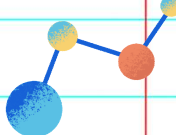



**CSIE52400/CSIEM0140**  
**Distributed Systems**

**Lecture 03**  
**Architectures and Models**

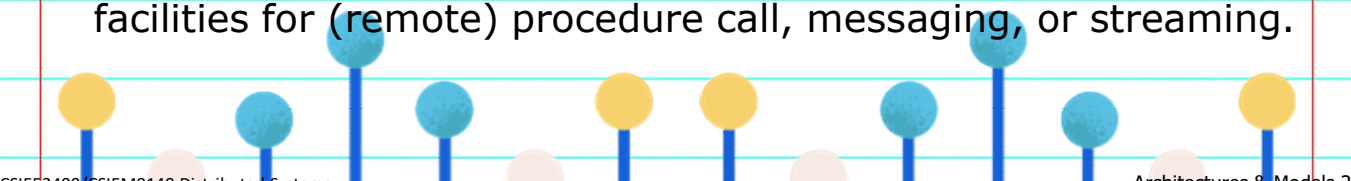
**Shiow-yang Wu (吳秀陽)**  
Department of Computer Science and Information Engineering  
National Dong Hwa University

CSIE52400/CSIEM0140 Distributed Systems 1

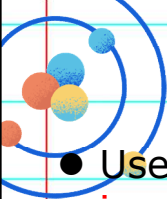


## Architectural Styles

- An **architectural style** is formulated by:
  - (replaceable) **components** with well-defined **interfaces**
  - the **way** that components are **connected** to each other
  - the **data exchanged** between components
  - how these components and connectors are jointly **configured** into a **system**.
- **Connector**: A mechanism that mediates communication, coordination, or cooperation among components. Example: facilities for (remote) procedure call, messaging, or streaming.

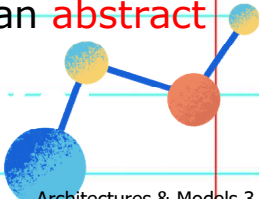


CSIE52400/CSIEM0140 Distributed Systems Architectures & Models 2




## System Models

- Use **models** to capture and discuss the **properties** and **design issues** of distributed systems.
- Types of models
  - **Physical models** – Describe a system by its **hardware composition** of **computers** and **networks**.
  - **Architectural models** – Describe a system by its **computational** and **communication tasks** performed by its **computational elements**.
  - **Fundamental models** – Describe a system from an **abstract perspective** to examine **individual aspects**.

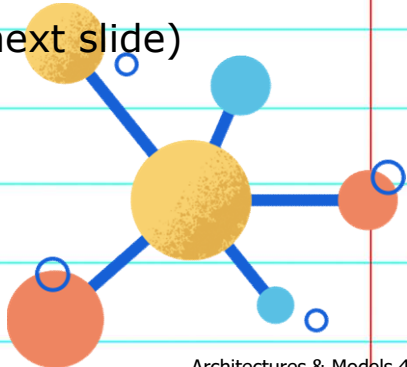


CSIE52400/CSIEM0140 Distributed Systems Architectures & Models 3



## Physical Models

- An **abstract representation** of the **hardware elements** of a distributed system.
- **Baseline physical model**: a set of computer **nodes** interconnected by a **network** and coordinated by passing **messages**.
- **Three generations** of distributed systems: (next slide)
  - Early
  - Internet-scale
  - Contemporary(當代的)



CSIE52400/CSIEM0140 Distributed Systems Architectures & Models 4

# Generations of Distributed Systems

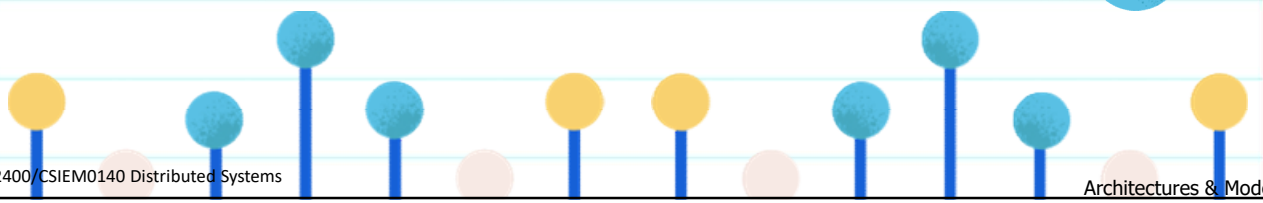
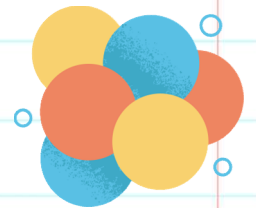
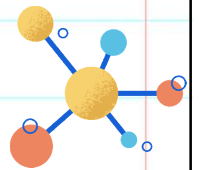
<i>Distributed systems:</i>	<i>Early</i>	<i>Internet-scale</i>	<i>Contemporary</i>
<i>Scale</i>	Small	Large	Ultra-large
<i>Heterogeneity</i>	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
<i>Openness</i>	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
<i>Quality of service</i>	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

# Architectural Models

- **System architecture** is the fundamental **organization** of a system, embodied in its **components**, their **relationships** to each other and the **environment**, and the **principles** governing its design and evolution.
- Describe architectural models from **three perspectives**
  - Architectural **elements**
  - Architectural **patterns**
  - **Middlewares**

# Architectural Elements

- Communicating **entities** – Basic elements
- **Communication paradigm** – How do elements comm.
- **Roles** and **responsibilities** of elements.
- **Placement** – How are they mapped on to the physical infrastructure?



# 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  
Processes

Objects  
Components  
Web services  
Agents

Message passing  
Sockets  
Multicast

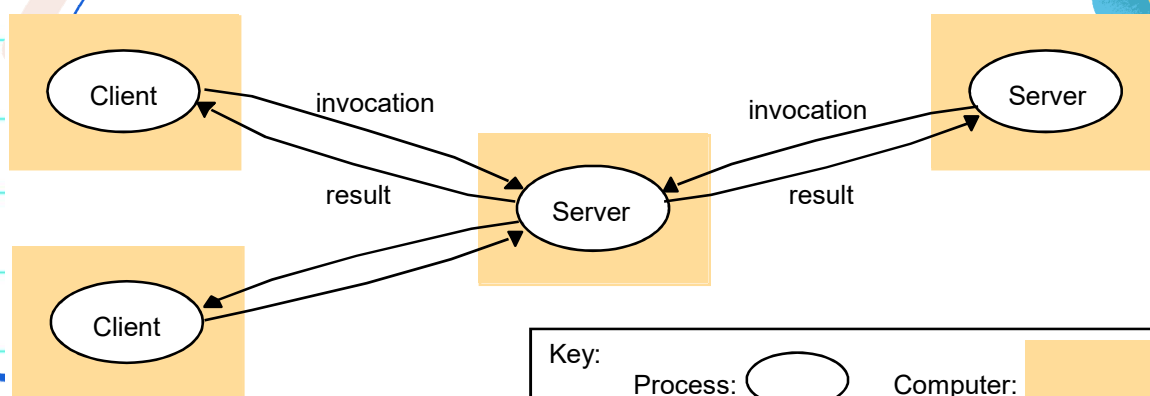
Request-reply  
RPC  
RMI

Group communication  
Publish-subscribe  
Message queues  
Tuple spaces  
DSM

## Roles and Responsibilities

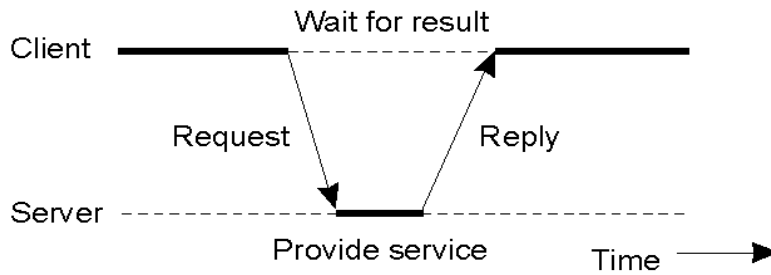
- Entities can take on different **roles** in a distributed system.
- These roles are fundamental in establishing the overall architecture.
- Two examples of architectural styles with distinctive roles:
  - Client-server
  - Peer-to-peer (P2P)

## Client-Server Architecture

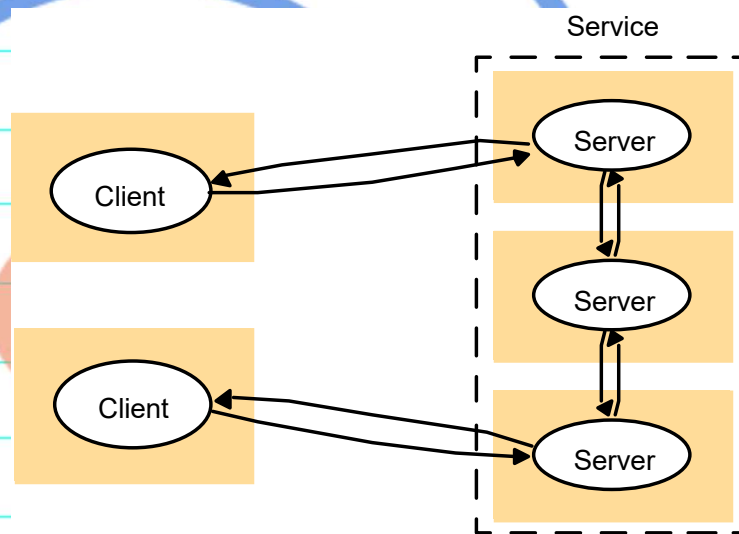


# Client-Server Interaction

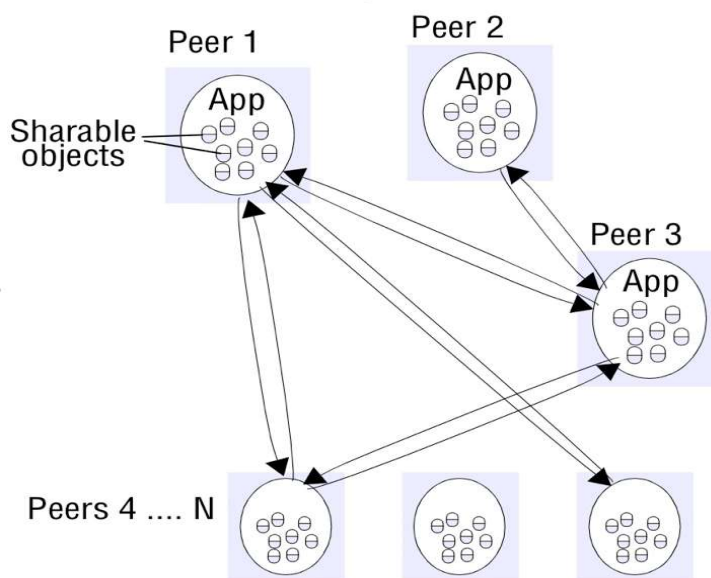
- General interaction (request-reply behavior) between a client and a server.



