6.033 计算机系统工程

2009年春季

6.033 第三讲：命名

系统
  系统 = 一堆资源，通过名称粘合在一起
    通常是查看或构建的最佳方式
    通常最有趣的操作在名称中
  [带有组件和连接的云]
  资源是什么？
    通常可以将它们视为三种基本类型

* 基本抽象
  存储
    内存，磁盘
    分层：数据结构，文件系统，磁盘阵列
  解释器
    CPU
    编程语言例如Java虚拟机
  通信
    电线，以太网
    分层：互联网，SSH，Unix管道
  我们将在课程中讨论所有这些抽象

为什么使用这些抽象？
  许多组件适应这些模具
  它们很好地协同工作
  硬件版本便宜
  对于这些抽象，已知一般技术
    以提高性能，容错性和功能性

名称
  所有三个抽象都依赖于名称
    内存存储命名对象
    解释器操作命名对象
    通信链路连接命名实体
  系统设计的很大一部分是命名 --- 粘合剂

DNS示例
  web.mit.edu是什么意思？
  你可以查找并将其转换为IP地址
    18.7.22.69
    互联网知道如何将数据发送到IP地址
  翻译是如何工作的？

大局观
  我的笔记本操作系统附带了一些"根名称服务器"的IP地址
    这些是硬编码在一个文件中的，大约有十几个
    198.41.0.4（弗吉尼亚）
    128.63.2.53（马里兰）
    ...
    这个列表不会经常变化，否则旧计算机会出问题
  我的笔记本选择一个根服务器，向其发送查询，
    请求字符串"web.mit.edu"的IP地址
  每个根服务器都有完全相同的表，即根"区域"
    edu NS 192.5.6.30
    edu NS 192.12.94.30
    com NS 192.48.79.30

...
查找与"web.mit.edu"后缀匹配的根区条目
将这些记录返回给我的笔记本电脑
现在我的笔记本电脑知道该向谁询问有关edu的信息
重新将请求发送到其中一个edu服务器
每个edu服务器都有edu区的副本
nyu.edu NS 128.122.253.83
mit.edu NS 18.72.0.3
mit.edu NS 18.70.0.160
将两个MIT记录返回给我的笔记本电脑
现在我的笔记本电脑将请求转发到一个MIT服务器
每个MIT服务器都有mit.edu区的副本
web.mit.edu A 18.7.22.69
bitsy.mit.edu A 18.72.0.3
csail.mit.edu NS 18.24.0.120
...

DNS具有分层命名空间
树状图

区域文件从哪里来?
每个区域都由独立管理
根区由国际非营利组织IANA管理
com区由VeriSign公司管理
麻省理工学院维护其区域文件，在其服务器上进行更新

为什么选择这个设计?
每个人都看到相同的命名空间（至少对于完整的名称）
性能可扩展
通过简单性，根服务器可以回答许多请求
通过缓存，这里没有真正描述
通过委托（保持表格大小）
与单个中央服务器形成对比
管理可扩展
麻省理工学院通过运行自己的服务器来控制自己的名称
容错性

存在问题:
谁应该控制根域名？ .com?
对根服务器的负载
拒绝服务攻击
安全问题：客户端如何知道数据是否正确?
当我要求更改NS时，Verisign如何知道我是ibm.com?

DNS非常成功
已有25年历史
设计在1000倍缩放后仍然有效
您可以从笔记的第4章了解更多关于DNS的内容

系统如何使用DNS?
用户友好的命名（输入URL）
网页告诉浏览器在按钮按下时要做什么
即连接一个页面到下一个页面
Adobe PDF Reader包含以后要检查升级的DNS名称
允许服务更改服务器的IP地址（更改托管服务）
DNS服务器可以向不同的客户端提供不同的答案

将www.yahoo.com转换为靠近客户端的IP地址

命名目标
- 让我们退后一步
- 1. 用户友好
- 2. 共享
  - 图表：A和B都知道N，指的是...
- 3. 检索（跨时间共享）
  - A知道N，两次引用...随时间变化
- 4. 间接
  - 图表：A知道固定N，引用可变N'，引用...
- 5. 隐藏
  - 隐藏实现
  - 控制访问（只有知道名称或挂钩到手动ACL才能访问）

命名系统很常见，因为非常强大
- 电话号码
- 文件/目录
- 电子邮件地址
- 程序中的函数/变量名

示例：虚拟寻址
- 通常称为"虚拟内存"或"分页"

为什么选择这个例子?
- 命名机制
- 设计良好
- 解决了许多问题
- 是许多精彩操作系统功能的基础

内存
- 这是来自6.004的复习
- 也是笔记中的第5章
- 图表：CPU，地址/数据线，DRAM
- DRAM是字节数组
- 使用"物理地址"进行索引
- 假设是32位地址
- 最多可以有4GB的DRAM
- 示例：LD R4，0x2020

为什么物理寻址不够?
- 有很多原因，我们将会看到
- 这里有一个原因：如果程序太大无法放入DRAM中怎么办?
  - 但仍然适合2^32字节
  - 例如，两兆字节的程序，只有一兆字节的DRAM

需求分页
- diagram: cpu, dram, disk
- 希望一些数据在内存中，一些在磁盘上
  - 当程序引用不在内存中的数据时，
  - 将其他数据从内存移到磁盘上，
  - 从磁盘加载所需的数据到内存中
  - 称为"需求分页"
- 这是最初的动机，现在不太相关了

虚拟寻址思想

diagram: MMU, 两个内存单元数组
从左到右的一组箭头
"虚拟地址" vs "物理地址"
软件仅使用虚拟地址进行加载/存储
物理地址仅出现在映射表中
概念上，对于所有2^32个地址进行转换
一些引用物理内存
其他引用磁盘位置
如何实现?
如果按字节映射，16 GB!
每个字节地址都有一个翻译表是不切实际的

页表
如何使映射表变小?
将虚拟地址分成连续对齐的"页"
上面（比如）20位是"虚拟页号"
MMU只将页号映射到物理地址的前20位
然后使用低12位虚拟地址位作为低12位物理地址位

例子:
页表:
0: 3 (即物理页3)
1: 0
2: 4
3: 5
寄存器R3包含0x2020
LD R4, (R3)
CPU将（R3）= 0x2020发送到MMU
虚拟页号为2
偏移量为0x20
因此物理地址= 4 * 4096 + 0x20 = 0x4020
MMU将0x4020通过线路发送到DRAM系统

页表足够小，可以放入DRAM中
有2^20个条目，每个条目可能是4个字节
现在页表只有4 MB，不算太大

页表在哪里?
在内存中
因此可以用普通指令进行修改
CPU页表基址寄存器指向基址
因此，内存引用在概念上需要*两次*物理内存获取
一次获取页表项
一次获取实际数据
允许我们快速切换多个页表
以后会派上用场...

标志
"页表项"
20位，加上一些标志
有效/无效，可写/只读
MMU仅在有效且正确读/写时进行转换
否则会强制发生"页错误":
保存状态
转移到操作系统中已知的"处理程序"函数

需求分页
页表：
0: 1, V
1: -, I
2: 0, V
而且只有两页物理内存！
如果程序使用va 0x0xxx或0x2xxx，MMU进行转换
如果程序使用va 0x1xxx，则发生错误
处理程序(vpn)：
(ppn, vpn') = 选择一个驻留页进行驱逐
写dram[ppn] -> disk[vpn']
pagetable[vpn'] = -, I
读dram[ppn] <- disk[vpn]
pagetable[vpn] = ppn, V

需求分页讨论
选择要驱逐的页面需要巧妙的方法
e.g. 最近最少使用
我们将DRAM视为磁盘上"真实"内存的缓存
可以这样做是因为程序使用"名称"(va)来表示内存
我们对名称的含义有完全控制权
映射到DRAM
或者在页面错误处理程序中进行任意操作

操作系统使用页表做什么？
需求分页
大型程序的延迟加载
零填充
模块化的地址空间，防止错误
每个运行中程序一个页表
一个程序甚至不能命名另一个程序的内存！
用于快速通信的共享内存
对同一物理页面有两个不同的名称
对共享内容有严格控制
写时复制分叉
分布式共享内存

结束
注意命名系统
在自己的工作中要注意
何时引入命名可能有帮助

****

在下一天开始时完成

以虚拟寻址作为命名的例子
已经证明是强大的，一个基本机制有多种用途
图表：CPU，MMU，RAM，简化的映射表
程序使用虚拟地址"命名"数据
映射有效（指DRAM），或导致操作系统"页错误"
o/s填写条目，返回
程序从未知晓
你可以用这个命名机制做什么？
从磁盘获取需求页面：支持大型程序
从磁盘开始懒加载

地址空间：不同程序的不同映射
    o/s根据当前运行的程序进行切换
    甚至不能*命名*彼此的内存
    包含错误
共享内存，例如x客户端和服务器（在同一台机器上）
分布式共享内存
    在一组独立机器上构建并行程序
        但通过共享内存，所有都是一个大程序
    图表：绘制其他机器，其RAM和映射
    页错误：获取，标记为无效

6.033 计算机系统工程

2009年春季

6.033讲座4：客户端/服务器

模块化
  如何对复杂程序进行排序?
  分解为模块
  [大型Therac-25块=>组件，交互]
  目标：解耦，缩小交互集

模块化（2）
  模块化采取什么形式?
  交互是过程调用
    C -> P，P返回给C
  过程澄清接口，隐藏实现
    P(a) { ... }
    C() { ... y = P(x); ... }

强制执行吗?
  接口是否被强制执行?
  实现是否被隐藏?

当C调用P时会发生什么?
  6.004
  它们通过共享的堆栈在内存中进行通信
  [堆栈：寄存器，参数，返回地址，P的变量，可能还有参数...]
  C：推送寄存器，推送参数，推送PC+1，跳转到P，弹出参数，弹出寄存器，...R0
  P：推送变量，...，弹出变量，将xx移动到R0，弹出PC

    调用合约
      P返回
  P返回到C指定的位置
  P恢复堆栈指针
  P不修改堆栈（或C的其他内存）
  P不会使机器崩溃（例如使用完所有堆内存）

软模块化
  在低级别上，调用合约的任何部分都不被强制执行!
  希望：规范+合约
  规范：我们无法强制执行（例如sqrt()是否返回正确答案？）
  合约：纯机械的，我们可以尝试强制执行
  目标：程序员只需关注规范，而不是合约
  == 强制模块化
    有很多方法来实现模块化
    今天我们将看一种方法

客户端/服务器
  注意，共享同一台机器和内存可能会导致很多问题
  因此：将C和P放在不同的机器上
  只通过消息进行交互
  [图表：两个方框之间有一根线]
  示例：Web客户端/服务器，AFS，X窗口系统（即将阅读）

代码+时间图
  客户端：
    将参数放入消息中（例如URL）
    发送消息
    等待回复

从回复中读取结果
服务器：
等待请求消息
获取参数
计算...
将结果放入回复消息中（例如index.html的内容）发送回复
返回起始位置

C/S的好处
1. 保护控制流（例如返回地址）
2. 保护私有数据
3. 没有命运共享（崩溃和死锁不会传播）
4. 强制窄规范（无论好坏，没有全局变量）
C/S强制执行大部分调用契约
错误仍然可以通过消息传播
但这是一个相对受限制的通道
比检查整个内存要容易
规范仍然是程序员的问题：确保服务器返回正确的答案

C/S有助于多方参与
事实证明，C/S具有更多好处，不仅仅是强制模块化
[图表：emacs，AFS，打印]
工作站上的emacs，AFS，打印服务器
工作站使用多个服务器
AFS有多个客户端
打印服务器可能是AFS的服务器和客户端
例如，客户端将文件名发送到打印服务器

AFS服务器是一个"可信中介"
1. 从多个物理位置访问您的数据
2. 与其他客户端共享
3. 控制共享（私有数据与公共数据，访问控制列表）
私有，好友，公共
一个客户端中的错误不会影响其他客户端
4. 服务器物理安全，可靠，易于找到
这些C/S的好处与强制模块化一样重要
例如，电子邮件服务器，eBay

RPC
C/S涉及许多令人讨厌且容易出错的消息格式化等
为什么不将细节隐藏在真实的过程接口后面?

例子：
想象这是你原始的单程序 eBay 实现：

```
ui(){
  打印出价（"123"，10.00）；
}
出价（物品，金额）{
  ...
  返回获胜者；
}
```

你想在不重写这段代码的情况下在出价()处分割

客户端存根
想法：一个*看起来*像出价()但发送消息的过程
客户端存根：

```
    出价（物品，金额）{
      将物品和金额放入消息中；
      发送消息；
      等待回复；
      获胜者 <- 回复消息；
      返回获胜者；
  现在原始的ui()正在使用一个服务器!

服务器存根
  想在服务器上使用原始的出价()代码
  服务器存根：
  调度(){
    while(1){
      读取消息
      物品，金额 <- 消息；
      w = 出价(物品，金额)
      消息 <- w；
      发送消息；
    }
  }

编组/解组
  需要一种标准的方法将数据类型放入消息中
    消息是一系列字节
  整数的标准大小，字符串和数组的平面布局
  典型的请求消息格式：
    proc# id 3 '1' '2' '3' 10.00???

自动生成存根
  用于查看参数和返回类型的工具：在bid()上运行它
  生成编组和解组代码
  生成存根过程
  保存编程
  确保例如参数类型的一致性

RPC在简化客户端/服务器方面非常成功，但是

RPC != PC
  尽管语法相似
  亚马逊网络，仓库后端
  [时间线]
  用户（浏览器）想要下订单
  RPC 1：检查库存（isbn）-> 计数
    如果网络丢失/损坏请求怎么办?
    丢失回复?
    仓库崩溃?
    存根可以通过超时+重发来隐藏这些故障
      在存根周围包装循环
    导致重复--没问题，没有副作用
  RPC 2：发货（isbn，地址）-> 是/否
    请求丢失?
    回复丢失?
    仓库崩溃?
    存根可以通过重试来隐藏这些故障吗?
    重复是否可以接受?
```

RPC系统如何处理故障?
  一些常用的方法，通常称为"故障语义"
  1. 至少一次
     客户端持续重新发送直到服务器响应
     -> 多次
     -> 错误返回和成功
     仅对没有副作用的RPC有效，例如检查库存
  2. 最多一次
     客户端不断重新发送（可能超时+错误）
     服务器记住请求并抑制重复
     -> 错误返回和成功
     对于ship()来说可以吗?
       如果我们还想在书籍发货时扣款，那就不行
       如果客户端放弃或崩溃，书籍发货了吗?
  3. 确切一次
     这通常是我们真正想要的：只是做它
     以有用的方式很难做到，稍后会看到实际版本
  大多数RPC系统执行#1或#2

RMI代码幻灯片
  将第一部分输入到rmic存根生成器中
    它为main()生成BidInterface的实现
    和服务器分派存根
    服务器存根调用您的bid()实现（未显示）
  公共接口确保客户端/服务器就参数（消息格式）达成一致
  对bid()的过程调用看起来非常普通!
    但是在它周围有非标准的方面
  Naming.lookup() -- 与哪个服务器通信?
    服务器已向命名系统注册
    lookup()返回一个代理BidInterface对象
    具有bid()方法，还记住服务器身份
  try/catch
    这是新的，每个远程调用都必须有这个，用于超时/无回复

摘要
  通过C/S实施强制模块化
  RPC自动化细节
  RPC语义不同
  下一步：在同一台机器上实施强制模块化

RMI代码幻灯片（l4.ppt）：

```
public interface BidInterface extends Remote {
  public String bid(String) throws RemoteException;
}

public static void main (String[] argv) {
    try {
      BidInterface srvr = (BidInterface)
           Naming.lookup("//xxx.ebay.com/Bid");
      winning = srvr.bid("123");
      System.out.println(winning);
    } catch (Exception e) {
      System.out.println ("BidClient exception: " + e);
    }
  }
```

6.033 计算机系统工程

2009年春季

6.033 第5讲：操作系统组织

接下来几节课的计划
  总体主题：模块化
  刚刚讨论了每台计算机一个模块（c/s）
    稍后在课程中详细介绍基于网络的c/s
  现在：同一台计算机内的模块
    L5：操作系统组织 - 实现模块化的工具
    L6：并发性
    L7：线程
  操作系统本身是一个有趣的设计产物

操作系统的目标
  多路复用：一个计算机，多个程序
  合作：帮助程序相互交互/通信/共享数据
  保护（错误，隐私）
  可移植性（隐藏硬件差异和例如RAM的数量）
  性能（改进，并且不妨碍）

实现这些目标的关键技术是什么？
  虚拟化
  抽象化

    虚拟化
  计算机具有硬件资源
  几个CPU，一个内存系统，一个磁盘，一个网络接口，一个显示器
  但是你想运行多个程序（X，编辑器，编译器，浏览器等）
  思路：给每个程序一组“虚拟”资源
    就像它拥有自己的计算机一样
    CPU，内存

虚拟化示例：CPU
  将一个CPU多路复用给多个程序
  因此，每个程序都认为它有一个专用的CPU
  将CPU依次分配给每个程序的1/n的时间
  硬件定时器每秒中断100次
  保存当前程序的寄存器（CPU寄存器）
  加载先前保存的另一个程序的寄存器
  继续执行
  对程序透明！
  在接下来的几堂课中还有很多要说的

    一些系统将所有硬件虚拟化
      "虚拟机监视器"
  给每个子系统（程序？）一个完整的虚拟计算机
    精确模拟完整的硬件接口
  早期的IBM操作系统以这种方式共享计算机
    一个计算机，多个虚拟机
    每个用户可以运行自己的操作系统
  Mac上的VMWare，Parallels，Microsoft Virtual PC
  仅仅虚拟化会使合作变得困难
    编译器希望看到编辑器的输出
    如果每个人都认为自己有一个专用的虚拟磁盘，那就不那么容易了

抽象
  抽象：虚拟化但改变接口

改变接口以实现共享、合作和可移植性

抽象的例子：
真正的"抽象虚拟资源"
磁盘 -> 文件系统
显示器 -> 窗口系统
CPU -> 只有"安全"指令，例如不能访问MMU、时钟等
??? -> 管道

抽象是什么样子的?
通常以一组"系统调用"的形式呈现给程序员
请参见幻灯片（l5.ppt）
你熟悉UNIX论文中的FS
chdir要求内核更改目录
代码的其余部分读取文件
系统调用看起来像是过程，我们将看到它在内部是不同的

```
主函数() {
  int fd, n;
  char buf[512];

  chdir("/usr/rtm");

  fd = open("quiz.txt", 0);
  n = read(fd, buf, 512);
  write(1, buf, n);
  close(fd);
}
```

