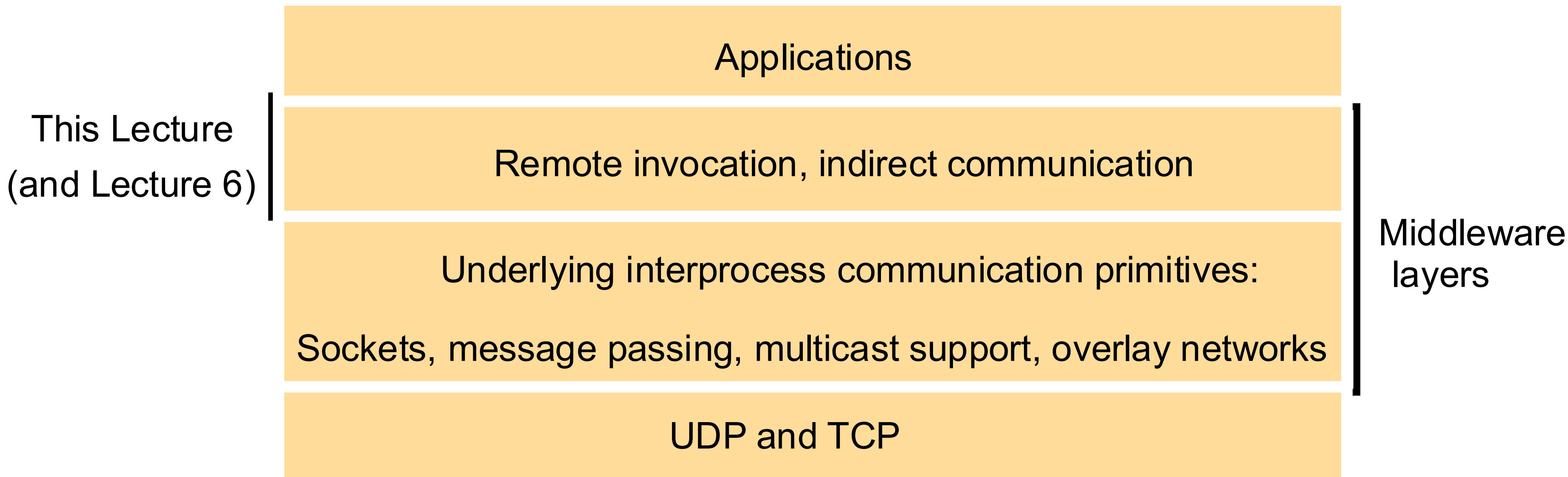
Remote invocation

CECS 327 Introduction to Networks and Distributed computing

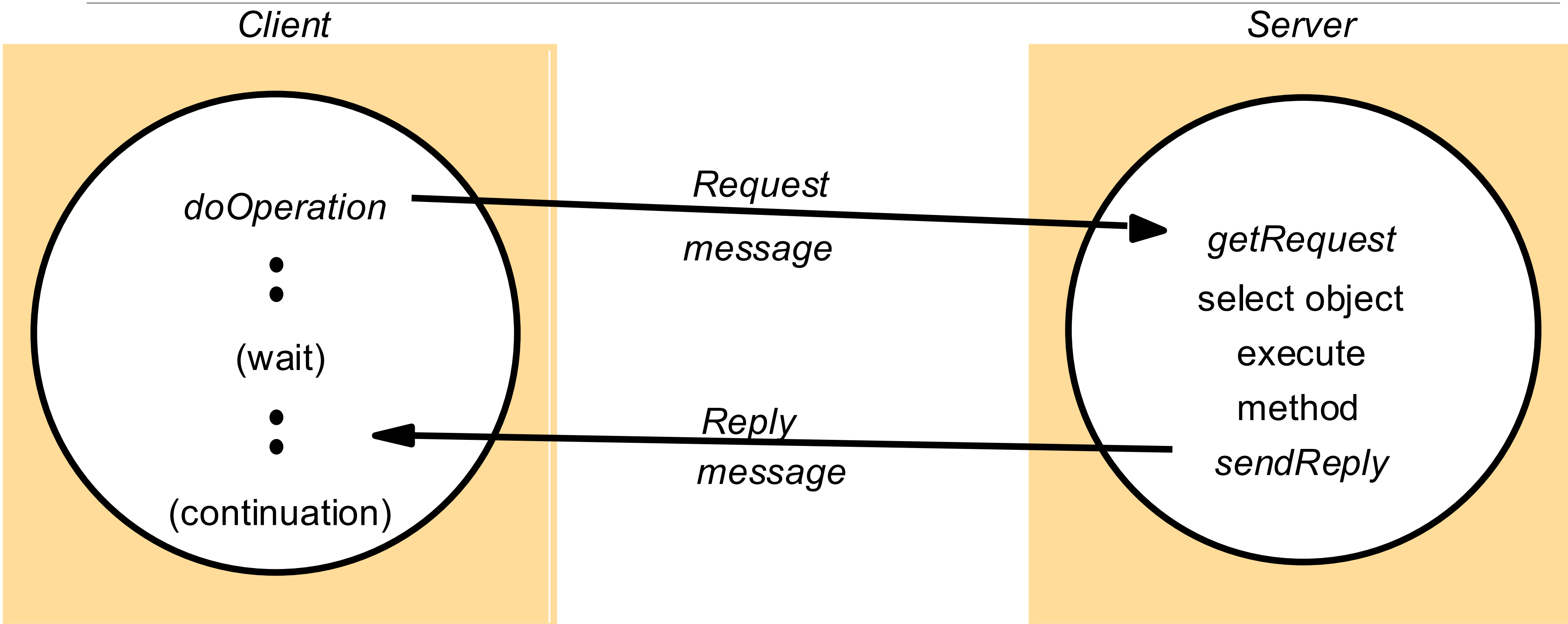
Oscar Morales-Ponce

CECS, CSULB

Middleware layers



Request-reply communication



Operations of the request-reply protocol

1. Client Stub

def do_operation(remote_ref, operation_id: int, arguments: bytes) -> bytes

Sends a request:

$$Req = (\mathrm{remote}_{\mathrm{ref}}, \mathrm{operation}_{\mathrm{i}}, \mathrm{args})$$

Waits for and returns the **reply**.

2. Server Stub

def get_request() -> (bytes, tuple)

Waits on server port.

Returns **(request, client info)**.

3. Reply

def send_reply(conn, reply: bytes)

Sends **reply message** back to the requesting client.

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*array of bytes*

Request-reply protocols

Message Identifiers: Each request has a unique **Request ID**, **Remote Ref**:

$$Req = (ReqID, RemoteRef, Data)$$

Failure Model (UDP)

- **Timeouts:** NoReply$(t > T_{max}) \quad \Rightarrow \quad retransmit(Req)$
- **Duplicate requests:** must discard if ReqID already processed.
- **Lost replies:** Safe if operation is **idempotent**:

$$f(x); f(x) = f(x) \ (\textit{same effect if repeated})$$

History Table: Server keeps history of processed ReqID to detect duplicates.

TCP Option: Using **TCP streams** for request–reply avoids most of these issues, since TCP already ensures:

reliable delivery, ordering, no duplicates.

RPC exchange protocols

Name	*Messages sent by*		
	Client	*Server*	*Client*
R	*Request*		
RR	*Request*	*Reply*	
RRA	*Request*	*Reply*	*Acknowledge reply*

RPC Concepts

Programming with Interfaces

- Client and server communicate through **well-defined interfaces**.
- The client only needs the **interface**, not the server's internal code.

Call Semantics

- RPC (Remote Procedure Call) behaves like a **local function call**, but runs on a remote server.
- Semantics: at-~~most~~ least-once, exactly-once, or maybe (depending on failure handling).

Transparency

- Goal: make remote calls look the same as local calls.
- The programmer should not need to know if a call is local or remote.

CORBA IDL example

```
// In file Person.idl
struct Person {
        string name;
        string place;
        long year;
} ;
interface PersonList {
        readonly attribute string listname;
        void addPerson(in Person p) ;
        void getPerson(in string name, out Person p);
        long number();
};
```

Techniques for Reliable RPC

Retry Request Message

If no reply within timeout T_{max}: Resend$(ReqID)$

Duplicate Filtering

Server checks **history table**:

If $ReqID \in H \quad \Rightarrow \quad$ *discard duplicate*

Retransmission of Results

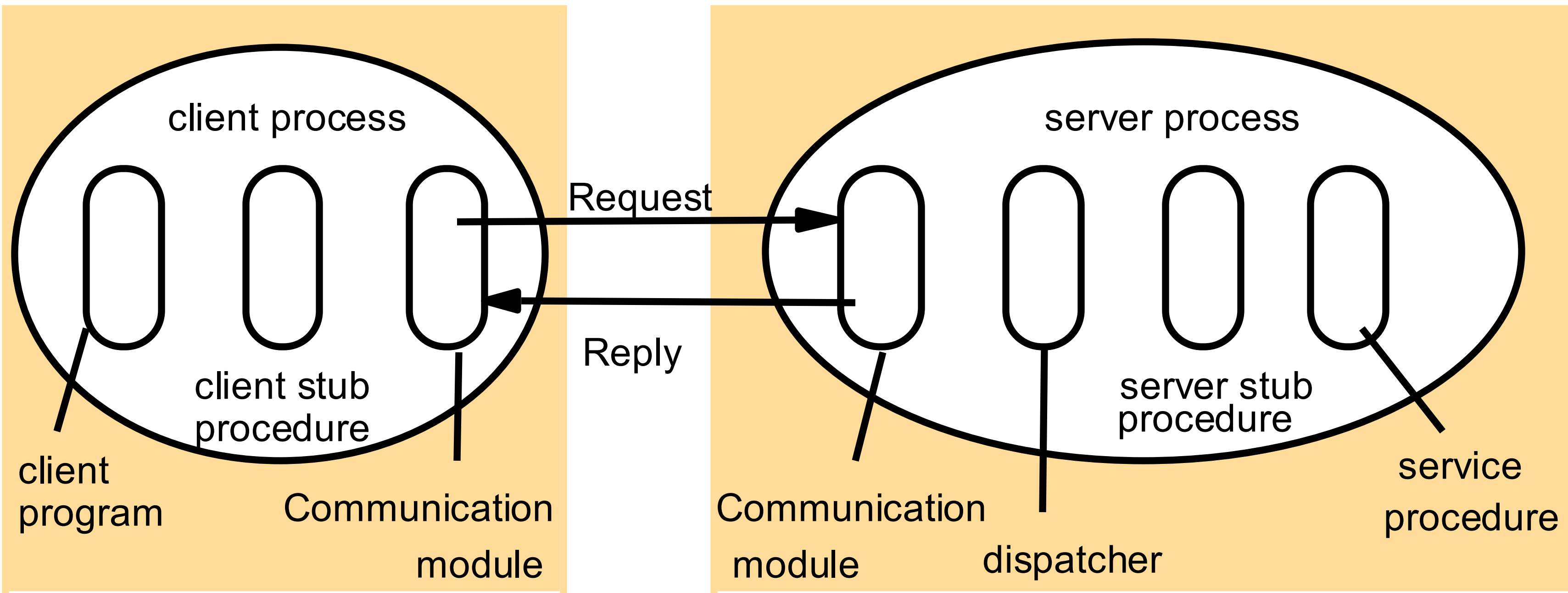
If duplicate request received, server **resends stored reply**:

$$\text{Reply}(ReqID) = H[ReqID]$$

Call semantics

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Implementation of RPC: Role of client and server stub procedures in RPC



Remote method invocation

Interfaces: Programs interact via **defined interfaces**, not raw message passing.

Built on Request–Reply: RPC/RMI = abstraction layer above request–reply protocols.

Call Semantics: Provides reliability options:

$$\text{Semantics} \in \{At-least-once, At-most-once\}$$

Transparency: Remote calls look like **local method calls** (hides networking).

Object-Oriented Support

- Uses **object identity**:

$$\text{RemoteRef} = (ObjectID, ServerAddr)$$

- Full power of **OOP** (methods, encapsulation, polymorphism).

The object model and distributed object model

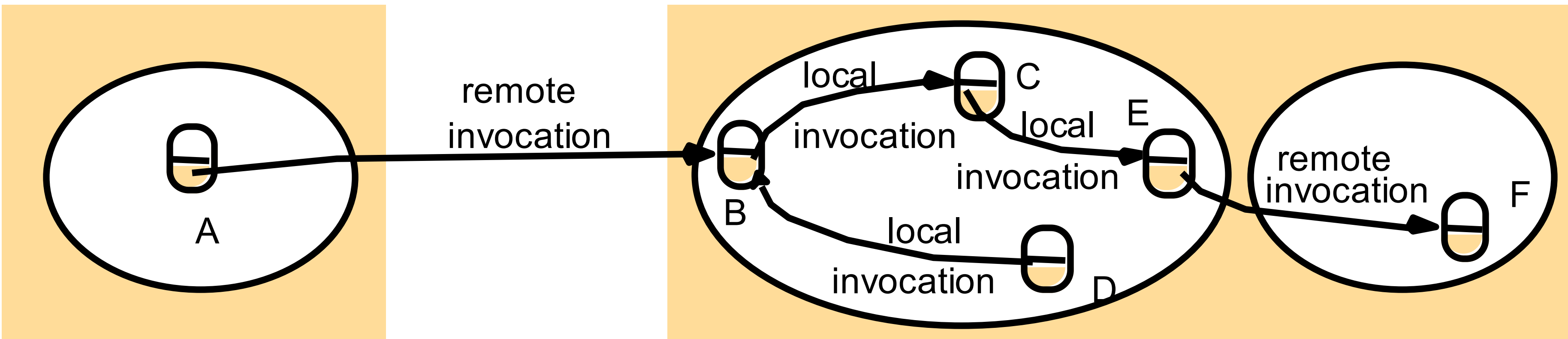
Object Model

- **Object References** → identifiers used to access objects.
- **Interfaces** → define available methods.
- **Actions** → method calls performed on objects.
- **Exceptions** → error conditions during method calls.
- **Garbage Collection** → automatic cleanup of unused objects.

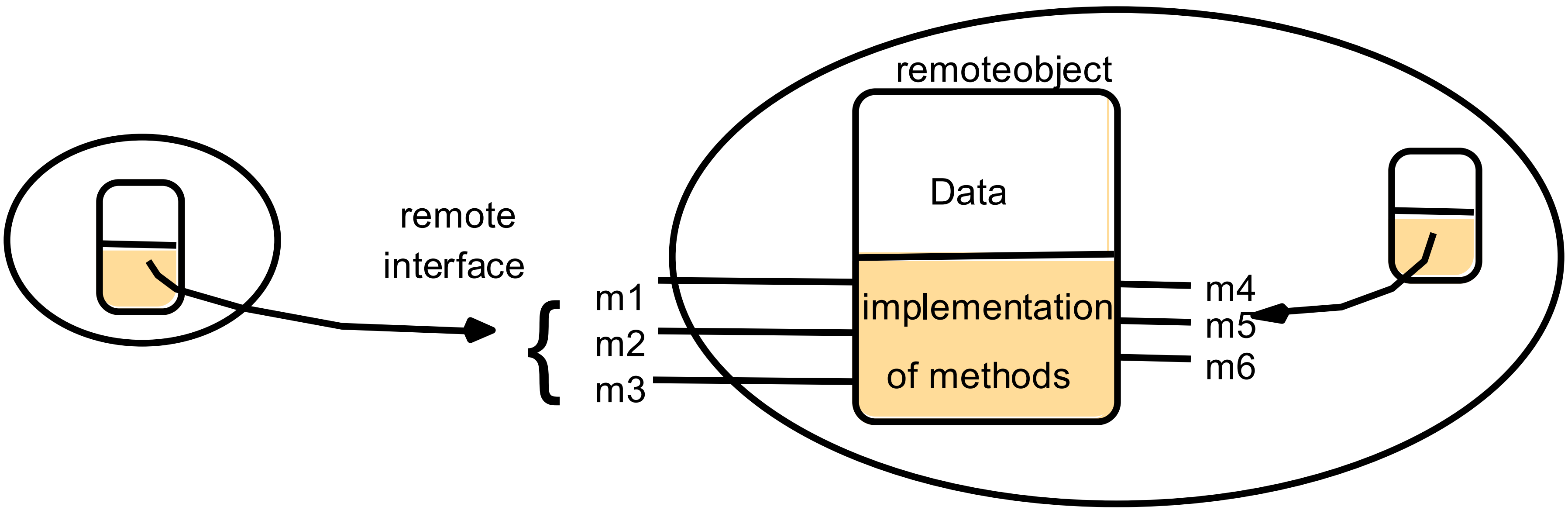
Distributed Object Model

- **Remote Object References** → allow access to objects on other machines.
- **Remote Interfaces** → define methods that can be invoked **remotely**.