# Multiple Servers Architecture

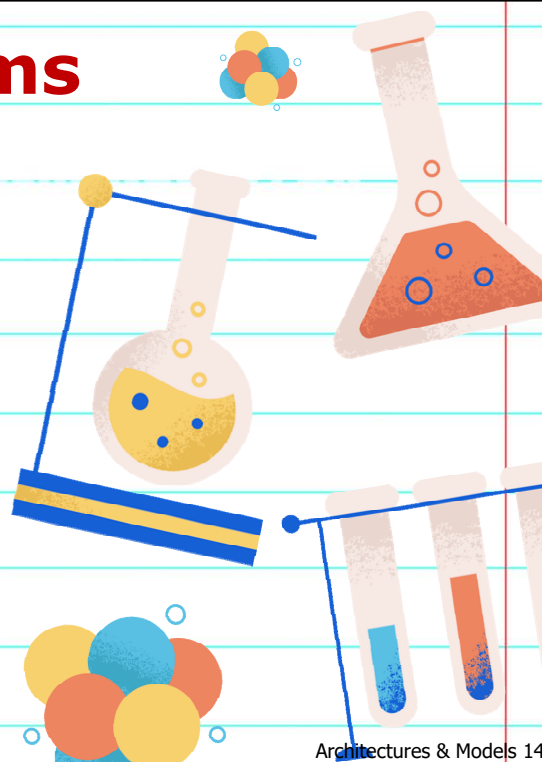


## Peer-to-peer Architecture



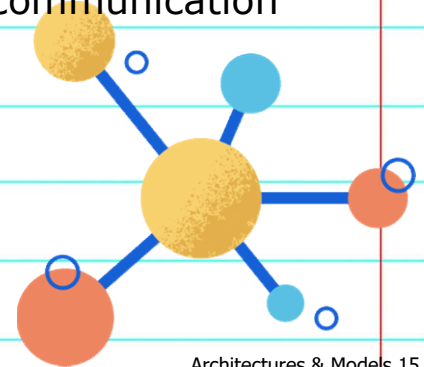
## Peer-to-Peer Systems

- Each component is **symmetric** in functionality
  - **Servent**: Combination of server-client
- How does a node find the other?
  - No "well-known" centralized server



## Overlay Network

- A **logical network** consisting of participant components (processes/machines)
  - Built on top of physical network
- Can be thought of as a **graph**
  - **Nodes** are processes/machines, **links** are communication channels (e.g., TCP connections)



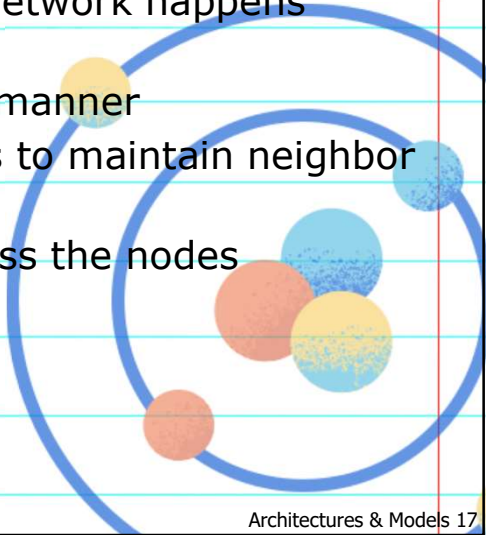
## Types of P2P Systems

- **Unstructured**: Built in a **random** manner
  - Each node can end up with any sets of neighbors, any part of application data
  - E.g.: Gnutella, Kazaa
- **Structured**: Built in a **deterministic** manner
  - Each node has well-defined set of neighbors, handles specific part of application data
  - E.g.: CAN, Chord, Pastry



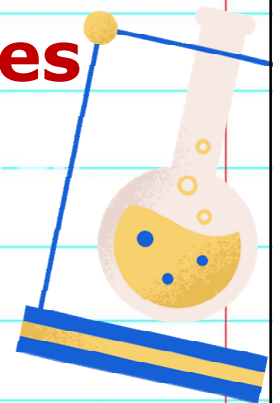
## Unstructured P2P Architectures

- Each node has a **list of neighbors** to which it is connected
  - Communication to other nodes in the network happens through neighbors
  - Neighbors are discovered in a random manner
  - Exchange information with other nodes to maintain neighbor lists
- Application data is **randomly spread** across the nodes
  - **Flooding**: To search for a specific item

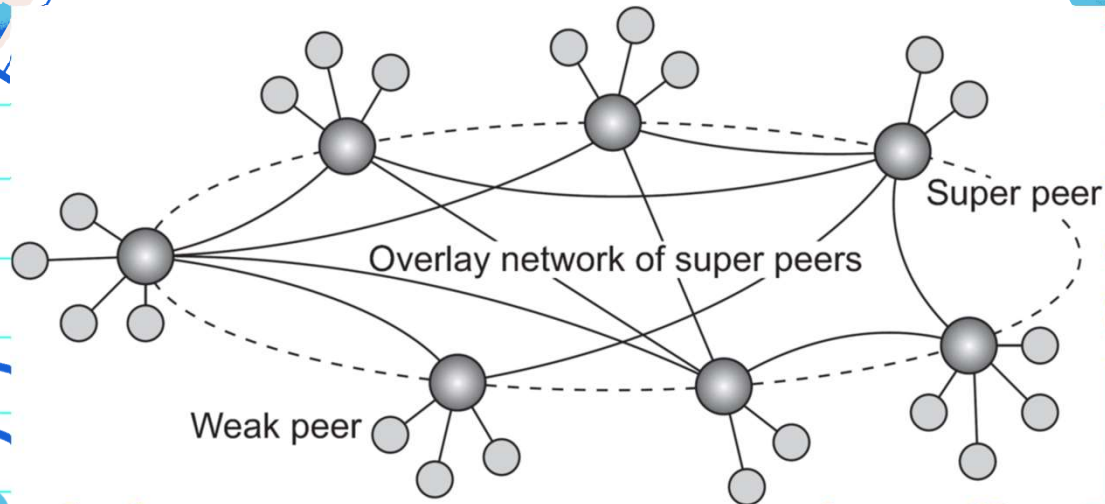


## Structured P2P Architectures

- Nodes and data are organized **deterministically**
- **Distributed Hash Tables (DHT)**
  - Each node has a well-defined **ID**
  - Each data item also has a **key**
  - A data item resides in the node with nearest key
- Each node has information about neighbors in the ID space
- **Searching** for a data item:
  - **Routing** through the **DHT overlay network**

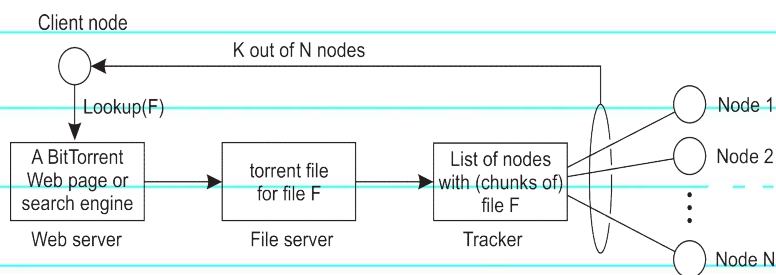


## Super-peer Networks



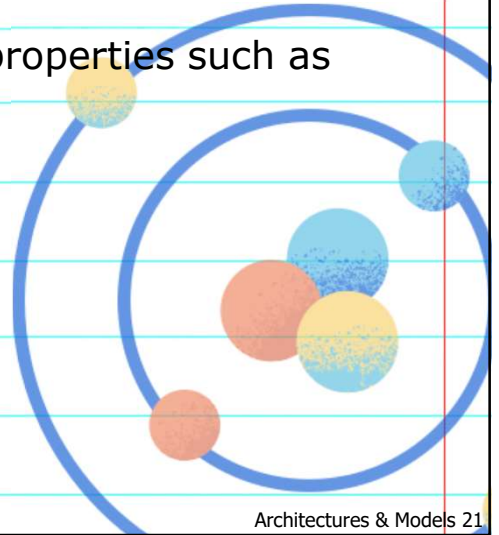
## Collaboration: BitTorrent

- Principle: search for a file  $F$ 
  - Lookup file at a global directory  $\Rightarrow$  returns a **torrent file**
  - Torrent file contains reference to **tracker**: a server keeping an accurate account of **active** nodes that have (chunks of)  $F$
  - $P$  can join **swarm**, get a chunk for free, and then trade a copy of that chunk for another one with a peer  $Q$  also in the swarm.

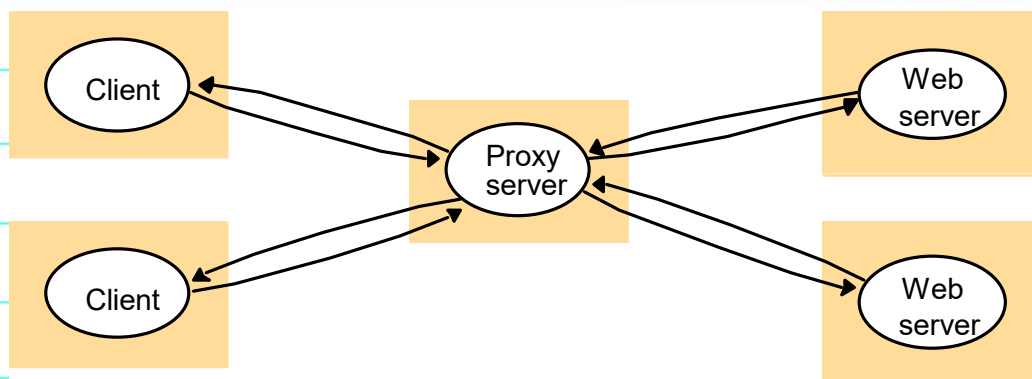


# Placement

- The **mapping** of entities on to physical infrastructure.
- **Where** to place different entities?
- Placement is crucial in determining the properties such as
  - Performance
  - Reliability
  - Security



# Proxy Servers and Cache Architecture



# Web Applets

a) client request results in the downloading of applet code

```

    graph LR
      subgraph ClientBox [Client]
        C((Client))
      end
      subgraph WebServerBox [Web server]
        WS((Web server))
      end
      WS -- Applet code --> C
  
```

b) client interacts with the applet

```

    graph LR
      subgraph ClientAppletBox [Client]
        C((Client))
        A((Applet))
        C <--> A
      end
      subgraph WebServerBox [Web server]
        WS((Web server))
      end
  
```

CSIE52400/CSIEM0140 Distributed Systems Architectures & Models 23

# Mobile Code/Agent

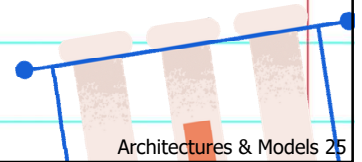
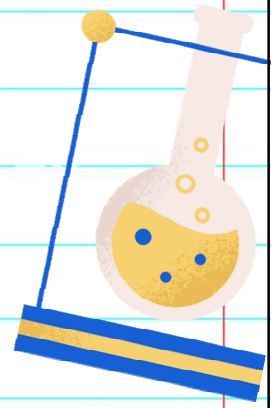
```

    graph TD
      S1([Server]) --> S2([Server])
      S2 --> S3([Server])
      S3 --> S1
  
```

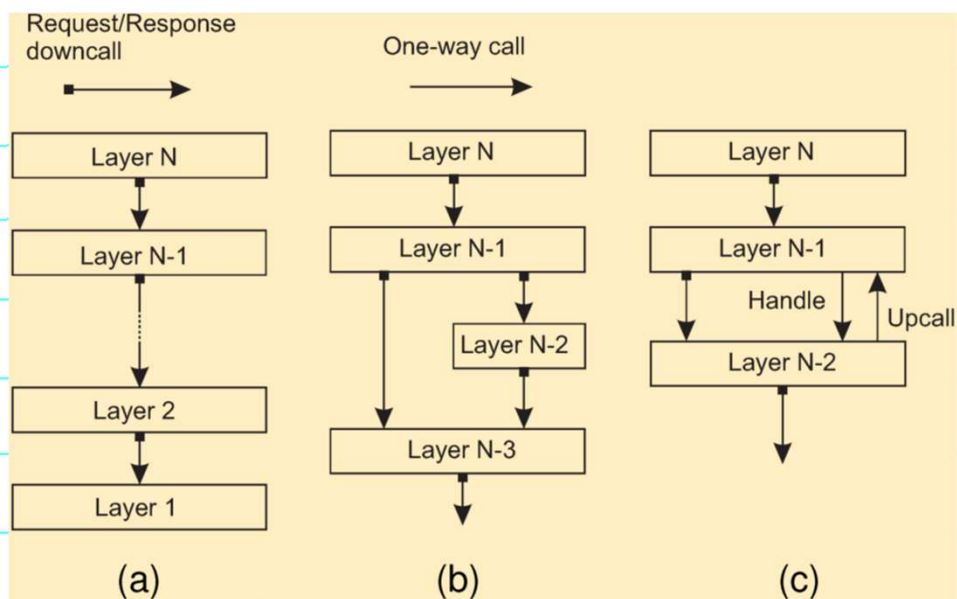
CSIE52400/CSIEM0140 Distributed Systems Architectures & Models 24

# Architectural Patterns

- **Composite structures** on top of architectural elements
- Some **patterns** have been shown to work well in given circumstances
- Not necessarily complete solutions but offer **partial insights**.



