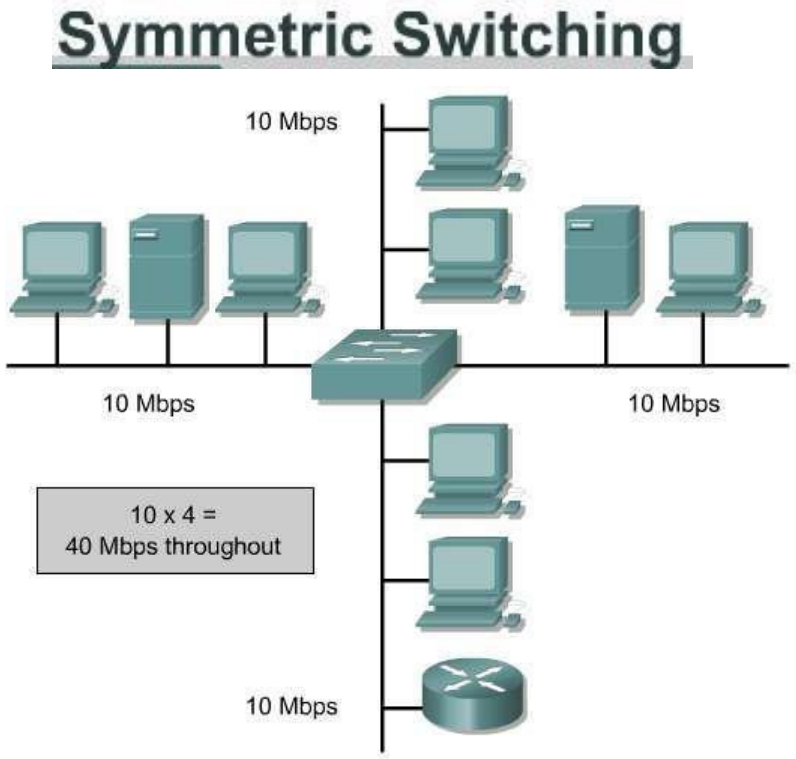
- Router operates at the network layer and uses the IP address to determine the best path to the destination node.
- Bridges and switches provide segmentation within a single network or subnetwork.
- **Routers provide connectivity between networks and subnetworks.**
- **Routers also do not forward broadcasts while switches and bridges must forward broadcast frames.**

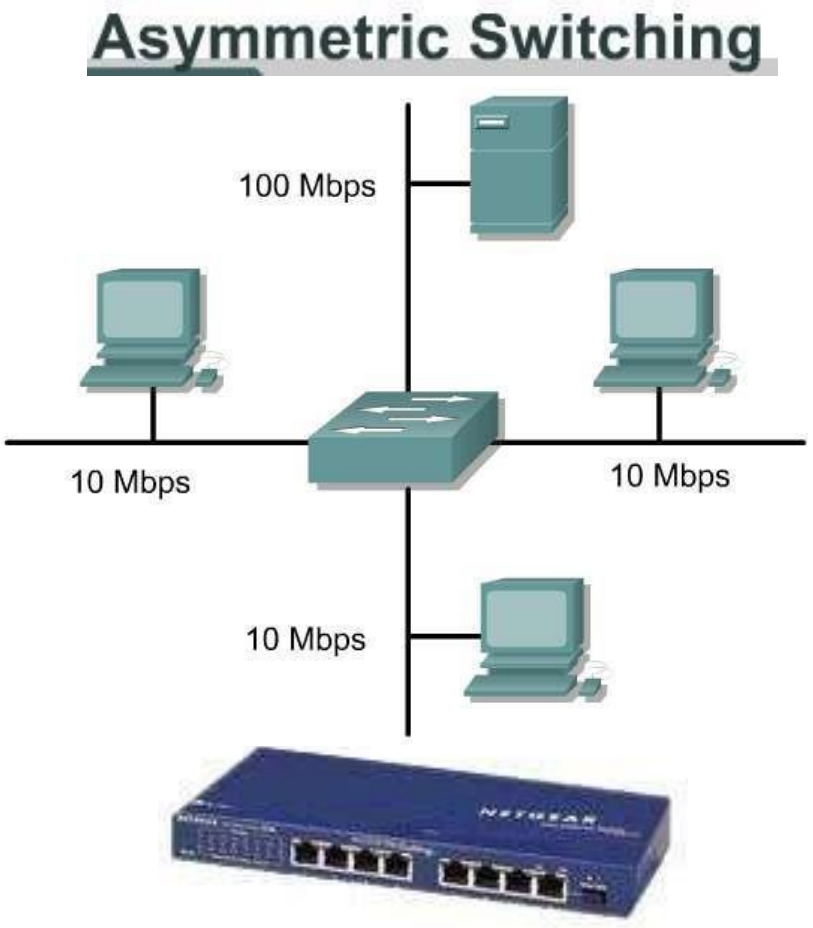
Layer 2 and layer 3 switching

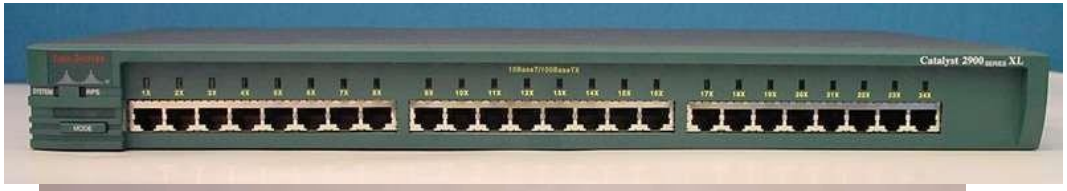


- The Layer 3 header information is examined and the packet is forwarded based on the IP address.

