Distributed Storage Systems

Introduction

Basics of Socket Programming

Credits: Some slides adapted from David M. Beazley

Agenda

Today's topics

- Course introduction
- Introduction to distributed storage systems
- Internet technologies
    - Some background
    - UDP/TCP/IP
- Sockets
    - Basics
    - Python examples

Course Introduction

- **Objectives:** introducing students to fundamentals of distributed storage systems
- **Learning outcomes:**
  - Understand fundamentals of distributed storage systems,
  - Analyse performance trade-offs between different storage approaches,
  - Explain the operation of relevant systems and protocols for distributed storage,
  - Evaluate and test algorithms in controlled, local deployments

Format of the course

- Lectures + Practical work every week
  - Learn theory (2h)
  - Get your hands dirty with practical systems: not as a user, but as a system implementer/designer (2h)

- Typical meeting times:
  - Mondays 12-16

- Exam: Individual Oral Examination (20 min each) based on Report and Course Slides
  - Report: focused on prototypes developed

Class Structure

Week #	Lab #	Date	Lecture	Lab
Week 1	Lab 1	Aug 30	Course intro, networking basics, socket programming	Python sockets
Week 2	Lab 2	Sept 6	RPC, NFS, JSON RPC, REST API	JSON-RPC with tinyrpc, REST API with Flask
Week 3	Lab 3	Sept 13	AFS, reliable storage intro	ZeroMQ, Protocol Buffers
Week 4	Lab 4	Sept 20	Hard drives, RAID levels	RPi Stack Handout RPi stack intro, RPi RAID with ZMQ
Week 5	Lab 5	Sept 27	Finite fields, Reed-Solomon Codes	Kodo intro, RS and RLNC with Kodo
Week 6	Lab 6	Oct 4	Repair problem, RS vs Regenerating codes	RPi simple distributed storage with Kodo RS
Week 7	Lab 7	Oct 11	Regenerating codes, XORBAS	RPi Regenerate lost fragments with RS
Week 8	-	Oct 18	Autumn Break	
Week 9	Lab 8	Oct 26	Hadoop	RPi RLNC, recovery with recode
Week 10	Lab 9	Nov 1	Storage Virtualization, Network Attached Storage, Storage Area Networks	Mini project introduction + RPi basic HDFS (namenode+datanode, read and write pipeline)
Week 11	Lab 10	Nov 8	Object Storage	RPi basic S3 API
Week 12	-	Nov 15	Compression, Delta Encoding	Mini project consultation
Week 13	Lab 11	Nov 22	Data Deduplication	RPi compression+dedup
Week 14	-	Nov 29	Fog storage	Mini project consultation
Week 15	-	Dec 6	Security for Storage Systems	Mini project consultation

Goals

Today's Learning Goals

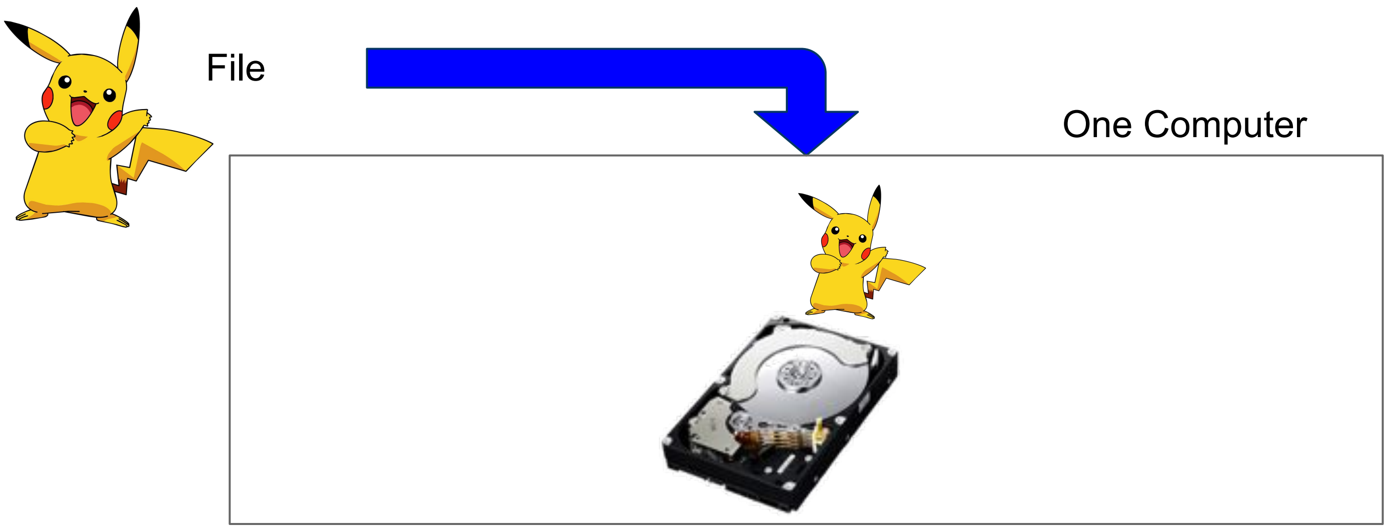
- Understand distributed storage systems background
- Review basics of TCP/IP
- Be able to create a small application communicating with another application over TCP/IP

Data Storage

- Basic problem:
  - Storing data persistently
  - Storing data reliably: no corruption, no loss (almost)
  - Cope with increasing amount of data

Data Storage

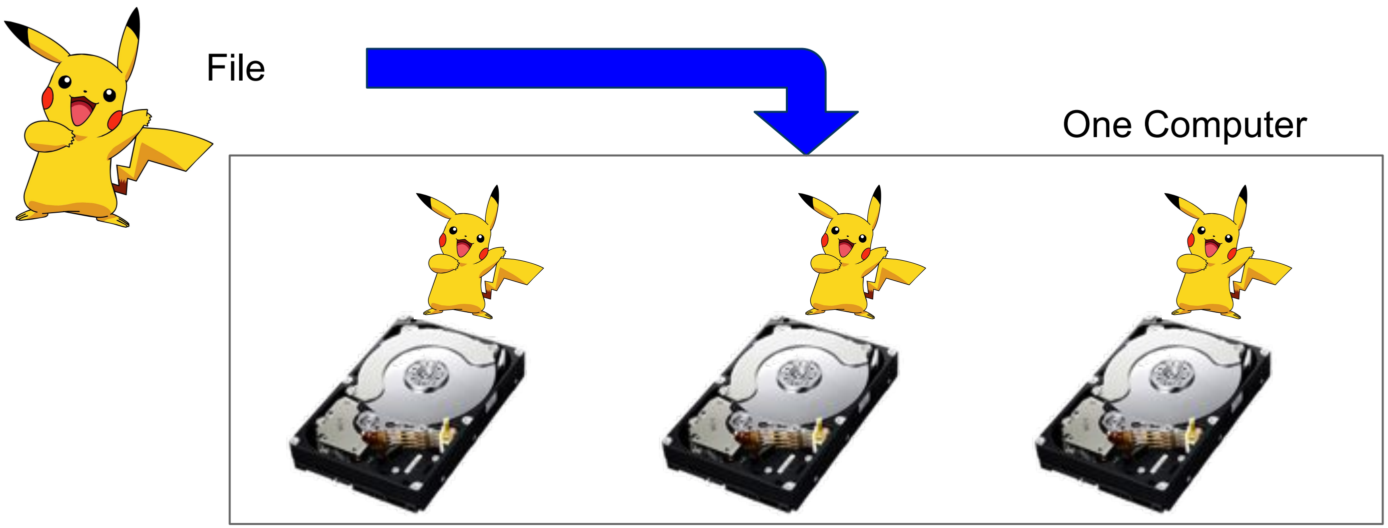
- Basic problem:
  - **Storing data persistently**
  - Storing data reliably: no corruption, no loss (almost)
  - Cope with increasing amount of data



Individual **disks** are unreliable

Data Storage

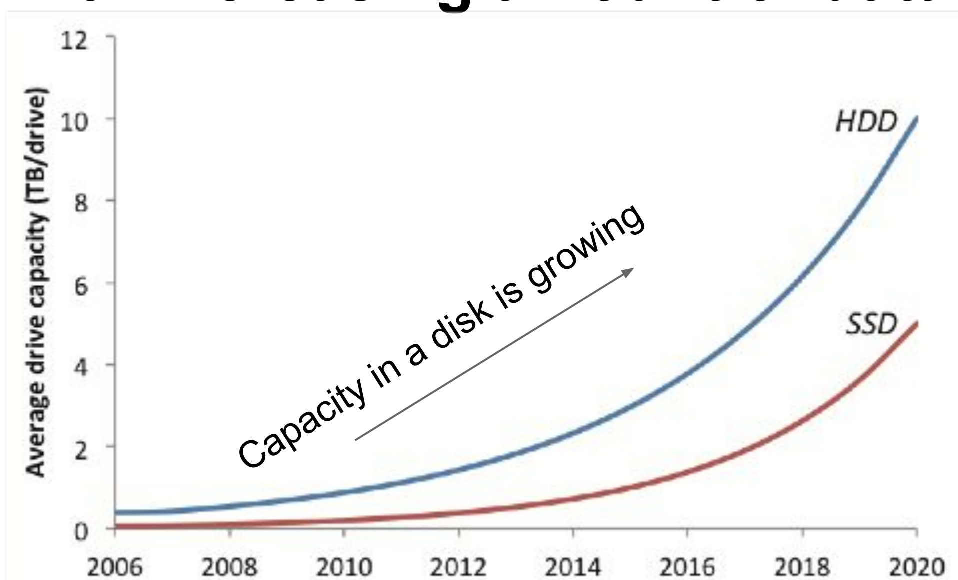
- Basic problem:
  - Storing data persistently
  - **Storing data reliably: no corruption, no loss (almost)**
  - Cope with increasing amount of data



Individual **servers** are unreliable

Data Storage

- Basic problem:
  - Storing data persistently
  - Storing data reliably: no corruption, no loss (almost)
  - **Cope with increasing amount of data**



Size of data exceeds that of a single drive (or several drives in one server)

Data Storage

- Basic problem:
  - Storing data persistently
  - Storing data reliably: no corruption, no loss (almost)
  - Cope with increasing amount of data
- More advanced problem:
  - Availability of data
  - Access speed
  - Computation over large amount of data

Distributed Storage Systems

Reliability requirements, physical limitations of components, size of data, computation