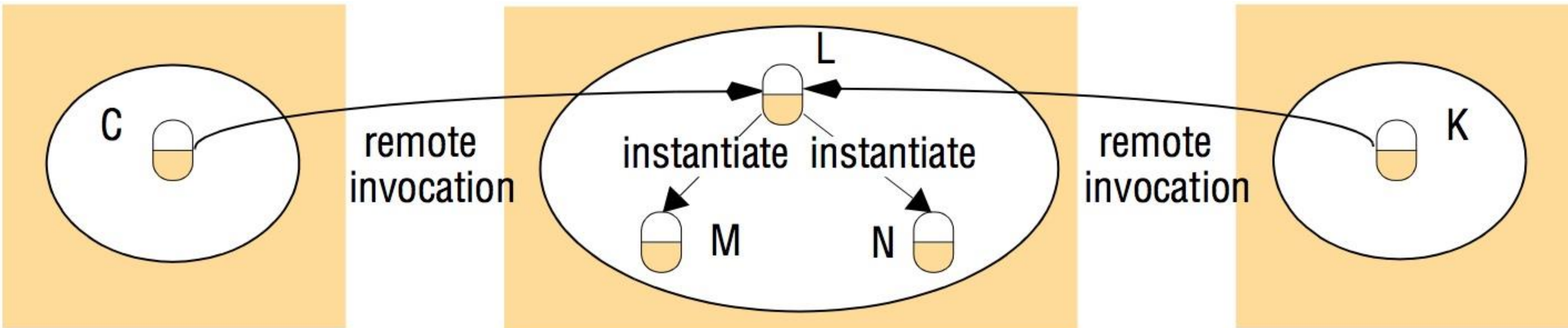
Remote and local method invocations



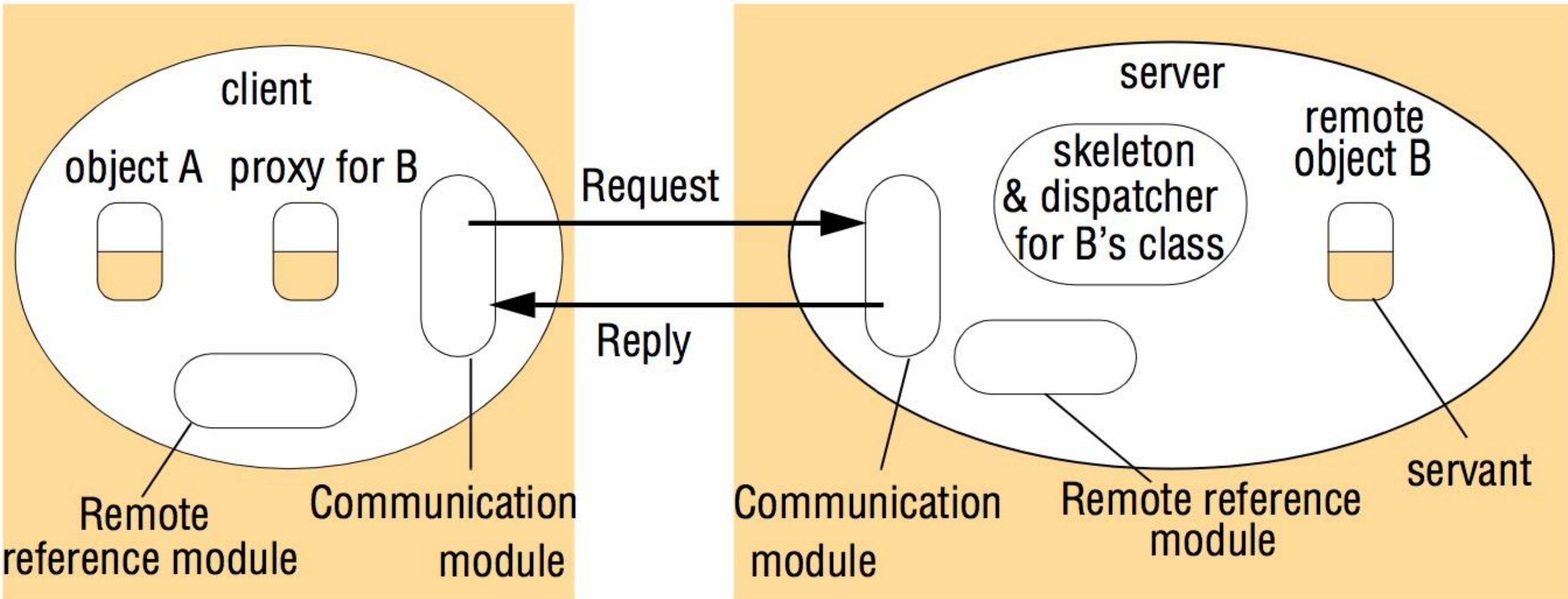
A remote object and its remote interface



Instantiation of remote objects



Implementation of RMI: The role of proxy and skeleton in remote method invocation



Define a Remote Object in Python

```python
# remote_object.py
from Pyro5.api import expose

@expose
class RemoteCounter:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1
        return self.count

    def get(self):
        return self.count
```

Start the Pyro Daemon (Server)

```
# server.py
import Pyro5.api
from remote_object import RemoteCounter

daemon = Pyro5.api.Daemon()
uri = daemon.register(RemoteCounter)
print("Ready. Object URI =", uri)
daemon.requestLoop()
```

Client to Access Remote Object

```
# client.py
import Pyro5.api

uri = input("Enter the server URI: ")
remote = Pyro5.api.Proxy(uri)

print(remote.increment())
print(remote.get())
```

Register an Object with a Name

```
# server.py
from Pyro5.api import expose, Daemon, locate_ns

@expose
class MyService:
    def greet(self, name):
        return f"Hello, {name}!"

# Register with Name Server
daemon = Daemon()
ns = locate_ns()
uri = daemon.register(MyService)
ns.register("example.greeting", uri)

print("Service registered. Waiting for requests...")
daemon.requestLoop()
```

Client Looks It Up by Name

```
# client.py
from Pyro5.api import locate_ns, Proxy

ns = locate_ns()
uri = ns.lookup("example.greeting")
remote = Proxy(uri)

print(remote.greet("Alice"))
```

Interprocess Communication

CECS 327 Introduction to Networks and Distributed computing

Oscar Morales-Ponce

CECS, CSULB

Sockets

- Socket = endpoint for communication

$$\text{Socket} = (IP, Port)$$

- Used to send/receive data across a network

Socket Types

Stream Socket (TCP)

- Connection-oriented
- Reliable, ordered delivery

Datagram Socket (UDP)

- Connectionless
- Unreliable, unordered

IPC Properties

Validity: every sent msg is eventually delivered

Integrity: no corruption, no duplication

Ordering: FIFO by sender

UDP Basics

Datagram = fixed-length packet

$$P = (Src_{IP}, Src_{port}, Dst_{IP}, Dst_{Port}, Data)$$

- No ACK, no retries, $P_{loss} > 0$.

UDP Reliability Extension

- Add ACKs + retransmissions.
- Cost: more overhead.

UDP Example (Python)

- One short server snippet (recvfrom, sendto).

UDP Reliability Extension

Add ACK + retransmission:

$$m \;\Rightarrow\; ack(m)$$

Retransmit if no ACK received

Tradeoff: higher reliability $\leftrightarrow$ more overhead

UDP Example (Python)

```
import socket
sock = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
sock.bind(('localhost', 12345))
data, addr = sock.recvfrom(1024)
print("Received:", data.decode())
sock.sendto(b"Reply", addr)
sock.close()
```

Blocking vs Non-Blocking

Aspect	Blocking	Non-Blocking (Polling)
Thread Usage	1/thread	Single thread
CPU Efficiency	Idle when waiting	CPU $\propto$ polling rate
Response Time	Higher (waits)	Lower (immediate return)

Non-Blocking Example

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('localhost', 12345))
sock.setblocking(False)
while True:
    try:
        data, addr = sock.recvfrom(1024)
        print("Received:", data)
    except BlockingIOError:
        continue
```

TCP Features

Data segmentation:

$$M = \{m_1, m_2, \dots, \mathrm{m_k}\} \quad | \, m_i \, | \leq MTU$$

Reliability: ACK + retransmit

Flow control:

$$\text{sendable} \leq \min(\text{cwnd}, \text{rwnd})$$

Ordering & Duplication

Sequence numbers ensure order:

$$i < j \;\Rightarrow\; recv(m_i)\ before\ recv(m_j)$$

Duplicates discarded

TCP Connection Setup

- **3-Way Handshake**
  - Client → SYN(x)
  - Server → SYN(y), ACK(x+1)
  - Client → ACK(y+1)

Connection Lifecycle

- Listening socket: accepts multiple clients
- New socket created per client
- Closing: all sockets terminate when process exits

TCP client

```
import socket
import sys

def main():
    try:
        message = sys.argv[1]
        host = sys.argv[2]
        server_port = 7896

        # Create a socket and connect to the server
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.connect((host, server_port))
            # Send the message
            s.sendall(message.encode('utf-8'))
            # Receive the response
            data = s.recv(1024).decode('utf-8')
            print("Received:", data)
    except IndexError:
        print("Usage: python3 TCPClient.py <message> <hostname>")
    except socket.gaierror as e:
        print("Socket error:", e)
    except ConnectionRefusedError as e:
        print("Connection error:", e)
    except Exception as e:
        print("Error:", e)
if __name__ == "__main__":
    main()
```

TCP server

```
import socket
def main():
    host = '0.0.0.0'  # Listen on all interfaces
    port = 7896       # Same port used by the client
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:
        server_socket.bind((host, port))
        server_socket.listen(1)
        print(f"Server listening on port {port}...")
        while True:
            conn, addr = server_socket.accept()
            with conn:
                print("Connected by", addr)
                data = conn.recv(1024).decode('utf-8')
                if not data:
                    break
                print("Received:", data)
                response = f"Echo: {data}"
                conn.sendall(response.encode('utf-8'))
if __name__ == "__main__":
    main()
```

Marshalling

- Marshal: structure → bytes

$$(d_1, d_2, \ldots, d_k) \ \rightarrow \ (b_1, b_2, \ldots b_k)$$

- Unmarshal: bytes → structure
- Needed for portability

Example (Python Pickle)

```
import pickle
data = {"name": "Alice", "year": 1984}
serialized = pickle.dumps(data)
obj = pickle.loads(serialized)
print(obj)
```

XML

```
<person id="123">
  <name>Smith</name>
  <place>London</place>
  <year>1984</year>
</person>
```

Pros: portable, schema-based
Cons: verbose

JSON

{"id": 123, "name": "Smith", "place": "London", "year": 1984}

- Lightweight
- Widely used in APIs

Multicast Basics

One sender → many receivers

Group = Class D IP range:

$$224.0.0.0 \ \leq \ IP \ \leq \ 239.255.255.255$$

Dynamic join/leave

Multicast Example (Python)