Symmetric and asymmetric switching

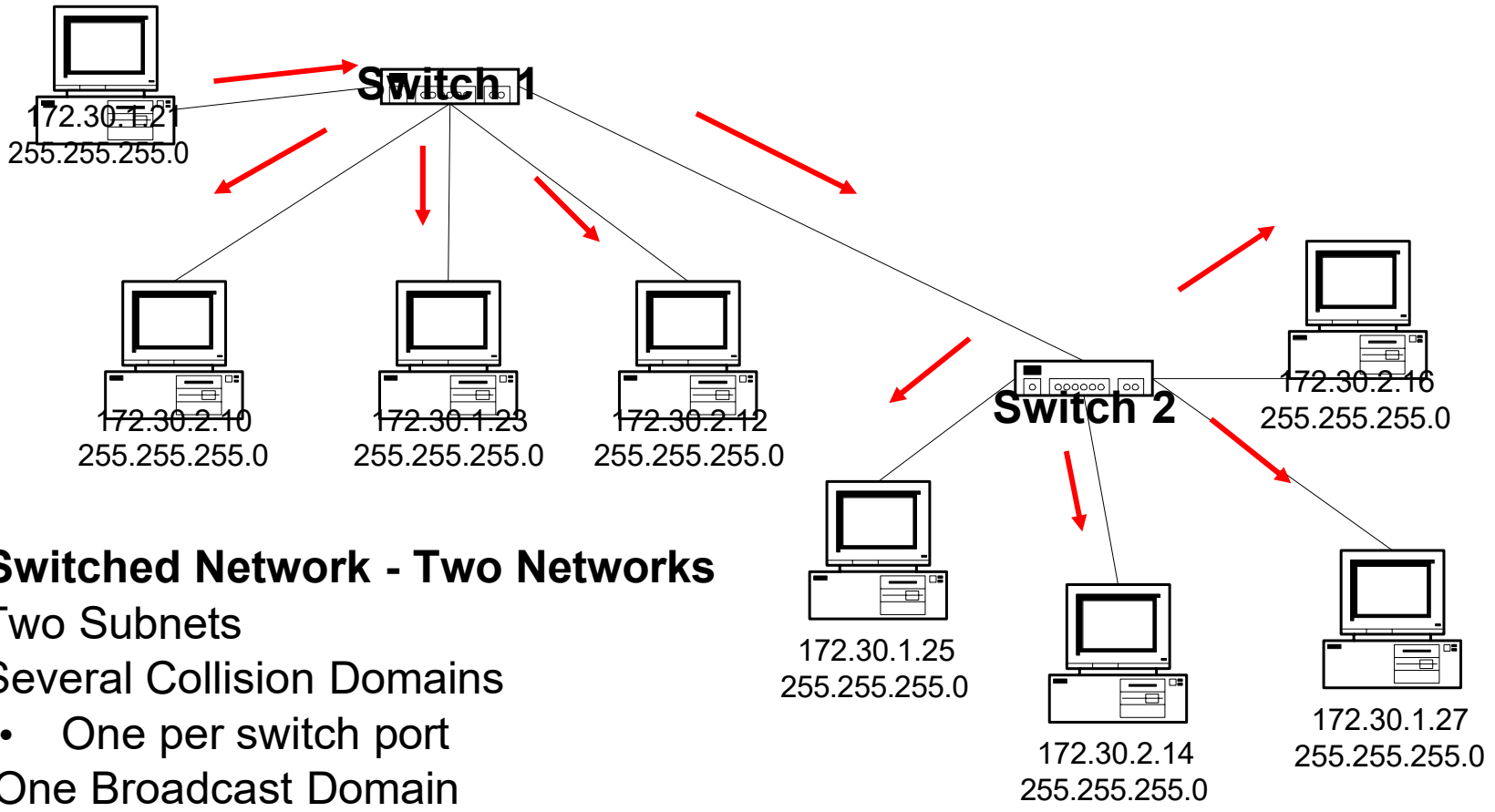






Note: Most switches are now 100/1000, which allow you to use them symmetrically or asymmetrically.

Broadcast domains



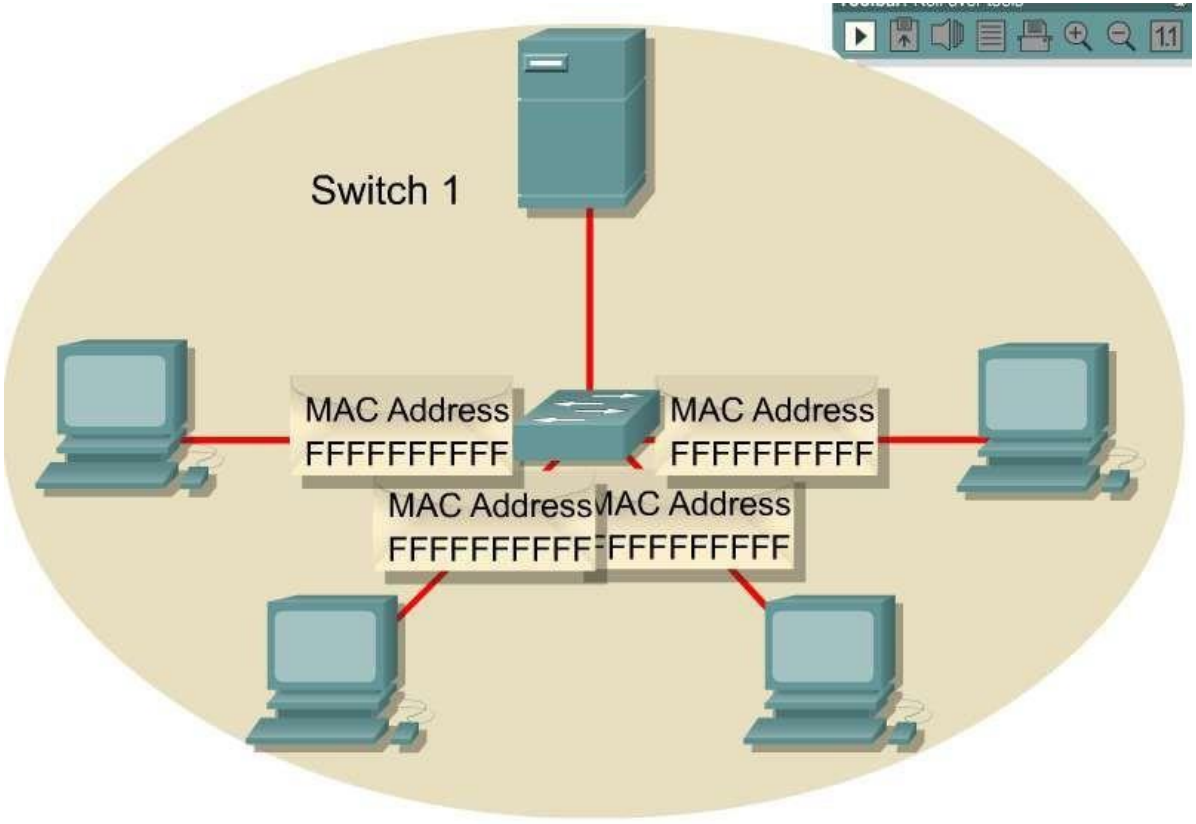
All Switched Network - Two Networks

- Two Subnets
- Several Collision Domains
  - One per switch port
- One Broadcast Domain

- Even though the LAN switch reduces the size of collision domains, all hosts connected to the switch are still in the same broadcast domain.
- Therefore, a broadcast from one node will still be seen by all the other

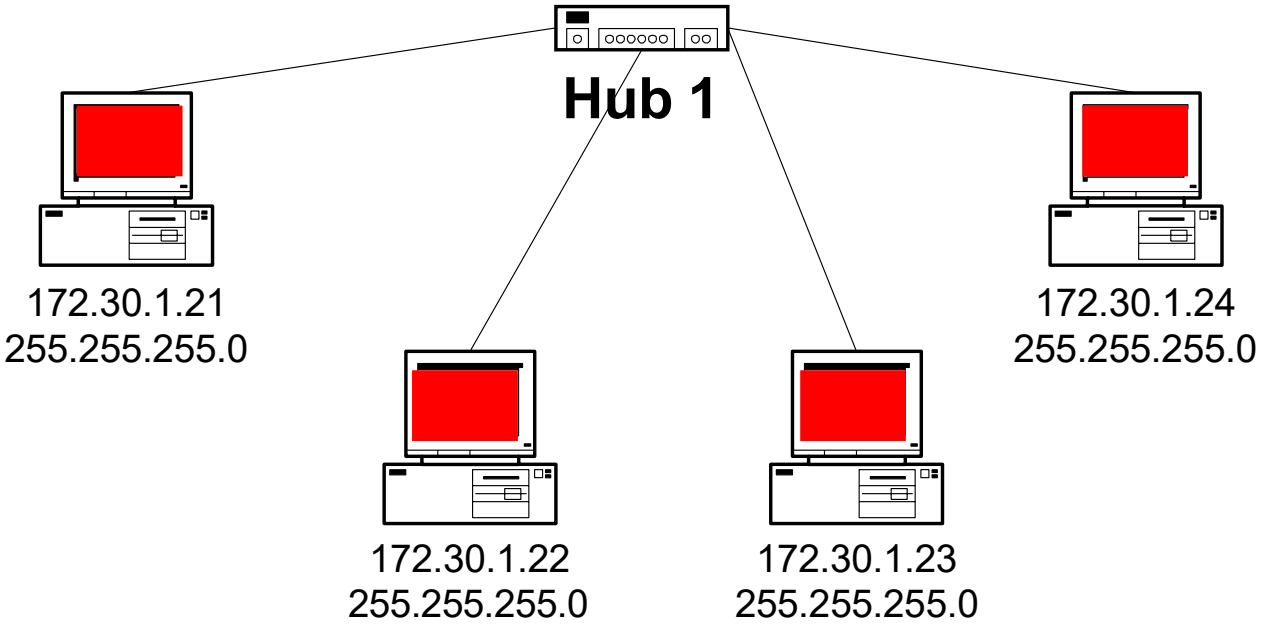
nodes connected through the LAN switch.

