1.1 分布式系统简介
分布式系统定义：一个其硬件或软件组件分布在连网的计算机上，组件之间通过消息进行通信或动作协调的系统。
显著的特征：并发、缺乏全局时钟、故障独立性
构造分布式系统的主要动力之一是来源于资源共享
1.2 分布式系统的例子
分布式系统包含了近年来许多最重要的技术发展
分布式系统底层技术是现代计算的核心
本地化系统 <-----> 全球数百万台节点
数据为中心 <-----> 计算密集型任务
传感器系统 <-----> 强大计算单元系统
嵌入式系统 <-----> 复杂交互式系统
300ms内完成
整个任务对分布式系统设计是一个巨大的挑战
1.2.1 Web搜索
Google是计算历史上最大和最复杂的分布式系统
一个底层的物理设施
位于全世界的多个数据中心联网组成
一个分布式文件系统
支持超大文件，并根据Google的应用分别进行了深度优化
一个相关的结构化分布式存储系统
超大数据集的快速访问
一个锁服务
提供诸如分布式加锁和协定等分布式系统功能
一个编程模型
支持超大并行分布式计算管理
1.2.2 大型多人在线游戏
客户-服务器模式
集中式服务器上维护了游戏世界状态的单个拷贝
服务器由多个计算机集群组成
分布式体系结构
游戏世界（或称宇宙）被划分到大量服务器上
服务器可在地理上分散部署
P2P模式
每个参与者贡献（存储和处理）资源来容纳游戏
1.2.3 金融交易
金融行业一直处于分布式 系统的最前沿
特别是在实时访问大范围的信息源方面
金融行业的需求：可靠和及时地传递事件给大量对此信息可能感兴趣的客户。
1.3 分布式系统的趋势
1.3.2 移动和无处不在的计算
分布式系统正在经历巨大的变化：
- 泛在互联网技术
- 移动和无处不在的计算
- 分布式多媒体系统
- 分布式计算作为公共设施

1. 互联网也是一个超大的分布式系统
服务集是开放的，要解决异构、安全、可伸缩等挑战
2. 设备小型化、无线网络发展
设备的便携性和方便的网络连接能力
移动计算：用户在移动或访问某个非常规环境时执行计算任务的性能
移动性为分布式系统带来了一系列的挑战
变化的连接、短时中断等
无处不在的计算是指对用户的物理环境中存在的多个小型、便宜的计算设备的利用。
小型计算设备在不引人注意的日常物品中普及
各处的计算机在相互通信时才变得有用
自发互操作需求
用户访问某地时，通过移动电话获得股票信息、导航等服务
访问某单位时，将数码相机的照片直接发送到会议室投影机或打印机
将设备与合适的本地设备相关联---服务发现
3. 分布式多媒体系统
分布式系统中支持多媒体服务的需求
离散型媒体的传输、存储和展示
图片、文本消息
连续类型媒体的传输、存储和展示
视频、音频
连续类型媒体具有时间维度，需要在媒体类型元素之间保持实时关系
网络播放—分布式多媒体技术应用
编码和加密格式的支持
服务质量保障
资源管理策略
适配策略
4. 云计算
分布式资源<---->其他公用设施
资源通过合适的服务提供者提供，被最终用户有效的租赁而不是拥有
云计算（Cloud Computing）用来刻画计算作为公共设施
通常云实在集群计算机上，从而提供必要的伸缩和性能
集群计算机（Cluster Computer）
互连的计算机集合，它们紧密协作提供单一的、集成的高性能计算能力
集群服务器的总目的是提供高性能计算能力、大容量存储等。
网格计算，偏重于支持科学计算，通常被看作云计算这种通用模型的先驱

1.4 资源共享
大家已经习惯了资源共享带来的好处，以至于很容忽视它们的重要性
资源共享的模式随其工作范围与用户工作的密切程度不同而不同
Web搜索引擎为全世界用户提供工具
计算机支持协同工作（CSCW）在封闭小组内资源共享
服务—表示计算机系统中管理相关资源并提供功能给用户和应用的一个单独的部分
服务将资源访问限制为一组定义良好的操作
分布式系统的资源是物理地封装在计算机内
其他计算机只能通过通信才能访问
服务器----是指在连网的计算机上的一个运行的程序（进程），这个程序接收来自其他计算机上正在运行的程序的请求，执行一个服务并适当地做出响应。
发出请求的进程称为客户
整个方案称为客户—服务器计算
注意：客户和服务器指的是进程而不是运行客户或服务器的计算机
1.5 挑战
1.5.1 异构性
互联网使得用户能在大量异构计算机和网络上访问服务和运行应用程序
网络；
计算机硬件；
操作系统；
编程语言；
软件实现。
中间件---是指一个软件层，它提供了一个编程抽象，同时屏蔽了底层网络、硬件、操作系统和编程语言的异构性。
公共对象请求代理（Common Object Request Broker, CORBA）
JAVA远程调用方法（Remote Method Invocation，RMI）
大多数中间件基于互联网协议实现，这些协议屏蔽了底层网络的差异，但是中间件要解决操作系统和硬件的不同。
1.5.1 异构性
移动代码---是指从能从一台计算机发送到另一台计算机，并在目的计算机上运行的代码（例如：Java applet）
虚拟机方法提供了一种代码可以在任何计算机上运行的方法：某种语言的编译器生成一台虚拟机代码而不是某种硬件代码，虚拟机通过解释的方式来执行它。
1.5.2 开放性
开放性决定系统能否以不同的方式被扩展和重新实现
分布式系统的开放性主要取决于新的资源共享服务能被增加和供多种客户程序使用的程度
开放的分布式系统的特点：
发布系统的关键接口开放系统的特征
基于一致的通信机制和发布接口访问共享资源
能用异构硬件和软件构造
1.5.3 安全性
分布式系统中使用和维护的众多信息资源，具有很高的内在价值，安全性相当重要。
机密性、完整性、可用性
1.5.4 可伸缩性
如果资源数量和用户数量激增，系统仍能保持其有效性，那么该系统就称为可伸缩的。
面临的挑战：
控制物理资源的开销、控制性能损失、防止软件资源用尽、避免性能瓶颈
理想状态下，系统规模增加时系统和应用程序应该不需要随之改变。
1.5.5 故障处理
故障不可避免
分布式系统故障一般来说是部分的
有些组件正常而有些组件出现故障
故障处理技术：
检测故障 掩盖故障 容错 故障恢复 冗余
1.5.6 并发性
分布式系统中，服务和应用均提供可被客户共享的资源
几个客户同时试图访问一个共享资源
代表共享资源的任何一个对象必须负责确保在并发环境中操作正确，不仅适用于服务器，也适用于应用中的对象
1.5.7 透明性
最重要的两个透明性：访问透明性和位置透明性
如果一个分布式系统仅可以利用FTP访问远程计算机上的文件，那么它就是缺乏透明性的例子
Web中的URL具有位置透明性
1.5.8 服务质量
服务质量：系统的主要非功能特性
影响用户体验的服务质量包括：
可靠性 安全性 性能 适应性
1.6 实例研究：万维网
Word Wide Web 是一个不断发展的系统，用于发布和访问互联网上的资源和服务
WWW是一个开放的系统，可以被扩展

基于自由发布和广泛实现的通信标准和文档/内容标准
可以在其上发布和共享资源
基本体系架构
超文本标记语言（HTML）
统一的资源定位器（URL）
标准交互规则（HTTP）
Figure 1.7
Web servers and web browsers
1.6 实例研究：万维网
HTML
超文本标记语言用于指定组成Web页面内容的文本和图像，以及显示给用户的布局方式
文本编辑器手写生成、也可以由所见即所得的编辑器编辑
URL 统一资源定位器
作用是识别资源
Web体系文档中使用URI（统一资源标识符）
浏览器检查URL以便访问相应的资源
URL有两个顶层的部分组成
模式：模式特定的标识符
第一个部分模式，声明了URL的类型，例如：ftp，http，mailto等
HTTP URL的主要工作
识别出哪一个服务器维护资源
识别出该服务器上的哪些资源是被请求的
http://服务器名[:端口][/路径名][?查询][#片段]
方括号中的项是可选的
服务器名表示成为一个域名系统（Domain Name System, DNS）
HTTP超文本传输协议：定义了浏览器和其他类型的客户与服务器的交互方式
主要特征
请求--应答交互
内容类型
一次请求一个资源
简单的访问控制
Web服务器上运行的为客户生成内容的程序通常称为公共网关接口（Common Gateway Interface, CGI）程序
除浏览器之外的程序也可以是Web的客户
通过程序访问Web资源
HTML不适合于程序之间的互操作
XML是一种标准的、结构化的、特定于应用的格式表示数据方式。
REST体系结构

2.1 简介
不同类型的分布式系统具有一些重要的基本特征，也有公共的设计问题
分布式系统的困难和威胁：
使用模式的多样性、系统环境的多样性、内部问题、外部威胁
物理模型：描述系统的最显式的方法，系统的硬件组成
体系结构模型：描述系统的计算元素执行的计算和通讯任务
基础模型：采用抽象的观点描述分布式系统的某个方面，交互模型、故障模型和安全模型
2.2 物理模型
物理模型是从计算机和所用网络技术的特定细节中抽象出来的分布式系统底层硬件元素的表示
基线物理模型：联网计算机上的硬件和软件组件仅通过消息传递进行通信和协调动作的系统，一组可以扩展的计算机节点，这些节点通过计算机网络相互连接进行所需的消息传递
**早期的分布式系统**
20世纪70年代晚期到80年代早期
局域网互联的10～100个节点组成
与互联网连接并支持少量的服务
单个系统大部分是同构的，开放性不是主要问题
服务数量有限
**互联网规模的分布式系统**
20世纪90年代开始
Google搜索引擎1996年第一次发布
利用互联网，基础设施变成全球化
异构性问题很突出
操作系统、网络、计算机体系结构、语言、开发者开放标准和相关中间件技术重要性不断增加
当代的分布式系统
上述系统节点通常是台式机，因此相对静态的、分立的、自治的
分布式系统的趋势
泛在互联网技术
移动和无处不在的计算
分布式多媒体系统
分布式计算作为公共设施
**系统的分布式系统**
超大规模分布式系统（Ultra Large Scale, ULS）
复杂系统
包含一系列的子系统
每个子系统本身也是系统
一起完成一个或多个特定任务
洪水预测管理系统
传感器网络

环境参数
集群计算机模拟系统
2.3 体系结构模型
一个系统的体系结构是用独立指定的组件以及这些组件之间的关系来表示的结构
整体目标是确保结构能够满足现在和将来可能的需求
主要设计目标：可靠性、可管理性、适应性、性价比
2.3 体系结构模型
2.3.1. 体系结构元素
支撑现在分布式系统的核心基本元素
2.3.2. 体系结构模式
分布式系统解决方案中单独或组合使用的体系结构模式
2.3.3. 相关中间件解决方案
2.3.1 体系结构元素
四个关键问题
通信的实体是什么?
它们如何保持通信，使用什么通信范型?
扮演什么角色，承担什么责任?
如何被映射到物理分布式基础设施上?
通信实体：从系统角度来看，分布式系统中通信的实体通常是进程
注意:
一些原始环境中（如：传感器网络），操作系统可能不支持进程抽象，因此这些系统中通信的实体是结点
大多数分布式系统中，用线程补充进程。
从编程观点来看，面向问题有更多的抽象：
对象：在分布式系统中使用面向对象方法；对象通过接口访问，用一个相关的接口定义语言（IDL）提供定义在一个对象上的方法的规约
组件：与对象类似，通过接口访问；与对象的区别是，显式的给出其他组件/接口的假设
Web服务：基于行为封装和通过接口访问
利用Web标准表示和发现服务
通信范型—分布式系统中实体如何通信
进程间通信 、远程调用、间接通信
**进程间通信**：是指用于分布式系统进程之间通信的相对底层的支持，包括消息传递原语、直接访问由互联网协议提供的API和对多播通信的支持。
**远程调用**：是分布式系统中最常见的通信范型，覆盖一系列分布式系统中通信实体之间基于双向交换的技术，包括调用远程操作、过程或方法。
请求—应答协议：构建在底层消息传递服务至上，用于支持客户—服务器计算。
通常涉及一对消息的交换，消息从客户到服务器，接着从服务器返回客户。
远程过程调用（Remote Procedure Call，RPC）
远程计算机上进程中的过程调用与本地调用一致
远程方法调用（Remote Method Invocation, RMI）
与远程过程调用类似，但仅应用于分布式对象环境
底层细节对用户隐藏
上述技术的共同点：
通信代表发送者和接收者之间的双向关系
发送者显式地把消息/调用送往相关的接收者
接收者通常了解发送者的标识
大多数情况下，双方必须同时存在
**间接通信**：通过第三个实体，允许在发送者和接收者之间的深度解耦合
发送者不需要知道正在发送给谁（空间解耦合）
发送者和接收者不需要同时存在（时间解耦合）
组通信：组通信涉及消息传递给若干接收者，是支持一对多通信的多方通信泛型
发布—订阅系统：大量生产者（或发布者）为大量的消费者（订阅者）发布他们感兴趣的信息项，关键特征：中间服务—用于确保由生产者生成的信息被路由到需要这个信息的消费者
消息队列：提供点对点服务。生产者发送消息到指定队列，消费者能从队列中接收消息，或者被通知队列有消息到达
元组空间：进程可以把任意的结构化数据项（元组）放到一个持久元组空间，其他进程可以指定感兴趣的模式，从而可以在元组空间读取或者删除元组。
分布式共享内存（Distributed Shared Memory，DSM）：系统提供一种抽象，用于支持在不同共享物理内存的进程之间共享数据
角色和责任
进程相互交互完成一个有用的活动
进程在系统中扮演给定的角色
客户---服务器
客户---服务器模式直接、简单，但是伸缩性差，集中化的提供服务和管理，受到计算机处理资源和网络带宽等资源条件限制大
对等体系结构
不区分客户和服务器或运行它们的计算机
所有参与进程运行相同的程序并且在相互之间提供相同的接口集合
目的是利用大量参与的计算机的资源来完成某个给定的任务或活动
Eg. BitTorrent
放置：对象或者服务这样的实体如何映射到底层物理分布基础设施

物理分布基础设施由大量机器组成，机器通过任意复杂的网络互联
决定了分布式系统的特性：性能、可靠性、安全性
如何放置客户和服务器需要仔细设计
- 考虑实体间的通信模式
- 机器的可靠性和负载
- 不同机器间通信质量
- 通用的指导方针

将服务映射到多个服务器
一个服务实现为一个单独主机上的几个服务器进程，必要时进行交互以便为客户进程提供服务
服务器可以将服务所基于的对象分区将这些分区分部到各个服务器上
服务器可以在几个主机上维护复制的对象集
缓存：用于存储最近使用的数据对象
当服务器接收到一个新对象时，将它存入缓存，必要时进行更新替换
当客户需要一个对象时，缓存服务首先检查缓存
如果有最新的拷贝，则直接提供缓存中的对象
如果没有可用对象，则去取一个最新的拷贝
移动代码：是一个运行的程序（包括代码和数据），它从一台计算机移动到网络上的另一台计算机，完成指定任务，最后返回结果。
移动代理：一个移动代理可能多次调用所访问地点的本地资源
移动代码与移动代理一样，对所访问的计算机而言是一个潜在的威胁

2.3.2 体系结构模式

体系结构模式构建在体系结构元素之上，提供组合的、重复出现的结构
分布式系统的体系结构模式是核心，同时也是一个很大的主题，本节介绍几个关键的体系结构模型
- 分层体系结构
- 层次化体系结构
- 瘦客户

分层体系结构
- 与抽象紧密相关
- 一个复杂的系统被分成若干层
- 每层利用下层提供的服务

平台：一个服务于分布式系统的应用的平台由最底层的硬件和软件层组成
Intel x86/Windows， Intel x86/Mac OS X，ARM/Linux等
中间件：定义为一个软件层，目的是屏蔽异构性
表示成一组计算机上的进程或对象，相互交互实现分布式应用的通信和资源共享支持
通过对抽象的支持提升应用程序通信活动的层次
层次化体系结构
- 层次化体系结构与分层体系结构互补的
- 分层将服务垂直组织成抽象层
- 层次化用于组织给定层的功能
- 主要与应用和服务的组织最相关，但也可以应用于所有层
- 考虑如下一个给定应用的功能分解：