Multiple machines for a task

Cope with reliability, availability requirements

Slide adapted from Daniel Leblanc

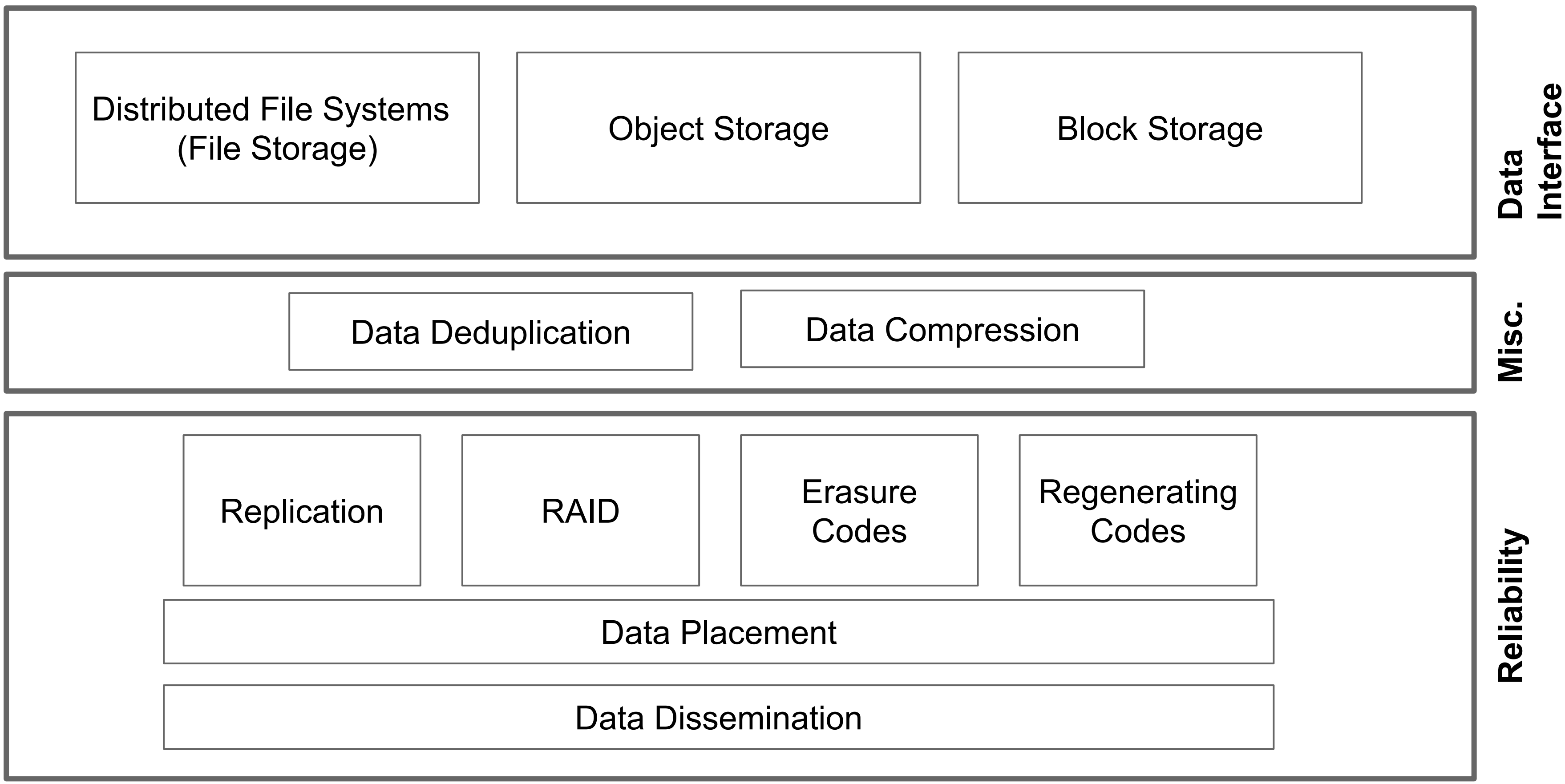
Distributed Storage Systems

- Programming on a distributed system is much more complex
  - Synchronizing data exchanges
  - Managing a finite bandwidth between servers
  - Ensuring consistency of data
- Distributed (storage) systems must be designed with the **expectation of failure**

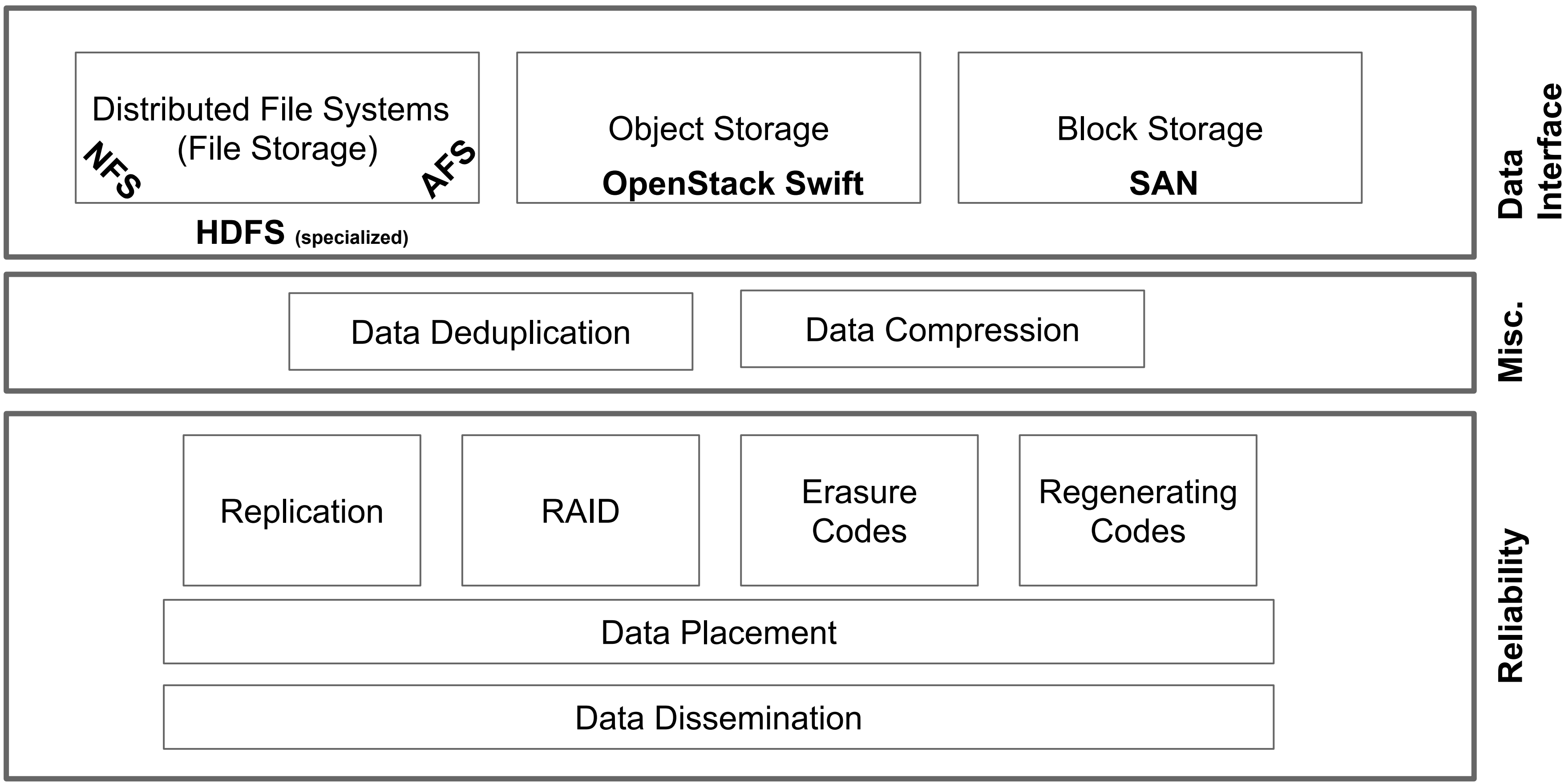
What will we do in the course?

- Basics of networking Socket programming, Remote Procedure Calls (RPC)
- NFS as an example of RPC for distributed storage
- Managing reliability, e.g., use of replication, RAID
- Theoretical aspects of distributed storage, including, basics of erasure codes (e.g., RAID, Reed-Solomon, network coding) for reducing costs of reliability and network use
- Theoretical and practical discussions on data locality for data repair, but also in the context of BigData analysis (e.g., MapReduce)
- Architectures
- Basics of operation of real systems (e.g., Hadoop - HDFS, OpenStack Swift)
- Compression and Data Deduplication

