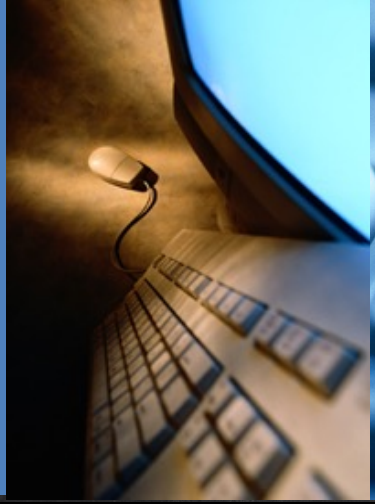
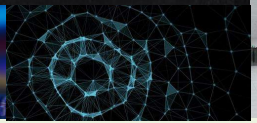



分布式系统
Distributed Systems

陈鹏飞
数据科学与计算机学院
chchenpf7@mail.sysu.edu.cn
办公室：超算5楼529d
主页：http://sdcs.sysu.edu.cn/node/3747



第二讲 — 分布式系统架构

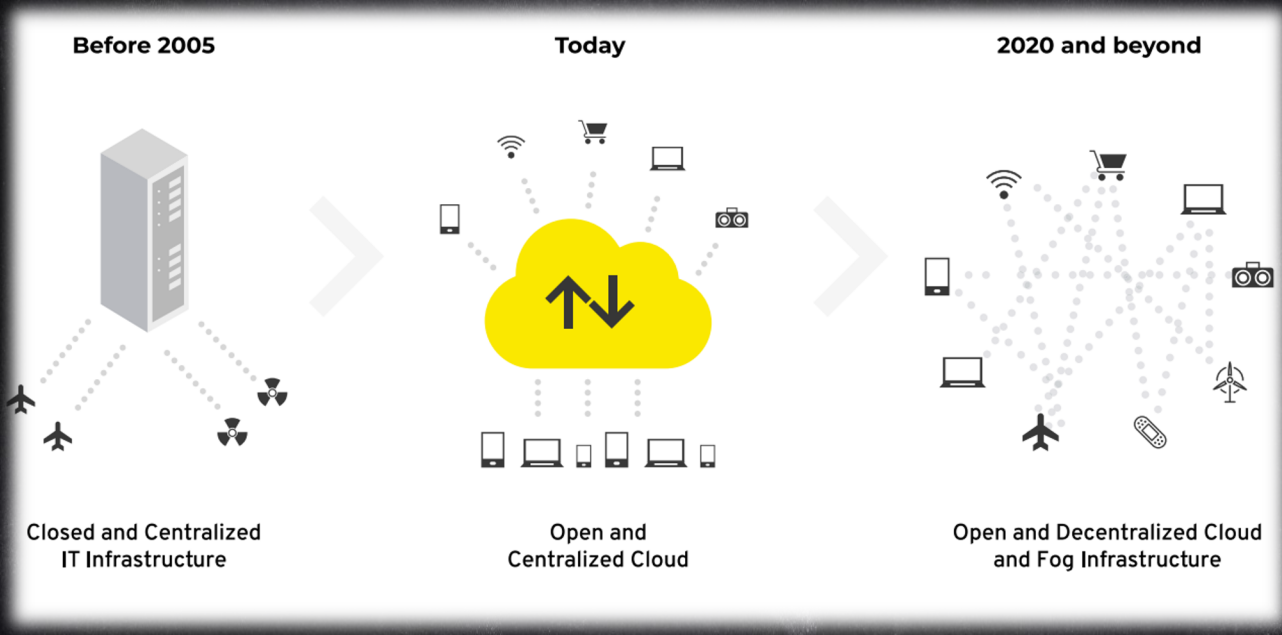




分布式系统案例

- 分布式文件系统（HDFS、NFS、Ceph）
- 分布式数据库（Mongodb、Cassandra、ElasticSearch）
- 分布式处理框架（Hadoop、MPI、Spark、Storm）
- 分布式调度器（YARN、Mesos、Slurm）
- 分布式操作系统（Kubernetes、OpenStack、OpenShift）

分布式系统架构



软件体系结构 | 系统体系结构



体系结构样式

➢ 基本思想

“样式”包含以下几个方面：

- （可替换性）组件具有定义良好的接口；
- 组件之间互联的方式；
- 组件之间交换的数据；
- 组件以及连接器是如何协同配置的；

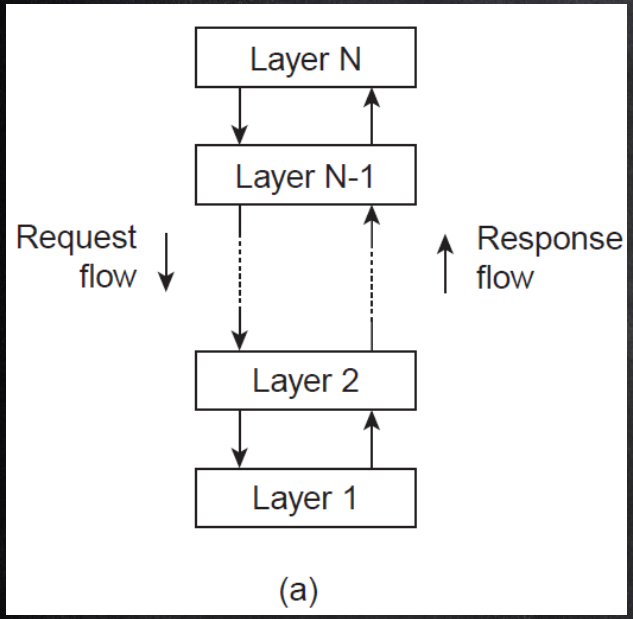
➢ 连接器

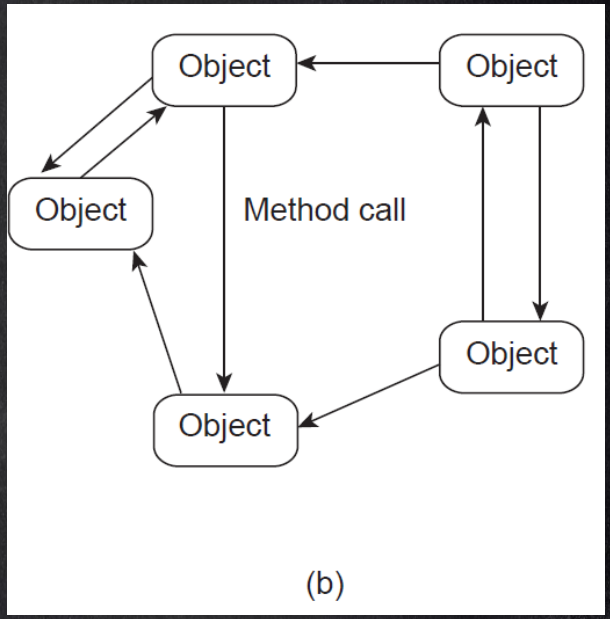
- 连接器提供了一种机制，在组件之间传递通信、使组件相互协调和协作。

➢ 分类

- 根据组件和连接器的使用，可以出现不同的配置，从而划分不同的体系结构，包括：分层体系结构；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。