表示逻辑、应用逻辑、数据逻辑

AJAX（Asynchronous Javascript And XML）
Web所使用的标准客户—服务器交互方式的扩展
满足了Javascript 程序与服务器细粒度通信的需求
**标准Web交互流程**
- 浏览器发送HTTP请求给服务器，请求制定的URL页面、图像或其他资源
- 服务器发送整个页面作为应答，由服务器上一个文件读取、或者由一个程序生成
- 浏览器根据其MIME类型相关显示方式呈现

标准交互方式的约束
- 一旦浏览器发送了一个新的Web页面请求，用户不能与页面交互，直到html内容被浏览器收到并呈现，这个时间间隔不确定
- 为了用来自服务器的额外数据修改当前页面的一小部分，也需要请求和显示整个新的页面
- 客户显示的页面内容不能被更新

JavaScript是跨平台、跨浏览器语言，能下载到浏览器中并执行
需要Web应用允许用户访问和更新大量共享数据集
- 数据集通常很大，还可能是动态的
- 用户一次更新数据量较少

AJAX提供了一套通信机制
运行在浏览器的前端组件能发送请求，并从运行在服务器上的后端组件接收结果
瘦客户
- 分布式计算的趋势是将复杂性从最终用户设备移向互联网服务
- 它使得能以很少的对客户设备的假设或需求，获得对复杂网络化服务的访问，这些服务可以通过云解决方案提供
- 指的是一个软件层，在执行一个应用程序或访问远程计算机上的服务时，由该软件层提供一个基于窗口的本地用户界面

瘦客户端的优势：极大的简化了本地设备
瘦客户端的缺点：交互频繁的图形活动中，受到网络和操作系统的影响大
虚拟网络计算（Virtual Network Computing, VNC）
- 为远程访问提供图形用户界面
- RealVNC 软件解决方案
- Adventiq硬件解决方案

其他经常出现的模式
代理（Proxy）
提供远程过程调用或远程方法调用的位置透明
Web服务中的业务代理（Brokerage）
复杂分布式基础设施中支持互操作性的体系结构
反射（Reflection）
Introspection （系统动态发现的特性）
Intercession（动态修改结构或行为的能力）
2.3.3 相关的中间件解决方案
中间件的任务是为分布式系统的开发提供一个高层的编程抽象，通过分层对底层基础设施中的异构性提供抽象，从而提升互操作性和可移植性
中间件的类别
远程过程调用包和组通信，之后出现了大量不同风格的中间件
中间件的限制
依靠中间件支持的开发，能大大简化分布式系统变成
但系统可依赖性的一些方面要求应用层面的支持
• Email发送，构建在TCP之上
Saltzer, Redd, Clarke的观点是：通信相关功能可以只依靠通信系统终点的应用的知识和帮助，即可完整、可靠的实现。
2.4 基础模型
系统模型各有不同，但具有一些基本特征
所有模型都由若干进程组成
进程之间通过在计算机网络上发送消息而相互通信
共享的设计需求
- 进程及网络的性能和可靠性
- 系统中资源安全
基础模型的目的是：
显式的表示有关我们正在建模的系统的假设
给定这些假设，就什么是可能的、什么是不可能的给出结论
2.4 基础模型
基本模型中提取的分布式系统能解决下列问题：
交互
- 计算在进程中发生，进程通过传递消息交互，并引发进程之间的通信和协调
- 交互模型必须反映通信所带来的延迟
故障
- 分布式系统中任一计算机出现故障或连接他们的网络出现故障，分布式系统的正确操作就会受到威胁
- 模型要对这些故障进行定义、分类、分析和容忍
安全
- 分布式系统的模块特性和开放性，使其暴露在外部代理和内部代理的攻击下
- 分析系统的威胁以及设计抵御这些威胁的系统
2.4.1 交互模型
分布式系统由多个以复杂方式进行交互的进程组成，例如：
- 多个服务器进程进行相互协作提供服务
- 对等进程能相互协作获得一个共同的目标
算法---采取一系列步骤以执行期望的计算
- 算法中的每一步都有严格的顺序
- 算法决定程序的行为和程序变量的状态
分布式算法---定义了组成系统的每个进程所采取的步骤
- 包括进程之间的消息传递，以便协调每个进程所采取的步骤
- 每个进程的执行速率和进程之间消息传递的时限通常不能预测，因此描述分布式算法的所有状态也十分困难，要处理所涉及的一个或多个进程的故障或消息传递的故障
进程交互完成了分布式系统中所有的活动
每个进程有自己的状态，该状态由进程能访问和更新的数据集组成
属于每个进程的状态是私有的
影响进程交互的两个重要因素：
- 通信性能经常是一个限制特性
- 不可能维护一个全局时间概念
通信信道的性能
通信信道在分布式系统中可用许多方法实现
计算机网络上的流或简单消息传递
计算机网络上的通信有下列性能特征：
延迟（Latency）
从一个进程开始发送消息到另一个进程开始接收消息之间的间隔时间称为延迟
带宽（Bandwidth）
给定时间内网络能传递的信息总量
抖动（Jitter）
传递一系列消息所花费的时间的变化值
计算机时钟和时序事件
分布式系统中每台计算机有自己的内部时钟
不同计算机的时钟会提供不同的值
计算机时钟和绝对时间之间有偏移，每台计算机的偏移率互不相同
时钟漂移率
指的是计算机时钟偏离绝对参考时钟的比率
即使分布式系统中所有计算机时钟在初始情况下都设置成相同时间，它们的时钟最后也会相差巨大，除非进行校正
时钟的校正方法

- GPS，以大约1μs的精度接收时间读数
- 具有精确时间源的计算机可以发送时序消息给网络中其他计算机

交互模型的两个变体

**同步分布式系统**

进程执行每一步的时间有一个上限和下限

通过通道传递的每个消息在一个已知的时间范围内接收到

每个进程有一个本地时钟，与实际时间的偏移率在一个已知的范围内

**异步分布式系统**——对下列因素没有限制的系统

进程执行速度

消息传递延迟

时钟漂移率

对异步分布式系统有效的任何解决方案对于同步系统同样有效

Pepperland协定

红师和蓝师在邻近两座山的山顶

事件排序

许多情况下我们需要知道一个进程中的一个事件（发送或接受一个消息）是发生在另一个进程的另一个事件之前、之后或同时。

例如：

X,Y,Z,A之间的邮件交换

如果X,Y,Z计算机上的时间可以同步，每个消息发送时可以携带本地时间戳，那么消息可以按照正常的顺序排列

如果一个分布式系统中时间不能精确同步，

Lamport提出了逻辑时间的模型，为在分布式系统中运行于不同计算机上的进程的事件提供顺序

使用逻辑时间不需要借助时钟就可以推断出消息的顺序

分布式系统中，进程和通信通道都可能出故障

故障模型定义了故障可能发生的方式，以便理解故障所产生的影响

遗漏故障 随机故障 时序故障

2.4.2 故障模型

遗漏故障：是指进程或者通信通道不能完成它应该做的动作

进程遗漏故障中最常见的是进程崩溃

崩溃检测方法依赖超时的使用

异步系统中超时只能表明进程没有响应

崩溃 执行速度慢 消息没有送达

故障—停止，如果能够确切检测到进程崩溃

同步系统中，确定消息送达后，超时可用于检测故障—停止行为

通信遗漏故障

通信原语send和receive

进程p将消息m插入到它外发消息缓冲区来执行sen

通信通道将m传输到q的接受消息缓冲区

进程q通过将m从接受消息缓冲区取走并完成传递来执行receive

2.4.2 故障模型

如果通信信道不能将消息从p传递到q，那么就产生了遗漏故障

接收端或中间网关缺乏缓冲区

网络传输错误 发送遗漏故障

接收遗漏故障 通道遗漏故障

2.4.2 故障模型

故障检测

Pepperland，同步系统，异步系统

面对通信故障时达成协定的不可能性

Ringo证明， Pepperland环境中，在通信不能保证的情况下，两师不能一致的决定做什么

2.4.2 故障模型

随机故障（拜占庭故障）

用于描述可能出现的最坏的故障，此时可能发生任何类型的错误

例如：一个进程可能在数据项中设置了错误的值，响应一个调用返回一个错误的值

进程的随机故障

是指进程随机地省略要做的处理步骤或执行一些不需要的处理步骤

不能通过查看进程是否应答调用来检测

通信信道随机故障

例如：消息内容被破坏或传递不存在的消息，也可能多次传递实际消息

通信软件可以识别这类故障并拒绝出错的消息

时序故障

时序故障适用于同步分布式系统

对进程执行时间、消息传递时间和时钟漂移率均有要求

异步分布式系统中，一个负载过重的服务器响应时间可能很长，但我们不能说它有时序故障

实时操作系统是以提供时序保证为目的而设计的

- 设计复杂
- 需要冗余硬件保障
- 大多数通用操作系统不能满足实时约束

时序与音频和视频通道的多媒体计算关系尤为密切

故障屏蔽

分布式系统中的每个组件通常是基于其他一组组件构造的

利用存在故障的组件构造可靠的服务是可能的

例如：有数据副本的多个服务器中一个服务器崩溃时能继续提供服务

一个服务通过隐藏故障或者转换为一个更能接受的故障类型来屏蔽故障
例如：通过校验可以屏蔽错误消息，将随机故障转化为遗漏故障
一对一通信的可靠性
可靠通信可从下列角度定义：
有效性：外发消息缓冲区中的任何消息最终能传递到接受消息缓冲区
完整性：接收到的消息与发送消息一致，没有消息被传递两次
2.4.3 安全模型
体系结构模型：进程，进程间交互
通过保证进程和用于进程交互的通道的安全以及保护所封装的对象免遭未经授权访问可实现分布式系统安全
保护对象
- 对象可由不同的客户按照不同的方式调用
- 访问权限指定了允许谁执行一个对象的操作
- 需要将每个调用和每个结果均与对应的授权方相关联
- 一个授权方成为一个主体（principal）
- 一个主体可以是一个用户或进程

服务器验证每个调用的主体身份，检查是否有足够的访问权限在所调用的某个对象上完成所请求的操作，如果没有就拒绝它们的请求。客户可以检查服务器的主体身份以确保结果来自所请求的服务器
保护进程和它们的交互
进程通过消息进行交互
消息易收到攻击（所使用的网络和通信服务是开放的）
服务器和对等进程暴露它们的接口
分布式系统经常在可能受到来地敌对用户的外部攻击的任务中使用和部署。
为了识别和抵御这些威胁，需要分析安全威胁的模型。
**敌人**
假定敌人能给任何进程发送任何消息，并能读取或复制一对进程之间的任何消息。
攻击可能来自合法连接到网络的计算机或以非授权的方式连接到网络的计算机
潜在敌人的威胁包括对进程的威胁和对通信信道的威胁
对进程的威胁
- 处理到达的请求的进程可以出接收来自其他进程的消息，但是未必能能确定发送方的身份
- 缺乏消息源的可靠的知识对服务器和客户的正确工作而言是一个威胁

对通信信道的威胁
- 对人在网络和网关上能复制、改变或插入消息
- 试图保存消息的拷贝并在以后重放这个消息
- 利用安全信道可以解除这些威胁

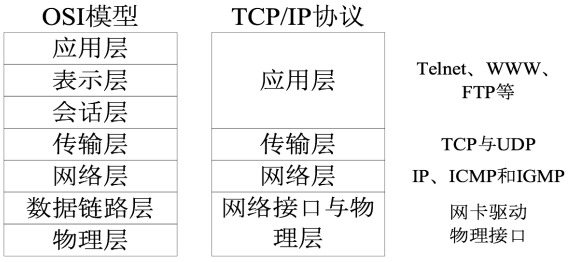
**解除安全威胁**
密码学和共享秘密
假设一对进程共享一个秘密
如果一对进程交换的消息包括证明发送方共享秘密的信息，那么接收方就能确认发送方是一对进程中的另一个进程
密码学：保证消息安全的科学
加密是将消息编码以隐藏其内容的过程
认证：证明由发送方提供的身份
基础是共享秘密和加密
在消息中包含加密部分，该部分中包含足够的消息内容以保证它的真实性
安全通道：是连接一对进程的通信通道
加密和认证用于构建安全通道，安全通道作为已有的通信服务层之上的服务层，具有以下特征
- 每个进程确切的知道其他正在执行的进程所代表的主体身份
- 安全通道确保在其上传送的数据的私密性和完整性
- 每个消息包括一个物理的或逻辑的时间戳以防消息被重放或重排序

其他可能的来自敌人的威胁
**拒绝服务**
通过超量地、无意义地调用服务或在网络上进行消息传送，干扰授权用户的活动，导致物理资源的过载
**移动代码**
如果进程接收和执行来自其他地方的程序代码，那么这些移动代码就会来带新的安全问题。
特洛伊木马，生成完全无害的事情但事实上包括了访问或修改资源的代码
安全模型的使用
安全技术和访问控制的使用会产生实质性的处理和管理开销
需要对系统网络环境、物理环境和人际环境等进行评估分析

3.1 简介
UDP的应用程序接口提供了消息传递（Message Passing）抽象
- 进程间通信的最简单形式
- 使得发送进程能够给一个接收进程传递一个消息
- 包含消息的独立的数据包称为数据报（datagram）

TCP的应用程序接口提供了进程对之间的双向流（two-way stream）抽象
相互通信的信息由没有消息边界的一连串数据项组成
双向流为生产者—消费者通信提供了构造成分
3.2 互联网协议的API
3.2.1 进程间通信的特征
由Send和Receive这两个消息通信操作来支持一对进程间进行的消息传递，均用目的地和消息定义
一个进程发送一个消息到目的地，在目的地的另一个进程接收消息
该活动设计发送进程到接收进程间的数据通信，涉及两个进程的同步
同步和异步通信
每个消息目的地与一个队列相关
- 发送进程将消息添加到远程队列中
- 接收进程从本地队列中移除消息

发送进程和接收进程间可以是同步也可以是异步的
同步（synchronous）通信
发送进程和接收进程在每个消息上同步
Send和Receive都是阻塞操作
异步（asynchronous）通信
Send操作是非阻塞的，只要消息复制到本地缓冲区，发送进程就可以继续其他处理
消息传递和发送进程并行
Receive操作包括阻塞和非阻塞两种形式
消息的目的地
互联网协议中解释了消息是如何发送到 <互联网地址，本地端口>
本地端口是计算机内部使用的消息目的地，用一个整数指定
一个端口只能有一个接收者（组播除外），但可以有多个发送者
进程也可以使用多个端口接收消息
任何知道端口号的进程都能向端口发送消息
如果客户使用一个固定的互联网地址访问一个服务，那么该服务必须总在该地址所代表的计算机上，以保持该服务的有效性；
可以通过客户程序使用名字服务来避免这个情况，以提供位置透明性；
但是不能实现迁移
迁移是指在系统运行时移动服务所在的位置
**可靠性**
有效性：如果一个点对点消息服务在丢失了“合理”数量的数据包后，仍能保证发送消息，那么该服务就称为可靠的；
相反，只丢失一个数据包，消息就不能保证发送，那么这个点对点消息服务仍是不可靠的。
完整性：到达的消息必须没有损坏，且没有重复
排序：有些应用要求消息按发送方的顺序发送
与发送方顺序不一致的消息发送会被这样的应用认为是失败的发送
网络的体系结构
网络采用分而治之的方法设计，将网络的功能划分为不同的模块，以分层的形式有机组合在一起。
每层实现不同的功能，其内部实现方法对外部其他层次来说透明，每层向上层提供服务，也可以使用下层提供的服务
网络体系结构即指网络的层次结构和每层所使用协议的集合
两类非常重要的体系结构：OSI与TCP/IP
OSI开放系统互联模型
OSI模型相关的协议已经很少使用，但模型本身非常通用
共有七层
TCP/IP协议族的体系结构
TCP/IP协议是Internet事实上的工业标准。
一共有四层
TCP/IP协议通信模型




数据的封装与传递过程
一些基本概念
IP地址：IP地址是Internet中主机的标识
Internet中的主机要与别的机器通信必须具有一个IP地址，一个IP地址为32位（IPV4），或者128位（IPV6）
每个数据包都必须携带目的IP地址和源IP地址，路由器依靠此信息为数据包选择路由
特殊的IP地址：广播地址、多播地址
表示形式：常用点分形式，如202.38.64.10，最后都会转换为一个32位的整数。

