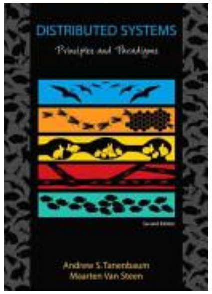
Chapter 2: ARCHITECTURES

Software architectures and System architectures

Logical organization and Physical realization



Thanks to the authors of the textbook [**TS**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.

Turgay Korkmaz
korkmaz@cs.utsa.edu

Chapter 2: ARCHITECTURES

- ARCHITECTURAL STYLES (SOFTWARE ARCHITECTURES)
- SYSTEM ARCHITECTURES
  - Centralized Architectures
  - Decentralized Architectures
  - Hybrid Architectures
- ARCHITECTURES VERSUS MIDDLEWARE
  - Interceptors
  - General Approaches to Adaptive Software
- SELF-MANAGEMENT IN DISTRIBUTED SYSTEMS
  - The Feedback Control Model
  - Example: Systems Monitoring with Astrolabe
  - Example: Differentiating Replication Strategies in Globule

Objectives

- To learn how to **organize** a distributed system whose components are dispersed across multiple machines
- To understand the differences between
  - **software** architecture (**logical** organization) and
  - **system** architecture (**physical** realization)
- To understand trade-offs when providing **distribution transparency**
- To understand adaptability and self-mng issues and mechanisms for flexibility and efficiency

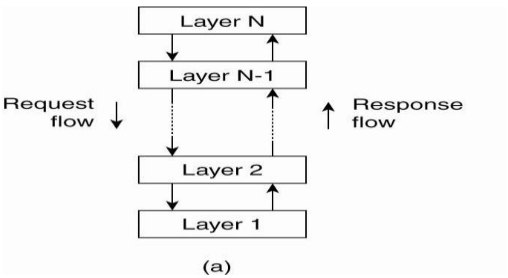
Software Architecture
(Architectural Style, Logical organization)

- Divide the system into *logically* different software components, distribute them over multiple machines, and allow them to communicate through connectors
  - Component: a modular unit with well-defined *required* and *provided* interfaces,
  - Connector: a mechanism that mediates communication, coordination, and cooperation (e.g., RPC, msg passing)
- Using components and connectors, we can create different configurations, which are classified into the following architectural styles:
  - Layered
  - Object-based
  - Event-based
  - Data-centered

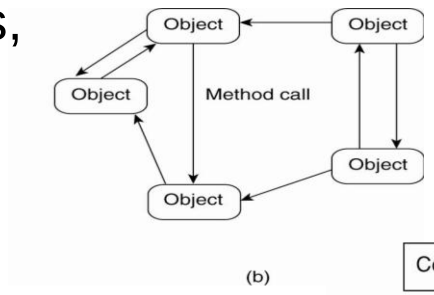
All try to achieve distributed transparency at a reasonable level
and
Each style would be more appropriate for a different application

Software Architecture
(Architectural Style, Logical organization)

- **Layered style**
  - used for client-server systems, request/reply model

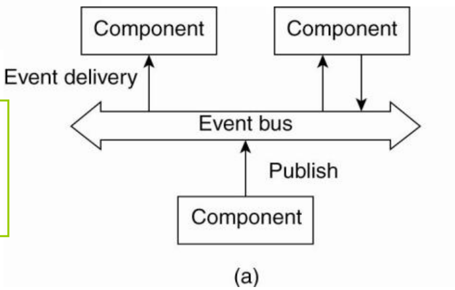


- **Object-based**
  - Used for distributed object systems, request/reply model

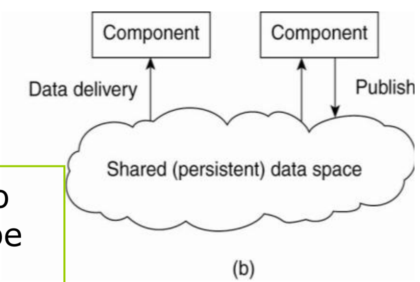


- **Event-based:**
  - Publish/subscribe systems
  - Loosely coupled components
    - decoupled in space or referentially decoupled

(processes do not need to refer to each other)



- **Data-centered:**
  - Communicate through common repository (e.g., shared distributed file system)
  - Can be combined with event-based, yielding shared dataspace
    - processes are now decoupled in space and time



(processes do not need to be active at the same time)

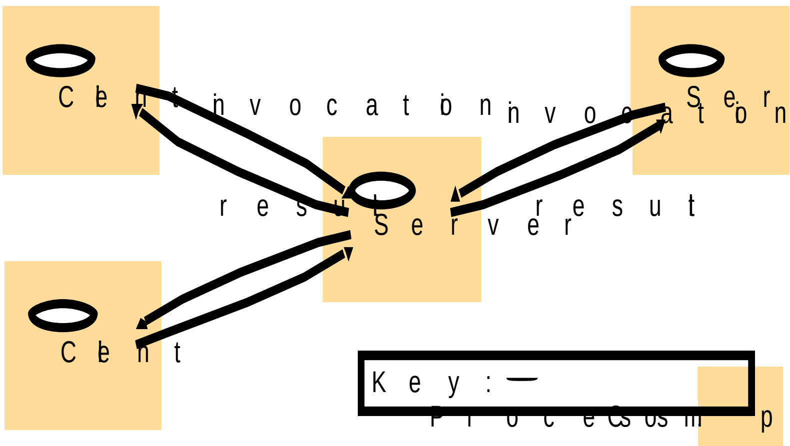
System Architecture
(Physical realization)

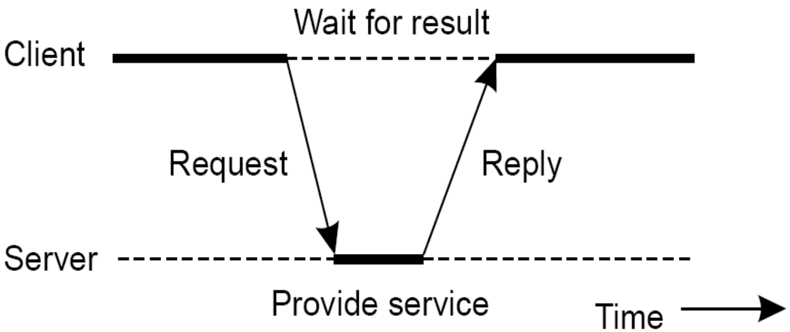
- Consider how and where to place software components and realize their interactions
- There are three major *physical realization* approaches:
  - Centralized — client-server
  - Decentralized — P2P (Structured vs. unstructured)
  - Hybrid: — combination of centralized and P2P

System Architecture: Client-Server

- There are processes offering services (servers)
- There are processes that use services (clients)
- Clients and servers can be on different machines
- Clients follow request/reply model to use services
- Connection-oriented vs. connectionless

(most use TCP vs UDP)