```
import socket, struct
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM,
socket.IPPROTO_UDP)
sock.bind(('', 6789))
group = socket.inet_aton('228.5.6.7')
mreq = struct.pack('4sL', group, socket.INADDR_ANY)
sock.setsockopt(socket.IPPROTO_IP,
socket.IP_ADD_MEMBERSHIP, mreq)
data, addr = sock.recvfrom(1024)
print("Received:", data.decode())
```

Networking and Internetworking

CECS 327 Introduction to Networks and Distributed computing

Oscar Morales-Ponce

CECS, CSULB

Network performance

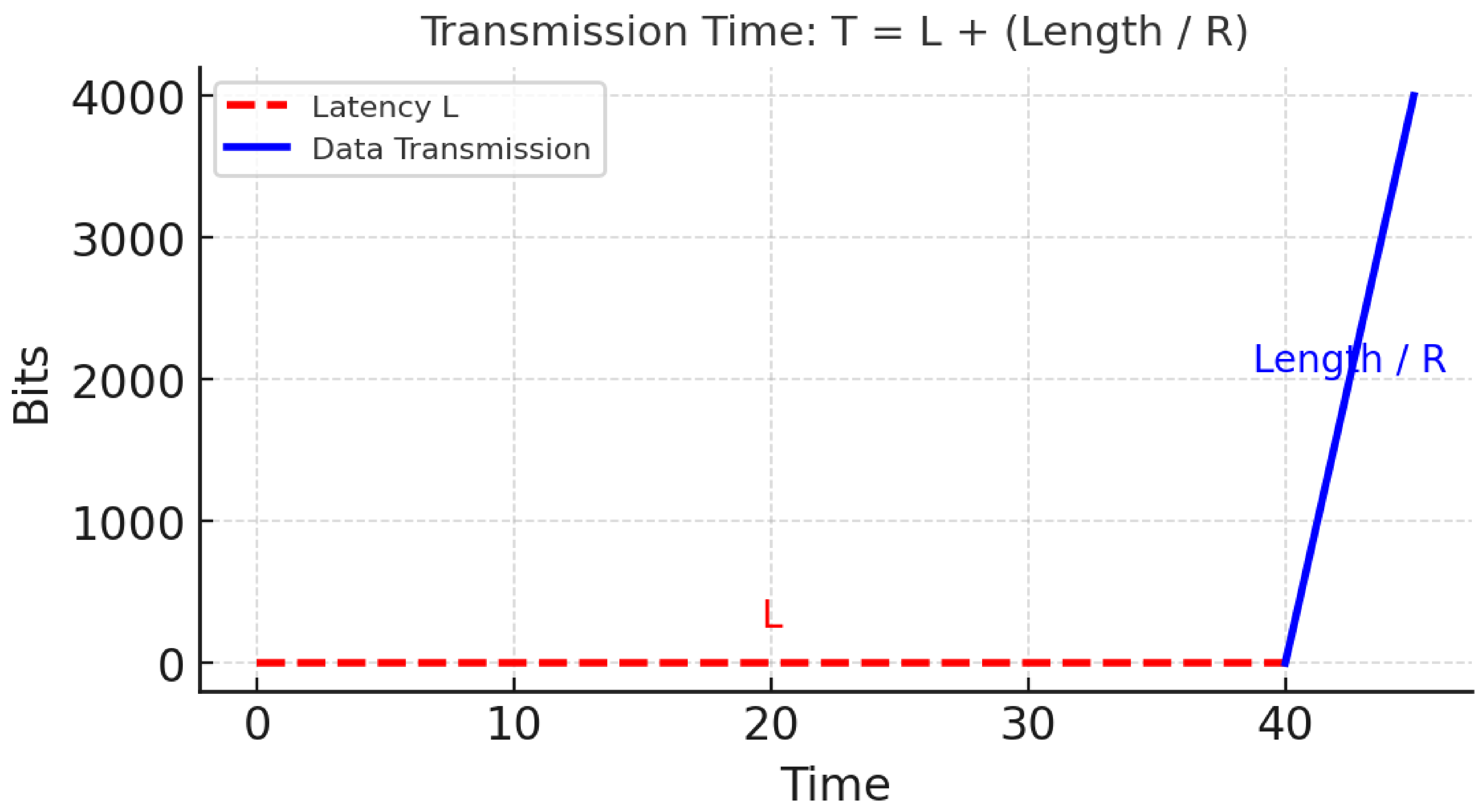
- **Latency** (L)
  Delay before first bit arrives (ms).
- **Data Rate** (R)
  Transmission speed (bits/s).
- **Message Transmission Time** (T)

$$T = L + \frac{\text{Message Length (bits)}}{R}$$

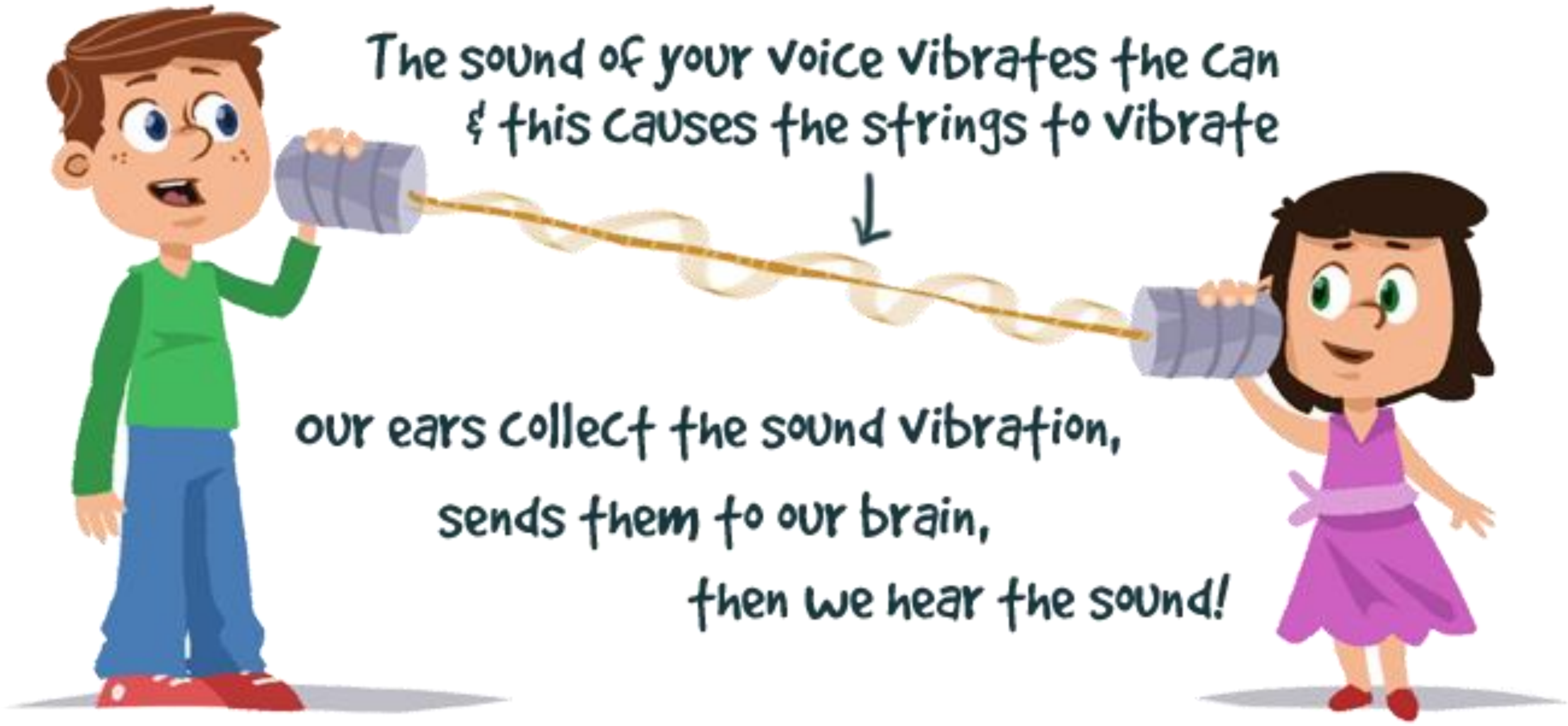
- **Throughput** (BW)

$$BW = \frac{\text{Total data transferred}}{\text{Time}}$$

Network performance

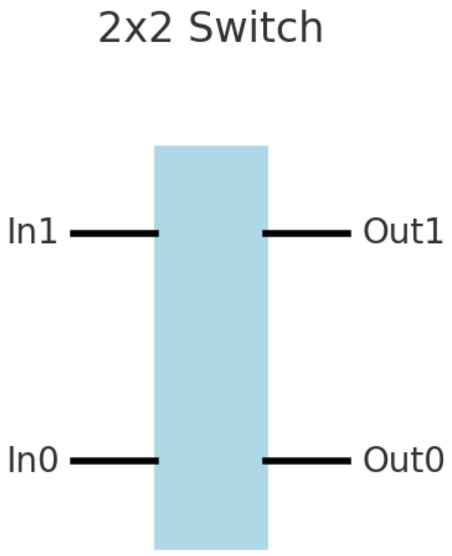


Switch Network



Switch Network

- **Definition**:
  Data moves via **switches** that select paths between nodes.
- **Circuit Switching**



Fixed Path:

$$\text{Path} = (n_1 \rightarrow n_2 \rightarrow \cdots \rightarrow n_k)$$

Transmission Time:

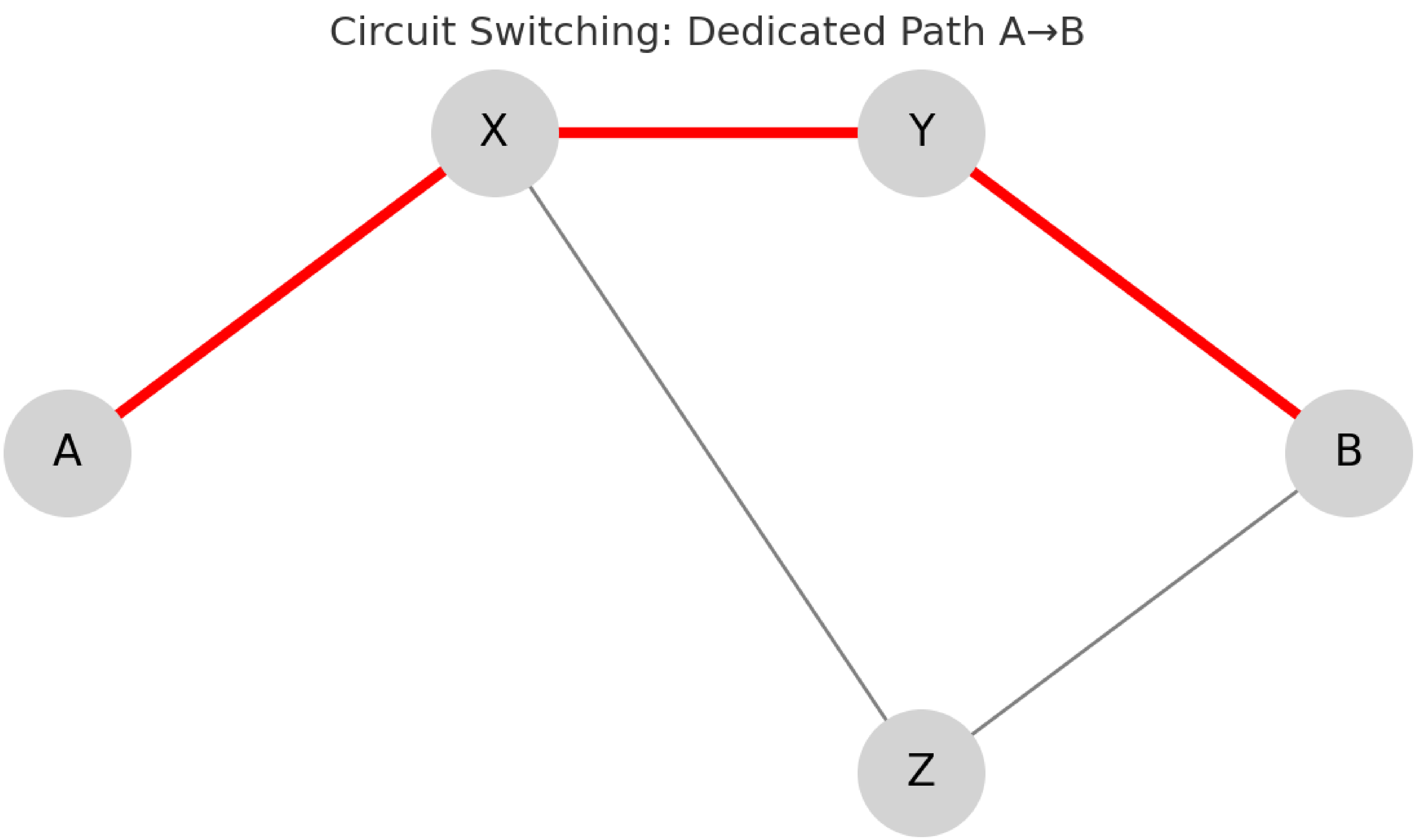
$$T = L_{setup} + \frac{Length}{R}$$

where L_{setup} **=**connection setup delay.

Pros: Constant rate (R), predictable latency.

Cons: Idle path = wasted resources.

Switch Network



Omega Network Basics

Multistage Interconnection Network (MIN)

$N = 2^k$, Stages = k

Each stage: N/2 switching elements (2x2)

Path selected using destination address bits

Stages = $\log_2(N)$

Switches per stage = N/2

8×8 Omega Network

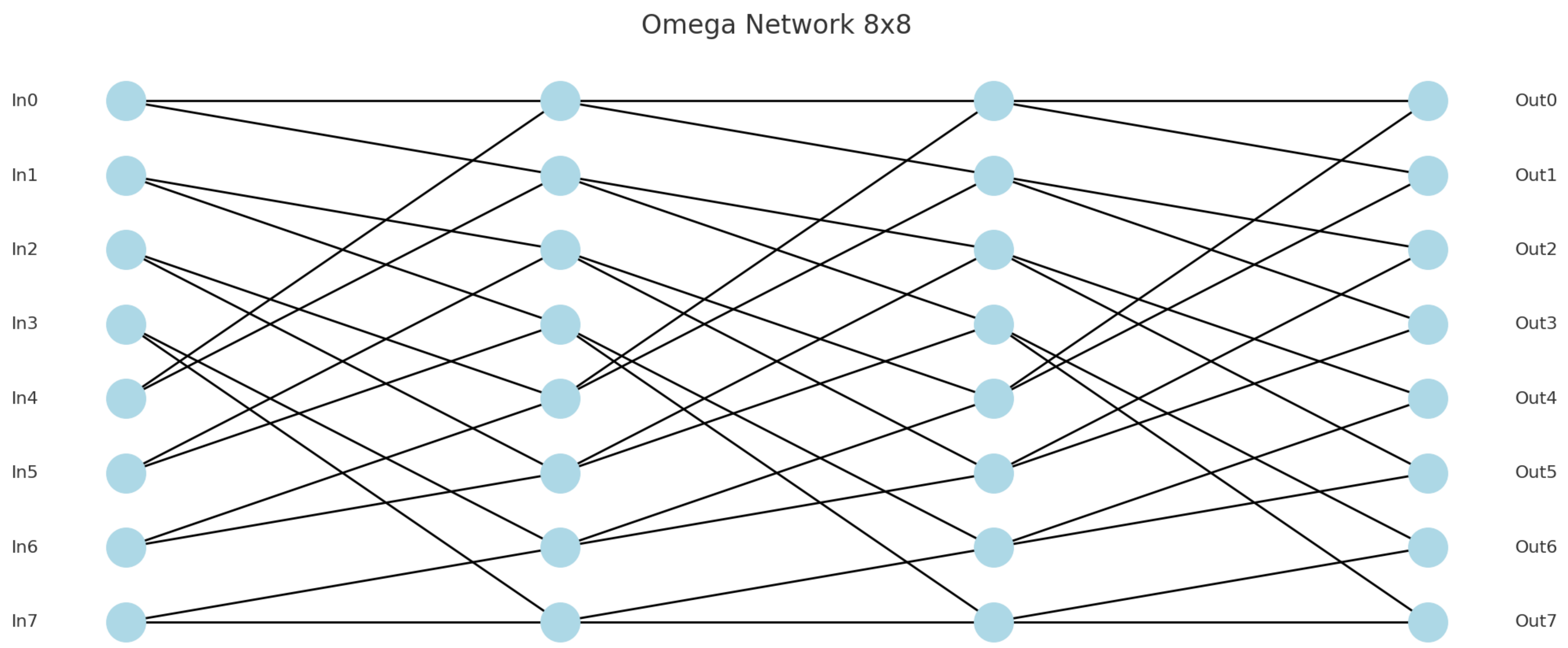
For an $N = 2^k$ Omega Network:

Stage $j (0 \leq j < k)$:
Each switching element uses the j^{th} **bit of the destination address** (from MSB to LSB) to decide:

$$f(i, d, j) = \begin{cases} upper\ output, & if\ bit_j(d) = 0 \\ lower\ output, & if\ bit_j(d) = 1 \end{cases}$$

Path Selection:
The **binary address of the destination** uniquely determines the route.

8×8 Omega Network

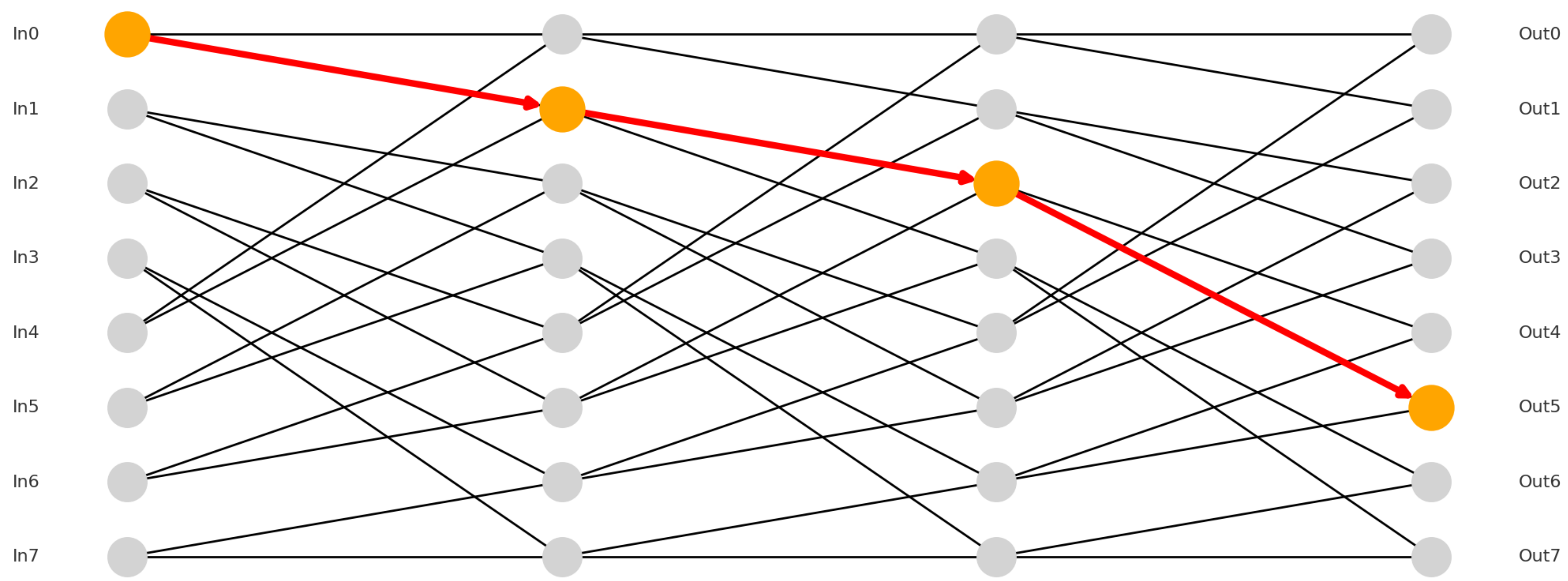


8×8 Omega Network

Example (N=8, In0 → Out5=101_2):

- Stage 1: bit0=1 → lower
- Stage 2: bit1=0 → upper
- Stage 3: bit2=1 → lower

Omega 8x8 Network: Path In0 → Out5



Packet Network

Model:

Message M split into n packets:

$$M = \{\mathrm{P}_1, \mathrm{P}_2, \dots, \mathrm{P}_\mathrm{n}\}, \qquad | P_i | \leq L_{max}$$

Routing:

Each P_i may follow a different path:

$$\text{Path}(P_i) = \text{f(src, dest, i)}$$

Advantages:

High efficiency (links shared by many flows).

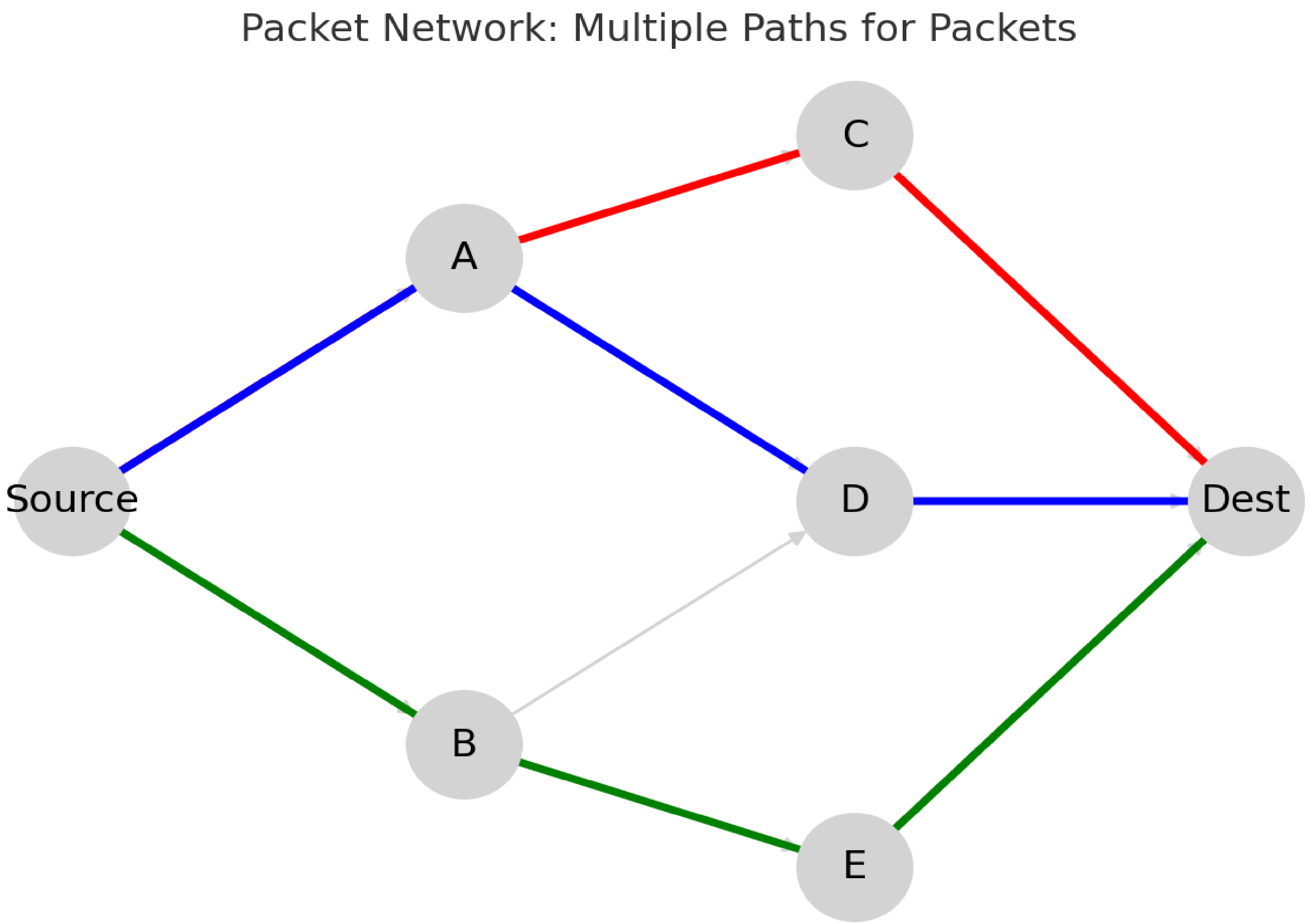
Scalable to large networks.

Disadvantages:

Variable delay (Δt_i)

Possible packet loss ($P_{loss} > 0$).

Packet Network



Message Switching

Model:
Entire message M(size $|M|$) transmitted as a **single unit**. Each intermediate node **stores** M before forwarding.

Delay Formula:
If there are h hops,

$$T = h \cdot \left(L + \frac{|M|}{R} \right)$$

where:

- L= latency per hop
- R= transmission rate

Pros:

- No dedicated path required.

Cons:

- Each switch must have buffer $\geq |M|$.
- High delay due to **store-and-forward**.

Terminology

Message

$$M = (d_1, d_2, \dots, d_n), n \in \mathbb{N}$$

Arbitrary length sequence of data items.