体系结构风格



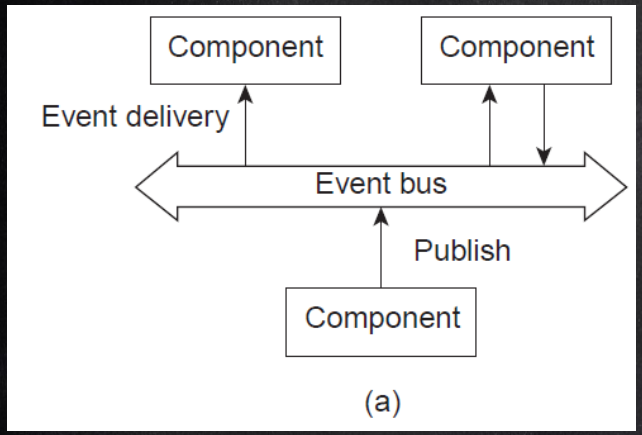


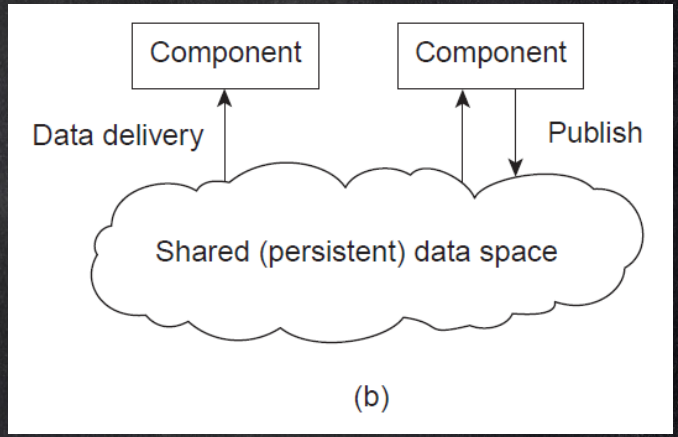
分层结构主要用于客户端-服务器模型

面向对象结构主要用于分布式对象系统

体系结构风格

- 进一步将过程的空间（“匿名化”）和时间（“异步化”）解耦，形成如下两种结构；





基于事件的结构：如Pub/Sub结构

共享数据空间结构

系统体系结构

- 系统体系结构（System Architecture）

  确定软件组件、这些组件的交互以及它们的位置就是软件体系结构的一个实例，称为系统体系结构。

  - 集中式体系结构

    整个系统包含一个控制中心，协同系统的运行；

  - 非集中式组织结构

    系统没有一个整体的控制中心，各个节点独立自主运行；

  - 混合组织结构

    系统中既包含集中式结构也包含了非集中式结构；



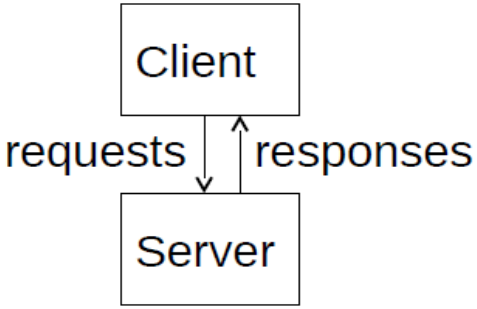
客户端-服务器模型（Client-Server）

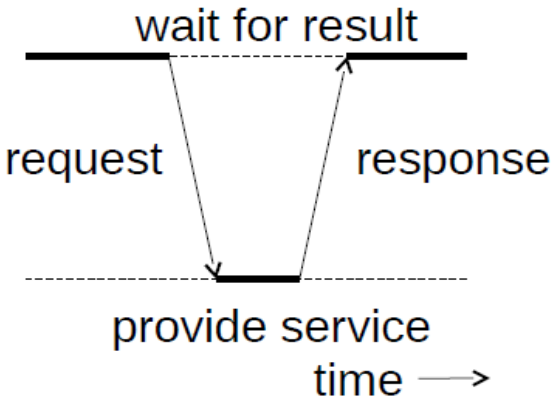
- 客户-服务器模型的特征：
  - **Server进程**

    实现特定服务的进程；
  - **Client进程**

    通过往服务器发送请求来请求服务、然后等待服务器回复的进程；
  - 客户端和服务器进程可以分布在不同的机器上；
  - 客户端在使用服务时采用请求-响应模式；

客户端-服务器模型（Client-Server）



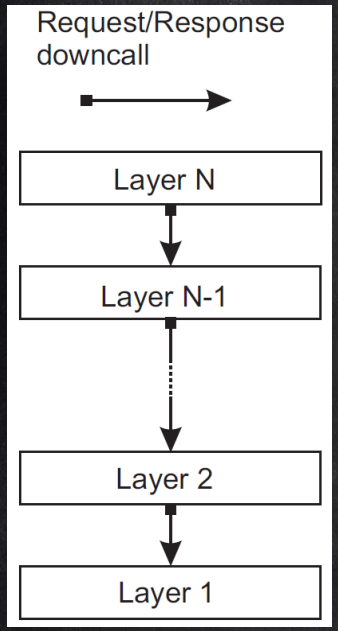


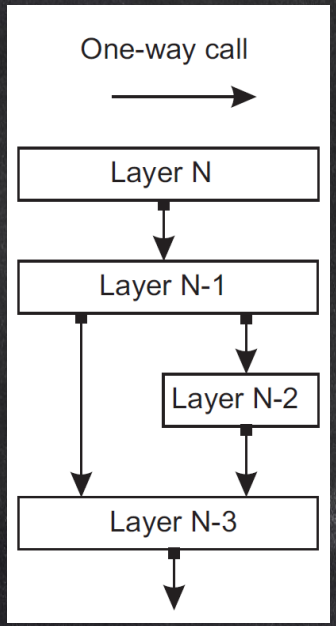
主流的客户端-服务器之间的通信协议包括：

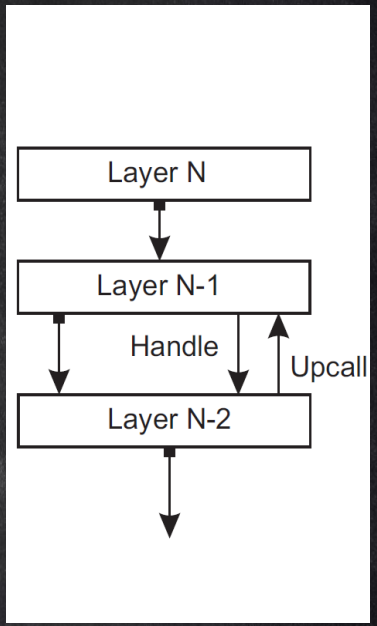
- HTTP、HTTPS： REST API (GET、POST…)；
- AJAX：异步请求；
- RPC： XMLRPC、SOAP、gRPC，web服务通过API调用；

应用分层

应用分层类别：



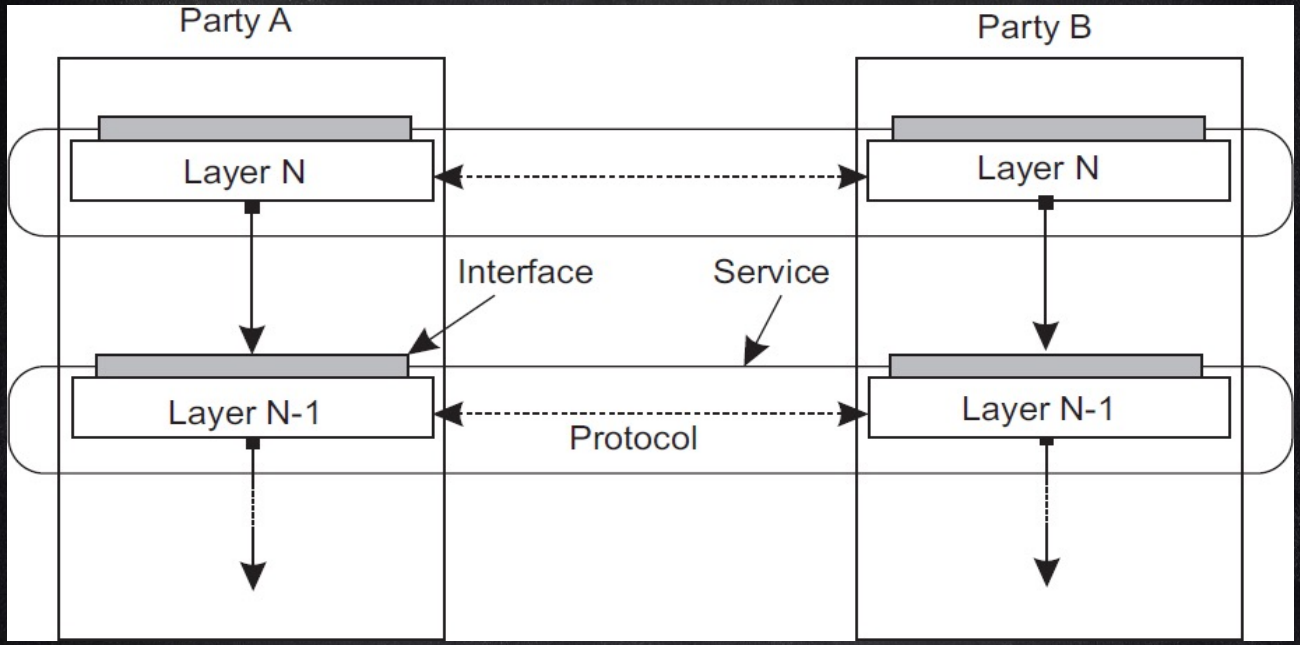


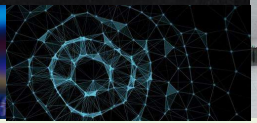


什么样的场景？

分层通信协议

- 协议、服务和接口




两节点通信

Server

```
1 from socket  import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept()  # returns new socket and addr. client
4 while True:                # forever
5   data = conn.recv(1024)   # receive data from client
6   if not data: break       # stop if client stopped
7   conn.send(str(data)+"*") # return sent data plus an "*"
8 conn.close()               # close the connection
```

Client

```
1 from socket  import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # connect to server (block until accepted)
4 s.send('Hello, world')  # send some data
5 data = s.recv(1024)     # receive the response
6 print data               # print the result
7 s.close()               # close the connection
```