Switches and broadcast domains



- When a device wants to send out a Layer 2 broadcast, the destination MAC address in the frame is set to all ones.
- A MAC address of all ones is FF:FF:FF:FF:FF:FF in hexadecimal.
- By setting the destination to this value, all the devices will accept and process the broadcasted frame.

Single Hub



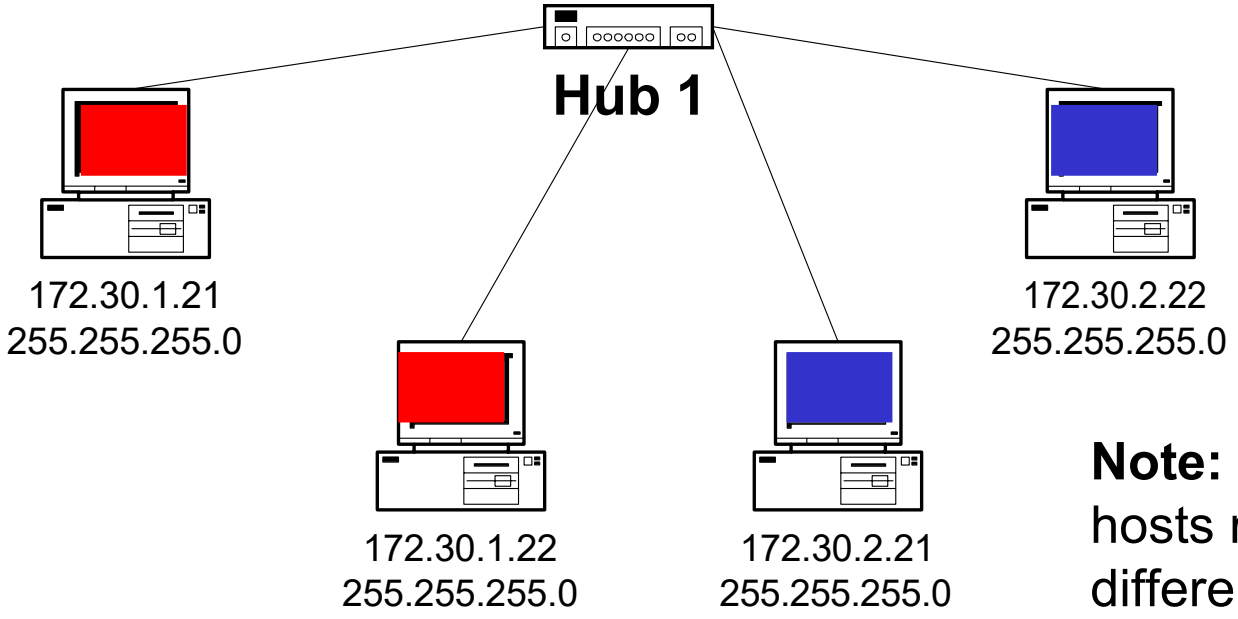
Single Hub

- One Network (IP Network Address - usually)
- One Collision Domain
- One Broadcast Domain

Single Hub

This is fine for small workgroups, but does not scale well for larger workgroups or heavy traffic.

Single Hub



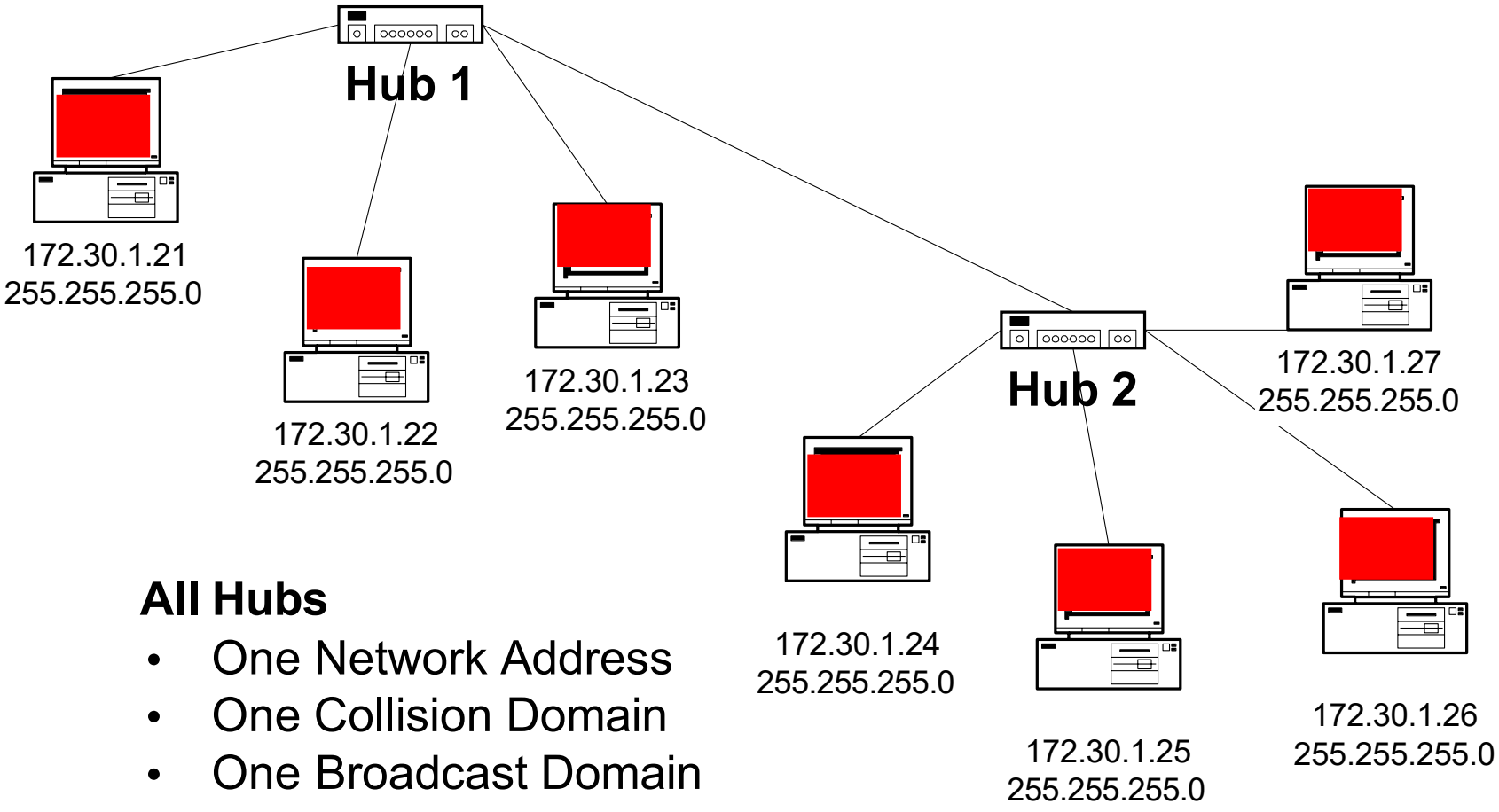
Note: Different color hosts refer to different subnets.

