BDIC3005J Distributed Systems

Chap1. Basics of Distributed Systems

Definition of Distributed Systems:

Definition 1:

Distributed system is a collection of independent computers that appears to its users as a single coherent system:

From a hardware perspective: Computers are autonomous

From a software perspective: Users think they use a single coherent system

Definition 2:

Any system that consists of hardware or software components located at networked computers that communicate and coordinate their actions via message passing.

Characteristics of Distributed Systems:

1. **Hidden**: Hide the difference and the communication among all kinds of computers.
2. **Consistent and Uniform**: No matter where and when a user interacts with a distributed system, they use a consistent and uniform method.
3. **Scalability**: even if some part of the whole system is not working, the system should be available all the time without user noticing components are replaced or updated: this is called “Independent Failures”.
4. **Concurrency**: Coordination among different concurrently executing programs that share resources.
5. **No Global Clock**: Difficult to synchronize all clocks in a network, but the sequence between actions is important, therefore message passing is used.

Distributed system和Centralized system的区别：

Centralized system是一个computer+很多peripheral devices & remote terminals

Distributed System是很多个computer interconnected by high-speed network

Goals:

1. **Making resources accessible**: For economic reasons, Collaborate and exchange information and for security problems(Eavesdropping or Communication intrusion)

2. **Distribution Transparency**: 分布式系统对于用户应当是不可见的; 但是并不是所有时候都可以隐藏这个属性, 比如delay会存在，并且不总是好的: 在transparency和system performance之间有trade-off, 比如维持多份服务器之间的consistency

3. **Openness**: The service provided by the system is according to standard, which describes the syntax and semantics of the services(由于有标准定义，很容易达成一致，别人也更容易使用，因此open)---IDL(Interface Definition Language), 所有的service provided by DS都由interface定义, 由IDL定义interprocess communication. Flexibility: 所有人在一个DS的deployment中可以各司其职，只需要定义一个好的interface; separate policy from mechanism(需求和实现分离).

4. **Scalability**: Scalable in different aspects: Size, Geography and Administration

但是从这几方面的Scalability也会带来问题：

从size上讲尽管更加容易增删user resource了，但是没有一台机器有整个系统的完整信息；所有的decision只能利用local information做出；并且一般不认为有global clock的存在.

从Geography上讲user和resource可能离得很远；使用WAN的communication是unreliable的；Distributed中有很多的centralized components；Synchronized communication should be block due to performance reasons.

从Administration上讲，不同的units有不同的policy，这之间会产生conflicts，如何administrate所有的components？

使用的Scaling Techniques：

Hiding communication latency：使用asynchronous communication；让client进行一些计算

Distribution：将一个component分成很多个小块，分布在整个系统中. E.g. DNS

Replicate a component across a Distributed System：目的是增加availability，通过load balancing实现performance，并且隐藏WAN中的Latency. E.g. Caching(一种special form)，但是如何维持不同的replicas之间的consistency又会成为新的问题

Hardware Definition:

Homogeneous Multicomputer System:

The processor’s type, the memory size, I/O band width, and the interconnection form of the network are all the same

Heterogeneous Multicomputer System:

Include different and independent computers

Can be connected by different network connections

No global system view

Need complex software

Software Definition:

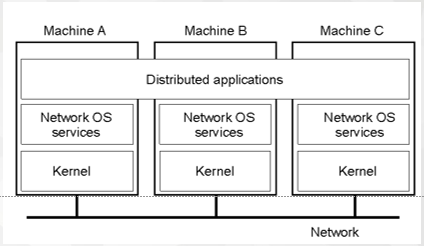
Distributed Operating System:

紧密耦合的系统：try to maintain a single, global view of the resources it manages

在homogeneous multicomputer system中使用

Network Operating System:

松耦合的系统：一组OS，每个computer都有不同的OS；computers和OS是不同的，通过network environment相连接；提供tools以获取独特的service of one computer, 比如一台电脑专门提供File Server，另一台提供



在heterogeneous multicomputer system中使用

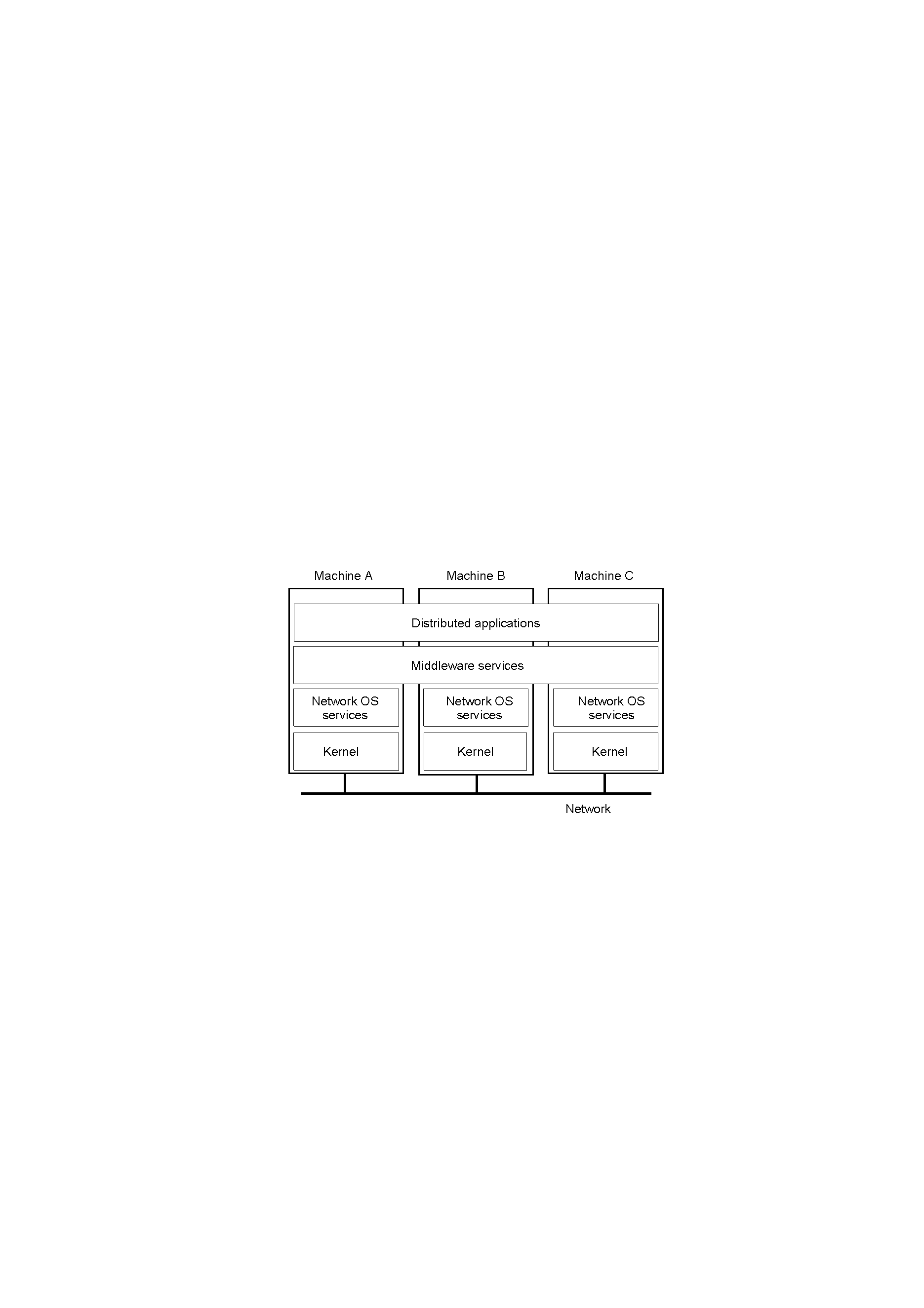
Middleware:

松耦合的系统

在heterogeneous multicomputer system中使用

Tries to hide the heterogeneity of underlying platform and improves distributed transparency

不基于network environment而是基于Middleware services



System models:

Physical models: Hardware

Hardware composition of a system in terms of the computers (and other devices) and their interconnecting networks.

Architectural models: Communication entities

Processes and threads

Objects: 通过interface实现的methods access（通过IDL统一）

Components：多个Object在一起可以组合成Component，problem-Oriented，通过interface 实现access；可以由不同的第三方开发；不同的components之间没有dependencies，Promotes a purer compositional approach to constructing distributed systems

Web Services：integrated into WWW，利用web standards represent and discover services. 通过定义的interface（通过XML）实现software applications，通过URI标识

Architectural elements：

1. Communication Paradigms：

Interprocess communication：使用message-passing primitives，使用网络服务提供的API，为Multicast提供support

Remote Invocation：Request-reply protocols，通过Remote procedure calls或者Remote method invocation实现

Indirect Communication：Group Communication，Publish-Subscribe Systems，Message Queues，Distributed Shared Memory

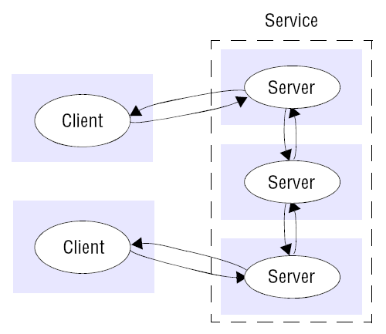
1. Roles and Responsibilities：

Client-Server：每个client process和一个individual server process进行interact，server不一定是server，也可以是别的server的client

Peer-to-Peer：在P2P中所有的process的role都一样，每个peer有一小部分data，互相之间传输

1. Placement：

Mapping of service to multiple servers：



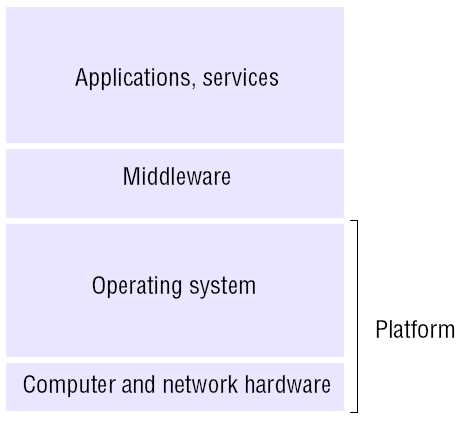
Caching：使用Proxy server

Mobile code：client从webserver上获得Applet code，之后Client只需要和Applet Interact

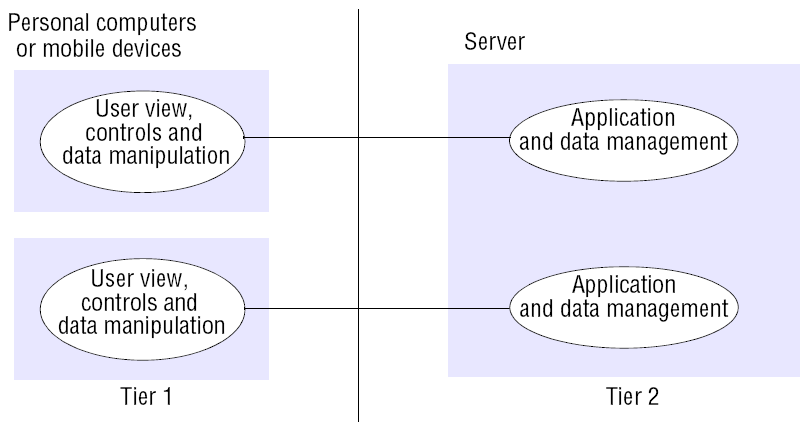
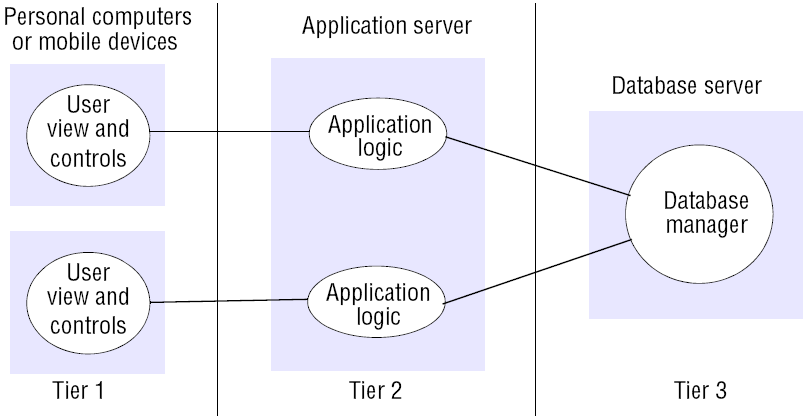
Mobile Agents：一个program，Travels from one computer to another in a network carrying out a task on someone’s behalf，collecting information, and eventually returning with the results.

Architectural Patterns：

Layering Architecture：Client process需要维护一个system-wide view of the service‘s resources



Tiered Architecture：

Thin Clients：

轻量化的clients，将复杂度从client移动到server，使用network访问复杂的servers

Moving complexity away from the end-user device towards services in the Internet

Can access to sophisticated networked services on a remote computer with few assumptions or demands on the client device.

Fundamental Models：

Interaction model:

主要影响interacting process的主要因素有两个：

Performance of communication channels & no global notion of time

1. Latency: delay between the start of a message’s transmission & the start of its receipt.
2. Bandwidth: the total amount of message that can be transmitted over a channel at the same time.
3. Jitter: 由于congestion造成的delay的上下抖动.

