中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

Lecture 13：云存储

SSE316：云计算技术
Cloud Computing Technologies

陈壮彬

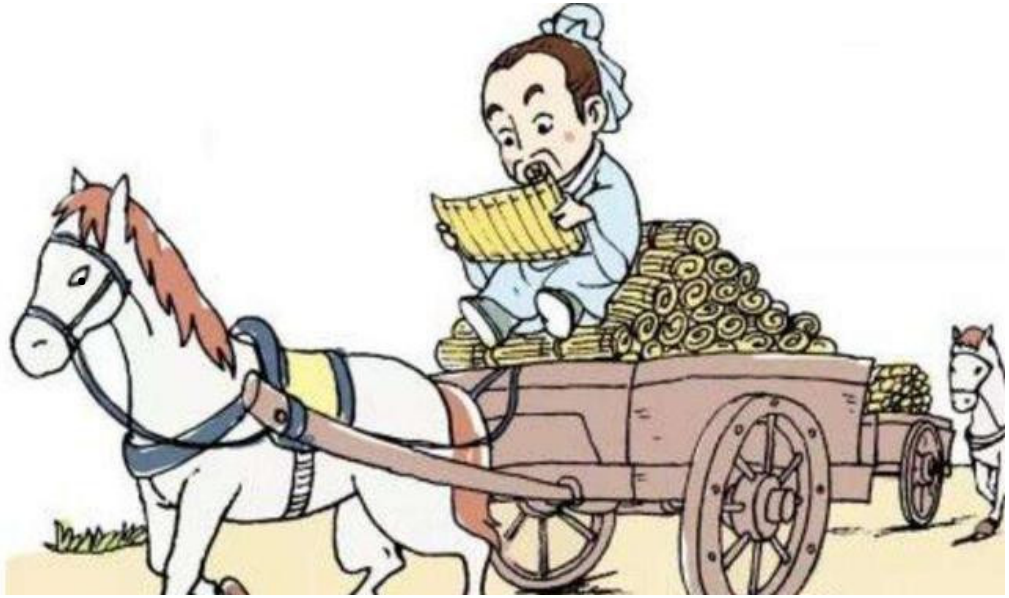
软件工程学院

chenzhb36@mail.sysu.edu.cn

Today’s topics

- 云存储介绍
- 数据存储模型
- 分布式系统中的共识问题
- 数据可靠性

竹简刻字



曾考证过，3000 多片竹简能写大约 10 万字，重量约 12 公斤，而古代马车的载重约 200 公斤，五车竹简就是 1000 公斤。因此，

“学富五车” ≈ 800万字

https://new.qq.com/rain/a/20201109A0DJ9F00

蔡伦造纸

学富五车 = 

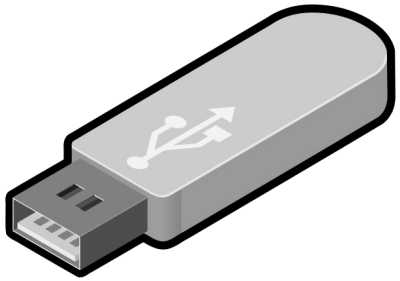
×10?

89.7万字

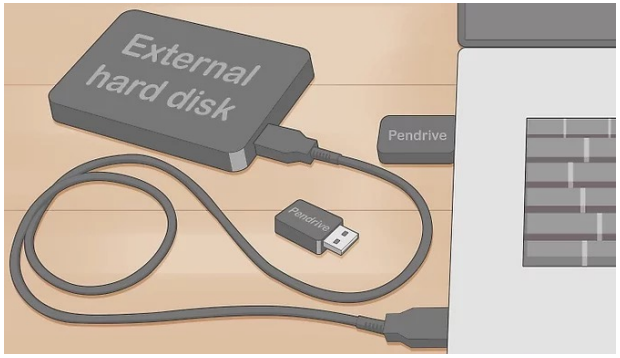
CD-ROM

比尔·盖茨在 1994 年展示了一张 CD-ROM 可以容纳比一堆纸 (33 万张) 更多的信息 (700MB)

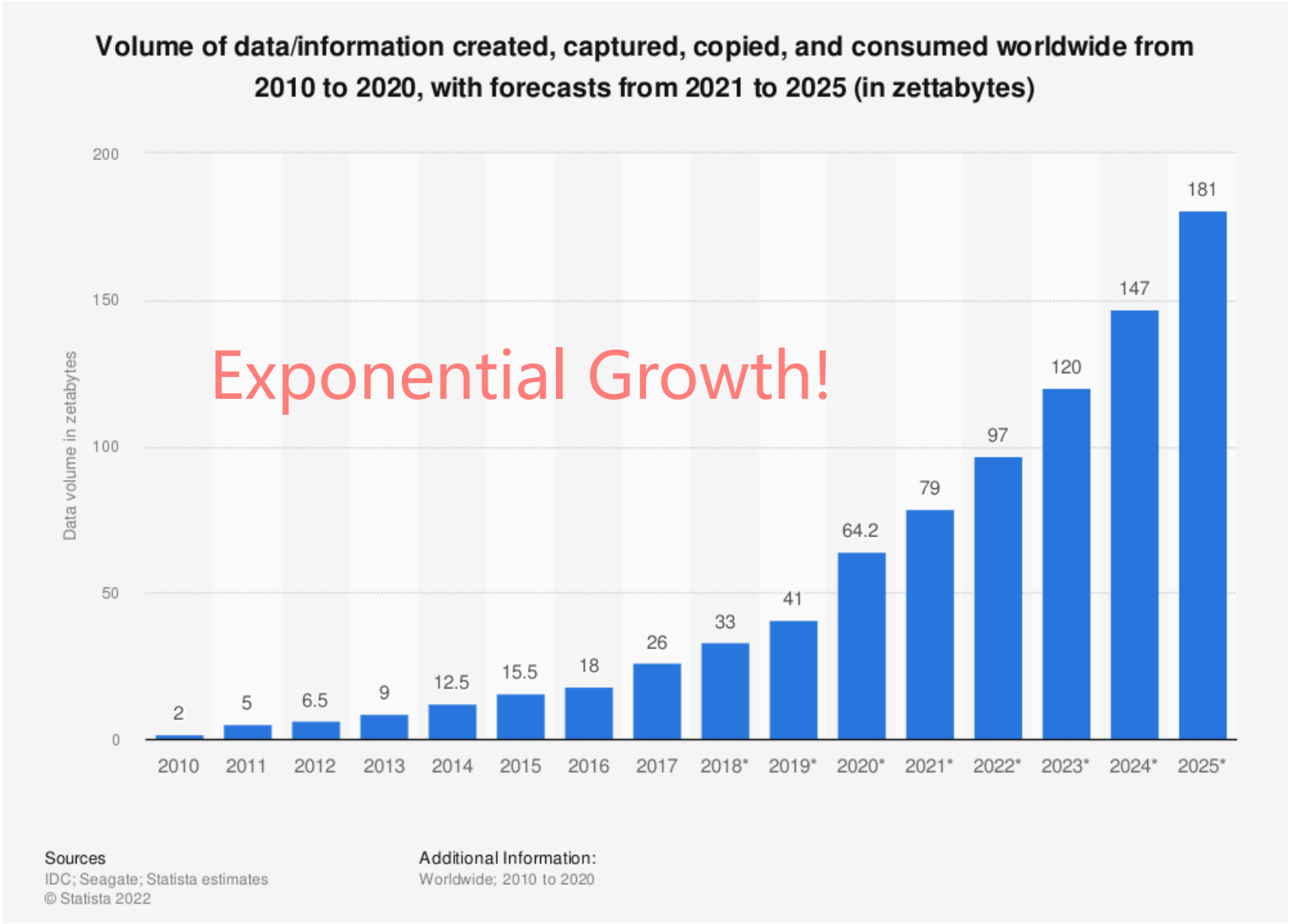
移动存储



2GB - 2TB



1TB - 18TB

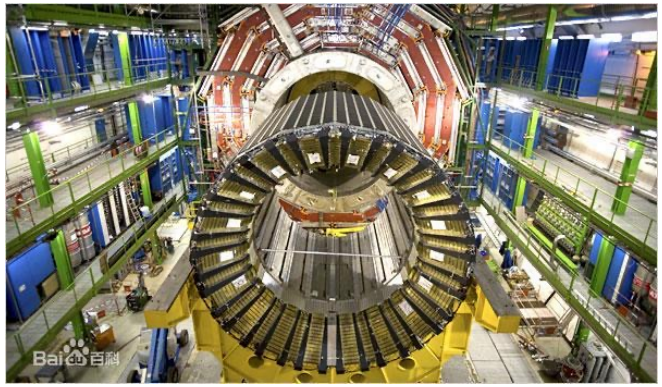


$$1\ \text{ZB} = 1024\ \text{EB} = 1024^2\ \text{PB} = 1024^3\ \text{TB} = 1024^4\ \text{GB}$$

https://www.statista.com/statistics/871513/worldwide-data-created/

超大规模数据的应用（1）

口科学研究



大型强子对撞机

平方千米阵列射电望远镜

NASA 宇宙观测

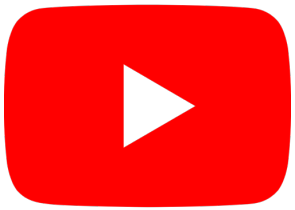
- 大型强子对撞机每年产生 25PB 的数据
- 电射望远镜每年产生 600PB 的数据

超大规模数据的应用（2）

口社交媒体平台

WeChat

Facebook



YouTube

- 2024 年，微信月活用户达 13.59 亿
- 2021 年，Facebook 的月活用户达 28 亿；每天新产生 47.5 亿条新内容，45 亿个 likes 和 3.5 亿张图片
- 2021 年，YouTube 用户每分钟上传超过 500 小时的视频

超大规模数据的应用（3）

口搜索引擎

Google

Bing

Baidu

- 截至 2021 年，谷歌检索超过 130 万亿的网页，需要 EB 级的存储
- 2021 年，谷歌每天需要处理超过 35 亿的查询，等同于每秒 4 万次查询

磁盘技术的发展

Parameter	1956	2016
Capacity	3.75 MB	10 TB
Average access time	≈ 600 msec	2.5–10 ms
Density	200 bits/sq. inch	1.3 TB sq. inch
Average life span	≈ 2 000 hours/MTBF	≈ 22, 500 hours/MTBF
Price	$9,200/MB	$0.032/GB
Weight	910 Kg	62 g
Physical volume	1.9 m^3	34 cm^3

- 磁盘密度增加了 650×10^6 倍!
- 磁盘价格下降了 300×10^6 倍!
- 磁盘容量增加了 2.7×10^6 倍!

课堂思考

□你每天产生了多少数据?

- 想一想你日常使用的 App（微信、抖音、网盘等），一天大约产生多少数据?

□这些数据存储在哪里?

- 手机本地？云端服务器？如何保证你的数据不丢失?

□如果管理全世界的数据，你会怎么设计存储系统?