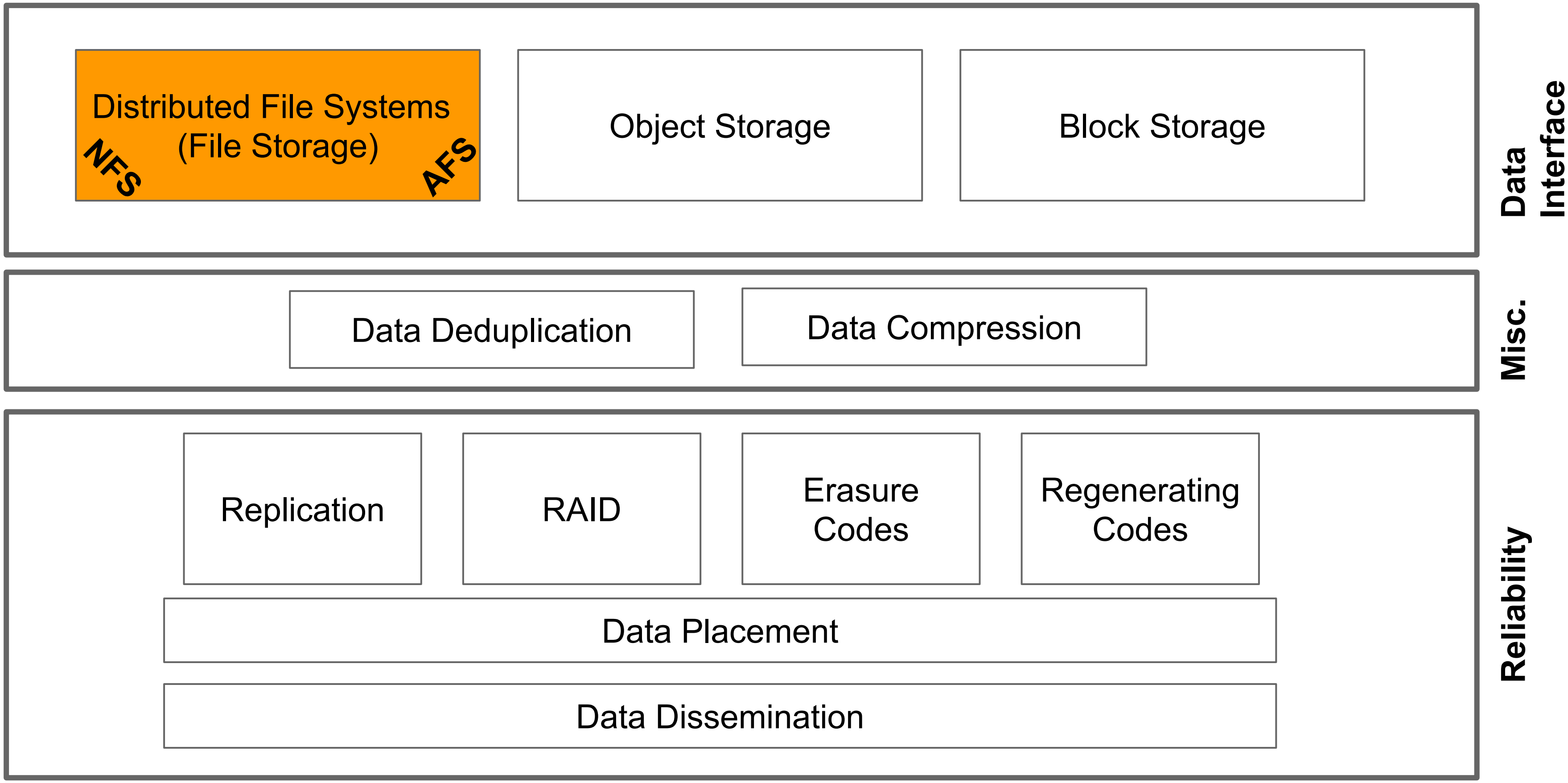
Storage Systems



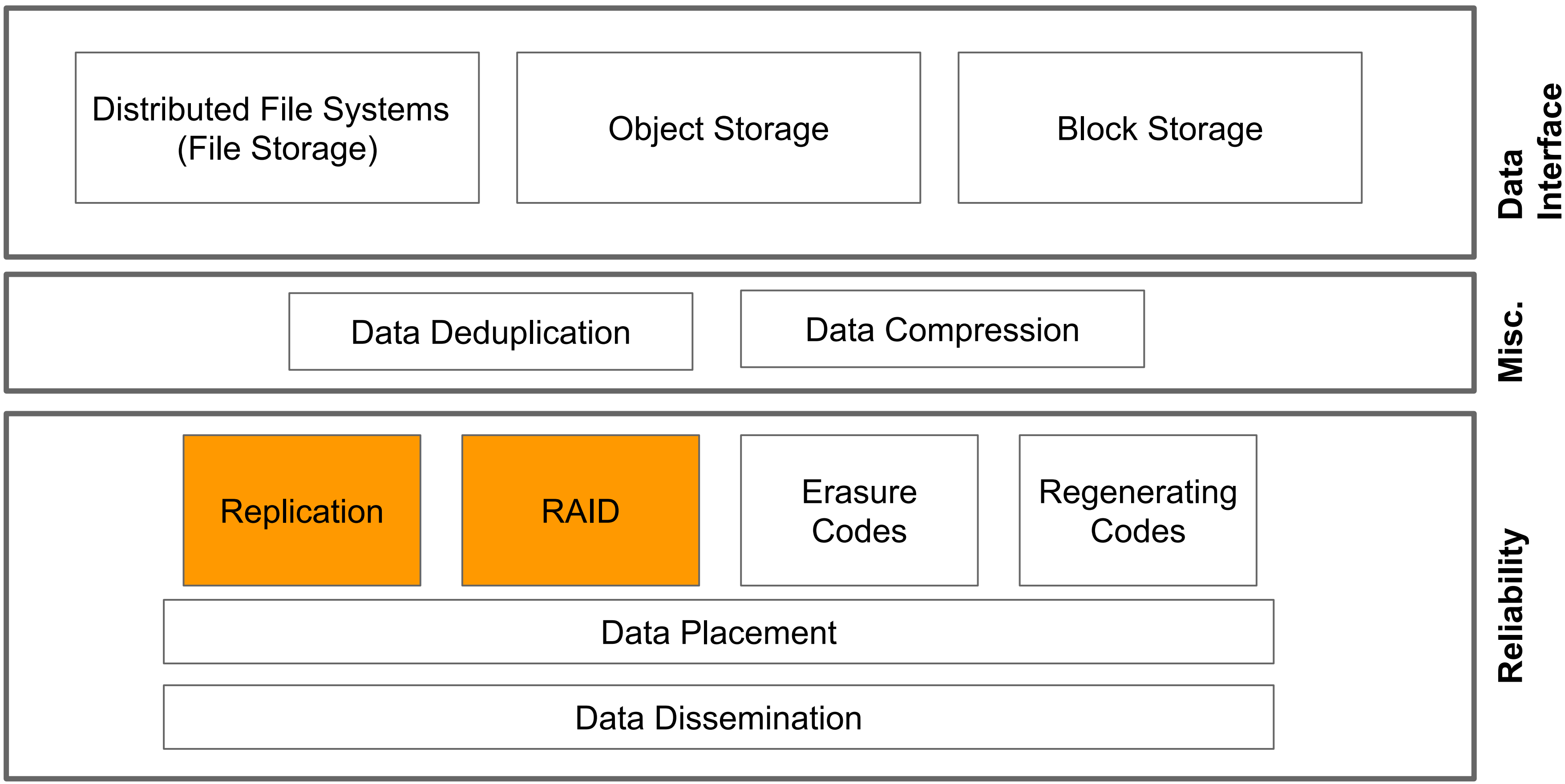
Storage Systems



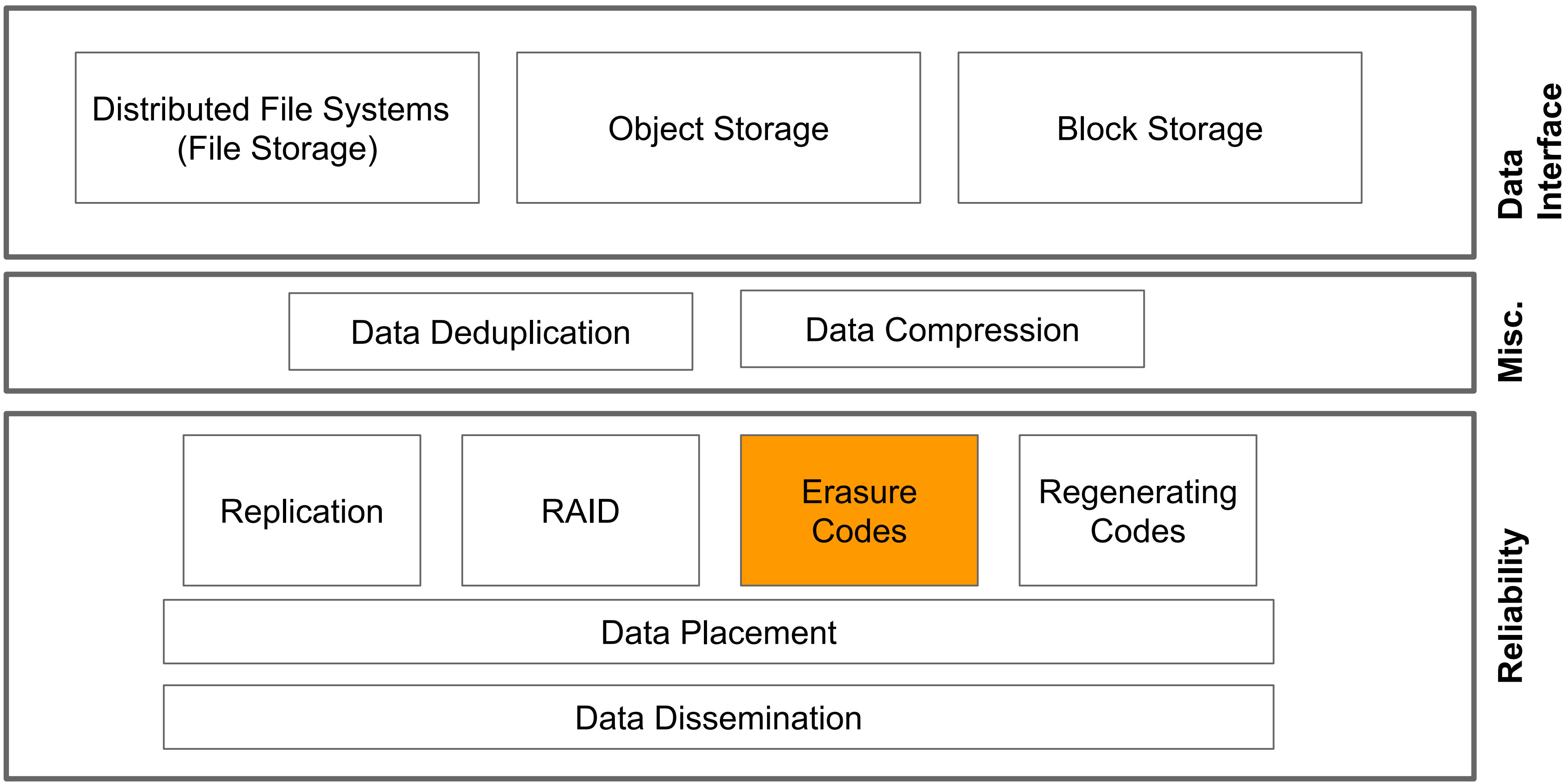
Storage Systems



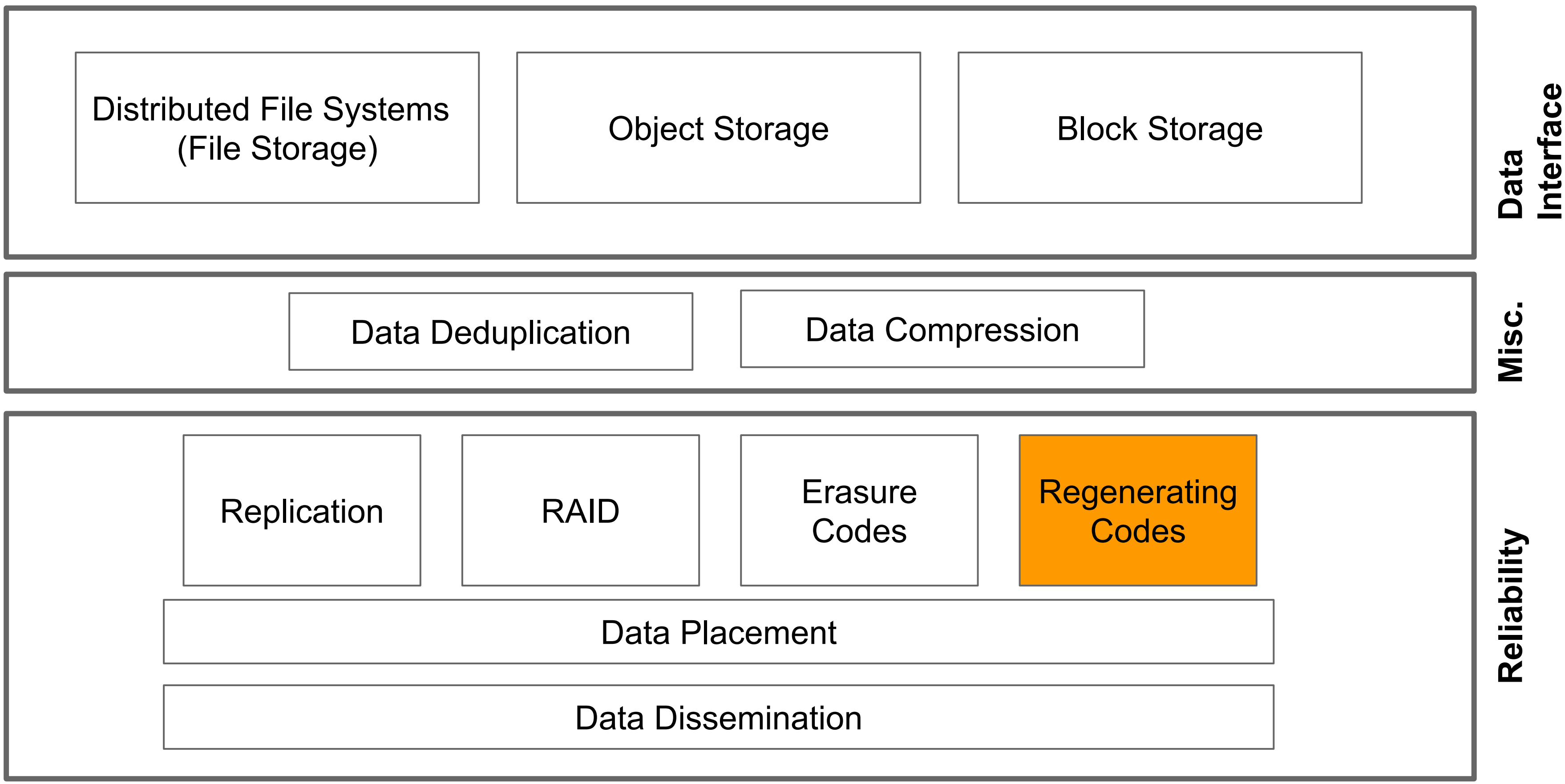
Storage Systems



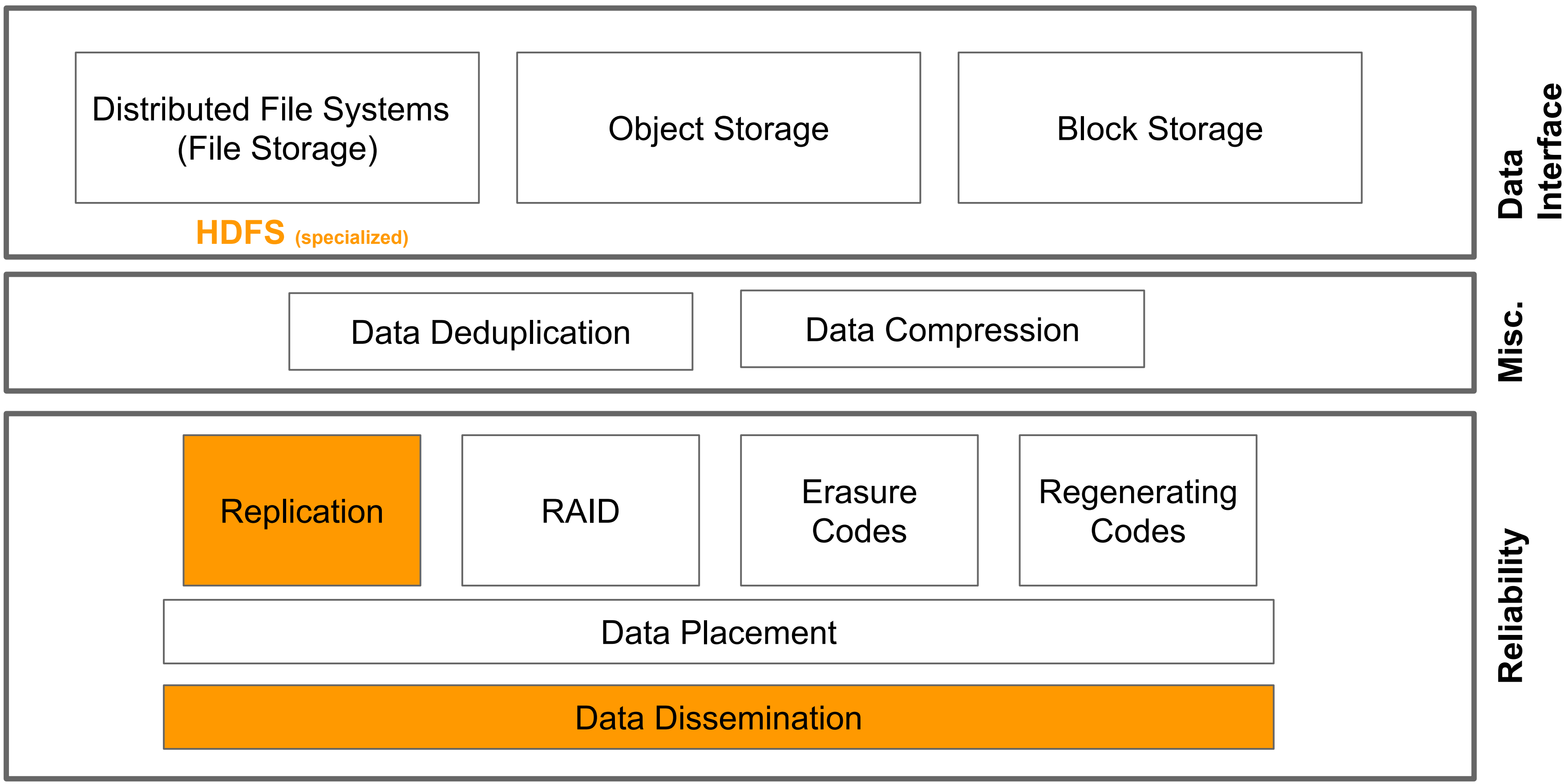
Storage Systems



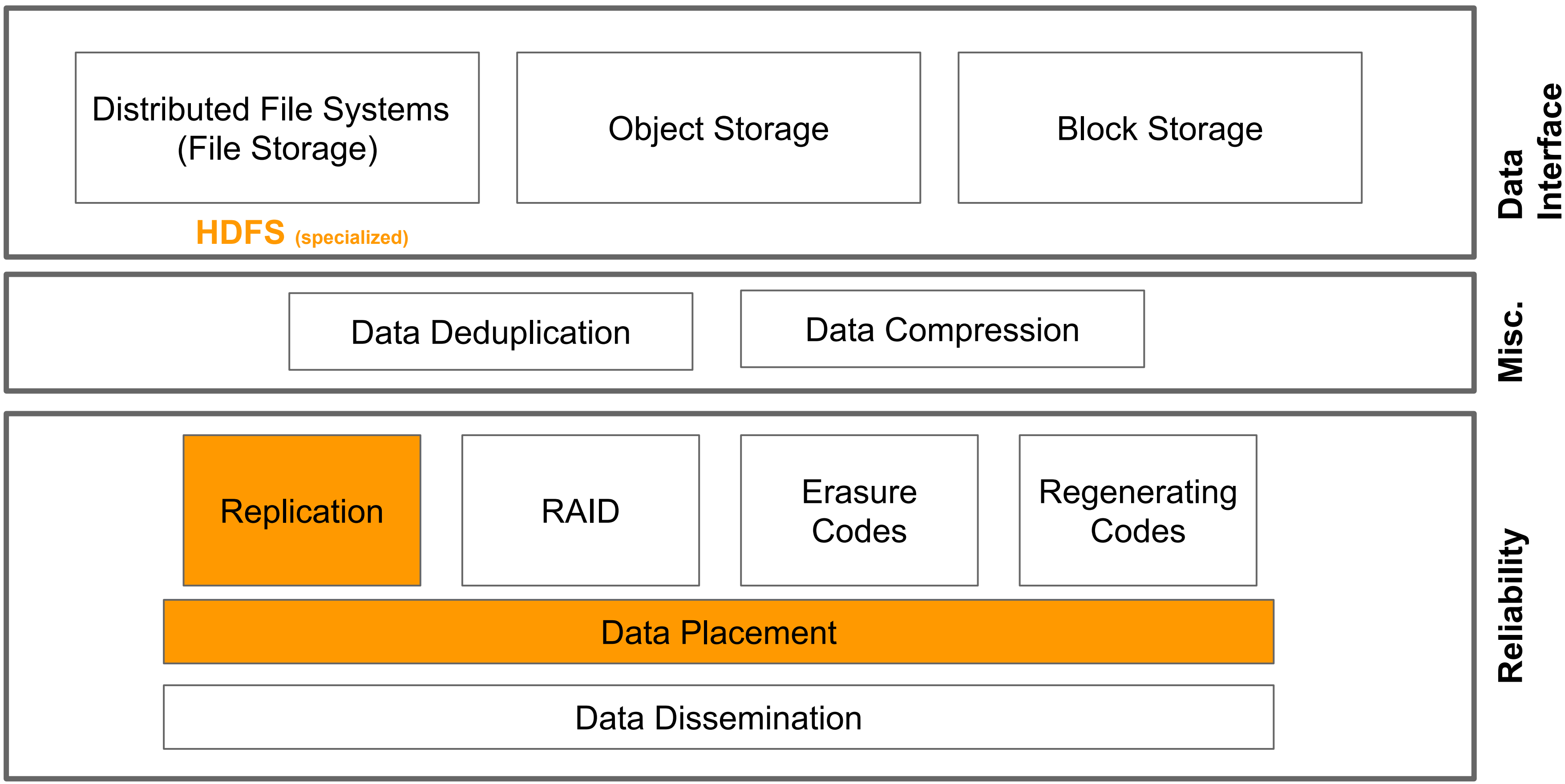
Storage Systems



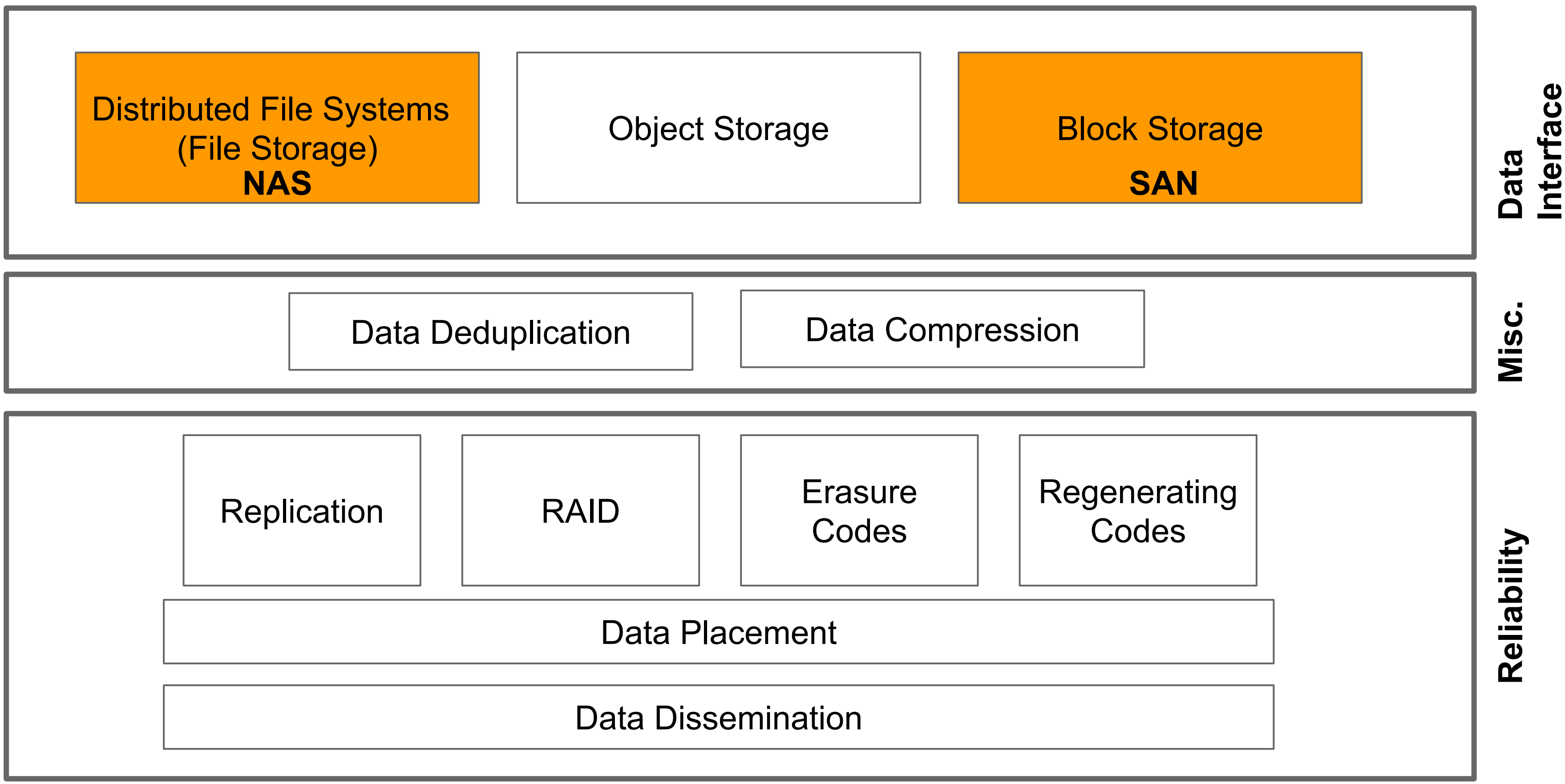
Storage Systems



