高级操作系统

Advanced Operating System

Distributed Systems

Principles and Paradigms

朱 青 Qing Zhu,

Department of Computer Science, Renmin University of China

zqruc2012@aliyun.com

高级操作系统

Advanced Operating System

Distributed Systems

Concepts and design

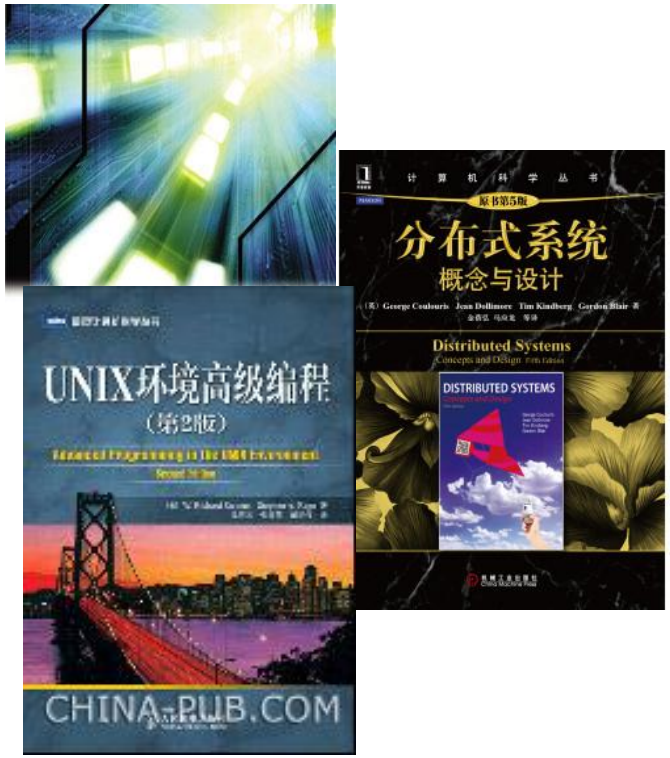
朱 青 Qing Zhu,

Department of Computer Science, Renmin University of China

zqruc2012@aliyun.com

Chapter 2

Architectural Models　系统架构模型



朱　青

信息学院计算机系，

中国人民大学，

zqruc2012@aliyun.com

Chapter 2
Architectural Models 系统模型

Teacher: Qing Zhu 朱 青

Department of Computer Science,
Information School,
Renmin University of China
zqruc2012@aliyun.com

第2章 体系结构

- 2.0 简介
- 2.1 软件体系结构样式
- 2.2 系统体系结构
- 2.3 体系结构与中间件
- 2.4 分布式系统的自我管理
- 2.5 基础模型
  - 交互模型
  - 故障模型
  - 安全模型

2.0 简介

- 物理模型：考虑组成系统的计算机和设备、以及它们之间的互联（硬件组成）。
- 体系结构模型：从系统的计算元素执行的计算、通信任务来描述。
  - 客户服务器
  - 对等模型
- 基础模型：采用抽象的观点描述系统的某个方面。
  - 交互模型
  - 故障模型
  - 安全模型

体系结构模型

- 分布式系统逻辑组织——软件体系结构样式
- 分布式系统物理组织——系统体系结构
- 体系结构与中间件——系统透明性
- 分布式系统的自我管理——系统可适应性
- 基础模型 ——分布式系统重要模块

物理模型

- 1. 早期的分布式系统
  - 局域网 + 以太网
  - 通过局域网连接的10~100个结点

- 2. 互联网规模的分布式系统
  - 大规模的分布式系统，例如：Google在1996年第一次发布；
  - 通过互联网连接，跨组织提供分布式系统服务
  - 特点：结点通常是台式机，静态、分立和自治的

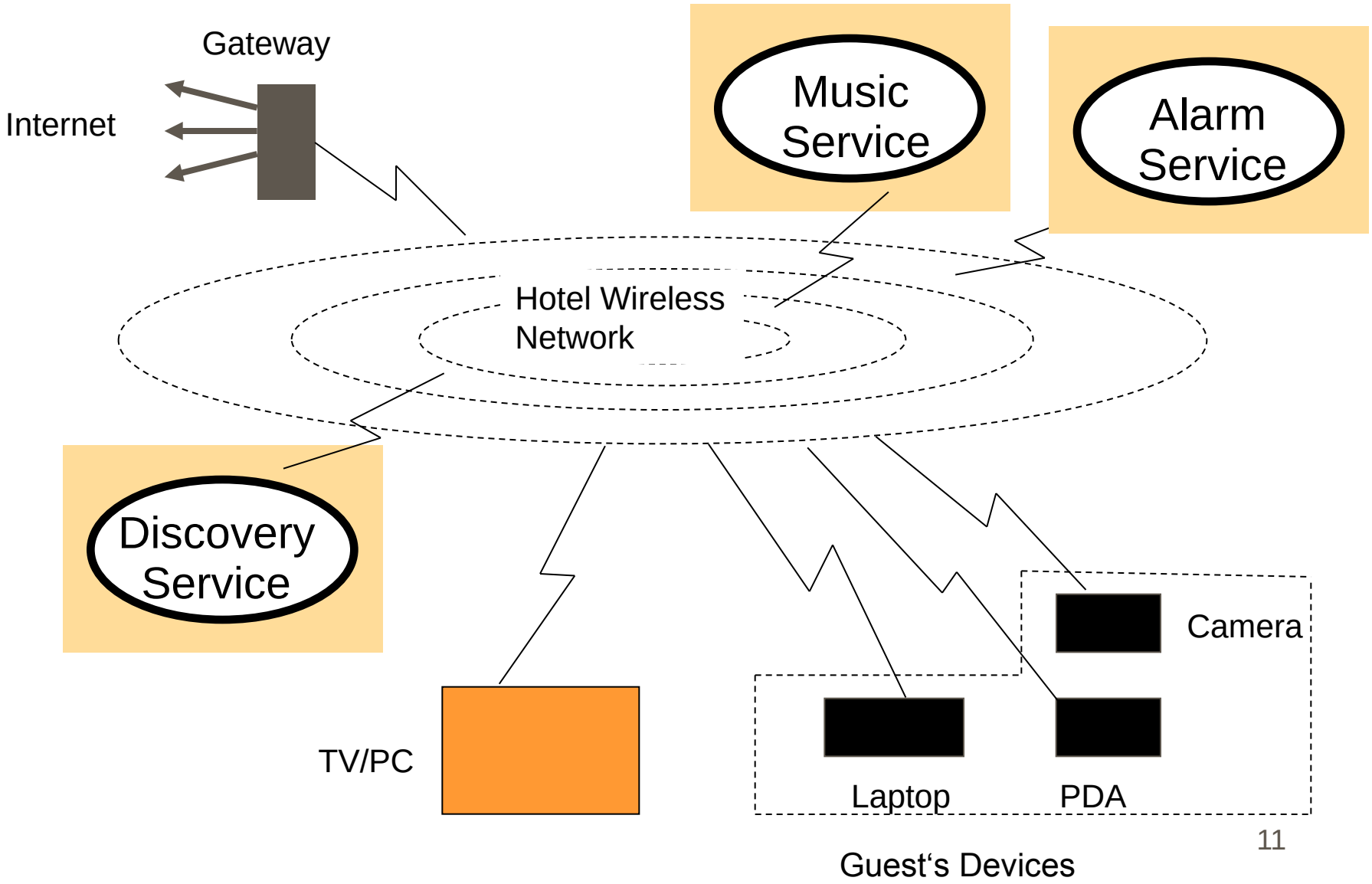
当代的分布式系统

- 移动计算的出现导致物理模型的变化
- 无处不在的计算导致体系结构从分离结点转向嵌入系统。
- 云计算与集群体系结构从自治结点完成任务转向一组结点提供给定的服务。
- 系统的复杂性——系统的分布式系统，复杂、异构。
- 例如：洪水测试的环境管理系统。

Figure 2.1
Generations of distributed systems 分布式系统分代

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

Example of Spontaneous Networking



Architectural Model体系结构模型

The architectural model of a distributed system addresses
分布式系统体系结构模型涉及:

- the placement of its components组件的位置,
  - distribution of data and workload数据与工作量的分布
- the relationships between them.
  - functional roles and communications patterns 角色和通信模式

Examples of architectural models:

- Client/server model,
- Peer process model

Architectural Model

Modifications of the client/server model:

- **Partition of data or replication of data at cooperating servers**
  - 在合作的服务器上进行数据的分区或复制。
- **Caching of data by proxy servers and clients**
  - 由代理服务器和客户进行数据缓存。
- **Use of mobile code and mobile agents**
  - 使用移动代码和移动代理。
- **Methods to add and remove mobile devices**
  - 以便利的方式增加和删除移动设备。

分布式系统的困难和威胁

- 使用模式的多样性：
  - 系统组件承受各种工作负载
    - 例如：Web每天有几百万的访问量
  - 系统断线或连接不稳定
    - 例如：系统中包括移动计算机
  - 系统对带宽与延迟的特殊要求
    - 例如：多媒体应用

分布式系统的困难和威胁

- 系统环境的多样性：
  - 容纳异构硬件、操作系统和网络
  - 网络在性能上有很大的不同
    - 例如：无线网的速度只达到局域网的几分之一
  - 支持不同规模的系统
    - 从几十台计算机到几百万台计算机

分布式系统的困难和威胁

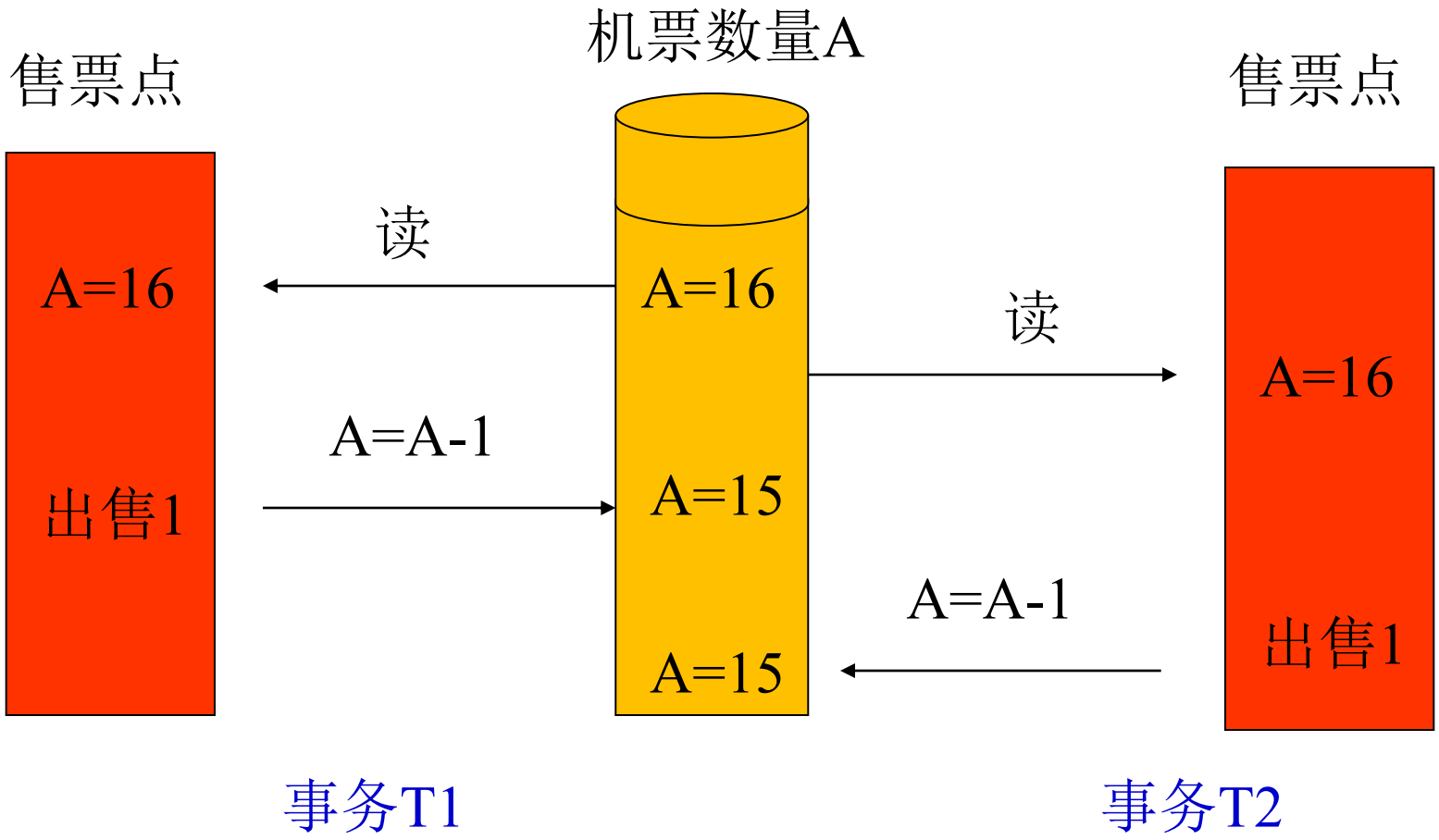
- 内部问题：
  - 非同步时钟
  - 冲突的数据更新
  - 系统组件的软硬件故障
- 外部问题：
  - 数据完整性
  - 保密性的攻击
  - 服务拒绝攻击

并发引起的数据不一致——内部问题

一个最常见并发操作的例子是飞机订票系统中的订票操作。例如在该系统中的一个活动序列：

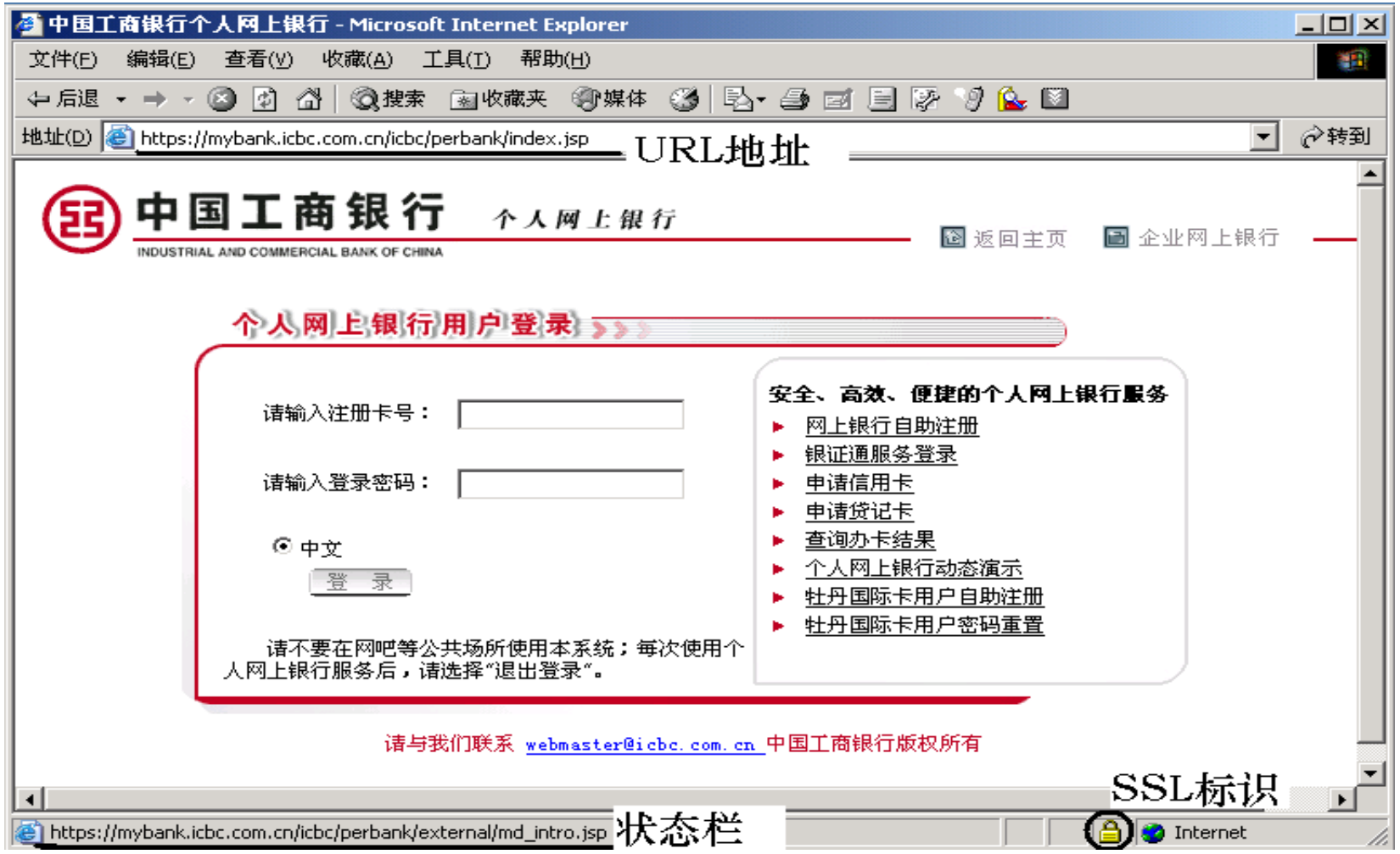
- ①甲售票点（甲事务）读出某航班的机票余额A，设A＝16；
- ②乙售票点（乙事务）读出同一航班的机票余额A，也为16；
- ③甲售票点卖出一张机票，修改余额A　A-1，所以A为15张，把A写回数据库；
- ④乙售票点也卖出一张机票，修改余额A　A-1，所以A为15张，把A写回数据库。
- 结果明明卖出两张机票，数据库中机票余额只减少1。
- 这种情况称为数据库的不一致。这种不一致是由并发操作引起的。

并发引起内部问题



⌘这种数据库的不一致是由并发操作引起的

Web欺骗及防范技术——外部问题



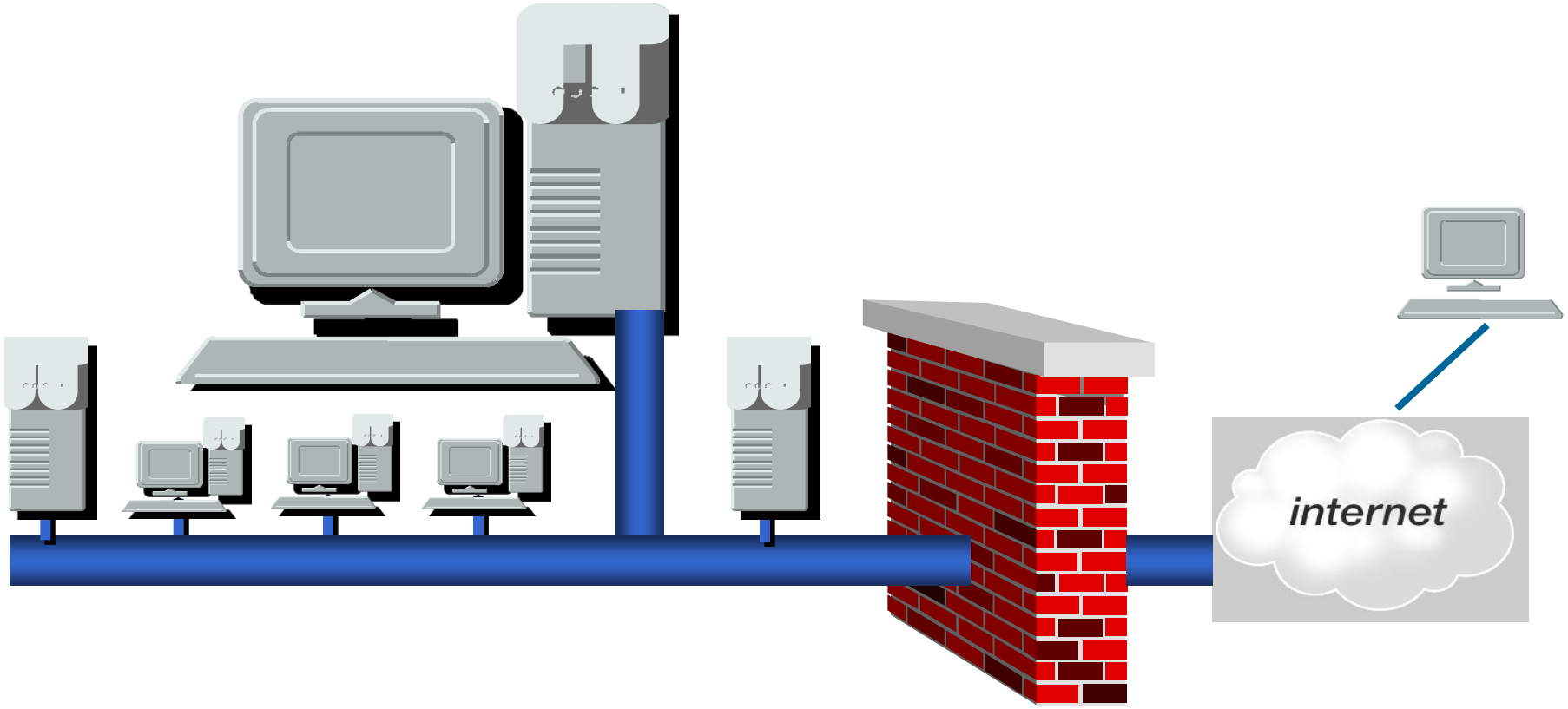
口令攻击

1　口令暴力攻击：
　　生成口令字典，通过程序试探口令。

2　窃取口令文件后解密：
　　窃取口令文件（UNIX环境下的Passwd文件和Shadow文件），通过软件解密。

方法与对策：
1、限制同一用户的失败登录次数
2、限制口令最短长度，要求特权指令使用复杂的字母、数字组合。
3、定期更换口令，不要将口令存放到计算机文件中

常用的安全防护措施—防火墙



- 访问控制
- 认证
- NAT
- 加密
- 防病毒、内容过滤
- 流量管理

软件体系结构建模

◇ “4+1”模型概述

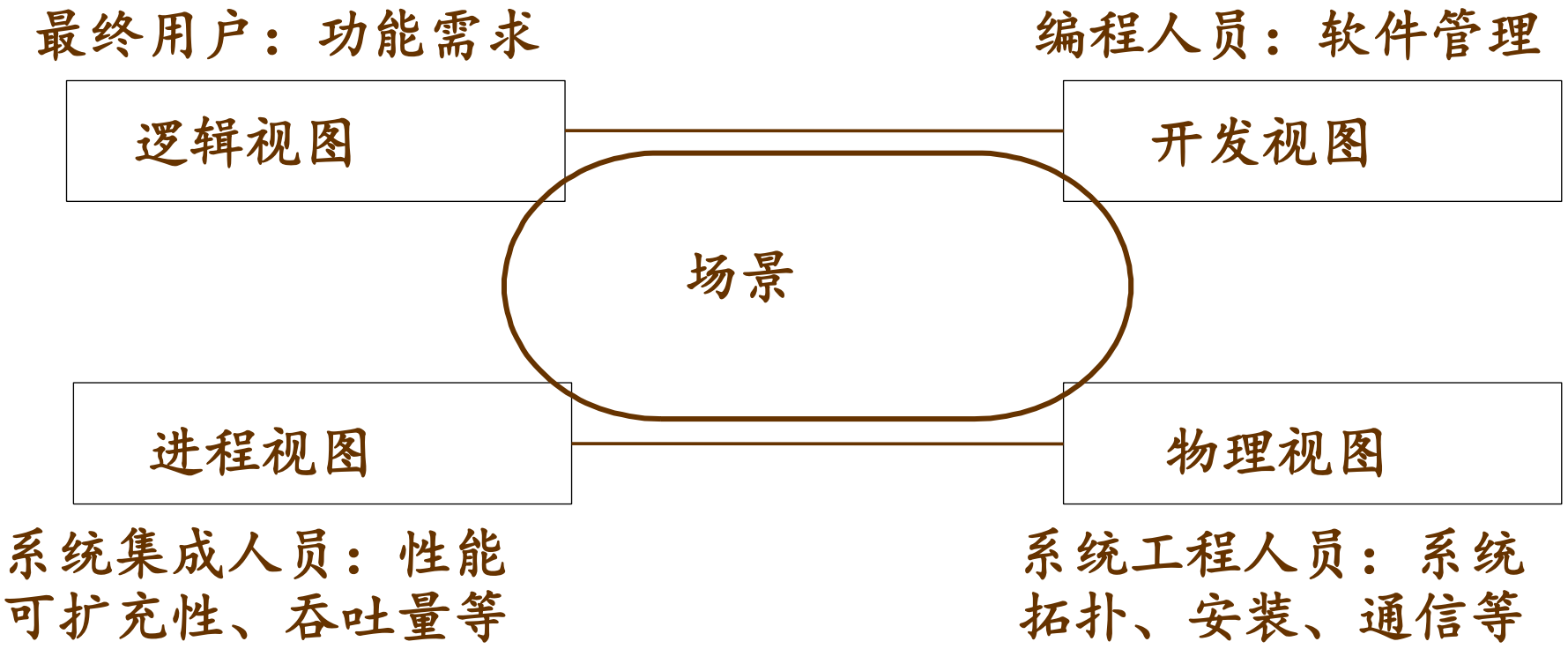
Kruchten在1995年提出了“4+1”的视图模型。

“4+1”视图模型从5个不同的视角包括逻辑视图、进程视图、物理视图、开发视图和场景视图来描述软件体系结构。

每一个视图只关心系统的一个侧面，5个视图结合在一起才能反映系统的软件体系结构的全部内容。

软件体系结构建模

◇ “4+1”模型概述



◇ 逻辑视图

逻辑视图主要支持系统的功能需求，即系统提供给最终用户的服务。在逻辑视图中，系统分解成一系列的功能抽象，这些抽象主要来自问题领域。这种分解不但可以用来进行功能分析，而且可用作标识在整个系统的各个不同部分的通用机制和设计元素。

在面向对象技术中，通过抽象、封装和继承，可以用对象模型来代表逻辑视图，用类图来描述逻辑视图。

◇ 开发视图

开发视图也称模块视图，主要侧重于软件模块的组织和管理。

开发视图要考虑软件内部的需求，如软件开发的容易性、软件的重用和软件的通用性，要充分考虑由于具体开发工具的不同而带来的局限性。

开发视图通过系统输入输出关系的模型图和子系统图来描述。

◇ 进程视图

进程视图侧重于系统的运行特性，主要关注一些非功能性的需求。

进程视图强调并发性、分布性、系统集成性和容错能力，以及从逻辑视图中的主要抽象如何适合进程结构。它也定义逻辑视图中的各个类的操作具体是在哪一个线程中被执行的。

进程视图可以描述成多层抽象，每个级别分别关注不同的方面。在最高层抽象中，进程结构可以看作是构成一个执行单元的一组任务。它可看成一系列独立的，通过逻辑网络相互通信的程序。它们是分布的，通过总线或局域网、广域网等硬件资源连接起来。

◇ 物理视图

物理视图主要考虑如何把软件映射到硬件上，它通常要考虑到系统性能、规模、可靠性等。解决系统拓扑结构、系统安装、通讯等问题。

当软件运行于不同的节点上时，各视图中的构件都直接或间接地对应于系统的不同节点上。因此，从软件到节点的映射要有较高的灵活性，当环境改变时，对系统其他视图的影响最小。

