大纲

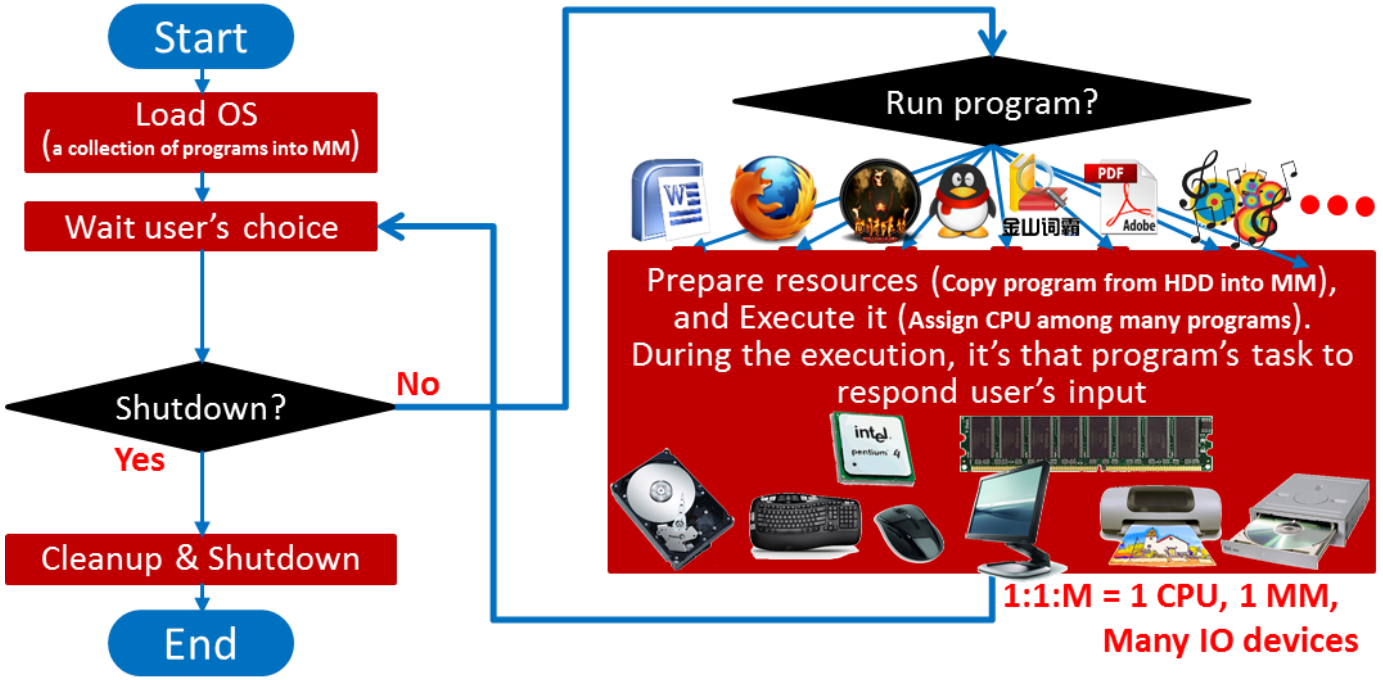
- **前言**
  - 为什么需要“大规模计算” [HPC, DL, Business platform system, Cloud已经合流]
    - 导入 – 科学计算(天气预报)，DL, 互联网平台(Google, Amazon, Alibaba, MeiTuan, …)
- **基础篇**
  - 并发程序的样子 – Divide & Conquer, Model & Challenges, PCAM, Data/Task, …
    - 天气预报的计算
  - 运行环境
    - 硬件 – 自己梳理的3个方案 – Shared/Unshared Memory, Hybrid
    - 系统软件 – 协议栈, Modern OS, Distributed Job Scheduler, GTM等
- **算法级篇**
  - OpenMP, MPI, CUDA (DL的实现), Big Data 中的MR/Spark等 (只涉及在Big Data SDK之上的编程；大数据本身的介绍放到后一部分)
- **系统级篇 – 互联网平台的实现**
  - “秒杀”的技术架构
  - 计算广告
  - 系统架构 (HTAP等)
    - Flink, ClickHouse, MaxCompute, ELK …

Chapter 5: Distributed OS

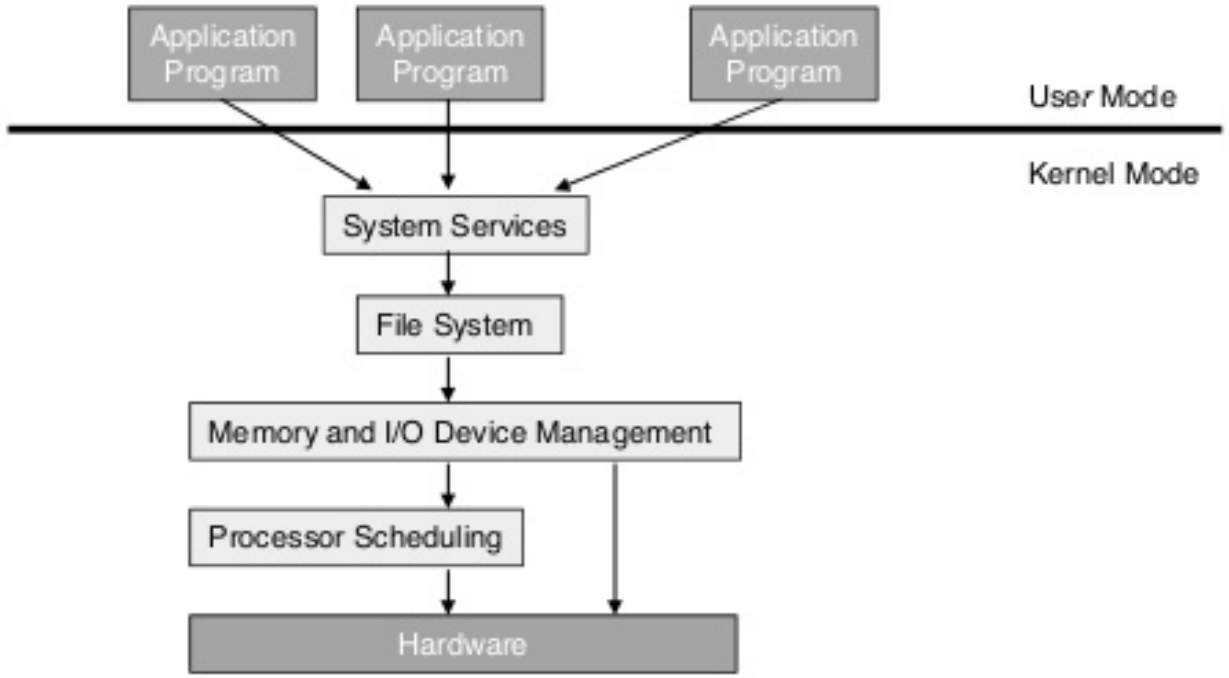
- **Support the execution of many execution units – parallel or distributed**
  - OS's Primary function is to support the execution of many (distributed) programs - **Protocols**
    - **Resource availability & Dispatching**
    - Successful cooperation – circumstance is stable or not
  - Evolution of related frameworks/platforms
    - From Amoeba [变形虫] to Micro-services [微服务]

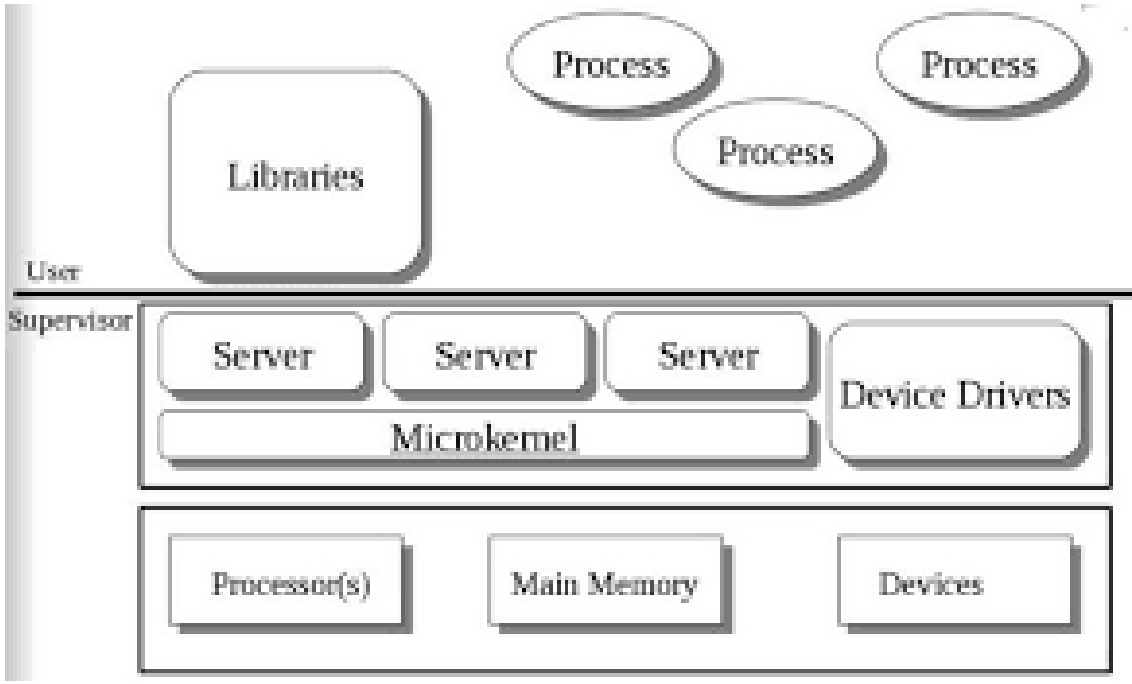
OS? – Process scheduling + Synchronization

- **As roles you've learned from OS course**
  - Goal: Support the concurrent execution of many processes
  - 2 roles
    - Resource manager + (~~Friendly interface~~) Cooperation Monitor

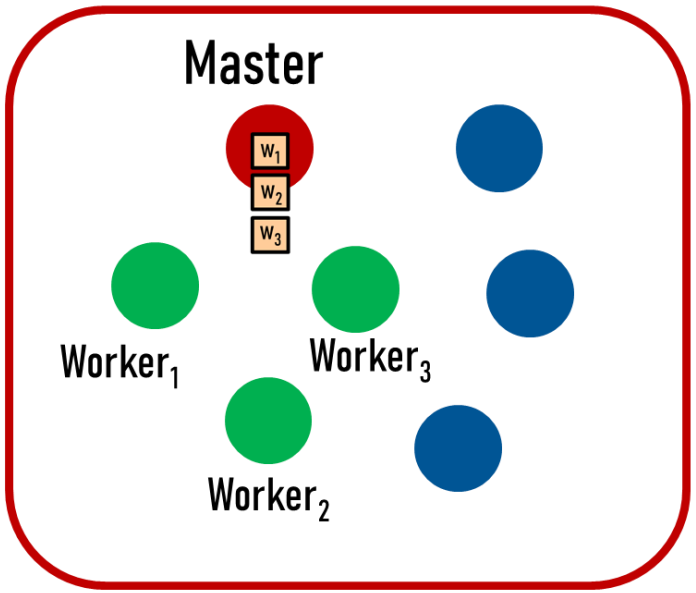


OS's programs are modularized and structured

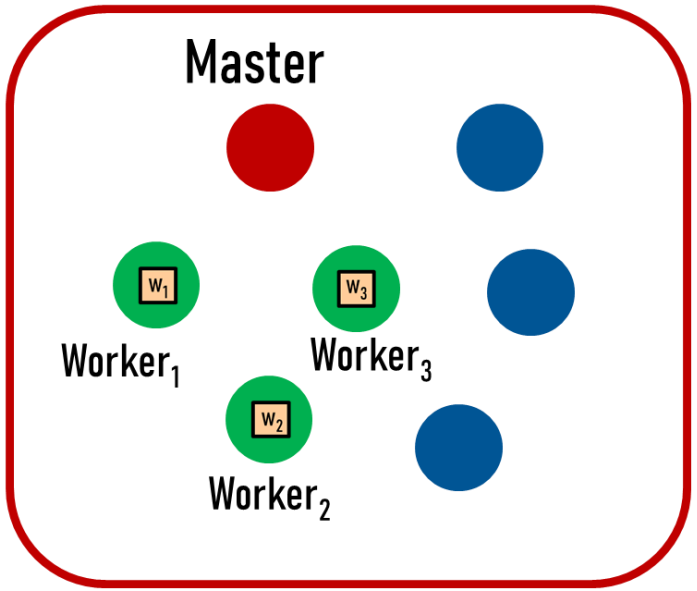




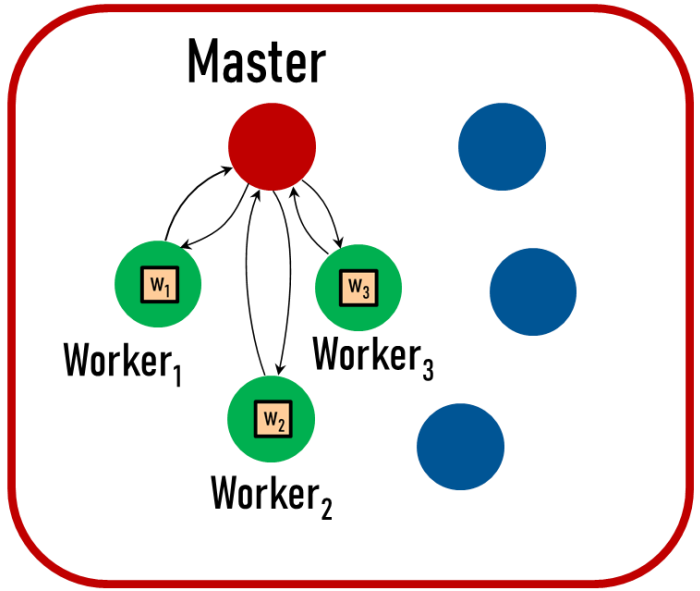
- “modularized and structured” are also the way to organize the services to support the execution of many distributed programs



(a) Master received the Work

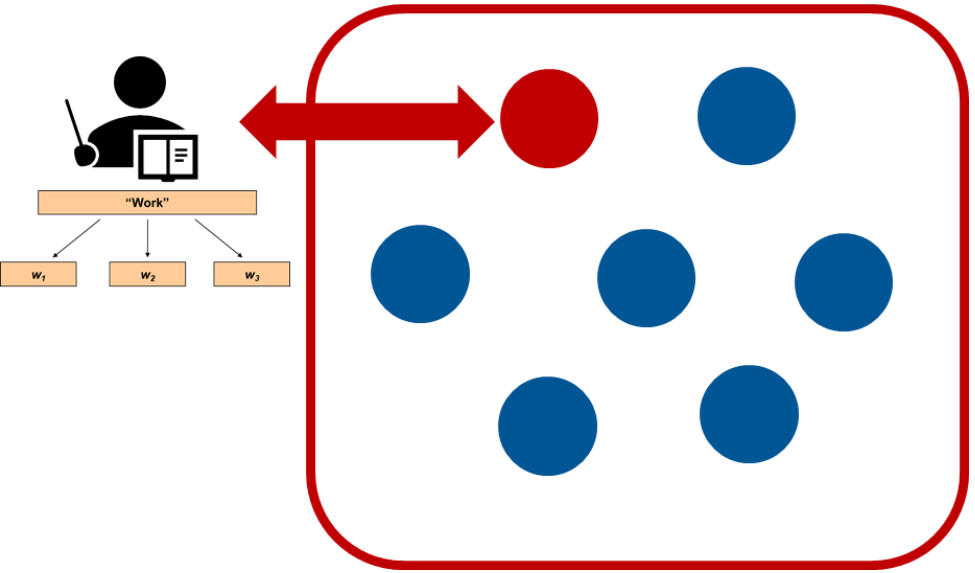


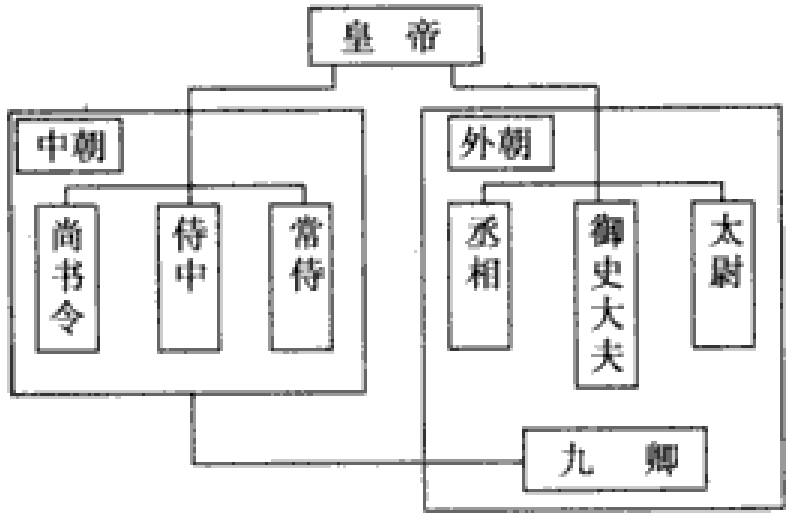
(b) Master dispatches work to Workers



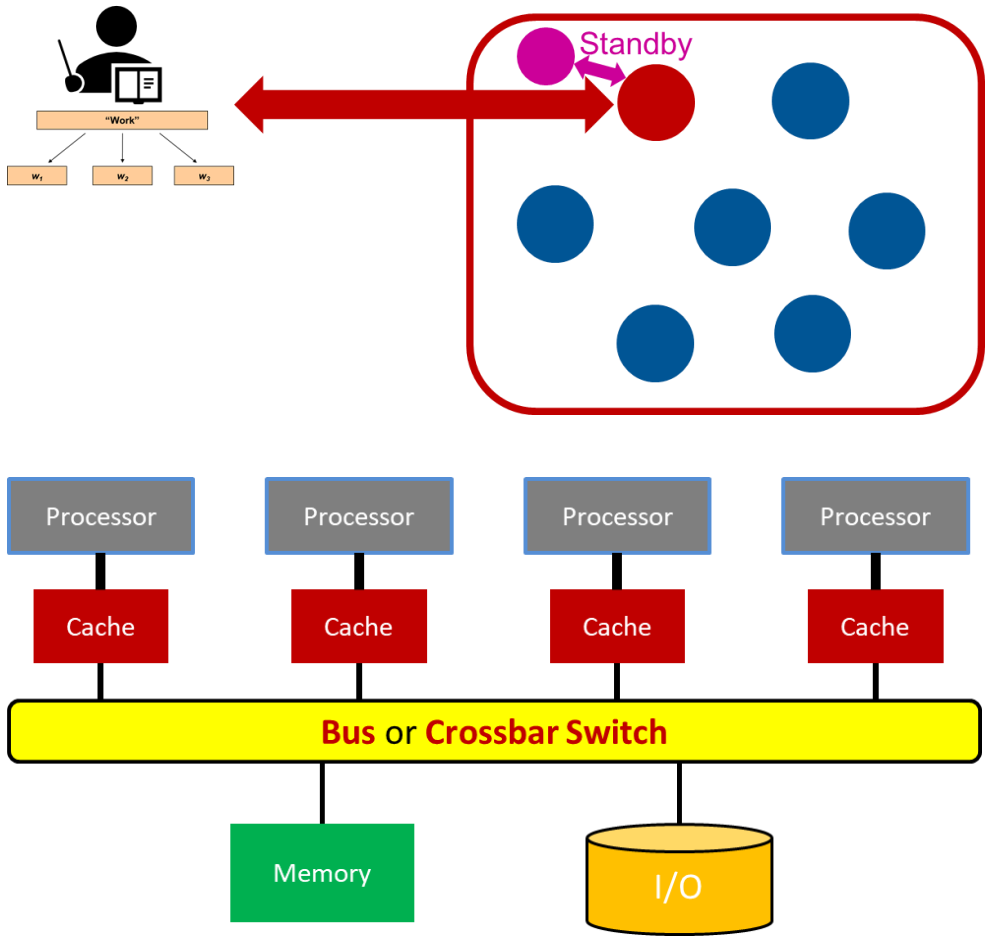
(c) Go and return between Master and Workers

借助生活中的例子来理解 Large Scale Computing Program 运行时的样子

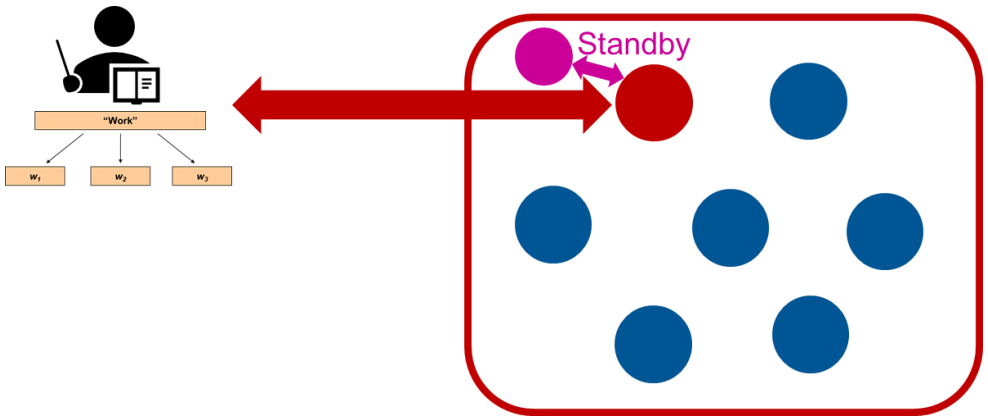


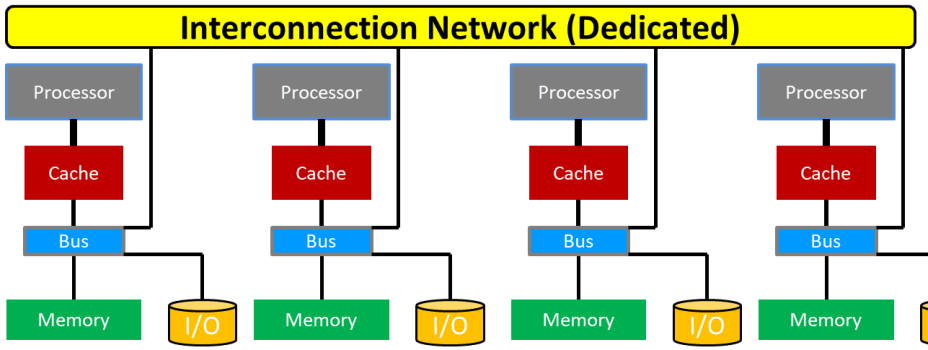


- **也就是说，一群算力单元要为许多用户的并发程序进行服务**
- **类似社会团体，算力单元群一般**
  - 有一个对外的服务节点
    - 负责管理可用的算力节点
  - 当有任务提交时，服务节点根据任务的工作量选择适当的算力节点协同完成计算
    - 过程中服务节点与算力节点间，甚或算力节点间有交互的协同要求(彼此往复发送信息)
- **类似社会团体，算力单元群也可能不稳定**
  - 服务节点损坏
  - 算力节点损坏
  - 节点间通讯不畅
  - 。。。