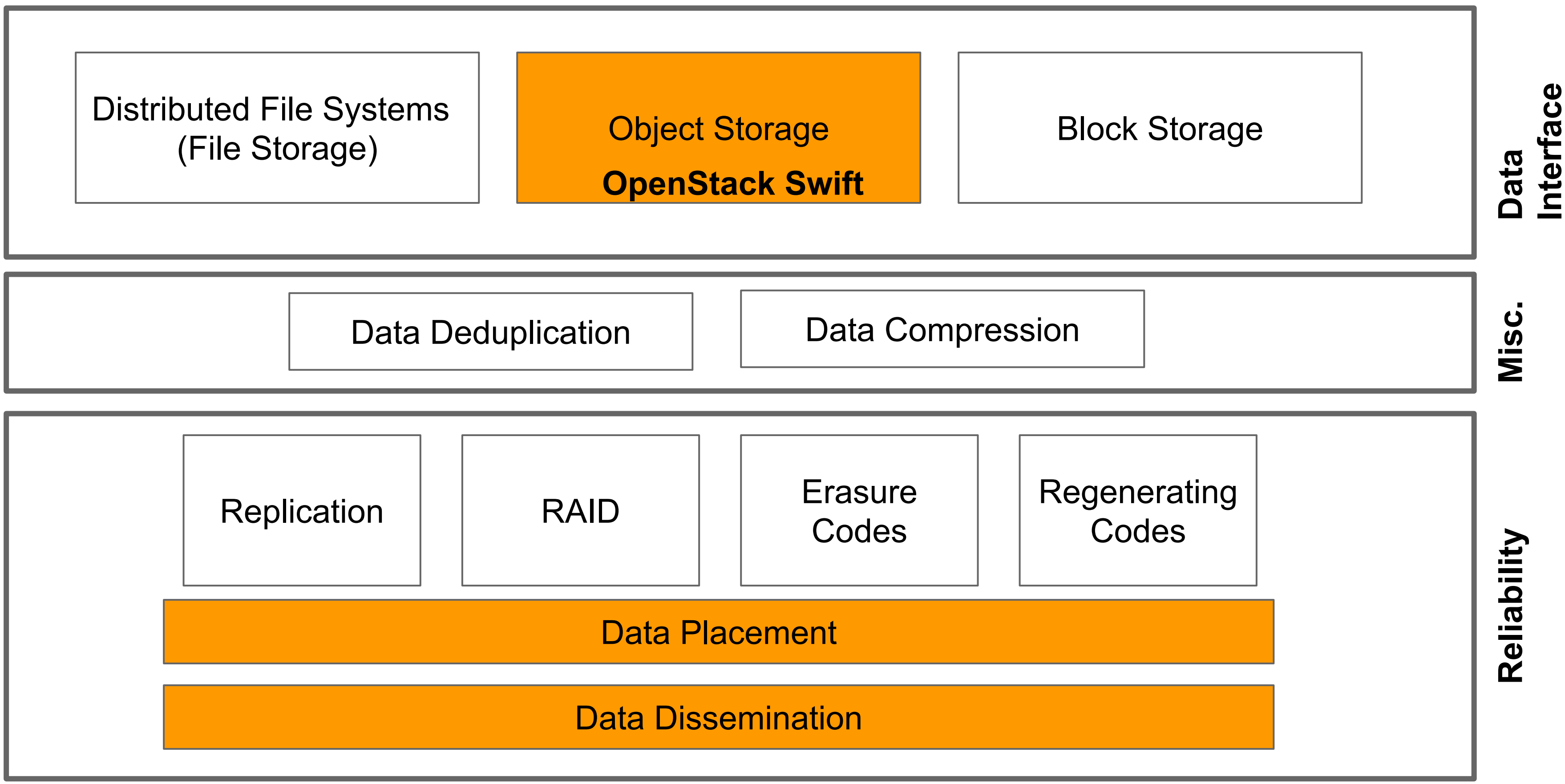
Storage Systems



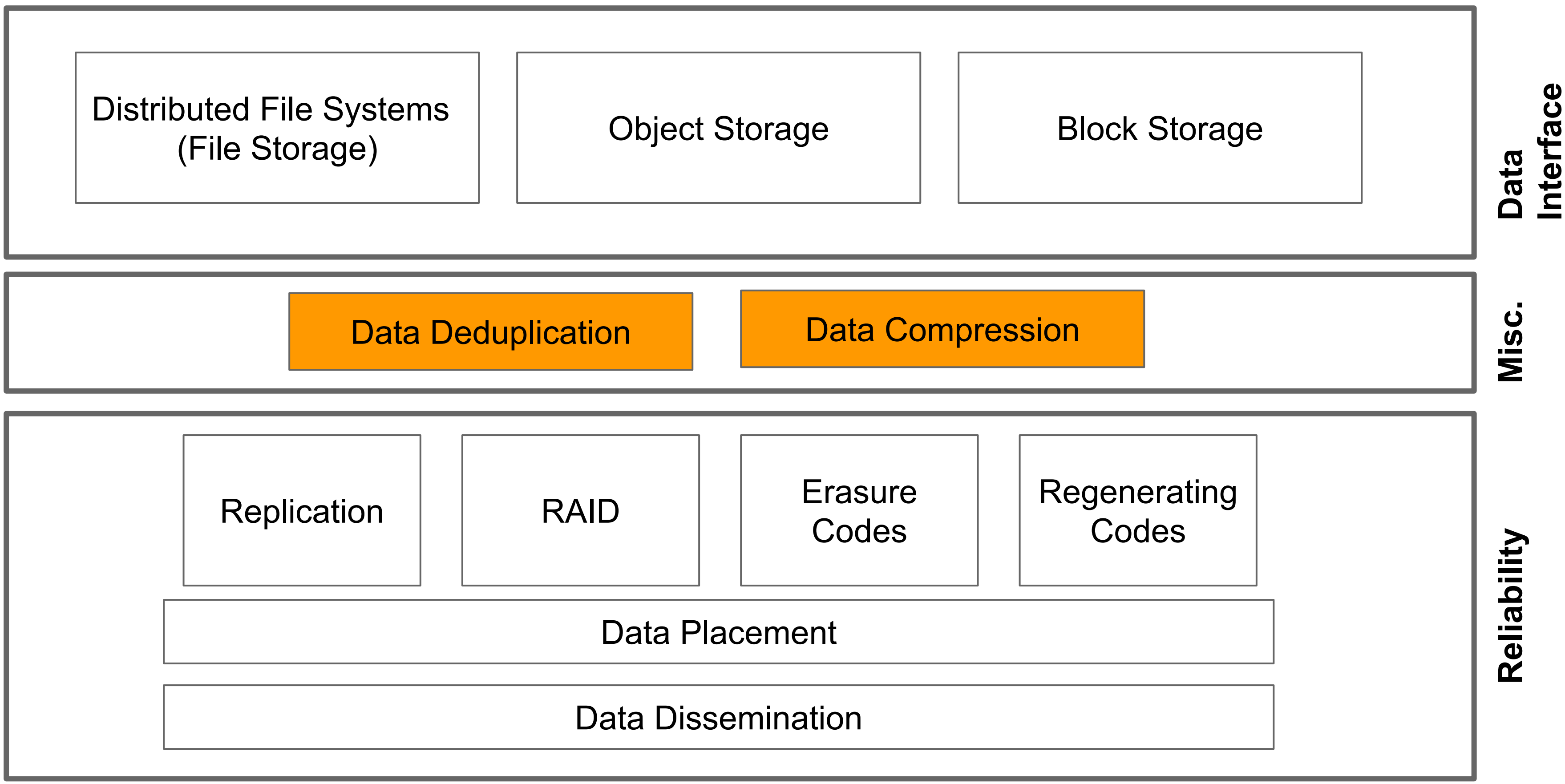
Storage Systems



Storage Systems

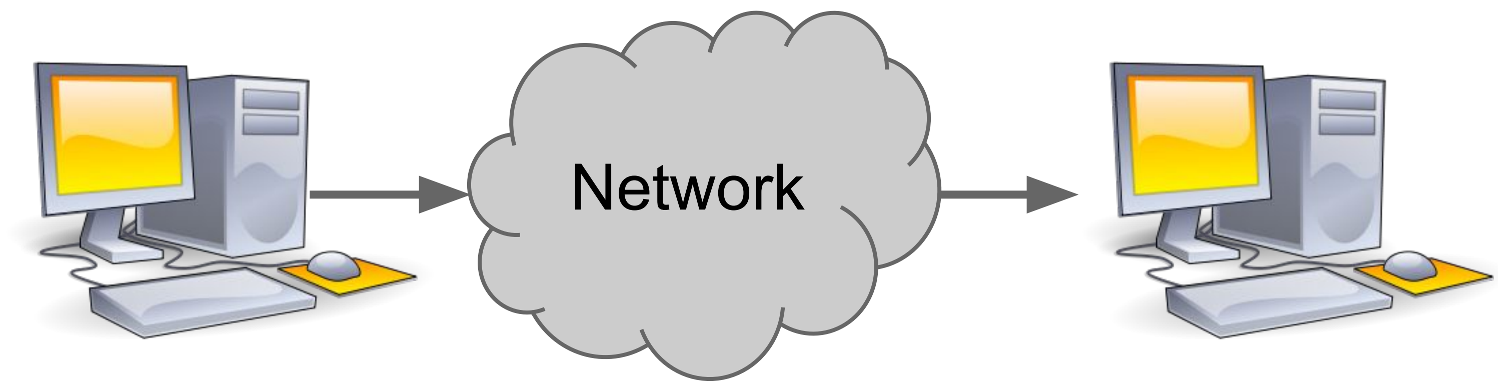


Storage Systems



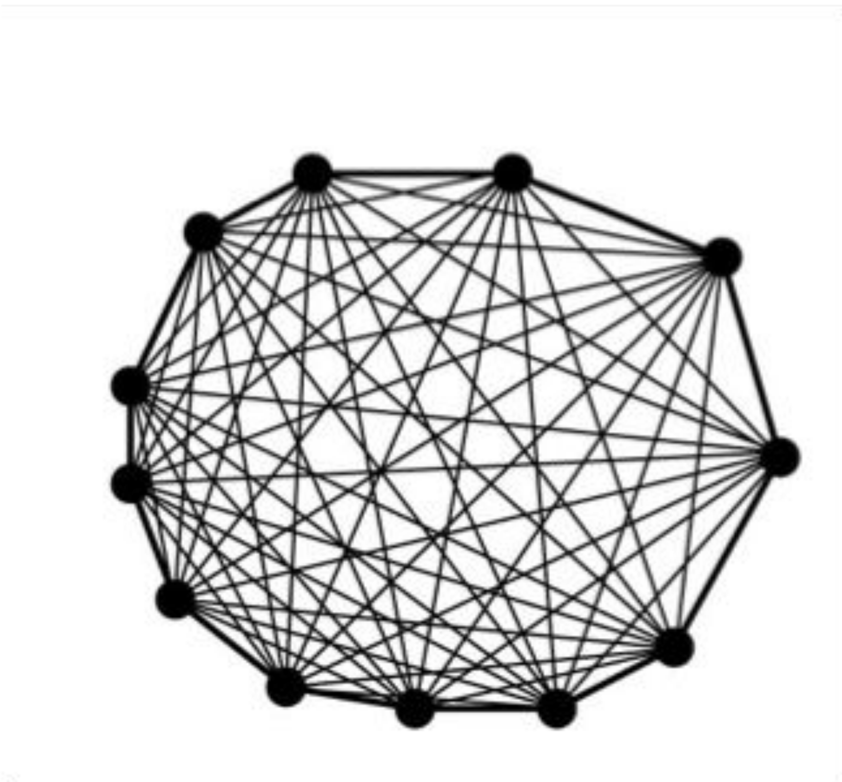
Networking:
Key to Distributed Storage

- Basic problem: Communication between users/computers

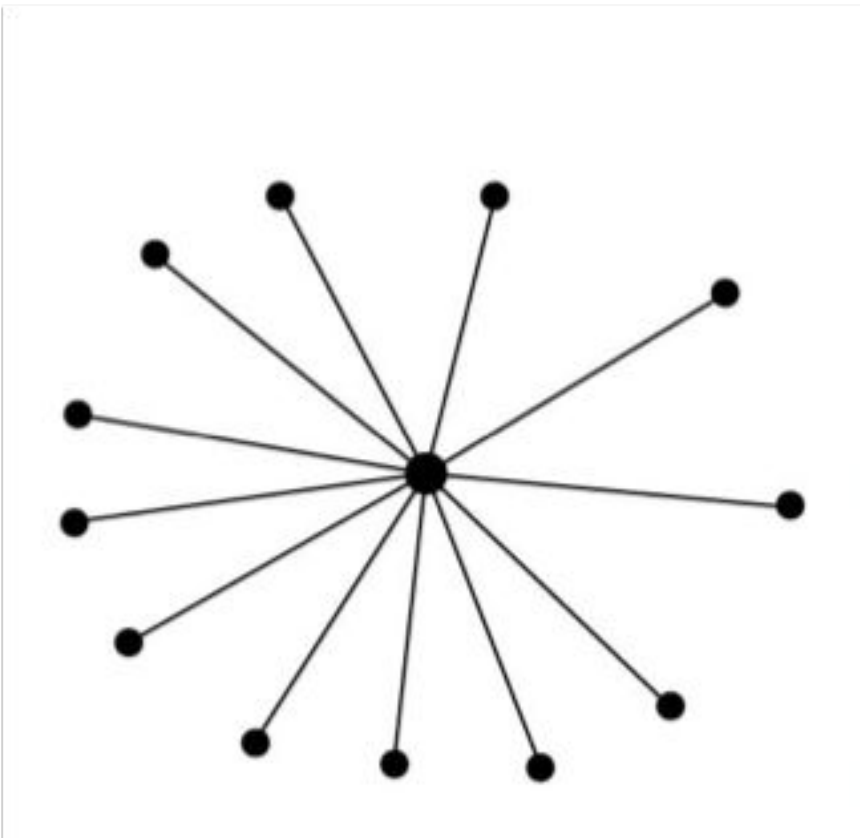


- Sending and receiving data (bits)