◇ 场景

场景可以看作是那些重要系统活动的抽象，它使四个视图有机联系起来，从某种意义上说场景是最重要的需求抽象。在开发体系结构时，它可以帮助设计者找到体系结构的构件和它们之间的作用关系。同时，也可以用场景来分析一个特定的视图，或描述不同视图构件间是如何相互作用的。

场景可以用文本表示，也可以用图形表示。

第2章 体系结构

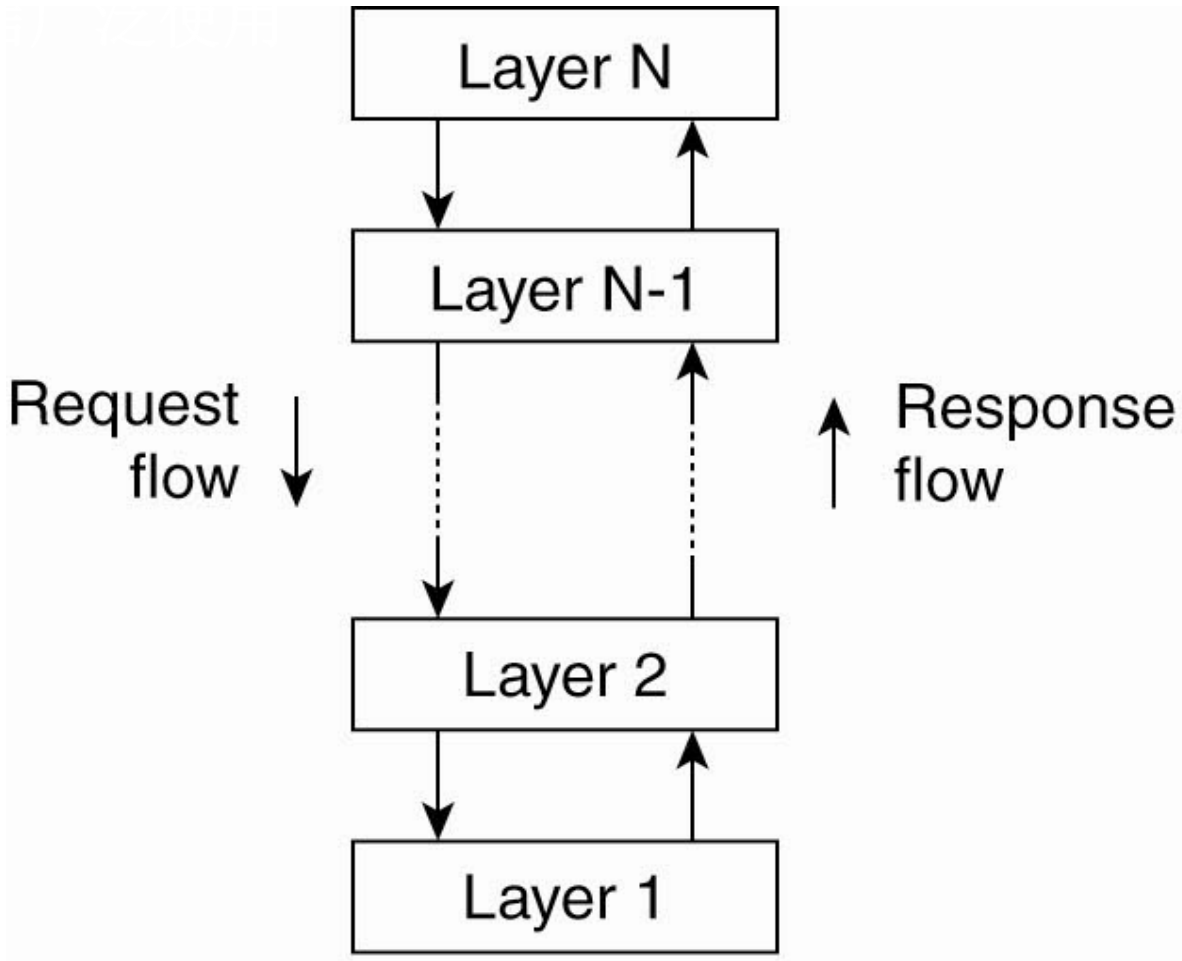
- 2.0 简介
- 2.1 软件体系结构样式
- 2.2 系统体系结构
- 2.3 体系结构与中间件
- 2.4 分布式系统的自我管理
- 2.5 基础模型 Fundamental Models

2.1 体系结构样式 Architectural Styles

体系结构样式：软件架构根据组件、组件之间的连接方式、数据交换以及这些元素如何集成到一个系统来定义：

- 分层体系结构（Layered architectures）
- 基于对象的体系结构（Object-based architectures）
- 以数据为中心的体系结构（Data-centered architectures）
- 基于事件的体系结构（Event-based architectures）

分层体系结构



(a)

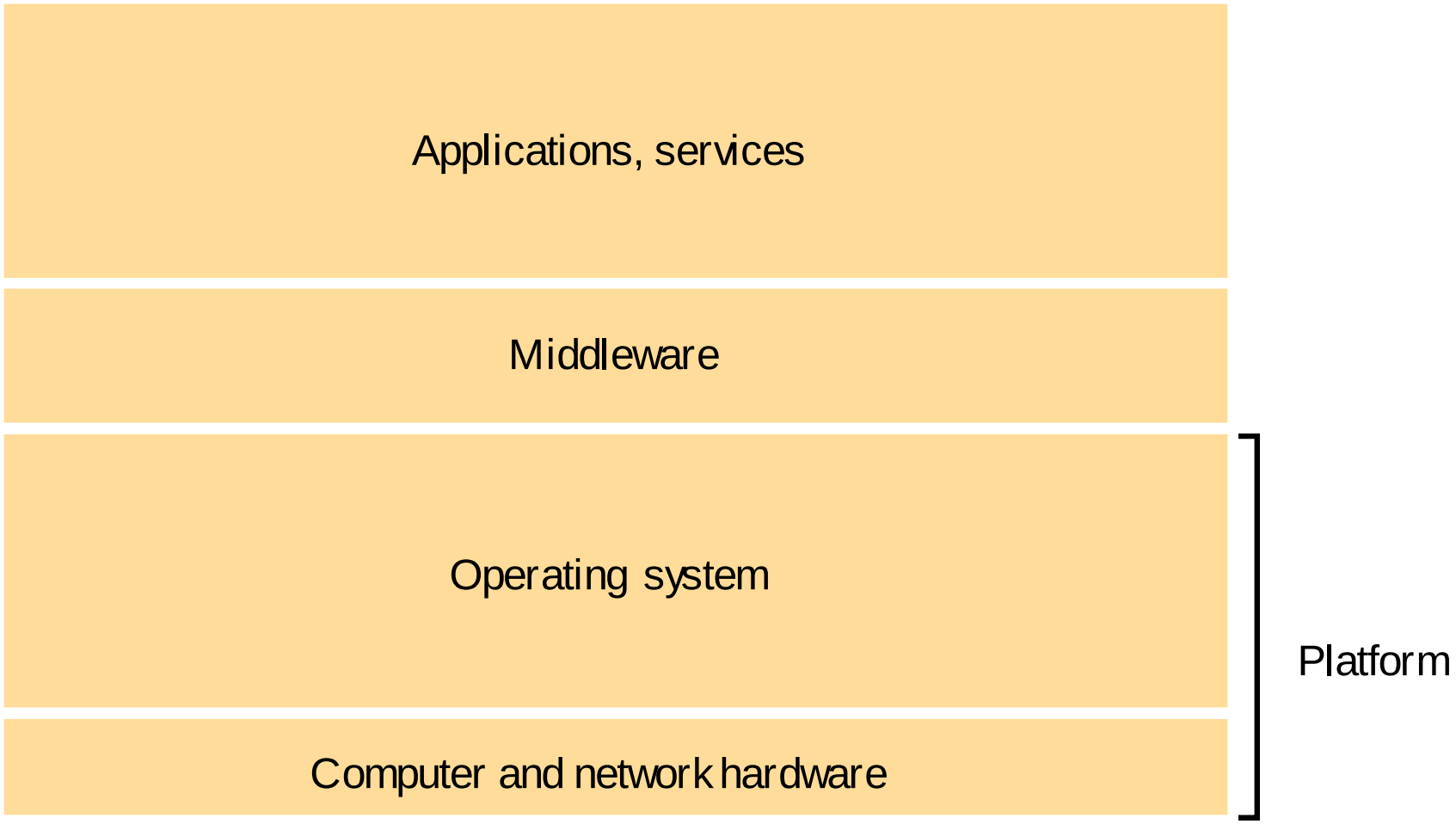
Software Layers

- For ease of design and implementation the software architecture of a distributed systems is divided into layers.分布式系统软件结构的设计和实现被划分出层次。
  - Heterogeneity and openness can be addressed best by using layers.异构和开放
  - Potential loss of performance
    - Example: Error checking in each layer

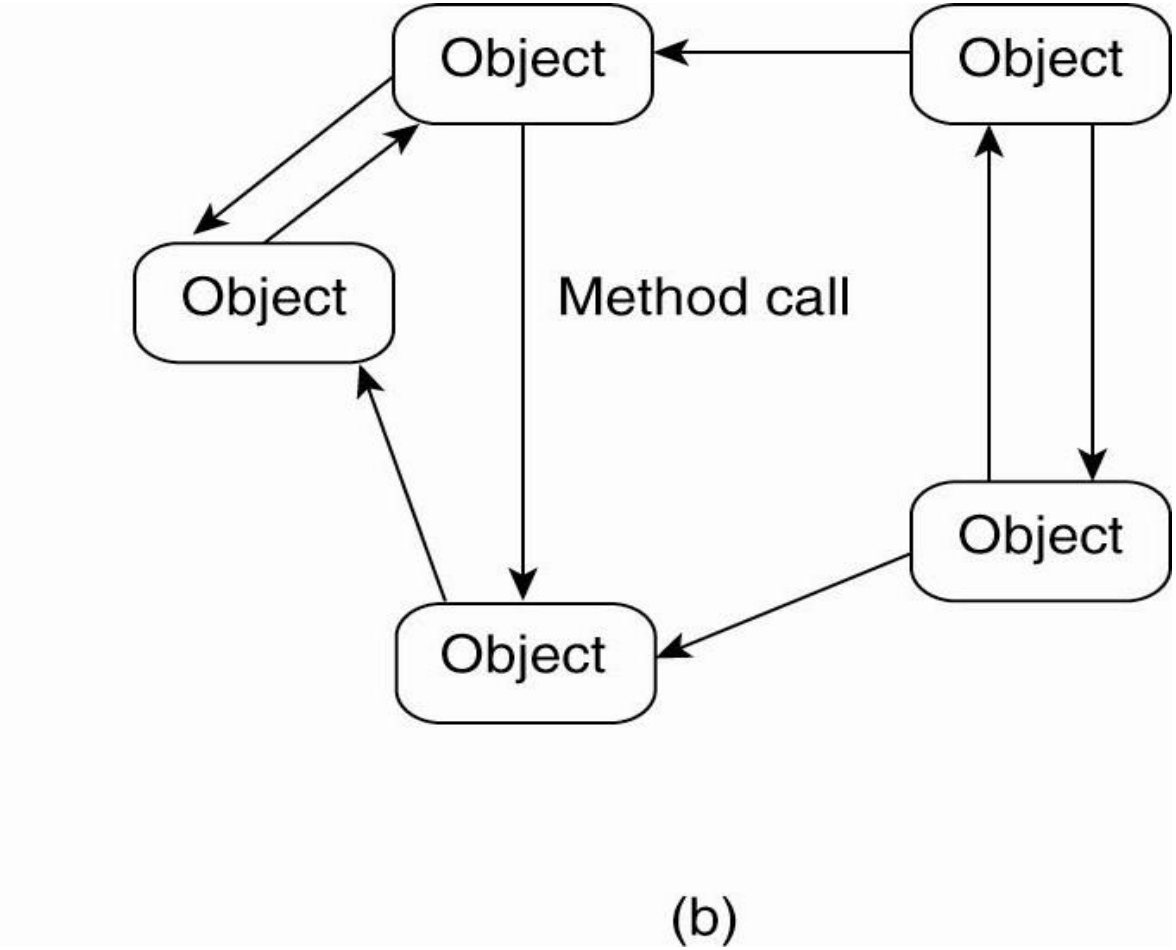
Software Layers

- **Platform layer平台层: The hardware and the lowest layer of software**
  - **Operating systems of the various components**
- **Middleware layer中间件层: 目的是屏蔽异构性，为应用程序员提供方便的编程模型。**
  - **Masking of heterogeneity**
  - **Convenient programming model for application programmers**
- **Application layer应用层: Applications are implemented on top of a given middleware.**

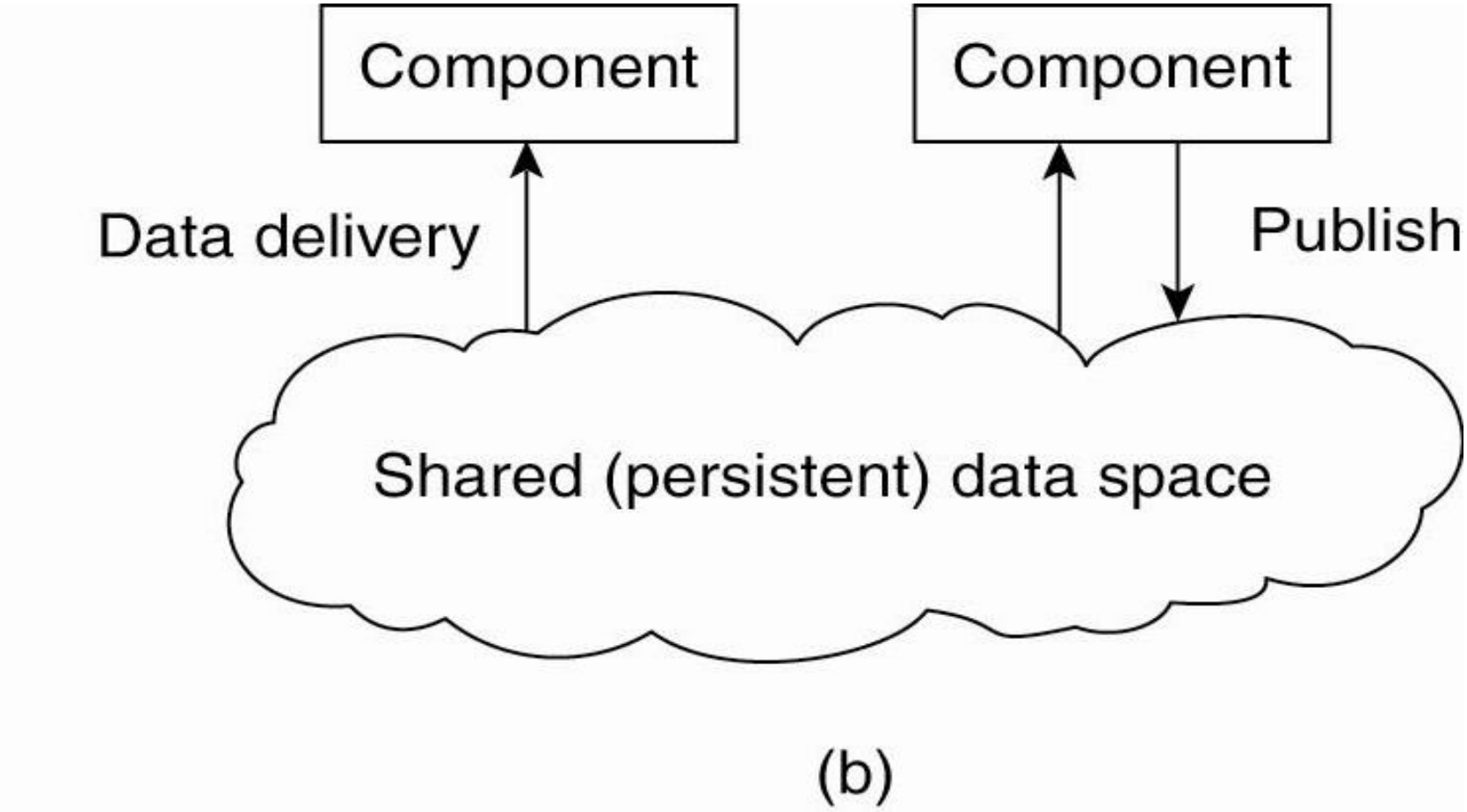
Figure 2.1
Software and hardware service layers in distributed systems



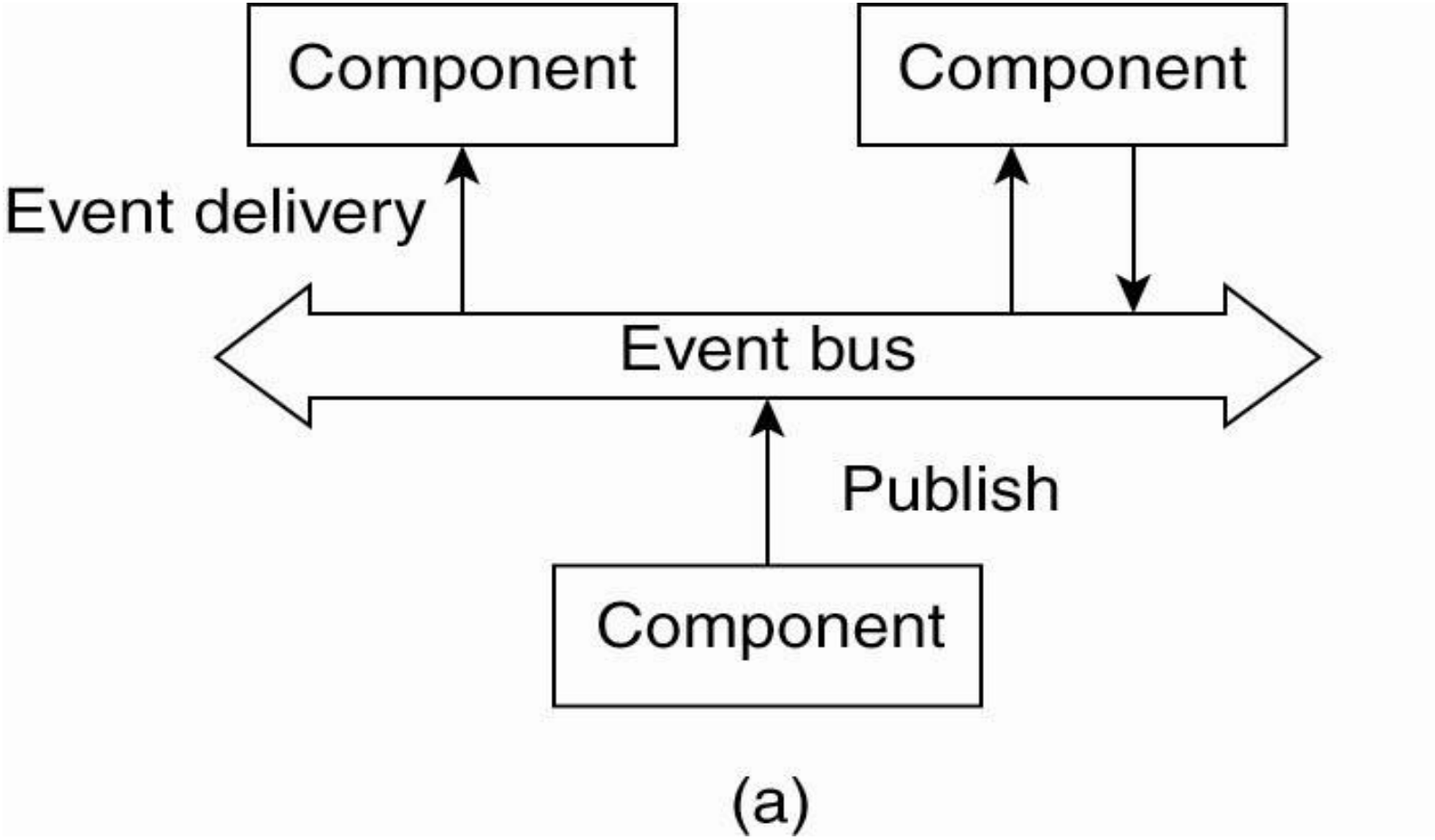
基于对象的体系结构



以数据为中心的体系结构



基于事件的体系结构


Component
Component
Event delivery
Event bus
Publish
Component

(a)

第2章 体系结构

- 2.0 简介
- 2.1 软件体系结构样式
- 2.2 系统体系结构
- 2.3 体系结构与中间件
- 2.4 分布式系统的自我管理
- 2.5 基础模型
  - 交互模型
  - 故障模型
  - 安全模型

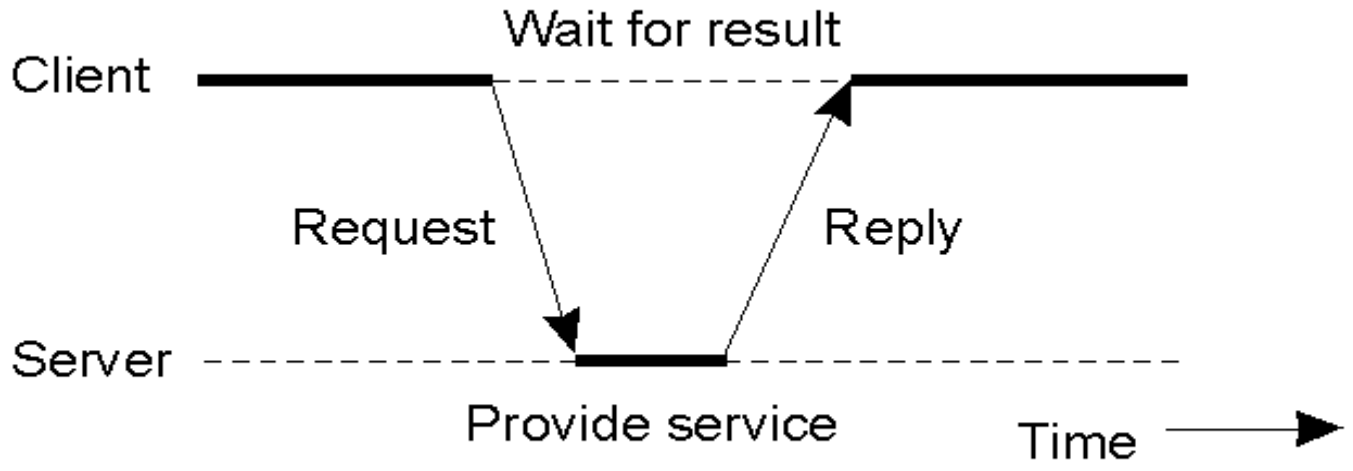
2.2 系统体系结构

系统体系结构：软件体系结构的实例，确定软件组件、组件的交互以及它们的位置

- 集中式体系结构
- 非集中式体系结构
- 混合体系结构

集中式体系结构
客户端-服务器模型

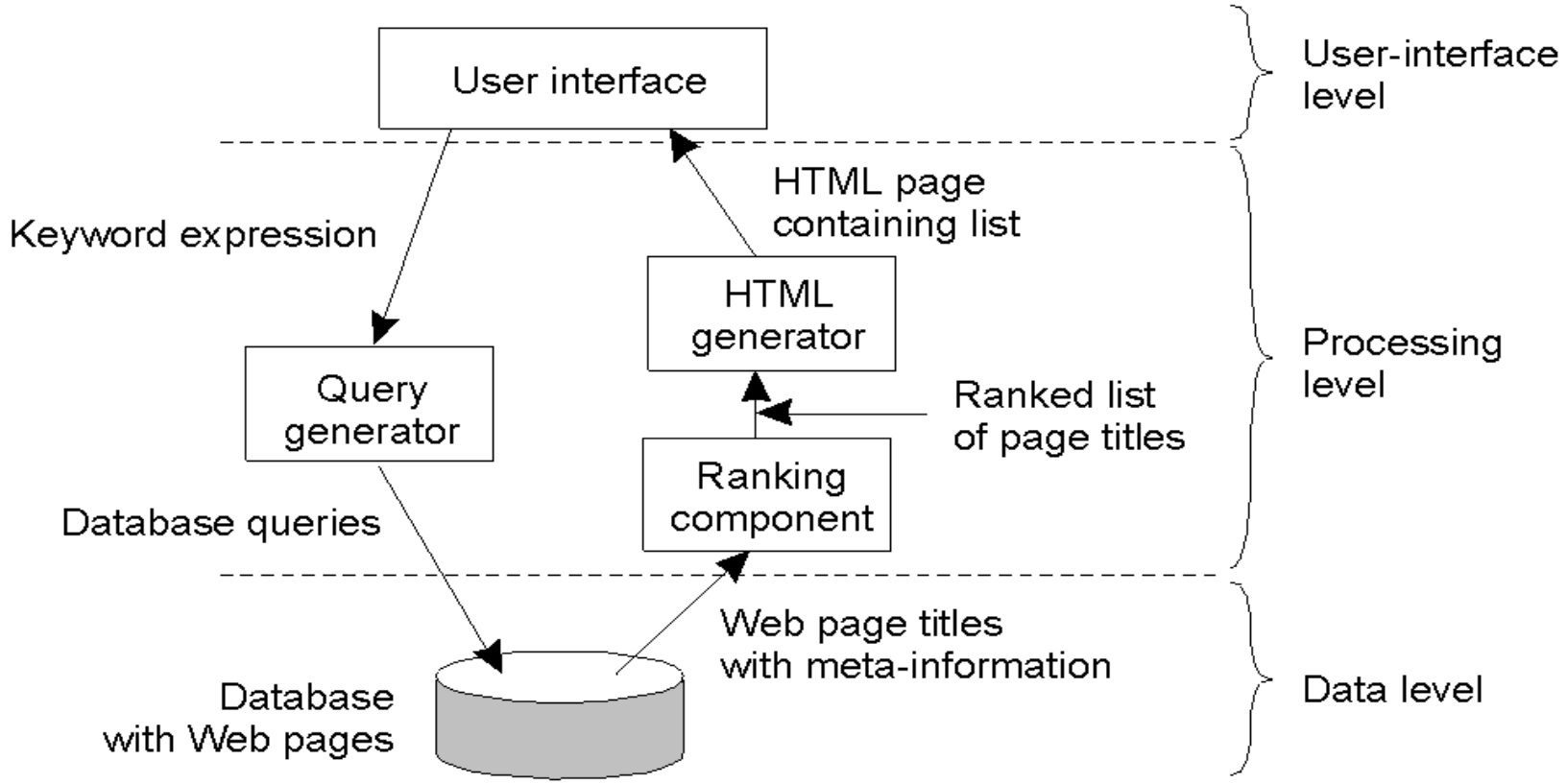
- 服务器（server）：实现某个特定服务的进程
- 客户（client）：向服务器请求服务的进程
- 客户端-服务器之间的一般交互：请求/回复
- 无连接的协议：高效，受传输故障的影响，适合局域网
- 基于连接的协议：性能相对较低，适合广域网（TCP/IP）



应用程序的分层

客户服务器应用程序通常组织为三个层次(搜索引擎、金融决策支持系统)：

- 用户界面层：用户交互所需的一切
- 处理层：应用程序核心功能
- 数据层：操作数据或文件系统，保持一致性



Network Computers and Thin Clients
（网络计算机和瘦客户）

The **network computer** downloads its **operating system** and any needed **application software** from a **remote file server**.