如何管理/实现抽象?
如何强制模块化?
例如，希望程序仅通过系统调用使用FS
机制：内核与用户
[图表：内核，程序，磁盘等]
内核可以对硬件做任何喜欢的事情
用户程序只能直接使用自己的内存，不能使用其他任何东西
通过系统调用要求内核执行操作
通过这种安排，执行更加清晰，分为两个部分：
将程序保留在其自己的地址空间内
控制用户和内核之间的转移

强制执行地址空间边界
使用每个程序的页面映射来禁止地址空间外的写入
内核在切换程序时切换页面映射
这就是操作系统保护一个程序免受另一个程序影响的方式
程序的地址空间分割
内核在高地址
用户在低地址
PTEs有两组访问标志：KR，KW，UR，UW
所有页面都有KR，KW
只有用户页面有UR，UW
方便：允许内核直接使用用户内存来实现系统调用

CPU有一个"监管模式"标志
如果打开，Kx PTE标志适用，修改MMU，与设备通信
如果关闭，Ux PTE标志，没有设备

需要确保只有在执行内核代码时才将监管模式设置为打开

从用户到内核的有序转移
1. 设置监管标志
2. 切换地址空间
3. 跳转到内核
并防止用户随意跳转到内核中的任何位置
并且不能信任/使用用户堆栈
不能信任用户提供的任何内容，例如系统调用参数

系统调用机制
应用程序：
```
chdir("/usr/rtm")
  R0 <- 12（系统调用号）
  R1 <- "/usr/rtm" 的地址
  SYSCALL
```
SYSCALL 指令：
保存用户程序状态（PC，SP）
设置超级用户模式标志
跳转到内核系统调用处理程序（固定位置）
内核系统调用处理程序：
在每个程序表中保存用户寄存器
将 SP 设置为内核堆栈
调用 sys_chdir()，一个普通的 C 函数
...
恢复用户寄存器
SYSRET
SYSRET 指令：
清除超级用户模式标志
恢复用户 PC，SP
继续进程执行

这是强制模块化的过程调用
虽然只保护内核免受用户程序的影响
用户程序只能做两件事
只能使用自己的内存
或者跳入内核，但此时内核负责
SYSCALL 结合了设置超级用户模式和跳转到已知位置
如果让用户程序指定目标，将是一场灾难

如何使用内核的强制模块化？
如何为系统服务安排模块边界？
两个主要派别：微内核 vs 单体内核

微内核的愿景
将主要操作系统抽象实现为用户空间服务器文件系统、网络、窗口/图形界面应用程序、服务器通过消息进行交互内核提供最低限度的
支持服务器进程 - 地址空间 + 运行程序大部分交互通过消息进行

只有两个主要系统调用：发送（send()）和接收（recv()）

为什么微内核有吸引力？
优雅、简单

直接支持客户端/服务器
内核很小，bug较少
内核可以优化以做好一件事（消息）
不同服务器之间有严格的模块化，减少bug
统一的端口方案 -> 统一的安全计划
通过给予（或不给予）端口访问权限来控制访问
更容易分发（消息允许在其他主机上提供服务）

这个想法去哪了？
实现在1990年左右很慢 - 例如Mach
很多消息，特别是如果有很多服务器
往往会得到几个巨大的服务器，所以几乎没有模块化的优势
消息的想法很有影响力，例如苹果的OSX

单体内核 - 到目前为止，这是获胜的设计
将许多服务器分开太麻烦
拥有一个大程序更容易和更高效

Linux内核中有什么？

文件描述符 进程

文件系统

磁盘
终端网络缓存

驱动程序 分配

以太网磁盘 内存、CPU、内存管理单元

复杂：300多个系统调用，不仅仅是两个以上的微内核
大部分复杂性隐藏在统一接口之下 - 面向对象的风格
例如，所有存储设备对文件系统都是相同的
USB闪存驱动器、硬盘
对于网络代码来说，所有网络硬件都是相同的
所以没有看起来那么复杂

为什么单体内核大多数情况下胜过微内核？
传统，微内核的优势微妙
效率：过程调用或内存引用而不是消息
效率：通过模块之间的集成获得更好的算法
在文件系统缓存和进程内存之间取得平衡
容易添加微内核风格的消息传递、用户级服务器
X、sshd、DNS、apache、数据库、打印机

什么在[单片机?]内核中不太好运行？
设备驱动程序有错误，导致崩溃
有很多规则，例如用作参数传递的用户指针
容易出错的编程环境
在共享资源管理（内存，磁盘）方面表现不佳
编译器使用大量内存->笔记本电脑无法使用
备份程序使用磁盘->笔记本电脑无法使用（所以我不备份...）
模块化对资源管理来说是软性的

摘要

操作系统为共享/多路复用进行虚拟化
为可移植性和协作提供抽象
以两种有用的方式提供强制模块化：
    程序/内核
    程序/程序
支持两种类型的c/s：程序->内核，程序->程序
接下来：单台计算机上的客户端/服务器的具体技术

6.033 计算机系统工程

2009年春季

6.033 讲座6：单台计算机上的客户端/服务器

介绍
  如何在一台计算机上实现c/s
  本身就很有价值
  涉及并发性，是一个独立有趣的主题
    DP1完全关于并发性

我们想要什么?
  [图表：X客户端，X服务器，尚无内核]
  客户端想要发送例如图像到服务器
  目标：保持距离，使X服务器不易受到客户端错误的影响

思路：让内核管理交互
  客户端/服务器与可信内核进行交互，而不是彼此
  [图表：X客户端，X服务器，内核]
  让我们专注于单向流动（可以使用两个进行远程过程调用）
  内核中的缓冲内存
    每个条目保存一个消息指针
  send(m)将消息添加到缓冲区
  receive(m)从缓冲区读取消息
  有限的缓冲区，因此send()可能需要等待
  缓冲区可能为空，因此receive()可能需要等待
  为什么缓冲区有多个条目?
    发送者/接收者速率可能围绕平均值变化
    当发送者更快时，让发送者积累积压
                  这样接收者在发送者较慢时有输入
  非常类似于UNIX管道

问题：并发性
  内核内部的某些数据结构，s()和r()对其进行修改
  如果s()和r()同时活动会怎样? 可能会干扰
  并发性是一个巨大的问题，会多次出现
  让我们从简单开始：
    每个程序都有自己的CPU
    只有一个内存系统
  [图表：两个CPU，一个内存]
  系统调用在两个CPU上运行!
    即如果程序A调用send()，send()将在A的CPU上运行
  send()和receive()通过单个共享内存系统进行交互

数据结构
  "有界缓冲区"
  [图表：缓冲区[5]，输入，输出]
  每个数组条目：指向消息缓冲区的指针
  输入：放入BB的消息数
  输出：从BB中读取的消息数
  IN mod 5是send()写入的下一个位置
  OUT mod 5是receive()查找的下一个位置
  示例：in = 28，out = 26
    有两个等待的消息，位置1和位置2
  in > out => BB非空
  in - out < 5 => 非满

send()代码幻灯片
  p是"端口"，指向BB的实例，因此我们可以有很多个

例如每个c/s对或每个UNIX管道一个
循环等待直到有空间（"忙等待"）
写入槽
增加输入计数

receive()代码幻灯片
循环等待直到发送的次数大于接收的次数
如果有消息
在复制消息之后增加p.out
因为p.out++可能会导致send()覆盖
如果send()正在等待空间

我相信这个简单的BB代码是有效的
[显示同时显示两个幻灯片]
即使同时调用send()和receive()
并发很少能够如此顺利工作!

简单的BB发送/接收的假设
1. 一个发送者，一个接收者
2. 每个都有自己的CPU（否则循环会阻止其他运行）
3. 输入和输出不会溢出
4. CPU按照程序顺序执行内存读写
哎呀！这段代码可能不会按照所示的方式工作!
编译器可能会将in/out放入寄存器中，看不到其他的更改
CPU可能在写入buffer[]之前增加p.in
我将假设内存读写按照程序顺序执行

假设我们想要多个发送者
例如，许多客户端可以向X服务器发送消息
我们的send()会起作用吗?

并发发送()
A: 发送(p, m1) B: 发送(p, m2)
我们希望发生什么?
最有用的行为是什么?
目标:
buf中有两个消息，in == 2
我们不关心顺序

并发发送()的示例问题
在不同的CPU上，同时在同一个p上

```
A             B
r in, out
              r in, out
w buf[0]
              w buf[0]
r in=0
              r in=0
w in=1
              w in=1
```

结果：in = 1，有一个消息在有界缓冲区中，另一个消息丢失了!

这种类型的错误被称为"竞争"
一旦A将数据放入缓冲区，它必须赶快完成对p.in的增加!

代码中的其他竞争

假设只剩下一个插槽
A和B可能都观察到in - out < N
将*两个*项目放入buf，覆盖最旧的条目

种族是一个严重的问题
容易犯的错误 - send()看起来完全合理!
很难找到
取决于时间，可能很少发生
例如Therac-25，只有经验丰富的操作员打字速度足够快

如何修复send()的竞争条件?
原始代码假设没有其他人干扰p.in等
一次只有一个CPU在send()中
==隔离执行
我们能恢复那种隔离吗?

锁
锁是一个具有两个操作的数据类型
获取(l)
s1
s2
释放(l)
锁包含状态：锁定或解锁
如果您尝试获取一个已锁定的锁
获取将等待直到它被释放
如果两个acquire()同时尝试获取锁
一个将成功，另一个将等待

如何使用锁修复send()?
[锁定send()幻灯片]
将锁与每个BB关联
在使用BB之前获取锁
在使用BB完成后释放
高级视图：
不会交错多个send()的执行
只有一个send()会执行send()的内部
如果单发送者的send()是正确的，那么很可能是正确的

send()如何使用锁有关吗?
将acquire移动到IF之后? [幻灯片]

为什么每个有界缓冲区都有单独的锁?
而不是所有的BB都使用相同的锁
这样只允许一个BB处于活动状态
但是如果不同BB上的send()是并发的，那是可以的
每个BB都有一个锁可以提高性能/并行性

死锁
大型程序可能有数千个锁，一些是每个模块的一旦系统中有多个锁，就必须
担心死锁问题

死锁：两个CPU都有一个锁，每个都在等待另一个释放
例子：
实现文件系统
需要确保两个程序不会同时修改一个目录
每个目录都有一个锁

```
    create(d, name):
      获取d.lock
      如果名称存在:
        错误
      否则
        为名称创建目录实体
      释放d.lock
  将文件从一个目录移动到另一个目录怎么样? 像mv一样
    move(d1, name, d2):
      获取d1.lock
      获取d2.lock
      从d1中删除名称
      将名称添加到d2
      释放d2.lock
      释放d1.lock
  这里有什么问题?

避免死锁
  查找所有持有多个锁的位置
  确保对于每个位置，它们按相同的顺序获取
    这样就不会出现锁环和死锁
  对于move()函数:
    按i节点号对目录进行排序
    先锁定较低的i节点号
    所以:
      如果d1.inum < d2.inum:
        获取d1.lock
        获取d2.lock
      否则:
        获取d2.lock
        获取d1.lock
  这可能很痛苦: 需要全局推理
    获取l1
    打印("...")
  print()函数是否获取锁? 它会与l1发生死锁吗?
  好消息是，一旦发生死锁，它们就不会很微妙

锁粒度
  如何决定有多少个锁，它们应该保护什么?
  一个光谱，粗粒度 vs 细粒度
  粗粒度:
    只有一个锁，或者每个模块一个锁
    例如，整个文件系统一个锁
    更有可能是正确的
    但是CPU可能会等待/自旋等待锁，浪费CPU时间
    "串行执行"，性能不比一个CPU好
  细粒度:
    将数据分成许多片段，每个片段都有一个单独的锁
    不同的CPU可以使用不同的数据，不同的锁
      并行操作，完成更多的工作
    但是更难正确实现
    需要更多的思考来确保对不同片段的操作不会相互影响
      例如，在目录之间移动时发生死锁
  始终从粗粒度开始!
    只有在低并行性能的情况下才使用多个锁
```

如何实现获取和释放?

这是一个行不通的计划:

```
获取(l)
  当 l == 0 时
          什么都不做
  l = 1
释放(l)
  l = 0
```

存在一个熟悉的竞争条件:

A 和 B 都看到 l = 0
A 和 B 都设置 l = 1
A 和 B 都持有锁!

如果我们能够进行l==0的测试和l=1的不可分割...

大多数CPU提供我们需要的不可分割指令!
根据CPU的不同，但通常类似于:

```
RSM(a)
  r <- mem[a]
  mem[a] <- 1
  return r
```

将内存设置为1，返回旧值
RSM = 读取和设置内存

硬件如何使RSM不可分割?

一个简单的计划
两个CPU，一个总线，内存
只有一个总线，只有一个CPU可以使用它
有一个仲裁器来决定
所以CPU抢占总线，进行读取和写入，释放总线
仲裁器强制一个RSM在另一个开始之前完成

如何使用RSM进行锁定?

```
获取(l)
  while RSM(l) == 1
    do nothing
```

总是将锁设置为1
  如果已经锁定：无害
  如果未锁定：锁定
RSM返回0当且仅当未锁定
一组并发CPU中只有一个会看到0
小提示：你可以使用普通的LOAD和STORE来实现锁
  即没有硬件支持的RSM
  只是有点笨拙和慢
  查找Dekker算法

摘要

BB是你在一台计算机上进行客户端/服务器的需要
并发编程很困难
锁可以帮助并发看起来更顺序
注意死锁
下一步：一个CPU上的多个程序

6.033 计算机系统工程

2009年春季

6.003讲座7：线程和条件变量

主题：虚拟处理器/线程
  星期一：客户端/服务器/有界缓冲区，每个程序一个CPU
  今天：比CPU多的程序
    只有一个CPU（不忙等！）
    或者几个CPU，更多的程序
  也比CPU少的程序（CPU可能需要空闲！）
  目标：虚拟化处理器
    在许多"线程"之间复用CPU

线程抽象
  可运行程序的状态
  所以CPU复用==挂起X，恢复Y，挂起Y，恢复X
  还有其他用于复用CPU的抽象
    这是一个有用且传统的抽象
  由"线程管理器"或"调度器"控制
  所需状态是什么？
  如何保存挂起的状态？
  如何从保存的状态恢复？

来自上一堂课的send()函数
  说明了为什么我们需要线程和多路复用[发送幻灯片]

  循环等待BB非满
  消耗大量CPU时间
  如果只有一个CPU，可能永远不会运行receive()函数！
  我们希望让receive()函数运行...

使用yield的send函数[发
  送/接收幻灯片]yield()
  函数放弃CPU让其他线程运行

    例如，一个receive()函数可能一直在等待并调用yield()
  总有一天yield()函数会返回
    在其他线程调用yield()函数之后
    例如，它尝试接收()函数，但是BB是空的

如何实现yield()函数？
  yield()函数是线程实现的核心
  挂起一个线程，恢复另一个线程

数据：
  threads[]表：状态，sp
  线程堆栈0，堆栈1，...
  cpus[]表：线程
  t_lock（粗粒度...）

[yield版本1幻灯片]

yield()函数中发生了什么？
  send函数调用yield()函数
  它如何知道哪个线程正在运行？
    每个CPU寄存器CPU()包含cpu#
    cpus[]表示每个CPU上正在发生的情况
  RUNNING -> RUNNABLE

    RUNNING表示某个CPU正在执行它
    RUNNABLE表示未执行，但可能执行
  保存SP（CPU寄存器）
  寻找一个不同的线程来运行
    忽略RUNNING状态的线程
  将“新”线程标记为RUNNING，以便其他CPU不运行它
  恢复新线程的保存的SP
    即将其加载到CPU的SP寄存器中
  返回

问题：
  yield保存了什么状态？
    只有SP
    栈上有什么？ 局部变量，RA，send()的保存寄存器等
    我们可能还需要保存/恢复被调用者保存的寄存器
  恢复后返回时会发生什么？
  这种对SP的使用可能不起作用，取决于编译器
    我假设编译后的代码在yield()的主体中不会改变SP
    并且返回基本上只是从堆栈中弹出RA
    在实际情况中更复杂
  t_lock保护了什么？
    不可分割的.state和.sp集合
    不可分割的查找RUNNABLE并将其标记为RUNNING
    在我们切换之前，不要让另一个CPU获取当前堆栈

有问题吗？

激励通知/等待
  [使用yield幻灯片发送]
  send()和receive()仍然消耗CPU时间
    e.g. send()等待接收方在BB中释放一个插槽
    e.g. receive()等待BB非空
    如果有许多线程等待，重复使用yield()会很昂贵
  希望send能够自我挂起
    当有空间时，让receive唤醒它
    do it in a general way
    不希望receive必须知道所有等待在send中的线程

"条件变量"
  作为会合点的对象
  两个方法：
    wait(cvar, lock) -- 释放(lock)，让出控制权，notify(cvar)后返回
    notify(cvar) -- 唤醒当前在wait(cvar)中等待的所有线程
  notify没有记忆性：如果没有线程在wait()，则没有任何效果
    先wait()再notify()：wait返回
    先notify()再wait()：wait不返回

每个BB都有两个条件变量：
  notfull（如果满了，send在此等待）
  notempty（如果为空，recv在此等待）

[使用wait/notify发送]
  如果满了，就等待，接收方总会释放一个槽位并通知(p.notfull)
  在循环中等待，等待返回后重新检查
    可能有多个发送方在等待，但只有一个槽位被释放
    也就是说，wait()的返回只是一个提示

你总是应该明确检查条件
在有一个或多个接收方在等待的情况下通知notempty
如果没有人在等待，也没有任何损害
在测试buffer并使用之间保持锁定
因此，没有其他send()可以偷走buffer[]的槽位

为什么wait()会释放p.lock？ 为什么不让send()释放它？
也就是说，为什么不
当p.in - p.out == N时
释放
等待(p.notfull)
获取
通知可能发生在释放和等待之间
在那一点上没有影响，因为没有线程在等待
然后send()的wait()不会返回，即使有消息！
这就是"丢失通知"问题

避免丢失通知
等待(cvar, lock)
调用者必须持有锁
wait()原子性地释放锁并将线程标记为等待状态
因此没有通知可以干预
在返回之前重新获取锁
notify(cvar)
调用者必须持有锁
因此，条件变量始终与特定的锁相关联