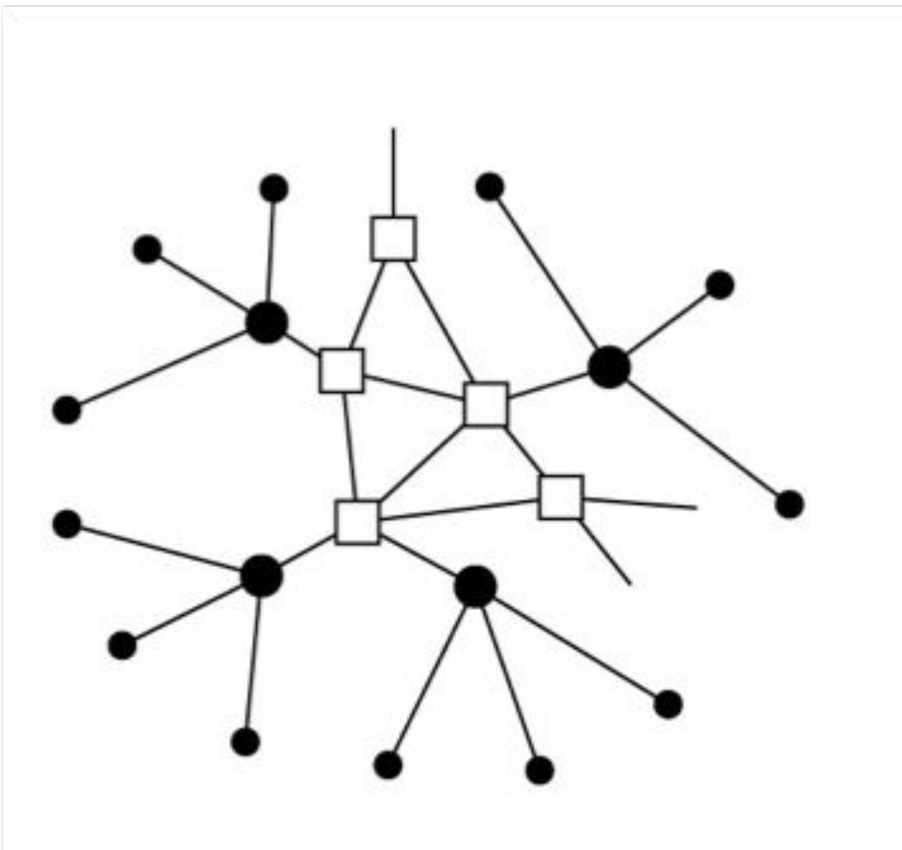
Let’s go back in time: telephone

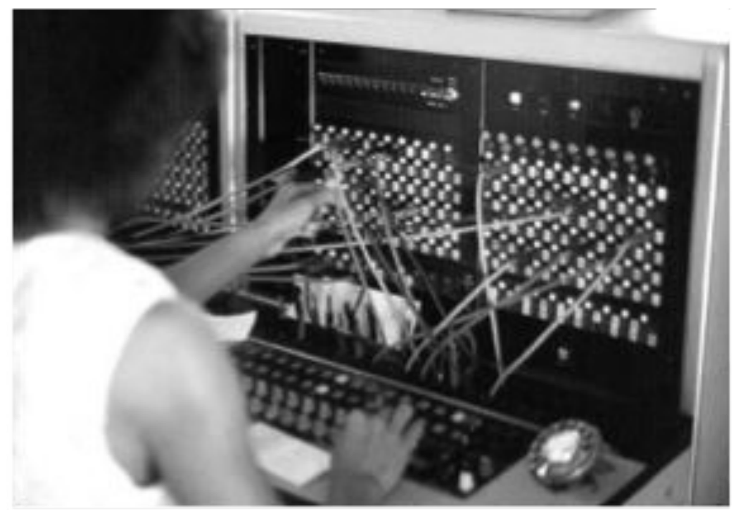


Let’s go back in time: telephone

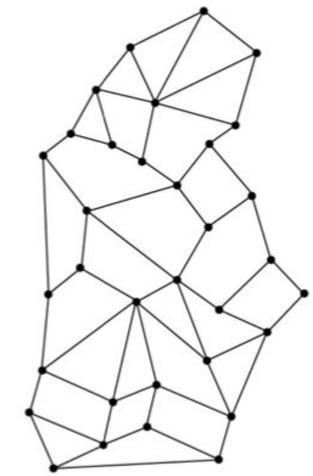


Let’s go back in time: telephone





How about the Internet?



- The internet
    - A global system of interconnected computer networks

ADVANCED RESEARCH PROJECTS AGENCY
Washington 25, D. C.

April 25, 1963

