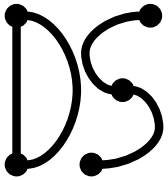


Research Group
Distributed Systems


# Distributed Systems

## Architectures & Processes

Olaf Landsiedel

# Notes

- Labs
  - Please submit in teams of two!

# Last Time

- Communication
  - Tour de Protocol Stack
  - Hour glass model

# Today

- Architectures
  - Design choices for distributed systems
- Processes
  - Refresh your knowledge on concurrency etc.

Part I

# Architecture

# A Distributed System

- A Distributed System is characterized by?
  - Multiple devices
  - Connected by a network
  - Cooperating on some task

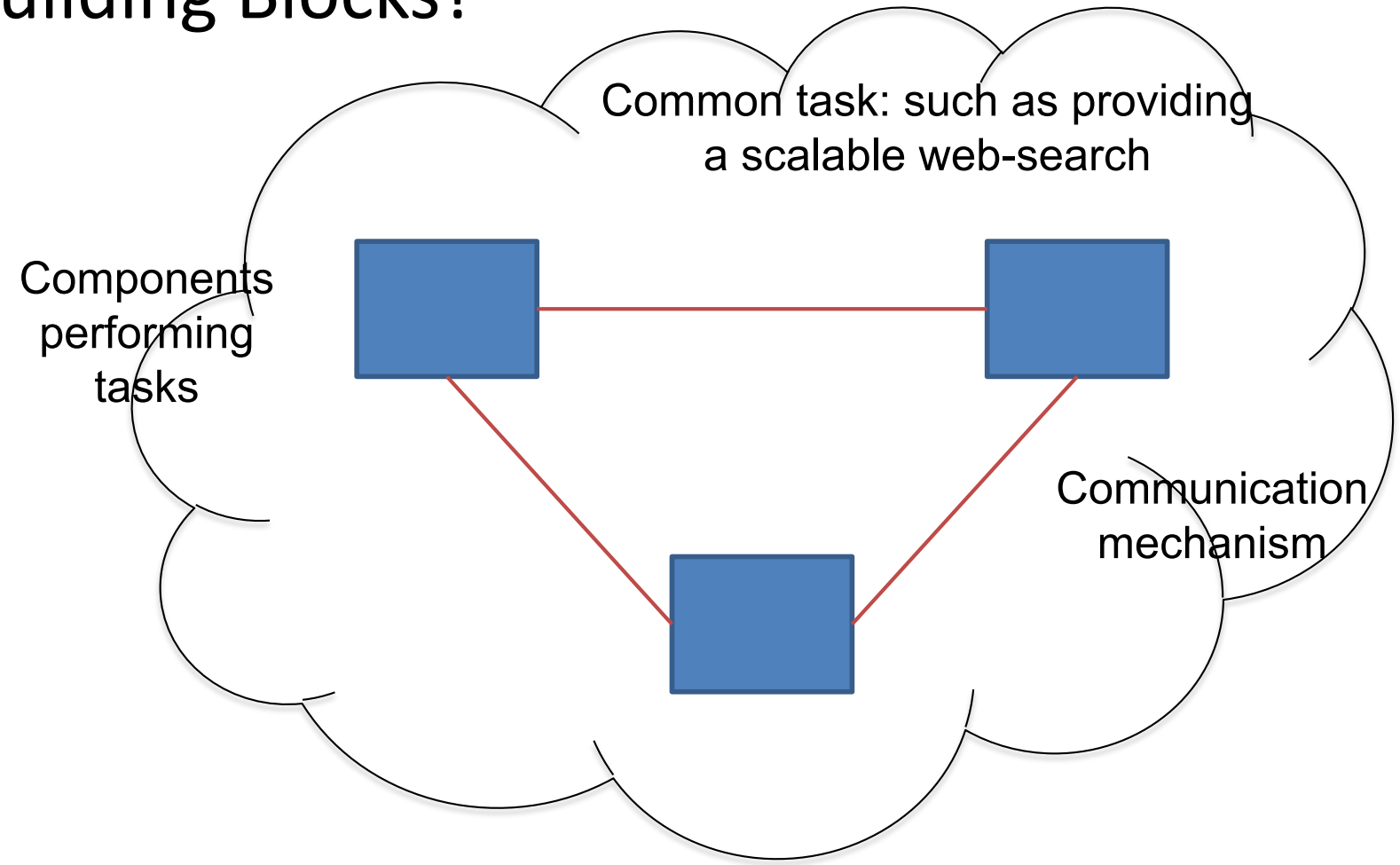
- You know when you have one …

  … when the failure of a computer you've never heard of stops you from getting any work done

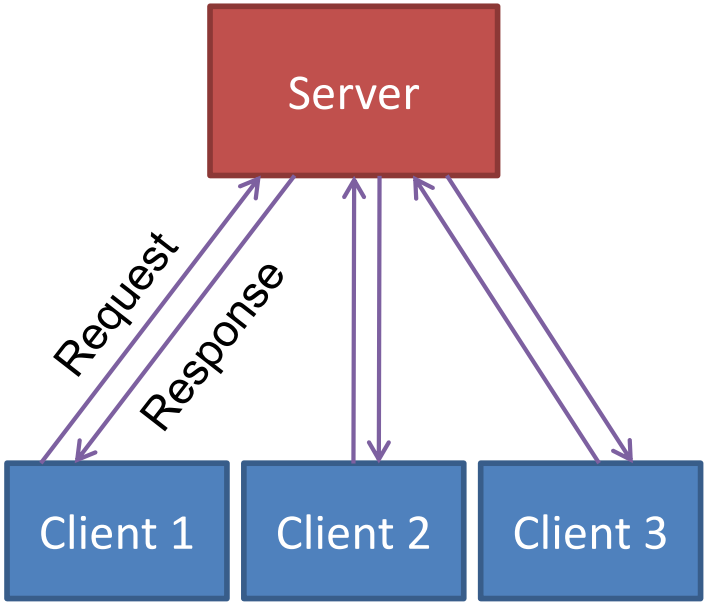
  (*L. Lamport*)

# A Distributed System

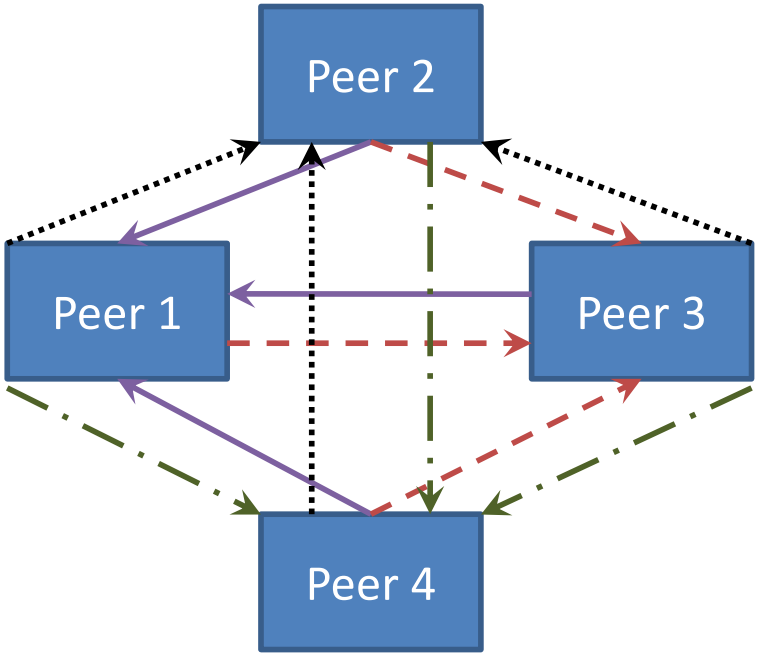
- Building Blocks?



# Building Distributed Systems: Design Choices




Client – Server Model




Peer-To-Peer Model

## Pros and Cons?

# Classification of Distributed Systems

- What are the entities that are communicating in a DS?
  a) Communicating entities
- How do the entities communicate?
  b) Communication paradigms
- What roles and responsibilities do they have?
  c) Roles and responsibilities
- How are they mapped to the physical distributed infrastructure?
  d) Placement of entities

# Classification of Distributed Systems

- What are the entities that are communicating in a DS?
  a) Communicating entities
- How do the entities communicate?
  b) Communication paradigms
- What roles and responsibilities do they have?
  c) Roles and responsibilities
- How are they mapped to the physical distributed infrastructure?
  d) Placement of entities

