

# **DISTRIBUTED SYSTEMS**

## **CS6421**

# **DISTRIBUTED ARCHITECTURE**

Prof. Rozbeh HaghNazari

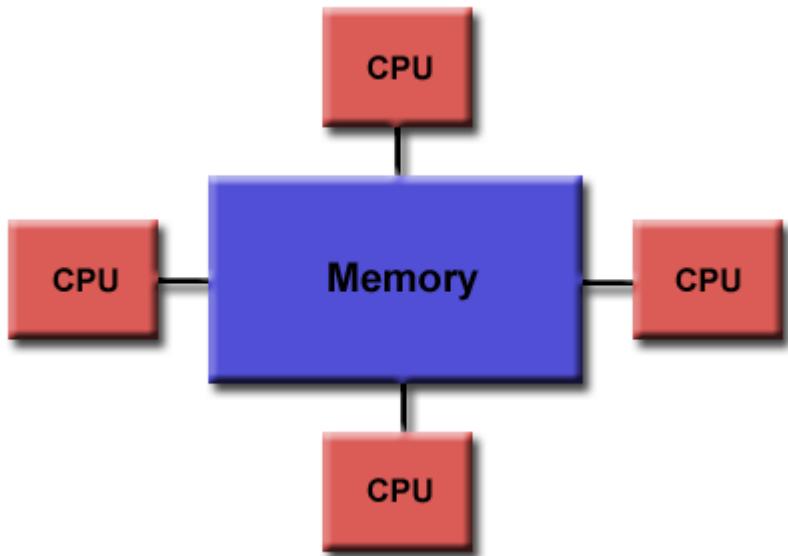
Slides Credit:

Prof. Tim Wood and Prof. Rozbeh HaghNazari

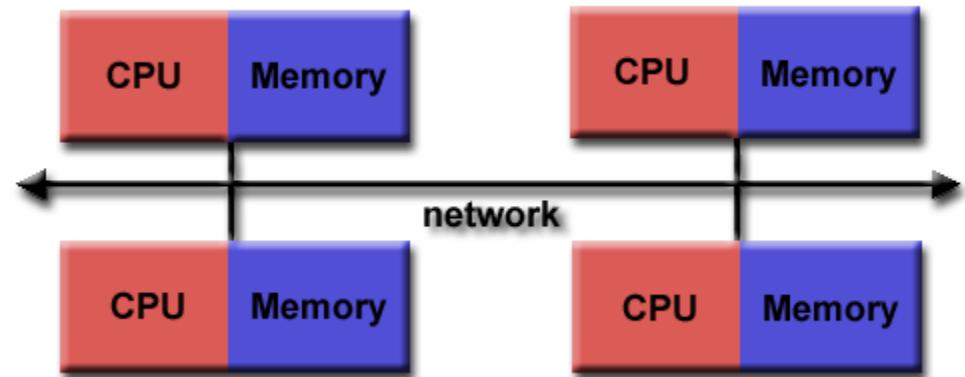
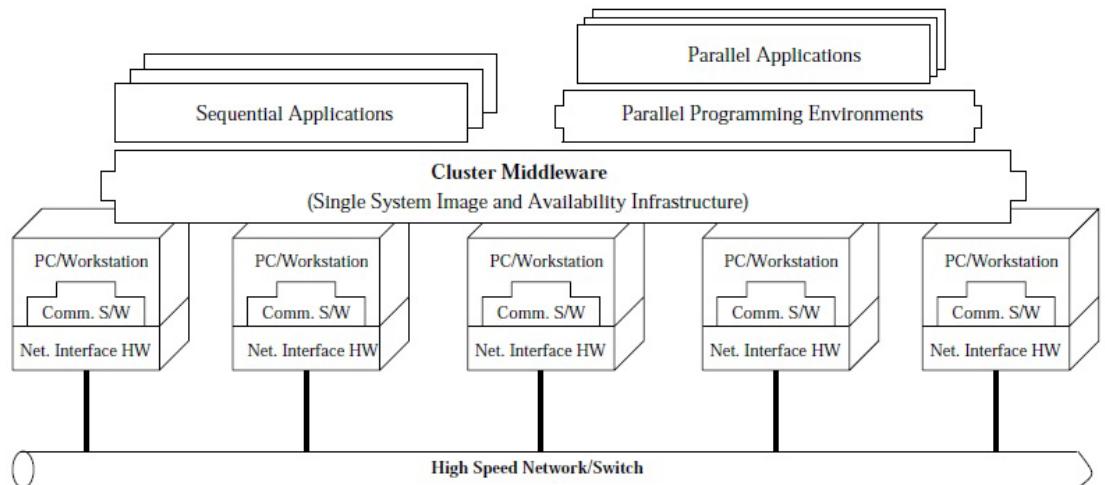
# TYPES OF DISTRIBUTED SYSTEMS

- Distributed Computing Systems
  - Clusters
  - Grids
- Distributed Information Systems
  - Transaction Processing Systems
  - Enterprise Application Integration
- Distributed Embedded Systems
  - Home systems
  - Health care systems
  - Sensor networks

# CLUSTER COMPUTING



Shared Memory: Uniform Memory Access Obtained from [wwwcomputing.llnl.gov](http://wwwcomputing.llnl.gov)



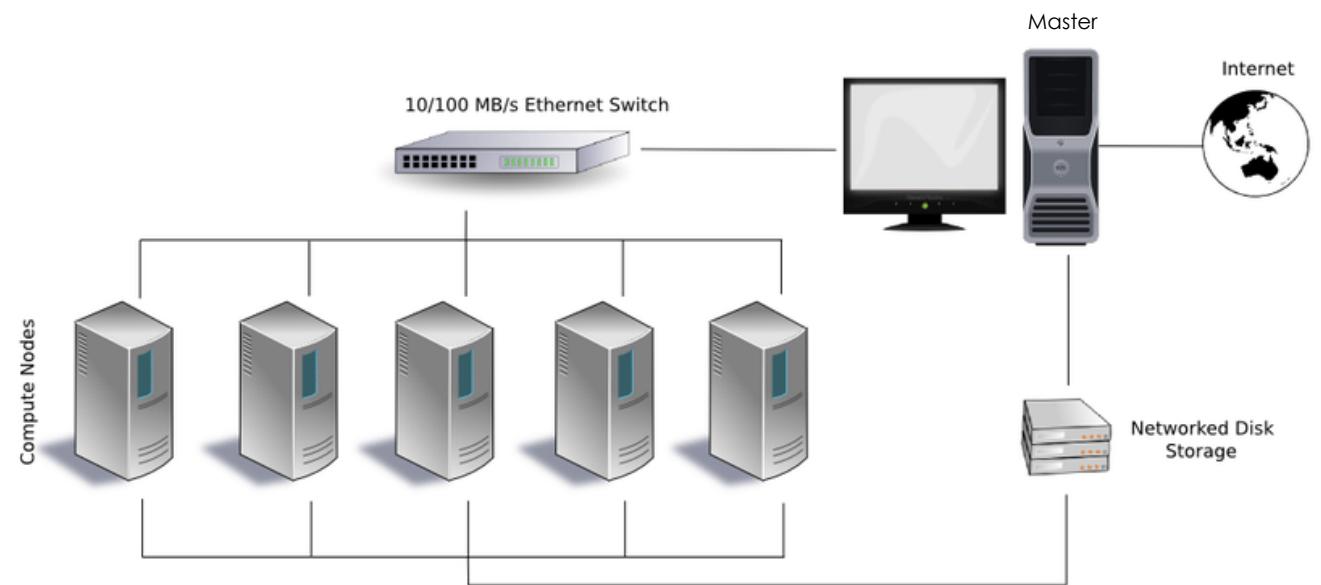
Distributed Memory System Obtained from [wwwcomputing.llnl.gov](http://wwwcomputing.llnl.gov)

# CLUSTERS CLASSIFICATIONS

- High Performance
- Expandability and Scalability
- High Throughput
- High Availability

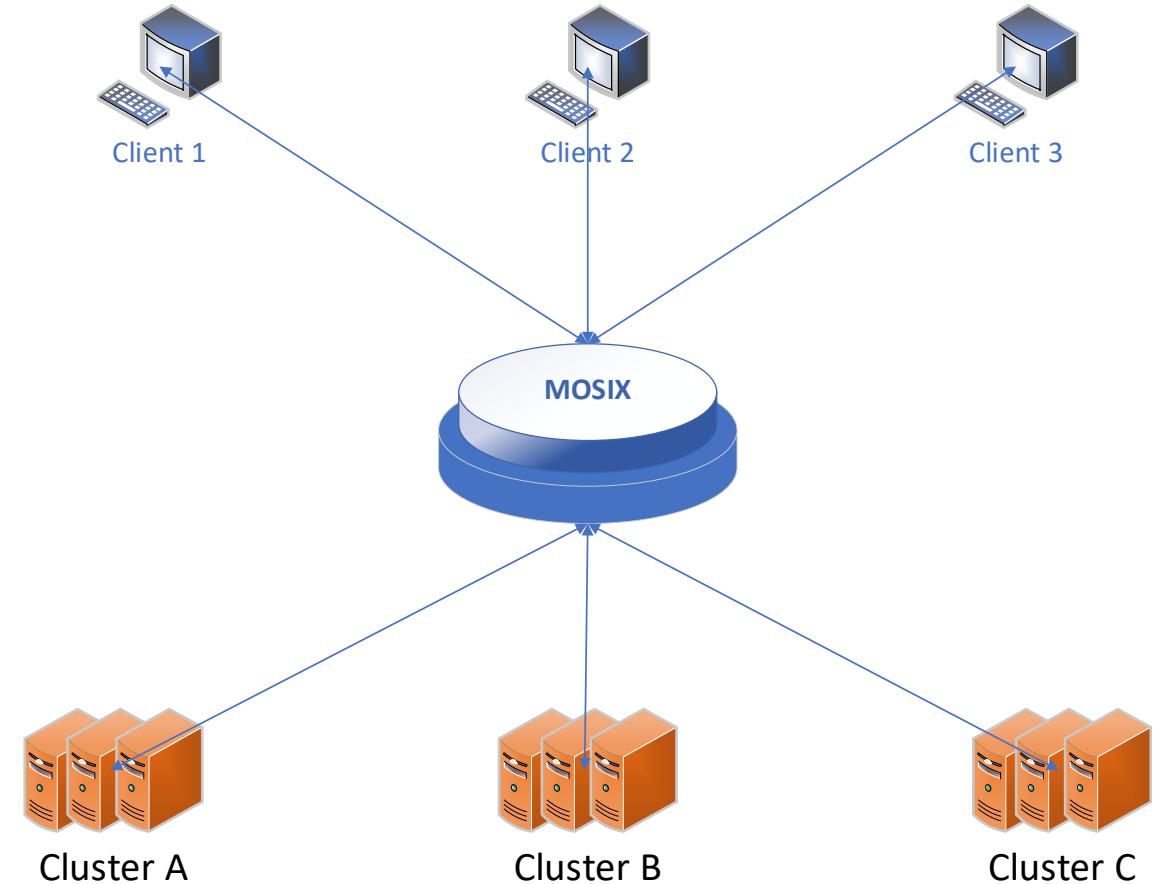
# CLUSTERS – BEOWULF MODEL

- Master-slave paradigm
  - One processor is the master; allocates tasks to other processors, maintains batch queue of submitted jobs, handles interface to users
  - Master has libraries to handle message-based communication or other features (the middleware).
- Proper for parallel programs



# CLUSTERS – MOSIX MODEL

- Provides a **symmetric**, rather than **hierarchical** paradigm
  - Single system image simplifies deployment
  - Processes can migrate between nodes dynamically
- “Operating-system-like”; looks & feels like a single computer with multiple processors
  - Provides resource discovery and automatic workload distribution among clusters



# GRID COMPUTING SYSTEMS



Highly heterogeneous with respect to hardware, software, networks, security policies, etc.



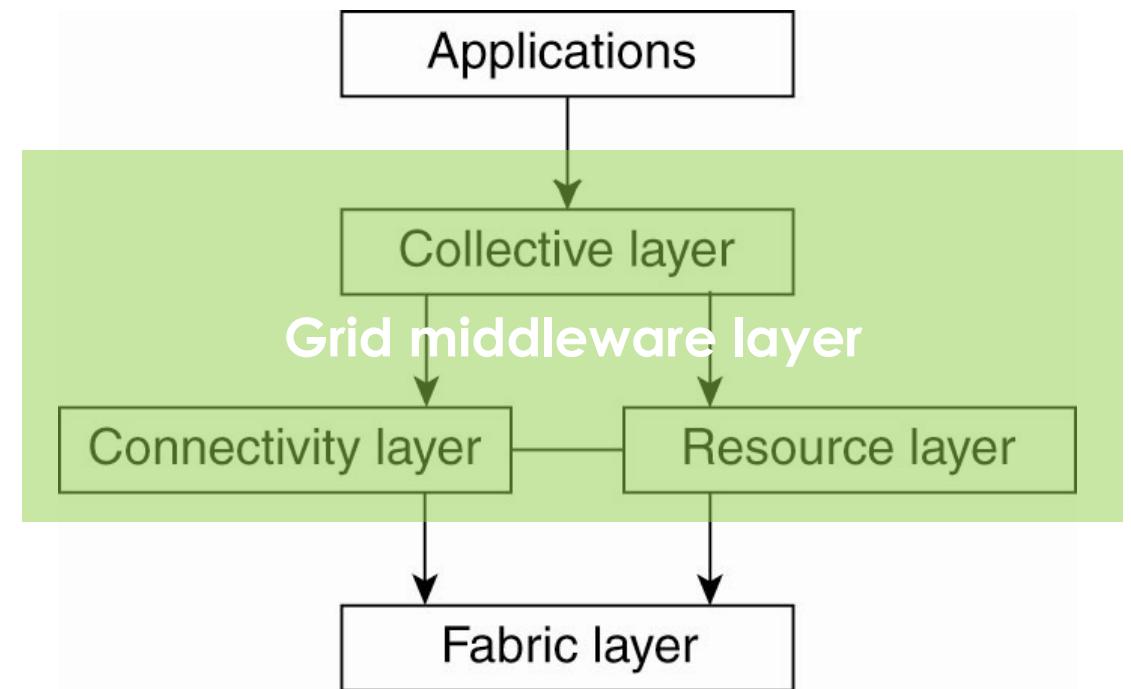
Grids support **virtual organizations**: a collaboration of users who pool resources (servers, storage, databases) and share them



Grid software is concerned with managing sharing across administrative domains.

# A PROPOSED ARCHITECTURE FOR GRID SYSTEMS

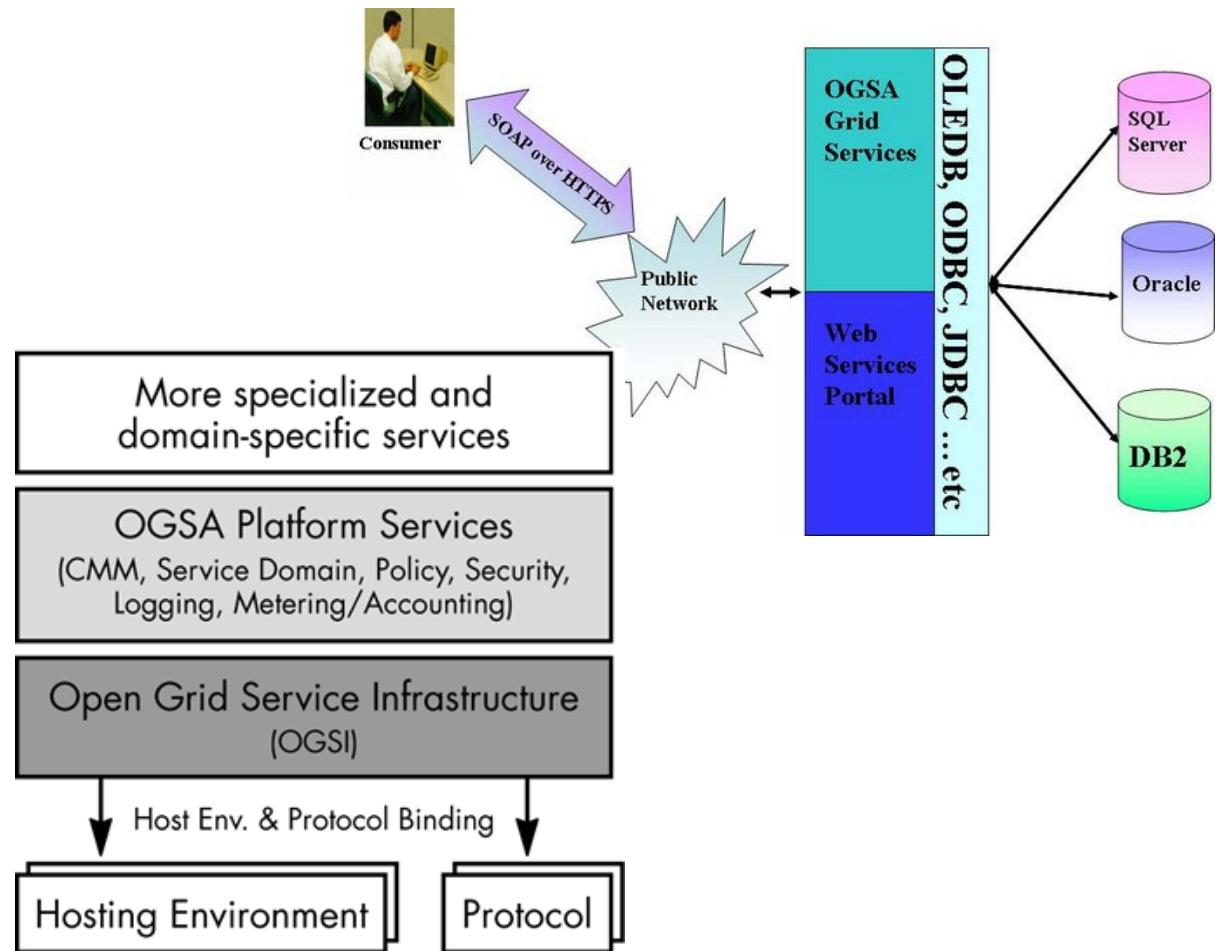
- **Fabric layer:** interfaces to local resources
- **Connectivity layer:** supports usage of *multiple* resources for a single application; e.g., access a remote resource or transfer data between sites
- **Resource layer** manages a *single* resource
- **Collective layer:** resource discovery, allocation, etc.
- **Applications:** use the grid resources
- The collective, connectivity and resource layers together form the middleware layer for a grid



. A layered architecture for grid computing systems

# OGSA – ANOTHER GRID ARCHITECTURE

- Open Grid Services Architecture (OGSA) is a service-oriented architecture
  - Sites that offer resources to share do so by offering specific Web services.
  - The architecture of the OGSA model is more complex than the previous layered model.