网络计算机从远程文件服务器下载它的操作系统和任何需要的应用软件。

- The user need not maintain the local software base.

用户不需要保存本地软件库。

- Applications are run locally while the files are managed by the remote file server.
- 可在本地运行应用，但文件有远程文件服务器管理。
  - The user can migrate from one network computer to another.用户可以从一台网络计算机迁移到另一台。

Network Computers and Thin Clients
（网络计算机和瘦客户）

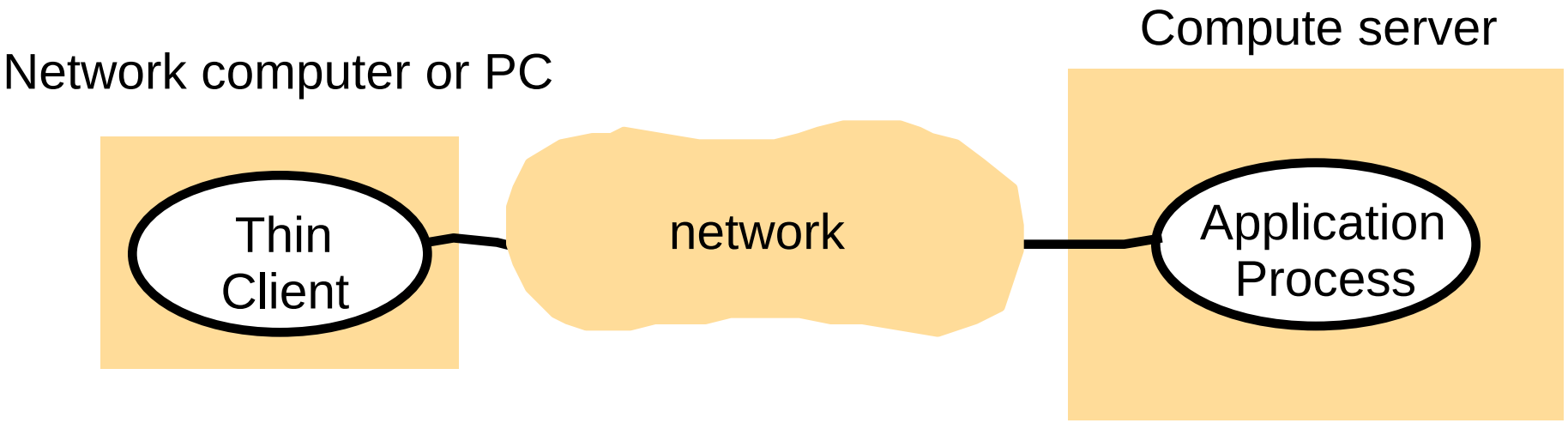
- Little processor and memory capacities are required.
- 处理器和内存容量受限。
  - Costs are reduced.代价减少
  - Memory is mainly used as a cache.内存主要用作缓存。

A **thin client** only supports a graphical user interface and runs the application code on a remote server.

瘦客户只是支持图形用户界面和在远程计算机上运行应用程序代码。

- Potentially long response times.
- 不足：潜在的长响应时间。

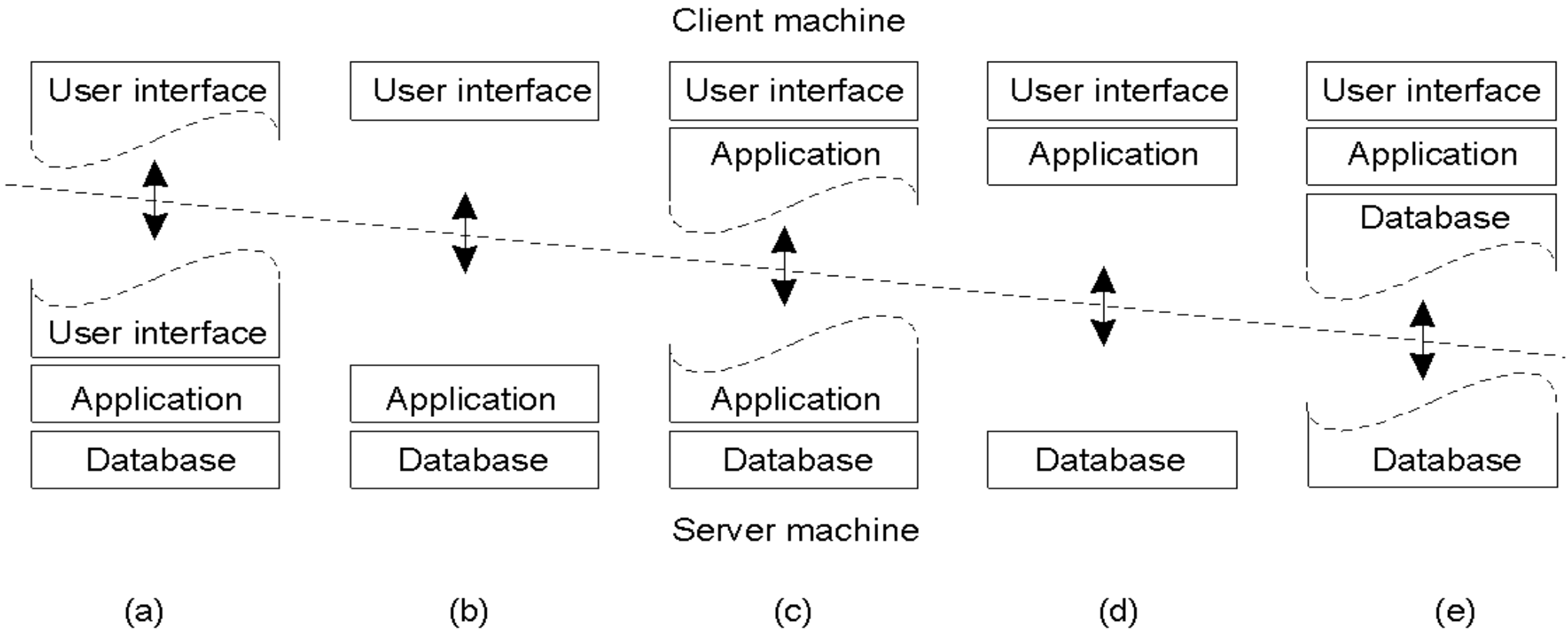
Figure 2.7
Thin clients and compute servers



瘦客户指的是一个软件层。在执行远程计算机上的应用程序代码时，由它在用户本地的计算机上支持支持基于窗口的用户界面。

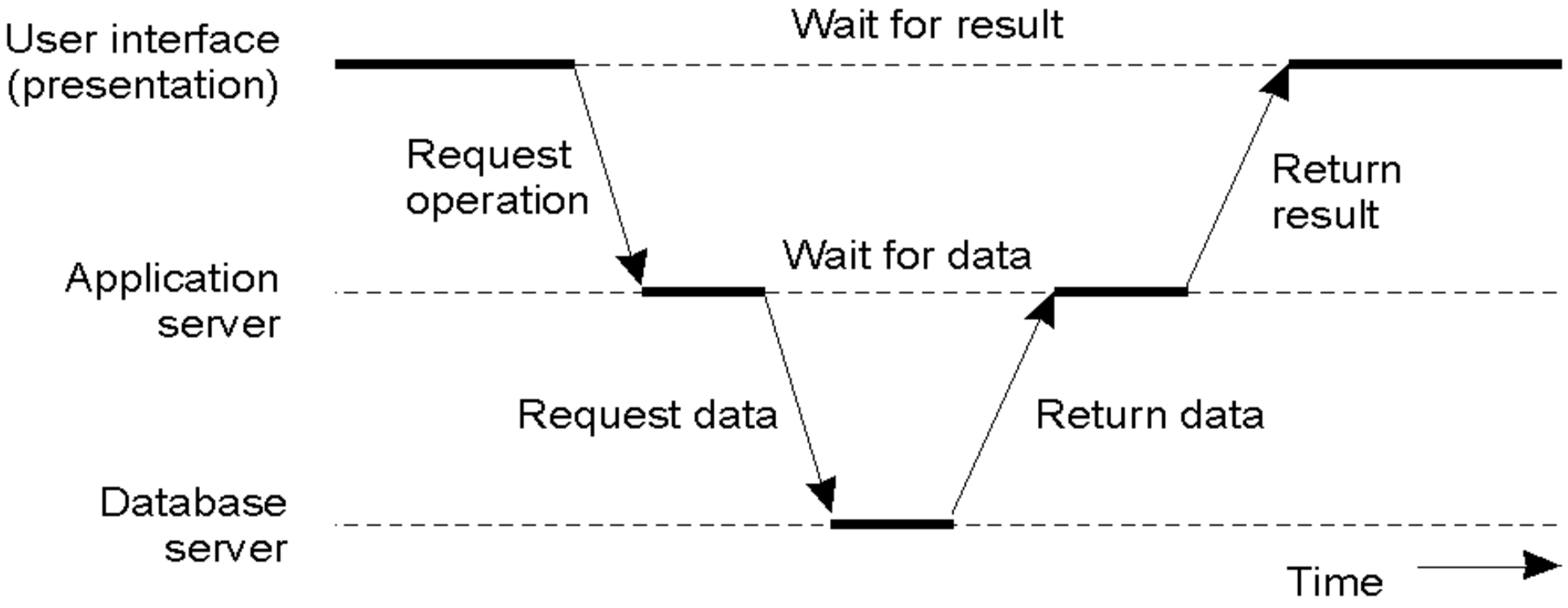
多层体系结构（1）

- 客户端-服务器模型可能的组织结构（a）– （e）.



多层体系结构（2）

- 服务器充当客户端角色的例子



Client and Server

A server is a running program (process) that

- accepts requests from programs (usually running on other computers),
- performs a service,
- responds appropriately by sending messages.

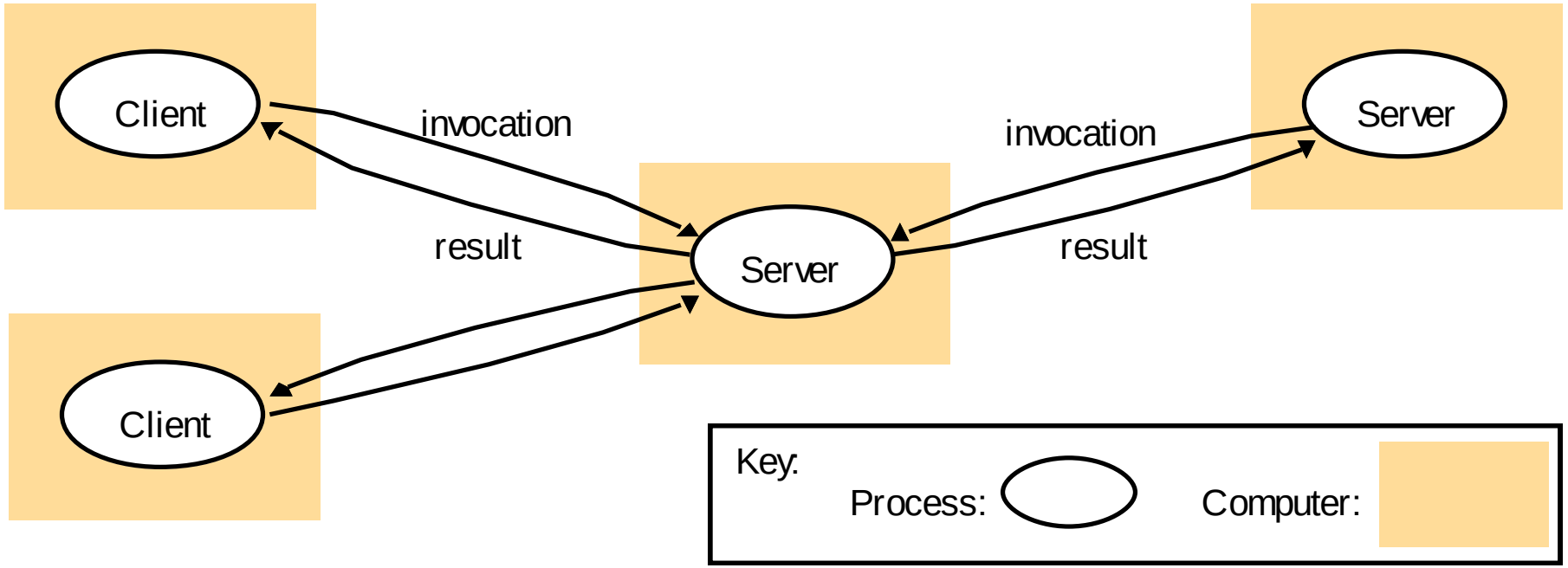
A client is a process that

- requests a service in form of messages.

A client **invokes an operation** upon a server.

- Different locations of client and server: **remote invocation**
- Object oriented programming:
  a client object invokes a method on a server object.

Figure 2.2
Clients invoke individual servers



Client and Server

- 搜索引擎：用户通过因特网查找Web页面上可用的信息总汇。这些信息通过“网络爬虫”的程序形成。
- 搜索引擎既是服务器又是用户
  - 回答用户查询（Server）
  - 运行“网络爬虫”的程序,向其它Web服务器发请求（Client）
  - 线程运行“网络爬虫”的程序。

Proxy Server 代理服务器

Caches are used in all computer systems to hide delays in accessing data due to different speeds of storage devices. 缓存用于计算机系统中缓解（由于存储设备的速度不同造成的）访问数据的延迟。

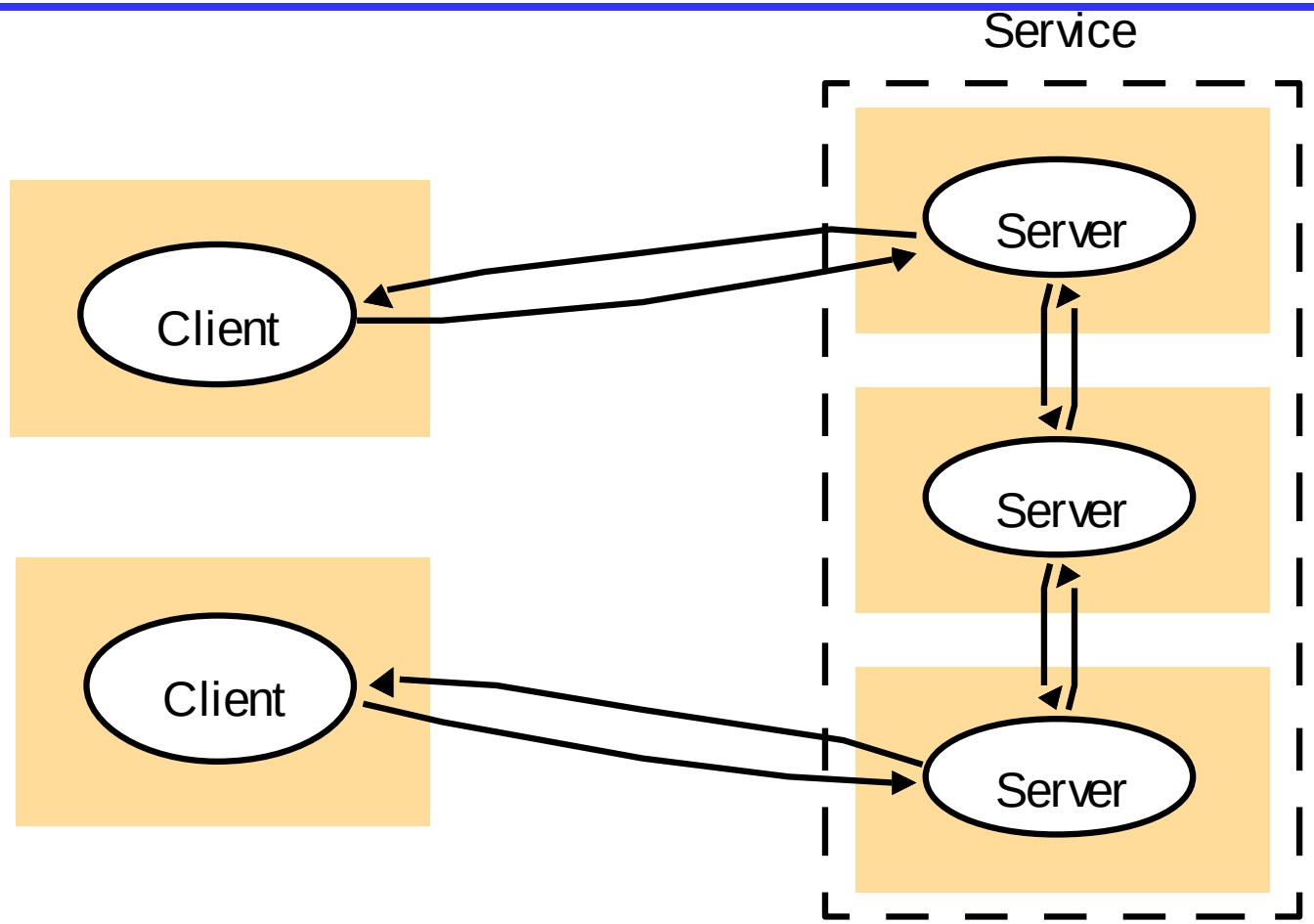
In distributed systems access delays can be caused by **remote data locations** and **slow network connections**.（远程数据定位和慢速网络连接会造成分布式系统的访问延时）。

- Client caches客户缓存
- Specific cache servers: Proxy servers代理服务器
  - shared cache for several servers and several clients
  - 客户和服务器共享缓存

Proxy Server

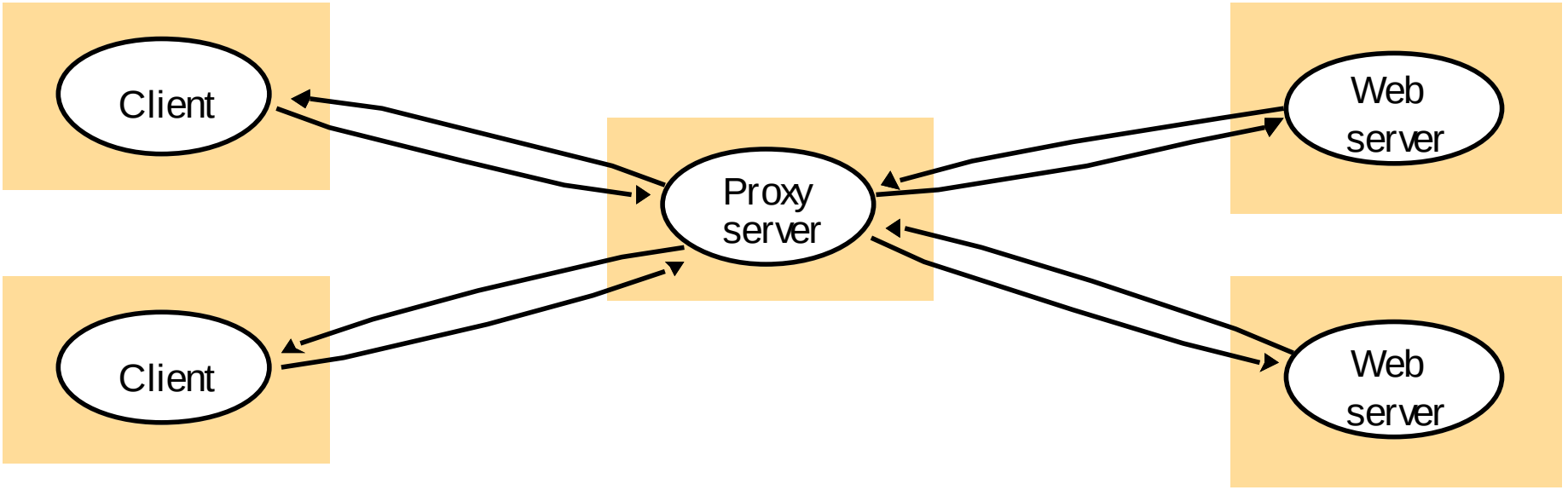
- Upon a request from a client the proxy server determines the validity of the cached data with the original server.客户对代理服务器的请求决定源服务器缓存数据的有效性。
  - fast interaction快速交互
- If the data is not valid it retrieves a new copy from the server. 如果数据无效，从服务器收取新的副本。
- The proxy server sends the data to the client.代理服务器送数据给客户

Figure 2.4
A service provided by multiple servers



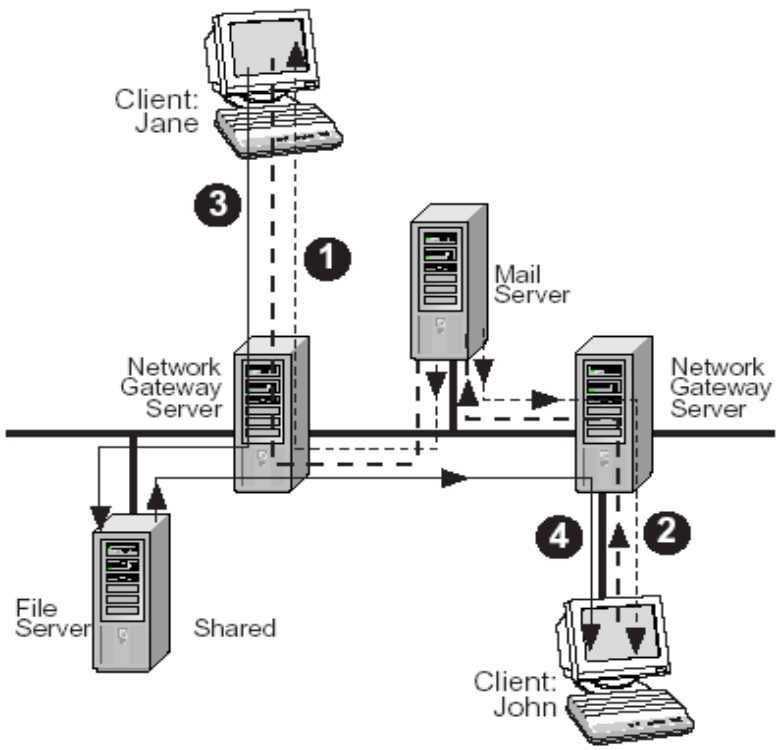
与客户进行必要交互的几个服务器进程可在一个单独主机上实现，以便给客户进程提供服务。

Figure 2.5
Web proxy server

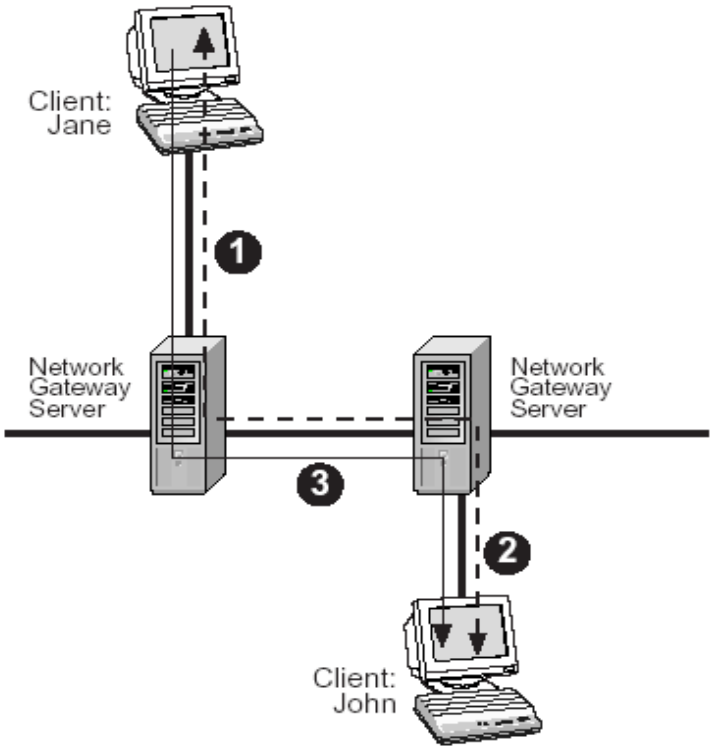


代理服务器的目的是通过减少广域网和WEB服务器的负载提高服务的可用性和性能。

Client-Server vs. Peer-to-Peer Example



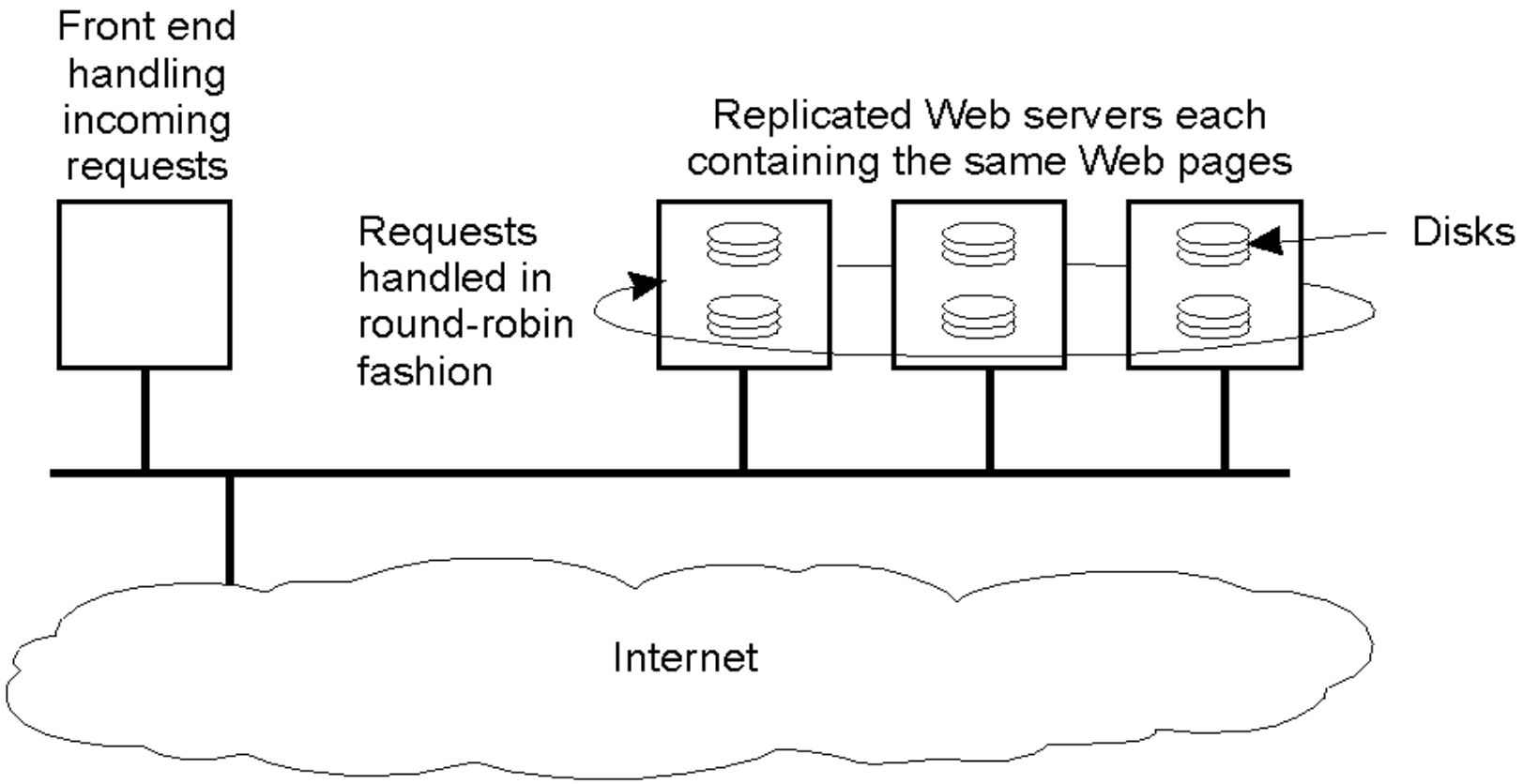
1. John mails Jane: "I need the file "Plan.doc." Jane reads when synchronizing her inbox.
2. John "pulls" Jane's mail reply: "I will place Plan.doc on group's shared drive."
3. Jane uploads file to shared drive.
4. John downloads the file.