实现等待
线程表添加：
新状态：等待
threads[].cvar（以便notify可以找到我们）
重要问题：在哪里释放锁？
[wait()幻灯片]
首先获取t_lock，然后释放锁，然后等待
确保notify()同时持有两者！
顺便说一下，需要修改yield()
[wait+notify()幻灯片]
notify()调用者持有锁，notify()获取t_lock
因此接收notify()持有两个锁
要么在发送线程获取锁之前执行
要么在发送线程暂停之后执行
（但不在发送的检查和暂停之间执行）
如果在之前：
send()等待直到接收完成
send()将看到空槽并且不会等待
如果在之后：
notify()将看到WAITING发送线程，并将其标记为RUNNABLE

但现在我们必须重新访问yield()
[yield v1再次]
t_lock已经持有，不需要设置状态（容易）
yield可能会发现没有RUNNABLE！！！（更难）
此线程WAITING，但在另一个CPU上运行receive()
持有t_lock的循环永远不会结束
因此，没有其他CPU可以执行notify()
因此，没有线程将会是RUNNABLE

系统将会挂起

如何修复yield()?
[yield版本2幻灯片]
不要获取t_lock，不要设置为RUNNABLE
在"空闲循环"中释放+获取
仍然无限循环，没有可运行的线程
但允许其他CPU执行notify()
返回时保持t_lock，但wait()释放它
请注意，我还将SP设置为每个CPU的堆栈，在空闲循环之前
为什么?
yield() v1在调用线程的堆栈上运行空闲循环
有人可能会notify()它
空闲循环中的其他CPU可能会运行该线程
现在两个CPU正在同一个堆栈上执行
例如调用像acquire这样修改堆栈的函数
因此每个CPU堆栈用于yield()在没有任何线程的情况下使用

抢占式调度
如果一个线程从不调用yield()会怎么样?
我们有麻烦了，没有办法复用那个CPU
计算密集型，或者长代码路径，或者有问题的用户程序
要求程序员插入yield()太烦人
我们想要强制周期性的yield

如何进行抢占?
定时器硬件，每秒生成10次中断
中断保存状态，跳转到内核中的处理程序
timer():
yield()
return

结果堆栈会正确恢复吗?
中断将PC +寄存器推入当前线程的堆栈
当不运行时，堆栈看起来像:
...
中断时的线程RA
寄存器
RA到timer()
所以yield()返回到中断处理程序，然后返回到中断的代码

如果在yield中间发生定时器中断会怎么样?
会递归调用yield
死锁：已经持有t_lock
acquire应该禁用中断
release应该重新启用
不仅仅是这里，而是所有锁的用途

如果空闲循环释放t_lock后，定时器中断会怎样?
再次，递归yield()
但是无效的cpus[][CPU()].thread
所以修复yield()将.thread置空
并且修复定时器中断，只有在有效的.thread时才进行yield

摘要
结束思考：如何终止一个线程？可能正在运行...

线程是虚拟处理器
- 允许多个线程，少量CPU
- 是时间共享的基础

我们必须整合：
- yield()
- 条件变量
- 用于抢占的中断

缺失的功能：创建（简单），退出（更难）

6.033 计算机系统工程

2009年春季

6.033 2009年第8讲：性能

性能：为什么我要讲这个？
  通常对设计有巨大影响
  随着负载增加，通常需要进行重大重新设计
  更快的CPU并没有“解决”性能问题
    问题也在增加，例如互联网规模
    磁盘和网络延迟没有改善

性能介绍
  你的系统太慢了
    [图表：客户端，网络，服务器CPU，服务器磁盘]
    可能有太多用户，并且他们在抱怨
  你能做些什么？
  1. 测量以找到瓶颈
     可能是任何组件，包括客户端
     希望找到一个主导的瓶颈！
  2. 放松瓶颈
     提高效率或添加硬件

确定你对"性能"的理解
  吞吐量：每秒请求数（适用于多个用户）
  延迟：单个请求的时间
  有时相反：
    如果需要0.1秒的CPU时间，并且只有一个CPU，则吞吐量为10/秒
  通常不是相反的
    使用2个CPU，延迟仍为0.1秒，但吞吐量为20/秒
    排队和流水线

我将重点关注吞吐量，适用于负载较重的系统
  大多数系统随着用户数量的增加而变慢
    起初，每个新用户使用一些空闲资源
    然后他们开始排队
  [图表：用户数量，回复/秒，线性上升，达到平台期]
  [图表：用户数量，延迟，保持为零，然后线性上升（排队延迟）]

如何找到瓶颈？
  1. 测量，也许发现例如CPU使用率达到100%
     但也许不是，例如cpu使用率50%和磁盘使用率50%，只是在不同的时间2. 模型

     网络应该需要10毫秒，50毫秒的CPU，10毫秒的磁盘
  3. 概要
     告诉你CPU时间去哪了
  4. 猜测
     你可能还是需要这样做
  test假设通过修复系统中最慢的部分
    can很困难：
    如果磁盘一直忙，你应该买一个更快的磁盘/两个磁盘吗？
      还是更多的RAM？

您可能需要使应用程序逻辑更加高效
  更少的功能，更好的算法
  我无法帮助您解决这个问题 - 应用程序特定的
  但有一些通用的技术

主要的性能技术
  1. 缓存
  2. I/O并发
  3. 调度
  4. 并行硬件（两个磁盘，两个CPU等）
  这些在有许多等待请求时最有用
    但如果您的服务器负载很重，这通常是情况

我将专注于磁盘瓶颈
  每年都越来越难成为CPU限制
  如何提高磁盘性能的技巧?

日立7K400：400 GB。7200。8.5毫秒读取，9.2毫秒写入。
567到1170秒/磁道。10个磁头。大约87000个柱面。

关于磁盘的入门知识
  [磁盘幻灯片]
  物理排列
    旋转盘：7200 RPM，每秒120次，每转8.3毫秒
      连续旋转
    圆形磁道，512字节扇区，大约1000个扇区/磁道
    大约100,000个磁道，同心圆环
    多个盘片，7K400有5个，所以有10个表面
      柱面：垂直对齐的磁道集合
    一个磁臂，每个表面一个磁头，它们一起移动
      一次只能从一个磁头读取
  需要三个动作:
    "寻道"臂到所需磁道：根据磁道数量变化，1到15毫秒
    等待磁盘将所需扇区旋转到磁头下：0到8.3毫秒
    在7200转速下读取旋转的位

磁盘性能?
  大规模多轨顺序传输:
    每次旋转一个轨道
    512*1000 / 0.0083
    62兆字节/秒
    这很快！对于大多数系统来说不太可能成为瓶颈
  小规模传输，例如5000字节的网页:
    从*随机*磁盘位置
    寻道 + 旋转 + 传输
    平均寻道时间：9毫秒
    平均旋转时间：对于随机块来说是1/2个完整旋转
    传输时间：大小 / 62 MB/秒
    9 + 4 + 0.1 = 13.1毫秒
    速率 = 5000 / 0.0131 = 0.4 MB/秒
    即大顺序速率的1%。这很慢。
    可悲的是，这在现实生活中很常见

教训：将数据按顺序排列在磁盘上!

缓存磁盘块
  使用一部分RAM来记住最近读取的磁盘块
    这通常是RAM的主要用途...
  你的操作系统内核会为你做这个
  表格:
    BN 数据

... ...
读取（bn）：
如果块在缓存中，则使用它
否则：
驱逐一些块
从磁盘读取bn
将bn、块放入缓存中
命中成本：从RAM中提供小文件约需0秒
未命中成本：从磁盘读取小文件需0.010秒

驱逐/替换策略
重要：不想驱逐即将使用的内容！
最近最少使用（LRU）通常效果好
如果最近使用过，很快会再次使用
对于不适合的大型顺序数据，LRU效果不好
如果你反复读取它（甚至只读取一次）
如果最近使用过，一段时间内不会再次使用！
不想从缓存中驱逐其他有用的内容
随机？MRU？

如何判断缓存是否可能有效？
思考工作集大小与缓存大小的关系是有益的
你的磁盘上有1 GB的数据和0.1 GB的RAM
这样会有好的效果吗？
也许是的：
如果小的子集被广泛使用（热门文件）
如果用户一次只关注其中一些（仅活跃登录的用户）
如果"命中"时间<<"未命中"时间，例如磁盘缓存
如果磁盘I/O在整体时间中占据显著比例
也许不是：
如果数据只使用一次
如果在重新使用之前读取超过0.1 GB（即人们读取整个GB）
如果人们读取随机块，那么只有10%的命中概率
如果命中时间与未命中时间相差不大
例如，缓存计算结果

I/O并发
如果大多数请求都命中，但有些必须访问磁盘怎么办？
而且你有很多未完成的请求
[时间图表：短短长短短]
我们不希望因为等待磁盘而阻塞一切
思路：同时处理多个请求
一些请求可能在等待磁盘，其他请求从缓存中更快地完成
可以使用线程
注意：即使只有一个CPU，线程在这里也很方便
特殊情况：预取（适用于只有一个线程）
特殊情况：写后台

调度
有时改变执行顺序可以提高效率
如果有很多小的磁盘请求在等待
按磁道排序
结果：短寻道时间（1毫秒而不是8毫秒）
积压越大，平均寻道时间越小
系统负载越高，效率越高！
更高的吞吐量

“电梯算法”
也许你也可以按磁道排序，但那很难
缺点：不公平--增加了一些延迟

如果你无法提高效率
购买更快的CPU/网络/磁盘如果可以的话
否则购买多台服务器（CPU + 磁盘）--但如何使用？
严格分割工作：最简单的
用户a-m在服务器1上（CPU + 磁盘）
用户n-z在服务器2上（CPU + 磁盘）
没有交互，所以例如不需要锁，没有竞争
如果一些操作涉及多个用户，就会很困难--银行转账
也许将计算与存储分离出来
前端/后端
前端可以从后端获取所有所需用户的数据
如果受限于CPU或前端可以缓存，效果很好

语用学
程序员的时间成本很高
硬件很便宜
如果编程能让你使用更多硬件，那就是最值得的

****

展示幻灯片
我们将看一道几年前的测验题作为练习，并且用来说明性能的概念测验经常会
提供一些新的虚拟系统，问很多相关问题

值得多练习一下（旧考试题），需要一些适应时间

没钱了

通过网络向客户端提供电影文件
单个CPU，单个磁盘
每个文件大小为1 GB，分成8 KB的块，随机分布

问题1：1153秒
寻道时间 + 半圈旋转时间 + （8192 / 10 MB/秒）
0.005 + 0.003 + 0.0008 = 0.0088秒/块
乘以131072 = 1153

如果布局更好，需要多长时间？

马克添加了一个一GB的整个文件缓存

问题2：1153秒

这是否意味着缓存方案不好？

问题3：B

如果这是真的，我们会对每个原因期望看到什么行为？

在什么情况下，这种缓存方案会工作得很好？

如何改进它?

线程：有什么意义？ 我们是否期望线程能够帮助?

为什么我们认为它告诉我们在哪里调用了yield？ 这些点的意义是什么？ （I/O并发）

为什么非抢占式可能很重要?

新的缓存代码：

1. 4 GB

2. 独立读取每个块：*块*缓存，而不是整个文件

为什么按块缓存可能很重要？ 可能是二阶的，为了修复问题中的一个错误：否则GET_FILE_FROM_DISK可能会阻止所有其他活动。

问题4：100%命中率。

可能是发送了第一个非并发请求的结果。
如果他一开始就发送了许多并行请求会怎样?

问题5：本是正确的。 一个CPU，线程是非抢占式的。

如果有两个CPU，或者抢占？ 这个锁足够吗?

可能需要保护缓存、磁盘驱动程序、网络软件的所有使用，
可能最好在这些模块内部完成。

可能希望采取一些措施防止同时读取相同的磁盘块，即使IF、GET和ADD是不可分割的。 虽然自旋锁可能不是正确的答案。 每个块都有繁忙标志，等待（wait()），通知（notify()）。

问题6：0.9 * 0 + 0.1 * 1153 = 115.3

问题7：E

缓存比总使用数据还大，因此不需要替换。

6.033 计算机系统工程

2009年春季

6.033 第9讲：网络介绍

更多内容的想法
  负载调节/反馈选项（定价等）
  为什么互联网方法获胜？
    另一个计划是什么？
  互联网上的语音/视频实际上是有效的
    尽管没有最佳努力的保证
    为什么？视频是弹性的。
    尽管比特率低，但语音实际上更难
  为什么不需要QoS（弹性自适应应用程序？）

突发性和固定速率的管道使定价复杂化
  我的DSL公司以一种方式销售，以另一种方式批量购买
  他们承担了很大的风险--如果我每天开始下载24小时会怎么样？

旧的互联网地图
  http://www.nthelp.com/maps.htm
  MCIWorldcom 2000最令人毛骨悚然

数据网络的前5讲之一。
  今天的概述：识别问题和设计空间
  在接下来的四个讲座中深入细节。
  *网络是一个有用的系统构建模块
  *内部工作是系统设计的一个很好的案例研究。
    互联网特别是一个非常成功的系统的例子。
    足够复杂，可以成为科学的研究对象，就像天气一样

数据网络的目标是什么？
  允许我们将多台计算机组建成系统。
  你的问题可能对一台计算机来说太大了。
  出于结构原因，您可能需要多台计算机：客户端/服务器。
  更根本的原因：
    计算机的一个关键用途是人与组织之间的通信。
    人们分布在不同的地理位置，他们的计算机也是如此。
  因此，我们不得不处理计算机之间的通信。

系统图：
  主机。可能有数百万个。遍布全球。
  [尚无云，但留出空间。循环主机。]

需要解决的关键设计问题是什么？
  我将其分类为三种类型：
  O：组织性，以帮助人类设计师应对。
  E：经济性，以节省运营商的费用。
   P：物理性。

    E：通用性。
  希望有一个网络，多种用途
    而不是许多小型不兼容/断开连接的网络世界。
      例如，联邦快递为其客户建立了自己的数据网络，以跟踪包裹私有网络有时对于可靠性、可预测的服务和安全性是有好处的

    使用网络的人越多，对每个人都越有用。
    对我来说，价值与使用同一网络的人数成正比。
    对社会来说，价值与人数的平方成正比。

哪些技术工具帮助我们实现普适性?
标准协议使系统构建变得容易。
但不希望通过标准冻结设计来阻止演进。
困难之处在于只标准化使网络普遍有用的必要部分，
而不是可能需要稍后更改的部分。
一个通用网络意味着它不是每个系统设计的一部分。
它是一个黑盒子；简化了使用它的系统的设计。
象征性地，它是一个隐藏内部工作的云。
[绘制网络云]

E：拓扑结构。
[三张新图片；圆形主机，方形交换机]
看看云的内部。
每对之间都有一根电线吗?
优点：网络（电线）是透明的，可以传递模拟信号。
它从不忙碌。 永远不会有如何联系某人的问题。
缺点：主机电子设备。铺设电线昂贵。
星型网络。"交换机"。 在孟菲斯的联邦快递。
优点：较低的电线成本。易于找到路径。
缺点：中心枢纽的负载、成本和不可靠性。
网状拓扑结构。路由器和链路。
优点：低线路成本。有限的路由器扇出。一些路径冗余。
缺点：共享链接，负载较高。复杂的路由。

目标：路由。
给定目标“地址”，找到路径。
在网状网络中比星型或层次结构中更困难。
[图表：两条可用路径]
远距离更改可能需要重新路由
-> 具有全局元素，因此存在扩展问题
许多目标：
效率：低延迟，快速链接
分担负载
适应故障
最小化支付
应对巨大规模
路由是一个关键问题。

目标：寻址。
主机需要能够向网络指定目标。
具体而言，您向路由系统提供的地址
18.7.22.69，而不是web.mit.edu
理想情况：
每个主机都有一个唯一的ID，可能是随机分配的128位
路由系统可以找到地址，无论它在哪里
所以我可以连接到我的PDA，无论我是把它留在家里还是办公室
没有人知道如何为大量的平坦地址构建路由系统!
也许可以在其他系统之上进行分层，但不能直接实现
实际上，地址中编码位置提示。
分层地址，例如18.7.任何
[图表：18区，18.7区，...]
互联网的其余部分只知道18，而不是18中的每个主机
如果主机移动或拓扑发生变化，会有麻烦。
很难利用非分层链接（互联网的其余部分无法使用MIT-
哈佛）
路由和寻址经常非常交织在一起

E：共享。
必须在每根物理电线上多路复用许多对话。
1.如何进行多路复用?
A.等时的。
[输入链接，路由器1，路由器2，链接，重复周期]
保留带宽，可预测的延迟，最初设计用于语音
称为TDM
B.异步的
数据流量往往是突发性的-不均匀间隔
你思考，然后点击一个加载大量大图像的链接
为对话保留带宽是浪费的
所以只要数据准备好就发送和转发
[图表：输入链接，路由器1，链接，路由器2，输出链接]
将数据分成数据包，每个数据包都有关于目的地的头部信息
2.如何跟踪对话?
A.连接
像电话系统一样
你告诉网络你想要和谁交谈
提前找到路径，然后你交谈
对等时钟流量所需
可以允许异步流量的负载平衡控制
也许允许较小的数据包头（只有小的连接ID）
连接设置复杂，转发简单
B.无连接/数据报
许多应用程序与网络级别的连接不匹配，例如DNS查找
每个数据包都是独立的，单独处理
数据包包含完整的源地址和目的地址
每个数据包可能通过网络采用不同的路径!
将负担从网络转移到端点（连接抽象）

E：过载
需求超过容量
共享、增长和突发源的结果
它是如何出现的?
等时网:
新呼叫被阻塞，现有呼叫不受干扰
如果应用程序具有恒定且不可更改的带宽要求，则有意义
异步网:
[图：具有多个输入的路由器]
过载发生在网络内部，对发送主机不可见。
网络必须对多余的流量采取措施，取决于需求大于容量
的时间尺度非常长的时间尺度：购买更快的链路短时间尺
度：告诉发送者减速反馈

弹性应用（文件传输，带参数压缩的视频）
可以随时添加更多用户，但对每个人来说速度会变慢
通常比阻塞调用更好
非常短的时间尺度：不丢弃--缓冲数据包--"排队"
需求随时间波动
[图：需求变化，固定链路容量的线]
缓冲区允许您将高峰延迟到低谷
但前提是低谷来得相当快!
增加了复杂性。 如果过载太多，必须丢弃。 大多数互联网的来源
丢失。

