分布式系统

余盛季

计算机学院

2023/8/27

基本信息

- 余盛季
  - 办公地点：主楼A2-408
  - 联系方式：飞书
- 成绩构成
  - 50% 平时成绩
    - 20% 课堂练习
    - 30% 大作业
  - 50% 期末考试（闭卷）

如何学好分布式系统

- 先行课
  - 计算机网络 / 操作系统 / C、Java语言
- 课堂形式
  - 理论探讨为主
  - 应用验证为辅
  - 认真做大作业
- 为什么要学这门课？
  - 经典的硬核课程：综合性强
  - 理解实际系统的关键：平衡
  - 启发研究工作

教材及主要参考书目

- **Distributed Systems – Concepts and Design**
  分布式系统概念与设计
  - 中文版，原书第5版
  - 机械工业出版社
  - 金蓓弘 曹冬磊 等译

- Distributed Systems Principles and Paradigms
  分布式系统原理与范型
  - 中文版，原书第2版
  - 清华大学出版社
  - 辛春生 陈宗斌 等译

目标

- 对分布式系统的
  - 设计
  - 分析
  - 实现
- 所涉及的关键问题进行探讨和研究
- 海外高校课程模式
  - Topic driven paper reading
  - MIT 6.824

主要内容

- 分布式系统的特征
- 系统模型
- 时间和全局状态
- 协调和协定
- 事务和并发控制
- 复制
- 分布式文件系统
  - 谷歌文件系统GFS
- P2P系统

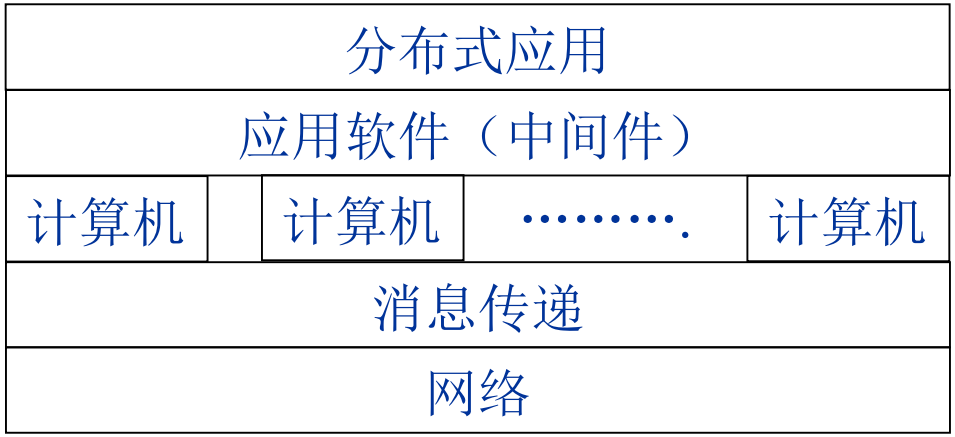
第1章 分布式系统的特征

- 引言
  - 分布式系统的定义、目标、特点
- 分布式系统举例
- 分布式系统趋势
- 挑战
- 总结

定义

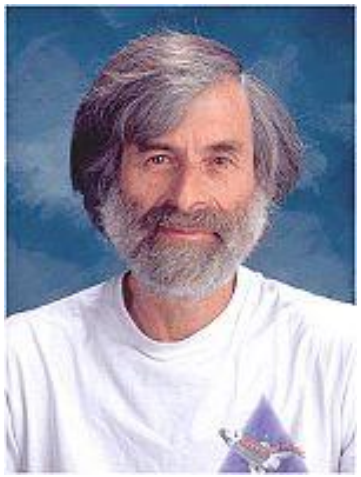
- 什么是分布式系统？

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.



定义

A distributed system is one on which I cannot get any work done because some machine I have never heard of has crashed. --- Leslie Lamport



A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system. --- Tanenbaum

分布式系统的目标

- 资源共享 (resource sharing)
  - 一些计算机通过网络连接起来，并在这个范围内有效地共享资源，协作实现共同目标
  - 硬件的共享，软件的共享，数据的共享，服务的共享
  - 媒体流的共享（动态的资源形式）
- 协同计算 (collaborative computing)
  - 并行计算，分布式计算

为什么需要分布式系统?

- Functional Separation（功能分离）
  - Existence of computers with different capabilities and purposes
    - Clients and Servers
    - Data collection and data processing
- Inherent distribution（固有的分布性）
  - Information: different information is created and maintained by different people (e.g., Web pages)
  - People: computer supported collaborative work (virtual teams, engineering, virtual surgery)
- Power imbalance and load variation（负载均衡）
  - Distribute computational load among different computers.
- Reliability（可靠性）
  - Long term preservation and data backup (replication) at different locations.
- Economies（经济性）:
  - Building a supercomputer out of a network of computers.

特点

- 分布式系统的三个基本特点：

- 并发性 (concurrency)
  - 多个程序（进程，线程）并发执行，共享资源
- 无全局时钟 (global clock)
  - 每个机器有各自的时间，难以精确同步，程序间的协调靠交换消息
- 故障独立性 (independent failure)
  - 一些进程出现故障，并不能保证其它进程都能知道

第1章 分布式系统的特征

- 引言
- 分布式系统举例
- 分布式系统趋势
- 挑战
- 总结

分布式系统举例

- 1、WEB搜索
- Google：最大最复杂的分布式系统之一
  - 底层物理设施（异地，失效是正常）
  - 分布式文件系统
    - GFS
  - 分布式存储系统
    - F1, Spanner, Bigtable
  - 锁服务
    - Chubby
  - 编程模式
    - MapReduce

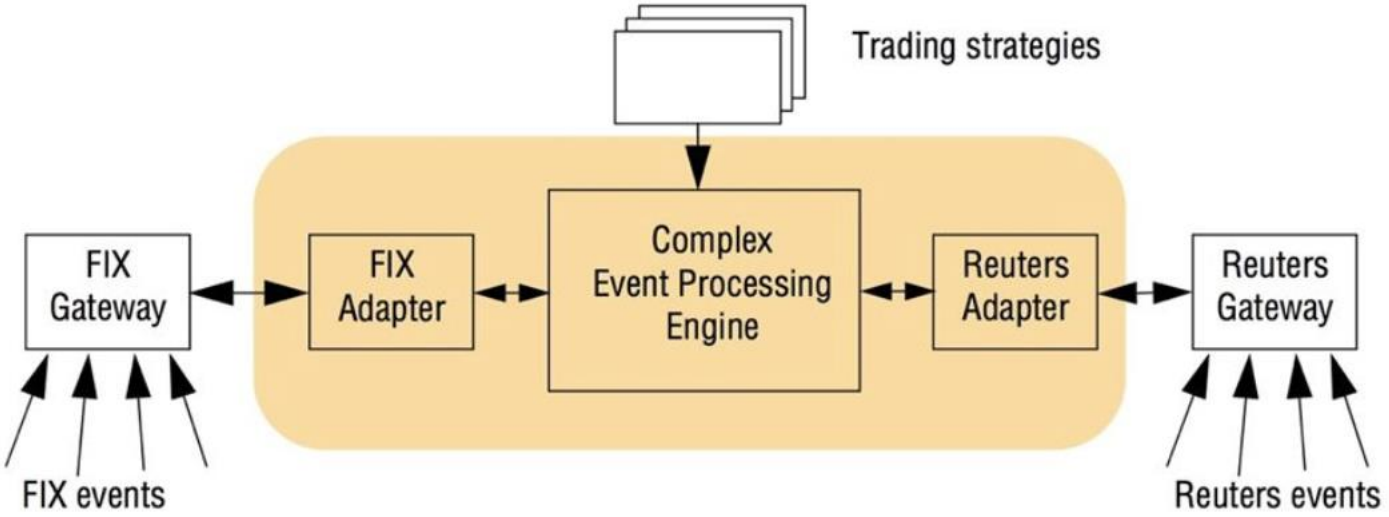
分布式系统举例

- 2、大型多人在线游戏
  - Massively Multiplayer Online (MMO) / Role-Playing (RPG)
  - 王者荣耀 / 魔兽

  - 快速响应的实时要求
  - 对共享世界的一致视图

分布式系统举例

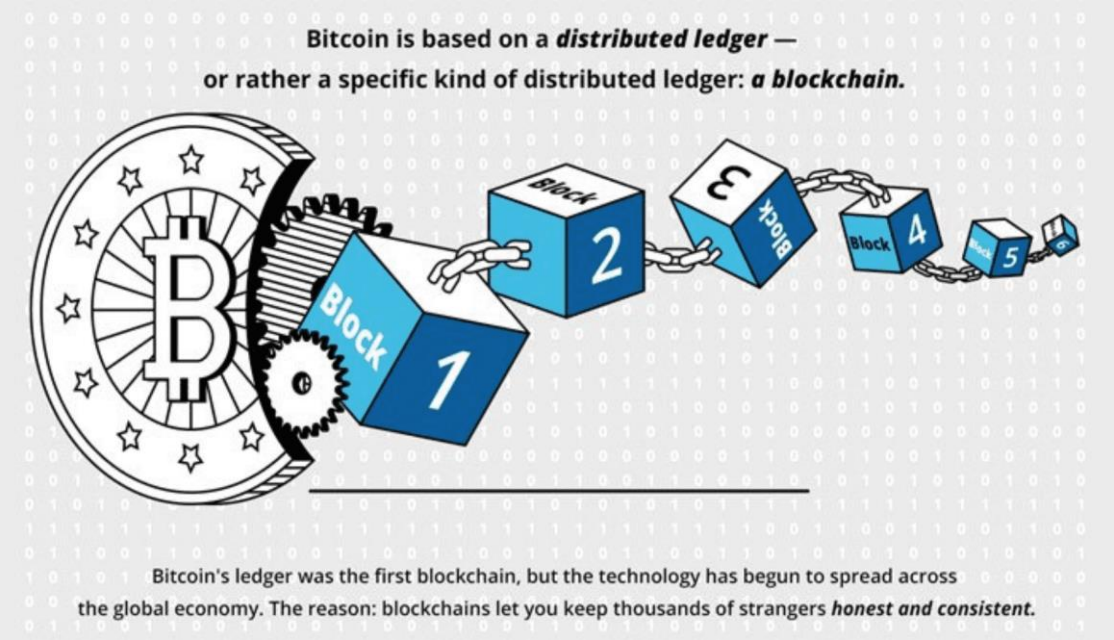
- 3、金融交易
  - 重点是对事件的通信与处理，这样的系统通常采用分布式基于事件的系统