# TYPES OF DISTRIBUTED SYSTEMS

- Distributed Computing Systems
  - Clusters
  - Grids
- **Distributed Information Systems**
  - Transaction Processing Systems
  - Enterprise Application Integration
- Distributed Embedded Systems
  - Home systems
  - Health care systems
  - Sensor networks

# DISTRIBUTED INFORMATION SYSTEMS

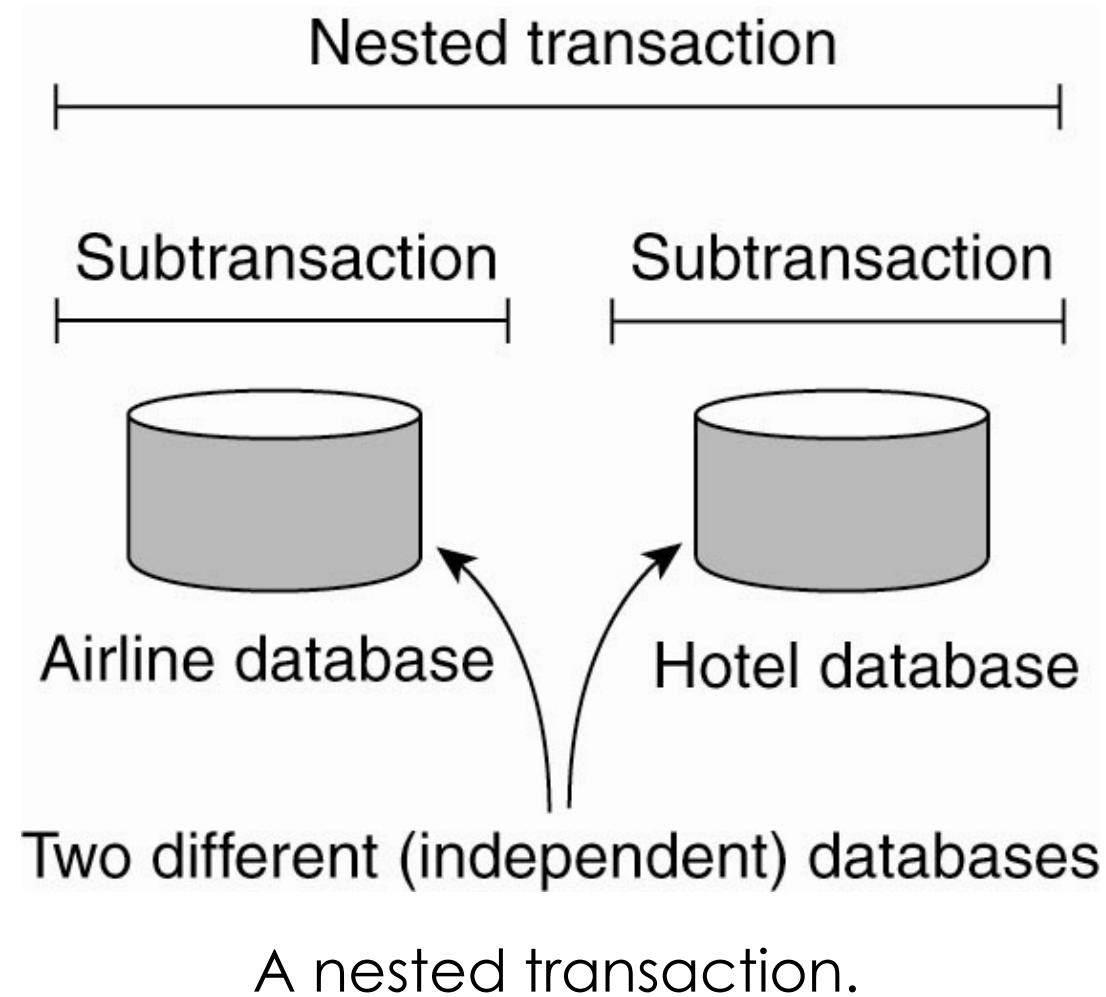
- Business-oriented
- Systems to make a number of separate network applications interoperable and build “enterprise-wide information systems”.
- Two types discussed here:
  - Transaction processing systems
  - Enterprise application integration

# TRANSACTION PROCESSING SYSTEMS

- Provide a highly structured client-server approach for database applications
- Transactions are the communication model
- Obey the ACID properties:
  - Atomic: all or nothing
  - Consistent: invariants are preserved
  - Isolated (serializable)
  - Durable: committed operations can't be undone

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

# TRANSACTION PROCESSING SYSTEMS



# IMPLEMENTING TRANSACTIONS

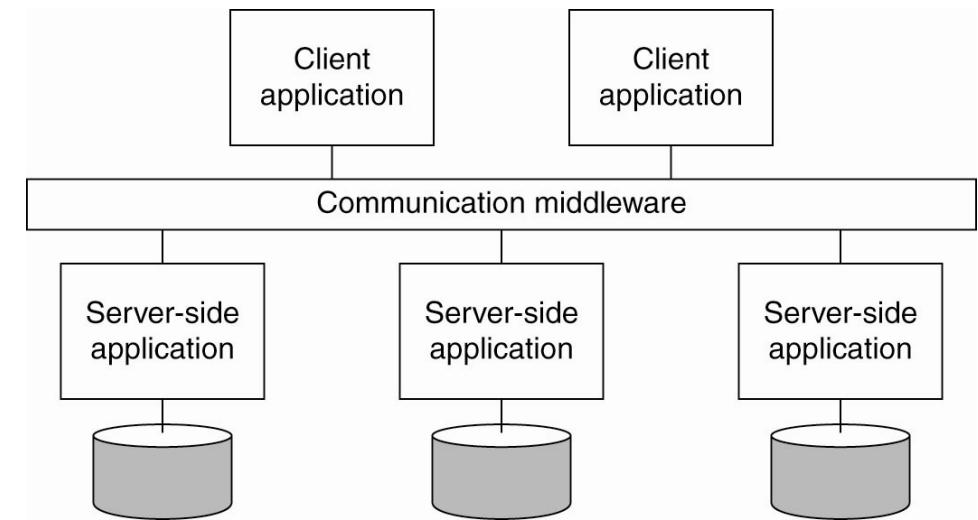
- Conceptually, private copy of all data
- Actually, usually based on logs
- Multiple sub-transactions – commit, abort
  - Durability is a characteristic of top-level transactions only
- Nested transactions are suitable for distributed systems
  - Transaction processing monitor may interface between client and multiple data bases.

# ENTERPRISE APPLICATION INTEGRATION

- Supports a less-structured approach (as compared to transaction-based systems)
- Application components are allowed to communicate directly
- Communication mechanisms to support this include CORBA, Remote Procedure Call (RPC), Remote Method Invocation (RMI), and Message-Oriented middleware (MOM).

## Examples?

Tell some software architectures that can be applied on this model



Middleware as a communication facilitator in enterprise application integration.

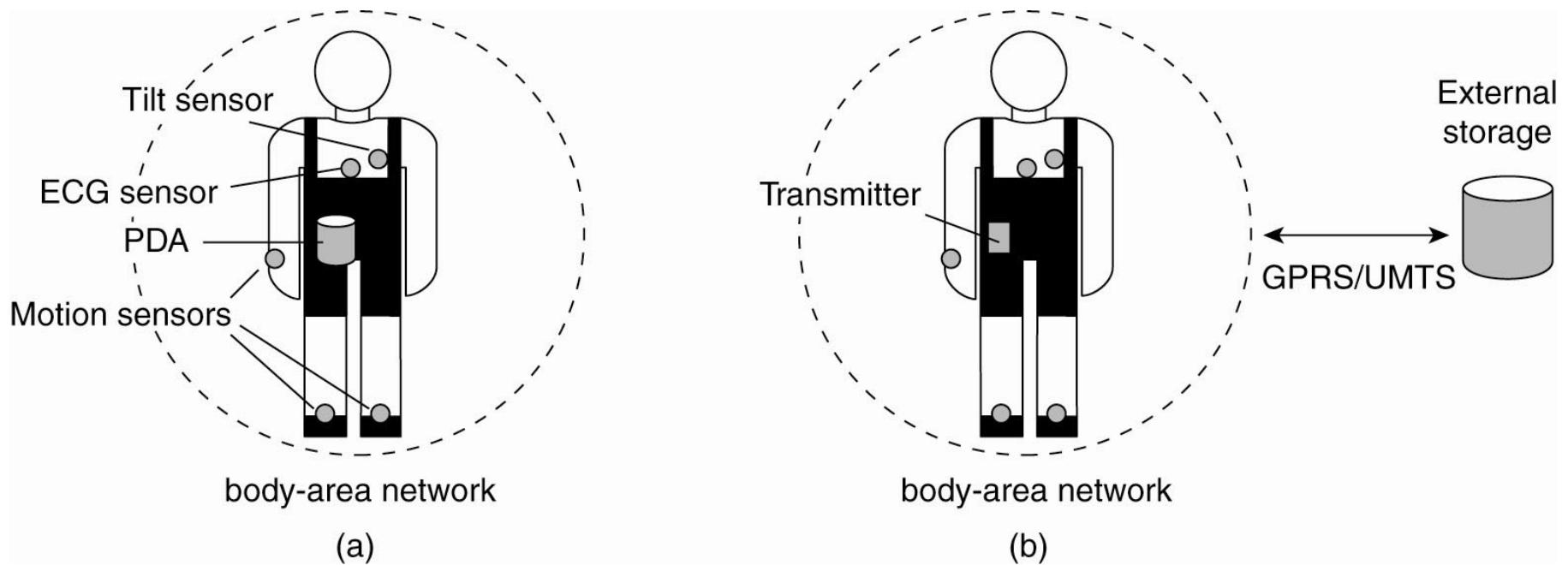
# TYPES OF DISTRIBUTED SYSTEMS

- Distributed Computing Systems
  - Clusters
  - Grids
- Distributed Information Systems
  - Transaction Processing Systems
  - Enterprise Application Integration
- **Distributed Embedded Systems**
  - Home systems
  - Health care systems
  - Sensor networks

# DISTRIBUTED PERVASIVE SYSTEMS

- The first two types of systems are characterized by their stability: nodes and network connections are more or less fixed
- This type of system is likely to incorporate small, battery-powered, mobile devices
  - Home systems
  - Electronic health care systems – patient monitoring
  - Sensor networks – data collection, surveillance

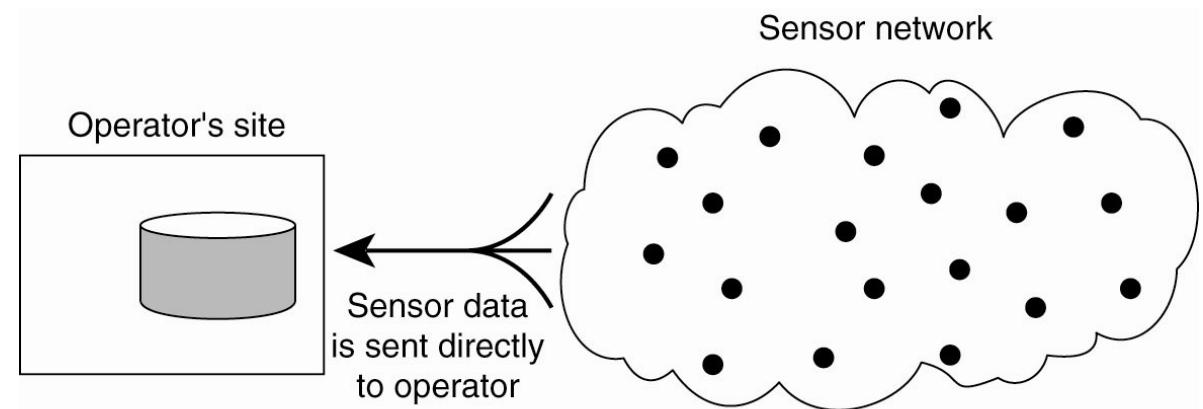
# ELECTRONIC HEALTH CARE SYSTEMS



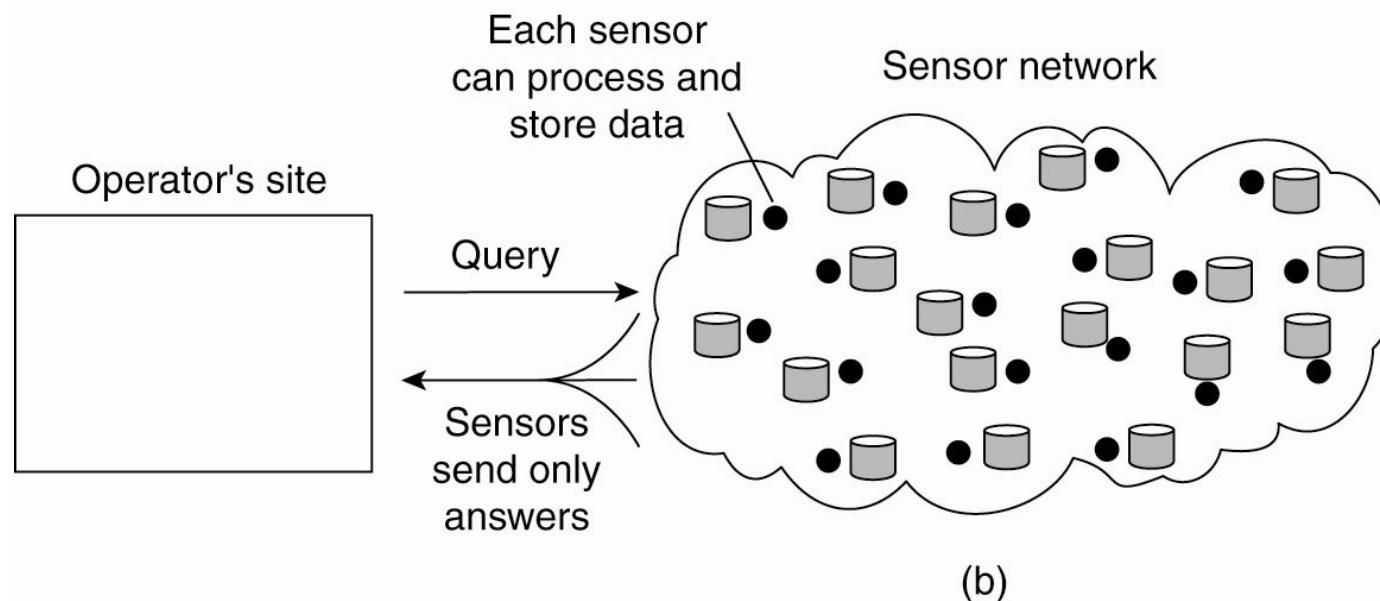
Monitoring a person in a pervasive electronic health care system, using (a) a local hub or (b) a continuous wireless connection.

# SENSOR NETWORKS

Organizing a sensor network database, while storing and processing data (a) only at the operator's site or (b) only at the sensors.



(a)



(b)

Prof. Tim Wood & Prof. Roozbeh Haghnazari

# ARCHITECTURES

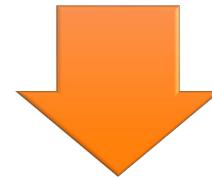


# DEFINITION OF *ARCHITECTURE*

- The art or science of building
  - *specifically*: the art or practice of designing and building structures and especially habitable ones
- Formation or construction resulting from or as if from a conscious act or a unifying or coherent form or structure
- A method or style of building
- The manner in which the components of a computer or computer system are organized and integrated

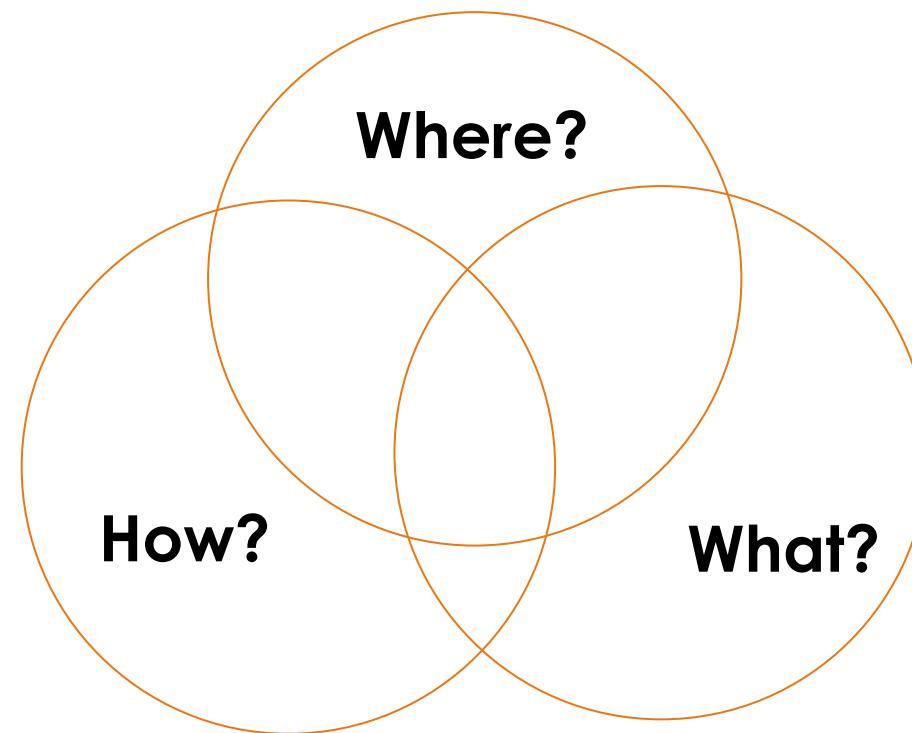
# SOFTWARE/SYSTEM ARCHITECTURE

**Software Architectures** – describe the organization and interaction of software components; focuses on logical organization of software (component interaction, etc.)

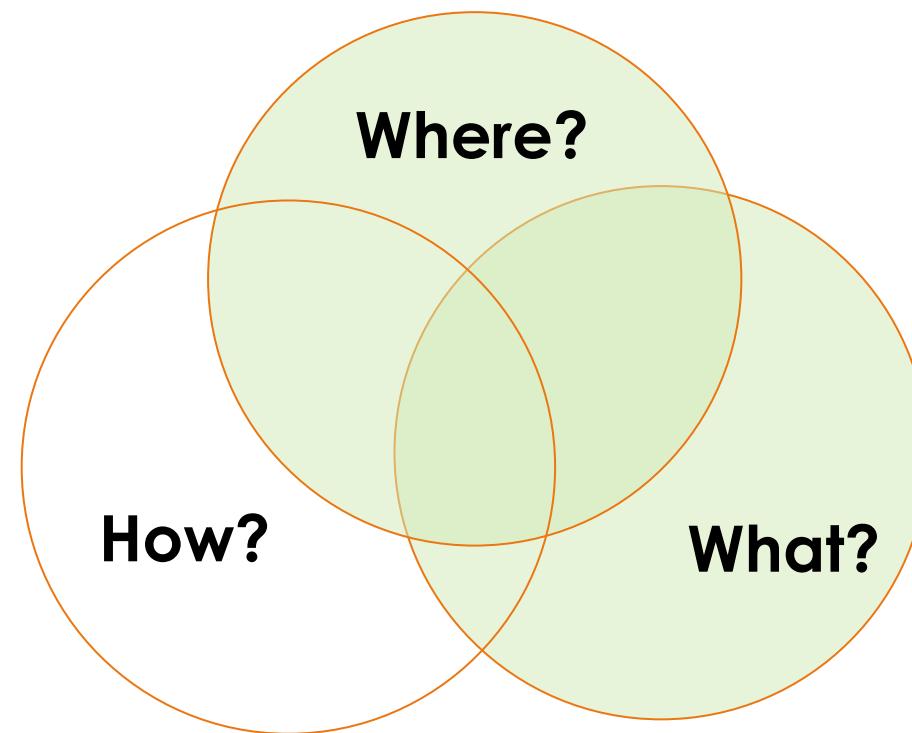


**System Architectures** - describe the communication and placement of software components on physical machines

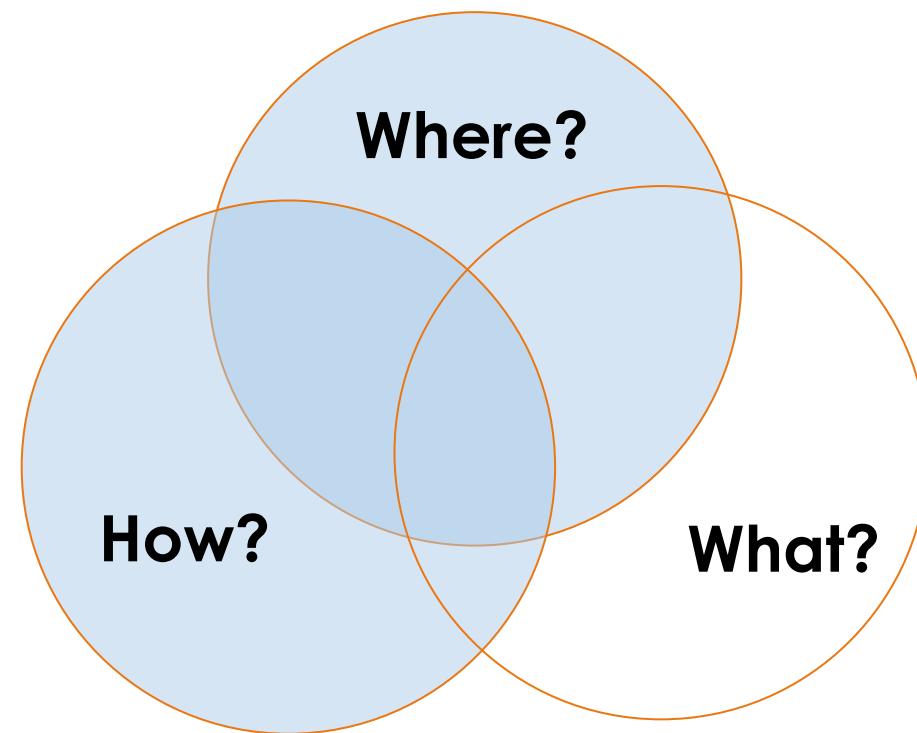
# ARCHITECTURE VS DESIGN



# ARCHITECTURE VS DESIGN



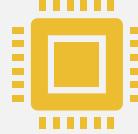
# ARCHITECTURE VS DESIGN



# COMPONENT



A component is an encapsulated part of a software system



A component has an interface



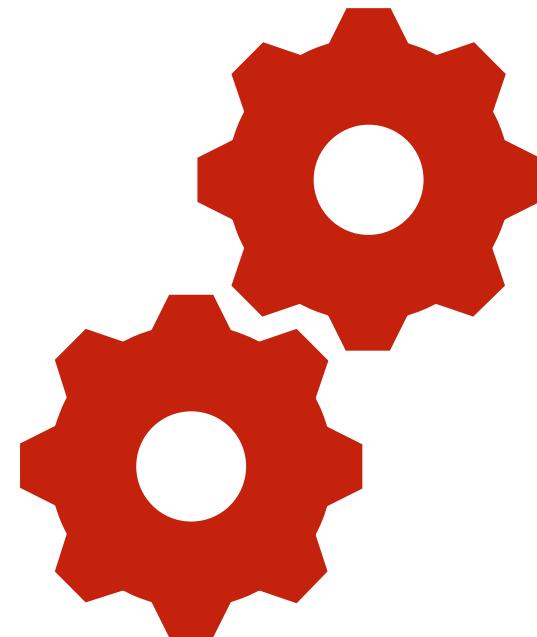
Components serve as the building blocks for the structure of a system



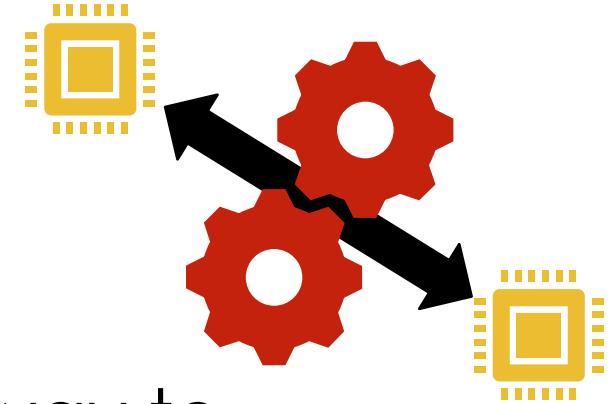
At a programming-language level, components may be represented as modules, classes, objects or as a set of related functions

# SUBSYSTEM

- A subsystem is a set of collaborating components performing a given task
- A subsystem is considered a separate entity within a software architecture
  - It performs its designated task by interacting with other subsystems and components...