- 带着这个问题，我们来看看云存储技术的发展

数据存储模型

存储模型

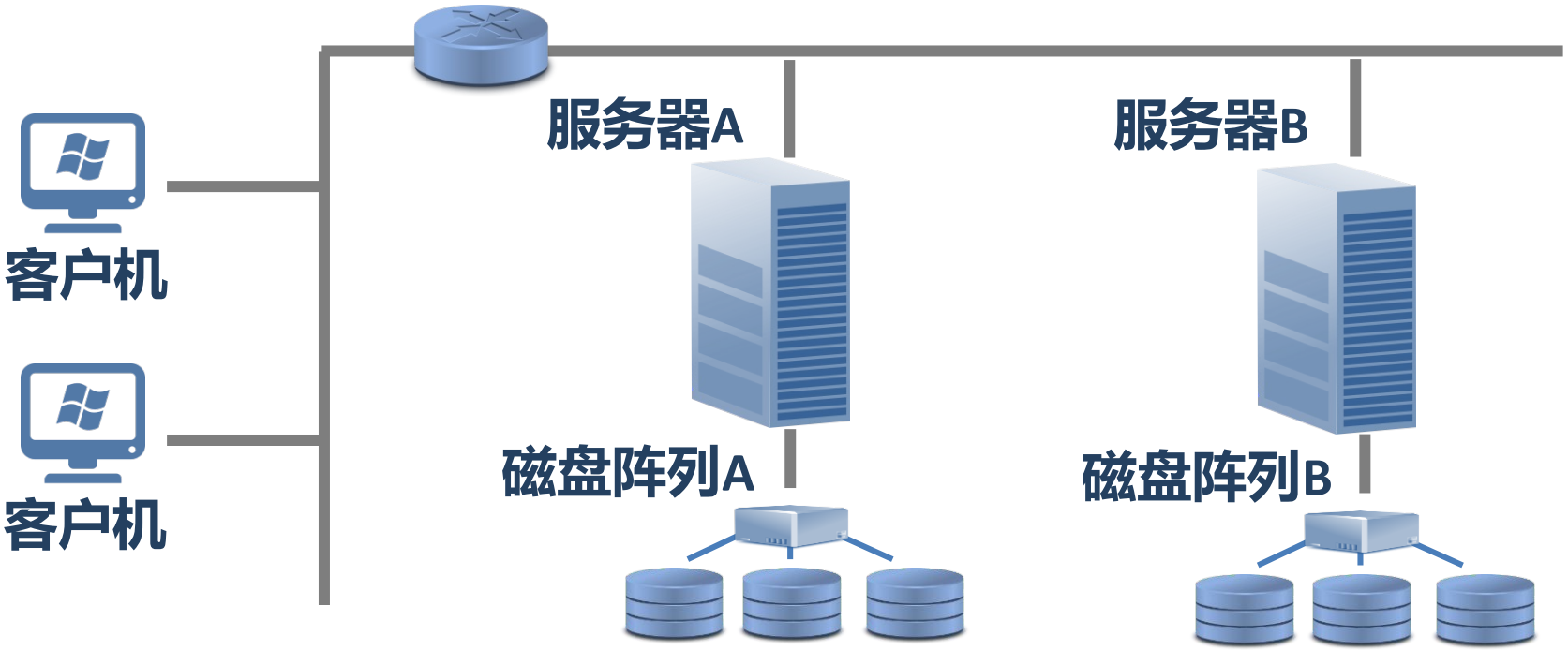
存储模型（Storage Model）是一种描述计算机数据存储结构和管理方式的概念模型

数据放在哪里、谁来管理、服务器/客户端通过什么方式访问它

常见的存储模型包括：

- 直接连接存储 (DAS, Direct Attached Storage)
- 网络连接存储 (NAS, Network Attached Storage)
- 存储区域网络 (SAN, Storage Area Network)
- 云存储 (Cloud Storage)

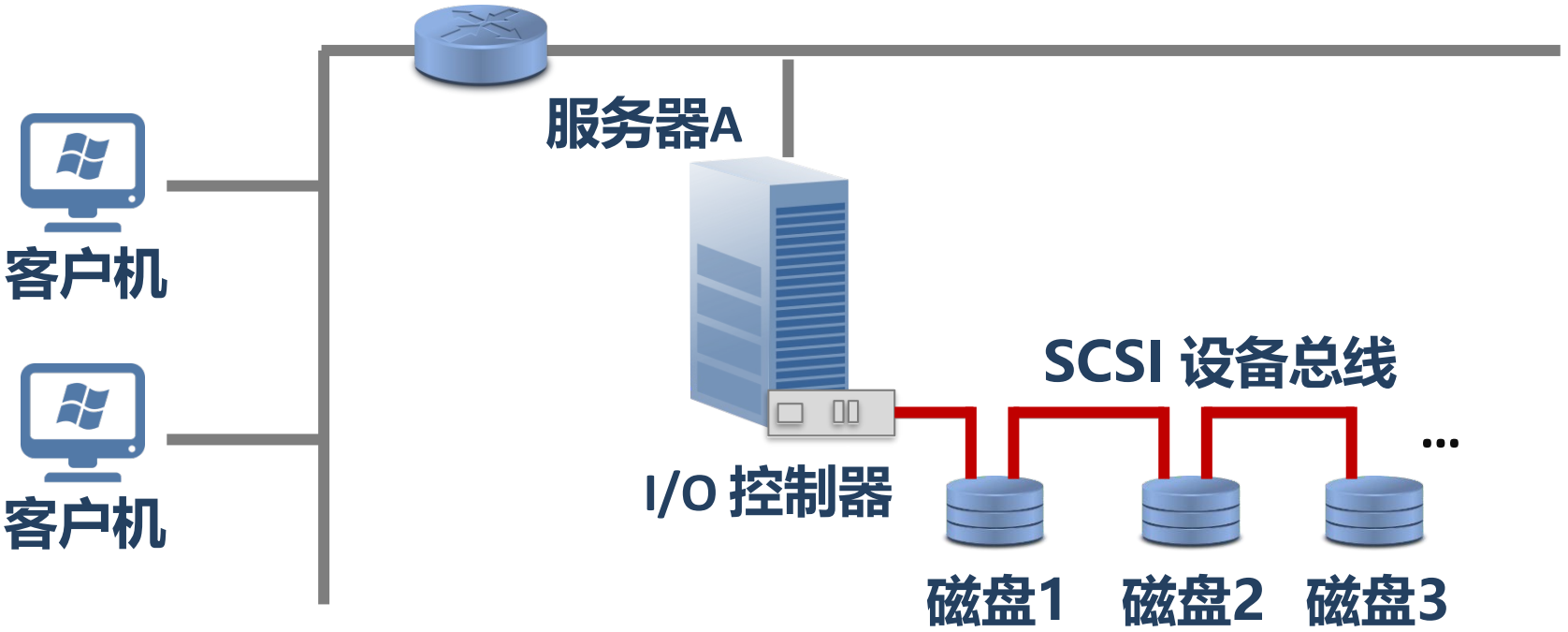
直连式存储



一种将存储设备（如硬盘、SSD、磁带驱动器）直接连接到服务器或工作站的存储模型（硬盘直接插在服务器上）

设置和管理简单，不需要复杂的网络配置，数据传输延迟低，适用于小型企业/办公室

直连式存储

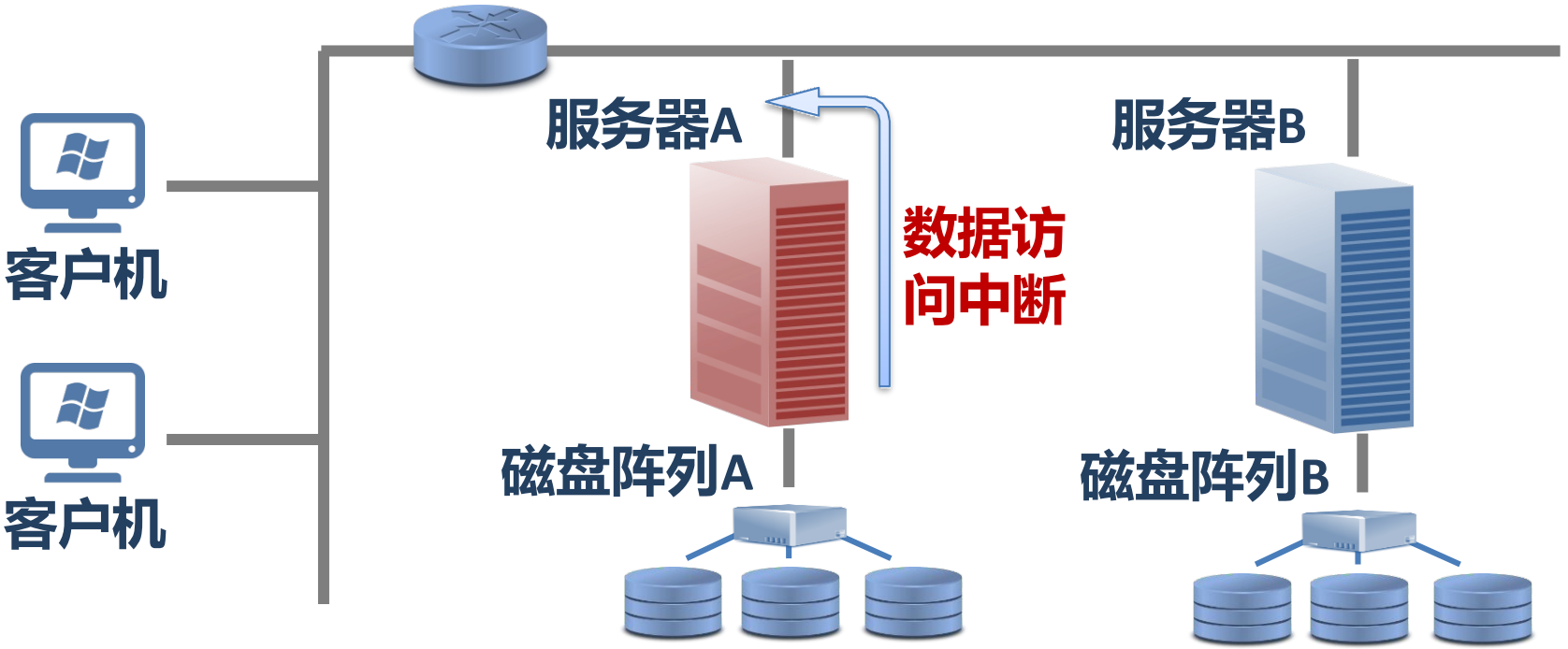


4 位设备 ID = 最多连接 15 个设备+1 台控制器

直连式存储有什么问题?

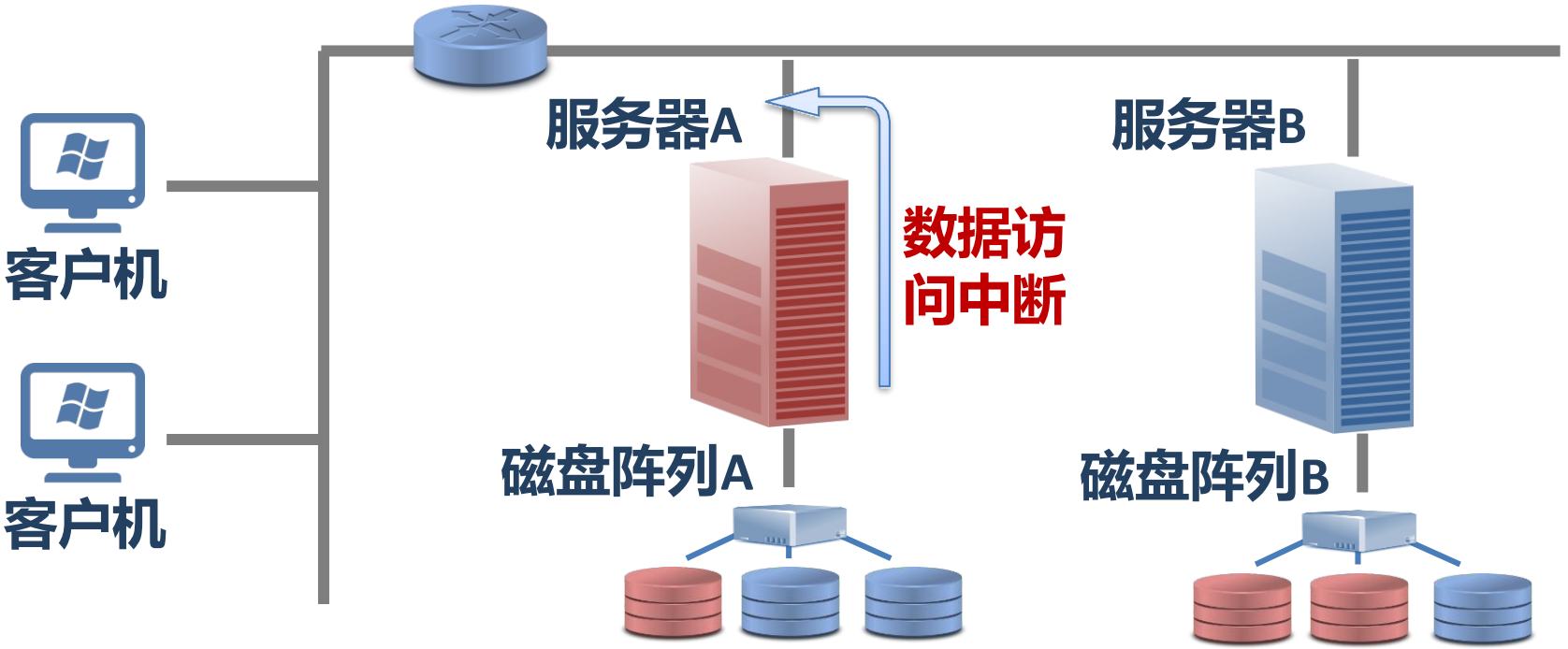
SCSI (Small Computer System Interface) 是一种用于连接计算机和外围设备（如硬盘驱动器、光驱、扫描仪和打印机等）的标准接口