超载时的行为是关键。
[图：输入速率与输出速率，拐点，崩溃]
崩溃：消耗资源来应对超载。
这是一个重要且困难的主题：拥塞控制

E：高利用率。
我们必须支付多少昂贵的容量?
容量>=峰值需求将使用户满意。
那意味着什么?
单用户流量是突发性的!
[典型用户时间与负载图 - 平均低，峰值高]
最坏情况是平均值的100倍，因此网络将浪费99%!
例如，我的一兆比特的DSL线几乎完全闲置
core中购买Nusers * peak太昂贵
但是当你聚合异步用户时 - 峰值和谷值相互抵消。
[几个图表，越来越多的用户，越来越平滑]
峰值越来越接近平均值
100个用户<< 1个用户所需的100倍容量。
空闲容量越来越少，每比特传输的成本越低
"统计复用。"
对于互联网的经济非常重要
假设用户是独立的
大部分是真实的 - 但是白天/夜晚，闪电般的人群
50%的白天利用率可能是核心链路的典型情况

一旦购买了容量，您希望尽可能接近100%。
固定成本，希望最大化收入/工作
可能需要反馈告诉发送者加快速度!
希望保持在输入/输出图的第一个拐点。
或者为了限制延迟而减少
[图表：负载与排队延迟]

P：错误和故障。
通信电线是模拟的，容易受到噪音的干扰。
挖掘机、不可靠的控制计算机。
一般的解决方法：检测、发送冗余信息（重传）或重新路由。
如何分配责任?
你付费给你的电信提供商，所以他们最好是完美的?
导致复杂、昂贵的网络：每个部分都是100%可靠的。
你不信任网络，所以你自己检测和修复问题。
导致复杂的主机软件。
选择智能网络还是智能主机的决策非常重要。
复杂性的位置既付出成本，也具有权力。
“最佳设计”取决于你是运营商还是用户。

P：光速。
每纳秒一英尺。
从海岸到海岸需要14毫秒，也许带电子设备需要40毫秒。
请求/响应延迟。 每个往返时间的字节->每秒36字节。
[图片：数据包通过电线传输-和空闲电线]
需要更高效。
更糟糕的是：延迟会影响控制算法。延迟期间发生变化。
例子：减速/加速。
结果：在过载和浪费空闲之间振荡。

O：广泛的适应性范围。

一个设计，多种情况。
应用：文件传输，游戏，语音，视频。
延迟：机房 vs 卫星。
链路比特率：调制解调器 vs 光纤。
管理：少数用户，整个互联网。
解决方案：自适应机制。

设计空间中的互联网
异步（无预留）
分组（无连接）
没有带宽或延迟保证
可能丢失、重复、重新排序或损坏数据包 - 并且通常会发生
对终端主机没有太多直接用处！终端主机必须修复
但是给予主机灵活性，创新的空间
"尽力而为"

下一堂课：开始讨论解决方案。

6.033 计算机系统工程

2009年春季

6.033 第10讲：分层，链路层

让我们设计一个数据网络。
  我们如何组织我们的设计?

提醒：网状网络。
  [绘制主机/交换机/网状图]

想法：协议。
  两个实体（对等体）正在交谈。
  协议正式定义了对话的结构。
  通常是一系列消息 - 数据包。
  格式：
    特定的消息类型集合。
    消息的每个位代表什么意思?
    [例如：以太网论文，目的地，源地址，数据，校验和]
  下一步发生的规则，状态机。
  语义：它是什么意思?
  通常情况下，您可以从格式中学到所需的所有内容...

但是通信有许多层次。
  [在*网状*网络上绘制，嵌套...]
  电线上的模拟波形。
  从一个交换机到下一个交换机的消息。
  主机/交换机也是如此。
  应用程序到应用程序。
  这里有许多相互作用的协议。
  我们如何组织它们?

想法：层次。
  直觉：协议嵌套。
  内部协议是外部协议的构建块。
  让我们形式化这种组织多个协议的方式。
  选择并定义一个有用的协议。[盒子<-->盒子]
  定义软件接口，以便其他层可以使用它。[堆叠]
  它可能反过来使用更原始的层。[堆叠向下]
  可以通过这种方式构建功能。
  但使用模块和抽象来控制复杂性。
  困难的部分：选择有用的层边界。

6.033层模型：
  [为主机、交换机、主机绘制堆栈]
  物理层：模拟波形 -> 位。
  链路层：位 -> 数据包，单根电线。
  网络层：在电线上的数据包 -> 目标数据包。
  端到端：数据包 -> 连接或流。
  应用层。
  物理层几乎总是与链路层紧密绑定。
  而应用层不是一个普遍有用的工具。
  注意：层可以有许多客户端
    多个应用程序使用端到端，多个端到端使用网络
    需要一种多路复用它们的方法
  注意：网络层可以使用多个链路
  所以在一个主机中的真实情况
    应用1 应用2 应用3
      TCP UDP

IP
以太网 WiFi
层 == 数据网络主题的大纲。

层的堆栈：
层之间交互的重复方案
每个层都会添加/剥离自己的头部。
将更高层的数据封装为“有效负载”。
[添加到数据包图中；内部头部等]
每个层可能会分割更高层的数据。
[将流分割成数据包的负载]
每个层都复用多个更高层。
[将协议号字段放入数据包]
每个层对更高层（大多数情况下）是透明的。
在远端传递的数据 == 输入的数据

物理层
由电线构成。
我们只会讨论一个方向。
只需构建两个用于双向通信。
[图片：发送方，电线，接收方]
可以发送"信号"：电压，光功率水平。
我们想要的是一串比特流。
问题：
哪个值，0还是1？ 与噪声相比
何时在接收端采样

第一个草稿：
3根电线：数据位，准备好，确认
[图片]
光速限制了数据速度。
从海岸到海岸：30比特/秒
一英里：200千比特/秒

第二个草稿：
假设发送方和接收方具有准确的振荡器。
即相同的频率。
[画出两个振荡器]
发送方每个时钟周期发送一个比特。
[波形图。标记采样点。]
简单问题：电线延迟导致相位变化。
困难问题：
你可能可以建造一个准确的时钟。
你可能无法建造两个相同的时钟。
在分布式系统中，你永远不能假设时钟是同步的。

草人三：
将时钟发送到单独的电线上？
昂贵，如果快速或长距离会导致电线之间的偏差

正确答案：
思路：从信号本身恢复时钟。
接收器保持本地时钟。
如果信号转换不在时钟转换上，调整本地时钟。
平衡信号：你需要转换！不能有连续的零。
因此不能直接将0/1编码为高/低信号。

以太网/曼彻斯特有四种波形 == "符号"
LH：0位
HL：1位
LL：空闲
HH：未使用
因此只使用了4个代码中的2个。[图片]
可以更高效：每2位有8个符号值。[图片]
LLH：00
LHL：01
HLL：10
HLH：11
其他未使用的，例如LLL，以帮助时钟恢复
为什么不是每127位有128个信号？
单个信号错误会破坏127位...

我们能将其声明为一层吗？
这是一个普遍有用的抽象吗？
不是的：没有共享的方式。只有位。

链路层：从位到数据包/帧
为什么是数据包？为什么不是连续的数据？
需要将来自不同对话的数据包混合在一起。
我们如何界定数据包？

数据包帧
链路有两种状态：空闲，发送数据。
需要信号状态转换。
空闲 -> 发送（数据包开始）。
发送 -> 空闲（数据包结束）。
让我们用全低信号来表示空闲。
曼彻斯特不使用LL来发送位
数据包开始很容易：只需发送一个一位。
可能希望一些更长的交替序列用于时钟恢复。
例如，8个LH后跟HL，以防接收器错过了前几个LH
称为前导码。
[数据包图片：前导码，数据，结束标记]
标记数据包的结束？
特殊代码通常不能出现在数据流中。
转换数据流以消除魔术位序列。
在曼彻斯特方案中，OR两个零信号。
如果包结束信号损坏了？破坏下一个包吗？
为什么我们不在开头使用长度字段而不是特殊代码？

我们可以在这里放一个接口，然后创建一个层吗？
还没有：我们怎么知道谁的包是谁的？
真正指示我们上面的哪一层。
我们只需要在包头中有一个数字。

我们可以将这个变成一个层。
按照惯例，链路层检测错误。

我们应该怎么处理错误？
例如，噪音会破坏一些位。
我们需要定义链路层抽象。
它能合理地*保证*什么？
可以忽略这个问题，让更高层来处理。

小心的更高层可能会进行检查。
我们能保证完美地传递所有数据包吗?
链路层很难做出这个保证。
向前的交换机重新发送?复杂，需要缓冲。
如果交换机出现故障怎么办? => 链路无法保证!
也许应用程序也不在乎，例如实时语音。
现实的答案:
保证所有传输的数据包都是正确的。
我们被允许（也被期望）丢弃损坏的数据包。
“尽力而为”合同的例子。
这意味着链路层只是*检测*位错误。

错误检测

最佳方案取决于错误模式。
由于短暂的噪声导致单个位错误。
微波炉会清除许多数据包。
编码会清除例如2位的一个数据块。
示例：在末尾异或字节。
[图片：垂直列的字节，底部是异或]
检测任何一个位错误，一些多位错误。
不能检测例如字节交换。
还有更复杂的方案可用!
通常称为“校验和”。

错误纠正
一个单个位错误会导致链路层丢弃整个数据包。
看起来很浪费。
我们通常可以挤出更多的性能。
这是一个纠正单个位错误的计划。
[在每个字节中添加第9位。]
如果有两个错误的位仍然必须丢弃。

我们的链路层协议是什么样的?
[参考协议板]
格式：主要是帧格式，也有原型号。
什么是抽象?
暴露数据包。（不是例如文件...）
允许在更高层协议之间共享。
尽力而为：可能无法传递数据包。
很少会给你一个损坏的数据包。

以太网MAC在这个层次上运行。

现在我们可以沿着一根电线传递数据包。
下一堂课：沿着交换机路径传递数据包。

6.033 计算机系统工程

2009年春季

第11讲

Sam Madden

网络层

今天：网络层。

网络层的工作是在多跳网络中找到并转发发送方和接收方之间的数据包。

展示一个网络的示例：

"路由器"

网络层接口

两个主要任务：

转发 - 根据路由表通过链路发送数据
路由 - 构建路由表的过程

转发：

展示网络中的路由表

指出可能存在多个路由表（示例）

展示堆栈 - 注释上面的"net_send"和"link_send"和"net_handle"和"e2e_handle"调用，封装

在堆栈中显示链接选择

转发 - 机械。只需在表中执行查找。

伪代码。

转发表 t

```
net_send(payload, dest, e2eprot):
        pkt = new packet(payload,dest,e2eprot)
        net_handle(pkt)

net_handle(pkt):
        if (pkt.dest == LOCAL_ADDR):
                e2e_handle(pkt.payload, pkt.e2eprot)
        else:
                link_send(t[p.dest].link, pkt)
```

路由 - 计算转发表

如何计算转发表？ 手动 - 不可扩展。

集中 - 不是一个好主意（为什么？）
- 需要一个路由算法来收集
- 收集需要很多消息
- 难以适应变化

路径向量算法 - 分布式

每个节点维护一个转发表 T，具有：
"e2e_handle" 调用，封装
目的地 链接 路径

两个步骤：
广告（定期）
将 T 发送给邻居

integrate(N,neighbor, link) -- 在从邻居接收到广告时
将从邻居处在链接上收到的邻居表 N 合并到 T 中

合并：
对于每个目标d和路径r在N中：
如果d不在T中，则将（d，link，neighbor ++ r）添加到T
如果d在T中，则替换为（neighbor ++ r）比旧路径更短

例子：

（如果每个人都选择到每个目标的最佳路径，你会发现对于一个由N跳分开的最远节点的网络，在N轮中每个人都会知道如何在N步内到达其他人。）

问题：保持路径在表中的目的是什么?

问题：

- 永久循环?
  - 如果我们添加一个规则，不选择包含自己的路径，这个问题就不会出现
  这就是我们需要路径的原因!

- 临时循环 - 由于两个节点可能略有不同
  例子
  - 解决方案：向数据包添加发送计数 - "TTL"

- 失败/更改 - 定期重复广告，
  从表中删除未重新广告的路径
  （例如，以路由器R开头的路径P应该在表中）

下一个广告来自R.)

- 图形变化--与故障相同

这在互联网上是如何工作的：

起初，互联网是一个像这样的小网络

展示演化幻灯片

在这里使用路径向量的问题是什么？

网络很庞大

> 10亿个节点在网络上

即使我们假设其中大部分是仅连接到本地路由器的计算机（因此实际上不需要运行路径向量协议），互联网上仍然有数百万个路由器每个路由器都需要知道如何到达这数十亿台计算机中的每

一台使用纯路径向量，每个节点都有一个数十亿条目的表（需要千兆字

节的存储空间）

每个路由器都必须将这些千兆字节的表发送给其每个邻居；数百万个广告传播。灾难。

解决方案：分层路由

将网络划分为区域；具有多级路由

每个区域有一个代表节点；在区域级别执行路径向量。 在每个区域内，可以自由进行任何操作。 （例如，使用更多层次结构。）（例如，以路由器R开头的路径P应该在

如何命名节点：

区域名称

在互联网上，这是IP地址

例如，18.7.22.69 -- 这是mit.edu

运行的互联网路由器 -- BGP -- 广告前缀的这些地址

显示广告（例如，"18.*.*.*"...）
17.1*.*.*

6.033 计算机系统工程

2009年春季

6.033 第12讲

Sam Madden

端到端层

上次：网络层 -- 如何在多个链路的网络中传递数据包

回想一下，网络层是尽力而为，意味着：

- 数据包可能丢失、重排序、损坏
- 由于没有路由、队列溢出、数据包损坏/冲突等原因导致的丢失
- 由于数据包采用不同的路由、重传而导致的重排序

网络层也是面向消息的。

端到端层解决了这些限制。 应用程序可能需要的一些功能：

1. 连接、数据流，而不是需要明确地将数据分成
消息

2. 所有数据包都到达（可靠）

3. 没有重复（最多一次）

4. 数据包按顺序到达

5. 效率

并非所有应用程序都需要这些功能 - 例如，在语音聊天应用程序中，您可能更喜欢
具有一些数据包丢失的可预测延迟，而不是所有数据包
以不可预测的延迟到达。

上面的列表大致是互联网的TCP协议所提供的。互联网还提供其他
协议，如UDP（上述协议都不包括）或RTP（流媒体，但不可靠）

不同的端到端层将提供这些功能的不同子集。 目标是在不修改网络层的情况下构建
这些功能。

端到端接口

图表：（包括端口->应用程序表），数据包

流连接（向端口->应用程序表添加条目）

源	目标
conn = open_stream(dest, port)	conn = listen(port)
send(conn, bytes1)	
send(conn, bytes2)	bytes = recv(conn)
close_stream(con)	

端到端层将流分割成数据包，并将其发送到网络层。

可靠性 --> 至少一次

所有数据包都到达，且没有数据包损坏

对于损坏，通常数据包具有某种校验和，在接收到数据包时进行检查。

如何确保所有数据包都到达?

想法:

每个数据包都由接收方发送确认。 在发送方设置一个定时器，当在规定时间内没有收到确认时，重新发送。

为每个数据包关联一个唯一的序列号。

将序列号附加到确认中。

图表:

如何设置定时器间隔？1秒？1毫秒？ 网络差异很大！在本地有线网络中，可能会看到端到端延迟<1毫秒。 在蜂窝网络或洲际网络中，可能会看到延迟>1秒。

太短 --> 始终重新发送
太长 --> 网络利用不足（展示）

那么我们应该怎么做？调整超时时间！

测量往返时间（RTT）并进行调整。

RTT = 数据包发送和确认接收之间的时间
只使用一个样本吗？不。
仅使用上次的时间+错误吗？不行。-- 变化太大，即使在一个网络上也是如此（展示幻灯片）

使用最近数据包的平均值。

为了避免需要保留一段时间的平均窗口，可以使用指数加权移动平均：

（展示幻灯片）

好的，现在我们适当地设置定时器。 确保每个数据项至少到达一次。

我们可以有重复吗？（是的，为什么？）

对此应该怎么办？

至多一次的传递

在接收器上，跟踪已经确认的数据包（在列表中）。 当收到重复的数据包时，重新发送确认，但不交付给应用程序。

图表：

何时可以从acks列表中删除某个内容？由于acks可能会延迟或丢失，可能会相当晚地收到重发的数据包。一个典型的解决方案是附加最后一条消息

从发送者接收的消息上的id。

至少一次传递和最多一次传递一起是确切一次传递.