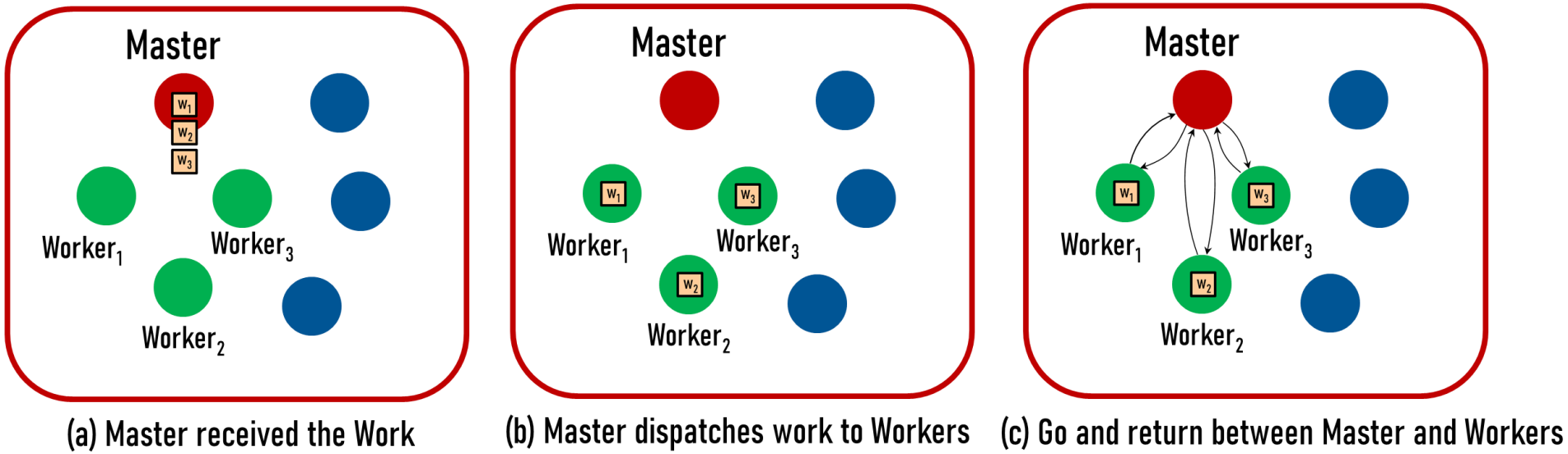


- **对于单机多核计算机系统 (SMP – UMA or NUMA)，一个OS管理全部的资源**
  - 包括运行着OS之外的其他 CPUs
  - Master-Workers 可以是线程或进程
  - 用户计算程序框架 可以使用 Pthreads, Java, OpenMP, OpenCL, CUDA, MPI等
- **现代OS基本能够支持**
  - 任务的分派；通信；同步；死锁等功能
- **发展到现在，这类系统故障率已经很低了**
  - 但，考虑容错(Fault Tolerance)仍然是有价值的



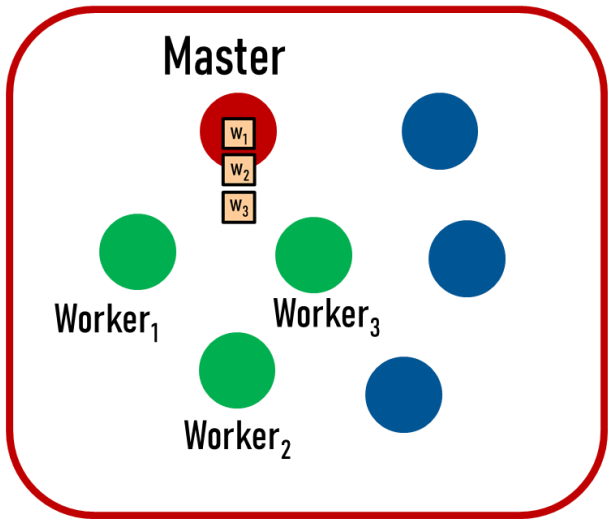


- **对于多机系统 (MPP 和 Cluster)，每个物理节点都有自己的 OS，**
  - 每个物理节点上运行着不同角色的服务进程
    - Distributed File, Transaction, Monitor etc.
    - 其中有一个Job Scheduler管理用户并发程序的执行
- **任务的分派；同步；死锁等功能有相应的服务进程群提供支持**
  - 一如人类社团内由不同的小群体实现不同的职能划分
    - 如一致性算法 包括： 2PC 、 3pc 、 paxos 、 Raft、ZAB等
      - ✓ 2PC / 3PC 协议用于保证属于多个数据分片上操作的原子性。这些数据分片可能分布在不同的服务器上，2PC / 3PC 协议保证多台服务器上的操作要么全部成功，要么全部失败
      - ✓ Paxos、Raft、Zab 算法用于保证同一个数据分片的多个副本之间的数据一致性
- **容错是需要认真对待的问题！**
- **用户计算程序框架 可以使用 MPI, Big Data, MPI+ 等**

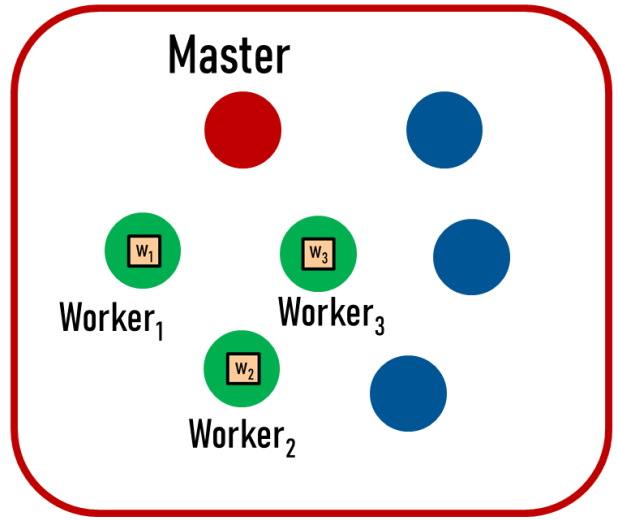


(a) Master received the Work (b) Master dispatches work to Workers (c) Go and return between Master and Workers

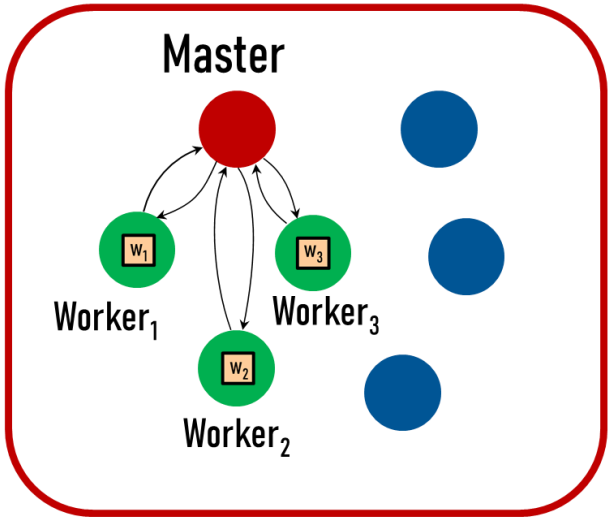
- **Many challenges**
  - Available resources – alive or not, busy or not?
  - Cooperation under unstable circumstance –
    - Serialize data access – event ordering, distributed transaction, …
    - Fault tolerance – WAL, Standby, …



(a) Master received the Work



(b) Master dispatches work to Workers



(c) Go and return between Master and Workers

负载均衡

数据访问的原子性要求

数据备份一致性的保障

系统具备高可用的特征

程序调用

存活与否

标识资源访问的次序

具有容错能力

具有灾难中恢复的能力

备份的功能

资源的互联

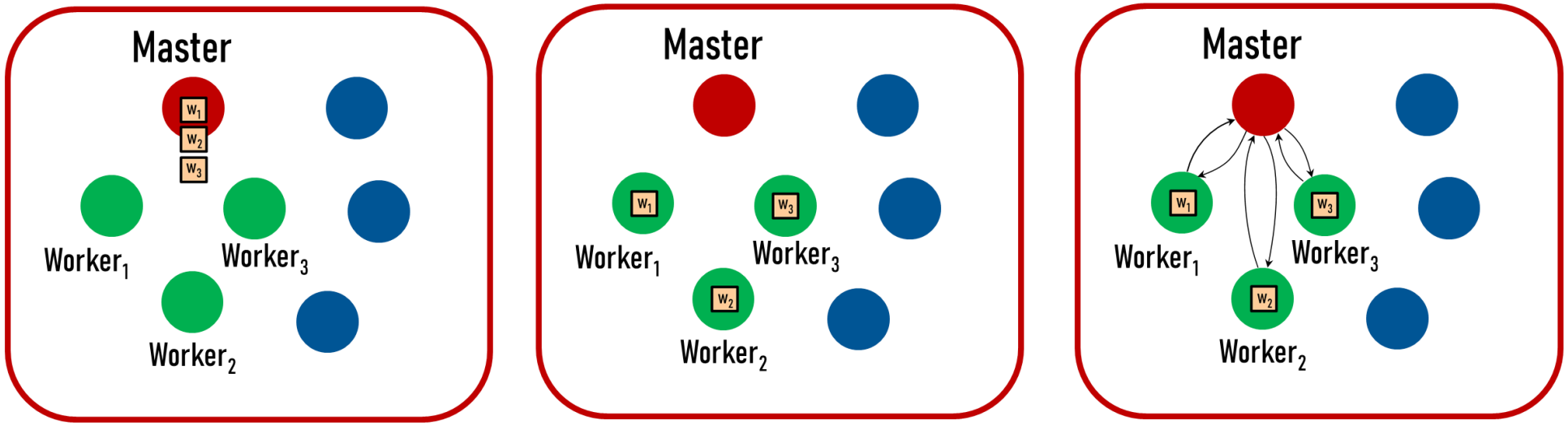
找到并使用资源

协同任务的可靠性

系统的整体功能需求

Chapter 5: Distributed OS

- **Support the execution of many execution units – parallel or distributed**
  - OS's Primary function is to support the execution of many (distributed) programs - **Protocols**
    - **Resource availability & Dispatching**
    - Successful cooperation – circumstance is stable or not
  - Evolution of related frameworks/platforms
    - From Amoeba [变形虫] to Micro-services [微服务]



(a) Master received the Work (b) Master dispatches work to Workers (c) Go and return between Master and Workers

System requirements

Cooperating safely

Resource Allocation & Monitoring

Load Balancing (DRF etc.)

Transactional Isolation (WAL, Snapshot, …)

Consensus (PAXOS, RAFT, Byzantine, ZAB, …, Leader-Follower Election)

RPC (RMI, gRPC, …)

Heartbeat

Event Ordering (Real time based, Lamport's Logical Clock, …)

Networking (TCP/IP, InfiniBand, etc.)

HA (High Availability)

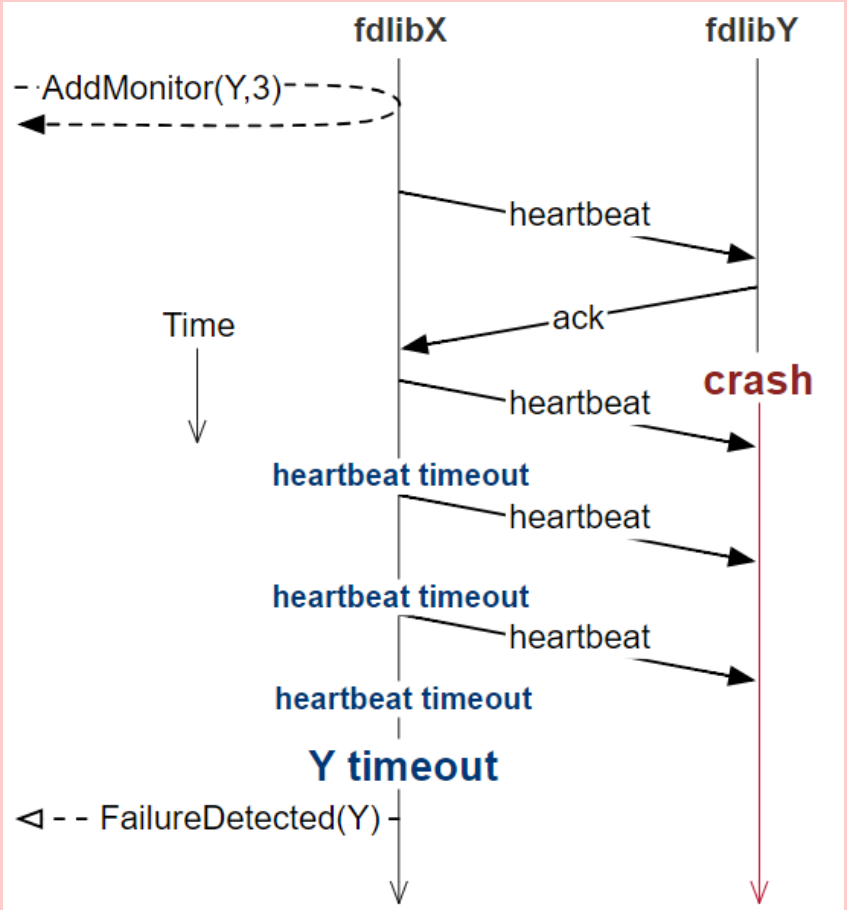
Fault Tolerance

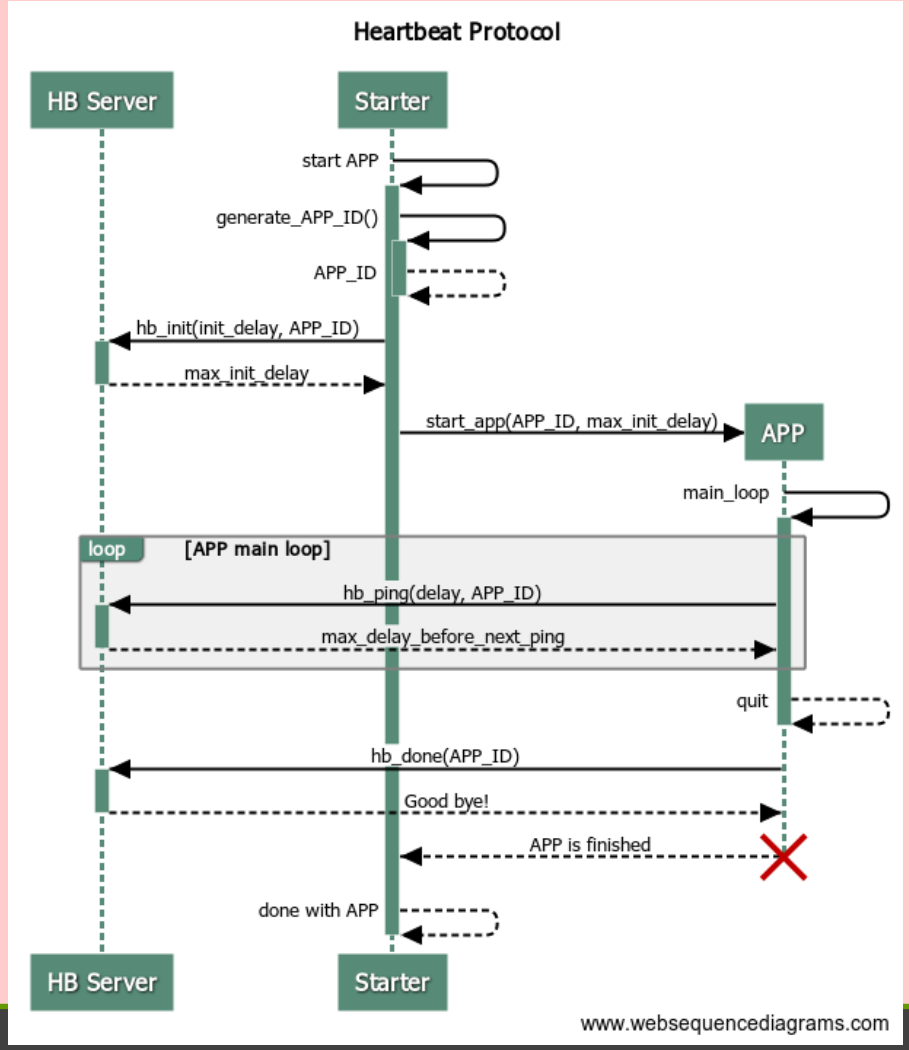
DR (Disaster Recovery)

Backup(Replication)

Heartbeat

Alive or Dead? - Heartbeat protocol





Dominant Resource Fairness (DRF)

- **A user's *dominant resource* is resource user has biggest share of**
  - Example:

    Total resources: **8 CPU** **5 GB**

    User 1's allocation: **2 CPU** **1 GB**

    25% CPUs **20% RAM**

    Dominant resource of User 1 is CPU (as **25%** > 20%)

- **A user's *dominant share:* fraction of dominant resource allocated**
  - User 1's dominant share is **25%**

Dominant Resource Fairness: Fair Allocation of Multiple Resource Types
Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, Ion Stoica, NSDI'11

https://www.cnblogs.com/yuananyun/p/5137321.html

考虑一个有9个cpu和18GB的系统，有两个用户：用户A的每个任务都请求（1CPU，4GB）资源；用户B的每个任务都请求（3CPU，1GB）资源。如何为这种情况构建一个公平分配策略?

对于用户A，每个任务需要消耗的资源为<1/9,4/18>=<1/9,2/9>,所以A的dominant shares为内存，比例为2/9

对于用户B，每个任务需要消耗的资源为<3/9,1/18>=<1/3,1/18>,所以B的dominant shares为cpu，比例为1/3

$$\max(x, y) \quad \text{(Maximize allocations)}$$

subject to

$$x + 3y \leq 9 \quad \text{(CPU constraint)}$$

$$4x + y \leq 18 \quad \text{(Memory constraint)}$$

$$\frac{2x}{9} = \frac{y}{3} \quad \text{(Equalize dominant shares)}$$

Solving this problem yields[2] $x = 3$ and $y = 2$. Thus, user A gets $\langle 3\text{ CPU}, 12\text{ GB}\rangle$ and B gets $\langle 6\text{ CPU}, 2\text{ GB}\rangle$.

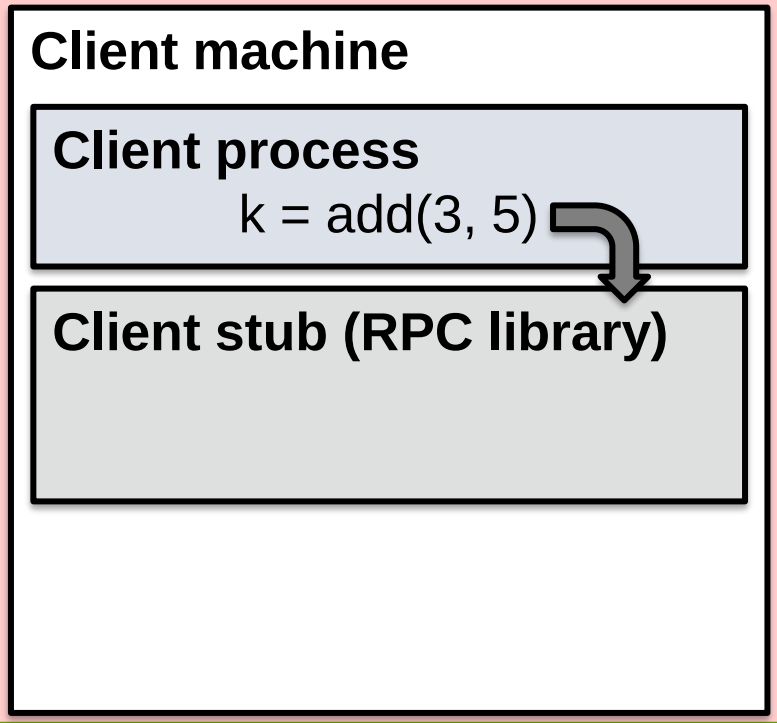
通过列不等式方程可以解得给用户A分配3份资源，用户B分配2份资源是一个很好的选择。

Chapter 5: Distributed OS

- **Support the execution of many execution units – parallel or distributed**
  - OS's Primary function is to support the execution of many (distributed) programs - **Protocols**
    - **Resource availability & Dispatching**
    - Successful cooperation – circumstance is stable or not
      - ✓ RPC, Event Ordering, Transaction control (2/3 PC)
      - ✓ WAL, Consensus (PAXOS, RAFT, ZAB, etc.), ← Fault Tolerance (or as HA)
  - Evolution of related frameworks/platforms
    - From Amoeba [变形虫] to Micro-services [微服务]

A day in the life of an RPC

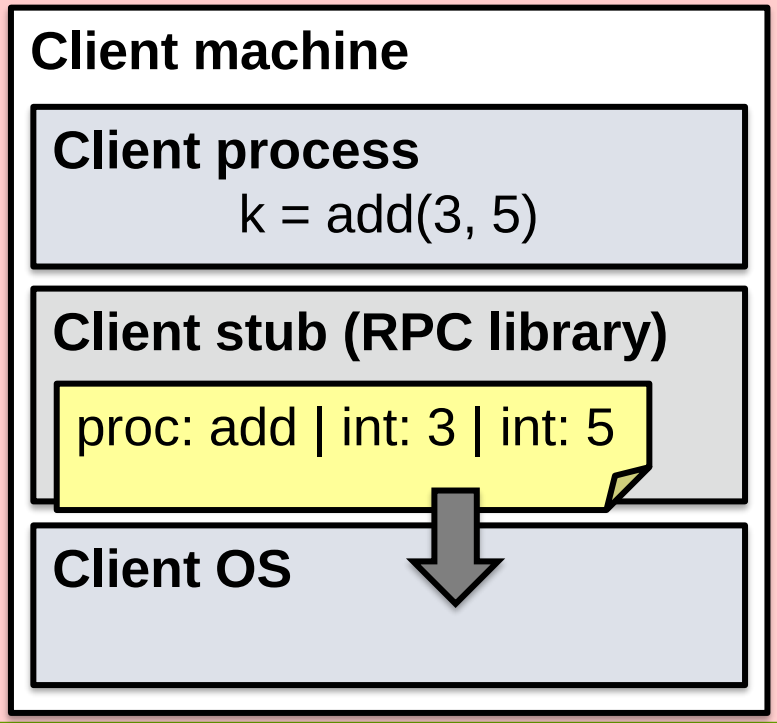
1. **Client calls stub function (pushes params onto stack)**



A day in the life of an RPC

1. Client calls stub function (pushes params onto stack)

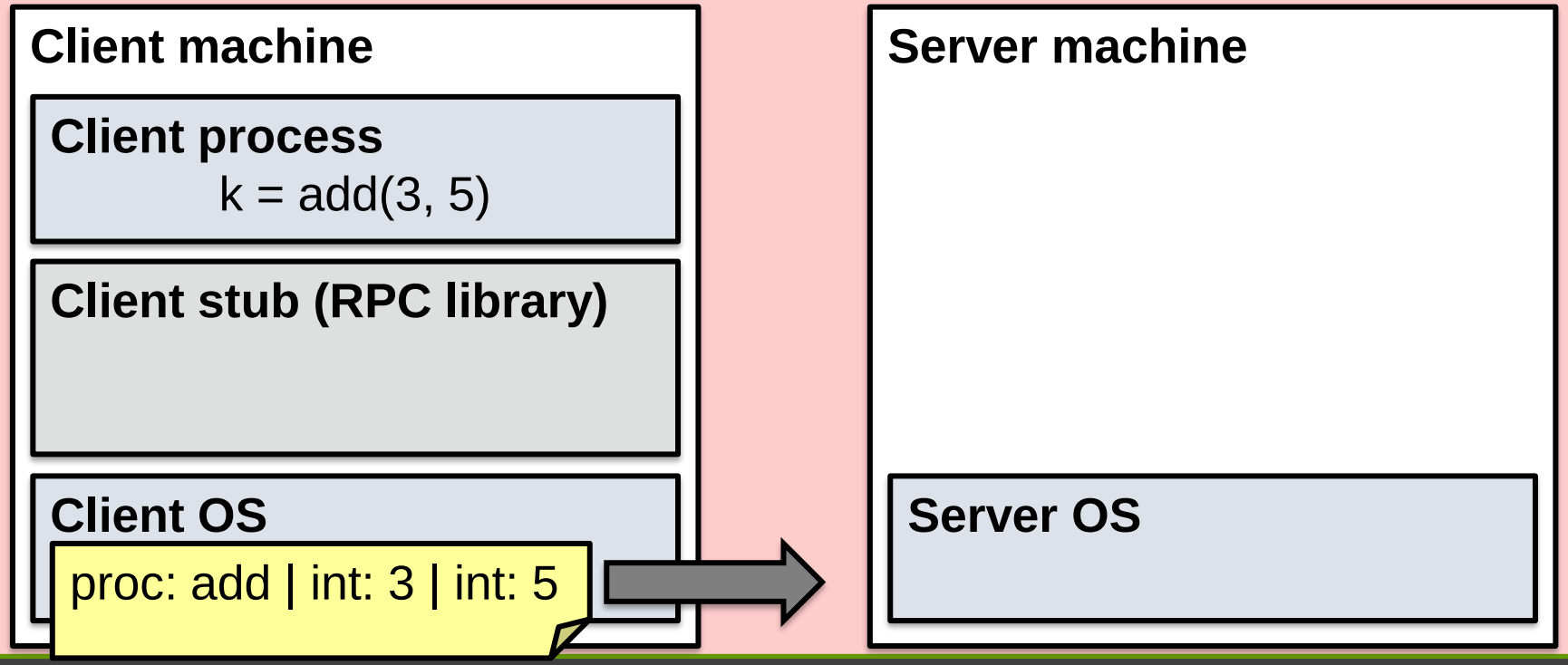
2. Stub marshals parameters to a network message