Financial Information eXchange

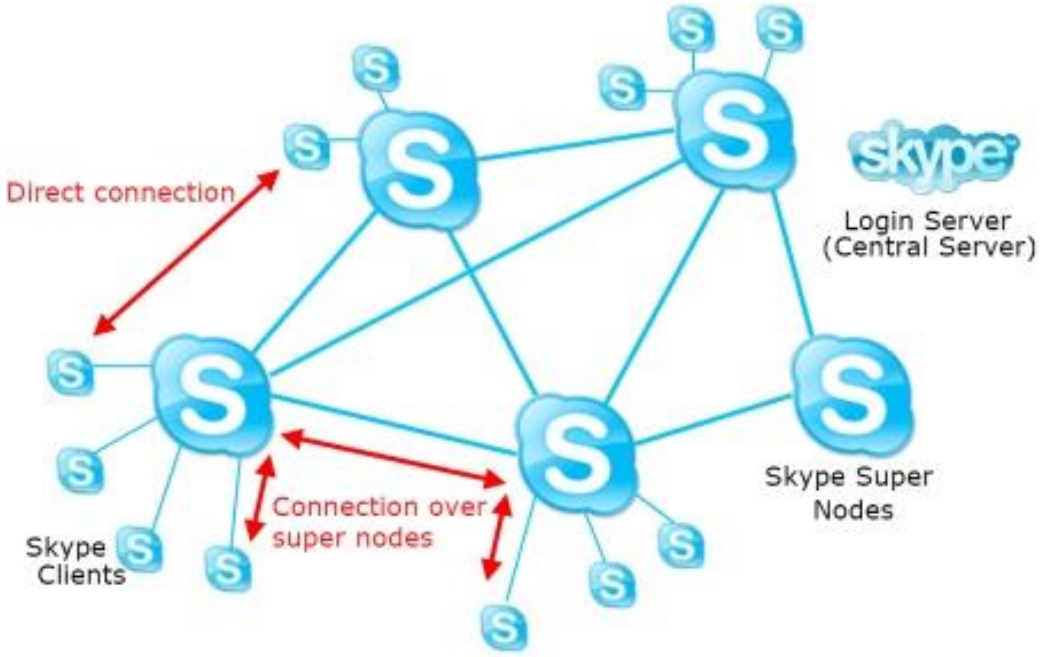
分布式系统举例

- 4、区块链系统
  - 分布式账本、不可篡改
  - 共识、容错 vs. 性能



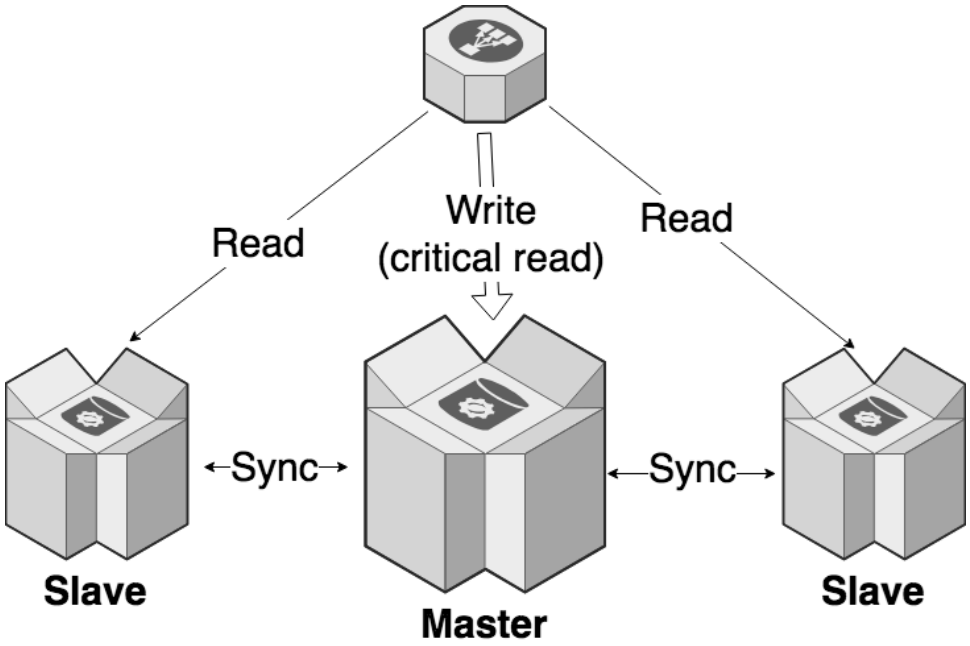
分布式系统距离

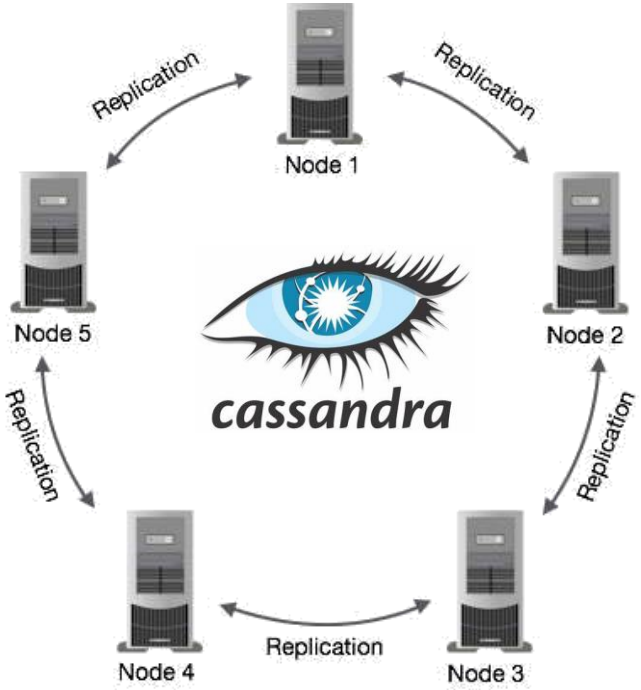
- 5、语音系统
  - 集中 vs. 对等
  - 中转 vs. 直连
  - 随时上下线



分布式系统举例

- 6、数据库系统
  - Scale-up vs. scale-out





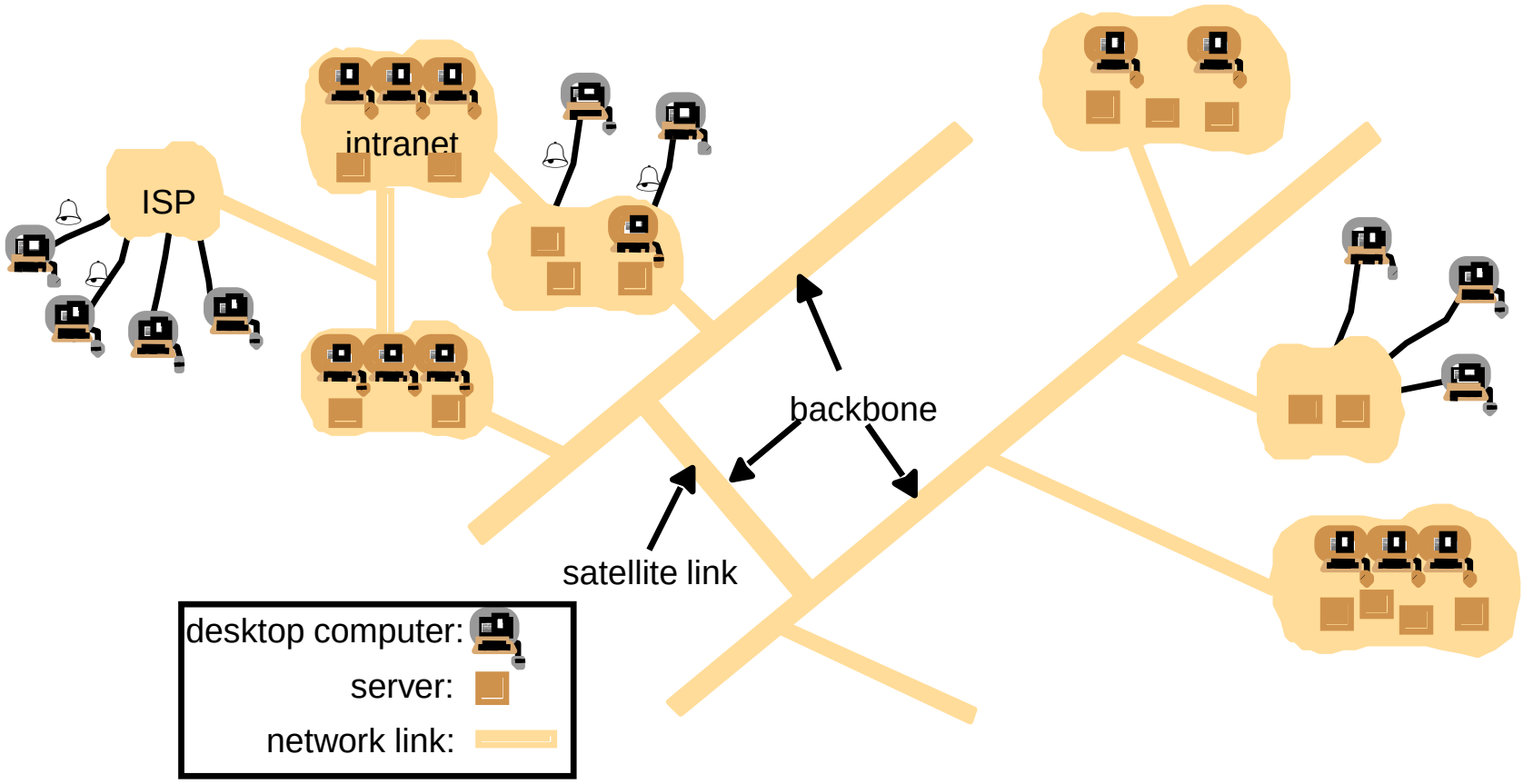
第1章 分布式系统的特征

- 引言
- 分布式系统举例
- 分布式系统趋势
- 挑战
- 总结

分布式系统趋势

- 泛在联网和现代互联网
- 移动和无处不在的计算
- 分布式多媒体系统
- 将分布式计算作为公共设施

泛在联网和现代互联网


intranet
ISP
backbone
satellite link
desktop computer:
server:
network link:

泛在联网和现代互联网

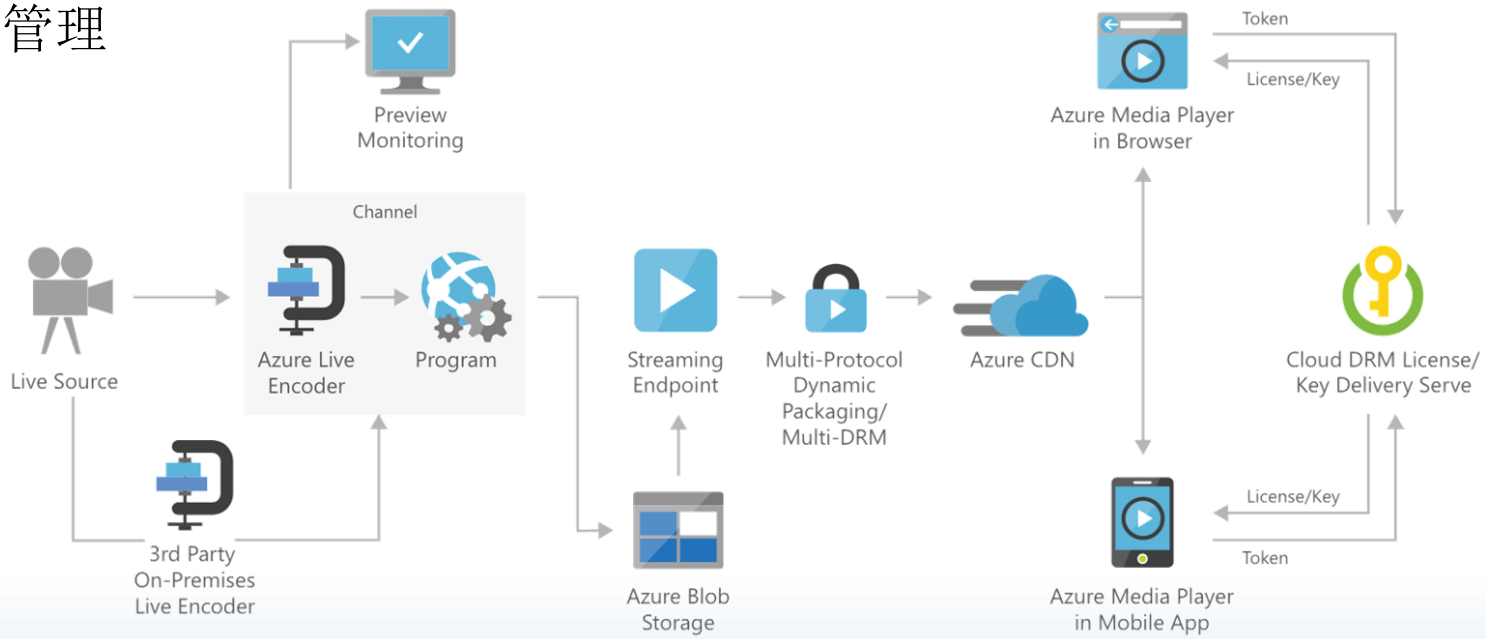
- Internet
- 难点：
  - 可扩展性(DNS, IP)
  - 资源的定位
  - 异构
- 成就：
  - TCP/IP协议是因特网最重要的技术成果

移动计算/物联网

- 笔记本电脑
- 手持设备
  - PDA，手机，摄像机，数码照相机
- 可穿戴设备
  - 智能手表
  - 数字眼镜
- 家电设备、传感器
  - IoT: internet of things

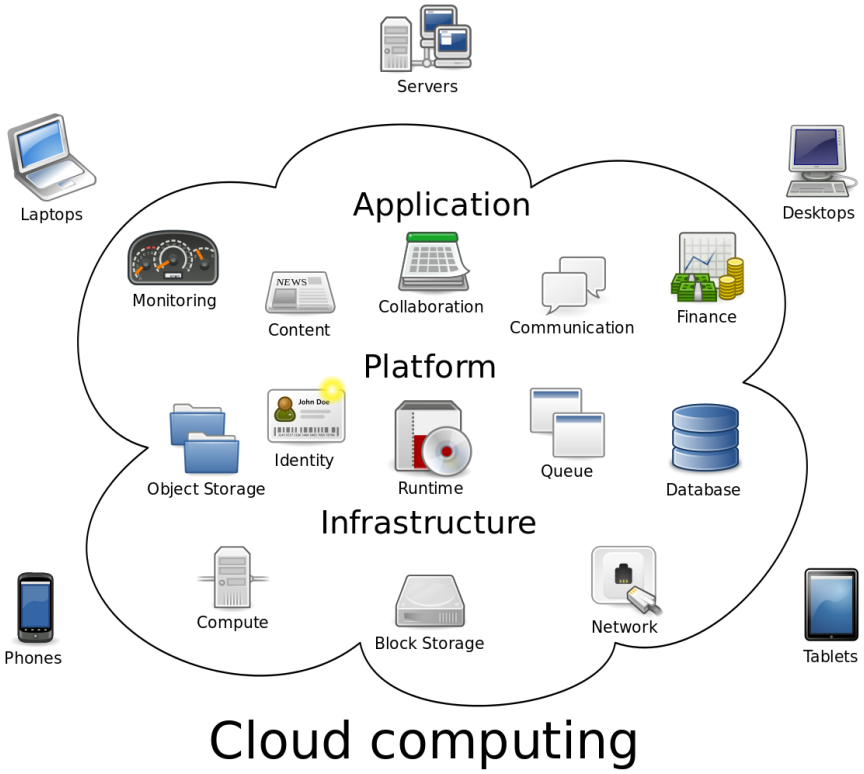
分布式多媒体系统——流媒体

- 分布式多媒体系统应该能够对连续类型媒体（如音频和视频）完成存储和定位以及网络传输功能，同时具备在一组用户中共享多种类型媒体的能力。
- 分布式多媒体应用面临的问题
  - 对一系列编码和加密方式的支持
  - 资源管理
  - QoS