有点名不副实。

问题：

- 假设接收方在接收到消息后但在发送确认之前崩溃；重新启动
--> 它是否处理了该消息，如果再次接收到消息会怎样？

我们将在春季假期后更详细地讨论构建可靠系统和这些问题。

- 确认信号意味着什么？ 通常只表示端到端层将数据包交给了应用程序，而不表示应用程序是否处理了该消息。

(跳过？) 序列号要多大？

16位？ 在千兆比特网络中，使用1千比特的数据包，每秒发送1百万个数据包。因此，在大约60毫秒内将耗尽序列号空间。数据包可能会被延迟这么长时间。

32位？4000秒。 更好，但某些连接将持续这么长时间。 需要回收序列号。

请注意，到目前为止，该协议按顺序传递数据，因为一次只有一个未完成的消息。

到目前为止，我们已经构建了一个流式、仅一次、有序的传递机制。
问题是它很慢。

显示图表：

假设往返时间为10毫秒。每秒只能发送100个数据包。 如果数据包是1000位，那么我们只能以100千比特/秒的速度发送。不好！

我们可以让数据包变大吗？

为了千兆以太网，它们需要多大？比现在大10000倍 --> 10兆比特。 太大了！重传非常昂贵，而且我们在网络复用方面做得不好！

解决方案：更小的数据包，但同时有多个未完成的数据包。

未完成数据包的"窗口"

显示图表：

仍然浪费，因为发送方等待一个往返时间。解决方案：滑动窗口：

请注意，如果窗口大小太小，可能仍然需要等待。

那么，窗口要多大？ 希望能够"覆盖"延迟。
假设我们有一个往返时间的测量值。

并且假设我们知道网络可以发送的最大速率（无论是由于链路、网络中的延迟还是发送方或接收方的限制）

最大未完成数据包数

窗口 = 往返时间 x 速率

"带宽延迟乘积"

当然，这假设我们可以准确测量速率，但事实证明这很困难，因为速率取决于网络内部发生的事情（例如，网络内部其他节点正在做什么）。这是下次讨论的主题。

确认和窗口 - 两种可能性：

- 确认是按消息计算的，如上所述。
    + 精确（只重传一个数据包）
    - 如果确认丢失，可以重新发送不需要重新发送的数据

- 累积确认
    例如，指示已经接收到的序列号，+ 一个确认可以覆盖多个数据包（减少确认数）
    - 发送方不知道确切丢失了什么，因此必须重新传输更多数据

实际上，两者都使用：
    - 在常见情况下使用累积确认（例如，每n个数据包）
    - 在存在单个丢失时使用选择性确认

乱序传递

尽管所有数据包都到达，但数据包可能不按顺序到达，因为窗口控制

解决方案：重新排序缓冲区。

接收方固定大小的缓冲区记录乱序的数据包。 如果数据包按顺序到达，交付给应用程序。否则，将其添加到乱序缓冲区，同时等待更近期的数据包到达。图示：

当丢失的数据包到达时，立即将缓冲区的前缀发送给应用程序。

6.033 计算机系统工程

2009年春季

6.033 第13讲

山姆・麦登

拥塞控制

上次，我们看到端到端层如何确保仅一次传递，使用窗口来高效利用网络。

还有几个问题需要解决：

- 重新排序 -- 数据包可能乱序到达接收方

```
接收(p)
        槽 = p.sno - 上一个
        if (槽 > 上一个 + 大小)
                丢弃
        else
                新的 = 槽为空
                if (新的) 将p放入槽中
                确认p
        if (槽 == 上一个 and 新的)
                将前缀交付给应用程序
                槽 += 大小(前缀)
```

累积确认

到目前为止，已经将确认作为选择性的方式呈现- 用于特定数据包

在互联网中，有时会使用累积确认

在上面的图中，当p1到达时，会发送确认(2)，表示已经接收到所有的数据包直到2

+ 对丢失的确认不敏感
    (如果使用选择性确认，丢失一个确认需要重新传输一个数据包。在这里，如果P2的确认丢失了，当P0到达时，接收方知道P2已经到达。)

- 发送方不知道哪些数据包丢失了

    在这里，发送方已经发送了p3、p4和p5；接收方已经接收到了p4和p5，但是只收到了p2的确认。

    可能导致过多的重传

在TCP中，同时使用累积确认和选择性确认：

    通常情况下，使用累积确认，这样丢失一个确认不会有问题当确认一个后续数据包到达时，使用选择性确认，这样发送方就知道哪些数据包已经到达了

选择窗口大小

上次，我们发送了窗口大小应该是：

    窗口 = 速率 x RTT

其中速率是接收方可以接受数据包的最大速率

这将使接收方始终保持忙碌状态

例如，假设接收方可以处理1个数据包/毫秒，RTT为10毫秒；那么
    窗口大小需要为10个数据包，以使接收方始终保持忙碌状态

2个问题：

1）接收方如何知道自己可以处理数据包的速度有多快?
2）如果网络无法以接收方可以处理的速度传递数据包怎么办?

1）实际上并不需要知道。在实践中，这并不太重要 - 只需将其设置为足够大即可。例如，在100 Mbit以太网上，RTT为10毫秒，

窗口 = 100 Mbit/秒 * .01秒 = 1 Mbit = 128 KB

那么如果网络无法以这个速率处理数据包，会有问题吗?

假设发送方和接收方都可以以3个数据包/秒进行通信，但是在发送方和接收方之间存在一个瓶颈路由器，该路由器只能以1个数据包/秒的速度发送。假设RTT为2秒（因此接收窗口为6个数据包）

1秒后，这个路由器的队列中将有2个数据包
2秒后，将有4个数据包

拥塞崩溃 = 传输负载持续超过可用带宽

请注意，如果我们能够快速调整发送方的往返时间（RTT），那将会有所帮助，因为我们将发送更少的重复数据包。

但是，如果我们有一个大窗口并且存在一个慢速瓶颈，我们仍然可能遇到麻烦。

为什么会出现拥塞问题?

互联网中间的路由器通常很大且速度很快 - 比终端主机快得多。 所以上述情况不应该发生，对吗?

问题：共享。 源之间彼此不知道，因此可能同时尝试使用某个链接（例如，突发人群）。 这甚至可能使最强大的路由器不堪重负。

例子：9/11 - 每个人都去CNN。苹果 - 发布新产品。

对此应该怎么办：

避免拥塞：

增加资源吗？ 昂贵且反应缓慢；不清楚它是否有帮助。 在电话网络中使用

。 一些网络服务器会这样做，以减轻负载。 但在网络内部很难做到这一点，因为路由器对应用程序或它们可能呈现的负载没有一个很好的模型。

拥塞控制 - 要求发送方减速。

发送方如何了解拥塞情况？

选项：

1）路由器发送反馈（直接发送给发送方，或通过标记数据包，
接收方在其确认中注意到并传播）。

这种方法有效，但如果网络拥塞，这些消息可能无法传递（或可能被延迟很长时间）。2）假设数据包超时表示拥塞。对于拥塞应该怎

么办：- 增加超时时间。

- 减小窗口大小。

需要非常积极地这样做 - 否则拥塞很容易崩溃产生。 拥塞控制很难恢复。

TCP同时执行以下操作：

超时：指数退避

在重传的数据包上，设置：

设置超时时间 = 超时时间 * c ; c = 2

直到达到某个最大值

超时时间呈指数增长

当确认开始到达时（丢包停止）
超时时间基于RTT EWMA进行重置（与上一节课相同）

2个窗口：接收窗口 + "拥塞窗口"：

窗口大小 = 较小的（拥塞窗口，接收窗口）

在重传的数据包上：

拥塞窗口 = 拥塞窗口/2

（最小大小 = 1）

在确认的数据包上：

拥塞窗口 = 拥塞窗口 + 1（最多达到接收窗口大小）

为了快速初始化窗口，使用TCP "慢启动"：

在连接开始时，将拥塞窗口加倍，直到达到接收窗口大小，除非发生丢包：

慢启动 - 仅在连接开始时

每次确认时：

拥塞窗口 = 拥塞窗口 * 2（最大接收窗口）

TCP是否"公平"？
不公平；应用程序不必使用TCP，如果不使用，则不遵守拥塞规则。 应用程序还可以使用多个TCP连接。

那么为什么这个有效呢？ 显然是因为大部分流量是TCP。

6.033 计算机系统工程

2009年春季

DNS，CDN 第14讲

Sam Madden

DNS

域名（例如youtube.com）和IP地址之间的关系是什么?

DNS是确定这种映射的系统。

基本思想：

你联系一个DNS服务器，给它一个查询

它会回答你的问题，或者将你的答案转发给其他服务器例子：

每当一个服务器回答一个查询时，它会将其添加到缓存中

当你得到一个响应时，你将其添加到缓存中

每个缓存值都有一个持续时间，在此之后你应该重新查找名称

在DNS层次结构的顶部是根服务器

然后是.com，.edu等服务器...

然后是MIT，youtube等服务器...

显示缓存

dig +norec csail.mit.edu

A记录与NS记录
TTL

观察到A记录的过期时间比NS记录短得多

例子：

csail.mit.edu

dig
dig edu@E.ROOT-SERVERS.NET.
dig mit.edu @G.GTLD-SERVERS.NET.
dig csail.mit.edu @bitsy.mit.edu.

你如何最初获取你的DNS服务器?

- 你的管理员告诉你
- DHCP（域主机配置协议）--你在本地链路上广播--DHCP服务器通过另一个广播消息回应：- 为你提供一个IP地址

  - 一个"网关"路由器的IP地址供你使用
  - 一个要使用的DNS服务器的IP地址
  - ....

DNS有一些高级功能：

- 一个物理机器可以有多个名称

- 一个名称可以对应多个IP地址
  DNS负载均衡
  DNS服务器选择要返回的名称
    大多数DNS服务器以"轮询"方式循环遍历这些

示例：

dig google.com

;; ANSWER SECTION:
google.com. 282 IN A 209.85.171.100

google.com. 282 IN A 74.125.45.100
google.com. 282 IN A 74.125.67.100

内容分发网络（CDN - 例如，Akamai）

使用DNS提供可扩展性并适应负载

假设我有一些经常访问的内容--例如，YouTube上的视频

我可以使用DNS负载平衡在我的本地服务器之间平衡负载，但我仍然需要拥有这些服务器。

这会给我的服务器带来巨大的负载，无法适应负载峰值（例如，“slashdot”效应）。

此外，对于远离的用户（例如，亚洲或澳大利亚的用户），他们必须通过长距离和狭窄的管道下载该内容，而澳大利亚的ISP必须支付很高的带宽费用。

对于像这个视频这样经常访问的内容，每次都必须到加利福尼亚州圣布鲁诺去获取是否更好。

想法：创建一个本地缓存

解决方案1：代理缓存。 对于每个URL，在本地缓存中查找（可能由您的ISP运行）以查看内容是否存在。如果存在，则获取它。否则，获取原始数据。 就像DNS请求一样，网页可以与其关联的缓存生命周期，代理缓存会尊重这些生命周期。

问题：

- 对ISP有帮助，但不一定对内容提供商有帮助
- 要求客户端配置其浏览器以使用代理缓存
- 不能解决Slashdot效应

存在所谓的"透明代理"，可以自动进行此过滤，但这可能对用户来说不太好，特别是如果他们的ISP在这样做（公司经常这样做）

解决方案2：内容分发网络（例如，Akamai）

显示图表：

举例：

nytimes.com
dig graphics8.nytimes.com

来自麻省理工学院网络和移动网络

观察到答案的TTL非常短 - 为什么？ -- 处理Slashdot效应 - 可以动态地开始使用越来越多的Akamai服务器来处理特定请求

Akamai - 公司

为内容提供商提供了一个卸载服务器负载的方法

如果服务器位于ISP内部，还可以帮助ISP（为ISP参与提供了激励！）

Akamai在世界各地拥有数千个缓存

当收到来自images8.nytimes.com等内容的请求时，它使用DNS将用户转发到最近的服务器。

确定“最近”的服务器和分配多少服务器是他们的秘密酱。

对于动态内容，Akamai也可以工作。

示例 - 显示尝试选择返回Akamai的最佳路由，维护可达性信息等。

Akamai是一个更一般的例子，被称为超级网络。

覆盖是一种通过在现有网络之上构建来创建具有新功能或不同结构的网络的方法。

Akamai在IP网络之上创建了一个覆盖网络，从一组IP服务器中选择最佳路由。

覆盖广泛用于通过新功能扩展网络，例如：

提供不同的拓扑结构（例如，客户端的直接连接）
- 例如，出于管理原因（VPN）
- 或出于性能原因（Akamai）

提供不同的寻址机制（例如，内容寻址，P2P）

VPN示例：

公司希望允许远程用户或站点访问其企业互联网站点

过去的日子可能会用调制解调器来做这个，但那太慢了，而且很麻烦。

VPN为远程用户提供了一种在实际上是外部的情况下，看起来是在内部网络上的方法。

思路是使用覆盖网络（客户端和服务器在网络内部）在公共互联网上“隧道”传输流量：

图表：

这只是一个简单的例子，但原则上可以创建复杂的多跳VPN。

6.033 计算机系统工程

2009年春季

第15讲　　从不可靠的组件构建容错系统

山姆・麦登

到目前为止，我们看到了什么：

模块化
　　RPC
　　进程
　　客户端/服务器
网络
　　实现客户端/服务器

看到了一些处理故障的例子--例如，在
一个尽力而为的系统中提供恰好一次的语义。

本课程的目标是了解如何从一组不可靠的部件系统地构建可靠的
系统。

基本的工作方式是引入冗余，例如：

- 在TCP中，这采用多次发送数据的形式；
"- 今天：一种常见的技术 - 关键系统组件的复制
"- 下一次：在文件系统或数据库中，它通常以在多个位置写入数据的形式出现。

"一些高级要点：

　　"- 这些技术提高了可靠性，但构建一个完全可靠的系统是不可能的。 "我们能做的最好的就是无限接近可靠。 "我们的保证基本上是概率性的。

　　"- 可靠性是有代价的 - 硬件、计算资源、磁盘带宽等。更高的可靠性通常意味着更高的成本。 "必须决定 - 就像TCP一样 - 额外的可靠性是否值得成本。总是有一个权衡。

"本讲座：

　　"了解如何量化可靠性

　　"了解我们用于容忍故障的基本策略

　　"将复制作为提高可靠性/掩盖故障的技术来看待

　　"将在磁盘系统的背景下进行，但其中许多教训也适用于其他组件。

"（将在最后回顾此内容。） "

为什么是磁盘?

图示（磁盘，电缆，计算机，控制器，文件系统软件）

故障发生在哪里?
（到处都有。）展示幻灯片。

系统中的任何故障都会使计算机对用户不可用。

目标是提高系统的整体可用性。

测量可用性

平均故障时间（MTTF）

平均修复时间（MTTR）

可用性 = MTTF / (MTTF + MTTR)

MTBF = MTTF + MTTR

假设我的操作系统每个月崩溃一次，并且需要10分钟恢复
MTTF = 720小时（43,200分钟）
MTTR = 10
43200 / 43210 = .9997（"三个九"）-- 每年2小时的停机时间

一般的容错设计过程：

- 量化每个组件的故障概率
- 量化故障的成本
- 量化实施容错的成本

这里许多组件的故障概率都很高，但是

然而，与其他组件不同，磁盘是唯一一个如果故障了，你的数据就会丢失的组件--可以随时更换CPU或电缆，或重新启动文件系统。

你可以更换硬盘，但是你的系统就不能工作了。 所以硬盘故障的成本很高。

好的，那么为什么硬盘会出现故障呢?

显示硬盘幻灯片

旋转盘片
磁性表面
数据按磁道和扇区排列
磁头“浮动”在表面上，进出以读写数据

可能出什么问题?
整个硬盘可能停止工作（电气故障，断线等）

扇区故障：

可能从错误的位置读取或写入（例如，如果驱动器被撞击）
可能由于磁头过高或涂层过薄而无法读取或写入
磁头可能与驱动器表面接触并划伤表面

前一张幻灯片表明硬盘经常出现故障。 制造商怎么说?

显示希捷幻灯片

平均故障间隔时间：700,000小时
80年？80年前我们还在用算盘编程。

这里发生了什么？ 他们是如何测量的?

运行，比如说，1000个硬盘，运行时间为3000小时（总共300万小时）
出现了4次故障
所声称的是每750,000小时1次故障

实际故障率是什么样的？浴缸（5年）

那么这700,000小时告诉我们什么？ 浴缸底部的故障概率。如果我在一个拥有一千台机器的数据中心运行，并且想知道我需要多频繁更换硬盘，这将很有帮助。 每年每个硬盘的故障率约为1%。

购买更贵的硬盘会有帮助吗?

实际上并不会（展示幻灯片）- 增加MTTR两倍，价格增加5倍

应对故障：

第一类故障 - 扇区错误

我能做什么
（展示架构图 - 包括硬盘，固件，驱动程序，文件系统）

1）悄无声息地返回错误答案

2）检测故障 -
每个扇区都有一个带有校验和的头部
每次磁盘读取都会获取一个扇区及其头部
固件将校验和与数据进行比较
如果校验和不匹配，则向驱动程序返回错误

3）正确/掩盖
如果固件信号错误（如果存在对齐问题），请重新阅读
（还有其他许多策略）

这些是一般的方法--例如，在应用程序级别可以有校验和（在文件中），或者备份以从坏文件中恢复/修正。

磁盘具有校验和--它们有多大帮助?

（展示SIGMETRICS幻灯片）

170万个驱动器
10%的驱动器在2年内出现错误

这些是无法恢复的读取故障
会导致错误数据进入文件系统

磁盘是否能捕捉所有错误？（展示FAST08）

150万个驱动器，41个月，1000个磁盘上的40万个损坏读取
（不知道进行了多少次读取，但每个驱动器可能读取了数百万个块--所以我们在谈论数万亿次读取。）

未被发现故障的非常小的比例。

但是它们确实发生；然而，许多系统假设它们不会发生（例如，你的桌面文件系统。）

整个磁盘故障怎么办?

强大的解决方案：复制

将所有内容写入两个或更多个磁盘

如果一个读取失败，从另一个读取

图表：

写入（扇区，数据）：

写入（磁盘1，扇区，数据）
写入（磁盘2，扇区，数据）

读取（扇区，数据）

数据 = 读取（磁盘1，扇区）
如果出错
数据 = 读取（磁盘2，扇区）
如果出错
返回错误
返回数据

这解决了所有问题吗？（不）

未捕获的错误
可以进行投票/三重模冗余 -- 图表

系统的其他部分出现故障
展示非停机幻灯片
复制所有内容，包括操作系统 -- 运行N个
每个应用程序的副本，比较每个内存读取/写入
成本巨大，性能支持

假设故障是独立的
（那么停电怎么办）
这不是应用程序想要的 -- 应用程序正在写入跨多个块的大型数据项；如果在写入过程中发生停电怎么办

总结
磁盘：
寿命约为5年
每年1%的随机总故障
10%的磁盘在2年内会出现无法读取的扇区
少量未被捕捉到的错误
复制是一种可以用来应对的通用策略

网址:

http://h20223.www2.hp.com/NonStopComputing/downloads/KernelOpSystem_datasheet.pdf

HP Integrity NonStop NS16000服务器数据表 - PDF（2005年5月）

http://www.usenix.org/events/fast08/tech/full_papers/bairavasundaram/bairavasundaram.pdf

http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_7200_10.pdf

6.033 计算机系统工程

2009年春季

6.033 讲座 16 -- 原子性

Sam Madden

上周: 看到如何利用不可靠的部件构建容错系统

看到磁盘如何使用校验和来检测坏块，以及我们如何使用
磁盘复制来掩盖坏块错误。

今天的目标是开始讨论如何将多步操作看起来像一个
单一的、原子的操作。

为什么这很重要？例子>

两个银行账户，余额存储在磁盘上

转账程序

```
xfer(A, B, x):
        A = A - x //写入磁盘
                        <--------------崩溃!
        B = B + x //写入磁盘
```

丢失了$x。 请注意，复制不会修复这个问题（除非我能阻止系统完全崩溃）。即使
对A的写入顺利进行，我们仍然丢失了x美元。请注意，发生的情况是我们暴露了这个操

作的一些内部状态，这本应该是原子的。

我们真正想要的是使这个操作要么发生要么不发生的方式。
例如，为了保护金钱。

这被称为“全有或全无的原子性”-- 要么发生，要么不发生。

相关问题：　　**并发**　　。 假设有两个人运行：

```
A = 10; B = 20
U1                  U2
----                ---
转账(A,B,5)         转账(A, B, 5)
```

假设 A = A-X 实际上

```
R0 = 读取(A)
R0 = R0 - X
```

写入(A, R0)

结果:

A = 0, B = 30 (正确)

U1	U2
RA	
	RA
	WA
WA	

A = 5 (不正确!)