2 variants of the interaction model:

Synchronous distributed systems: process的每一个step的处理时间有已知的上下限；通过信道传输的message都有一个已知的时间限制；每个process都有一个local clock，local clock和真实时间的差值有一个已知的上下限

Asynchronous distributed systems:

对于上面的这三条都没有限制的bounded time

Failure: how to deal with failure

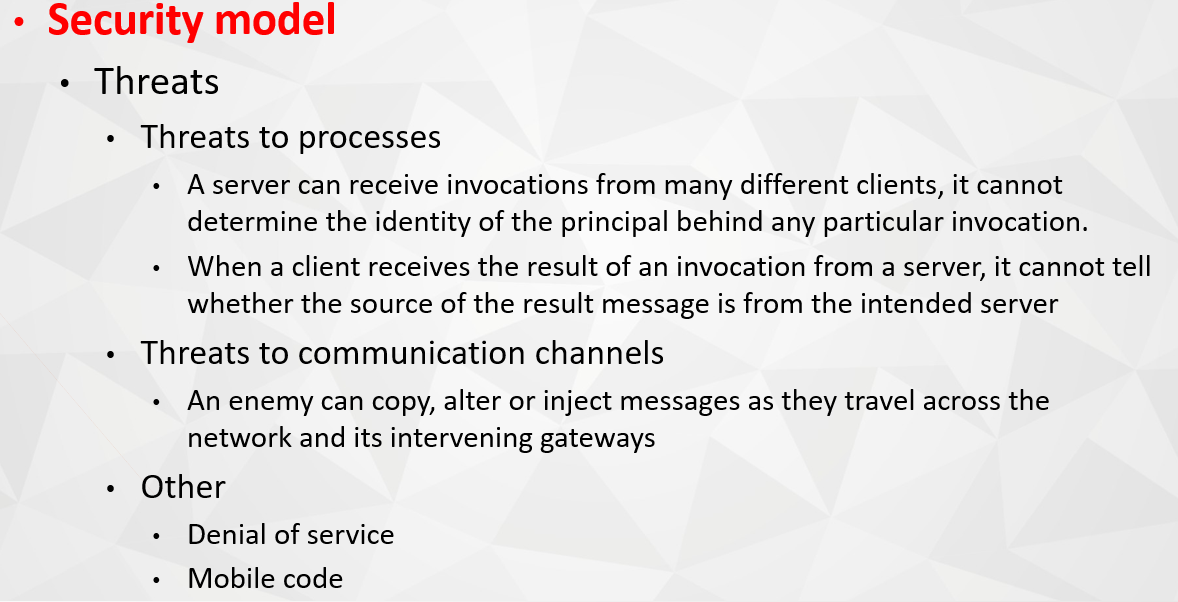
1. Omission failure: process or communication channel failure

Process omission failure: crash---在synchronous DS中可以通过timeout机制检测，但是在Asynchronous DS中很难检测

Communication omission failure: Dropping message

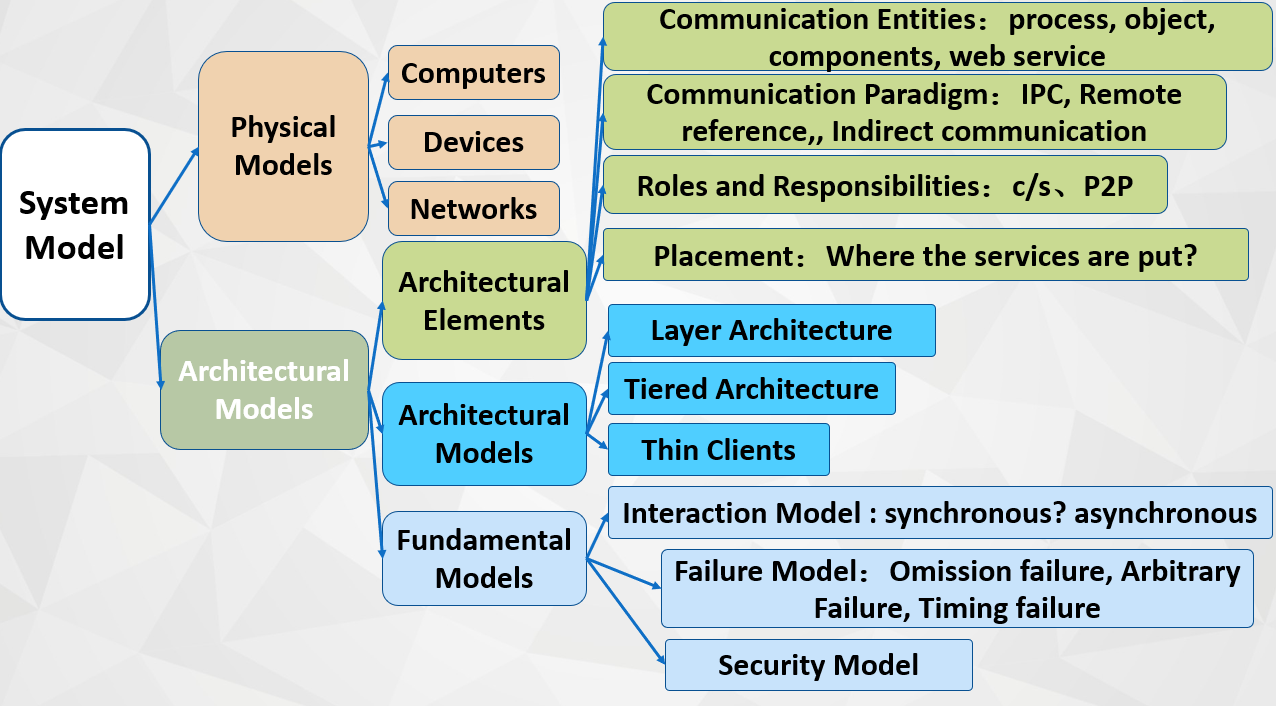
1. Arbitrary failure: 任意一种error
2. Timing failure: the step of process time bound/process local clock time drift/message transmission time这三种超出time bound都是timing failure

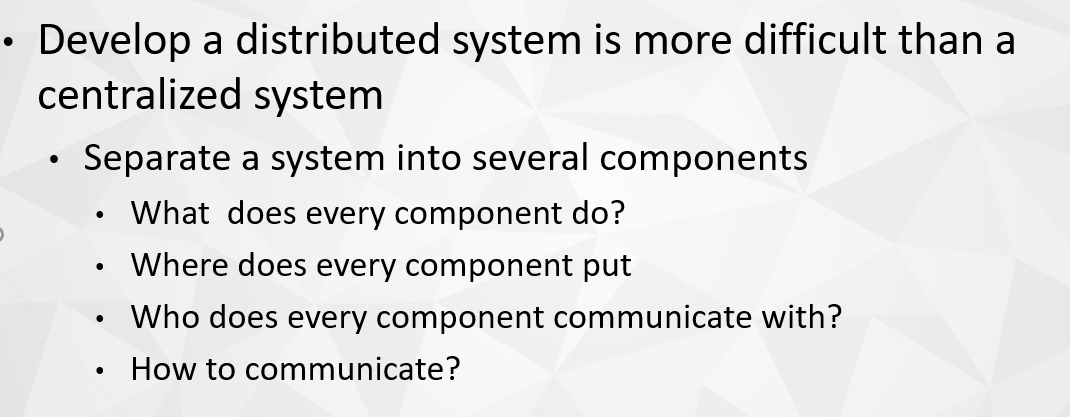
处理failure：通过replicas of server掩盖(mask)发生的failure；对于omission failure---再发一次



如何deal with threat：使用cryptography and shared secrets(私钥)或者authentication

Secured channels(VPN & SSL)：双方的身份对于对方都是已知的，这个channel可以保证传输的信息的privacy and integrity，利用time stamp来prevent replayed/reordered message

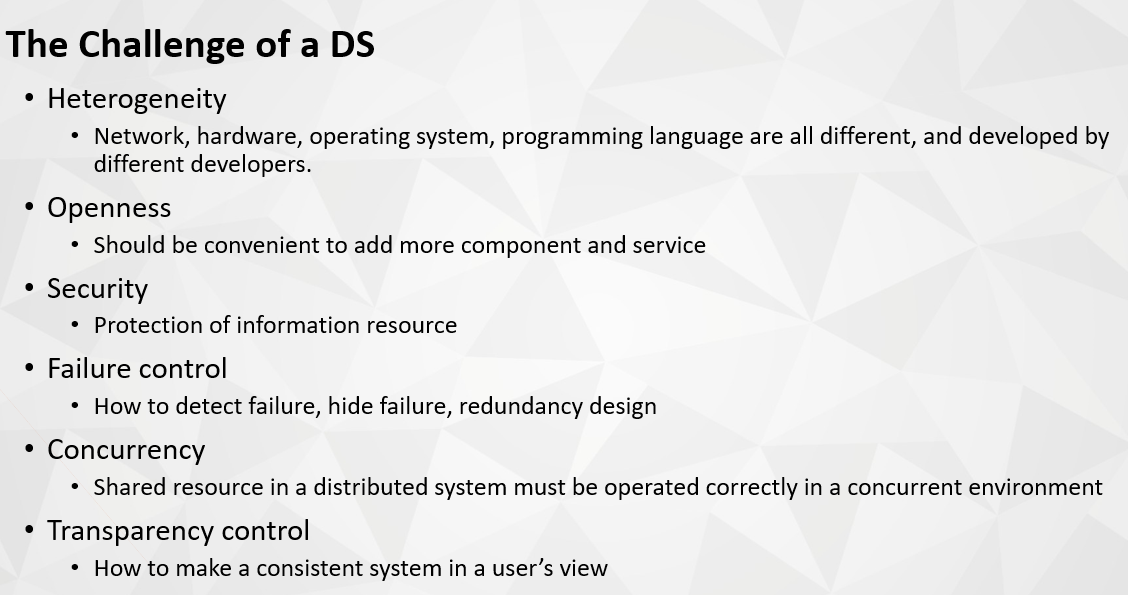




Distributed Systems的优缺点:

Advantages: Economy: processor的价格比mainframe要便宜很多; Speed: total computation capability stronger than mainframe; Distribution: some apps have distribution attributes; Reliability: avoid single point failures.

Disadvantages: new task for system design; Network connection is important; Security and privacy; 需要更多的开发经验(middleware/Os/programming/…)

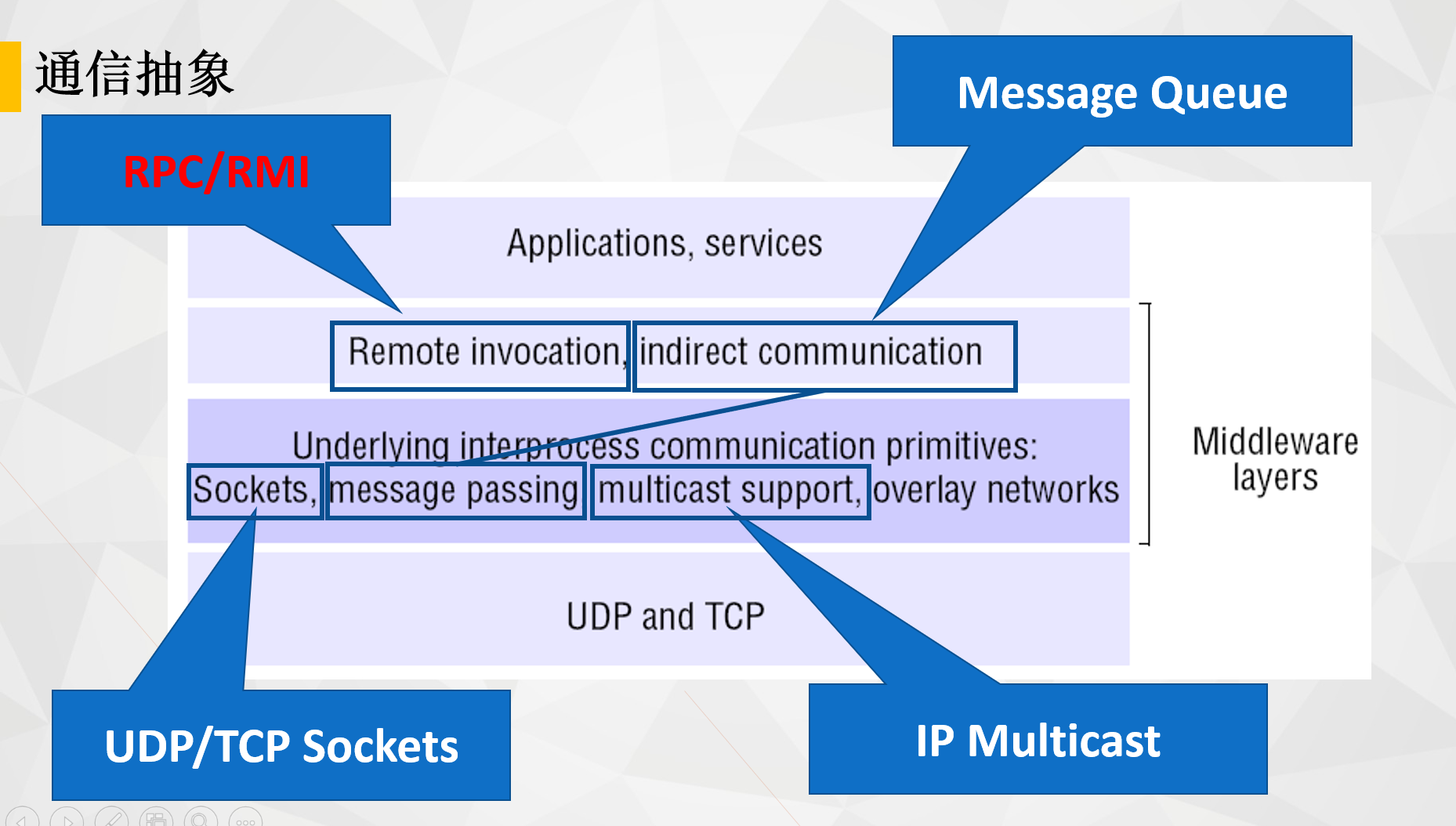


Chap2.