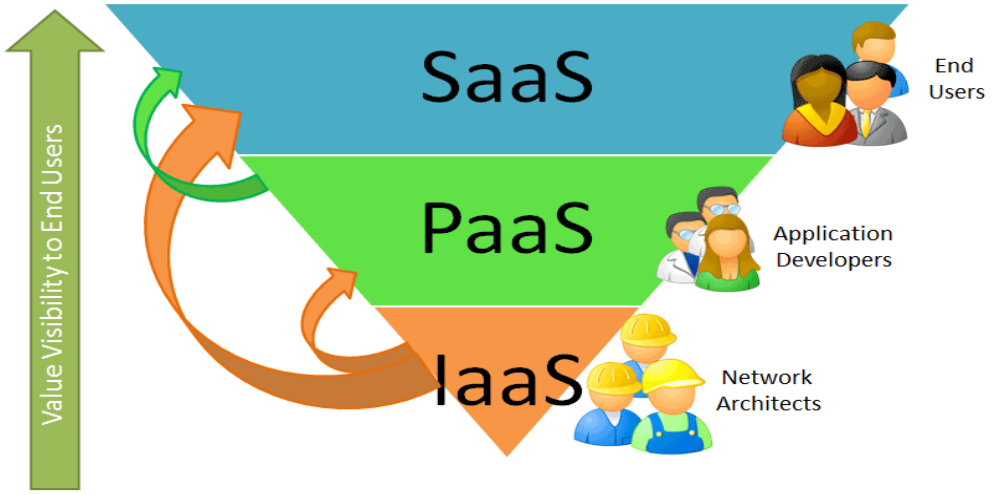
把分布式计算作为一个公共设施

- 云计算
  - 一种基于互联网的计算方式，通过这种方式，共享的软硬件资源和信息可以按需提供给计算机和其他设备：按需使用



把分布式计算作为一个公共设施

- 云计算



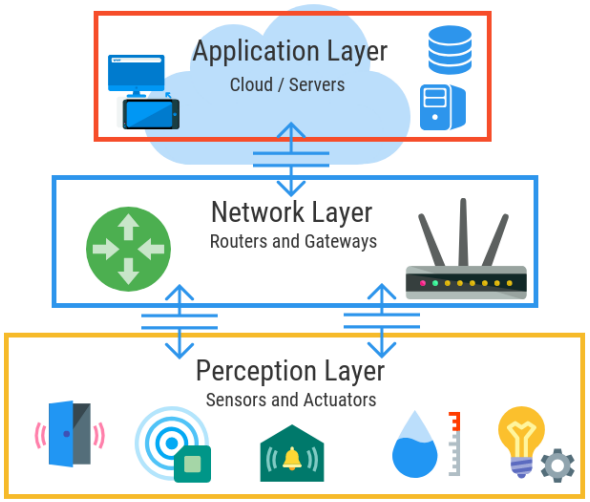
服务多租赁化

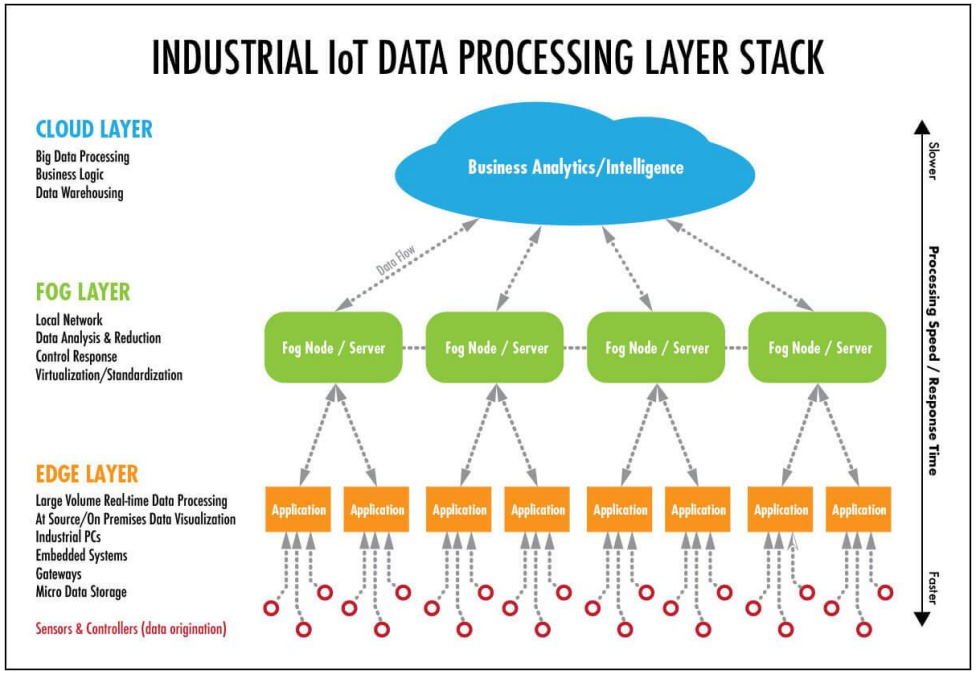
平台可伸缩化

资源虚拟化

将分布式计算作为公共设施

- 物联网：把传感器装备到电网、铁路、桥梁、隧道、公路、建筑、供水系统、大坝、油气管道以及家用电器等各种真实物体上，通过互联网联接起来，进而运行特定的程序，达到远程控制或者实现物与物的直接通信。
- 边缘计算 Edge Computing
- 雾计算 Frog Computing





第1章 分布式系统的特征

- 引言
- 分布式系统举例
- 分布式系统趋势
- 挑战
- 总结

假设

- 如果网络能保证信息不丢失
- 如果所有消息都能在预期的时间内到达
- 如果一系列操作都能成功
- 如果每个机器上的时钟都是精准的
- 如果机器不宕机，进程不出故障
- 如果消息都是可信的

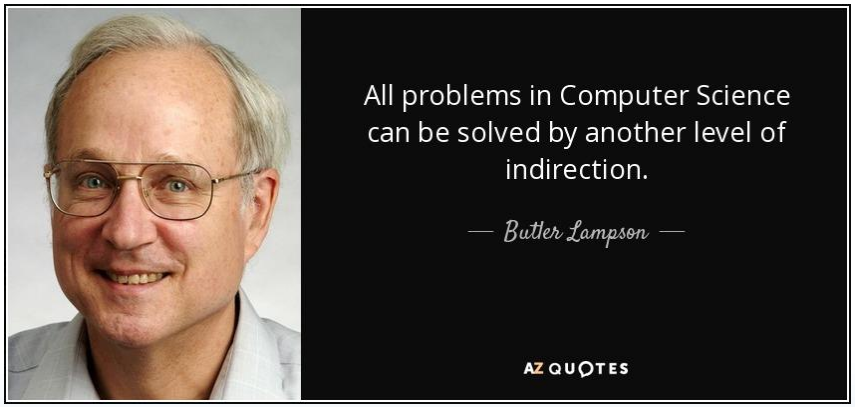
- 没有实践经验的开发者，通常会在上述假设下，理想化系统的实现：只注重应用需求，而忽略实际中会出现的问题。
- **这样的系统，在实际中根本不能用**

挑战——异构性(Heterogeneity)

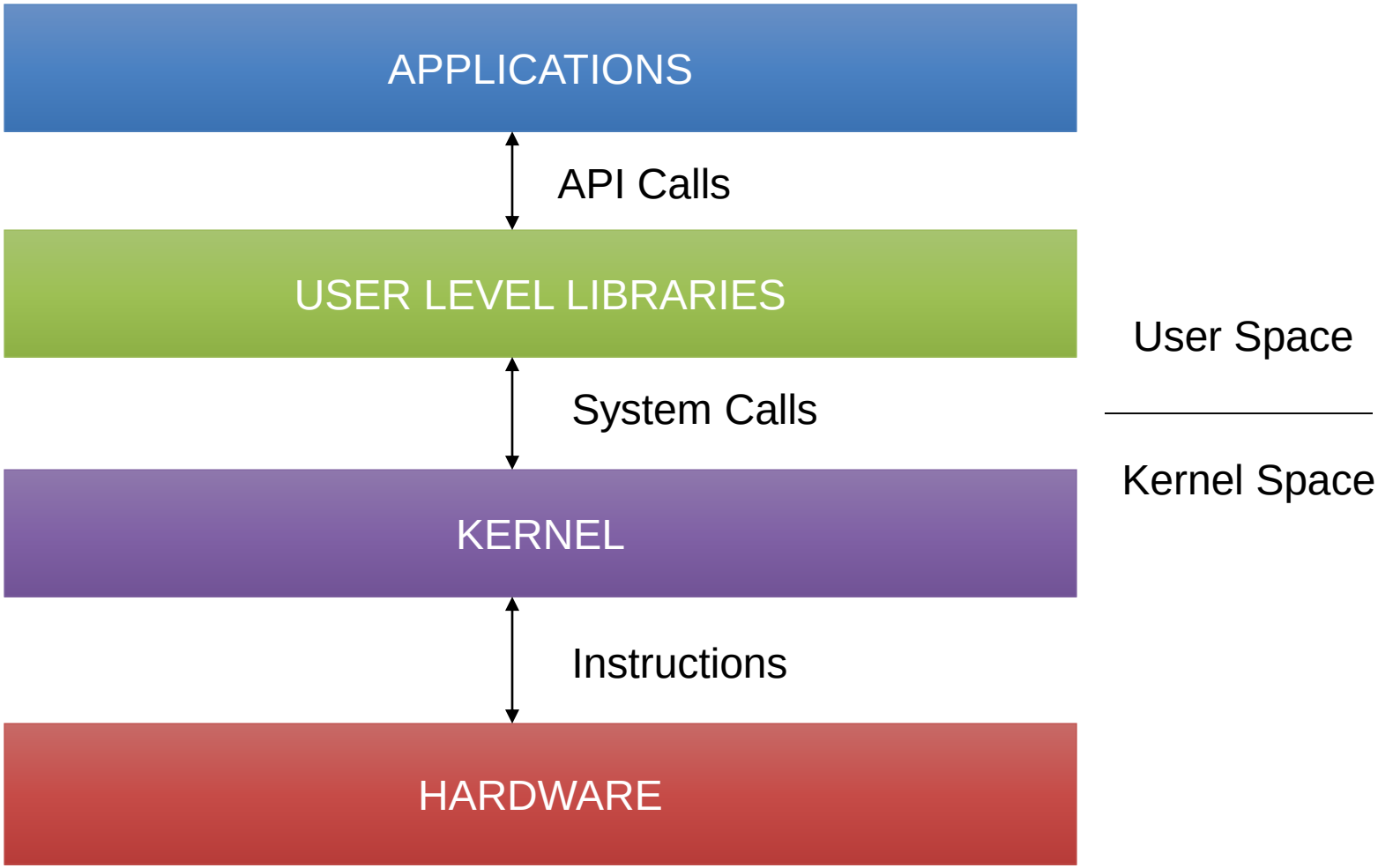
- 网络协议
  - Ethernet, token ring, etc.
- 硬件
  - big endian / little endian, ISA
- 操作系统
  - different APIs of Unix and Windows for the same features
- 编程语言
  - different representations for data structures
- 开发者实现方式的不同
  - no application standards

挑战——异构性(Heterogeneity)

- 中间件 (Middleware)
  - 应用到软件层，用来屏蔽底层的异构性。
    - 如Java RMI，提供远程调用接口，可在任何操作系统上运行。
    - DB proxy屏蔽分库分表差异
- 移动代码 (Mobile code)
  - 在不同的机器间移动并执行，须解决异构问题。
  - 虚拟机运行在不同的机器或系统上，代码在虚拟机上运行
    - 虚拟机类型



Virtualization Levels

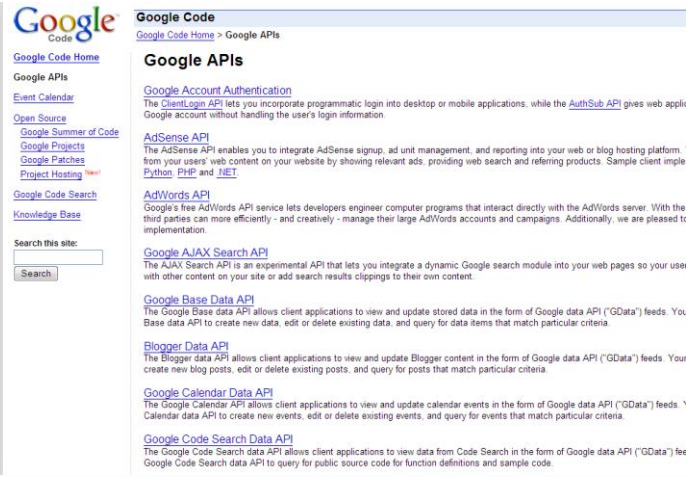


Categories

- Instruction Set Architecture Level
  - Bochs/Crusoe/Qemu/BIRD/Dynamo → Binary translation
- Hardware Abstraction Layer (HAL) Level
  - VMWare/Virtualbox/Denali/Xen/L4/Plex86/User-mode Linux/Cooperative Linux
- Operating System Level
  - Jail(chroot)/Virtual Environment/Ensim's VPS/FVM
- Library (user-level API) Level
  - Wine/WABI/LxRun/Visual MainWin
- Application (Programming Language) Level
  - JVM/.NET CLI/Parrot/Lua