A day in the life of an RPC

2. Stub marshals parameters to a network message

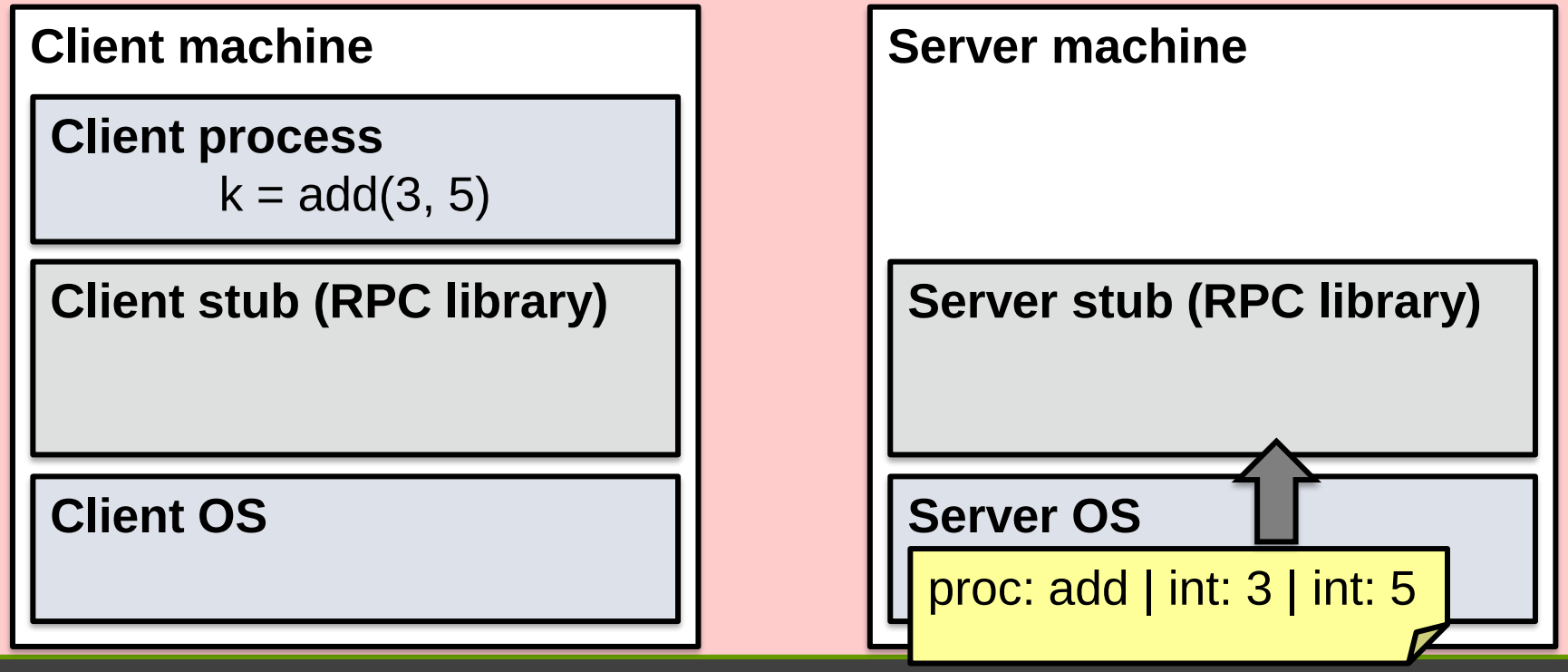
3. OS sends a network message to the server



A day in the life of an RPC

3. OS sends a network message to the server

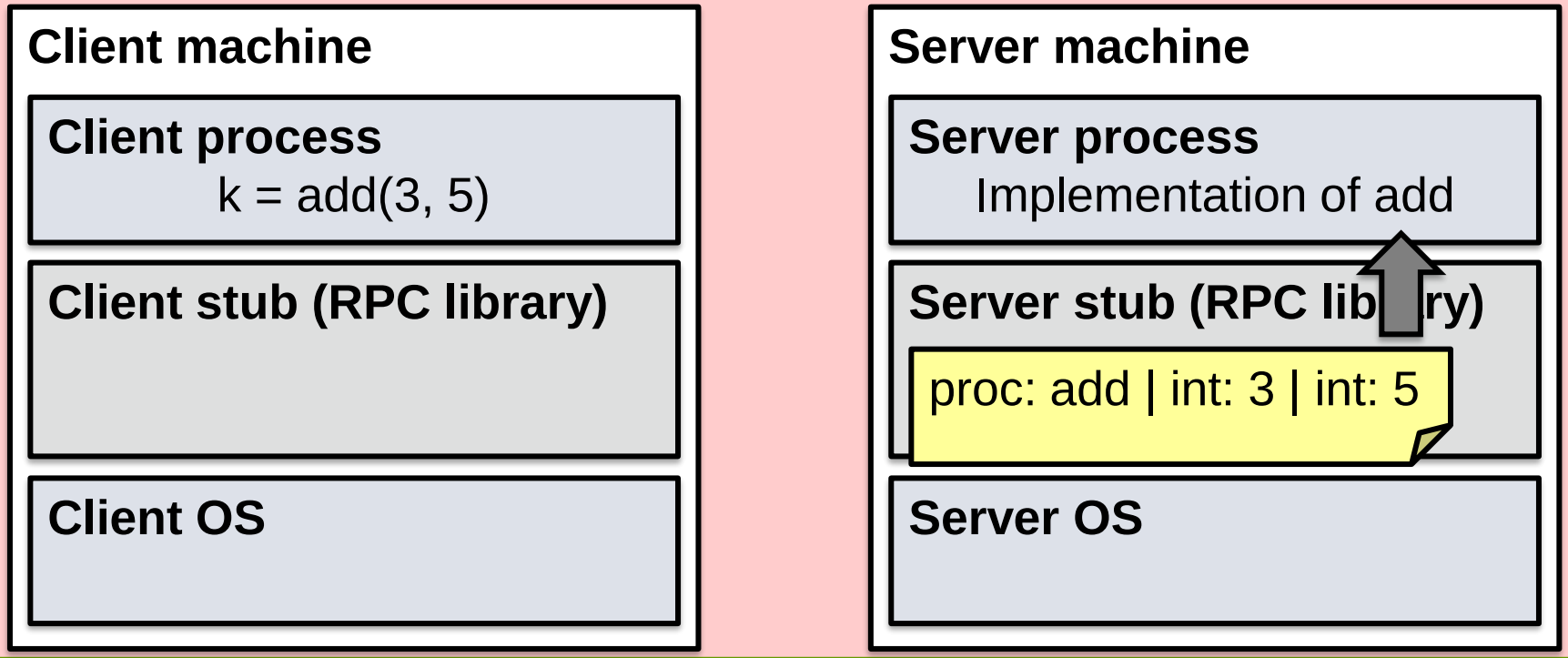
4. Server OS receives message, sends it up to stub



A day in the life of an RPC

4. Server OS receives message, sends it up to stub

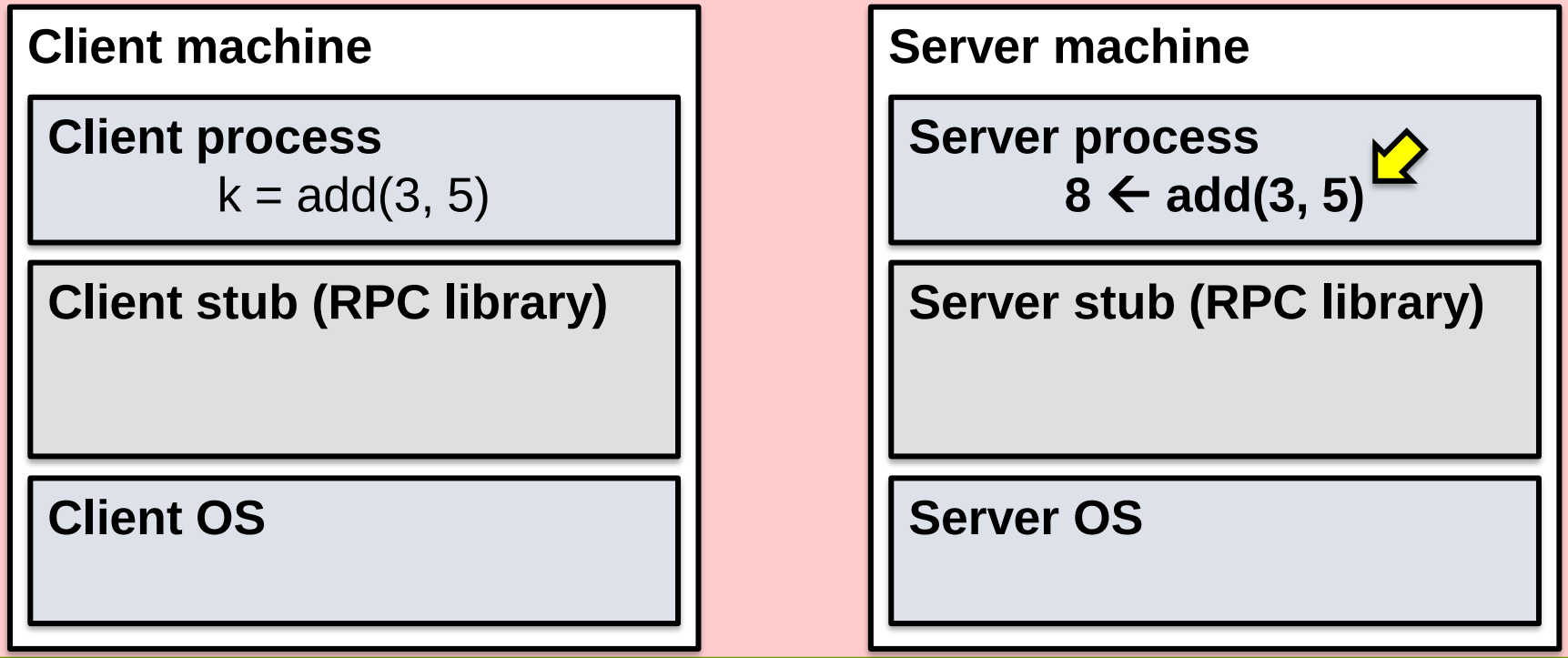
5. Server stub unmarshals params, calls server function



A day in the life of an RPC

5. Server stub unmarshals params, calls server function

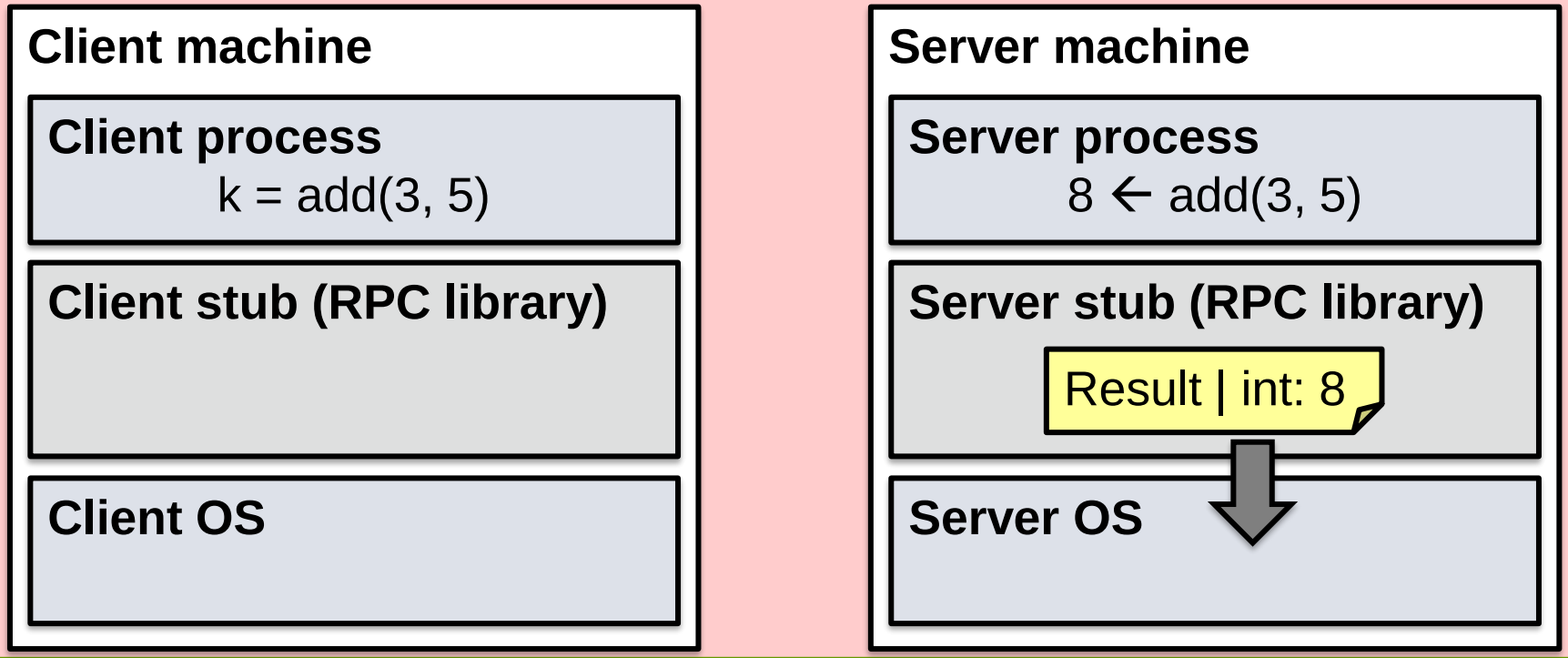
6. **Server function runs, returns a value**



A day in the life of an RPC

6. Server function runs, returns a value

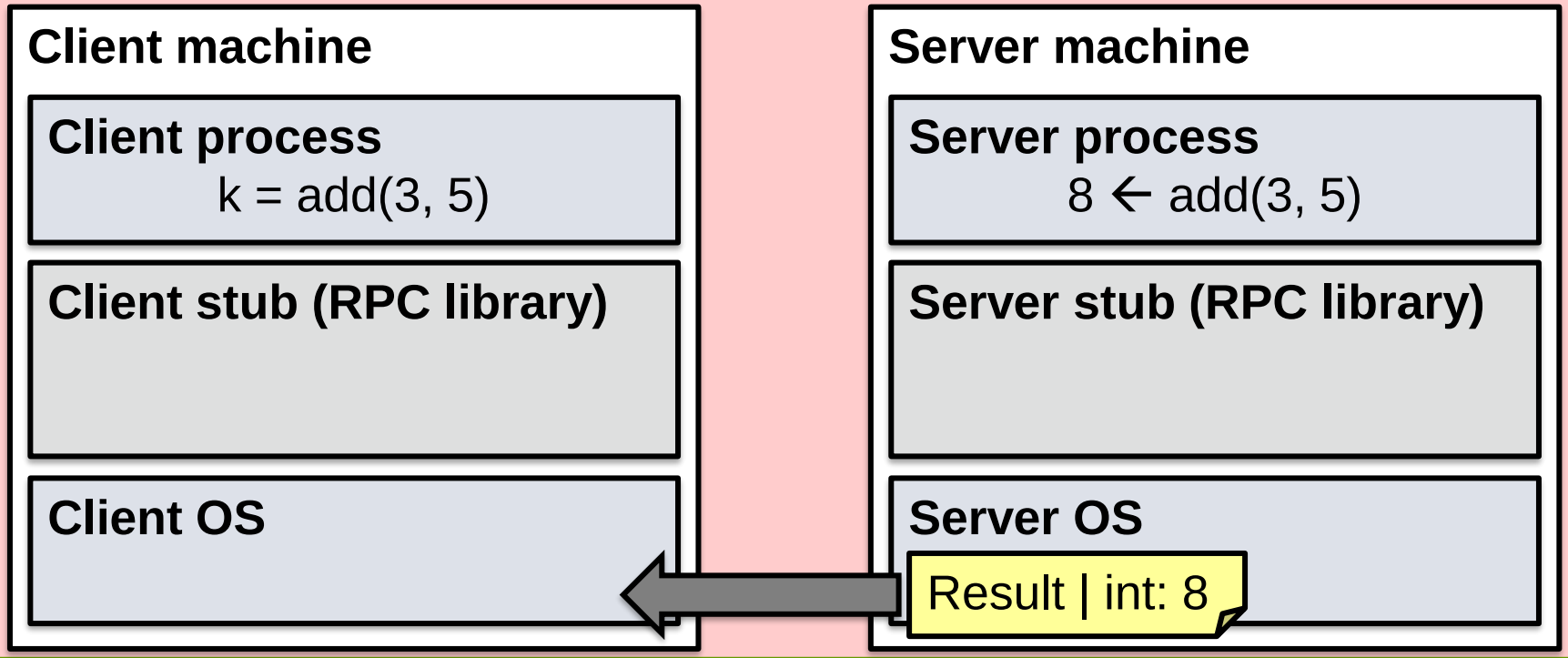
7. **Server stub marshals the return value, sends msg**



A day in the life of an RPC

7. Server stub marshals the return value, sends msg

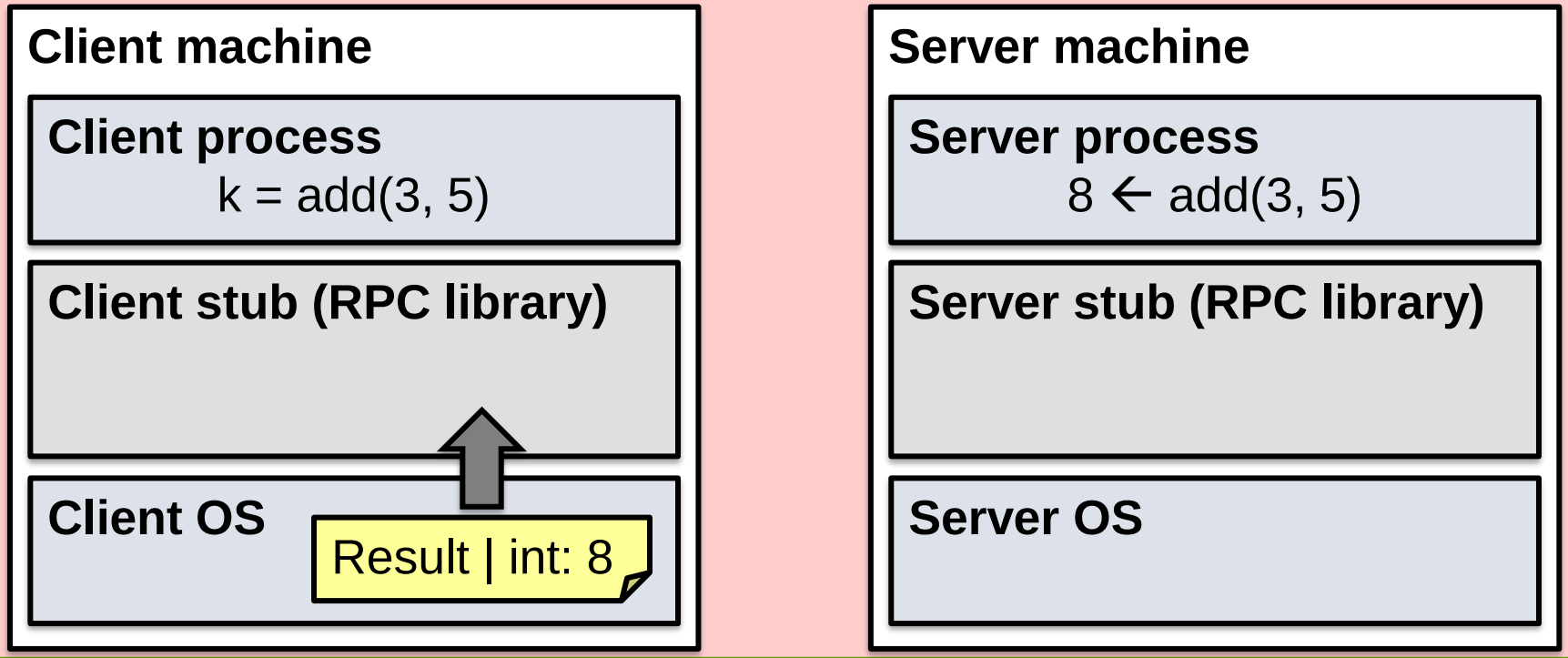
8. **Server OS sends the reply back across the network**



A day in the life of an RPC

8. Server OS sends the reply back across the network

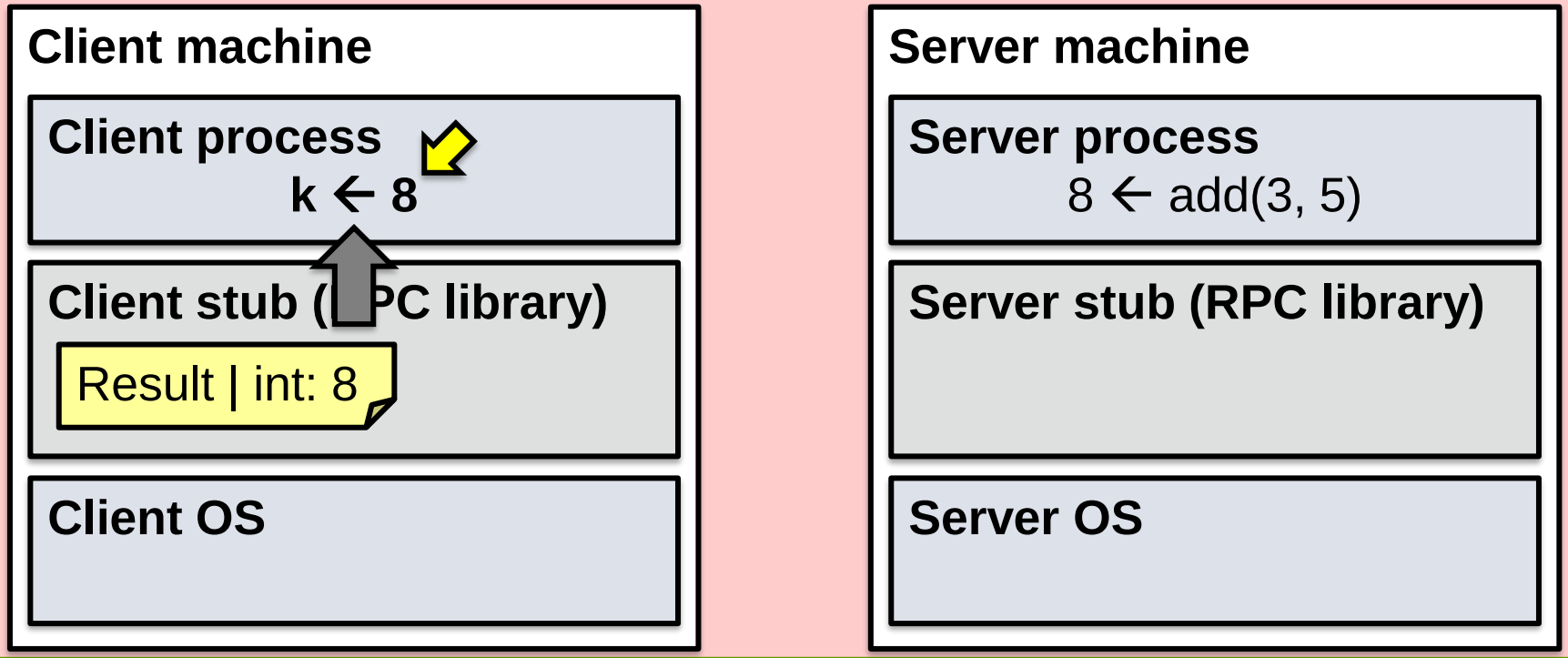
9. Client OS receives the reply and passes up to stub