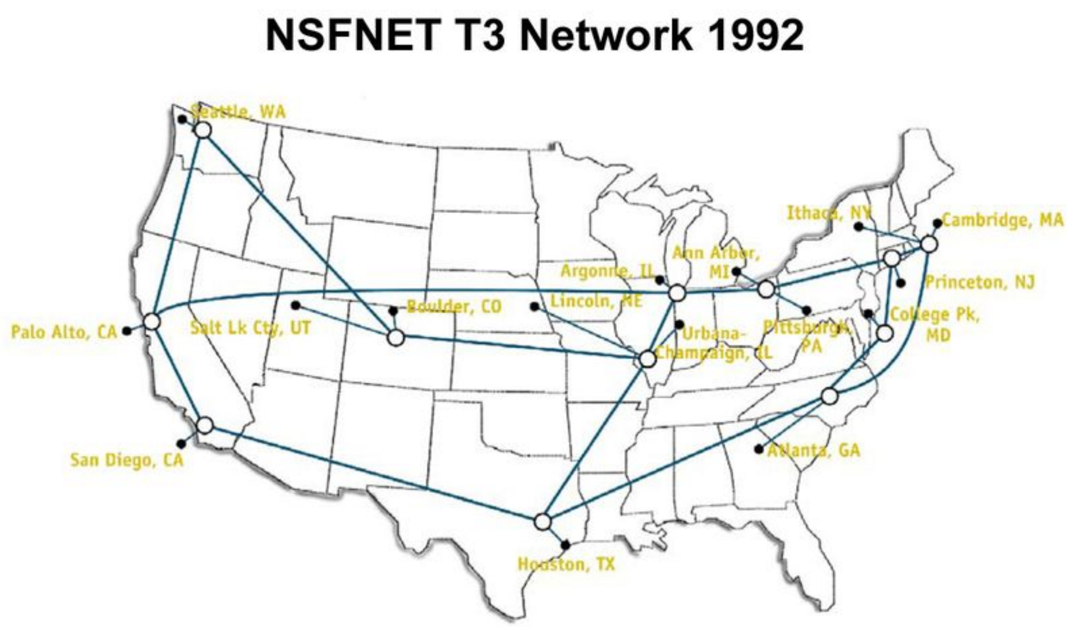
MEMORANDUM FOR: Members and Affiliates of the Intergalactic Computer Network

FROM : J. C. R. Licklider

SUBJECT : Topics for Discussion at the Forthcoming Meeting

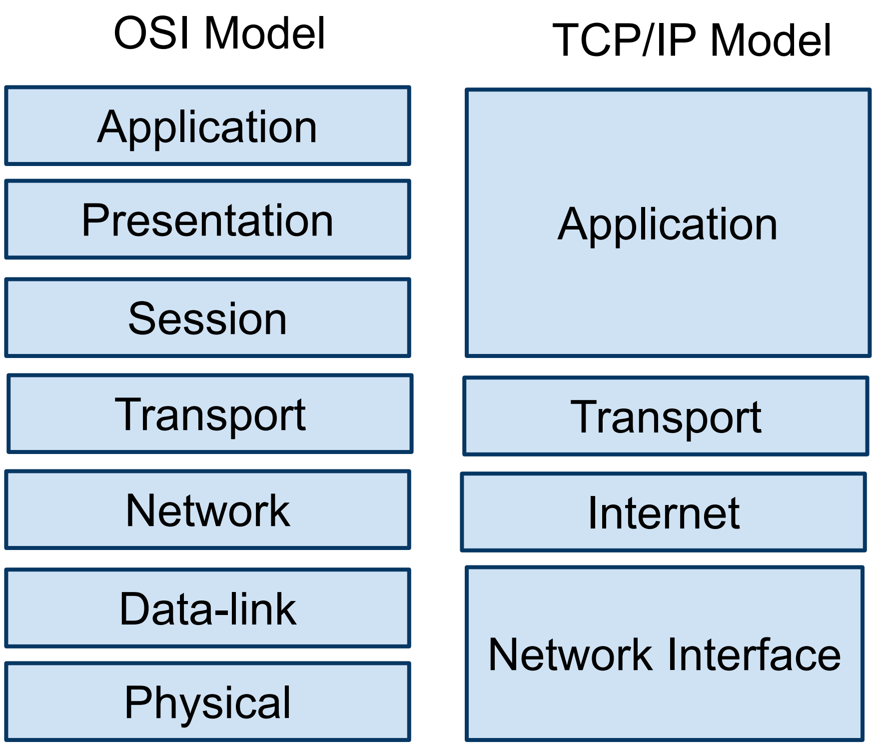
Internet Technologies

- Some significant game changers
  - Moved from circuit switched to packet switched networking
  - **Routing / addressing**
  - **Protocols**