Packet

$$P = (\text{Header}(\text{src}, \text{dest}), \text{Payload}), \mid P \mid \leq L_{max}$$

Fixed-length unit with addressing info.

Streaming: Continuous flow:

$$\{p_1, p_2, \dots, \mathrm{p_t}\}, t \to \infty$$

Used for real-time audio/video.

Broadcast

One-to-all delivery:

$$\text{Send}(src, M) \quad \Rightarrow \quad \forall\, d \in Network,\ d \neq src$$

Internetworks

Router: Forwards packets using **routing function**:

$$f_{router}(p) = out\_port(dest(p),\ routing\_table)$$

Bridge: Connects heterogeneous networks:

$$f_{bridge}:\ N_a \leftrightarrow N_b$$

Hub: Broadcast device (no filtering):

$$f_{hub}(p) = \{p \ \mid\ \forall\ d \in segment\}$$

Switch: Like router, but local (LAN scope). Uses **MAC table** for selective forwarding:

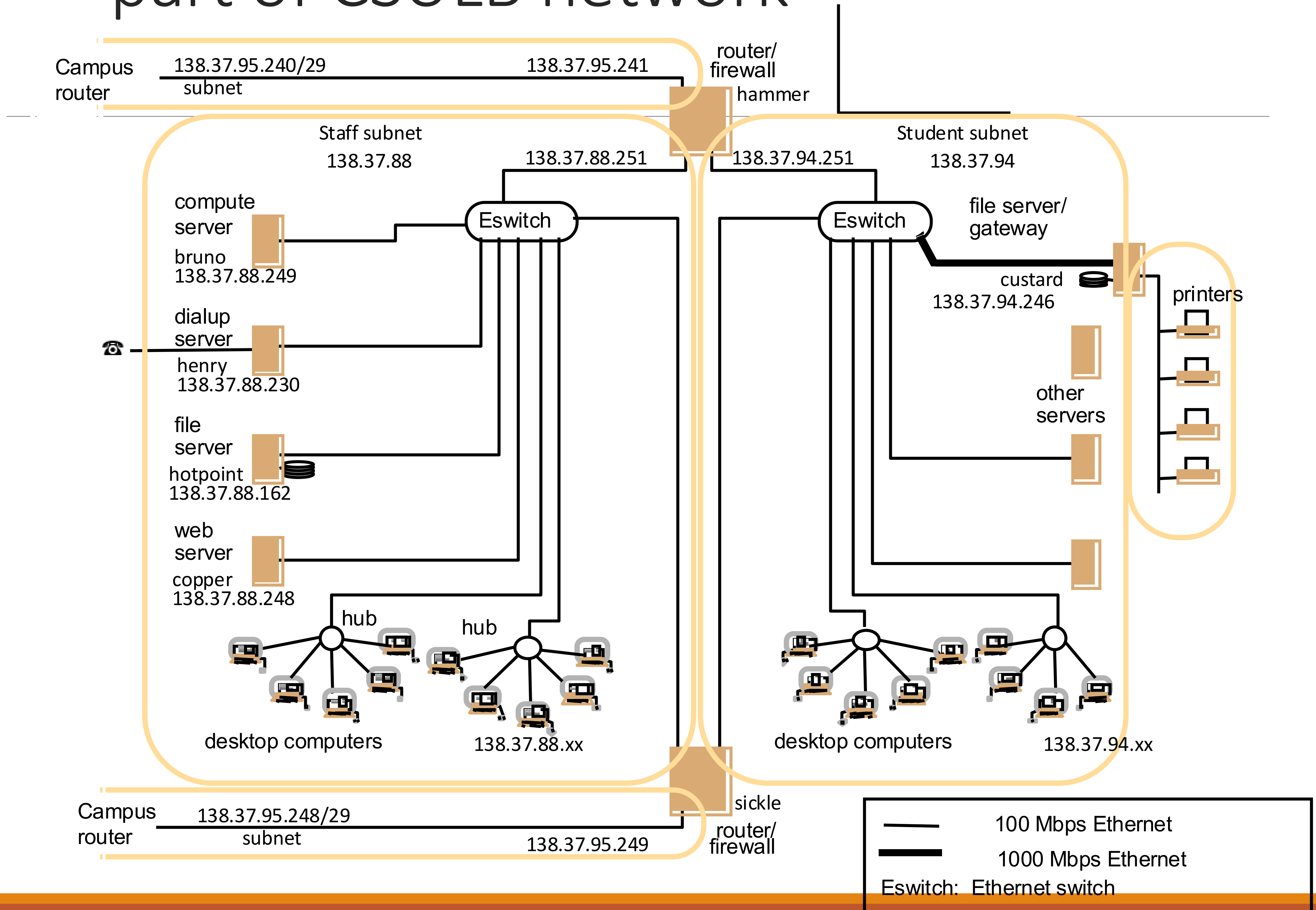
$$f_{switch}(\mathrm{src}, \mathrm{dest}) \rightarrow specific\ port$$

Tunneling: Encapsulation of a packet p inside another:

$$p' = Encap(p,\ tunnel\_header)$$

Enables forwarding across an "alien" network.

Internetworking: Simplified view of part of CSULB network



Types of networks

Personal Area Network (PAN)

Scope:

$$d \leq 10\ m$$

Nodes: wearable + mobile devices.

Example: Smartwatch $\leftrightarrow$ *Phone (Bluetooth)*

Local Area Network (LAN)

Scope:

$$d \leq 1\ km$$

Performance:

$$R \gg 100\ Mbps, L \ll 10\ ms$$

Ownership: private (home, office, campus).

Example: Laptop $\leftrightarrow$ $Wi{-}Fi\ AP \leftrightarrow Printer$

Types of networks

Metropolitan Area Network (MAN)

Scope:

$$1\ km < d \leq 50\ km$$

Use: Interconnects multiple LANs in a city.

Performance:

$$10\ Mbps \leq R \leq 600\ Mbps, 10\ ms \leq L \leq 50\ ms$$

Control: ISPs or municipalities.

Example: City-wide Wi-Fi.

Types of networks

Wide Area Network (WAN)
Scope:

$$d > 100\ km\ (national\ /\ global)$$

Use: Internet, corporate backbones.
Performance:

$$R \approx 0.5\ Mbps\ to\ 600\ Mbps, L \approx 100 - 500\ ms$$

Characteristics: Uses routers + public transmission systems.
Example: The Internet.

Network performance

	Example	*Range*	*Bandwidth (Mbps)*	*Latency (ms)*
Wired:				
LAN	Ethernet	1–2 kms	10–10,000	1–10
WAN	IP routing	worldwide	0.010–600	100–500
MAN	ATM km	2–50 kms	1–600	10
Internetwork	Internet	worldwide	0.5–600	100–500
Wireless:				
WPAN	Bluetooth (IEEE 802.15.1)	10–30m	0.5–2	5–20
WLAN	WiFi (IEEE 802.11)	0.15–1.5 km	11–108	5–20
WMAN	WiMAX (IEEE 802.16)	5–50 km	1.5–20	5–20
WWAN	3G phone	cell: 1—5	348–14.4	100–500

Datagram Delivery

Datagram Delivery

- **Model:** Each packet P_i is independent.

$$P_i = (\mathrm{Src}, \mathrm{Dest}, \mathrm{Payload_i})$$

- **Routing:**

$$\mathrm{Path}(P_i) = f(Src, Dest, i)$$

- **Characteristics:**
  - No setup phase.
  - Each packet may follow a different route.

Virtual Circuit Delivery

- **Setup:** Before transmission, establish path with **circuit ID** VC.

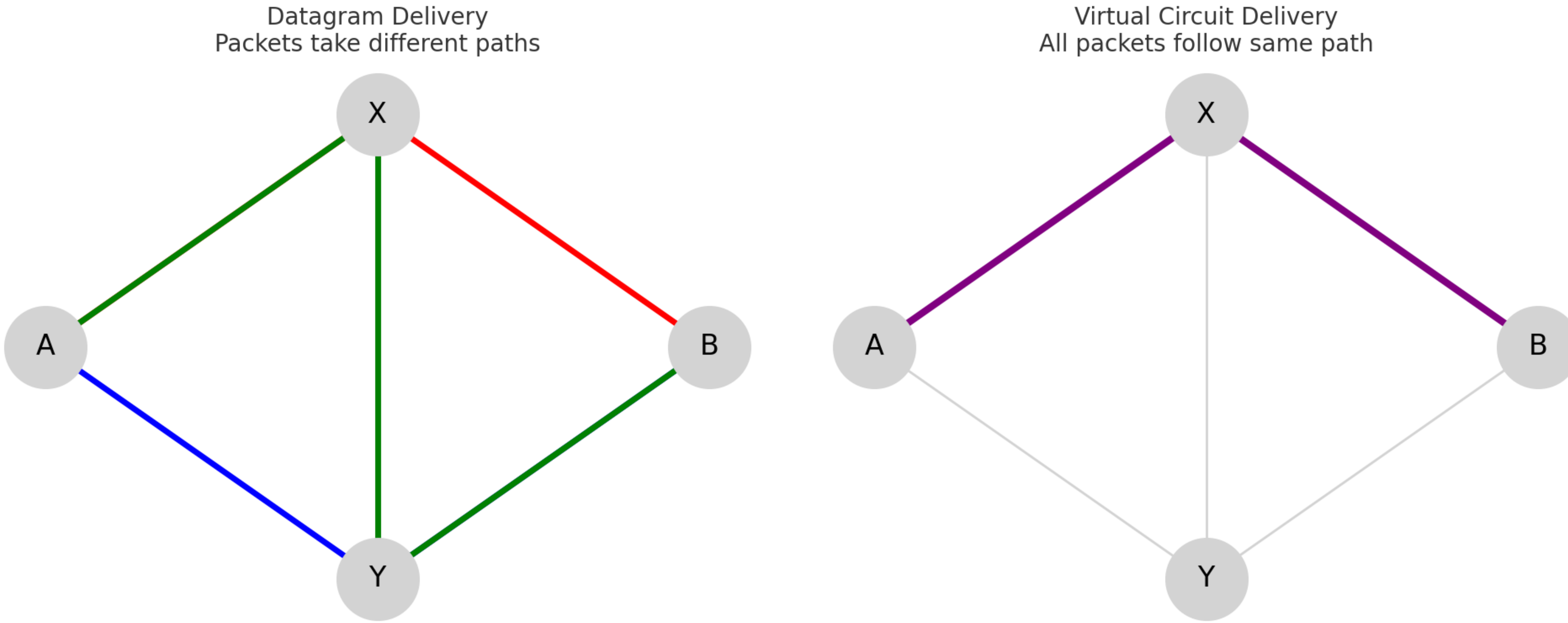
$$\text{Setup}(A, B) \quad \Rightarrow \quad VC_{AB}$$

- **Packets:**

$$P_i = (BC_{AB}, Payload_i)$$

- **Routing:** All packets follow same pre-defined path.
- **Characteristics:**
  - Connection-oriented.
  - Lower per-packet overhead.

Packet Delivery



Protocols

Definition: A protocol P is a tuple:

$$P = (\Sigma, F)$$

where:

Σ= ordered sequence of messages

F= format of each message