# Communicating Entities

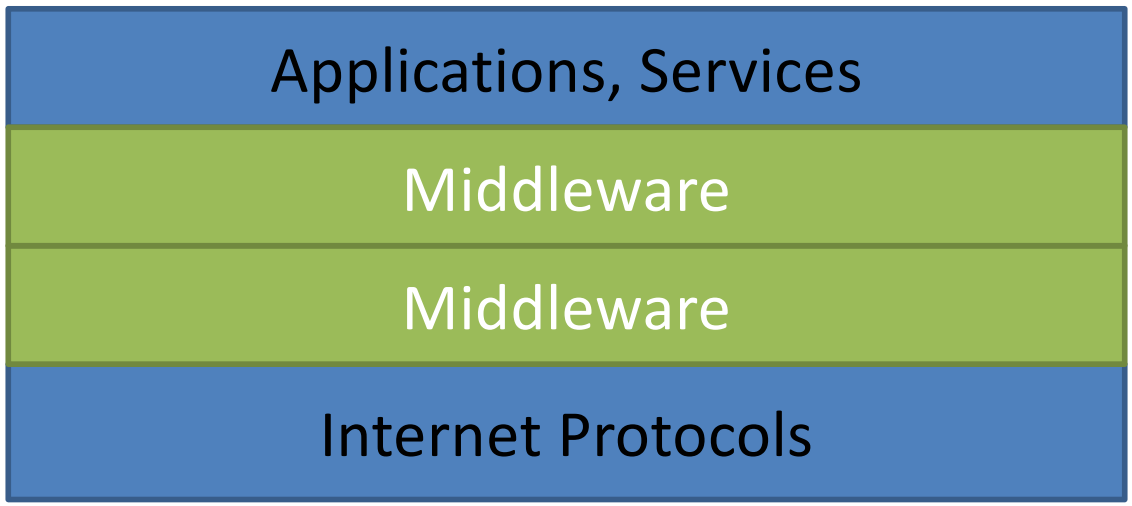
- What entities are communicating in a DS?
  - System-oriented entities
    - Processes
    - Threads
    - Nodes
  - Problem-oriented entities
    - Objects (in *object-oriented programming* based approaches)

# Classification of Distributed Systems

- What are the entities that are communicating in a DS?
  a) Communicating entities
- How do the entities communicate?
  b) Communication paradigms
- What roles and responsibilities do they have?
  c) Roles and responsibilities
- How are they mapped to the physical distributed infrastructure?
  d) Placement of entities

# Communication Paradigm

- Middleware
  - Service “Library”
  - On server and client




Example

HTML, JS, …

AJAX, WebSockets, …

HTTP, SPDY, …

TCP, UDP,

# Problems

- Space Coupling:
  - Where the procedure resides should be known in advance
- Time Coupling:
  - On the receiver, a process should be explicitly waiting to accept requests for procedure calls

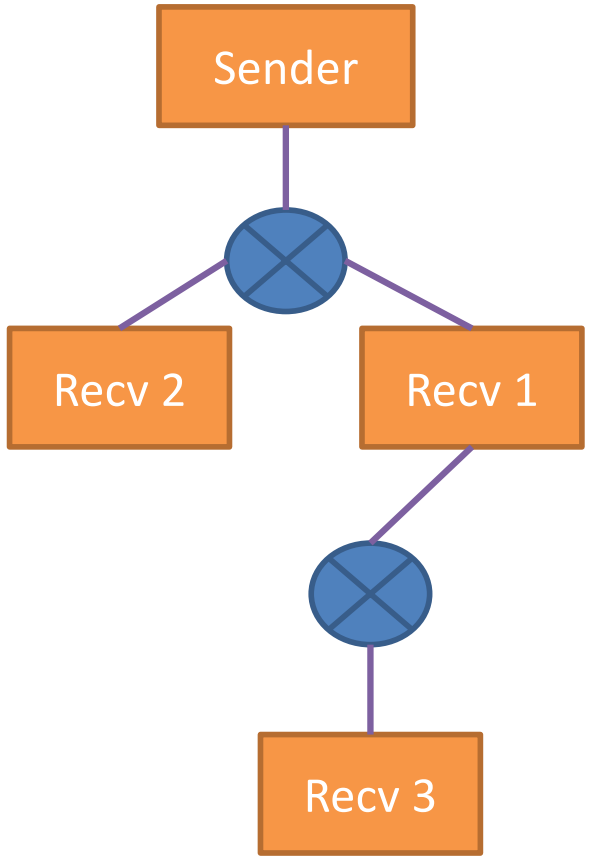
- Next: Indirect communication
  - Aiming to fix this

# Indirect Communication Paradigm

- Indirect communication uses middleware to:
  - Provide one-to-many communication
  - Some mechanisms eliminate space and time coupling
    - Sender and receiver do not need to know each other's identities
    - Sender and receiver need not be explicitly listening to communicate
- Approach used: Indirection
  - Sender → A middle-man → Receiver
- Types of indirect communication
  1. Group communication
  2. Publish-subscribe
  3. Message queues

# 1. Group Communication

- One-to-many communication
  - Multicast communication
- Abstraction of a group
  - Group is represented in the system by a *groupId*
  - Recipients join the group
  - A sender sends a message to the group which is received by all the recipients

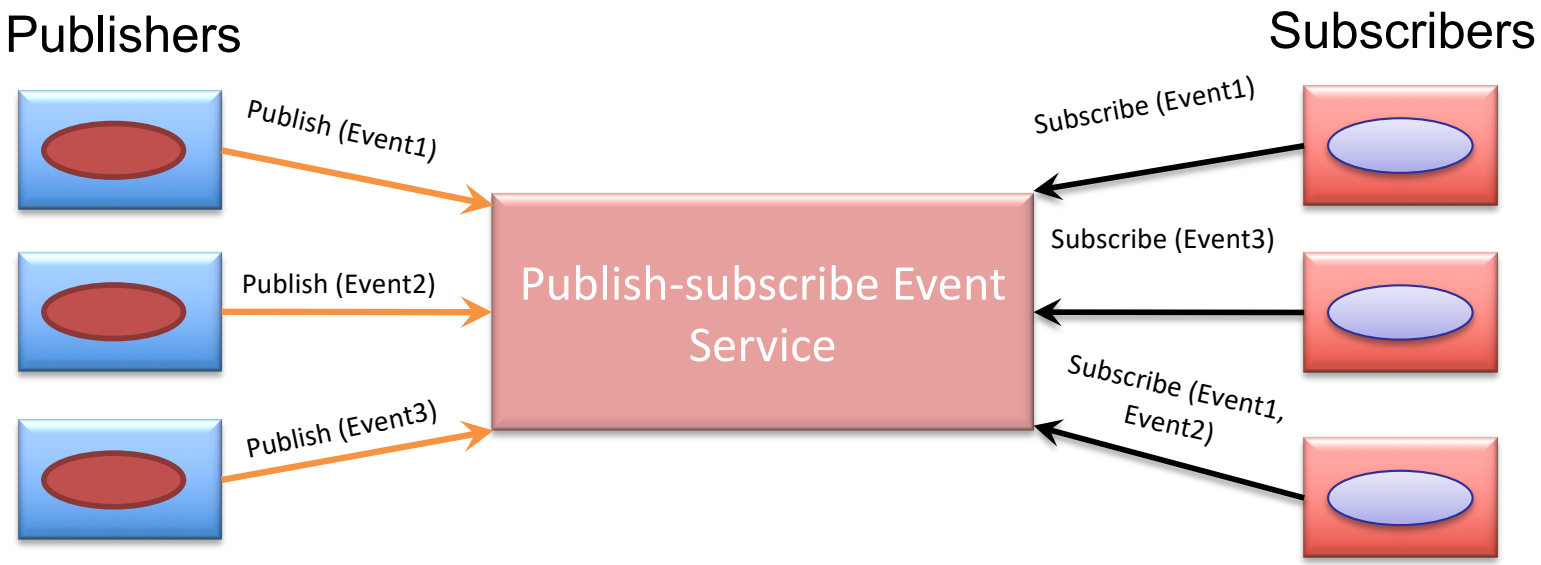


# 1. Group Communication (cont'd)

- Services provided by middleware
  - Group membership
  - Handling the failure of one or more group members
- Advantages?
  - Enables one-to-many communication
  - Efficient use of bandwidth
  - Identity of the group members need not be available at all nodes
- Disadvantages
  - Time coupling

# 2. Publish-Subscribe

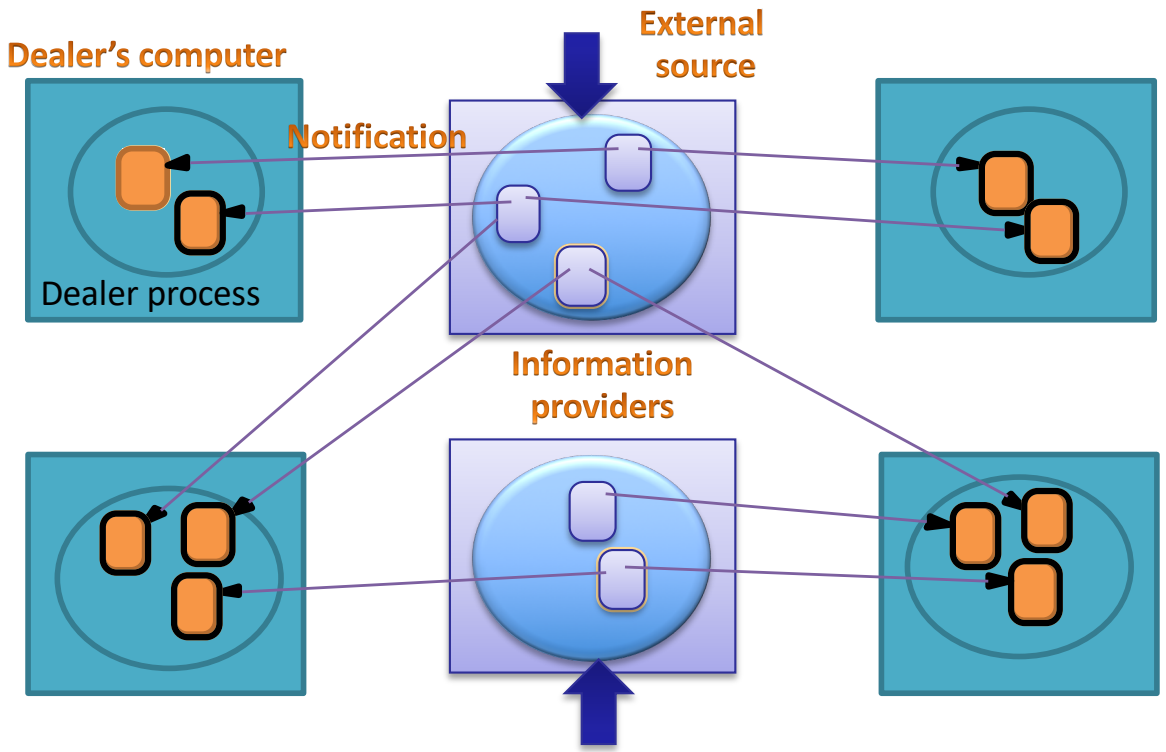
- An event-based communication mechanism
  - Publishers publish events to an event service
  - Subscribers express interest in particular events