这两个动作不是隔离的 - 它们看到了彼此的中间状态.

应该只允许 A = 0, B = 30.

隔离目标: 并发动作的结果等同于对这些动作进行某种串行执行.

例如, A & B == A 然后 B 或者 B 然后 A.

请注意，我们已经看到了使用锁来实现这一点的方法，但这需要复杂的手动推理. 例如，假设用户发出一系列应该原子执行的转账:

转账(A,B)
转账(C,D)
转账(D,A)

不能仅仅将锁定逻辑推入转账中 - 执行转账的过程必须进行锁定以提供隔离.

目标是开发一个提供并发操作的全或无操作原子性和串行等价性的抽象。

这个抽象称为"原子操作"或"事务"

工作原理如下：

T1	T2
开始	开始
转账(A,B,20)	转账(B,C,5)
扣款(B,10)	存款(A,5)
...	....
结束	结束
也称为"提交"	

特点：

- 用户不需要在任何地方显式加锁。

- 全部或无

- 串行等价

- 不需要预先声明操作 - 当发出"结束"时，事物变得可见。

非常强大的抽象。 实现起来有些棘手，但使程序员的生活变得轻松。

使用这些技术的最常见的系统是数据库。 它将数据表示为具有记录的表，而不是抽象文件。

例如：

账户：

客户：

...

可以编写读取和修改多个表的混合事务，例如

- 在账户之间进行复杂的转账操作或
- 删除一个客户及其所有账户。

...

今天 - 开始讨论如何在没有隔离的情况下实现全有或全无属性（我们将在几节课中讨论）。

假设我们有一个存储在单个文件中的数据库余额文件（例如，Excel电子表格）

想象一下，该文件被加载到内存中，并且“保存”操作的工作方式如下：

```
保存（文件）：
    对于文件中的每个记录r：
        如果r已更改
            将b写入文件
```

问题是什么？

可能在写入文件的过程中崩溃，导致文件处于某种中间状态

版本2（“影子副本”）

```
保存（文件）：
    复制（文件，fnew）
    对于fnew中的每个记录r：
        如果r是脏的，则将r写入fnew
    移动（fnew，文件）//重命名 --- 提交点！
```

需要原子重命名操作

崩溃后，要么有旧版本，要么有新版本

原子重命名：

```
unlink(文件)
                    <--- 崩溃！（文件消失）
link(新文件,文件)
unlink(新文件)
```

实际：

```
link(新文件, 文件)
                    <-- 崩溃！（两个文件的副本！）
unlink(新文件)
```

link命令是“提交点”-在它发生后，原子操作保证完成（“全部”部分的“全部或无”）

磁盘上的文件系统状态

目录块：

名称	inode#
文件	1
新文件	2

inode 1:
块：A，B，C
引用计数：1

inode 2:
块：D，E，F
引用计数：1

链接：

<---- 崩溃！ //文件和新文件都存在，但不同
（CP）修改目录块 //新文件和文件都指向新文件
<---- 崩溃！ //引用计数错误
更新引用计数

取消链接:

<---- 崩溃！// fnew和file都存在，引用计数错误
修改目录块
<---- 崩溃！// 只有file存在，引用计数错误
更新引用计数

假设单个磁盘写入是原子的 - 大多数磁盘都实现了这一点（例如，使用一个电容器来“完成”写入）

将在本周四的复习课上详细介绍如何在没有这个假设的情况下，文件系统如何确保此属性（或参见笔记 - 第9.2节讨论）；基本思想是写入两个副本的块，使用校验和返回有效的副本

"恢复"：（重新启动处理以确保全部或无）
扫描文件系统，修复引用计数
如果存在(fnew)和存在(file)
删除(fnew) // 文件是旧的还是新的

问：何时是提交点？ 在链接中修改目录块后

朝着我们的原子性愿景迈出了一步；问题：

- 每次都必须复制整个文件，即使我们只修改了几条记录

- 只适用于一个文件 -- 如果更改跨越多个文件怎么办？
(例如，客户和账户表)

- 可以将要编辑的所有数据库放在一个目录中，
原子性地重命名新目录

- 需要事先修复目录结构

- 并发性工作机制不清楚。 如果我们只允许一个用户同时访问，可
以实现隔离，但这非常限制。

(我们宁愿允许不同的用户同时访问文件的不同部分)

下次我们将解决这些限制，但在此之前，让我们介绍一下这种
方法的一些关键特点：- 我们写入两份数据副本

- 切换到新副本
- 删除旧副本

我们所有的原子性方案都将按照这种方式工作 --
恢复性的黄金法则：

永远不要覆盖唯一的数据副本。

下次我们将开发一个适用于多个对象的通用原子性保持方案。

不需要我们复制对象的所有块--只需要修改的块。

思路是记录我们所做的每个更改的日志，反映更改之前和之后的对象的状态。 然后
可以使用此日志从任何中间状态转到“之前”状态或“之后”状态。

例如：数据库A，数据库B；A10 v0 B7 v0
之前：

A10 v0 B7 v0

之后：

A10 v1 B7 v1

日志：

W(A10 v0,A10 v1)
W(B7 v0, B7 v1)

（日志记录包含旧版本和新版本）)

假设崩溃，重新启动，发现磁盘上有A10 v1，B7 v0
可以使用日志将两者都恢复为版本0或版本1

6.033 计算机系统工程

2009年春季

6.033 第17讲--日志记录

山姆・麦登

上次：介绍了原子性

两个关键思想：
- 全部或者什么都没有
- 隔离

我们看到了事务，这是一种确保原子性的强大方式

开始
转账(A,B,10)
借记(B,10)
提交（或者中止）

开始讨论

没有隔离的全部或者什么都没有
（下次讨论隔离）

我们看到了影子副本的概念，你将更改写入到一个次要副本中，然后
原子性地安装该副本

（展示幻灯片）

这涉及到原子性的关键思想 - 黄金法则

永远不要覆盖唯一的副本

大多数原子性方案都是这样工作的 - 你对数据的某个独立的副本进行更改，然后有一种方法在需要时切换到该新副本。

如果你在进行更改的过程中崩溃，旧副本仍然存在。 通常当
你崩溃时，因为事务不完整，你只需恢复到旧副本。

影子副本方法有一些限制；它们是什么？

- 一次只能处理一个文件（可以使用影子目录等方式来修复）

- 需要我们对文件进行完整的复制

在许多地方使用影子副本 -- 例如，文本编辑器，(emacs)

今天我们将学习一种适用于全有或全无原子性的通用方法，该方法解决了这些限制 -- 记录日志。

基本思想是，在每次更改后，记录所更改对象的前后值。

让我们来处理银行账户余额；假设我们在内存中有以下表格

账户ID	余额
A	100
B	50

假设我执行以下操作

开始
借记 (A,10)
借记 (B, 20)
结束

开始
存款 (A,50)

日志中保存哪些内容：

- 事务开始 / 提交 / 中止
  - 事务ID
- 更新
  - TID
  - 更新的变量
  - 之前（撤销）/ 之后（重做）的状态

日志：

开始 1
更新 1 A 之前：100；之后：90
更新 1 A 之前：50；之后：30
提交 1

开始 2
更新 2 A 之前: 90; 之后: 140
crash!

这个日志现在记录了我需要确定一个对象的值的所有内容

```
读取(var, 日志):
        cset = {}
        对于 r 在 日志(len-1) 中 ... 日志(0)
                如果 r 是提交
                        cset = cset U r.TID
                如果 r 是更新
                        如果 (r.TID 在 cset 中且 r.var = var)
                                返回 r.after
```

但这样做非常慢，因为我必须在每次读取后扫描日志

什么是提交点？ (当提交记录进入日志时)

(写入操作很快，因为我只是顺序地追加到日志中)

如何使这个过程更快？ 保留两个副本 -- "单元存储" -- 例如，实际表内容，除了磁盘上的日志。 读取可以直接从单元存储中读取当前值，写入可以同时写入两个位置。

```
读取(var)
        返回读取(单元存储, var)

write(TID, 变量, 新值)
        append(日志, TID, 变量, 存储单元(变量), 新值)
        write(存储单元, 变量, 新值)
```

让我们看看会发生什么：

状态：

A	100 --> 90 --> 140
B	50 --> 30 -> 60 -> 30 -> 40

开始	日志：
借记 (A,10)	开始 T1
借记 (B, 20)	更新(A, T1, 100, 90)
提交	更新(B, T1, 50, 30)
	提交 T1
开始	开始 T2

存款 (B,30)	更新(B,T2, 30, 60)
中止	中止 T2
开始	开始 T3
借记 (B,20)	更新(T3,B, 30, 40)
提交	提交 T3
开始	开始 T4
存款 (A,50)	更新(T4,A,90,140)
崩溃	中止 T4

崩溃后，存储单元中的A值是错误的。我们应该怎么办？

需要添加一个恢复过程，并且需要在日志上进行反向传递，以撤销未提交事务的影响。（撤销"失败者"）

还有其他需要做的事情吗？

还需要在日志上进行正向传递，以重做在写入单元存储之前但在追加操作之后的单元存储写入的影响。（重做"胜利者"）

恢复（日志）：

```
cset = {}
对于日志中的每个记录r（从log[len(log)-1 ... 0]）//
        撤销如果r是提交，则cset = cset U r.
        TID如果r是更新且r.TID不在cset中，
                则写入（单元存储，r.var，r.before
）对于日志中的每个记录r（从log[0...len(log)-1]
        ）//重做如果r是更新且r.TID在cset中
                ，则写入（单元存储，r.var，r.afte
```

为什么要进行反向传递来进行撤销？ 必须进行扫描以确定cset。 如果我们进行这个反向扫描，我们可以与撤销结合起来。

为什么前向传递到REDO？ 希望单元存储器显示最近提交的事务的结果。

其他可能的变体 - 例如，先重做再撤消，向前传递以进行撤消（使用先前的扫描来确定c集合），等等。

如果在恢复过程中崩溃怎么办？

好的 - 恢复是幂等的

示例：崩溃后：A = 140; B = 40
cset = {1, 3}

撤消
A -> 90
B -> 30
重做
A -> 90
B -> 30
B -> 40
T1和T3已提交，并按照此顺序。

优化1：“不要对已经在单元存储器中更新的内容进行重做”

可以通过不对已经在单元存储器中反映的更新进行重做来对此进行一定程度的优化。 最简单的方法是在每个单元格上记录一个最近在单元格中反映的日志记录ID。图表：

更新(tid,logid,var,before,after)

(在我们的例子中，我们只需要撤销A->90)

问：为什么我在更新单元格存储之前写日志记录？ 如果我按相反的顺序做这些会发生什么?
单元格存储将具有值，但可能不在日志中
恢复将无法正确撤消更新的影响

这被称为"先写日志"--在更新数据之前始终先写日志

优化2：
延迟单元格存储更新

可以在内存中保留单元格存储的缓存，并在需要时刷新。并不重要

只要日志记录先写入，日志记录将确保我们对已提交的事务重新执行任何单元格存储更新。

读取(var)：
    如果var在缓存中
        返回缓存[var]
    否则
        val = 读取(cell-storage, var)
        将(var,val)添加到缓存
        返回val

写入(var,val)
    append(log,TID, var, cell-storage(var), newval)
    add (var,val) to cache

优化3:

截断日志。

如果日志无限增长，恢复需要很长时间，并且浪费磁盘空间。

问题: 哪些日志前缀可以丢弃?

答案: 任何已完成事务的部分，其更改明确地反映在单元存储中（因为我们永远不需要重新应用这些更改）。

想法:

检查点:
    写检查点记录
    刷新单元存储
    截断检查点之前的日志

写检查点。
将所有未完成的更新写入磁盘的单元存储。 这个状态写入磁盘明确地反映了检查点之前的所有日志记录的更新。 在检查点之前截断日志。

大多数数据库通过将日志分割成多个文件来实现截断
这些文件链接在一起。 截断涉及删除不再
使用的文件。

图表：

日志 --

良好的读取性能 -- 单元存储，加上内存缓存

不错的写入性能 --

必须写日志，但可以顺序写入；

单元存储延迟写入

恢复速度快 -- 仅读取未截断的日志部分

限制 --

外部操作 -- "发放钱款"，"发射导弹"

无法将它们与磁盘更新合并为一个原子操作

全部写两次 -- 更慢吗?

下一次 --

多个并发事务的隔离

6.033 计算机系统工程

2009年春季

第18讲 -- 隔离 + 并发操作

山姆・麦登

关键思想:

串行等价
两阶段锁定
死锁

到目前为止，我们只想象了一次只有一个事务 -- 实际上，整个系统（数据库、文件系统等）都被一个大锁包围。

这有什么问题？ 为什么一次只运行一个事务比串行运行所有事务利用系统资源更少？ （可能会期望相反的情况是真的，因为在事务之间切换可能会有一些开销！）

- 很难利用多个处理器
- 可能希望在一个事务等待磁盘或CPU时代表另一个事务执行处理

我们在隔离中的目标是串行等价性 - 希望事务的最终结果与一个接一个地运行事务相同

A = 50
B = 10

xferPercent(A,B,.2) xferPercent(A,B,.2)

A = 40
B = 20

A = 32
B = 28

xfer(A,B,10)
RA // t = .1A
WA // A = A - t
RB
WB // B = B + t

示例"日程表":

RA
WA // a= 40
        RA
        WA // a= 32
RB
WB // B = 20
        RB
        WB // B = 28

可串行化吗？是的。为什么?
结果是相同的。

RA
        RA
WA // a= 40
        WA // a = 40
RB
WB //B = 20
        RB
        WB // B = 30

不可序列化!

是否有一些测试可以进行序列化?

第二个例子出了什么问题?

T1对A的读取先于T2对A的写入，并且T2对A的读取先于T1对A的写入。

为了形式化，定义"冲突"--如果T1中的操作o1和T2中的操作o2冲突，则o1和o2都不是写操作，并且都是对同一个对象的操作。

(不用担心两个读操作。)

如果在一对事务T1和T2中，所有冲突的操作都按照相同的顺序排序，那么调度是可序列化的--例如，T1-T2或T2-T1

为什么？假设它们不是。 那么一个事务可能在另一个事务更新之前读取某些内容，并在之后更新它，就像上面的例子一样。首先

事务的效果会丢失!

测试可序列化的简单方法：冲突图

用事务作为节点绘制图形
如果T1中的op1与T2中的op2冲突且op1在op2之前，则从T1到T2绘制箭头

例子：

现在我们理解了可串行化调度的含义，我们的目标是设计一个能确保其的锁定协议。

我们希望确保所有冲突操作以相同的方式排序。使用锁来实现。 不要求程序员手动获取锁。

相反，事务系统在读取对象时自动获取锁。

锁定协议：
在读取/写入对象之前，获取其锁定（如果锁不可用，则阻塞）

我可以立即释放吗？（不行！示例）：

```
T1              T2
锁定A
                锁定A
        RA      阻塞
        WA
释放A
                锁定A     //T2 "潜入"并进行更新，违反了可串行性
                锁定B
                        RA
                        WA
                        RB
                        WB
                释放A
                释放B

锁 B
        RB
```

WB
释放 B

在更新之前，T2不应该能够看到 B 的暴露值 -- 不可序列化。

锁定协议：
在读取/写入对象之前，获取其锁定（如果锁不
可用，则阻塞）

事务提交后释放锁

锁 A		锁 A (块)
	RA	I
	WA	I
锁 B		I
	RB	I
	WB	I
提交		I
释放 A, B		I
		v
		RA
		WA
		锁 B
		RB
		WB

(强制 T2 在 T1 之后完全执行 -- 显然放弃了一些并发性；注意，如果 T2 先访问其他对象，例如 C，那就不是问题。

一分钟内将解决这个限制。)

问题：死锁

如果 T2 首先尝试获取锁 B 怎么办?

没有事务能够取得进展。

我们可以要求事务总是按相同的顺序获取锁吗？ 不行，因为事务可能不知道它们将要获取的锁 -- 例如，被更新的事物可能依赖于被读取的账户（假设我们从所有值 > x 的账户中扣除）。

那么我们对死锁怎么办呢？ 简单：中止其中一个事务并强制其重新运行。 这是可以的，因为应用程序必须始终准备好事务系统崩溃并中止其操作的可能性。

我们如何判断事务是否发生了死锁?

有两种基本方法:

1）如果一个事务等待锁的时间超过t秒，就假设它发生了死锁并重新启动它。

（简单，可能会将一个长时间运行的事务误认为是死锁。）MySQL就是这样做的。

2）构建一个“等待图”——如果T1正在等待T2持有的锁，则从T1到T2绘制一条边。如果存在循环，则存在死锁。

例子:

使锁定协议更高效的优化:

优化1:

一旦一个事务获得了所有的锁，它就可以在没有其他事务干扰的情况下继续进行。

锁定点:
它将在与其冲突的任何其他事务之前进行序列化。

如果一个事务在读/写一个对象后完成，它可以释放该对象上的锁而不影响序列化顺序。 由于它不会再次使用该值，所以不必等到事务结束。

例子:

锁A		
	RA	锁A（块）
	WA	
锁B <--- 锁点		
释放A		
		RA
		WA
		锁B（块）
	RB	
	WB	
释放B		

RB
WB

这被称为两阶段锁定：

第一阶段 -获取锁，直到锁点

第二阶段 - 释放锁，在完成对象操作和锁点之后

优化2：

共享锁与独占锁

注意，同一对象的两个读操作不会冲突 - 例如，如果我有两个只读事务，我可以随意交错它们的操作并仍然得到相同的答案。 然而，协议不允许这样做。

想法 - 有两种类型的锁：共享（S）锁和独占（X）锁。 可以有多个持有S锁的事务，但只能有一个持有X锁的事务。

如果只有一个S锁，事务可以从S升级到X。

两阶段锁定协议带有共享锁

在读取对象之前，对其进行S锁定

在写入对象之前，对其进行X锁定

在锁定点之后释放锁定

实际上，使用了两阶段锁定的变体：

1）在事务提交之后才释放写锁？为什么？

T2在T1提交之前读取了T1写入的A -- 现在T1中止，T2也必须中止。

严格的两阶段锁定

2）在事务提交之前不释放任何锁定？为什么？

无法确定何时完成锁定

严格的两阶段锁定

6.033 计算机系统工程

2009年春季

Lec 19：嵌套原子操作和多站点原子性

山姆・麦登

到目前为止，我们专注于事务仅在一个系统上运行的情况，
并且所有事务要么提交要么中止。

今天 -- 还有几个额外的目标

- 嵌套原子操作 -- 具有可能成功或失败的子操作，不影响整个操作的结果（嵌套原子操作）

- 多站点操作 - 跨多个站点的事务
(分布式事务); 为什么呢?