Single Hub - Two subnets

- Two subnets
- One Collision Domain
- One Broadcast Domain

- What if the computers were on two different subnets?
- Could they communicate within their own subnet? **Yes**
- Between subnets? **No, need a router. The sending host will check the destination IP address with its own IP address and subnet mask. The AND operation will determine that it is on a different subnet and cannot be reached without sending the packet to a default gateway (router). This is even though they are on the same physical network.**

Multiple Hubs



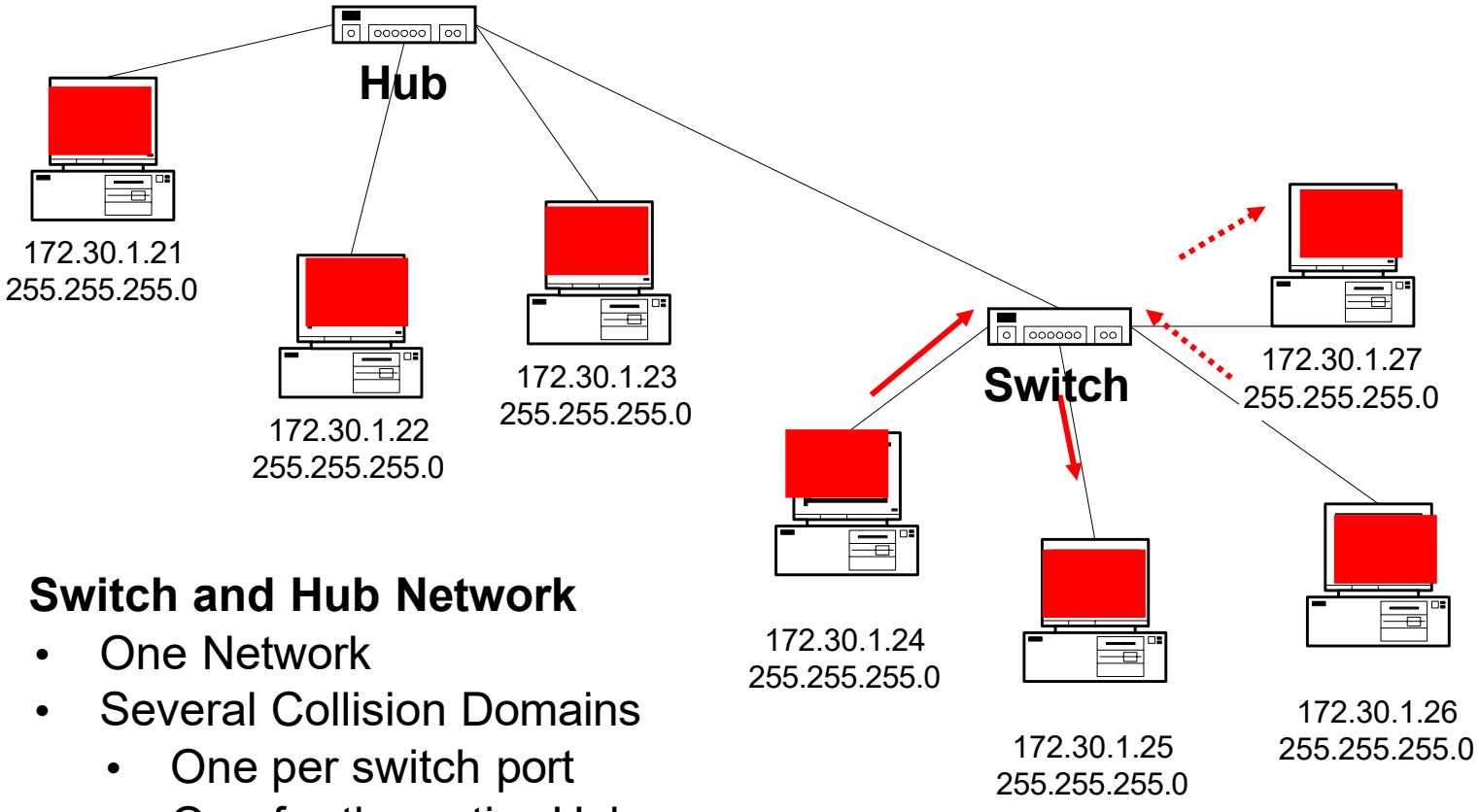
All Hubs

- One Network Address
- One Collision Domain
- One Broadcast Domain

Using Switches

- Layer 2 devices
- Layer 2 filtering based on Destination MAC addresses and Source Address Table
- One collision domain per port
- One broadcast domain across all switches

Switches create multiple parallel paths



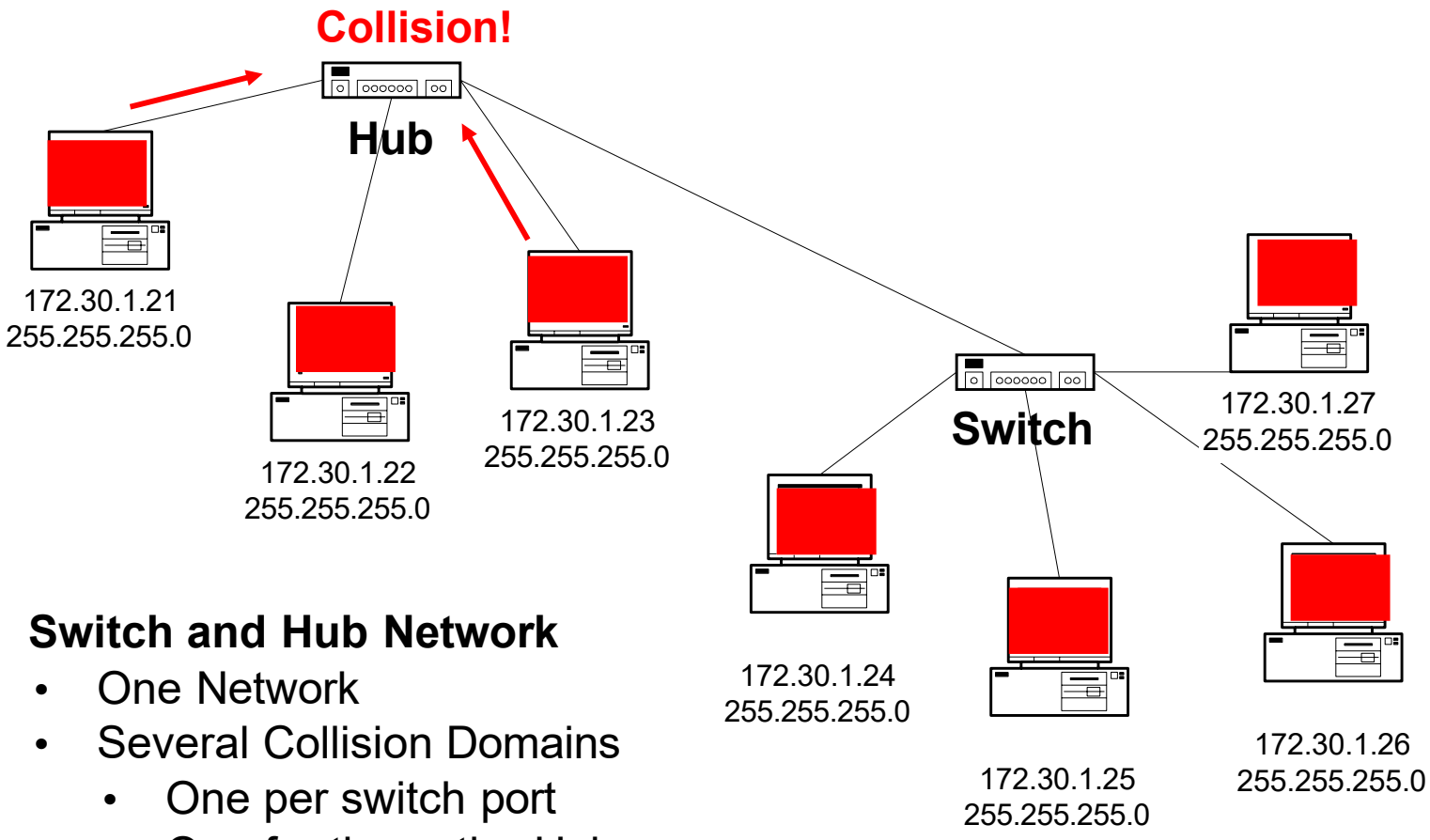
Switch and Hub Network

- One Network
- Several Collision Domains
  - One per switch port
  - One for the entire Hub
- One Broadcast Domain