挑战—开放性（openness)

- 一个系统是否可以扩充或以不同的方式重新实现
- 分布式系统的开放性
  - 在多大程度上新的资源共享服务可以加到系统中来
- 关键：公开接口 (API)
  - 传输协议
  - 序列化
  - 抽象方式
    - RPC
    - Restful



挑战—安全性（security）

- 机密性 (Confidentiality)
  - 防止未经授权的个人访问资源
  - e.g. ACL in Unix File System, encryption
- 完整性 (Integrity)
  - 防止数据被篡改和破坏
  - e.g. checksum, signature
- 可用性 (Availability)
  - 防止对所提供服务的干扰
  - e.g. Denial of service

挑战—可伸缩性（Scalability）

- 系统规模扩展后，无论是资源还是用户，系统的性能保持在一定的水平
- 设计挑战
  - 控制物理资源的代价
    - e.g.,随着用户数的增长，服务器的增长代价不能超过 O(n)
  - 控制性能损失
    - e.g., DNS no worse than O(logn)
  - 控制软件资源被耗尽
    - e.g., IP address
  - 防止性能瓶颈
    - e.g., partitioning name table of DNS, cache and replication

挑战—故障处理

- 检测故障
  - e.g. 用校验和检测数据
  - 分布式系统中确切地知道远程服务器是否出现故障很难做到
- 屏蔽故障
  - e.g. 重发没有收到的消息, 备份服务器等
- 故障容错
  - e.g. 无法做到屏蔽故障，至少让用户知道出现了问题，让用户自由选择是否继续请求服务。
- 故障恢复
  - e.g. 操作日志，恢复。
- 冗余策略
  - e.g. IP route, replicated name table of DNS

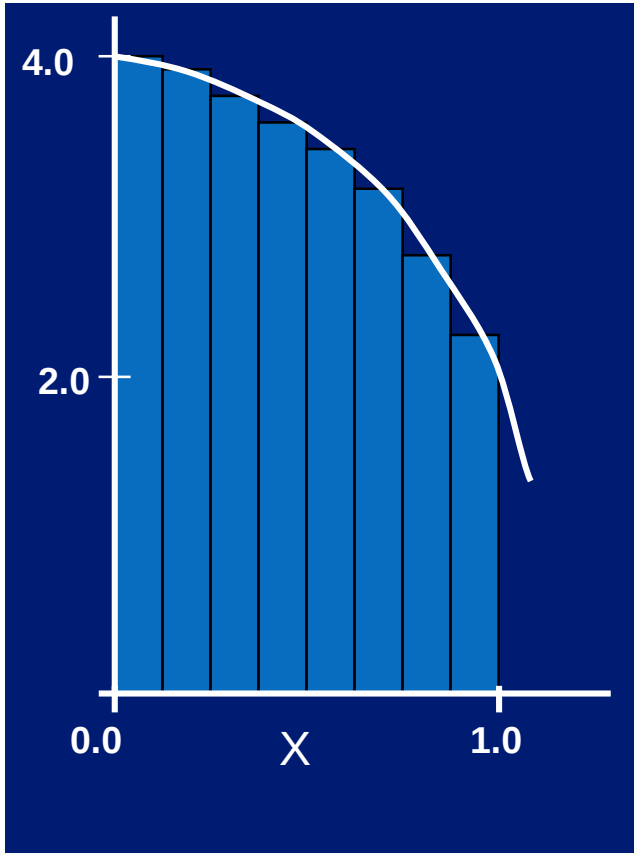
挑战—并发（Concurrency）

- 正确性
  - 多个进程并发访问共享资源，要保证被访问数据的正确性，不能出现不一致
  - DB transaction
- 性能 (Performance)
  - 多个并发操作保证性能
  - HPC 典型案例

MPI: PI

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

```
#include <stdio.h>
#include "mpi.h”

static long num_steps = 100000;
double step;

void main (int argc, char** argv) {
 int i, id, num_procs;
 double x, pi, sum = 0.0;

 MPI_Init(&argc, &argv);

 step = 1.0/(double) num_steps;

 MPI_Comm_rank(MPI_COMM_WORLD, &id);
 MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
```



MPI: PI

```
for (i=(id+1);i<= num_steps; i+=num_procs){
   x = (i-0.5)*step;
   sum = sum + 4.0/(1.0+x*x);
 }

 MPI_Reduce(&sum, &pi, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
 if (id == 0){
   pi *= step;
   printf("pi is %f \n",pi);
 }
}
```

挑战—透明性（Transparency）

- 访问透明（Access transparency）
  - 使用同样的操作去访问本地资源和远程资源。
  - E.g. NFS / Windows File Sharing
- 位置透明（Location transparency）
  - 访问资源的时候，不需要知道资源的位置。
  - E.g. URL
- 并发透明（Concurrency transparency）
  - 几个进程同时访问资源，互不干扰
  - E.g. DB

挑战—透明性（Transparency）

- 复制透明（Replication transparency）
  - 使用多个资源的副本来提高可靠性和性能，用户或者应用程序开发者并不需要了解副本技术。
  - E.g. Keepalive (VRRP)
- 故障透明（Failure transparency）
  - 在存在故障的情况下，用户和应用仍可完成他们的任务
  - E.g., email

挑战—透明性（Transparency）

- 移动透明（Mobility transparency）
  - 资源或者客户端的移动不影响用户及程序的操作。
  - E.g. mobile phone
- 性能透明（Performance transparency）
  - 允许系统重新配置改善性能，例如改变负载。
  - E.g. auto-scaling
- 扩展透明（Scaling transparency）
  - 允许系统和应用扩大规模无需改变系统的结构和用算法。
  - Stateless design
  - E.g., auto scaling

第1章 分布式系统的特征

- 引言
- 分布式系统举例
- 挑战
- 总结

总结

- 分布式系统无处不在（pervasive，ubiquitous）
- 构造分布式系统的主要动机是资源共享和协同计算
- 分布式系统的特点
  - 并发性
  - 没有全局时钟
  - 故障独立性

总结

- 构造分布式系统面临的挑战
  - 异构性（Heterogeneity）
  - 开放性（Openness）
  - 安全性（Security）
  - 可伸缩性（Scalability）
  - 故障处理（Failure handling）
  - 并发行（Concurrency）
  - 透明性（Transparency）

查阅并了解以下概念

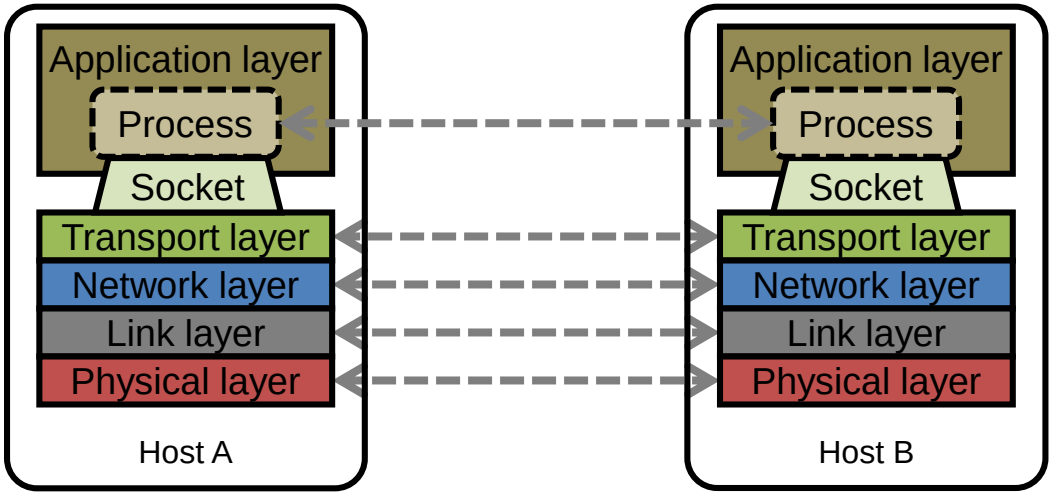
- Time synchronization
- Leader election
- Mutual exclusion
- Distributed snapshot
- Routing
- Consensus
- Replica management
- Transactions
- Trust model

RPC

Remote Procedure Call

Socket-based communication

- Socket: The interface the OS provides to the network
  - Provides inter-process explicit message exchange
- Can build distributed systems atop sockets: send(), recv()
  - e.g.: put(key,value) → message