直连式存储



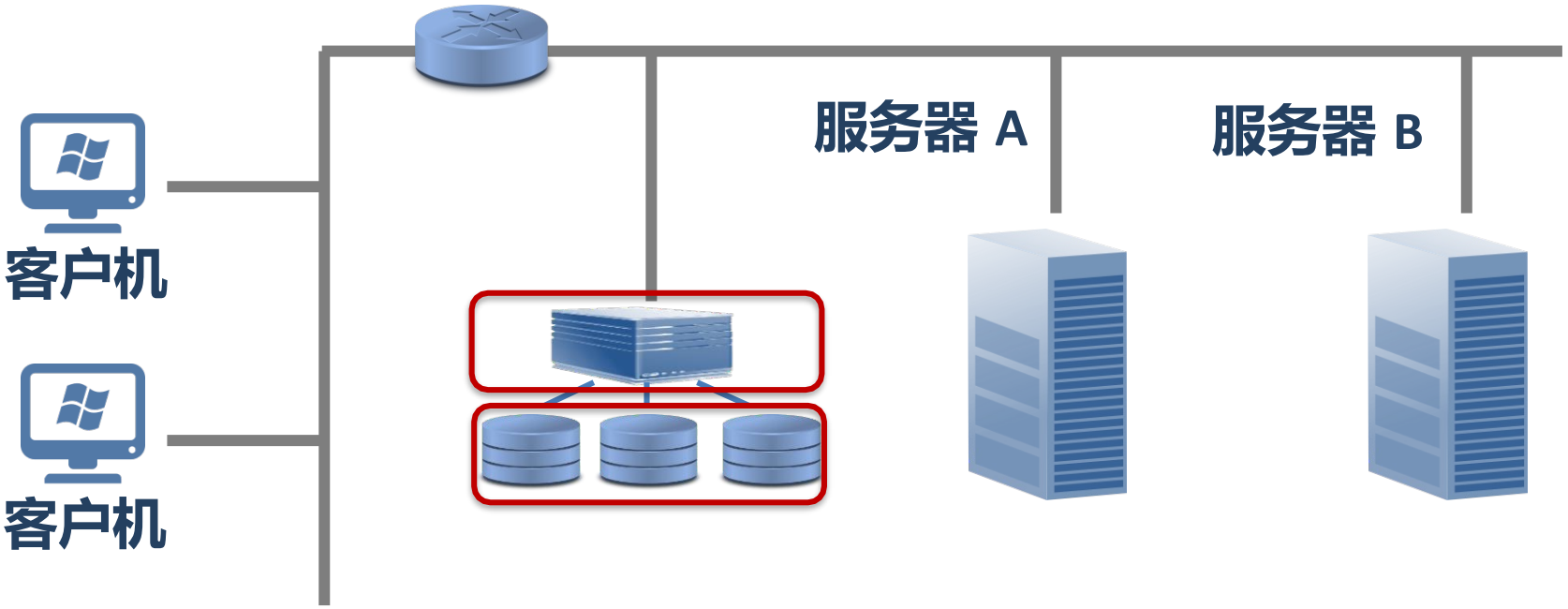
1. 单点故障： 如果服务器发生故障，其连接的 DAS 存储上的数据也无法访问，即使磁盘本身没坏

直连式存储



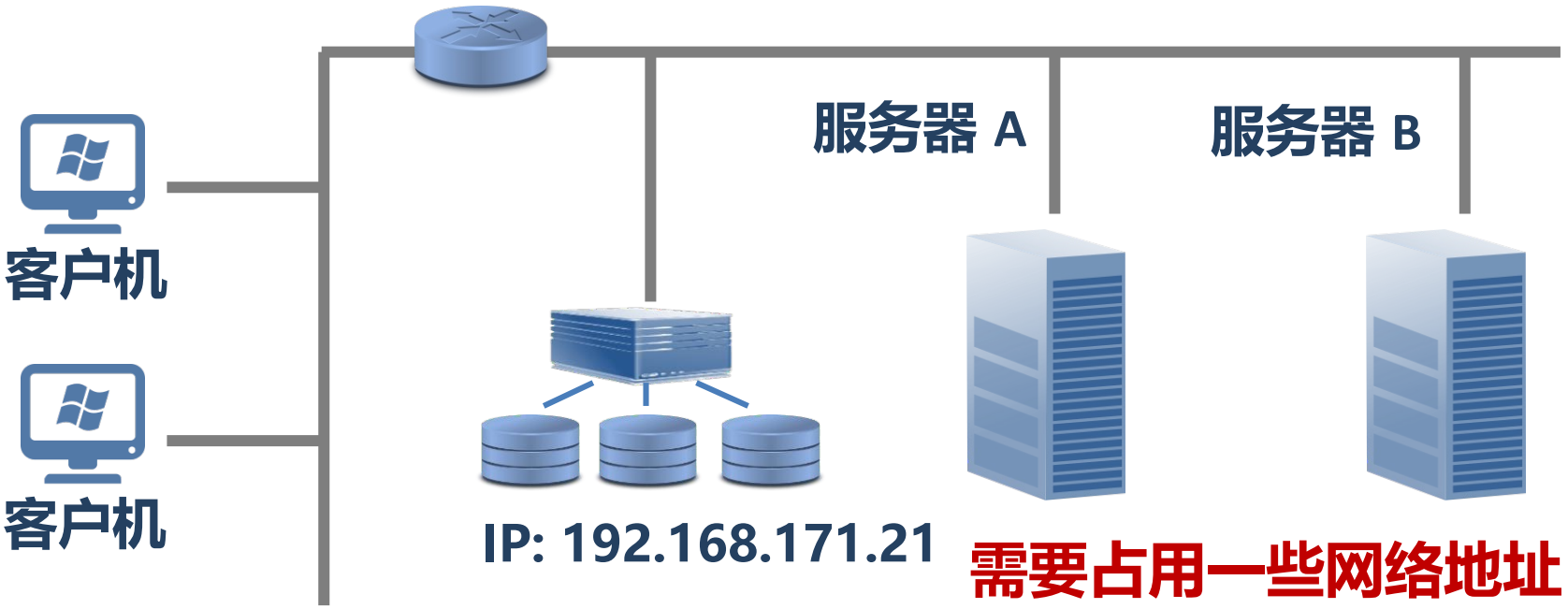
2. 资源利用率低：存储资源通常只能被连接的单一服务器访问，难以在多台服务器之间共享，资源无法有效调配

网络附加存储



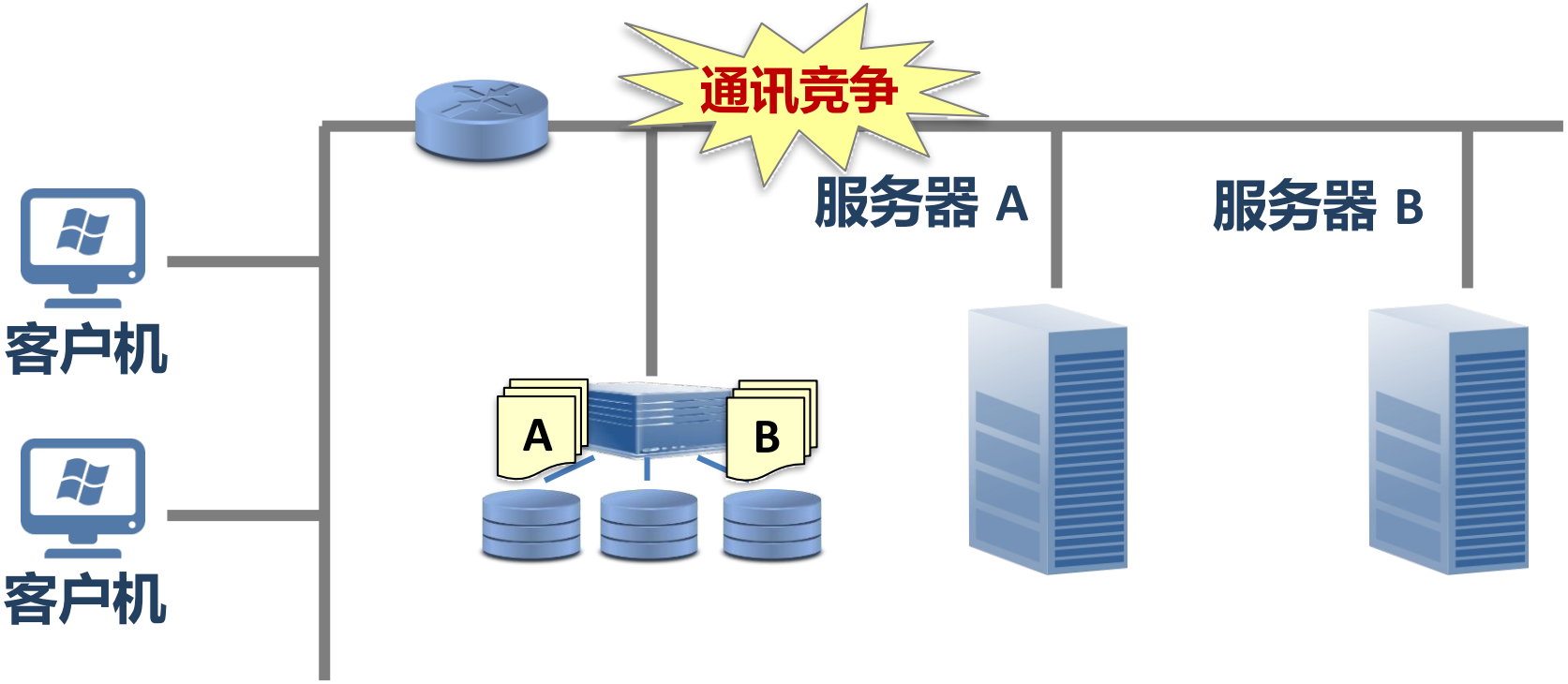
NAS 是一种专用的文件存储设备，它通过标准的以太网连接到局域网 (LAN)，为网络中的多个客户端（计算机、服务器）提供文件级别的共享存储服务