Example:

$$\Sigma = \{\mathrm{m_i}, \mathrm{m_2}, \ldots, \mathrm{m_n}\}, \mathrm{m_i} = \{\mathrm{Header}, \mathrm{Payload})$$

Properties:

Enables **independent development** of sender/receiver modules.

Implemented as **paired software modules**:

$$P = \{\mathrm{M_{Send}}, \mathrm{M_{recv}}\}$$

Protocol Layers (hierarchy)

Arranged as stack:

$$L_n \to L_{n-1} \to \cdots \to L_1$$

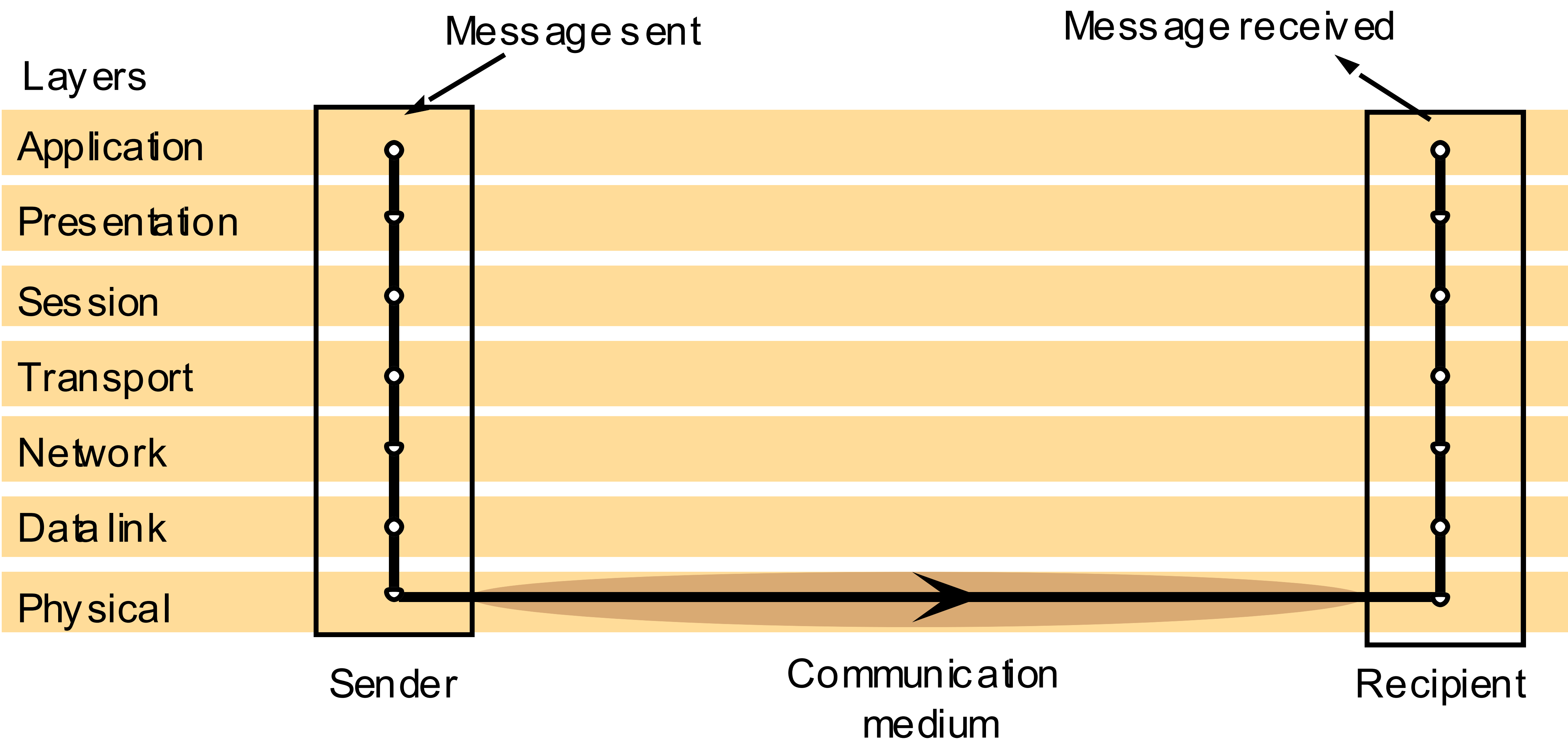
Each layer adds its **header** H_i:

$$\text{Message}_{out} = H_n \ \| \ H_{n-1} \ \| \ \ldots \ \| \ H_1 \ \| \ \textit{Payload}$$

Advantages

- Manage complexity
- Encapsulation and modularity
- Interoperability and standardization
- Separate concerns (e.g., transmission vs. application logic)

Protocol layers in the ISO Open Systems



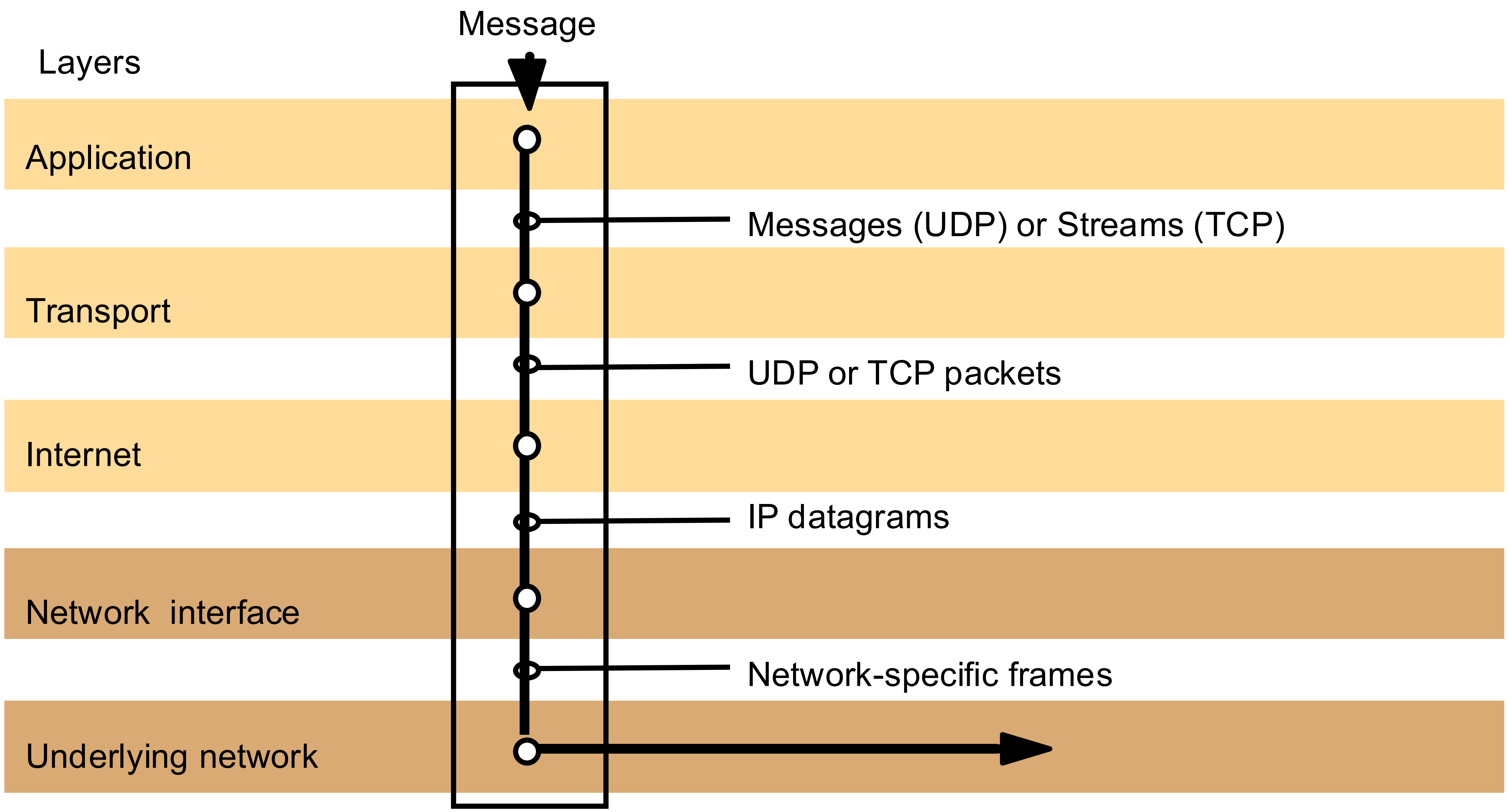
OSI protocol summary

Layer	Description	Examples
Application	Interfaces with end-user applications; defines communication services.	HTTP, FTP, SMTP
Presentation	Translates data formats, encryption, compression.	SSL/TLS, JPEG, MPEG
Session	Establishes, manages, and terminates sessions between applications.	Session, Checksum
Transport	Reliable or unreliable delivery of messages between processes.	TCP, UDP
Network	Logical addressing, routing between networks.	IP, ATM
Data Link	Node-to-node transfer; error detection, framing, and MAC addressing.	Ethernet, MAC
Physical	Transmission of raw bits over physical medium (electrical, optical, radio).	Ethernet cables, Fiber optics

Real-World Analogy

- Sending a letter:
- You write (application)
- Envelope (transport)
- Address (network)
- Mail truck (data link)
- Road (physical)

TCP/IP layers



IP Address and ports

- **Network Address (Host ID):**
  A host is identified by an IP:
  $$IP = (a_i, a_2, a_3, \mathrm{a}_4), 0 \leq a_i \leq 255$$
  (IPv4 as 32-bit integer).
- **Port Number (p):**
  Software-defined identifier for a process endpoint:
  $$0 \leq p \leq 65535$$
- **Port Ranges:**
  $0 \leq p \leq 1023$ *Privileged (system services)*
  $1024 \leq p \leq 49151$ *Registered (apps)*
  $49152 \leq p \leq 65535$ *Dynamic / private*

Routing

- **Hop-based forwarding:** If source and destination are not on the same LAN:

$$\text{Path}(S \to D) = \{R_1, R_2, \dots, R_h\}$$

  where h= number of hops (routers).

- **Routing Function** (per router R):

$$f_{route}(p) = \arg\min_{n \in N(R)} C(R,n) + D(n, dest(p))$$

  where:

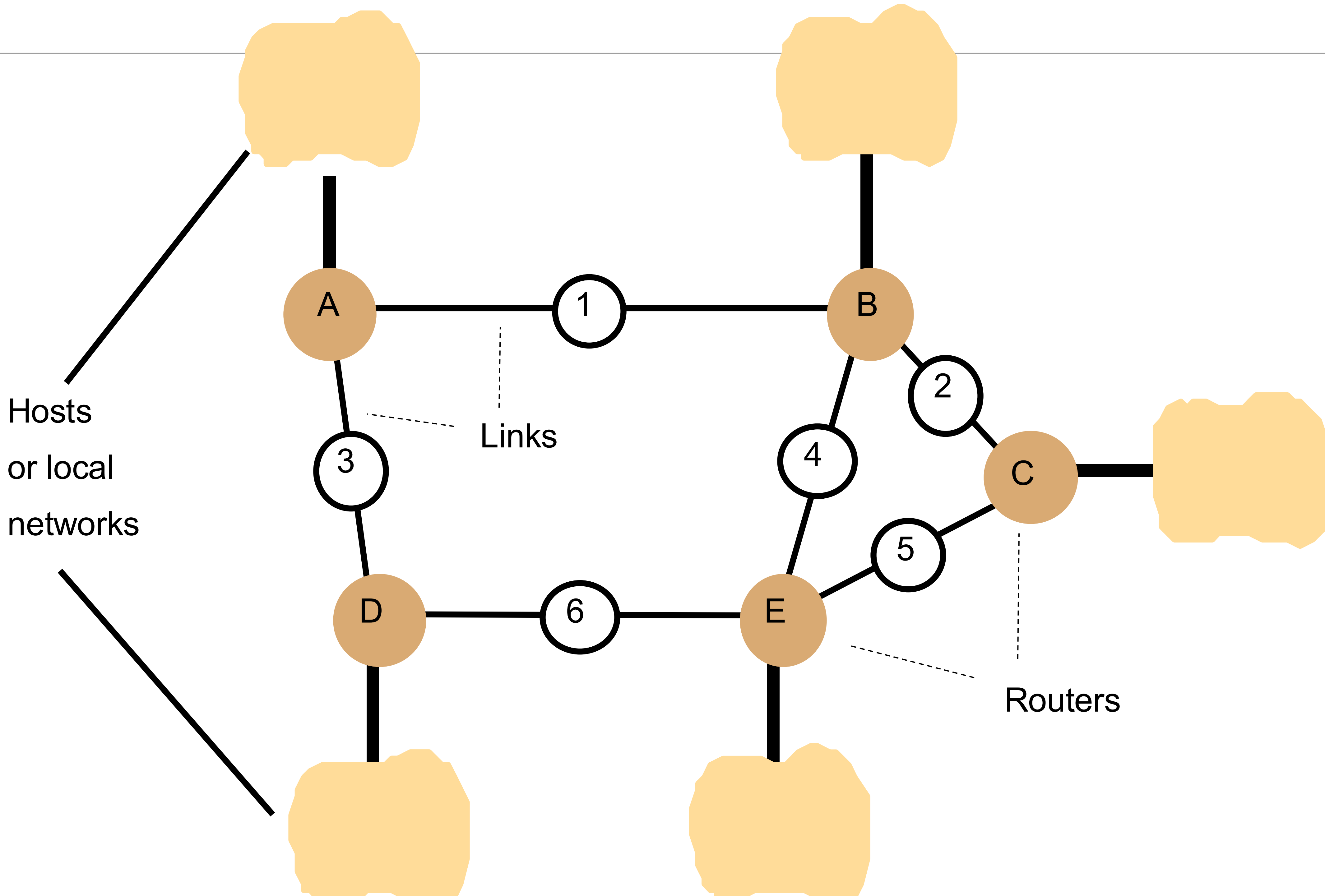
  - $N(R)$ =neighbors of router R,
  - $C(R,n)$ =cost of link $R \to n$,
  - $D(n, dest)$ =estimated distance from n to destination.

- **Distance-Vector (RIP) Update Rule:**

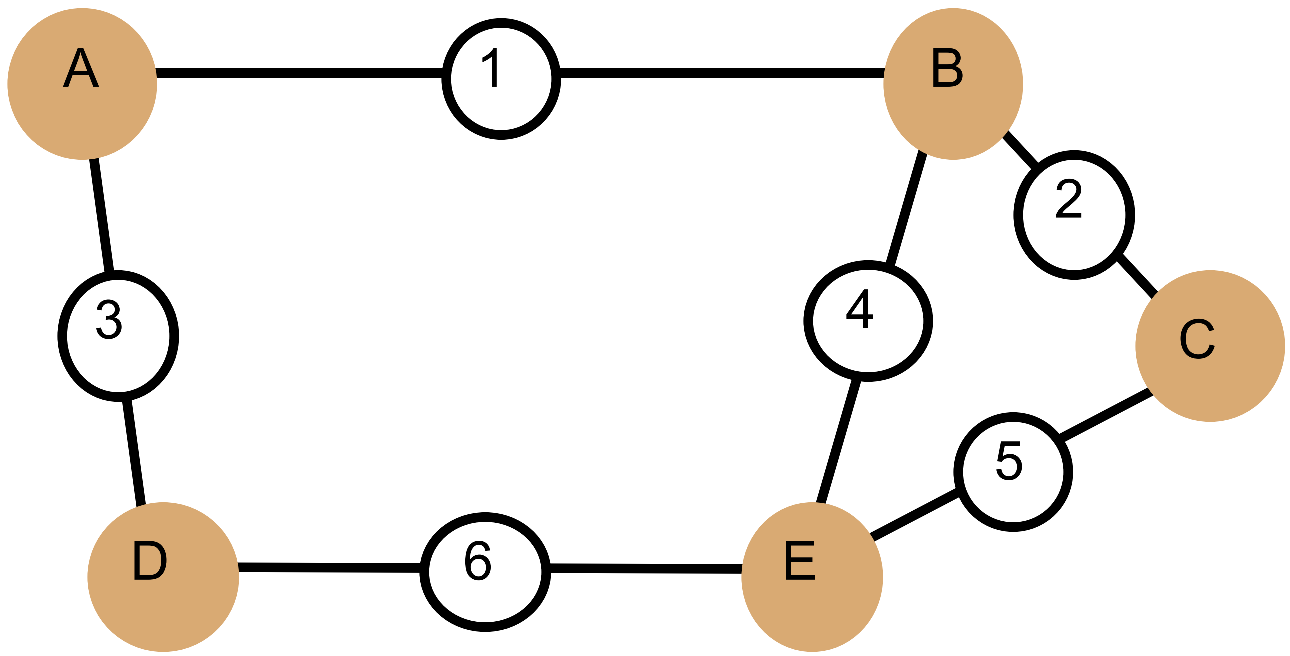
$$D_{new}(R, \mathrm{d}) = \min_{n \in N(R)}(C(R,n) + D(n,d))$$

  Each router sends its table $D(R,\cdot)$ to neighbors every t seconds.

Routing in a wide area network



Routing in a wide area network



Routing tables for the previous network

Routings from A

To	*Link*	*Cost*
A	local	0
B	1	1
C	1	2
D	3	1
E	1	2

Routings from B

To	*Link*	*Cost*
A	1	1
B	local	0
C	2	1
D	1	2
E	4	1

Routings from C

To	*Link*	*Cost*
A	2	2
B	2	1
C	local	0
D	5	2
E	5	1

Routings from D

To	*Link*	*Cost*
A	3	1
B	3	2
C	6	2
D	local	0
E	6	1

Routings from E

To	*Link*	*Cost*
A	4	2
B	4	1
C	5	1
D	6	1
E	local	0

Pseudo-code for RIP routing algorithm