- 性能 (2台机器存储数据库，两台机器参与
查询或更新他们所拥有的部分数据库。)

- 管理 - 两个不同组织拥有的两个数据库，
希望运行跨两个数据库的命令。

例子：旅行网站：

单一系统：

希望嵌套操作允许子事务完成而不提交，并且然后提交子事务

开始 (父事务 P)
开始 (A)
预订 JB
结束
开始 (B)
预订 USAir ---> 如果此操作中止，JB 预订不会丢失
结束 ---> 如果此操作中止，P 可以尝试其他操作而不会丢失 JB 预订
结束
父事务 P
子事务 A I 仅在提交时执行

子事务 B I P 提交

目标：

只有在 P 提交后，A + B 的影响才对外可见。
(一旦 P 提交，A + B 的影响都可见)

A + B 可以独立中止

B不应该看到A的影响，除非A已经完成。

一旦A到达并且它不应该中止，除非A中止（因为B可能使用它的结果
）

我们将说已经到达其"结束"的嵌套事务是"暂时提交"的--它准备好提交，但
正在等待其父事务的结果。

协议的变化：
锁定：

何时释放A/B的锁？在P提交之前吗？ 不是--因为P可能中止！只有在P提交之后
才能释放。

但是B应该能够在A到达"结束"后读取A的数据。

更改锁获取协议以检查是否正在等待来自具有相同父事务的暂时
提交事务的锁

日志记录：
需要为子操作保留单独的开始/结束，以防其中一个中止并且我们崩溃，以确保
其效果被撤消。

在日志处理期间，当我们遇到子事务时，只有在其父事务提交时才想要提交
它。

开始 P
开始 A
更新座位图1
尝试提交 A (父节点 P)
开始 B
更新座位图2
尝试提交 B (父节点 P)
结束 P

恢复：向后扫描，确定胜者，撤销失败者

（例如，中止的子操作或其父操作中止的子操作）向前扫描，重做胜者多站点操作

类似的协议和规则 - 假设现在 JB 和 USAir 是独立的预订系统，每个系统都有自己的数据。

有一个协调员，用户连接到该协调员并在这些子系统上进行预订。图表（带有互联网）

与之前一样，只有当 USAir 提交时 JB 才提交，反之亦然。

比之前更困难，因为现在 JB / USAir 可能会独立崩溃，消息可能会丢失/重新排序。

每个站点都有单独的日志，包括协调员。

为了处理消息丢失，将使用类似一次性 RPC 的协议。

图表：

基本协议如下：

协调员将任务发送给工人

一旦所有任务完成，协调员需要让工人进入准备好的（暂时提交）状态

这里的暂时提交意味着如果协调员告诉他们这样做，工人们将肯定提交；协调员不能单方面提交

协议（无丢失）：（准备，投票，提交，确认）

为什么不在他们完成工作后向所有站点发送提交消息？

其中一个可能在提交处理过程中失败，这将要求我们能够中止已经提交的子任务。

假设消息丢失了？ 使用超时重新发送准备和提交消息

崩溃了？ 需要确保工人的日志能够恢复到暂时提交的状态。

日志记录：
在"是"投票之前，在工人上写入TC记录。

提交记录仍然在协调员上写入 - 这是整个事务的提交点

协调员还写入"完成"消息

假设工作人员崩溃：

准备之前?

准备之后?

假设协调员崩溃：

准备之前

发送一些准备之后

写入提交之后?

写入完成之后?

协调员如何回应"TX?"查询? 它是否永远保持所有事务的状态? (不会 - 一旦它收到所有工作人员的确认，它就知道他们已经收到结果。)

请注意，工作人员在听到协调员的提交/中止之前不能忘记事务的状态，即使他们崩溃。 这使得协议在跨组织环境中有些不切实际。

那么应该怎么办呢?
使用补偿操作（例如，航空公司允许您在预订后的几个小时内免费取消购买。）

2PC提供了一种分布式节点达成一致（例如，提交或中止）的方法。但请注意，它只保证所有节点最终了解结果，而不是在同一时刻达成一致。

"两将军问题"（幻灯片）

无法保证在有限时间内达成协议（尽管最终很可能会发生）

6.033 计算机系统工程

2009年春季

今天：一致性和复制

山姆·麦登

考试：平均分55分。 如果你的成绩很低 - 不要担心。重要的是你相对于平均水平的表现。

在最近的几堂课中，我们看到了如何使用日志记录和锁定来确保原子操作，即使在面对故障的情况下也是如此。

但是 - 我们可能希望做得比仅仅从崩溃中恢复更好。 我们可能希望能够屏蔽系统的故障 - 使系统更可用。

之前我们已经看到复制是一个有助于实现这一目标的强大工具。 今天我们将更详细地研究复制，并探讨在维护复制系统时出现的一些问题。

复制在本地区域和广域网中都被使用 - 例如，在机房的同一机架上有两个副本，或者在不同建筑物、州等地有两个副本。

广域网复制对于灾难恢复非常重要。 "飓风的一半宽度。"

让我们来看看如何将复制应用到系统中：

（向两者发送请求；如果其中一个宕机，只发送给另一个。）

可能出现什么问题：

（网络分区。其他客户端向其他机器发送请求。现在它们不同步了。）

解决方案：

- 最终一致性 - 在事后同步的一种方式（暴露的不一致性 - 如果我们没有一个复制系统，这种情况可能永远不会发生。）

- 严格一致性 - "单副本一致性" - 确保外部用户无法区分一台机器和多台机器之间的区别。

严格一致性和可用性在面对网络分区时根本无法兼顾。

在上述示例中，当最终一致性方法允许任何一个客户端接收答案时。

在严格一致性方法中，我们能做的最好的办法是指定一个副本作为"主"（或权威），并允许与其通信的客户端继续进行。 其他客户端 - 与其他副本通信 - 将无法获得服务。

最终一致性

通常比严格一致性更容易提供。 通常的工作方式是为每个数据项关联一个时间戳（或版本号）。

定期比较时间戳的副本，使用最新的。 这就是Unison does的功能。

DNS是最终一致性的一个例子--假设我改变：

将db.csail.mit.edu的IP地址从X更改为Y

效果不会立即可见；相反，当DNS记录的超时到期时，它将传播。 记住--DNS服务器会缓存查找结果一段固定的时间。

（展示幻灯片）

这意味着具有IP地址X的机器可能在一段时间内收到对db.csail.mit.edu的请求。

在这里，不一致被认为是可以接受的。 在这里提供严格的一致性将会很困难：我必须联系世界上的每个缓存，并确保它们

已更新，或者必须没有缓存，并且每个请求都要来自
csail.mit.edu的每次查找。两者都不可行。

最终一致性的一个挑战是如果对同一数据进行并发更新时该怎么办。 在DNS中不是一个问题，因为每个子域名（例如，csail.mit.edu）都有一个主服务器来完全排序更新。 但是在像Unison这样的系统中，我们可以对同一数据项进行多次更新，这是有问题的，需要一些手动的冲突解决。

下一次讲座将会看到另一个最终一致性的例子，并了解并发更新同一项可能导致的一些额外问题。

严格一致性 - 复制状态机

基本思想是创建*n*个系统副本。相同的起始状态。 相同的输入，按相同的顺序。 确定性操作。 确保相同的最终状态。

如果一个副本崩溃，其他副本有冗余的状态副本可以替代它。当该副本恢复时，它可以从其他副本复制其状态。

但是网络分区怎么办 - 副本可能会错过一个请求，甚至可能不知道它错过了！

如果来自不同分区的不同客户端发出请求，则副本
可能会看到不同的请求。图表：

一种选择：创建一个完全有序的请求排序服务。 当发生网络分区时，一个副本可能会错过一些请求，但它可以检测到这一点

并从其他副本请求这些操作，然后继续进行。
图表：

(单点故障 - 一会儿会回到这个问题。)

退化版本：单主复制

图表：

所有更新都发送到一个单独的主节点，该主节点负责将更新传播到
副本。

看起来可能解决了问题，因为现在只有一个主节点。

如果主节点故障，切换到其中一个副本。这里有什么问题?

可能已经有一些写操作传播到某些副本，但没有传播到其他副本。 副本如何知道当前状态是什么?

(它们不知道 - 可能认为它们可以相互交流并找出它们中任何一个已经看到的最新状态。但是，崩溃的主节点可能看到了它们没有看到的东西。)

两阶段提交解决这个问题吗?

（不。假设以下

M R

WA

WA

准备

准备

好的

提交

-----崩溃！-------

)

一种方法是简单地容忍一些不一致性 - 例如，接受副本可能不知道某些事务的结果。

实际上，在广域复制中就是这样做的 - 例如，在数据库系统中。

客户端如何知道要与哪个服务器通信？

排序服务

如何构建可靠的排序服务？

1个节点 - 可用性低。

使用复制状态机。图示：

仍然存在分区问题。 通常的解决方案是在大多数节点上达成一致意见后接受答案。
挑战：网络分区、消息重排序、丢失消息。

这个问题被称为一致性：

一致性协议（草图）

领导者：

next(op)

accept(op)

commit

工作节点

agree(op)

accept_ok(op)

阶段1：领导者节点向所有其他节点提议下一个操作（提议）

阶段2：如果大多数节点同意，它告诉所有其他节点这是下一个操作（接受）

阶段3：一旦大多数节点确认接受，
发送下一个操作！

这个协议的原因是我们希望在领导者离开时允许其他节点继续运行。

其他节点：

如果在随机超时后无法联系到领导者，则自己提出作为新的领导者。 如果大多数节点同意，则它们成为新的领导者。

（节点只响应来自当前领导者的准备/接受请求。）一旦选择了新的领导者，

它会检查是否有任何节点已经接受了任何东西。 如果没有，它会重新启动协议。

如果一些节点已经同意，但没有节点看到接受，那没关系，因为该事务可能尚未提交，所以领导者可以安全地中止它。

一旦领导者从大多数节点收到接受_ok，即使存在网络分区，它知道自己处于少数分区，多数分区将至少有一个已经提交的节点。

否则，多数分区可能只有一个准备好的节点，新节点可能会认为该事务未提交并提出其他建议。

大多数之外的节点不一致，需要恢复，即使它们已经发送了accept_ok消息。

当n/2+1个工作者写入accept(ok)时，达到提交点。

示例1 - 网络分区

示例2 - 领导者故障

这些协议协议非常棘手，这就是为什么我只是在这里勾勒出想法。 学习更多，请参考6.828课程。

协议草图：

```
N：副本数量（奇数，例如3或5）
投票数=0
投票者={}
R=0 //必须在重启后保持
opQ=new Queue() //必须在重启后保持

执行(op)：
        opQ.append(op)
        广播("下一个操作是opQ[R]，在第R轮");

定期执行：
        如果(R < opQ.len)
                广播("下一个操作是opQ[R]，在第R轮");

下一个操作(op,来自,轮次)：
        如果(R < 轮次)：
                等待看是否还有更多的选票投给R
                如果（R < round）：
                        恢复
                        返回

        如果（op == opQ[R] && from不在选民中）：
                投票++;
        选民 = 选民 U 来自


        如果（投票 > N/2）：
                通知工人（opQ[R]）
                R = R + 1
                选民 = {}
                投票 = 0

        如果（N/2 - 投票 > N - len(选民)
                死锁
```

6.033 计算机系统工程

2009年春季

安全介绍尼古拉·泽
尔多维奇==========
===

今天的关键思想：
    安全的关键是了解攻击者能做什么
    原则：减少信任（最小特权），机制经济性

--- 第1个板块 ---

安全/保护
    渗透计算机系统设计
    如果你不在设计初期就设计好，以后很难修复
    就像命名、网络协议、原子性等一样
    会影响上述所有内容

举例说明安全问题
    幻灯片：一般安全统计数据
      关键问题，允许攻击者控制
      Windows机器--大约每周一次
    幻灯片：因此，中国控制大使馆的计算机并不令人意外
    幻灯片：甚至像起搏器这样的医疗设备也容易受到攻击

那么什么是保护？ 回到第一个板块
    防止坏人访问
    允许好人访问
    策略[有很多，我们无法涵盖所有可能性]
    [..所以和其他主题一样] -> 机制

--- 第二个板块 ---

现实世界与计算机安全
    相同之处：
      锁 - 加密
      检查点 - 访问控制
      法律，惩罚
    不同之处：
      攻击快速，廉价，可扩展
          攻击1台或100万台机器所需的努力相同
          可以攻击世界各地的机器
          无需亲自在机器旁
          没有强烈的身份概念
      全球化；法律很少

--- 第三个板块 ---

策略目标：积极 vs 消极
    尼古拉可以读取grades.txt -- 容易
      为什么容易？ 构建系统，如果不起作用，就不断修复直到起作用
    约翰无法读取grades.txt -- 困难
      似乎与上述相反
      我们可以尝试让约翰登录并尝试访问grades.txt
      不够：必须量化约翰可能获取grades.txt的所有可能方式
          尝试直接访问存储文件的磁盘
          尝试通过浏览器访问（也许Web服务器有权限？）
          尝试在Nickolai的编辑器退出后读取未初始化的内存

尝试拦截正在读写grades.txt的NFS数据包

试图向您出售恶意副本的Windows
试图从服务器中取出物理磁盘并复制它
试图从垃圾桶中窃取打印输出的副本
打电话给系统管理员并假装是Nickolai
很难说"无论发生什么，约翰都不会得到grades.txt"

仅通过一个接口控制访问还不够
必须确保所有可能的访问路径都是安全的

我们已经在6.033中看到了一些积极的目标（例如命名）
也有一些负面目标（事务不能被"损坏"）
安全性更难，因为攻击者可以做很多事情
在交易中，我们知道会发生什么（在任何时候崩溃）
大多数安全问题都是这样的负面目标

--- 第4个板块 ---

威胁模型
最重要的是要了解攻击者能做什么
然后你可以设计你的系统来防御这些事情

C -> I -> S

典型设置：
客户端名为Alice，服务器名为Bob
网络中的攻击者（路由器），Eve，正在窃听
或者，恶意对手Lucifer可以发送、修改数据包

攻击者控制客户端吗？服务器？

常见的假设：
没有物理、社会工程攻击
只拦截/发送消息
可能会或可能不会危及服务器、客户端

即使在单台机器上也适用这个图像
来自不同进程的 用户向操作系统内核发起调用

也要考虑成本（安全和不安全都有价格）
便利性、硬件费用、设计等等

板块的右侧：
基本目标
- 认证[幻灯片：肯塔基传真]
- 授权[谁有权释放囚犯?]
- 保密性 注意：与认证非常不同!
- 审计
- 可用性

--- 板 5 ---

策略 / 机制
硬件：限制用户代码
机制：虚拟内存，监管位

认证：内核初始化页表，监管位
硬件知道当前状态
授权：可以访问当前页表中的任何虚拟内存地址
无法访问特权CPU状态
Unix：私有文件
机制：进程，用户ID，文件权限
认证：用户密码，当用户启动进程时
授权：内核检查文件权限
防火墙：限制连接
机制：数据包过滤
认证：连接建立
授权：允许/禁止连接列表
看似脆弱的机制，但在实践中非常强大
银行ATM：只能从你的账户中提取，最高到余额
机制：服务器进程中的应用级检查
认证：卡片和PIN
授权：跟踪账户余额
密码学：下一讲

--- 第6个板块 ---

挑战
错误
构建无错误系统很困难，编写完美代码
预计会出现错误，在设计系统时尽量使其安全
明天的讲座中，我们将研究其中一些错误
完全的中介
需要仔细的设计
幻灯片：paymaxx错误
许多机制：难以强制执行一致的策略
希望确保遵循银行政策
我们有哪些机制？
虚拟内存隔离进程
内核，文件系统实现访问控制列表
银行ATM实施自己的检查
网上银行可能实施其他检查
银行员工使用的系统有其他检查
网络中的不同位置有防火墙
层之间的交互
[缓存/定时，命名，内存重用，网络重放]
幻灯片：符号链接攻击的命名问题
幻灯片：逐个字符检查密码

--- 第7个板块 ---

安全网方法
要做到偏执--明确假设
攻击者将尝试所有可能的边界情况
考虑环境
如果你依赖网络安全，请检查是否有开放的无线网络
如果你重复使用，依赖于另一个组件，请确保它是安全的
代码用于在非网络系统上运行，用于网络上使用？
从未预料到会处理恶意输入
考虑使用的动态
假设只有尼古拉应该访问grades.txt
谁可以为成绩文件指定权限？

谁可以修改Athena上的编辑器？ 或者对其设置权限？
谁可以控制该文件的名称转换？

深度防御
即使你在一个"安全"的公司网络上有一个服务器，仍然希望要求密码。 如果有人带来了一个被感染的笔记本电脑怎么办？

黑板的右侧：
人类：最薄弱的环节
- 用户界面
- 安全的默认设置

--- 第8个黑板 ---

设计原则
开放设计，最小化秘密
找出什么将区分坏人和好人
专注于保护那个，将其他一切公开
身份验证：ID公开，证明你是该ID的东西是秘密的
社会安全号码，信用卡号码在这方面失败
社会保障号码既用作身份证明，也用作身份验证的凭据
不清楚信用卡号的哪一部分是真正的秘密
一些收据会屏蔽前12位数字，其他则屏蔽最后4位数字
机制的经济性
简单的安全机制
多个安全机制相互干扰
努力将安全策略减少到现有机制

设计以最小化安全机制之间的"阻抗不匹配"通常在客户端和真实对象之间有多个应用层右侧：图表：客户端-Web应用程序-FS-磁盘假设这是存储用户税务数据的paymaxx
如果策略可以直接在对象上执行，那就不必信任服务器应用程序代码假设Obj是文件--机制是文件权限如果不同的用户将其数据存储在一个文件中，则无法使用操作系统保护如果我们仔细设计每个用户一个文件，可能可以使用操作系统

最小化TCB（受信任计算基础）
TCB（受信任计算基础）
通常不希望信任网络（下一讲将展示如何）
将应用程序分解为小组件，每个组件具有最少所需的
权限

6.033 计算机系统工程

2009年春季

尼古拉·泽尔多维奇

今天的关键思想：
开放设计
身份与验证器
验证消息与主体（消息完整性，绑定数据）
公钥身份验证

--- 新委员会 ---

安全模型
C - I - S
上次：
讨论了一些构建安全系统的一般思路
防御性设计：预期妥协，分解成部分，降低权限
上次的讲座中，看到了可能导致部分受损的示例
接下来的讲座：
假设我们可以设计端点正确且安全
（虽然困难，但暂时先这样）
找出如何在面对攻击者时实现安全
攻击者可以查看、修改和发送消息
我们想要实现的基本目标
服务器内部：守卫 - 服务

身份验证
授权
机密性

--- 新委员会 ---

基本构建块：加密
让我们看看如何实现加密
两个函数，加密和解密

C -> E -> I -> D -> S

军事系统，E和D是秘密的
封闭设计

问题：如果有人窃取了你的设计，你会陷入大麻烦
很难在不失去保密性的同时分析系统
构建安全系统的关键原则：最小化秘密！

--- 新委员会 ---

开放设计
巨大优势：如果有人窃取了设计和密钥，可以直接更换密钥
可以将系统与特定的秘密密钥分开分析
最小化秘密

设计系统的重要原则：
准确确定区分坏人和好人的秘密
保守秘密非常困难
知道什么是重要的将使你能够专注于正确的事情

相同的图表，但密钥进入E和D

对称密钥加密的示例：一次性密码本
    用随机位与消息进行异或运算，这些位是密钥
    快速描述异或运算，为什么会得到原始消息
    问题：密钥很大（但方案是完全安全的）

流密码：生成随机外观位的各种算法
    不再是完全不可破解的，只需要大量计算
    幻灯片：RC4

如果密钥被重用，则会受到攻击
    C->S：加密(k, "信用卡号码NNN")
    S->C：加密(k, "谢谢，...")