Two parallel paths: (complete SAT tables)

- Data traffic from 172.30.1.24 to 172.30.1.25
- Data traffic from 172.30.1.26 to 172.30.1.2

- Hubs do not create multiple parallel paths



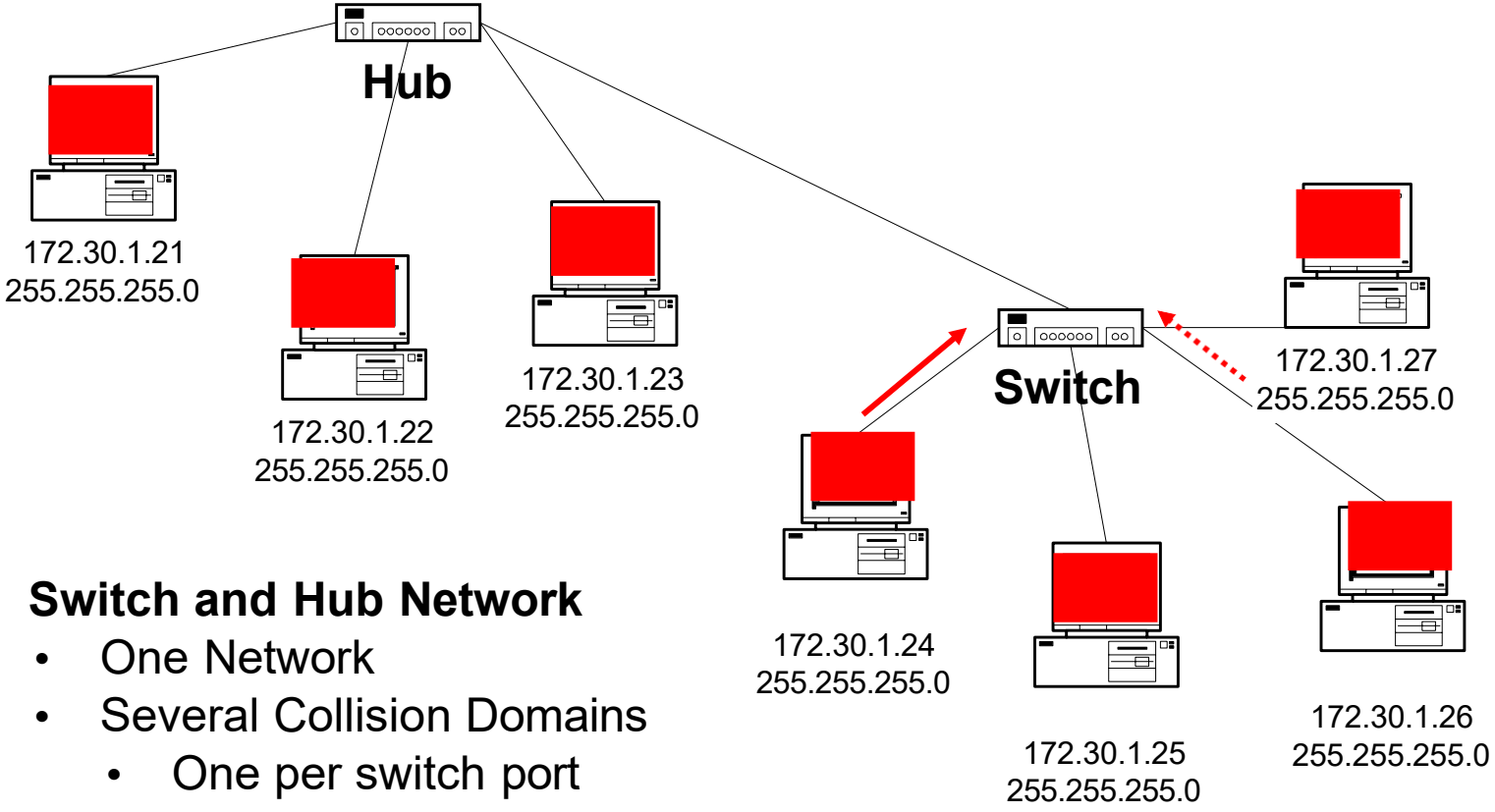
Switch and Hub Network

- One Network
- Several Collision Domains
  - One per switch port
  - One for the entire Hub
- One Broadcast Domain

As opposed to the Hub:

- Data traffic from 172.30.1.21 to 172.30.1.22
- Data traffic from 172.30.1.23 to 172.30.1.24

Switches create multiple parallel paths



Switch and Hub Network

- One Network
- Several Collision Domains
  - One per switch port
  - One for the entire Hub
- One Broadcast Domain