网络附加存储



方便多用户、多平台之间共享文件，数据集中存放，便于管理、备份和维护，适用于中小型企业

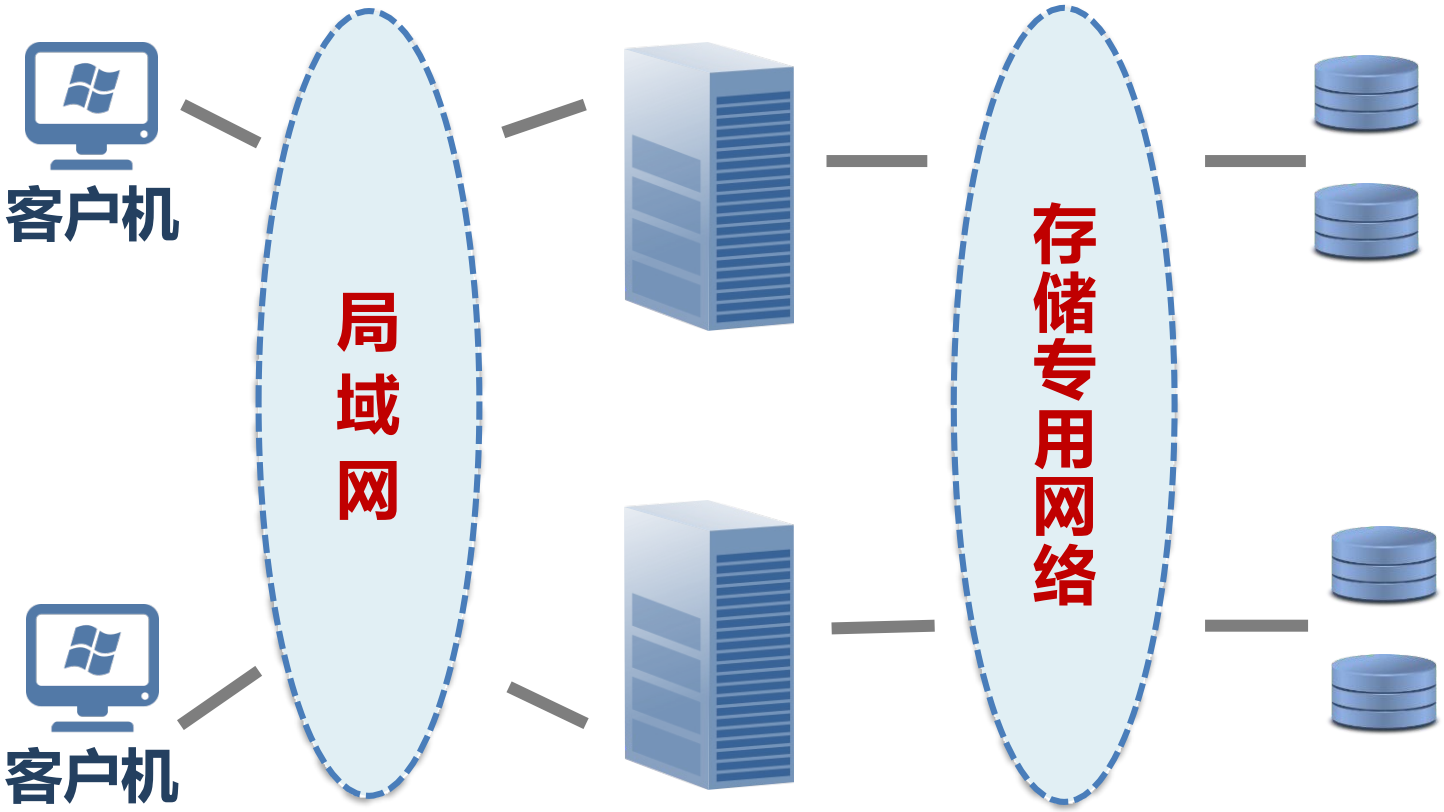
网络附加存储



1. 性能瓶颈： 性能受限于网络带宽和 NAS 设备自身的处理能力，在高并发访问或大数据块传输时，局域网可能成为瓶颈

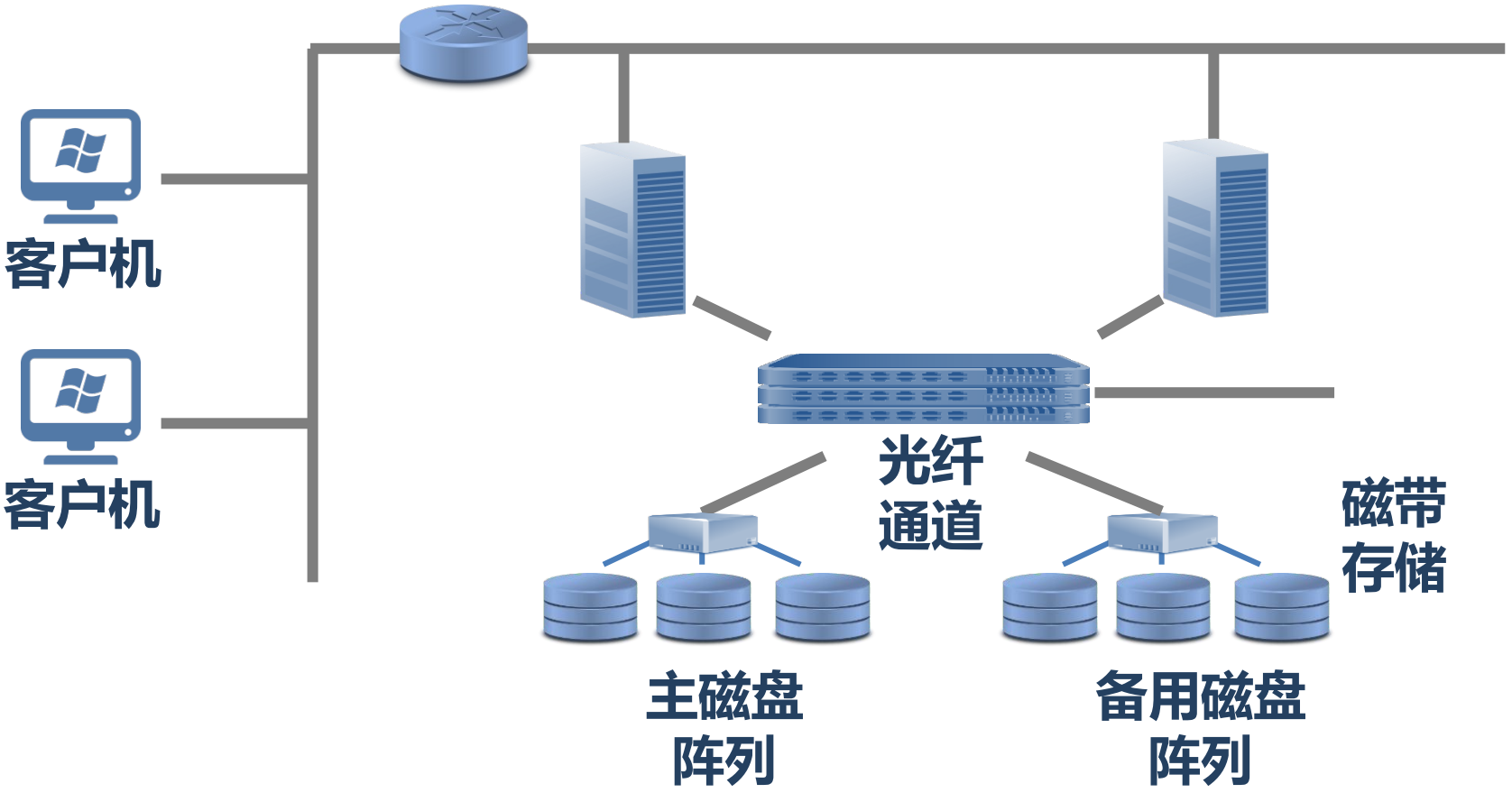
2. 文件级访问： 主要提供文件级访问，不适合需要块级访问的应用（如某些数据库系统）

存储区域网络



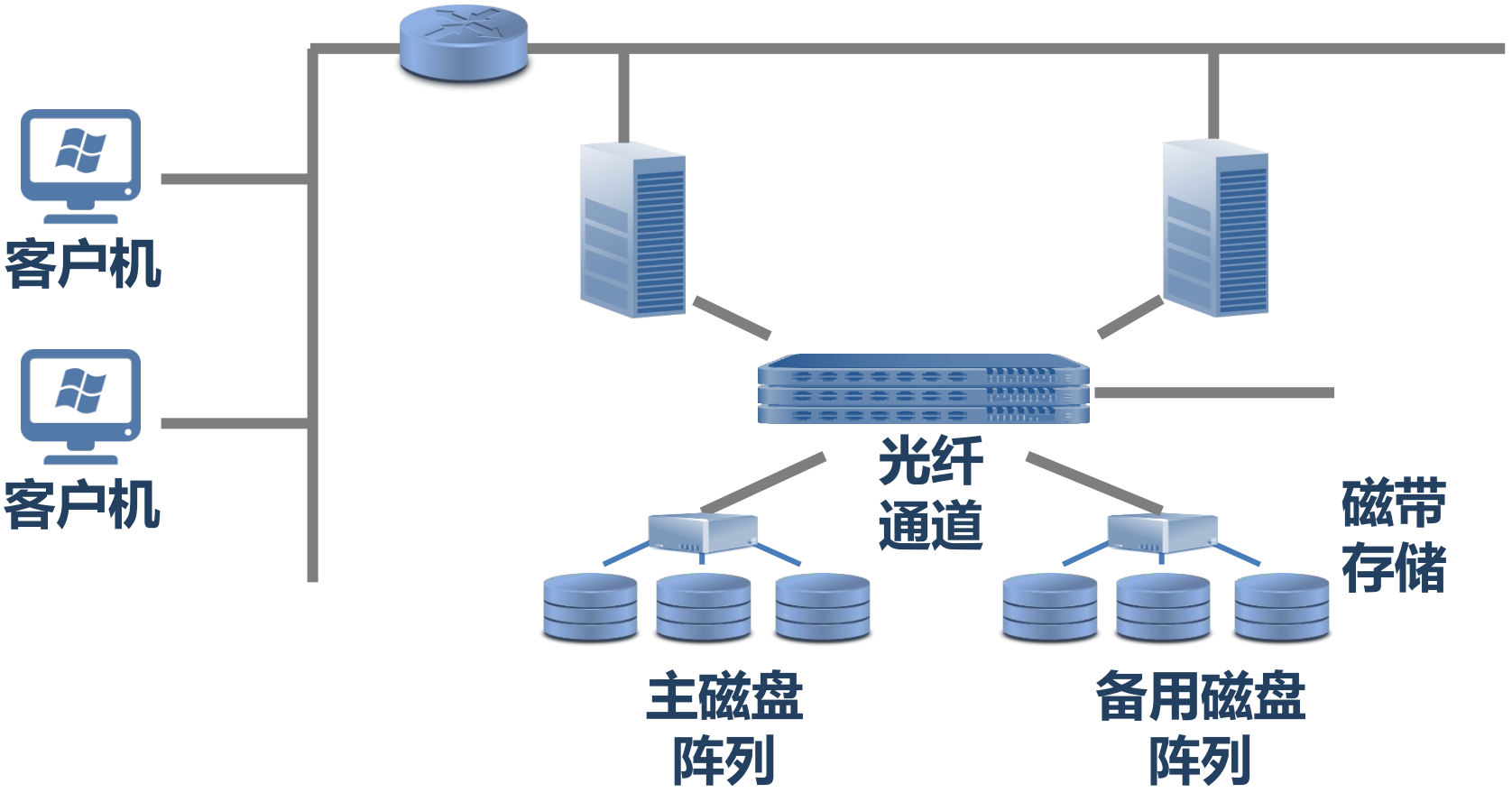
一种通过专用高速网络（如光纤通道或 iSCSI）将存储设备连接到共享存储设备池（如磁盘阵列、磁带库）的存储模型。SAN 提供块级存储服务，通常用于**企业级应用**