1. Instant Message (IM), John to Jane: "I will need the file Plan.doc."
2. IM, Jane to John: "It's on my shared storage now."
3. John downloads the file from Jane's computer.

非集中式体系结构

- 垂直分布性：按逻辑把不同的组件放在不同的机器上。
- 水平分布性：客户或服务器按照在物理上被分割成逻辑上相同的几部分:点对点系统。



P2P 技术

- P2P 应用
  - 文件内容共享和下载，例如Napster、Gnutella、eDonkey、eMule、Maze、BT等；
  - 计算能力和存储共享，例如SETI@home、Avaki、Popular Power等；
  - 协同与服务共享平台，例如JXTA、Magi、Groove等；
  - 即时通讯工具，包括ICQ、QQ、Yahoo Messenger、MSN Messenger等；
  - P2P通讯与信息共享，例如Skype、Crowds、Onion Routing等；
  - 网络电视：沸点、PPStream、 PPLive、 QQLive、 SopCast等。

Peer Processing

In this model all processes play the same role and interact as peers to perform a distributed activity.

所有进程扮演相同的角色。

- No distinction between client and server processes

- Some interprocess communication delays are reduced.
  - No chain of interactions is required.
  - 消除服务器进程可以减少为访问本地对象带来的进程间通信延迟。

Peer Processing

- The total network load is not reduced due to less hierarchy in the system.
- 由于系统缺少层次，总的网络负载不会减少。
  - Additional messages are required to
    - synchronize the peers and使用附加信息同步peers并且
    - maintain the system status and保持系统状态
    - consistency in all peers.所有peer的一致性
  - Coordination Code manages those messages.

Figure 2.3
A distributed application based on peer processes

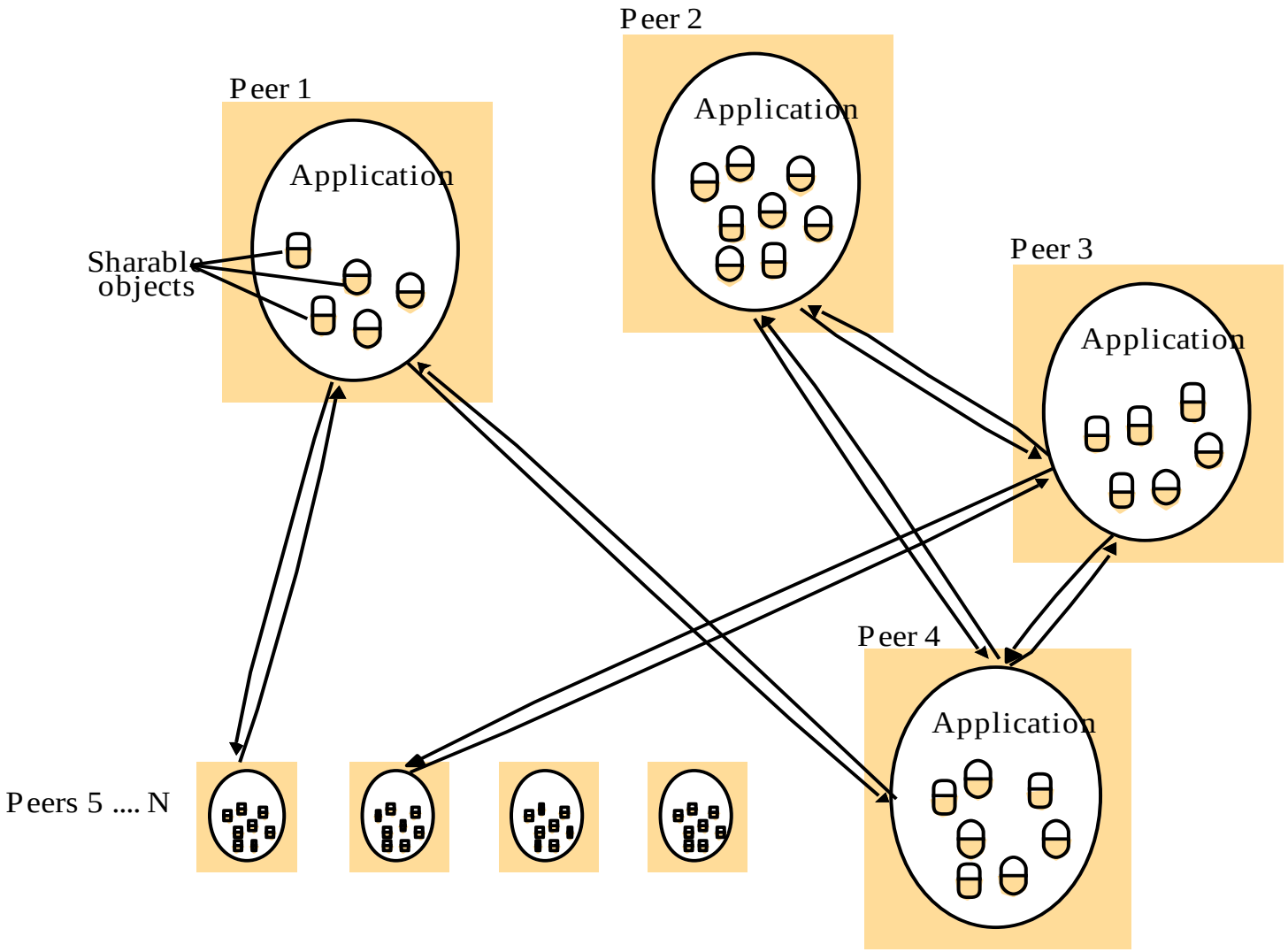


Figure 10.2: Napster: peer-to-peer file sharing with a centralized, replicated index

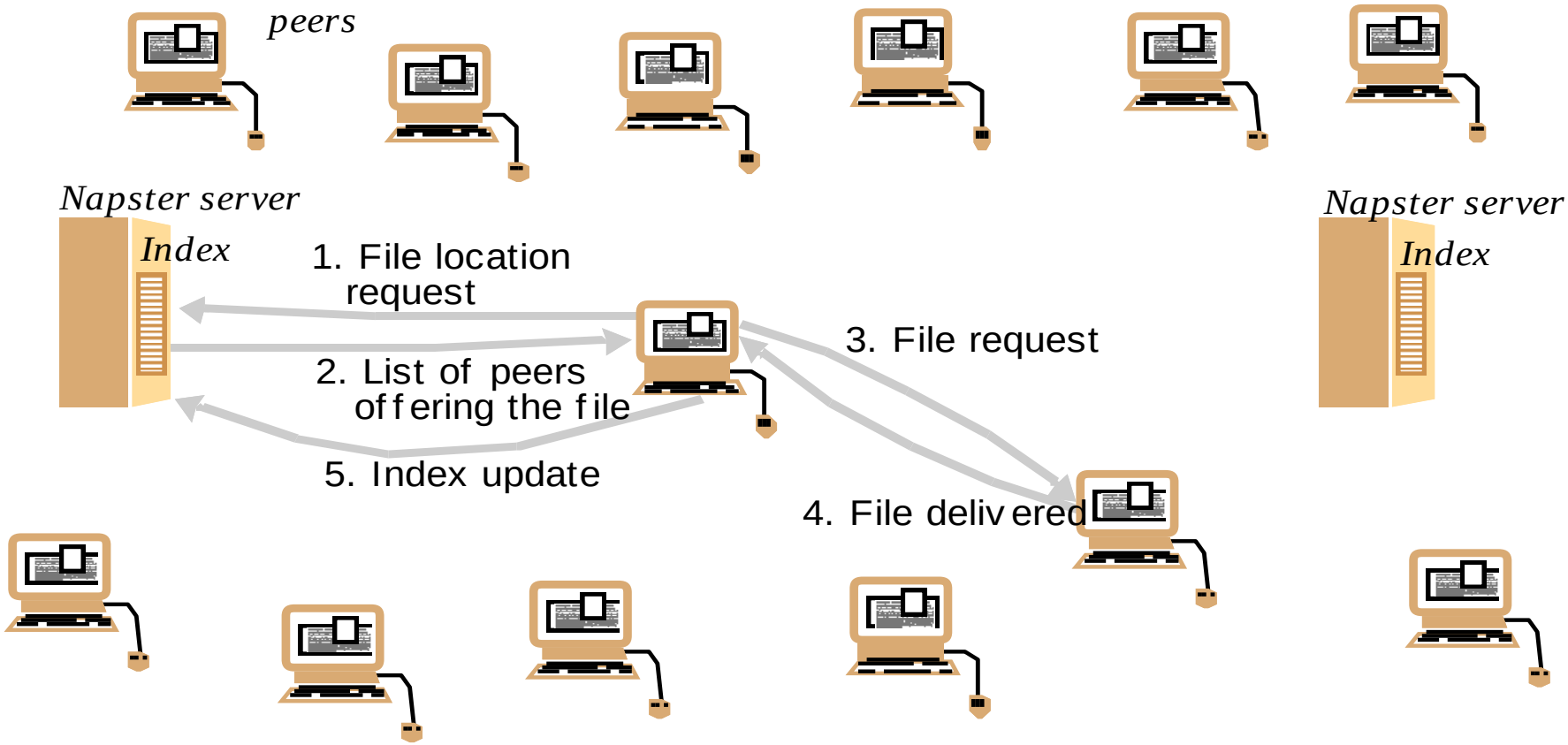
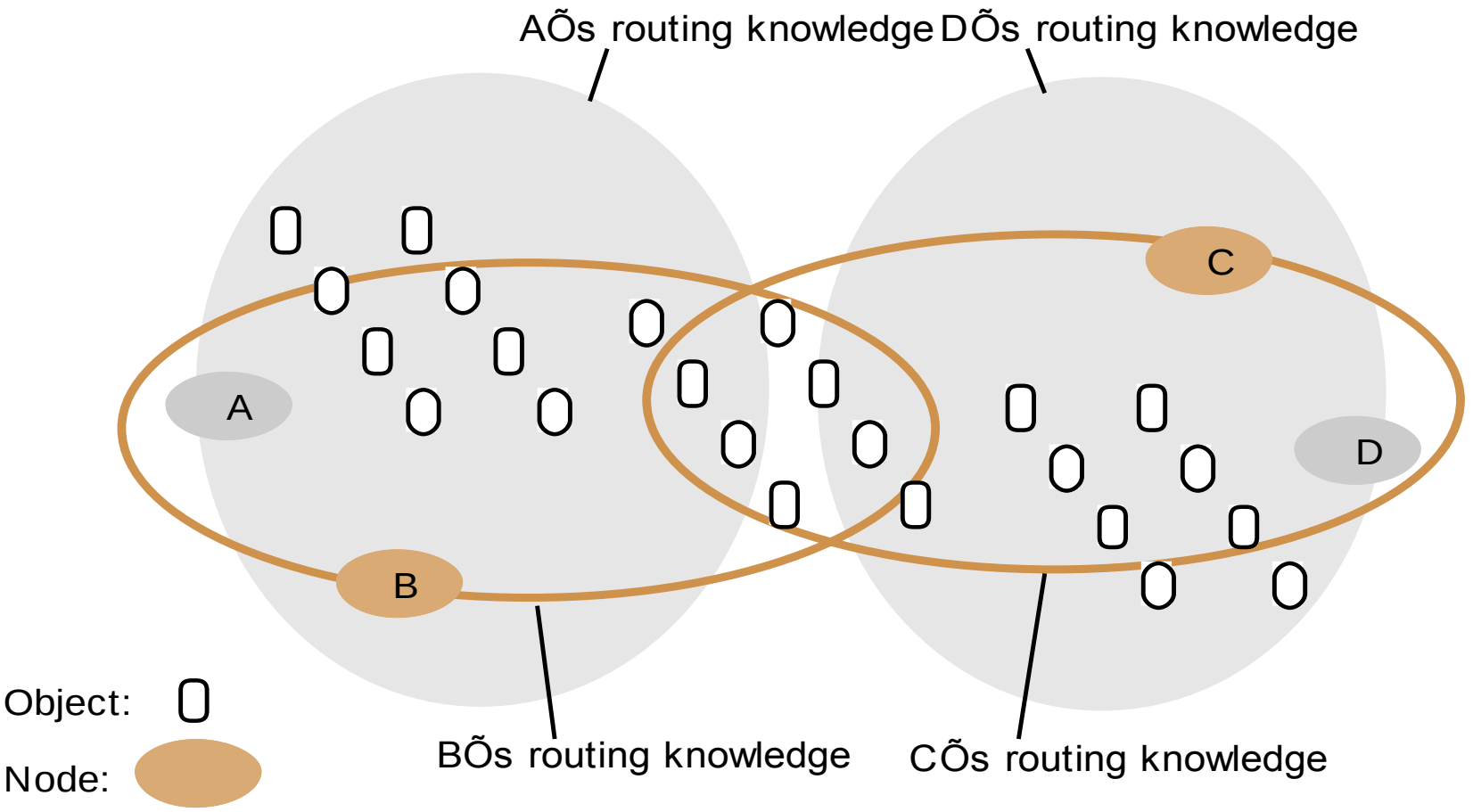


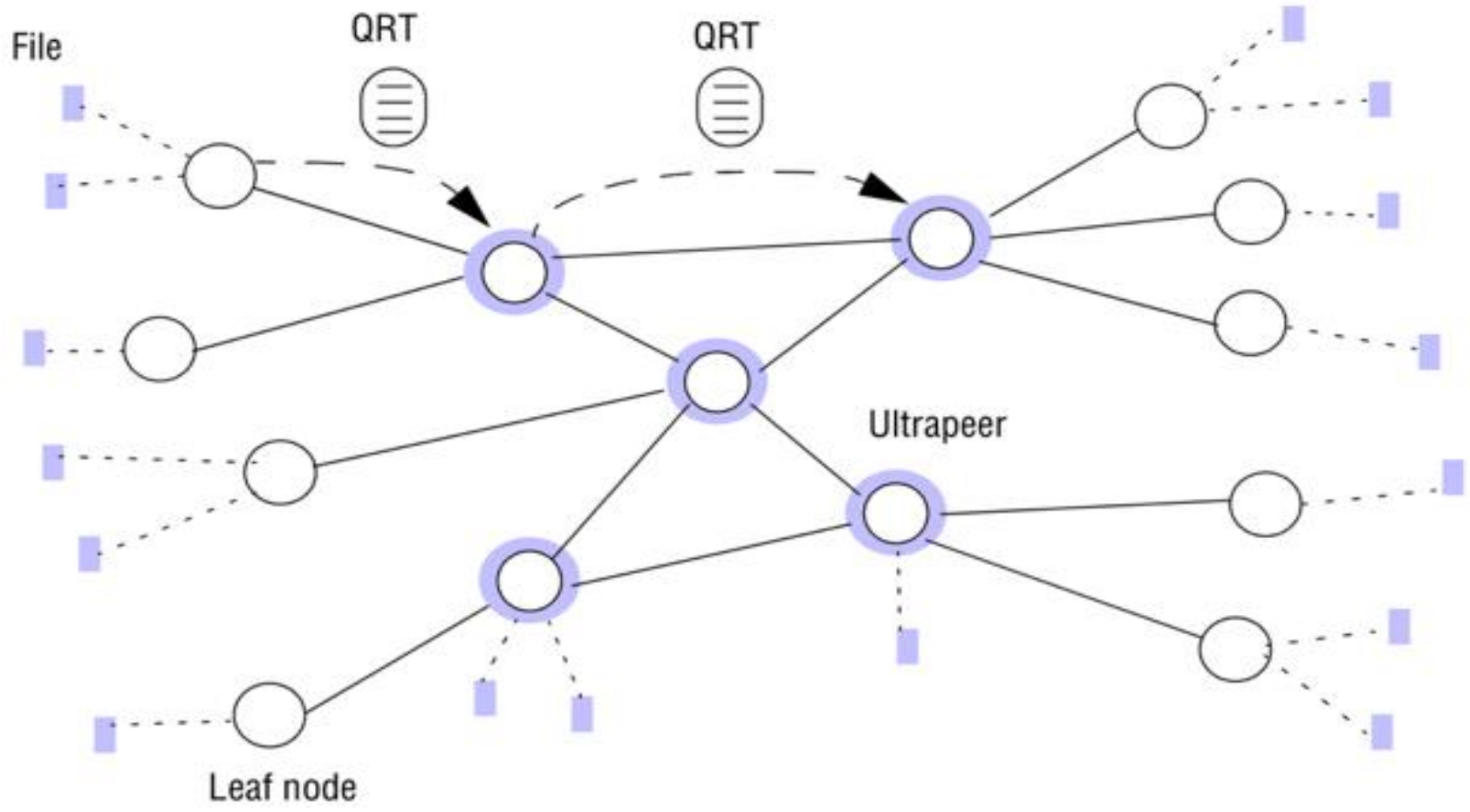
Figure 10.3: Distribution of information in a routing overlay



非结构化点对点体系结构

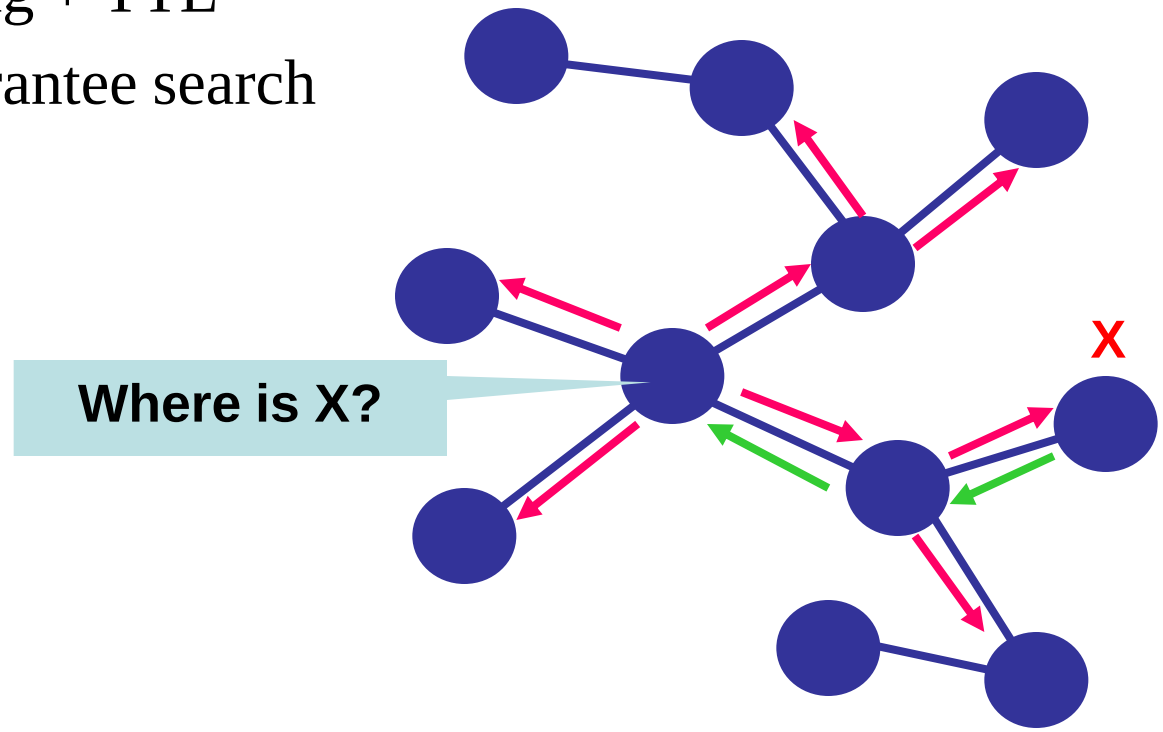
- Flooding
- Random Walk

Key elements in the Gnutella protocol



Search on Unstructured P2P

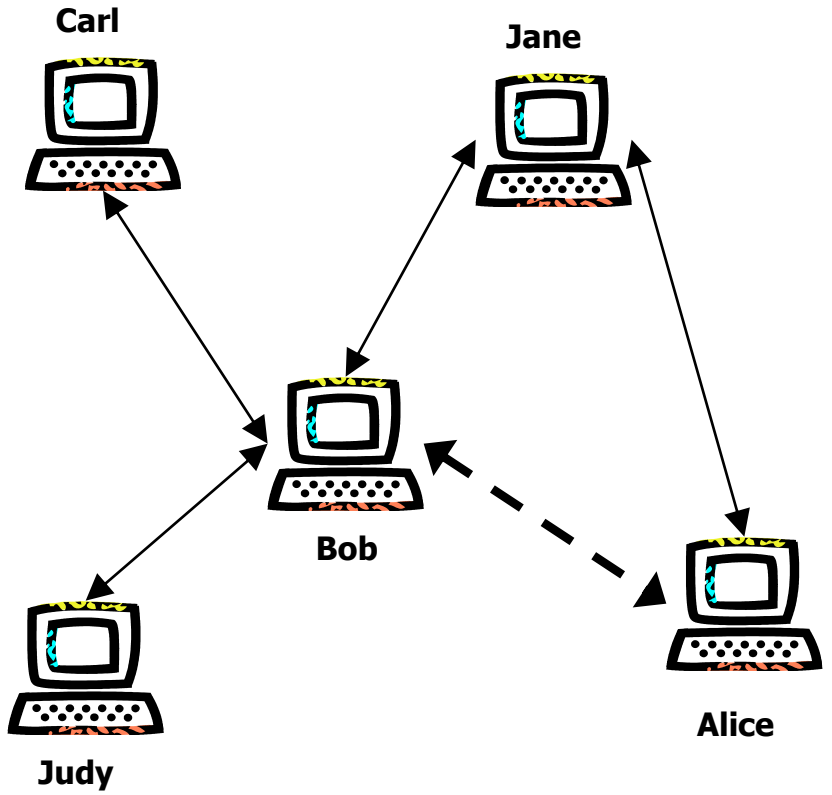
- Example: Gnutella
- Solution: Broadcasting + TTL
- Constraints: non-guarantee search



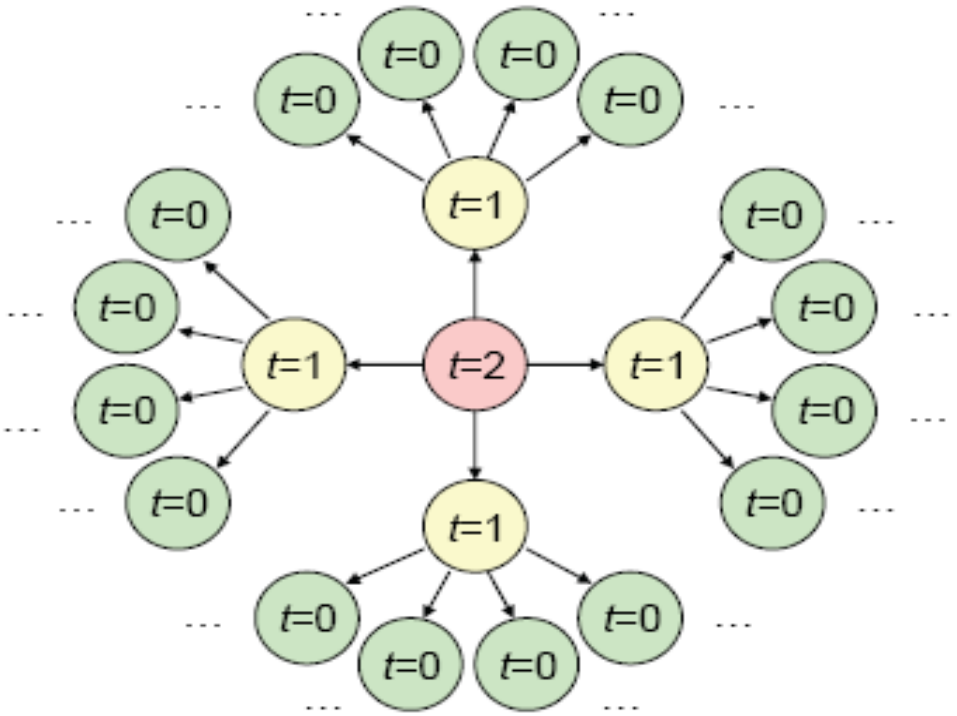
- **Research topics**
  - **- Exploring strategies**
  - **- Linking strategies**
  - **- Routing strategies**

Flooding

- Gnutella model
- Benefits:
  - No central point of failure
  - Limited per-node state
- Drawbacks:
  - Slow searches
  - Bandwidth intensive

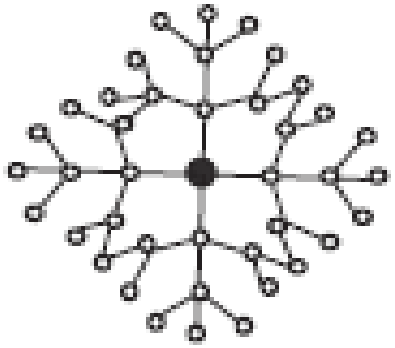


Flooding



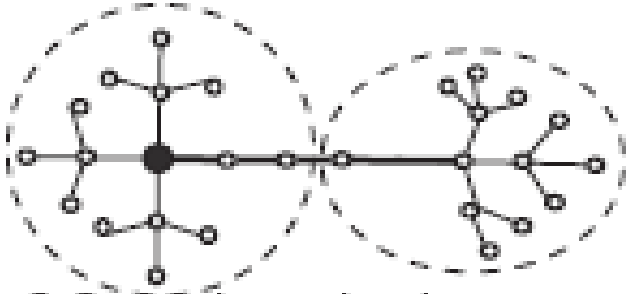
Simple Four-nary Tree Topology with Depth Two

Search on Unstructured P2P

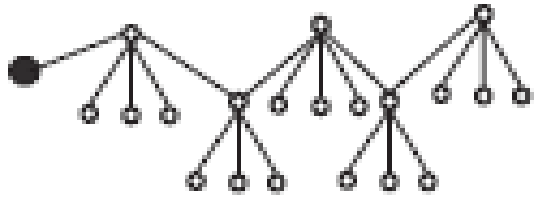


A. Flooding

B. Long random walk



C. General search scheme
(e.g. flooding with direction)



D. Short random walk with
local flooding
(replication, look ahead)

Flooding

- **Flooding** is the predominant search technique in **unstructured peer-to-peer** (P2P) networks.
- If we measure performance as the number of exchanged messages per distinct response, flooding with **small time-to-live performs well** in regular networks.
- However, its **performance deteriorates as the time-to-live increases**, or if the topology of the underlying network is not regular. In addition, flooding has poor granularity.