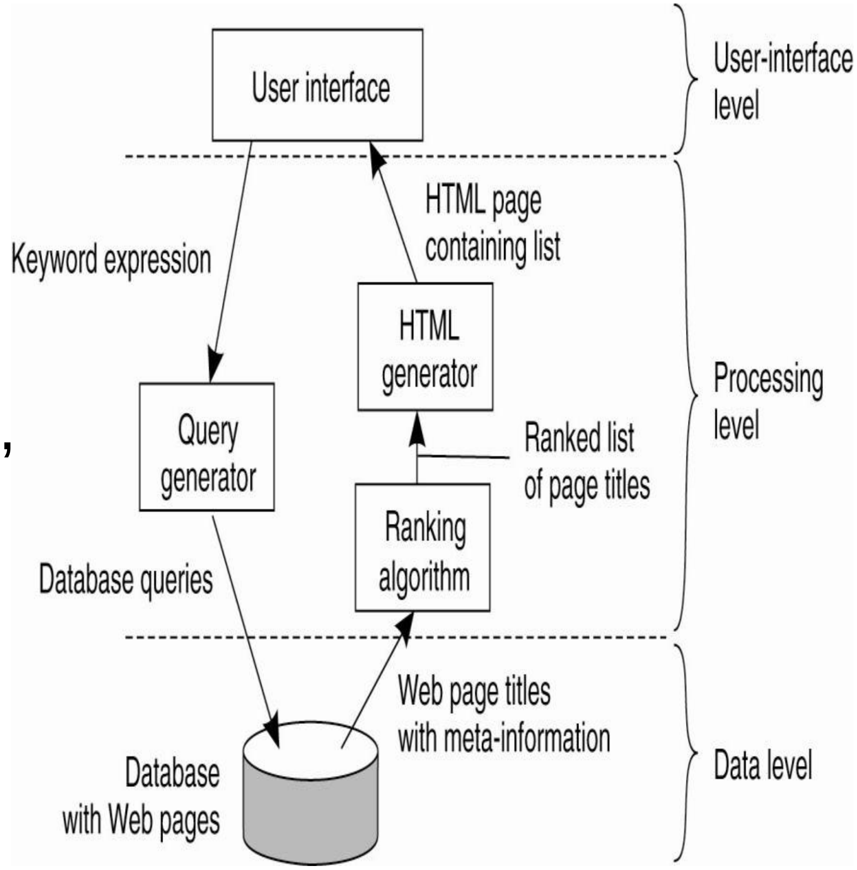




Application Layering (logical)

How to draw a clear line between client end server?

- User-interface layer
  - units for an application's user interface
- Processing layer:
  - functions of an application, i.e. without specific data
- Data layer:
  - data that a client wants to manipulate through the application components

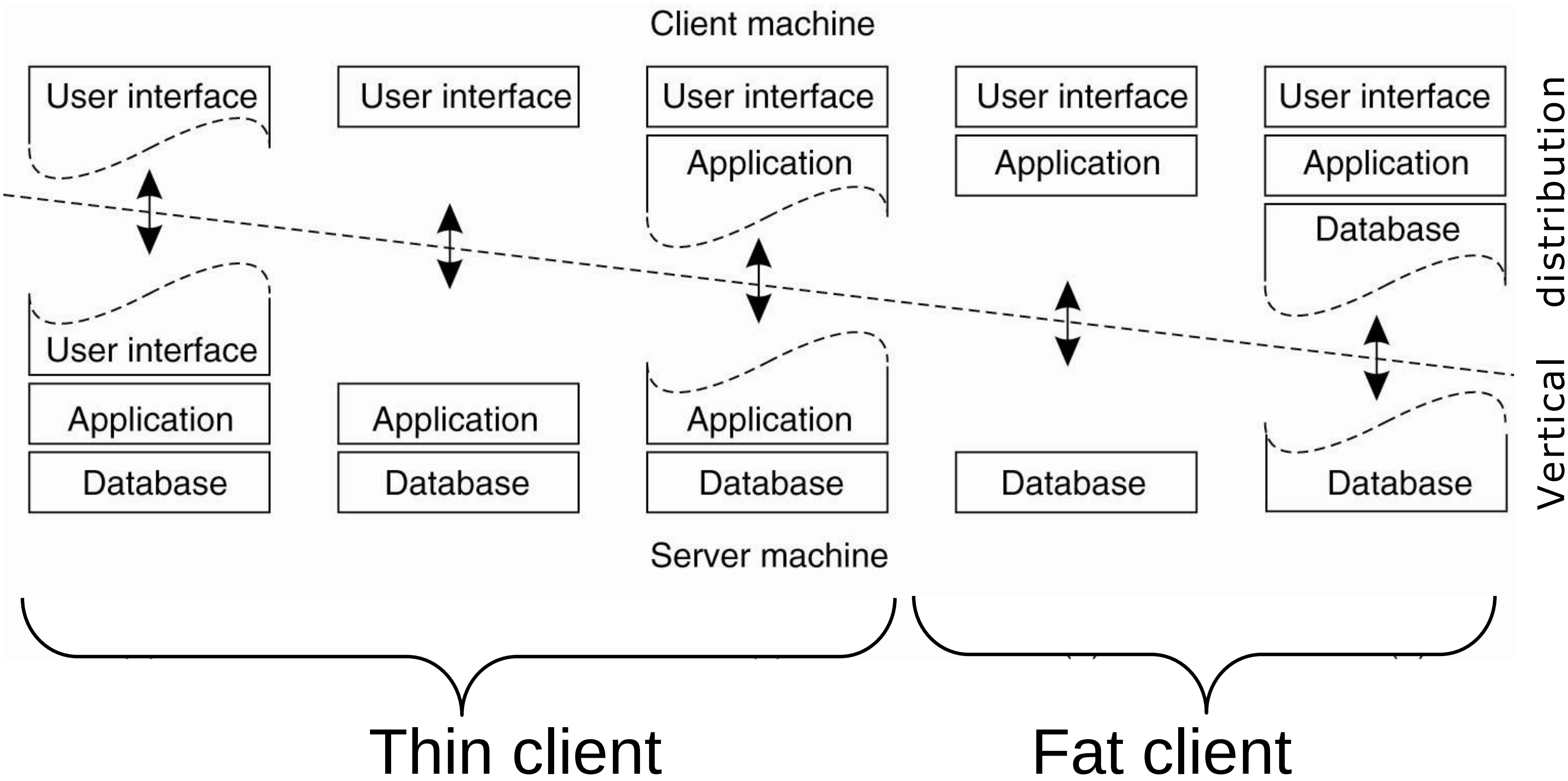


Observation: layering is found in many distributed information systems, using traditional database technology and accompanying applications.

Multitiered Architectures (physical realization)

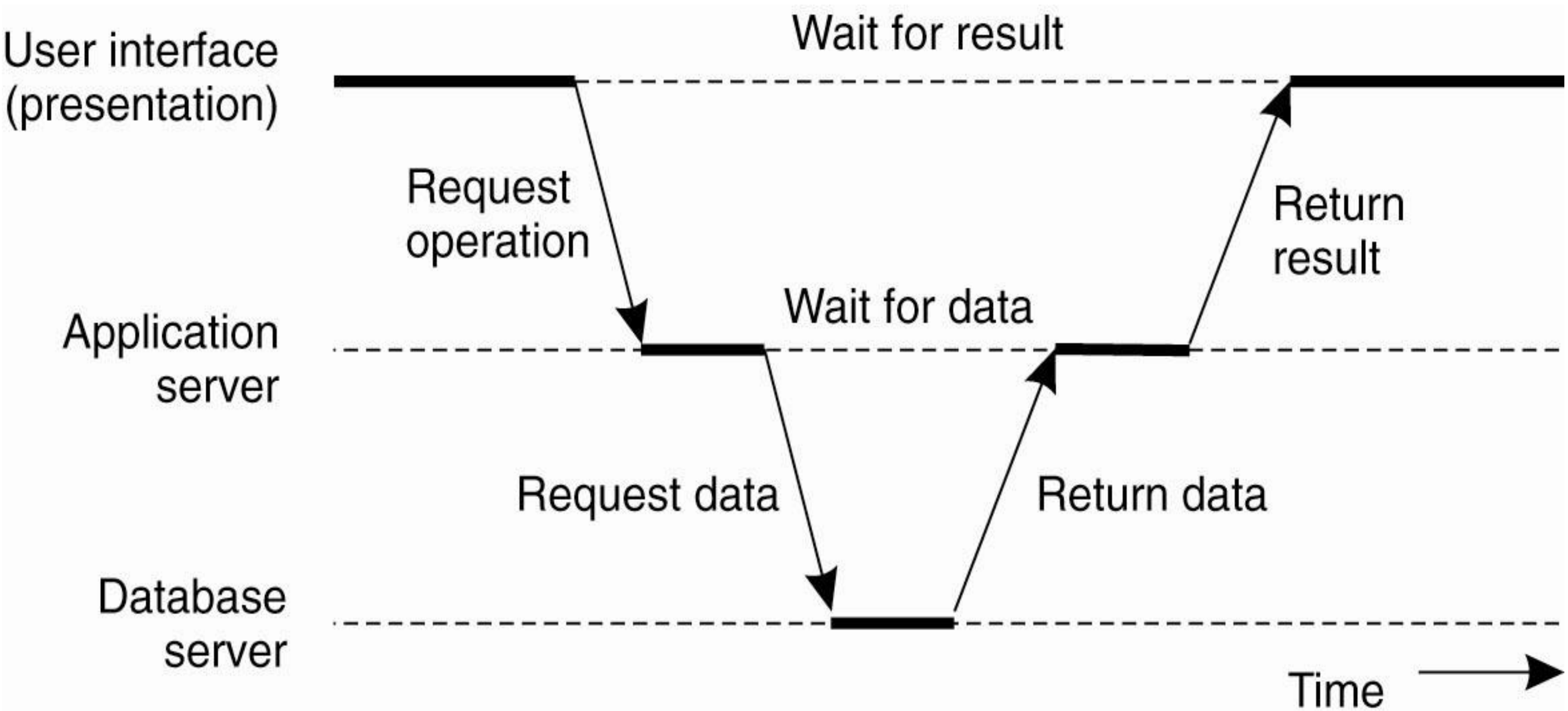
How to place the three layers on client and server?