# ARCHITECTURAL STYLES



- An **architectural style** describes a particular way to configure a collection of components and connectors.
  - **Component** - a module with well-defined interfaces; reusable, replaceable
  - **Connector** – communication link between modules
- An architectural style is a coordinated set of architectural **constraints** that restricts the relationships among those elements

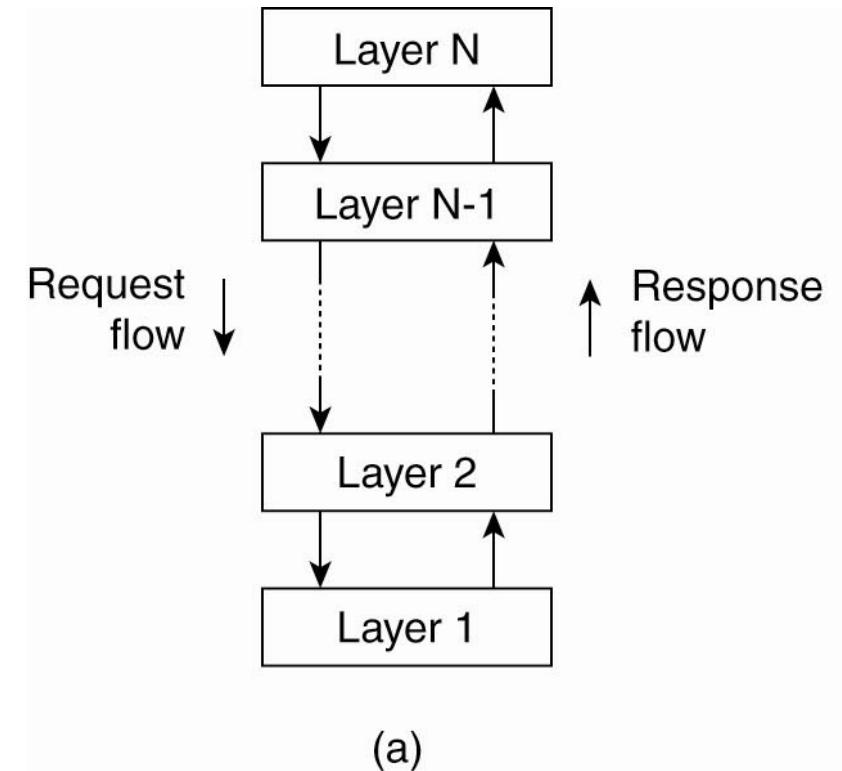
# ARCHITECTURAL STYLES

1. Layered architectures
2. Object-based architectures
3. Data-centered architectures
4. Event-based architectures

# 1. LAYERED ARCHITECTURES

- Components of layer  $N_i$  is only allowed to call components at the underlying layer  $N_{i-1}$

Why? Example?

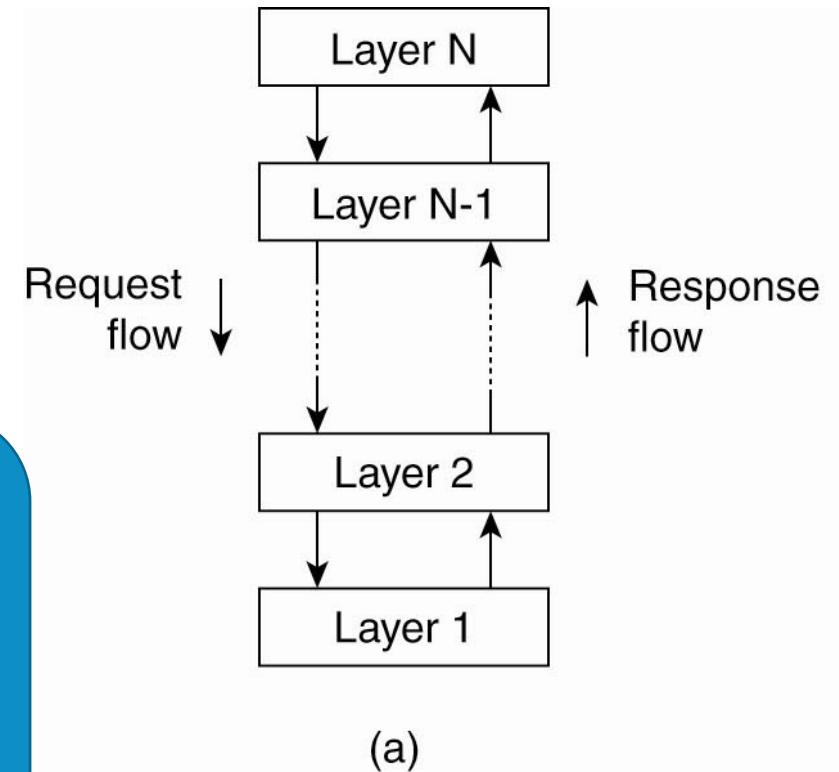


# 1. LAYERED ARCHITECTURES

- Components of layer  $N_i$  is only allowed to call components at the underlying layer  $N_{i-1}$

**Why:** hides information,  
interchangeable layers

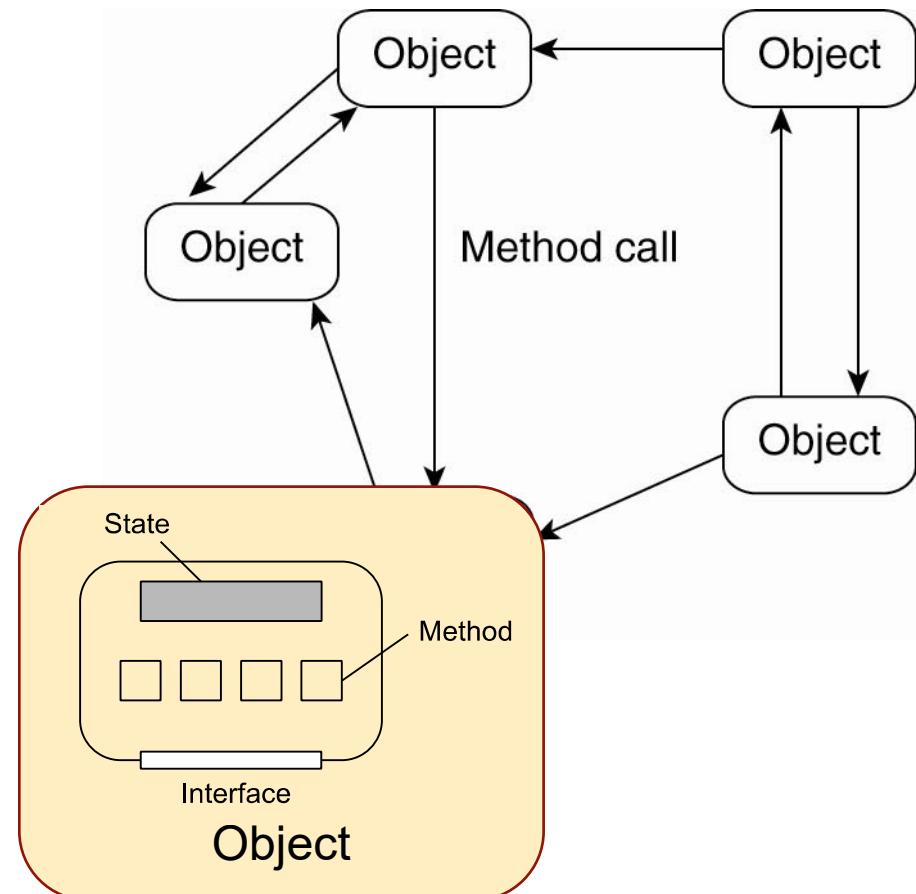
**Example:** Network stack, LAMP stack



## 2. OBJECT-BASED ARCHITECTURES

- Each object is a component that encapsulates data and methods
- They are connected through a well defined remote API that hides internals

**Why? Example?**

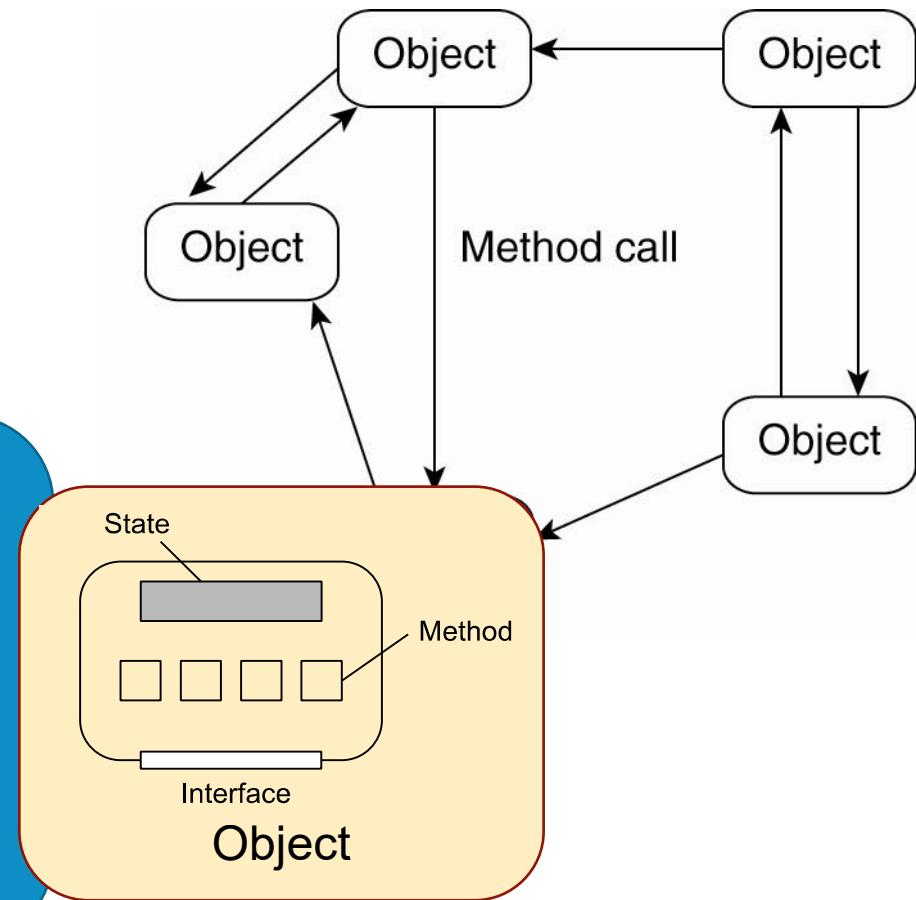


## 2. OBJECT-BASED ARCHITECTURES

- Each object is a component that encapsulates data and methods
- They are connected through a well defined remote API that hides internals

**Why:** components can be individually scaled/developed/managed

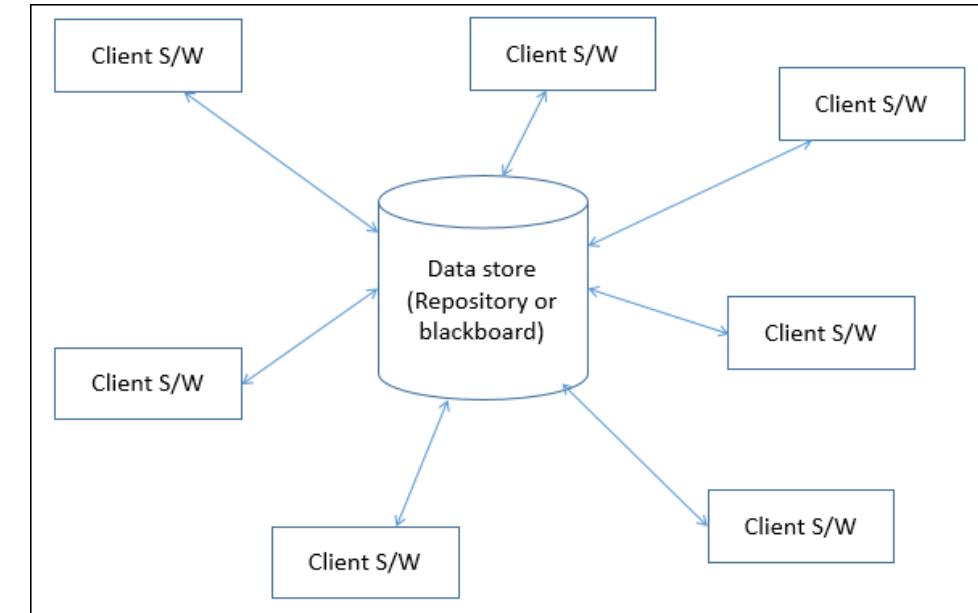
**Example:** MapReduce, microservice web architectures



# 3. DATA-CENTERED ARCHITECTURES

- Main purpose: data access and update
- Processes interact by reading and modifying data in a **centralized** shared repository

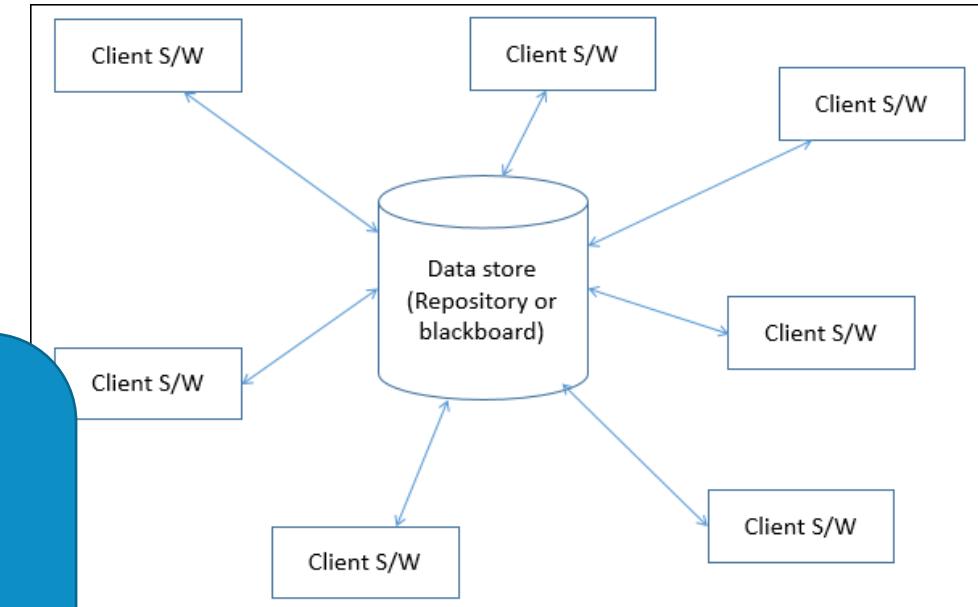
Why? Example?



### 3. DATA-CENTERED ARCHITECTURES

- Main purpose: data access and update
- Processes interact by reading and modifying data in a **centralized** shared repository

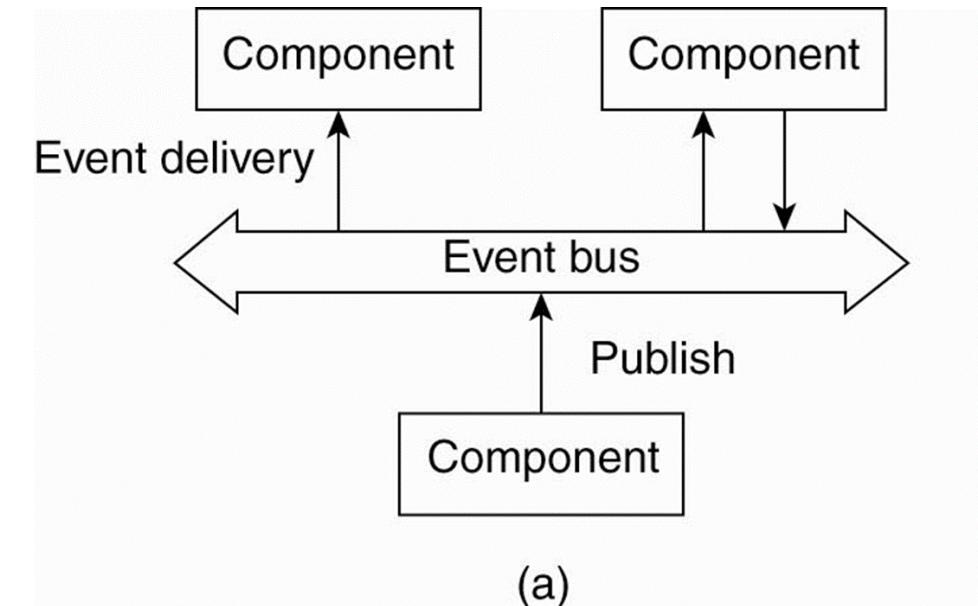
**Why:** simplifies data management  
**Example:** Dropbox, Message board systems, Email



# 4. EVENT-BASED ARCHITECTURES

- Communication via event propagation
  - Publish-subscribe
  - Broadcast
  - Point-to-point

**Why? Example?**

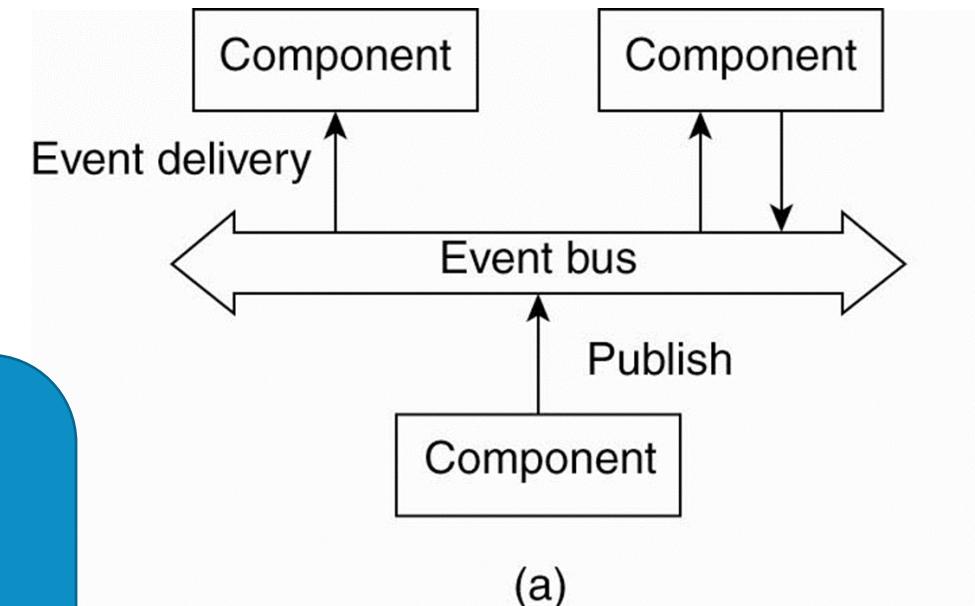


# 4. EVENT-BASED ARCHITECTURES

- Communication via event propagation
  - Publish-subscribe
  - Broadcast
  - Point-to-point

**Why:** decouples sender/receiver,  
asynchronous

**Example:** Slack, Security monitoring



# ARCHITECTURAL STYLES

1. Layered architectures
2. Object-based architectures
3. Data-centered architectures
4. Event-based architectures

Each style constrains how we will build the system. Following a style makes development and extensibility easier.

But sometimes we need a **hybrid** style!