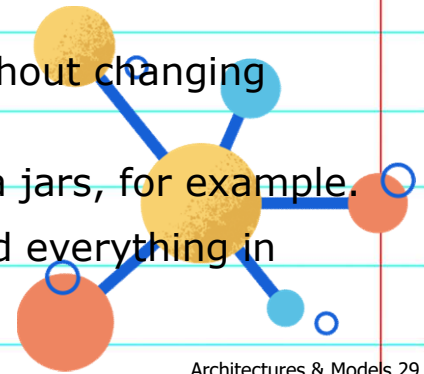
# Layered Architecture



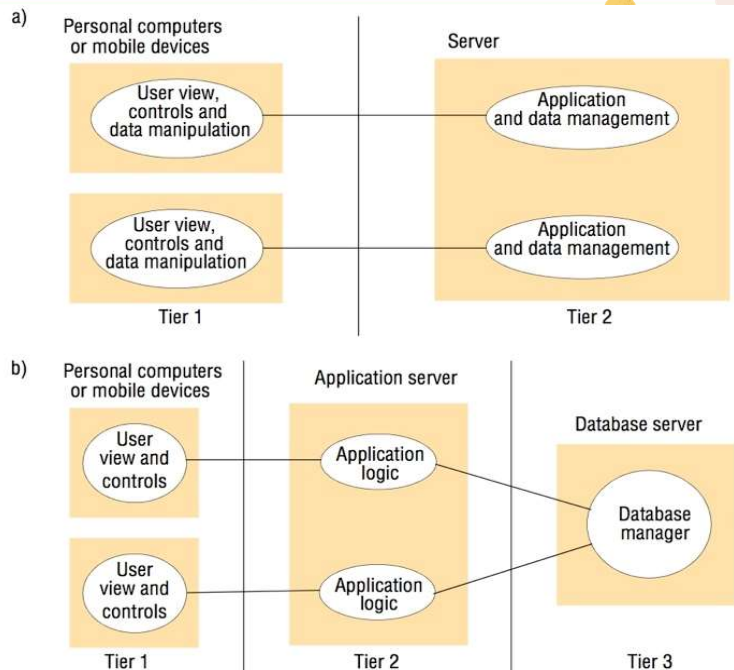


# Layering

- Can also think about layering in terms of **dependencies**:
  - A layer A is above layer B if changes to the interfaces provided by layer A do not require changes to the code of layer B.
- Why layer?
  - **Flexible** — You can add functionality without changing underlying layers.
  - **Reuse** — Many applications can use Java jars, for example.
  - **Reduce complexity** — Too hard to hold everything in your head at once.

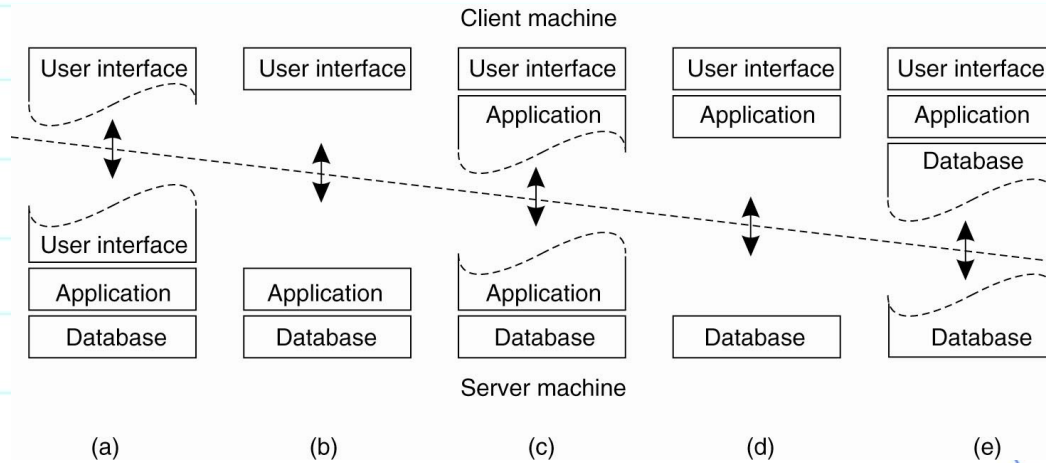


# Two-tier and three-tier architectures



## Alternative Architectures

- Alternative client-server organizations (a) – (e).



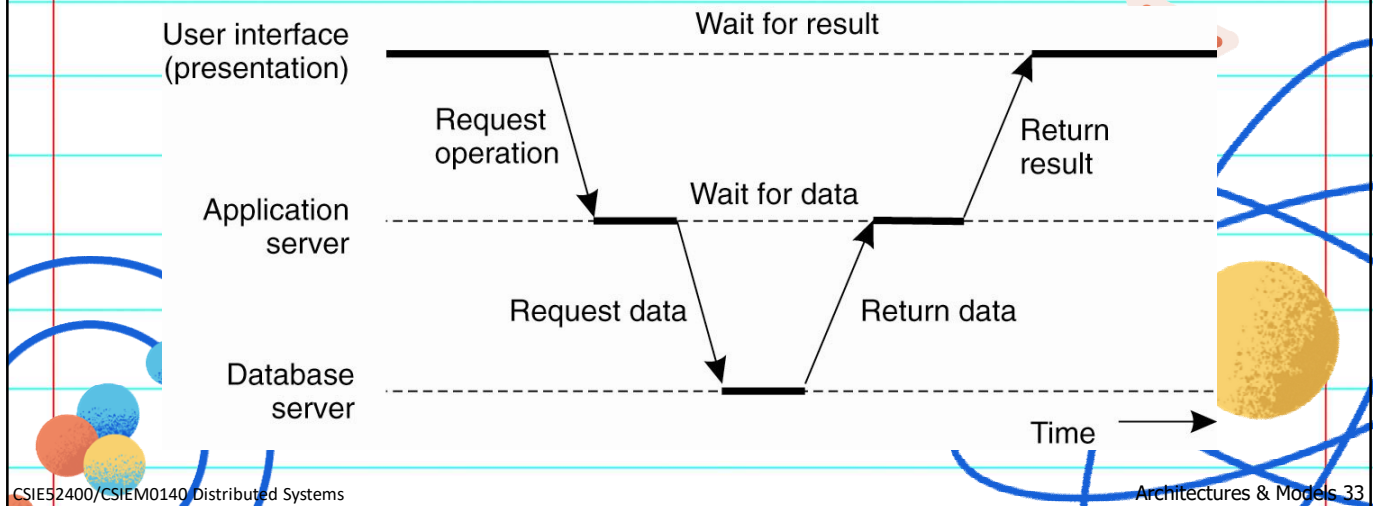
## Examples of Alternative Architectures

- (a): server-side has some control over UI.
- (c): form checking.
- (d): banking application just uploads transaction.
- (e): Local caching
- Also known as **multitiered architectures**.
- What's good about moving things out to desktop machines?
- What's bad?



## Three-tiered Architecture

- Note that the application server acts as client when requesting to the database server.



CSIE52400/CSIEM0140 Distributed Systems

Architectures &amp; Models 33

## Logical Architecture vs. Physical Architecture

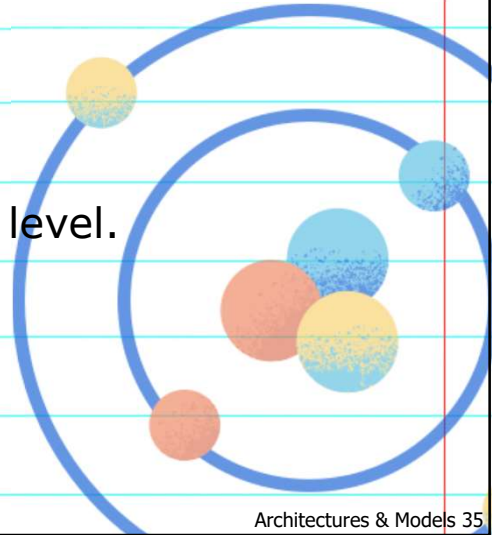
- Physical architecture may or may not match the logical architecture.
- The simplest organization is to have only two types of machines:
  - A **client machine** containing only the programs implementing (part of) the user-interface level
  - A **server machine** containing the rest,
    - the programs implementing the processing and data level
- Or could have other partitioning methods.

CSIE52400/CSIEM0140 Distributed Systems

Architectures &amp; Models 34

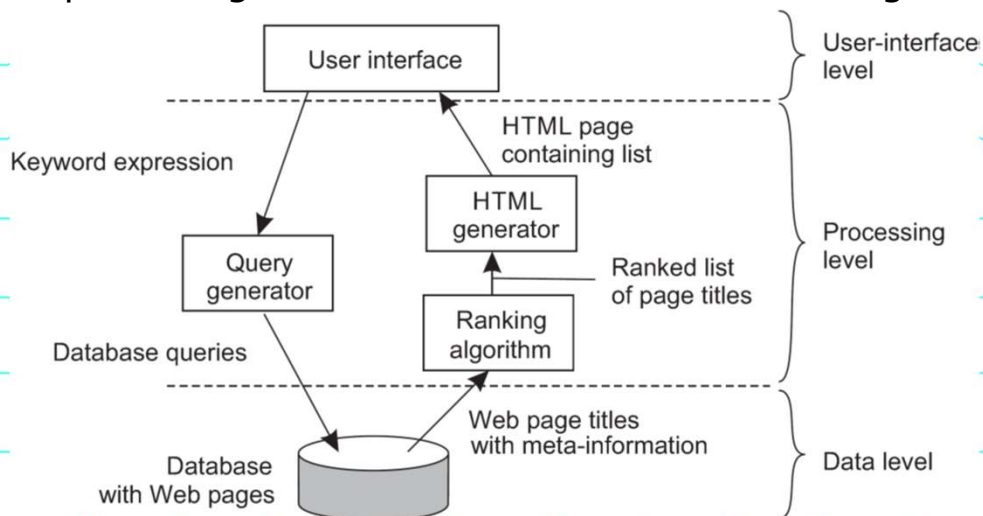
## Application Layering

- Client-server applications are usually constructed with a distinction between three levels:
  - User-interface level
  - Processing level
  - Data level
- Clients implement the user-interface level.
- Servers implement the rest.



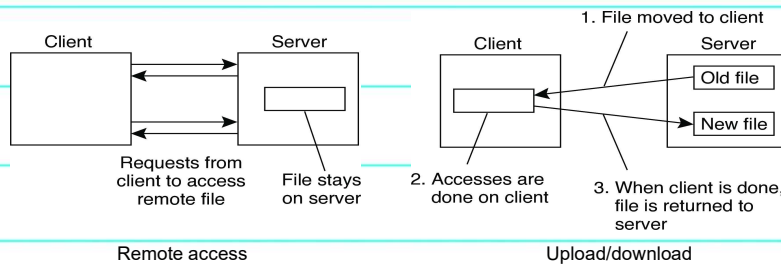
## Example: Search Engine

- The simplified organization of an Internet search engine



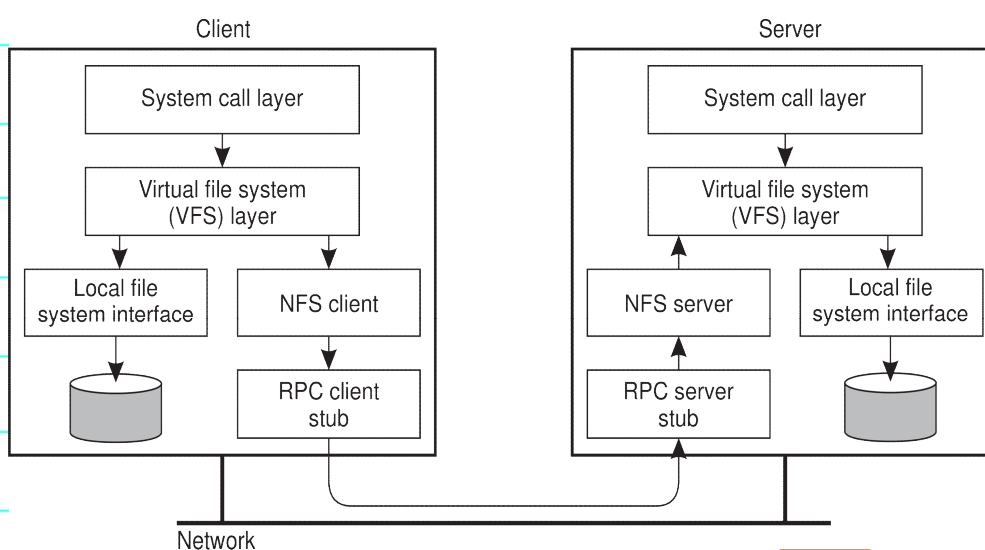
# Example: Network File System

- Each **NFS server** provides a standardized view of its local file system: each server supports the same model, regardless the implementation of the file system.
- The NFS remote access model:



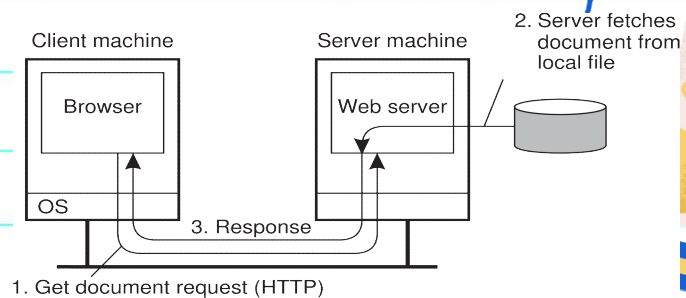
- NFS is a typical upload/download model. The same can be said for systems like Dropbox.