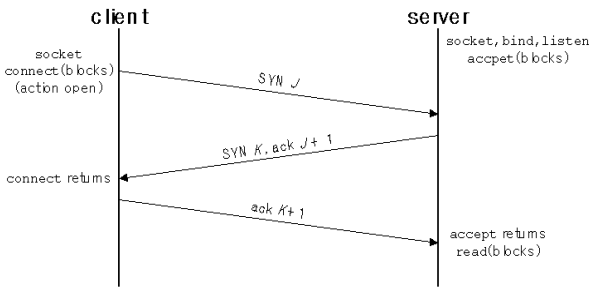
为了区分一台主机接收到的数据包应该递交给哪个进程来进行处理，使用端口号
TCP端口号与UDP端口号独立
端口号一般由IANA (Internet Assigned Numbers Authority) 管理
众所周知端口：1~1023，1~255之间为大部分众所周知端口，256~1023端口通常由UNIX占用
注册端口：1024~49151
动态或私有端口：49151~65535
**为什么需要Socket**
普通的I/O操作过程
打开文件－>读/写操作－>关闭文件
TCP/IP协议被集成到操作系统的内核中，引入了新型的“I/O”操作
网络协议具有多样性，如何进行统一的操作
需要一种通用的网络编程接口：Socket
什么是Socket：独立于具体协议的网络编程接口
在ISO模型中，主要位于会话层和传输层之间
BSD Socket（伯克利套接字）是通过标准的UNIX文件描述符和其它程序通讯的一个方法，目前已经被广泛移植到各个平台。
Socket类型
流式套接字(SOCK_STREAM) **TCP**
提供了一个面向连接、可靠的数据传输服务，数据无差错、无重复的发送且按发送顺序接收。内设置流量控制，避免数据流淹没慢的接收方。数据被看作是字节流，无长度限制。
数据报套接字(SOCK_DGRAM) **UDP**
提供无连接服务。数据包以独立数据包的形式被发送，不提供无差错保证，数据可能丢失或重复，顺序发送，可能乱序接收。
原始套接字(SOCK_RAW)
可以对较低层次协议，如IP、ICMP直接访问。
Windows Socket
简称Winsock，是在Windows环境下使用的一套网络编程规范，基于4.3BSD的BSD Socket API制定
1991年Winsock 1.1，16位，由WINSOCK.DLL支持，主要用在Windows 95中
1997年Winsock 2.2 版，32位，由WSOCK32.DLL支持，主要用在Windows 98及以后的版本中
已经成为Windows环境下网络编程的事实标准
Linux Socket
基本上就是BSD Socket
需要使用的头文件
数据类型：#include <sys/types.h>
函数定义：#include <sys/socket.h>
Socket常用函数介绍
基本函数
网络连接函数
socket　创建套接字
bind　绑定本机端口
connect　建立连接
listen　监听端口
accept　接受连接
recv, recvfrom　数据接收
send, sendto　数据发送
close, shutdown　关闭套接字
3.2.2 套接字
UDP和TCP都是用套接字（Socket）抽象
套接字提供进程间通信的一个端点
进程间通信是在两个进程各自的一个套接字之间传送一个消息
3.2.2 套接字
用于互联网地址的Java API
Java提供了一个InetAddress类，用以表示互联网地址
可以通过调用InetAddress的静态方法来获得地址
3.2.3 UDP数据报通信
由UDP发送的数据报从发送进程传输到接收进程，不需要确认或重发。如果发生故障，消息可能无法到达目的地。
当一个进程send数据报，另一个进程receive该数据报时，数据报就会在进程间传送。
要发送或接收消息，进程必须首先创建于一个本地主机的互联网地址和本地端口绑定的套接字。
服务器将它的套接字绑定到一个服务器端口（server port），该端口应让客户端知道，以便客户端给该端口发送消息
客户将它的套接字绑定到任何一个空闲的本地端口。

Receive方法除了获得消息外，还获得发送方的互联网地址和端口，这些信息允许接收方发送应答。
消息大小（message size）
接收进程要指定固定大小的用于接收消息的字节数组。如果消息大于数组大小，那么消息会在到达时被截断。
阻塞（blocking）
套接字通常会提供非阻塞的send操作和阻塞的receive操作
除非套接字上设置了超时，否则receive方法将会一直阻塞直到接收到一个数据报为止。
超时（timeout）
在有些程序中，receive操作不适合无限制等待下去
任意接收（receive from any）
receive方法不指定消息的来源
UDP数据报故障

遗漏故障（omission failures）消息偶尔丢失
排序（ordering）消息有时没有按发送方顺序发送
为了获得所要求的可靠通信的质量，使用UDP数据报的应用要自己提供检查手段
对一些应用而言，使用偶尔有遗漏故障的服务是可接受的：域名服务、VOIP等
UDP的优势在于没有保证消息传递相关的开销
UDP的开销主要包括
- 需要在源和目的地存储状态信息
- 传输额外的消息
- 发送方的延迟

3.2.3 UDP数据报通信
3.2.4 TCP流通信
TCP协议的API源于BSD 4.x UNIX，它提供了可读写的字节流。
流抽象可以隐藏网络的下列特征：
- 消息大小（message size）
- 消息丢失（lost message）
- 流控制（flow control）
- 消息重复和排序（message duplicate and ordering）
- 消息的目的地（message destination）

3.2.4 TCP流通信
连接建立过程




客户和服务器的套接字对由一对流相连接，每个方向一个流
每个套接字都有一个输入流，一个输出流
进程对中的任何一个进程都可以通过将信息写入它的输出流来发送信息给另外一个进程，而另一个进程可以通过读取它的输入流来获得信息
当一个应用close一个套接字时，表示它不再写任何数据到它的输出流。输出缓冲区的中所有数据被送到流的另一端，放在目的地套接字的队列中，并指明流已断开。
流通信的相关问题
数据项匹配（matching of dataitem）两个通信进程需要对流上传送的数据内容达成一致
阻塞(blocking)写入流的数据保存在目的地套接字队列中，当进程试图从输入通道读取数据时，它将直接从队列中获得数据或一直阻塞直到队列中有可用的数据为止。
线程(thread)当服务器接受连接时，它通常创建一个新线程用于与新客户通信。
故障模型
通信完整性保障
- 使用校验和检查并丢弃损坏的数据包
- 使用序列号检测和丢弃重复的数据包

通信有效性保障
- 使用超时和重传来处理丢失的包

如果连接上的包丢失超过了限制以及连接一对通信的进程的网络不稳定或严重阻塞，那么负责发送消息的TCP软件将收不到确认，这种情况持续一段时间后，TCP就会声明该连接中断。
连接中断后，会造成如下后果：
- 使用连接的进程不能区分是网络故障还是另一端进程故障
- 通信进程不能区分最近它们发送的消息是否已被接收

TCP的使用
在TCP连接上经常使用的服务，使用保留端口号
HTTP：80
FTP：21
TELNET：23
SMTP：25
3.3 外部数据表示和编码
存储在运行的程序中的信息都表示成数据结构，而消息中的信息由字节序列组成
不论使用何种通信形式，数据结构在传输时都必须转换成字节序列，到达目的地后再重构
在消息中传送的单个简单数据项可以是不同类型的数据值
不是所有的计算机都以同样的顺序存储整数这样的简单值
大数法、小数法
浮点数的表示也随体系结构不同而不同
用于表示字符的代码集不同
ASCII
Unicode
下列方法可以用于两台计算机交换数据值：
值在传送前先转换成一致的外部数据，然后在接收端转换成本地格式
值按照发送端格式传送，同时传送所使用格式的标志，接收方进行转换

注意：字节本身在传送过程中是不改变的；为了支持RMI或RPC，任何能作为参数或作为结果返回的数据类型必须被转换为字节序列。
表示数据结构和简单值的一致标准称为外部数据表示（external data representation）
3.3 外部数据表示和编码
编码(marshalling)是将多个数据项组成合适消息传送的格式的过程
解码(unmarshalling)是在消息到达后分解消息，在目的地生成相等的数据项的过程
本节中我们讨论三种外部数据表示和编码方法
- CORBA的公共数据表示
- Java的对象序列化
- XML（可扩展标记语言）
3.3.1 COBRA的公共数据表示
CORBA的公共数据表示（Common Data Representation, CDR）是CORBA 2.0定义的外部数据表示。
CDR能表示所有在CORBA远程调用中用作参数和返回值的数据类型
15种简单类型（short，long，unsigned short，…）
1套复合类型
基础类型（Primitive）：
CDR定义了大序法排序和小序法排序的表示
值按照发送端消息中指定的顺序传送
接收端如果要求不同的顺序就要进行翻译
结构化类型（Constructed Types）：
组成每个结构化类型的简单类型值按特定的顺序加到字节序列中
根据在消息中传送的数据项类型的规约，可以自动生成编码操作
数据结构的类型和基本数据类型用CORBA IDL描述
CORBA接口编译器根据远程方法的参数类型和结果类型的定义为参数和结果生成适当的编码和解码操作
3.3.2 Java对象序列化
在Java RMI中，对象和简单数据值都可以作为方法调用的参数和结果传递。一个对象是一个Java类的实例。
例如：CORBA IDL中定义的Person struct作用相当于Java类
上面的类表明实现了Serializable接口，该接口没有方法，意味着它的实例能够序列化
在Java中，serialization指的是将一个对象或者一组关联的对象打平成适合于磁盘存储或消息传送的串行格式，例如：RMI中的参数或结果
解序列化是指从串行格式中恢复对象或一组对象的状态
假设：解序列化进程事先不知道序列化格式中对象的类型，需要将关于每个对象类的一些信息包含在序列化格式中，使得解序列化时能装载恰当的类
类信息由类名和版本号组成，当类有大的改动时要修改版本号
可以由程序员设置
也可以自动根据类名和它的实例变量、方法和接口的名字的Hash值计算
解序列化对象的进程能检查它的类版本是否正确
Java对象可以包含对其他对象的引用，当对象序列化时，它所引用的所有对象也随它一起序列化，以确保对象在目的地重构时它引用的对象也能恢复。
引用对象序列化成句柄----在这种情况下，句柄（handle）是在序列化格式内对一个对象的引用。
序列化过程必须确保对象引用和句柄之间是一一对应的
也必须确保每个对象只能写一次-----在对象第二次出项及之后再出现时，写入句柄而不是对象
对象序列化是一个递归过程
整型、字符型、字节等基本类型实例变量可以通过ObjectOutputStream类来写成一个可移植的二进制格式。
字符串和字符使用writeUTF方法写入
反射的使用
反射实现了根据类名创建类的能力，以及为给定的类创建具有给定参数类型的构造函数
反射使得以完全通用的方式进行序列化和反序列化成为可能
Java对象序列化使用反射找到要序列化的对象的类名，以及该类的实例变量的名字、类型和值
对解序列化而言，序列化格式中的类名用于创建类。然后用类名创建一个新的构造函数，它具有与指定在序列化格式中的类型相应的参数类型。最后，新的构造函数用于创建新的对象，其实例变量的值可以从序列化格式中读取。
3.3.3 可扩展标记语言
可扩展标记语言（Extensible Markup Language，XML）
XML是可扩展的，这意味着用户能定义自己的标记
元素：XML中元素是由匹配的开始标记符和结束标记符包围的字符数据组成
一个元素可以包含其他元素，使得XML具有表示层次的能力
属性：一个开始标记可以选择地包含关联的属性名和属性值

二进制数据：XML元素中所有信息必须表示成字符数据。
解析和良构的文档
XML文档必须是良构的，即它的结构必须符合规则，开始标记必须都要有一个匹配的结束标记
所有的标记都要正确嵌套
3.3.4 远程对象调用
客户端调用远程对象中的一个方法时，就会向存放远程对象的服务器进程发送一个调用消息。这个消息需要指定哪一个对象具有要调用的方法。
远程对象引用（remote object reference）是远程对象的标识符，在整个分布式系统中有效。
远程对象引用在调用消息中传递，以指定调用哪一个对象。
远程对象应用必须确保空间和时间唯一性的方法生成
在远程对象上有许多进程，所以远程对象引用在分布式系统的所有进程中必须是唯一的
一种确保远程对象引用唯一的方法是通过拼接计算机的互联网地址、创建远程对象应用的进程的端口号、创建时间和本地对象编号来构成远程对象引用。
每次进程创建一个对象，本地对象编号就增加1。
3.4 组播通信
一个进程与一组进程通信是必要的，消息成对交换不是一个进程到一组进程通信的最佳模式
组播操作（multicast operation）是更合适的方式，将单个消息从一个进程发送到一组进程的每个成员的操作
组的成员对发送方通常是透明的
组播的行为有很多种情况，最简单的组播不提供消息传递保证或排序保证
组播消息为构造具有下列特征的分布式系统提供了基础设施：
基于复制服务的容错：一个复制服务由一组服务组成，客户请求被组播到所有成员，每个成员执行相同操作
在自发网络中发现服务：客户和服务器使用组播消息来找到可用的发现服务
通过复制的数据获得更好的性能
事件通知的传播：在发生某些事件时通知相关进程
3.4.1 IP组播---组播的通信实现
IP组播（IP multicast）
在网际协议IP的上层实现
IP组播使发送方能够将单个IP数据包传送给组成组播组的一组计算机
发送方不清楚每个接收者身份和组的大小
组播组（multicast group）由D类互联网地址指定
IPV4中，前4位是1110的地址（224.0.0.0～239.255.255.255）
组播成员允许计算机接收发送给组的IP包。组播成员是动态的，计算机可以在任何时间加入或离开，计算机也可以加入任意数量的组。可以无需成为组成员就像一个组发送数据报。
IP组播只能通过UDP实现，应用程序通过发送具有组播地址和普通端口号的UDP数据报完成组播
IPv4组播具有下列特点：
组播路由器（multicast router）：IP数据包既能从局域网上组播，也能在互联网上组播。
- 本地组播使用了局域网的组播能力
- 互联网上的组播利用了组播路由器
- 将单个数据发送到其他成员所在的网络的路由器上，再通过路由器组播到本地成员
- 为了限制组播数据报传播的距离，发送方可以指定允许通过的路由器数量（存活时间TTL）

组播地址可以是永久的，也可以是暂时的
即便没有任何成员，永久组仍然存在
剩下的组播地址可用于临时组，这些组必须在使用前创建，在所有成员离开时消失。创建一个临时组，需要一个空闲的地址，以避免意外的加入到一个已有组中。
- IP协议没有解决这个冲突的问题
- 本地通信可以利用设置小的TTL
- 互联网上需要利用组播地址分配架构（Multicast Address Allocation Architecture）

组播数据报的故障模型
IP组播上的数据报组播与UDP数据报有相同的故障特征：会遭遇遗漏故障，有部分组成员接收到消息
3.4.2 组播的可靠性和排序
IP组播会遇到遗漏故障
一个组播路由器发送到另一个路由器数据报可能会丢失
局域网能也可能由于接收者的缓冲区满而丢失包
组播路由器可能会出现故障
- 局域网内可以接收消息
- 通过路由器到达的成员将不能接收到组播消息

排序问题
互联网上的IP包不一定按照顺序到达
- 从同一个发送者处接收的数据报的顺序可能与其他成员接收的不一样
- 两个进程发送的消息不必以相同的顺序到达组的所有成员

可靠性和排序的效果举例
- 基于复制服务的容错
- 在自发网络中发现服务
- 通过复制的数据获得更好的性能

- 事件通知的传播

一些应用需要比IP组播更可靠的组播协议，需要可靠组播

即传输的任何消息要么被一个组的所有成员收到，要么所有成员都收不到

一些应用对顺序有严格要求，其中最严格的称为全排序组播

传输到一个组的所有消息要以相同的顺序到达所有成员

3.5 网络虚拟化：覆盖网络

互联网通信协议通过API提供了一组有效的构造分布式软件的构造块

- 不断增加的大量不同类型的应用在互联网上并存
- 改变互联网协议来适用运行在其上的每一个应用是不实际的
- IP传输服务是实现在网络技术之上的

网络虚拟化涉及在一个已有的网络之上构造多个不同的虚拟网络

每个虚拟网络被设计成支持一个特定的分布式应用

支持多媒体数据流、多人在线游戏

面向特定应用的虚拟网络构建在一个已有的网络上并为特定的应用进行优化，而不改变底层网络的特征

覆盖网络（overlay network）节点和虚拟链接组成的虚拟网络，位于一个底层网络之上

- 满足一类应用需求的服务或一个特别高层的服务
- 在一个给定的联网环境中的更有效的操作
- 额外的特色

覆盖网络的好处

- 不改变底层网络就能定义新的网络服务
- 鼓励对网络服务进行实验和对服务进行面向特定应用的定制
- 能定义多个覆盖网，它们能同时存在，从而形成更开放和扩展的网络体系结构