A day in the life of an RPC

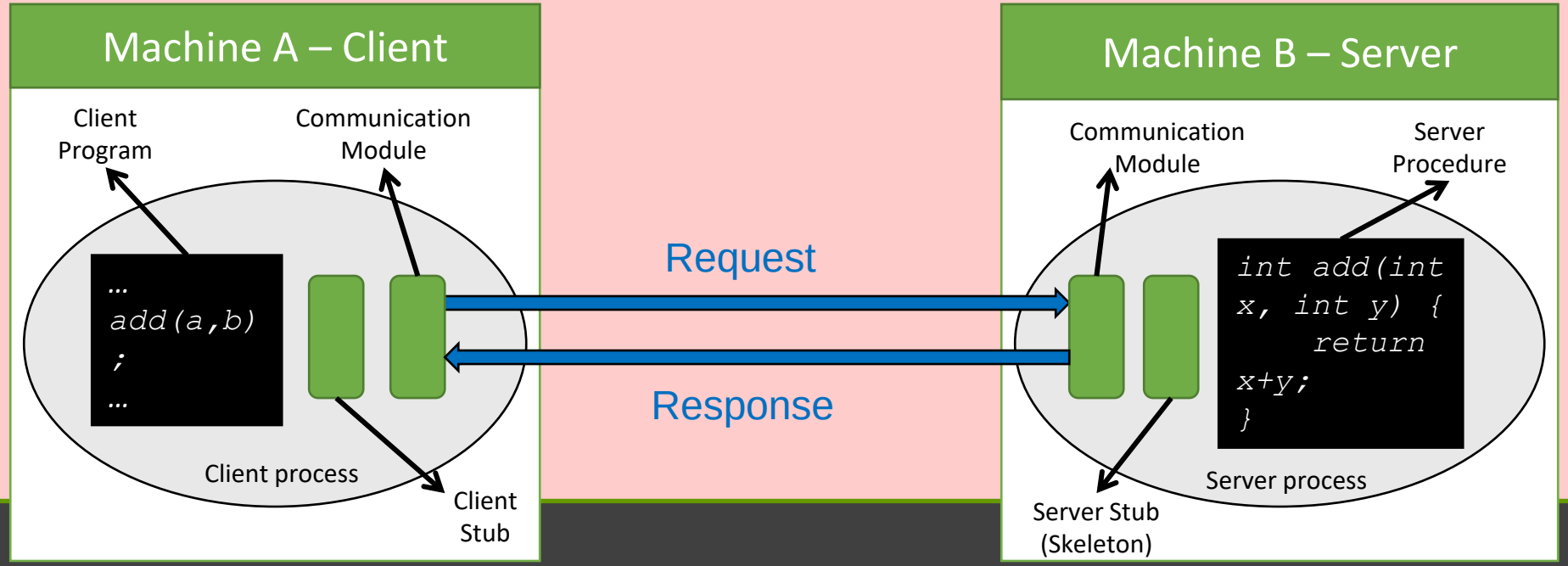
9. Client OS receives the reply and passes up to stub

10. **Client stub unmarshals return value, returns to client**



Remote Procedure Calls (RPC)

- RPC enables a sender to communicate with a receiver using a simple procedure call
  - No communication or message-passing is visible to the programmer
- Basic RPC Approach:



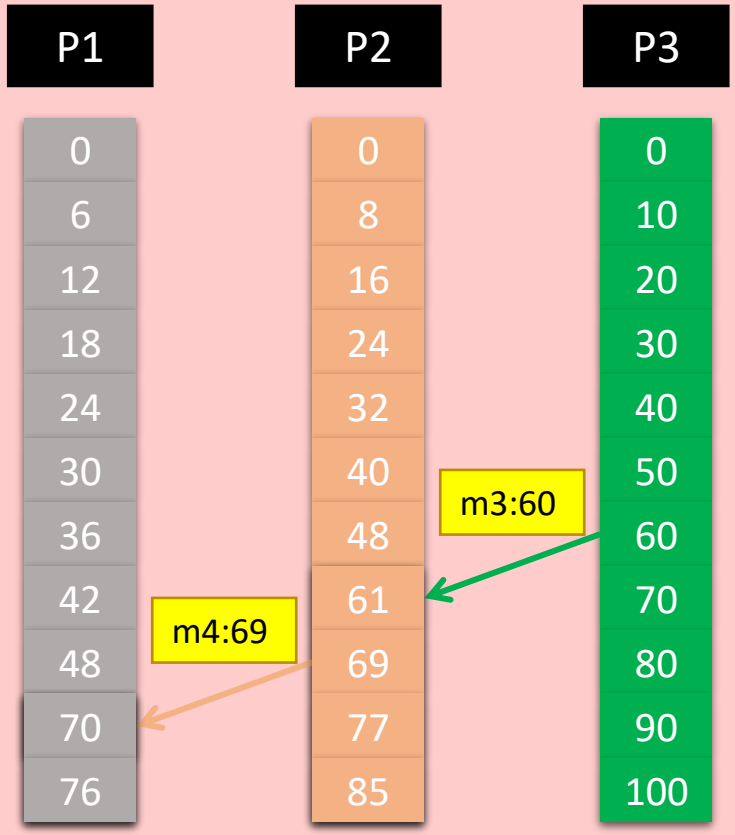
Event Ordering (Real time based, Lamport's Logical Clock, …)

Lamport's Logical Clock (Lamport's Clock Algorithm)

- When a message is being sent:
  - Each message carries a **`timestamp`** according to the sender's logical clock

- When a message is received:
  - If the receiver logical clock is less than the message sending time in the packet, then adjust the receiver's clock such that:

```
currentTime = timestamp + 1
```



Consensus

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated

- Will see how to relax this in case of site failures later → Majority Consensus (PAXOS, RAFT, ZAB, …)

Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an unreplicated data item *Q* residing at site S_i, a message is sent to S_i's lock manager.
  - If *Q* is locked in an incompatible mode, then the request is delayed until it can be granted.
    - ✓ More than (**n/2+1**) nodes respond OK
  - When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted

quorum英 ['kwɔːrəm] 美 ['kwɔːrəm]
•**n.** 法定人数

Quorum Consensus Protocol

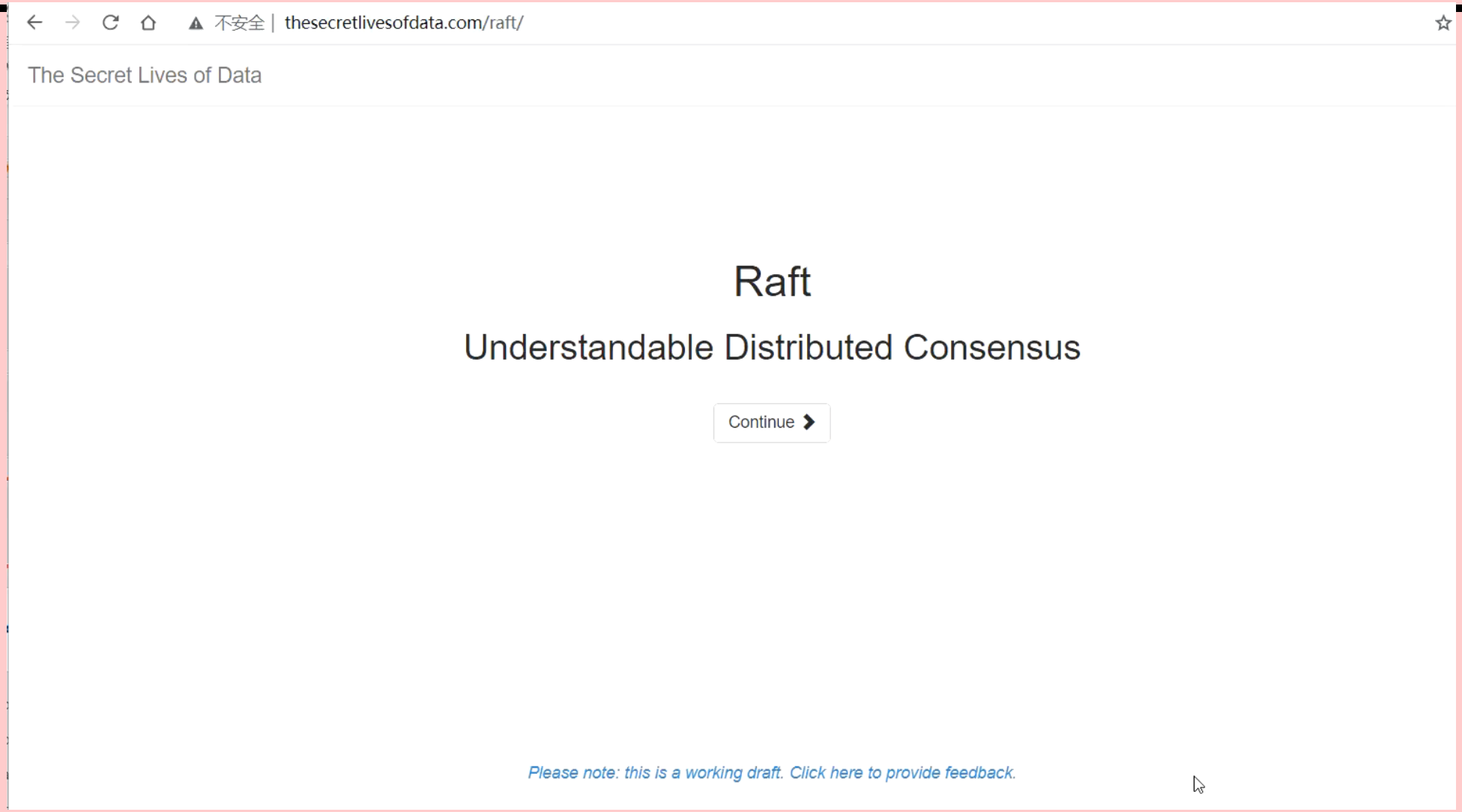
- A generalization of both majority and biased protocols
- Each site is assigned a weight.
  - Let S be the total of all site weights
- Choose two values read quorum Q_r and write quorum Q_w
  - Such that $Q_r + Q_w > S$ and $2 * Q_w > S$
  - Quorums can be chosen (and S computed) separately for each item
- Each read must lock enough replicas that the sum of the site weights is >= Q_r
- Each write must lock enough replicas that the sum of the site weights is >= Q_w
- For now we assume all replicas are written
  - Extensions to allow some sites to be unavailable described later

RAFT: In Search of an Understandable Consensus Algorithm

- **Raft basics: the servers**
  - A RAFT cluster consists of several servers
    - Typically five
  - Each server can be in one of three states
    - ***Leader***
    - ***Follower***
    - ***Candidate*** (to be the new leader)
  - Followers are passive:
    - Simply reply to requests coming from their leader

1. **Leader election**
   - Select one of the servers to act as cluster leader
   - Detect crashes, choose new leader
2. **Log replication (normal operation)**
   - Leader takes commands from clients, appends them to its log
   - Leader replicates its log to other servers (overwriting inconsistencies)
3. **Safety**
   - Only a server with an up-to-date log can become leader

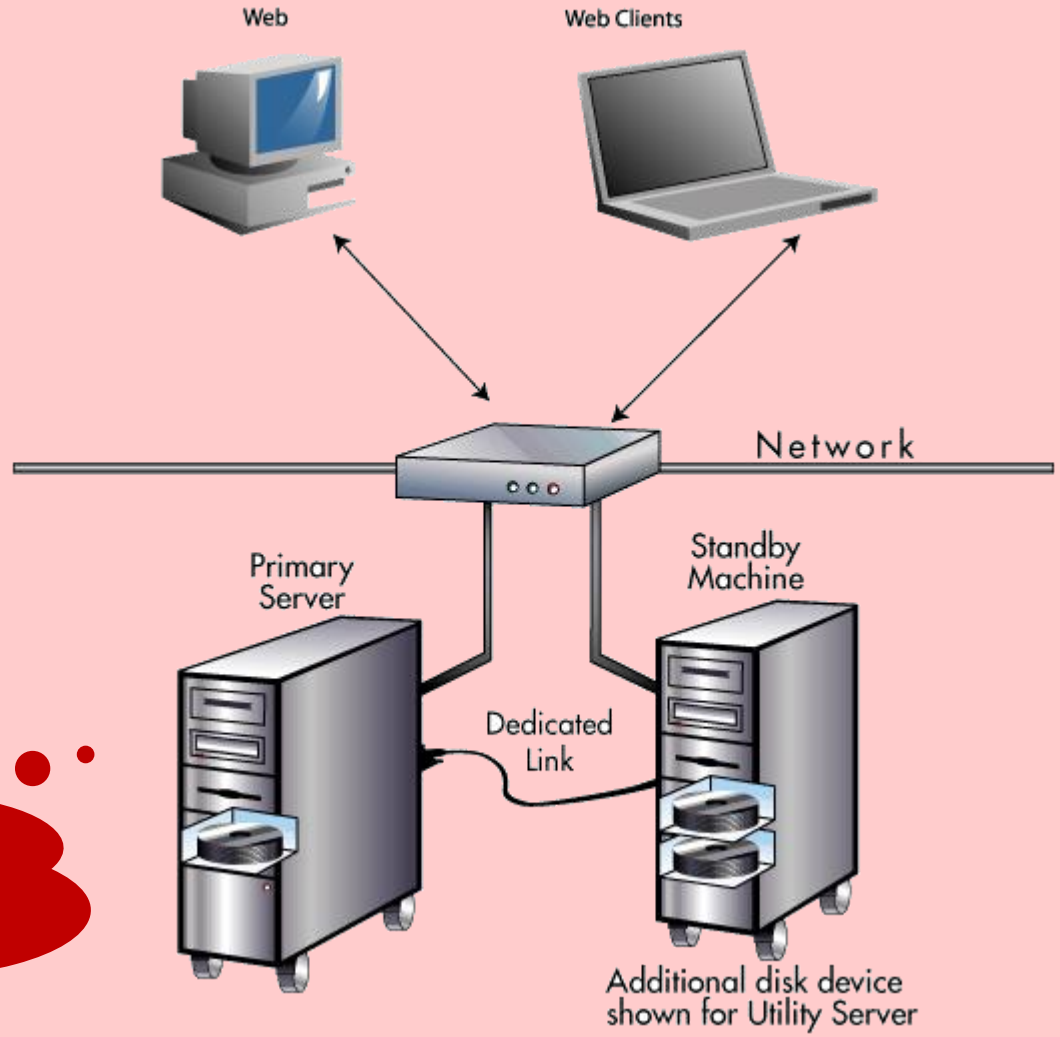
http://thesecretlivesofdata.com/raft/



Fault Tolerance

- **Fault tolerance (WAL is fundamental)**
  - Job scheduler is down ← standby Job scheduler
  - Job scheduler is OK, but PU(s) is unreachable ← Recover
    - All PUs are OK, but sub-work(s) is down

Also referred as a system with HA – High Availability



Fault tolerance

- **If an EU (work) crashes**
  - Retry on another node
  - If the same task repeatedly fails, end the job
- **If a node (PU) crashes**
  - Relaunch its current task on other nodes - what about task inputs? File system replication
- **If a task is going slowly (straggler)**

straggler 英 [ˈstræglə(r)] 美 [ˈstræglər]
n. 流浪者；落伍的士兵；离群的动物；[植] 蔓生的枝叶

  - Launch second copy of task on another node
  - Take the output of whichever finishes first

表 1 操作系统核心专题

1 goal	支持多个程序的并发运行	
3 core functions	- → 资源管理 - → 协作 - → 容错	
	单机 (冯•诺依曼架构, 多核, GPU)	分布式 (MPP, Cluster)
资源稳定，没有协作 (资源感知和分配)	- → Process/Thread - → Virtual Memory - → Hard Disk Space	- → TCP/IP - → RPC (Remote Process Call)/RMI ■ → PVM/MPI, Middleware (CORBA, DCOM…), Web Service (Globus, SOA…) etc. - → Heartbeat - → Load balancing (DRF etc.)
资源稳定，有协作 (数据不一致，死锁)	- → Synchronization - → Deadlock	- → Event Ordering (Lamport's clock, etc.) - → PAXOS basic - → Distributed Deadlock
资源不稳定，有协作 (容错 – Fault Tolerance)	有探讨，但解决不彻底	- → WAL (Write Ahead Logging) - → PAXOS advanced, RAFT, ZAB etc.

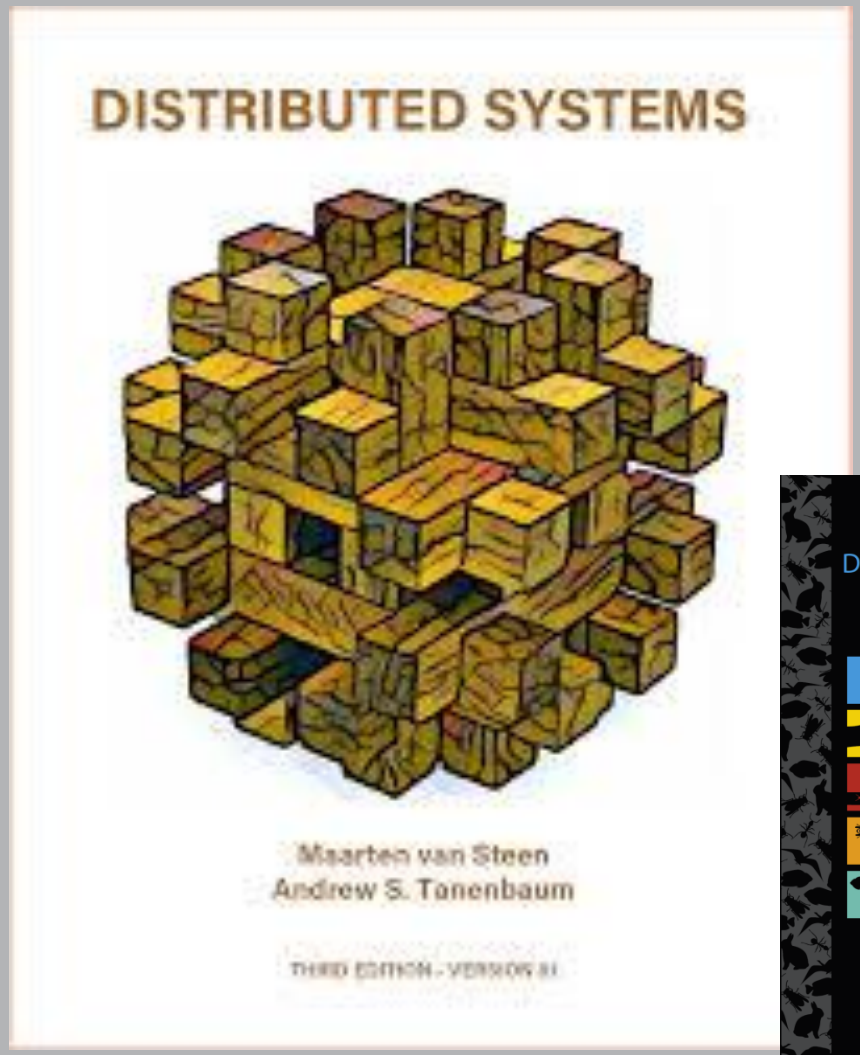
*1: 对于现代计算机系统，“资源稳定，没有协作 (资源分配和调度)” 是没有实际的实例的。这一点要先说明！

*2: 按说，还应该有“资源不稳定，没有**协调程序的执行 (Coordinate the execution of many programs)**”，这一层次其实也就可以归入“资源不稳定，有协调程序的执行”吧(?)