    对两个密文和已知的响应进行异或运算，以获取未知的请求消息！
    永不重用对称加密密钥！（一次性密码本！）

--- 新委员会 ---

以前需要共享密钥，不可扩展

RSA：公钥密码学
    用于加密和解密的密钥不同
    幻灯片：RSA算法
      简短的示例计算？

      p = 31, q = 23, N = 713
      e = 7, d = 283

      m = 5
      c = m^e mod N = 5^7 mod 713 = 408
      m = c^d mod N = 408^283 mod 713 = 5
    从d生成e，以及从e生成d很困难
    假设：分解N很困难！

    比对称密钥加密更加计算密集！重要属性：不需要每个参与方之间共享密钥为某人加密消息与解密消息不同。 服务器可以使用相同的密钥接收多个客户端发送的消息

    在实践中同样棘手
      如何表示消息？
      小消息是脆弱的
      大消息是低效的
      可以将消息相乘
      需要一种称为填充的东西

加密机制依赖于计算复杂性
    适当选择密钥大小--"有效窗口"

--- 新委员会 ---

主体认证
    主体/身份：表达你是谁的一种方式
    验证器：说服他人你的身份
      开放设计原则在这里有点适用

希望保持身份公开，验证器私有
关注区分好人和坏人的特点
通常有一个会面来达成可接受的验证器

验证器类型：板的右侧
现实世界：社会安全号码
糟糕的设计：混淆主体的身份和验证器
密码
假设用户是唯一知道密码的人，可以推断
如果有人知道密码，那一定是用户
服务器存储密码列表，如果泄露就是灾难
常见解决方案：存储密码的哈希值
定义一个密码哈希函数：
H(m) -> v，v很短（例如256位）
给定H(m)，很难找到m'，使得H(m') = H(m)
破坏了我们上次的时序攻击
理论上很难逆转
字典攻击：尝试短字符序列、单词
物理对象
磁卡：存储一个长密码，不是很有趣
智能卡：使用加密进行身份验证的计算机
生物特征
最古老的身份验证形式：人们记住面孔、声音
容易被盗取（你到处留下指纹、面部图像）
与密码不同，如果被破坏，很难更改
更像是一个身份而不是身份验证机制

需要信任/验证您提供身份验证器的对象！
假登录界面、假ATM机可以获取用户的密码/PIN
下一次复习中，您将更多地了解现实世界中发生的情况
网络钓鱼攻击：说服您对它们进行身份验证

--- 新委员会 ---

假设我们信任我们的客户端（例如笔记本电脑、智能卡等）
如何设计协议？
板：C - I - S图
客户端发送一条消息，内容是"购买10股谷歌股票"
简单版本：只需将密码通过网络发送
攻击者获得密码后，可以冒充用户
更好的版本？ 发送密码的哈希值
攻击者无法获取我们的密码（很好，可能）
但是哈希值现在同样有效--可以将其拼接到其他消息上！

** 需要同时进行身份验证和完整性验证 **

更好吗？ 包括消息的校验和，例如CRC
攻击者可以重新计算校验和！需要校验和有密钥

更好的方法是：发送[消息+密码]的哈希值，称为MAC
消息认证码
如果要这样做：请查阅HMAC
最好的方法是：建立一个会话密钥，尽量减少使用密码（长期秘密）
向对方发送一条消息，内容是"我将在一段时间内使用这个密钥"
使用该密钥对各个消息进行MAC

6.033 计算机系统工程

2009年春季

尼古拉·泽尔多维奇

今天的高级概念：公钥身份验证证书
　　颁发机构会话密钥

　　网络协议和攻击：要明确

认证，授权==========
=========

--- 第1个板块 ---

设置：客户端和服务器通过互联网发送消息
　　希望实现：认证，授权，机密性

上次：如何在共享密钥的情况下验证请求
　　股票交易示例：“购买100股谷歌”
　　密码，将请求和密码一起哈希

```
[密码]            [密码]
   C --- I --- S

    <-----------
    m，H（m + 密码）      m：“谷歌股票价格为100美元”

    ----------->
    m，H（m + 密码）      m：“购买100股谷歌”
```

--- 第二个板块 ---

现在：如果我们事先没有共享密钥怎么办？
　　例如，股票报价来自Web服务器

　　想要一对函数SIGN和VERIFY

	共享密钥	公共密钥
SIGN(m, K1) -> sig	H(m+K1)	m^e mod N
VERIFY(m, sig, K2)	sig==H(m+K2)	sig^d mod N == m
	MAC	签名

　　幻灯片：RSA提醒

　　告诉学生这两个示例都是有缺陷的
　　　　希望使用HMAC和更好的公钥签名方案
　　　　这些只是为了你的直觉

　　形式化以前基于密码的身份验证：MAC
　　新的：不同的签名，验证密钥
　　公钥签名的关键属性：其他人可以验证但不能生成！

对身份验证的SIGN，VERIFY的攻击[黑板的右侧？]
　　密码攻击

找出密钥或找到违反加密保证的方法
身份验证
修改
重新排序
扩展
拼接

例如，RSA如所述对乘法消息是脆弱的
还需要注意将字符串压缩为整数

--- 第三个板块 ---

在这种情况下，认证某人到底意味着什么?
你如何知道要为某人使用哪个公钥?
想法：分成验证名称，然后信任名称
有人将为我们管理名称

对称
S <-> PW <-> 信任

非对称
S <-> K1 <-> 名称 <-> 信任
\----V----/
相信有人告诉我们这些绑定

--- 第4个板块 ---

与服务器交谈，获得响应和带有服务器密钥的签名
现在与一个密钥命名服务器交谈，询问它，这个人的名字是什么?
键命名服务器响应，连同其自己的签名
必须提前信任键命名服务器
否则可能需要与另一个键命名服务器交谈...

SIGN("GOOG is $100", K1), K2

[C知道K2_ca]

C <---------- S
^
|
\----> CA
"K2是谁? "
SIGN("K2是quotes.nasdaq.com", K1_ca)

--- 板 5 ---

想法：实际上不需要一直查询键命名服务器
将键命名服务器的签名响应给原始服务器
通常称为"证书"

[K2_ca]
C <---------------- S

m: "GOOG is $100"
SIGN(m, K1_q), K2_q
SIGN("K2_q是quotes.nasdaq.com", K1_ca)

委托：证书可能被链接
否则每个人都必须与根证书颁发机构交谈
这大致是DNSSEC的工作原理
用户信任顶级证书

SIGN("K2_q是quotes.nasdaq.com", K1_nasdaq), K2_nasdaq
SIGN("K2_nasdaq是*.nasdaq.com", K1_ca)

整个链条必须是可信的!
实际上，构建链条的最薄弱方式就是问题所在

这些认证机构中的一个如何决定下一个人是谁?
过去你必须去市政府办公室获取营业执照
可以指定任何你想要的名字，可能会找到一些宽松的城镇
现在只需通过你的域名注册的电子邮件进行验证
一个受欢迎的认证机构会发送电子邮件给webmaster@domain.com询问是否可以

幻灯片：很容易伪造这个过程，即使对于知名公司也是如此!
重申：链条中最薄弱的环节才是最重要的

幻灯片：很少使用的机制对安全性来说是一场灾难
Verisign为此设计了一个证书吊销列表
不是常规工作流程的一部分，所以从未经过测试!
事实上，当需要使用它时，没有人知道如何找到这个证书吊销列表!

实际上，用户会认为一个安全的网站是安全的，就此结束
不会费心考虑他们实际上正在与哪个服务器通信
现在有一种新型的证书，涉及“扩展验证”
弹出一个绿色的栏，显示经过验证的公司名称

替代方案：盲目信任和记住密钥
第一次容易受攻击
SSH
在浏览器中添加证书异常

--- 第6个板块 ---

好的，现在双方都有公钥，接下来呢?
非对称加密/签名计算成本高昂
大多数协议使用公钥来建立对称会话密钥
然后使用对称加密或MAC进行后续消息

会话密钥协议：Denning-Sacco
A想与B交流，需要建立对称会话密钥
1. A -> S: A, B
2. S -> A: SIGN({A, Ka}, Ks), SIGN({B, Kb}, Ks)
3. A选择一个会话密钥Kab
A -> B: SIGN({A, Ka}, Ks), ENCRYPT(SIGN(Kab, Ka), Kb)
4. A和B使用Kab进行对称加密/MAC通信
(通常不建议将同一个密钥用于两个目的，所以
也许Kab实际上是两个密钥，一个用于加密，一个用于MAC)

--- 7号板 --- [向右边?]

期望的目标

1. 保密性：只有A、B知道Kab
2. 身份验证：A、B知道对方的身份

这是真的吗?
1. A知道Kab，而唯一能解密消息3的人是B
2. A知道唯一能解密Kab的人必须是B
   B知道唯一能发送Kab的人是A（签名！）

结果发现协议是有问题的!
接收者无法真正推断对方是A！假设B是恶意的（例如A访问了一个恶意网站）可以冒充A向其他方（例如另一个网站）发送{C:Kc}_Ks3. B -> C:

SIGN({A, Ka}, Ks), ENCRYPT(SIGN(Kab, Ka), Kc)

接收者能推断的是A签署了这条消息，但这有什么用呢?
它想要使用Kab与某人交谈，但不一定是我们（收件人）！

6.033 计算机系统工程

2009年春季

尼古拉·泽尔多维奇

今天的高级想法：
- 对网络协议的攻击
- 构建安全协议的原则
- 安全通信通道抽象；加密和消息认证码
- 授权：列表 vs 票据
- 如何将所有内容整合在一起（HTTP中的安全性）

协议安全性========
==========

幻灯片：提醒丹宁-萨科协议

--- 第1个板块 ---

协议级攻击
- 冒充：密码特别糟糕[如果在多个网站上使用]
- 重放
- 反射

幻灯片：提醒丹宁-萨科协议的破解
幻灯片：如何修复丹宁-萨科协议

协议目标[白板右侧]
- 适当性[明确性？]
  - 明确的上下文，命名原则
- 前向保密性[泄露的密钥不会揭示过去的通信]
  - 会话密钥，版本号
- 新鲜度[区分旧消息和新消息]
  - [稍后：]时间戳，随机数

--- 第二个板块 ---

重放攻击示例：

C -> S："购买100股GOOG"，HMAC(m，pw)

攻击者多次重发消息！

有什么解决办法？ 包括一个随机数、序列号、时间戳等

--- 第三个板块 ---

反射攻击示例
- 假设爱丽丝和鲍勃共享一个秘密（例如密码）
- 希望确保对方在场/活着
- A和B之间共享密钥K

A -> B：加密（Na，K）
B -> A：Na

B -> A：加密（Nb，K）
A -> B：Nb

攻击：露西法假装是爱丽丝，而爱丽丝实际上已经断开连接
- 鲍勃要求露西法使用他们共享的密钥解密挑战
- 露西法没有密钥，但可以要求鲍勃向爱丽丝证明自己

鲍勃将解密自己的挑战并将其发送给露西法!

--- 第4个板块 ---

在构建系统时，很难理解所有这些单独的事物
这些构建块太低级了
常见解决方案：安全通信通道的抽象

C <===> S

提供机密性和身份验证，或仅提供身份验证

+ 长期通信
- 短期会话，多个节点
- 离线消息，在模型之外[例如仅机密性]

对于 -：必须使用我们一直在讨论的低级抽象

计划：使用类似丹宁-萨科的协议建立会话密钥
或者两个密钥，一个用于加密，一个用于MAC
使用会话密钥进行对称加密以实现安全通信通道

需要在消息中包含序列号以避免重放和拼接攻击

示例：浏览器中的SSL/TLS
幻灯片：来自维基百科的SSL/TLS概述

[[加密攻击：
仅密文
已知明文
已选择明文
已选择密文
]]

幻灯片：只是为了好玩，观察无线网络上发生的事情
解释tcpdump的作用

幻灯片：解释我们从中获得的内容：数据包头等

幻灯片：Gmail未加密

幻灯片：Facebook未加密；明文

为什么人们不在所有地方都使用SSL?
他们可能应该
一些技术原因：性能
带宽，加密消息的CPU消耗现在可以忽略不计
但SSL需要握手：2个额外的往返时间

--- 板 5 ---

安全通信渠道 =/= 安全
通信渠道的安全性基于许多信任假设
声明：你正在向/从特定实体发送/接收数据
实体是谁？ 一个CA验证为该实体的某人
展示亚马逊的证书

所有的CA是谁？ 在浏览器中显示所有CA的列表？
编辑 -> 首选项 -> 安全 -> 列出证书
这些CA来自哪里？ 它们随Mozilla或MS的浏览器一起提供
可能出现什么问题？
也许我对名称的信任是错误的（混淆的名称？amaz0n.com？）
也许CA未能正确验证名称？
也许其中一个CA已被攻击？
也许其中一个CA的密钥被泄露、盗窃或猜测？
你的笔记本电脑被攻击，有人添加了一个额外的受信任的CA根？
或者，只是修改了你的浏览器以执行任意操作
更高级的问题
凭证泄露
错误的访问控制

--- 第6个板块 ---

授权
授权的生命周期：
1. 授权
2. 调解
3. 撤销

撤销 --\
授权 -\ |
V V
客户端 -> 守卫 -> 服务器 -> 对象
守卫检查请求是否应该被允许

--- 第7个板块 ---

这个守卫是如何工作的？
逻辑上，是一个访问矩阵

	F1	F2	F3
A	R	R	RW
B	RW	RW	- -
C	--	RW	R

一个轴上的主体 -- 需要认证
另一个轴上的对象 -- 守卫需要理解发生了什么
条目是允许的操作

这个矩阵存储在哪里？[白板的右侧]
列表（Unix中的ACL，AFS）：将对象的列存储起来
认证：弄清楚用户是谁
授权：将用户添加到ACL
调解：查找用户在对象列中的条目
票证（Java指针，某些URL，某些cookie，Kerberos，证书）：用户存储行
必须确保用户不能篡改自己的行！
授权：超出范围 -- 无论用户如何获得票证（例如密码）

中介：在用户提供的行中查找当前对象

--- 第8个黑板 ---

优点：票据模型可以轻松委托权限
优点：将授权检查与身份验证检查分离
可以单独更改系统的两个部分
缺点：需要先获取票据
缺点：撤销困难
计划A：追踪每个用户的每个票据
计划B：作废所有未完成的票据，要求仍然有权限的用户获取新的票据

--- 第9个板块 ---

生成票据的好方法?
HMAC很容易
[资源/权限，生成号/超时？，...]，使用服务器密钥进行HMAC加密
生成号支持撤销
超时同样有助于控制票据分发[Kerberos]

让我们看看这在实践中是如何工作的
访问目录/mit/6.01的票据
过期日期为11/11/2009

票据 = { /mit/6.01，11/11/2009，MAC(/mit/6.01 + 11/11/2009) }
票据 = { /mit/6.011，1/11/2009，MAC(/mit/6.011 + 1/11/2009) }

重要：确保所有签名的消息都能明确地编排!

能力 = 命名 + 票据
将名称和票据结合在一起
不能谈论一个对象而不谈论对它的权利
这样就可以避免符号链接攻击

Unix中的文件描述符有点像这样

管理这个矩阵可能很困难（无论是行还是列）
常见的简化方法是角色或组

--- 第10个板块 ---

所有这些组件是如何组合在一起的?
例子：Web浏览器/服务器交互
安全通信通道：SSL
客户端（浏览器）验证服务器的名称
服务器验证客户端的证书（MIT个人证书）否则，服务器将与尚未验证的某个人建立连接

消息序列
服务器：请自我识别 \
客户端：用户名，密码 > SSL（通常）
服务器：这是一个Cookie（票据） /
客户端：请求1和Cookie（票据）
客户端：请求2和Cookie（票据）
...

幻灯片：明文Cookie：足以登录Google日历
幻灯片：明文Cookie 2：其他网站的列表

通常这些cookie在你更改密码后仍然有效!
因为它只是一个带有你的用户名的签名票据
好主意：在cookie/ticket中包含pw gen#

票据和ACL并不像听起来那么不同
票据通常基于底层ACL发放
ACL通常用于根据较低级别的票据决定较高级别的操作
例如，服务器证书=服务器主体的票据
但其他检查适用于服务器的名称
客户端证书/cookie =该用户客户端权限的票据
但其他检查适用于客户端的名称