覆盖网的不足是引入了额外的间接层（因此可能会有性能损失）

覆盖网与熟悉的分层概念相关

一个覆盖网是一层，是标准体系结构（TCP/IP）外存在的一层

覆盖网可以定义如上所述的网络的核心元素：寻址模式、扫采用的协议、路由方法等

4.1 简介

远程过程调用范型

请求--应答协议，描述了一个基于消息传递的范型，支持在客户/服务器计算中遇到的消息双向传输

远程过程调用（RPC），将传统的过程调用模型扩展到分布式系统中。允许客户程序透明地调用在服务程序中的过程。

远程方法调用（RMI），基于对象的编程模型扩展，允许不用进程运行的对象通过其彼此通信。是对本地方法调用的扩展。

4.2 请求---应答协议

这种通信设计用于支持典型客户/服务器交互中任务和信息的交换

通常情况下，请求---应答通信是同步的，在来自服务器端的应答到达之前客户端进程是阻塞的。从服务器端的应答是对客户端进程的有效确认，因此也是可靠的。

异步的请求---应答通信是可选的，这种方式在客户允许检索应答延迟的情况下是有用的（7.5.2节）。

客户---服务器交换的实现通常采用TCP流的形式

TCP流协议存在不必要的开销

应答在请求之后，所以确认是多余的

建立连接需要三次握手协议

大部分只传递少量的参数，流控制也是多余的

消息标识符

如果需要提供类似于可靠消息传递或请求—应答通信等额外特性，那么任何消息管理方案都会要求每一个消息必须有唯一的消息标识符。通过消息标识符才可以引用消息。

消息标识符有两部分组成

requestID，发送进程从一个长度不断增加的整数序列获取requestID;

发送者进程的标识符，如它的互联网地址和端口号

历史

对于要求重新传输应答而不需要重新执行操作的服务器来说，可以使用历史。

历史，指的是包含已发送的（应答）消息记录的结构

历史的内容包含请求标识符、消息和消息被发送到的客户标识符。其目的是当客户进程请求服务器时让服务器重新传输应答消息

和历史相关的问题是它的内存开销

由于客户每次只能发送一个请求，服务器可以将每个请求解释成客户对上一次应答消息的确认。因此，历史中只需要包含发送个每个客户的最晚的应答消息。

交互协议的类型
为了实现多种类型的请求行为，可以使用三种协议。每种协议在出现通信故障时会产生不同的行为。
请求（R）协议
请求—应答（RR）协议
请求—应答—确认应答（RRA）协议
R协议中，客户端向服务器端发送一个单独的请求消息，这个协议可以用在不需要从远程操作返回值或客户端不需要得到远程操作执行确认的情况中。在发送请求后，客户端可以立即继续执行，而无须等待应答消息。
RR协议，对于绝大多数客户/服务器的交互是有用的，因为它是请求—应答协议的，不要求特殊的确认消息，服务器的应答消息可以看成客户端请求的确认。随后客户端的调用，可以视为服务器应答消息的确认。因UDP包丢失而引起通信故障可以通过带有重新过滤的请求传输和重新传输进行屏蔽。
RRA协议基于三种消息的交互：请求---应答---确认应答。
确认应答消息中包含了来自于应答消息的requestID。服务器可以利用这个ID来从历史中删除相应的条目。
对于到达的确认消息中的requestID来说，它可以看成所有requestID中比其更小的应答消息的确认。因此部分确认消息丢失也不会造成影响。
尽管该交互过程涉及附加的消息，也无须阻塞客户端进程，因为该确认可能在想客户端发送应答之后才传输的。
请求—应答协议的TCP流使用
在请求—应答协议中，服务器用缓冲区来接收请求消息，客户端用缓冲区来接收应答消息。
过程的参数和结果可能是任意长度的，所以数据报长度的限定（通常为8Kb）不适用于透明RMI或RPC系统使用。
实现基于TCP流的请求—应答协议的原因之一是期望避免实现多包协议，因为TCP流可以传输任意长度的参数和结果。
如果使用TCP协议，就能保证可靠的传输请求消息和应答消息，请求—应答协议就没有必要去处理消息的重传、重复消息过滤、历史使用等问题。流控机制也可以传递大量的参数和结果而不需采用特殊措施来避免大规模的接收。
TCP协议可以简化请求—应答协议的实现。
如果同一对客户/服务之间基于同一个流连续发送相同的请求—应答消息，也不需要在每次远程调用上都有连接开销。
如果应用不需要使用TCP提供的所有机制，那么更有效的方法是定制一个基于UDP实现的协议。
HTTP：请求—应答协议的例子
Web服务器有两种不同的实现管理资源的方法：
数据，如HTML网页的正文或图片或面板的类
程序，如Servlets或PHP或运行在Web服务器端的Python程序
客户端请求指定一个包含Web服务器上的DNS主机名和Web服务器上选择端口的URL和在该服务器上资源的标识符。
HTTP协议指定一个消息，该消息涉及：
请求—应答交互、方法、参数、结果及将它们的编码规则，支持一个固定的方法集合（GET，PUT，POST等
准许内容协商和密码式验证
内容协商（content negotiation）：客户端请求中包含说明他们能够接受的数据表示形式的信息，是服务器能选择出对于客户端最合适的数据表示形式
认证（authentication）：凭据（credential）和质询（challenge）用于密码式验证，试图去访问受密码保护的区域时，服务器的应答包含了适用于资源的质询，当客户端接收到质询，它令用户输入用户名和密码，并提交与后续请求关联的凭据。
HTTP基于TCP实现，协议初始版本中，客户/服务器交互由下列步骤组成：
- 客户端请求（连接），服务器在一个默认端口或URL指定的端口接受连接
- 客户端向服务器发送请求消息
- 服务器向客户端发送应答
- 连接断开

每个请求—应答交互都建立、断开连接的代价高昂，引起太多的消息通过网络发送。注意：浏览器一般会向相同的服务器发送多个请求。
HTTP方法
- 每个用户请求指定使用服务器资源的方式和该方法的URL
- 应答则说明该请求的状态
- 请求和应答也可能包含资源数据、表单内容或者运行在Web服务器上的程序资源输出。该方法包含以下内容：

GET HEAD POST PUT DELETE OPTIONS TRACE

### 4.3 远程过程调用

远程过程调用（RPC）是分布式计算的重大突破，使得分布式编程和传统编程相似，即实现了高级的分布透明性。
将传统的过程调用模型扩展到分布式环境方式实现在RPC中调用远程机器上的程序就像这些程序在本地的地址空间中一样。

底层RPC系统隐藏了分布式环境重要的部分，包括对参数和结果的编码和解码、消息传递以及保留过程调用要求的语义。

4.4 **远程方法调用**

远程方法调用（Remote Method Invocation，RMI）与RPC紧密联系，RMI扩展到了分布式对象的范畴
访问对象能够调用位于潜在的远程对象上的方法
RPC和RMI的共性如下：
都支持接口编程
都是典型的基于请求—应答协议构造的，并提供一系列如最少一次、最多一次调用语义
都提供相似程度的透明性
RMI的特殊性：
程序员能够在分布式系统软件开发中使用所有面向对象编程的功能
基于面向对象系统中对象标识的概念，在基于RMI系统中的所有对象都有唯一的对象引用
**伺服器：提供远程对象主体的类的实例**
由相应的骨架传送的远程请求最终是由伺服器处理
- 骨架：远程对象类中有一个骨架，用于实现远程接口中的方法。一个骨架将请求消息中的参数解码，并调用伺服器等待调用完成。之后将结果和异常信息编码，组成应答消息。

当远程对象被实例化时，就会生成一个伺服器。
伺服器的生命周期与远程对象相同，最终也会做为无用单元被回收
RMI软件
它位于应用层对象和通信模块、远程引用模块之间的软件层组成。主要包含如下几种角色：
- 代理
- 分发器
- 骨架

绑定程序是分布式系统中一个单独的服务，它维护一张表，表中包含从文本名字到远程对象引用的映射。
**分布式无用单元收集器**的目的是提供如下保证：
如果一个本地对象引用或者远程对象引用还在分布式对象集合中任何地方，那么该对象本身将继续存在，但是没有任何对象引用它时，该对象将被收集，并且它使用的内存将被回收。
Java分布式无用单元收集算法主要是是基于引用计数
一旦一个远程对象引用进入一个进程，进程就会创建一个代理，主要需要这个代理，它就会一直存在。

5.1 简介

间接通信：在分布式系统中实体通过中介者进行通信，没有发送者和接收者（们）之间的直接耦合
前两讲介绍的技术都是基于发送者和接收者之间的直接耦合
考虑简单的客户—服务器交互，因为是直接耦合，用相同功能的另一台服务器代替原来的服务器很困难。服务器故障后，客户必须显式的处理故障
间接通信避免了直接耦合，使用中介有两个主要特性：
空间解耦：发送者不知道也不需要知道接收者（们）的身份，反之亦然
参与者可以被替换、更新、复制或迁移
时间解耦：发送者和接收者（们）可以有独立的生命周期，发送者和接收者（们）不需要同时存在才能通信，在易变的环境下，发送者和接收者可以随时进入和离开
间接系统常常用于预期会发生改变的分布式系统
例如：移动环境中
间接系统还常用于分布式系统的事件分发，在系统中接收者未知，且易于改变
间接系统的主要缺点是：
增加间接层带来的性能开销
更加难以精确管理
与异步通信的关系
在异步通信中，发送者发送一个消息，然后继续工作（不阻塞），因此不需要与接收者在同一时间通信。
时间解耦增加了额外的维度，发送者和接收者（们）可以互相独立存在。例如：接收者在通信发起时可能不存在。

5.2 组通信

组通信（group communication）提供一种服务，在这种服务中，消息首先被发送到组中，然后该消息被传送到组中的所有成员。
在这个动作中，发送者不清楚接收者们的身份
组通信是对组播通信的抽象，可以通过IP组播实现或等价的覆盖网实现
增加了一些重要特性，如管理组的成员、检测故障、提供可靠性和排序保证
组通信的主要应用领域包括：
面向大量客户的可靠消息分发、支持协作应用、支持一系列容错策略、支持系统监控和管理，包括负载平衡策略

5.2.1 编程模型

在组通信中，核心概念是组和相关的组成员
进程可以加入或离开组

进程可以发送一个消息到组中，然后消息被传播到组中的所有成员，并在可靠性和排序方面提供一定的保证
组通信实现了组播通信，即通过一个操作，消息被发送到组中所有成员
与系统中所有进程通信，而不是其中子组，被称为广播（broadcast），而与单个进程通信被称为单播（unicast）
组通信的重要特征是一个进程事项只发起一个组播操作，它就可以将消息发送到一组进程中的每一个，而不是发起多个发送操作到每个进程

**进程组和对象组**

大多数组服务工作关注进程组（process group）概念，即通信实体是这个组中的进程。这种服务相对低级，因为：
消息被传递到进程，没有进一步提供对分发的支持
消息通常是非结构化的字节数组，不支持对复杂数据类型的编码
对象组（object group）提供更高级的组计算方法
一个对象组是一组对象的集合（形式上是同一个类的实例），这些对象并发地处理同一组调用，然后，各自返回其响应
客户对象不需要知道拷贝，它们调用一个本地对象上的操作，该对象充当组的代理
代理使用组通信系统向对象组的成员发送调用
对象参数和结果如在RMI中一样被编码，相关的调用自动分发到正确的目标对象/方法
其他主要的区别
已经开发了许多组通信服务，它们因各自的假设不同而不同：

**封闭和开放组**：

一个组只有组成员能组播给它，这样的组被称为封闭组。
如果组外的进程可以发送消息给这个组，那么这个组被称为开放组。

**重叠和非重叠组**

在重叠组，实体（进程或对象）可能成为多个组的成员
非重叠组意味着成员不会重叠（也就是说，任一个进程属于至多一个组）
同步和异步系统
需要在两种环境中考虑组通信
5.2.2 实现问题
主要考虑一下几个方面：
底层组播服务在可靠性和排序方面的特性
进程可以在任何时候加入、离开或失效的动态环境中，组成员管理十分重要
在组播中的可靠性和排序
在组通信中，所有成员必须收到发送给本组的消息的拷贝，并且一般具有传递保证
这个保证包括组中每个进程收到的消息应达成的协定
组成员间消息传递顺序应达成的协定
组播系统非常复杂，仅提供最少的传递保证的IP组播也需要很大的工程量
在组播中的可靠性和排序
可靠性
2.4.2节利用完整性、有效性来定义了点对点通信中的可靠性
可靠组播的性质建立在这些语义覆盖之上
完整性保证消息至多被正确的传输一次
有效性保证消息最终会被传递
除此之外，还扩展了第三个特性，即协定（agreement）
所谓协定是指，如果消息被传递到一个进程，那么该消息会被传递到本组中所有进程
排序
组通信还要求对传递到多个目的地的消息提供消息相对排序方面的额外保障
有序是不能由底层进程间通信源语保证的
在组播中的可靠性和排序
排序
组通信服务提供了有序组播（ordered multicast），提供如下一个或多个特性：
FIFO序：先进先出（First-In-First-Out）序，保证了从发送者进程的角度所看到的顺序
如果一个消息在另一个消息之前发送，那么将以这个顺序传递到组中所有进程
因果序：因果序考虑了消息之间的因果关系，如果分布式系统中一个消息在另一个消息之前发生，那么传递相关消息到所有进程时，这种所谓的因果关系将被保留
全序：在全序中，如果在一个进程中，一个消息在另一个之前被传递，那么相同的顺序将在所有进程上被维持
5.2.2 实现问题
组成员管理
组成员服务的四个主要任务：
提供组成员改变接口
组成员服务提供创建和删除进程组、在组中增加或者删除进程的操作
故障检测
检测组成员崩溃、通信故障等。检测器标记进程为可疑和非可疑的
使用故障检测器对组成员做出决策：当怀疑其已经出故障或变得不可达时，从成员中出去该进程

组成员改变时通知成员
当增加进程，或去除进程时（故障或进程有意退出），通知成员
执行组地址扩展
当进程组播一个消息时，它提供组标识而不是组中的一系列进程。成员管理服务将该标识扩展为要传递的当前组成员。服务通过控制地址扩展来协调成员改变时的组播传递
组成员管理
IP组播是一个较弱的组成员服务的例子
有一些组成员服务的特性，但不完全
允许进程动态的加入和离开组并执行地址扩展
对于组播消息，发送者只需提供一个IP组播地址作为目的地
但是IP组播本身不向成员提供当前成员信息，组播传递不会随成员改变而调整
需要维护组成员对基于组的方法有重要影响
尤其是，组通信在小规模、静态系统中很有效；大规模或者高度变化的系统中运行并不完善。

5.3 发布—订阅系统

发布—订阅系统（publish-subscribe systems）有时也称为基于事件的分布式系统（distributed event-based system）
在发布—订阅系统中，发布者（publisher）发布结构化的事件到事件服务，订阅者（subscriber）通过订阅（subscription）表达对特定事件感兴趣，其订阅可以是结构化事件之上的任意模式。
发布订阅系统的任务是把订阅和发布事件进行匹配，保证事件通知（event notification）的正确传递。
一个给定的事件将被传递到多个潜在的订阅者
发布—订阅系统的特征

**异构性**

事件通知被用作一种通信手段，分布式系统中没有被设计实现互操作的组件可以在一起工作。

**异步性**

通知是由生成事件的发布者异步地发送到所有对其感兴趣的订阅者的，发布者和订阅者之间进行了解耦。

5.3.1 编程模型

发布—订阅系统的表达能力由订阅（过滤器）模型决定
以下是几种已定义的常见模式：

**基于渠道 基于主题 基于内容 基于类型**

**基于概念的订阅模型**

过滤器可以根据事件的语义和语法进行表述
复杂事件处理
仅能对单个事件的查询是不够的，需要更为复杂的、能够识别复杂事件模式的系统

5.3.2 实现问题

发布—订阅系统的任务是清楚的：保证所有事件被有效地传递到有过滤器与事件匹配的所有订阅者。除此之外，还可以有安全性、可伸缩性、故障处理、并发和服务质量等额外需求。
本节主要考虑发布—订阅系统的实现问题和所需的系统体系结构
集中式实现与分布式实现
已有许多实现发布—订阅系统的体系结构，最简单的方法是单节点服务器方式的集中式实现。
服务器作为事件代理
发布者发布事件到该代理（还可以选择是否发布广告）
订阅者发送订阅到代理并接收返回的通知
与代理的交互是通过一系列点对点的消息，可以通过使用消息传递或远程调用来实现
集中式方法易于实现，但是设计缺乏弹性和可伸缩性，也意味着可能的单点故障和性能瓶颈
集中式代理被**代理网络**（network of broker）所取代，这些方法可以从故障中生存下来
基于内容的分布式实现方法比较复杂，需要进一步考虑