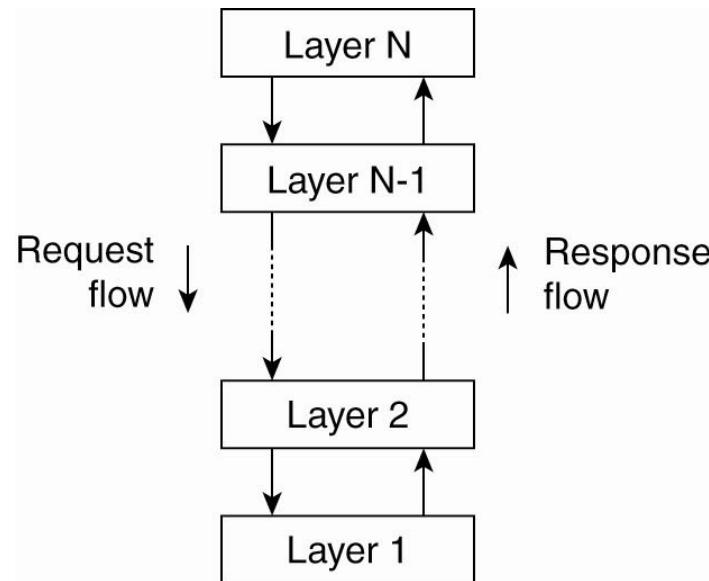
# SYSTEM CHARACTERISTICS

- **Centralized:** A single component/subsystem is “in charge”
  - **Vertical** (or hierarchical) organization of communication and control paths
  - Logical separation of functions into client (requester) and server (responder)
- **Decentralized:** multiple components/subsystems interact as peers
  - **Horizontal** rather than hierarchical communication and control
  - Communication paths may be less structured; symmetric functionality
- **Hybrid:** combine elements of both

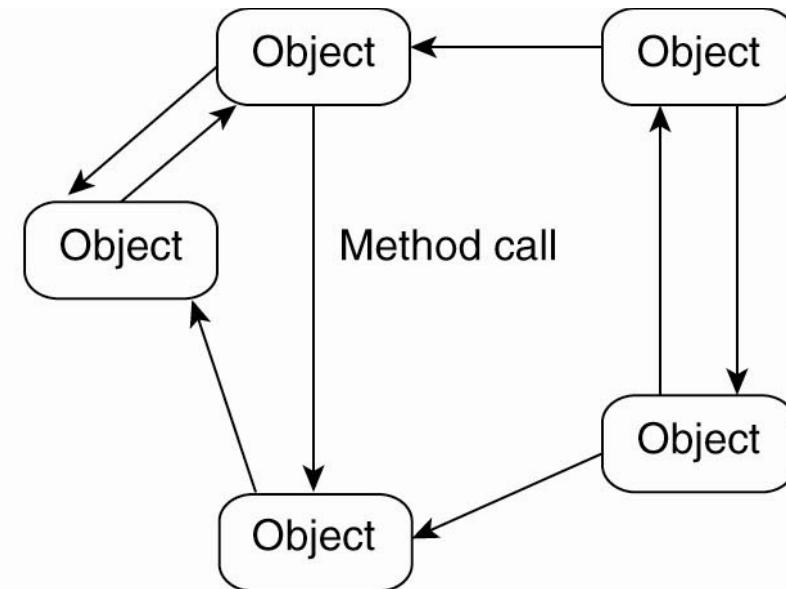
Classification of a system as **centralized** or **decentralized** primarily refers to **communication** and **control organization**

# VERTICAL VS HORIZONTAL

- Vertical: Layers with different functionality. Restricted communication

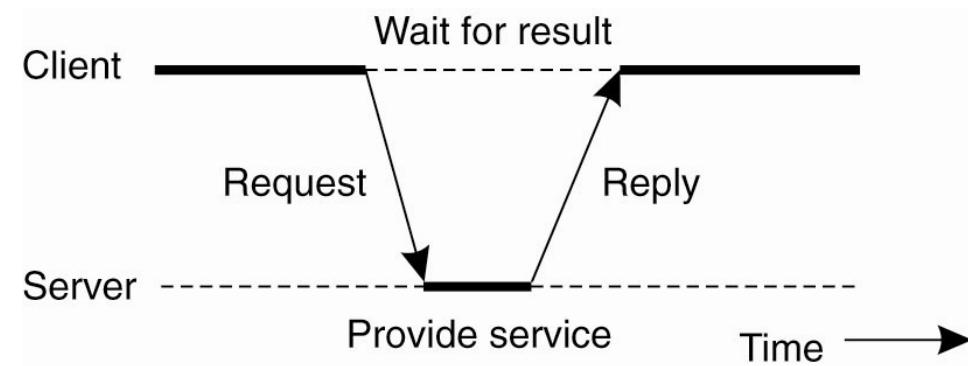


- Horizontal: Components with similar functionality or more diverse communication



# TRADITIONAL CLIENT-SERVER

- Processes are divided into two groups (clients and servers).
- Synchronous communication: request-reply protocol
  - Could be message oriented or RPC



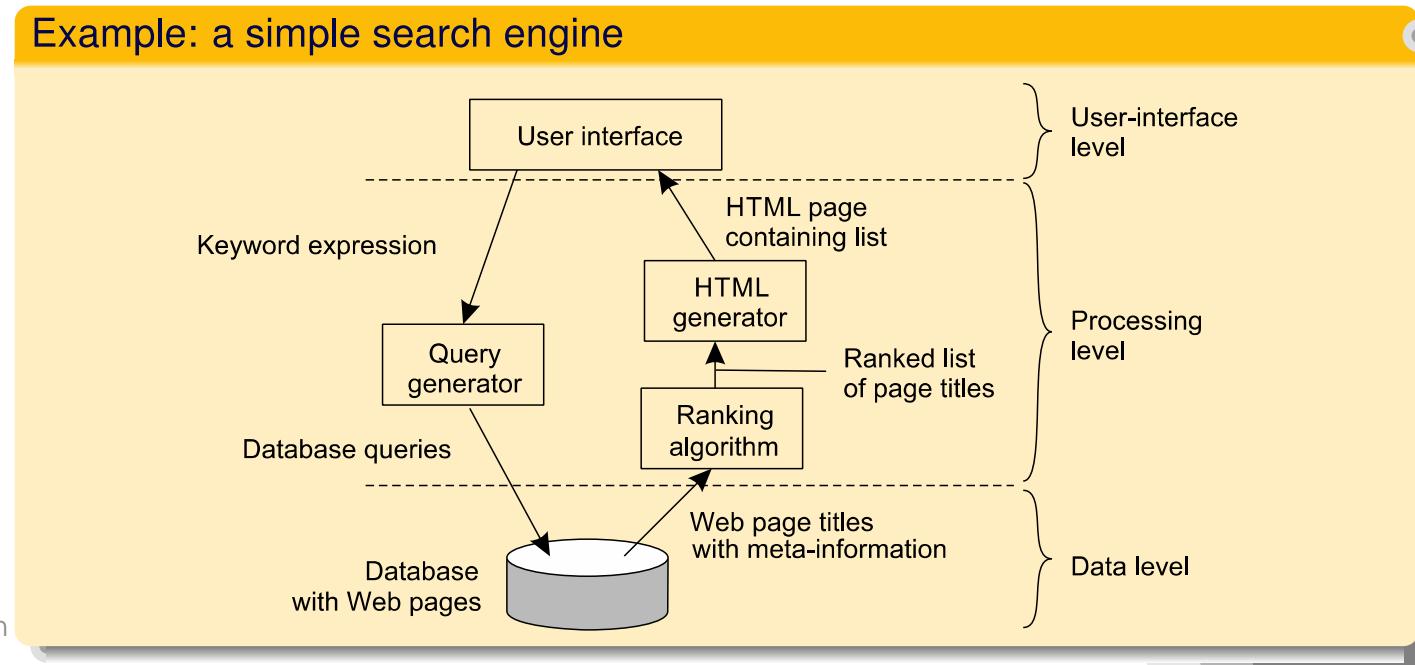
- Note: even in this simple example, lots could go wrong!

# CLIENT ARCHITECTURE

- Server provides processing and data management; client provides simple graphical display (**thin-client**)
  - Pro: Easier to manage, more reliable, client machines don't need to be so large and powerful
  - Con: Potential performance loss at client
- At the other extreme, all application processing and some data resides at the client (**fat-client** approach)
  - Pro: reduces workload at server; more scalable
  - Con: harder to manage by system admin, less secure

# LAYERED SERVER EXAMPLE

- **User-interface level:** GUI's (usually) for interacting with end users
- **Processing level:** data processing applications – the core functionality
- **Data level:** interacts with data base or file system



# TIERS, LAYERS, NODES, COMPONENTS

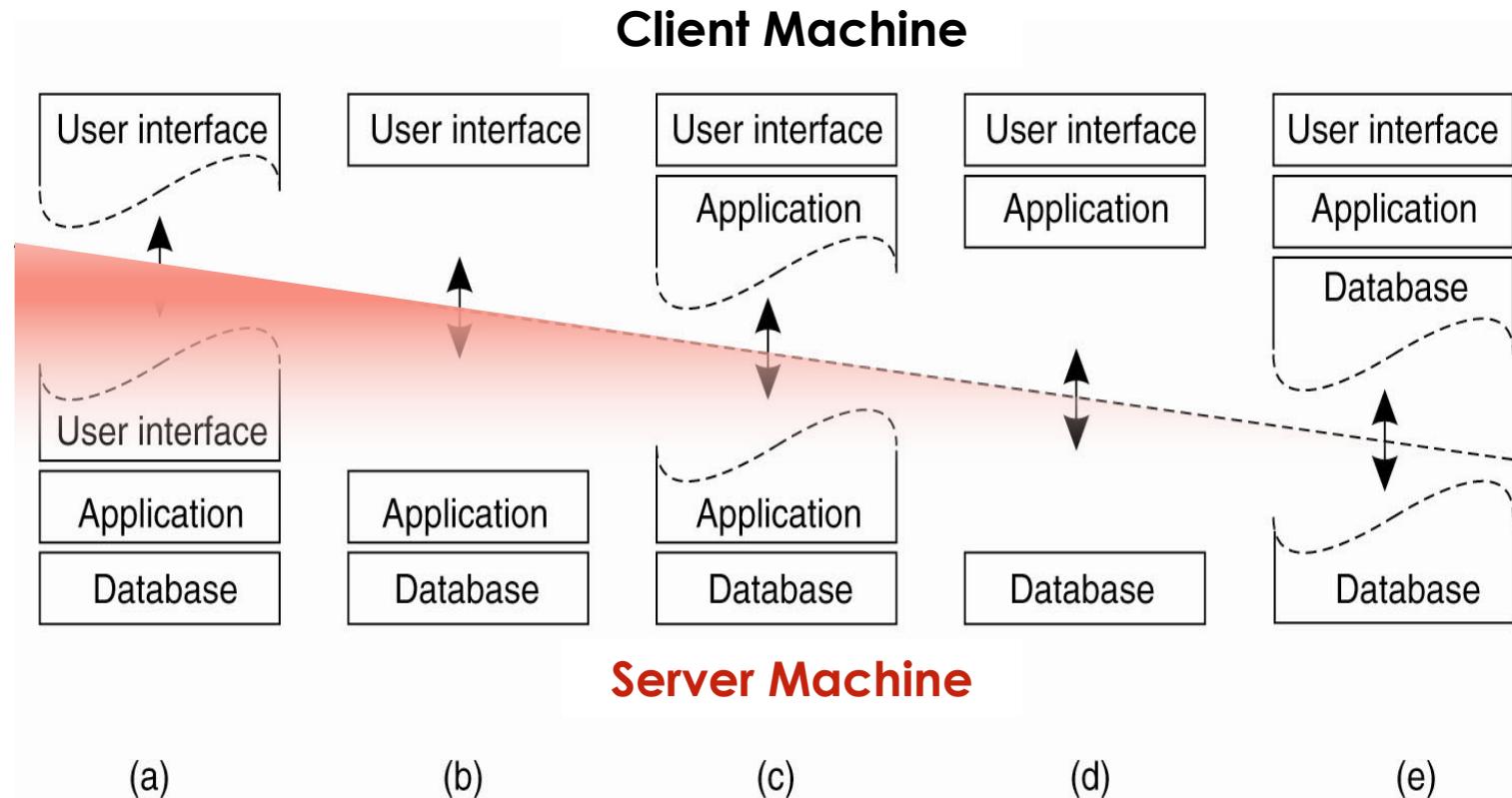
- **Layer** and **tier** are roughly equivalent terms, but *layer* typically implies software and *tier* is more likely to refer to component deployment on HW.
  - Several software layers might comprise a subsystem deployed as a single “tier” in a multi-tier web application
- **Components** are generally software, whereas a **node** could refer to a component deployed on a particular server

Layers / Components = Software

Tiers / Nodes = Software deployed on hardware  
(usually\*)

# CLIENT-SERVER SPLIT

Can you come up with an example service/application which uses each of these architectures?



# THREE-TIERED WEB ARCHITECTURE



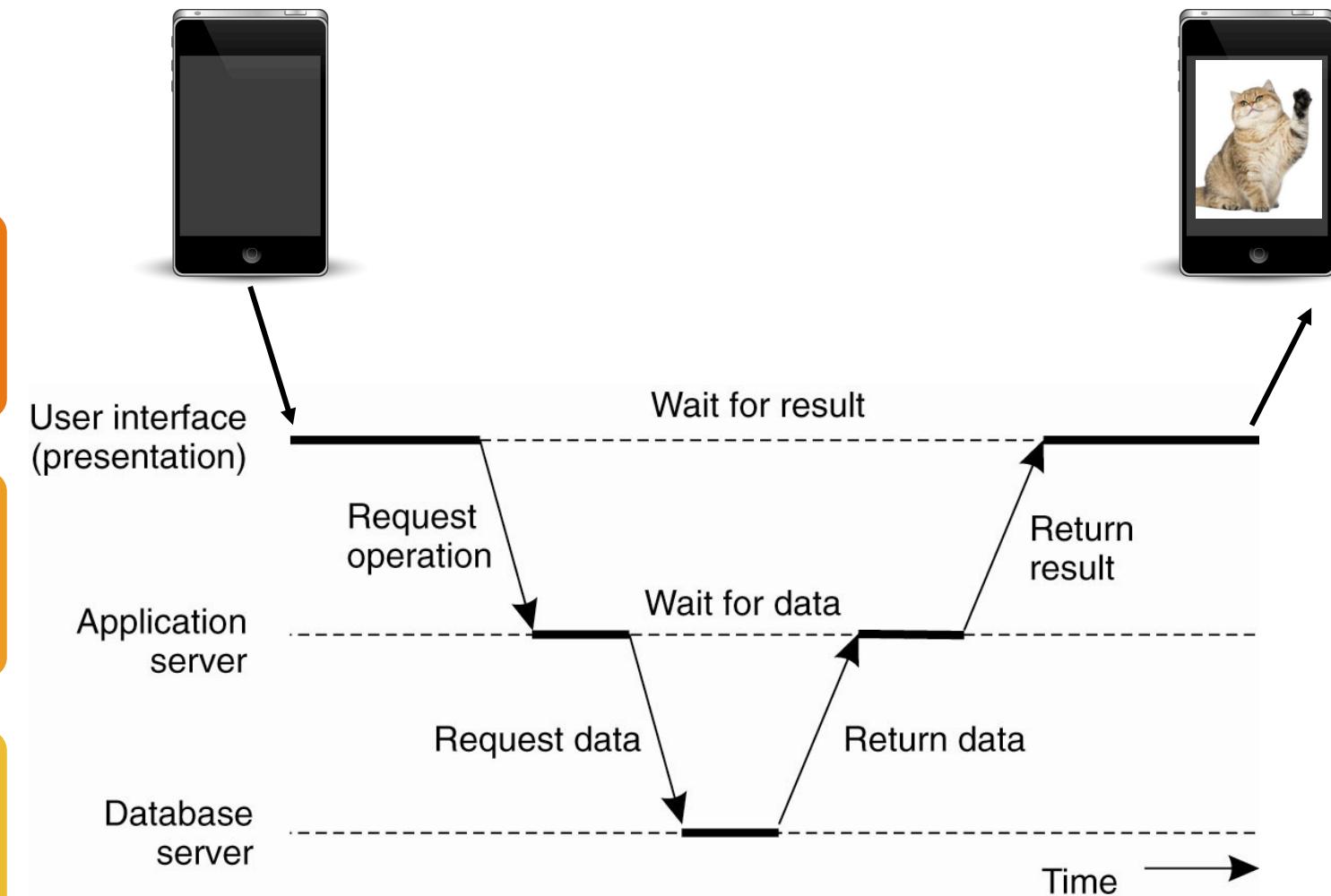
Tiers can be spread across multiple servers



Simplifies deployment, performance management, reliability



Servers also can play the role of client



“LAMP Stack” (Linux, Apache, MySQL, PHP) was the 3-tier standard

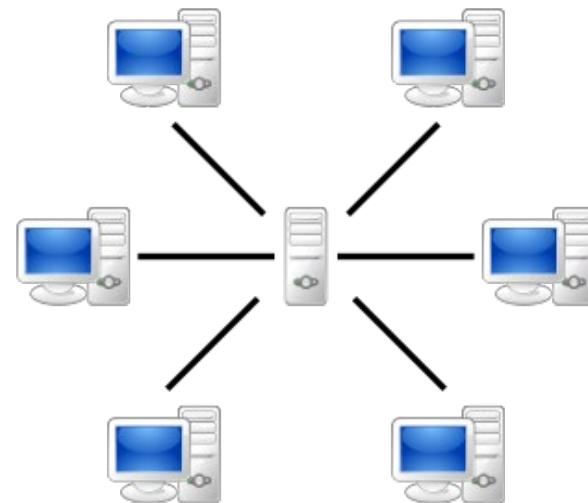
Prof. Tim Wood & Prof. Roozbeh Haghnazari

# DECENTRALIZED ARCHITECTURES

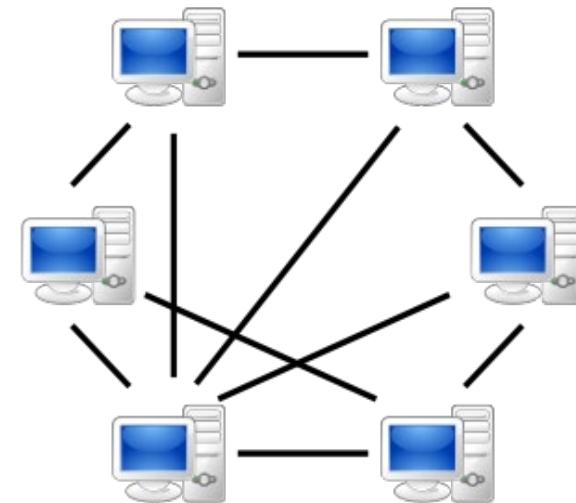


# PEER TO PEER SYSTEMS

- A distributed system that does not rely on centralized coordination
- Peers are *equipotent* and work together to provide a service



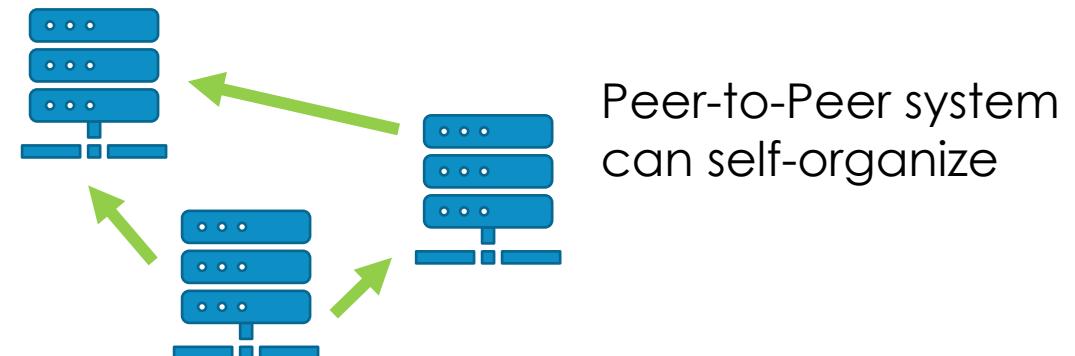
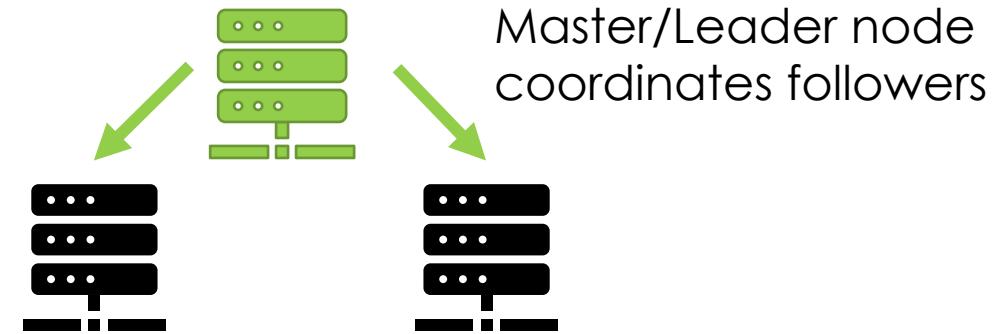
**Centralized**



**P2P**

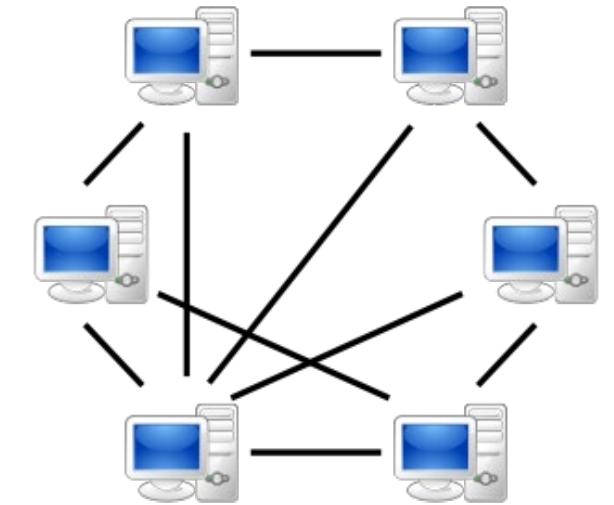
# DECENTRALIZATION BENEFITS

- Centralized systems may have **a single point of failure**
  - Affects reliability and may be a performance bottleneck
- Decentralization can make a system **more robust and performant**
  - But only if it is well designed!