Single-tiered: dumb terminal/mainframe configuration
Two-tiered: client/single-server configuration



Multitiered Architectures (physical realization)

- The server part could be distributed over multiple machines,
- Three-tiered: each layer on separate machine

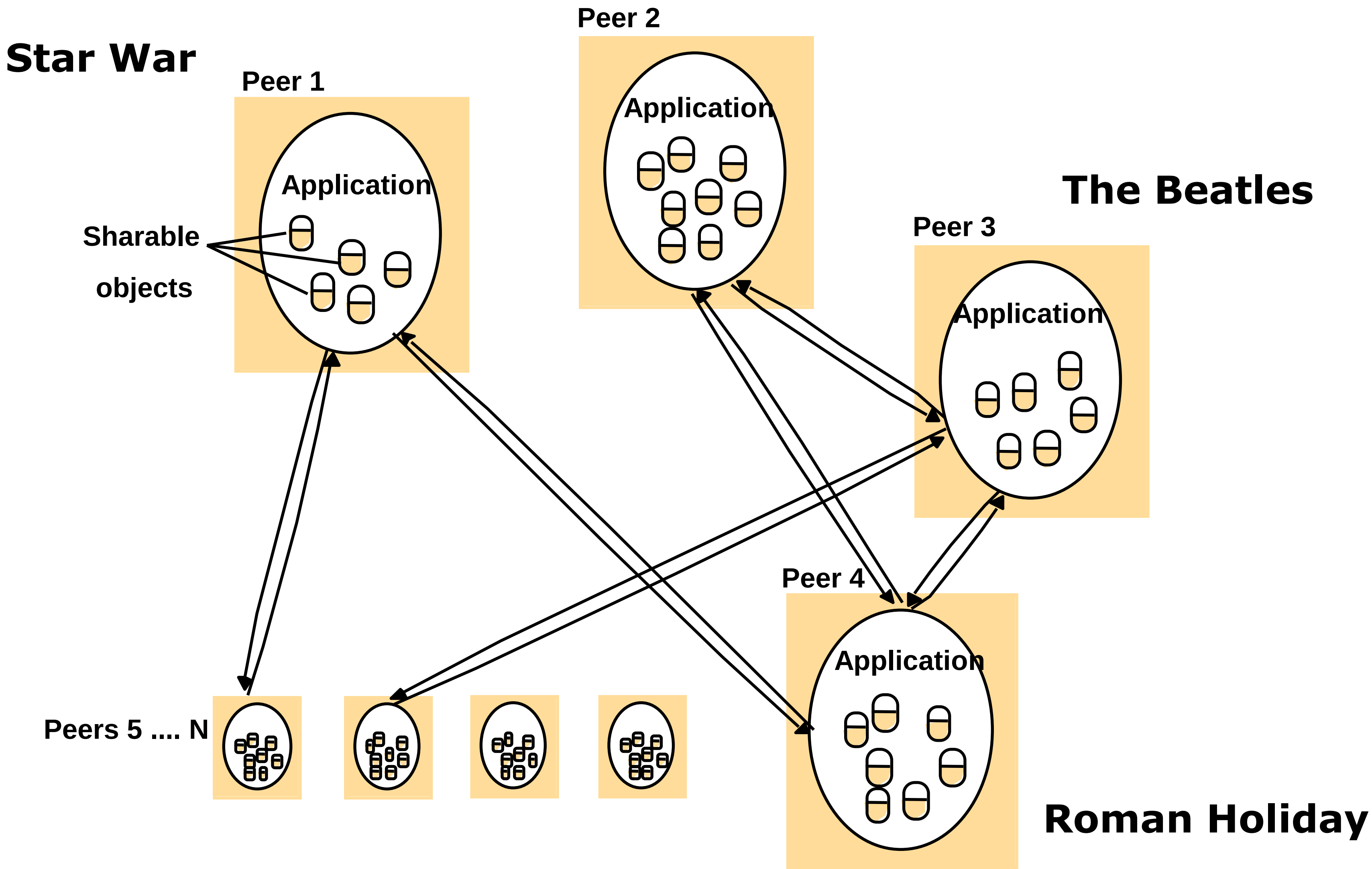


Decentralized Architectures: P2P Systems

- Multitiered architectures do vertical distribution:
- We place logically different components in client-server *(i.e., user interface, processing, data)* on different machines
- Processes are not equal and Interactions are asymmetric
  - One acts as client while the other acts as server
- Traditional approach

- P2P architectures do horizontal distribution:
- We split up clients and servers into logically equivalent parts and let each part operate on its own share
- Processes are equal and Interactions are symmetric
  - Each acts as both client and server
- Tremendous growth in the last couple of years

Decentralized Architectures: P2P Systems



Decentralized Architectures: P2P Systems

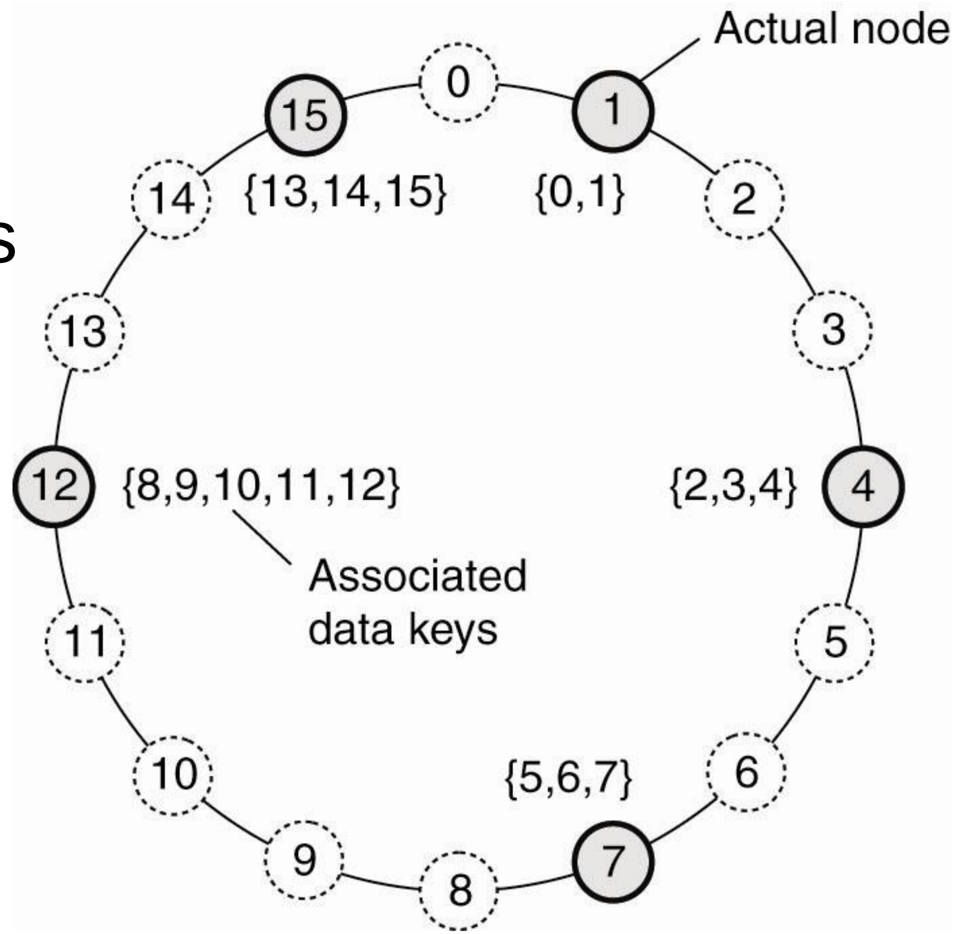
- Given the symmetric behavior, the key question is how to organize processes in an overlay network, where links are usually TCP channels…
- How about fully connected overlay network? -/+
- There are three approaches to organize nodes into overlay networks through which data is routed
  - **Structured P2P**: nodes are organized following a specific distributed data structure and deterministic algorithms
  - **Unstructured P2P**: randomly selected neighbors
  - **Hybrid P2P**: some nodes are appointed special functions in a well-organized fashion