- Large number of producers distribute information to large number of consumers

# 2. Publish-Subscribe (cont'd)

## Example: Financial trading

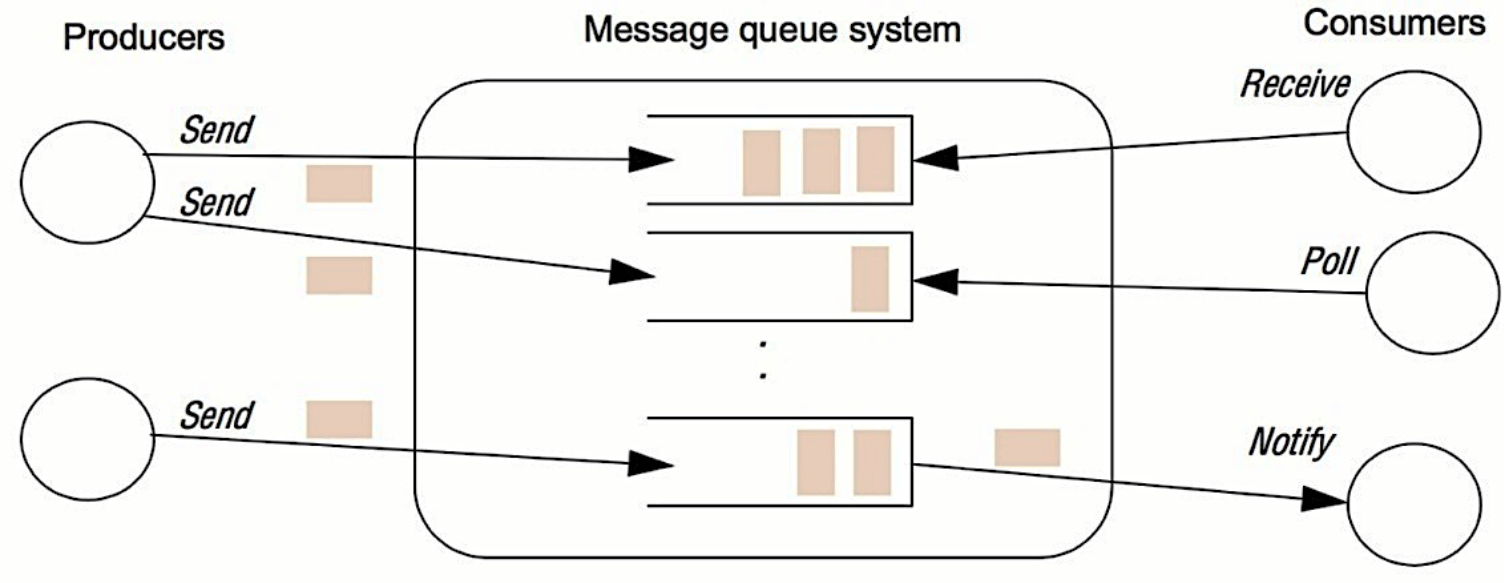



When is publish subscribe good?

# 3. Message Queues

- A refinement of Publish-Subscribe where
  - Producers deposit the messages in a queue
  - Messages are delivered to consumers through different methods
  - Queue takes care of ensuring message delivery

- Advantages
  - Enables space decoupling
  - Enables time decoupling



# Last Time

- Tour de Protocol Stack, Part II
- Architectures of Distributed Systems

# Recap: Communication Entities and Paradigms

| ***Communicating entities (what is communicating)*** | |
|---|---|
| *System-oriented* | *Problem-oriented* |
| • Nodes<br>• Processes<br>• Threads | • Objects |

| ***Communication Paradigms (how they communicate)*** | | |
|---|---|---|
| *"Raw"* | *Middleware* | *Indirect Communication* |
| • Sockets | • "Library" | • Group communication<br>• Publish-subscribe<br>• Message queues |

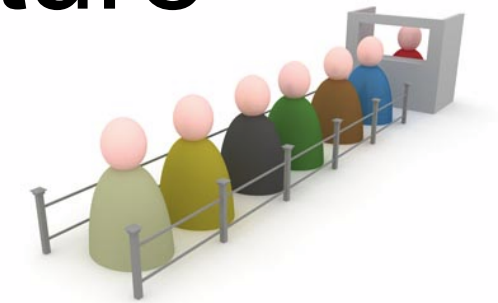
# Classification of Distributed Systems

- What are the entities that are communicating in a DS?
  a) Communicating entities
- How do the entities communicate?
  b) Communication paradigms
- What roles and responsibilities do they have?
  c) Roles and responsibilities
- How are they mapped to the physical distributed infrastructure?
  d) Placement of entities

# Roles and Responsibilities

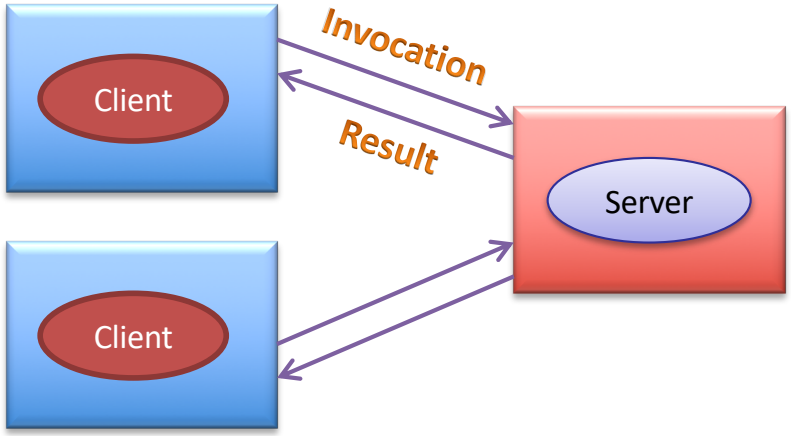
- In DS, communicating entities take on roles to perform tasks
- Roles are fundamental in establishing overall architecture
  - Question: Does your smart-phone perform the same role as Google Search Server?
- We classify DS architectures into two types based on the roles and responsibilities of the entities
  - Client-Server
  - Peer-to-Peer

# Client-Server Architecture



- Approach:
  - Server provides a service that is needed by a client
  - Client requests to a server (invocation), the server serves (result)

- Widely used in many systems
  - e.g., DNS, Web-servers

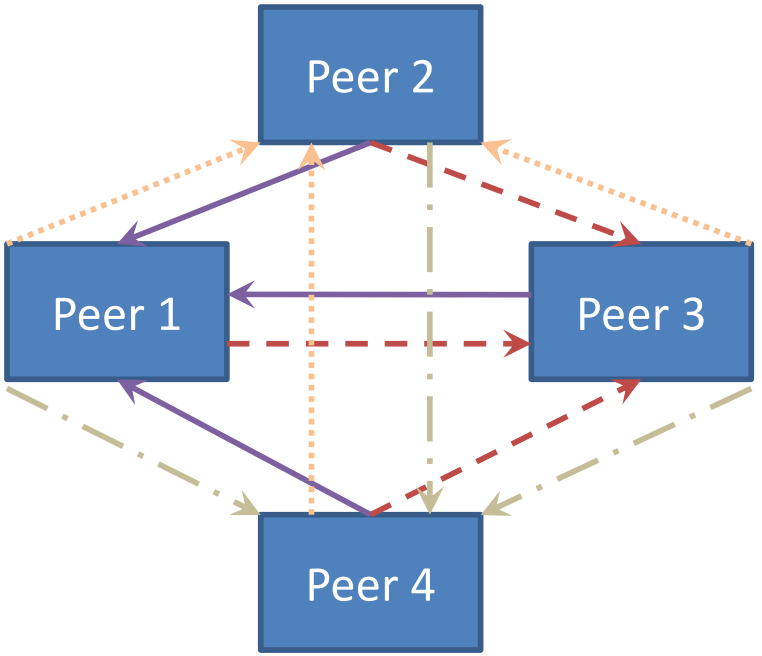


# Client-Server Architecture: Pros and Cons

- Advantages?
  - Simplicity and centralized control
  - Computation-heavy processing can be offloaded to a powerful server, Clients can be “thin”
- Disadvantages?
  - Single-point of failure at server
  - Scalability

# Peer to Peer (P2P) Architecture

- In P2P, roles of all entities are identical
  - All nodes are peers
  - Peers are equally privileged participants in the application
- Examples
  - Napster, Bittorrent, Skype, Spotify, …
    (see also next slides)