```
# SEND
Every t seconds or when TL changes:
    Send TL on each active outgoing link.

# RECEIVE
On receiving table TR from neighbor e:
    For each entry Rr in TR:
        Rr.cost = Rr.cost + 1
        Rr.link = e
        if Rr.destination not in TL:
            Add Rr to TL
        else if Rr.cost < TL[Rr.destination].cost or TL[Rr.destination].link == e:
            TL[Rr.destination] = Rr
```

Example

Suppose **Router D** the local routing table TL at D is

Destination	Link	Cost
A	3	3
B	6	3
C	6	3
D	local	0
E	6	1

Example

Suppose **Router D** receives a routing table from **Router E** over link DE.

Apply RIP update rules When D receives E's table:

Increment cost by 1 for all entries (distance-vector hop count).

- A via E: cost = 2 + 1 = 3
- B via E: cost = 1 + 1 = 2
- C via E: cost = 1 + 1 = 2
- D via E: cost = 1 + 1 = 2 (but destination is self, ignore)
- E via E: cost = 0 + 1 = 1

Set next hop = 6 for all updated entries.

Example

Compare with D’s current table

Now update:

A: Current cost 3 via 3. New cost 3 via 6. Equal cost, but 6 is a valid alternative.

B: Current cost 3 via 3. New cost 2 via 6 → **update: better route**.

C: Current cost 6 via 6. New cost 2 via 6 → **update: much better route**.

E: Already known (cost 1). No change.

Destination	Link	Cost
A	3 (or 6)	3
B	6	2
C	6	2
D	local	0
E	6	1

UDP (User Datagram Protocol)

Packet Format:

$$P = (SrcPort, DstPort, Len, Checksum, Data)$$

Constraints:

$$|\, Data \,| \leq 2^{16} - 1 \approx 64\ kB$$

Properties:

Connectionless

No guarantee of delivery ($P_{loss} > 0$)

No sequencing, retransmission, or flow control

TCP/IP Protocol Suite

- **UDP (User Datagram Protocol):**
  - Connectionless, unreliable.
  - Packets:

$$P_i = (\mathrm{Src}, \mathrm{Dest}, \mathrm{Payload_i}), \textit{delivery not guaranteed}$$

- **TCP (Transmission Control Protocol):**
  - Reliable, connection-oriented.
  - Stream of bytes, ordered delivery.

TCP (Transmission Control Protocol)

Stream Abstraction: Reliable byte stream:

$$D = \{b_1, b_2, \dots, \mathrm{b_n}\}, n \to \infty$$

Properties:

- **Connection-oriented** (requires handshake).
- **Sequencing:** Each byte indexed by sequence number.

$$\text{Seq}(b_i) = i$$

- **Flow Control:** Receiver window $rwnd$.

$$\text{Allowed } data \leq rwnd$$

- **Congestion Control:** cwnd.

$$\text{Effective window=}\min(rwnd, cwnd)$$

- **Reliability:** Lost packets retransmitted.
- **Checksum:** ensures error detection.

TCP Congestion Control

- Define **congestion window** $cwnd$.
- Sending rate:

$$R \approx \frac{cwnd}{RTT}$$

Rules:

On ACK received:

$$cwnd \leftarrow cwnd + \frac{1}{cwnd}$$

(*additive increase*).

On packet loss:

$$cwnd \leftarrow \frac{cwnd}{2}$$

(*multiplicative decrease*).

UDP vs TCP Features (Simplified)

Feature	UDP	TCP
Header	SrcPort, DstPort, Len, Checksum	SeqNum, AckNum, Flags, Window, Checksum
Delivery	Unreliable (P_loss > 0)	Reliable (retransmission + ACKs)
Connection	Connectionless	Connection-oriented (3-way handshake)
Data size	≤ 64 KB	Arbitrarily long stream
Sequencing	None	Seq(b_i) = i
Flow Control	None	Allowed ≤ min(rwnd, cwnd)
Checksum	Yes	Yes

IP addressing

Requirements:

1. **Universality:**
   Every host has a unique address $IP \in [0,2^{32}-1)$ (IPv4).
2. **Efficiency:**
   Address space must minimize waste.
3. **Routing suitability:**
   Structure must allow aggregation (prefixes).

IP Classes (Legacy IPv4)

Class A: Prefix $bits = 0, N_{hosts} = 2^{24} - 2$

Very large networks.

Class B: Prefix $bits = 10, N_{hosts} = 2^{16} - 2$

Medium-sized organizations.

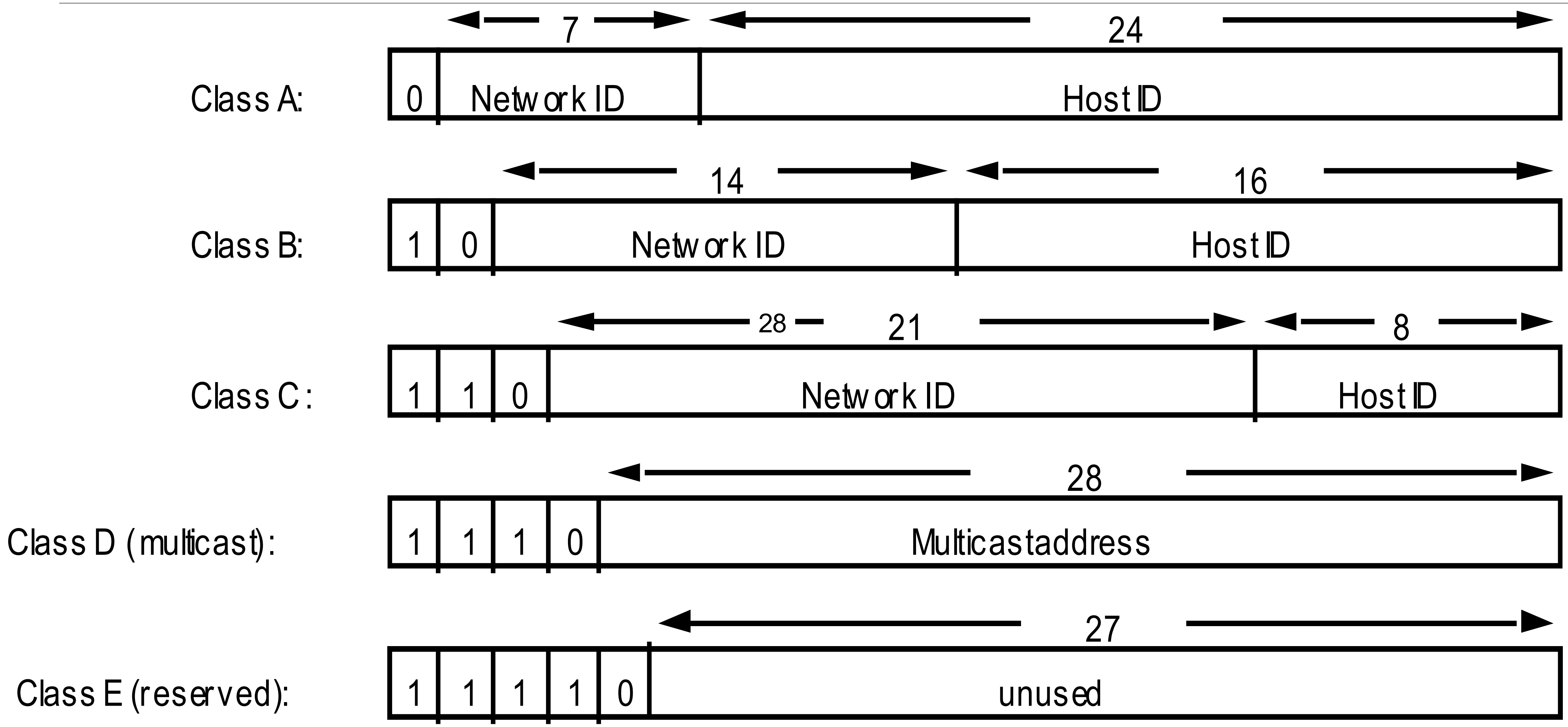
Class C: Prefix $bits = 110, N_{hosts} = 2^{8} - 2$

Small networks.

Class D: Prefix $bits = 1110, \mathit{Used\ for\ multicast}$

Class E: Prefix $bits = 1111, \mathit{Reserved\ (future\ use)}$

Internet address structure, showing field sizes in bits



Decimal representation of Internet addresses

	octet 1	octet 2	octet 3		Range of addresses
	Network ID	Host ID			
Class A:	1 to 127	0 to 255	0 to 255	0 to 255	1.0.0.0 to 127.255.255.255
	Network ID		Host ID		
Class B:	128 to 191	0 to 255	0 to 255	0 to 255	128.0.0.0 to 191.255.255.255
	Network ID			Host ID	
Class C:	192 to 223	0 to 255	0 to 255	1 to 254	192.0.0.0 to 223.255.255.255
	Multicast address				
Class D (multicast):	224 to 239	0 to 255	0 to 255	1 to 254	224.0.0.0 to 239.255.255.255
Class E (reserved):	240 to 255	0 to 255	0 to 255	1 to 254	240.0.0.0 to 255.255.255.255

Private vs Public IP Addresses

Not all hosts need global uniqueness.

Only edge devices (with Internet access) need **public IPs**.
Internal devices use **private IPs**.

DHCP (Dynamic Host Configuration Protocol):

Router dynamically assigns:

$$IP_{host} \sim Pool_{private}$$

(IP drawn from private block).

IANA-Reserved Private IPv4 Blocks

10.0.0.0 →10.255.255.255(2^{24} hosts)

172.16.0.0 →172.31.255.255(2^{20} hosts)

192.168.0.0→192.168.255.255(2^{16} hosts)

These addresses are **not routable on the global Internet**.

Access to Internet requires **NAT (Network Address Translation)** at the router.

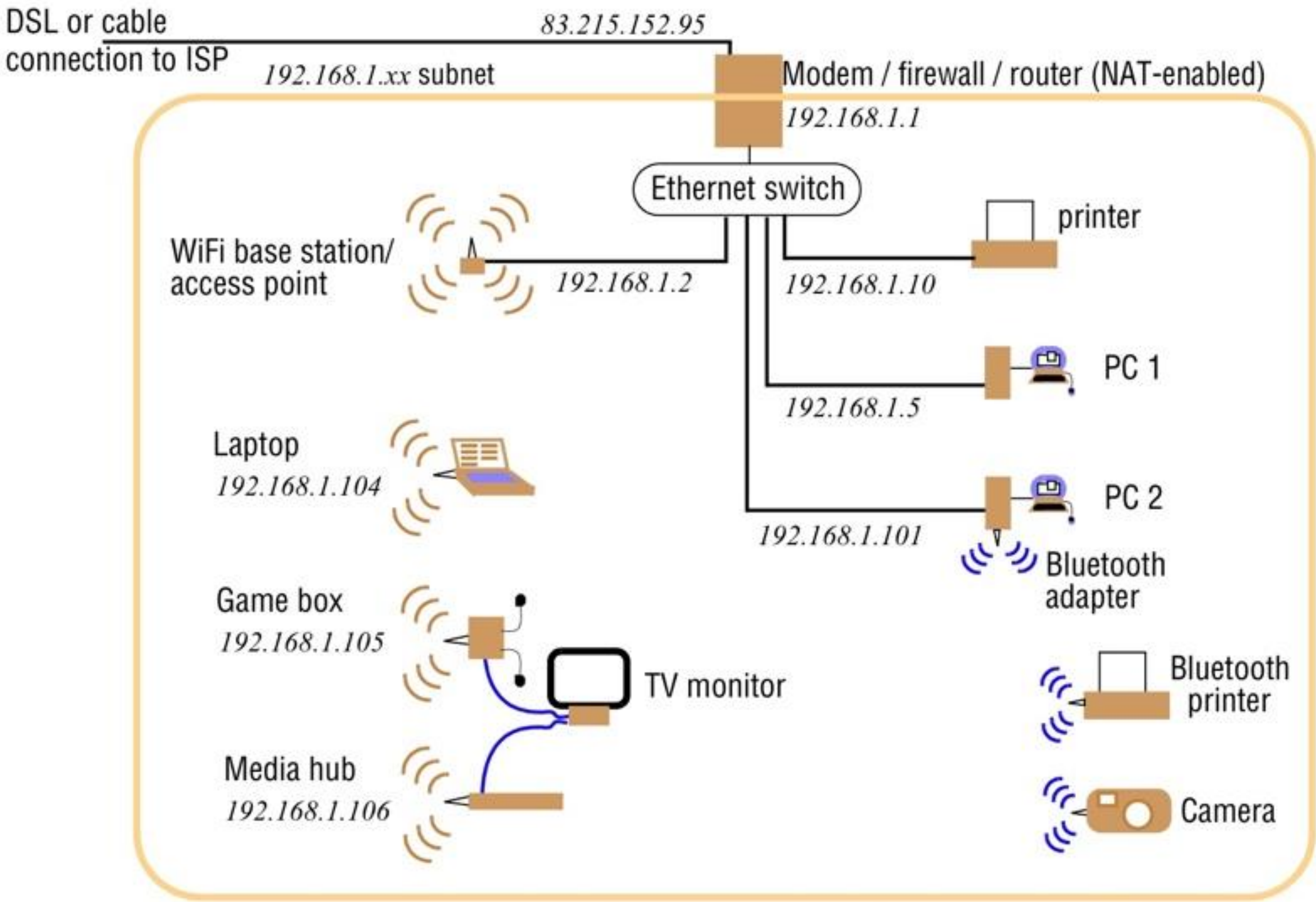
Public IP: unique in $[0, 2^{32} - 1]$.

Private IP: unique only within local subnet, reused globally.

NAT provides mapping:

$$(IP_{private}, port) \leftrightarrow (IP_{public}, port')$$

A typical NAT-based home network



IP version 6

Address Space:

$$|IP_{v6}| = 2^{128} \approx 3.4 \times 10^{38} \textit{ unique addresses}$$

Routing Efficiency: IPv6 header size is **fixed 40 bytes**. Processing per hop:

$$T_{proc}^{v6} < T_{proc}^{v4}$$

Services: Real-time traffic supported by **flow labels**. Next Header field → extensibility.

Multicast / Anycast: Multicast:

$$\text{Send}(src, M) \rightarrow \{d_1, d2_, \dots, d_k\}$$

Anycast: Send(*src, M*) $\rightarrow$ d_i *where* $d_i \in \{d_1, d2_, \dots, d_k\}$ *nearest*

Security: Built-in via **AH (Authentication Header)** and **ESP (Encapsulating Security Payload)**.

Architectural Models

CECS 327 Introduction to Networks and Distributed computing

Oscar Morales-Ponce

CECS, CSULB

Architectural models

Entities: Set of processes/hosts:

$$E = \{e_1, e_2, \dots, e_n\}$$

Each entity can send/receive messages.

Communication Paradigm: Function $C: E \times E \to M^*$

where M^* =set of message sequences. Examples: **message passing**, **RPC**, **publish–subscribe**.

Roles / Responsibilities Partition entities into roles:

$E = C_l \cup S_r \cup \cdots$ (e.g., Clients, Servers, Coordinators, Workers).

Topology: Graph representation:

$$G = (V, E_c), V = entities,\ \ E_c = connections$$

Can be **star, ring, mesh, tree** etc., mapping logical to physical infrastructure.

2

Communicating entities and communication paradigms

Communicating entities (what is communicating)		*Communication paradigms (how they communicate)*		
System-oriented entities	*Problem-oriented entities*	*Interprocess communication*	*Remote invocation*	*Indirect communication*
Nodes	Objects	Message passing	Request-reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Example

Online Banking System

Clients: User’s mobile banking app or browser.

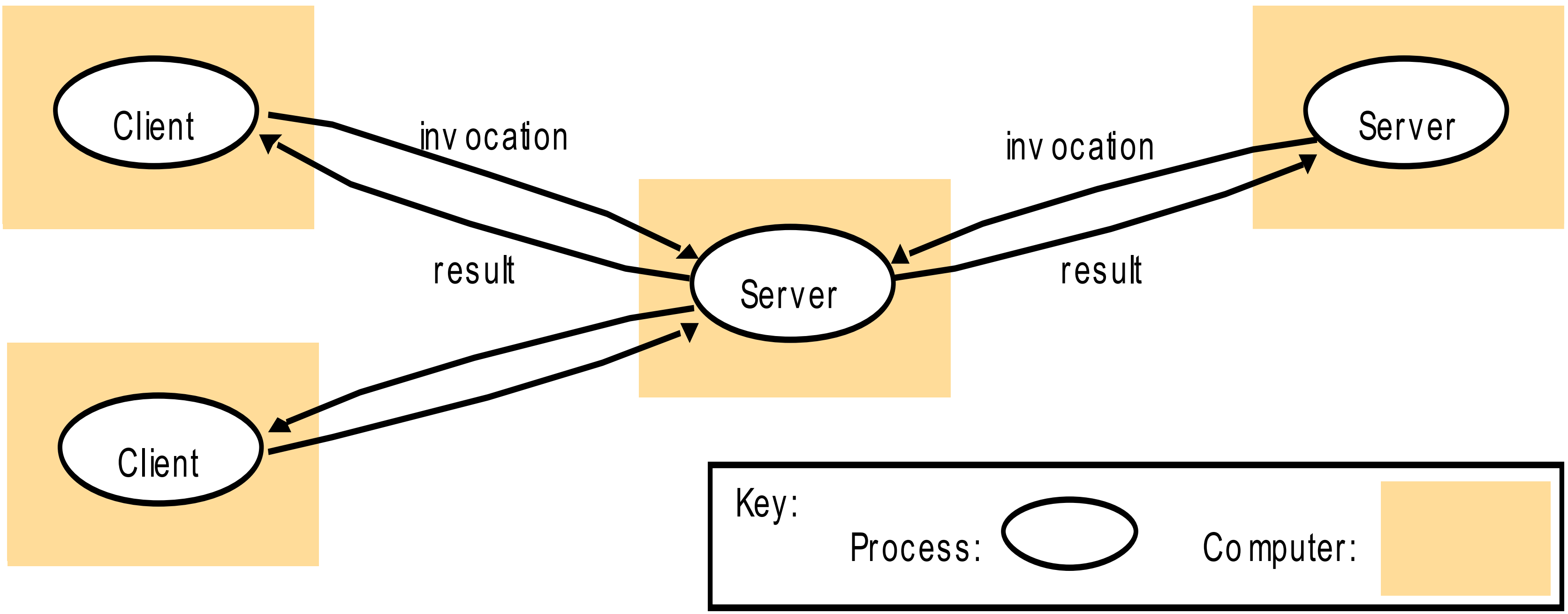
Servers:

- **Application server:** processes login, transactions, payments.
- **Database server:** stores account balances, transaction history.

Flow:

- Client sends request → “Check account balance.”
- Server processes request and queries the database.
- Response sent back to client for display.

Roles and responsibilities: Clients invoke individual servers



Example: **BitTorrent File Sharing**

Peers (1 … N):

- Each peer stores parts ("chunks") of a file.
- Peers upload/download chunks to/from each other.

Sharable Objects:

- File pieces are the sharable objects.
- Example: a movie file split into 100 chunks, spread across multiple peers.

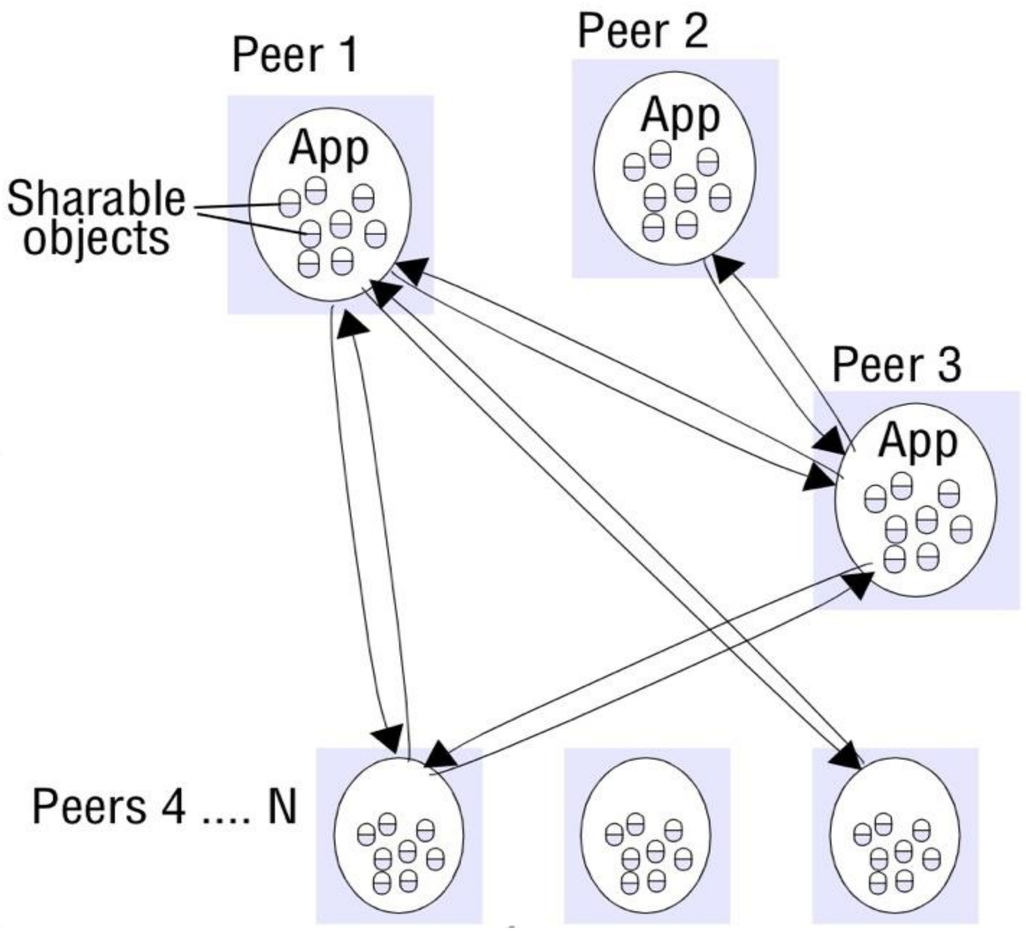
Application:

- BitTorrent client (uTorrent, qBittorrent, Transmission).
- Each peer runs the app and participates in both uploading and downloading.

Flow:

- Peer 1 requests missing chunks from Peer 2 and Peer 3.
- Peer 2 simultaneously downloads other chunks from Peer 4.
- Eventually all peers exchange pieces until each has a full copy.

Roles and responsibilities: Peer-to-peer architecture



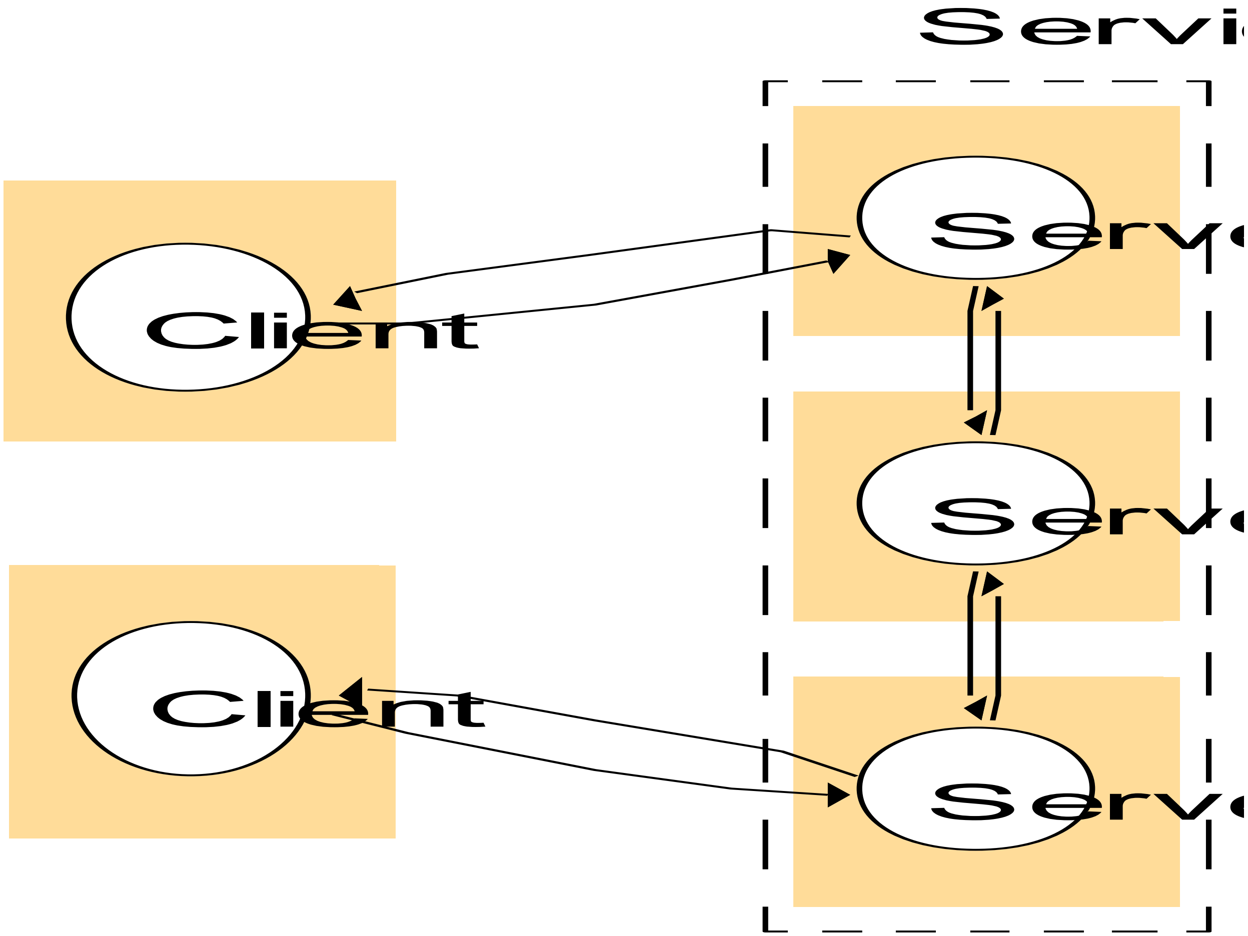
Example: Amazon.com

Clients: Shoppers using the Amazon website/app.

Servers:

- **Front-end servers** (load-balanced).
- **Application servers** (recommendation engine, checkout).
- **Database clusters** (product catalog, order history).

Placement: A service provided by multiple servers



Example: Online Multiplayer Game (e.g., World of Warcraft)

Clients:

- Player devices (PCs, consoles, mobile apps).
- They send actions (movement, chat, combat events) to the servers.

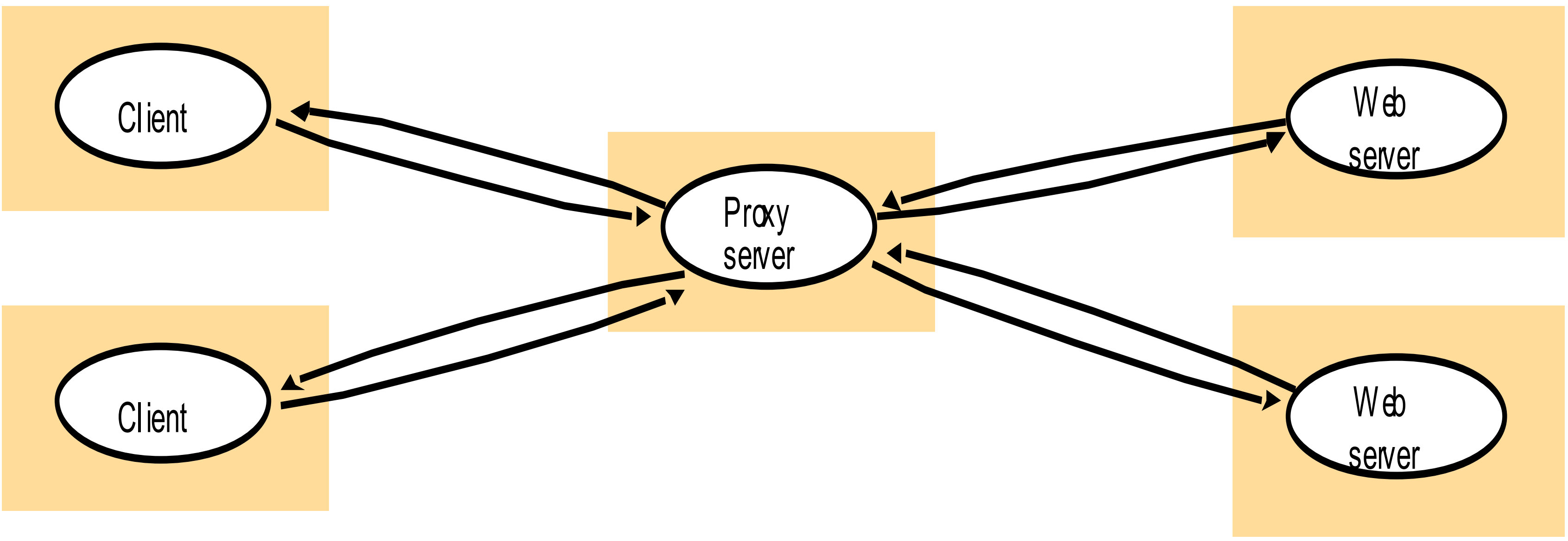
Servers:

- **Game Server 1:** manages part of the world (e.g., one game region).
- **Game Server 2:** manages another region.
- **Game Server 3:** coordinates special events or global state.
- Servers synchronize game state among themselves to keep consistency.

Flow:

- Client → sends player input to its assigned game server.
- That server processes the input and, if needed, communicates with other servers (e.g., cross-region interactions).
- Servers update their local state and return updates to clients.

Placement: Caching using a web proxy server



Architectural patterns: AJAX example

new Ajax.Request('scores.php?
game=Arsenal:Liverpool',
{onSuccess: updateScore});

function updateScore(request) {

.....

(*request* contains the state of the Ajax request including the returned result.

The result is parsed to obtain some text giving the score, which is used to update the relevant portion of the current page.)

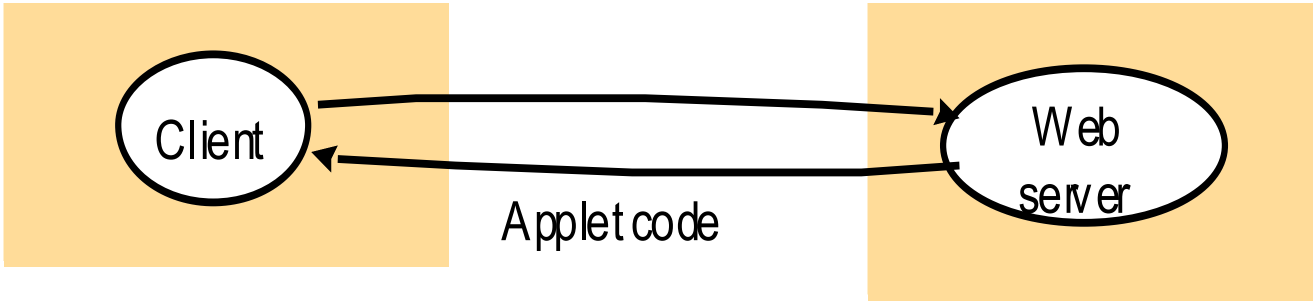
.....

}

12

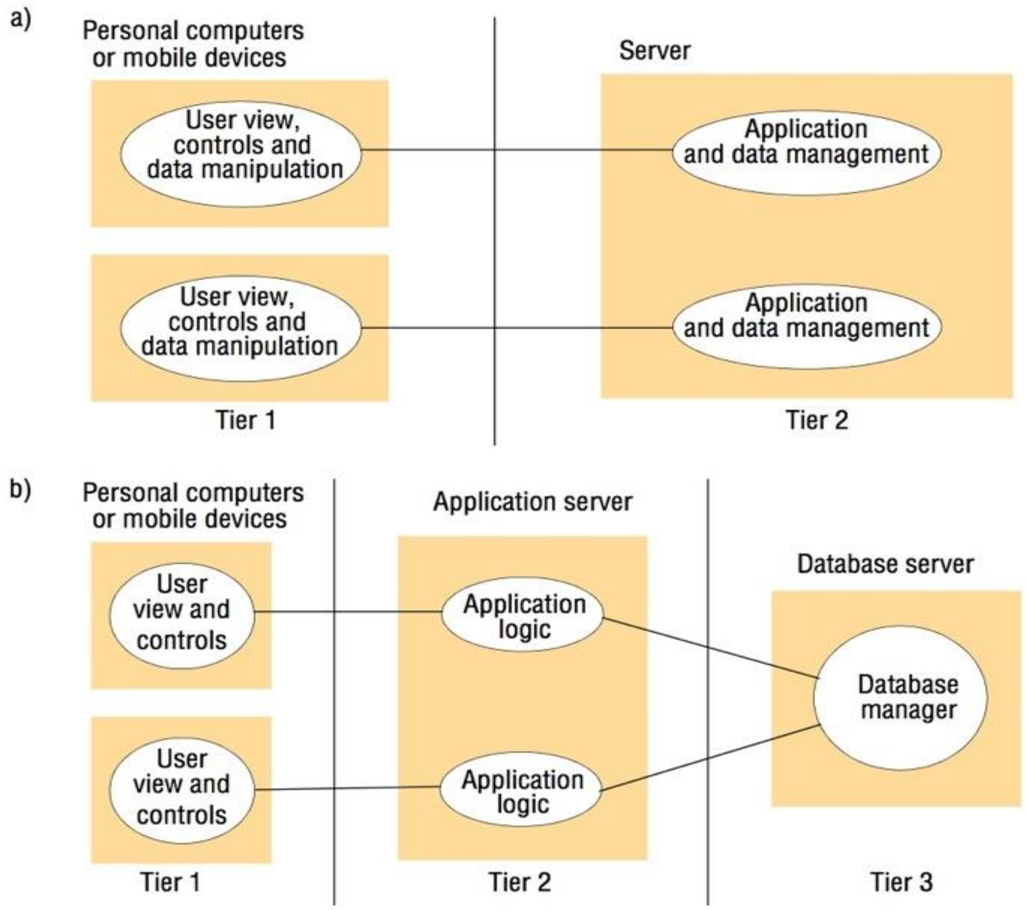
Placement: Mobile Code (web applets)

a) client request results in the downloading of applet code

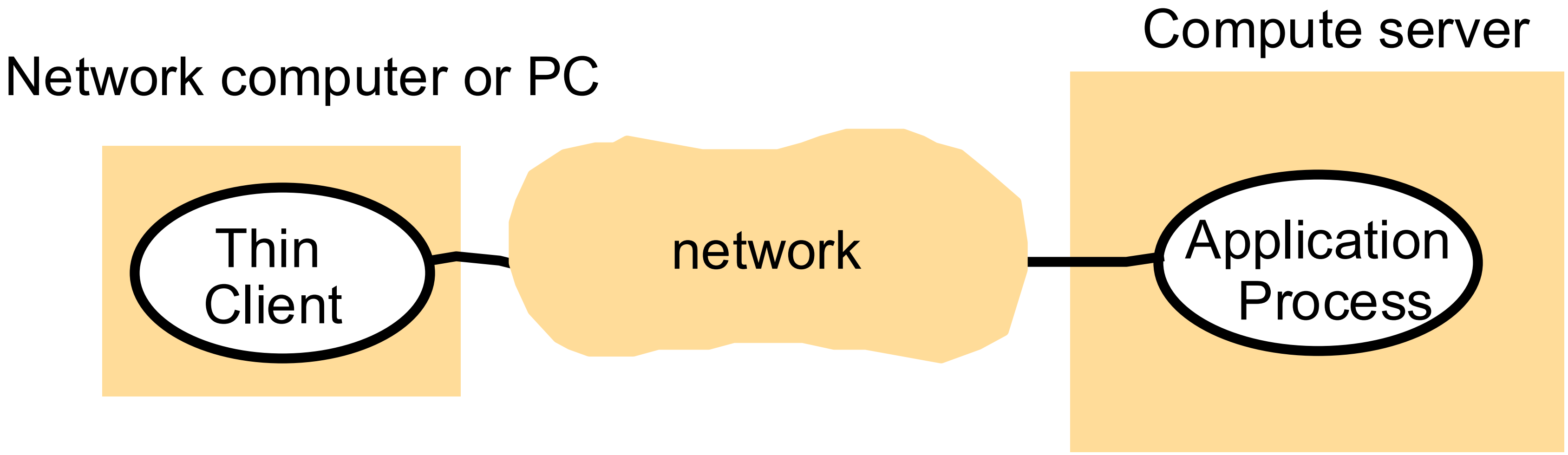


b) client interacts with the applet

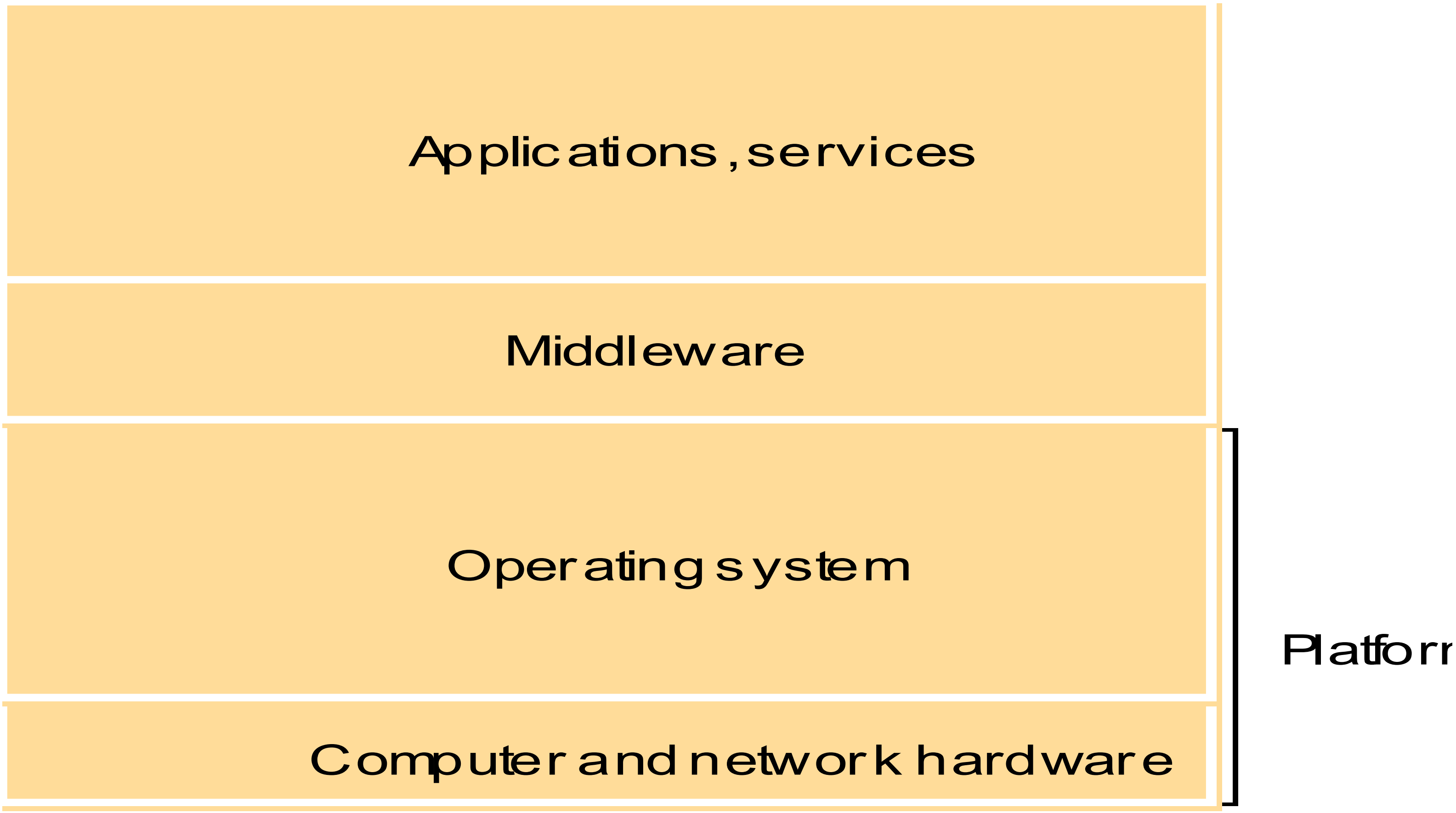
Architectural patterns: Two-tier and three-tier architectures



Thin clients:



Architectural patterns: Software and hardware service layers in distributed systems



Fundamental models

Interaction:

- Processes exchange messages: $m: p_i \rightarrow p_j$
- Must ensure **synchronization** and **ordering**:

$$\text{Order}(m_1, m_2) \in \{Casual, total, none\}$$

Failure:

- Fault model:

$$F = \{crash, omission, timing, Byzantine\}$$