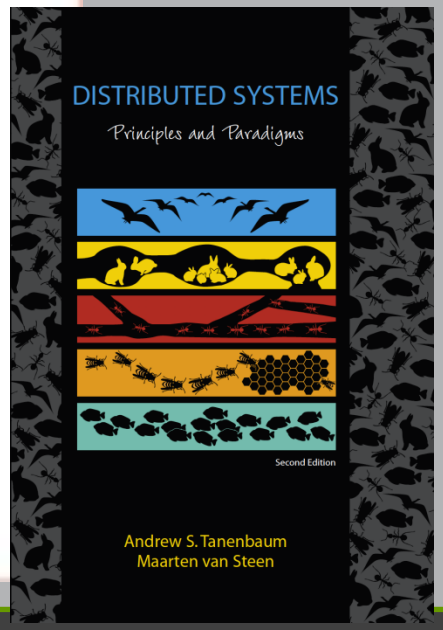
*3: 在浏览一些介绍分布式系统的视频资料，往往也直接将分布式数据存储的 2/3-PC (Phase Commit)协议纳入进来。但是，觉得这不属于“操作系统”的角色，一如传统课程中，不会将 Transaction Control 纳入到操作系统课程中一样。

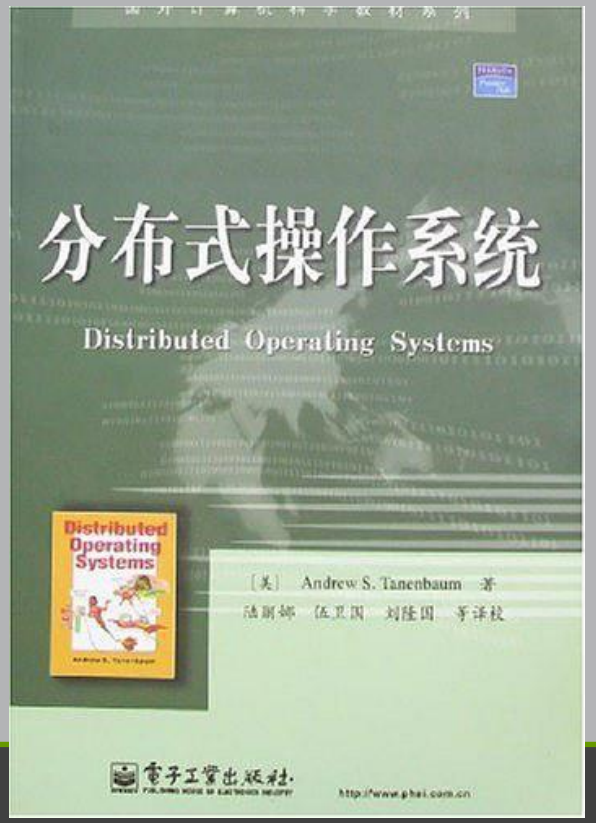
*4: 也进而将数据的分布式存储（P2P，大数据中的文件存储等）不作为“操作系统”的内容。不过，具有任务调度的 YARN、MESOS、K8S 等，视作“操作系统”的内容，因为它们都是要支持多个程序的并发执行。

前人的总结 – “雾里看花”的感觉



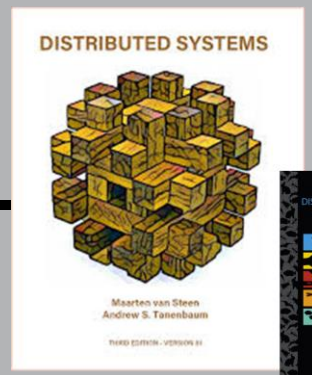
- Distributed Systems 3rd edition (2017)
- Andrew S Tanenbaum, Maarten Van Steen



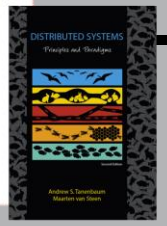


Traditionally, we have DOS, NOS, Middleware

- DOS (Distributed Operating Systems)
- NOS (Network Operating Systems)
- Middleware

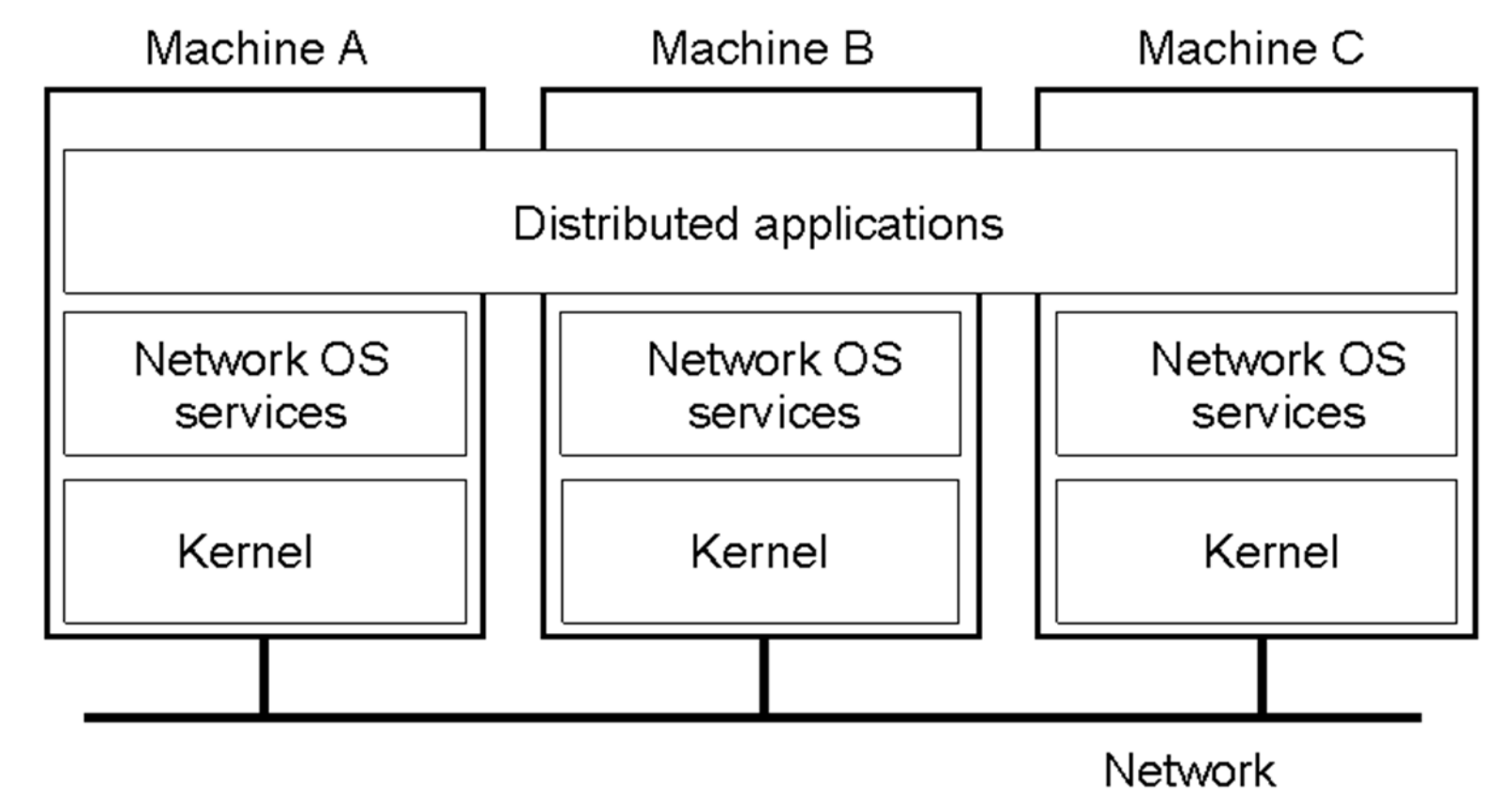


- Distributed Systems 3rd edition (2017)
- Andrew S Tanenbaum, Maarten Van Steen



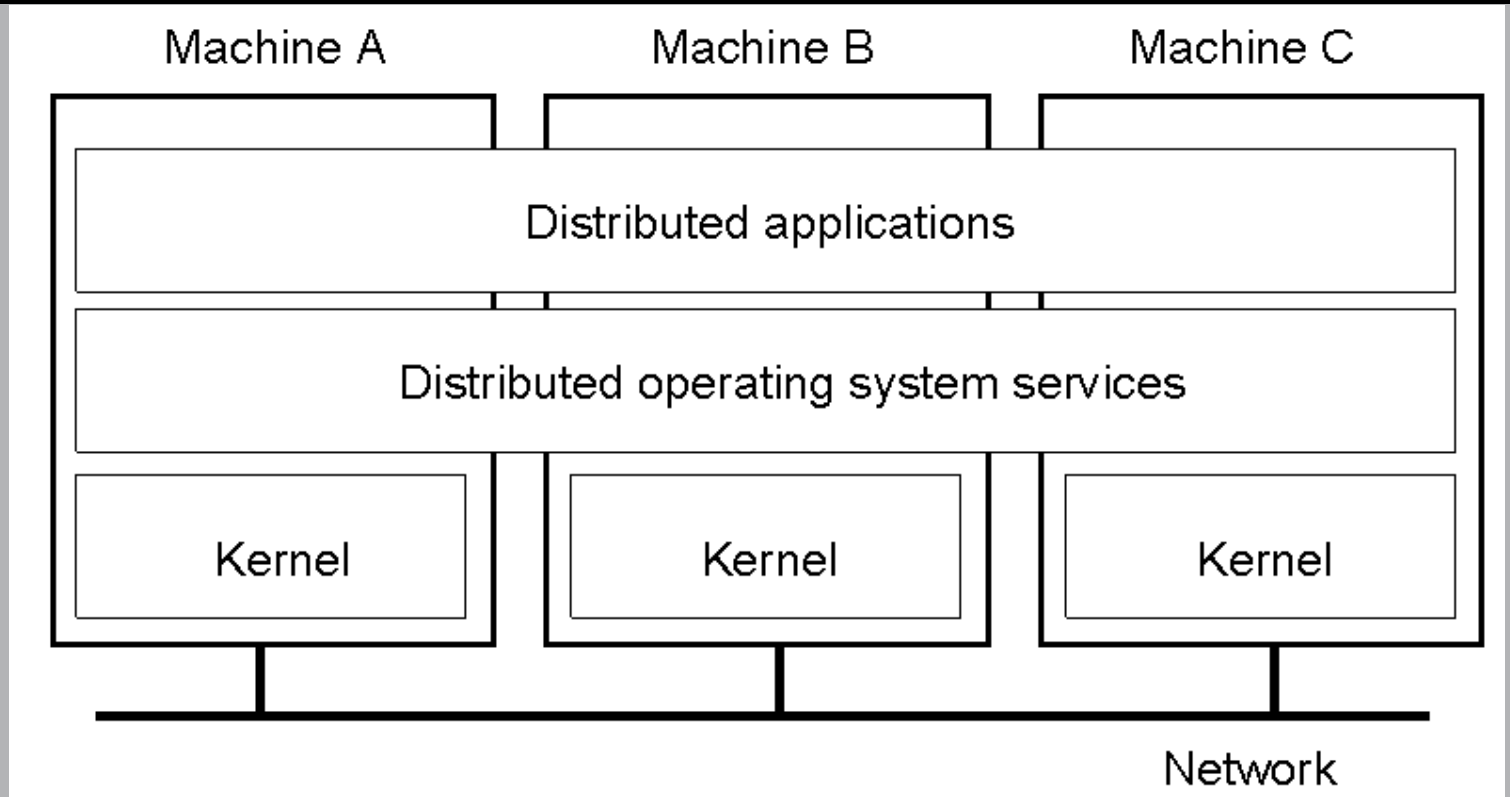
System	Description	Main Goal
DOS	Tightly-coupled operating system for multi-processors and **homogeneous multicomputers**	Hide and manage hardware resources
NOS	Loosely-coupled operating system for **heterogeneous multicomputers** (LAN and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general-purpose services	Provide distribution transparency

Traditionally, we have DOS, NOS, Middleware



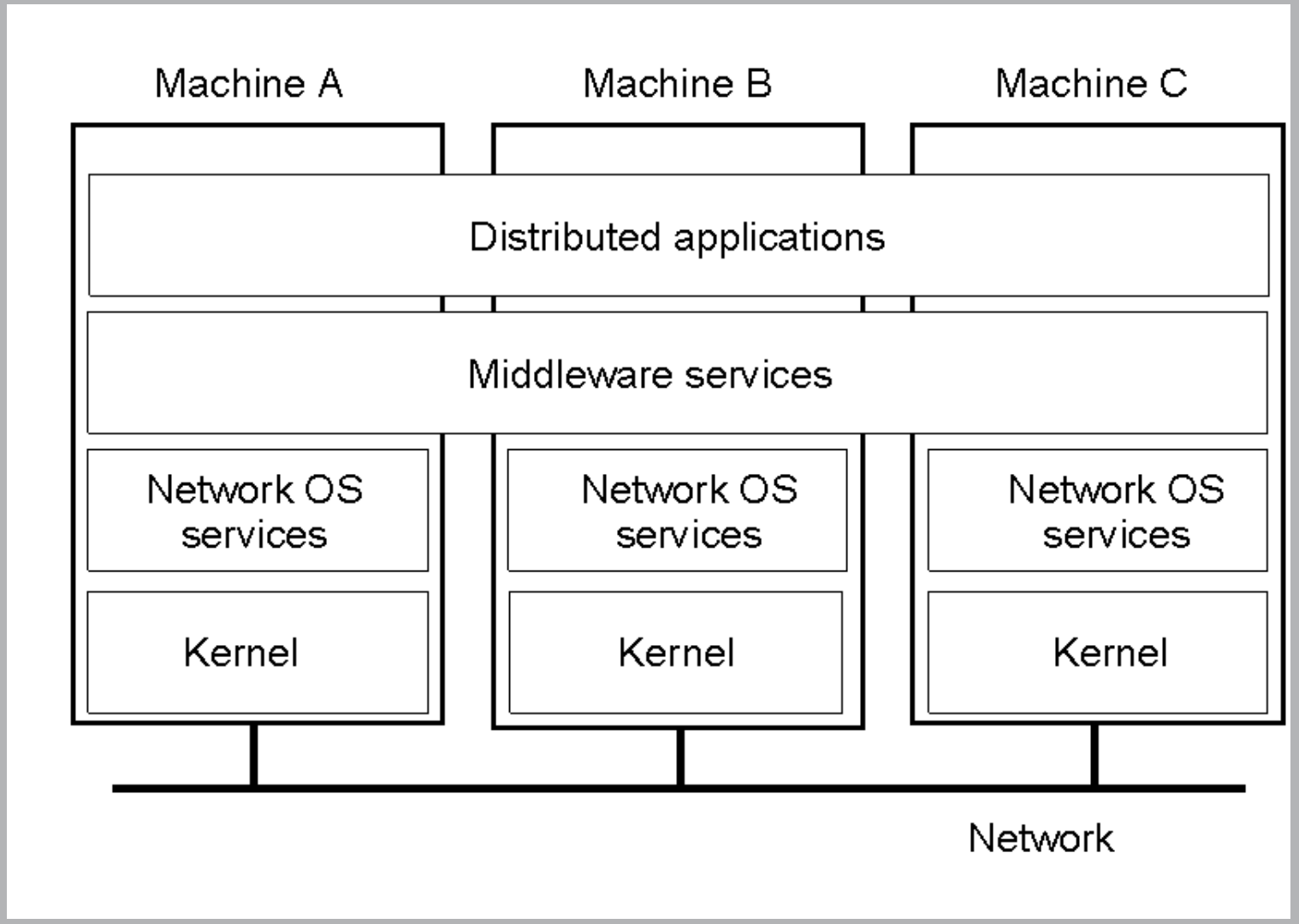
- OSes can be different (Windows or Linux)
- Typical services: rlogin, rcp
  - Fairly primitive way to share files

Traditionally, we have DOS, NOS, Middleware



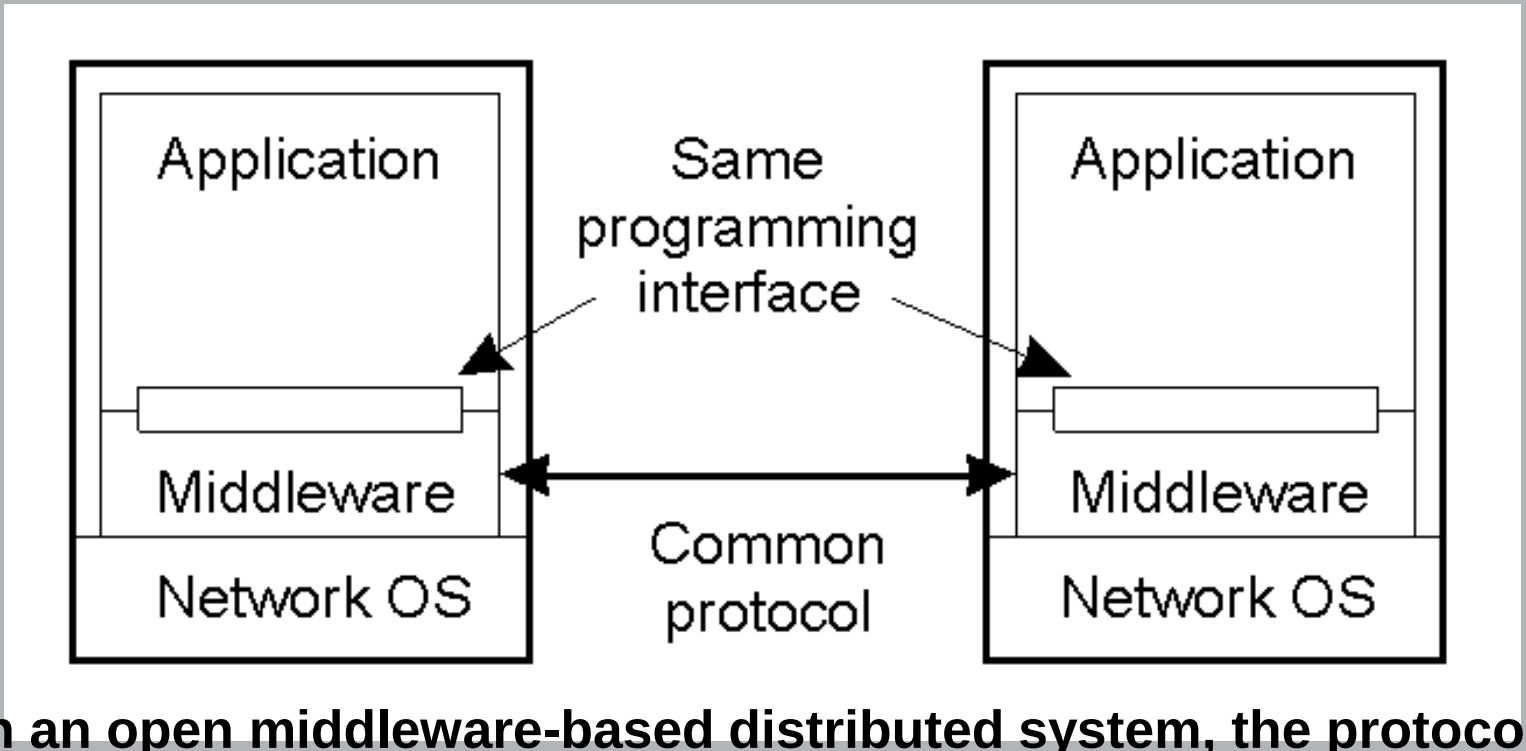
- **But no longer have shared memory**
  - Provide *message passing*
  - Can try to provide *distributed shared memory*
    - But tough to get acceptable performance

Traditionally, we have DOS, NOS, Middleware

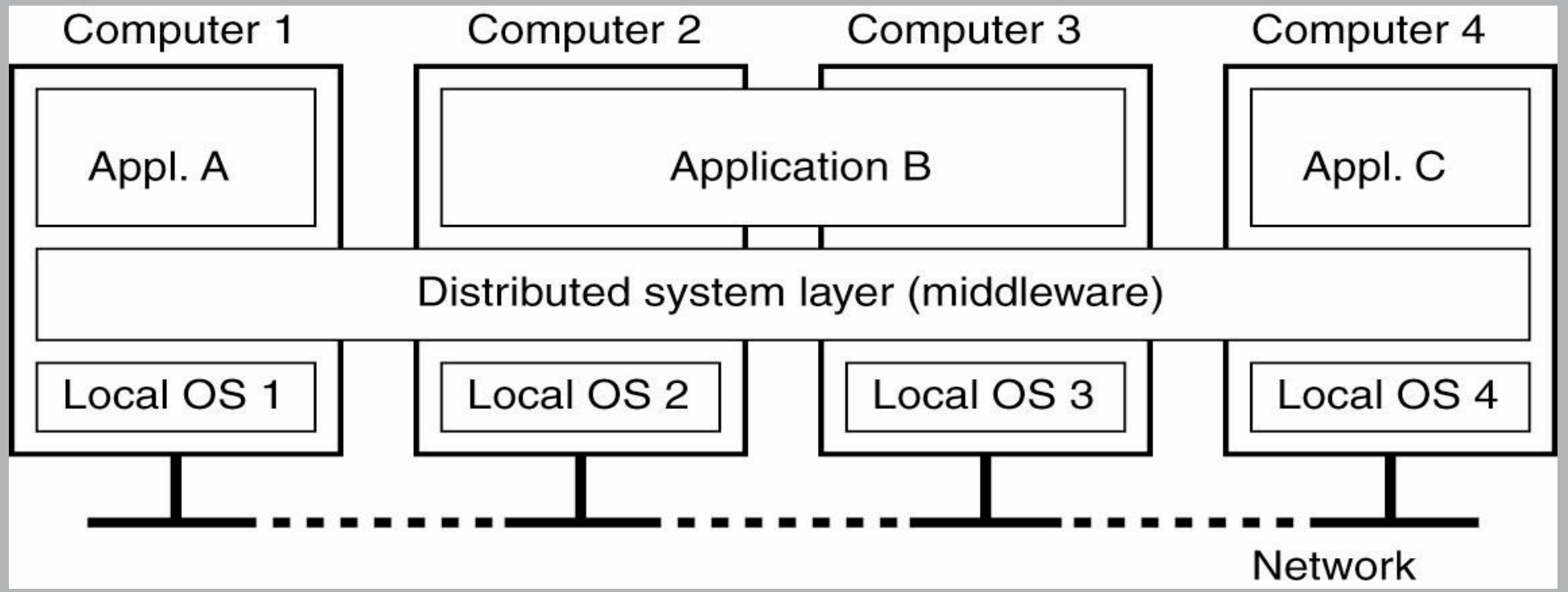


https://web.cs.wpi.edu/~rek/DCS/D04/DistributedSystems.ppt

Middleware and Openness

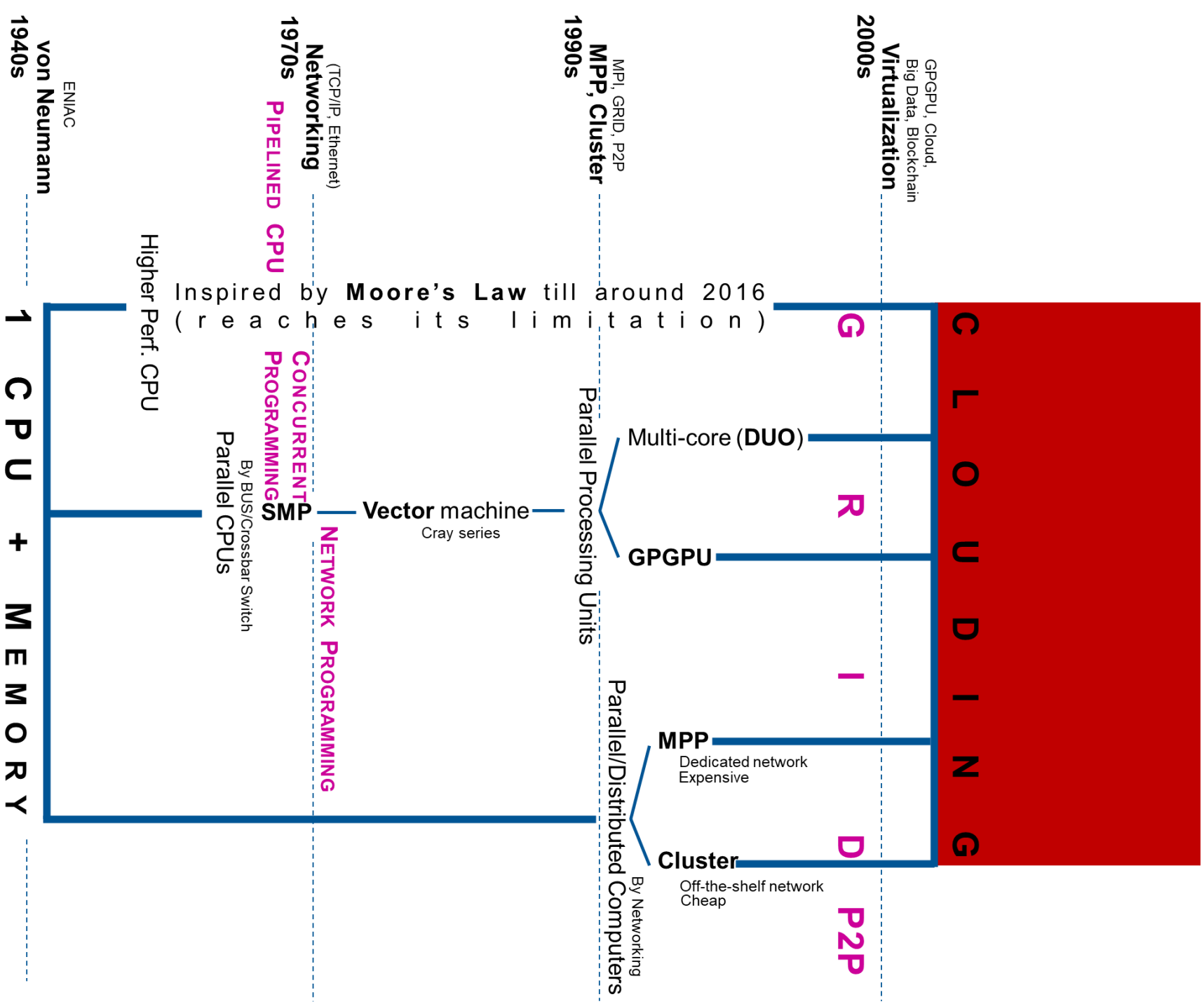


- **In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.**
  - If different, there will be compatibility issues
  - If incomplete, then users will build their own or use lower-layer services (frowned upon)


Computer 1
Computer 2
Computer 3
Computer 4
Appl. A
Application B
Appl. C
Distributed system layer (middleware)
Local OS 1
Local OS 2
Local OS 3
Local OS 4
Network