**泛洪**（Flooding）

最简单的方法是基于泛洪，也就是向网络中的所有节点发送事件通知，在订阅者端执行适当的匹配
另外一种方案，用泛洪发送订阅到所有可能的发布者，在发布端执行匹配，匹配成功的事件通过点对点通信被直接发送到相关的订阅者。

**过滤**

代理网络中采用过滤（filtering）是很多方法所采用的原则，这种方法被称为基于过滤的路由
代理通过一个有路径到达有效订阅者的网络转发通知，实现机制：
通过先向潜在的发布者传播订阅信息
然后在每个代理上存储相关状态
每个结点必须维护
邻居列表（该列表包含了该节点在代理网络中所有相连接的邻居）
订阅列表（该列表包含了由该节点为之服务的所有直接连接的订阅者）
路由表（维护该路径上的邻居和有效订阅列表）
这个方法需要在代理网络中的每个结点上实现匹配过滤

**广告**

上述纯基于过滤的方法会由于订阅的传播而产生大量的网络流，订阅本质上采用了泛洪的方法向所有可能的发布者推送

在广告系统中，通过与订阅传播类似的方式，向订阅者传播广告，这样流量负担可以减少
两种方法间需要权衡，一些系统相继采用了两种方法
**汇聚**（rendezvous）
将所有可能的事件集合看做一个事件空间，将事件空间的责任划分到网络中的代理集合上
汇聚结点是负责一个给定的事件空间的子集的代理节点
5.3.3 发布—订阅系统的例子
5.4 消息队列
分布式消息队列是间接通信系统的一个重要的类别
消息队列使用队列概念作为一种间接机制提供点对点的服务
消息队列可以实现时间和空间解耦
消息队列也称消息中间件
5.4.1 编程模型
它提供了在分布式系统中通过队列进行通信的一种方法，生产者进程发送消息到特定队列，其他（消费者）进程从该队列中接受消息
通常支持三种接收方式：
**阻塞接收**
**非阻塞接收（轮询操作）**
**通知操作**
一些进程能将消息发送到同一个队列，同样也有一些接收者能从队列中取出消息。
排队的策略通常是先进先出（FIFO），但是大多数队列的实现也支持优先级概念，即高优先级的消息先被传递。
消费者进程也能基于消息的优先级从队列中选择消息。
一条消息通常由以下内容组成
目的地（即一个指定目的队列的唯一标识符）
消息的相关元数据（包括：消息优先级、传递模式等）
消息体：消息体通常是不透明的，且未被消息队列系统改变过
通常通过定义在元数据上的谓词表示选择消息的规则
消息队列系统的一个重要特性是消息是持久的---也就是说，消息队列会无限期存储消息（直到它们被消费为止），并将消息提交到磁盘，以实现可靠传递。
5.5 分布式共享内存
共享内存的抽象是另外一种间接通信范型
为并行计算开发，分布式共享内存在读和写字节级别操作
分布式共享内存可以通过地址进行访问
分布式共享内存（DSM）是一种抽象，用于给不共享物理内存的计算机共享数据
进程通过读和更新看上去是其他地址空间中普通的内存来访问DSM
底层的运行时系统透明地保证运行在不同的计算机上的进程可以观察到其他进程的更新
5.5.2 元组空间通信
元组空间是耶鲁大学的David Gelernter 作为分布式计算的一种新形式引入的，它基于David Gelernter提出的“生成通信”
进程通过在元组空间放置元组间接地进行通信，其他进程可以从该元组读或删除元组
元组没有地址，但是可以通过内容上的模式匹配进行访问（内容可寻址的内存）
所形成的Linda编程模型有很广泛的影响力，并在分布式编程方面带来了重大的发展，包括：Agora系统，Sun的JavaSpaces和IBM的Tspaces
元组空间通信在无处不在的计算领域也很有影响
编程模型
在元组编程模型中，进程通过元组空间（一个共享的元组集合）进行通信。元组由一个或多个带类型的数据域组成
进程通过访问同一个元组空间实现共享数据：
- 通过使用write操作将元组放置在元组空间中
- 使用read或者take操作从元组空间读或提取元组

从元组空间中读或者删除元组时，一个进程提供了一个元组规约，元组空间返回符合该规约的任何元组
为了使得进程能够同步其活动，read和take操作都会阻塞，直到在元组空间中找到一个相匹配的元组
一个元组规约包括域的数量和所需的域值或者域类型
编程模型
在元组空间范型中，不允许直接访问元组空间中的元组，进程必须替换元组空间中的元组而不是修改它。元组是保持不变的。
与元组空间相关的特性
空间解耦
放置在元组空间中的元组可能源自任何数量的发送者进程，也可能被传递到任何一个潜在的接收者
时间解耦
放置在元组空间中的元组会保留在元组空间中直到被删除（可能是无限期的），因此，发送者和接收者不需要在时间上重叠
这些特性提供了一种在空间和时间上完全分布的方法，还通过元组空间提供了一种共享变量的分布式共享形式

8.1 简介
分布式系统中，时间一个重要的问题
在分布式系统中，物理时间的概念也是不准确的
这不是由于相对性的影响，相对性在常规计算机中可以忽略或不存在
主要是问题是：受目前技术能力的限制，不能准确记录不同结点上的事件的时间，以便知道事件发生的顺序或事件是否同时发生
分布式系统中没有绝对的全局时间
但是，分布式系统中需要确定事件的某些状态是否同时出现
例如：确认某一对象的引用是否不存在
8.2 时钟、事件和进程状态
进程交互模型
假设一个分布式系统由N个进程$p_i$(i=1,2,⋯,N)组成，记为P
每个进程在一个处理器上执行，处理器之间不共享内存
在P中，$s_i$是$p_i$的状态，通常在进程执行时进行状态变换
进程状态包括进程中所有变量的值，还包含它影响本地操作系统环境中的对象的值。
假设，进程除通过网络发送消息外，进程之间不能相互通信
当进程$p_i$执行时，它会采取一系列的动作，每个动作或是一个消息send/receive操作，或一个转换状态$p_i$的操作
即改变$s_i$中的一个或多个值
事件
定义：发生了一个动作（通信/状态转换），该动作由一个进程完成
进程$p_i$的事件序列可以用全序方式排列
事件之间的关系可以用i表示
进程$p_i$中事件e在e'之前发生，可以表示为$e \rightarrow_i e'$
在单个处理器上执行时，不论进程是否是多线程的，这个排序都是良好定义的
进程$p_i$的历史定义为在该进程中发生的一系列事件，按关系i排序：
$history(p_i) = h_i = <e_i^0, e_i^1, e_i^2, \cdots>$
时钟
解决如何事件标记时间戳的问题
每个计算机有它们自己的物理时钟
操作系统读取结点的硬件时钟值$H_i(t)$, 按照一定比例放大，再加上一个偏移量，从而产生软件时钟$C_i(t) = \alpha H_i(t)+\beta$,用于近似度量进程$p_i$的实际物理时间t
通常时钟不完全准确，因此$C_i(t)$与t不同。但是，如果$C_i$的表现足够好时，我们可以用它的值给$p_i$的事件打时间戳
时钟偏移和时钟漂移
两个时钟的读数之间的瞬间不同称为时钟偏移（clock skew）
时钟漂移是指单个时钟读数和名义上的完美的参考时钟之间的偏移
漂移率(drift rate)是指参考时钟度量的每个单位时间内，在时钟和名义上完美的参考时钟之间的偏移量。
通用协调时间
计算机时钟能与外部的高精度时间源同步
8.3 同步物理时钟
为了知道分布式系统P的进程中事件发生的具体时间，有必要用权威的外部时间源同步进程的时钟$C_i$ ---- 外部同步（external synchronization）。
如果时钟$C_i$与其他时钟同步到一个已知的精度，那么我们就能通过本地时钟度量在不同计算机上发生的两个事件的间隔 ----内部同步（internal synchronization）
我们在实际时间I的一个区间上定义两个同步模式
8.3 同步物理时钟
外部同步：
设一个同步范围D>0，UTC时间源为S，I中的所有实际时间t，满足$|S(t)-C_i(t)|<D$，其中i=1,2,⋯,N。时钟$C_i$在范围D中是准确的
内部同步：
设同步范围D>0，I中所有实际时间t，则有$|D_i(t)-C_j(t)|<D$，其中i,j=1,2,⋯,N。时钟$C_i$在范围D中是一致的
内部同步的时钟未必是外部同步的
如果系统P在范围D内是外部同步的，那么该系统在范围2D内是内部同步的
时钟正确性（correctness）通常定义为，如果一个硬件时钟H的漂移率在一个已知的范围ρ>0内（该值从制造商处获得，例如$10^{-6}$s/s），那么该时钟是正确的。
这表明度量实际时间t和t'的时间间隔的误差是有界的 $(1- \rho)(t'-t) \leqslant H(t')-H(t) \leqslant (1+ \rho)(t'-t)$
该条件禁止了硬件时钟值的跳跃，有时，软件也时钟也要求遵循该条件。但是用一个较弱的单调性条件就足够了。
单调性是指一个时钟C前进的条件

$$t'>t \Longrightarrow C(t') > C(t)$$

不满足正确性的时钟就被定义成是有故障(faulty)的
当时钟完全停止滴答，称为时钟的崩溃故障（crash failure）
其他时钟故障是随机故障（arbitrary failure）
例如：千年虫故障、时钟电池不足

注意：根据定义，时钟不必非常正确，因为其目标可以是内部同步而不是外部同步，正确的标准仅仅与时钟“机制”的正常运行有关，而不是它的绝对设置

8.3.1 同步系统中的同步

考虑最简单的情况：

在一个同步分布式系统中，两个进程之间的内部同步

在同步系统中，已知时钟漂移率的范围、最大的消息传输延迟和进程每一步的执行时间

一个进程在消息m中将本地时钟的时间发送给另一个进程。

原则上，接收进程可以将它的时钟设成$t+T_{trans}$

其中$T_{trans}$是在两个进程间传输m所花的时间。但是$T_{trans}$是常常变化和未知的。

根据定义，在同步系统中，用于传输消息的时间有一个上界max。

- 设消息传输时间的不确定性为u，那么u=(max-min)。
- 如果接收方将时钟设置为t+min，那么时钟偏移至多为u。
- 如果接收方将时钟设置为t+max，那么时钟偏移至多为u。
- 如果设置为t+(max+min)/2，那么是时钟偏移至多为u/2。
- 同步系统要同步N个时钟，可获得的时钟偏移最优范围是u(1-1/N)

大多数实际的分布式系统是异步的，导致消息延迟的原因很多，消息传输延迟没有上界max。对于一个异步系统，我们只能说$T_{trans}$ =min+x，其中$x \geqslant 0$

8.3.2 同步时钟的Cristian方法

Cristian[1989]建议使用一个时间服务器，它连接到一个接收UTC信号的设备上，用于实现外部同步

在接收到请求后，服务器S根据它的时钟提供时间

只有在客户和服务器之间的往返时间与所要求的精确性相比足够短，该方法才能达到同步

进程p在消息$m_r$中请求时间，从消息$m_t$中接收时间值t

进程p记录了发送请求$m_r$和接收应答$m_t$的整个往返时间$T_{round}$

如果时钟漂移率小，那么该值可以比较精确地度量这段时间

例如往返在LAN上应该达到1-10ms数量级，漂移率为$10^{-6}$s/s的时钟在这段时间里变化至多$10^{-5}$ms

假设S在$m_t$中放置t，往返时间在t时间点之前和之后平分，那么估计进程p应该设置它的时钟为$t+T_{round}/2$

假设最小传输时间min的值是已知的或者能保守地估计

- S能在$m_t$中放置的最早时间点是在p发出$m_r$之后的min
- 它能做此工作的最近时间点是在$m_t$到到达p之前的min
- 因此，应答消息到达S时，时钟的时间位于范围$[t+min, t+T_{round}-min]$
- 精度是$\pm(T_{round}/2-min)$

通过给S发送几个请求，并用$T_{round}$最小值给出最精确的估计

精确性要求越高，达到它的可能性越小

8.3.3 Berkeley算法

Gusella和Zatti[1989]描述了一个内部同步算法，用于运行Berkeley UNIX的计算机群。

算法的主要步骤

1. 该算法需要选择一台协调者计算机作为主机（master）
2. 主机定期轮询其他要同步时钟的计算机（从属机）
3. 从属机（slave）将它们的时钟值返回给主机
4. 主机通过观察往返时间来估计它们的本地时钟时间，并计算所获得值的平均值
5. 协议的准确性依赖于主从机之间的名义上最大往返时间
6. 主机发送每个从属机的时钟所需的调整量

主要采用容错平均值（fault-tolerant average）

时钟中选择差值小于一个指定量的子集

平均值仅根据这些时钟的读数计算

如果主机出现故障，要能选举另一个主机接管

8.3.4 网络时间协议

网络时间协议（Network Time Protocol，NTP）定义了时间服务的体系结构和在互联网上发布时间信息的协议

NTP主要的设计目标和特色如下：

提供一个服务，使得跨互联网的用户能精确地与UTC同步

提供一个能在漫长的连接丢失中生存的可靠服务

使得客户能经常有效地重新同步以抵消在大多数计算机中存在的漂移率

提供保护，防止对时间服务器的干扰，无论是恶意的还是偶然的

NTP服务由互联网上的服务器网提供

主服务器（primary server）直接连接到像无线电时钟这样的接收UTC源

二级服务器（secondary server）与主服务器同步

服务器在一个称为同步子网的逻辑层次中连接

因为在同步的每一层都会引入误差，层次数大的服务器上的时钟比层次数小的服务器上的时钟更容易不准确。
NTP在评估某个服务器拥有的计时质量时，也考虑了整个消息到根的往返时间延迟。
在服务器不可达或出现故障时，同步子网可以重配置。
NTP服务器用以下三种模式中的一种相互同步：
**组播**
**过程调用**
**对称模式**
组播模式
- 用于高速LAN，一个或多个服务器定期将时间组播到由LAN连接的其他结点，并设置它们的时间。需要基于延迟很小这一假设。

过程调用模式（procedure-call mode）
- 类似与Cristian算法
- 服务器从其他计算机接收请求，并用时间戳应答

对称模式（symmetric mode）
- 用于在LAN中提供时间信息的服务器和同步子网的较高层
- 一对服务器交换有时序信息的消息。时序数据作为服务器之间的关联的一部分被保留，时序数据可用于提高时间同步的准确性

所有模式都采用不可靠的UDP协议进行传输
过程调用模式和对称模式中，进程交换消息对
每个消息有最近消息的时间戳：发送和接收前一个NTP消息的本地时间，发送当前消息的本地时间。
NTP消息的接收者记录它接收消息的本地时间
对于两个服务器之间发送的每对消息，由NTP计算偏移$o_i$和延迟$d_i$
偏移$o_i$是对两个时钟之间实际偏移的一个估计
延迟$d_i$是两个消息整个的传输时间
如果B上的时钟相对于A的真正偏移是o，而m和m'实际的传输时间分别为t和t'，那么我们可以得到：
$T_{i-2} = T_{i-3} + t + o$ 和 $T_i = T_{i-1} + t' - o$
我们可以推出：
$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$
以及
$o = o_i + (t'-t)/2$，其中$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$
利用$t$和$t' \geqslant 0$的事实，有$o_i - d_i/2 \leqslant o \leqslant o_i + d_i/2$。$o_i$是偏移的估计，$d_i$是该估计的精确性度量
8.4 逻辑时间和逻辑时钟
从单个进程的角度看，事件可唯一地按照本地时钟显示的时间进行排序
可以采用类似物理因果关系的方法，这种排序基于以下两点：
如果两个事件发生在同一个进程$p_i$（i=1，2，…，N）中，那么它们发生的顺序是$p_i$观察到的顺序
当消息在不同进程之间发送时，发送消息的事件在接收消息的事件之前发生
Lamport将推广这两种关系得到的偏序关系成为发生在先关系。也称为因果序或潜在的因果序
我们可以按照如下方式定义发生在先关系（用表示）：
HB1：如果∃进程$p_i$：$e \rightarrow_i e'$，那么$e \rightarrow e'$
HB2：对任一消息m，send(m)→receive(m)
HB3：如果e, e', e" 是事件，且有$e \rightarrow e'$和$e' \rightarrow e"$，那么$e \rightarrow e"$
8.4 逻辑时间和逻辑时钟
**逻辑时钟**
Lamport[1978]提出了一种简单机制，数字化捕获发生在先排序
Lamport逻辑时钟是一个单调增长的软件计数器，它的值和任何物理时钟无关。
每个进程$p_i$维护它自己的逻辑时钟$L_i$，进程利用逻辑时钟给事件添加Lamport时间戳
$L_i(e)$表示$p_i$的事件e的时间戳，用L(e)表示发生在任一进程中的事件e的时间戳
为了捕获发生在先关系，进程按照如下规则修改它们的逻辑时钟，并在消息中传递它们的逻辑时钟值
LC1：在进程$p_i$发出每个事件之前，$L_i$加1：$L_i = L_i + 1$
LC2：
(a)当进程pi发送消息m时，在m中附加值t=Li；
(b)在接收(m, t)时，进程$p_j$计算$L_j = \max(L_j, t)$，然后在给receive(m)事件打时间戳时应用LC1
**全序逻辑时钟**
Lamport时间戳不能对不同进程生成的不同事件都进行排序
但是，很多情况下我们需要对所有事件排序，称之为事件的全序
如果e是在$p_i$中发生的事件，本地时间戳为$T_i$；e'是在$p_j$中发生的事件，本地时间戳为$T_j$，则这些事件的全局逻辑时间戳为$(T_i,i)$和$(T_j,j)$。
当且仅当$T_i < T_j$或$T_i = T_j$以及$i<j$时，定义$(T_i,i) < (T_j,j)$
这种排序没有通常的物理意义，但是有时是有用的
**向量时钟**
Mattern和Fidge开发了向量时钟，用以克服Lamport时钟的缺点： L(e)<L(e')不能推倒出e→e'
有N个系统的向量时钟是包含N个整数的数组
每个进程维护它自己的向量时钟$V_i$，用于给本地事件加时间戳
进程在发送给对方的消息中附件向量时间戳，更新时钟规则如下：
VC1：初始情况下，$V_i[j]=0$，i, j=1, 2, …, N.