Overview Internet protocol stack

- Fundamental models for networking
- The layers in the two models work together by encapsulating and decapsulating data
- Separation of responsibility
- End-to-end principle



IP (Internet Protocol)

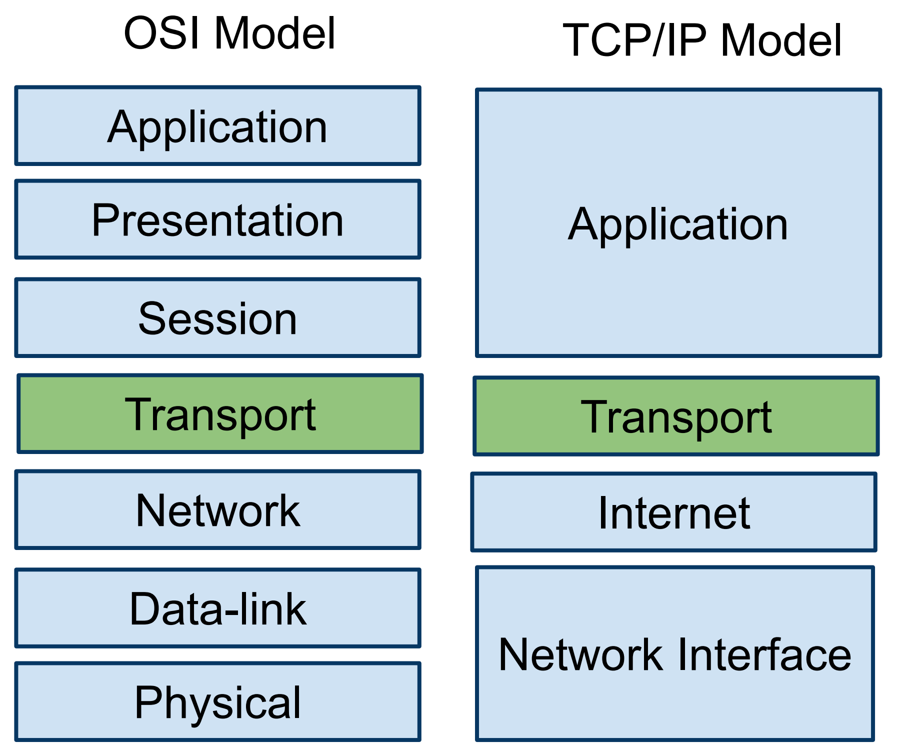
The Internet Protocol (IP) is a protocol used for communicating data across a packet-switched network using the Internet Protocol Suite, also referred to as TCP/IP

Two version exist:
- IPv4
- IPv6

Most significant change between IPv4 and IPv6 is the change from 32-bit addresses to 128-bit - i.e. from 4 billion available addresses to 340 undecillion

Overview Internet protocol stack

- Fundamental models for networking
- The layers in the two models above work together by encapsulating and decapsulating data
- Separation of responsibility
- End-to-end principle



Socket Interface

- Originated in BSD Unix
- One of the most widely-supported internet programming interfaces today
- A socket is an application-to-application channel
    - UDP and TCP protocols accepted
- A unique end-point is specified using the **(IP address, port)** tuple.

Python provides an object oriented API over the standard BSD socket interface.

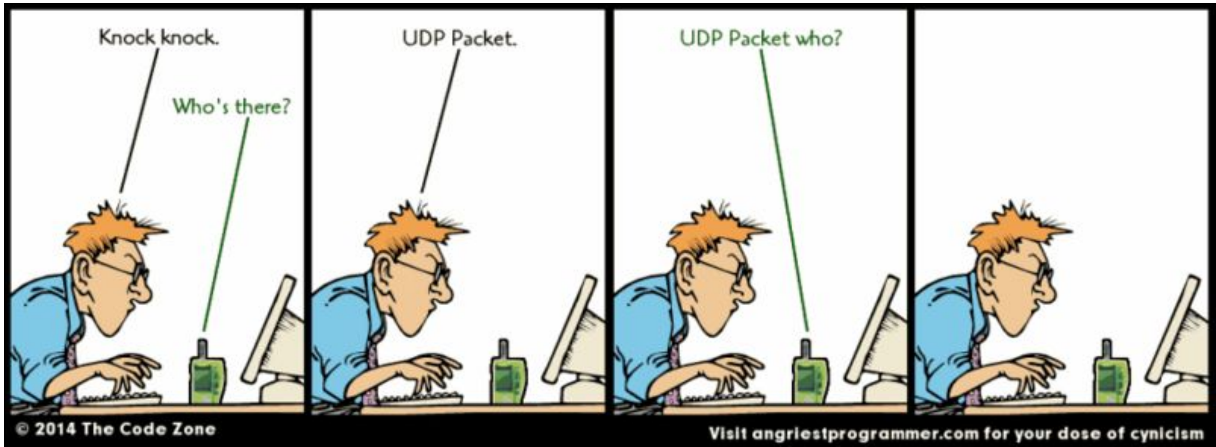
UDP (User Datagram Protocol)

UDP uses a simple transmission model **without**

- Hand-shaking
- Reliability
- Ordering

UDP's stateless nature is also useful for servers that answer small queries from huge numbers of clients

UDP is compatible with packet broadcast (sending to all on local network) and multicasting (send to all subscribers)



Datagram oriented

TCP (Transport Control Protocol)

Unreliable data delivery semantics provided by UDP are insufficient for many applications

TCP provides a connection-oriented communication:

- Reliable using retransmissions
- Flow control / Congestion control
- Ordered data transfer
- Data integrity guarantee using checksum
- Client-server oriented protocol
- Stream oriented

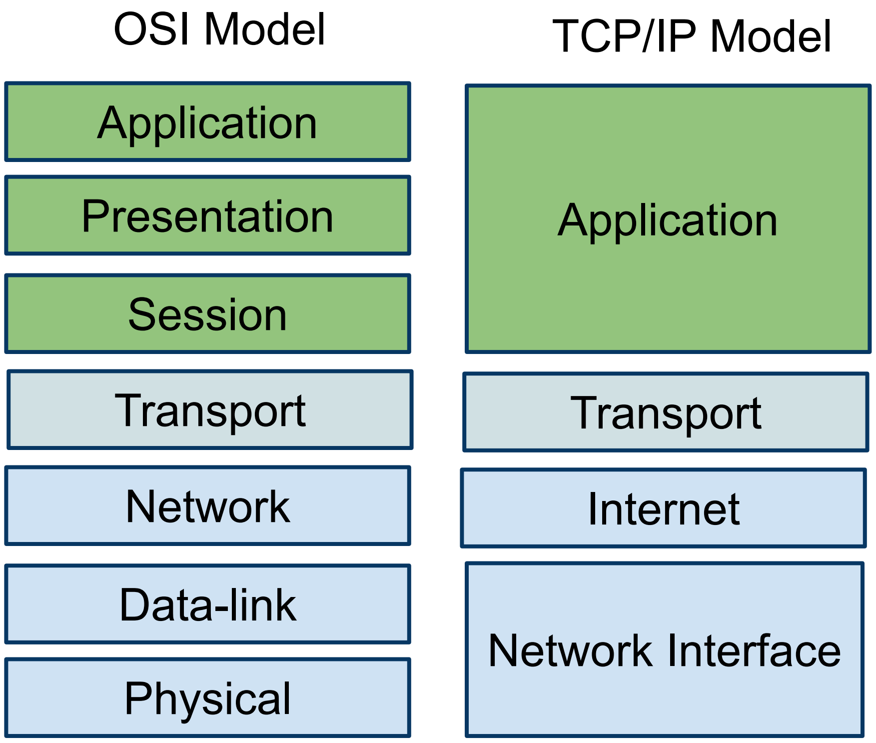
Meant for **Unicast**

"Hi, I'd like to hear a TCP joke."
"Hello, would you like to hear a TCP joke?"
"Yes, I'd like to hear a TCP joke."
"OK, I'll tell you a TCP joke."
"Ok, I will hear a TCP joke."
"Are you ready to hear a TCP joke?"
"Yes, I am ready to hear a TCP joke."
"Ok, I am about to send the TCP joke. It will last 10 seconds, it has two characters, it does not have a setting, it ends with a punchline."
"Ok, I am ready to get your TCP joke that will last 10 seconds, has two characters, does not have an explicit setting, and ends with a punchline."
"I'm sorry, your connection has timed out. ... Hello, would you like to hear a TCP joke?"

Stream oriented

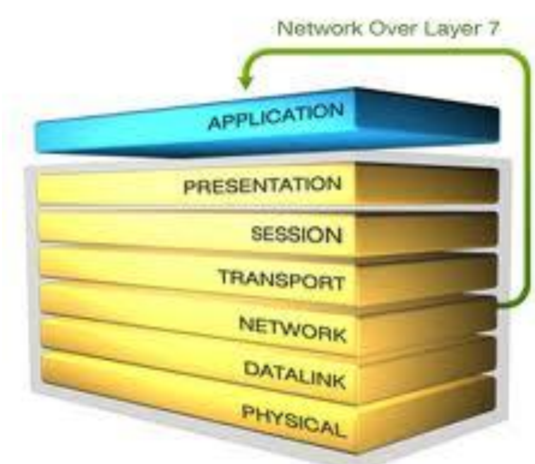
Overview Internet protocol stack

- Fundamental models for networking
- The layers in the two models above work together by encapsulating and decapsulating data
- Separation of responsibility
- End-to-end principle