# NFS Architecture



## Example: Simple Web Server

- Traditional Web server:

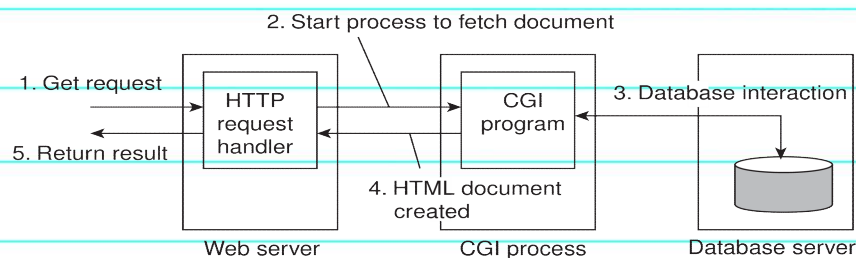


- Simple Web server:

- A website consisted as a collection of HTML files
- HTML files could be referred to each other by a hyperlink
- A Web server essentially needed only a hyperlink to fetch a file
- A browser took care of properly rendering the content of a file

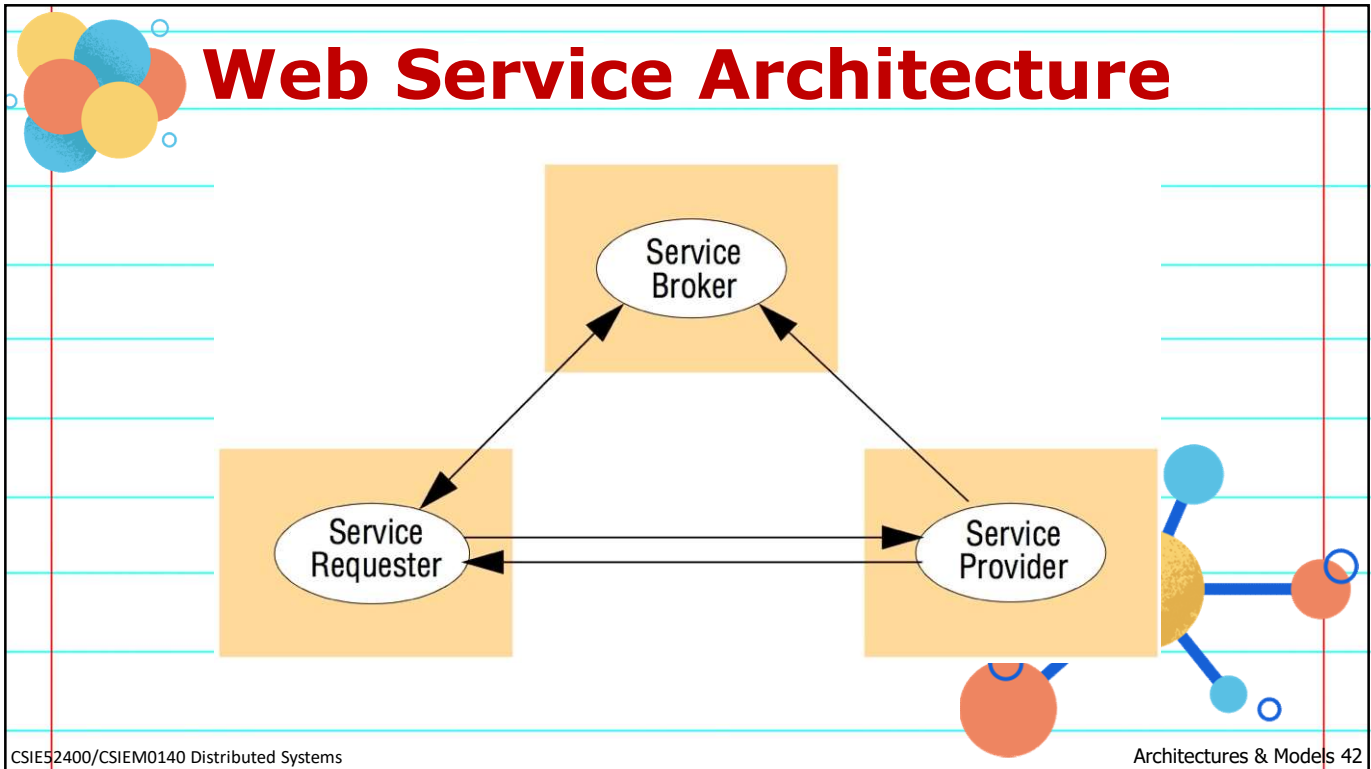
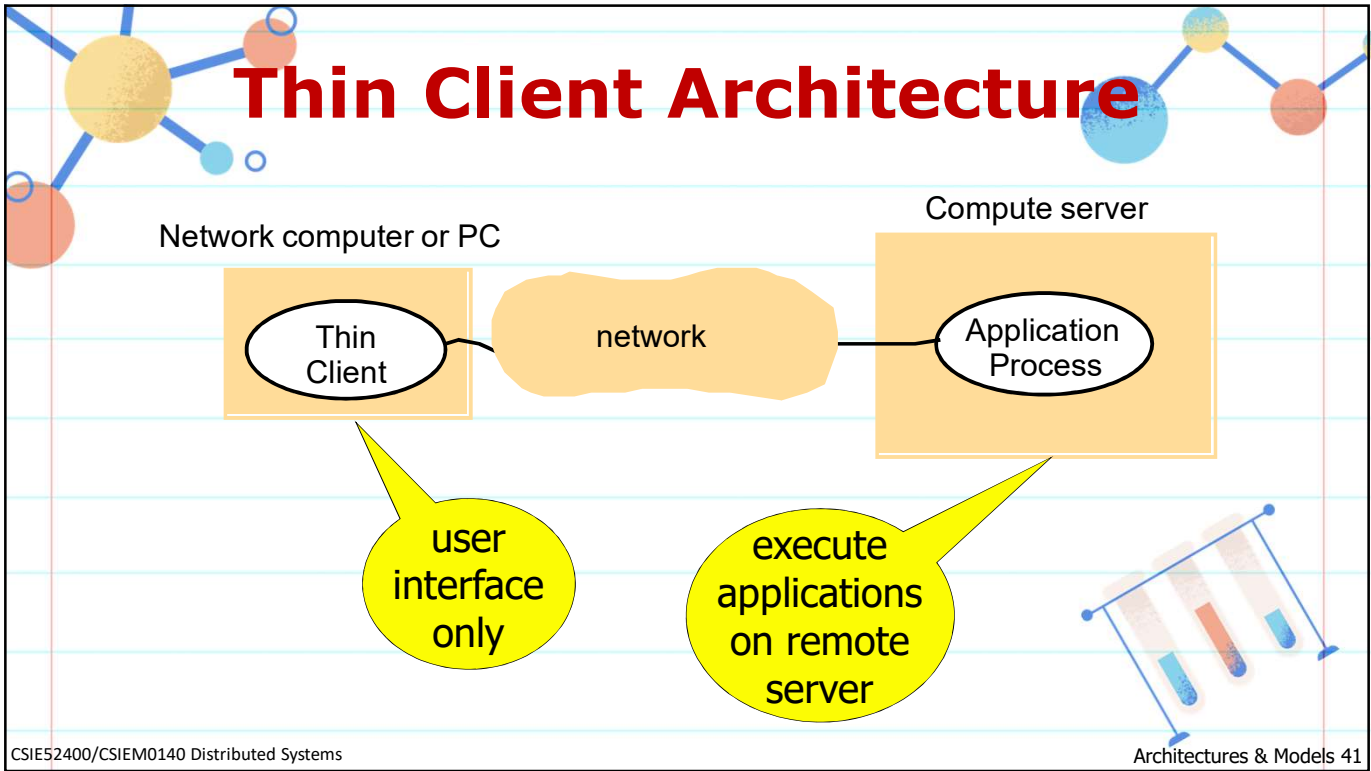
## Example: Less simple Web

- More dynamic Web with CGI and database



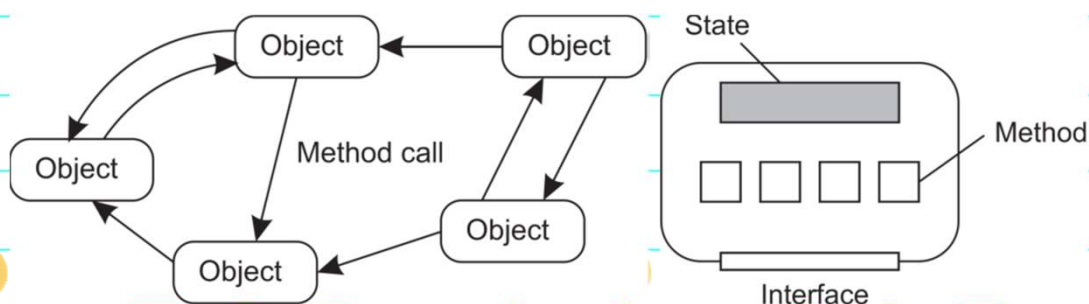
- More dynamic Web:

- A website was built around a **database** with content
- A Webpage could still be referred to by a hyperlink
- A Web server essentially needed only a hyperlink to fetch a file
- A separate program (**Common Gateway Interface**) composed a page
- A browser took care of properly rendering the content of a file



## Object-based Architecture

- Components are **objects**, connected to each other through **method calls**. Objects may be placed on different machines; calls can thus execute across a network.
- Objects are said to **encapsulate data** and offer **methods on that data** without revealing the internal implementation



## RESTful Architecture

- View a distributed system as a collection of **resources**, managed by **components**, and may be added, removed, retrieved, and modified by **applications**.
  - Resources identified through a single **naming scheme (URI)** and usually represented by **JSON** or **XML**
  - All **services** offer the **same interface (uniform interface)**
  - **Messages** sent to or from a service are fully **self-described**
  - After executing an operation, that component forgets everything about the caller (**stateless**)

## RESTful Interface

Method	Operation performed on server	Quality
GET	Read a resource.	Safe
PUT	Insert a new resource or update if the resource already exists.	Idempotent
POST	Insert a new resource. Also can be used to update an existing resource.	N/A
DELETE	Delete a resource .	Idempotent
OPTIONS	List the allowed operations on a resource.	Safe
HEAD	Return only the response headers and no response body.	Safe

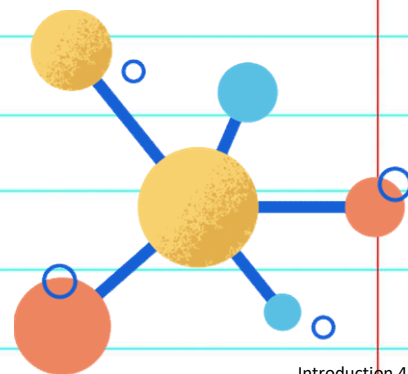
## Example: Amazon's Simple Storage Service (S3)