VC2：在$p_i$给事件加时间戳之前，设置$v_i[i]:=v_i[i]+1$
VC3：$p_i$在它发送的每个消息中包含t=V
VC4：当$p_i$接收到消息中的时间戳t时，设置$v_i[j]:=max(V_i[j],t[j]), j=1,2,\cdots,N$
这种取两个向量时间戳的最大值的操作称为合并（merge）操作
向量时钟$V_i$，$V_i[i]$是$p_i$已经附加时间戳的事件的个数，$V_i[j](j\neq i)$是在$p_j$中发生的可能会影响$p_i$的事件的个数
设V(e)是发生e的进程所应用的向量时间戳。通过在与事件e和e'相关的事件序列的长度上进行归纳，可以看到e→e' ==> V(e)<V(e')；如果V(e)<V(e')，那么e→e'

8.5 全局状态

**分布式无用单元收集**：如果分布式系统中不再对某个对象进行任何引用，那么该对象被认为是无用的。一旦认为对象是无用的，那么就要回收它所占用的内存。
为了检查一个对象是否是无用的，必须验证系统中对它没有任何引用。
**分布式死锁检测**：当一组进程中的每一个进程都在等待另一个进程给它发消息，并且在这种"等待"关系图中存在循环时，就会发生分布式死锁。
**分布式终止检测**：分布式系统中所有进程都已经停止

8.5.1 全局状态和一致割集

观察单个进程的连续状态是可能的，但查明系统的全局状态问题是非常困难的
本质问题是缺乏全局时间
从进程状态集中，我们可以判断进程是否发生死锁等
分布式系统P，包含N个进程$p_i$（i=1,2,⋯,N）
每个进程中发生一系列事件，我们通过每个进程的历史来描述每个进程的执行过程：
$history(p_i) = h_i = <e_i^0,e_i^1,e_i^2,\cdots>$
$h_i^k = < e_i^0,e_i^1,e_i^2,\cdots, e_i^k>$ 表示进程历史的任何一个有限前缀
每个事件或是进程的内部动作或是在进程相连的信道上发送或接收一个消息
每个进程记录本进程发生的事件，以及它经过的连续状态
$s_i^k$表示进程$p_i$在第k个事件发生之前的状态，$s_i^0$是进程$p_i$的初始状态
如果$p_i$记录它发送了消息m到进程$p_j$，那么通过检查$p_j$是否接收到该消息，就能推断出m是否是$p_i$和$p_j$之间信道状态的一部分
通过取单个进程历史的并集，我们可以得到P的**全局历史**(global history)
$H = h_0 \cup h_1 \cup h_2 \cdots \cup h_{N-1}$
一个全局状态相当于单个进程历史的初始前缀
**系统执行的割集**（cut）是系统全局历史的子集，是进程历史前缀的并集$C = h_1^{C1} \cup h_2^{C2} \cup \cdots \cup h_N^{CN}$
对于割集C的全局状态S中的状态$s_i$是由$p_i$处理的最后一个事件即$e_i^{ci};i(i=1,2,\cdots,N)$之后的$p_i$的状态。事件集$\{e_i^{ci}: i=1,2,\cdots,N\}$ 称为割集的边界
割集C是一致的，条件是对它包含的每个事件，它也包含了所有在该事件之前发生的所有事件，即
$\forall e \in C, fe \Rightarrow f \in C$
**一致的全局状态**（consistent global state）是指对应于一致割集的状态。我们可以把一个分布式系统的执行描述成在系统全局状态之间的一系列转换：
$S_0 \dashrightarrow S_1 \dashrightarrow S_2 \dashrightarrow \cdots$
在每个转换中，正好一个事件在系统的一个进程中发生。这个事件或是发送消息，或是接收消息，也可以是一个外部事件。如果两个事件同时发生，我们可以认为它们按一定的顺序发生，按照进程标识符排序。系统通过一致全局状态以这种方式逐步发展。
"走向"（run）是全局历史中所有事件的全序，并且它与每个本地历史排序i(i=1,2,⋯,N)是一致的。
线性化走向或一致的走向是全局历史中所有事件的全序，并且与H上发生在先关系是一致的。
不是所有的走向都经历一致的全局状态，但所有线性化走向只经历一致的全局状态。如果有一个经过S和S'的线性化走向，我们说状态S'是从状态S可达的(reachable)
有时，我们可以在一个线性化走向中变换并发事件的顺序，得到的走向仍是经历一致全局状态的走向
检测像死锁和终止之类的条件实际上是求一个全局状态谓词的值
全局状态谓词是一个从系统P的进程全局状态集映射到{Ture, False}的函数。
与对象成为无用、系统死锁、系统终止的状态相关的谓词是**稳定的**(stable)：一旦系统进入谓词值为True的状态，它将在所有可从该状态可达的状态中一直保持True。
**安全性**(safety)是一个系统全局状态的谓词的断言
假设有一个不希望有的性质α（例如，α可以是成为死锁的性质）
设$S_0$是系统的原始状态，关于α的安全性断言，即对所有可从$S_0$到达的所有状态S，α的值均为False
**活性**(liveness)
设β是系统全局状态希望有的性质（例如，可达终止的性质）
关于β的活性是对任一从状态$S_0$开始的线性化走向L，对可从$S_0$到达的状态$S_L$，β的值为True

9.1 简介

分布式系统中进程如何协调它们的动作或对一个或多个值达成协定。

分布式系统理论的一个基本结果：在异步分布式系统中，即使良性故障条件下，也不可能保证一组进程能对一个共享值达成协定。

故障假设和故障检测器

本章假设每对进程都可通过可靠的通道连接

假设进程故障不隐含对其他进程的通信能力的威胁

注意: 一个可靠的通道最终将消息传递到接收者的输入缓冲区。

在某个时间间隔内，一些进程之间的通信可能成功，而另一些进程之间的通信则被延迟。

**故障假设和故障检测器**

在点对点的网络上，复杂的网络拓扑结构和独立的路由选择意味着连接可能是非对称的(asymmetric)

连接还可能是非传递的

进程p到进程q，以及进程q到进程r都可以通信，但是进程p到进程r不能通信

因此，可靠性假设要包括任何有故障的链接或路由器最终会被修复或避开的内容。

**故障检测器**（failure detector）

用于处理有关某个进程是否已出现故障的查询

故障检测器通常是由每个进程中的一个对象实现的，此对象与其他进程的对应部分一起执行一个故障检测算法

每个进程中的这个对象叫做本地故障检测器（local failure detector）

不可靠故障检测器（unreliable failure detector）

当给出一个进程标识时，一个不可靠故障检测器会产生两个值：

- Unsuspected：表示检测器最近已收到表明进程没有故障的证据 例如：最近从该进程收到一个消息，但是进程可能在此后出现的了错误
- Suspected：表示故障检测器有迹象表明进程可能出现故障了 例如：在多于最长沉默时间里没有收到来自进程的消息

**可靠的故障检测器**（reliable failure detector）

能够精确检测进程故障的检测器

对于进程的问询，可以给出：

- Unsuspected：表示检测器最近已收到表明进程没有故障的证据
- Failed：表示检测器确定进程已崩溃

不可靠故障检测器实现

每个进程p向其他所有进程发送消息”p is here”，并且每隔T秒发送一次

故障检测器用最大消息传输时间D秒，作为评估值

如果进程q的本地故障检测器在最后一次T+D秒内没有收到”p is here”消息，则向q报告p是Suspected

但是如果后来收到”p is here”消息，则向q报告p是OK

9.2 分布式互斥

分布式进程常常需要协调它们的动作

- 如果一组进程共享一个或一组资源，那么访问这些资源时，常需要互斥来防止干扰并保证一致性
- 在操作系统领域中常见的临界区问题
- 在分布式系统中需要一个仅基于消息传递的分布式互斥问题解决方案

在某些情况下，管理共享资源的服务器也提供互斥机制。另外一些情况下，则需要一个单独的用于互斥的机制。

**互斥算法**

考虑无共享变量的N个进程$p_i$, (i=1，2，…，N) 的系统，这些进程只在临界区访问公共资源

假设：

- 只有一个临界区
- 系统是异步的
- 进程不出故障
- 消息传递是可靠的
- 任何消息被完整的恰好发送一次

执行临界区的应用层协议：

enter 0

resourceAccesses 0

exit 0

对互斥的基本要求：

ME1：安全性，在临界区（CS）一次最多有一个进程执行

ME2：活性，进入和离开临界区的请求最终成功执行，隐含既无死锁也无饥饿问题

ME3：顺序，如果一个进入CS的请求发生在先，那么进入CS时仍按此顺序

互斥算法性能的评价标准：

- 消耗的带宽，与每个entry和exit所发送的消息量成正比
- 每一个entry和exit操作由进程导致的客户延迟
- 算法对系统吞吐量的影响
- 一个进程离开临界区和下一个进程进入临界区之间的同步延迟（synchronization delay）来衡量这个影响

**中央服务器算法**

使用一个服务器来授予进入临界区的许可

满足安全性和活性要求，但不满足顺序要求。

性能

- 带宽消耗

enter():2个消息，即请求消息和授权消息

exit(): 1个消息，即释放消息
- 客户延迟
消息往返时间导致请求进程延迟
- 同步延迟
1个消息的往返时间
- 性能瓶颈
服务器
**基于环的算法**
满足安全性和活性要求，但不满足顺序要求。
性能
- 带宽消耗
由于令牌的传递，会持续消耗带宽
- 客户延迟
Min: 0个消息，正好收到令牌
Max: N个消息，刚刚传递了令牌
- 同步延迟
Min: 1个消息，进程依次进入临界区
Max: N个消息，一个进程连续进入临界区，期间无其他进程进入临界区
**使用组播和逻辑时钟的算法**
进程进入临界区需要所有其它进程的同意
组播+应答
并发控制：采用Lamport时间戳避免死锁
满足安全性、活性和顺序要求。
初始化：
state:=RELEASED;
为了进入临界区：
state:=WAITED;
组播请求给所有进程;
T:=请求的时间戳;
Wait until (接收到的应答数=(N－1));
state:=HELD;
在pj(i≠j)接收一个请求<Ti,pi>
if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
then 将请求放入pi队列,不给出应答;
else 马上给pi应答;
end if
为了退出临界区：
state := RELEASED;
对已进入队列的请求给出应答;
性能
- 带宽消耗
enter(): 2(N－1)，即(N－1)个请求、(N－1)个应答
- 客户延迟 1个消息往返时间
- 同步延迟 1个消息的传输时间
**Maekawa投票算法**
进程进入临界区需要部分其它进程的同意
每个进程$p_i$关联到一个选举集$V_i$
**On initialization**
state := RELEASED;
voted := FALSE;
**For pi to enter the critical section**
state := WANTED;
Multicast request to all processes in Vi;
Wait until (number of replies received = K);
state := HELD;
**On receipt of a request from pi at pj**
if (state = HELD or voted = TRUE)
then
queue request from pi without replying;
else
send reply to pi;
voted := TRUE;
end if
**For pi to exit the critical section**
state := RELEASED;
Multicast release to all processes in Vi;
**On receipt of a release from pi at pj**
if (queue of requests is non-empty)
then
remove head of queue – from pk, say;
send reply to pk;
voted := TRUE;
else
voted := FALSE;
end if
Maekawa算法会产生死锁
三个进程$p_1$、$p_2$和$p_3$，且$V_1=\{p_1, p_2\}$，$V_2=\{p_2, p_3\}$，$V_3=\{p_3, p_1\}$。若三个进程并发请求进入临界区，考虑下列情况：
1. $p_1$应答了自己，但延缓$p_2$；
2. $p_2$应答了自己，但延缓$p_3$；
3. $p_3$应答了自己，但延缓$p_1$。
Maekawa算法改进后可满足安全性、活性和顺序性
进程按发生在先顺序对待请求队列
性能
- 带宽消耗
即进入需要$2\sqrt{N}$个消息，退出需要$\sqrt{N}$个消息
- 客户延迟 1个消息往返时间
- 同步延迟 较差，1个往返时间，非单个消息的往返时间
容错
当消息丢失时会发生什么
当进程崩溃时会发生什么

如果通道不可靠，前面介绍的算法都不能容忍消息丢失

**基于环的算法不能容忍任何单个进程的崩溃故障**

**Maekawa算法可以容忍一些进程的崩溃**

崩溃的进程不在所需的投票集中

**中央服务器算法可以容忍一个既不持有也不请求令牌的客户进程的崩溃**

9.3 选举

选举算法(election algorithm)

选择一个唯一的进程来扮演特定角色的算法

召集选举(call the election)

一个进程启动了选举算法的一次运行

参加者(participant)

进程参加了选举算法的某次运行

非参加者(non-participant)

进程当前没有参加任何选举算法

进程标识符

唯一且可按全序排列的任何有用的数值

每个进程$p_i$有一个变量$elected_i$，用于包含当选进程的标识符。当进程第一次成为选举参与者时，变量值设置为⊥

基本要求

E1：安全性 参与的进程$p_i$有$elected_i = \perp$或$elected_i = P$（P是在运行结束时具有最大标识符的非崩溃进程）

E2：活性 所有进程$p_i$都参加并且最终置$elected_i \neq \perp$或进程$p_i$崩溃

- 带宽消耗
- 回转时间

从启动算法到终止算法之间的串行消息传输的次数

**基于环的选举算法**

目的：在异步系统中选举具有最大标识符的进程作为协调者

基本思想：按逻辑环排列一组进程

最初，每个进程标记为选举中的非参与者，任何进程可以开始一次选举

- 将自身标记为参与者
- $id_{msg} = id_{local}$，发送 选举消息{elect, $id_{msg}$}至邻居

非参与者转发选举消息

- 将自身标记为参与者
- 发送 {elect, MAX($id_{local}$, $id_{msg}$)}至邻居

当$id_{local}=id_{msg}$时，该进程成为协调者

- 将自身标记为非参与者
- $id_{coordinator} = id_{local}$, 发送当选{elected, $id_{coordinator}$ }至邻居