Structured P2P Systems

- Distributed Hash Table (DHT) is the most used one
  - Assume we have a large ID space Ω (e.g., 128-bit)
  - Assign random keys from Ω to data items
  - Assign random identifiers from Ω to nodes
  - The crux of every DHT is to implement an efficient and deterministic scheme that maps the key of a data item to node ID
  - When looking up a data item, the system should route the request to the associated node and return the network address of that node
- Example: Chord

A DHT Example: Chord

- Chord organizes the nodes in a structured overlay network such as a logical ring, and data item with key *k* is mapped to a node with the smallest *ID* >= *k*.
- This node is called as the successor of key *k* and denoted by *succ(k)*



LOOKUP(key=8) ?

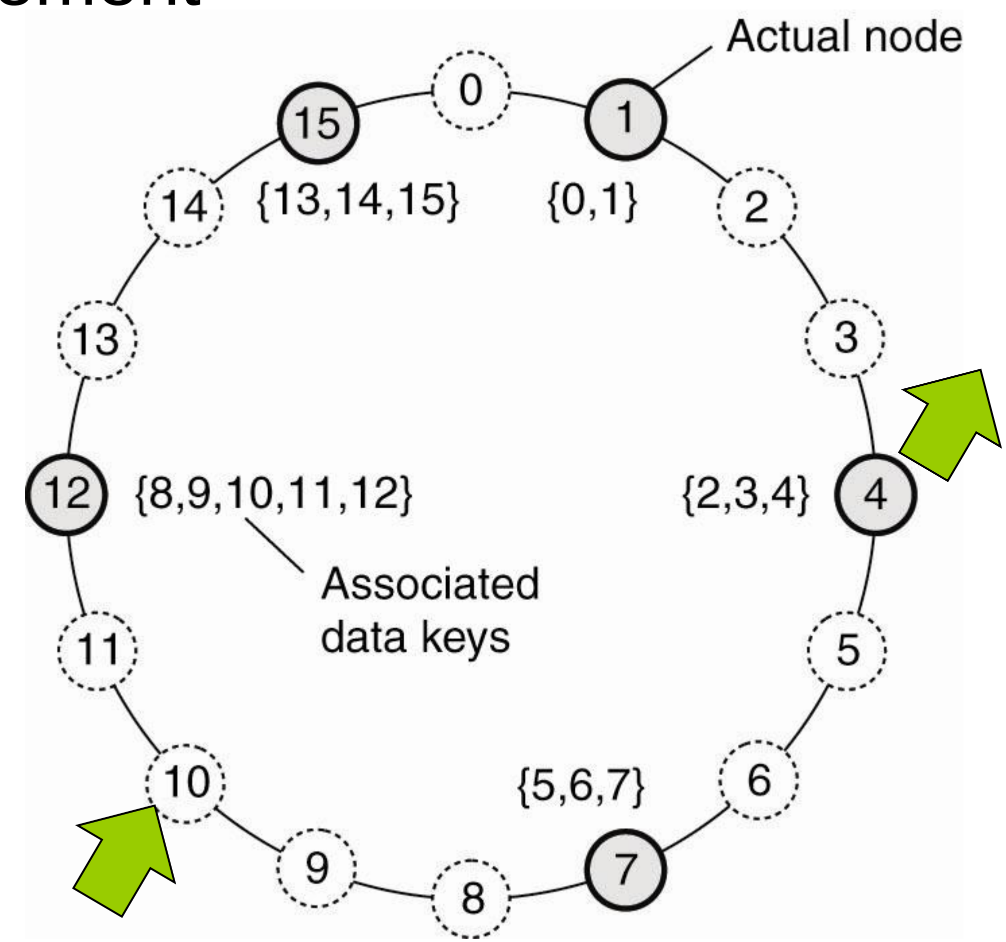
This should return *succ(8)* which is node 12.

(Details of how this is done is in Ch 5)

A DHT Example: Chord

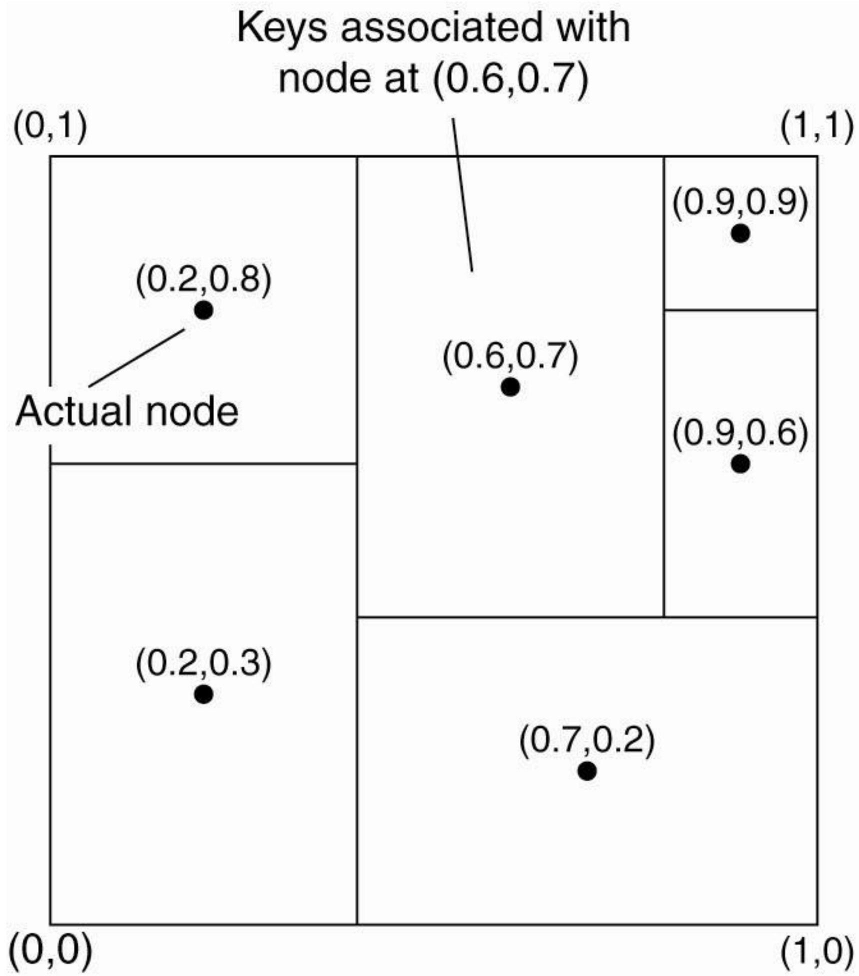
- Membership management
  - Join
  - Leave
- Lookup(key)