应用分层

- 传统的三层视图：
  - 用户接口层

    用户接口层包含系统与用户直接交互的单元；例如：显示管理等；
  - 应用处理层

    包含应用的主要函数，但是，不与具体的数据绑定；
  - 数据层

    数据层管理应用使用的实际数据。

分层结构是IT系统常见的组织结构形式，使用到了传统的数据库以及相关的应用程序

应用分层

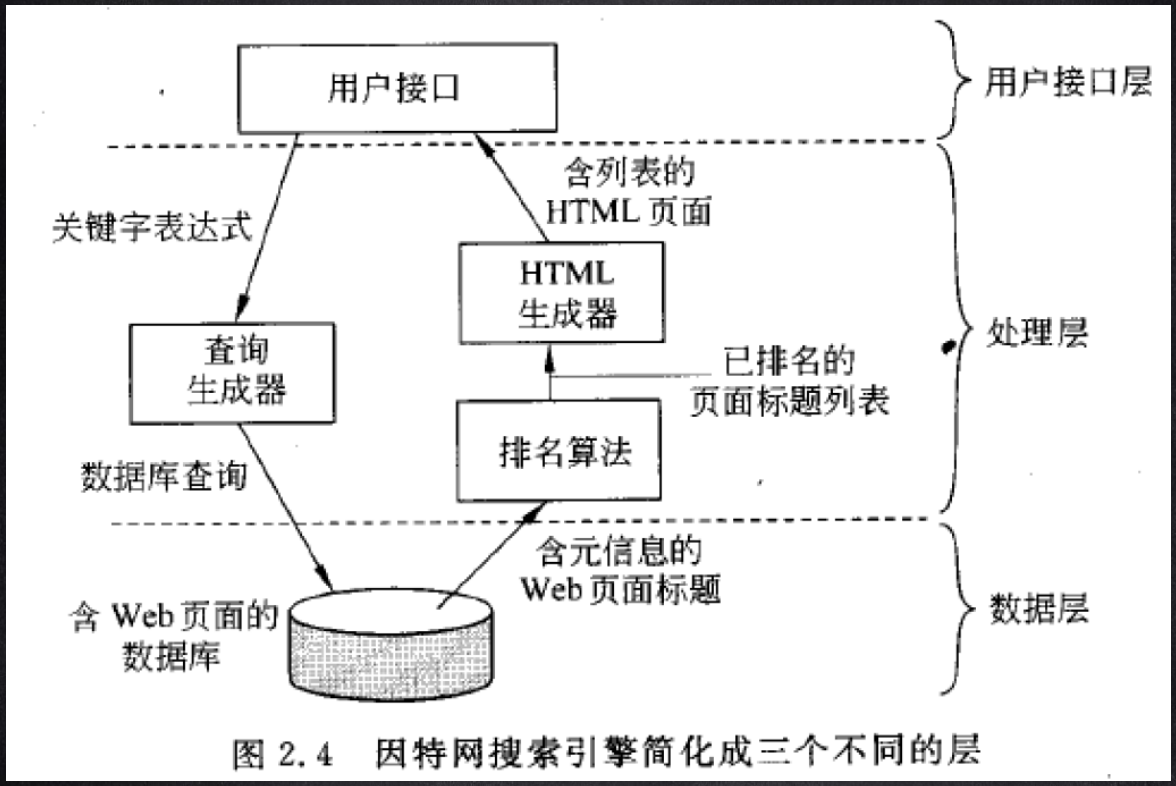
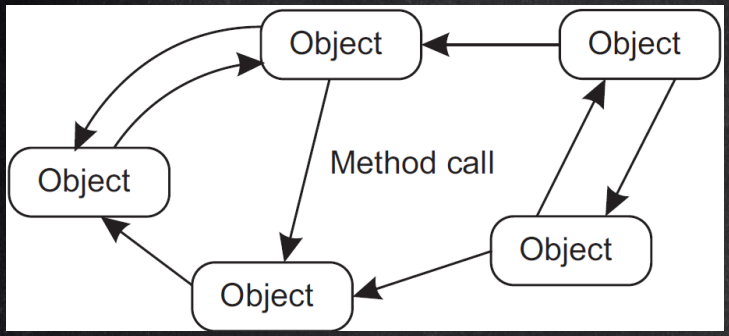


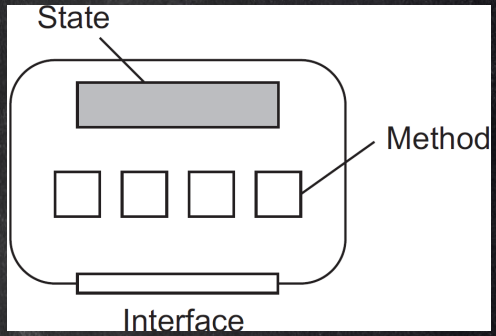
图 2.4 因特网搜索引擎简化成三个不同的层

基于对象的体系结构风格

- **本质**

  组件是由通过过程调用相互连接的对象构成。对象可以放置在不同的机器上，调用可以跨网络执行。





- **封装**

  对象封装了数据，并且提供面向数据的方法而没有展示内部实现；

RESTful架构

- **REST (Representational State Transfer)** 风格
  将分布式系统看做资源的集合， 由组件独立管理。这些资源可以由应用程序添加、删除、检索和修改。

- 资源可以通过命名机制标识；
- 所有的服务提供相同的接口；
- 从一个服务发出或者传入的消息是完全自描述的；
- 执行完操作后，执行组件不再记录调用者的信息；

Basic operations

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource by transferring a new state



样例：Amazon’s S3

- 原理
  - 对象（objects）放在桶（Buckets）中；
  - 桶是不能嵌套的；
  - 对于桶中对象的操作需要以下标识符：

    http://BucketName.s3.amazonaws.com/ObjectName

- 典型的操作：所有的对象是通过HTTP协议传输的；
  - 创建桶或者对象: 在一个 URI中执行 PUT操作；
  - 列出对象：在buket上执行GET操作；
  - 读对象：通过一个完整的URI执行GET；

关于接口

➢ 问题

- 人们都喜欢用RESTful的方法，因为接口简单；
- 代价：与资源相关的操作的参数空间设计复杂；

Amazon S3 SOAP interface

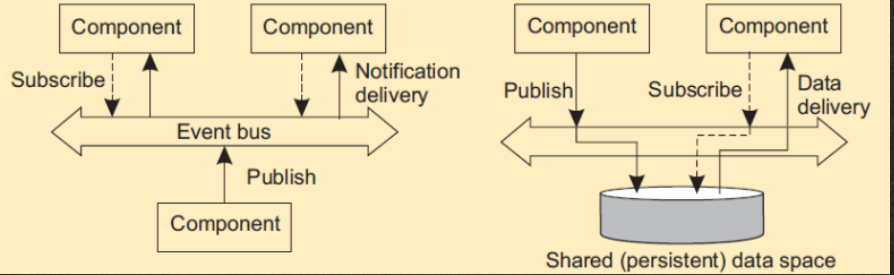
Bucket operations	Object operations
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus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

协同（Coordination）

- 时态和引用耦合

	Temporally coupled	Temporally decoupled
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

- 基于时间和共享数据空间



多层架构

- **单层架构：**
  简单客户端 / 大型主机配置；
- **两层架构：**
  客户端/单服务器配置；
- **三层架构：**
  每一层分布式在不同的机器上；