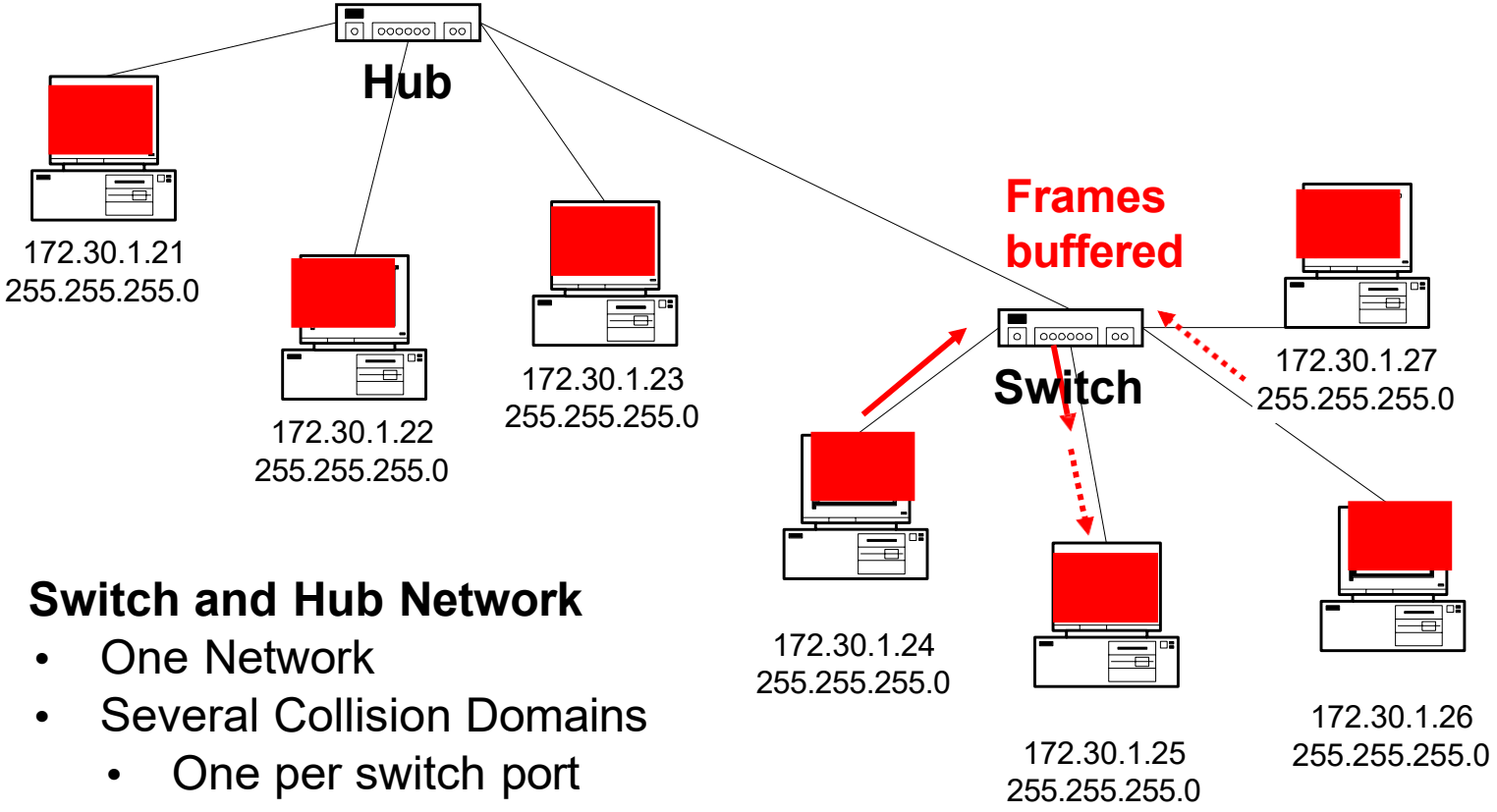
Switches create multiple parallel paths

Collisions and Switches:

What happens when two devices on a switch, send data to another device on the switch?

172.30.1.24 to 172.30.1.25 **and** 172.30.1.26 to 172.30.1.25

Switches create multiple parallel paths



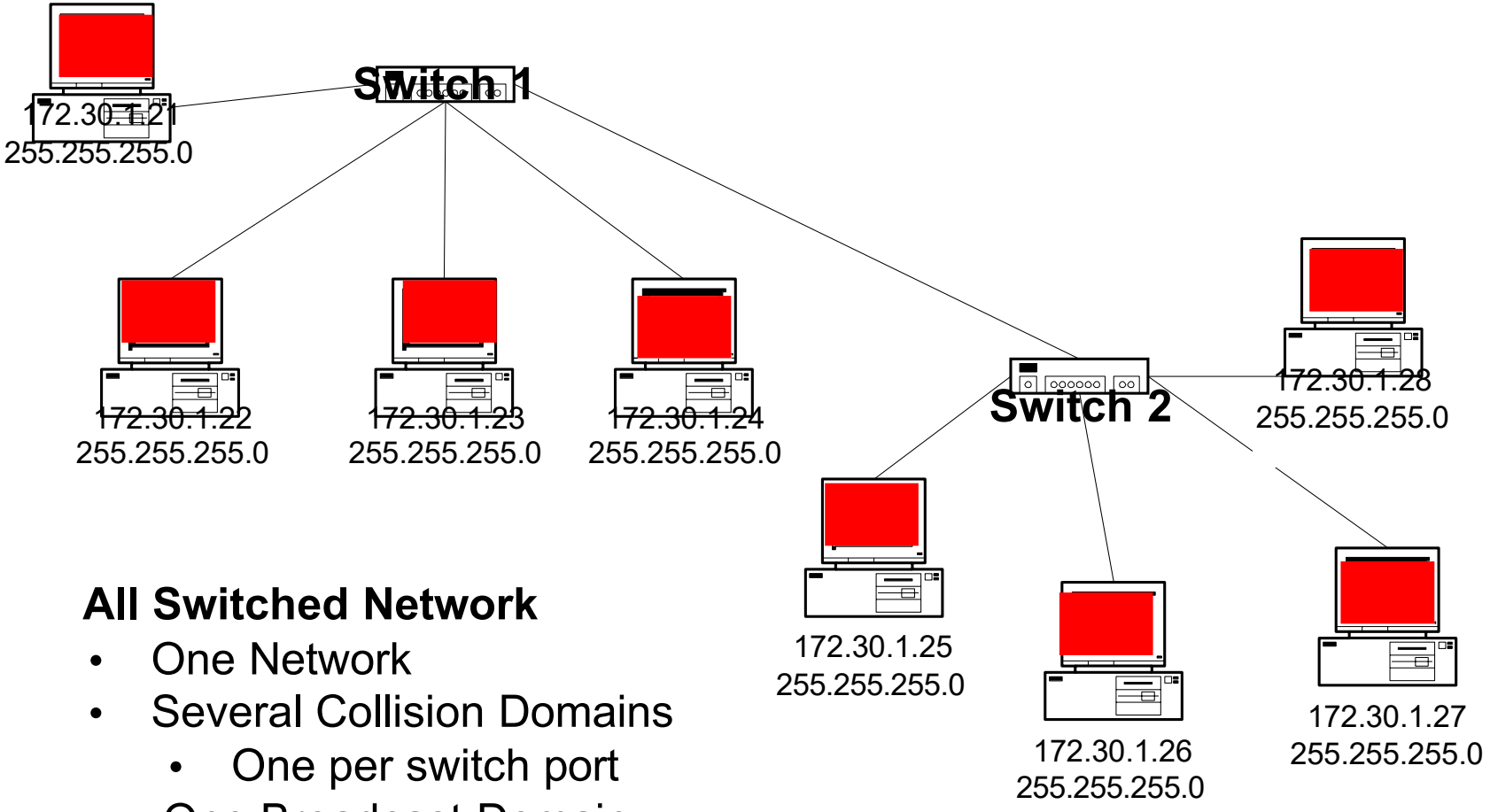
Switch and Hub Network

- One Network
- Several Collision Domains
  - One per switch port
  - One for the entire Hub
- One Broadcast Domain

The switch keeps the frames in buffer memory, and queues the traffic for the host 172.30.1.25.

This means that the sending hosts do not know about the collisions and do not have to re-send the frames.