Chapter 5: Distributed OS

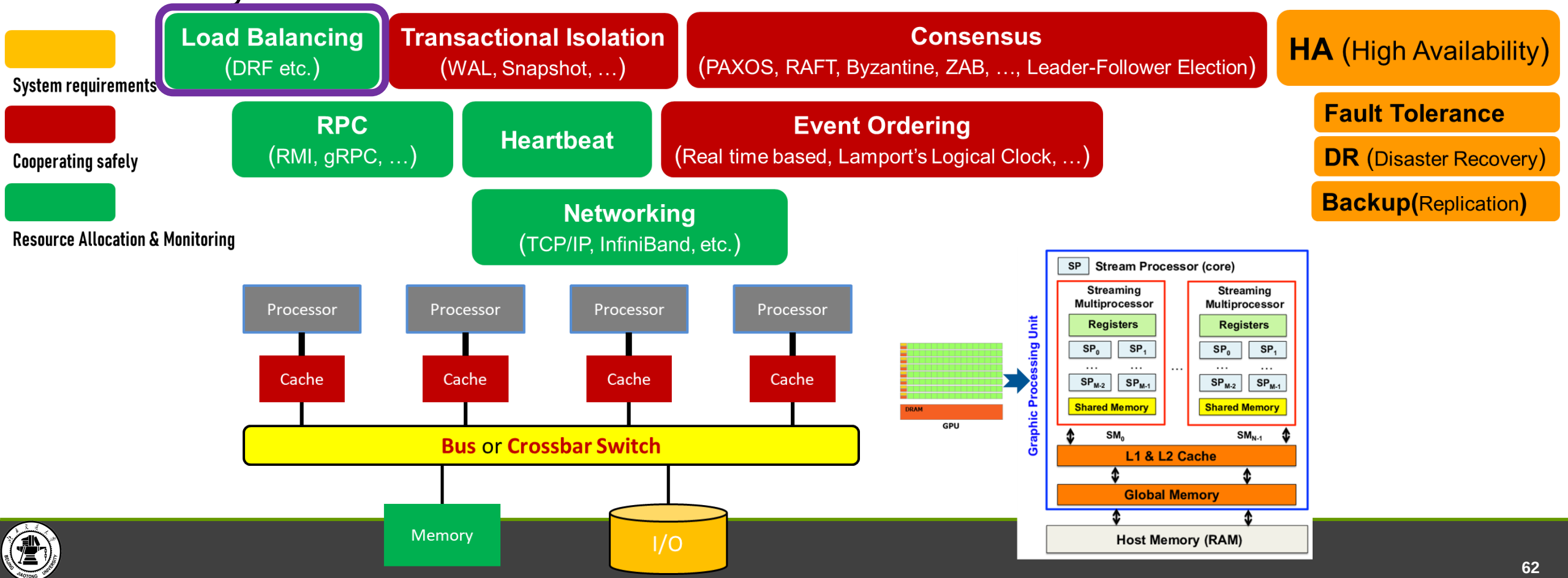
- **Support the execution of many execution units – parallel or distributed**
  - OS's Primary function is to support the execution of many (distributed) programs - **Protocols**
    - **Resource availability & Dispatching**
    - Successful cooperation – circumstance is stable or not
  - Evolution of related frameworks/platforms
    - From Amoeba [变形虫] to Micro-services [微服务]



- **OS's goal – no matter what kinds of computer systems**
  - Support the concurrent execution of many programs
- **Roles**
  - Resources manager
  - Cooperation monitor

Modern OS is OK for single computer – either multi-core or GPGPU (with GPU's SDK)

- Maybe a more module for Load balance



Computers cooperate together

- Different protocols for different systems (Storage systems, or other)

System requirements

Cooperating safely

Resource Allocation & Monitoring

Load Balancing (DRF etc.)

Transactional Isolation (WAL, Snapshot, …)

Consensus (PAXOS, RAFT, Byzantine, ZAB, …, Leader-Follower Election)

HA (High Availability)

Fault Tolerance

DR (Disaster Recovery)

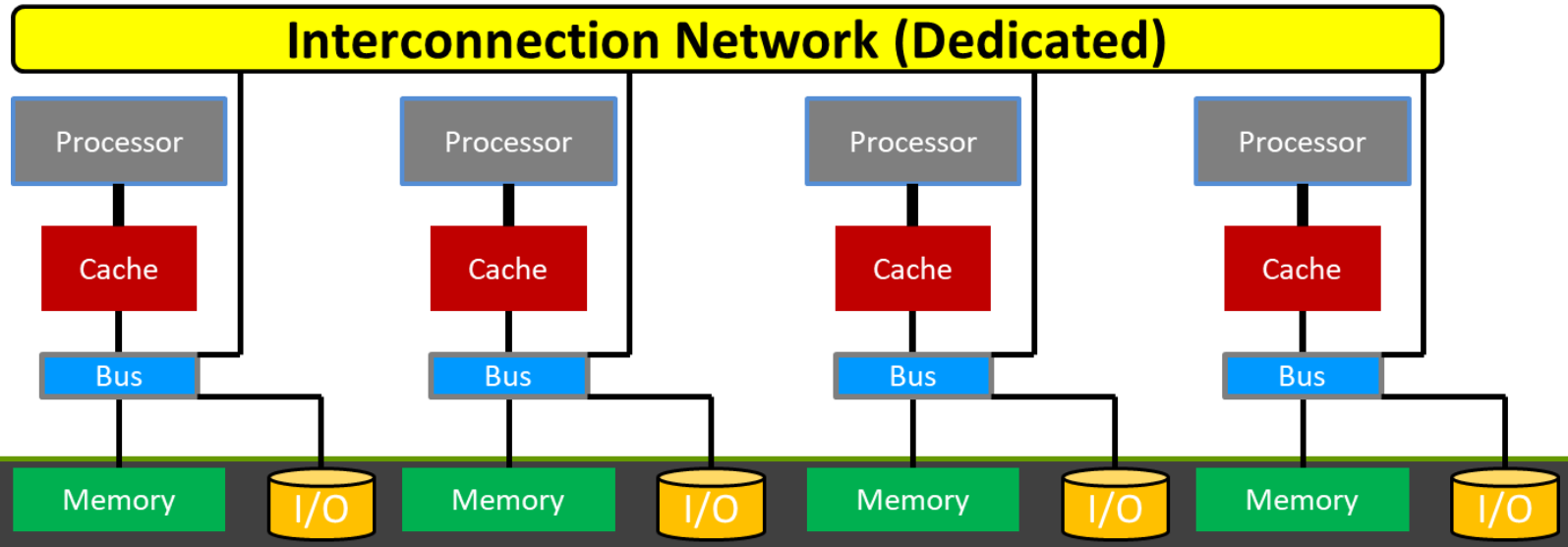
Backup(Replication)

RPC (RMI, gRPC, …)

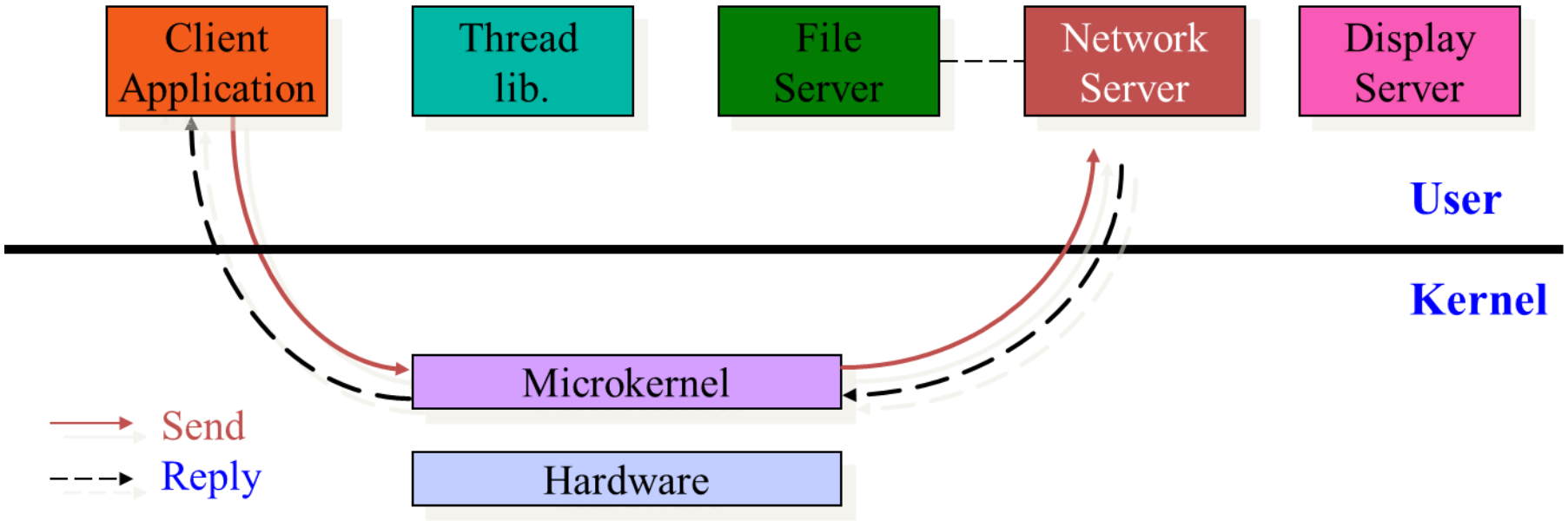
Heartbeat

Event Ordering (Real time based, Lamport's Logical Clock, …)

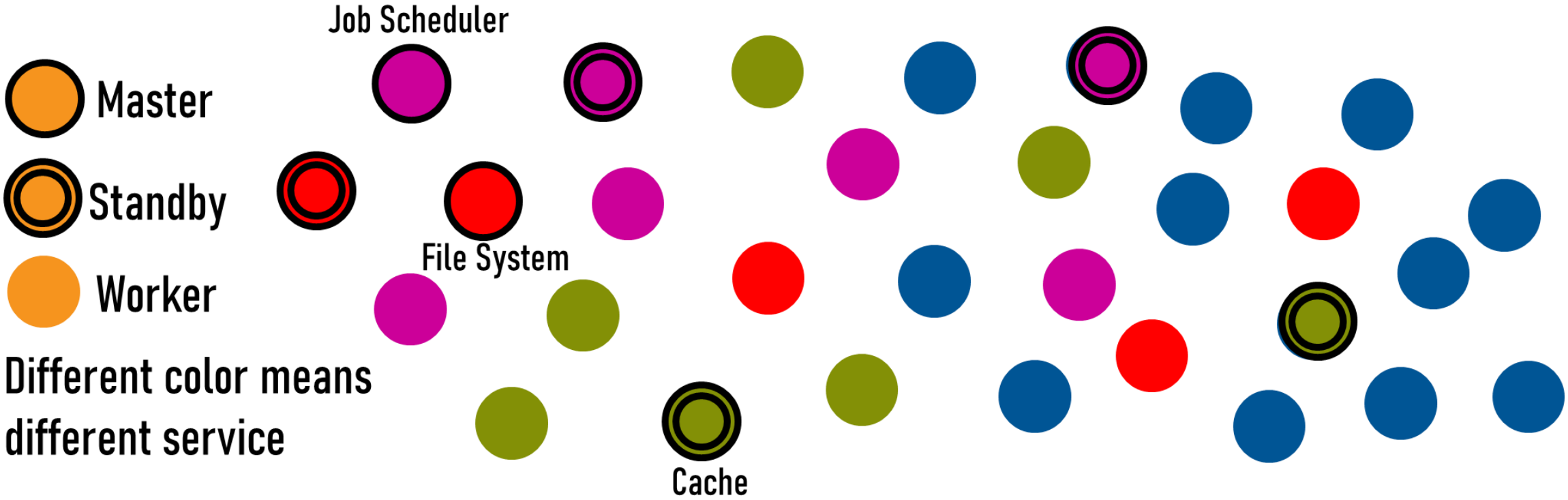
Networking (TCP/IP, InfiniBand, etc.)



- **DOS – Distributed OS? Not sure for this term.**
- **Job scheduler acts as the role of a modern OS for single computer**
  - All the other functions exist as services
  - Like Microkernel OS idea but services are distributed

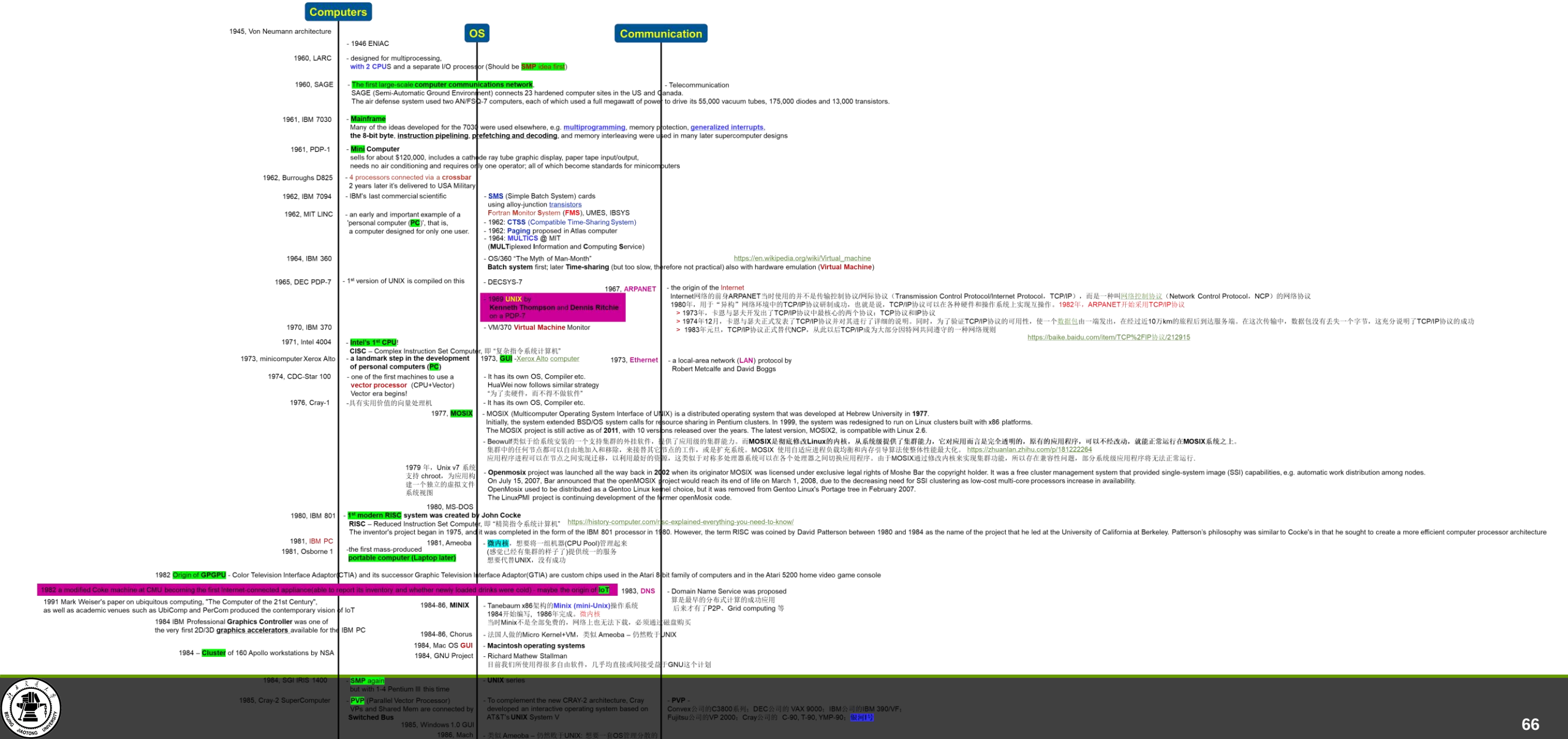


- **Usually all functions (DOS) are implemented as many programs scattered in a cluster of computers**



Attention: one node may contain several functions, such as FS's master node could also provide computing service, or is used as part of Cache

We have a big picture



Modern OS for single computer system

1962, IBM 7094 - IBM's last commercial scientific
- **SMS** (Simple Batch System) cards
using alloy-junction transistors
Fortran Monitor System (**FMS**), UMES, IBSYS

1962, MIT LINC - an early and important example of a 'personal computer (**PC**)', that is, a computer designed for only one user.
- 1962: **CTSS** (Compatible Time-Sharing System)
- 1962: **Paging** proposed in Atlas computer
- 1964: **MULTICS** @ MIT
(**MULT**iplexed **I**nformation and **C**omputing **S**ervice)

1964, IBM 360
- OS/360 "The Myth of Man-Month"
Batch system first; later **Time-sharing** (but too slow, therefore not practical) also with hardware emulation (**Virtual Machine**)

https://en.wikipedia.org/wiki/Virtual_machine

1965, DEC PDP-7 - 1st version of UNIX is compiled on this
- DECSYS-7

- 1969 **UNIX** by **Kenneth Thompson** and **Dennis Ritchie** on a PDP-7

1970, IBM 370
- VM/370 **Virtual Machine** Monitor

1981, Ameoba - 微内核，想要将一组机器(CPU Pool)管理起来
(感觉已经有集群的样子了)提供统一的服务
想要代替UNIX，没有成功

1986, Mach - 类似 Ameoba – 仍然败于UNIX: 想要一套OS管理分散的计算机，提供统一的对外服务，是走不下去了

Later

1991, **Linux**
- Linus Torvalds
以bash, gcc等工具写了一个小小的核心程序,
可以在Intel的386机器上面运行
1994年，发布了Linux的1.0版本
发布伊始就加入GNU，被广泛应用
现在已经成为服务器系统的de facto OS

- Mac OS, Windows, Linux, Android, EulerOS, 龙蜥

2021 年10 月 20 日，2021 云栖大会上，阿里云发布全新操作系统 "龙蜥" 并宣布开源。同时，阿里达摩院操作系统实验室也宣告成立龙蜥操作系统定位于服务器端，支持 x86、ARM 等多种芯片架构和计算场景，兼容 CentOS 生态，支持一键迁移，并提供全栈国密能力
> 其实，阿里做操作系统也很久了，但都失败了 – 如 所谓的 AliOS 云操作系统

Job schedulers

1989, **PVM** - Parallel Virtual Machine
为了支持多机器协作完成大计算任务的
设想，在一套微内核统一服务走不通后，
转而扩展已有OS的协同功能

1991, **PBS** - Portable Batch System as job scheduler for UNIX Cluster
allocate computational tasks, i.e., batch jobs, among the available computing resources, which is often used in conjunction with UN
3 current versions - OpenPBS, TORQUE and PBS Professional (PBS Pro)

1992, **LSF** - Load Sharing Facility
IBM Spectrum LSF is a workload management platform,
job scheduler, for distributed **HPC by IBM**,
based on the *Utopia* research project
"Utopia: a Load Sharing Facility for Large, **Heterogeneous Distributed Computer Systems**"

2000, **SGE** - Sun Grid Engine
Sun acquired Gridware a privately owned commercial vendor of advanced computing resource management softwa
Later that year, Sun offered a free version of Gridware for Solaris and Linux, and renamed the product **S**un **G**rid **E**ng

2003, **SLURM** - *SLURM: Simple Linux Utility for Resource Management*

2003~2004, **Google's Borg** - Paper published in 2015 "Large-scale cluster management at Google with Borg" 支持 MPI、MR等
Google 的 第一代 (Monolithic - 中央式) 集群管理 **Borg**

1996, **Globus, SOA** - Service Oriented Architecture:一个组件模型，
将应用程序的不同功能单元（称为服务）进行拆分，
并通过这些服务之间定义良好的接口和协议联系起来
Globus is an easy-to-use, high-performance data transfer to

2012, YARN
- 2012 年 8 月 Apache Hadoop YARN 成为 Apache Hadoop 的一个新子项目，
提供了一个更加通用的资源管理和分布式应用框架，不仅支持 Hadoop MapReduce，而且支持 **MPI**，图处理，在线服务（比如 Spark、Storm、HBase）等

2013, **Google's Omega**
- Omega: flexible, scalable schedulers for large compute clusters. SIGOPS European Conference on Computer Systems (EuroSys), ACM, Prague, Czech Republic (2013), pp.
共享状态式 (Shared-state)，是Borg的延伸。但因为 Docker 的火热，很快就有了 K8s

2014, Docker Swarm
- Docker Swarm 是 Docker 公司在 2014 年 12 月初发布的一套管理 **Docker 集群**的工具

2014, **Google's Kubernetes (K8s)**
- 第二代 双层式 (Two-level)
It works with a range of container tools and runs containers in a cluster, often with images built using Docker.
Many cloud services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) on which Kubernetes can be deployed as a platform-providing service.