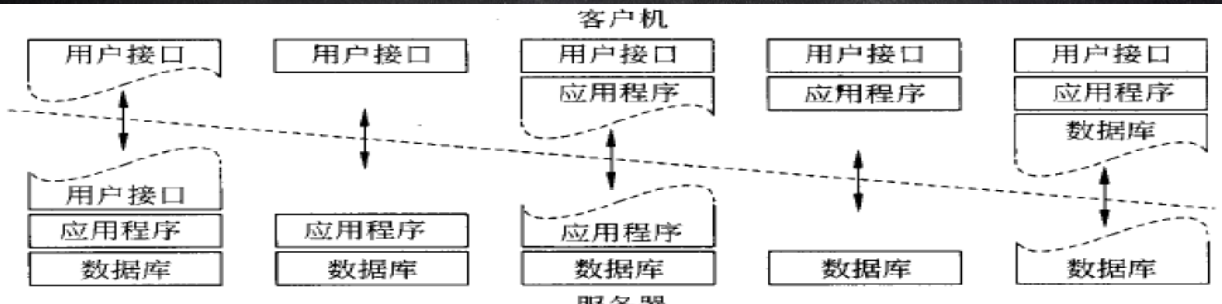
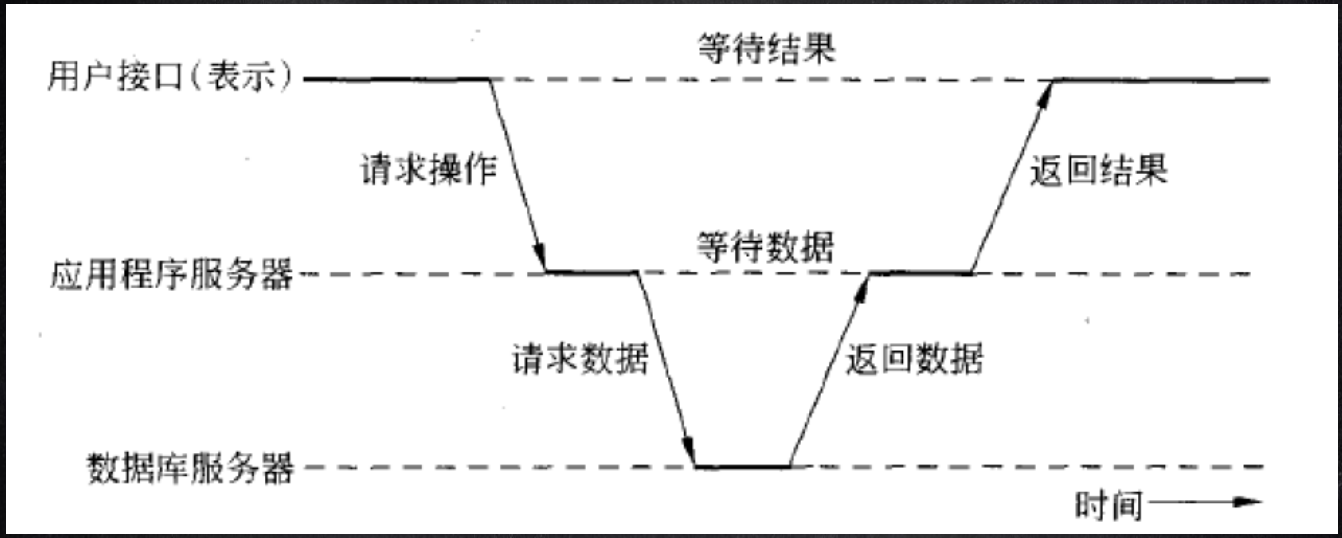


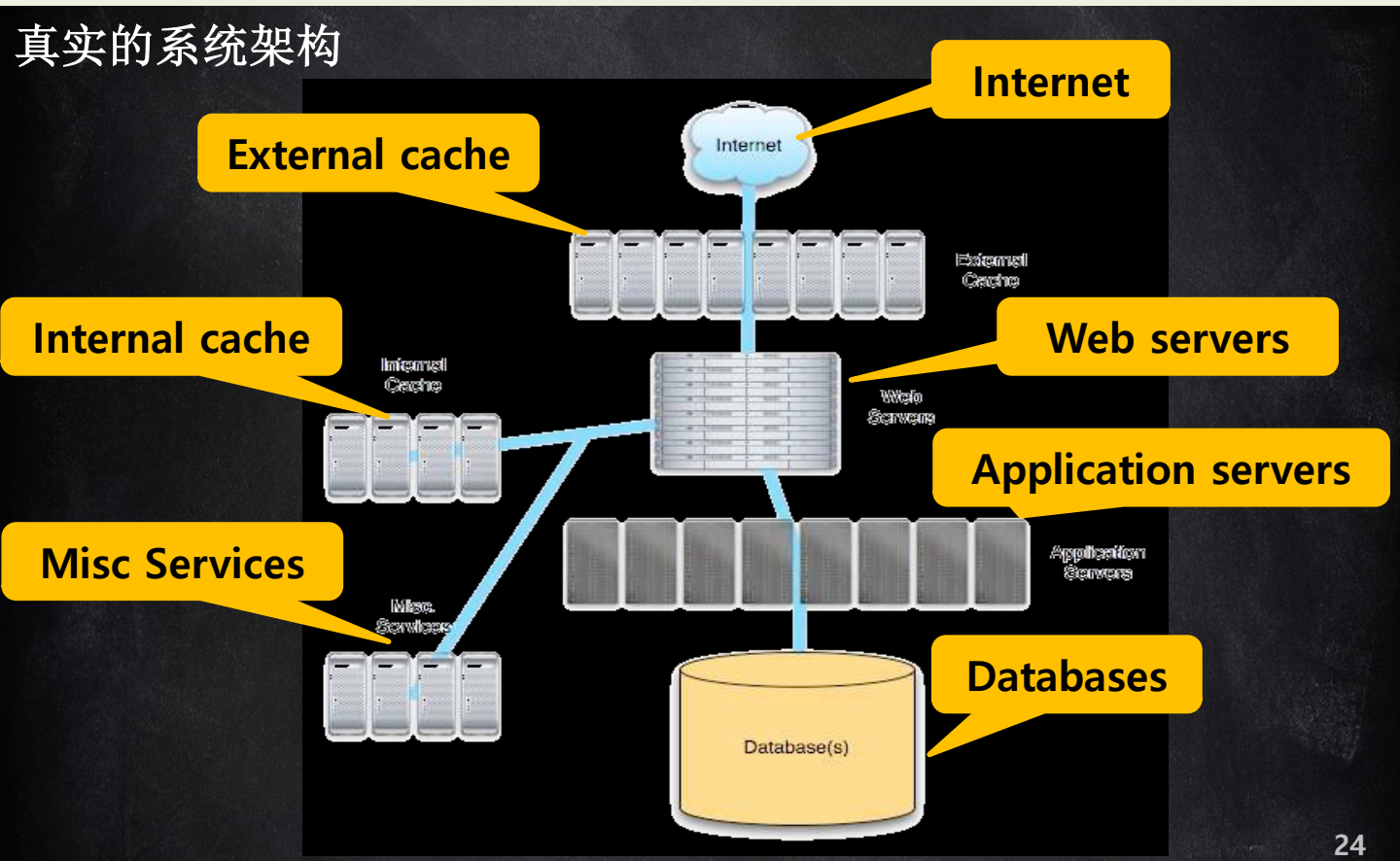
图 2.5 各种客户-服务器组织结构(a)～(e)

多层架构

- **三层架构：**
  有些服务端同时成为客户端，演变出三层甚至多层架构；



真实的系统架构


Internet
External cache
Internet
External Cache
Internal cache
Web servers
Internal Cache
Web Servers
Application servers
Misc Services
Application Servers
Misc. Services
Databases
Database(s)



非集中式体系结构

- **垂直分布（Vertical Distribution）：**
  系统逻辑分层，不同层次分布式不同的机器上；

- **水平分布（Horizontal Distribution）：**
  客户或者服务器在物理上分成逻辑上相等的几个部分，每个部分相对独立，且分布在不同的机器上；

- **点对点系统（Peer-to-Peer System）：**
  水平分布，构成点对点的系统的进程完全相同（既是客户端又是服务器、无中心化系统）；



非集中化体系结构

最近几年，P2P系统特别是区块链系统取得了高速的发展，P2P系统主要分为以下几种类型：

- **结构化的P2P系统**

  P2P系统中的节点按照特定的分布式数据结构组织；

- **无结构化的P2P系统**

  节点的邻居是随机选择的；

- **混合P2P系统**

  系统中的某些节点具有良好的组织方式；

在所有的P2P场景中，我们都会讨论“覆盖网络”：数据通过节点之间的连接传输，与应用层的广播类似。

结构化的点对点系统

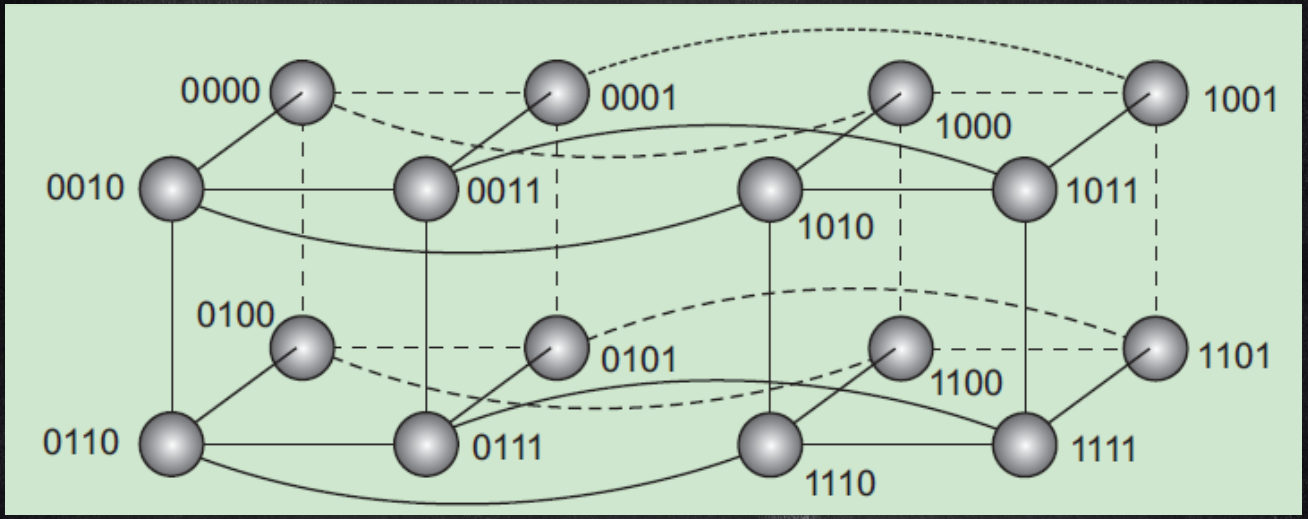
➢ 基本思想

- 将节点组织在一个特定结构的覆盖网络中，如环形结构，让这些节点根据自己的ID提供相应的职能；
- 语义无关的索引
- 数据与唯一的健值对应
- 最佳实践：利用Hash函数