参与者转发选举结果消息

- 将自身标记为非参与者
- 记录$id_{coordinator}$

基于环的选举算法的性能

最坏情况

- 启动选举算法的逆时针邻居具有最大标识符，共计需要3N − 1个消息，回转时间为3N − 1

最好情况

- 回转时间为2N

不具备容错功能

**霸道算法**

3种类型的消息

- 选举消息，用于宣布选举
- 应答消息，用于回复选举消息
- 协调消息，用于宣布当选进程的身份

一个进程通过超时发现协调者已经出现故障，并开始一个选举，几个进程可能同时观察到此现象

采用同步系统，因此可以构造一个可靠的故障检测器

进程P在发现协调者失效后启动一次选举，将选举消息发送给具有更大标识符的进程，并等待应答。

若进程P在时间T内没有收到回答消息，则认为自己为协调者，并给所有具有较小标识符的进程发送协调者消息。

若进程P收到回答消息，则等待协调者消息；若消息在一段时间内没有到达，则启动一次新的选举算法。

进程收到协调者信息后，设置 $elected_i = id_{coordinator}$

进程收到选举消息，回送一个应答消息，并开始另一次选举，除非它已经开始了一次选举

进程如果知道自己具有最大标识符，则会决定自己是协调者，并向其他进程宣布

最好情况 = N − 2

- 标识符次大的进程发起选举
- 发送N − 2个协调者消息
- 回转时间为1个消息

最坏情况 = $O(N^2)$

- 标识符最小的进程发起选举

9.4 组播通信中的协调与协定

组播：发送一个消息给进程组中的每个进程。

广播：发送一个消息给系统中的所有进程。

组播面临的挑战

**效率**

- 带宽使用
- 总传输时间

**传递保证**

- 可靠性
- 顺序

**进程组管理**

- 进程可任意加入或退出进程组

**系统模型**

系统包含一组进程，它们可以通过一对一的通道可靠地进行通信。进程在崩溃时才出现故障。

multicast(g, m)操作

1个进程发送消息给进程组g的所有成员

deliver(m)操作

传递由组播发送的消息到调用进程

9.4.1 基本组播

基本组播

- 一个正确的进程最终会传递消息
- 原语: B-multicast、B-deliver
- 可靠组播，与IP组播不同

简单实现

- B-multicast(g,m)：对每个进程p ∈ g，send(p,m)
- 进程p receive(m)时：p执行B-deliver(m)

多线程

- 利用线程来并发执行send操作

确认爆炸(ack-implosion)

- 确认从许多进程几乎同时到达
- 组播进程丢弃部分确认消息导致重发现象

9.4.2 可靠组播

**可靠组播**

性质

- 完整性(integrity)

一个正确的进程p传递一个消息m至多一次

- 有效性(validity)

如果一个正确的进程组播消息m，那么它终将传递m。

- 协定(agreement)——具有原子性

如果一个正确的进程传递消息m，那么在group(m)中的其它正确的进程终将传递m。

1. 用B-multicast实现可靠组播

算法评价

- 满足有效性

一个进程的最终将B-deliver消息到它自己。

- 满足完整性

B-multicast中的通信通道具有完整性

- 遵循协定

每个正确的进程在B-deliver消息后都B-multicast该消息到其它进程

- 效率低

每个消息被发送到每个进程|g|次。

2. 用IP组播实现可靠组播

特点

- 基于IP组播：IP组播通信通常是成功的
- 捎带确认：在发送给组中的消息中捎带确认
- 否定确认：进程检测到它们漏过一个消息时，发送一个单独的应答消息。

算法

$S_g^p$: 进程为它属于的组g维护的序号，初始化为0。

$R_g^q$: 进程记录来自进程q并且发送到组g的最近消息的序号。

R-multicast 一个消息到组g: 捎带$S_g^p$和确认，即<q, $R_g^q$ >; $S_g^p = S_g^p + 1$

R-deliver 一个消息:

1. 当且仅当m.S = $R_g^p$ +1传递消息； $R_g^p = R_g^p + 1$
2. 若m.S <= $R_g^p$, 则该消息已传递，直接丢弃。
3. 若m.S > $R_g^p$ +1或对任意封闭的确认<q, $R_g^q$ >有m.R > $R_g^q$, 则漏掉了一个或多个消息,将消息保留在保留队列中,并发送否定确认。

算法评价

- 完整性

通过检测副本和IP组播性质实现

- 有效性

仅在IP组播具有有效性时成立

- 协定

进程无限组播消息时成立

统一性质

- 统一：无论进程是否正确都成立的性质
- 统一协定：如果一个进程传递消息m，不论该进程是否正确还是出故障，在group(m)中的所有正确的进程终将传递m。

9.4.3 有序组播

**有序组播**

- FIFO排序：如果一个正确的进程发出multcast(g,m)，然后发出multicast(g,m')，那么每个传递m'的正确的进程将在m'前传递m。
- 因果排序：如果multcast(g,m) → multicast(g,m')，那么任何传递m'的正确进程将在m'前传递m。

```
On initialization
   Received := {};
For process p to R-multicast message m to group g
   B-multicast(g, m);          // p ∈ g is included as a destination
On B-deliver(m) at process q with g = group(m)
   if ( m ∉ Received )
   then
            Received := Received ∪ {m};
            if ( q ≠ p ) then B-multicast(g, m); end if
            R-deliver m;
   end if
```

- 全排序：如果一个正确的进程在传递m'前传递消息m，那么其它传递m'的正确进程将在m'前传递m。

实现FIFO排序：基于序号实现

FO-multicast/FO-deliver

算法：与基于IP组播的可靠组播类似，即采用$S_g^p$、$R_g^q$和保留队列
实现全排序：为组播消息指定全排序标识符
TO-multicast/TO-deliver
使用顺序者的全排序算法
1. 组成员p的算法
初始化：rg:=0;
为了给组g发TO-multicast消息：
B-multicast(g∪{sequencer(g)},<m,i>);
在B-deliver(Morder=<m, i>)时，其中g=group(m)
将<m,i>放在保留队列中；
在B-deliver(Morder=<"order",I,s>)时，其中g=group(Morder)
Wait until <m,i>在保留队列中并且S=rg;
To-deliver m; //在从保留队列删除它之后
rg=S+1;
2. 顺序者g的算法
初始化：sg:=0;
在B-deliver(<m,i>)时，其中g=group(Morder)
B-multicast(g,<"orer",I,sg>);
sg:=sg+1;

**全排序的ISIS算法**

$A_g^q$: 进程迄今为止 从组g观察到的最大的协定序号
$P_g^q$: 进程自己提出的最大序号
进程p组播消息m到组g的算法：
1. p B-multicasts <m, i>到g， 其中i 是m的一个委员的标识符
2. 每个进程： (1) $P_g^q = max(A_g^q, P_g^q)+1$; (2)把 $P_g^q$添加到消息 m，并把m放入保留队列; (3)用序号$P_g^q$ 回答p
3. P收集$P_g^q$,选择最大的数a作为下一个协定序号，然后 B-multicast<i,a> 到g
4. g中的每个进程q置$A_g^q$: = max($A_g^q$, a), 并把a附加到消息上。

- 正确的进程最终会对同一组序号达成一致；
- 序号是单调递增的；
- 不保证因果或FIFO序；
- 比基于顺序者的组播有更大的延迟

**实现因果排序**

向量时间戳：每个进程维护自己的向量时间戳
CO-multicast：在向量时间戳的相应分量上加1，附加时间戳到消息
CO-deliver：根据时间戳递交消息
使用向量时间戳的因果排序算法
对组成员$p_i$ (i=1,2,…,N)的算法
初始化：
$V_i^g[j]:=0$ (j=1,2,…M);
为了给组g发CO-multicast消息m:
$V_i^g[i]=V_i^g[i]+1$;
B-multicast(g,<$V_i^g$,m>);
在B-deliver(<$V_i^g$,m>)来自$p_j$(i≠j)的一个消息时，其中g=group(m);
将<$V_i^g$,m>放入保留队列,直到$V_j^g[j]=V_i^g[j]+1$和$V_j^g[k]<=V_i^g[k]+1$ (k ≠j);
CO-deliver m; //在把它从保留队列删除后
$V_i^g[j]=V_i^g[j]+1$;

**组重叠**

全局FIFO排序

- 如果一个正确的进程发出multicast(g,m)，然后发出multicast(g',m')，则两个消息被发送到g∩g'的成员。

全局的因果排序

- 如果multicast(g,m) → multicast(g',m')，则g∩g'中的任何传递m'的正确进程将在m'前传递m。

进程对的全排序

- 如果一个正确的进程在传递发送到g'的消息m'前传递了发送到g的消息m，则g∩g'中的任何传递m'的正确进程将在m'前传递m。

全局的全排序

- 令"<"是传递事件之间的排序关系。要求"<"遵守进程对的全排序，并且无环。

9.5 共识和相关问题
分布式系统中的协定问题
互斥：哪个进程可以进入临界区
全排序组播：组播消息的顺序
拜占庭将军：进攻还是撤退
共识问题

- 一个或多个进程提议了一个值后，应达成一致意见
- 共识问题、拜占庭将军和交互一致性问题

故障模型

- 进程崩溃故障、拜占庭进程故障

9.5.1 系统模型和问题定义
共识问题定义
符号

- $p_i$: 进程i
- $v_i$: 进程$p_i$的提议值
- $d_i$: 进程$p_i$的决定变量

共识算法的基本要求

- 终止性：每个正确的进程最终设置它的决定变量
- 协定性：如果pi和pj是正确的且已进入决定状态，那么di=dj ，其中i,j=1,2,…N。
- 完整性：如果正确的进程都提议了同一个值，那么处于决定状态的任何正确进程已选择了该值。

算法（进程不出现故障的系统）

1. 每个进程组播它的提议值
2. 每个进程收集其它进程的提议值
3. 每个进程计算V = majority ($v_1, v_2, \cdots, v_N$)

majority()函数为抽象函数，可以是max()、min()等等

算法分析

- 终止性

由组播操作的可靠性保证

- 协定性和完整性

由majority()函数定义和可靠组播的完整性保证

拜占庭将军问题

算法要求

- 终止性: 每个正确的进程最终设置它的决定变量
- 协定性: 如果pi和pj是正确的且已进入决定状态，那么di=dj ，其中i,j=1,2,…N
- 完整性: 如果司令是正确的，那么所有正确进程都采用司令的提议

**交互一致性**

就一个值向量达成一致

算法要求

- 终止性

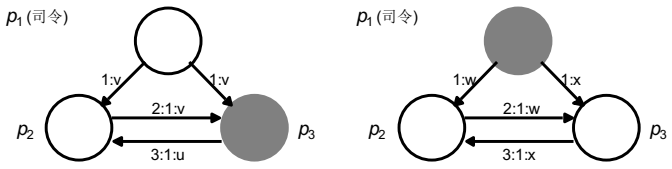
每个正确进程最终设置它的决定变量

- 协定性

所有正确进程的决定向量都相同

- 完整性

如果进程$p_i$是正确的，那么所有正确的进程都把$v_i$作为他们决定向量中的第i个分量。




有故障的进程用灰色表示

目的：重用已有的解决方案

问题定义

共识问题C：$C_i(v_1,v_2,\cdots v_N)$: 返回进程$p_i$的决定值

拜占庭将军BG：$BG_i(j,v)$: 返回进程$p_i$的决定值，其中$p_j$是司令,它建议的值是v

交互一致性问题IC：$IC_i(v_1,v_2,\cdots,v_N)[j]$: 返回进程$p_i$的决定向量的第j个分量

从BG构造IC

- 将BG算法运算N次,每次都以不同的进程pi作为司令
- ICi(v1,v2, …, vN )[j] = BGi(j,v), (i, j = 1, 2, …, N)

从IC构造C

- Ci(v1,v2, …, vN ) = majority(ICi(v1,v2, …, vN )[1], …, ICi(v1,v2, …, vN )[N])

从C构造BG

- 司令进程pj把它提议的值v发送给它自己以及其余进程
- 所有的进程都用它们收到的那组值v1,v2, …, vN作为参数运行C算法
- BGi(j,v) =Ci(v1,v2, …, vN ), (i = 1, 2, …, N)

同步系统中的共识问题

故障假设：N个进程中最多有f个进程会出现崩溃故障

算法

对$p_i \in g$的进程算法: 算法进行到f+1轮

初始化:

$values_i^1:=\{v_i\}$; $Values_i^0=\{\}$;

在第r轮($1 \leqslant r \leqslant f+1$)

B-multicast(g,$Values_i^r - Values_i^{r-1}$); //仅发送还没有发送的值

$Values_i^{r+1}:=Values_i^r$;

While(在第r轮) {

在B-deliver($V_j$)来自$p_j$的消息时:

$Values_i^{r+1}:=Values_i^{r+1} \cup V_j$;

}

在(f+1)轮之后

将d赋成min($Values_i^{f+1}$);