- **Objects** (i.e., files) are placed into **buckets** (i.e., directories).
- Buckets cannot be placed into buckets.
- Operations on ObjectName in bucket BucketName require the identifier:  
<http://BucketName.s3.amazonaws.com/ObjectName>
- All operations are carried out by sending **HTTP** requests:
  - Create a bucket/object: PUT, along with the URI
  - Listing objects: GET on a bucket name
  - Reading an object: GET on a full URI

## Why RESTful ?

- RESTful architecture is popular because the interface is so **simple**. The catch is that much needs to be done in the **parameter space**.
- As a comparison, the Amazon S3 SOAP interface

Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy



## RESTful vs SOAP

- Assume an interface **bucket** offering an operation **create**, requiring an input string such as **mybucket**, for creating a bucket "mybucket"

### SOAP

```
import bucket
bucket.create("mybucket")
```

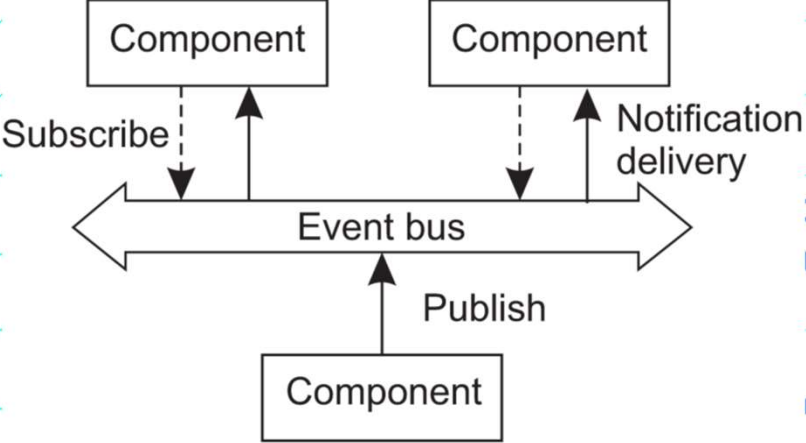
### RESTful

```
PUT "https://mybucket.s3.amazonsws.com/"
```



# Event-based Architecture

- The **event-based** architectural style

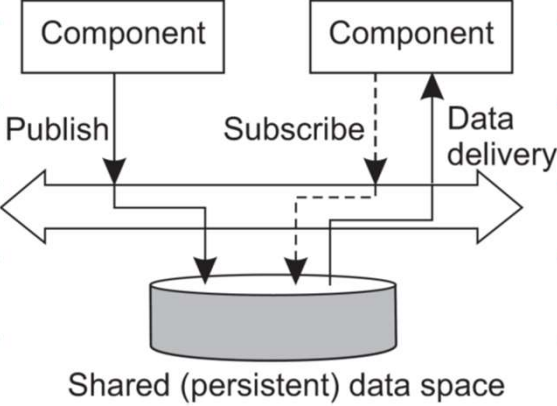


The diagram illustrates the event-based architectural style. It features three rectangular boxes labeled 'Component'. A central horizontal double-headed arrow represents the 'Event bus'. One component at the bottom has a solid arrow labeled 'Publish' pointing up to the event bus. Two components at the top have dashed arrows labeled 'Subscribe' pointing down to the event bus. From the event bus, solid arrows labeled 'Notification delivery' point up to each of the two top components.

CSIE52400/CSIEM0140 Distributed Systems Architectures & Models 49

# Shared Data-space Architecture

- The **shared data-space** architectural style.
  - Processes communicate through a **shared repository**.
  - WebDAV, Linda, tuple-space.



The diagram illustrates the shared data-space architectural style. It features two rectangular boxes labeled 'Component' at the top. A central horizontal double-headed arrow represents the 'Shared (persistent) data space'. The left component has a solid arrow labeled 'Publish' pointing down to the data space. The right component has a dashed arrow labeled 'Subscribe' pointing down to the data space. From the data space, a solid arrow labeled 'Data delivery' points up to the right component.

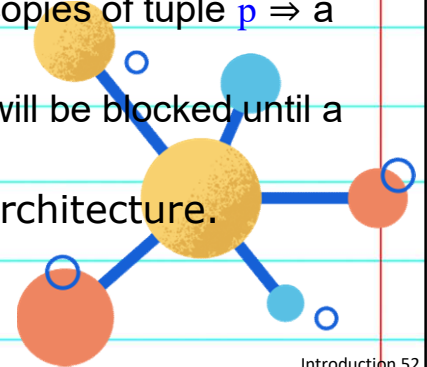
CSIE52400/CSIEM0140 Distributed Systems Architectures & Models 50

## Temporal & Referential Coupling

	Temporally Coupled	Temporally Decoupled
Referentially Coupled	Direct	Mailbox
Referentially Decoupled	Event-based	Shared data space

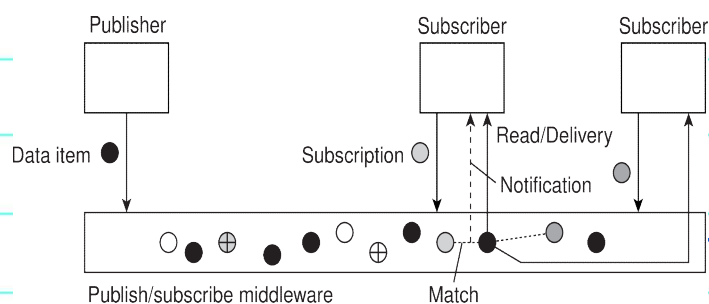
## Example: Linda Tuple Space

- Three simple operations
  - $in(t)$ : remove a tuple matching template  $t$
  - $rd(t)$ : obtain copy of a tuple matching template  $t$
  - $out(t)$ : add tuple  $t$  to the tuple space
- Calling  $out(p)$  twice in a row, leads to storing **two** copies of tuple  $p \Rightarrow$  a tuple space is modeled as a **multiset**.
- Both  $in$  and  $rd$  are **blocking** operations: the caller will be blocked until a matching tuple is found, or has become available.
- A system based on the **shared data-space** architecture.

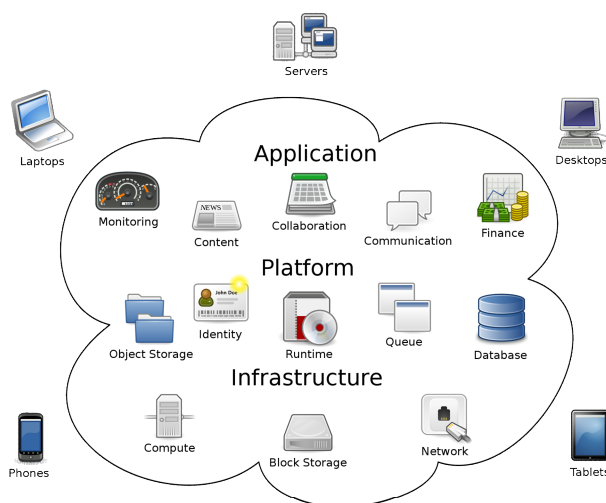


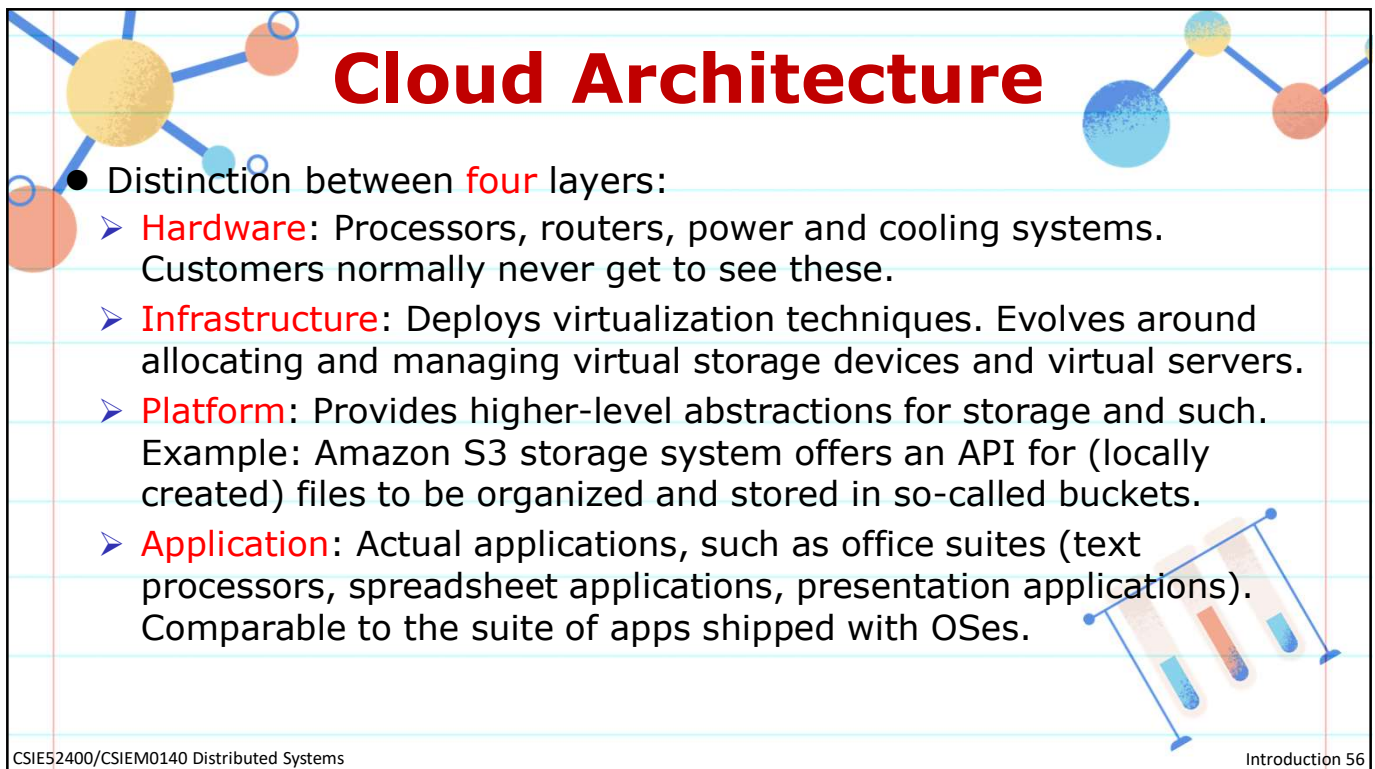
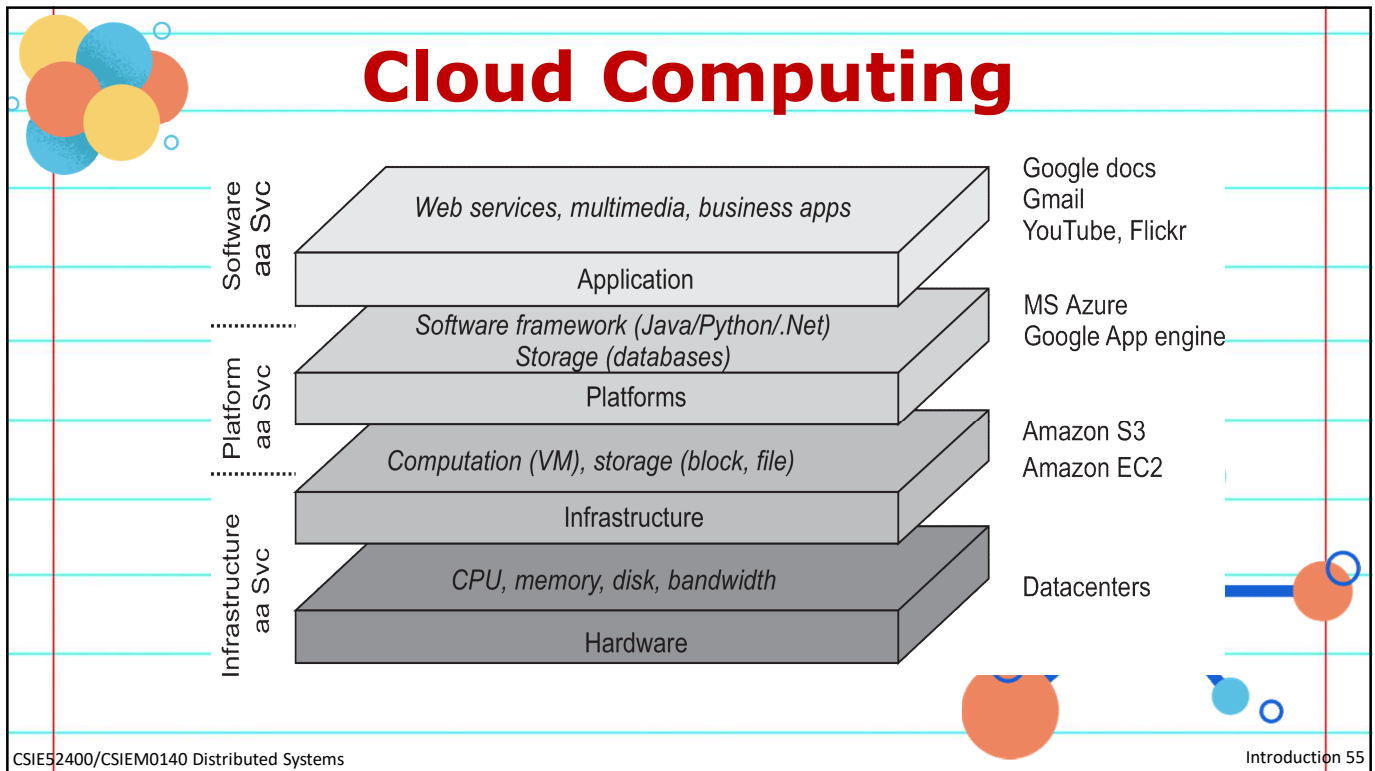
# Publish and Subscribe