Network and Networking:

Network protocols---OSI protocol stack and TCP/IP protocol stack

在application layer和transport layer之间的有另外两层，第一层包括remote invocation和indirect communication；另一层描述了underlying interprocess communication primitives: Sockets, message passing, multicast, overlay networks. 这两层被称为Middleware Layer



RPC: Remote procedure call(RPC) & parameter passing …

RPC Basics: Description of RPC---机器A上的process X calls机器B上的process Y，parameters被从X传到Y并传回结果，而programmer并不知道message passing的存在

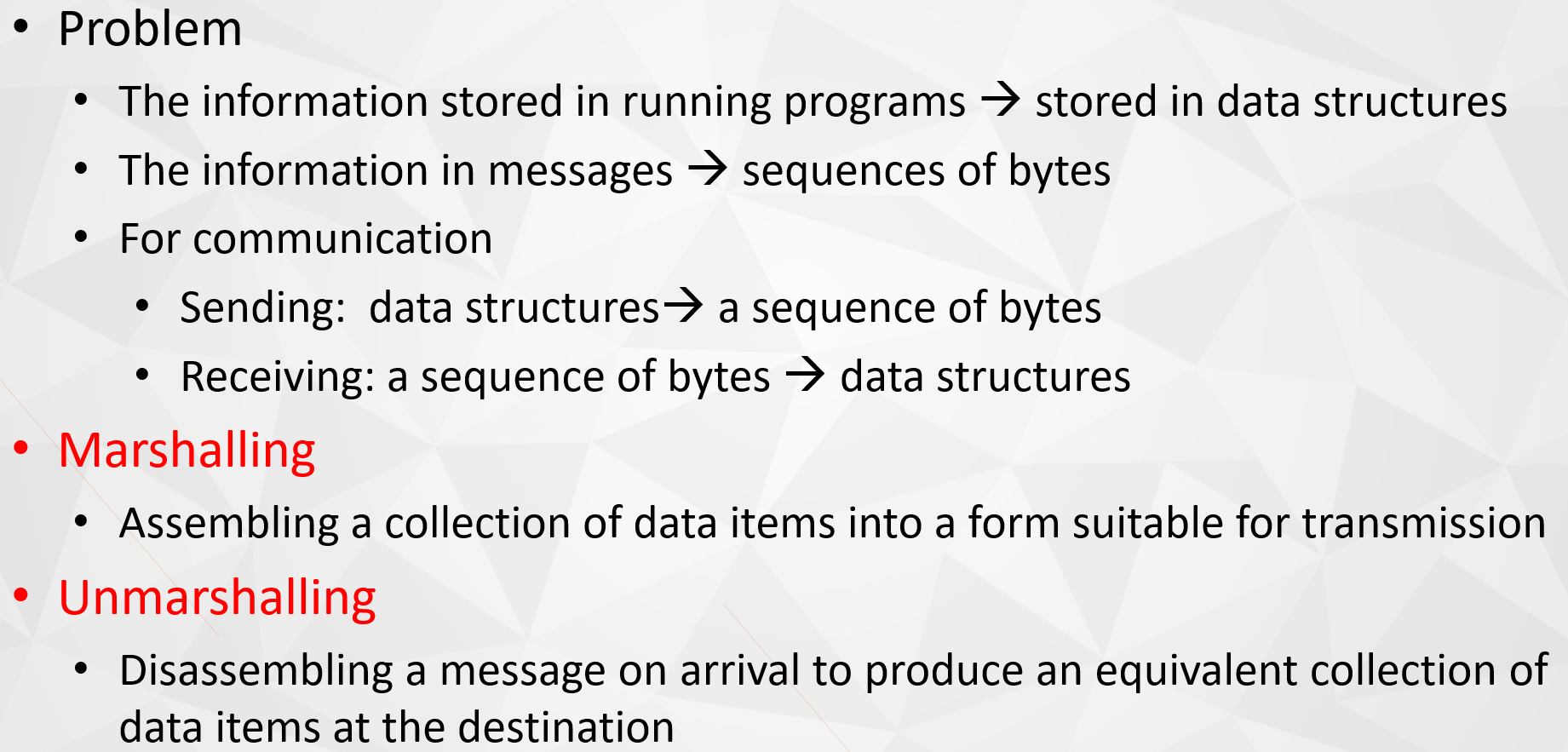
Complexity：caller和callee在不同的address space运行; Problems about message passing; Machine Failure

Transparency：RPC和LPC(Local Procedure Call)有相同的format---相同的interface

Client Stub and Server Stub：



Parameter Passing的问题：



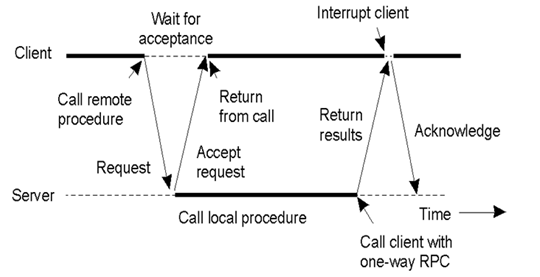
Traditional Parameter Passing: Call by value & Call by reference(传指针)

Problem1 of pass by value: 编码不同--Character: ASCII/EBCDIC; Integer: Endianness

Problem2 of pass by reference: 传的指针是没有用的，因为address space不同。解决方法：对于数组---copy & restore，Omit a copy process if stub知道array是input parameter可以实现Optimization（restore指的是将数组再从server传回client的过程，如果知道这个array只是input parameter的话就不需要传回client了）；对于传统的指针---发送pointer给server后，server会创建一个specific code用来call back client for data

Problem3 of how to create stub: 在同一个protocol下，所有的stubs唯一的不同就是interface，可以通过IDL实现解决

Asynchronous RPC：传统的RPC是synchronous的，会block直到收到return value---如果没有return值，那么这就是不必要的；Asynchronous RPC使用的是两个one-way RPC---用对于client的interrupt实现，但其实还是synchronous的功能实现



RMI：Remote Method Invocation

RMI和RPC类似，但是是Oop中的概念

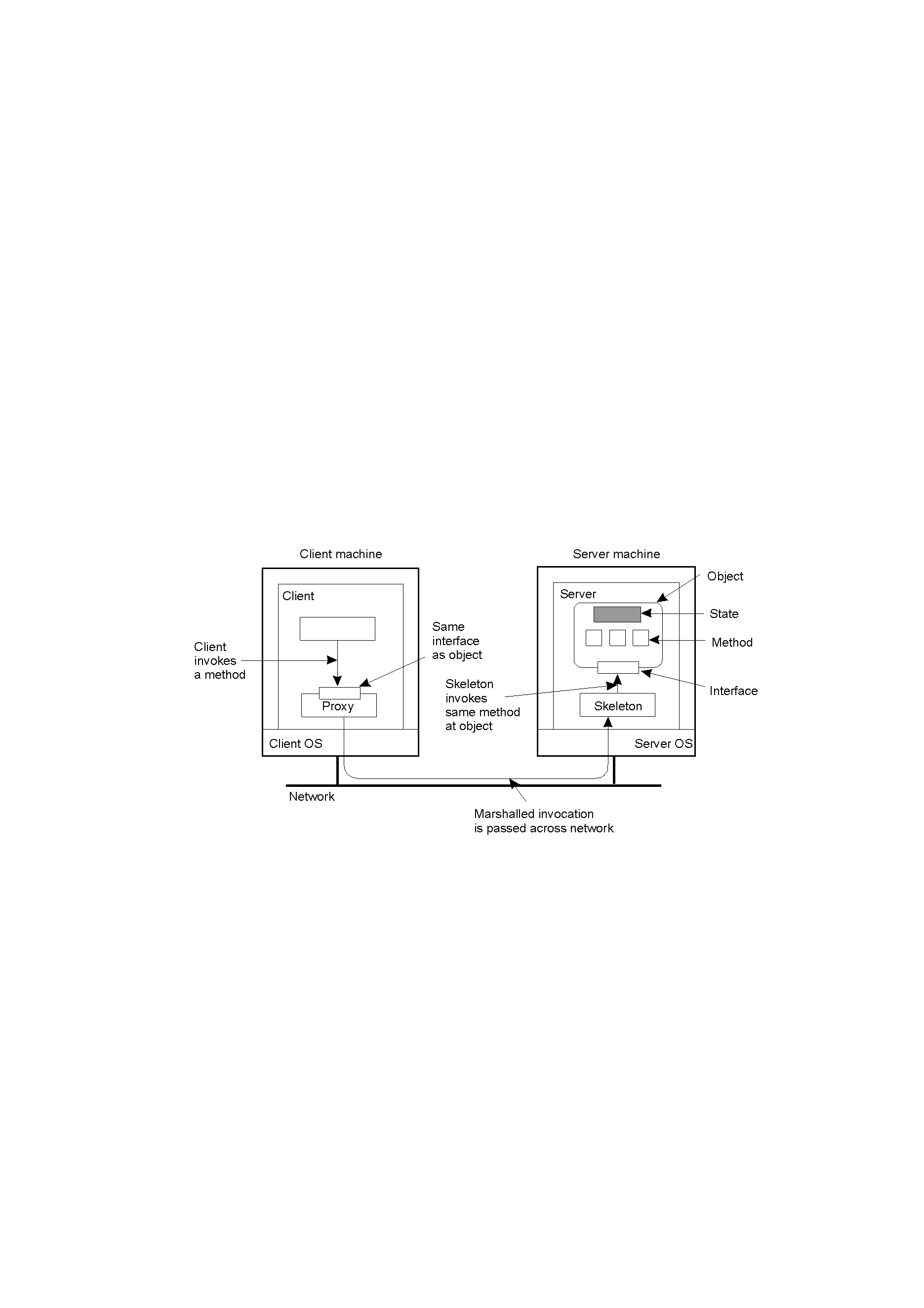
Distributed Object(Remote Object)：指的是不同的Object在不同的地方，而不是一个Object被拆开；这些Object被server和client管理，并通过RMI调用；Objects are replicated and migrated to enhance performance

RMI的工作原理：

Proxy---Client Stub implement Stub

Skeleton---Server Stub used for receiving requests

功能和Stubs是完全一样的



Remote Object Reference: 想要invoke一个method，需要Object reference，Method name，Necessary Arguments；对于RMI则需要Remote Object Reference

Remote Interface：Remote Object的实现需要通过remote interface：别的process中的object只能访问自己的remote interface中的方法

Parameter Passing: By value & By reference---Optimization：reference Object on other machine则by reference，reference Object on local machine则by value

Message Oriented Communication：

传输信息的过程有两个操作：send和receive---send message到一个queue里，receiving process从queue里提取message

Synchronous & Asynchronous Communication:

Synchronous: 发送和接受都是带有block的

Asynchronous: 发送是non-blocking的；接受可以是blocking或者non-blocking的，使用buffer进行接收

接受的destination使用IP + Port的二元组标识

IP Multicast: 和广播的区别在于广播是给所有人发，多播只给一定的IP地址发

并且即使不是组内成员也可以给一组multicast group发送多播

Transient Communication：

发送的message不会在communication system中储存，只是简单的基于message-oriented model在transport layer上的实现。E.g. Socket interface, MPI

MPI: Message Passing Interface standard

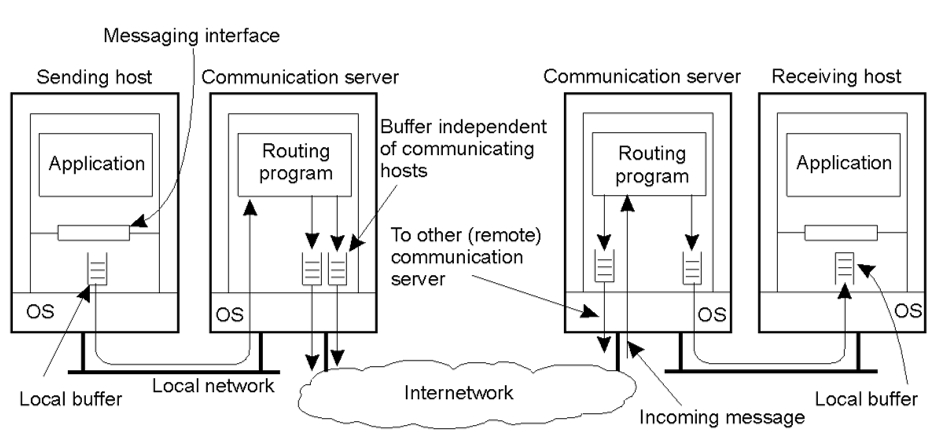
在grid computing中很重要

The goal of MPI：保留了message passing的简易，可操作性和效率；通过独立于编程语言和OS的interface提升了portability

Persistent Communication：

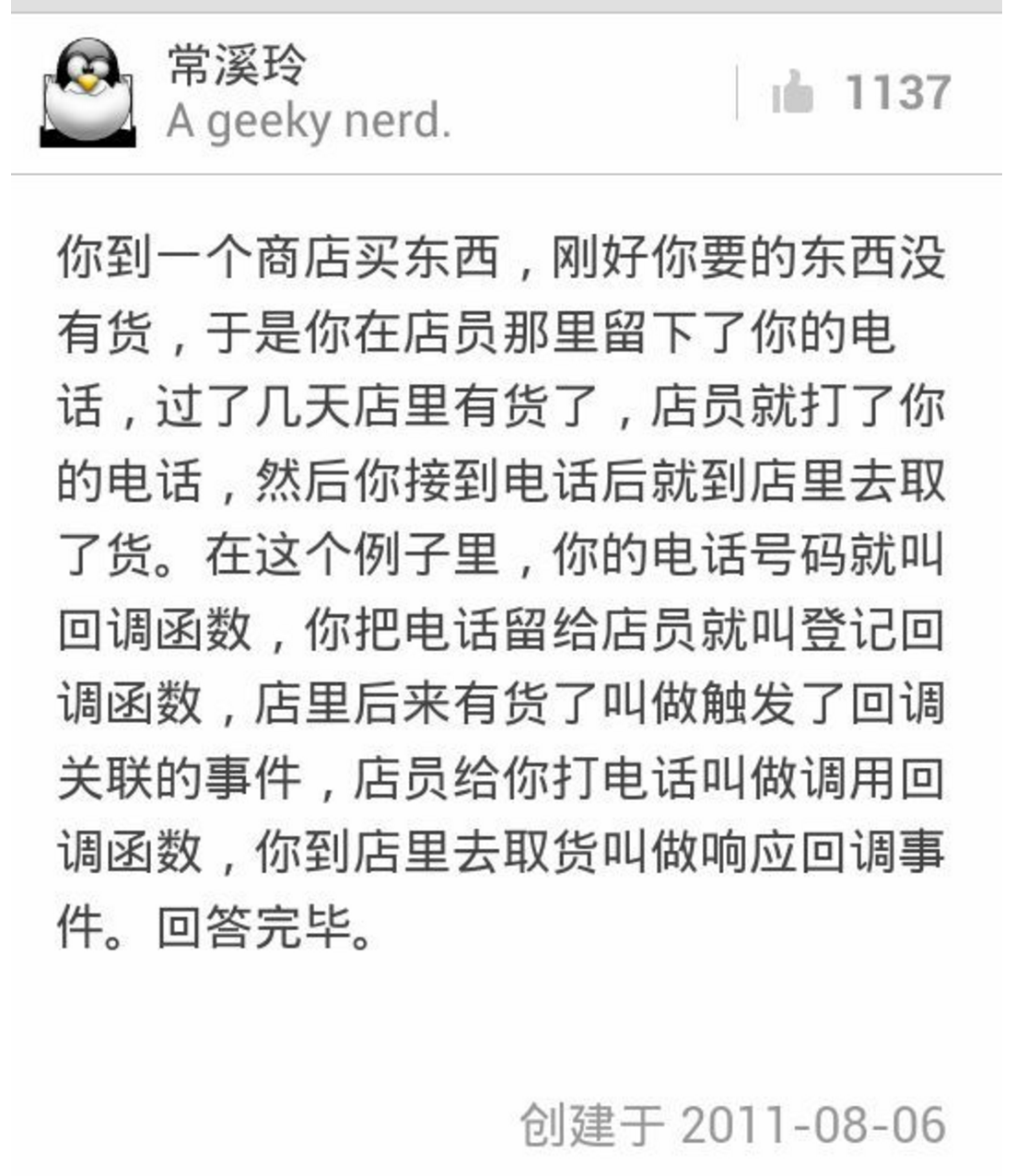
发送的message会被储存在communication system中，直到被发送到接收方。

A basic communication system:



主要就是利用了buffer实现了persistent communication，在sending host，communication server和receiving host处使用buffer来存储。多个buffer通过网络拓扑存储不同发送方向的message。Application与buffer交互即可。Message queue通过queue manager维护

Callback function：



Relay：中继

可以创建一个scalable message-queuing system，对message进行二次处理

Message broker：不同的发送端和接收端可能是用的是不同的format，需要通过message broker进行转换得到可以用的message

Stream-Oriented Communication：

Definition：a form of communication in distributed system to exchange the related information---radio stream / video stream

Media: storage & transmission & display media

Continuous/discrete representation media: 对于continuous来说data之间的temporal relationship对于meaning是很重要的，discrete反之

Data stream: a sequence of data units

Transmission types: Asynchronous---data item是一个一个传输的，没有time restriction; synchronous---对于每个data stream中的data item都有一个最大的end-to-end delay; Isosynchronous---data item must be transmitted on time, i.e. no delay for each item.