- 终止性质

由同步系统保证

- 协定性和完整性

假设$p_i$得到的值是v，而 $p_j$不是

$P_{k1}$在把v传送给$p_i$后,还没来得及传送给 $p_j$就崩溃了

$P_{k2}$在把v传送给$p_i$后,还没来得及传送给 $p_j$就崩溃了

$P_{k(f+1)}$在把v传送给$p_i$后,还没来得及传送给 $p_j$就崩溃了

但我们假设至多有f个进程崩溃

因此，$p_i$和$p_j$的值相同

因此，min($Values_i^{f+2}$)相同

9.5.4 异步系统的不可能性

**没有算法能够保证达到共识**

无法分辨一个进程是速度很慢还是已经崩溃

故障屏蔽

屏蔽发生的所有进程故障

使用故障检测器达到共识

在仅依靠消息传递的异步系统中，不存在完美的故障检测器，甚至是最弱的故障检测器。

使用随机化达到共识

引入一个关于进程行为的可能性元素。

10.1 简介

事务的目标：在多个事务访问对象以及服务器面临故障的情况下，保证所有由服务器管理的对象始终保持一个一致的状态。

**并发控制**

**增强可靠性**

- 可恢复对象
- 利用持久存储保存状态信息

事务是由客户定义的针对服务器对象的一组操作，它们组成一个不可分割的单元，由服务器执行。

**事务的故障模型——Lampson模型**

对持久性存储的写操作可能发生故障

写操作无效或写入错误的值

文件存储可能损坏

读数据时可根据校验和发现损坏数据

服务器可能偶尔崩溃

进程崩溃后，根据持久存储中的信息恢复

服务器不会产生随机故障

消息传递可能由任意长时间的延迟

消息可丢失、重复或者损坏

接收方能够检测受损消息

10.2 事务

事务的概念

以原子方式执行的一系列操作，即

- 它们不受其它并发客户操作的干扰
- 所有操作或者全部成功完成，或者不产生任何影响

全有或全无：或者完全成功，或者不留下任何效果

**故障原子性**

即使服务器崩溃，事务的效果也是原子的。

**持久性**

一旦事务完成，它的所有效果将被保存到持久存储中

**隔离性**

每个事务的影响不受其它事务的影响

**ACID特性**

**Atomicity 原子性**

**Consistency 一致性**

**Isolation 隔离性**

**Durability 持久性**

事务的完成通常需要一个客户程序、若干可恢复对象和一个协调者之间的合作

事务执行结果

- 完全成功
- 放弃事务

并发控制错误：

**更新丢失问题**

**不一致检索**

**串行等价性**

多事务正确运行效果推断

若每个事务知道它单独执行的正确效果，则可以推断出这些事务按某种次序一次执行一个事务的效果。

串行等价的交错执行

并发事务交错执行操作的效果等同于按某种次序一次执行一个事务的效果。

使用串行等价性作为并发执行的判断标准，可防止更新丢失和不一致检索问题。

**冲突操作**

串行等价性

两个事务串行等价的充要条件是，两个事务中所有的冲突操作都按相同的次序在它们访问的对象上执行。

**并发控制协议**

依据：串行等价性

目的：将访问对象的并发事务串行化

方法

- 锁
- 乐观并发控制
- 时间戳排序

**脏数据读取**

某个事务读取了另一个未提交事务写入的数据

事务可恢复性

策略：

推迟事务提交，直到它读取更新结果的其它事务都已提交。

连锁放弃，某个事务的放弃可能导致后续更多事务的放弃

防止方法：只允许事务读取已提交事务写入的对象

**过早写入**

一些数据库在放弃事务时，将变量的值恢复到该事务所有write操作的“前映像”。

为了保证使用前映像进行事务恢复时获得正确的结果，write操作必须等到前面修改同一对象的其它事务提交或放弃后才能进行。

**事务的严格执行**

严格执行：read和write操作都推迟到写同一对象的其它事务提交或放弃后进行

可以真正保证事务的隔离性

**临时版本**

目的：事务放弃后，能够清除所有对象的更新

方法：

- 事务的所有操作更新将值存储在自己的临时版本中
- 事务提交时，临时版本的数据才会用来更新对象

10.3 嵌套事务
概念
嵌套事务：允许事务由其它事物构成
顶层事务（Top-Level）
子事务(Sub-transaction)
优点
- 并发度高：子事务可并发运行
- 健壮性强：子事务可以独立提交和放弃
提交规则
事务在它的子事务完成以后，才能提交或放弃
子事务完成后，可独立决定是暂时提交或放弃
父事务放弃时，所有的子事务都被放弃
若子事务放弃，则父事务可以决定是否放弃
若顶层事务提交，则所有暂时提交的事务将最终提交
10.4 锁
**互斥锁是一种简单的事务串行化实现机制**
事务访问对象前请求加锁
若对象已被其它事务锁住，则请求被挂起，直至对象被解锁
**两阶段加锁**（two-phrase locking）
目的：保证两个事务的所有的冲突操作对必须以相同的次序执行
增长阶段：不断获取新锁
收缩阶段：释放锁
**严格的两阶段加锁**(strict two-phrase locking)
目的：防止事务放弃导致的脏数据读取、过早写入等问题
方法：所有在事务执行过程中获取的新锁必须在事务提交或放弃后才能释放
**读锁和写锁**
目的：提高并发度
支持多个并发事务同时读取某个对象
允许一个事务写对象
**事务的操作冲突规则**
如果事务T已经对某个对象进行了读操作，那么并发事务U在事务T提交或放弃前不能写该对象。
如果事务T已经对某个对象进行了写操作，那么并发事务U在事务T提交或放弃前不能写或读该对象。
锁的实现
锁的授予通常由服务器上一个对象实现—锁管理器
每个锁都是Lock类的一个实例
- 被锁住对象的标示符
- 当前拥有该锁的事务的标示符
- 锁的类型
嵌套事务的加锁需求
嵌套事务集
要求：不能观察到其它嵌套事务集的部分效果
实现方法：父事务继承子事务的所有锁，锁的继承从底层向高层传递。
嵌套事务集中的事务
要求：不能观察到同一事务集中其它事务的部分效果
实现方法：父事务不允许和子事务并发运行；同层次的事务可并发执行。
获得读锁
如果子事务获取了某个对象的读锁，那么其它活动事务不能获取该对象的写锁，只有该子事务的父事务们可以持有该写锁。
获得写锁
如果子事务获取了某个对象的写锁，那么其它活动事务不能获取该对象的写锁或读锁，只有该子事务的父事务们可以持有该写锁或读锁。
提交
子事务提交时，它的所有锁由父事务继承，即允许父事务保留与子事务相同模式的锁。
放弃
子事务放弃时，它的所有锁都被丢弃。如果父事务已经保留了这些锁，那么它可以继续保持这些锁。
死锁：两个事务都在等待并且只有对方释放锁后才能继续执行
死锁是一种状态，在该状态下一组事务中的每一个事务都在等待其它事务释放某个锁。
等待图（wait-for graph）表示事务之间的等待关系。
**预防死锁**
每个事务在开始运行时锁住它要访问的所有对象
- 一个简单的原子操作
- 不必要的资源访问限制
- 无法预计将要访问的对象
预定次序加锁
- 过早加锁
- 减少并发度
更新锁
- 避免死锁
- 在数据项上加更新锁的事务可以读该数据项，但该锁与加在同一数据项上的更新相冲突。
**死锁检测**
维护等待图
检测等待图中是否存在环路，若存在环路，则选择放弃一个事务
锁超时：解除死锁最常用的方法之一
每个锁都有一个时间期限，超过时间期限的锁成为可剥夺锁，若存在等待可剥夺锁保护的对象，则对象解锁
在加锁机制中增加并发度
- 双版本加锁

- 层次锁

**双版本加锁**

读/写操作

- 写操作对象可为临时版本
- 读操作对象为提交版本

读锁、写锁和提交锁

- 读锁：在读操作前为对象设置读锁
- 写锁：在写操作前为对象设置写锁
- 提交锁：收到提交事务请求后，将写锁转换为提交锁

**层次锁**

混合粒度的层次锁

每个事务按需要锁住部分数据

锁的拥有者能显式访问该节点并隐式访问它的子节点

给子节点加锁时，需要在父节点和祖先节点设置试图锁

在每一层，设置父锁与设置等价的子辈锁具有相同的效果

10.5 乐观并发控制

锁机制的缺点

- 维护开销大
- 会引起死锁
- 并发度低

**乐观策略**

基于事实：在大多数应用中，两个客户事务访问同一个对象的可能性很低。

方法

- 访问对象时不作检查操作
- 事务提交时检测冲突
- 若存在冲突，则放弃一些事务

事务的三个阶段

工作阶段

- 每个事务拥有所修改对象的临时版本
- 每个事务维护访问对象的两个集合：读集合和写集合

验证阶段

- 在收到closeTransaction请求，判断是否与其它事务存在冲突。

更新阶段

- 提交通过验证的事务

事务的验证

事务号

- 每个事务在进入验证阶段前被赋予一个事务号
- 事务号是整数，并按升序分配
- 事务按事务号顺序进入验证阶段
- 事务按事务号提交

冲突规则

- 事务Tv的验证测试
- Ti和Tv之间的存在冲突

事务$T_v$对事务$T_i$而言是可串行化的：

**向后验证**

检查它的读集是否和其它较早重叠事务的写集是否重叠

算法：分配就是更新结束

startTn: $T_v$进入工作阶段时已分配的最大事务号码

finishTn: $T_v$进入验证阶段时已分配的最大事务号码

```
Boolean valid = true
For ( int Ti = startTn +1; Ti <= finishTn; Ti ++) {
        if (read set of Tv intersects write set of Ti)
            valid = false
}
```

验证失败后，冲突解决方法：放弃当前进行验证的事务

**向前验证**

比较Tv的写集合和所有重叠的活动事务的读集合

算法

设活动事务具有连续的事务标示符$active_1$~$active_N$

验证失败后，冲突解决方法

放弃当前进行验证事务

推迟验证

放弃所有冲突的活动事务，提交已验证事务

$active_1$、 $active_2$是较$T_v$晚开始的事务

向前验证和向后验证的比较

- 向前验证在处理冲突时比较灵活
- 向后验证将较大的读集合和较早事务的写集合进行比较
- 向前验证将较小的写集合和活动事务的读集合进行比较
- 向后验证需要存储已提交事务的写集合
- 向前验证不得不允许在验证过程中开始新事务

**饥饿**

由于冲突，某个事务被反复放弃，阻止它最终提交的现象。

利用信号量，实现资源的互斥访问，避免事务饥饿

10.6 时间戳排序

时间戳

- 每个事务在启动时被赋予一个唯一的时间戳
- 时间戳定义了该事务在事务时间序列中的位置
- 不会引起死锁

冲突规则

- 写请求有效：对象的最后一次读访问或写访问由一个较早的事务执行
- 读请求有效：对象的最后一次写访问由一个较早的事物执行

**基于时间戳的并发控制**
临时版本
- 写操作记录在对象的临时版本中
- 临时版本中的写操作对其它事务不可见

写时间戳和读时间戳
- 已提交对象的写时间戳比所有临时版本都要早
- 读时间戳集用集合中的最大值来代表
- 事务的读操作作用于时间戳小于该事务时间戳的最大写时间戳的对象版本上

时间戳排序的写规则
是否接受事务$T_c$对对象D执行的写操作
if (Tc ⩾ D的最大读时间戳 && Tc > D的提交版本上的写时间戳)
在D的临时版本上执行写操作，写时间戳置为 Tc
else /* 写操作太晚了 */
放弃事务 Tc
时间戳排序的读规则
是否接受事务$T_c$对对象D执行的读操作
if (Tc ⩾ D提交版本的写时间戳) {
设Dselected是D的具有最大写时间戳的版本⩽Tc;
if (Dselected已提交)
在Dselected版本上完成读操作
else
等待直到形成版本的事务提交或放弃，然后重新应用读规则;
} else
放弃事务 Tc

10.7 并发控制方法的比较
时间戳排序
- 静态地决定事务之间的串行顺序
- 对读操作占优的事务而言，优于两阶段加锁机制
- 冲突规则

两阶段加锁
- 动态决定事务之间的串行顺序
- 对更新操作占优的事务而言，优于时间戳排序

时间戳排序和两阶段加锁均属采用悲观方法
**乐观方法**
- 并发事务之间的冲突较少时，性能较高
- 放弃事务时，需要重复大量工作

**悲观方法**
- 简单
- 并发度低

11.1 简介
资源共享是分布式系统一个重要特征
共享存储信息可能是分布式资源共享的一个重要方面
Web服务器提供了一种严格的数据共享方式
局域网和企业内部网中的共享
为客户端提供各种类型的程序和数据的持久存储
基本---- 分布式文件系统的主要目的是在多个远程计算机系统上模拟非分布式文件系统的功能
不支持文件多持久副本，不提供带宽和实时保证
11.1.1 文件系统的特点
文件系统
文件的组织、存储、检索、命名、共享和保护
11.1.2 Linux 文件系统
文件系统的最终目的是把大量数据有组织的放入持久性(persistant)的存储设备中
11.1.3 分布式文件系统的需求
**透明性**
- 访问

客户程序不应了解文件的分布性
**位置**
客户程序应使用单一的文件命名空间
**移动**
文件移动时，客户程序和客户结点上的系统管理表都不必进行修改
**性能**
服务负载在一个特定范围内变化时，客户程序性能可以得到满意的性能
**伸缩**
文件服务可以不断扩充
11.1.3 分布式文件系统的需求
并发文件更新、文件复制、硬件和操作系统异构性
容错、一致性、安全性、效率
分布式文件系统要在一致性、性能、可扩展性上进行折中
用户端缓存可以提高性能和可扩展性的工具
针对需求进行设计
■何谓“大型”网站?
大型网站架构的目标与挑战
■何谓“大型”网站?
大型网站架构的目标与挑战
■网站架构目标与挑战
DNS负载均衡
简单
缺少灵活性（DNS缓存）
反向代理负载均衡
负载均衡软件/硬件

13.1 为什么使用NoSQL

**NoSQL主要特征**

- 不使用SQL
- 通常是开源项目
- 大多数是为了在集群环境运行开发
- 数据模型与关系型数据库不同
- 不需要使用“模式”

13.2 数据模型

数据模型----数据库组织数据的方式

元模型（metamodel）

关系型数据库

每张表包含若干行（元组）

每行包含相关实体，实体通过列来描述，行列交汇处有单一值

NoSQL抛弃了原有的关系模型

**四大类：键值、文档、列族、图**

13.2 数据模型

聚合

- 一组相关联的对象视为一个整体单元进行操作
- 与数据存储通信时也以聚合为单位
- 程序员通过聚合结构来操作数据

聚合的边界一般都很难划分

可以按照不同的方式来查看数据

聚合模型适用于集群环境

集群环境运行时，需要把采集数据时所需的节点数降至最小

**键值数据库**

- 包含大量聚合
- 每个聚合中包含获取数据所用的key或ID
- 键值数据库聚合不透明
- 通过键值来搜索聚合内容

**文档数据库**

- 包含大量聚合
- 每个聚合中包含获取数据所用的key或ID
- 定义了其允许的结构和数据类型
- 查询词可以基于文档内部结构

**列族数据库**

- 带有稀疏列的无模式表结构
- “两级映射”（two-level map）

大部分数据库以行为单位存储数据

如果写入操作执行的少，需要一次读取若干行中的很多列

将所有行的某一组列（列族）作为基本数据存储单元效果很更好

可以列族模型其视为两级聚合结构（two-level aggregate structure）

第一个键通常代表行标识符，可以用它来获取想要的聚合

“行聚合”（row aggregate）本身又是一个映射，其中包含一些更为详细的值

这些“二级值”（second-level value）就叫做“列”

可以操作特定的列

get(‘1234’,’name’)

列族数据库将列组织为列族，每一列必须是列族的一部分

两种数据组织方式

面向行（row-oriented）：每一行都是一个聚合（例如ID为1234的顾客就是一个聚合），该聚合内部存有一些包含有用数据块（客户信息、订单记录）的列族。

面向列（column-oriented）：每个列族都定义了一种记录类型（例如客户信息），其中每行都表示一条记录。可以将数据库中的大“行”理解为列族中每一个短行记录的串接。

13.2 数据模型

数据库通过这些数据分组方式，可在存储及访问时利用此信息。

即使文档数据库声明了某种结构，每个文档也依然被视作独立单元。

“列族”体现了“列族数据库”二维映射这一特点。

关系

聚合的有用之处在于可以把经常访问的数据存放在一起

大部分数据库都提供描述这种关系的手段

面向聚合数据库获取数据时以聚合为单元

只保证单一聚合内部内容的原子性，如果一次更新多个聚合，需要应对中途发生的错误

**图数据库（Graph Database）**

处理相互间关系复杂的一小组记录

基本数据模型：边连接而成的若干节点

属性

针对图专门设计的查询操作

大部分操作是沿着网络的边来浏览数据库

**无模式数据库**

在关系数据中存储数据要先定义模式

哪些表格，哪些列

NoSQL数据库的数据存储就比较随意

- 键值数据库可以把任何数据存放在一个“键”名下
- 文档数据库对文档结构没有限制
- 列族数据库任意列都可以随意存放数据
- 图数据库中可以新增边，向节点和边添加属性

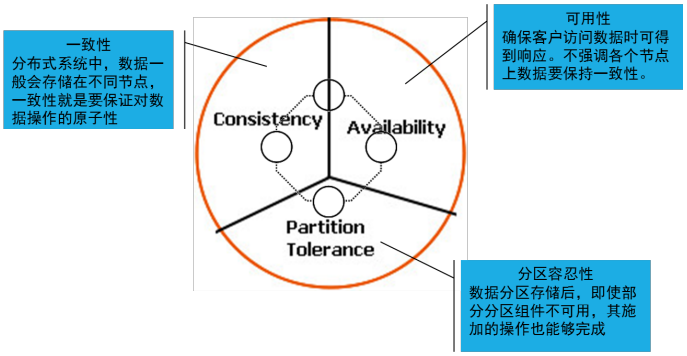
**分布式是NoSQL的主要特点**

数据分布两条途径

- 复制 replication
- 分片 sharding

复制与分片是“正交的”技术

**分片**
不同用户需要访问数据集中在不同部分
理想情况下不同服务器节点服务于不同用户
每位用户只需与一台服务器通信
要达到理想情况，必须保证需要同时访问的那些数据存在同一节点
负载均衡
“自动分片”Auto-sharding
分片对于提升性能尤其有用，同时提升读取和写入效率，对于改善“故障恢复能力”帮助并不大
**主从复制**
在“主从式分布”中，我们要把数据复制到多个节点上，一个节点是主节点，其余节点是从节点，复制操作就是从主节点到从节点同步
主从复制
优点
- 需要频繁读取数据集的情况下，主从复制有助于




提升访问性能
- 可以增强“读取操作的故障恢复能力”

缺点
- 数据的不一致性

**对等复制**
- 所有节点地位相同，都可以接受写入请求
- 丢失一个副本不影响整个数据库访问

对等复制
数据的不一致问题
两个节点可以同时处理写入操作，造成写入冲突
- 协调机制协调写入操作
- 相互冲突的写入操作合并

**更新一致性**
写冲突：两人在同一时刻更新同一条数据
服务器收到写入请求后，将其“序列化”，产生更新丢失问题
乐观方法
先让冲突发生，再检测冲突并对发生冲突的操作排序
版本控制
悲观方法
加锁
**读取一致性**
更够保持更新一致性的数据库，未必能保证用户所提交的访问请求总能得到内容一致的响应
**逻辑一致性**
不同数据项放在一起，其含义符合逻辑
为了避免读写冲突，关系型数据库支持“事务”
面向聚合的数据库支持原子更新，但仅限于单一聚合内部
在执行影响多个聚合的更新操作时，会留下一段时间空档，存在不一致风险的时间长度就叫“不一致窗口”inconsistency window
**复制一致性**
要求从不同副本中读取的同一个数据项时，所得到值相同
**最终一致性**
最终更新还是会传播到全部节点
存在“不一致窗口”，不同人在同一时刻可能会看到不同的数据
照原样读出所写内容的一致性
执行完更新操作后，紧接着必须能看到更新之后的值
**放宽“一致性”约束**
单服务器关系型数据库，通过事务加强一致性
性能影响太大
CAP定理

放宽“持久性”约束
如果数据大部分时间在内存中运行，更新操作直接写入内存
定期将数据变更写回磁盘
大大提高响应请求的速度
一旦服务器发生故障，未写会磁盘的更新数据会丢失