存储区域网络



专用的高速网络，提供极佳的I/O性能和低延迟，特别适合事务密集型应用，扩展性好，资源利用率高

存储区域网络



1. **高成本：** 光纤通道设备（HBA、交换机、存储阵列）价格昂贵
2. **复杂性：** 设计、部署和管理 SAN 需要专业知识和技能

DAS vs NAS vs SAN 对比总结

□DAS（直连式存储）

- 连接方式：直接连接服务器 | 访问级别：块级 | 成本：低
- 适用场景：小型办公室、单服务器环境

□NAS（网络附加存储）

- 连接方式：以太网 | 访问级别：文件级 | 成本：中
- 适用场景：文件共享、备份、中小企业协作

□SAN（存储区域网络）

- 连接方式：专用光纤网络 | 访问级别：块级 | 成本：高
- 适用场景：数据库、虚拟化、高性能企业应用

□核心权衡：成本 ↔ 性能 ↔ 共享能力 ↔ 管理复杂度

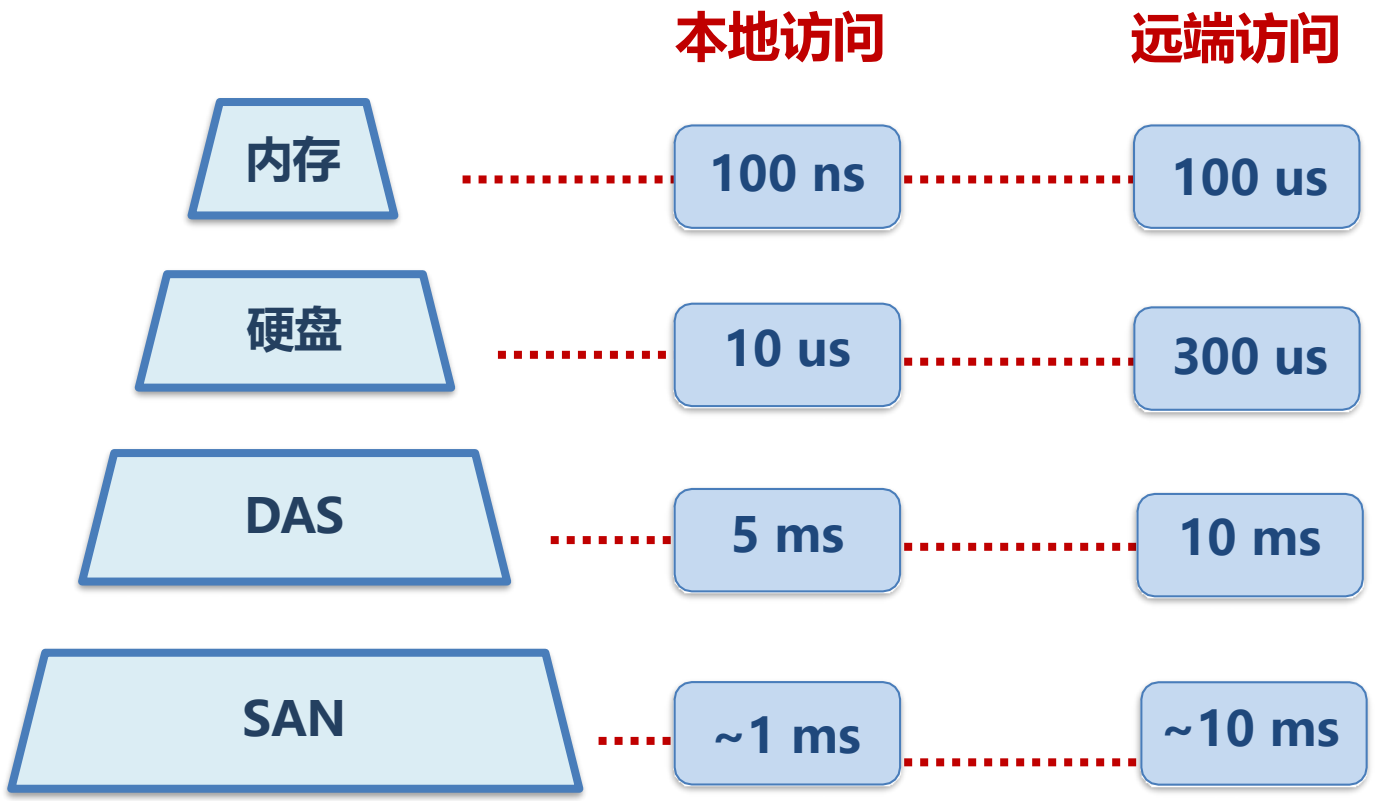
优化存储速度

Enterprise Storage Guide

HOME | NEWS | GUIDES | WHITEPAPERS | WEBINARS | VIDEOS | ABOUT US | CONTACT US | RSS FEED

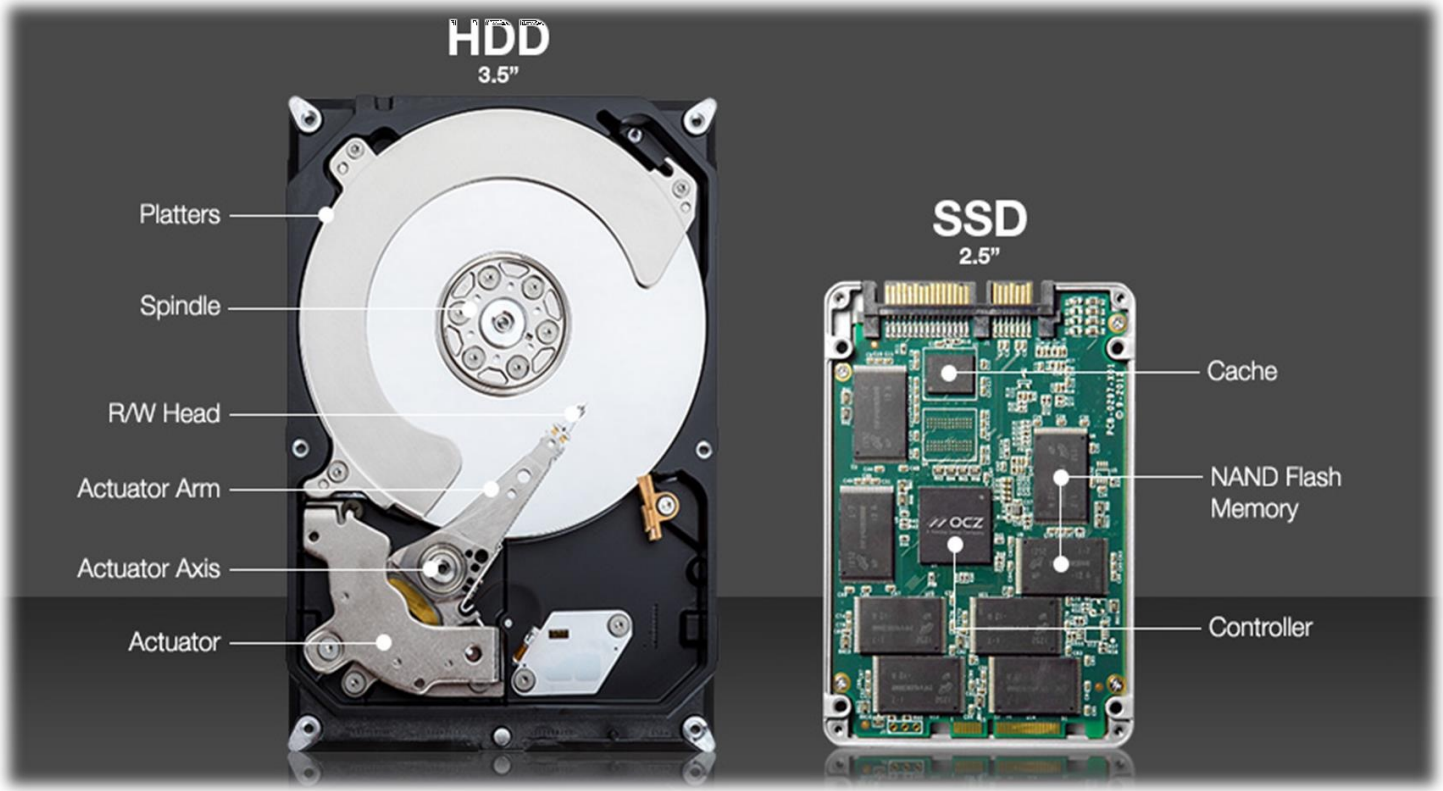
Latency: The King of Storage Performance Metrics

分布式存储系统下的层次化数据访问



固体硬盘

- 基于闪存技术的固态硬盘（SSD）得到快速发展

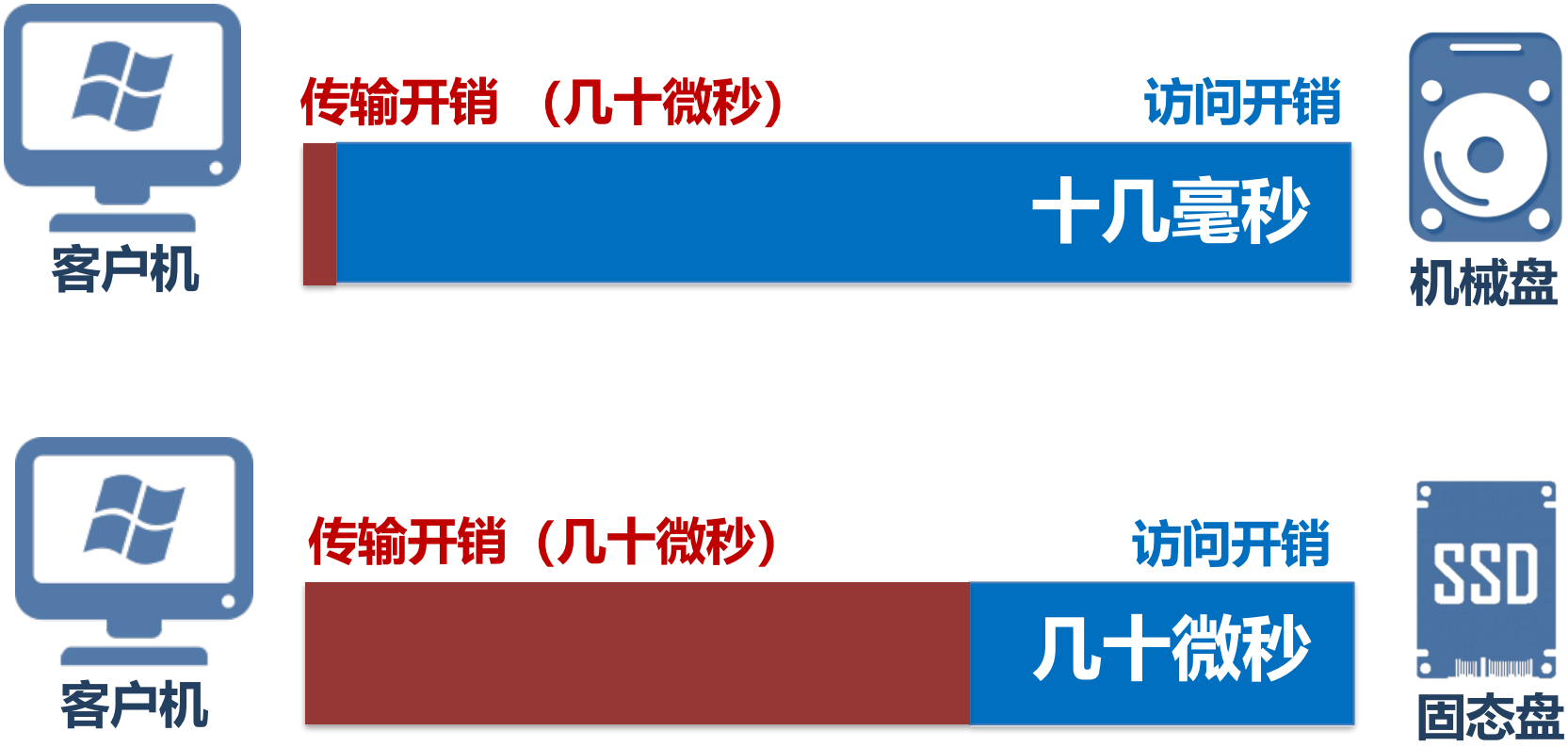


机械硬盘
Hard Disk Drive (HDD)

固态硬盘
Solid State Drive (SSD)

https://baike.baidu.com/starmap/view?nodeId=6774351af6c857499b335609

性能瓶颈的迁移



云存储和云数据库对比

- 云存储不是一个设备，而是一种云端的存储服务
  - 即把**数据存储和访问作为一种服务**并通过网络提供给用户
- 云存储提供基本存储资源，属于 IaaS 层
  - 用于存放各种类型文件
- 云数据库提供数据管理能力，属于 PaaS 层
  - 整体数据实现结构化，用于组织数据

SaaS 层的云存储服务?