Random Walk

- In regular topologies, the performance of the **random walk simulation method appears to be better than the performance of flooding**

- In addition, the random walk simulation method scales well and has **excellent granularity**. However, the simulation of a random walk is inherently sequential, which causes a large increase in the response time.

结构化点对点体系结构

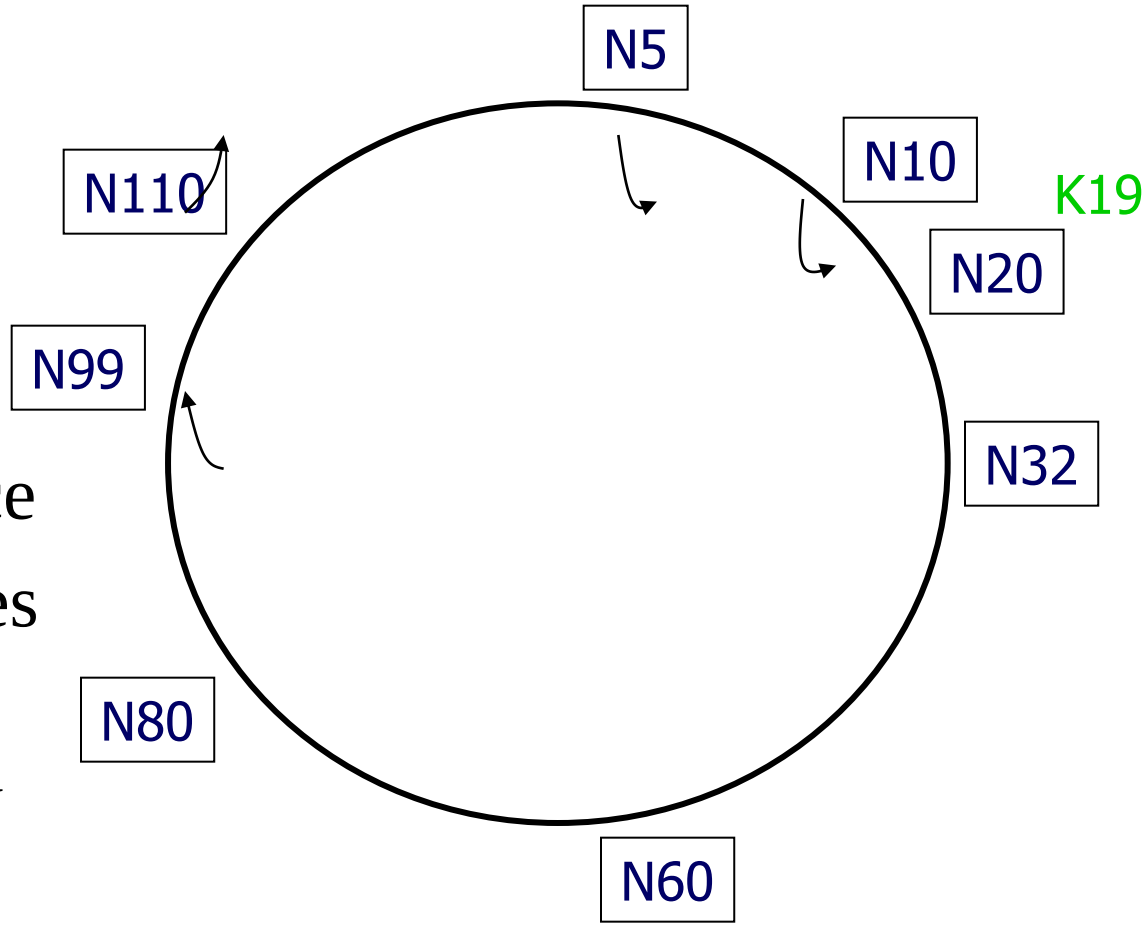
Search on Structured P2P

- Distributed Hashing with **structured** topologies
  - (Chord)
- Scalable Content-Addressable Network
  - (CAN)
- Distributed Tree-Index with **structured** topologies
  - A Balanced Tree (BATON)

结构化点对点体系结构——Chord

- Document Routing – Chord

- MIT project
- Uni-dimensional ID space
- Keep track of log N nodes
- Search through log N nodes to find desired key



Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Robert Morris
Ion Stoica, David Karger,
M. Frans Kaashoek, Hari Balakrishnan

MIT and Berkeley
http://www.pdos.lcs.mit.edu/chord

Routing challenges

- Define a useful key nearness metric
- Keep the hop count small
- Keep the tables small
- Stay robust despite rapid change

- Chord: emphasizes efficiency and simplicity

Chord properties

- Efficient: O(*log(N)*) messages per lookup
  - N is the total number of servers
- Scalable: O(*log(N)*) state per node
- Robust: survives massive failures

- Proofs are in paper / tech report
  - Assuming no malicious participants

Chord overview

- Provides peer-to-peer hash lookup:
  - Lookup(key) → IP address
  - Chord does not store the data

Chord IDs

- Key identifier = SHA-1(key)
- Node identifier = SHA-1(IP address)
- Both are uniformly distributed
- Both exist in the same ID space

Chord Overview

- Chord provides fast distributed computation of a hash function mapping keys to nodes responsible for them.
- It uses *consistent hashing* (high probability the hash function balances load)

Chord Overview

- Randomly assign each peer an ID between 0 and 2^m
- Hash key to find ID for an item
- Next peer after ID (mod 2^m)is responsible for storing item
- Each peer has routing table to subsequent peers
- Table is used for incremental routing

Scalable Content-Addressable Network (CAN)

Document Routing – CAN

- Associate to each node and item a unique *id* in an *d*-dimensional space
- Goals
  - Scales to hundreds of thousands of nodes
  - Handles rapid arrival and failure of nodes
- Properties
  - Routing table size O(*d*)
  - Guarantees that a file is found in at most $d*n^{1/d}$ steps, where *n* is the total number of nodes

Slide modified from another presentation

Document Routing – CAN

- Content-Addressable Network (CAN) as a distributed infrastructure that provides hash table-like functionality on Internet-like scales
- *Content-Addressable Network* (CAN)
  - a distributed,
  - Internet-scale,
  - hash table
- The CAN is
  - scalable, fault-tolerant and completely self-organizing,
  - robustness and low-latency properties

CAN Design

- This coordinate space is completely logical and bears no relation to any physical coordinate system

- This virtual coordinate space is used to store (key,value) pairs as follows: to store a pair (K1,V1), key K1 is deterministically mapped onto a point P in the coordinate space using a uniform hash function

CAN Design

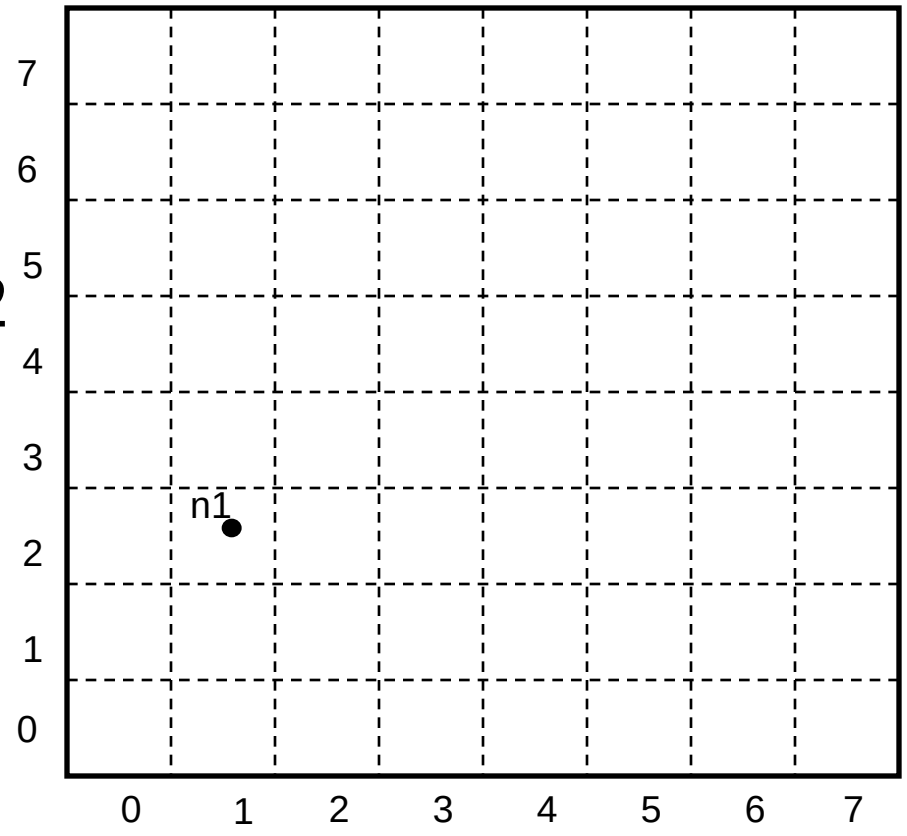
- Nodes in the CAN self-organize into an overlay network that represents this virtual coordinate space.
- A node learns and maintains the IP addresses of those nodes that hold coordinate zones adjoining its own zone.
- This set of immediate neighbors in the coordinate space serves as a coordinate routing

CAN Design

- CAN routing
  - maintains a coordinate routing table
  - Using its neighbor coordinate set, or flooding
- Construction of the CAN coordinate overlay
  - Finding a Zone
  - Joining the Routing
- Maintenance of the CAN overlay
  - Node departure
  - Recovery

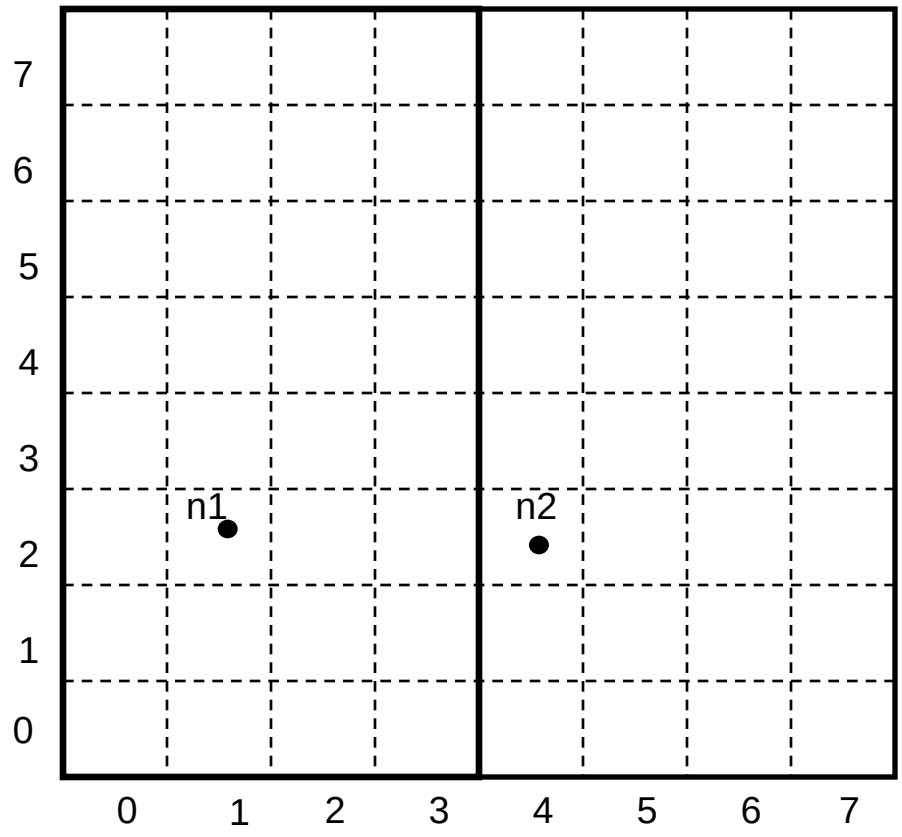
CAN Example: Two Dimensional Space

- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1:2 or 2:1
- Example:
  - Node n1:(1, 2) first node that joins → cover the entire space



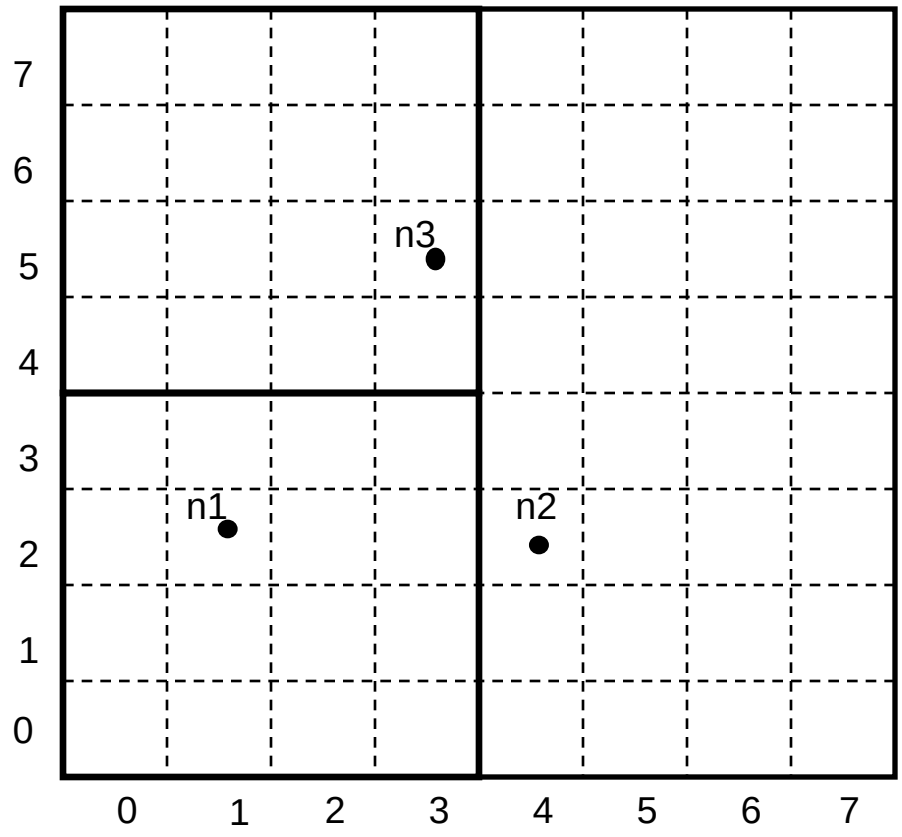
CAN Example: Two Dimensional Space

- Node n2:(4, 2) joins → space is divided between n1 and n2



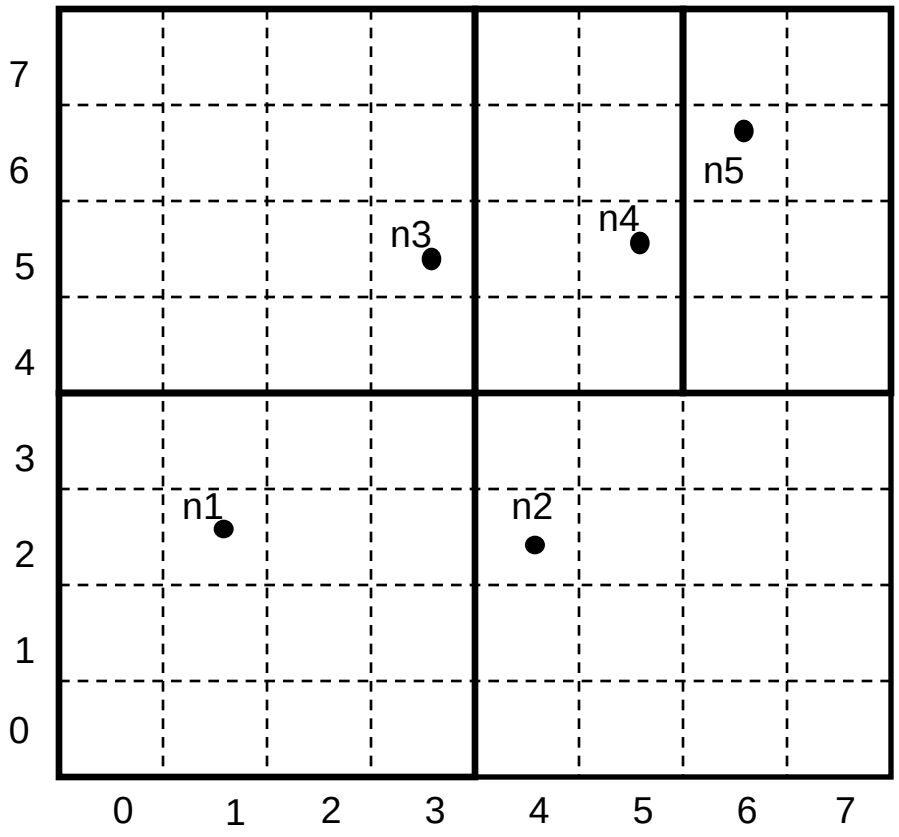
CAN Example: Two Dimensional Space

- Node n2:(4, 2) joins → space is divided between n1 and n2



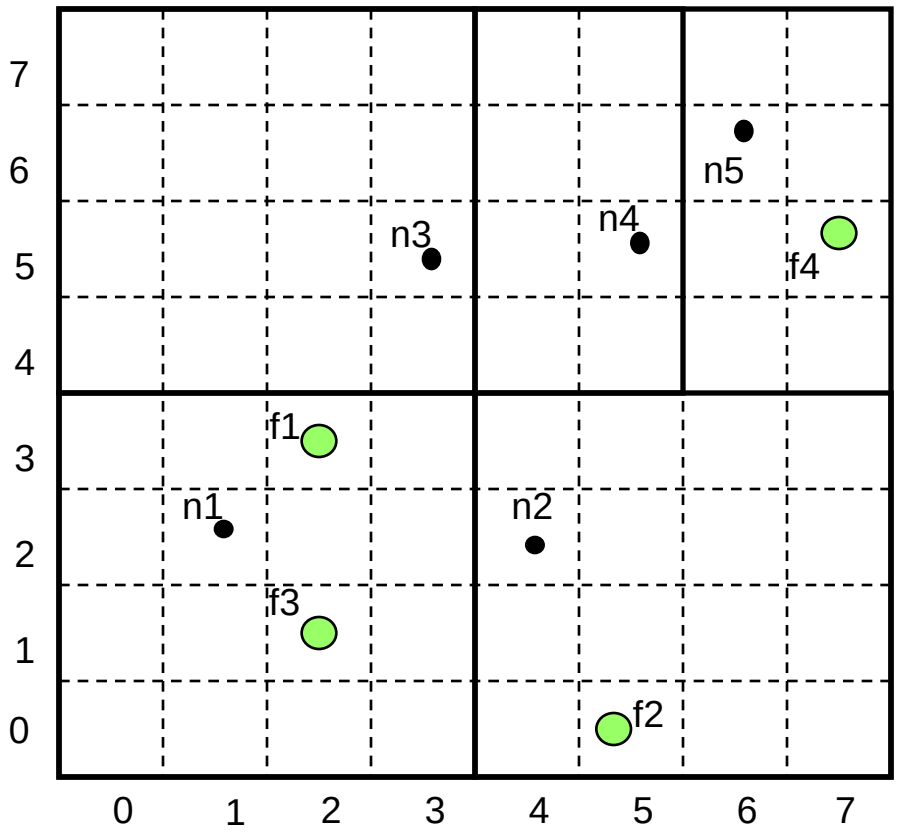
CAN Example: Two Dimensional Space

- Nodes n4:(5, 5) and n5:(6,6) join



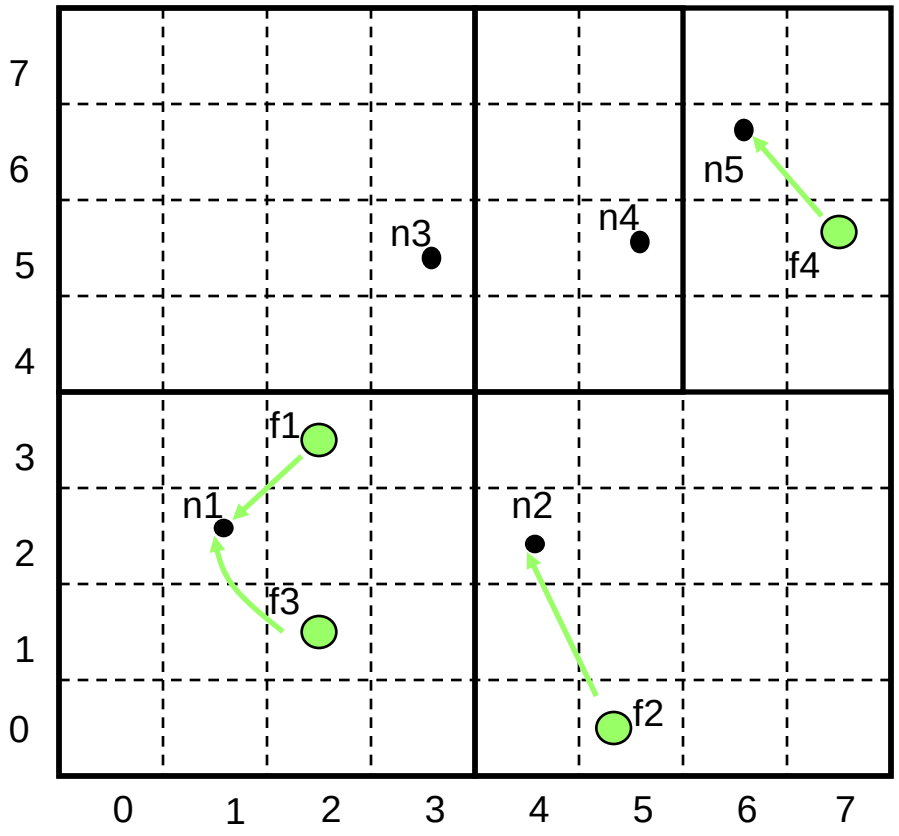
CAN Example: Two Dimensional Space

- Nodes: n1:(1, 2); n2:(4,2); n3:(3, 5); n4:(5,5);n5:(6,6)
- Items: f1:(2,3); f2:(5,1); f3:(2,1); f4:(7,5);



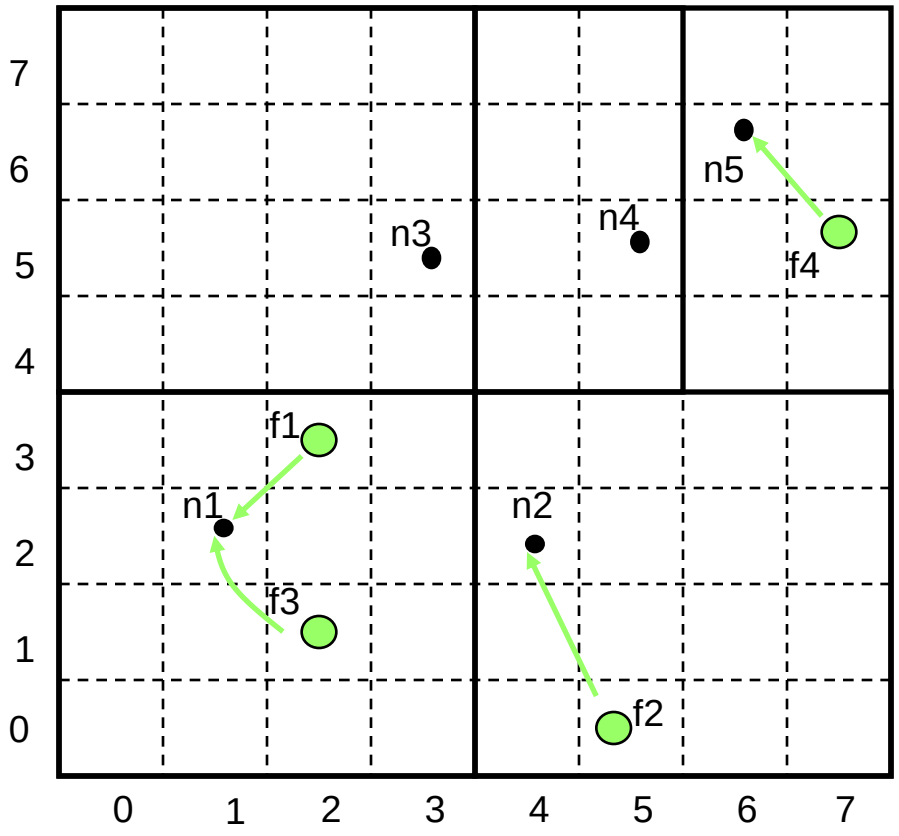
CAN Example: Two Dimensional Space

- Each item is stored by the node who owns its mapping in the space



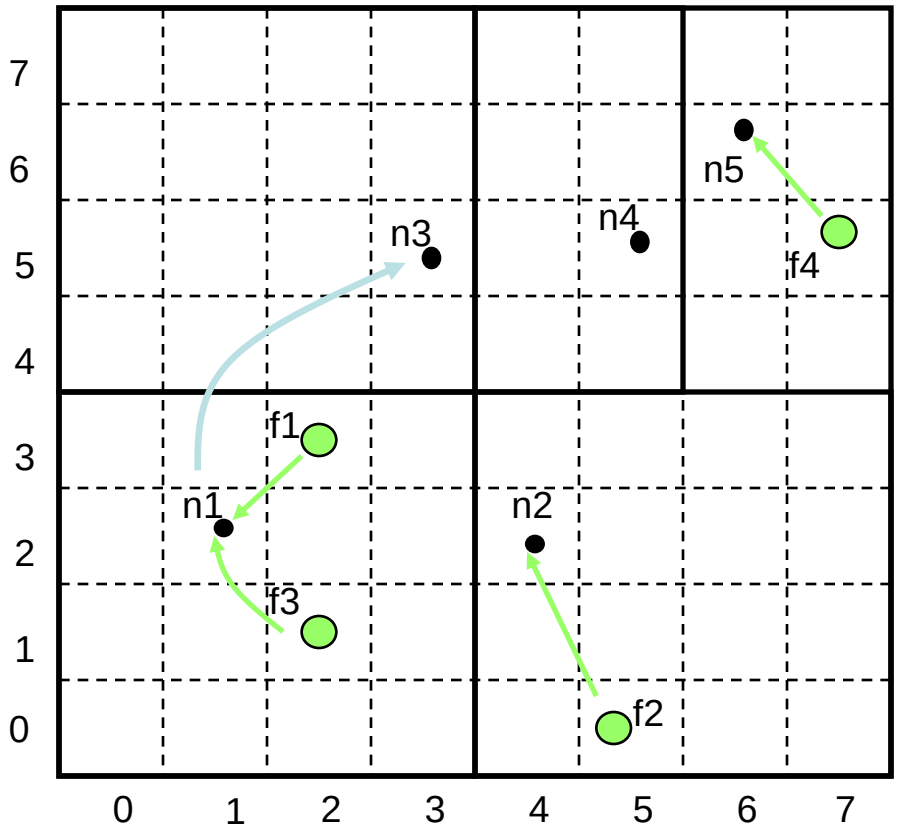
CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding



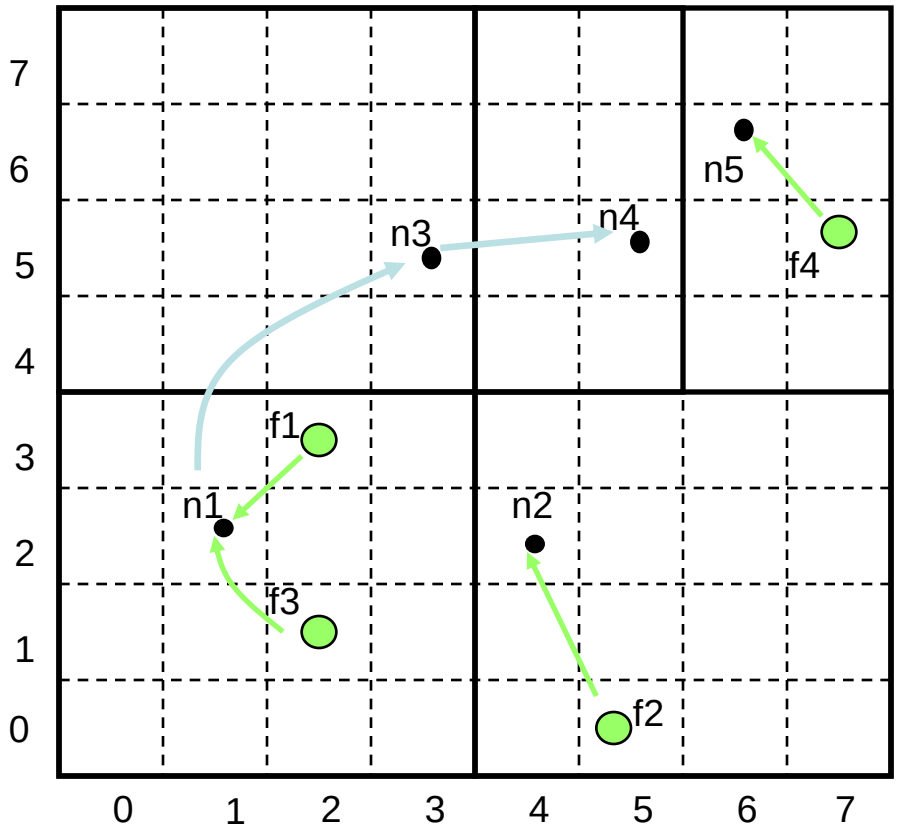
CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding



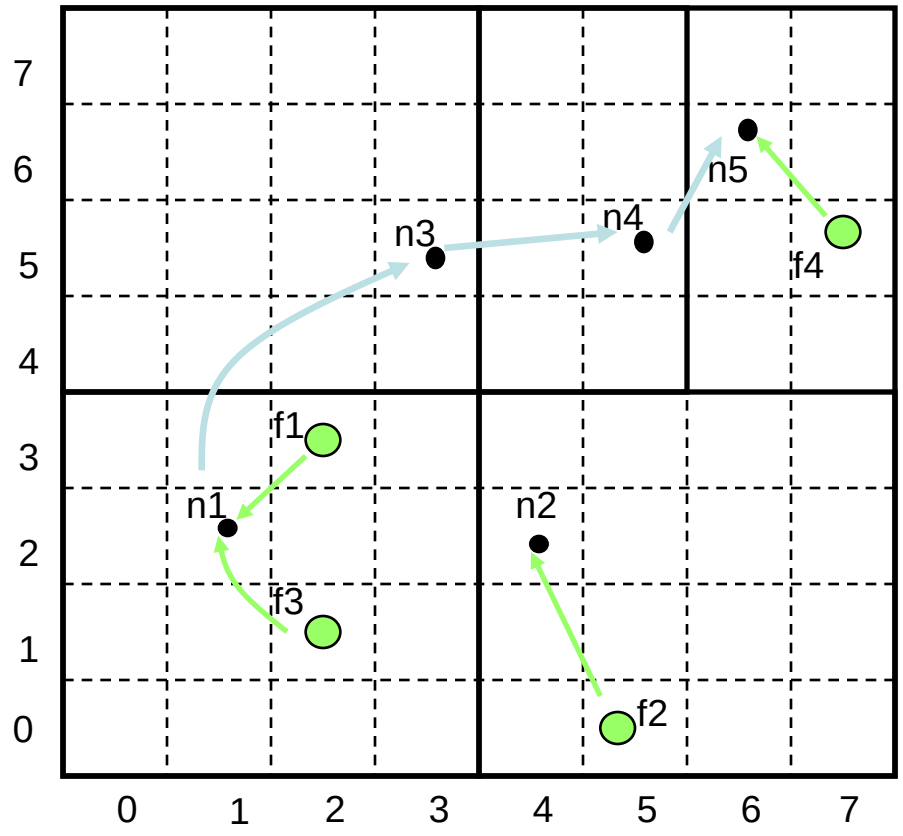
CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding



CAN: Query Example

- Each node knows its neighbors in the *d*-space
- Forward query to the neighbor that is closest to the query *id*
- Example: assume n1 queries f4
- Can route around some failures
  - some failures require local flooding



BATON Papers Overview

Jagadish, Ooi, and Vu: “BATON: A Balanced Tree Struture for P2P Networks,” VLDB 2005.

Jagadish *et al.*, “Speeding up Search in P2P Networks with a Multi-way Tree Structure,” SIGMOD 2006

Presented by Nick Taylor, October 4, 2006

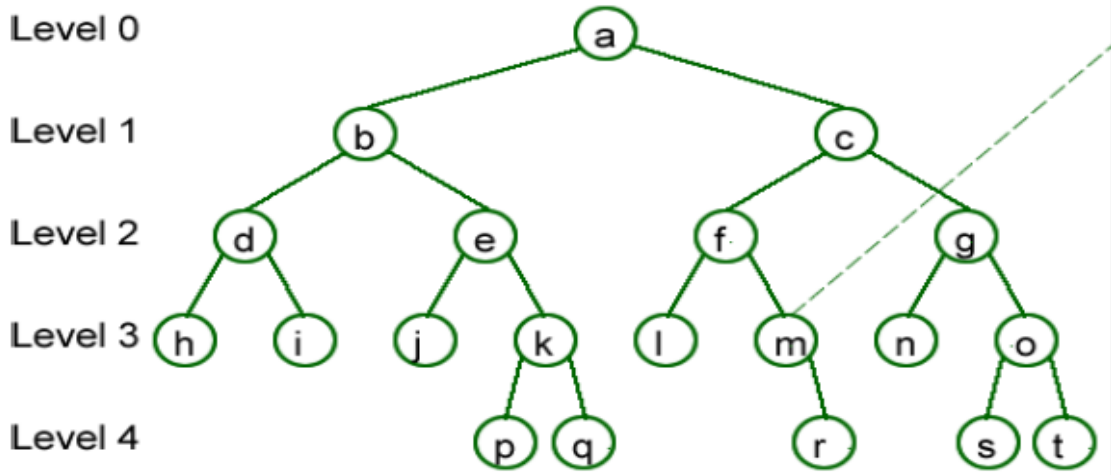
BATON: A Balanced Tree

- BATON: a balanced tree
  - binary balanced tree
- Capable of supporting
  - exact queries
  - range queries efficiently
  - sufficient fault tolerance to permit efficient repair
  - exact queries and range queries can be answered in O(logN) steps

Contributions

- First to build a P2P overlay network based on a balanced tree structure
- Log N steps for finding a place for the joining node or for finding a node to replace the leaving node
- Load balancing and adjust this range dynamically
- Fault Tolerance

BAlanced Tree Overlay Network



Node m: level=3, number=6
parent=f, leftchild=null, rightchild=r
leftadjacent=f, rightadjacent=r
Left routing table:

	Node	Left child	Right child	Lower bound	Upper bound
0	l	null	null	l_{lower}	l_{upper}
1	k	p	q	k_{lower}	k_{upper}
2	i	null	null	i_{lower}	i_{upper}

Right routing table:

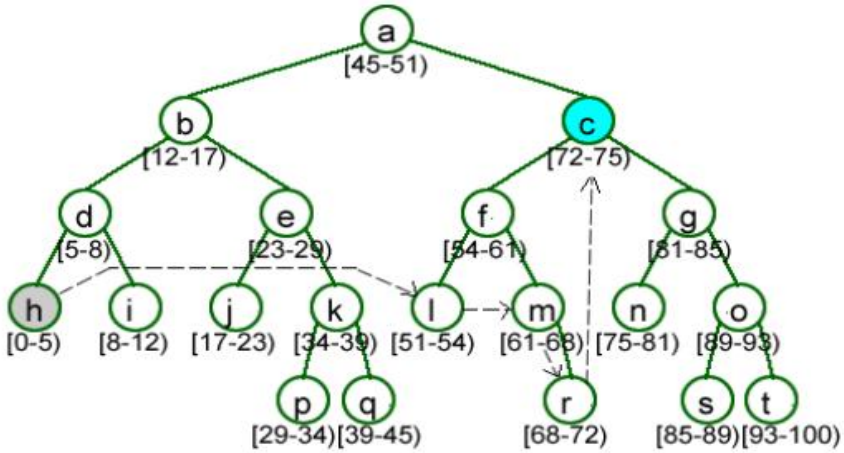
	Node	Left child	Right child	Lower bound	Upper bound
0	n	null	null	n_{lower}	n_{upper}
1	o	s	t	o_{lower}	o_{upper}

Note that routing table for a node *m* contains details about nodes $m\text{-}2^2$, $m\text{-}2^1$, $m\text{-}2^0$, $m+2^0$, $m+2^1$, sort of like Chord.

Lookup Algorithm

Algorithm: *search exact(node n, query q, value v)*

```
If ((LowerBound(n)<=v) and (v<=UpperBound(n)))
   q is executed at x [1]
Else
   If (UpperBound(n)<v)
      m=TheFarthestNodeSatisfyingCondition
           (LowerBound(m)<=v)
      If (there exists such an m)
         Forward q to m
      Else
         If (RightChild(n)!=null)
            Forward q to RightChild(n)
         Else
            Forward q to RightAdjacentNode(n)
         End If
      End If
   Else
      //A similar process is followed towards the left
   End If
End If
```

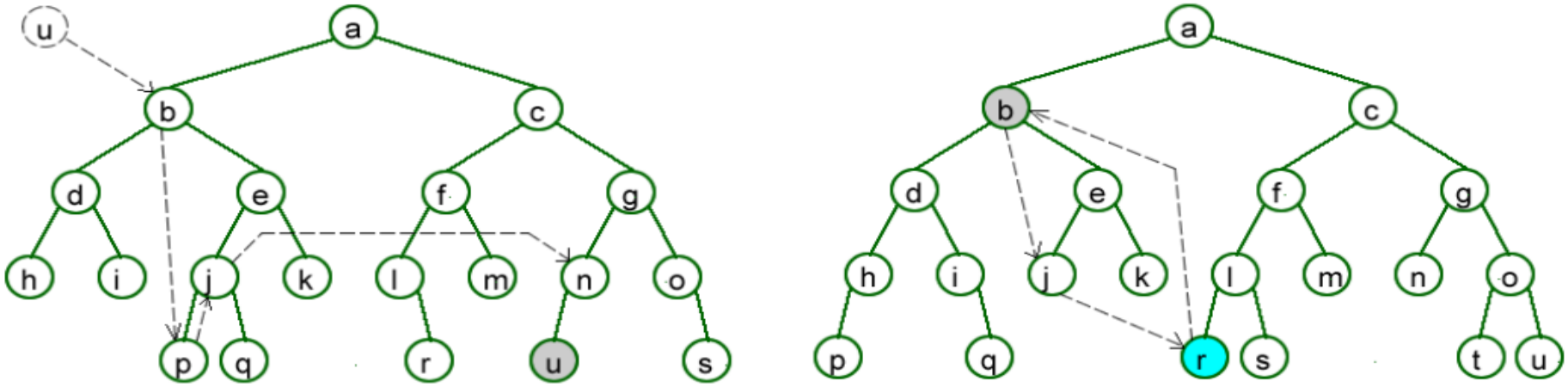


Lookup for 74 starting at node h

Lookup Algorithm (2)

- Rarely moves up tree
  - If at leaf node
  - If value is higher up tree
- At most log *N* messages sent to do a lookup
- Range query finds start of range using lookup procedure, then follows links between nodes
- Insertion and deletion use lookup procedure

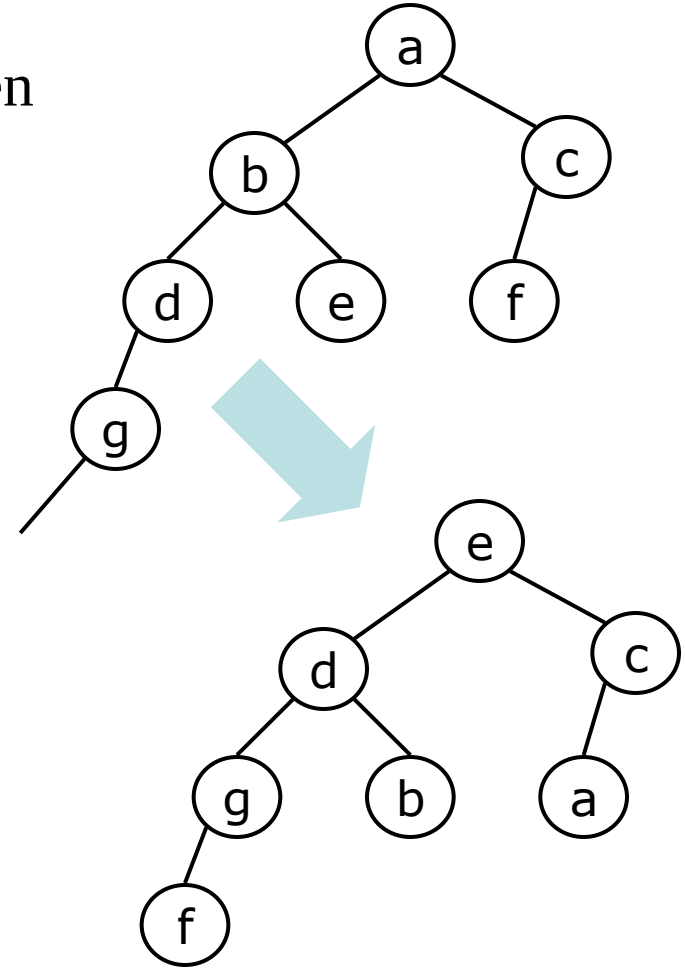
Node addition and removal rules



- Need to preserve balanced structure of tree
- Addition: forward request to node with full routing table but at most one child
- Removal: find leaf node that can be moved without causing imbalance

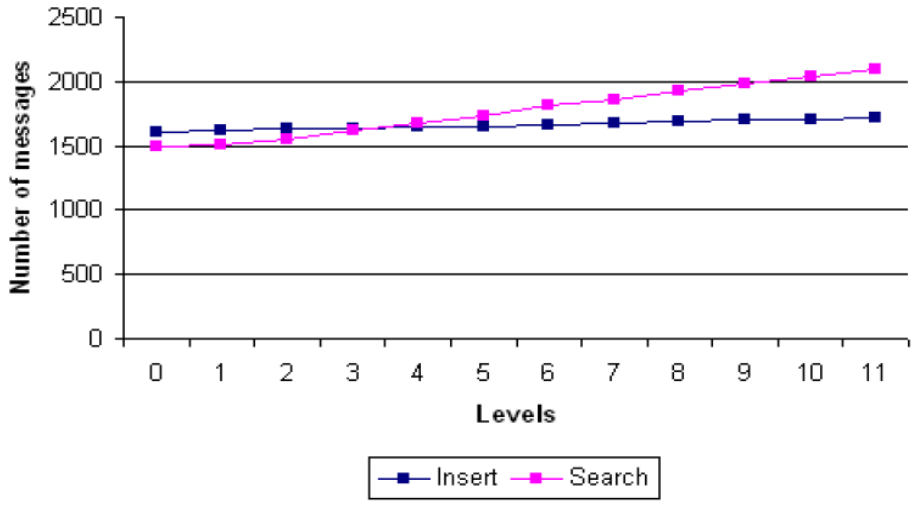
Load Balancing

- May need to redistribute ranges between nodes
- Simple case: just affects immediate neighbors
- General algorithm:
  - Identify overloaded node *g*
  - Identify lightly loaded node *f*
  - Transfer *f*'s data to *c*
  - Add *f* as child of *g*
  - If tree is unbalanced, perform rotation
- Need to update routing table
  - Still cheaper than moving data around



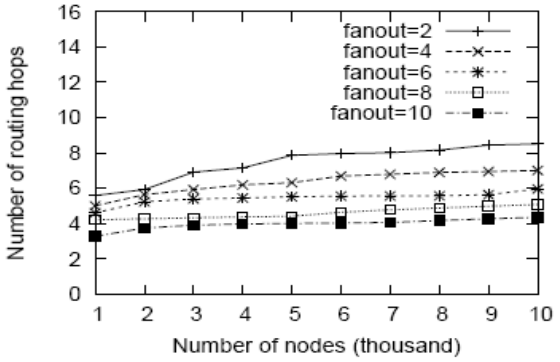
Experimental Results

- Maintenance slightly better than Chord
- Performance slightly worse than Chord
- Root node does not get overloaded
  - Leaves get slightly more traffic than root

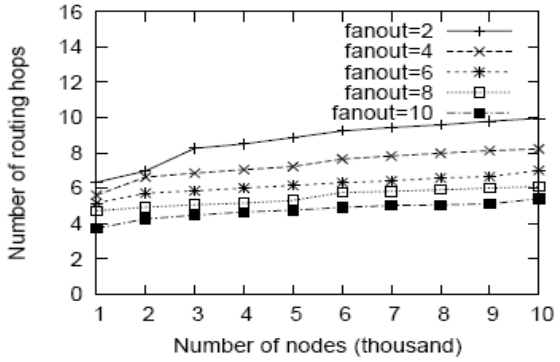


BATON*

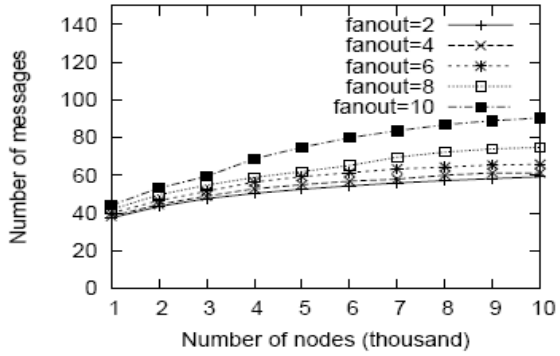
- Generalizes techniques from binary trees to *m*-ary trees
  - Improves query performance
  - Increases maintenance cost



(a) Cost of exact match query



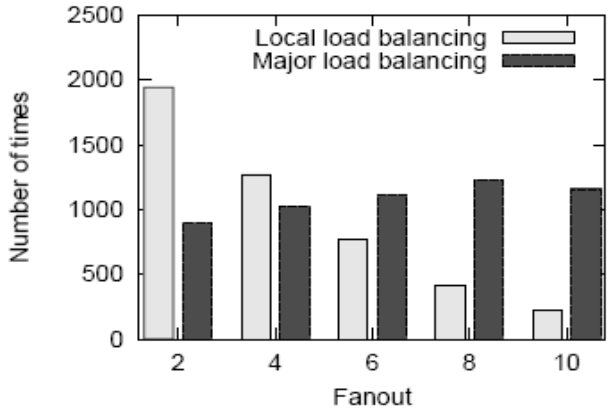
(b) Cost of range query



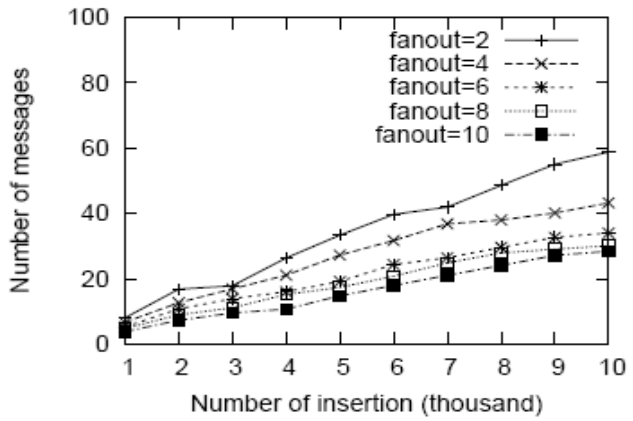
(c) Cost of updating routing table

Fault tolerance and load balancing

- *m*-ary trees are harder to partition
- *m*-ary trees have more leaf nodes
  - Easier to do "local" load balancing



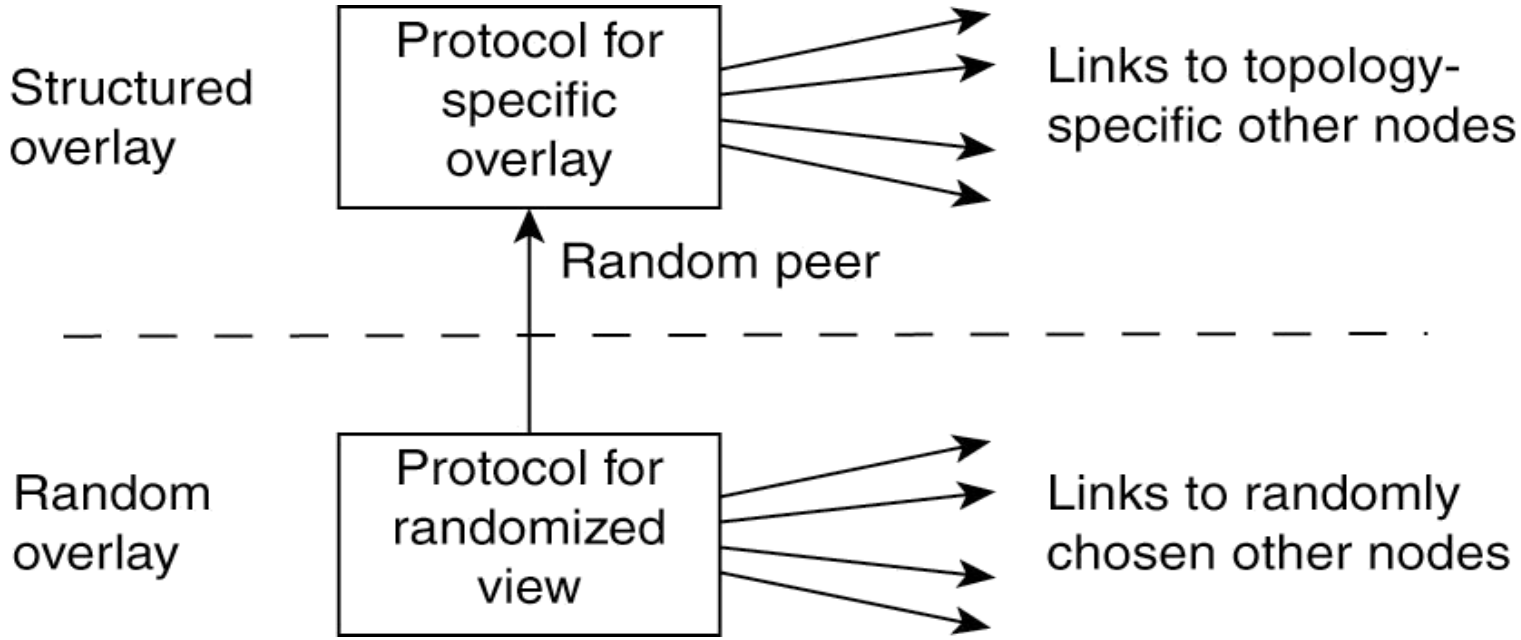
(a) Number of times each load balancing scheme is invoked