Other Switching Features



All Switched Network

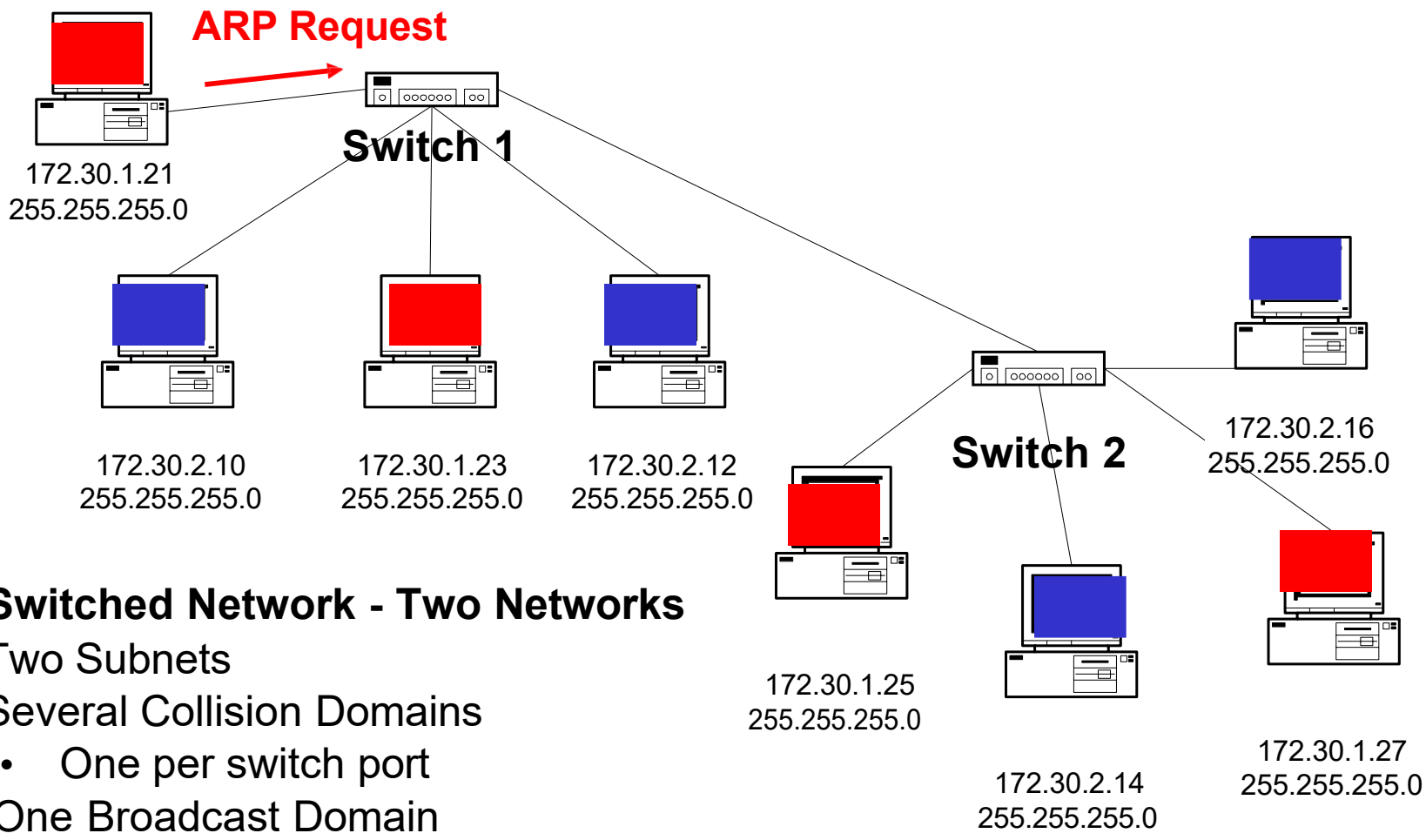
- One Network
- Several Collision Domains
  - One per switch port
- One Broadcast Domain

- Ports between switches and server ports are good candidates for higher bandwidth ports (100 Mbps) and full-duplex ports.
- Most switch ports today are full-duplex.

Introducing Multiple Subnets/Networks without Routers

- Switches are Layer 2 devices
- Router are Layer 3 devices
- Data between subnets/networks must pass through a router.

Switched Network with Multiple Subnets



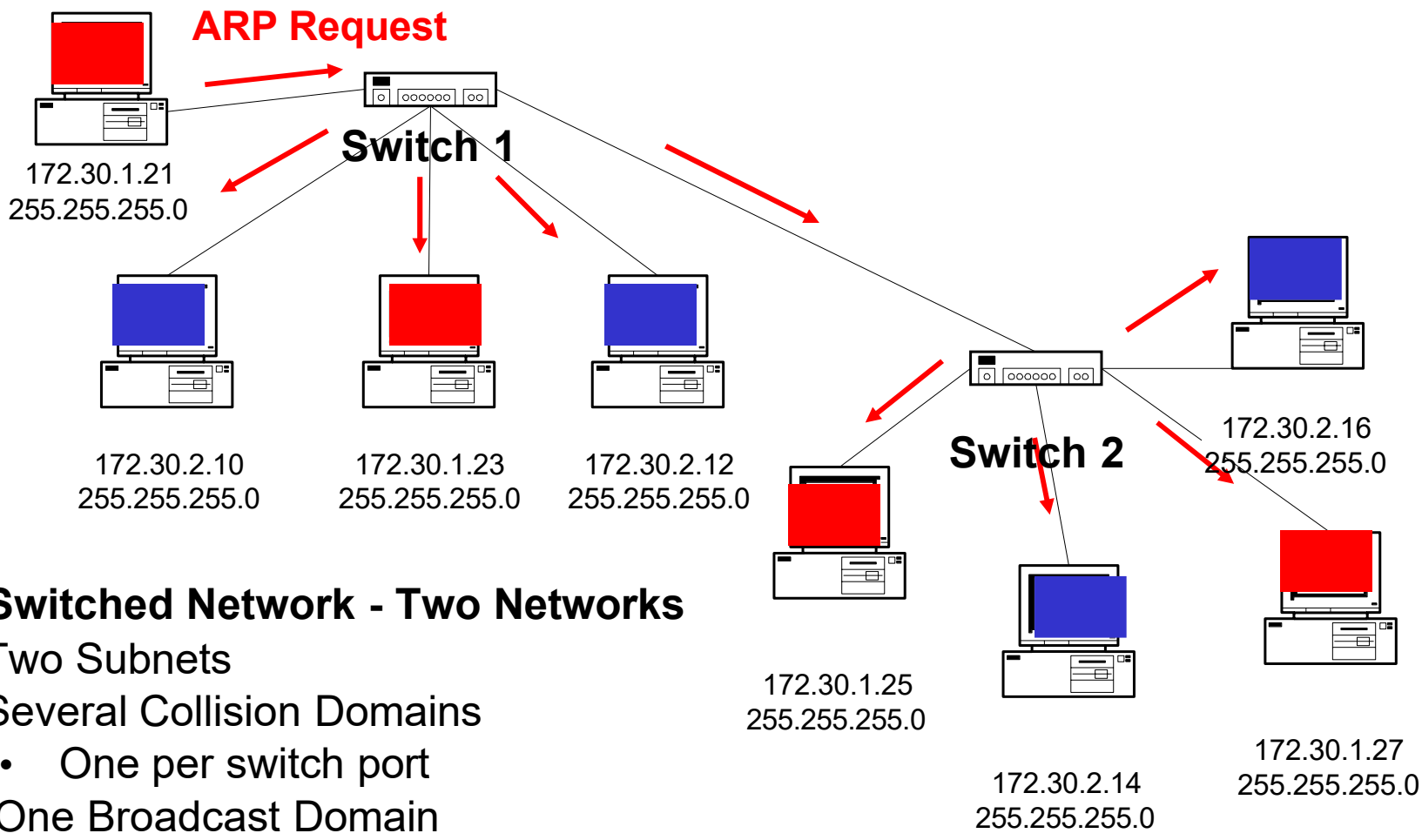
All Switched Network - Two Networks

- Two Subnets
- Several Collision Domains
  - One per switch port
- One Broadcast Domain

Switched Network with Multiple Subnets

- What are the issues?
- Can data travel within the subnet? Yes
- Can data travel between subnets? No, need a router!
- What is the impact of a layer 2 broadcast, like an ARP Request?