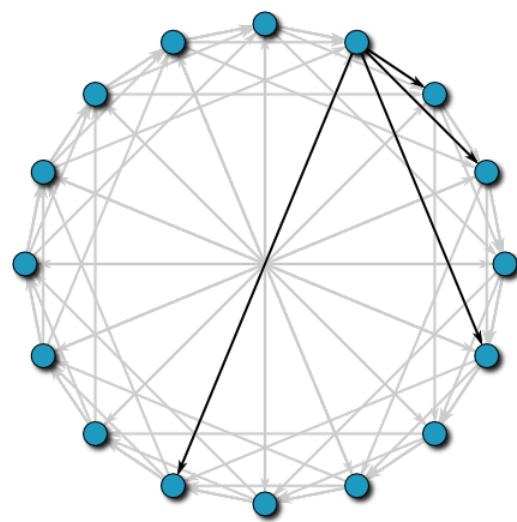
# P2P CHALLENGES

- Routing and Discovery
  - How to reach other nodes?
  - How to find out what other nodes exist?
  - How to bootstrap when you first join?
- Consistency
  - How to keep information consistent across the network?
- Reliability / Failure Handling
  - What happens when nodes crash and rejoin?
- Performance
  - How to get predictable performance with limited control?

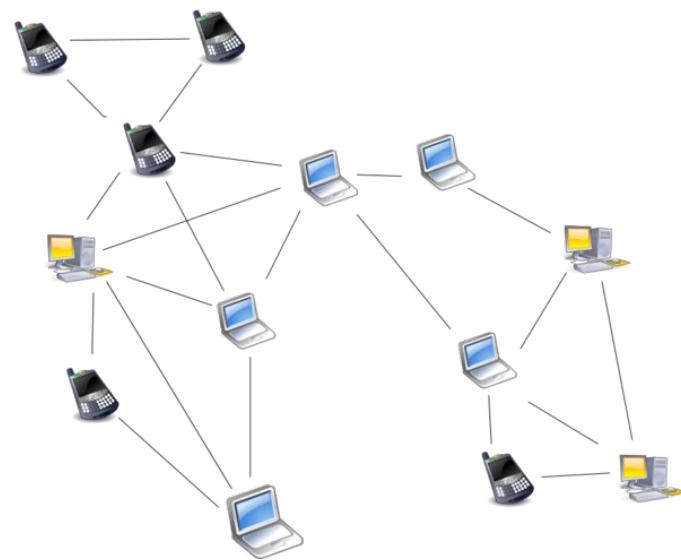


# P2P ARCHITECTURES

Structured



Unstructured



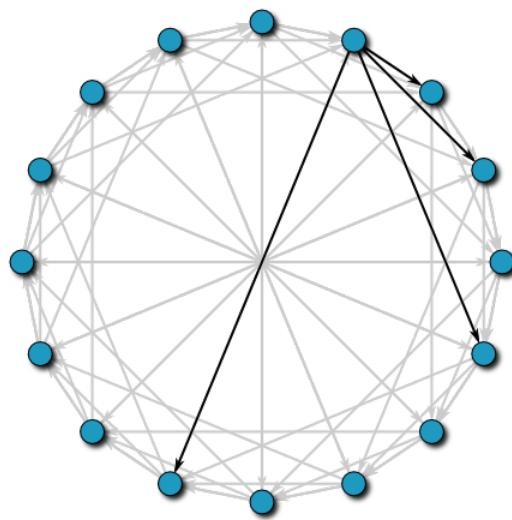
- The **peers** can be connected in a organized (**structured**) manner or in an ad-hoc (**unstructured**) manner
  - **Why/When might you choose one over the other?**

# EXTRA SLIDES

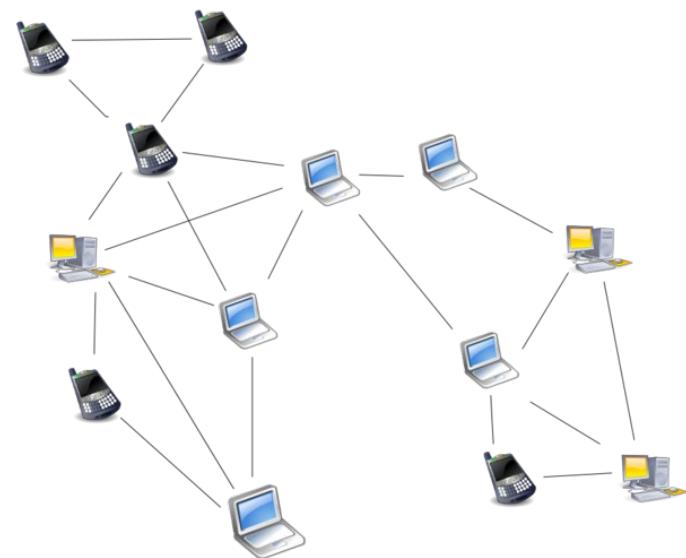
For later...

# BOOTSTRAPPING

Structured



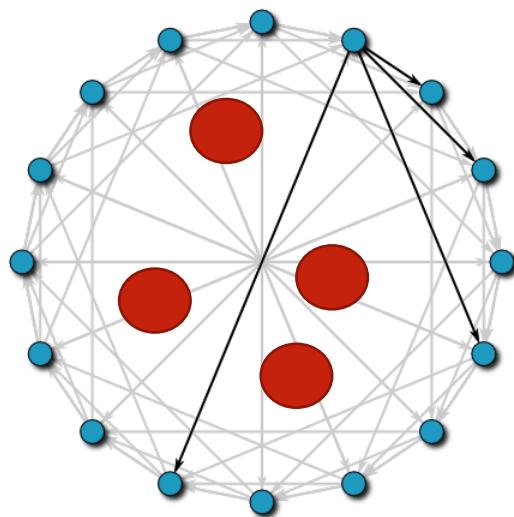
Unstructured



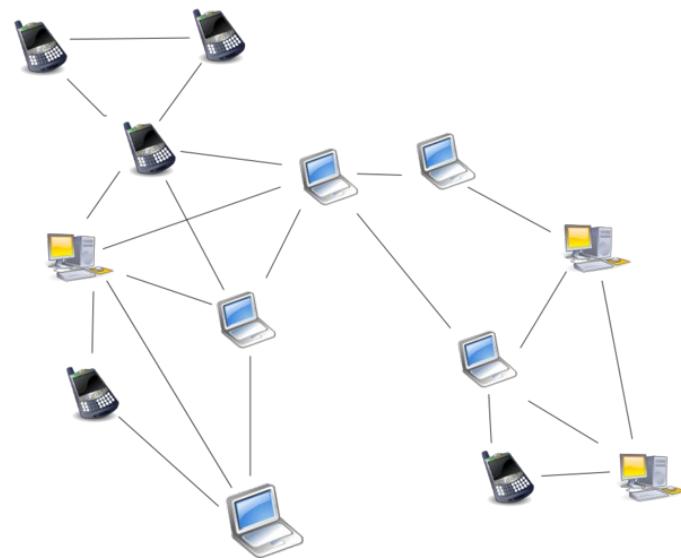
- How can a node join a P2P network if there is no centralized server to connect to?

# BOOTSTRAPPING

Structured



Unstructured

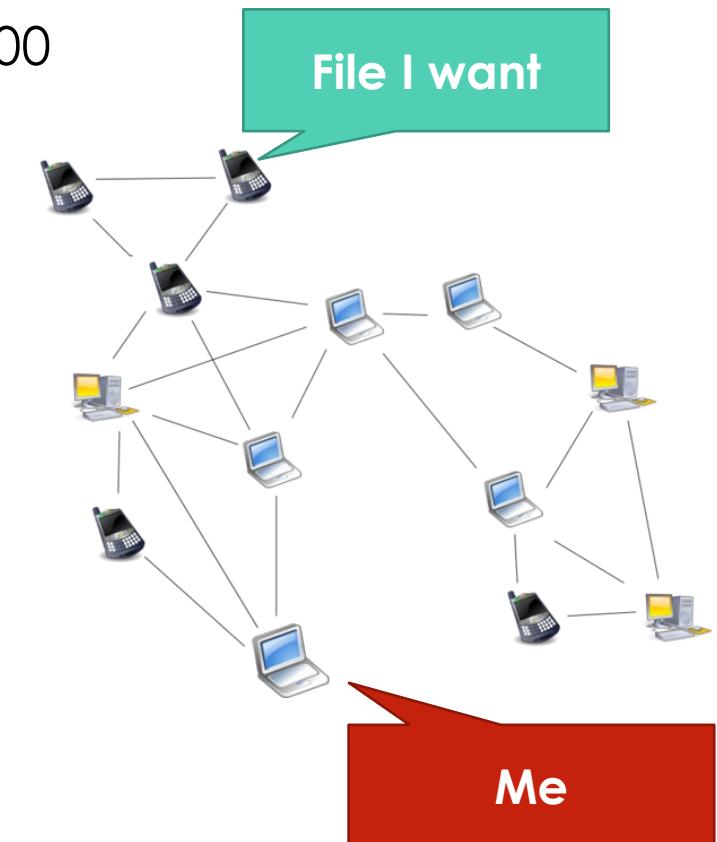


- Common Assumptions:
  - New nodes have address of at least one other active node
  - Special peers store extra information

- New nodes can broadcast on their radio to find close neighbors

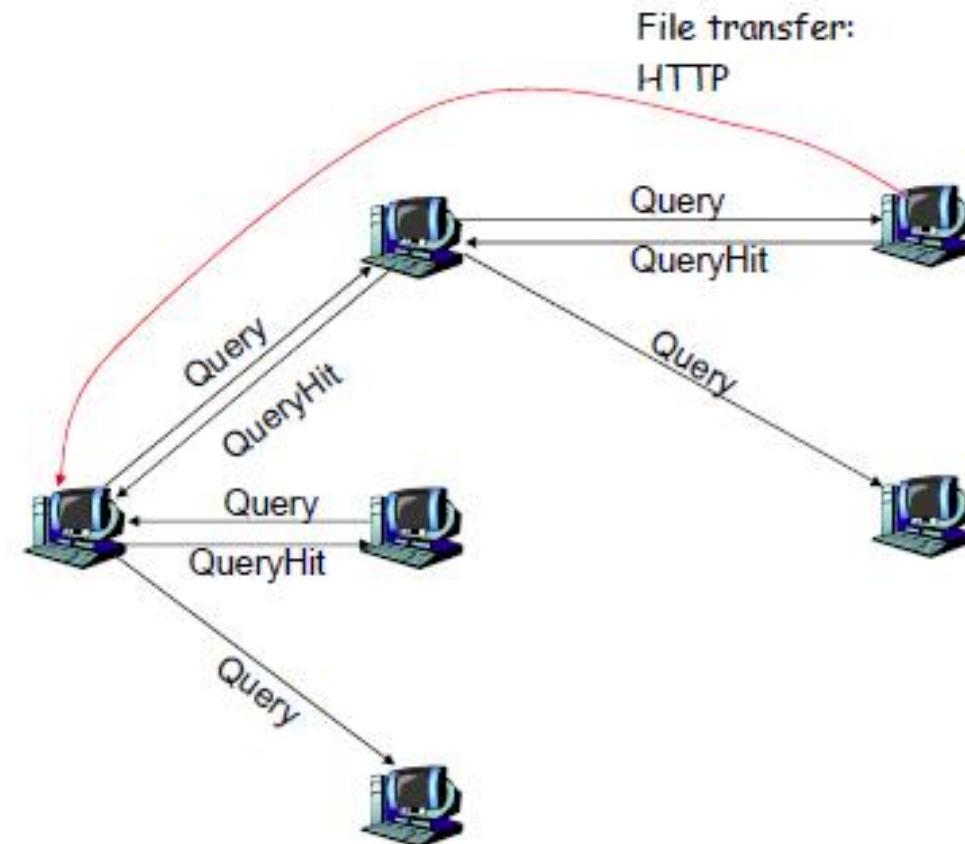
# GNUTELLA P2P FILE SHARING

- Peer-to-Peer file sharing service
  - Released by a company owned by AOL on March 14, 2000
  - AOL shut down the company the next day...
- **Unstructured** P2P system
  - Bootstrap using pre-defined addresses of starter nodes
  - Randomly pick a set of  $N$  neighbors ( $N=5$ )
  - Search for files by querying neighbors
  - Neighbors propagate searches up to  $H$  hops total ( $H=7$ )
  - Responses travel back the same path
- Once file is found, transfer over direct connection



# GNUTELLA

- At most how many neighbors will this search?
  - 5 neighbors per node
  - 7 hop max path
- This is a form of *flooding*
- What could make this more efficient?



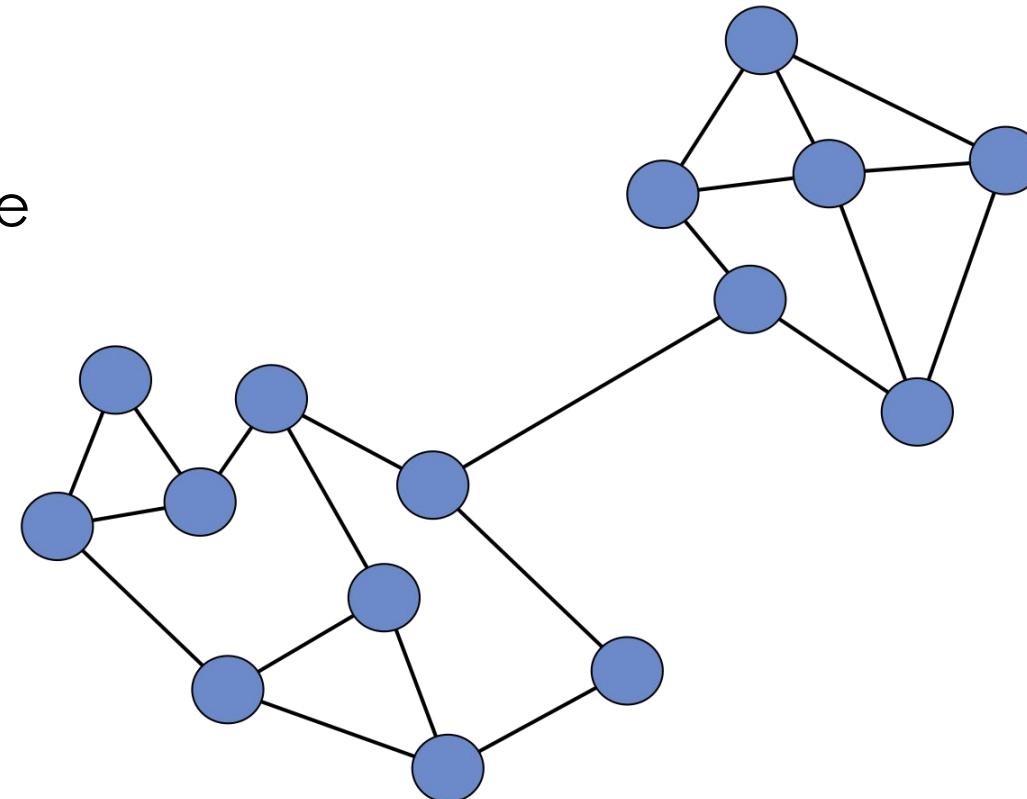
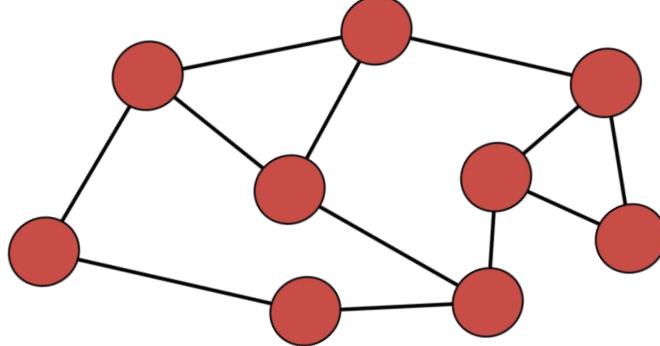
Figures from wikipedia

# NOT ALL PEERS ARE EQUAL

- Gnutella v0.6 added Ultra Peers and Leaves
- Leaf Node:
  - Connects to 3 Ultra Peers
  - Maintains an index of all its content
  - Send queries to Ultra Peer
- Ultra Peer:
  - Connects to 32 Ultra Peers
  - Forwards queries at most 4 hops (not 7)
  - Merges the content indexes of all leaf nodes
  - Shares content index with all adjacent Ultra Peers
  - Only send to an Ultra peer on the 4th hop if query is in index
- How does this change things?

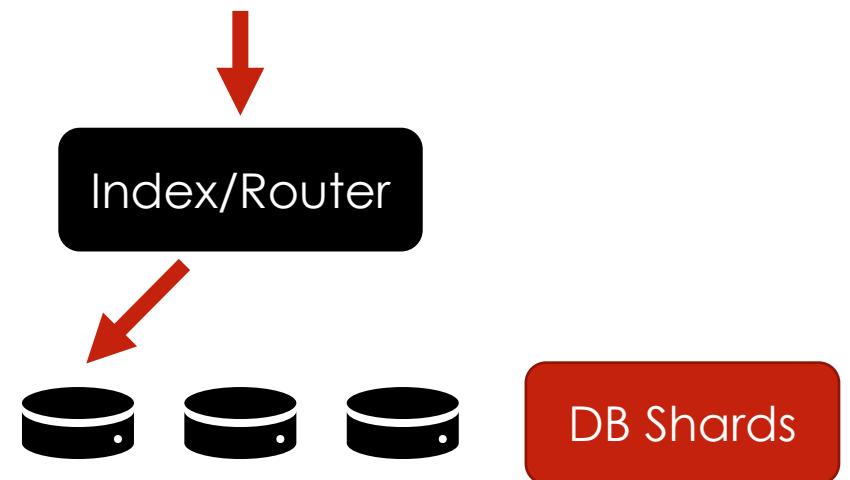
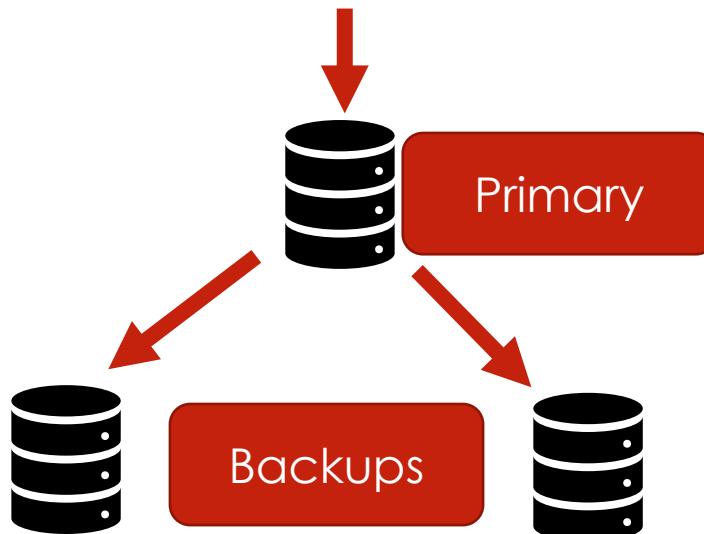
# HOW TO PICK NEIGHBORS?