- An architecture style with **publishers** and **subscribers**.
- Publishers generate and **publish data events**.
- Subscribers **subscribe** to **desired events**.
- Mechanisms of **matching/notification** of events:
  - Assume events are described by (attribute,value) pairs
  - **Topic-based subscription**: specify a “attribute = value” series
  - **Content-based subscription**: specify a “attribute ∈ range” series



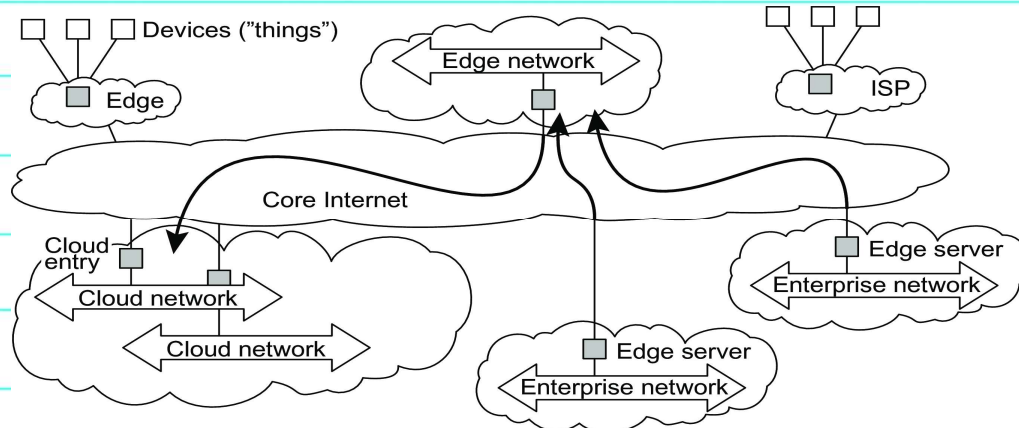
# Cloud Service Architecture





## Edge-server Architecture

- Systems deployed on the Internet where **servers** are placed at the **edge** of the network.



## Edge Advantages

- **Latency** and **bandwidth**: Especially important for certain real-time applications, such as augmented/virtual reality applications. Many people underestimate the latency and bandwidth to the cloud.
- **Reliability**: The connection to the cloud is often assumed to be unreliable, which is often a false assumption. There may be critical situations in which extremely high connectivity guarantees are needed.
- **Security** and **privacy**: The implicit assumption is often that when assets are nearby, they can be made better protected. Practice shows that this assumption is generally false. However, securely handling data operations in the cloud may be trickier than within your own organization.

## Edge Orchestration

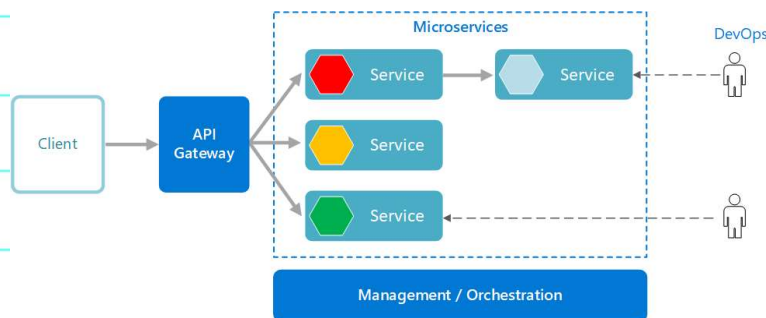
- Managing resources at the edge may be trickier than in the cloud
  - **Resource allocation**: we need to guarantee the availability of the resources required to perform a service.
  - **Service placement**: we need to decide **when** and **where** to place a service. This is notably relevant for mobile applications.
  - **Edge selection**: we need to decide which edge infrastructure should be used when a service needs to be offered. The closest one may not be the best one.
- There is still a lot of buzz about edge infrastructures and computing, yet whether all that buzz makes any sense remains to be seen.

## Microservice Architecture 1

- An **application** is structured as a **collection** of **collaborating services** that are:
  - Small, independent, and loosely coupled
  - Highly maintainable and testable
  - Independently deployable
  - Persisting their own data or external state
  - Organized around business capabilities
  - Communicate with others by using well-defined APIs
  - Owned by a small team
  - Support polyglot programming (can use different langs, technology stack, libraries, or framework)

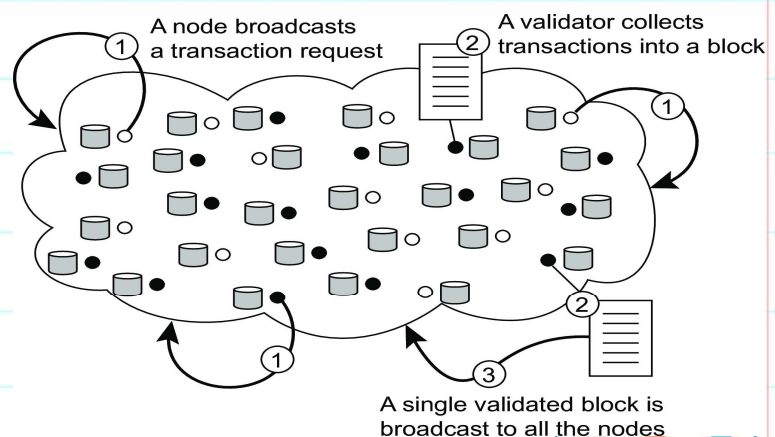
## Microservice Architecture 2

- Typical components and organization
  - **Management/Orchestration**: placing services, identifying failures, rebalancing services, etc.
  - **API Gateway**: entry point for clients, forwards client call to appropriate services



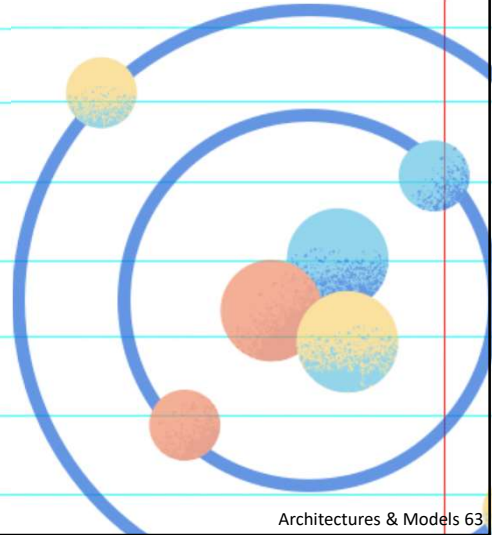
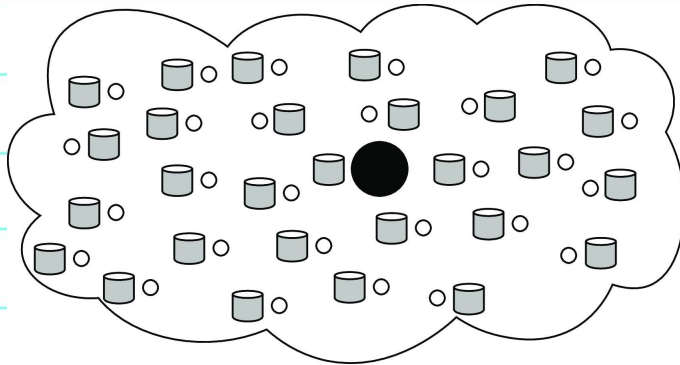
## Blockchains

- Principle working of a blockchain system
- Blocks are organized into an unforgeable **append-only** chain
- Each block in the blockchain is **immutable** ⇒ massive replication
- The real snag lies in who is allowed to append a block to a chain



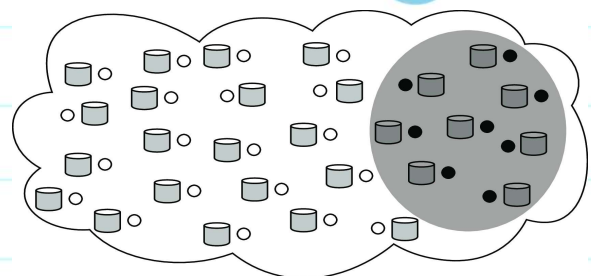
## Appending a block: distributed consensus

- Centralized solution: A single entity decides on which validator can go ahead and append a block.
- Does not fit the design goals of blockchains

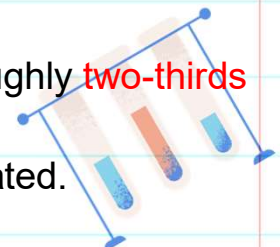


## Appending a block

- Distributed solution (permissioned)



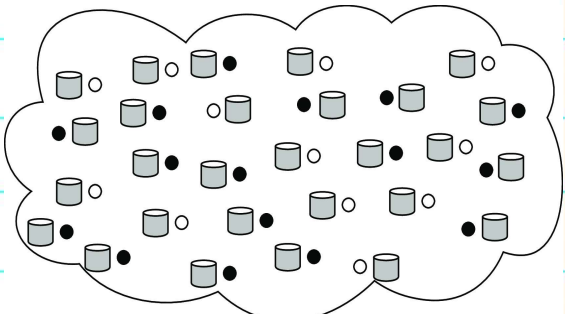
- A selected, relatively **small group** of servers jointly reach consensus on which validator can go ahead.
- None of these servers needs to be trusted, as long as roughly **two-thirds** behave according to their specifications.
- In practice, only **a few tens of servers** can be accommodated.





## Appending a block

- Decentralized solution (permissionless)

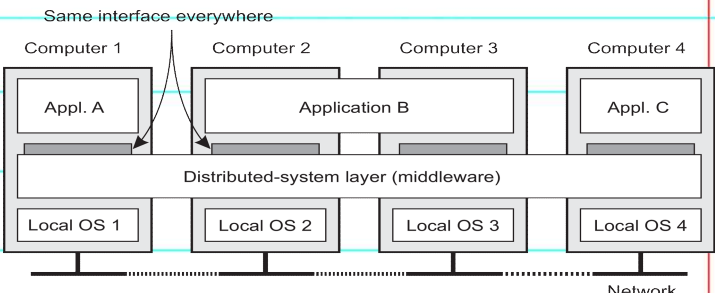


- Participants collectively engage in a **leader election**. Only the elected leader is allowed to append a block of validated transactions.
- Large-scale, decentralized leader election that is fair, robust, secure, and so on, is far from trivial.