Simple stream: 由一个data sequence组成

Complex stream: 由几个相关的simple stream组成: substreams

QoS：多媒体系统都是real-time的，由underlying system提供的服务好坏称为QoS

QoS requirements: Timing and other nonfunctional requirements of underlying distributed system and guarantee quality of service

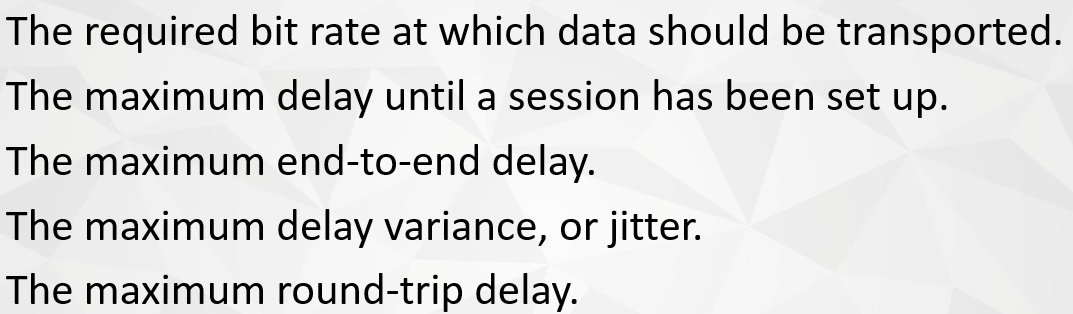
QoS management: planned allocation and scheduling to achieve the meets for multimedia is called QoS management---当前的OS和network很多时候都不提供QoS management，需要通过软件实现

通过QoS management可以让application得到足够多的资源和时间，别的applications于此同时竞争资源

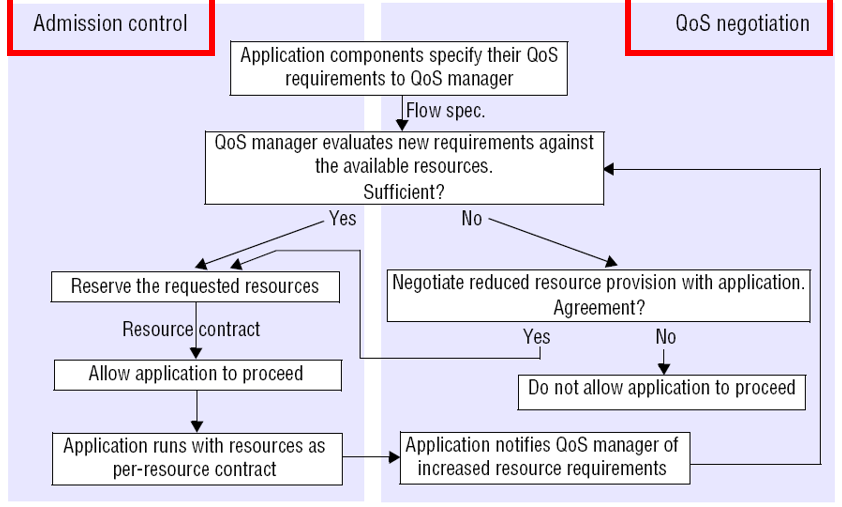
多媒体应用的特性：highly distributed，resource requirement is dynamic，balance is needed for user, between resource cost and other activities

Compression: 可以10-100倍地减少bandwidth requirements，但是timing requirements仍然是不变的。特殊的硬件会process and dispatch video and audio information，比如PC中的video codec

用于描述requirements的specifications：



QoS management tasks: Admission control & QoS negotiation



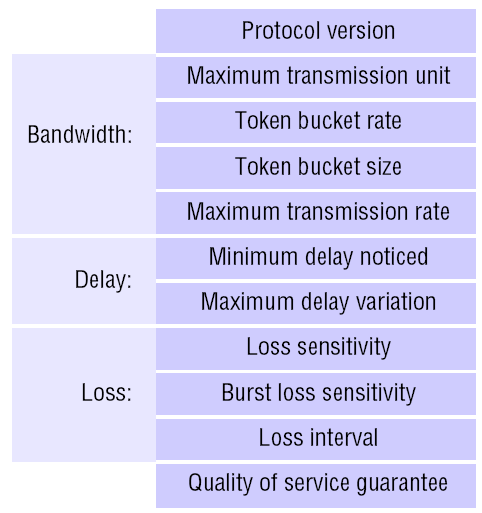
QoS negotiation: specify不同的stream的parameters---bandwidth，latency，loss rate

Traffic shaping：

Bucket algorithm---specify influx and network traffic

以固定的速度创建固定大小的token，用token向网络中传输，保证向network中的传输速度会比较固定

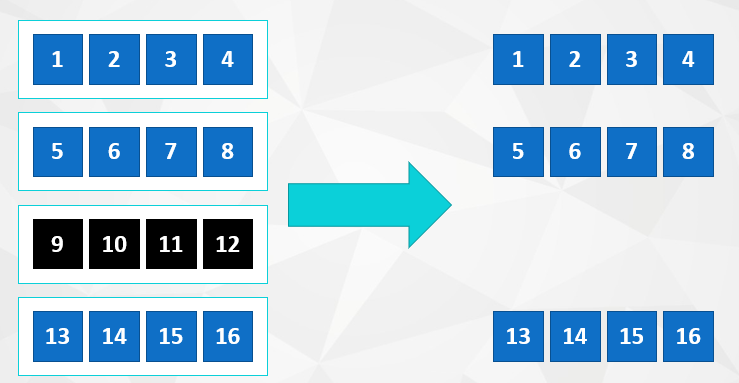
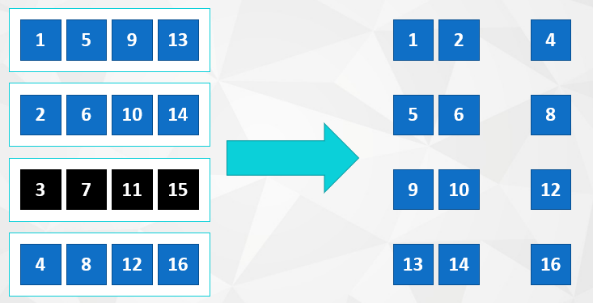
QoS requirements：



Admission control：Regulates access to resources to avoid resource overload and to protect resources from requests that they cannot fulfill.

Packet loss solutions：error correction---重传(多媒体中不适用)&encoding

但是如果是audio和video packet丢包了，会产生一个比较大的gap，因此使用了interleaving frames的机制：在打包的时候将连续的帧拆开(右图所示)：



Stream synchronization：多个continuous data item之间的synchronization & 一个continuous data stream和discrete data stream之间的synchronization

Low-level synchronization---通过不同的data item之间的操作实现同步，缺点由application实现。可以通过interface使得application更容易控制devices

High-level synchronization---一般通过多媒体middleware实现，由application控制并由中间件实现synchronization

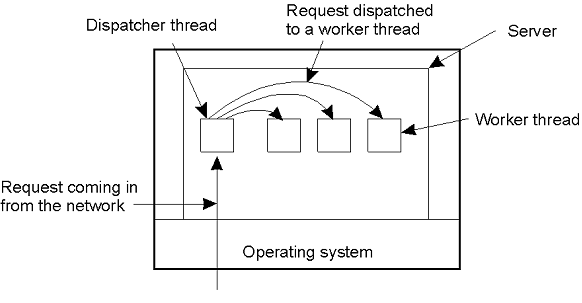
Chap3. Process

Threads:

Thread and process: 进程是executing program, 所有进程share resource transparently. 进程实现concurrency需要很高的cost, 并且process对于实现分布式系统来说过于庞大；线程是executing program或者executing program的一部分, 并且更容易实现transparency concurrency

Multithread Clients：can achieve distribution transparency by concealing message transmission time. E.g. browser---使用不同的线程获取一个网页中的不同元素

Multithread Servers：可以简化服务器的代码并且实现并发



分为dispatcher thread和worker thread：dispatcher负责读取request并且选择一个闲置的worker thread分配任务去执行

Clients and Servers:

Network User Interface:

Client的主要任务就是为用户提供与远端的server进行交互的接口，主要有两种形式: 每个service都有一个单独的部分用来contact server；convenient user interface

在client端的软件distribution transparency：

Access transparency---根据server提供的service在client创建stub，使得user并不知道自己所访问的服务是由本地或是远端提供的

Location/Migration/Relocation Transparency：使用convenient name system。如果server移动了，则很容易通知client并与client上的software cooperate

Replication Transparency：向多个复制发送请求，但是只返回一个答复

Failure Transparency：Client Middleware会尝试连接特定的服务器，在多次失败后尝试连接另外的服务器。用户并不会意识到failure

Server Classification：

Iterative Server：服务器负责处理请求并且返回给client

Concurrent Server：服务器并不负责处理client的请求，发送给一个thread或者process进行处理并等待下一个request。多线程服务器---Unix通过fork()函数生成新的进程处理新的request

How to connect client and server:

Client的request会被发送给server的endpoint或者port，但是如何得到endpoint---公知的端口号(e.g. HTTP 80, FTP 21) / use a daemon(守护线程) on server for tracing or use a super server for monitoring all endpoint

Communication interrupt: 如何中断服务器的service

Solution 1: exit the client application

Solution 2: Support out-of-band message sending---指的是服务器需要优先处理的某个client的data：第一种方法是使用两个端口号，一个负责监听out-of-band data，另一个正常监听；第二种通过同一个connection发送out-of-band data

Stateless server & Stateful server:

Stateless---不会保留client的信息，state改变时也不会通知client e.g. browser

Stateful---保留client的信息，如果server fails则所有的state都会被保留 e.g. file server

Object servers: 用于支持object的server，类似于一个装object的容器

和传统的服务器的区别在于服务器本身不提供服务，而是通过服务器维护的object提供

服务器可以invoke local object method。通过增删object可以增删服务器提供的服务

Activation policy：用于决定如何唤起每个method

Object adapters：用于实现某个特定的activation policy，object server可以支持多种activation policy

Cluster computing：底层硬件有多个相同或者类似的工作站或者PC，通过高速LAN相连

每个node运行相同的OS

Server clusters：一般的管理分为三个tier：①switch负责给request提供routing ②负责处理应用的服务器们 ③处理数据的服务器们。目标还是transparency：对于用户(client)隐藏多个服务器的事实。

Code migration：目的是process migration

Definition: process migration---整个process换了个机器

原因是①load balancing ②在离data近的地方进行处理 ③flexibility---parallel computing，将一个application分成多个部分并在不同的机器上运行

Performance是通过qualitative reasoning实现的，而不是mathematical modeling---heterogeneity of underlying platform & computer network（定性推理而不是数学模型---并不见得每一次好处都能够完全的体现出来，比如有些时候移动进程的开销可能比带来的收益更大）

Code migration models：focusing on code, state of executing program, suspended signal, other environmental elements

Process的三个components---code segments，resource segment & executing segment

Weak Mobility/Strong mobility：

Weak mobility：只移动code segment和一些initialization data，被移动的program总是从固定的位置开始执行。对于被迁移的机器只需要能运行被迁移的code即可，容易实现

Strong mobility：code segment和executing segment都会被移动，需要停止当前运行的进程，移动后从断点重新开始运行，较难实现

可以从发送端或者接收端发起 code migration

Sender initiated：程序所在/运行的机器发起，难以实现。比如向computing server上传计算

Receiver initiated：target发起，容易实现。比如client下载一个program

Methods：

For weak mobility：execute in browser process space/execute as a single process

For strong mobility：migrate process

Remote cloning：一份original process的copy在另一台机器上并行运行

Resource migration：code migration中最难的一部分

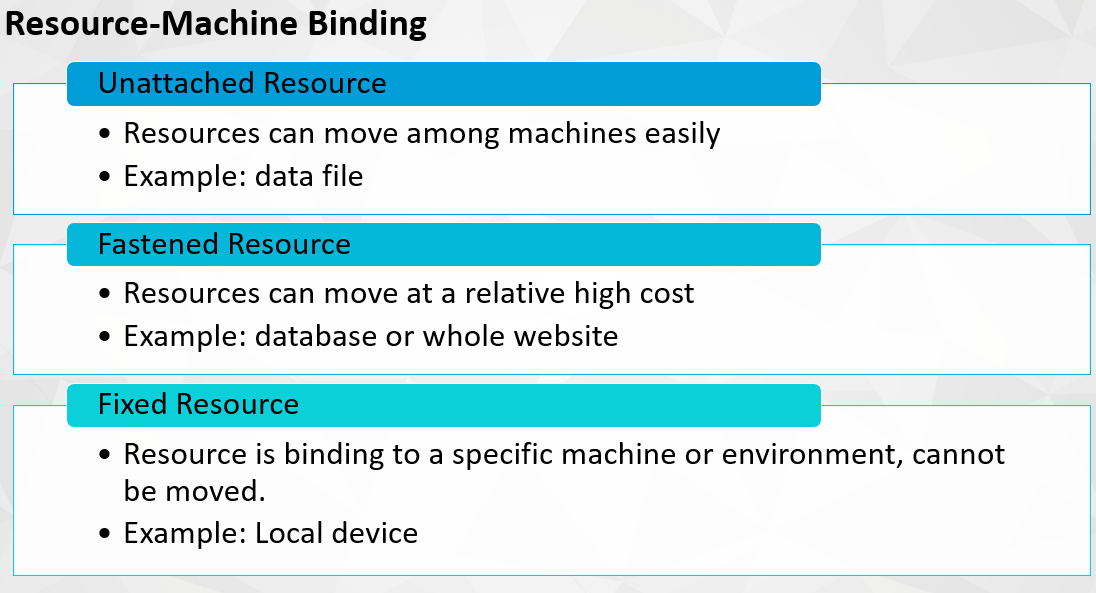
一个process中有code, resource and executing segment，如何在不改变其他部分的前提下移动resource segment

Process-resource binding：

Bind-by-ID：strongest，process可以清楚地访问resource

Bind-by-value：weaker，如果别的resource可以提供相同的value，则process的执行不会受到影响

Bind-by-type：weakest，process只会request指定的类型



对于heterogeneous system的migration：

Weak mobility：把代码对所有可能的平台都编译一遍

Strong mobility：问题在于正在运行的segment很依赖于当前的平台

Software agent：

Definition：Software agent is an autonomous process, it can initialize change in its environment, and cooperate with other software agents and users.