Switched Network with Multiple Subnets



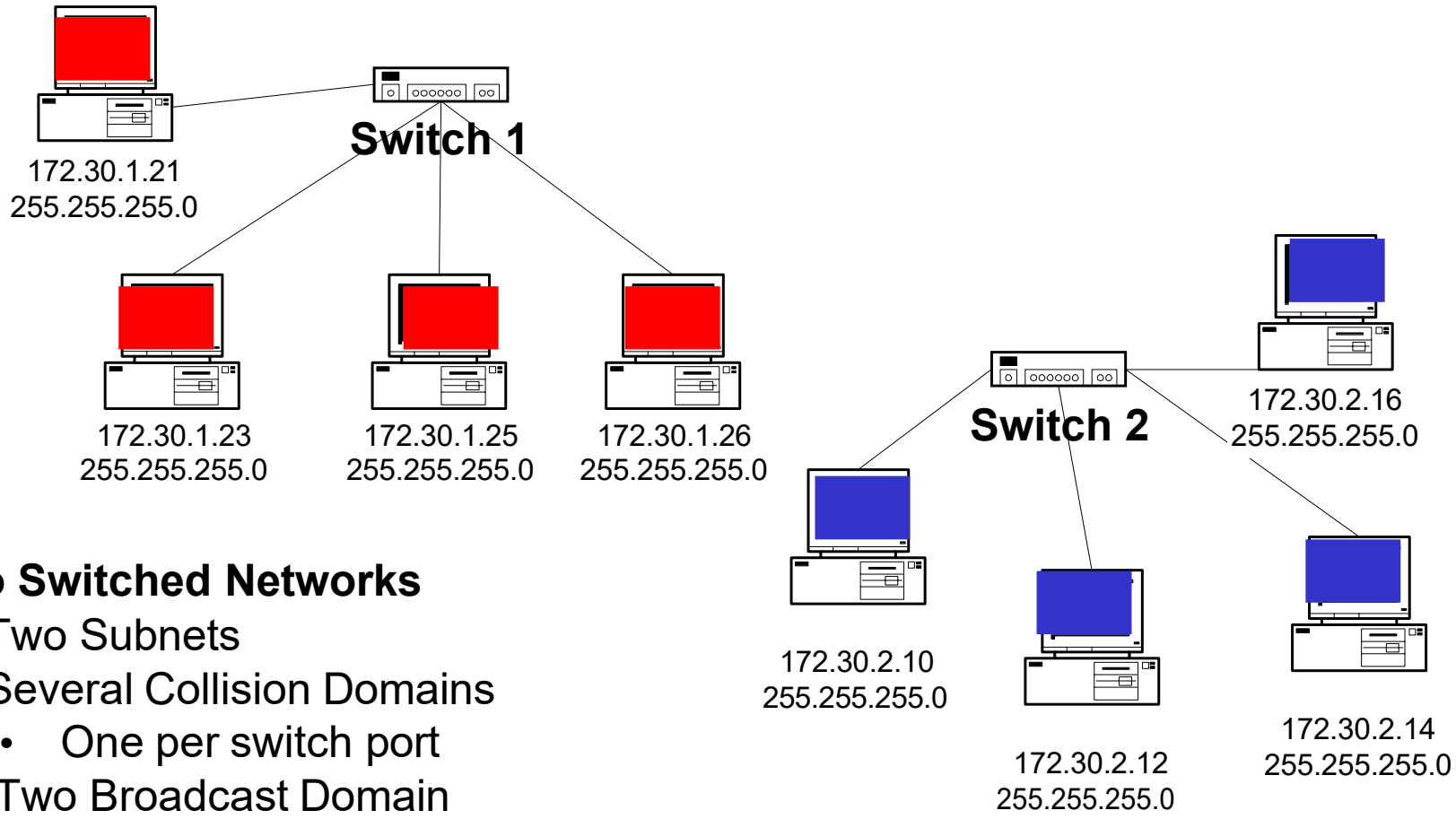
All Switched Network - Two Networks

- Two Subnets
- Several Collision Domains
  - One per switch port
- One Broadcast Domain

Switched Network with Multiple Subnets

- All devices see the ARP Request, even those on the other subnets that do not need to see it.
- One broadcast domain means the switches flood all broadcast out all ports, except the incoming port.
- Switches have no idea of the layer 3 information contained in the ARP Request.This consumes bandwidth on the network and processing cycles on the hosts.

One Solution: Physically separate the subnets

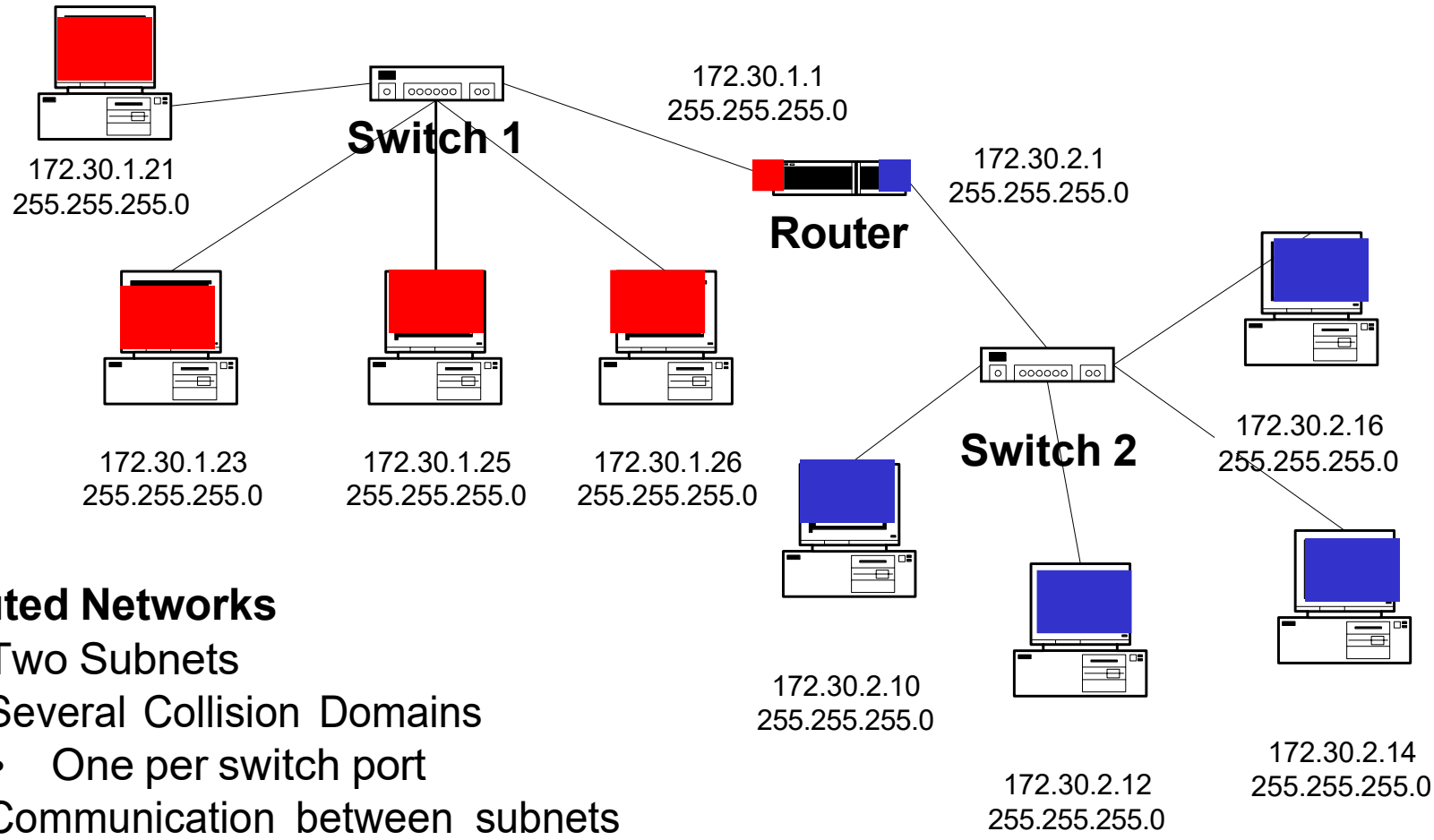


Two Switched Networks

- Two Subnets
- Several Collision Domains
  - One per switch port
- Two Broadcast Domain

- But still no data can travel between the subnets.
- How can we get the data to travel between the two subnets?

Another Solution: Use a Router



Routed Networks

- Two Subnets
- Several Collision Domains
  - One per switch port
- Communication between subnets

- Two separate broadcast domains, because the router will not forward the layer 2 broadcasts such as ARP Requests.