腾讯, 2009-至今
- 2009 的 T-Borg
2011 的 Torca (支持大数据)
2014 的 GaiaStack (迎合 Docker，基于开源的K8s)
2019 的 TKEStack (混合， K8s 编程了其中的一种底层应用实例)，于2020年开源

阿里, 2010-至今
- 2010年伏羲 (最初创立飞天平台时的三大服务之一: 分布式存储 Pangu，分布式计算 MaxCompute，分布式调度 Fuxi),对标开源系统的YARN
2013年伏羲在飞天5K项目中对系统架构进行了第一次大重构，解决了规模、性能、利用率、容错等线上问题，并取得世界排序大赛Sortbenchmark四项冠军，这标志着Fuxi 1.0的成
2019年Yugong 发布并论文被VLDB接受

华为, 2020推出 Donau (多瑙)
- 支持多场景统一调度，解决了多样性计算环境下，融合应用开发部署难的问题，同时，在集群规模、调度性能、资源利用率等核心调度指标方面实现了竞争力突破，业界领先

华为, 2020华为正式开源数据虚拟化引擎openLooKeng

Overview

→ Amoeba – 1981

IPC → RPC 1985

→ ORB (CORBA) 1991

→ MPI 1992

→ Cluster Beowulf 1994

→GRID Globus 1998

→ Linux SLURM 2003 (Linux Cluster Project)

→ Google's Borg 2003~2004

→ Big Data's

→ MR (2004)

→ Mesos (2009 C++), YARN (2012 Java 77MB),

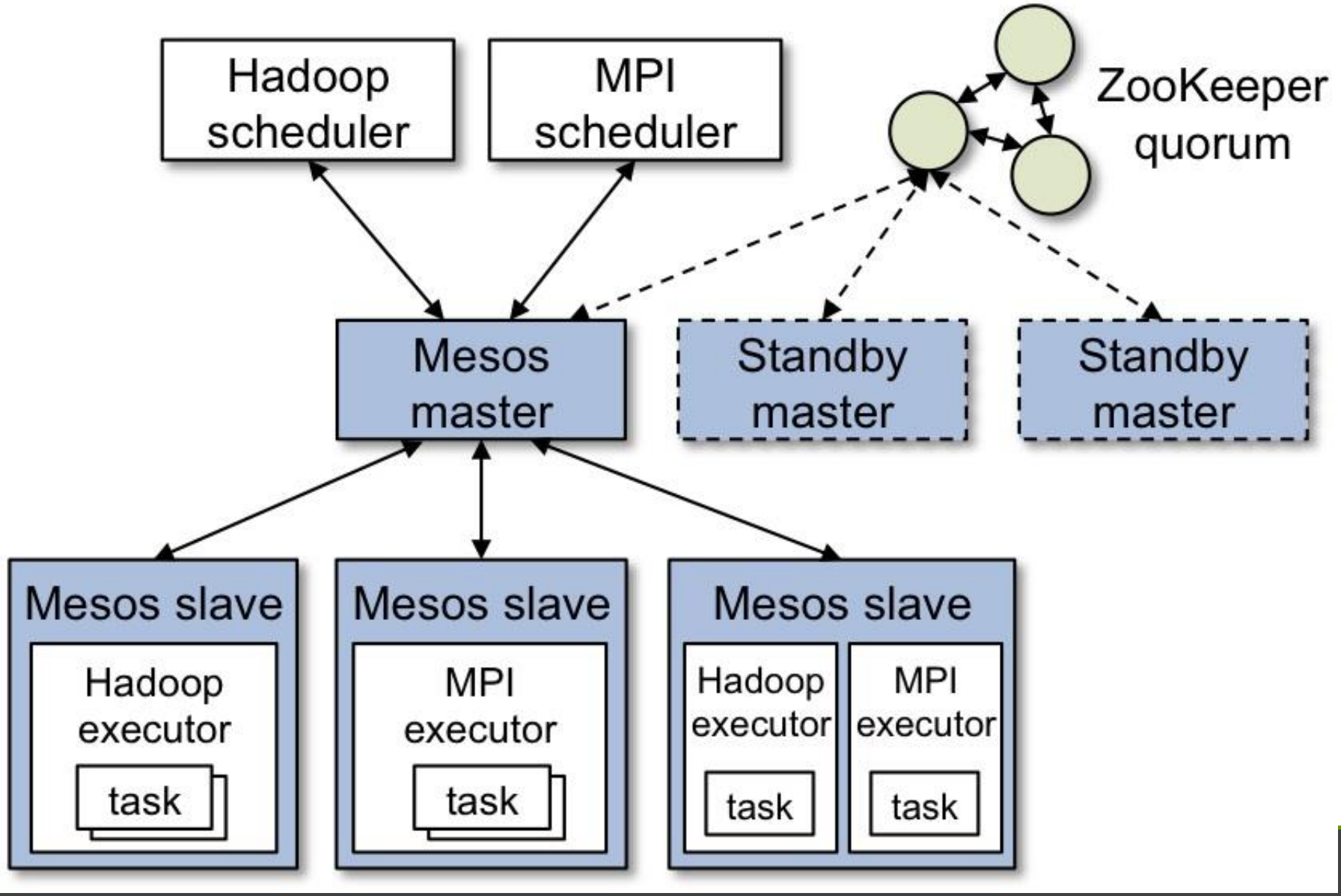
→ Kubernetes 2015 Go 40MB

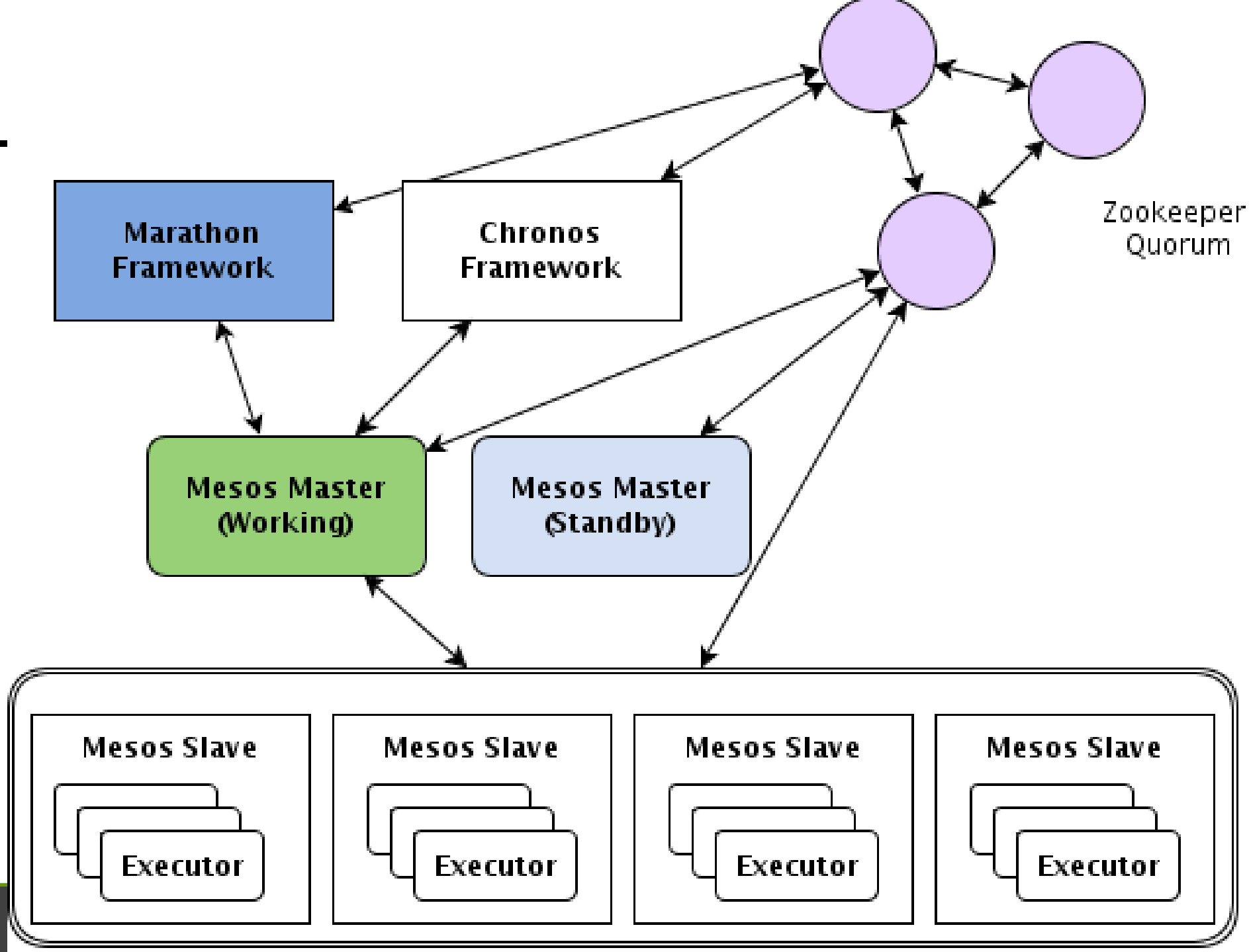
→ Swarm with Docker 2016 Go 2MB

→ Micro Service (微服务) 2016

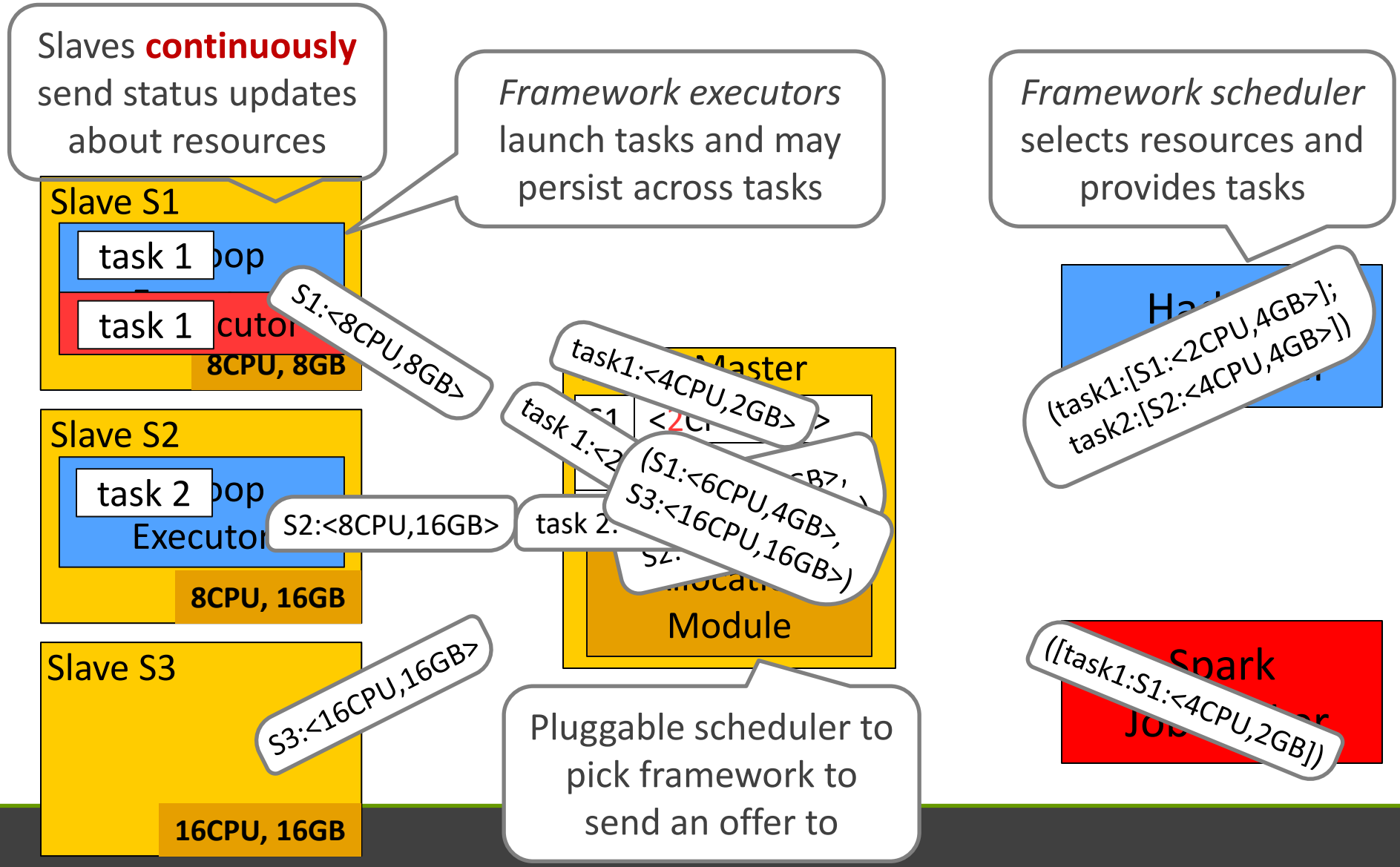
https://www.cnblogs.com/ee900222/p/docker_2.html

Mesos – 2009 (while still named Nexus)



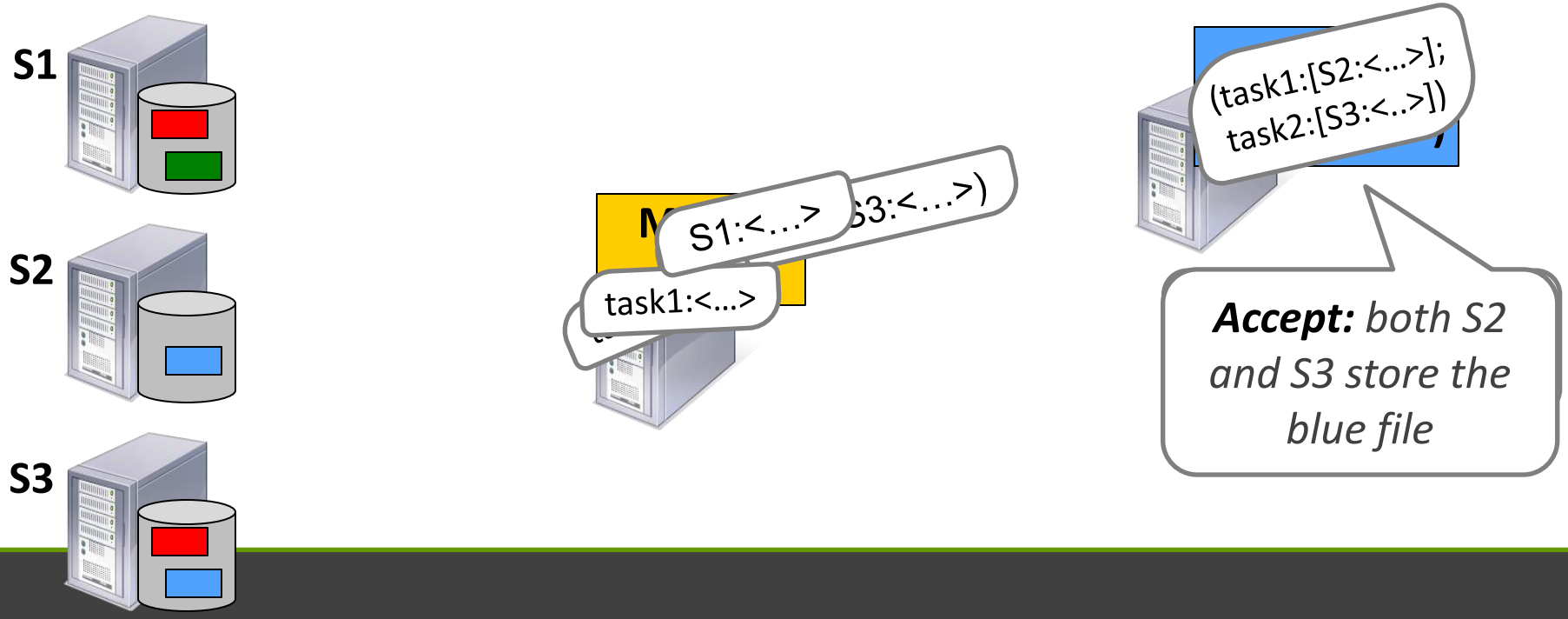

Marathon Framework
Chronos Framework
Zookeeper Quorum
Mesos Master (Working)
Mesos Master (Standby)
Mesos Slave
Executor
Mesos Slave
Executor
Mesos Slave
Executor
Mesos Slave
Executor

Mesos Architecture: Example



Why does it Work?

- A framework can wait for offer that matches its constraints or preferences, reject otherwise
- Example: Hadoop's job input is *blue* file

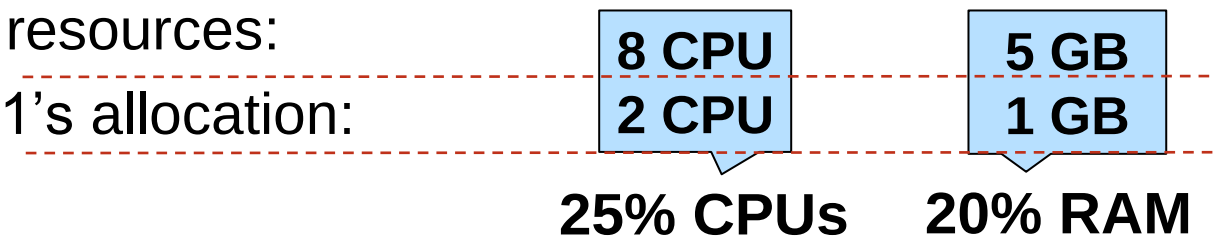


Two Key Questions

- **How long does a framework need to wait?**
  - Depends on distribution of task duration
  - "Pickiness" of framework given hard/soft constraints
- **How allocate resources of different types?**
  - Use **DRF**!

Dominant Resource Fairness (DRF)

- **A user's *dominant resource* is resource user has biggest share of**
  - Example:



Dominant resource of User 1 is CPU (as **25%** > 20%)

- **A user's *dominant share:* fraction of dominant resource allocated**
  - User 1's dominant share is **25%**

Dominant Resource Fairness: Fair Allocation of Multiple Resource Types
Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, Ion Stoica, NSDI'11

An Example

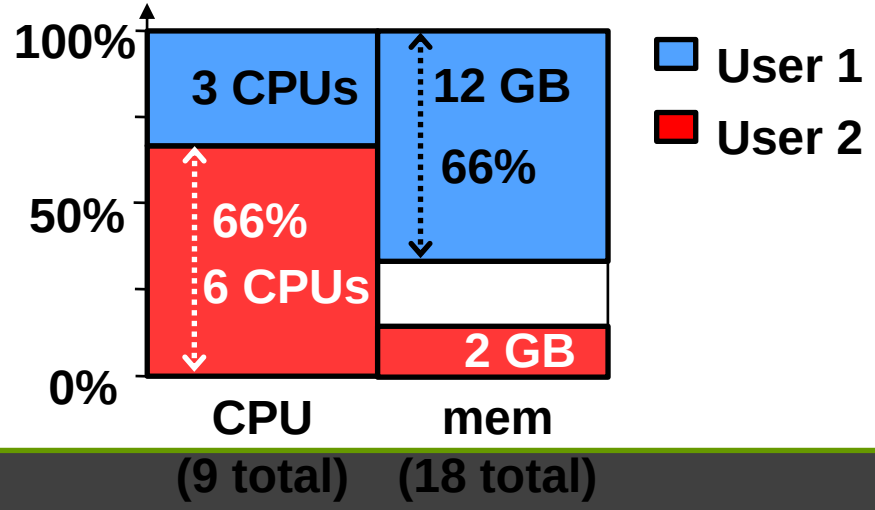
- **a system with of 9 CPUs, 18 GB RAM**
- **Two users, where**
  - user A runs tasks with demand vector <1 CPU, 4 GB>
  - and user B runs tasks with demand vector <3 CPUs, 1 GB> each.

- **Dominant share:**
  A:2/9 (memory) B:1/3 (CPU)

Dominant Resource Fairness (DRF)

- **Apply max-min fairness to dominant shares**

- **Equalize the dominant share of the users. Example:**
  - Total resources: **<9 CPU, 18 GB>**
  - User 1 demand: **<1 CPU, 4 GB>;** dom res: **mem** (1/9 < 4/18)
  - User 2 demand: **<3 CPU, 1 GB>;** dom res: **CPU** (3/9 > 1/18)



- With this allocation, each user ends up with the same dominant share, i.e., user A gets 2/3 of RAM, while user B gets 2/3 of the CPUs.
- This allocation can be computed mathematically as follows.
- Let x and y be the number of tasks allocated by DRF to users A and B

$$
\begin{aligned}
&\max(x, y) && \text{(Maximize allocations)}\\
&\text{subject to}\\
&x + 3y \leq 9 && \text{(CPU constraint)}\\
&4x + y \leq 18 && \text{(Memory constraint)}\\
&\frac{2x}{9} = \frac{y}{3} && \text{(Equalize dominant shares)}
\end{aligned}
$$

https://www.cnblogs.com/yuananyun/p/5137321.html

考虑一个有9个cpu和18GB的系统，有两个用户：用户A的每个任务都请求（1CPU，4GB）资源；用户B的每个任务都请求（3CPU，1GB）资源。如何为这种情况构建一个公平分配策略?

对于用户A，每个任务需要消耗的资源为<1/9,4/18>=<1/9,2/9>,所以A的dominant shares为内存，比例为2/9

对于用户B，每个任务需要消耗的资源为<3/9,1/18>=<1/3,1/18>,所以B的dominant shares为cpu，比例为1/3

$$\max(x, y) \quad \text{(Maximize allocations)}$$

subject to

$$x + 3y \leq 9 \quad \text{(CPU constraint)}$$

$$4x + y \leq 18 \quad \text{(Memory constraint)}$$

$$\frac{2x}{9} = \frac{y}{3} \quad \text{(Equalize dominant shares)}$$

Solving this problem yields[2] $x = 3$ and $y = 2$. Thus, user A gets $\langle 3\text{ CPU}, 12\text{ GB}\rangle$ and B gets $\langle 6\text{ CPU}, 2\text{ GB}\rangle$.

通过列不等式方程可以解得给用户A分配3份资源，用户B分配2份资源是一个很好的选择。

https://www.cnblogs.com/yuananyun/p/5137321.html

Algorithm 1 DRF pseudo-code

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ consumed resources, initially 0
$s_i \ \ (i = 1..n)$ ▷ user i's dominant shares, initially 0
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ \ (i = 1..n)$ ▷ resources given to user i, initially 0

pick user i with lowest dominant share s_i
$D_i \leftarrow$ demand of user i's next task
if $C + D_i \leq R$ **then**
    $C = C + D_i$ ▷ update consumed vector
    $U_i = U_i + D_i$ ▷ update i's allocation vector
    $s_i = \max_{j=1}^{m} \{u_{i,j}/r_j\}$
else
    return ▷ the cluster is full
end if

Kubernetes – 2015

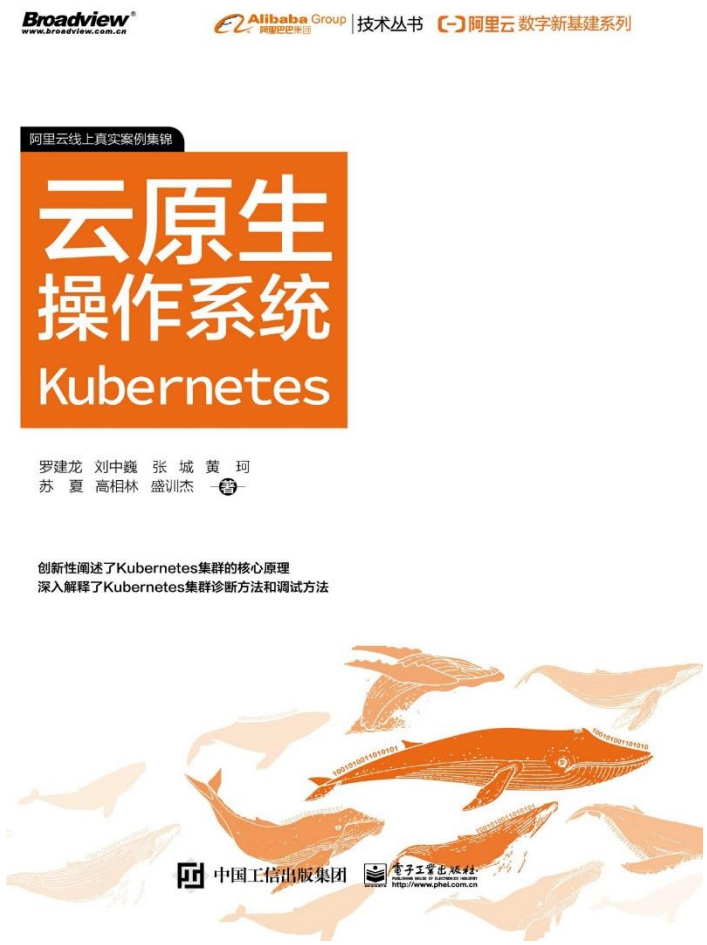


图0-2是传统操作系统（单机操作系统）和云原生操作系统（集群操作系统）的比较图。大家知道，像Linux或者Windows这些传统的操作系统，它们扮演的角色是底层硬件的抽象层。它们向下管理计算机硬件，如内存和CPU，然后把底层硬件抽象成易用的接口，向上对应用层提供支持。

而Kubernetes技术，我们也可以理解为一个操作系统。这个操作系统也是一个抽象层。它向下管理的硬件，不是内存或者CPU这种硬件，而是多台计算机组成的集群。这些计算机本身就是普通的单机系统，有自己的操作系统和硬件。Kubernetes把这些计算机当成一个资源池统一管理，向上对应用层提供支撑。

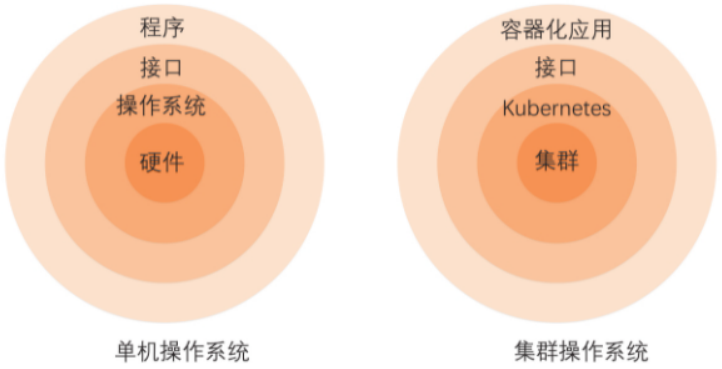
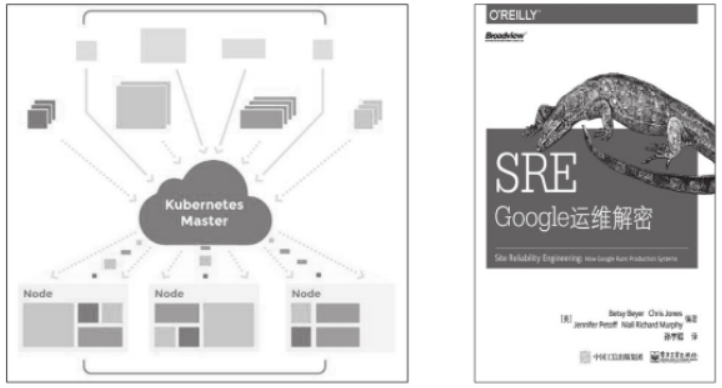