Classification：

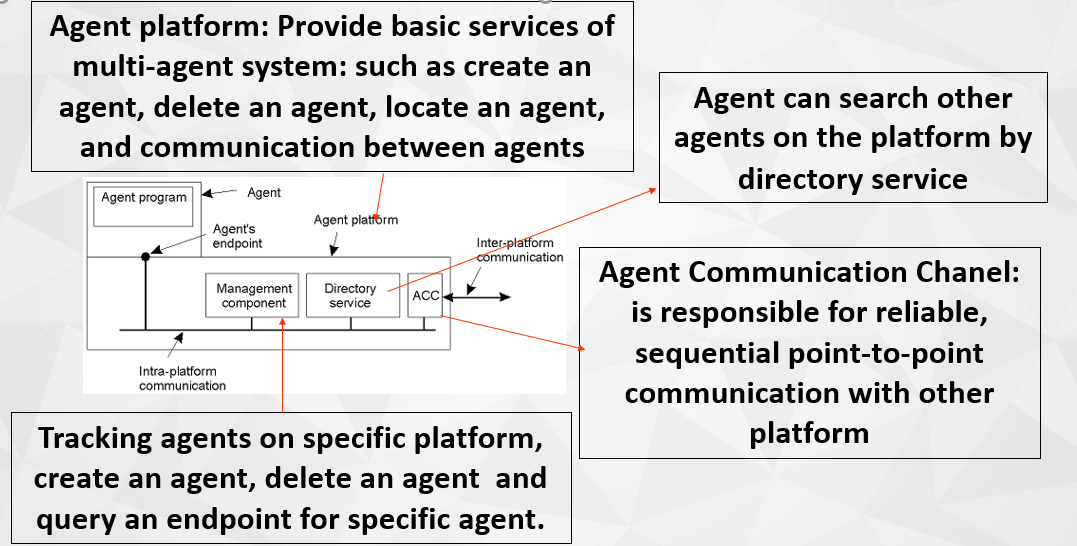
Collaborative agent：在一个multi-agent system中的所有agent以完成同一个目标

Mobile agent：agent可以在多个Machine中移动，需要很强的mobility

Interface agent：帮助terminal users to use one or more application, 需要study capability

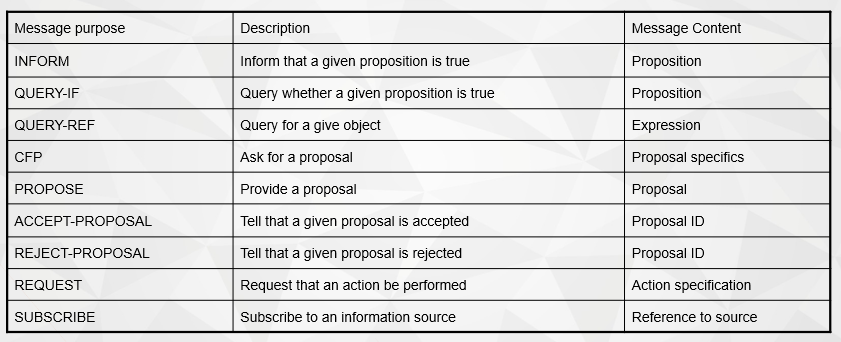
Information agent：manage各种resource，包括sorting，filtering and comparison

Foundation for Intelligent Physical Agents (FIPA)：



Agent communication language：application-level protocol，用于控制agents之间的通信

Purpose：让receiver provide specific service---response previous request, form all messages received, something in act of negotiation



And contents

Process allocation:

Definition: concerned with the required number of processors for a specific job

算法必须尽量优化

对于不同的system有不同的aim：processor utilization / minimum response time

有三个问题需要考虑:

Q1: load measurement

所有的算法都会假设每个机器都知道自己的load，并且可以判断自己是over-loaded or under-loaded和告知别的机器自己的当前状态

但是load measurement并不容易实现

可以通过process数量表示load，优点是简单，缺点是并不准确

也可以通过running & ready process的数量表示load，通过守护线程(Daemon)实现。会给系统带来一小点负荷

也可以通过processor利用率(running time/[running time + idle time])来表示，越大表示load越大，优点是reasonable

Q2: Extra cost

获取load info和给processor发送process都是需要extra cost的，但是很多算法忽略了这一点

一个好的算法需要考虑processor time used, memory use and network BW，是很难实现的

Q3: Complexity

Complexity会影响系统的performance, accuracy and robustness

但是问题在于如果新的算法比旧的performance更好，但是更加复杂，该如何处理？

Process allocation algorithm：

Deterministic vs. Heuristic：deterministic的算法需要知道process的所有信息---Computational requirement, file requirement, communication requirement，然后选出所有scheme中的最优解

Graph-based deterministic algorithm：

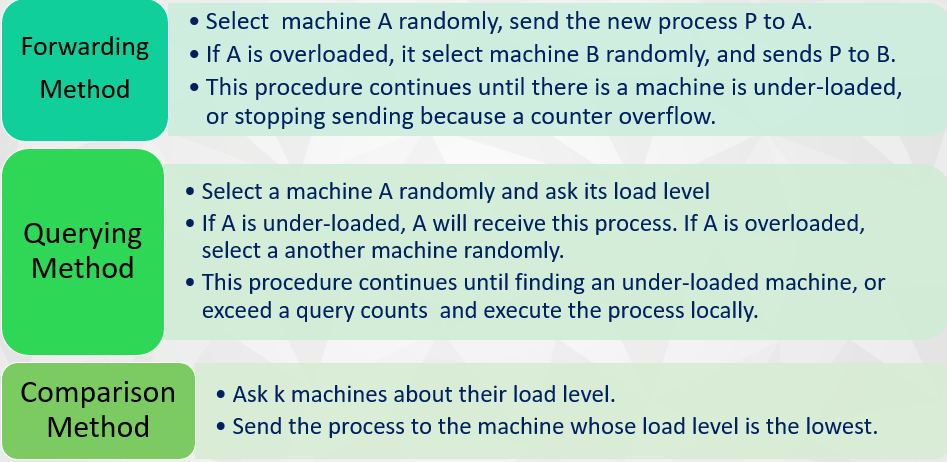
假设所有的process都知道自己的CPU和memory requirement以及两个process之间的平均communication traffic

目标是找到这个graph的一种partition method，使得所有sub-graph之间的平均communication traffic最小

Heuristic：系统的load是不确定的，不可预定的

EAGER heuristic algorithm：每台机器都知道自己的load，创建新的process时检查自己是否overloaded，如果超载了则寻找一台可以运行这个process的机器

启发式算法寻找另一台机器方法：



Local vs. Global：

Local：本地设置一个threshold，没超过的话process will stay locally，否则会被发送到另一台机器上

Global：先获得别的机器的load information，再决定要在哪台机器上运行

Centralized vs. Hierarchical：

Centralized：自顶向下的算法，由一个节点进行决定。是一个centralized heuristic algorithm without knowing all info（不是deterministic）：

通过一个coordinator维持一个table，其中记录了系统中所有节点的一个score，发生事件时向table发送一条消息update table。Coordinator会选择一个score最低的processor的请求优先处理：实际上是调度的公平性的保证

如果一个节点有运行在别的processor上的process，则每个time unit+1分

如果一个节点想要把process放在别的机器上运行但是被拒绝了，则每个time unit-1分

如果一个节点一直没有request，则每个time unit-1分直到最后为0分

优点是这是一个heuristic algorithm，保证了fairness

缺点是coordinator会成为系统的bottleneck，如果coordinator坏了则整个系统也就崩溃了

Hierarchical：保持了centralized algorithm的simplicity，并且可以适用于很大的分布式系统

相当于多层的centralized algorithm

如果要给task有k个process，则应该分配k个processor

预留R个processor(R>=k)---但是预留的不一定都会用上

每个internal node追踪并记录自己的field中的所有processor

如果不能满足所需求的processor数量，则向上传递请求

在预留了R processors之后请求会向下传输，被分配到进程的处理器会被标记为busy，分配的情况会被上传，实际上最后只分配了k个进程

如果node failure发生了，可以由上层分配，同级或者下级选出一个替代者

最顶层的node很重要，可以设立一个由多个这样的node组成的committee，如果一个坏掉了则committee中的其他node会完成它的工作

如果选择R也很重要，R必须足够大来保证能够分配足够的处理器给task---但是如果R太大则耗时太长，并且太多的处理器会被临时标记为busy并且最后被释放，消耗很多计算时间和资源

Overloader initiated vs. Under-loader initiated：

Overloader initialized: distributed heuristic algorithm，在一个被创建的进程发现机器超载时会被发起：如果超载了，则随机选择另一个机器并询问load level，超载则再询问别的机器，重复最多N次。在系统idle时性能优异，但是如果在系统很busy的情况下所有机器都无法执行，通信反而消耗了更多资源

Under-loader initialized：也是分布式启发算法，当一个进程结束时，系统会check机器的load level，如果是under-loaded则触发算法：结束时check load level，如果under-loaded则向别的机器请求一个进程来执行，如果这个机器也是under-loaded则重复，最多N次

N次之后会保持闲置一段时间，过一段时间之后再去进行询问。当系统很忙时性能优异，因为闲置的processor很少，很快就能找到要去执行的进程。但是闲置时所有机器都会寻找要去执行的进程，但是问题不大

Chap4. Name Service

Name Entity：

Name就是用来refer to某个entity的字符串。Entity就是可以被执行的单位

想要执行entity就需要访问，也就需要address：可变，一个entity可以有多个address

Location-independent-name：如果一直用address作为entity的name会非常inflexible and human unfriendly，并且如果一个entity有多个address或者address会变的话这种reference就不太准确，所以用location-independent-name来表示一个实体---ID & human-friendly name。于是就需要一个naming system来进行address---ID & name的转换

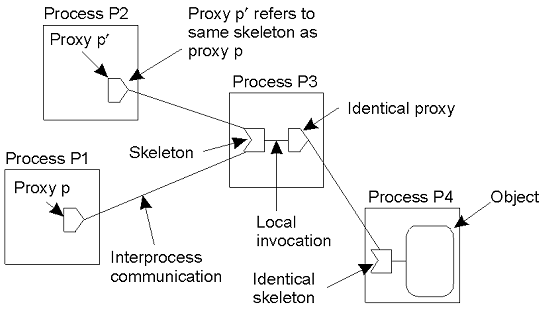
Flat naming：直接用ID---一串随机的数字来表示一个实体，ID并不包含如何访问的信息

因此需要location service来找到如何访问这个实体

……

可以通过forwarding pointer记录(相当于移动路径)，但是会变得很长，而且路径上的每台机器都需要保存这条路径的reference，还很容易断(broken links)

一个例子就是使用(Proxy, Skeleton)进行标识：一个Object在从A移动到B时，在A留下一个proxy，并在B创建一个skeleton。改进版本是object invocation会记录创建invocation的stub的ID，随后可以创建根据这个信息创建shortcut



Home-based Approaches：广播和forwarding pointer的scalability太差，因此如果要在大规模网络中标识mobile entity就需要记录当前位置，其被创建的位置称为home location：每当有别的程序想要访问这个entity的时候，就会访问其home agent，会告知其现在的位置

缺点是latency会比较高，而且home location必须一直存在

Hierarchical approaches：

Network被分为collections of domains，再细分为subdomains

最小的leaf domain一般是LAN或者蜂窝中的一个cell

每个domain中有一个directory node记录domain中的entity

在查询的时候根据locality进行查询，向dir(D)中查询，dir(D)中记录的就是pointer

通过使用cache可以减少上面的节点的workload

Update：从想创建replica的地方发送request，先找到要复制的在哪里然后复制；复制完成之后向父节点插入pointer，向上传递

Root node就是寻找和更新操作的bottleneck，因为所有的节点信息都要存一份在这里：解决的办法就是分成subnodes：让subnodes尽量的近，抑或随机分配（后面可以改变布局）

Structured naming：简单的，人可读的名字

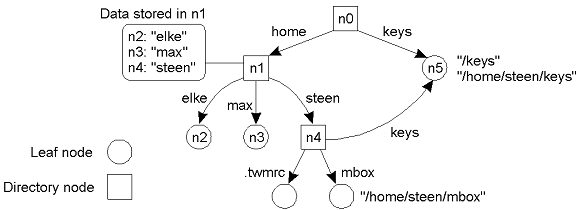
Name space：有两种节点的directed，labeled graph

Leaf node：没有出去的edge，就是一个entity

Directory node：有很多出去的edge，每个都有一个label

Root node只有出没有进

所有的边都是有方向的，也就是label：因此叫directed graph



Path也就是一串label---absolute path-从root开始，relative path-不用从root开始

Global name—在哪里找到的都一样，local name—取决于在哪里找到的不一样

Organization：Tree(每个节点都用绝对路径作为名字，除了root都至少有一个ingoing edge)或者Directed Acyclic graph(有向无环图，比如上图)或者other