  key(数据项) = hash(数据项的值)
- P2P系统用于存储（key, value）对

结构化的点对点系统

- 简单的例子：超立方体结构

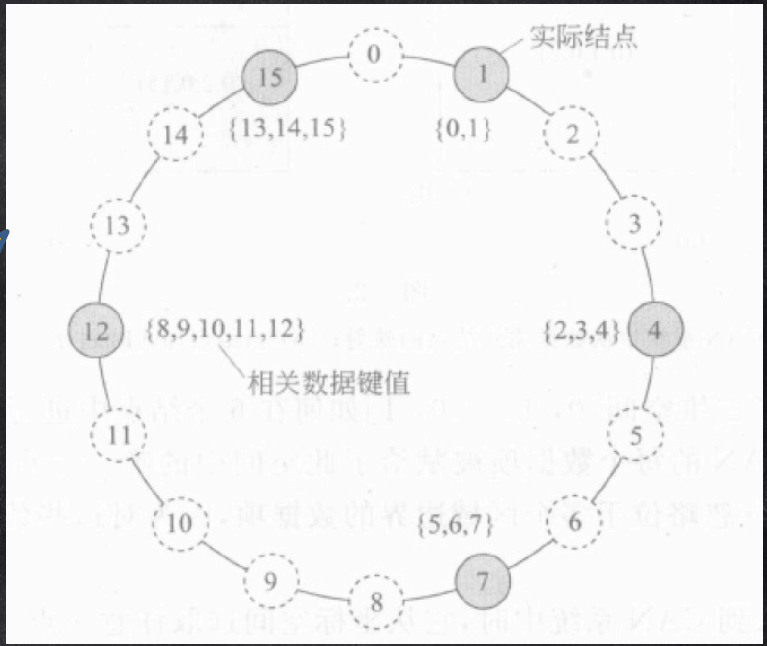


查找数据键值为 k 的数据 d，意味着将请求路由到节点标识符为 k 的节点

Chord结构

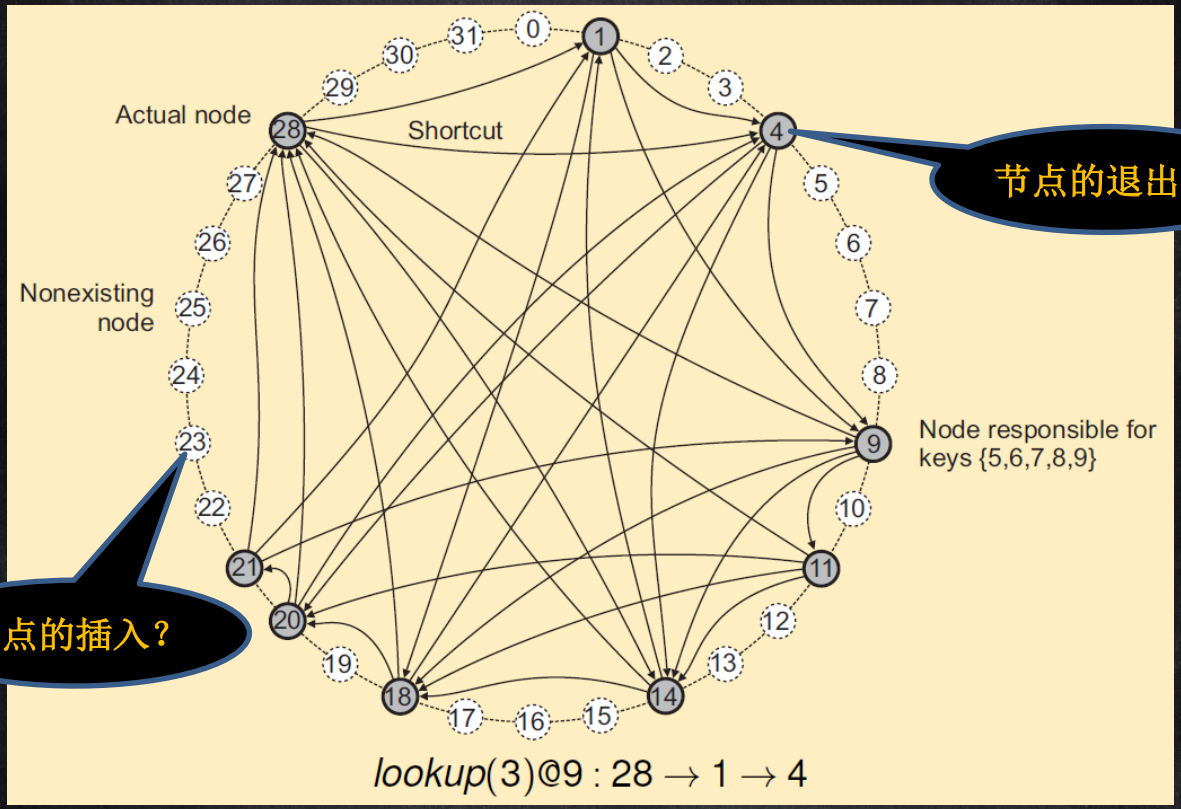
- 分布式哈希表（DHT）；
- 节点和数据项的key值在同一个随机空间中生成（160位标识符）；
- 数据项健值与节点标识符一一对应；

1. 键值为 k 的数据项保存在满足 id ≥ k的最小节点上 succ(k);
2. 环可以通过节点之间的捷径（shortcuts）进行扩展；



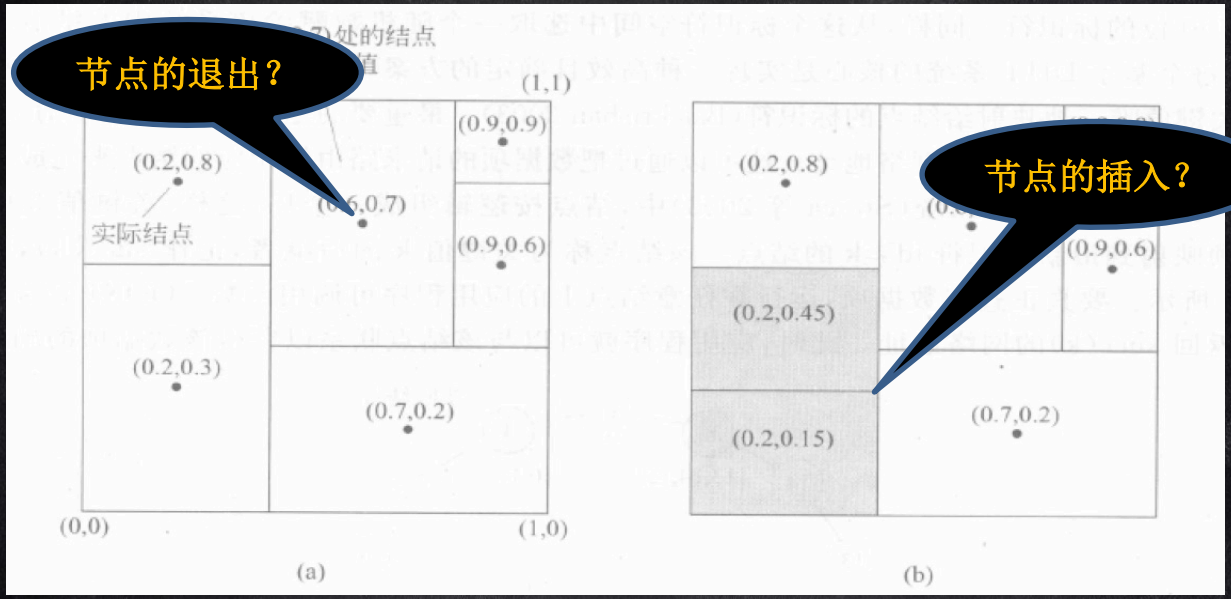
Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Chord结构