图0-2 传统操作系统与云原生操作系统

Kubernetes上的应用的特别之处在于，它们都是容器化应用。对容器不太了解的读者，可以简单地把它们理解成安装包。安装包里包括了所有的依赖库，如libc函数库等，使得这些应用不必依赖底层操作系统库文件就可以直接运行。

第三个角度，Kubernetes与Google运维解密

图0-3的左边是Kubernetes集群示意图，右边是一本非常有名的书《SRE Google运维解密》。相信很多人都看过这本书，而且有很多公司正在实践这本书里的方法，如故障管理、运维排班等。



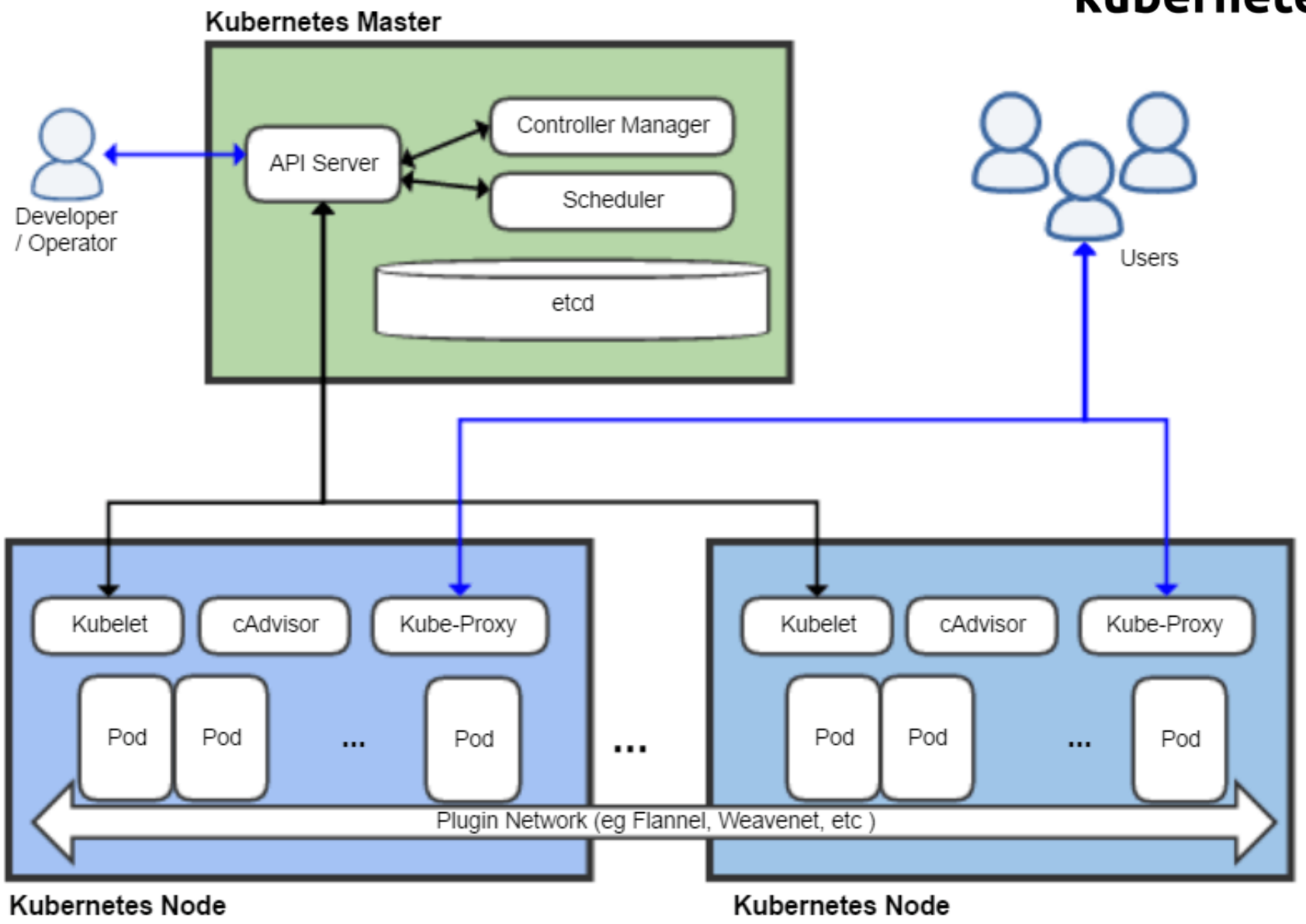
“剑法” “气功”

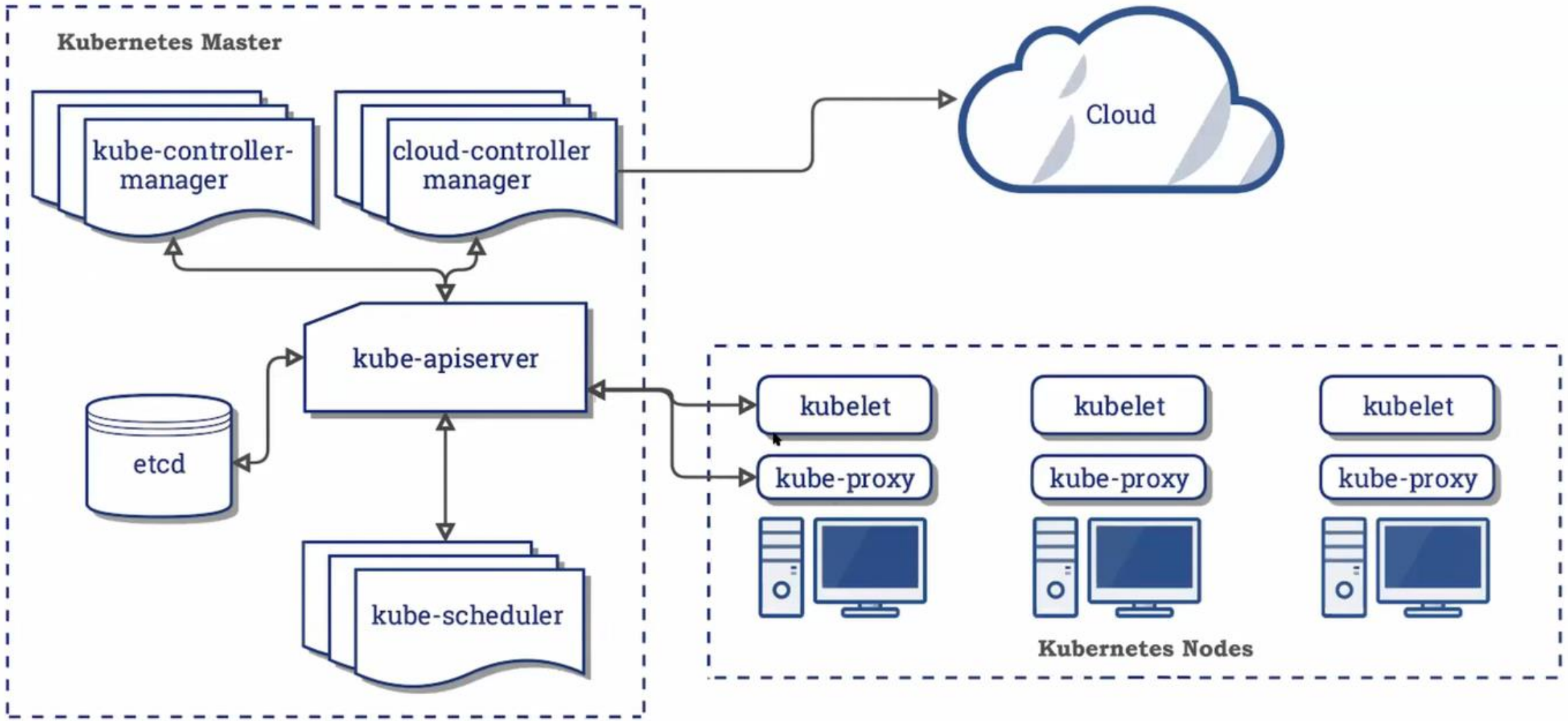
图0-3 Kubernetes集群与《SRE Google运维解密》

Kubernetes和这本书的关系，可以比作剑法和气功的关系。读者看过金庸的《笑傲江湖》的话，可能会记得，《笑傲江湖》里的华山派分为两个派别，剑宗和气宗。

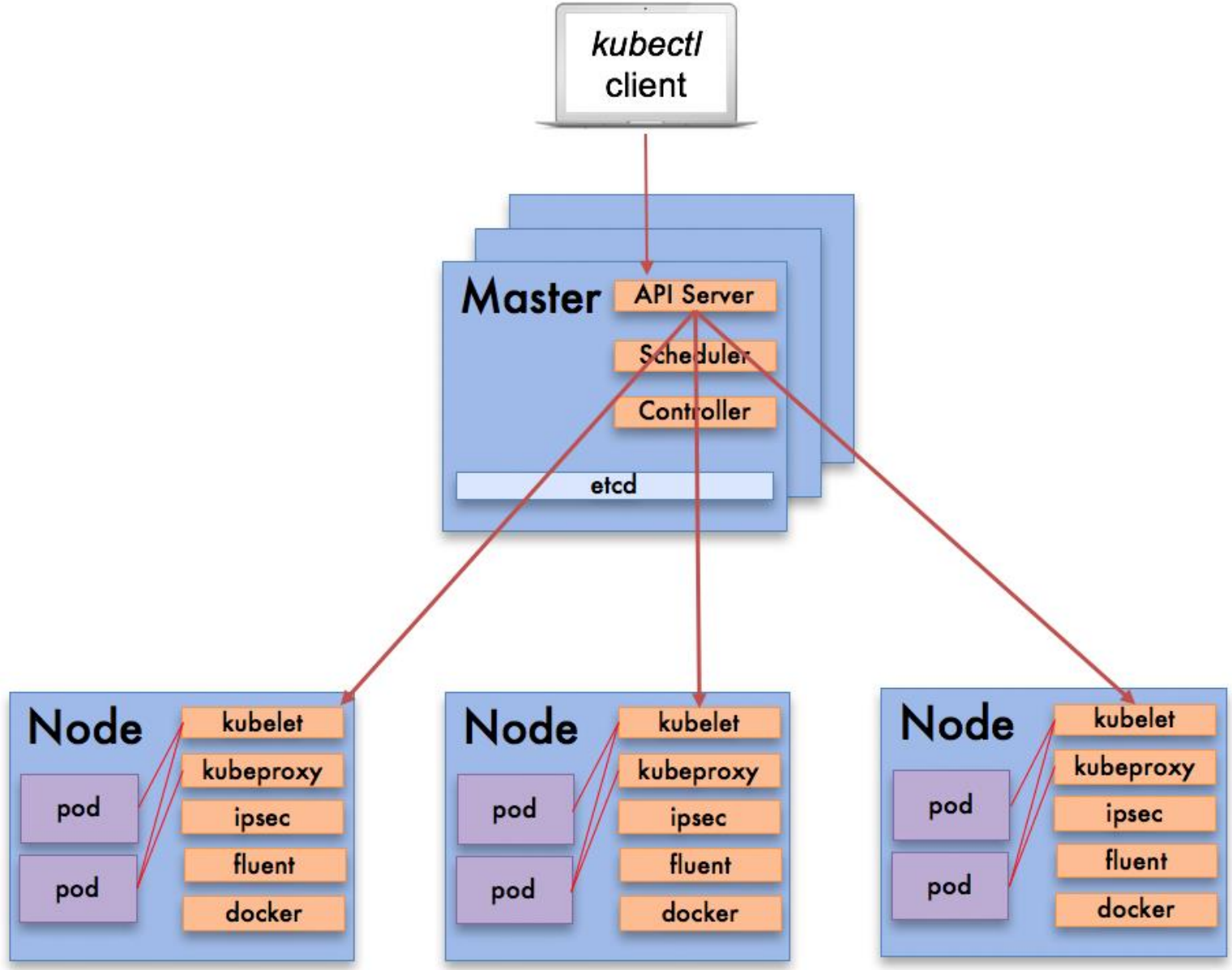
剑宗强调剑法的精妙，而气宗更注重气功修炼。实际上剑宗和气宗的“分家”，是因为华山派两个弟子偷学了同一本《葵花宝典》，但是两个人各记了一部分内容，最终因为观点分歧分成了两派。

Kubernetes实际上源自Google的集群自动化管理和调度系统Borg，也就是这本书里讲的运维方法所管理的对象。Borg系统和书


Kubernetes Master
Developer
/ Operator
API Server
Controller Manager
Scheduler
etcd
Users
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
...
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
Plugin Network (eg Flannel, Weavenet, etc)
Kubernetes Node
Kubernetes Node

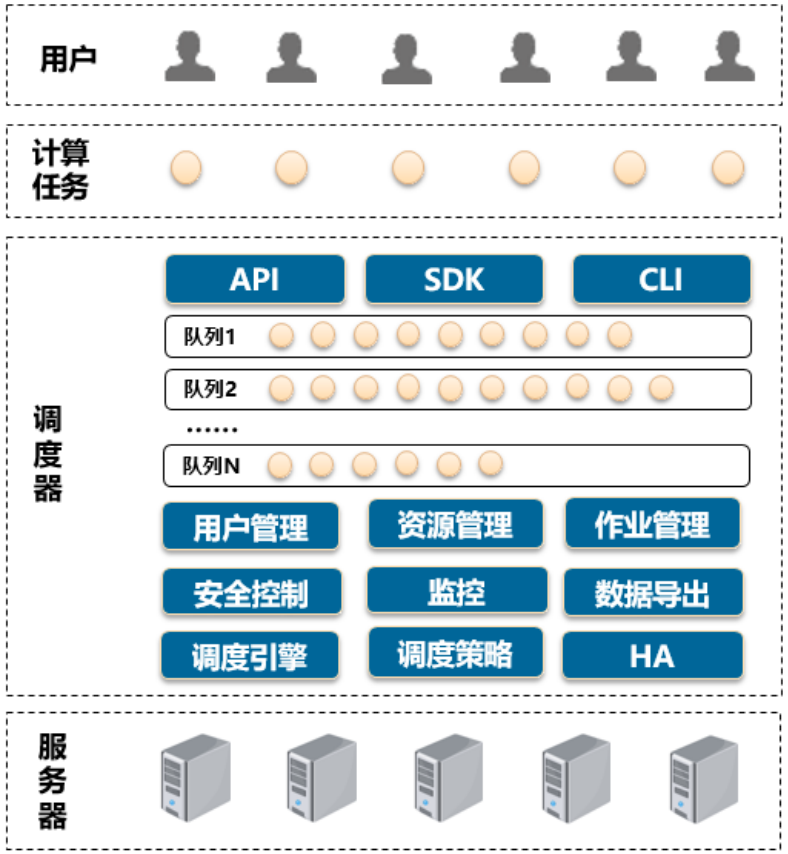

Kubernetes Master
kube-controller-manager
cloud-controller manager
Cloud
kube-apiserver
etcd
kube-scheduler
kubelet
kube-proxy
kubelet
kube-proxy
kubelet
kube-proxy
Kubernetes Nodes

https://www.dasher.com/containers-os-virtualization-to-workload-virtualization/



HuaWei Donau

华为自研调度器 Donau



特别鸣谢：计算技术开发部(云与计算BG)

吞吐量高

调度5K作业/秒
完成4M作业/小时

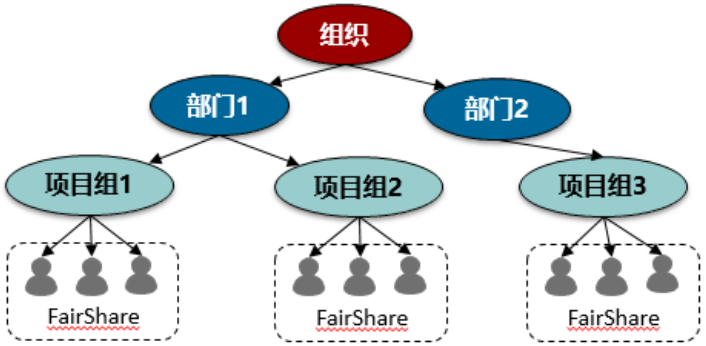
资源利用率高

资源利用率
90%+

扩展性强

最大集群规模
2w节点+

多层级 FairShare 资源公平调度策略



1、部门/组间自定义资源使用策略（配额、借用、预留、归还），支持资源互借，提高资源利用率；

2、组内用户保证公平使用资源；

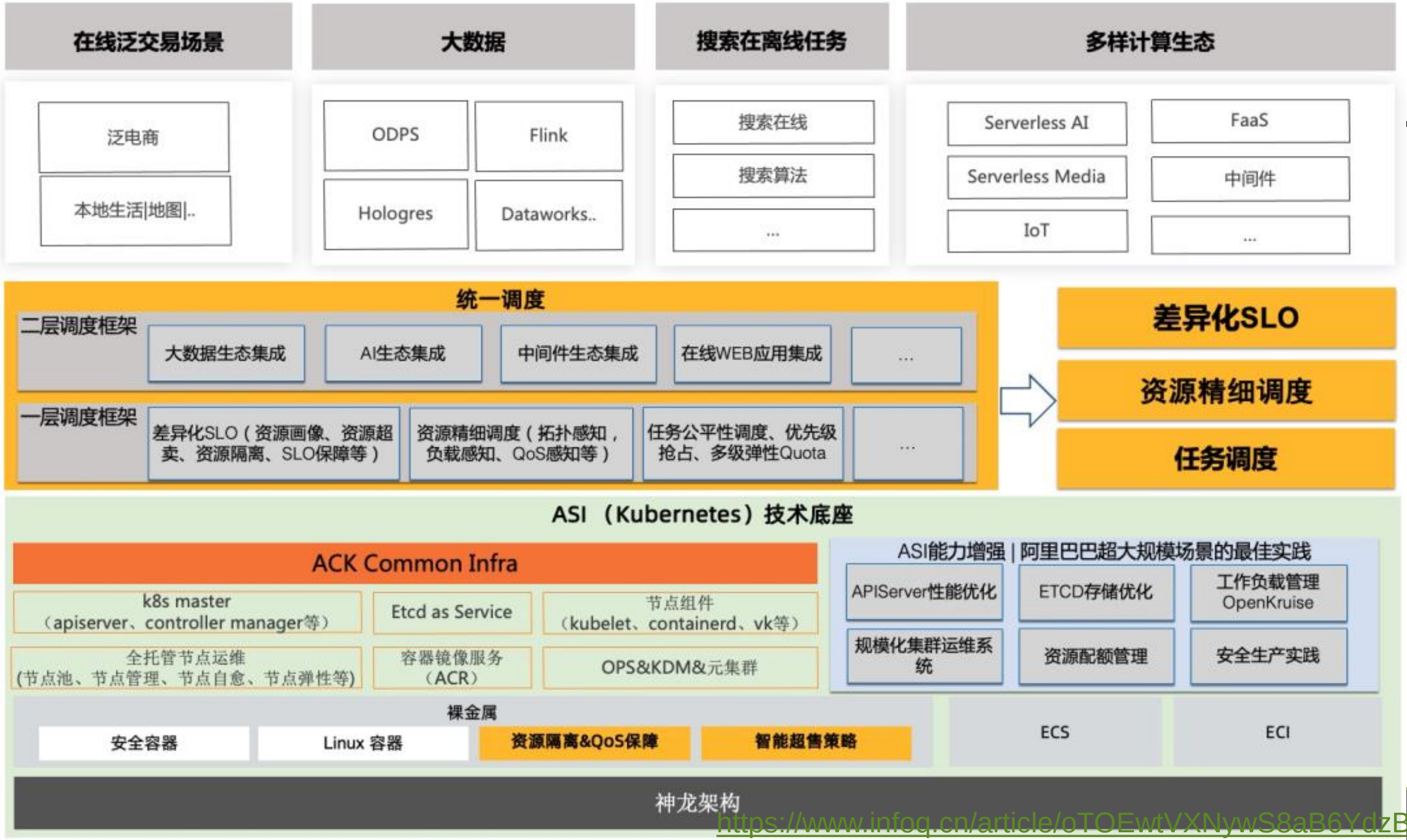
多样化负载支持

MPI Array WorkFlow HUAWEI

Alibaba ACK Anywhere

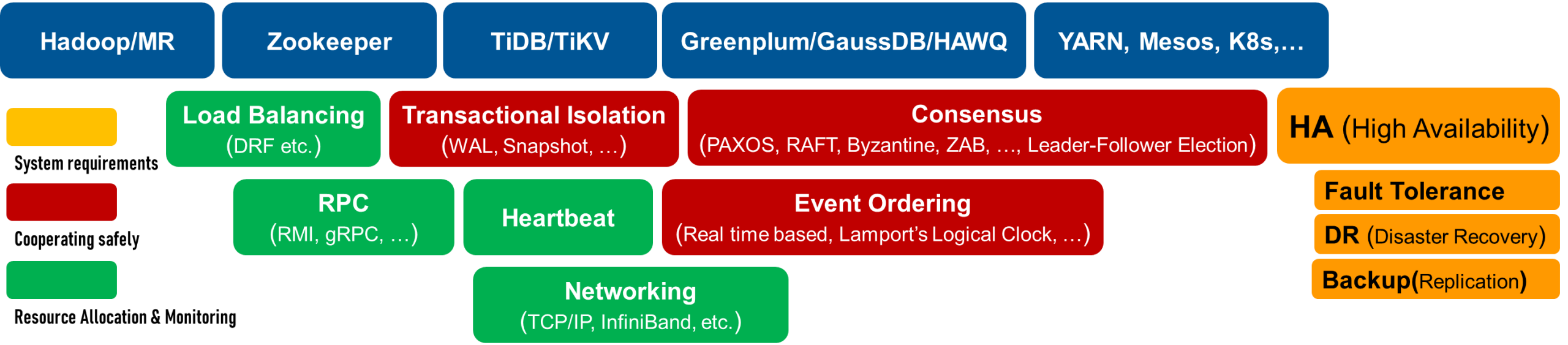
“刚开始是比较艰难的，尝试过好多版本，包括 Sigma on Kubernetes、Kubernetes on Sigma 等方式，最后还是决定用最标准、最原生的、完全基于 Kubernetes 的方式。”后面启动的 ASI 项目，它做的事情就是将整个调度框架以非常原生的标准方式搬到 Kubernetes 上，在 Kubernetes 基础上做到在线、离线调度的真正融合。而且在业务侧，阿里也专门组织了一支云原生团队来推进容器化，最终形成一个整体的云原生资源池。

https://www.infoq.cn/article/oTOEwtVXNywS8aB6YdzB



https://www.infoq.cn/article/oTOEwtVXNywS8aB6YdzB

Summary


Hadoop/MR
Zookeeper
TiDB/TiKV
Greenplum/GaussDB/HAWQ
YARN, Mesos, K8s,…
System requirements
Load Balancing
(DRF etc.)
Transactional Isolation
(WAL, Snapshot, …)
Consensus
(PAXOS, RAFT, Byzantine, ZAB, …, Leader-Follower Election)
HA (High Availability)
Cooperating safely
RPC
(RMI, gRPC, …)
Heartbeat
Event Ordering
(Real time based, Lamport's Logical Clock, …)
Fault Tolerance
DR (Disaster Recovery)
Resource Allocation & Monitoring
Networking
(TCP/IP, InfiniBand, etc.)
Backup(Replication)