Peer-To-Peer Model

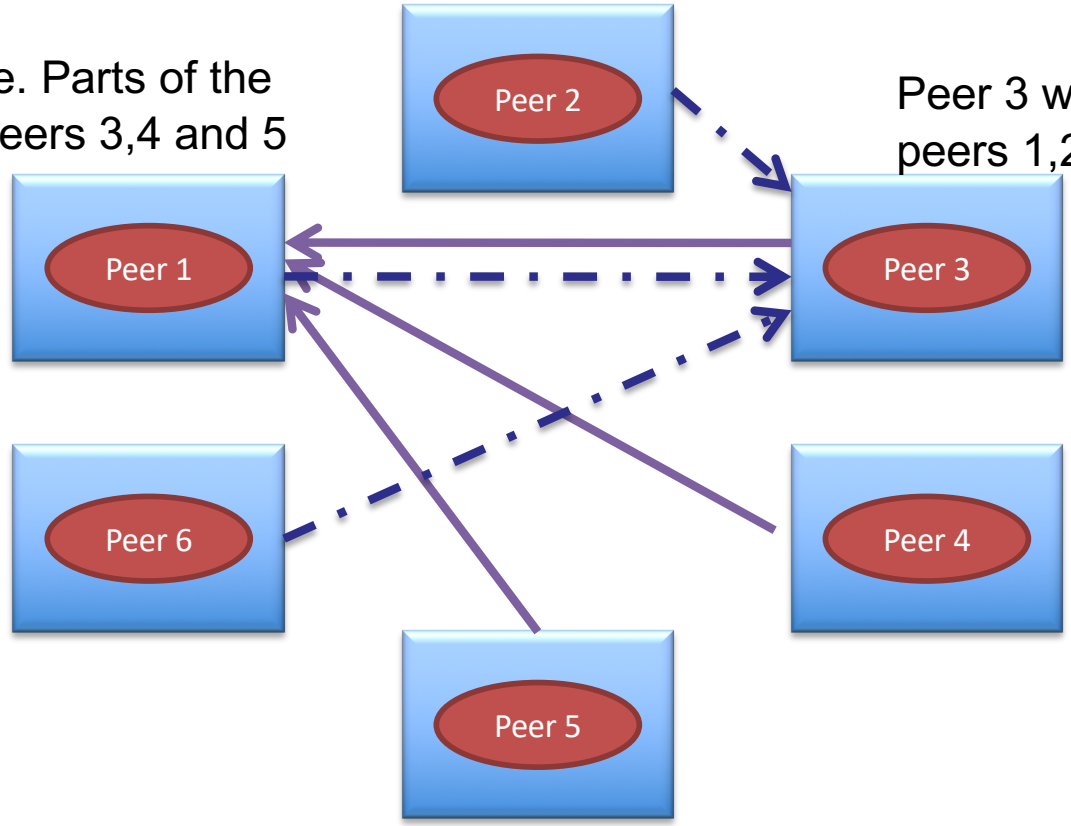
# Peer to Peer (P2P) Architecture

- Advantages
  - Scalable design
  - Limited cost for service provider
- Disadvantages
  - Limited control
  - Free riding
  - Churn: nodes coming and going
  - Bandwidth
  - ...
- For this reason:
  - Skype, Spotify, etc. moved away from P2P model
  - Used only during the beginning of the company
    - When they had limited financial ressources

# Peer to Peer Architecture

## Example: Downloading files from BitTorrent

Peer 1 wants a file. Parts of the file is present at peers 3,4 and 5




Peer 3 wants a file stored at peers 1,2 and 6

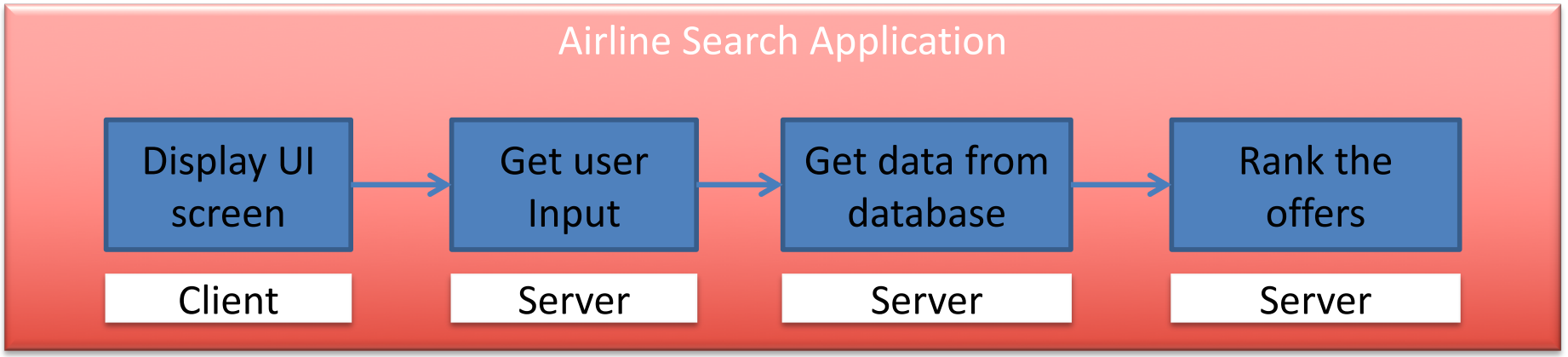
# Architectural Patterns

- Primitive architectural elements can be combined to form various patterns
  - Tiered Architecture
  - Layering

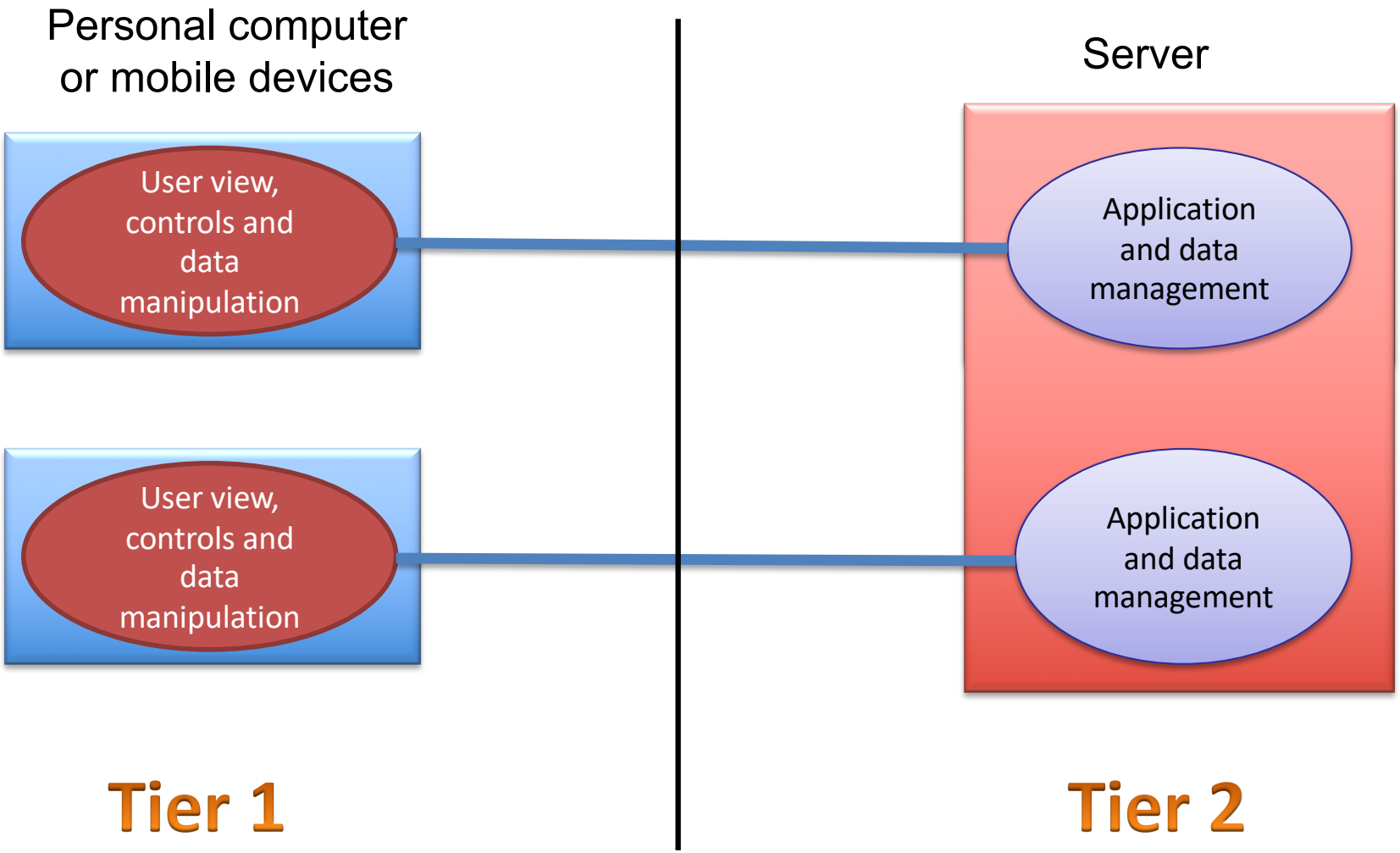
- Tiered architecture and layering are complementary
  - Layering = horizontal splitting of services
  - Tiered Architecture = vertical splitting of services