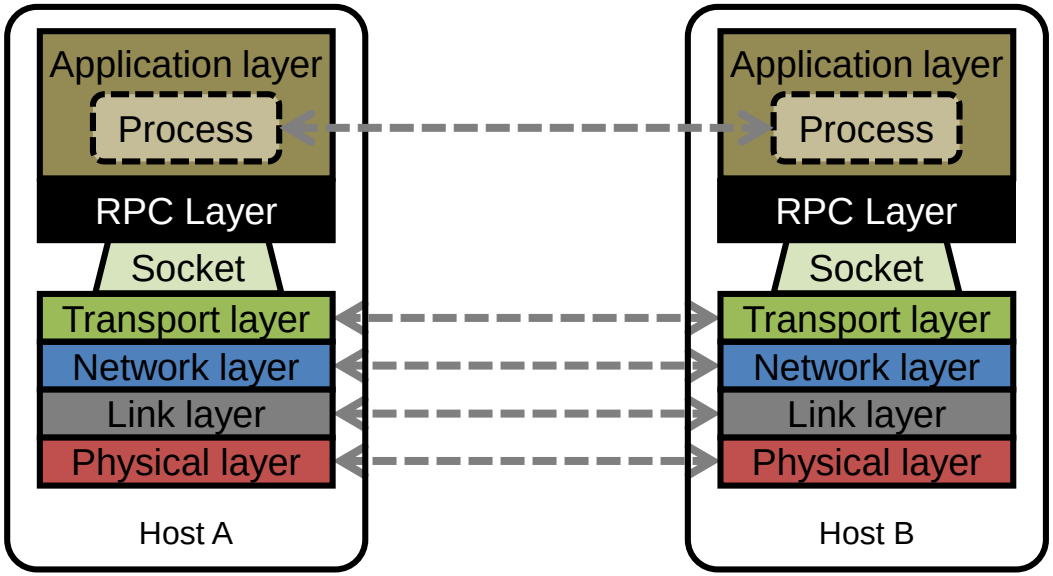


```
// Create a socket for the client
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror(”Socket creation");
  exit(2);
}

// Set server address and port
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = inet_addr(argv[1]);
servaddr.sin_port = htons(SERV_PORT); // to big-endian

// Establish TCP connection
if (connect(sockfd, (struct sockaddr *) &servaddr,
            sizeof(servaddr)) < 0) {
  perror(”Connect to server");
  exit(3);
}

// Transmit the data over the TCP connection
send(sockfd, buf, strlen(buf), 0);
```

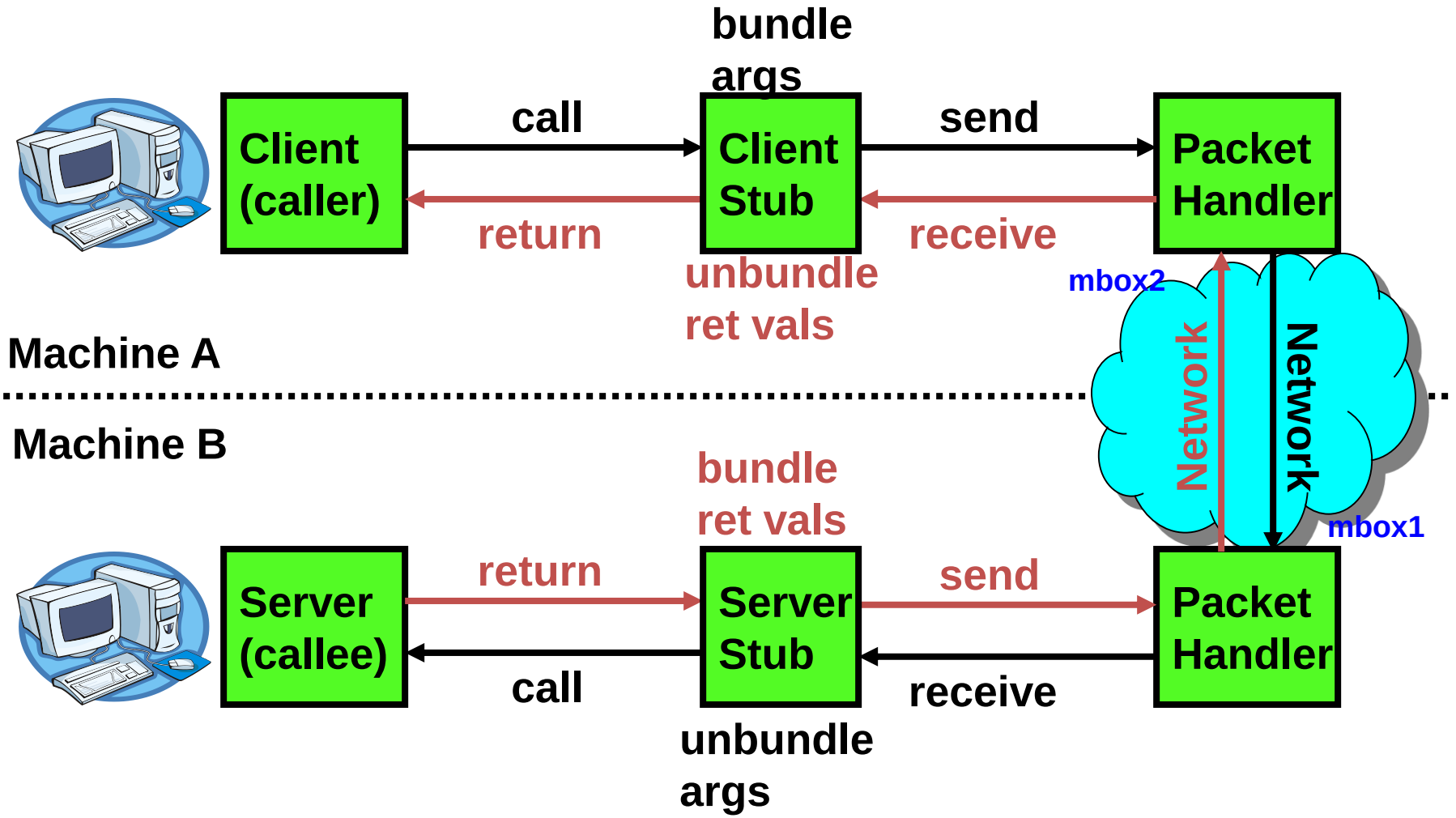
Socket: still not great

- Lots for the programmer to deal with every time
  - How to separate different requests on the same connection?
  - How to write bytes to the network / read bytes from the network?
  - What if Host A's process is written in Go and Host B's process is in C++?
- Still pretty painful… have to worry a lot about the network

Solution: Another layer!



RPC Information Flow



RPC Details

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform issues

Differences in data representation

- Not an issue for local procedure calls (LPC)
- For a remote procedure call, a remote machine may:
  - Run process written in a different language
  - Represent data types using different sizes
  - Use a different byte ordering (endianness)
  - Represent floating point numbers differently
  - Have different data alignment requirements
    - e.g., 4-byte type begins only on 4-byte memory boundary
  - Data in running programs NOT just primitives, but arrays, pointers, lists, trees, graphs etc.
    - Data being transmitted: Sequential!
    - Pointers make no sense.
    - Structures must be flattened.

External Data Representation

- An agreed standard for the representation of data structures and primitive values.
  - Internal to external: 'marshalling' (serialization)
  - External to internal: 'unmarshalling' (unserialization)
- Examples
  - Sun XDR
  - CORBA's Common Data Representation (CDR)
  - Java Object Serialization
  - XML
  - JSON
  - ...

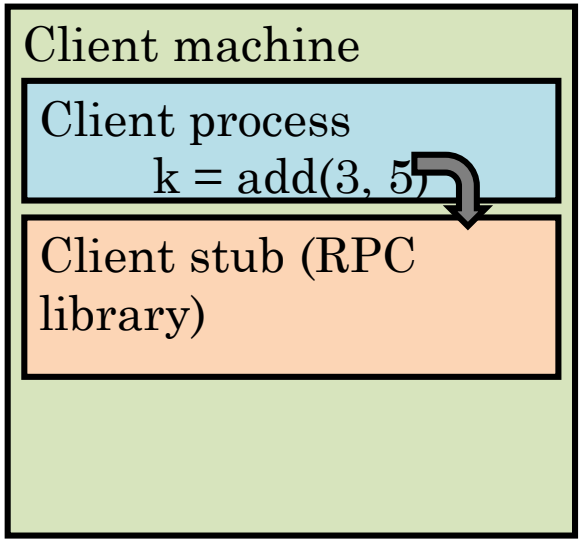
```
syntax = "proto2";

message Sensor {
  required string name = 1;
  required double temperature = 2;
  required int32 humidity = 3;
  enum SwitchLevel {
    CLOSED = 0;
    OPEN = 1;
  }

   required SwitchLevel door = 5;
 }
```

```
message Person {
  required string name = 1;
  required int32 age = 2;
  optional string email = 3;
}
```

```
#include “sensor.pb.h”

int main() {
  Sensor sensor;
  sensor.set_name("Laboratory");
  sensor.set_temperature(23.4);
  sensor.set_humidity(68);
  sensor.set_door(Sensor_SwitchLevel_OPEN);
}
```

A day in the life of an RPC

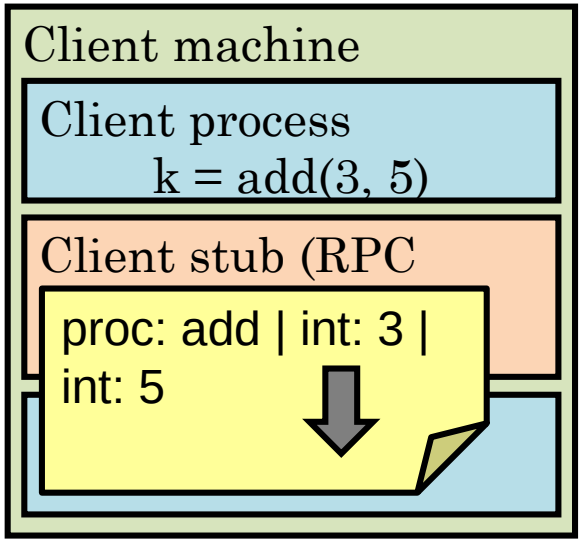
- Client calls stub function (pushes parameters onto stack)



A day in the life of an RPC

- Client calls stub function (pushes parameters onto stack)