Actual node
Shortcut
Nonexisting node
Node responsible for keys {5,6,7,8,9}
节点的退出？
节点的插入？
lookup(3)@9 : 28 → 1 → 4

CAN(Content Addressable Network)上下文可编址结构

- **核心思想：**
  将节点组织成 d 维空间，令每一个节点在一个特定的区域负责特定的任务。 当有新的节点加入时，将区域进一步划分。





非结构的点对点系统

- 本质
  - 每个节点维持一个动态随机的邻接表；
  - 目标是构建一个随机图；
  - 部分视图

    从当前节点集合中随机选择出来 c 个“活”节点；
  - 边<u,v>存在的概率为P(<u,v>)；
  - 节点 P 周期性地从部分视图中选择节点 Q， 并且交换信息；



非结构的点对点系统

- 部分视图构造

主动进程推送到对等节点 c/2 + 1 个表项；被动进程推送同样数量的表项给对等节点。

主动线程的动作（周期性重复）：
从当前的部分视图中选择一个对等体 P
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;
}
从当前视图和 P 的缓冲区构建一个新的部分视图；
增加新的部分视图中每个项目的年龄

(a)

被动线程的动作：
从任意进程 Q 中接收缓冲
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;
}
从当前视图和 P 的缓冲区构建一个新的部分视图；
增加新的部分视图中每个项目的年龄

(b)

非结构的点对点系统

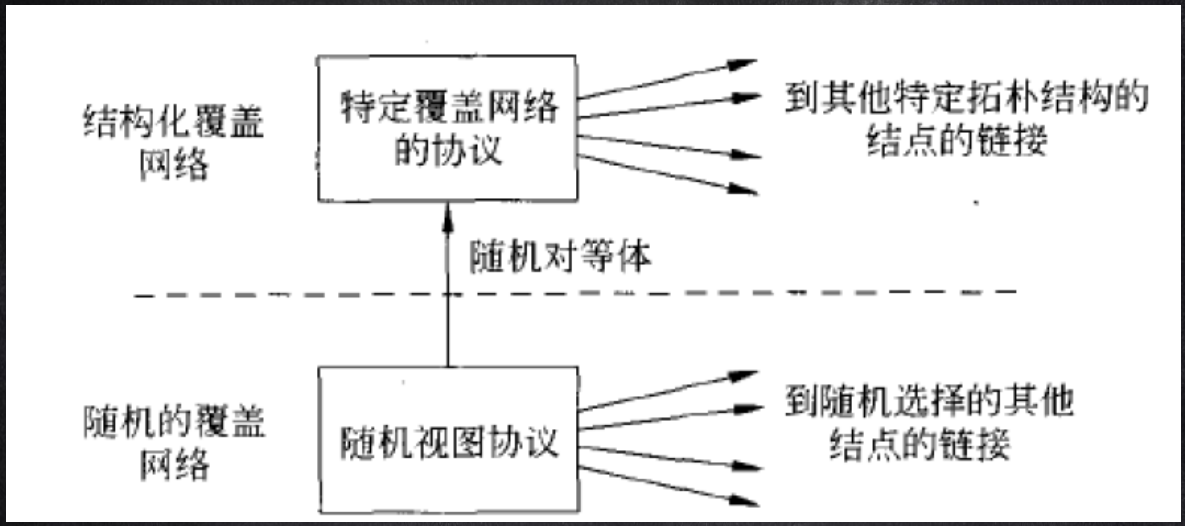
- P2P系统数据搜索方式
  - 泛洪方式

    请求发出节点 u 会其向其所有邻居节点发出数据搜索请求。如果之前已收到节点请求，则请求会被忽略。否则，节点会在本查找数据项。 请求会迭代传递下去。（有TTL限制、通信代价高）
  - 随机游走

    请求发出节点 u 从其邻居节点中随机选择一个节点，然后进行本地搜索。如果没有完成，继续从其邻居节点中选择一个随机节点向下进行。 （需要一种停止机制）

覆盖网络拓扑管理

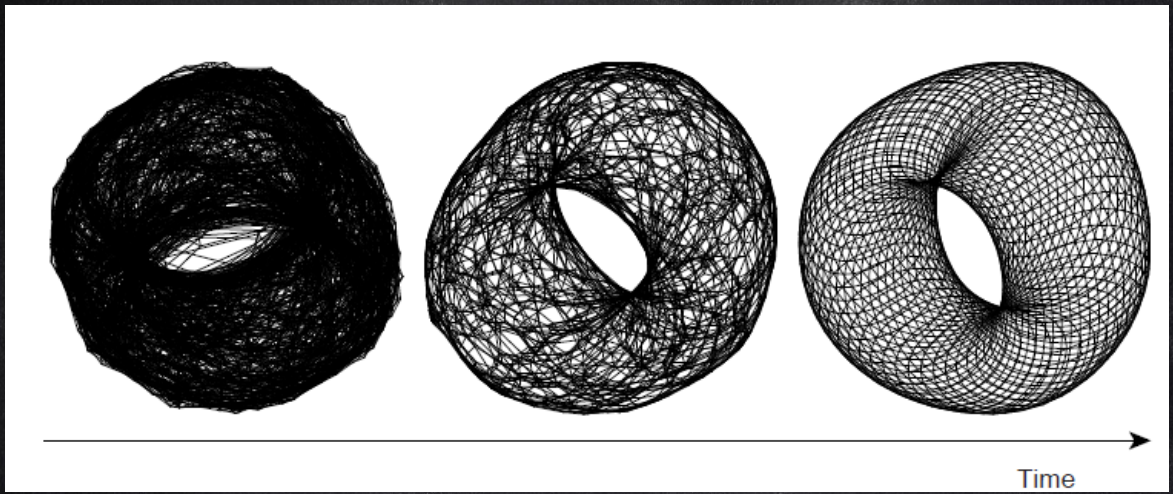
- **核心思想 （两层方法）：**
  - 在最底层维护随机的部分视图；
  - 在较高层的部分视图上进行表项选择；

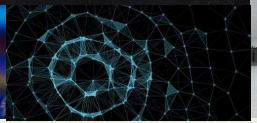


覆盖网络拓扑管理

- **样例：**

给定一个大小为 N x N 的逻辑网格， 每个节点维护一个含有 c 个最近邻节点的列表。 节点(a1, a2) 和 (b1, b2)之间的距离定义为：d1 + d2，且di=min(N-|ai- bi|, |ai-bi|)。



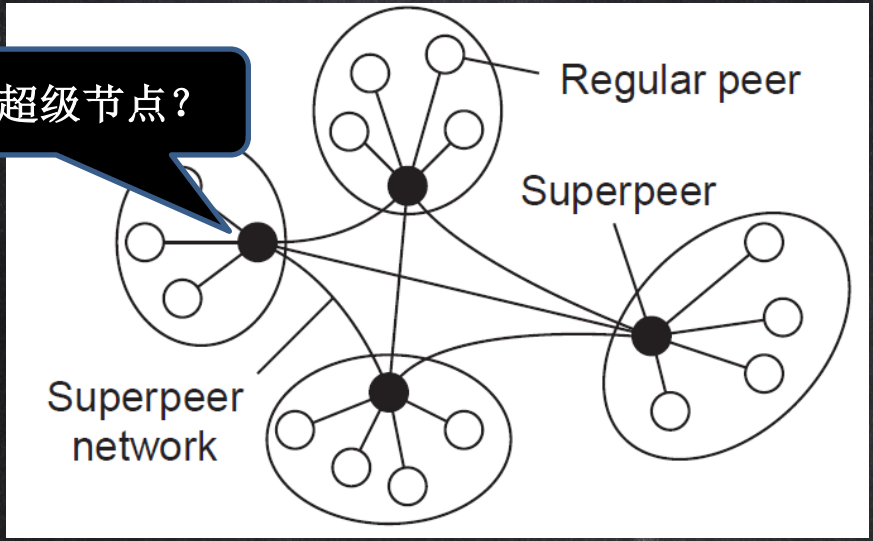

超级对等节点

- **核心思想：**
  - 非结构化的点对点系统中，数据项的定位非常困难；
  - 打破点对点网络的对称性；
  - 设置代理程序；

- **超级对等节点**
  - 维护一个索引或充当一个代理程序的节点；
  - 在非结构化的P2P进行搜索时，索引服务器会提高性能；
  - 通过代理程序可以更加有效地决定在哪里放置数据；