# Tiered Architecture

- A technique to:
  1. Organize the functionality of a service, and
  2. Place the functionality into appropriate servers

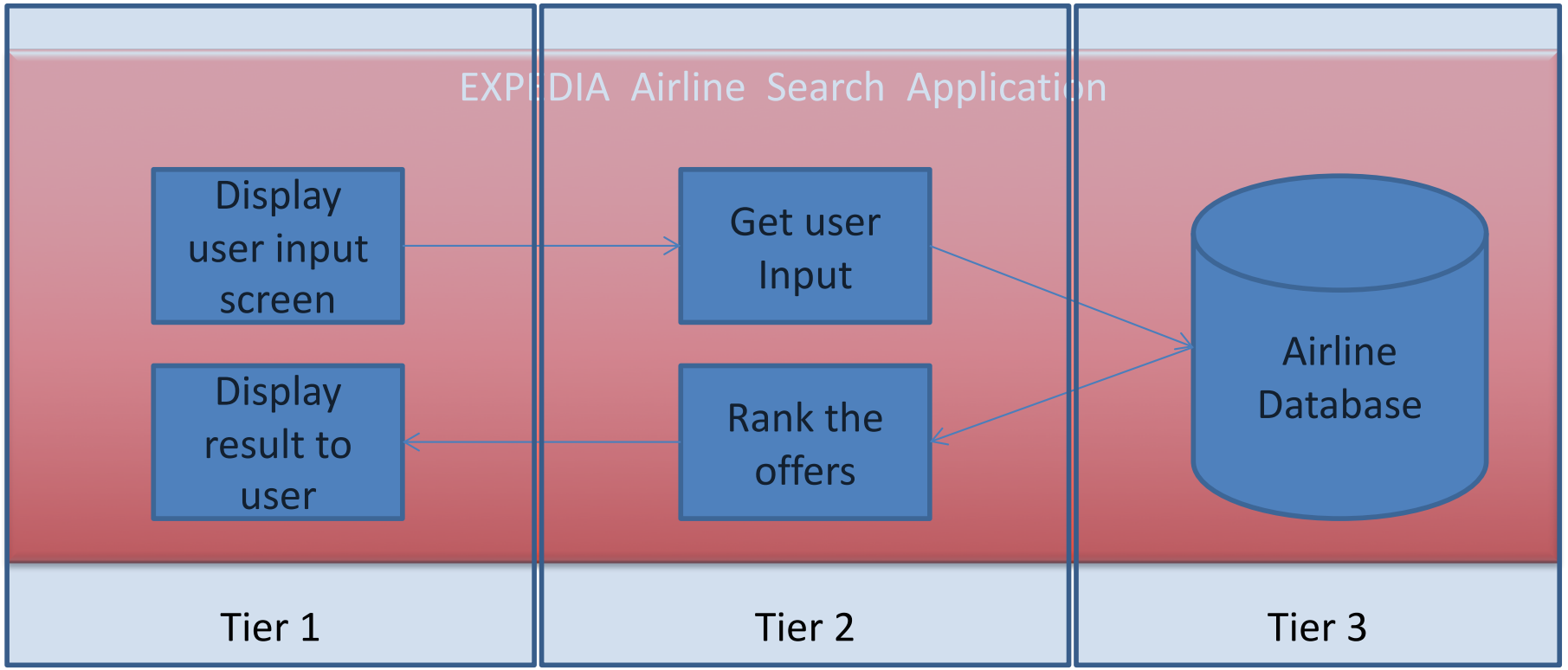


# A Two-Tiered Architecture



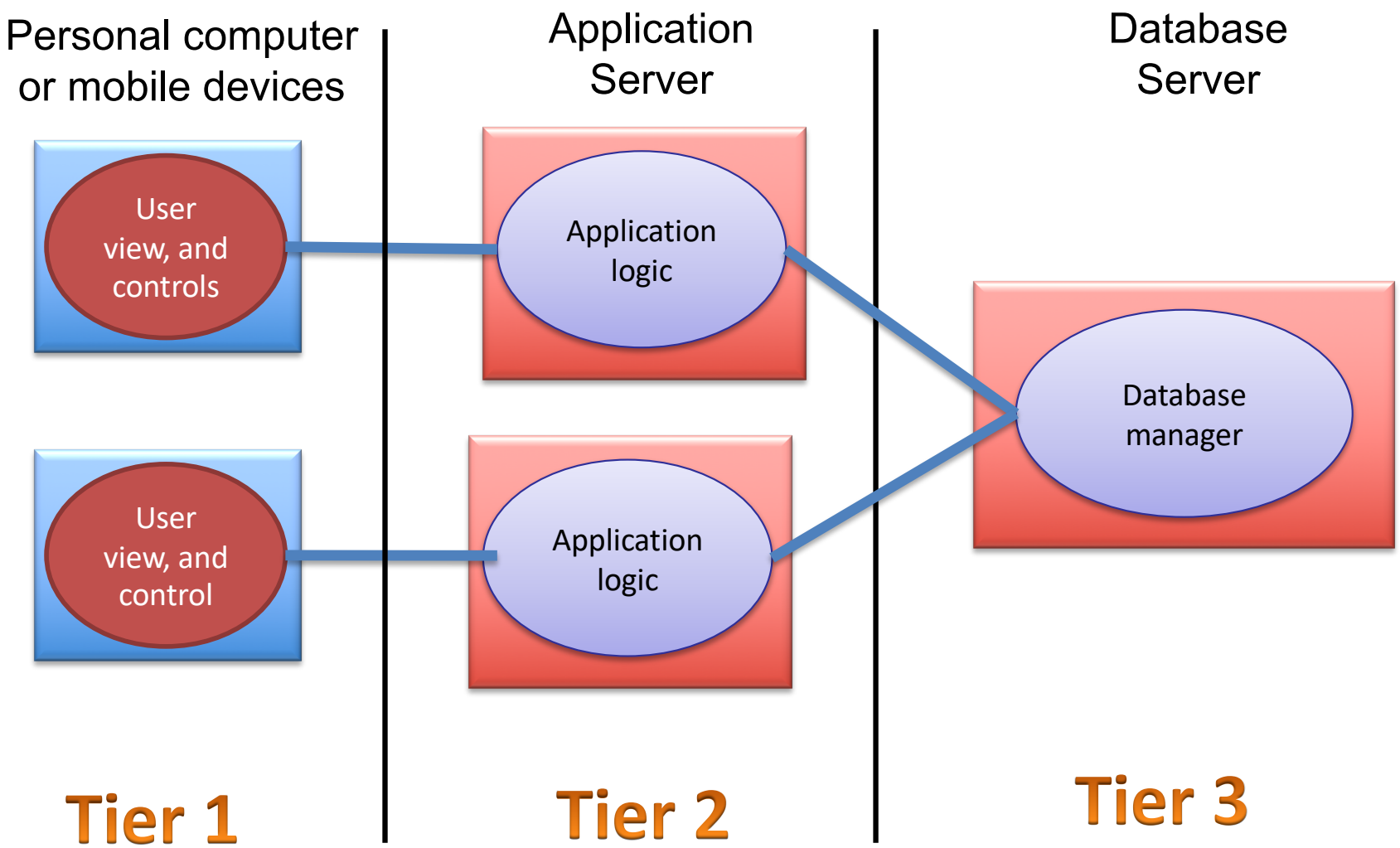
# A Three-Tiered Architecture

- How do you design an airline search application:



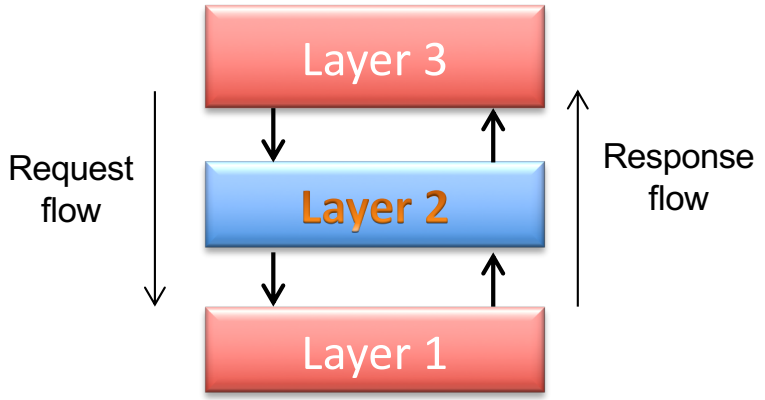

  - Organize functionality of a given layer

# A Three-Tiered Architecture



# Layering

- A complex system is partitioned into layers
  - Upper layer utilizes the services of the lower layer
  - A horizontal organization of services
- Layering simplifies design of complex distributed systems by hiding the complexity of below layers
- Control flows from layer to layer