(b) Average additional messages required for doing load balancing

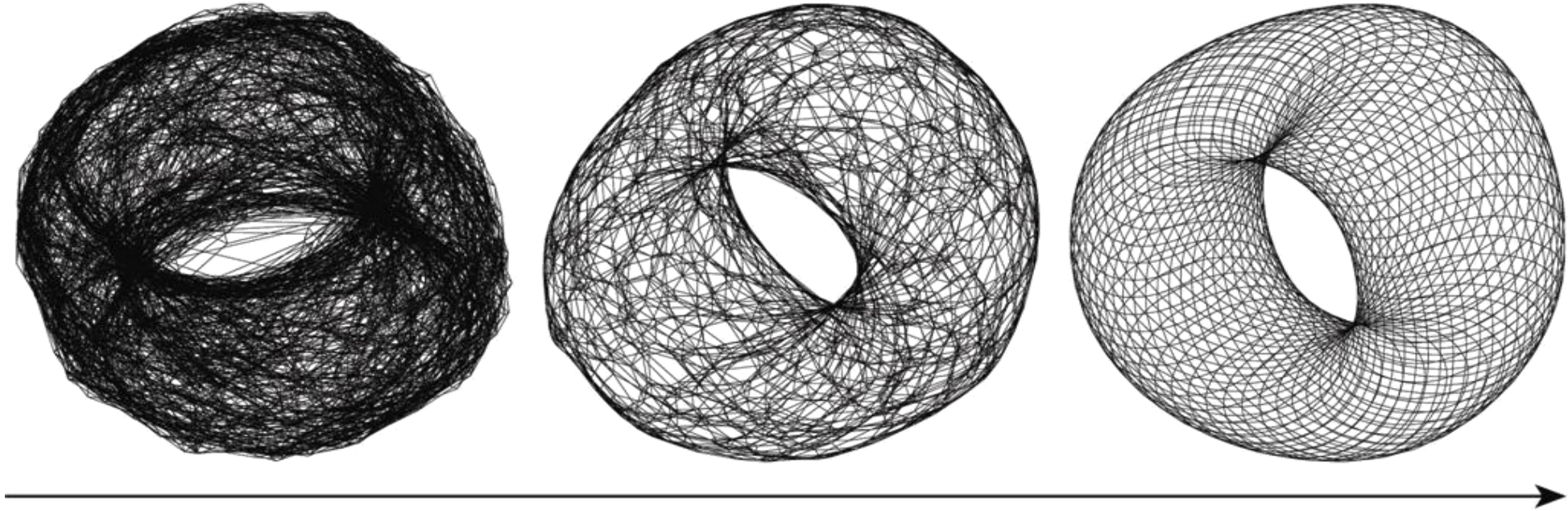
P2P网络的拓扑管理

- 使用无结构P2P技术构造特定的结构化P2P覆盖网络的两层结构



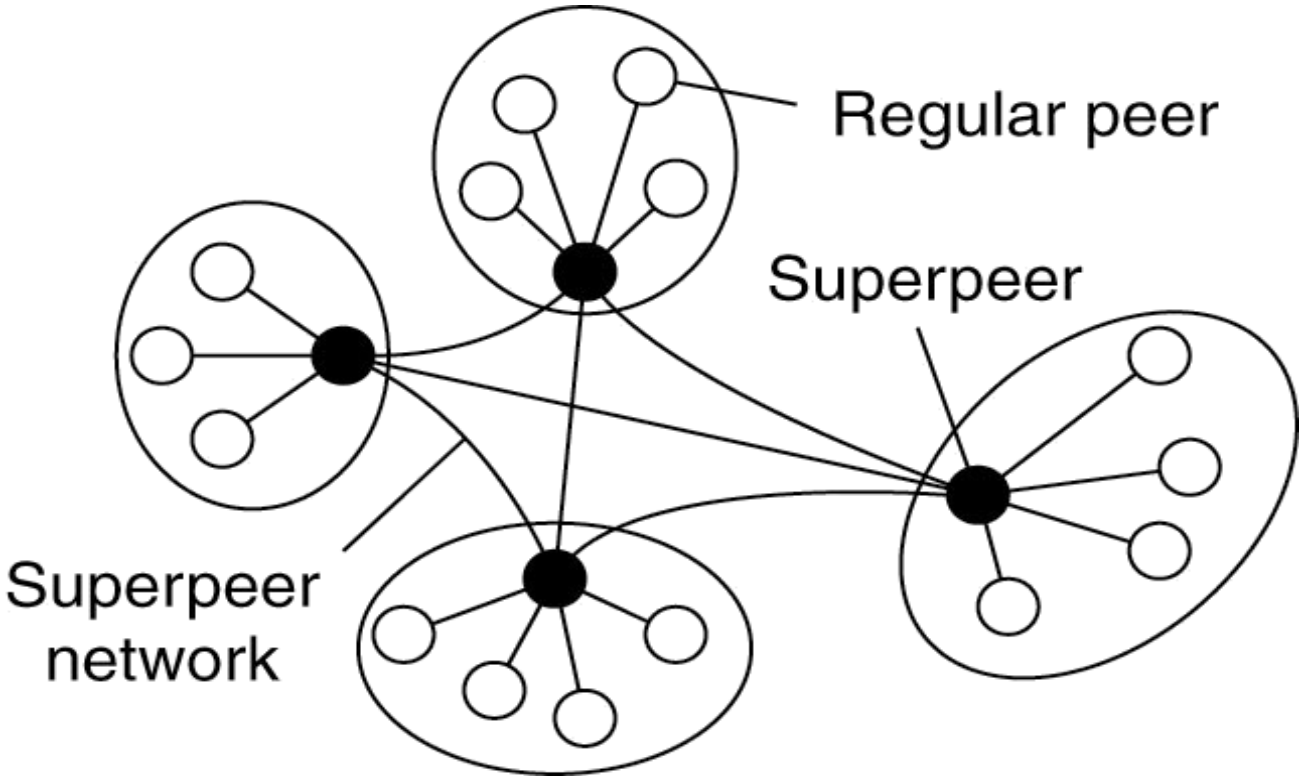
P2P网络的拓扑管理

- 使用两层的无结构P2P系统，最终生成的特定的覆盖网络

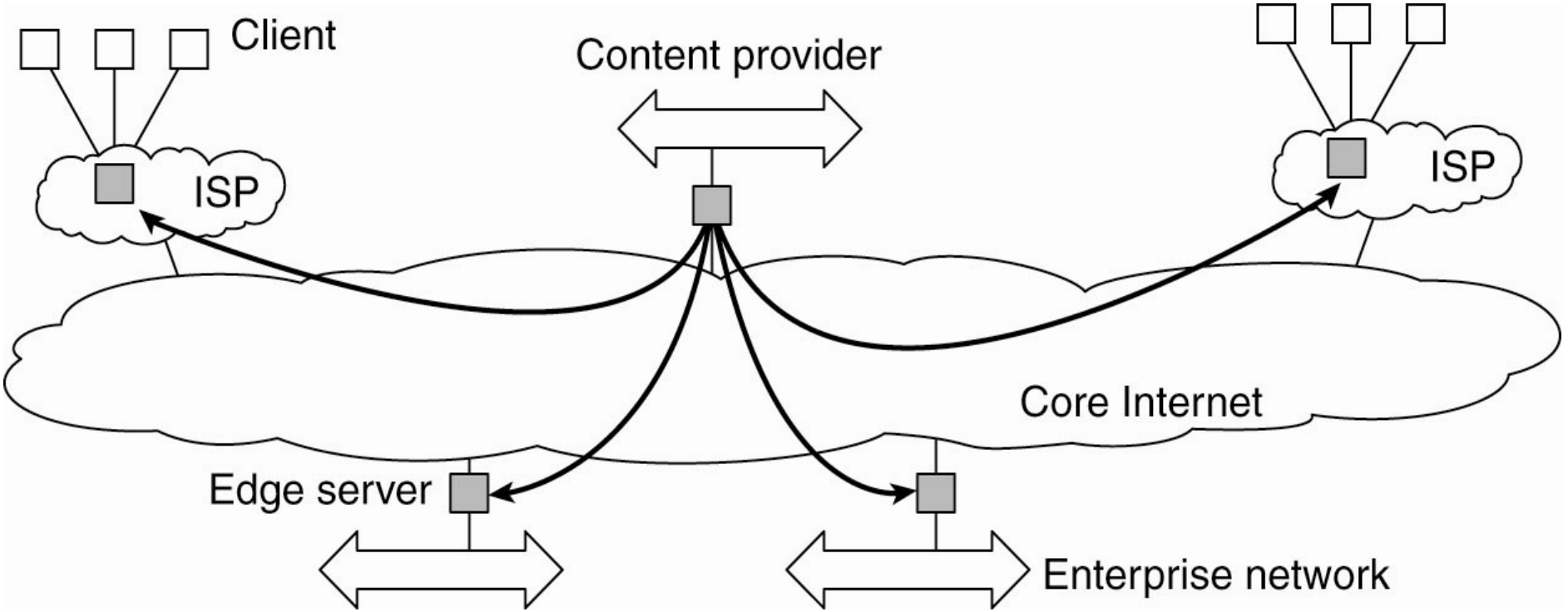


超级节点（Superpeer）

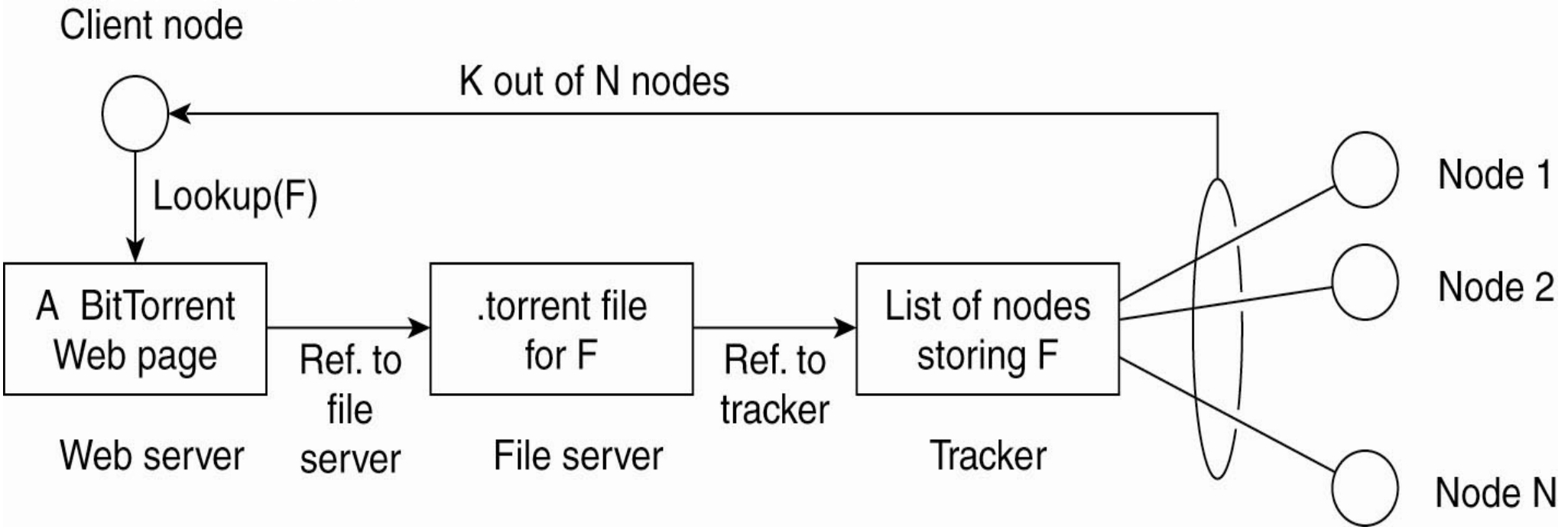
- 超级节点：能够维护索引或充当代理的节点



混合体系结构


Client
Content provider
ISP
ISP
Core Internet
Edge server
Enterprise network

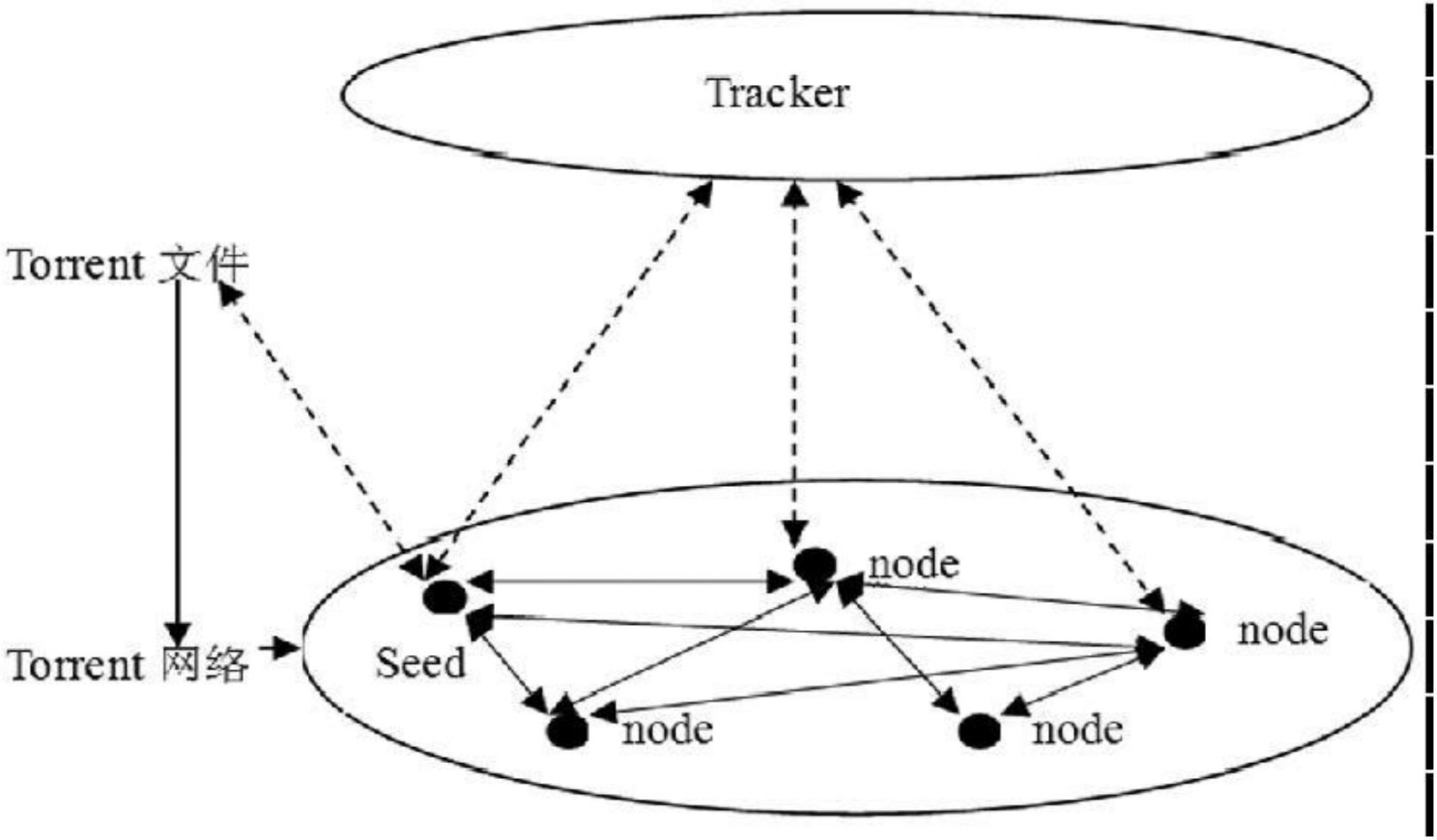
协作分布式系统


Client node
K out of N nodes
Lookup(F)
A BitTorrent Web page
Ref. to file server
.torrent file for F
Ref. to tracker
List of nodes storing F
Node 1
Node 2
Node N
Web server
File server
Tracker

BT 下载网络有三个关键静态组件：

- 跟踪器（Tracker）：Tracker 跟踪器是一个中央服务器，它主要跟踪系统中所有的参与结点，收集和统计这些结点的状态，帮助参与结点间互相发现并进行文件块的交换；
- 种子节点（Seed）:Seed 种子节点是指拥有完整文件的节点，提供上载服务；
- 下载节点（Downloader）。相对于Seed 的节点称为下载节点，一个下载节点完成下载后，可以成为种子节点

BitTorrent系统结构



动态流程

- 第一个用户通过BT工具制作要共享文件的Torrent 文件（Torrent文件包含共享文件的下载信息）并发布此Torrent文件到WWW中。
- 其他用户从WEB服务器上下载此Torrent 文件并通过节点跟踪器协议（如TrackerHTTP）去访问Tracker 跟踪器，参与到此Torrent网络中。
- Tracker跟踪器接收到一个新加入节点的下载请求后，随机选择部分此Torrent网络中的节点发送给新加入者作为邻居节点，并记录新节点。
- 新加入节点通过一定的算法同邻居节点连接进行文件的下载和上载直到文件下载完成，这一过程会根据一定的策略重复。如果继续上载，Tracker 服务器将此节点看作种子节点。
- 所有参与的节点将周期地报告自己的状态和进程给Tracker 跟踪器.

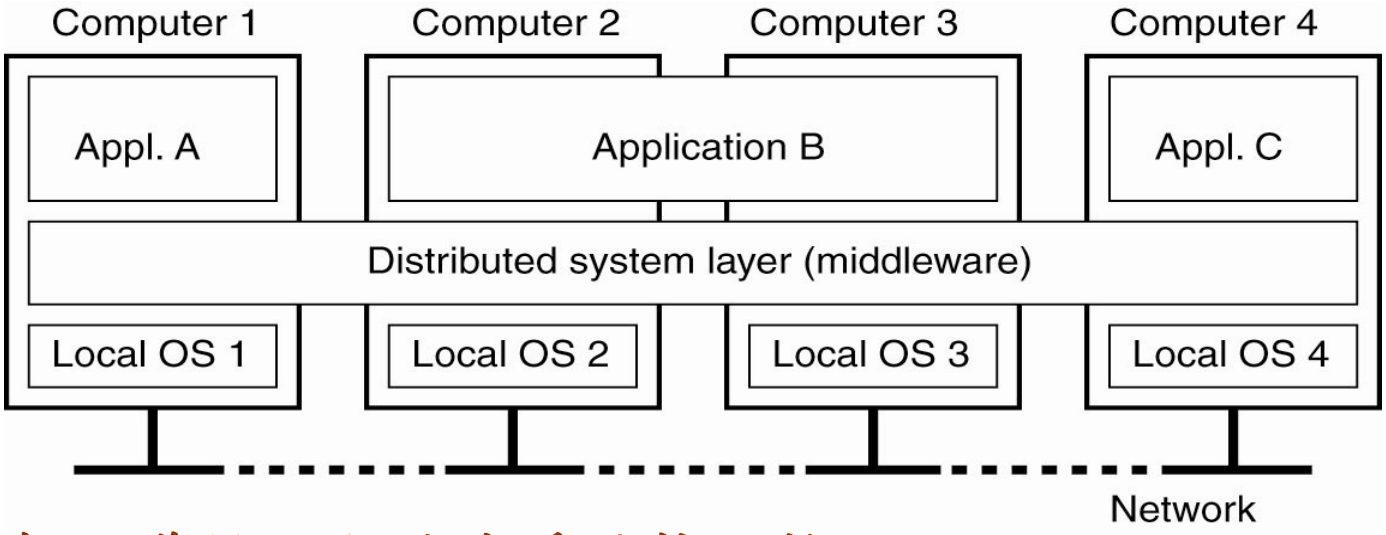
BT 关键技术

- BT 文件发布系统采用针锋相对（Tit_for_Tat）的方法来达到帕累托（pareto）有效，与当前其他的P2P技术相比，它达到了更高层次的鲁棒性和资源利用。
- 帕累托最优:指资源配置已达到这样一种境地，即任何重新改变资源配置的方式，都不可能使一部分人在没有其他人受损的情况下受益。
- 最少优先原则:对一个下载者来说，在选择下一个被下载的片断时，通常选择的是它的Peers 所拥有的最少的那个片断，也就是所谓的“最少优先”。

第2章 体系结构

- 2.0 简介
- 2.1 软件体系结构样式
- 2.2 系统体系结构
- 2.3 体系结构与中间件
- 2.4 分布式系统的自我管理
- 2.5 基础模型
  - 交互模型
  - 故障模型
  - 安全模型

2.3 体系结构与中间件



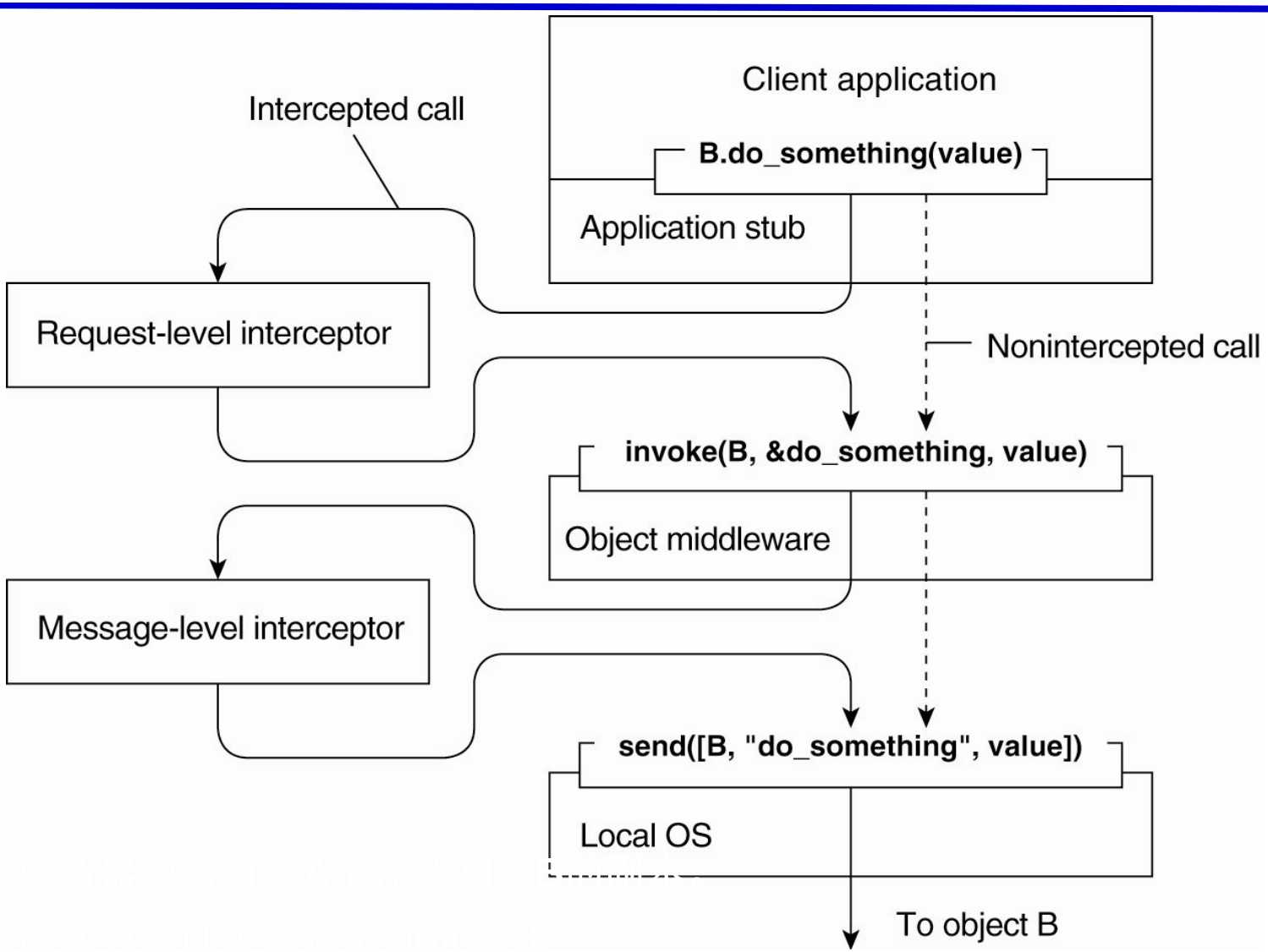
中间件一般要遵循一定的体系结构风格：

- 基于对象的体系结构
- 基于事件的体系结构
- 中间件可以按照应用程序的需求方便的进行配置、适应和定制：
  - 拦截器
  - 自适应软件的常见方法

拦截器（Interceptor）

- Interceptor是CORBA规范提出的一种重要思想。它允许扩充中间件的功能而无须改变中间件的核心构造。
- Interceptor pattern是一种设计模式,它允许将服务透明地添加到框架中,当某一事件发生时,可以自动触发。这种模式可以保护组件免受运行环境的影响,应用程序开发者可以集中精力于业务逻辑的开发。

使用拦截器处理远程对象激活



自适应软件的常见方法

实现软件自适应性的基本技术:

- 要点分离(Separation of concerns)：实现功能的部分与负责可靠性、性能和安全的部分分开
- 计算反射(Computational reflection)：程序检查自己，调整其行为的能力
- 基于组件的设计(Component-based design)：通过组件的不同组合来支持自适应

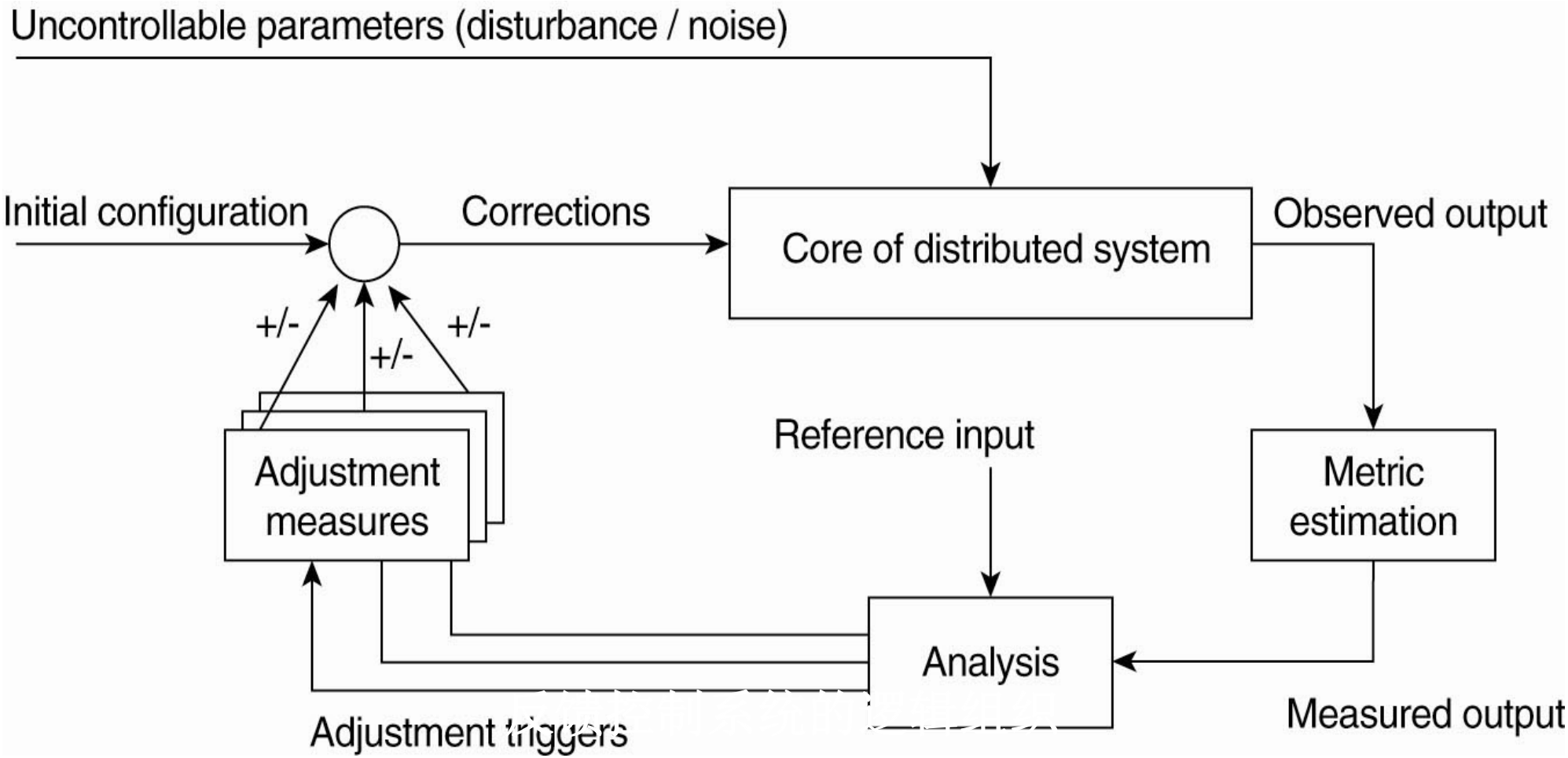
第2章 体系结构

- 2.0 简介
- 2.1 软件体系结构样式
- 2.2 系统体系结构
- 2.3 体系结构与中间件
- 2.4 分布式系统的自我管理
- 2.5 基础模型
  - 交互模型
  - 故障模型
  - 安全模型