超级对等节点

如何选择超级节点？



- 超级对等节点的作用：

  维护索引； 监控网络的状态；能够建立连接

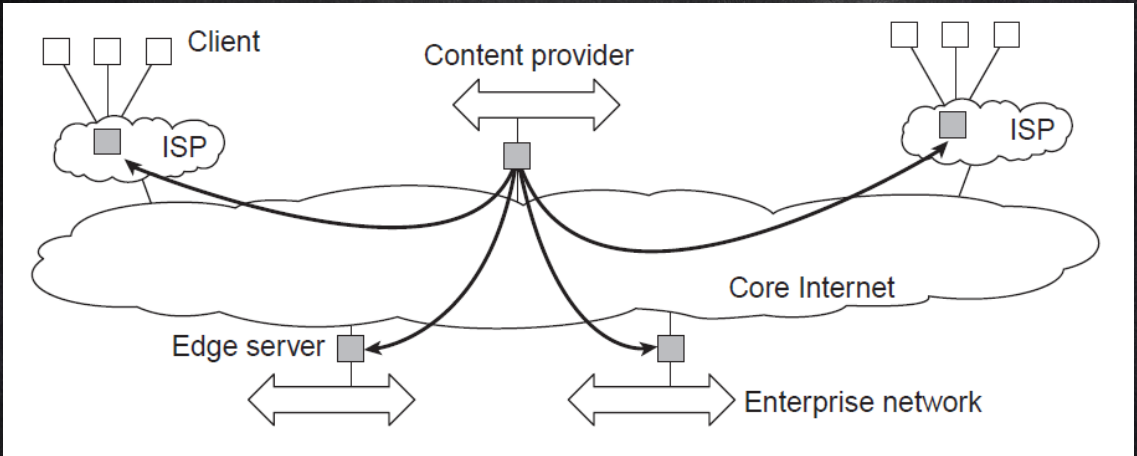


Skype’s 中 A 连接 B 的原理

- **A 和 B 均在公共网络上**
  - A 和 B 之间建立TCP连接用于控制报文的传输；
  - A 和 B之间实际通过 UDP 协议在商定的端口之间通信；
- **A 在防火墙内，而B在公网上**
  - A 和 超级对等节点 S 之间建立TCP连接用于控制报文的传输；
  - S 与 B节点建立TCP连接用于中继报文的传输；
  - A 和 B之间直接通过UDP协议传输数据；
- **A 和 B 都在防火墙后面**
  - A 与在线的超级对等节点 S 通过TCP连接；
  - S 通过 TCP协议与 B 连接 ；
  - 对于实际的呼叫，另外一个超级对等节点 R 作为中继节点：A 和 B 均会和 R 建立连接；
  - 所有的音频数据通过 R 以及两个 TCP连接转发；

混合架构之边界服务器

- **观察发现**
  在很多场景中，客户端-服务器架构是和P2P架构整合在一起的；
- **边界服务器系统：**
  服务器放置在网络的“边界”，常用与 CDN。

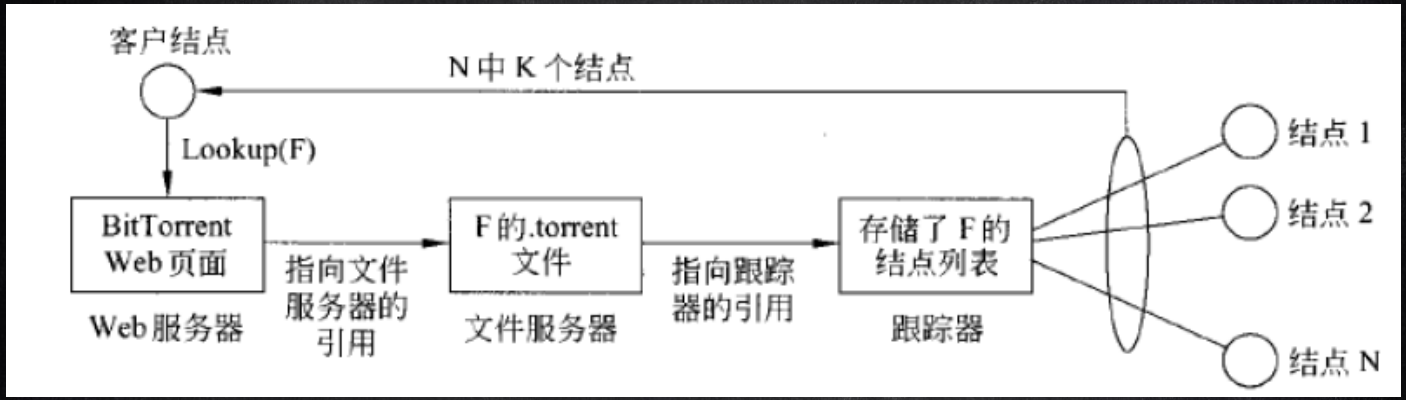


边界服务器主要的目的是进行过滤和编码转换后提供内容服务

混合架构之协作分布式系统

➢ Bittorrent

混合结构主要部署在协作分布式系统中。 Bittorrent融合了P2P下载协议和客户端-服务器架构。



➢ 基本原理

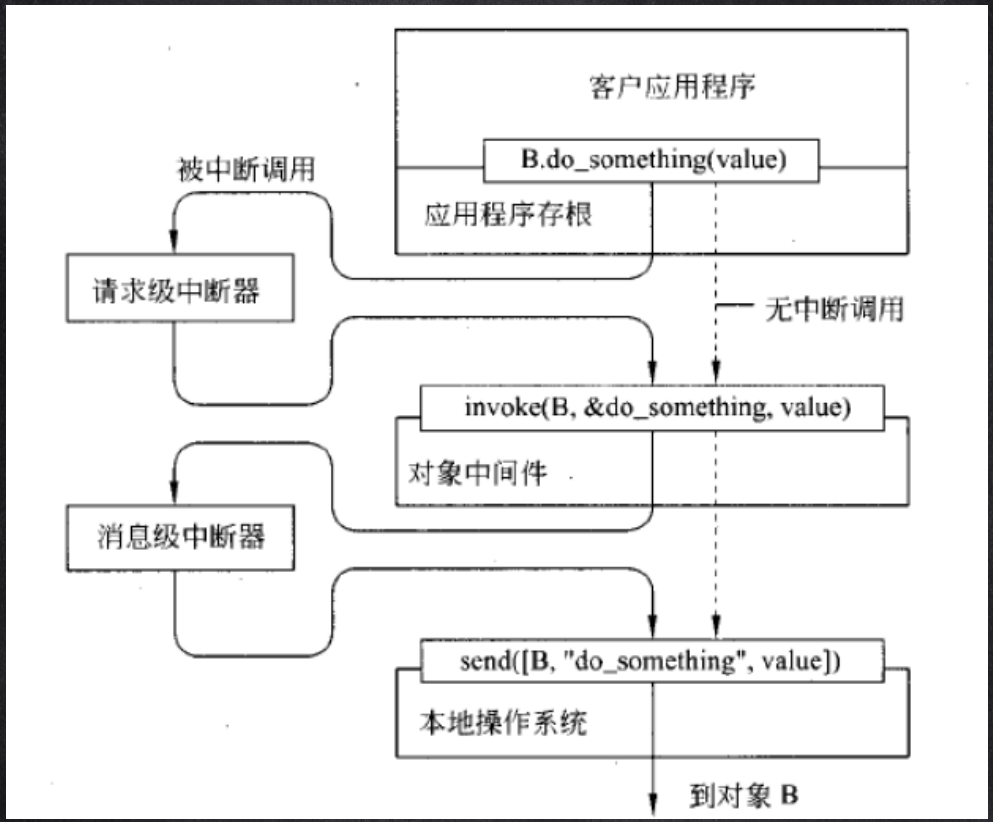
一旦节点识别出到哪里下载文件，它就会加入到一个下载机器的集合，这些机器并行地下载文件块，同时把这些数据块分发给其它节点



体系结构与中间件

- **问题**
  - 分布式系统的透明性；
  - 体系结构非最优，需要中间件提供动态自适应能力；
  - 中间件的自适应能力（策略和机制分离）；
- **中断器（Interceptors）**
  - 中断器是一种软件结构；
  - 中断器应具有通用性；
  - 基于对象的分布式系统的中断；

中断器


客户应用程序
B.do_something(value)
应用程序存根
被中断调用
请求级中断器
无中断调用
invoke(B, &do_something, value)
对象中间件
消息级中断器
send([B, "do_something", value])
本地操作系统
到对象 B