(search and routing is in Ch 5)



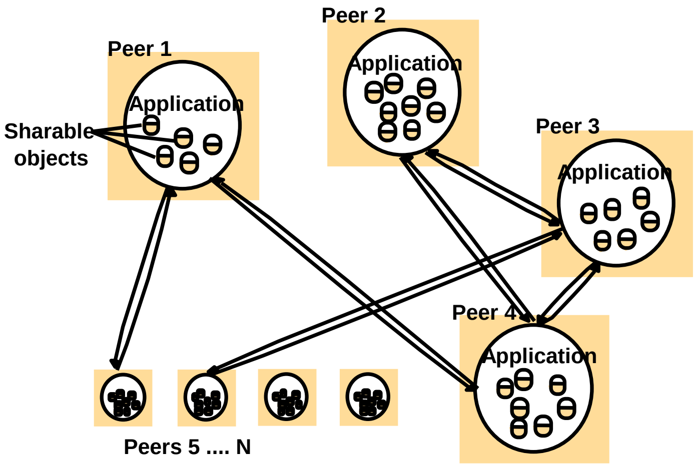
Another DHT Example: CAN

- Content Addressable Network
- Organize nodes in a *d*-dimensional space and let every node take the responsibility for data in a specific region.
- When a node joins, split a region.
- When a node leaves, merge regions.



(a)

Unstructured P2P Systems



- Maintain a random graph
- Data items are randomly placed on nodes
- How to do Lookup?
  - flooding
- Membership management
  - Join
    - Get a random list (from a well-known list or server)
    - Contact these nodes and run the algorithm presented next
  - Leave
    - Easy just leave…

How to maintain random graph

- Let each peer maintain a partial view of the network, consisting of c other nodes
- Each node P periodically selects a node Q from its partial view
- P and Q exchange information and exchange members from their respective partial views

Actions by active thread (periodically repeated):

```
select a peer P from the current partial view;
if PUSH_MODE {
    mybuffer = [(MyAddress, 0)];
    permute partial view;
    move H oldest entries to the end;
    append first c/2 entries to mybuffer;
    send mybuffer to P;
} else {
    send trigger to P;
}
if PULL_MODE {
    receive P's buffer;
}
construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

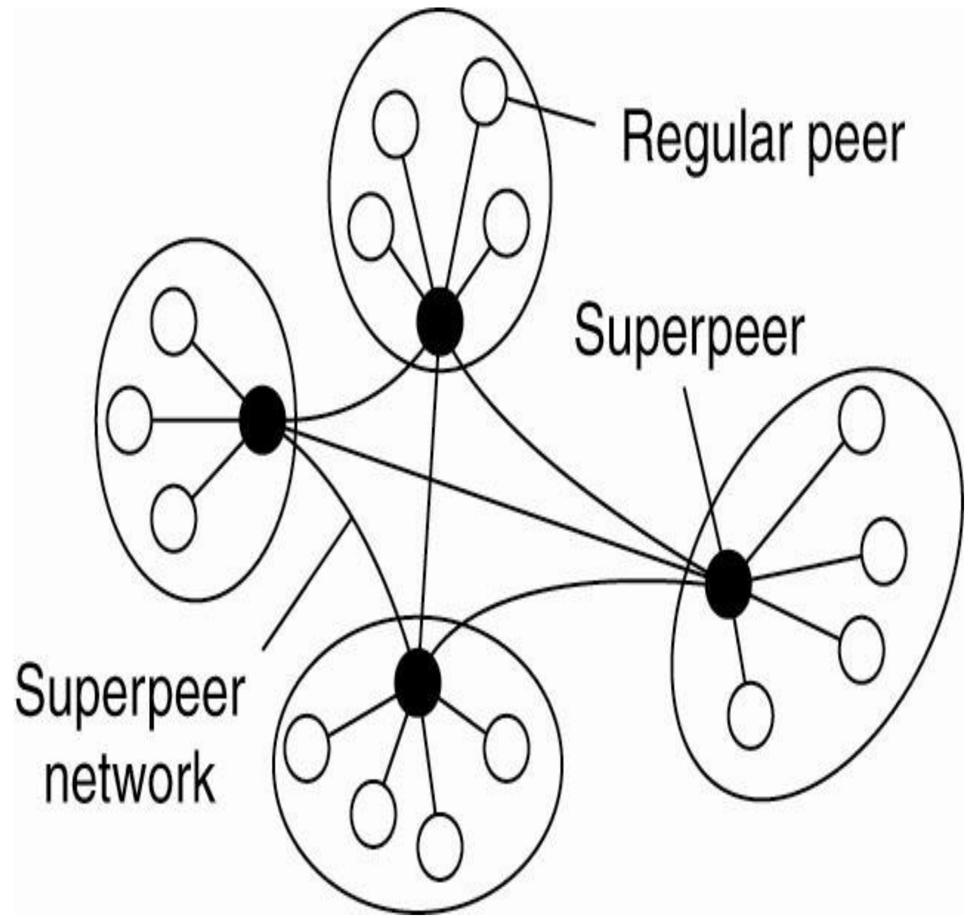
Actions by passive thread:

```
receive buffer from any process Q;
if PULL_MODE {
    mybuffer = [(MyAddress, 0)];
    permute partial view;
    move H oldest entries to the end;
    append first c/2 entries to mybuffer;
    send mybuffer to P;
}
construct a new partial view from the current one and P's buffer;
increment the age of every entry in the new partial view;
```

(b)

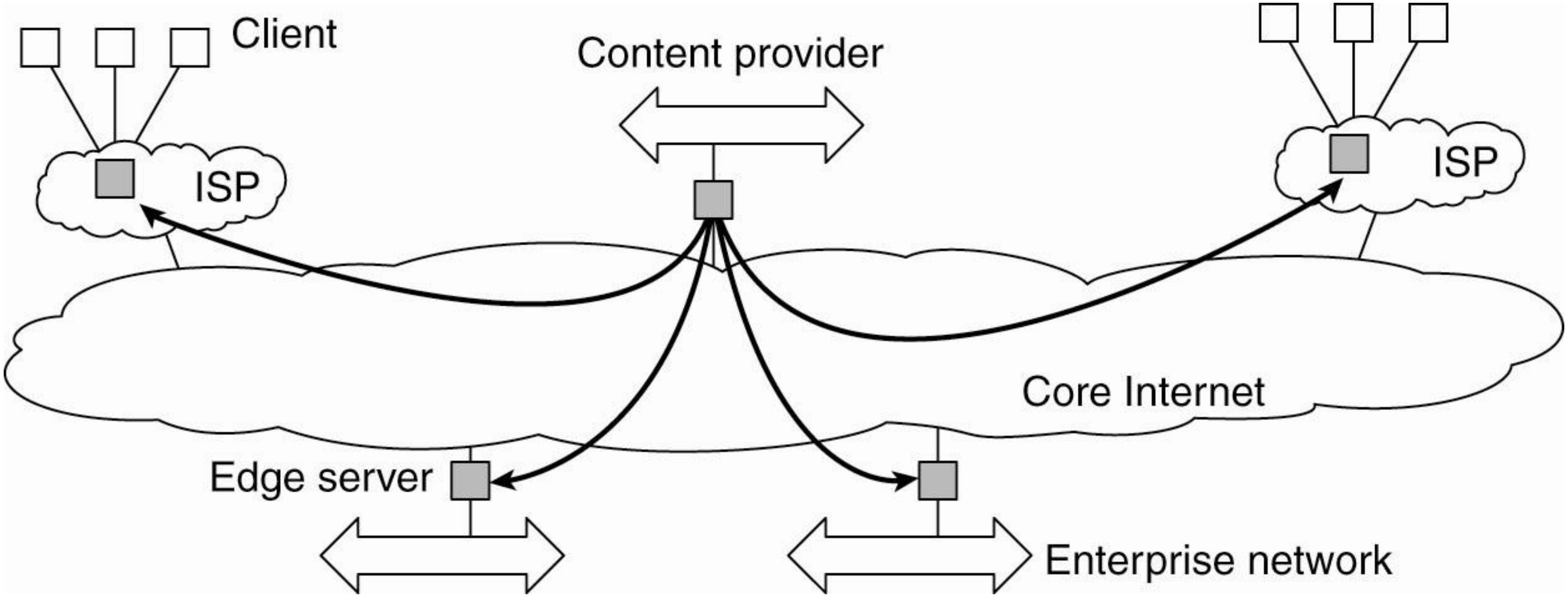
Superpeers

- When random graph gets bigger, it will be very hard to perform look up
- Use supperpeers to maintain an index
- Join/leave is easy
- How about Lookup?
- Regular peers may elect the supperpeer (Ch 6)



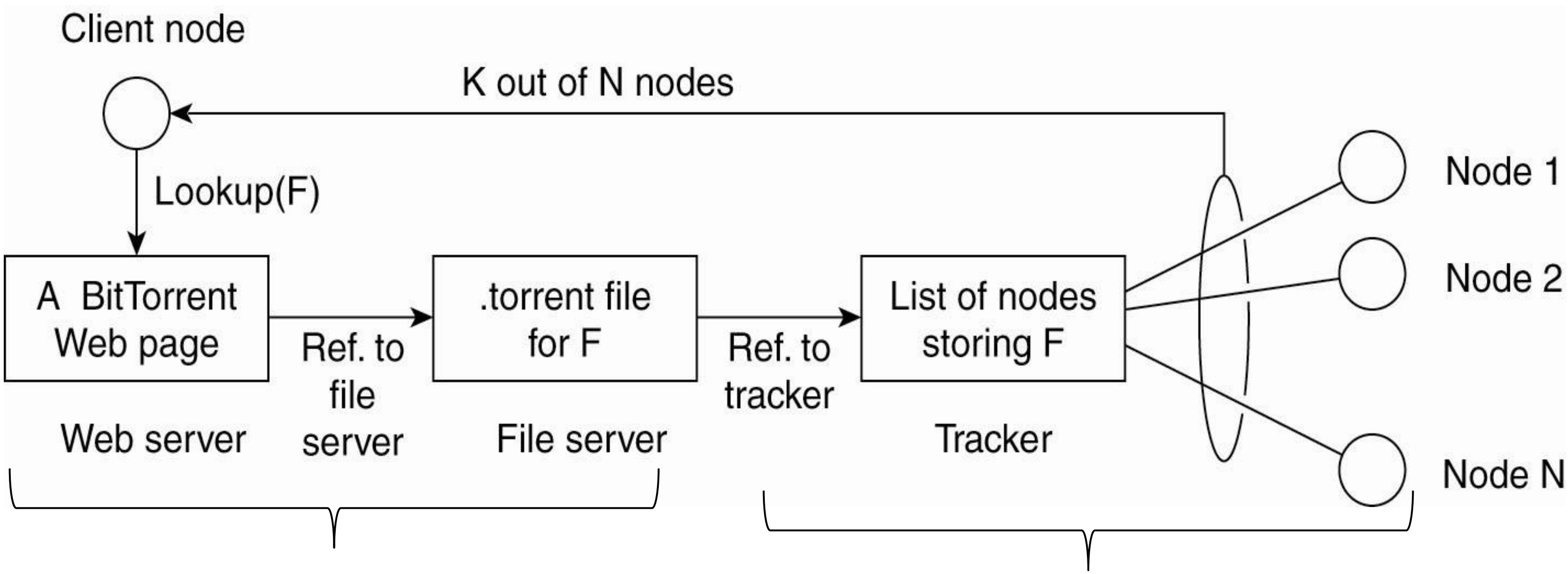
Hybrid Architectures: Edge-server systems

- Content Distribution Network (CDN)
- Edge servers can be used to optimize content distribution



Hybrid Architectures: Collaborative Distributed Systems

- Combining a P2P with a client-server architecture
- **Basic idea:** a node identifies where to download a file from and joins a **swarm** of downloaders; who get file chunks in parallel from the source, and distribute these chunks amongst each other



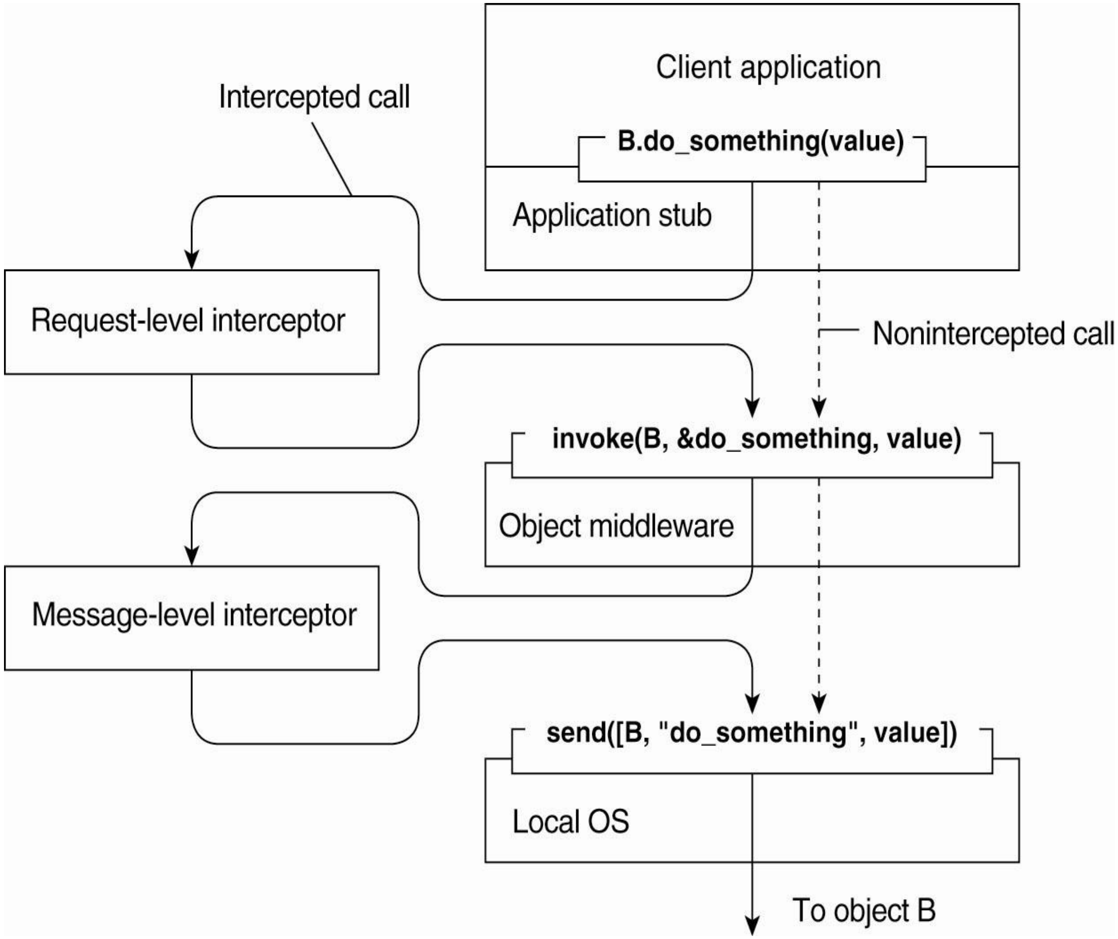
WHERE MIDDLEWARE FITS IN ALL THESE ARCHITECTURES?

Architectures Vs. Middleware

- Middleware is between application and local OS and provides some degree of transparency
- In practice, middleware systems follow a specific architectural style (software architecture, logical organization):
  - Layered
  - Object-based
  - Data centered
  - Event-based
- Adv/DisAdv
  - + makes app design simple
  - - may not be optimized for what an app needs
  - - adding more features complicates the middleware