- Want to avoid disconnected components and weak connectivity between groups!
- This is why some networks enforce a structure or hierarchy



# SEMI-DECENTRALIZED

- Replicated Database
  - All writes go to Primary
  - Reads can be distributed
- Sharded Database
  - Requests are routed to different DB “shards” that store part of data

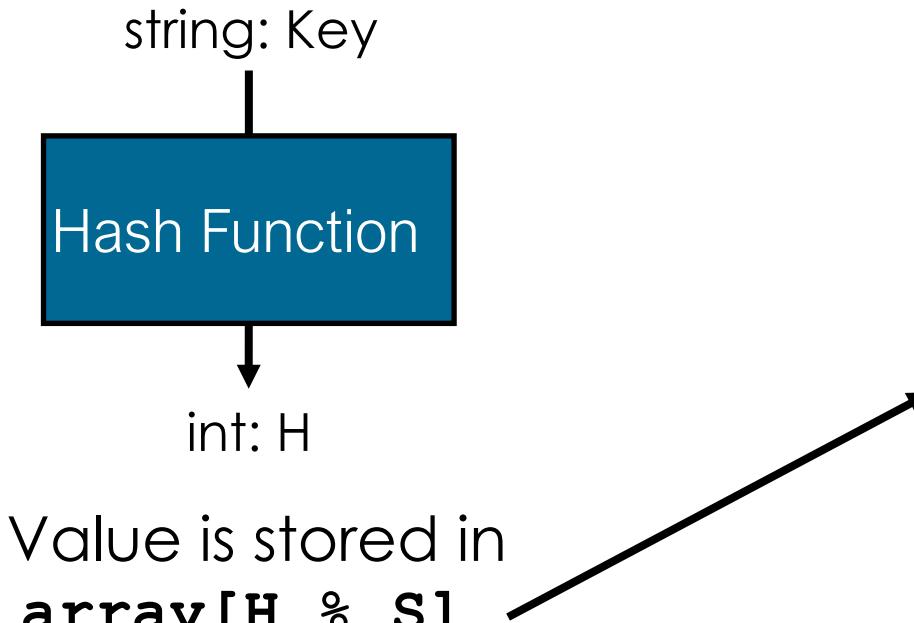


# FULLY DECENTRALIZED P2P

- How to design a fully decentralized Key Value store?
  - Record <Key, Value> pairs like a hash table
- Distributed Hash Table (DHT) goals:
  - Evenly partition data across nodes
  - Efficiently find where data is stored
  - Gracefully handle nodes joining/leaving

# SIMPLE HASH TABLE

- Find the value for a key in an array

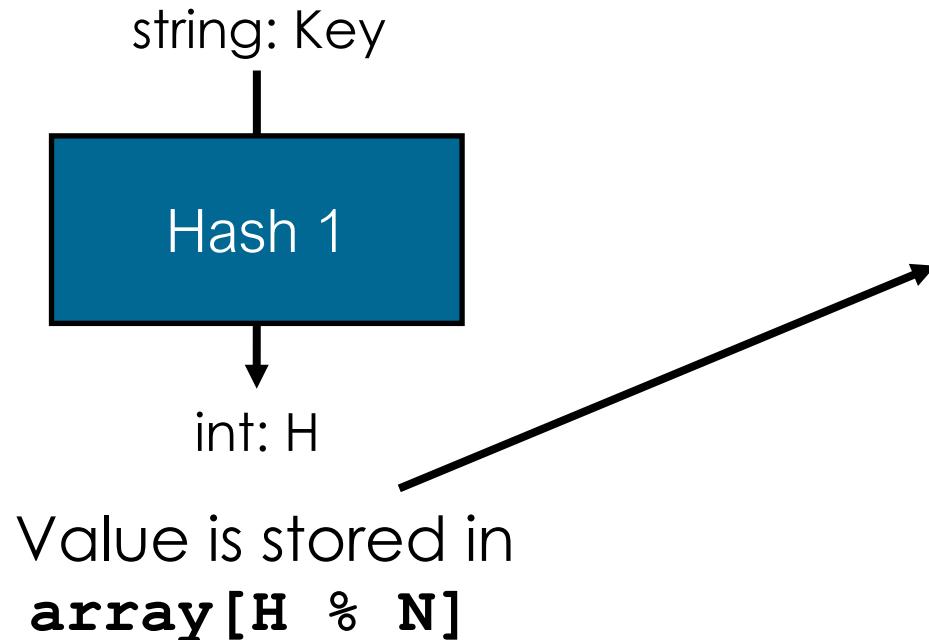


$S = \text{array size}$

Array Index	Value
X	
1	v1
2	v2
...	...
S	vS

# SIMPLE DHT V1

- What if one node can't fit all the data?
- Do two hash lookups!



$N = \# \text{ nodes}$

Node Address	
1	
2	
...	
N	

$S = \text{array size}$

Array Index	Value
1	v1
2	v2
...	...
S	vn

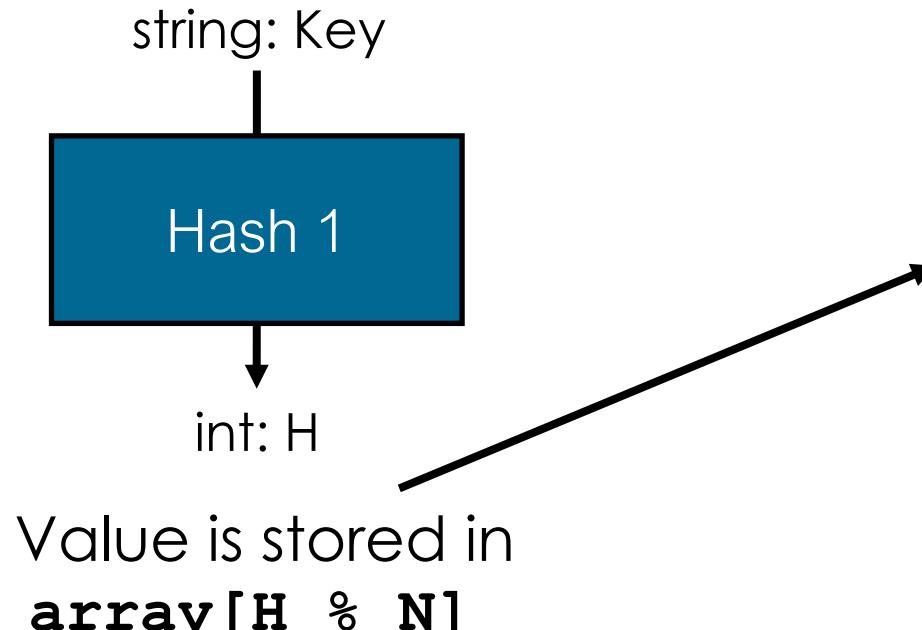
Hash 2

Array Index	Value
1	v1
2	v2
...	...
S	vn

Array Index	Value
1	v1
2	v2
...	...
S	vn

# SIMPLE DHT V1

- When will this perform poorly?



N = # nodes

Node Address	
1	
2	
...	
N	

S = array size

Array Index	Value
1	v1
2	v2
...	...
S	vn

Hash 2

Array Index	Value
1	v1
2	v2
...	...
S	vn

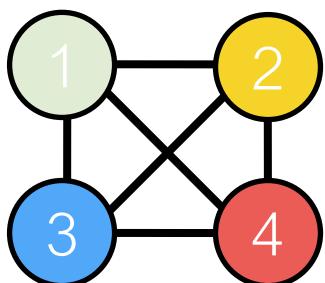
Array Index	Value
1	v1
2	v2
...	...
S	vn

# DHT CHURN

- Churn is when nodes are frequently joining or leaving
  - In a DHT it is OK to lose data when a node leaves, but it shouldn't cause all other nodes to reshuffle their data!

## Simple DHT

Value is stored on  
**node [H % 4]**



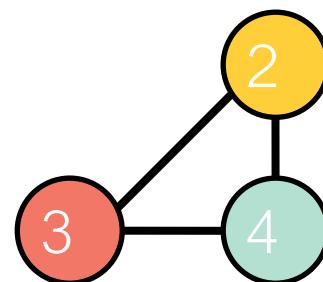
Divides hash space  
into 4 equal partitions  
for 4 servers

# CHURN

- Churn is when nodes are frequently joining or leaving
  - In a DHT it is OK to lose data when a node leaves, but it shouldn't cause all other nodes to reshuffle their data!



Oops!  
Green node  
failed!



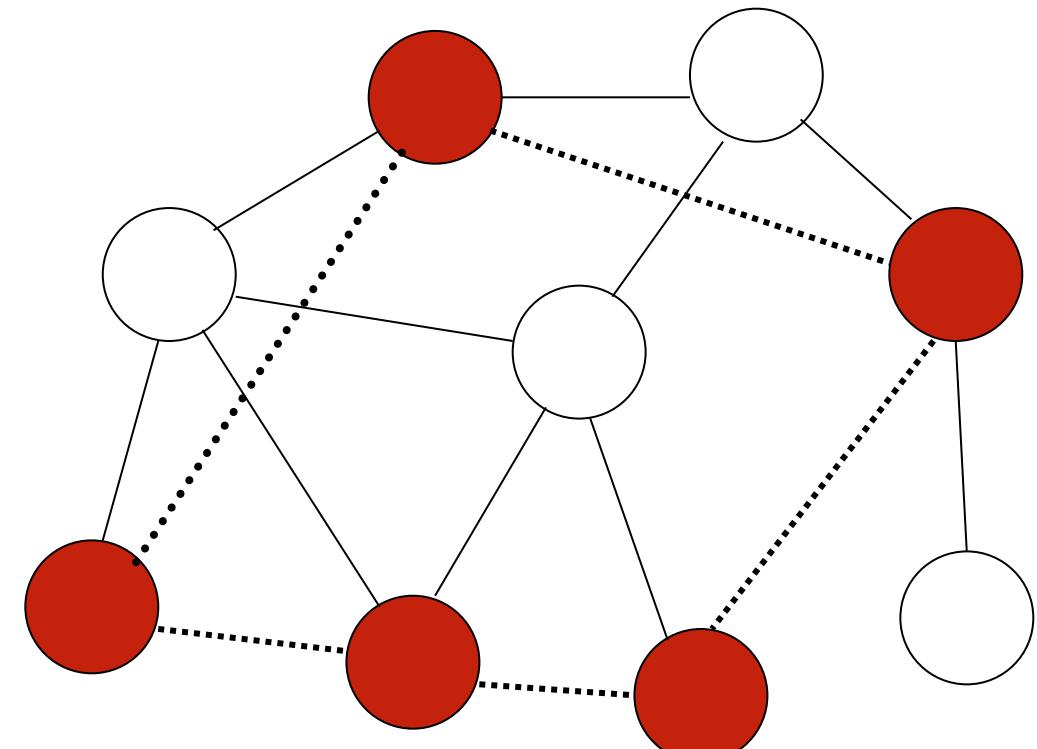
All nodes  
needs to be  
reorganized!

# STRUCTURED PEER TO PEER

- Symmetric behavior, peer-to-peer architectures evolve around the question how to organize the processes in an **overlay network**, that is, a network in which the nodes are formed by the processes and the links represent the possible communication channels.
- The overlay network is constructed using a deterministic procedure.
- By far the most-used procedure is to organize the process through a **distributed hash table(DHT)**

# OVERLAY NETWORK

- Are logical or *virtual* networks, built on top of a physical network
- A link between two nodes in the overlay may consist of several physical links.
- Messages in the overlay are sent to logical addresses, not physical (IP) addresses
- Various approaches used to resolve logical addresses to physical.



# DISTRIBUTED HASH TABLE

- Allows you to insert, delete and lookup objects with keys
- Performance concerns:
  - Load balancing
  - Fault tolerance
  - Efficiency of lookup and inserts
  - Locality
- Napster, Gnutella, FastTrack, etc. are kind of DHT

# CHARACTERISTICS OF DHT

- Scalable – to thousands, even millions of network nodes
  - Search time increases more slowly than size; usually  $O(\log(N))$
- Fault tolerant – able to re-organize itself when nodes fail
- Decentralized – no central coordinator  
(example of decentralized algorithms)

# STRUCTURED P2P ARCHITECTURES

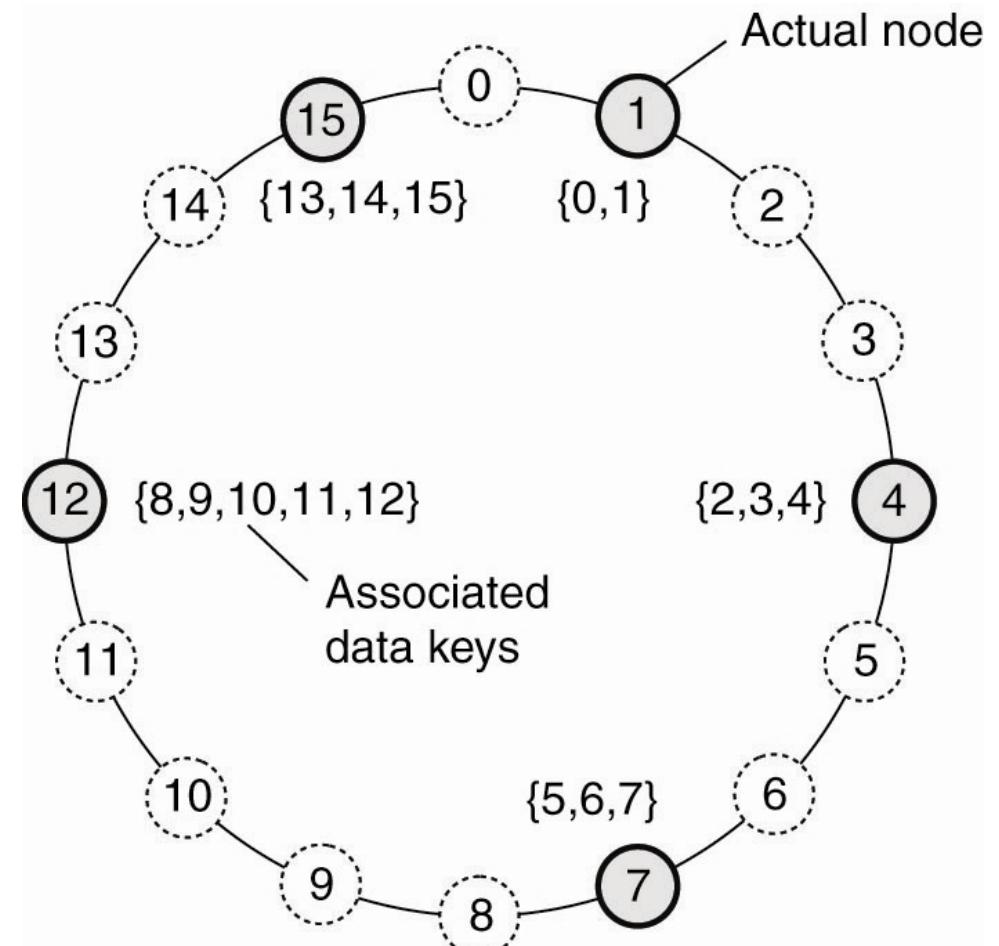
- In a DHT, data objects and nodes are each assigned a key which hashes to a random number from a very large identifier space (to ensure uniqueness)
- A mapping function assigns objects to nodes, based on the hash function value.
- A lookup, also based on hash function value, returns the network address of the node that stores the requested object.
- Chord is a structured peer to peer system which addresses the mentioned concerns

# CHORD

- Nodes are logically arranged in a circle
- Intelligent choice of neighbors to reduce the latency and message cost of routing
- Uses consistent hashing on Nodes' addresses
  - Ex. **SHA-1(ip,port) -> 160 bit string**
  - e.g., a node's key is a hash of its IP address and a file's key might be the hash of its name or of its content or other unique key.
  - The hash function is consistent; which means that keys are distributed evenly across the nodes, with high probability.

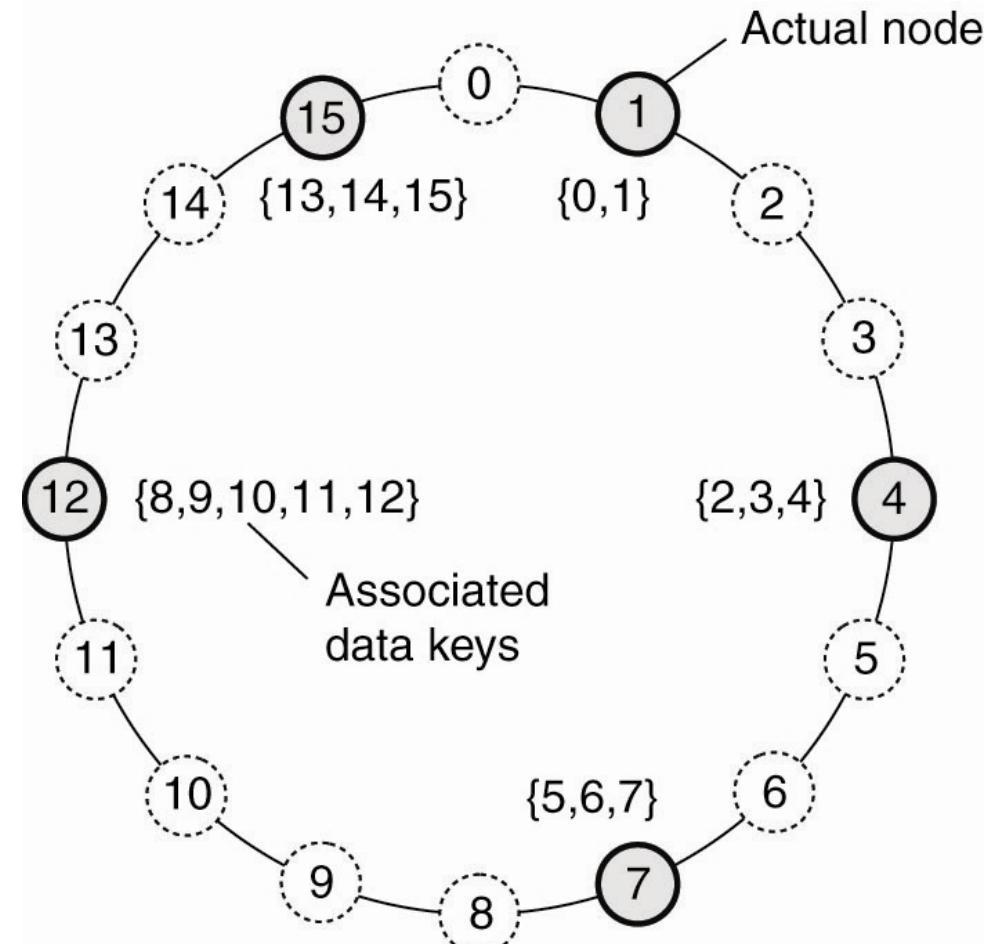
# INSERTION IN CHORD

- A data item with key value  $k$  is mapped to the node with the smallest identifier  $id$  such that  $id \geq k$
- This node is the successor of  $k$ , or  $\text{succ}(k)$
- Modular arithmetic is used



# LOOKUP IN CHORD

- Each node in the network knows the location of some fraction of other nodes.
  - If the desired key is stored at one of these nodes, ask for it directly
  - Otherwise, ask one of the nodes you know to look in *its* set of known nodes.
  - The request will propagate through the overlay network until the desired key is located
  - Lookup time is  $O(\log(N))$

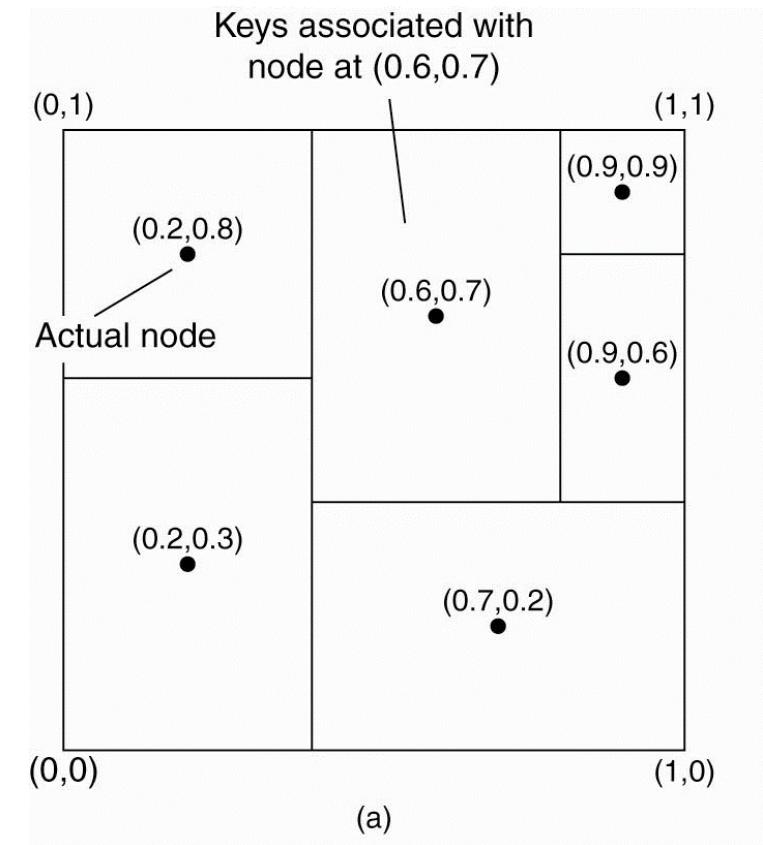


# JOINING & LEAVING THE CHORD

- Join
  - Generate the node's random identifier,  $\text{id}$ , using the distributed hash function
  - Use the lookup function to locate  $\text{succ}(\text{id})$
  - Contact  $\text{succ}(\text{id})$  and its predecessor to insert self into ring.
  - Assume data items from  $\text{succ}(\text{id})$
- Leave (normally)
  - Notify predecessor & successor;
  - Shift data to  $\text{succ}(\text{id})$
- Leave (due to failure)
  - Periodically, nodes can run “self-healing” algorithms