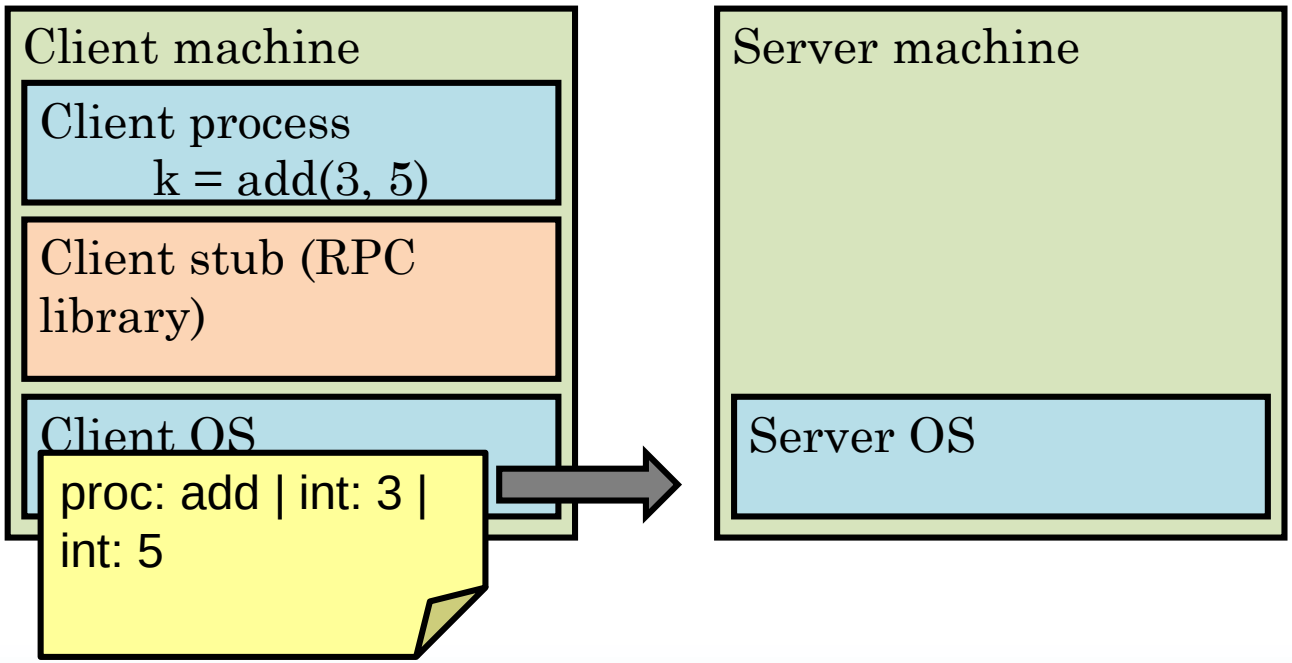
- Stub marshals parameters to a network message



A day in the life of an RPC

- Stub marshals parameters to a network message

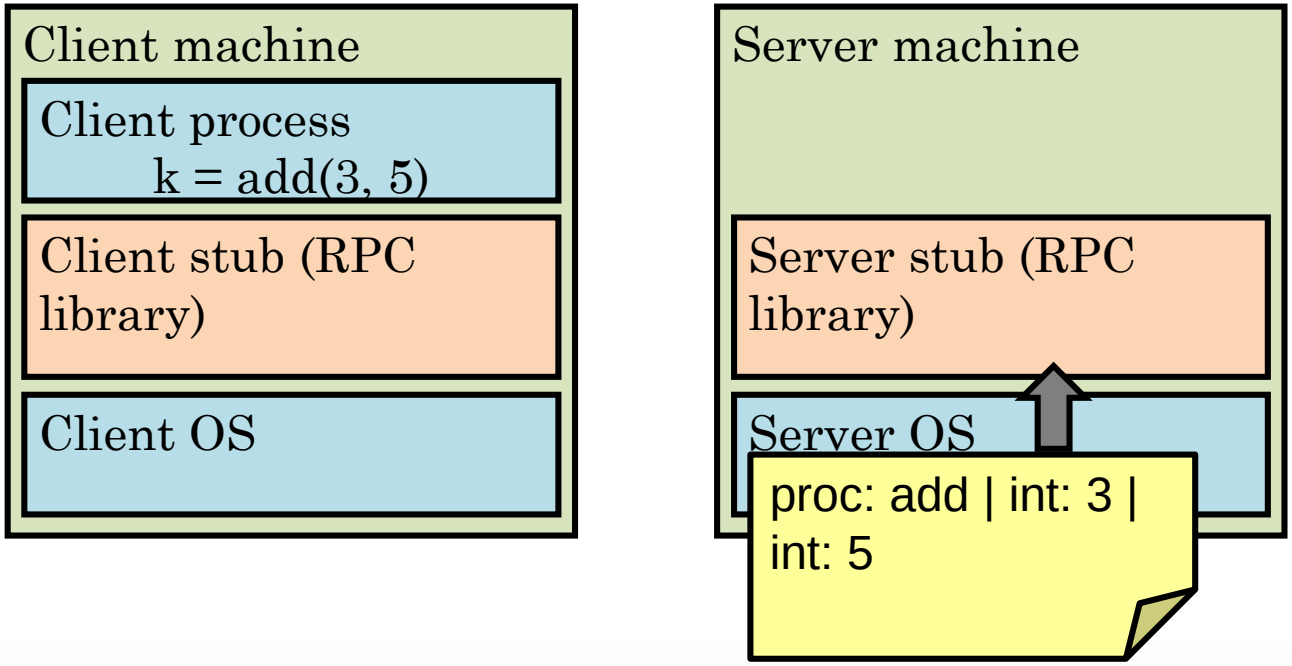
- OS sends a network message to the server



A day in the life of an RPC

- OS sends a network message to the server

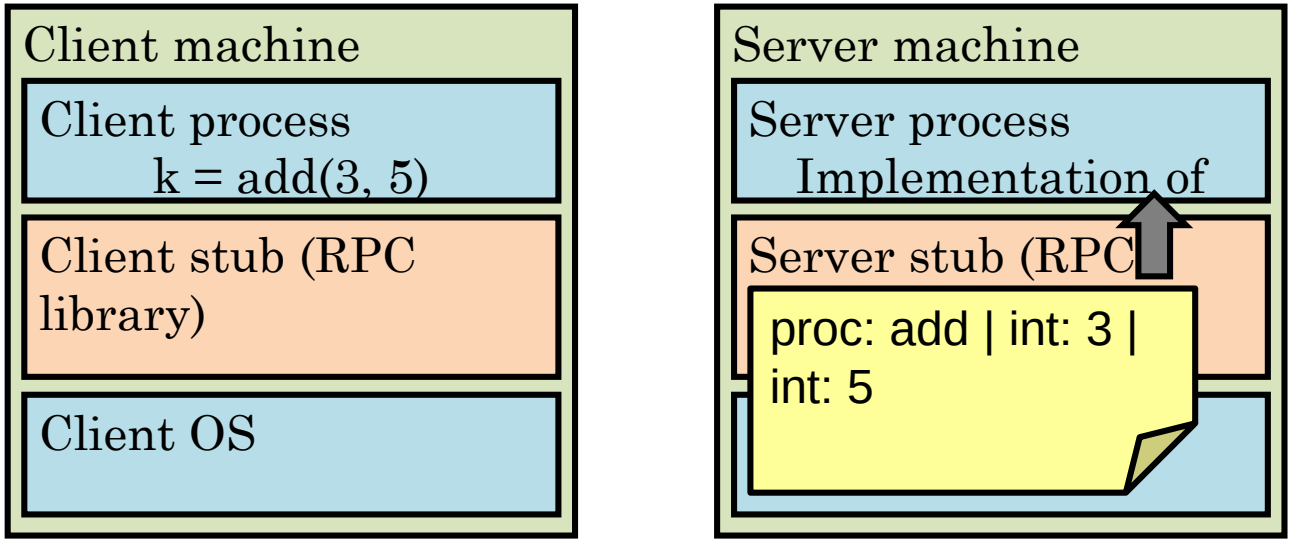
- Server OS receives message, sends it up to stub



A day in the life of an RPC

- Server OS receives message, sends it up to stub

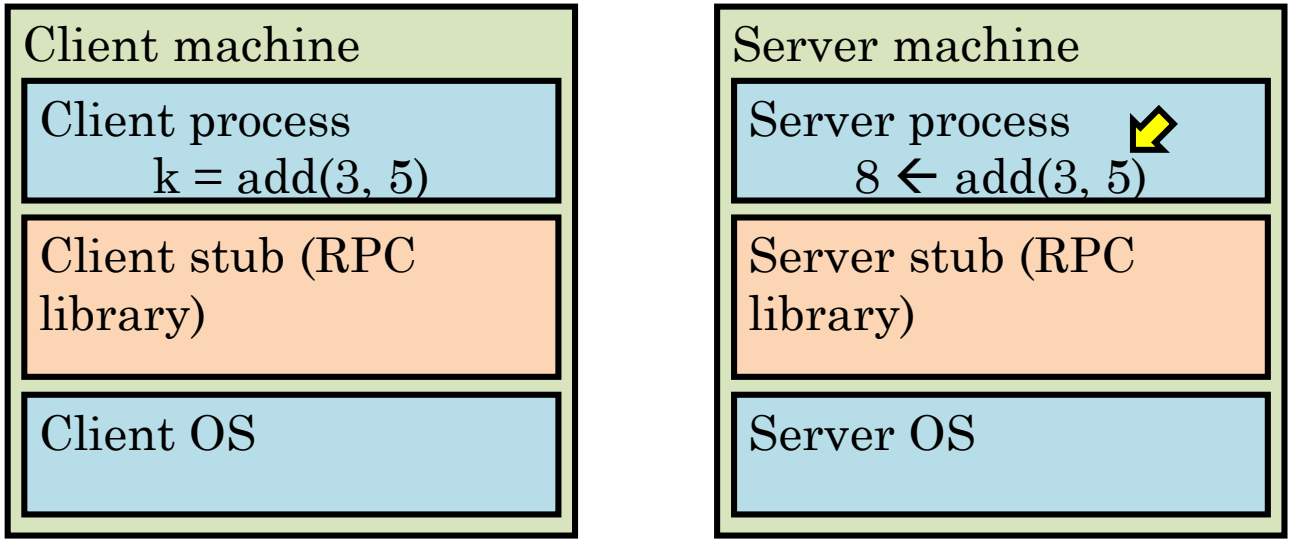
- Server stub unmarshals params, calls server function