Azure Blob Storage

amazon RDS

Azure SQL

Cloud SQL

云存储 (Cloud Storage)

云数据库 (Cloud Database)

云存储面临的问题和挑战

- **性能 (Performance)**
  - 通过设备并行性或者通过新型硬件技术提升性能
- **成本 (Cost)**
  - 存储设备成本惊人，只有热点数据适合多副本机制
- **容错 (Fault tolerance)**
  - 设备失效概率随着复杂性上升而提高
- **扩展 (Scalability)**
  - 如何扩展存储系统，尤其是在线扩展
- **其它 (Others)**
  - 数据一致性、功耗能耗、使用寿命等

主流云存储服务一览

□Amazon S3 (Simple Storage Service)

- 全球最大的对象存储服务，11个9的数据持久性 (99.999999999%)
- 按使用量付费，支持生命周期管理、版本控制

□阿里云 OSS (Object Storage Service)

- 国内市场份额领先，与 CDN 深度集成
- 支持图片/视频处理、数据湖分析等增值功能

□华为云 OBS (Object-Based Storage)

- EB 级海量存储，支持多 AZ 高可用部署

□Google Cloud Storage

- 多存储类别 (Standard/Nearline/Coldline/Archive)，按冷热分层

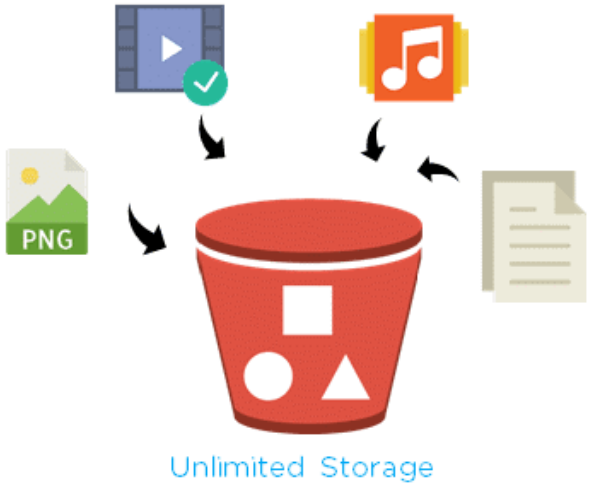
分布式对象存储

分布式对象存储是一种用于存储、管理和检索非结构化数据的存储方案，如图片、视频、日志文件等

口分布式对象存储特点

- **可扩展性**：在多台服务器、多个数据中心甚至多个地理区域之间分散存储数据
- **高可用性**：在多个物理位置存储数据的多个副本
- **简单的接口**：用户可以通过 RESTful API 进行存储和检索操作
- **数据管理功能**：数据生命周期管理、版本控制、访问控制

对象和对象存储设备



- 两个基本概念：
  - **桶（Bucket）**和**对象（Object）**

- 桶是储存对象的容器
  - 桶名不能重复，可用于计费、权限控制等
  - 桶中的所有对象都处于同一逻辑层级

- 对于对象存储来说，对象是数据存储的基本单位
  - 传统的存储系统中，文件和块是基本存储单位
  - 对象的大小可以从几个字节到几个 TB

分布式对象存储

键值 (Key)	对象的名称，为经过UTF-8编码的长度大于 0 且不超过1024的字符序列。一个桶里的每个对象须拥有唯一的对象键值
元数据 (Metadata)	对象的描述信息，如日期、内容类型；元数据以键值对 (Key-Value) 被上传到 OBS (Object Storage Service) 中
数据 (Data)	文件的数据内容

Huawei 分布式对象存储服务 OBS：https://www.huaweicloud.com/product/obs.html

对象存储优势总结

- DAS 和 SAN 属于**块存储 (Block-level)** 类型
  - 数据被分割为固定大小的块，每个块有唯一的标识
  - 优点是传输速度较快，常用于需要高性能和低延迟的环境
- NAS 产品往往是**文件级 (File-level)** 存储
  - 文件按照目录层次结构进行组织
  - 优点是价格低，方便共享
  - 存储性能可能不如块级存储
- 对象存储兼具 SAN 和 NAS 的特点
  - 扩展性强：比传统单机 NAS 更容易扩展到 PB、EB 级规模
  - 适合海量非结构化数据：数量多、单个对象可能较大、访问方式相对简单、更新不频繁
  - 通过 HTTP/API 访问：非常适合云环境、Web 应用和跨地域访问

分布式系统中的共识

共识 (Consensus)

在分布式系统中，共识是指多个计算节点就某个特定值或状态达成一致的过程。这在分布式系统设计中非常重要，因为系统中的各个节点需要协调和同步，以确保数据一致性和系统可靠性

□为什么需要共识?

- **资源访问控制**：确保在同一时间只有一个节点能够访问某个共享资源（互斥）
- **领导者选举**：在分布式系统中选举出一个领导节点来协调其他节点的操作
- **事件顺序一致性**：确保所有节点以相同的顺序处理事件，以便系统状态一致

为什么共识如此重要？

□场景：你向网盘上传了一份重要文件

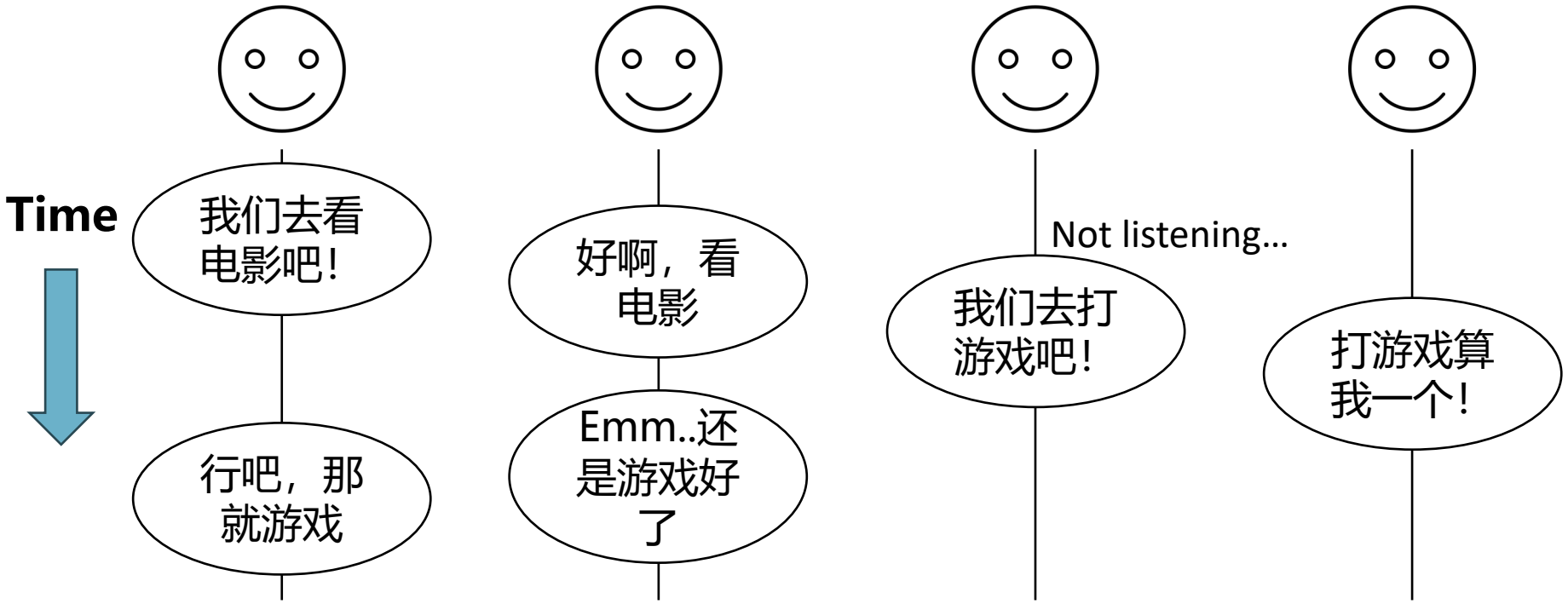
- 文件被复制到 3 台不同的服务器上
- 此时其中 1 台服务器突然宕机了……

□问题来了：

- 剩余服务器如何判断这次写入是否已经成功提交？
- 宕机服务器恢复后，如何知道自己是否缺少最新版本？
- 多个客户端同时修改同一文件时，系统如何决定最终版本？

□这就是分布式共识要解决的核心问题！

达成共识是什么意思?



- 共识是就一个结果达成一致
- 各方希望就任何结果达成一致，而不仅仅就自己的提议达成一致
- 一旦多数人同意一项提案，那就是共识
- 达成的共识最终可以被所有人知道
- 通信信道可能出现故障，也就是说，信息可能会丢失

共识 (Consensus)

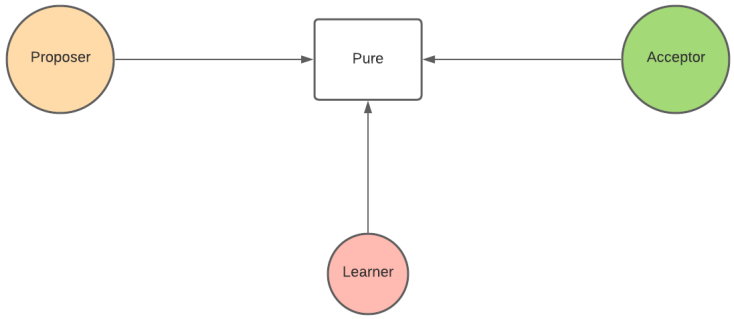
□共识问题的定义

- 共识问题可以简单地描述为：一个或多个系统可能会提出某个值，如何让一组计算机就其中一个提出的值达成一致？

□共识协议的属性：

- **有效性 (Validity)**：只有提出的值可以被决定。如果一个进程决定了某个值 v，那么一定有某个进程提出了 v
- **一致性 (Uniform Agreement)**：没有两个正确的进程（那些不会崩溃的进程）可以决定不同的值
- **完整性 (Integrity)**：每个进程最多只能决定一个值
- **终止性 (Termination)**：所有进程最终都会决定一个结果

Paxos 算法



Paxos 算法用于在通过异步网络通信的一组分布式计算机之间实现共识。当运行 Paxos 的系统中多数 Acceptors 同意其中一个提议的值时，共识就达成了

Paxos 是最早被广泛研究、严格证明并在实际系统中产生巨大影响的异步崩溃容错共识算法之一

- Paxos 提供可中止的共识 (abortable consensus)
  - 当客户端向 Paxos 提出一个值时，若有一个竞争的并发提议获胜，那么提议的值可能会失败。客户端需再次向另一轮 Paxos 算法提出该值

Lamport, Leslie. ”The Part-Time Parliament“ ACM Trans. Comput. Syst., 1998.