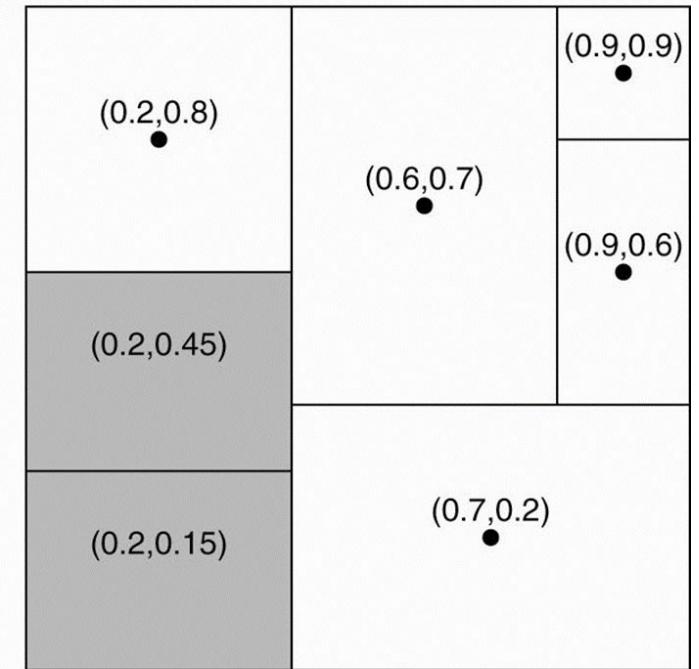
# CONTENT ACCESSIBLE NETWORK (CAN)

- 2-dim space  $[0,1] \times [0,1]$  is divided among 6 nodes
- Each node has an associated region
- Every data item in CAN will be assigned a unique point in space
- A node is responsible for all data elements mapped to its region



# CONTENT ACCESSIBLE NETWORK (CAN)

- To add a new region, split the region
- To remove an existing region, neighbor will take over



# STRUCTURED P2P SUMMARY

- Deterministic: If an item is in the system it will be found
- No need to know where an item is stored
- Lookup operations are relatively efficient
- DHT-based P2P systems scale well

# UNSTRUCTURED P2P ARCHITECTURE

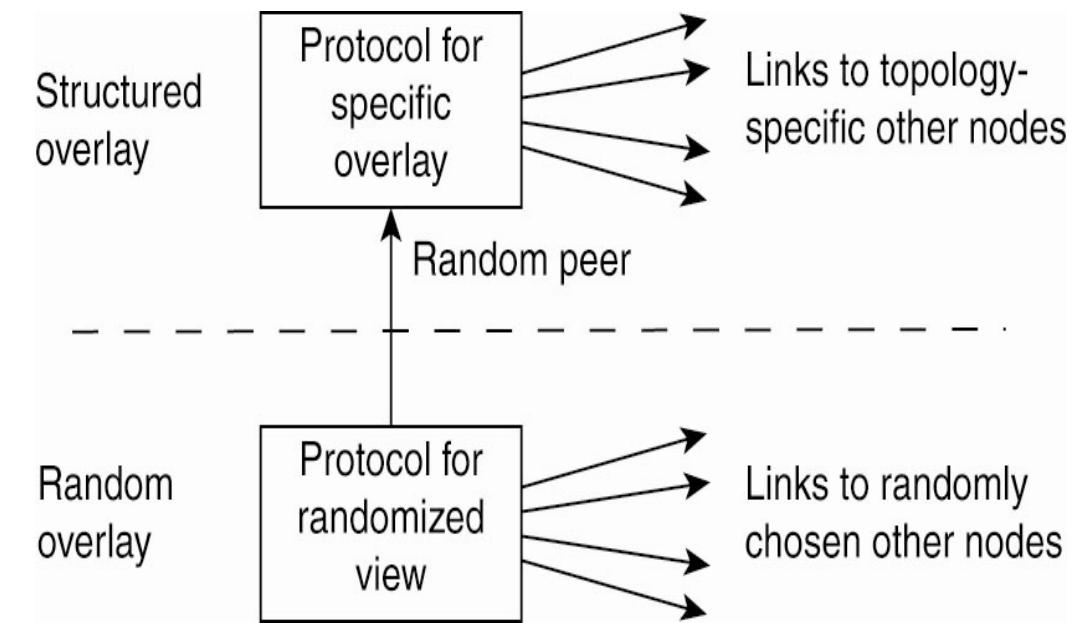
- Largely relying on randomized algorithm to construct the overlay network
  - Each node has a list of neighbors, which is more or less constructed in a random way
- One challenge is how to efficiently locate a needed data item
  - Flood the network?
- Many systems try to construct an overlay network that resembles a random graph
  - Each node maintains a **partial view**, i.e., a set of live nodes randomly chosen from the current set of nodes

# PARTIAL VIEW CONSTRUCTION

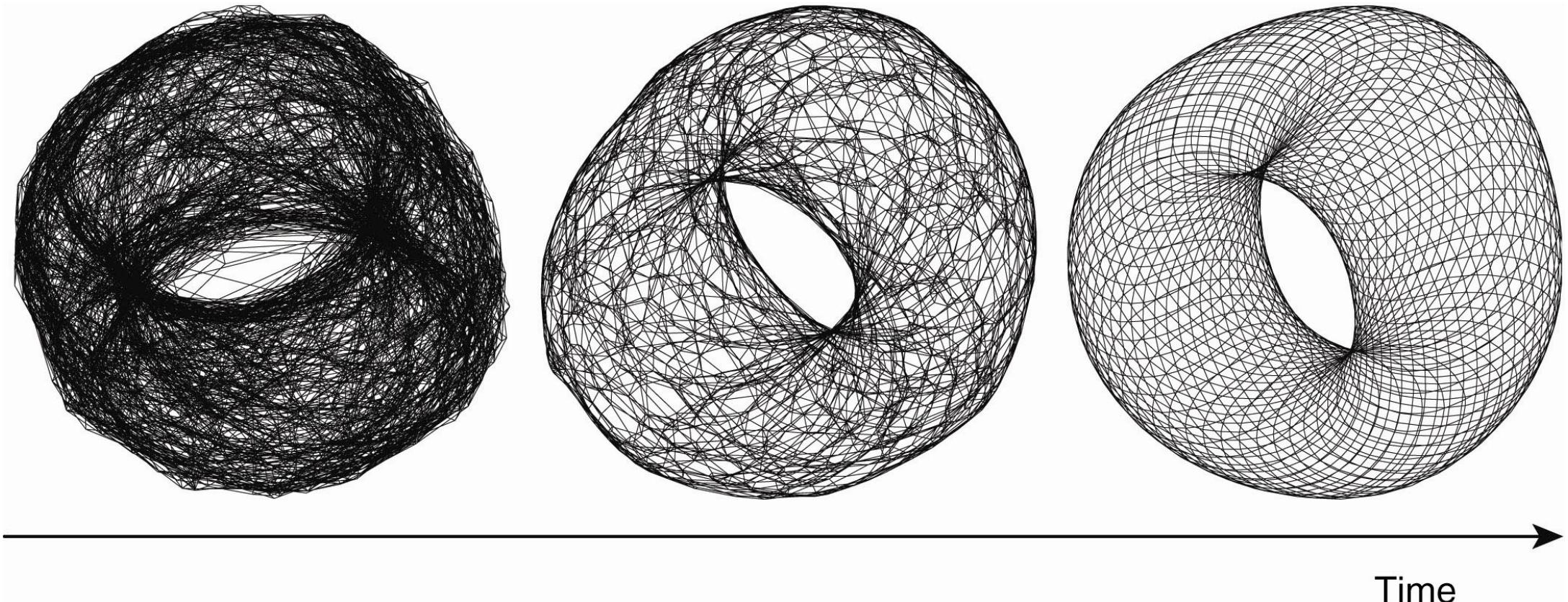
- Nodes exchange entries from their partial view regularly
  - Each entry is associated with an age tag
- Consists of an active thread and a passive thread
  - The active thread initiate the communication with a selected peer for partial view propagation
  - The passive thread waits for response from another peer and update its partial view accordingly
  - A node can be in PUSH mode or PULL mode

# TOPOLOGY MANAGEMENT

- Some specific topologies may benefit the applications in a given P2P system
  - E.g., only including nearest peers in the partial view may reduce the latency of data delivery
- How to constructing a specific topology from a unstructured P2P systems
- Solution: two-layered approach
  - Lower layer: unstructured P2P outputs a random graph
  - Higher layer: carefully exchange and selecting entries to build a desired topology



# TOPOLOGY MANAGEMENT

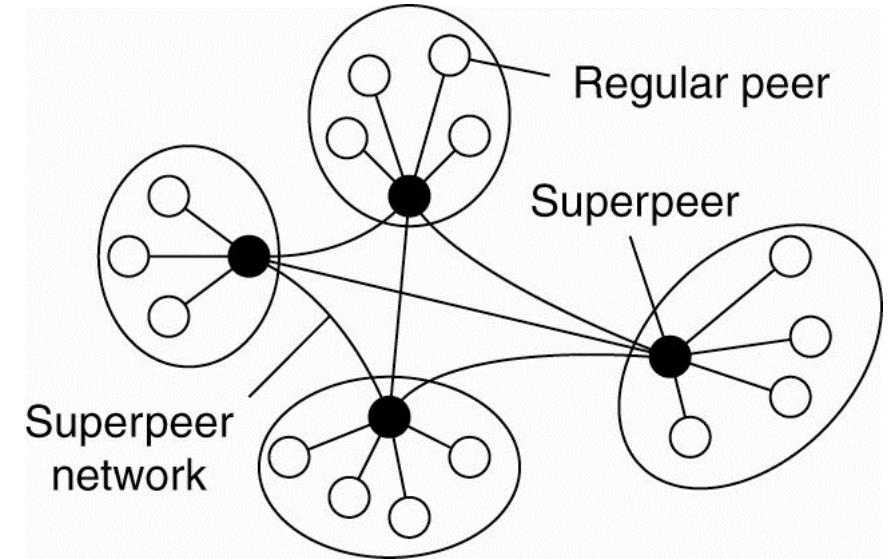


# FINDING DATA ITEMS

- This is quite challenging in unstructured P2P systems
  - Assume a data item is randomly placed
- Solution 1: Flood the network with a search query
- Solution 2: A randomized algorithm
  - Let us first assume that
- Each node knows the IDs of  $k$  other randomly selected nodes
- The ID of the hosting node is kept at  $m$  randomly picked nodes
  - The search is done as follows
- Contact  $k$  direct neighbors for data items
- Ask your neighbors to help if none of them knows
  - What is the probability of finding the answer directly?

# SUPERPEERS

- Maintain indexes to some or all nodes in the system
- Supports resource discovery
- Act as servers to regular peer nodes, peers to other superpeers
- Improve scalability by controlling floods
- Can also monitor state of network
- Example: Napster



# HYBRID ARCHITECTURES

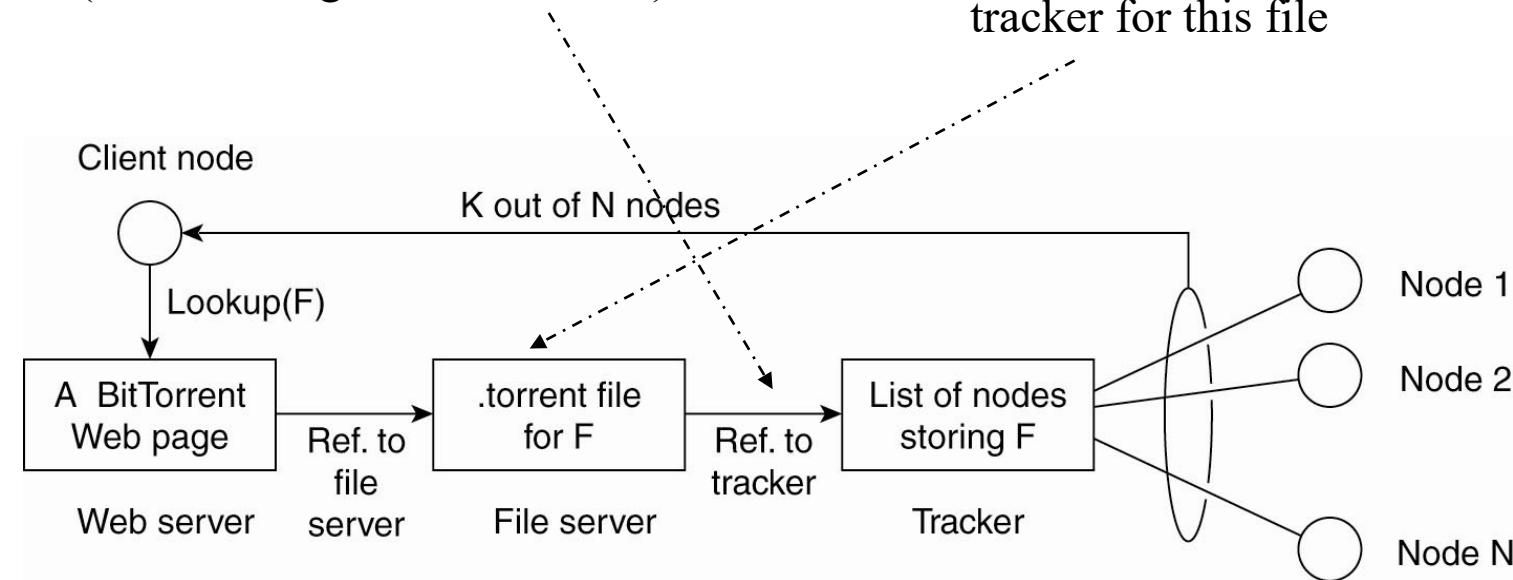
- Combine client-server and P2P architectures
  - Edge-server systems; e.g. ISPs, which act as servers to their clients, but cooperate with other edge servers to host shared content
  - Collaborative distributed systems; e.g., BitTorrent, which supports parallel downloading and uploading of chunks of a file. First, interact with C/S system, then operate in decentralized manner.
- Many real distributed systems combine architectural features
  - E.g., the superpeer networks – combine client-server architecture (centralized) with peer-to-peer architecture (decentralized)
- Two examples of hybrid architectures
  - Edge-server systems
  - Collaborative distributed systems

# COLLABORATIVE DISTRIBUTED SYSTEMS (BIT TORRENT)

- Designed to force users of file-sharing systems to participate in sharing.
- Simplifies the process of publishing large files, e.g. games
- When a user downloads your file, he becomes in turn a server who can upload the file to other requesters.
- Share the load – doesn't swamp your server

Trackers know which nodes are active  
(downloading chunks of a file)

Tells how to locate the  
tracker for this file



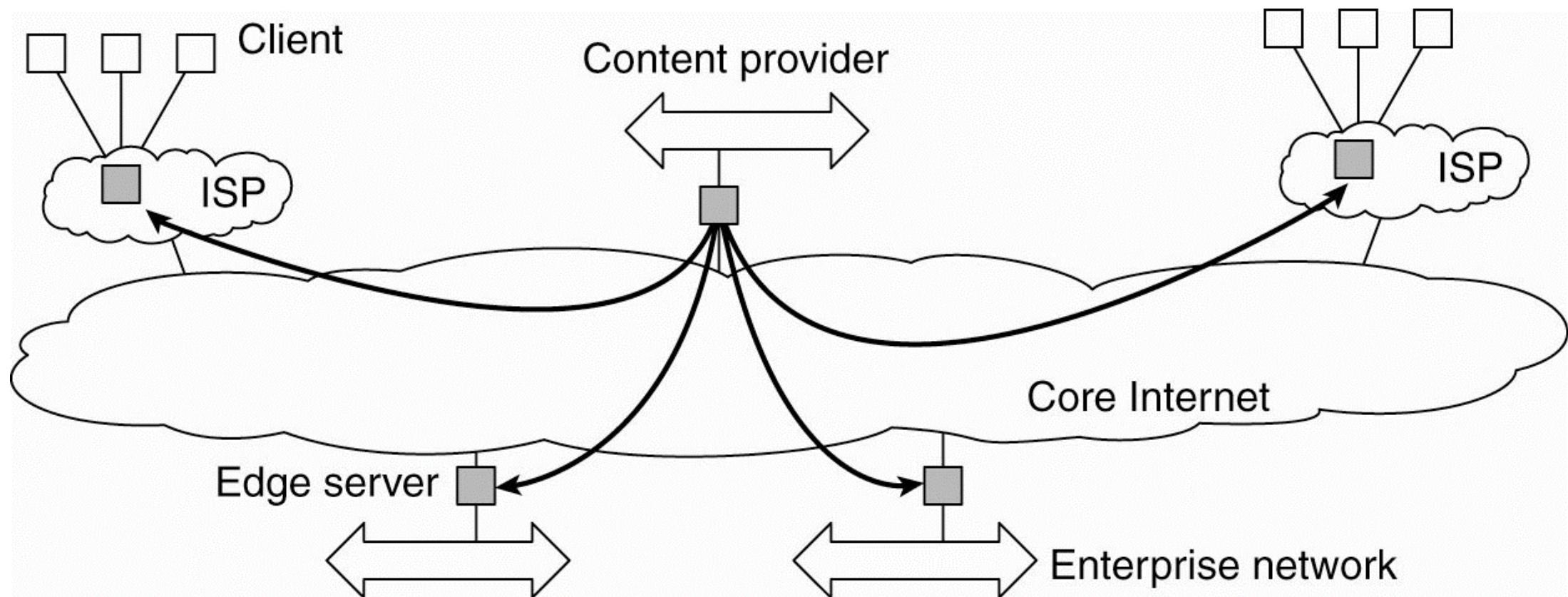
# GLOBULE

- Collaborative content distribution network:
  - Similar to edge-server systems
  - Enhanced web servers from various users that replicates web pages
- Components
  - A component that can redirect client requests to other servers.
  - A component for analyzing access patterns.
  - A component for managing the replication of Web pages.
- Has a centralized component for registering the servers and make these servers known to others

# GLOBULE

- Example:
  - Alice has a web server; Bob has a web server
  - Alice's server can have replicated contents of the Bob's server and vice versa
- Good if your server goes down
- Good if too much traffic that your server can not handle or gets too slow
- Better Geographic diversity
  - Allow users to get quick response from the nearest server with the replicated page

# EDGE-SERVER SYSTEMS



# ARCHITECTURE VERSUS MIDDLEWARE



Middleware: the software layer between user applications and distributed platforms.



Purpose: to provide distribution transparency

Applications can access programs running on remote nodes without understanding the remote environment



Many middleware follows a specific architecture style

Object-based style, event-based style  
Benefits: simpler to design application  
Limitations: the solution may not be optimal



Should be adaptable to application requirements

Separating policies from mechanism 36

# ARCHITECTURE VERSUS MIDDLEWARE

Middleware may also have an architecture

- e.g., CORBA has an object-oriented style.

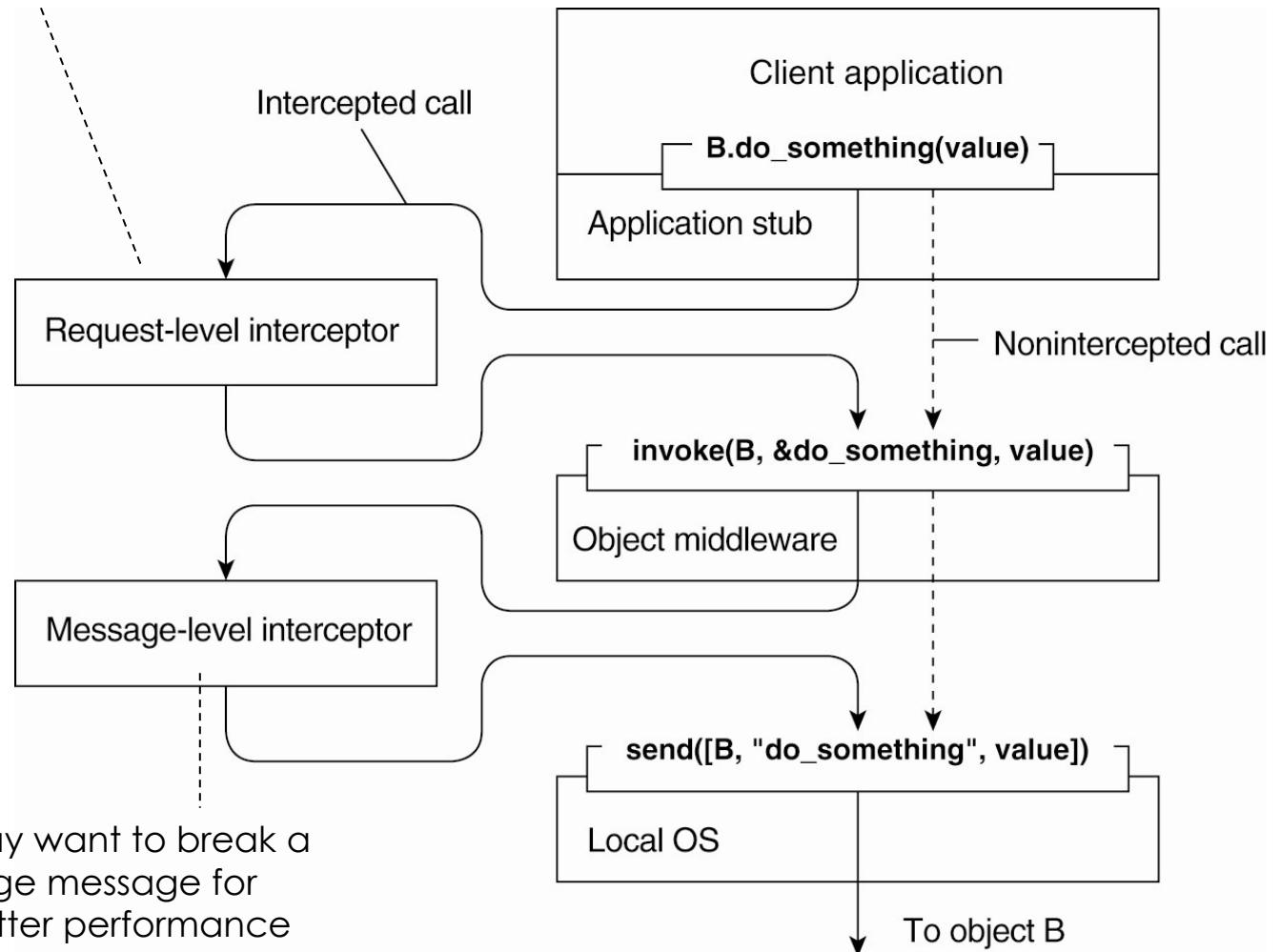
Use of a specific architectural style can make it easier to develop applications, but it may also lead to a less flexible system.