Architectures & Models 65

## Middleware: OS of distributed systems

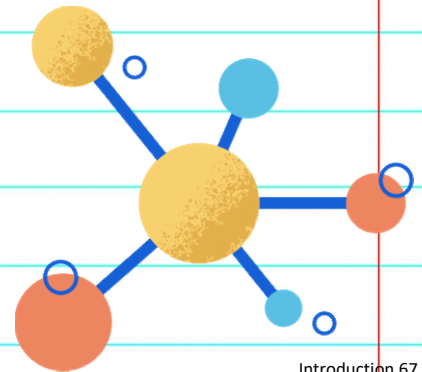
- **Middleware**
  - Provide higher-level abstraction
  - Hide the heterogeneity
  - Promote interoperability and portability
  - Often based on the corresponding architectural models



Architectures & Models 66

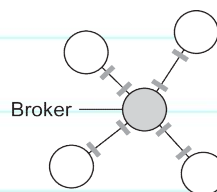
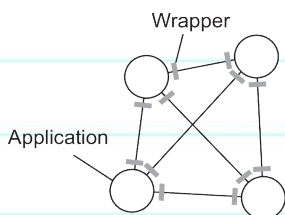
## Middleware: How

- Can use **legacy** to build. But the interfaces of legacy components are most likely not suitable for all applications.
- **Solution:** A **wrapper** or **adapter** offers an interface acceptable to a client application. Its functions are transformed into those available at the component.

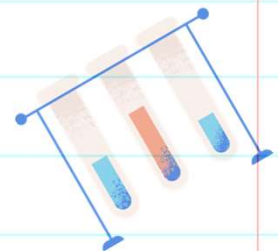


## Organizing Wrappers

- **Two** solutions: **1-on-1** or through a **broker**

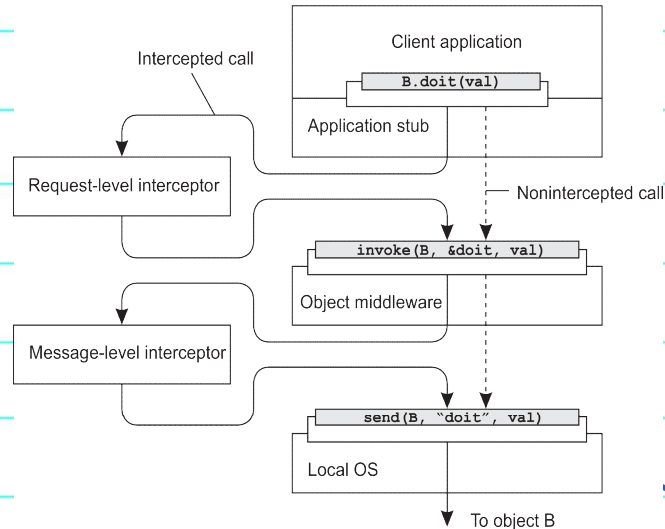


- Complexity with  $N$  applications
  - **1-on-1:** requires  $N \times (N - 1) = O(N^2)$  wrappers
  - **broker:** requires  $2N = O(N)$  wrappers



# Adaptable Middleware

- Middleware contains solutions that are good for **most** applications.
- May want to **adapt** its behavior for specific applications.
- Example: Can intercept the usual flow of control

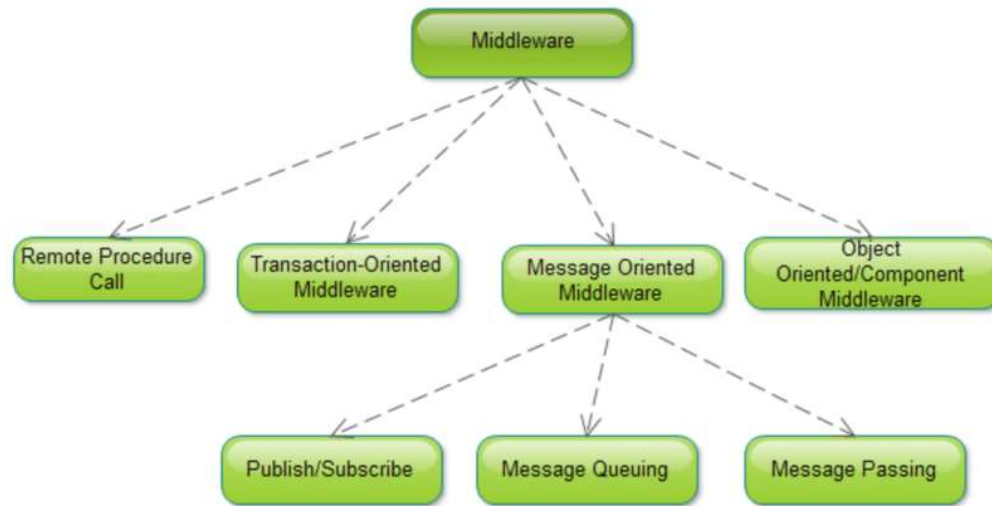


# Categories of Middleware

- Distributed objects
- Distributed components
- Publish-subscribe
- Message queues
- Web services
- Peer-to-peer

Major categories:	Subcategory	Example systems
<i>Distributed objects (Chapters 5, 8)</i>	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
<i>Distributed components (Chapter 8)</i>	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
<i>Publish-subscribe systems (Chapter 6)</i>	-	CORBA Event Service
	-	Scribe
	-	JMS
<i>Message queues (Chapter 6)</i>	-	Websphere MQ
	-	JMS
<i>Web services (Chapter 9)</i>	Web services	Apache Axis
	Grid services	The Globus Toolkit
<i>Peer-to-peer (Chapter 10)</i>	Routing overlays	Pastry
	Routing overlays	Tapstry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

# Categories of Middleware



# Models of Distributed Systems

## Fundamental Models

- **Abstract models** to discuss **individual aspects** of a distributed system
- Focus on three aspects:
  - **Interaction model**: Addresses communication and coordination between processes
  - **Failure model**: Defines and classifies faults and methods of recovery or tolerance
  - **Security model**: Defines security threats and mechanisms for resisting them

## Interaction Model

- Distributed systems are composed of **interacting processes**.
- Behaviors of processes are captured by **distributed algorithms** describing the **computing steps** and **message transmission** of processes.
- The **rate** of each process and the **timing** of message transmission **cannot** in general **be predicted**.
- Each process can only access its **own state**.
- **No** direct access to the **global state** of the system.
- **No global time**.

## Communication Channels

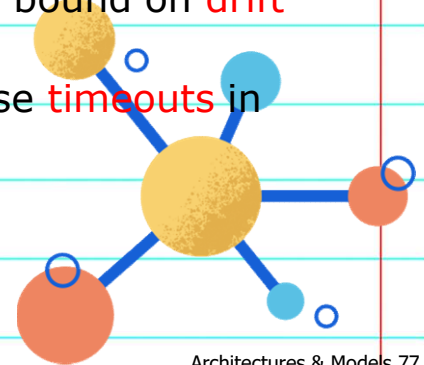
- Channels can be modeled in various ways
  - Streams
  - Message passing networks
- Performance characteristics
  - Latency – The delay between the start of message transmission and the beginning of reception.
  - Bandwidth – The total amount of info that can be transmitted over a given time.
  - Jitter – The variation in message delivering time.

## Clocks and Timing Events

- Each computer has its own **clock**.
- Different clocks have different **drift rates** (the rate a clock deviates from a perfect clock).
- **Clock synchronization** is to synchronize the clocks of a set of computers.
- In most cases, **relative ordering of events** is more important than absolute timing.
- It is possible to construct **logical clocks** for process synchronization.

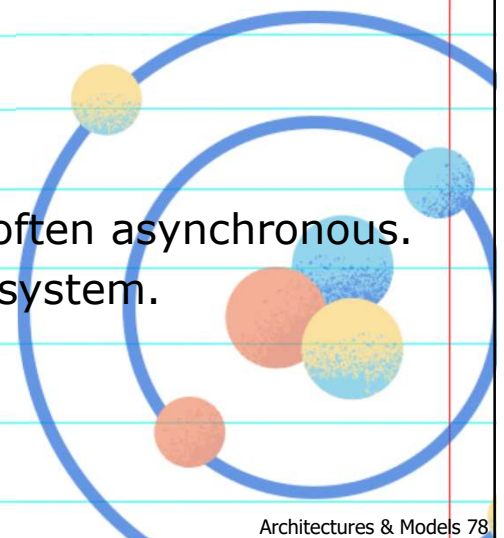
## Two Variants of Interaction

- **Synchronous distributed systems** – Systems in which the following **bounds** are **defined**:
  - Each **execution step** has known lower & upper bounds.
  - Each **message transmission** is received within known bound.
  - Each process has a local **clock** with known bound on **drift rate**.
- In a synchronous system, it is possible to use **timeouts** in distributed system design.



## Two Variants of Interaction

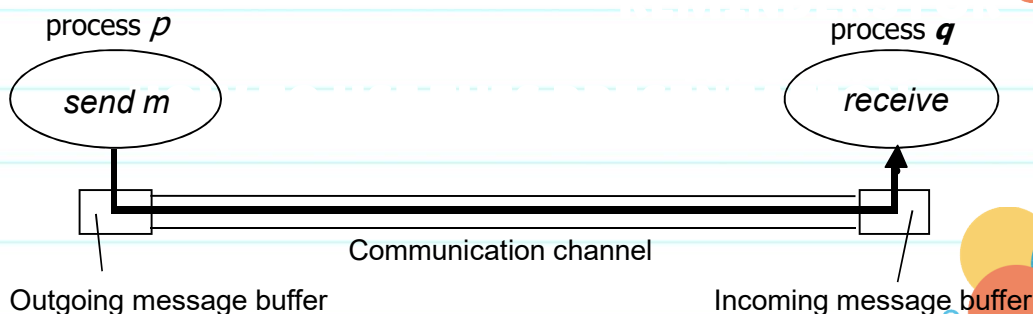
- **Asynchronous distributed systems** – Systems with **no bounds** on:
  - Process execution speeds
  - Message transmission delays
  - Clock drift rates
- Actual distributed systems are very often asynchronous.
- Internet is exactly an asynchronous system.







# Processes and Channels



# Omission and Arbitrary Failures

<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
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 <code>send</code> , 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.

## Timing Failures

<i>Class of Failure</i>	<i>Affects</i>	<i>Description</i>
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

## Masking Failures

- Failures are unavoidable.
- We can only **mask failures**
  - By **hiding** it altogether
  - By **converting** it into a more acceptable type of failure
- Examples of techniques for masking failures
  - Message checksums
  - Retransmission
  - Replication
  - ... (more in later chapters)

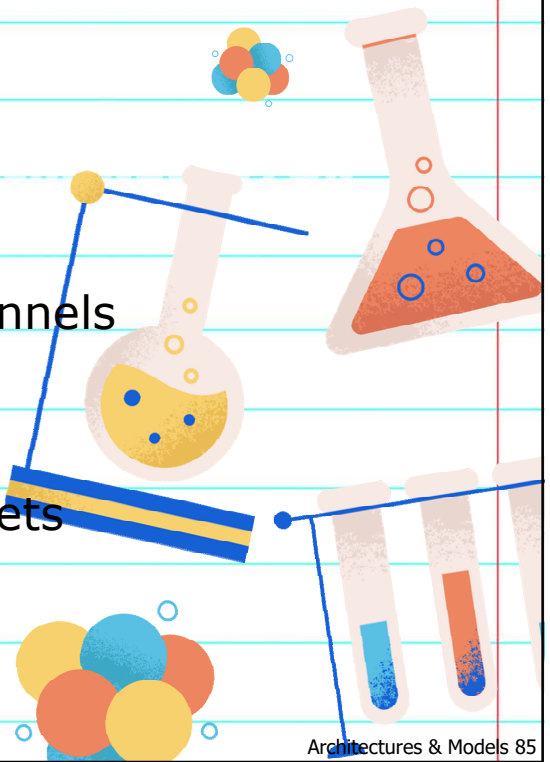
## Security Models

### ● Threats:

- threats to processes
- threats to communication channels
- denial of service

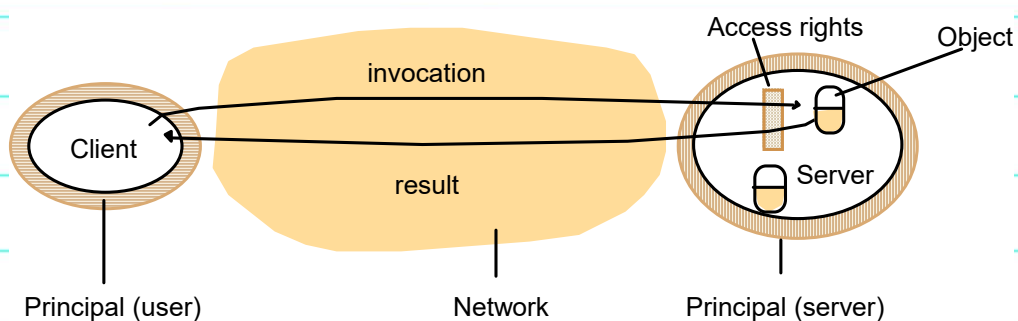
### ● Protection:

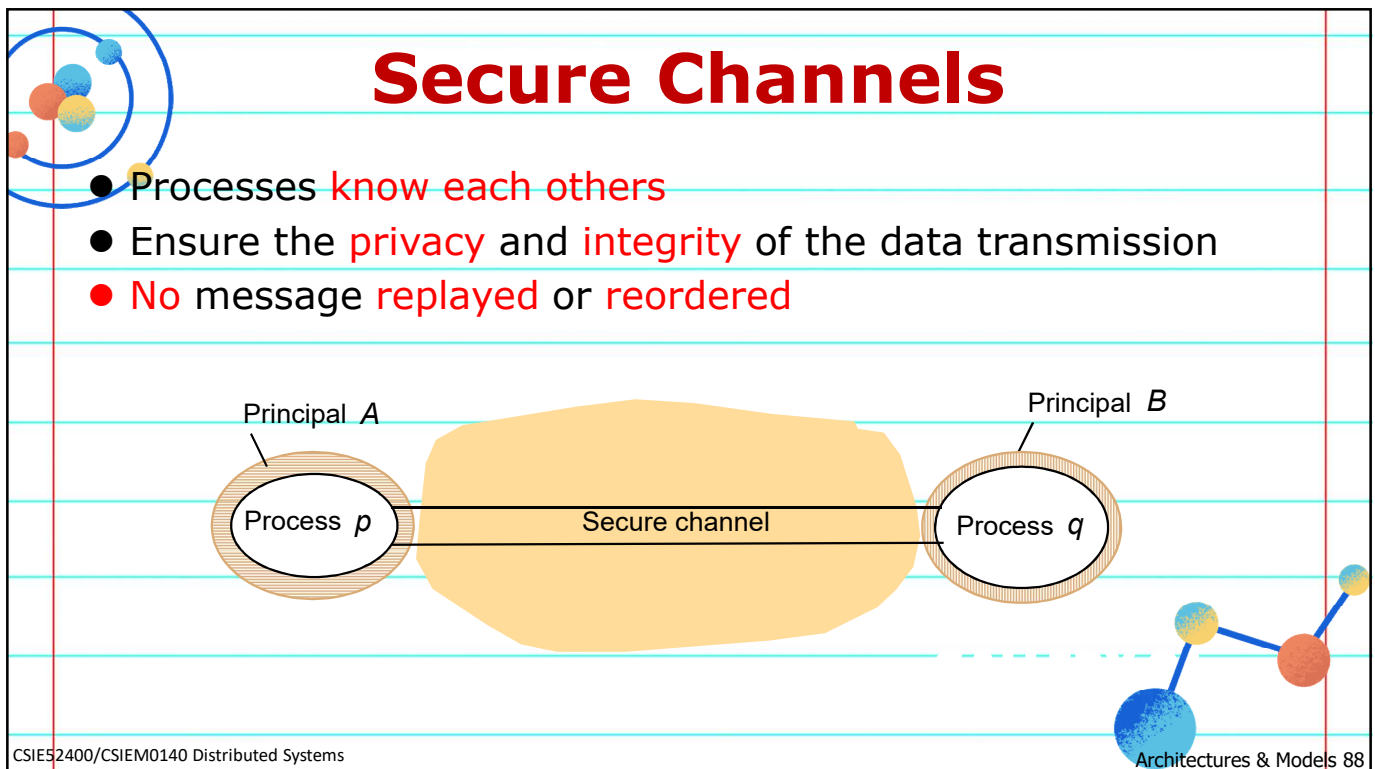
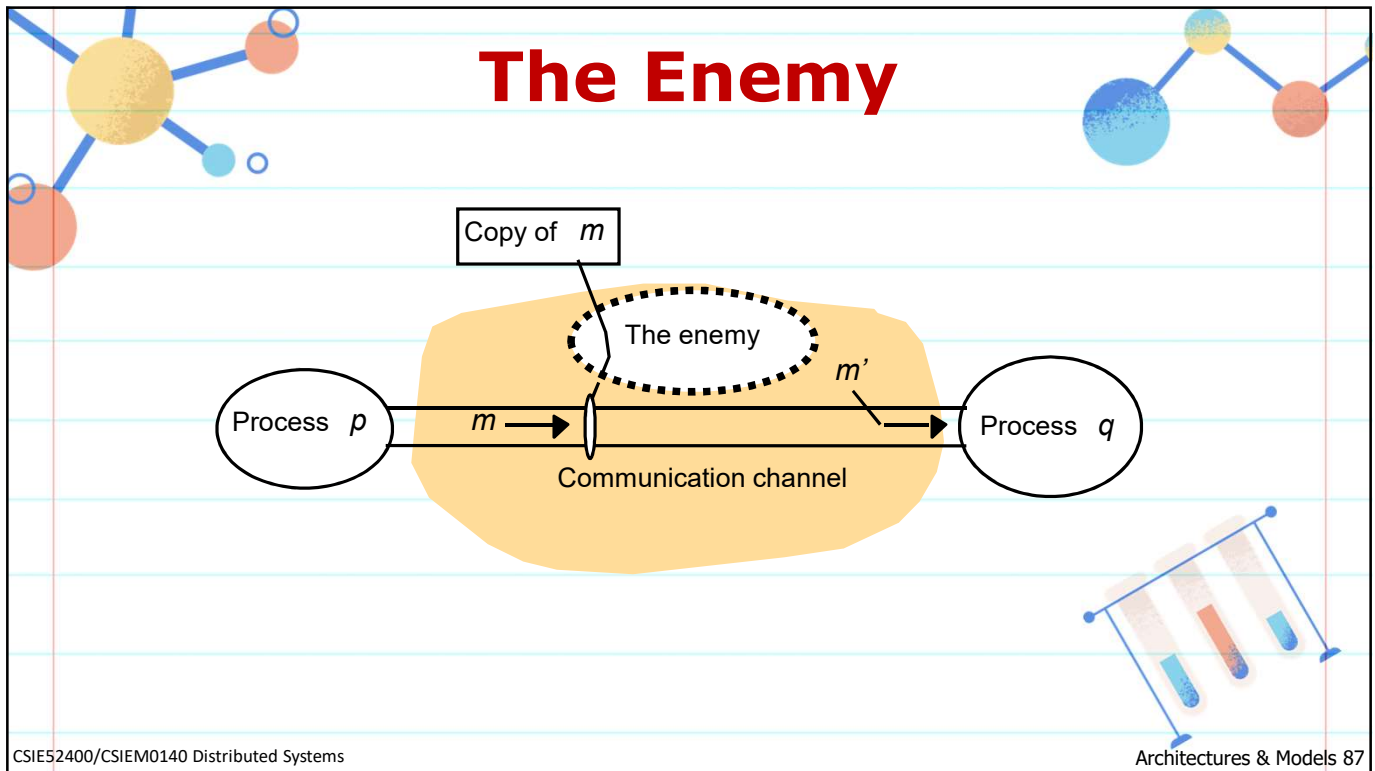
- cryptography and shared secrets
- authentication



## Objects and Principals

- **Access rights:** who can invoke the operations
- **Principal:** the authority on which an invocation or result is issued.



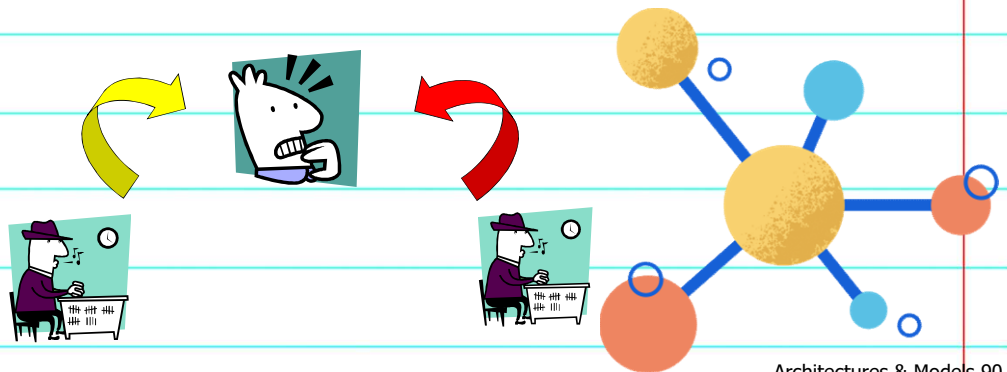


## Security Threats

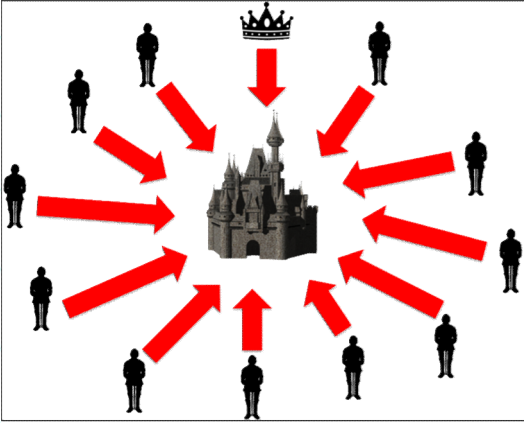
- Security threats can come from any place.
- Two interesting examples
  - **Denial of service (DOS)** – Enemy interferes with the activities of authorized users by making **excessive** and **pointless** accesses in a network.
  - **Mobile code**
    - Mobile code raises new and interesting security problems.
    - Can easily play a **Trojan horse** role.
    - Can be carried in many ways: emails, Web pages, applets, Active X, ...

## HW 1: Byzantine Generals Problem

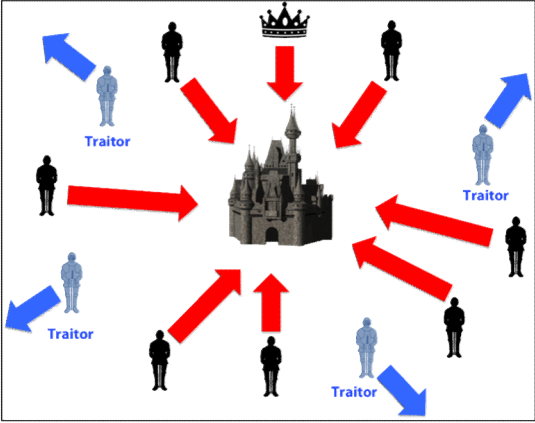
- This is a classic problem in distributed system design.
- In a distributed system, failed components can send conflicting information.
- Different parts of the system receive different information.



## Attack or Retreat ?



**Coordinated Attack Leading to Victory**



**Uncoordinated Attack Leading to Defeat**

How to reach the same agreement among loyal generals?

CSIE52400/CSIEM0140 Distributed Systems
Architectures & Models 91

## The Classic Problem

- Each **division** of the Byzantine army are directed by its own **general**.
- Some of the generals may be **traitors**.
- Generals communicate with each other by **reliable messengers**.
- Requirements:
  - All **loyal generals** decide upon the **same plan** of action.
  - A **small number of traitors** cannot cause the loyal generals to adopt a bad plan.

CSIE52400/CSIEM0140 Distributed Systems
Architectures & Models 92

## Variations and Impossibility

- How many traitors does it take to make the agreement among loyal generals impossible ?
- What if the messengers were not reliable ?
- There are several variant problems. Can you think out a different one by yourself ?
- Do not try to look for answer from the net or AI. It will lose all the fun of this assignment.
- You may do that AFTER thinking about the problem thoroughly and propose your answers.

## Reference

- Lamport, L., Shostak, R., Pease, M. "The Byzantine Generals Problem". *ACM TOPLAS*. Vol 4. Num. 3, July, 1982.
- There are several variant problems based on the classic problem.
- Due date: Mar 19, 2024
- Submit through the NDHU e-learning (東華e學苑). My email and disk quota is almost full and therefore not reliable.