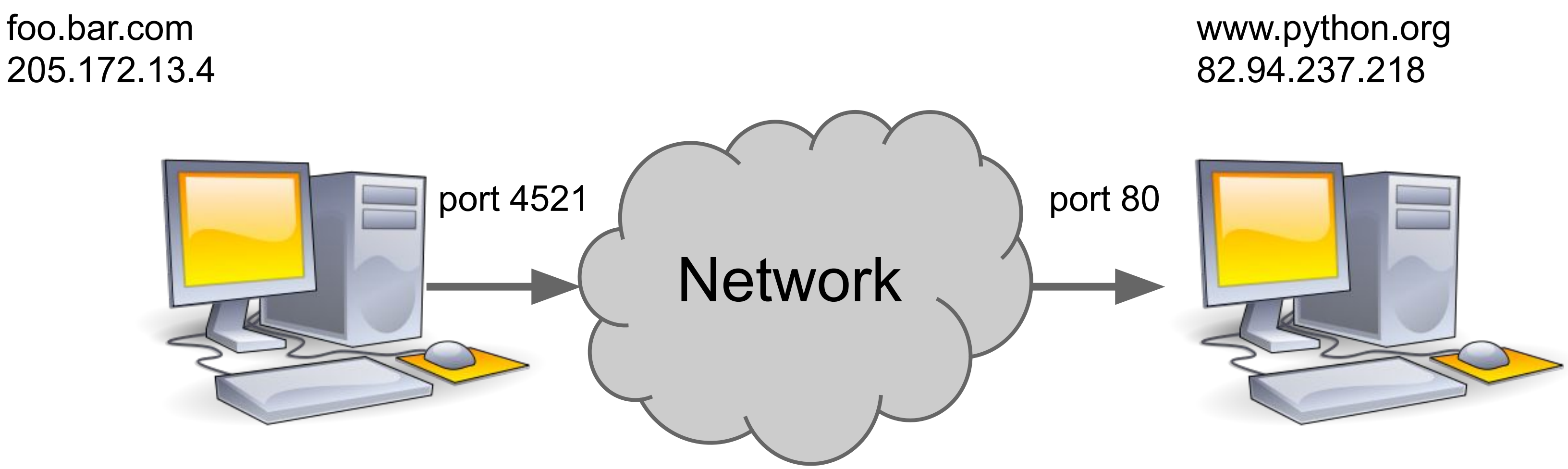
Application Layer Protocols

- HTTP (Hyper Text Transport Protocol)
  - Used when browsing the web to request web pages
- There are many other protocols built on top of UDP/TCP/IP, e.g. VoIP, RTP, BitTorrent, SMTP, POP, IMAP, SSH, TELNET, TFTP, etc
- Our own protocols are defined here



Network Addressing

- Machines have a hostname and IP address
- Programs/services have port numbers



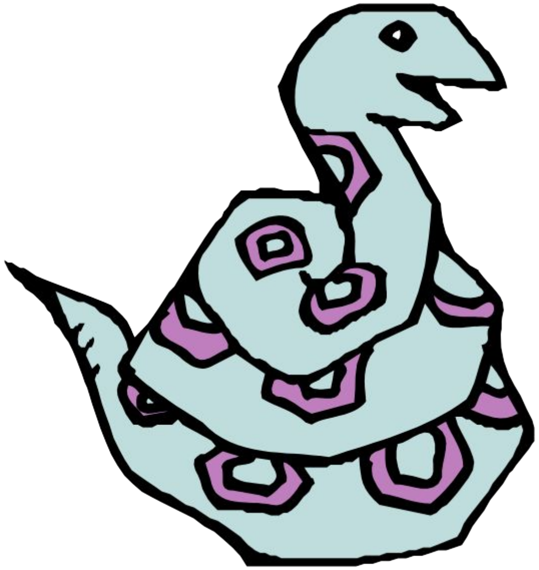
Standard Ports

- Ports for common services are preassigned

```
21          FTP
22          SSH
23          Telnet
25          SMTP (Mail)
80          HTTP (Web)
110         POP3 (Mail)
119         NNTP (News)
443         HTTPS (Web)
```

- Other port numbers may just be randomly assigned to programs by the operating system

Writing Networking applications



Connections

- Each endpoint of a network connection is always represented by a `host` and `port #`
- In Python you write it out as a tuple

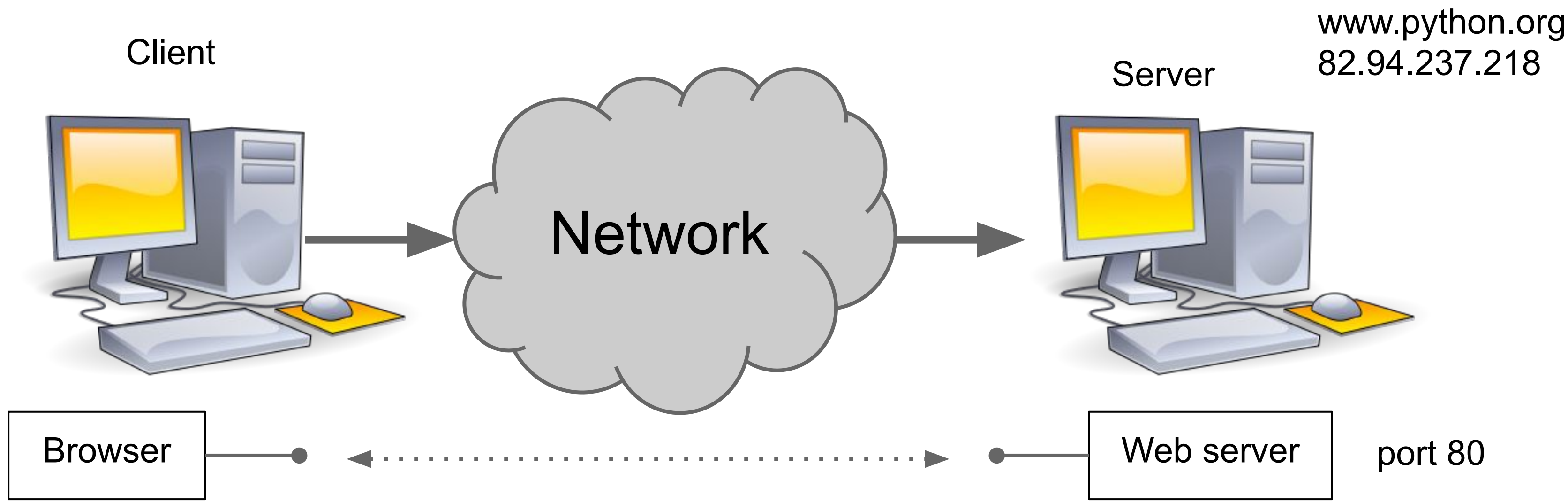
```
(host,port)


("www.python.org" 80)
("205.172.13.4", 443)
```

- In almost all of the network programs you'll write, you use this convention to specify the network address

Client/Server Concept

- Each endpoint is a running program
- Servers wait for incoming connections and provide a service (e.g., web, mail)
- Clients make connections to servers



Request/Response Cycle

- Most network programs use a request/response model based on messages
- Client sends a request message (e.g. HTTP)

```
GET /index.html HTTP/1.1
```

- Server sends back a response message

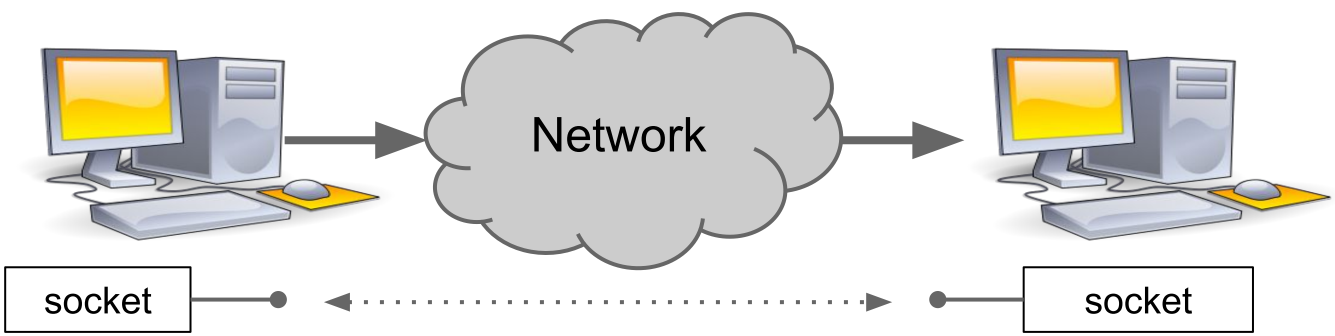
```
HTTP/1.1 200 OK
Content-type: text/html
Content-length: 48823

<HTML>
...
```

- The exact format depends on the application

Sockets

- Programming abstraction for network code
- Socket: A communication endpoint



- Supported by socket library module
- Allows connections to be made and data to be transmitted in either direction

Socket Basics

- To create a socket

```
import socket
s = socket.socket(addr_family, type)
```

- Address families

```
socket.AF_INET        Internet protocol (IPv4)
socket.AF_INET6       Internet protocol (IPv6)
```

- Socket types

```
socket.SOCK_STREAM    Connection based stream (TCP)
socket.SOCK_DGRAM     Datagram (UDP)
```

- Example

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
```

Socket Types

- Almost all code will use one of following

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s = socket(AF_INET, SOCK_DGRAM)
```

- Most common case: TCP connection

```
s = socket(AF_INET, SOCK_STREAM)
```

Using a socket

- Creating a socket is only the first step

```
s = socket(AF_INET, SOCK_STREAM)
```

- Further use depends on the application
- Server
  - Listen for incoming connections
- Client
  - Make an outgoing connection

TCP Client

- How to make an outgoing connection

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.connect(('www.google.com', 80))
s.send(bytes('GET / HTTP/1.1\r\n\r\n', 'utf-8'))
data = s.recv(10000)
print(data)
s.close()
```

- `s.connect(addr)` makes a connection

```
s.connect((‘www.google.com’, 80))
```

- Once connected, use `send()`, `recv()` to transmit and receive data
- `close()` shuts down the connection

Server Implementation

- Network servers are a bit more tricky
- Must listen for incoming connections on a well-known port
- Typically run forever in a server-loop
- May have to service multiple clients

TCP Server

- A simple server

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

- Send a message back to a client

```
$ telnet localhost 9000
Connected to localhost.
Escape character is '^]'.
Hello 127.0.0.1
Connection closed by foreign host.
```

Server message

TCP Server

- Address binding

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

- Addressing

```
s.bind(("",9000))
s.bind(("localhost",9000))
s.bind(("192.168.2.1",9000))
s.bind(("104.21.4.2",9000))
```

If the system has multiple IP addresses we can bind to a specific address

TCP Server

- Start listening for connections

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

Tells the operating system to start listening for connections on the socket

- `s.listen(`*`backlog`*`)`
- backlog is # of pending connections to allow
- Note: not related to maximum number of clients

TCP Server

- Accepting a new connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

Accepts a new client connection

- `s.accept()` blocks until a connection is received
- Server sleeps if nothing is happening

TCP Server

- Accepting a new connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

Accept returns a pair `(client_socket, addr)`

<socket.socket object, fd=5, family=2, type=1, proto=0>

('127.0.0.1', 47027)

TCP Server

- Accepting a new connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

Send data to the client

Note: The returned socket is for transmitting data. The server socket is only for accepting new connections.

TCP Server

- Closing the connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

- Note: Server can keep client connection alive as long as it wants
- Can repeatedly receive/send data

TCP Server

- Closing the connection

```
from socket import *
s = socket(AF_INET,SOCK_STREAM)
s.bind(("",9000))
s.listen(5)
while True:
    c,a = s.accept()
    print('Received connection from {}'.format(a))
    c.send(bytes('Hello {}\n'.format(a[0]), 'utf-8'))
    c.close()
```

Wait for next connection

- Original server socket it re-used to listen for more connections
- Server runs forever in a loop like this

Exercises

- Socket Server and Client: text message
  1. Both on same machine: use localhost
  2. Client sends message, e.g., name
  3. Server receives message, replies: “I have received: ” and ends connection
  4. Server reports IP and message in the terminal
  5. Test with colleagues: Use your client to send data to their server
- Socket Server and Client: File transmission
  - Instead of a message, transmit contents of a file and save as file in the server