自适应中间件

- **要点分离**

把要实现功能的部分与非功能部分如可靠性等分开如面向方面的编程，annotation （注解）；

- **计算反射**

让程序在运行时刻检查自己的行为，如果有必要，调整其行为，Java、Pyth

软件的自适应、自演化能力的重要性

- **基于组件的设计**

通过组件的不同组合来支持自适应。 系统可以在设计时静态配置，或者是在运行时动态配置。



分布式系统的自我管理

➢ 观察发现

在需要完成自适应功能时， 系统架构和软件架构之间的界线逐渐模糊。

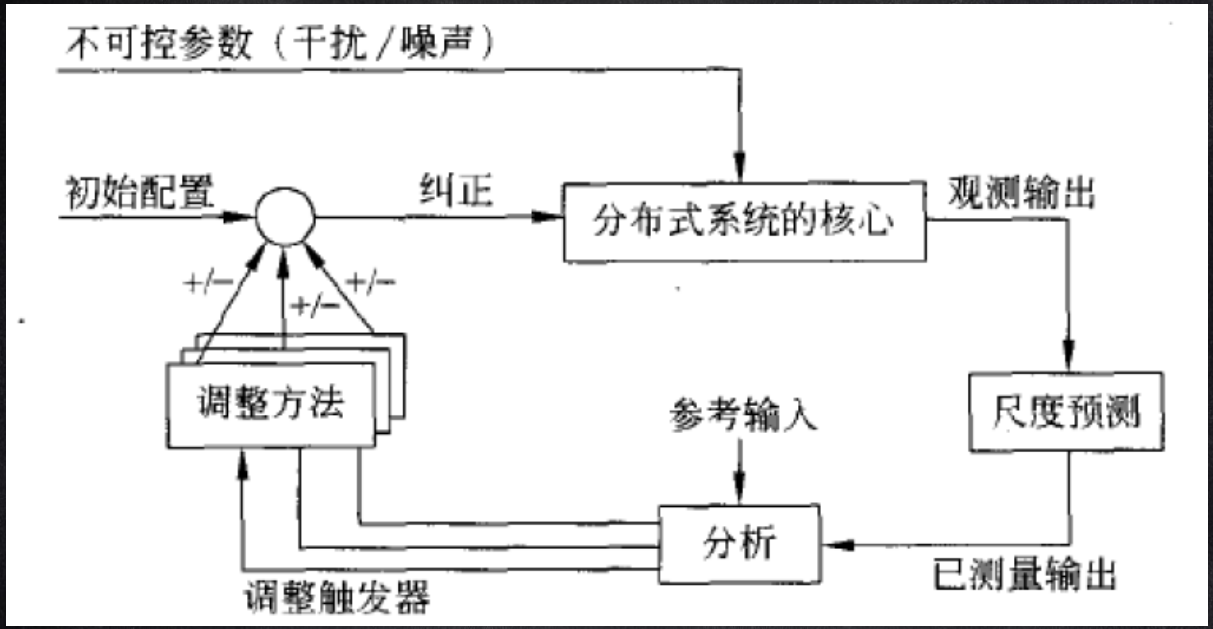
➢ 自主计算

- 自我管理；
- **自我恢复；**
- **自我配置；**
- **自我优化；**
- 自我*

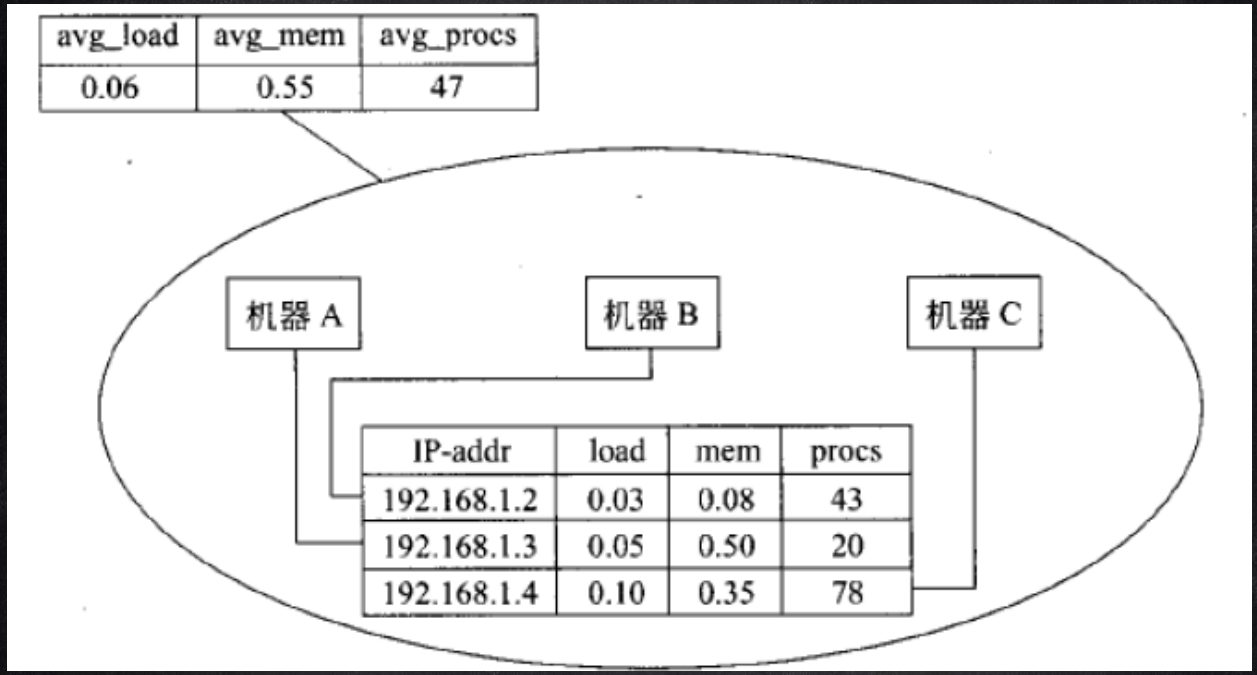
反馈控制模型

- **反馈控制循环**

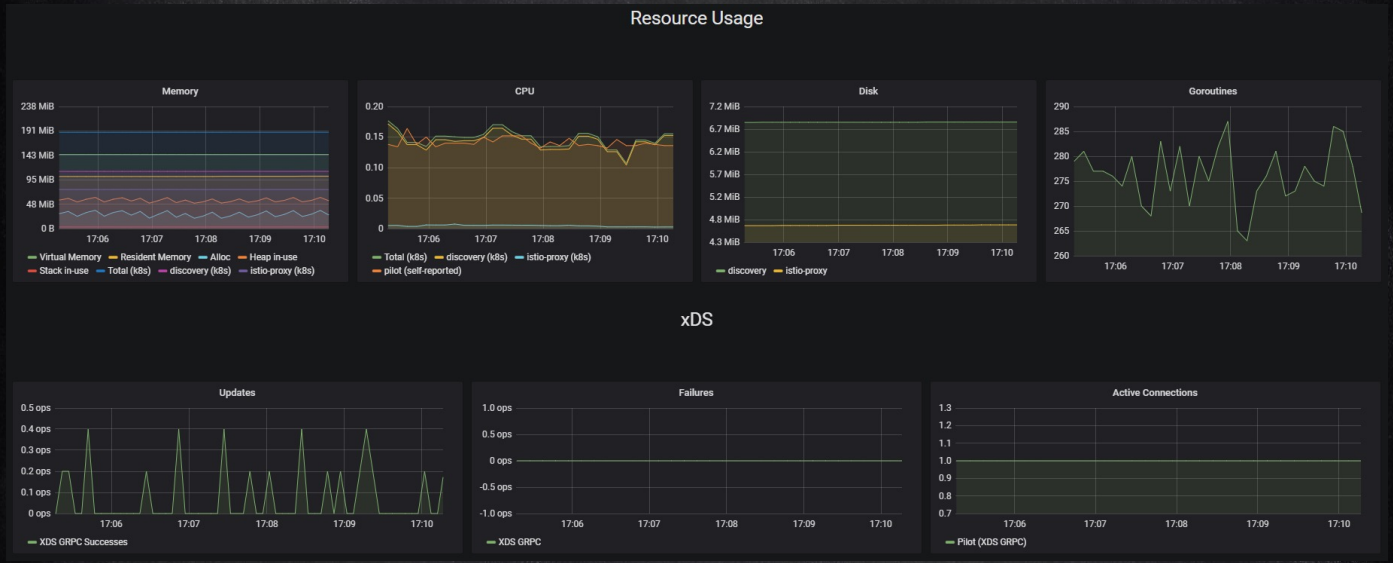
  自主计算往往通过反馈控制循环实现自适应性

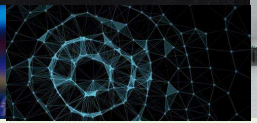


示例：Astrolabe监视系统



Prometheus + Grafana




总结

- 分布式软件架构和系统体系结构；
- 体系结构风格；
- 客户端-服务器模型
- 点对点系统；
- 分布式系统的自我管理；