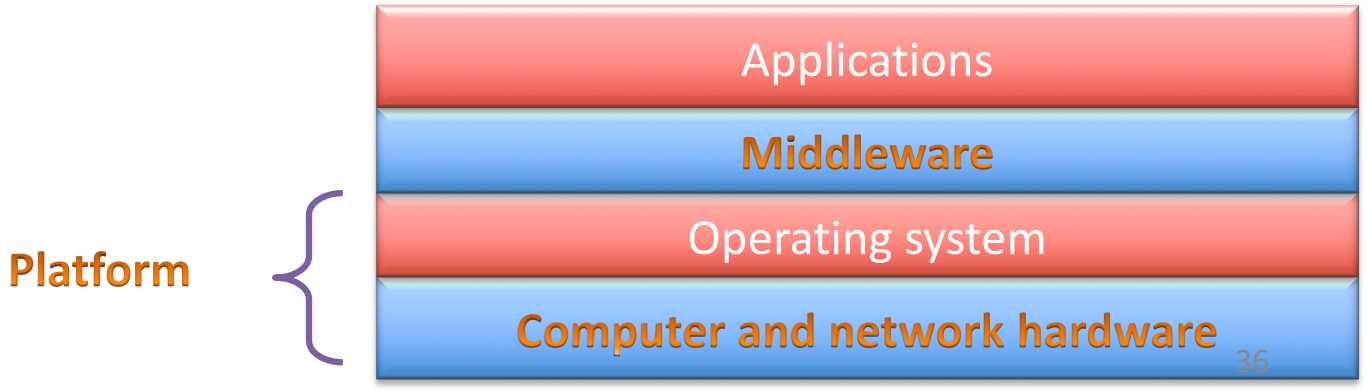
# Layering – Platform and middleware

Distributed Systems can be organized into three layers

1. Platform
   - Low-level hardware and software layers
   - Provides common services for higher layers

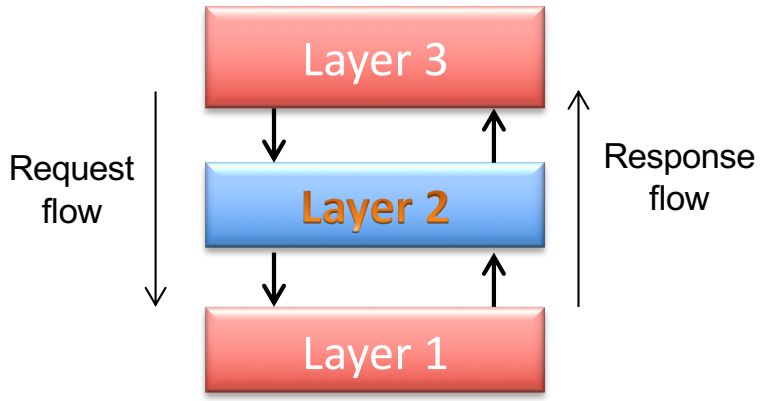
2. Middleware
   - Mask heterogeneity and provide convenient programming models to application programmers
   - Typically, it simplifies application programming by abstracting communication mechanisms

3. Applications

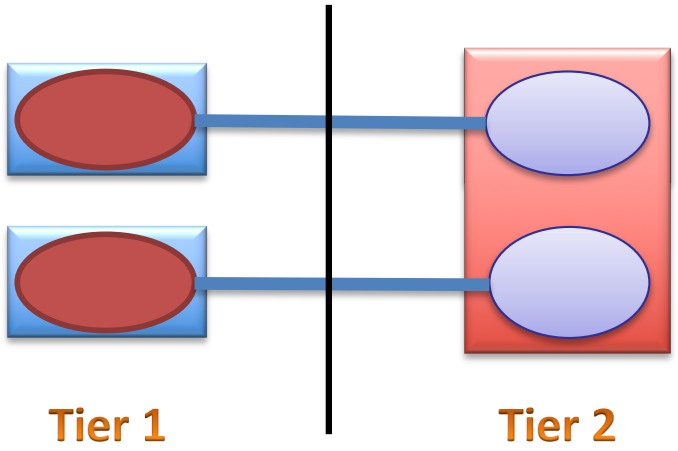


# Layering vs. Tiers

- Layers: Horizontal




- Tiers: Vertical




Most large-scale DS utilize a combination of both

# Classification of Distributed Systems

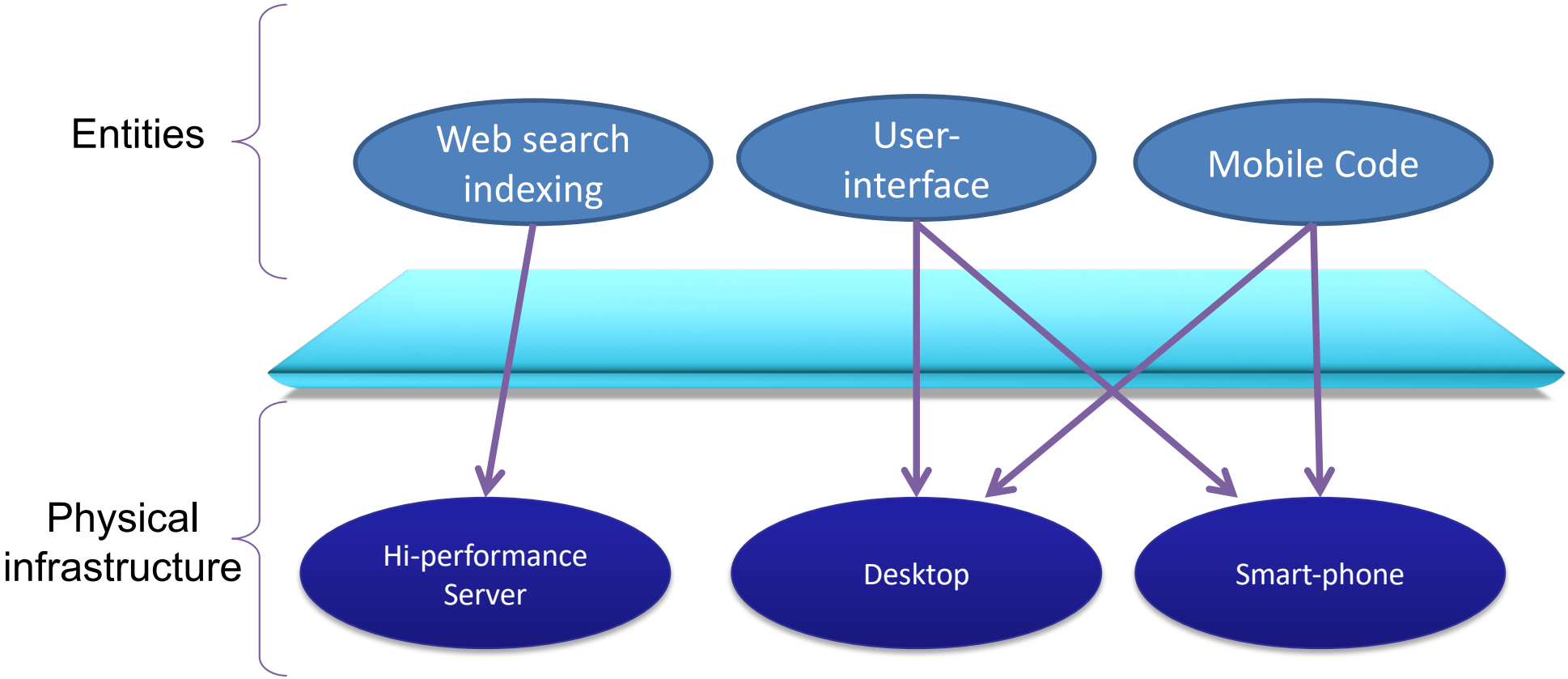
- What are the entities that are communicating in a DS?
  a) Communicating entities
- How do the entities communicate?
  b) Communication paradigms
- What roles and responsibilities do they have?
  c) Roles and responsibilities
- How are they mapped to the physical distributed infrastructure?
  d) Placement of entities

# Placement

- Observation:
  - A large number of heterogonous hardware (machines, network).
  - Smart mapping of entities (processes, objects) to hardware helps performance, security and fault-tolerance.

- “Placement” maps entities to underlying physical distributed infrastructure.
  - •Placement should be decided after a careful study of application characteristics
  - •Example strategies:
  - •Mapping services to multiple servers
  - •Moving the mobile code to the client

# Placement


Entities
Web search indexing
User-interface
Mobile Code
Physical infrastructure
Hi-performance Server
Desktop
Smart-phone

# Placement

- Why are there data centers all over the planet
  - Placement!
  - Goal:
    - Place data and service close to the user
    - Better performance (delay, bandwidth, …)
- Example
  - Videos that go “viral”
  - Where to place them?
  - Data travels with the daytime
    - From Europe to the US to Asia to …

# Recap Architecture

- So far, we have covered primitive architectural elements
  - Communicating entities
  - Communication paradigms of entities
    - Middleware, Indirect Communication
  - Roles and responsibilities that entities assume, and resulting architectures
    - Client-Server, Peer-to-Peer, Hybrid
  - Placement of entities

Part II

# Processes & Concurrency

# Concurrency

Example

```
int y = 0, z = 0;

thread A {
  x = y + z;
}
thread B {
  y = 1; z = 2;
}
```

- OS Scheduler
  - can interrupt a thread at any point in time
  - and schedule another thread

# Concurrency

- Possible results for x?