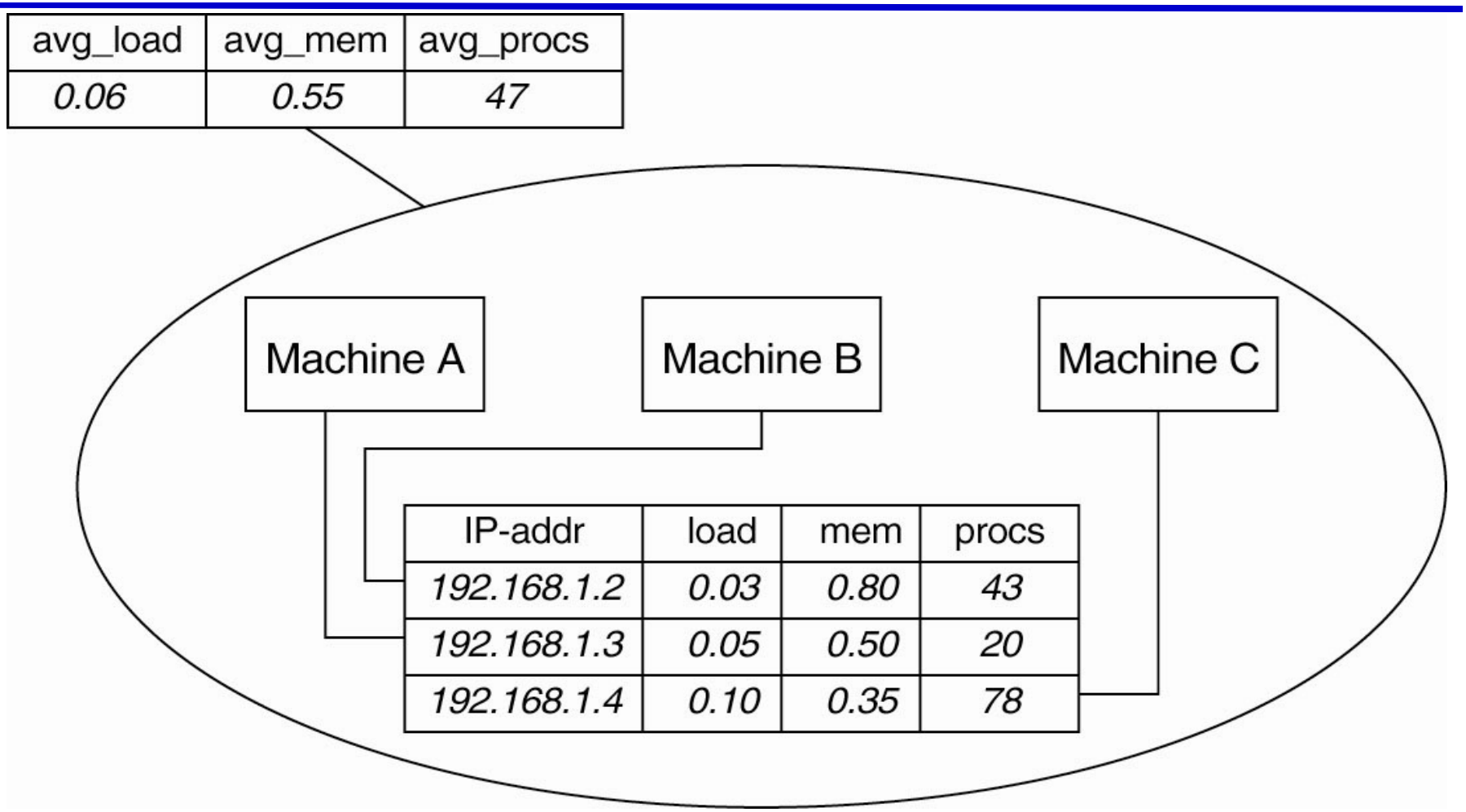
2.4 分布式系统的自我管理

- 以高级反馈控制系统的形式来组织分布式系统，允许自动适应变化
- 自治计算或自主系统：自我管理、自我恢复、自我配置和自我优化
- 反馈控制模型
  - Astrolabe监视系统
  - Globule中的差分复制策略
  - Jade的自动组件修复管理

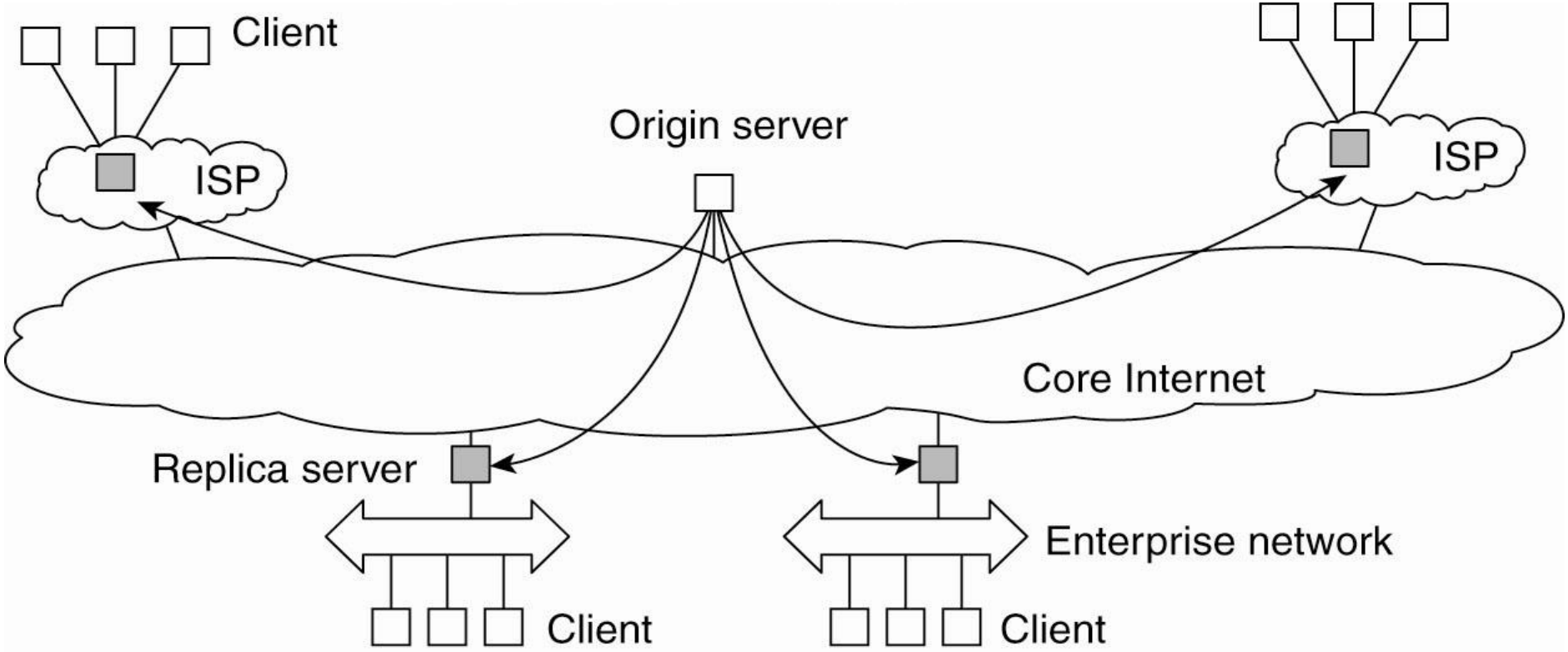
反馈控制模型


Uncontrollable parameters (disturbance / noise)
Initial configuration
Corrections
Core of distributed system
Observed output
+/-
+/-
+/-
Adjustment measures
Reference input
Metric estimation
Analysis
Measured output
Adjustment triggers

示例：Astrolabe监视系统



示例:Globule中的差分复制策略



示例:Jade的自动组件修复管理

- 对于基于组件的服务器，可以检测组件故障，并自动代替它们
- 修复管理域
- 结点管理器
- 故障检测器:每个结点一个
- 修复策略的执行步骤:
  - 终止无故障结点组件与已发生故障结点组件之间的所有绑定
  - 请求节点管理器启动和添加新结点到该域
  - 配置新节点使之具有与已崩溃结点相同的组件
  - 重新建立前面已终止的绑定

第2章 体系结构

- 2.0 简介
- 2.1 软件体系结构样式
- 2.2 系统体系结构
- 2.3 体系结构与中间件
- 2.4 分布式系统的自我管理
- 2.5 基础模型
  - 交互模型
  - 故障模型
  - 安全模型

2.5 基础模型 Fundamental Models

Distributed systems based on different architectural models share some fundamental properties. They are described in fundamental models

基于不同结构模型的分布式系统有相同的基本特性。

1. Interaction model 交互模型

- Communication delays 通信延迟
- Accuracy of process coordination进程协调的精确性
- Notion of time in a distributed system
- 分布式系统中的时间概念

- **2. Failure model故障模型**
  - Definition and classification of faults
  - 错误的定义与分类
- **3. Security model安全模型**
  - Classification of attacks攻击的分类
  - Security threat analysis安全威胁分析

Interaction Model 交互模型

- Performance of communication channels通信信道的性能
  - **Latency**: Delay between sending a message by one process and receiving it by another.
  - 等待时间：一个进程发送消息另一个进程接受消息之间的延时。
    - software layer delay, network access delay, transmission delay
  - **Bandwidth**: Amount of information that can be transmitted over a network in a given time.
  - 带宽：给定时间内网络上能传递消息的总量。
  - **Jitter**: Variation in the time taken to deliver a series of messages.
  - 抖动：传递一系列消息所用时间的变化值。

Interaction Model 交互模型

- Computer clocks and timing events计算机时钟和时序事件
  - **Clock drift rate**: The relative amount that a local computer clock differs from a perfect clock
  - 时钟漂移率：计算机时钟不同于绝对参考时钟的相对量。
  - Local computer clocks can be synchronized by messages.
  - 本地计算机时钟能被消息同步。
    - This procedure is subject to message delays.
    - 进程受限于消息延迟。

Synchronous vs. Asynchronous（同步vs. 异步）

Synchronous distributed systems同步的分布式系统

- The time to execute each step of a process has known lower and upper bounds.
- 进程执行每一步的时间有一个上限和下限。
- Each message transmitted over a channel is received within a known bounded time.
- 通过通道传递的消息在一个已知时间内收到。

Synchronous vs. Asynchronous（同步vs. 异步）

- Each process has a local clock whose drift rate from real time has a known bound.每一个处理有一个本地时钟，它与实际时间的漂移率在一个已知的范围内。

It is difficult to obtain **realistic** values for the bounds and to **guarantee** them.

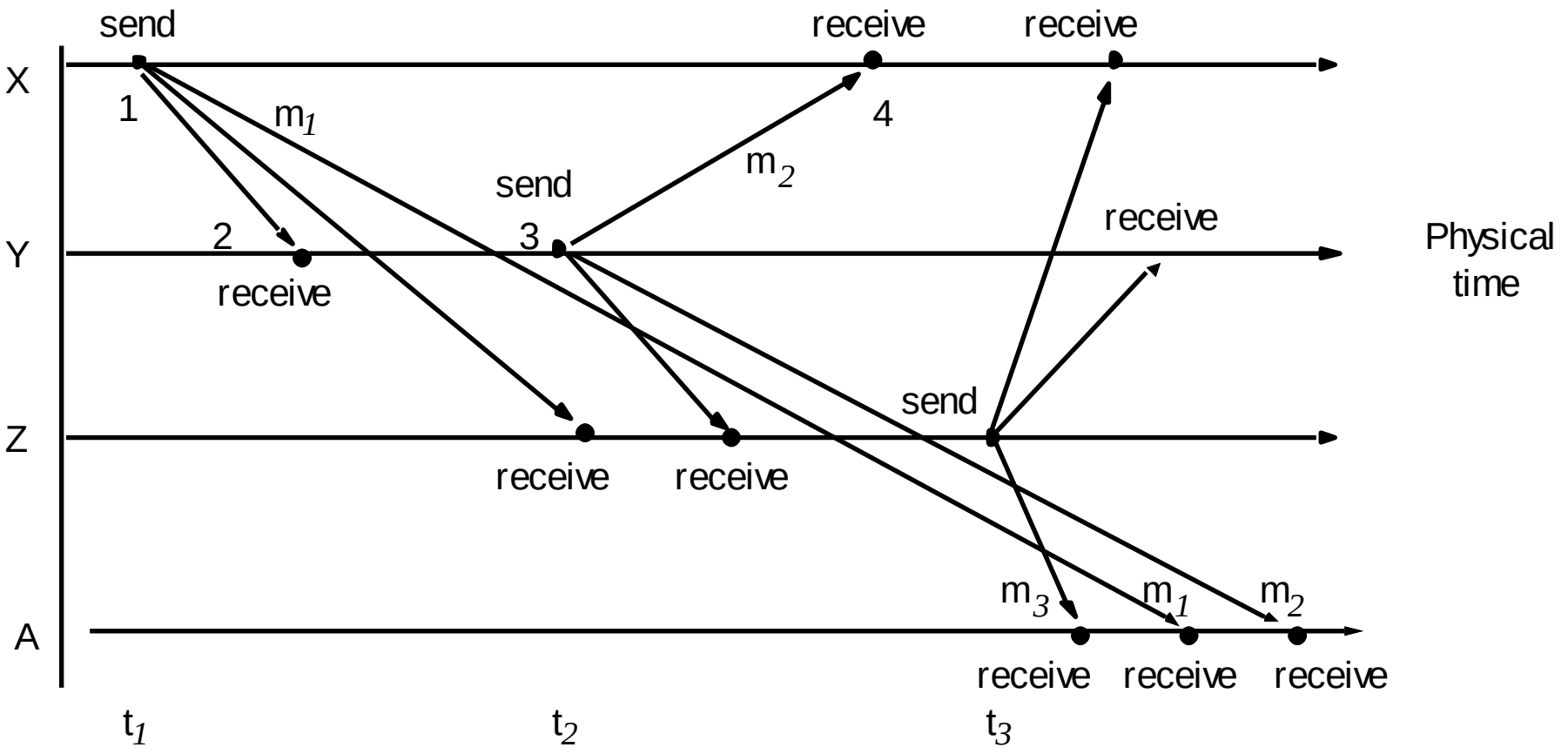
达到实际值和提供对所选值的保证是比较困难的。

Asynchronous distributed systems异步的分布式系统

There are no bounds on process execution speed, message transmission delays and clock drift rates.

- 对进程执行速度、消息传递延时和时钟漂移率没有限制

Figure 2.8
Real-time ordering of events 实时时间排序



Failure Model　故障模型

Processes **and** communication channels may fail in a distributed system.

在分布式系统中，进程和信道都可能发生故障。

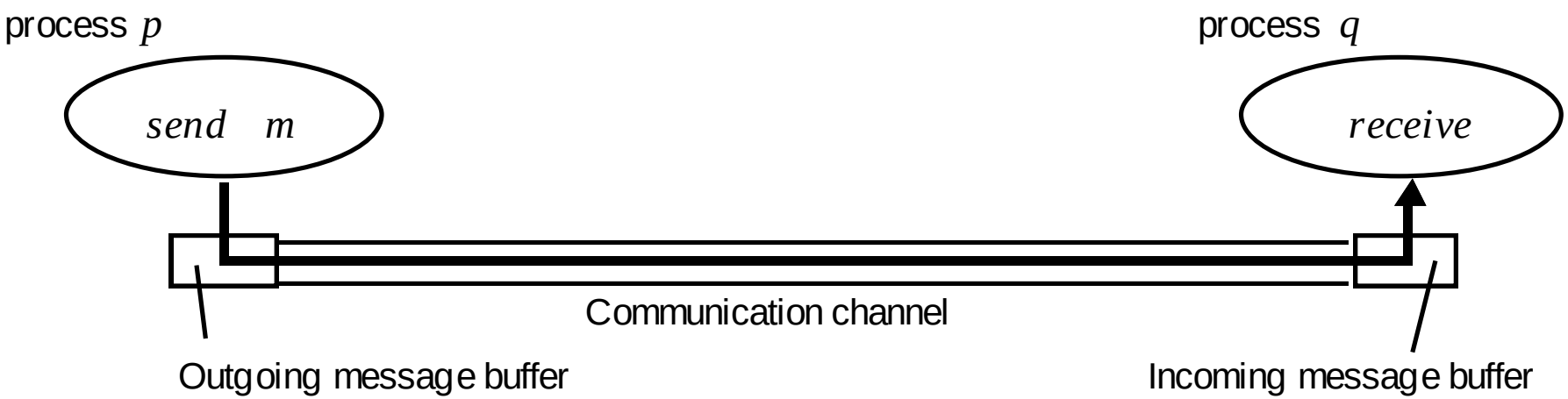
- **Omission failure**: 遗漏故障

  A process or a communication channel fails to perform actions it is supposed to do.

  进程或信道不能完成所要做的动作。

通信遗漏故障：考虑通信原语send和receive。

Figure 2.9
Processes and channels



Failure Model 故障模型

- **Arbitrary (Byzantine) failures**: 随机故障（拜占庭故障）

  Any type of failure may occur. 可能发生的任何类型的故障

  - A process may execute wrong steps or it may selectively execute only a few of the necessary steps.

- **Timing failures**:时序故障

  Those failures occur only in synchronous distributed systems when bounds on the process execution time, message delivery time or clock drift rate are exceeded.

  这个故障发生在同步的分布式系统中。进程的执行时间，消息传递时间和时钟漂移率均有限制。

Omission And Arbitrary Failure 遗漏和随机错误

	Class of failure	*Affects*	*Description*
故障-停止	Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
崩溃	Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
遗漏	Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
发送遗漏	Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
接收遗漏	Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
随机（拜占庭）	Arbitrary (Byzantine)	Process or Channel	A channel may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step: **arbitrary behaviour**

Reliable Communication可靠通信

If omission failures are masked then a communication is called **reliable**. This includes:

- **Validity**: Any message in the outgoing message buffer is eventually delivered to the incoming message buffer.
- 遗漏故障被标记，我们称通信是可靠的。包括：
- 有效性：
- 任何发出消息缓冲的消息最终被传递到接收消息缓冲区。

- **Integrity**: The message received is identical to the one sent, and no message is delivered twice.
- 完整性：接收到的消息与发送的一致，没有消息被两次发送。
  - A protocol does not support integrity if it retransmits messages but does not reject a message that arrives twice.如果重发消息但不拒绝到达两次的消息的协议，那么，协议不支持完整性
  - Integrity is not preserved if malicious users can
    - inject spurious messages,
    - replay old messages or
    - tamper with messages

    without detection.
- 如果恶意用户，插入伪造信息

Timing Failures时序故障

Class of failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.进程的本地时钟超过了与实际时间的偏移率的范围
Performance	Process	Process exceeds the bounds on the interval between two steps.进程超过了两个进程之间的间隔的范围
Performance	Channel	A message's transmission takes longer than the stated bound.消息传递花了比规定的范围更长的时间

Security Model 安全模型

Security of a distributed system can be achieved by

- securing the processes,
- 进程的安全
- securing the channels used for interaction between the processes
- 安全通道用于进程间的交互
- protecting the objects against unauthorized access保护对象阻止非授权访问
  - encapsulating the objects 压缩的目标

Security Model 安全模型

- managing access rights 管理访问权限

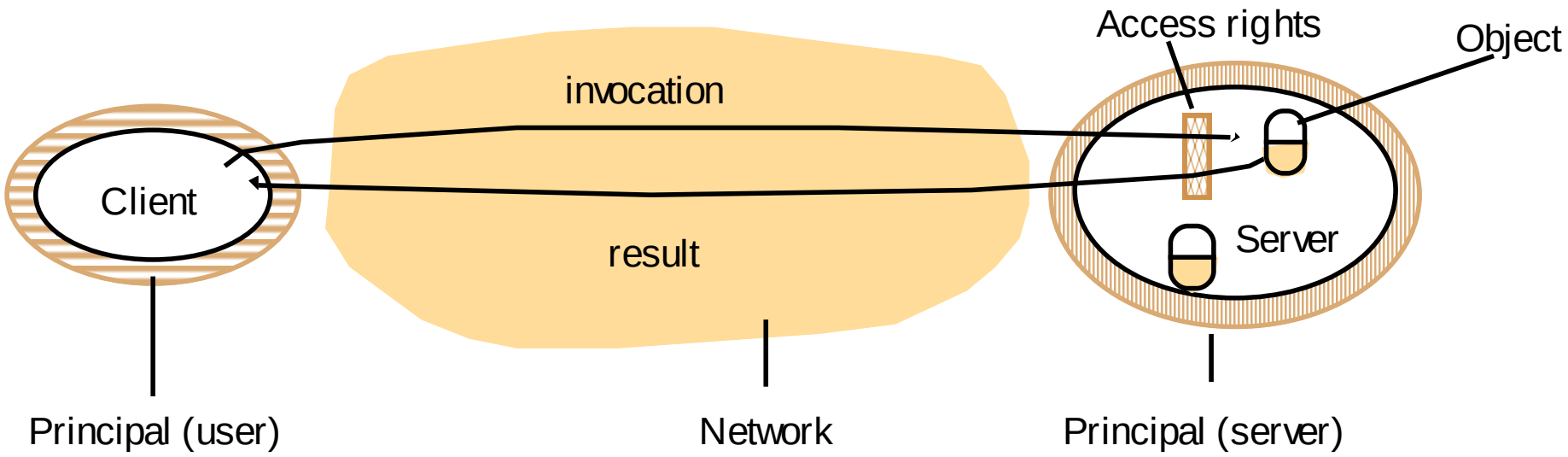
  Users and processes are beneficiaries of access rights. 用户和进程是访问权限的受益者。

Each invocation and each result must be associated with an authority (**principal**) on which behalf it is issued. 调用和结果与授权有关。

The server (client) is responsible to verify the identity of the principal behind each invocation (result) and to check whether the principal has sufficient access rights to perform the desired operation.

客户服务器有权验证身份、检察访问权限。

Figure 2.12
Objects and principals



Threats to Processes and Channels 对进程和通道的威胁

- Servers can receive invocations from many different clients without being able to reliably determine the identity of the principal of each invocation without affecting its functionality.
- 服务器从不同的客户端接收调用。
  - Example: Web server
  - The server must be protected against invocations from malicious processes.
  - 服务器必须被保护防止恶意进程的调用。

- Clients may receive results whose server address has been changed or results from an unrelated invocation.
  - 当服务器地址被改变或不相关调用的发生，客户可以接收结果
- An enemy can copy, alter or inject messages into the communication channel.
- 一个敌人可以复制、修改或插入消息到通信通道中。
  - The integrity the information is threatened.
  - 信息的完整性受到威胁。
  - A *secure channel* must be used.安全通道被使用

Figure 2.13
The enemy

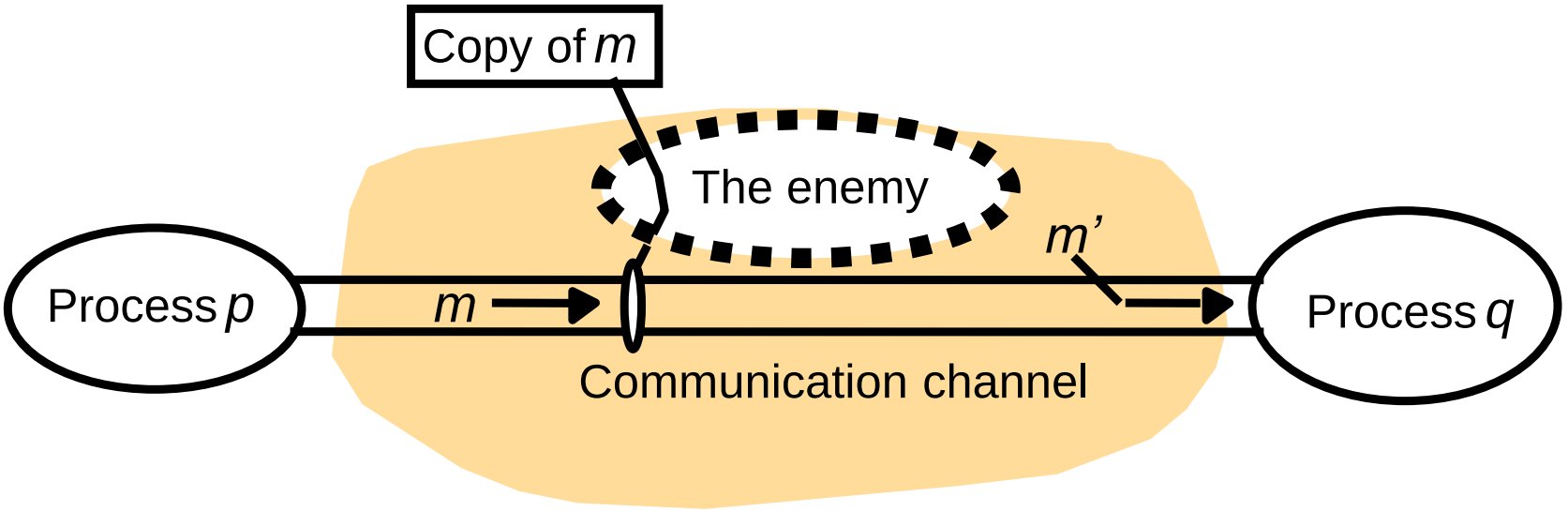
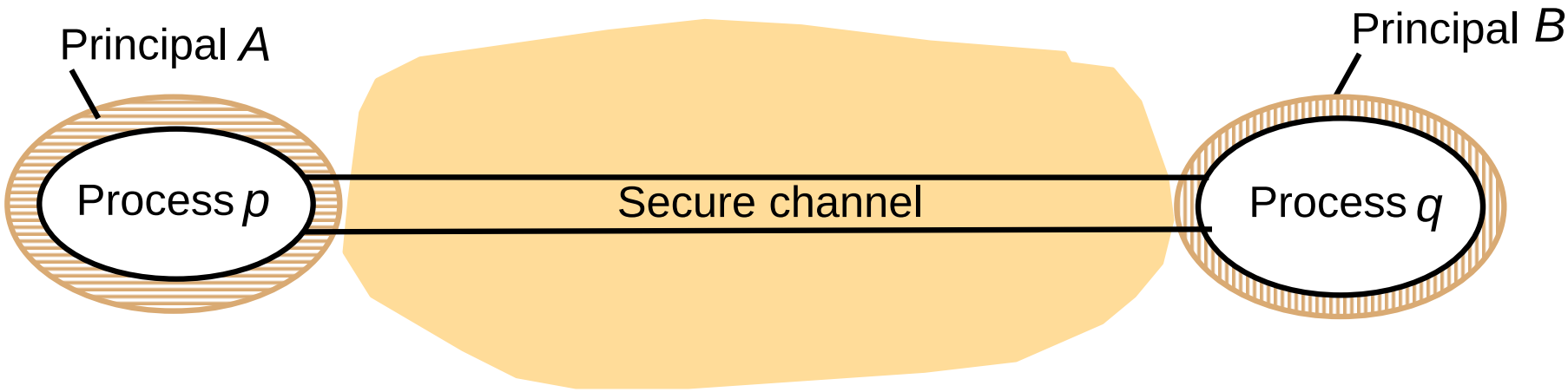


Figure 2.14
Secure channels



Design Issues 设计问题

Quality of service 服务质量

- Performance
  - Responsiveness: Fast and consistent response to interaction 响应性：对交互的快速和一致的应答。
  - Throughput: Rate at which computational work is done
  - 吞吐量：计算工作进行的比例。

    This is supported by load balancing.

    这需要负载平衡的支持。

Design Issues设计问题

- Reliability可靠性
  - Correctness: Any result returned to a user is correct.
  - 正确性：任何结果返回给用户都是正确的。
  - Fault tolerance: The system can tolerate many failures.
  - 容错性：系统应该可以容错许多错误。
- Adaptability： to meet changing system configurations and dynamic resource availability.
- 适应性:满足变化的系统配置和动态的资源可用性。
- Security安全性;
- Protection of user data and user resources
- 对用户数据和用户资源的保护。

第2章　小结

- 2.0 简介
- 2.1 软件体系结构样式
- 2.2 系统体系结构
- 2.3 体系结构与中间件
- 2.4 分布式系统的自我管理
- 2.5 基础模型
  - 交互模型
  - 故障模型
  - 安全模型

第2章　思考题

- XXX分布式系统的处理示例：
- 0. 软件架构图与功能模块
- 1. 软件体系结构样式
- 2. 系统体系结构设计
- 3. 体系结构与中间件实现
- 4. 分布式系统的自我管理机制
- 5. 基础模型的设计模块