- **Security**
  - Threat model:

$$S = \{eavesdrop, modify, impersonate, denialofservice\}$$

- **Distributed Algorithms**
  - Each process $p_i$maintains **state** $s_i(t)$.
  - Behavior defined by transition function:

$$s_i(t+1) = f(s_{i'}(t) m_{in}(t))$$

17

Network Performance Metrics

Latency (L): Delay between transmission start and first bit arrival:

$$L = t_{receive\_start} - t_{send\_start}$$

Bandwidth (BW): Maximum data transferred per unit time:

$$BW = \frac{\text{bits transmitted}}{\text{time}}$$

Jitter (J)
Variation in message delivery time:

$$J = Var(t_{delivery})$$

or equivalently:

$$J = \max_i \quad (t_i - t_{i-1}) - \min_i \quad (t_i - t_{i-1})$$

18

Clock Behavior in Distributed Systems

Local Clock: Each processor i has its own clock:

$$C_i(t) = t + \epsilon_i(t)$$

where t= true (reference) time, $\epsilon_i(t)$ =clock error.

Clock Drift: Real clocks are imperfect:

$$\frac{dC_i(t)}{dt} = 1 + \rho_i$$

where ρ_i =**drift rate** (deviation from perfect clock).

Bound on Drift: Typically:

$$|\rho_i| \leq \rho_{max}$$

e.g., $\rho_{max} \approx 10^{-6}$)1 microsecond drift per second).

19

Synchronous distributed systems

Process Execution Bounds: Each step of a process p_i executes within:

$$T_{min} \leq T_{step}(p_i) \leq T_{max}$$

Message Delay Bounds: For a message $m$sent over a channel:

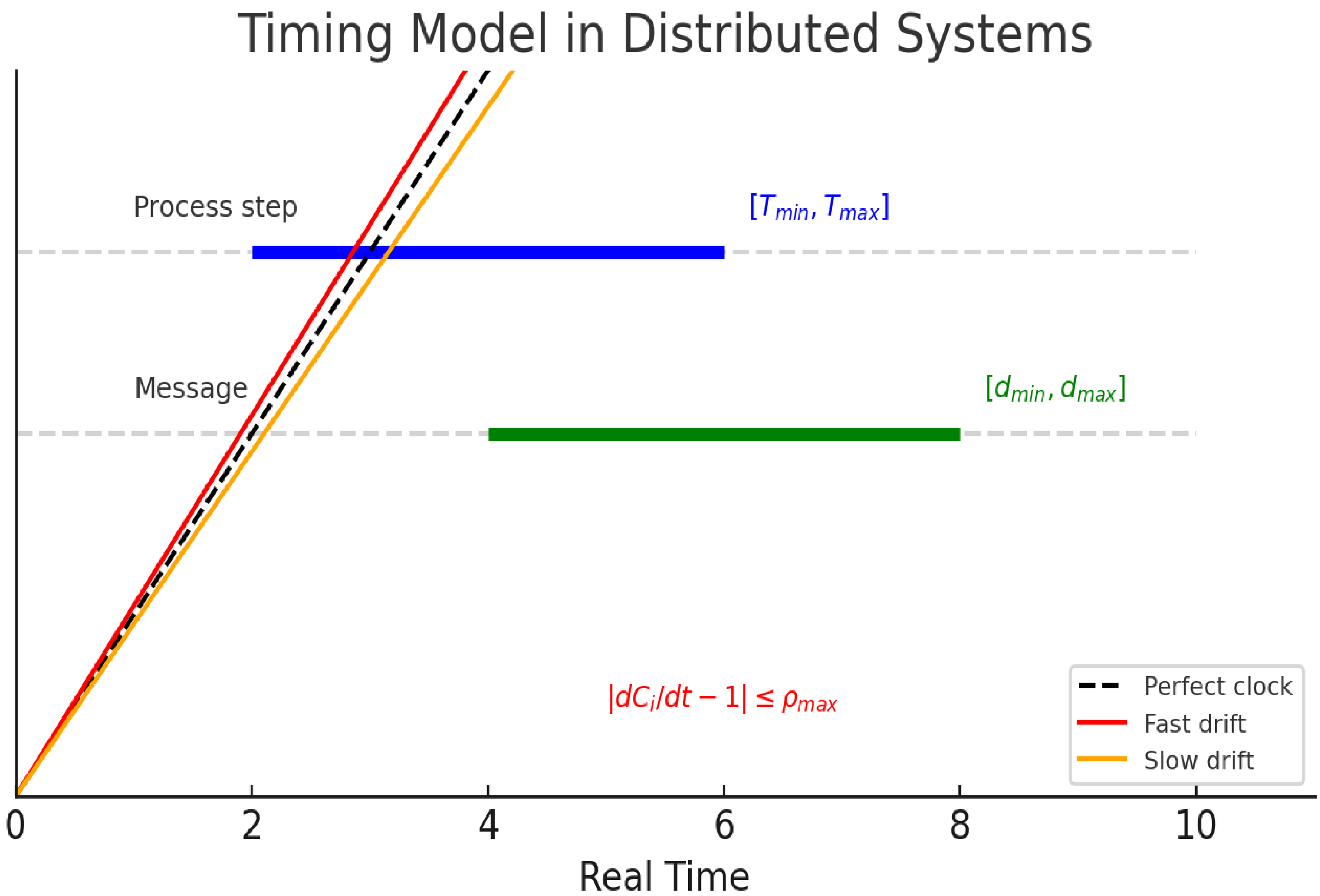
$$d_{min} \leq T_{msg}(m) \leq d_{max}$$

Clock Drift Bounds: For local clock $C_i(t)$:

$$\left| \frac{dC_i(t)}{dt} - 1 \right| \leq \rho_{max}$$

where $\rho_{max}$is the maximum drift rate.

20



Timing Model in Distributed Systems
Process step
$[T_{min}, T_{max}]$
Message
$[d_{min}, d_{max}]$
$|dC_i/dt - 1| \leq \rho_{max}$
Perfect clock
Fast drift
Slow drift
0
2
4
6
8
10
Real Time

21

Asynchronous distributed systems

Process Execution:

$$T_{step}(p_i) \quad \textit{unbounded}$$

(no upper or lower bound on execution time).

Message Transmission:

$$T_{msg}(m) \quad \textit{unbounded}$$

(delivery may take arbitrarily long, or never).

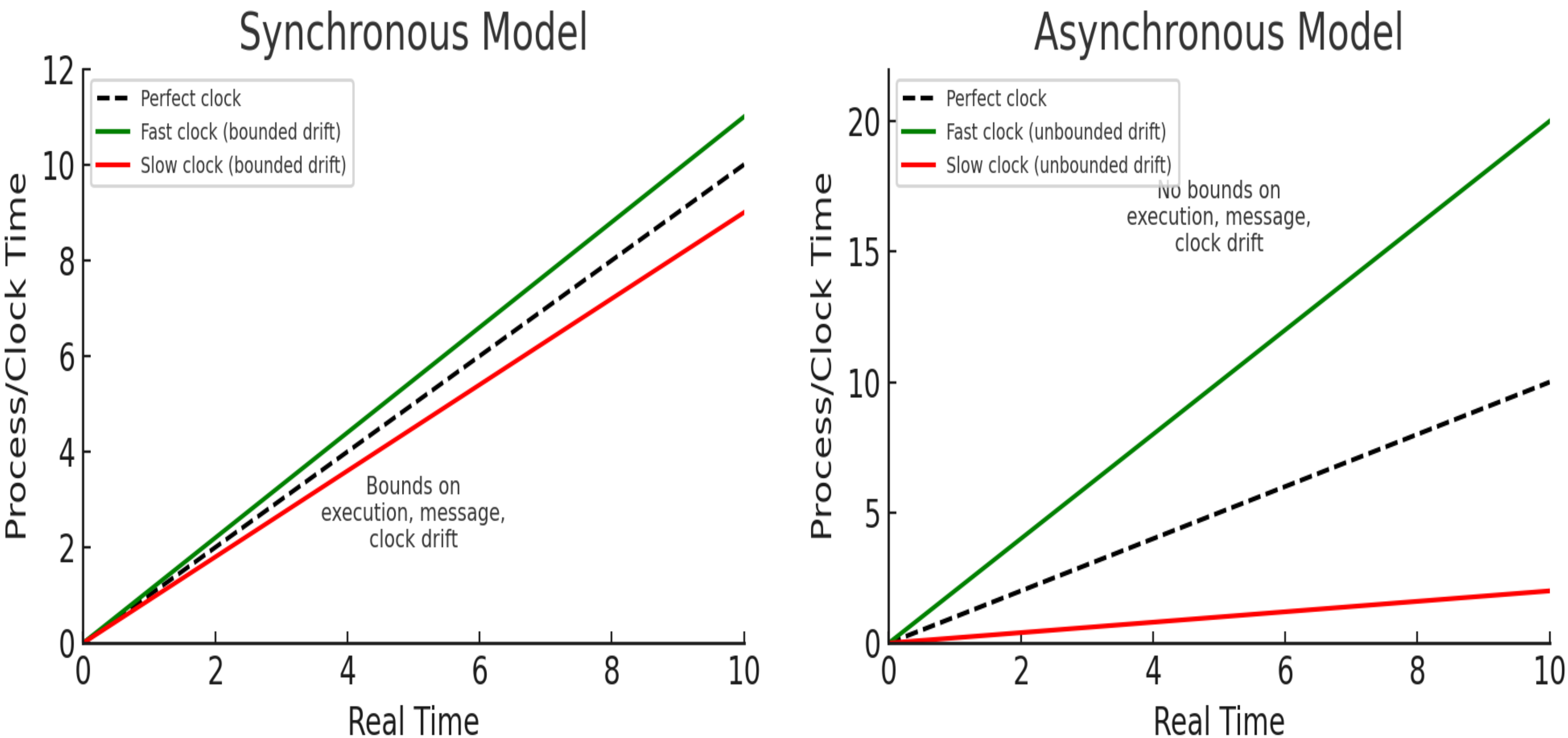
Clock Drift:

$$\left| \frac{dC_i(t)}{dt} - 1 \right| \quad \textit{unbounded}$$

(no limit on local clock deviation from real time).

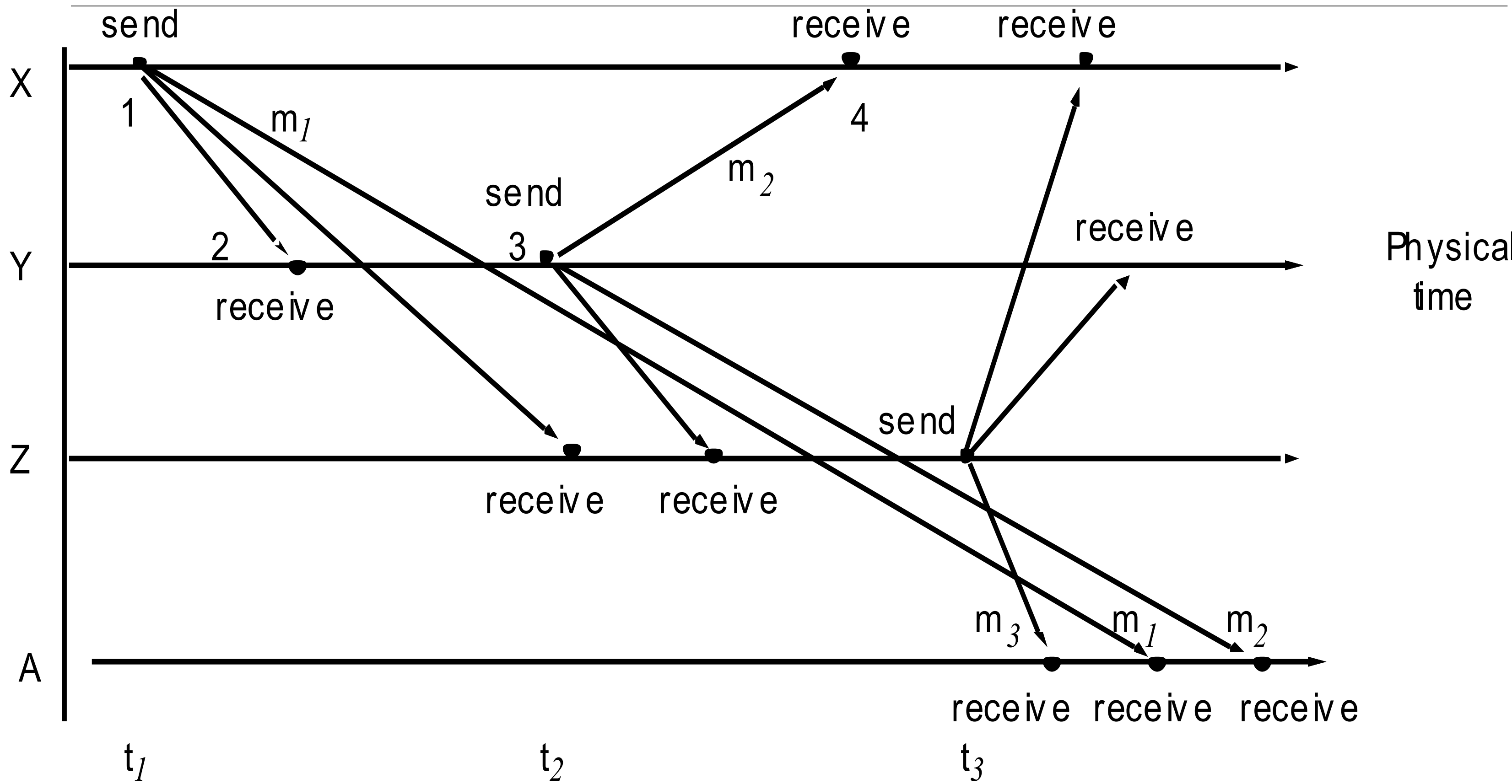
22

Synchronous vs Asynchronous Models



23

Real-time ordering of events



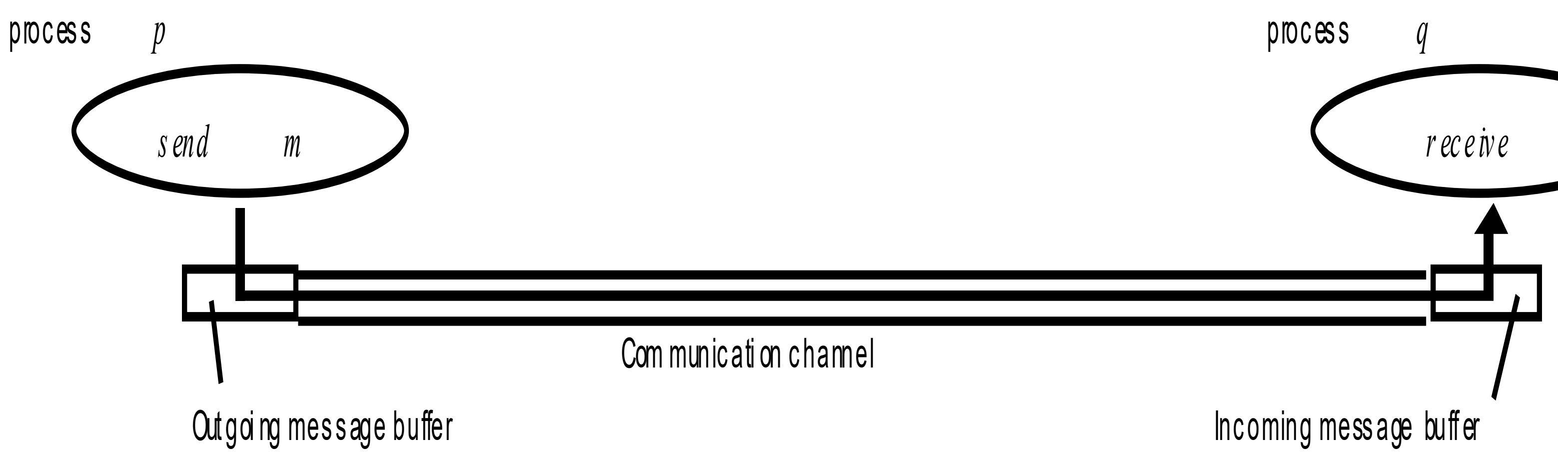
Failure Model: Omission and arbitrary failures

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send,* but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Failure Models in Distributed Systems

Failure Class	Who is Affected	Description (Simplified + Math)
Crash	Process / Node	Process halts and takes no further steps. After time t_c ,no actions: $\forall t > t_c,\ state(p) = \emptyset$.
Omission	Process or Channel	Messages or responses are lost. Send/receive not guaranteed: $m \notin I_j$.
Timing	Process / Network	Actions occur outside expected bounds. $T < T_{min}$ or $T > T_{max}$.
Arbitrary (Byzantine)	Process / Node	Any behavior possible: crash, lies, inconsistent or malicious. Can send different values to different processes.

Failure Model: Processes and channels



Failure Model: Timing failures

Class of Failure	Who is Affected	Description
Crash	Single clock / process	Clock stops advancing (frozen time).
Omission	Processes relying on sync	Missed updates or lost clock messages.
Drift (Timing)	All processes using local clock	Clock runs faster/slower: $\frac{dC_i(t)}{dt} = 1 + \rho_i$,
Byzantine	Other processes in system	Arbitrary/malicious faults: clock gives inconsistent or incorrect values.

Reliability of one-to-one communication

Validity (Liveness: eventually delivery): If a process p_i sends message m to p_j:

$$m \in O_i \quad \Rightarrow \quad \exists t: m \in I_j$$

(E*very message placed in the sender's outgoing buffer* O_i *is eventually delivered to the receiver's incoming buffer* I_j)

Integrity (safety: correct content, no duplication).

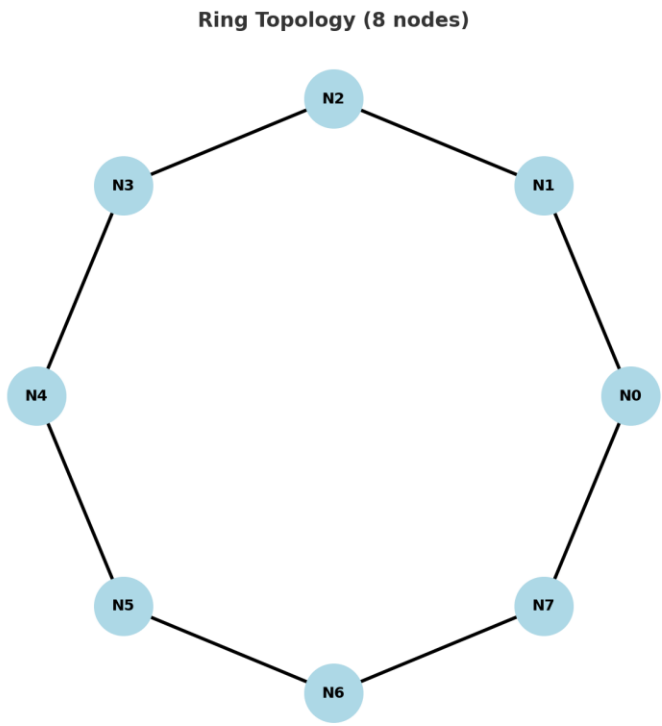
- No modification:

$$m_{recv} = m_{sent}$$

- No duplication:

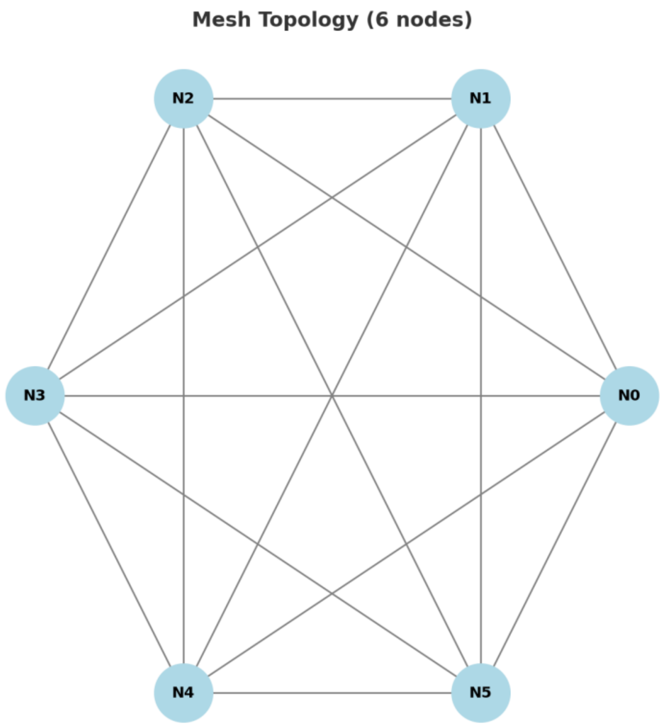
$$\#recv(m) \leq 1$$

Ring with n nodes



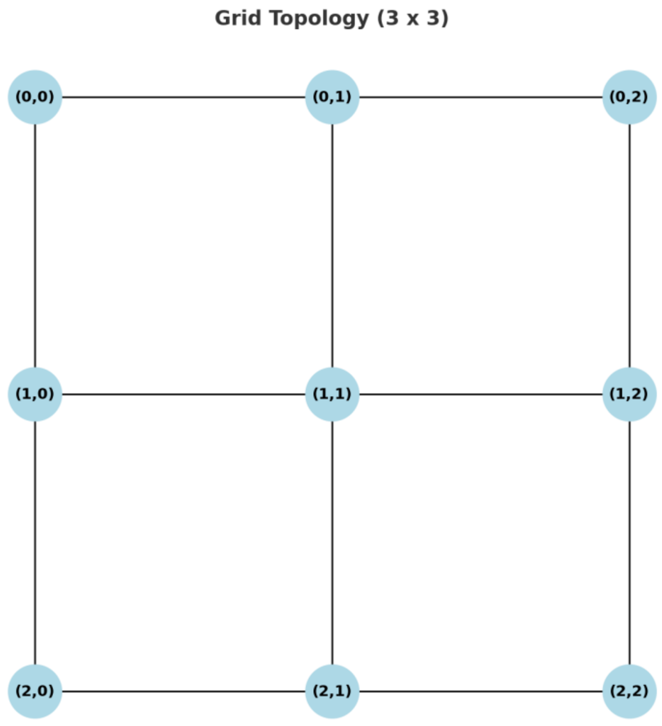
Number of nodes = n
Degree per node = 2
Diameter = floor(n/2)
Resilient but slower than mesh

Complete graph with n nodes



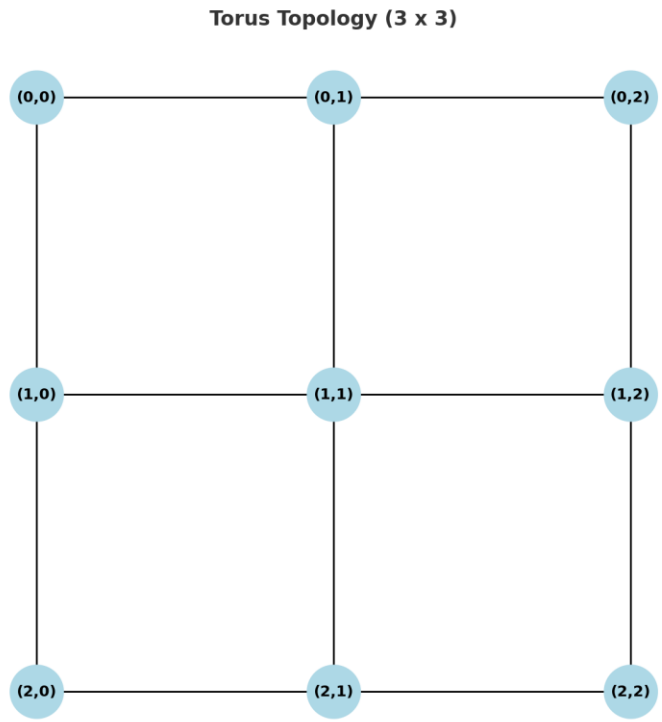
Number of nodes = n
Number of links = n(n-1)/2
Degree per node = n - 1
Diameter = 1 (any node directly connects)

Grid nxm



Number of nodes = $n \times m$
Max degree per node = 4 (interior nodes)
Boundary nodes have degree ≤ 3
Corner nodes have degree = 2
Diameter = $(n - 1) + (m - 1)$

Torus Topology (3x3 Grid with Wrap-around)



Number of nodes = n × m
Degree per node = 4 (all nodes equal)
Diameter ≈ floor(n/2) + floor(m/2)
Wrap-around edges connect borders

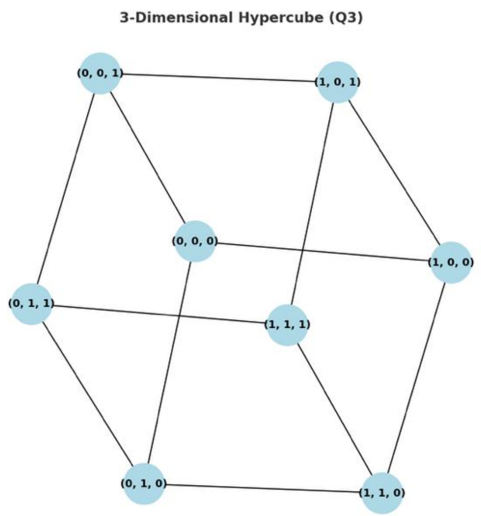
Hypercube of dimension d

1. Start with Q1: two nodes connected by an edge.

2. To form Q(k+1): take two copies of Qk and connect corresponding nodes.

3. Each node in Q3 is represented by an 3-bit binary string.

4. Two nodes are connected if their binary labels differ in exactly one bit.



Number of nodes = 2^3 = 8
Max degree per node = 3
Diameter = 3