Paxos 算法的假设

- **并发提议 (Concurrent proposals)**：一个或多个系统可能会同时提出一个值
- **有效性 (Validity)**：所选的值必须是提议的值之一
- **多数规则 (Majority rule)**：一旦 Paxos 中大多数 Acceptors 同意其中一个提议的值，该值便达成共识
- **异步网络 (Asynchronous network)**：网络是不可靠的和异步的，消息可能会丢失或任意延迟
- **停止故障 (Fail-stop faults)**：系统可能出现故障，但不是拜占庭式的。重启后能记住之前的状态，以确保不会改变主意
- **单播 (Unicasts)**：通信是点对点的。没有机制可以原子地将消息多播到 Paxos 服务器集
- **公告 (Announcement)**：一旦达成共识，可以让所有人知道

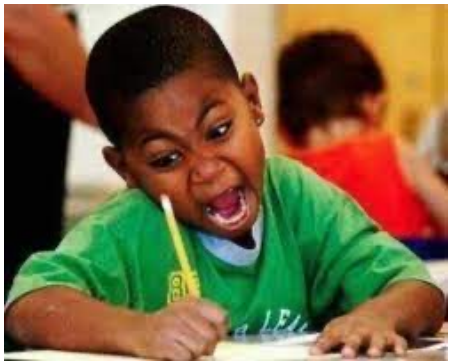
Paxos 基础

- Paxos 定义了三种角色
  - ✓**提议者（proposer）**：提出游玩提案让大家讨论
  - ✓**接受者（acceptor）**：进行提案讨论/裁决使共识达成
  - ✓**学习者（learner）**：学习已达成的提案

- 比喻

提议者（proposer）
老师

接受者（acceptor）
一起讨论的同学



学习者（learner）
抱佛脚的同学

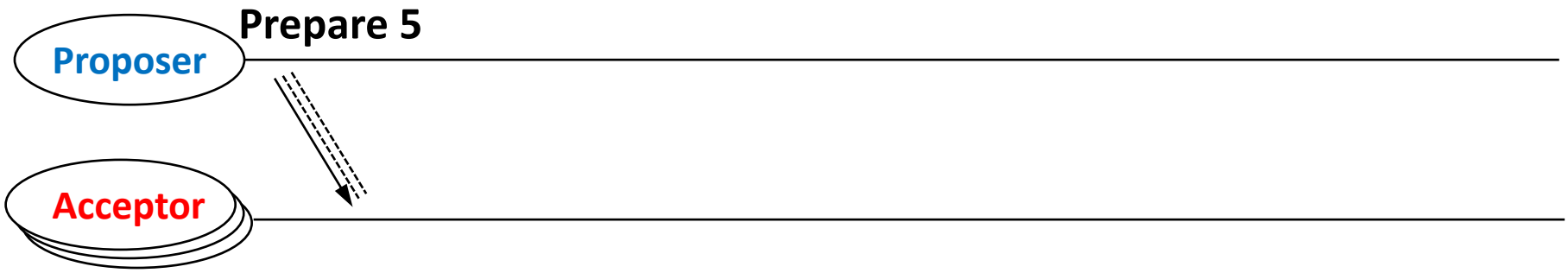
Paxos 基础

- Paxos 定义了三种角色
  - ✓**提议者（proposer）**：提出游玩提案让大家讨论
  - ✓**接受者（acceptor）**：进行提案讨论/裁决使共识达成
  - ✓**学习者（learner）**：学习已达成的提案

- **Paxos 节点**可以承担多个角色，甚至所有角色
- **Paxos 节点**知道多少个 acceptor 节点才算是大多数（比如 2/3）
- **Paxos 节点**必须是持久的，意味着节点不会忘记已接受的提案

- 一次 Paxos 运行旨在达成一个共识
- 当共识达成，算法结束，不会继续运行达成其他共识
- 若要达成其他共识，必须重新执行一次 Paxos 运行

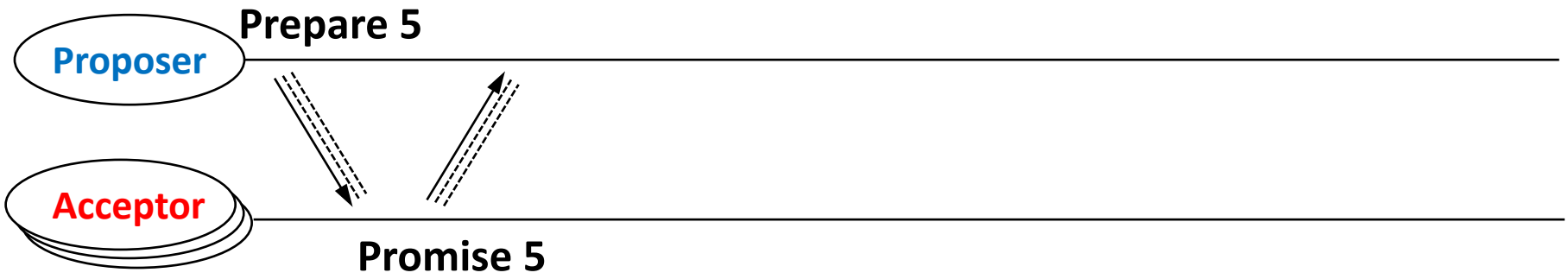
Paxos 算法



Proposer 发起提案 (来自 **client** 的 **value**)

- 发送 **Prepare ID** 给多数 (或所有) Acceptor
- **ID** 在多个 **Proposer** 中唯一 (时间戳 + Proposer ID)
  - ✓ 即若有其他 **Proposer** 将使用全局唯一的 **ID**
    - ❖ **Proposer 1**: 1, 3, 5, 7
    - ❖ **Proposer 2**: 2, 4, 6, 8
- 超时则重传，使用新的 **ID**

Paxos 算法

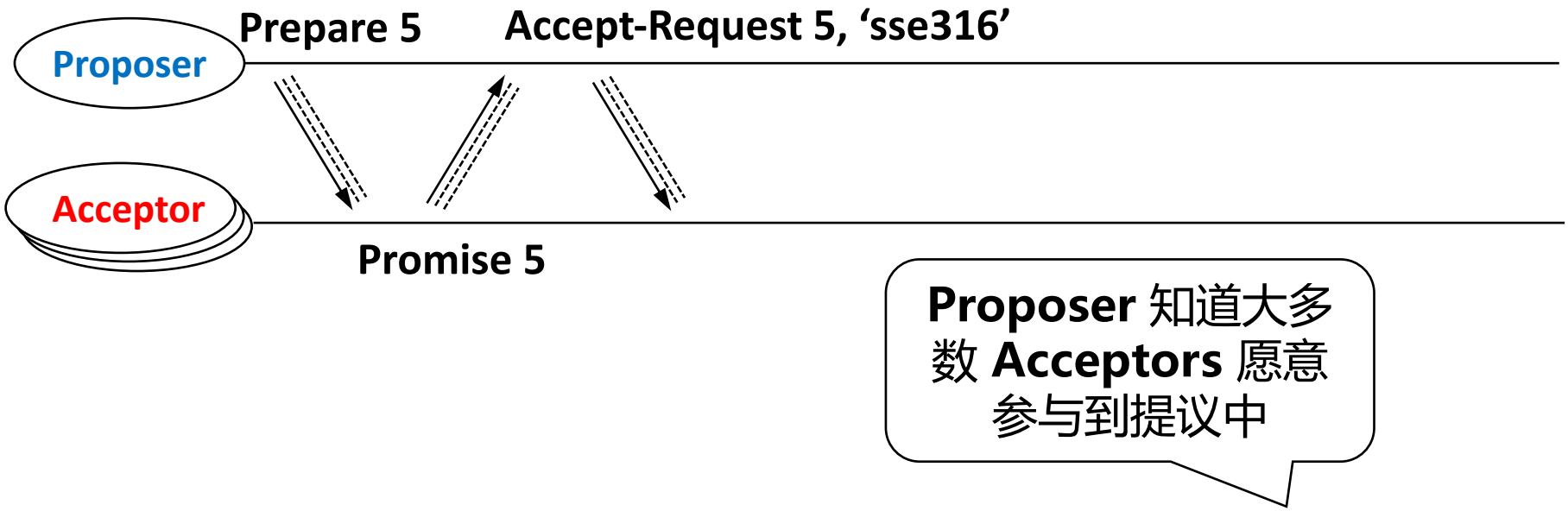


Acceptor 接收到提案 **Prepare ID**

- 判断是否承诺忽略此 **ID**
  - ✓ 是 -> 忽略 (或者回复 **fail** 消息)
  - ✓ 否 -> 将承诺忽略比此 **ID** 更小的提案，回复 **Promise ID** 消息 (?)

若 **Acceptor** Promise 了某个 **ID**，其他更小或相同的 **ID** 将不会被 Promise

Paxos 算法



Proposer 收到的大多数 **Promise** 回复均为某个 **ID**

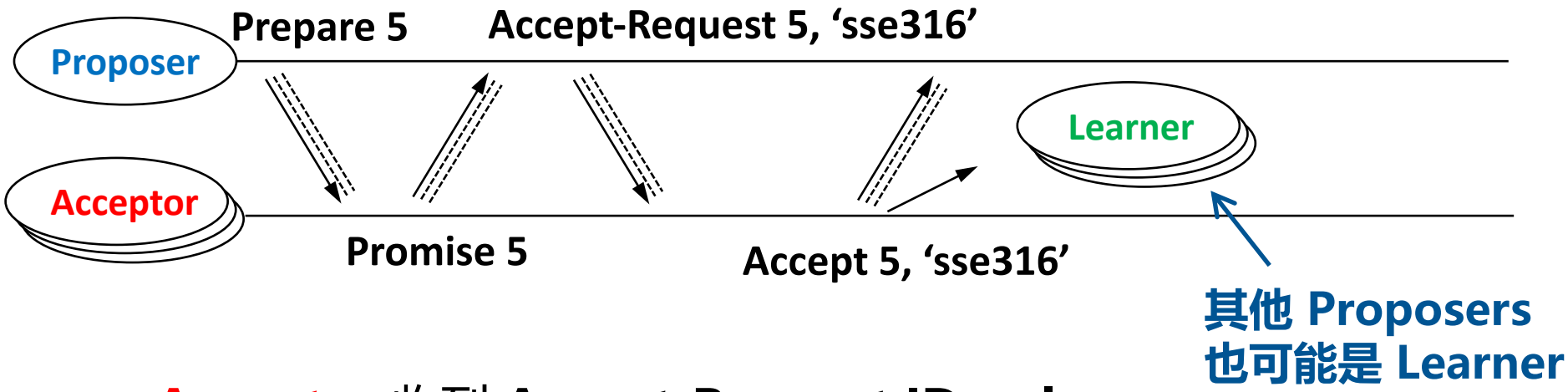
- 发送 **Accept-Request ID, value** 给多数 (或所有) **Acceptors**
- **value** 的值可以任意选 (?)

ID 用来决定“谁的提案优先级更高”；
value 才是最终要达成共识的内容

Value 场景举例

场景	Paxos 要决定什么	value 是什么
文件上传	文件版本是否提交	commit(file_v1)
文件冲突	哪个版本是最新版	latest(file)=v2A
分布式锁	谁获得锁	owner(lock)=NodeA
Leader 选举	谁是 leader	leader=NodeB
日志复制	某个日志槽位写什么	log[10]=put(x,1)
配置变更	集群成员如何变化	config={A,B,C,D}
事务处理	事务提交还是中止	commit(T100) 或 abort(T100)

Paxos 算法

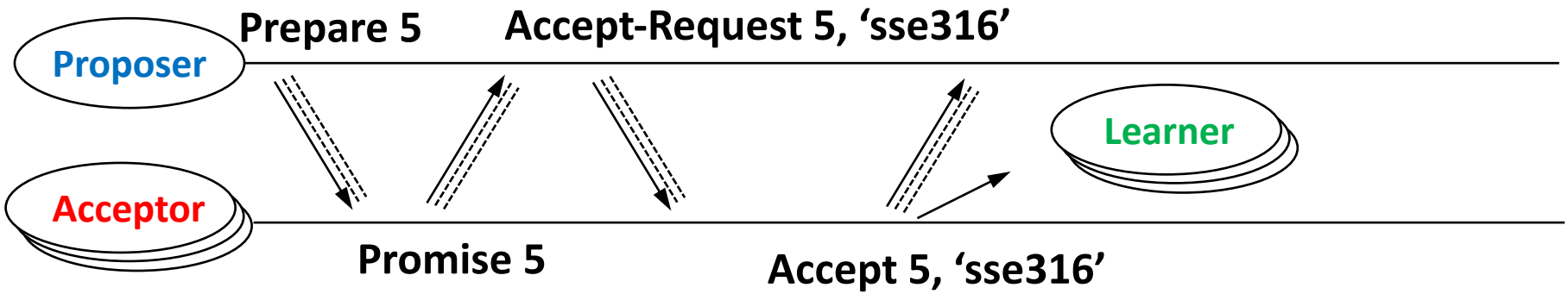


Acceptor 收到 **Accept-Request ID, value**

- 判断是否承诺忽略此 ID
  - ✓ 是 -> 忽略 (或回复 **fail** 消息)
  - ✓ 否 -> 回复 **Accept ID, value** 并发送给 **Proposer** 以及所有 **Learner**