Possible solution: develop middleware that can be customized as needed for different applications.

# INTERCEPTORS

- An Interceptor is a software that
  - breaks the usual flow of control and
  - allows other (application specific) code to be executed
- It makes middleware more adaptable to
  - application requirements and changing environment
- Interceptors are good for
  - providing transparent replication and
  - improving performance

May want to send to many other B's (i.e. replicated)



# GENERAL APPROACHES FOR ADAPTABILITY

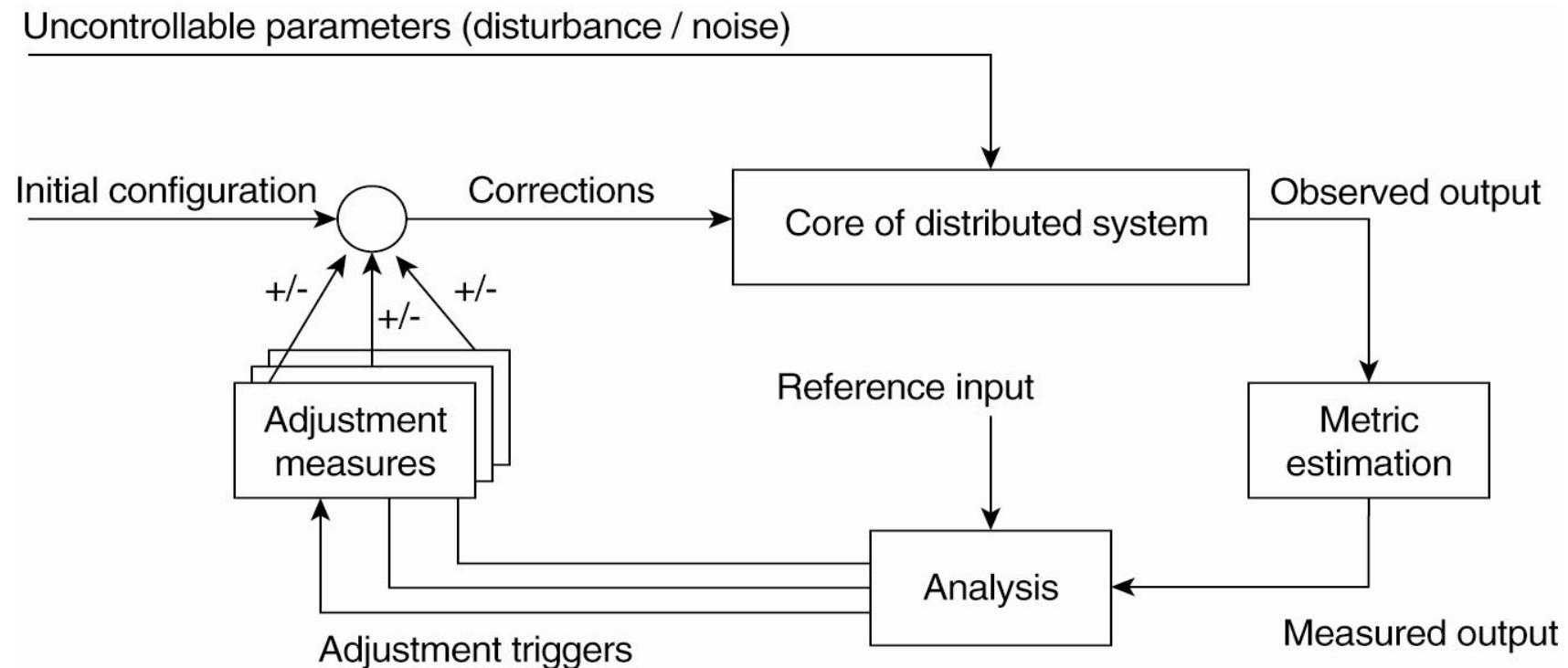
- Separation of concerns:
  - Modularizing the system and separate security from functionality
  - However, the problem is that a lot of things you cannot easily separate, e.g., security
- Computational reflection
  - Ability to inspect itself, and if necessary, adapt its behavior
  - Reflective middleware has yet to proof itself as a powerful tool to manage the complexity of distribute systems
- Component-based design
  - However, components are less independent than one may think
  - Replacement of one component may have huge impact on others

# SELF-MANAGEMENT IN DISTRIBUTED SYS.

- Distributed systems are often required to adapt to environmental changes by
  - switching policies for allocation resources
- The algorithms to make the changes are often already in the components
  - But the challenge is how to make such change without human intervention
- A strong interplay between software architectures and system architectures
  - Organize components in a way that monitoring and adjustment can be done easily
  - Decide where the processes to be executed to do the adaption

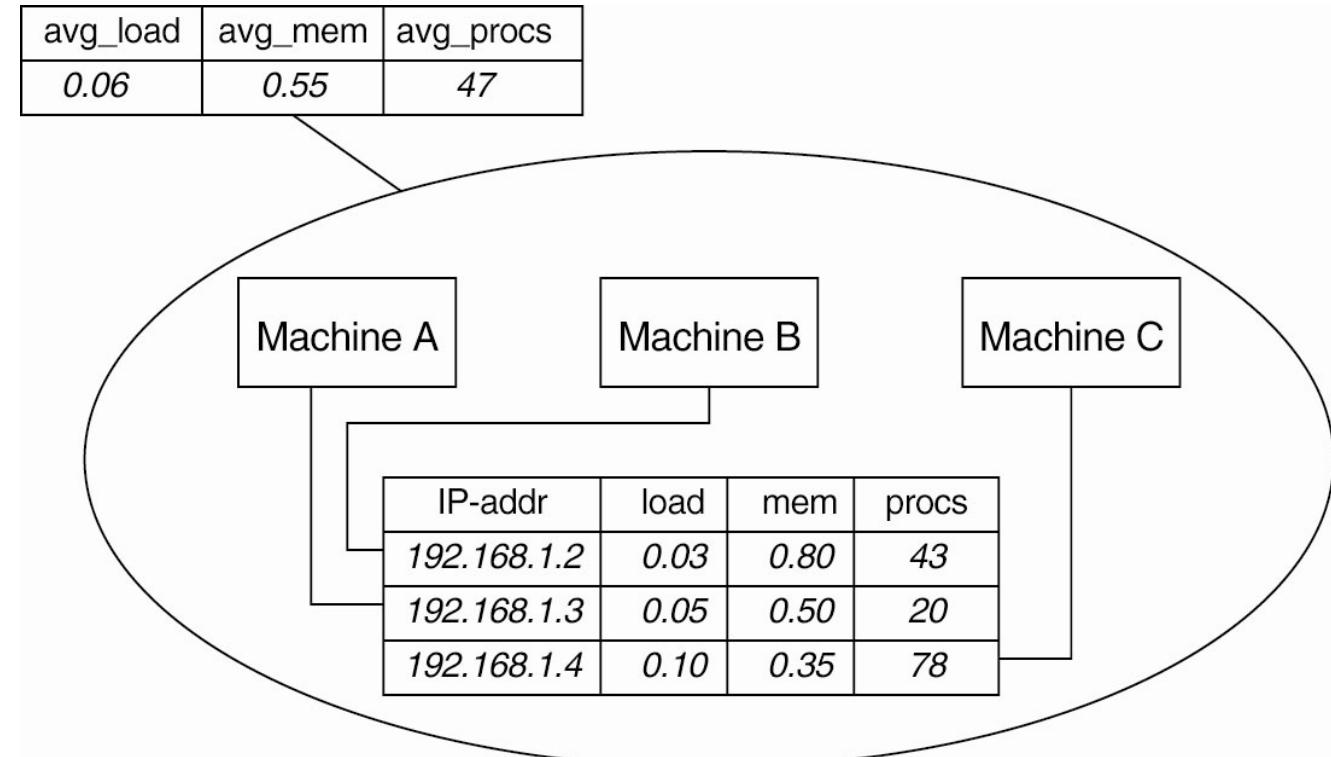
# THE FEEDBACK CONTROL SYSTEMS

- Allow automatic adaption to changes by means of one or more feedback control loops
  - self-managing, self-healing, self-configuration, self-optimization, etc.



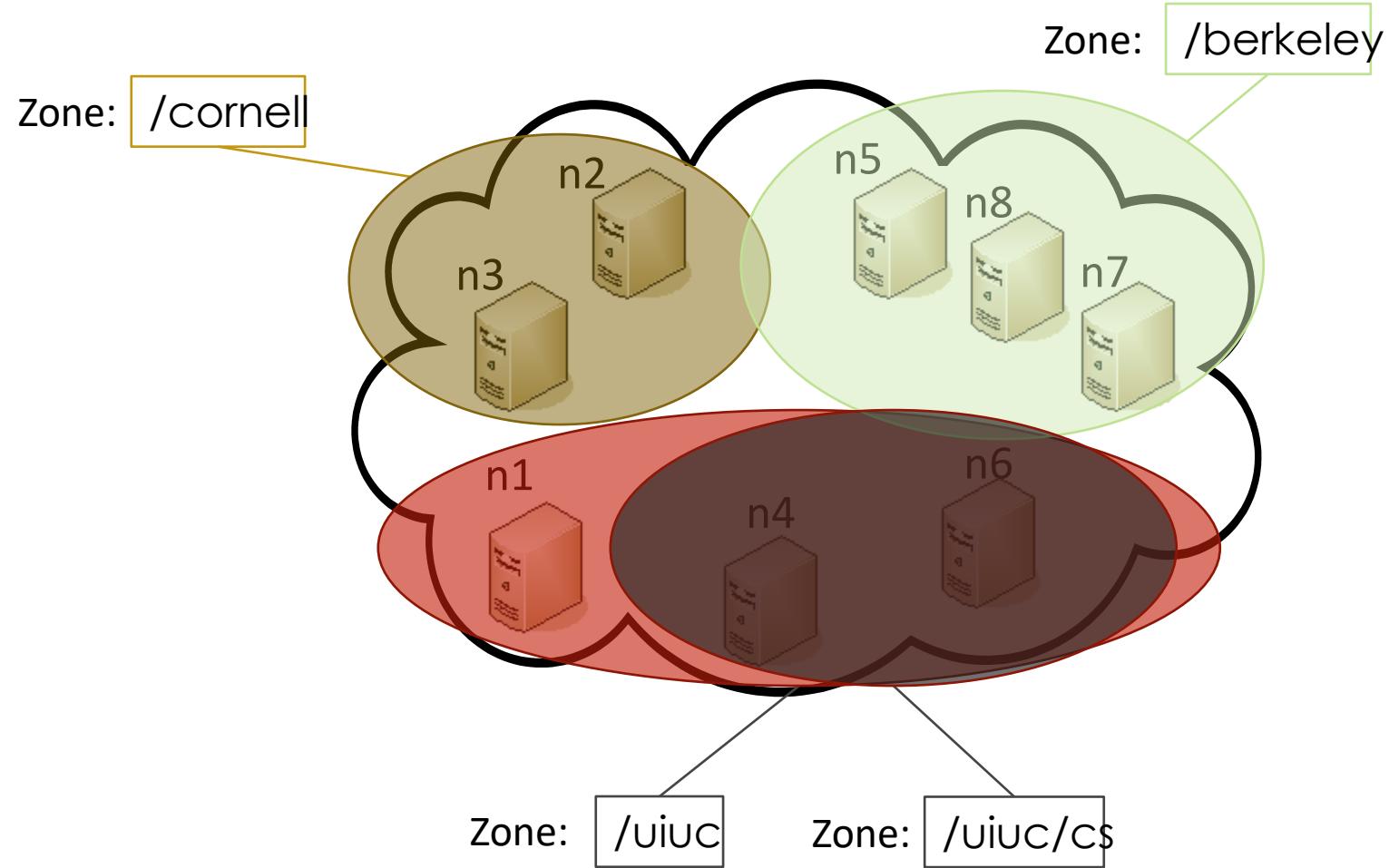
# SYSTEM MONITORING WITH ASTROLABE

- A tool for observing system behavior in a distributed system
  - One component of the feedback control system
- Every host runs a Astrolabe process, called an agent
  - Agents are organized as a hierarchy zones



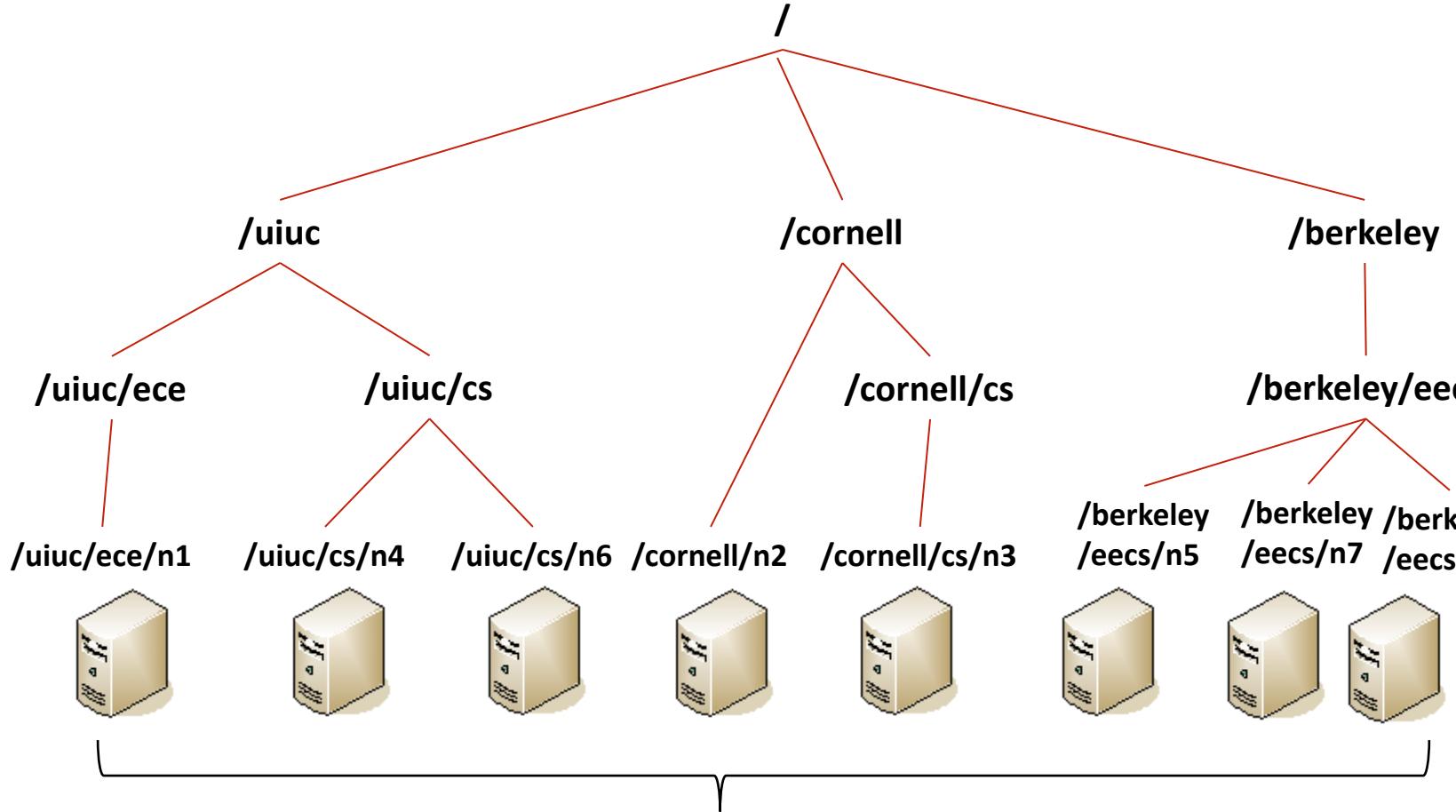
# SYSTEM MONITORING WITH ASTROLABE

- A tool for observing system behavior in a distributed system
  - One component of the feedback control system
- Every host runs a Astrolabe process, called an agent
  - Agents are organized as a hierarchy zones



# SYSTEM MONITORING WITH ASTROLABE

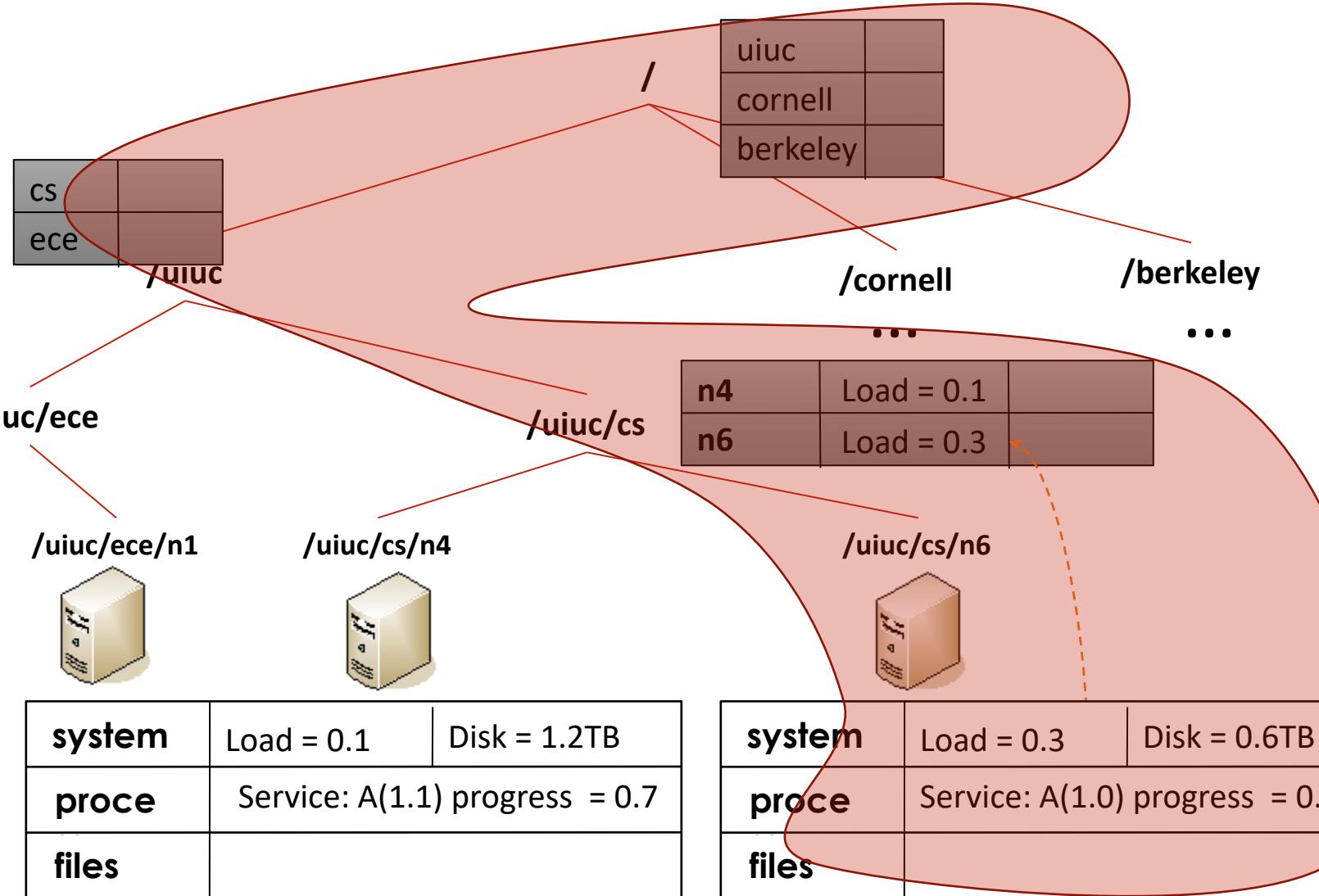
- Zone hierarchy is determined by the administrators (less flexibility).
- Assumption: zone names are consistent with the physical topology.



It's a virtual hierarchy. Only the host in leaf zone runs an Astrolabe agent.

# SYSTEM MONITORING WITH ASTROLABE

- Agent (/uiuc/cs/n6) has its local copy of
- these management table of MIBs.



# STATE AGGREGATION