Example

```
int y = 0, z = 0;

thread A {
  x = y + z;
}
thread B {
  y = 1; z = 2;
}
```

http://olafland.polldaddy.com/s/concurrency

# Concurrency

Example

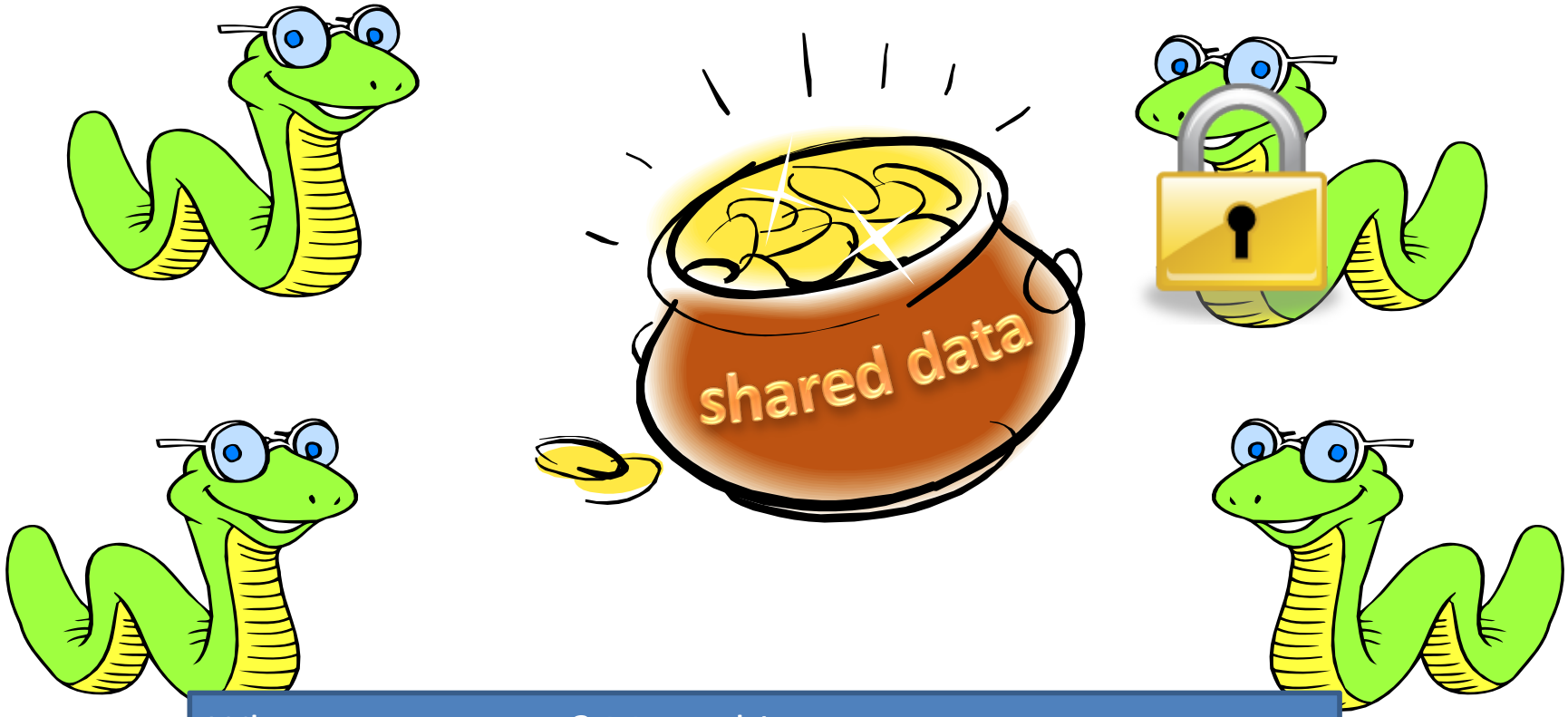
```
int y = 0, z = 0;

thread A {
  x = y + z;
}
thread B {
  y = 1; z = 2;
}
```

- Possible results for x?
  - 0, 1, 2, 3
- What causes this problem?
  - Concurrency
- How to fix?
  - Use locks

# Thread Synchronization

If lock is not available, threads wait
-> Execution becomes serialized (one after another gets access)

# Thread Synchronization




What can go wrong? Many things..

- We forget to release a lock (bug)
- What happens when the process holding a lock crashes?
- ….

# But

- In distributed settings
  - Multiple machine or processes

Locks alone are not always sufficient!

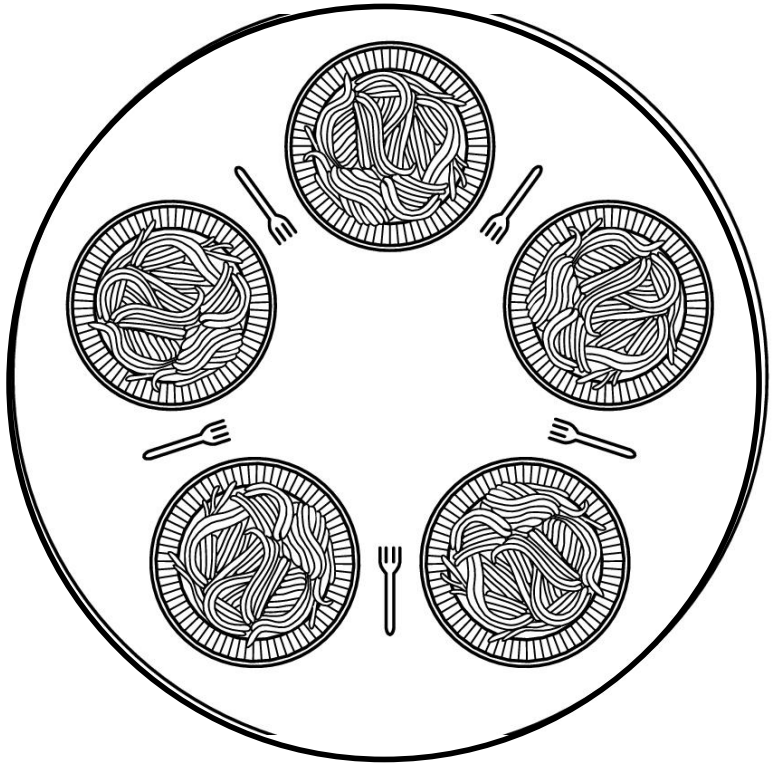
# Dining Philosophers

- A problem that was invented to illustrate a access to shared resources
- Our focus here is on the notion of sharing resources that *only one user at a time* can own
  - Such as a keyboard on a machine with many processes active at the same time
  - Or a special disk file that only one can write at a time (bounded buffer is an instance)

# Dining Philosopher's Problem

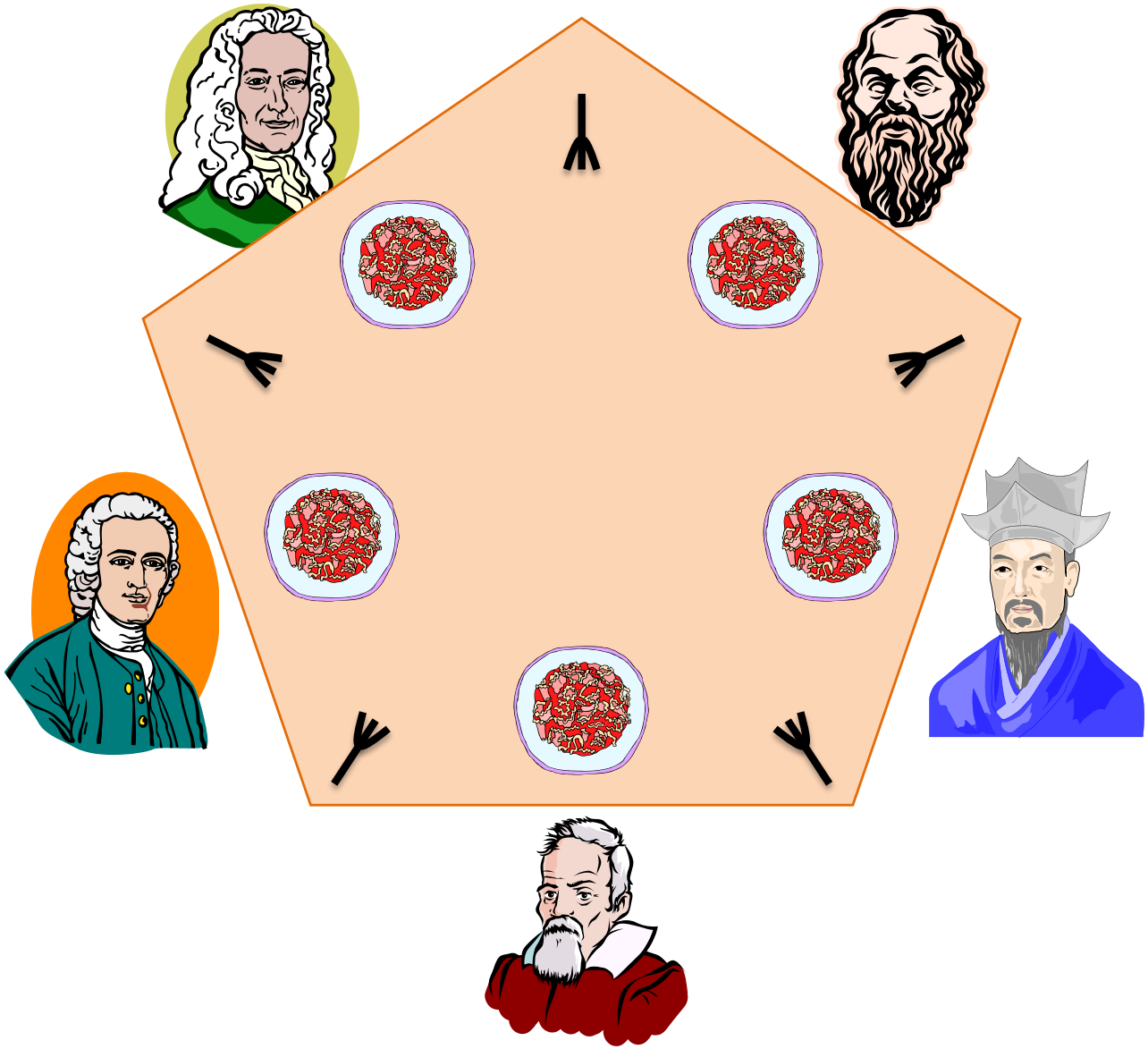
- Invented by Dijkstra in 1965
  - As exam question ;-)

- Philosophers eat/think
- N Philosophers, N forks
- Eating needs two forks
- Pick one fork at a time



Idea is to capture the concept of multiple processes competing for limited resources

# Dining Philosopher's Problem



Ok!