A day in the life of an RPC

- Server stub unmarshals params, calls server function

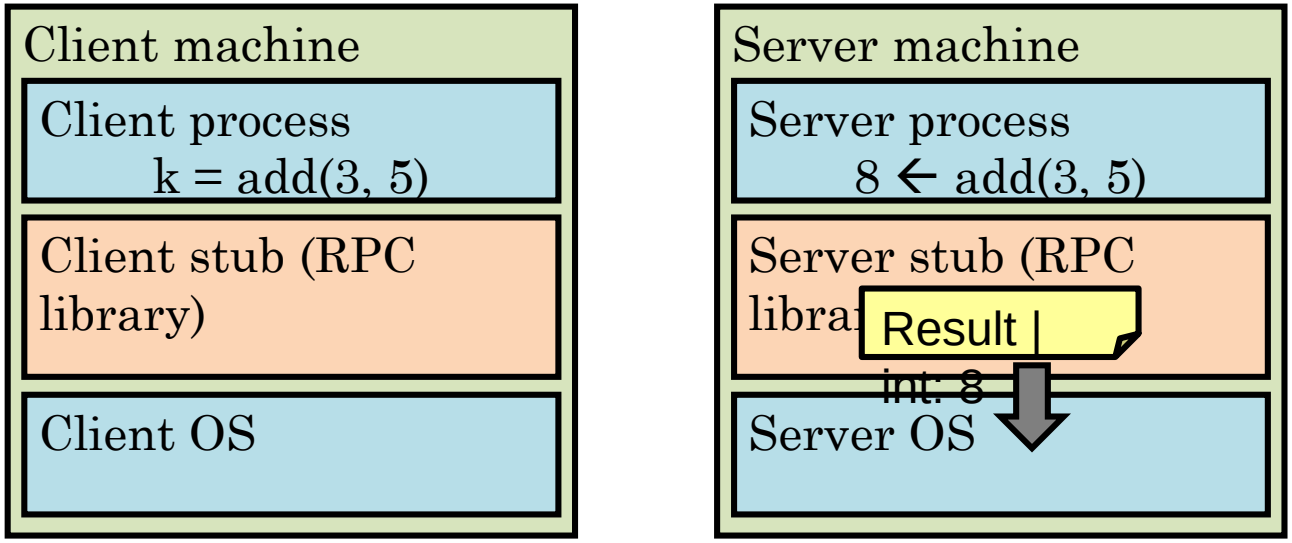
- Server function runs, returns a value



A day in the life of an RPC

- Server function runs, returns a value

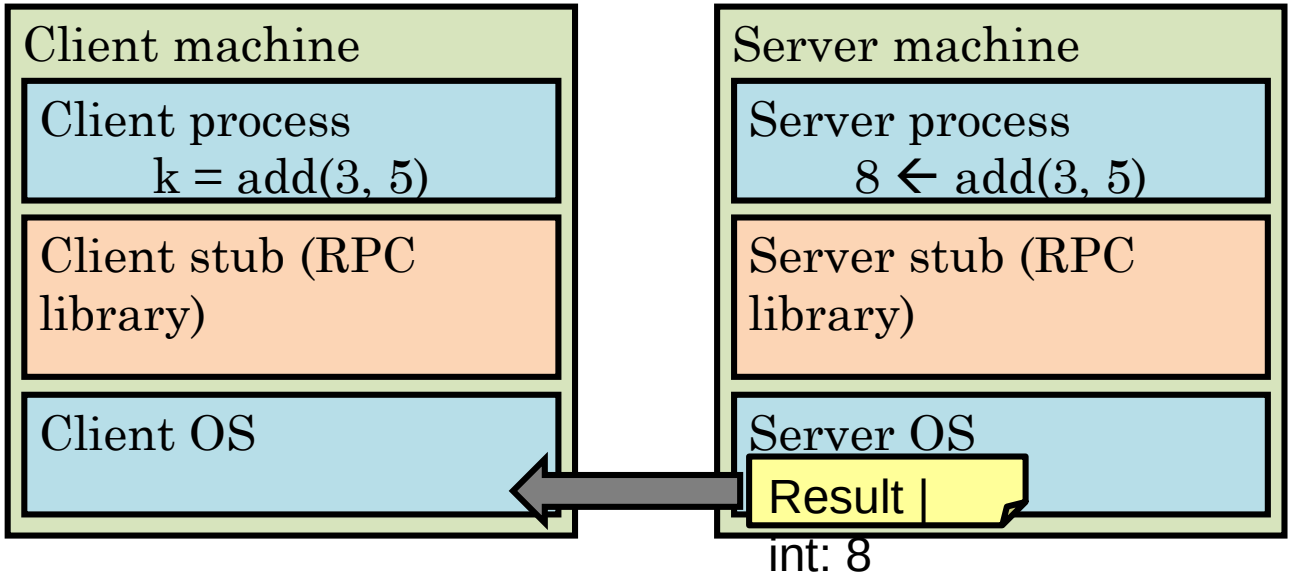
- Server stub marshals the return value, sends message



A day in the life of an RPC

- Server stub marshals the return value, sends message

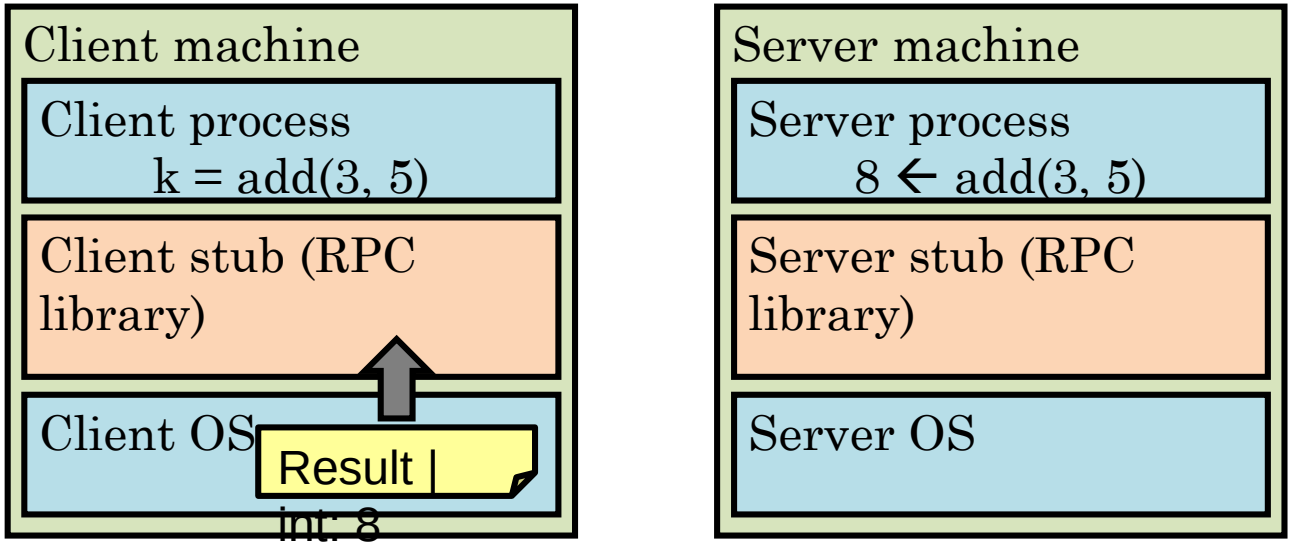
- Server OS sends the reply back across the network



A day in the life of an RPC

- Server OS sends the reply back across the network

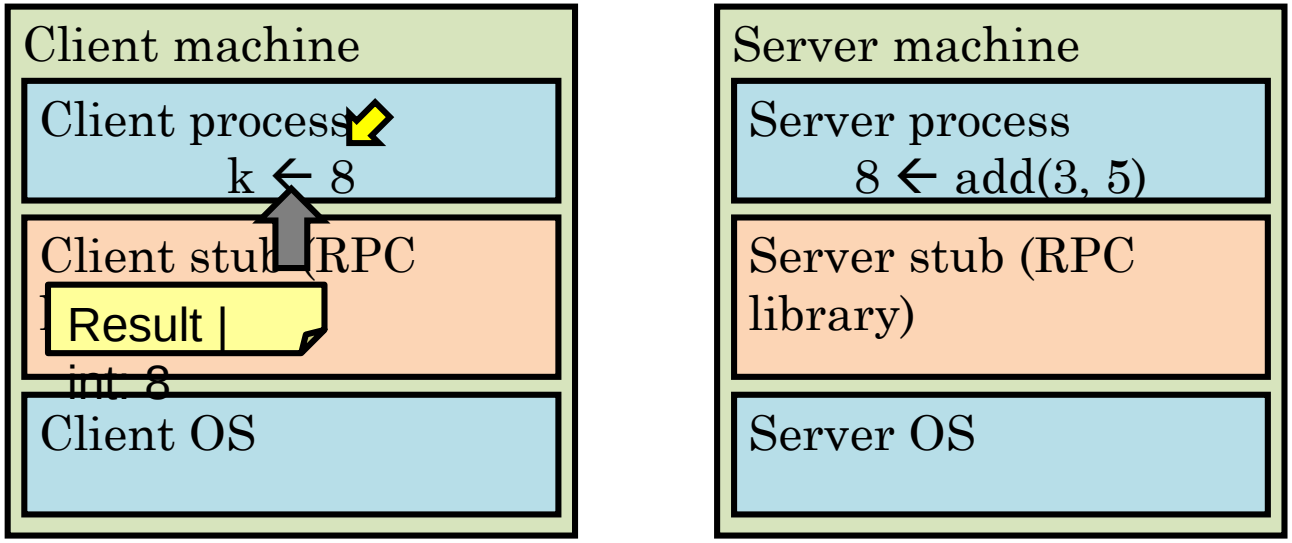
- Client OS receives the reply and passes up to stub



A day in the life of an RPC

- Client OS receives the reply and passes up to stub

- Client stub unmarshals return value, returns to client



Thanks