    - CORBA was initially object-based, later added msg passing
- Middleware should be adaptable to applications
  - Several different versions, configurable, separate policy and mechanisms

How to achieve adaptability?

- Interceptors: a software construct that will break the usual flow of control and allow other (app specific) code to be executed



General Approaches to Adaptive Software

- In many cases, distributed systems/applications are developed according to a specific architectural style. The chosen style may not be optimal in all cases Then there is a need to (dynamically) adapt the behavior of the middleware.
- Three basic approaches to adaptive software:
  - **Separation of concerns**:
    - Try to separate extra functionalities and later glue them together into a single implementation → aspect-oriented SW ,only toy examples so far.
  - **Computational reflection**:
    - Let a program inspect itself at runtime and adapt/change its settings dynamically if necessary → mostly at language level and applicability unclear.
  - **Component-based design**:
    - Organize a distributed application through components that can be dynamically replaced when needed (complex for DS, components are not independent)
- Do we really need adaptive software or adaptive system that reacts to changes (self-management)

Do we really need adaptive software?

- Software should expect all the environment changes and should have code in it to handle them
- DS should be able to react to changes in environment by switching policies or mechanisms in the system
- The challenge is how to achieve this reactive behavior without human intervention

Self-managing Distributed Systems

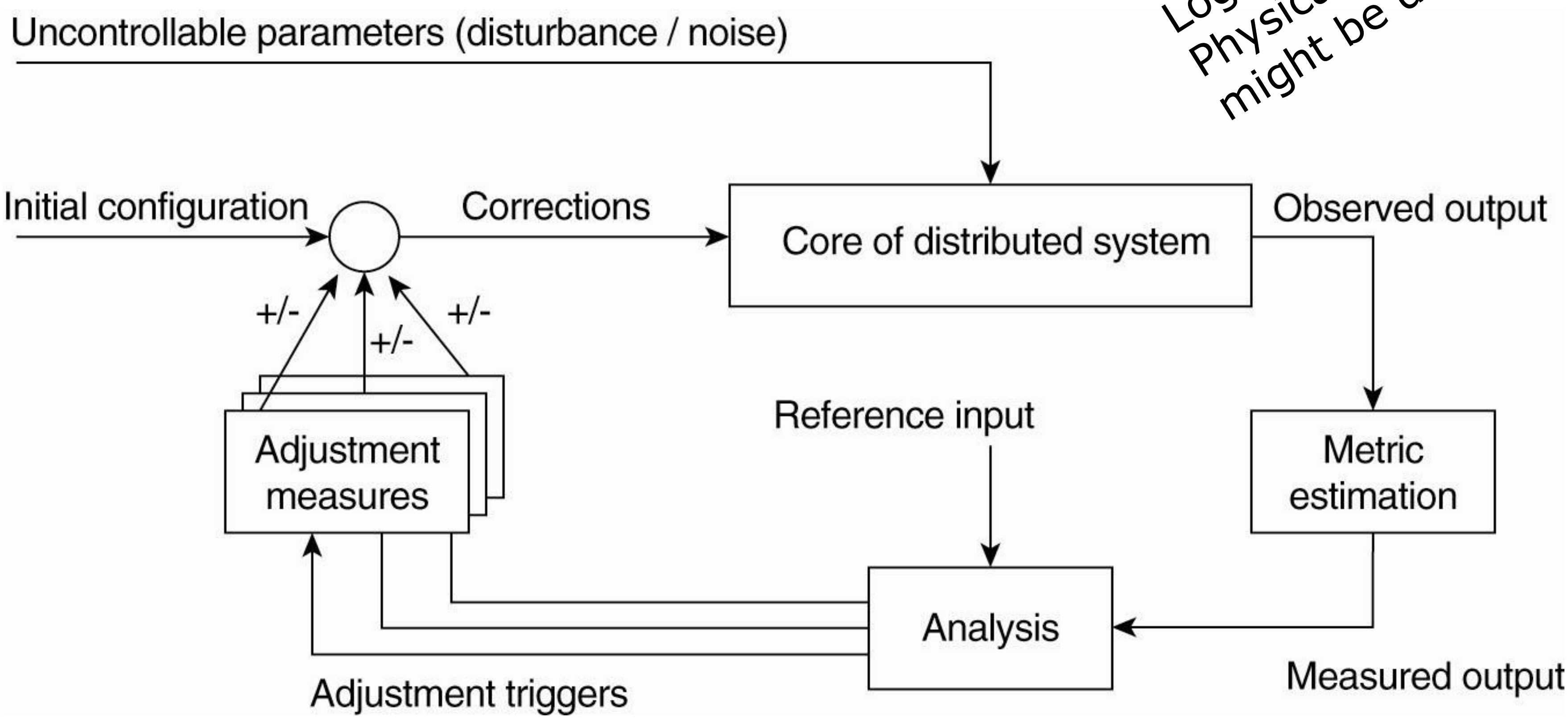
- Distinction between system and software architectures blurs when automatic adaptivity needs to be taken into account:
- Self-configuration
- Self-managing
- Self-healing
- Self-optimizing
- Self-*

Warning
There is a lot of hype going on in this field of autonomic computing.

Feedback Control Model

- In many cases, self-* systems use a feedback control loop.

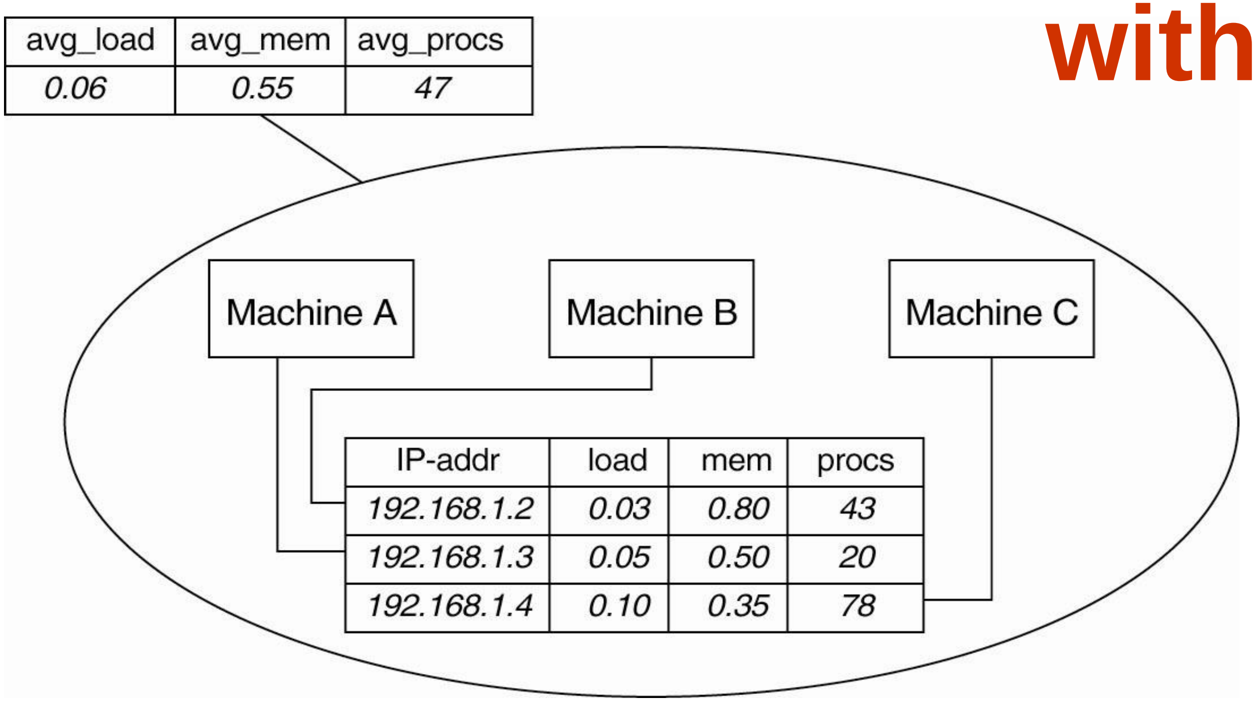
Logical view!
Physical realization
might be distributed



if time permits

FEEDBACK CONTROL EXAMPLES

Example: Systems Monitoring with Astrolab



A general tool for observing system behavior

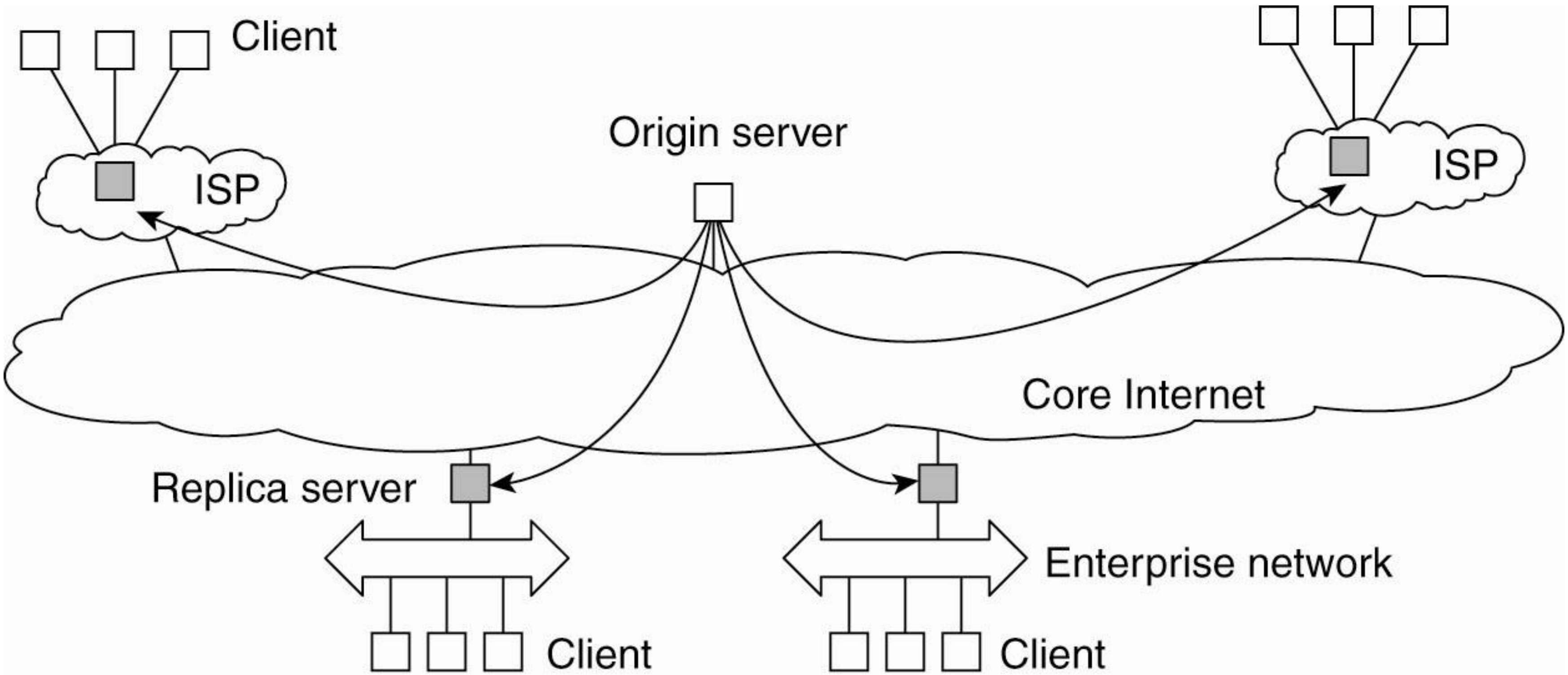
Organize hosts into a hierarchy of zones.

Collect information about each host and summarize it,

Exchange this information so all agents will see the same view.

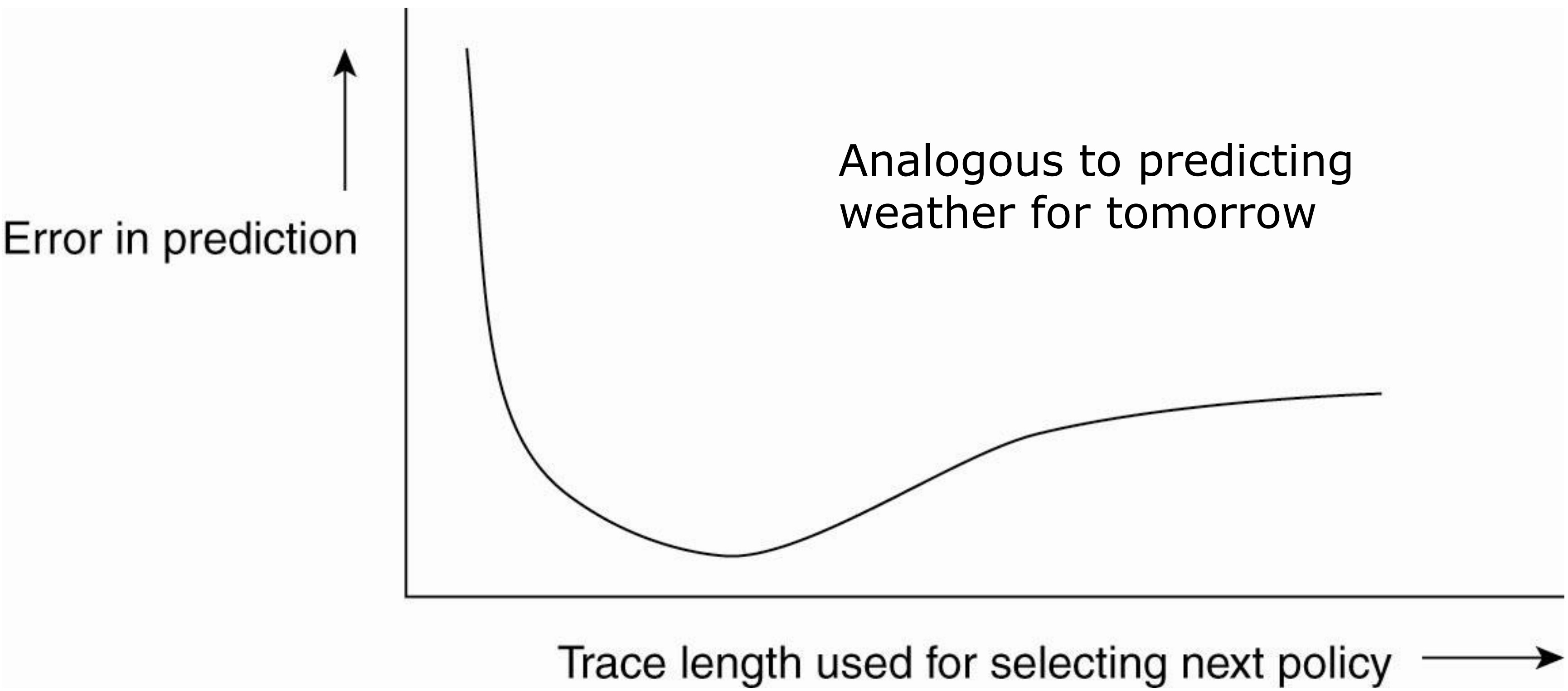
Example: Differentiating Replication Strategies in Globule (1)

- A collaborative CDN tries to minimize performance by replicating web pages.



Example: Differentiating Replication Strategies in Globule (2)

- Figure 2-19. The dependency between prediction accuracy and trace length.



Example: Automatic Component Repair Management in Jade

- Steps required in a repair procedure:
  - Terminate every binding between a component on a nonfaulty node, and a component on the node that just failed.
  - Request the node manager to start and add a new node to the domain.
  - Configure the new node with exactly the same components as those on the crashed node.
  - Re-establish all the bindings that were previously terminated.