Linking：

Alias：another name for the same entity

Hard link---多个absolute path，比如上图

Symbolic link---用leaf node表示entity，在其中储存绝对路径名字

Mounting：Name resolution只能在一个name space中完成

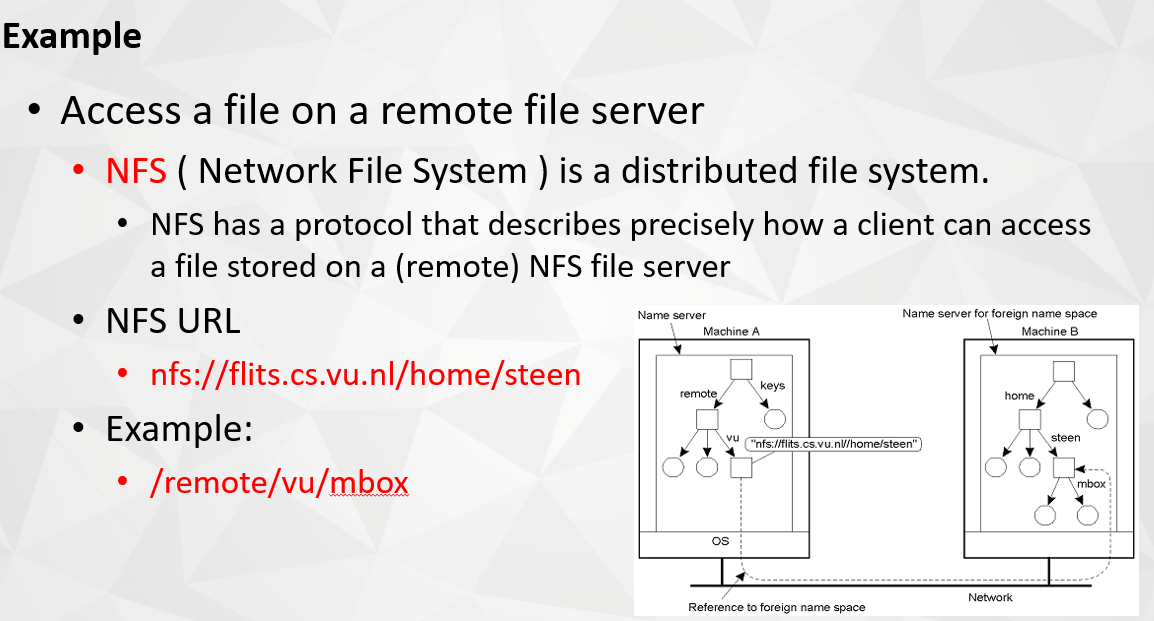
Mounting merges different name spaces in a transparent way

用一个directory node储存另一个name space中的信息

Mount point: directory node in this name space

Mounting point: directory node in another name space

实现mounting需要：The name of an access protocol，The name of the server and The name of the mounting point in the foreign name space



Implementation of name space：

Naming service---add，remove，loop up

Name service implemented by name servers: 小的系统就用一个name server，大规模的系统就在多个name sever上实现name space

Name space distribution：最上面global layer，中间administration layer，下面是managerial layer

Availability and performance：

Global layer要求很好的availability，需要replicate server；performance依赖于client-side cache

administration layer和global layer一样

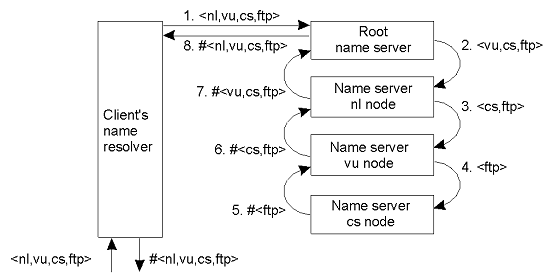
managerial layer对availability要求不是很高，但是对performance要求较高，而且在这一层cache已经不太管用了，因为需要经常更新

Name Resolution implementation：先假设没有replica of name server和cache，通过name resolver来进行name resolution

Iterative Name Resolution：比如对于root:<a, b, c, d.txt>：client先对root进行访问，root name server会去掉a并且返回<b, c, d.txt>；client再访问b这一层的name server，去掉b，返回c的name server的地址：依此类推，最终找到这个文件所在的位置



Recursive Name Resolution：Client访问Root，Root访问b，b访问c……找到后d返回c，c返回b，b返回root，root返回client



优点是减少了communication cost，caching result is more effective

缺点是对name server的performance需求更高，并且要完成整个解析

Attribute-based naming：Directory Service

之前的flat naming和structure-based naming提供了绝对唯一且独立于地理位置的指明实体的方法，但是location independence和human-friendliness并不是唯一的标准

可以使用一种新的方法，需要用户提供一个大概的对于需求的实体的描述以进行搜索

对于entity来说，通过一组(Attribute, value) Pairs来进行描述

可以有多个属性，用户可以通过其中的某些属性进行搜索，所以设计这些属性很重要

Delete Unreferenced Entity：

Why Delete---save resource

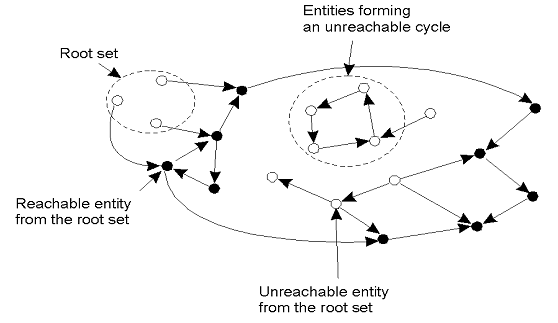
在传统的系统中可以清晰地直接删除entity，但是在分布式系统中很难知道一个entity是否是被referenced---distributed garbage collector

这里做出的assumption是需要删除两种object：一种没有remote reference的，另一种是只有两个object之间的互相的关联的object

Reference graph：

Root Set---root set中的节点必须存在，是请求的发起点

所有的白点是无法从root set达到的，应该被删除



Simple Reference Counting：使用一个reference counter

增加一个reference---counter+1

删除一个reference—counter-1

Counter=0时删除object

在DS中，reliability of communication会造成一些问题:

比如说请求增加一个reference时，返回的ACK丢包了，于是重传了一份请求并正确地ACK了，于是只增加了一个reference但是counter+2

Reference Listing：

在skeleton中用一个list记录所有的referenced proxies

因此重复的操作并不会产生任何影响

但是这样做会使得scalability变得很差

Tracing-based garbage collection：用来找出所有的unreferenced object

从root set出发，trace DS中所有的object，所有无法被达到的object都可以被删除

问题是scalability同样很差

可以通过hierarchical structure解决这个问题：使用层级的group进行tracing

Group就是一组进程的集合

先从小的group开始收集garbage，再在更大一级的group中重复收集

算法是通过skeleton的传播过程进行标记：①先找到一个有remote reference的并标记

②从skeleton传到proxy ③从proxy再传到下一个的skeleton

④重复②③ ⑤garbage reclamation

Chap5. Synchronization：

Physical Clock Synchronization：

由于各个机器上的clock可能不同，会造成混淆

不同的机器上的quartz crystals的频率可能是不同的，不同机器上的时间差称为clock skew

因此需要external physical clock：

一秒的标准就是solar day/86400，solar day的长度通过取大量测量的平均值

现在通过Atomic clock计算一秒International Atomic Time(TAI)，即铯133每9,192,631,770次跃迁记为一秒，但是由于实际的一天和原子钟记时的一天是有时间差的，所以引入了闰秒

UTC提供了WWV，每秒开始的时候发送一个WWV脉冲

因此如果一个机器有WWV receiver，别的机器就可以和它进行同步，有WWV receiver的机器被称为time server；如果没有机器有WWV receiver，那么就尽量维持所有的机器同步

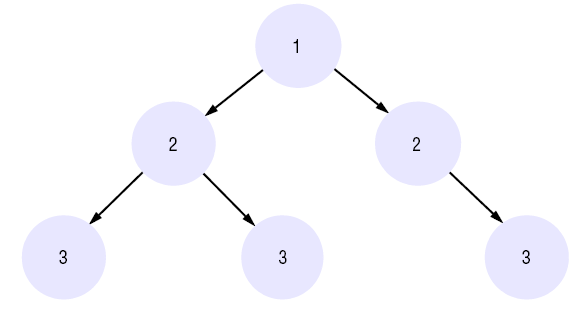
Cristian Algorithm：在同步时通过多次测量估算delay；时间不可以倒流，因此通过设置interrupt来进行时间的回调(也就是等)：正常的interrupt以此增加10ms，可以通过调整成为9或者11ms来进行时间的调整

Berkeley Algorithm：timer server进行轮询，求时间的均值后告知；在这个算法中不需要WWV receiver，在内部进行同步；这个算法中不考虑事件不可以倒流，即可以回调时间

Averaging Algorithm：所有机器通过广播进行同步：所有机器都广播自己的时间，设置一个synchronization interval S(为了避免特别高的延迟)，在S时间之内收到的时间去掉m个最高最低，求均值后再进行调整。但是只能在小系统中用

Network Time Protocol(NTP)：

最顶端的primary server接受UTC时间，每个节点和parent node进行同步



Logical Clock：实际上process关心的只是事件发生的先后顺序而已，不一定必须要和时间的时间进行同步

Lamport‘s Logical Clocks：

a🡪b表示a在b之前发生

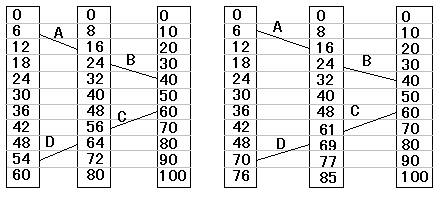
如果a🡪b和b🡪a都不为真，则认为他们是concurrent的

记C(a)为a事件发生的时间，如果a🡪b则C(a)<C(b)；时间只能increase不能decrease

假如a🡪b，则C(b)至少等于C(a)+1，以此来标识两个事件的先后顺序

每个process记录一个local counter，每次处理事件之前+1；在发送message m时，附上m的时间戳ts(m)；接收时将本地的counter和ts(m)进行对比，取较大的赋给counter，并+1，将message递交给application

比如在下面的例子中，C的发送是60，但是接受是56，因此直接将接受改为61



Total ordering of 2 events：不允许两个事件同时发生，在time stamp后加上一个process ID以区别同一时间两个时间的先后顺序

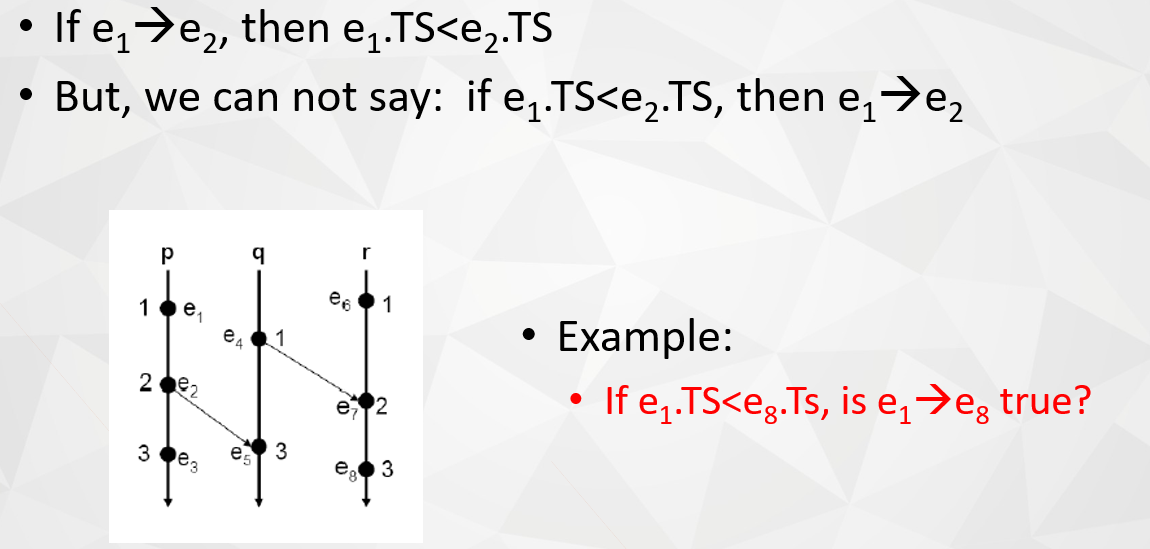
Total-ordered Multicasting: 假设一个数据库有多个replica，请求总是会发送给最近的replica，维护之间的操作的synchronization就需要通过Total-ordered Multicasting：

一组process互相之间广播message，消息通过本地的逻辑时钟印上timestamp

在多播的时候也发送给sender一份，假设所有的message是按照发送的顺序被接受的并且没有loss，接受者将所有接受的message放到queue中并且排序，随后发送ACK给所有别的process，使用Lamport’s algorithm to adjust local time，最后所有的processes都有一份相同的local queue的copy

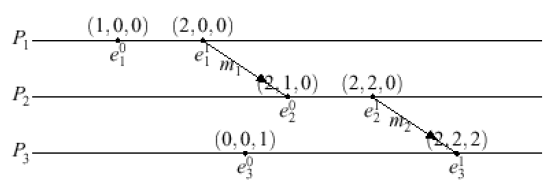
一个Process仅可以在两种情况下向application递交local queue：当其为queue中的队首时 & 当local queue已经被ACK的情况下

Limitation of Lamport’s Algorithm:



新的方案：Vector Clocks

Global Timestamp，V(e)和事件的先后是一致的，V(e)是一个向量(t1,…, tn)



每个process记录自己内部的时间戳，初始值为0，每一次内部时间会使得自己的tn + 1

每个process发送和接受消息的时候在自己的分量上+1即可

在接受的时候，会将V(e)中的所有元素取最大值

Global State：process的集合的state

尽管说DS的特点包括no global state，但是有一个全局状态还是比较方便的

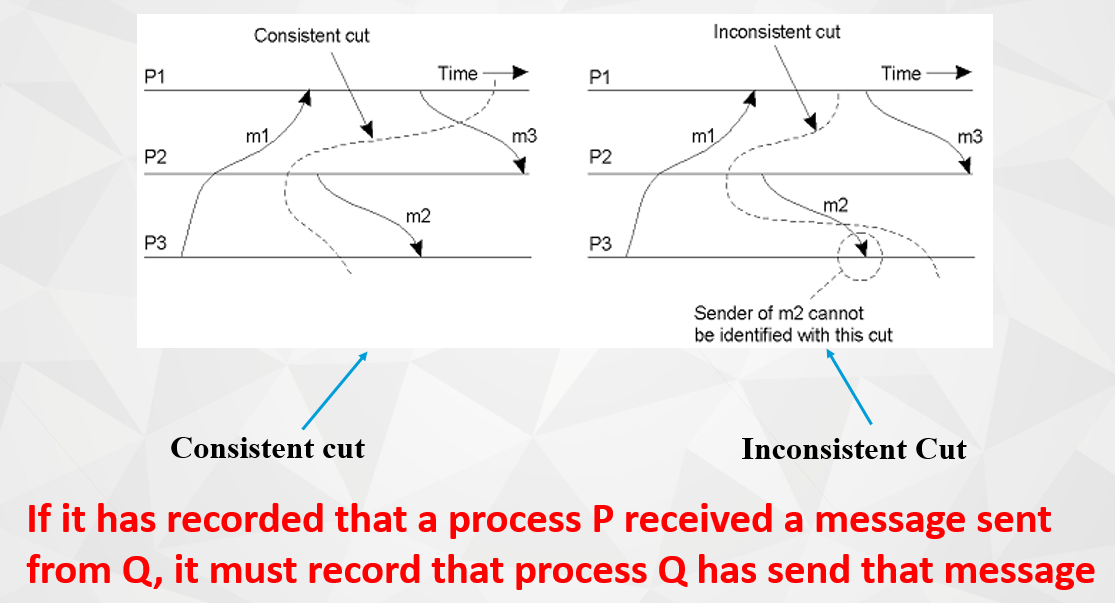
任意的process的集合的state在一起即为global state S = {s1, s2, s3, …, sn}