Not good!

# Rules of the Game

- The philosophers are very logical
  - They want to settle on a shared policy that all can apply concurrently
  - They are hungry: the policy should let everyone eat (eventually)
  - They are utterly dedicated to the proposition of equality: the policy should be totally fair

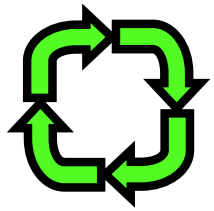
# Note

- We want fully distributed solution
  - Do not want philosophers to make arrangements with their neighbors
    - Cascade: these would also need to make arrangements with their neighbors
  - This would limit scalability and make things complex

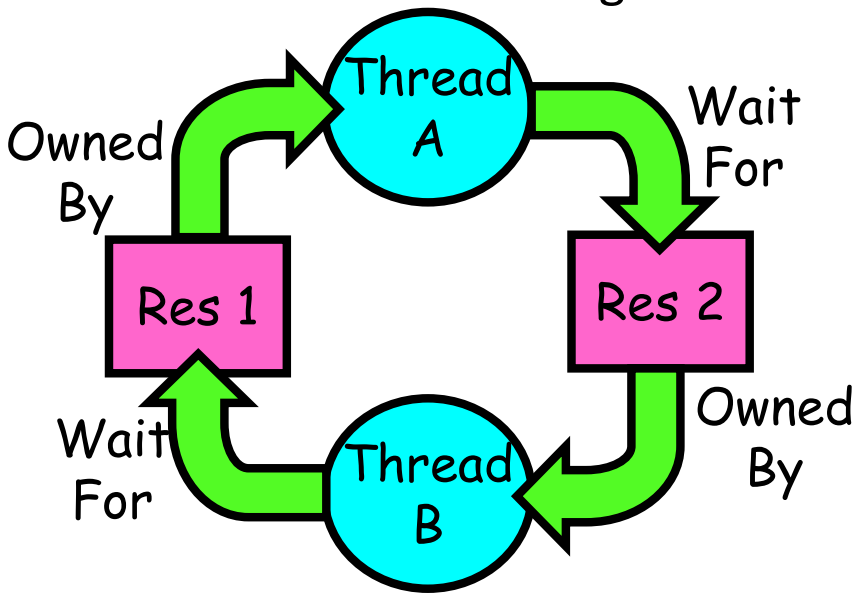
# What can go wrong?

- Primarily, we worry about?
  - Starvation: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)
  - Deadlock: A policy that leaves all the philosophers "stuck", so that nobody can do anything at all
  - Livelock: A policy that makes them all do something endlessly without ever eating!

# Starvation vs Deadlock



- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1



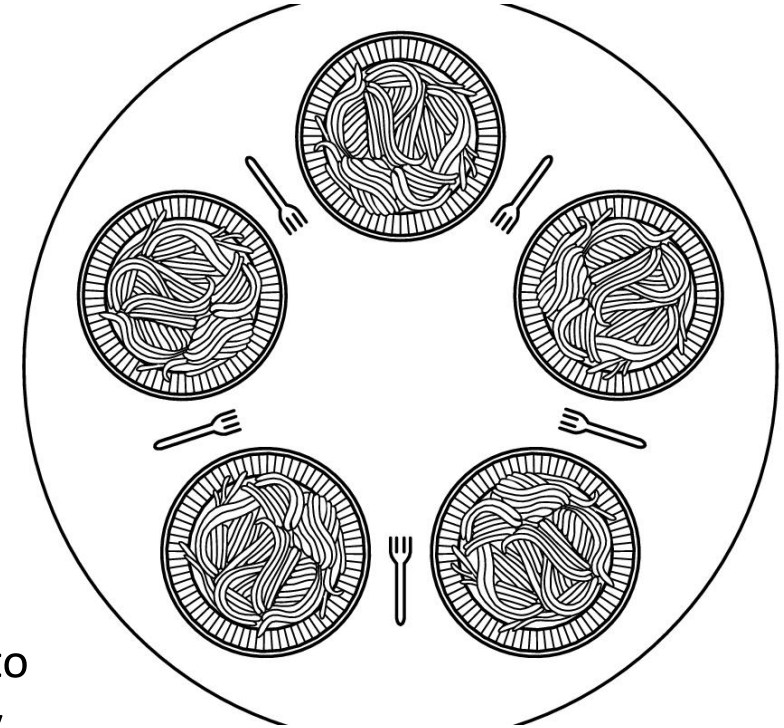

  - Deadlock ⇒ Starvation but not vice versa
    - Starvation can end (but doesn't have to)
    - Deadlock can't end without external intervention

# A flawed conceptual solution

```
# define N      5
```

Philosopher i (0, 1, .. 4)

```
do {
    think();
    take_fork(i);
    take_fork((i+1)%N);
    eat(); /* yummy */
    put_fork(i);
    put_fork((i+1)%N);
} while (true);
```



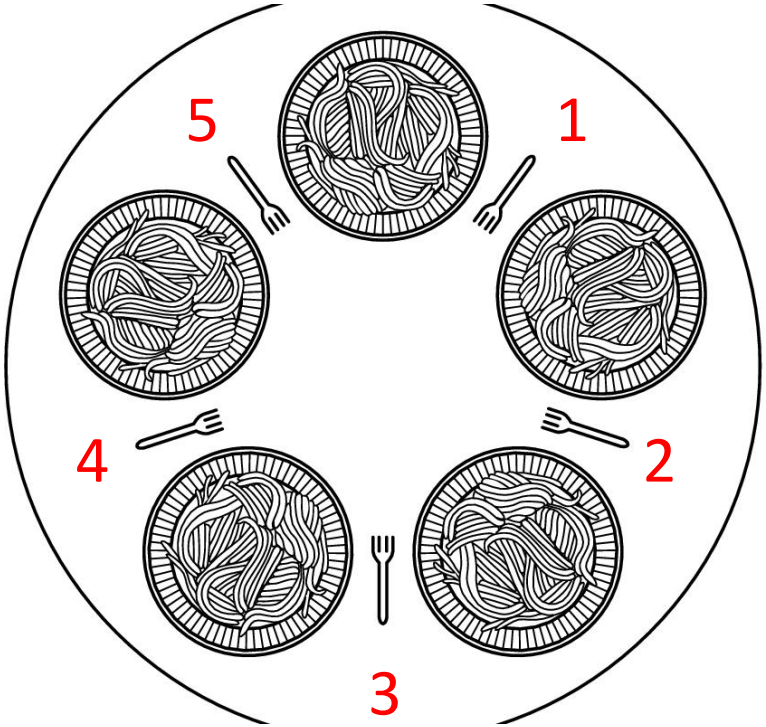
Oops! Subject to deadlock if they all pick up their "right" fork simultaneously!

# Solutions?

- Central entity: waiter
  - Assigns who can eat
  - Problem?
    - Does not scale, single point of failure
- Distributed
  - Naïve “Solution”:
    - Talking to determine who can eat
    - Requires global state (does not scale)
  - Solution 1: Assign order to resources (see next slides)
  - Solution 2: Probabilistic (see next slides)
    - Add randomization

# Solution 1: Partial Order

- Assign order to forks: 1 to n
  - Always try to pick the resource with the lower ID first
  - Avoids deadlocks, starvation, …
  - But…
    - Not always practicable to number resources



# Solution 2: Probabilistic Solution

- Assume Random "coin toss"
  - Guaranteed with probability 1 to break symmetry
- Idea: Try to get one first
  - Then get other
  - If can't get other, put first down and try again
- But don't go for the same fork first every time

```
Think
trying = true
While trying
    s = random(left,right)
    Wait for fork s then take it
    If fork ~s available
        take it
        trying = false
    else
        drop fork s
Eat
drop s=random(left,right)
drop fork ~s
```

Lehman, D. and Michael O. Rabin. "On the advantages of free choice: A symmetric and fully distributed solution of the dining philosophers problem," Proceedings of the Eighth ACM Symposium on the Principles of Programming Languages. (1981)

# Recap Concurrency

- Concurrency
  - Once multiple entities operate on the same data
    - Processes, multiple machines, …
  - Use Locks
  - Avoid deadlocks, starvation, ...
- Concurrency is natural in distributed systems
  - See your labs ;-)
  - Important to understand concurrency on a single system
    - Before moving on to large-scale DS

# Next Time

- Naming

# Questions?