A consistent global state is one that corresponds to a consistent cut

每个系统的状态通过transition of state记录：s1➡s2➡s3

何所谓consistent cut：

接受必须要记录发出，发送不一定要接受



Snapshot algorithm：跳过了

Election Algorithm：选举coordinator和initiator的算法，goal是所有的process都认同选举的结果

Bully Algorithm：一个process P发现当前的coordinator不能够回应request，则发送Election message给所有PID大于P的程序，如果没有process回应则P赢得选举成为coordinator，如果Q回应了则P放弃选举，Q重复这个过程

最终剩下的一个没有放弃的process成为coordinator并且发送消息告知别的进程

Ring Algorithm：

假设所有的process都有一个成环的逻辑顺序号，每个process都知道自己的下一位

一个process P发送Election message，在其中加上自己的PID并且发送给下家，每个process都在message中加入自己的PID并且传给下家，最后回到P，由P告知所有process Coordinator消息

Ring & Bully algorithm都是基于消息传递是可靠的和网络拓扑不变的前提下进行的，并不适用于wireless environment

Wireless environment中的election：

每个node有自己的capacity value，coordinator的capacity是最大的

每个node都可以开始election，向neighbor发送Election message，接收者记录第一次接收到Election message的发送者为parent；当一个节点收到非parent发送的Election时，不会继续发送Election message给neighbor

在所有的children返回ACK后，capacity最大的node会被acknowledged为coordinator，这个election的发起者可以知道coordinator是哪个节点

Mutual Exclusion：

Centralized algorithm: 需要一个coordinator P

如果Q想要进入critical section，需要向P请求granting

如果critical section是空的，则返回许可；如果非空，则将Q加入等待队列中

当一个process退出critical section时，向queue中第一个process发送granting

优点是公平，不会有starvation；缺点是coordinator failure会导致mutual exclusion不能实现---如果在重新选举之前critical section非空，重新创建之后会导致之前在queue中的process一直处于waiting状态，starvation

Distributed Algorithm：

当P想要进入critical section时，发送一个消息，其中包括critical section的名字，P的PID，Timestamp

将这个message发送给所有其他process，等待granting

当Q收到message时，如果Q不想进入critical section，则返回OK

如果Q在critical section中，则不返回response

如果Q正在waiting，则会比较自己的请求的timestamp和接受的消息的timestamp，如果接受的在前则返回OK，在自己之后则不做response并且将请求放入queue中

优点是没有deadlock和starvation

缺点是one-point failure会导致N-point failure；需要维护一个member list，并且需要使用group communication primitive

Token Ring Algorithm：

所有的process形成一个逻辑环，其中有一个token，拿到token的process可以进入critical section，否则将这个token向下传递

如果token丢失了，则重新创建一个token，但是如何判定token丢失需要考虑

一个process fail则整个算法可能fail

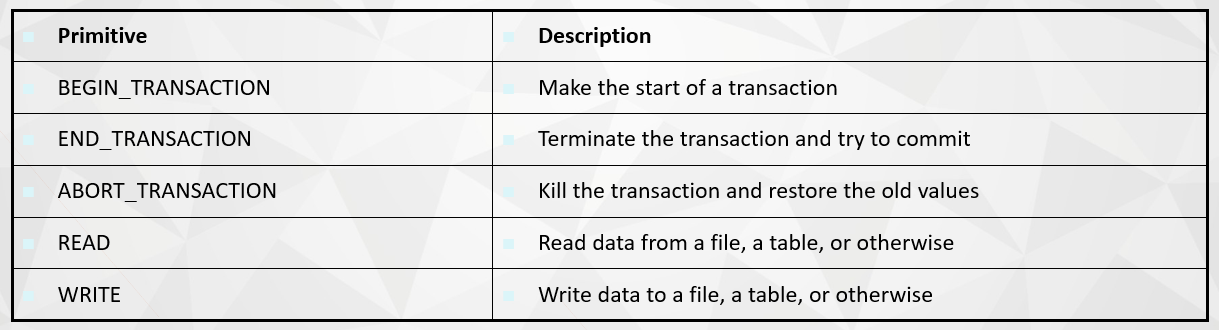
Distributed Transaction：

Transaction protect shared resource from concurrent access

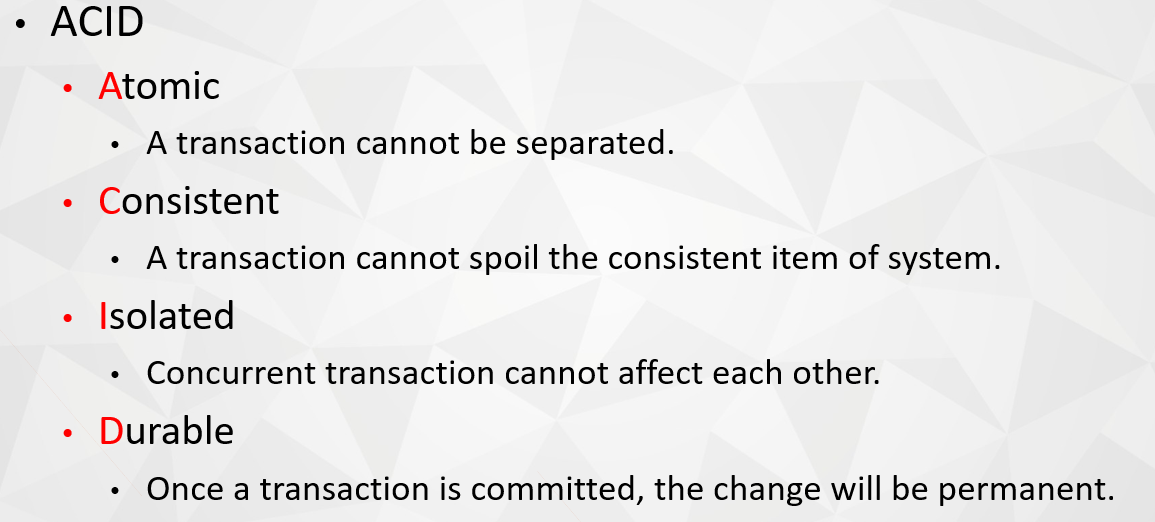
Permit a process 对多个item进行读写操作，把这几步当成一个atomic operation

All-or-nothing property：要不然都做，要不然都不做

Primitives：系统需要提供原语用来支持transaction



Transaction的特点：



Nested Transaction：对应的是flat transaction

Nested Transaction指在别的机器上有运行的sub-transactions

如果parent fail，则恢复transaction开始之前的状态，所有的sub-transactions都应当rollback

只有top transaction产生的变化是durable的；如果一个sub-transaction fail，由top transaction决定是否abort

实现rollback的方法：

Private Workspace：

在transaction开始时会分配private workspace，用于存放各种数据，读写都在private workspace中完成，知道transaction提交或者失败

Optimization：当一个进程只需要读取文件时，不需要copy data；当一个进程想要写入时，只需要在第一次访问数据时将其拷贝至workspace

Just copy file index---只对metadata进行修改

Writeahead log：在更新一个文件之前，在日志中添加一条记录

记录中包括进行修改的transaction，哪些数据被更新，更新之前的值和之后的值

只有日志写入成功之后才可以进行修改；后续可以通过日志进行rollback

Concurrency Control：

在多个transaction同时进行时维持数据的consistency

可以通过让所有的transaction按照一定的顺序执行来实现

为了解决这种问题提出了serial equivalence：

多个transaction可以同时执行，但是最后的结果要和串行执行的效果是一样的

Conflicting operations：当有两个操作都想访问同一个item且至少一个为写入时的情况

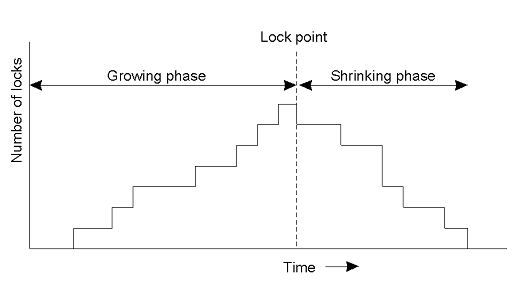
如果想要两个transactions serially equivalent，充分必要条件是两个transaction的conflicting operation对于所有的object的访问顺序都是相同的

通过lock实现：

当某个object被client transaction访问时server会对其上锁，如果client transaction试图访问一个上锁的object则请求会被挂起知道锁被解除

2-phase locking：oldest & most widely used synchronization algorithm

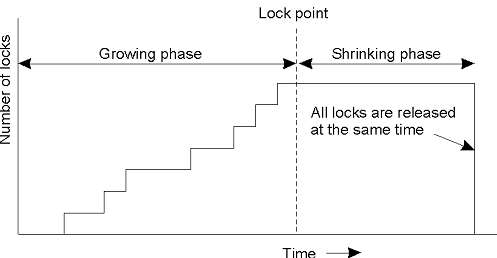
Transaction在释放锁之前不能获取同一个锁：growing phase获取锁，shrinking phase释放



Strict 2-phase locking：

只有在transaction完全结束(结束或者失败)之后才释放所有的锁：

优点是避免了cascaded abort---在释放了一个锁之后想要rollback，但是如果这个锁已经被别的transaction获取 则会导致cascaded abort



Many readers/single writer scheme：

多个transaction可以同时读/一个transaction可以写 一个object，但是不能同时发生

T and U are two concurrent transactions

1. If T read an object, then U must not write that object until T commits or aborts.

2. If T write an object, then U must not read or write that object until T commits or aborts

3. A read lock can be promoted to a write lock if the read lock is not shared with other transactions.

Deadlock：

两个transaction以不同的顺序访问同样两个锁，则会产生死锁

可以通过设置对于锁的访问顺序；通过distinct graph检测loop；Timeout(简单，但是有点不合理)

两种concurrent control methods：

Pessimistic approaches：

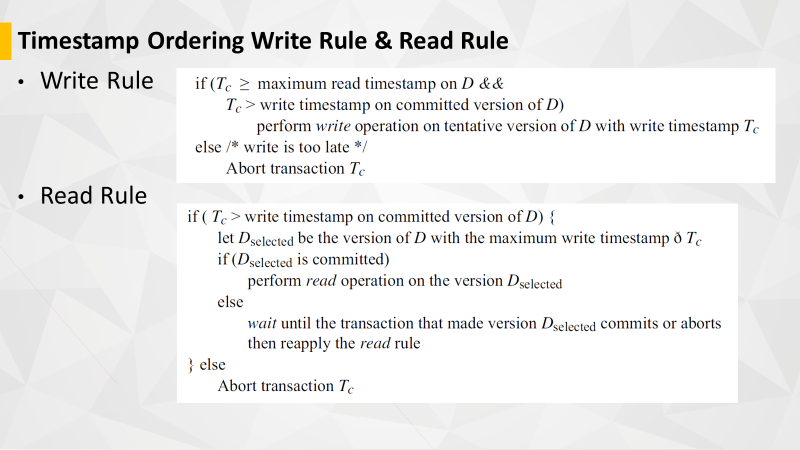
Murphy law：如果一个transaction可能fail，则必须fail

通过timestamp ordering：在执行之前所有操作之前进行同步

写操作只有在一个object被比自己早的transaction进行完读写时才有效，如果有比自己晚的transaction进行了读写则需要rollback

读操作只有在一个object被比自己早的transaction进行完写时才有效，如果有更晚的进行了写入则也需要rollback

每个transaction有一个自己的timestamp，每个object有两个timestamp，分别记录最后一次读和写操作的时间戳



Optimistic approaches：

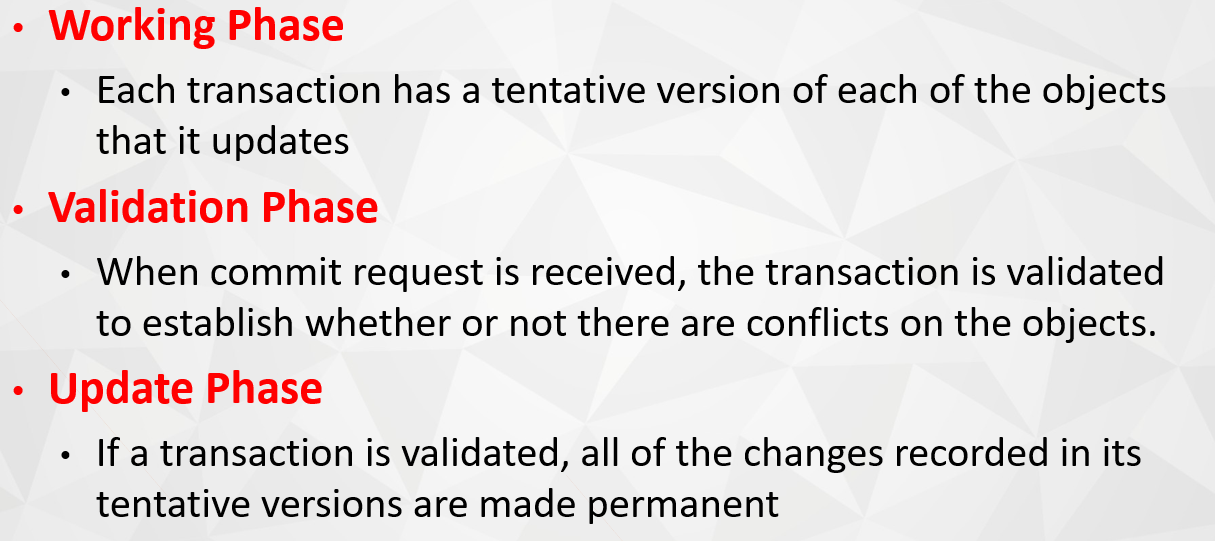
在transaction完成时进行同步，如果产生冲突就进行rollback

记录所有的读写操作，在提交transaction时检查开始之后文件有没有更新

如果有更新，则transaction abort，如果没有则transaction committed

当confliction比较少的时候会比较有效

Working principle：



几种方式之间的比较：

Locks---有overhead，可能死锁，并发性并不是很好

Timestamp ordering---在transaction开始前通过静态的方式来决定serialization order，而lock是动态的

对于read-only transaction来说 timestamp ordering更好，而locking在transaction主要是更新操作时比较好

Optimistic approach在confliction较少时好使，但是一旦发生会需要很多操作来进行abortion

Deadlock problem：

Ostrich Algorithm---Ignore the problem

Deadlock prevention：避免deadlock的发生

通过一些方法实现：让每个process只有一个资源/process开始时就有全部的资源/请求新的资源时释放所有现有资源/通过某个顺序请求资源以避免loop

Deadlock avoidance：

Timestamp algorithm---需要global time & atomic transaction

当transaction开始时分配一个timestamp，每个transaction有不同的timestamp，通过Lamport timestamp实现：

当一个进程请求资源并且被block时，仅当waiting状态的进程的时间戳早于(也可以是晚于)占用资源的进程时允许等待，否则abort---无论是早于还是晚于都不会出现死锁，因为确定了等待的顺序，不会有环的出现，这种算法被称为Wait-die algorithm

如果允许preemption的话，也可以使用wound-die algorithm---当请求资源的进程的时间戳早于(也可以是晚于)占用资源的进程时进行抢占并且使被抢的transaction回调，否则等待

Wait-die可能会导致starvation

Detect and recover：

Centralized deadlock detection---每个机器要知道自己的资源分配，有一台机器作为coordinator并且有全部资源的分配图，当发现有环时abort a process

维护global resource graph的方法：①当增删edge时向coordinator发送消息 ②周期性地告知coordinator增删edge的消息 ③coordinator在需要时请求消息

Fake Deadlock：

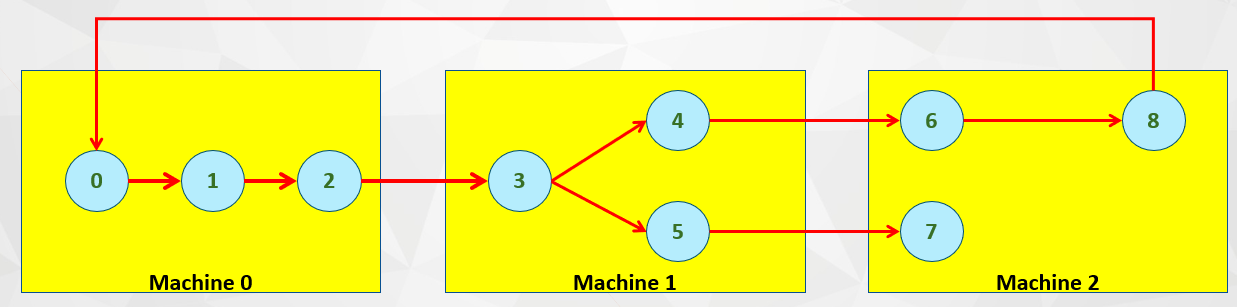


可以通过Lamport algorithm来提供一个global timestamp，每个消息都有自己的global timestamp；当收到一个会导致死锁的消息时，给所有的机器发送一个包含这条信息的时间戳的消息询问有没有在这条消息之前的还未到达的消息。Cost 非常大并且需要全局时间戳

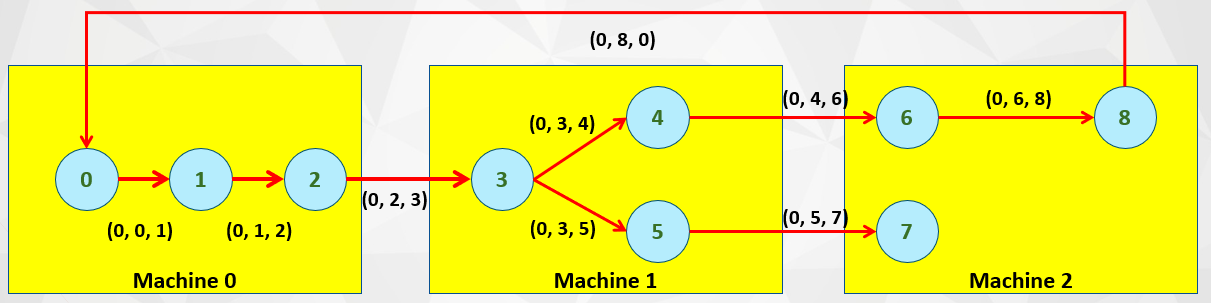
Distributed deadlock detection：

CMH algorithm---允许一个进程同时请求多个资源，一个process可以等待多个processes

通过进程的等待关系记录resource allocation：



通过发送一条消息来检测环的存在：每个消息包括(发起者，上一个发送消息的进程，接受的进程)，当发送和接受相同时即出现了环



第一种方法：发起这个算法的进程commit suicide；但是如果太多的进程发起了这个算法就会有过多的进程自行终结

第二种方法：消息中包括所有进程的PID，出现环时PID最大的(即最新的)进程会被killed

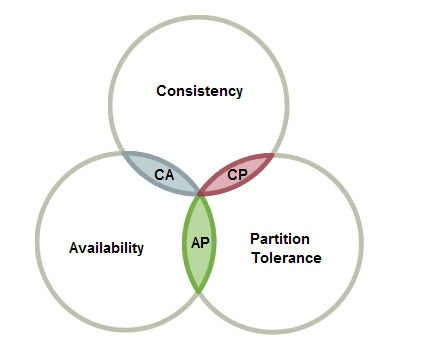
Chap6. Replication and Consistency：

Reasons of replication：reliability，performance & fault tolerance

但是如果有超过一份的replicas就有consistency的问题

使用replication的目的是实现scalability，但是执行replication时会需要更大的带宽，从而导致scalability problem---解决办法是update作为atomic operation/transaction对所有的副本进行更新，或者relax consistency restrict

CAP：分布式系统的三个indexes---Consistency，Availability，Partition tolerance



Partition tolerance---整个分布式系统被分为了多个subnet，每个subnet成为一个partition；当partition之间的通信失效时，整个系统仍能正常工作

Consistency---所有的副本都是一致的，返回的结果也应当相同

Availability---用户在需要时可以访问到所有的nodes

三者无法同时实现

Consistency model：

一份数据在分布式系统中会在多个机器上有一份备份，每个访问数据的进程都应当有一份完整的数据的副本

Consistency model就是进程和数据之间的contract---所有的进程都应当遵循这个规则；当一个进程进行读操作时，应当返回最后一次写操作的结果(看起来是一句废话，但是在有副本的情况下就需要考虑consistency的问题)：在没有global clock的情况下很难判断哪个写操作时最后一次写操作

Data-Centric Consistency model：

Continuous consistency：

方案一：如果存的数据是数值的话，可以允许deviation in numerical values

可以是绝对值，比如$0.2；也可以是相对值，比如0.5%

方案二：deviation in staleness---某些应用可以接受较旧的数据：这个数据是不是太过于老旧? 主服务器可以定时更新replicas

方案三：Deviation with respect to the ordering of update operations---假设更新的顺序对于数据不会有影响的话

试探性地向本地的副本进行写入，并等待所有的replicas发送agreement；可能会需要后续的rollback

Strict Consistency：严格的一致性，需要通过global clock实现，在单处理器系统中容易实现，DS中很难实现

非常依赖于global time---可以通过将时间分为连续的time interval，根据时间的区间添加timestamp，但是对于同一个interval内的只能通过PID？进行比较 probably

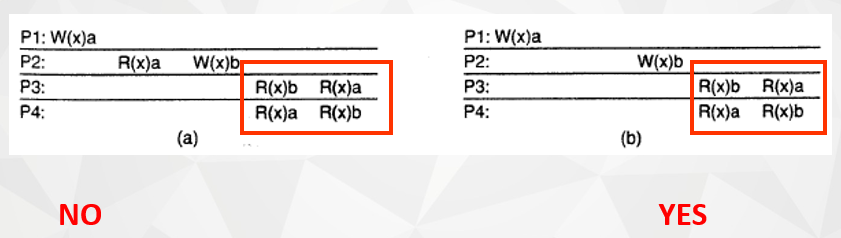
Sequential consistency：所有进程的读写操作都按照某个顺序进行操作，这个顺序由program进行定义（这个顺序可以是不对的，但是要被所有的进程认可）（一台机器上的顺序必须是固定的，不能被更改）

Linearizability：

如果每个操作都有一个timestamp，并且timestamp是同步的且符合sequential consistency，且如果一个操作的timestamp早于另一个操作，则timestamp早的操作应当在另一个操作之前完成，则认为这份数据是linearizable的

Causal consistency：所有结果可能相关的写操作在所有进程读取时必须是同一个顺序读出的，也就是说符合因果一致性的关系

比如下图中：在左图中P2读取x为a后对x写入b，可以认为是读取这个操作与后续的写入操作有因果关系，所以当后续的读取操作不符合先a后b的顺序时就不会被允许；但是右图中写入a和b不具有一定的因果关系，因此任意的读取顺序都会被允许



Grouping Operation：

别的算法都是单独在读写操作上实现的，这种方式通过对一组操作进行定义，通过提升操作的granularity达到应用的granularity

通过shared synchronization variable实现：类似于critical section的方法

Client-Centric Consistency model：

基于的假设是：所有的数据都不会自动进行更新；如果能够更新，问题就可以轻易地解决；更多的操作是对数据进行读取

Eventual consistency：最终会变成一致的

Concurrency在这种model下只会以特殊的形式呈现：

比如在数据库中只有很少的进程有写入的权限，大部分进程只能读取；DNS namespace的更新只能由部分人完成，不会有write-write conflict；web page也是一样的，只能由一个web master来更新，不需要解决write-write conflicts

Definition：

很多分布式系统对于不一致性的容忍度很高，如果长时间没有更新则所有的replica最后都会变成一致的，称为eventual consistency：

Data store在没有更新的情况下最后会变成consistent的

只要求更新要能够传播到所有的replica

Write-write conflict是很容易解决的

Cheap to implement

几种方案：

Monotonic read consistency：如果一个进程读取了一个data item X，则该进程任何连续对于X的读操作应当返回相同的值；一个进程对X的写操作在下一个连续的写操作之前应当被完成，在所有副本上都以这个顺序完成

记录WS(Xi(t))为对Xi(t)的操作；WS(Xi(t), Xj(t))为第一个操作传播至另一个副本

Read your writes consistency：

某个进程对于X的写操作的影响在被读取时应当以相同的顺序读取写的结果

Writes follow reads consistency：

当一个写操作在读操作之后时，对于X的写操作应当在最近被读的X的值上进行操作

Replica management：

Server placement：理论上讲应该被算出来的，虽然通常受限于商业因素，但还是可以进行优化

选择位置的原则：选择一个新的server时，使新的server位置被确定时所有用户到server的平均距离最小

每个区域都是想访问某些数据的集合，选择需求最大的区域

Permanent replicas：

初始的数据存储，别的都是这个replica的副本

Server-initiated replicas：data store的拥有者决定开始复制的过程

问题是决定何时何地某个replica应当被删除或者创建

e.g. Dynamic replication algorithm: 记录访问次数，如果访问很多的话可以动态地决定一个新的replica的位置

Client-initiated replicas：由client请求初始一个replica，在cache中储存一段固定的时间

Cache总应该被放在client端，也可以放在WAN中的某个机器上，称为cache server

Update propagation/content distribution：

三种方法：

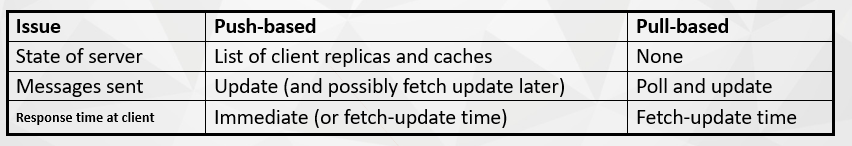
只传播update的一个notification---告知别的replicas其现在的数据副本已经无效

从一个copy将数据传给另一个---读写比例高时使用，可以通过发送日志实现

将update传播给彼得copies---告诉别的copies如何进行update

Push protocols：update传输至别的replicas

Pull protocols：replica自身询问是否有update



Lease：lease指的是server会在固定的一段时间后向client push updates

当lease expired，client需要向server进行轮询并进行修改(如果需要的话)，如果没有过期的话，server还是会进行push updates。相当于交钱享受push的服务

Communication：unicast，multicast

单播通过N条消息向N个replica进行update

多播通过底层的网络支持实现，一次对多个replica进行update

Epidemic protocol：为了实现eventual consistency的协议，用更少的消息实现全部更新

分为三种节点：如果一个server已经知道了update并且想要传播，称为infective；如果一个server并不知道update，则称为susceptible；如果一个节点已经知道了更新并且不想传播，则称为removed

Anti-entropy model：

一个server P随机选择另一个server Q并且交换updates

可以是push-based，pull-based和push-pull approach

Pull-based的效率比push-based更高，因为会有conflict

Gossiping Model：如果一个server有了update，随机选择另一个server push update。如果对方已经得知了update，则停止传播的过程。

问题是不能保证所有的server都获取到更新

优点是scalability，不需要很多的synchronization就可以进行传播

缺点是propagation dilemma：如果一个老版本的update晚于删除到达，则会被当成是update来处理，出现inconsistency---通过记录被删除的数据一段时间来解决(类似回收站的概念)。可以设置一个death certificate，在过了这个时间之后会彻底删除。

Consistency protocols：

Bounding numerical deviation：

控制一个deviation在某个bound之内，可以是numerical deviation，staleness deviation或ordering deviation

Primary based protocol：

每个data item有一个primary X，primary负责同步对于X的操作

Remote-write protocols：所有的写操作都在一个特殊的服务器上完成；数据不能复制或者移动

Primary backup protocols：允许副本产生，但是每次写操作都需要被传播到某个固定的primary

Local-write protocols：每个data item X都有一个unique replica，想要读写的话把primary移动到client附近

问题是需要keep trace of every data item

也可以允许副本的存在，但每个副本的更新都需要传送给primary进行。primary写入时需要对所有的副本进行通知---这样的话连续的写操作可以同步进行

Replicated write protocols：没有primary，写操作可以在多个replicas上进行，一个写操作可以对任意一个replica进行

Active replication：每个replica都有一个负责进行update的进程

Ordering又成了问题，可以通过sequencer解决

Quorum-based protocols：进行读写操作时，通过voting来实现

在读的时候需要询问足够多的servers for data version number

如果所有的版本号一致，则是最新的版本

记录写入的数量为NW，读取为NR：限制是NR + NW 的值要比N大，NW要比N/2大