若大多数 **Acceptor** 接受了某个 **ID, value**，则共识达成！
共识达成的是 **value**，而不一定是 **ID**，**ID** 可能会变

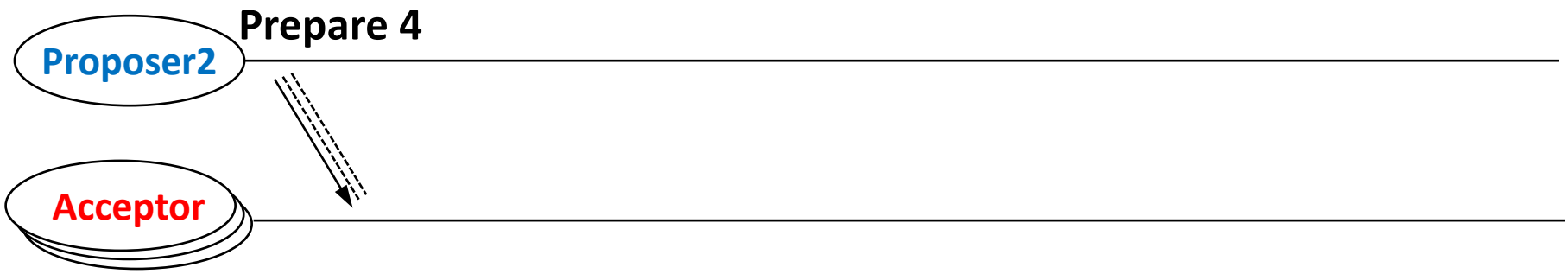
Paxos 算法



Proposer 或者 **Learner** 接收到 **Accept ID, value** 信息

如果一个 **Proposer** 或者 **Learner** 知道 **ID** 已被大多数接受，则他们知道已达成共识 **value**（而不是 **ID**）

Paxos 算法

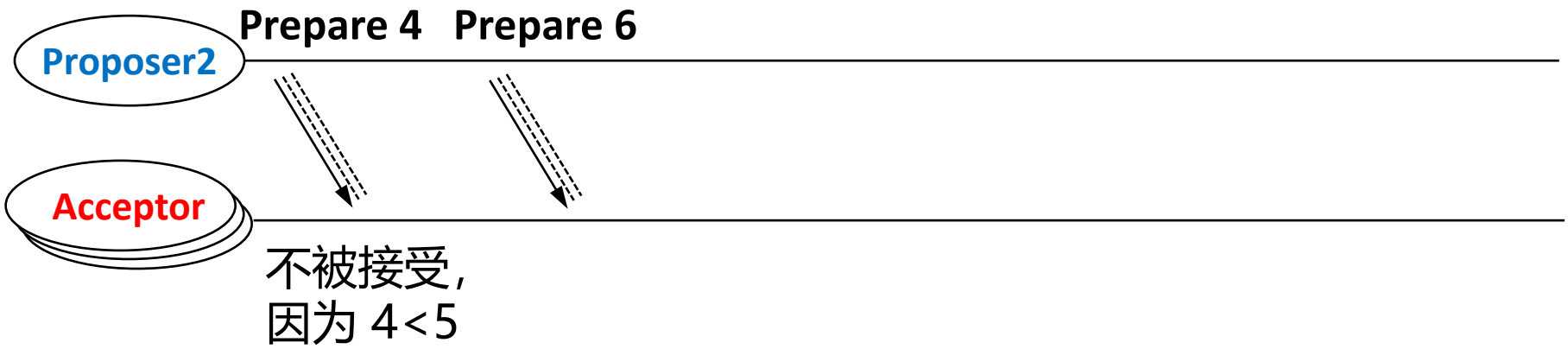


Paxos 算法

Acceptor 接收到提案 **Prepare ID2**

- 判断是否承诺忽略此 **ID2**
  - ✓ 是 -> 忽略
  - ✓ 否 -> 将承诺忽略比此 **ID2** 更小的提案回复 **Promise ID** 消息 (?)

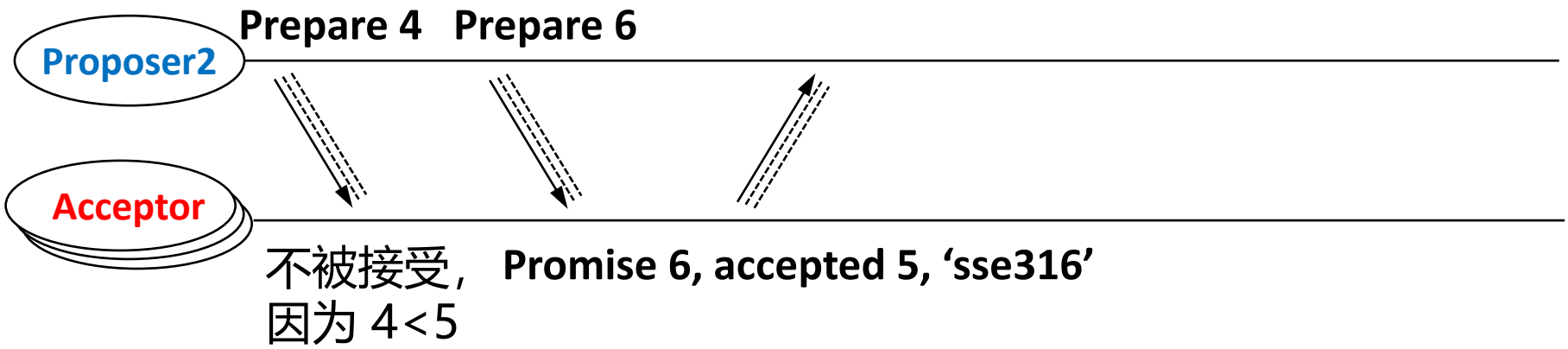
Paxos 算法



Acceptor 接收到提案 **Prepare ID2**

- 判断是否承诺忽略此 **ID2**
  - ✓ 是 -> 忽略
  - ✓ 否 -> 将承诺忽略比此 **ID2** 更小的提案
    回复 **Promise ID** 消息 (?)

Paxos 算法



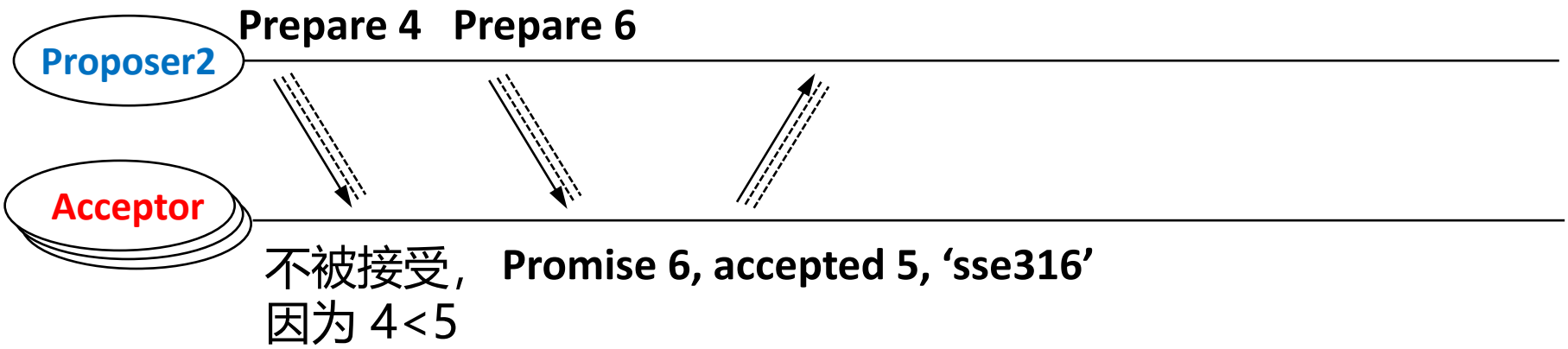
Acceptor 接收到提案 **Prepare ID2**

- 判断是否承诺忽略此 **ID2**
  - ✓ 是 -> 忽略
  - ✓ 否 -> 将承诺忽略比此 **ID2** 更小的提案
    ~~回复 **Promise ID** 消息 (?)~~

检查是否已经接受了某个 **value**

- ✓ 是 -> 回复 **Promise ID2, accepted ID, value** 消息
- ✓ 否 -> 回复 **Promise ID2** 消息

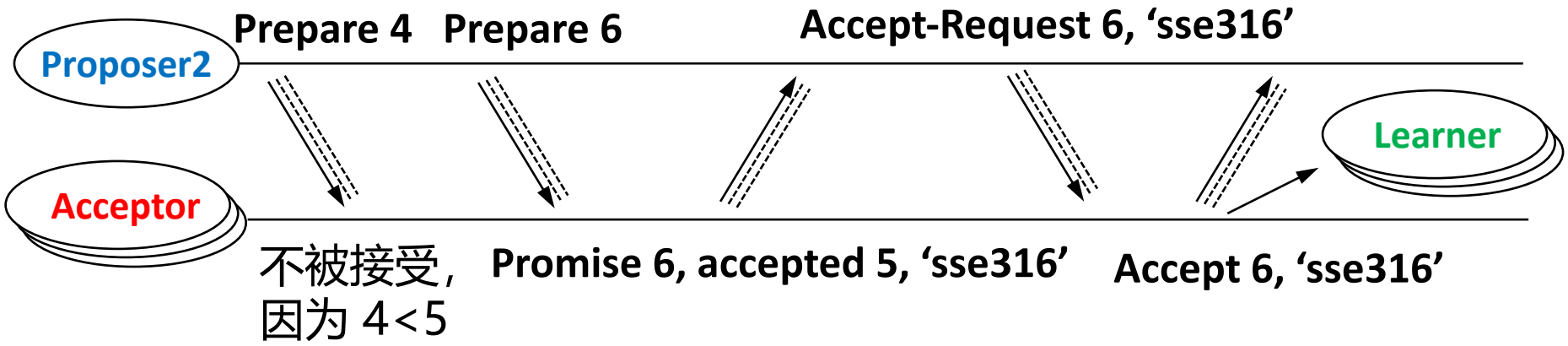
Paxos 算法



Proposer 收到的大多数 **Promise** 回复均为某个 **ID**

- 发送 **Accept-Request ID, value** 给多数 (或所有) **Acceptors**
- **value** 的值可以任意选 (?)

Paxos 算法



Proposer 收到的大多数 **Promise** 回复均为某个 **ID**

- 发送 **Accept-Request ID, value** 给多数 (或所有) **Acceptors**
- ~~**value** 的值可以任意选 (?)~~

- 判断 **Promise** 消息中是否包含已接受的 **value**
  - ✓ 是 -> 从最大的 **ID2** 中选择 **value**
  - ✓ 否 -> 选择任意的 **value**

Paxos 算法伪代码

☐Phase 1a: Proposer (PREPARE)

A proposer initiates a PREPARE message, picking a unique, ever-incrementing value.

```
ID = cnt++;
send PREPARE(ID)
```

☐Phase 1b: Acceptor (PROMISE)

An acceptor receives a PREPARE(ID) message:

```
if (ID <= max_id)
    do not respond (or respond with a "fail" message)
else
    max_id = ID // save highest ID seen so far
    if (proposal_accepted == true) // a proposal already accepted?
        respond: PROMISE(ID, accepted_ID, accepted_VALUE)
    else
        respond: PROMISE(ID)
```

Paxos 算法伪代码

☐Phase 2a: Proposer (PROPOSE)

The proposer now checks to see if it can use its proposal or if it has to use the highest-numbered one it received from among all responses:

```
did I receive PROMISE responses from a majority of acceptors?
if yes
    do any responses contain accepted values (from other proposals)?
    if yes
        val = accepted_VALUE // value from PROMISE message with the highest accepted ID
    if no
        val = VALUE // we can use our proposed value
    send PROPOSE(ID, val) to at least a majority of acceptors
```

Paxos 算法伪代码

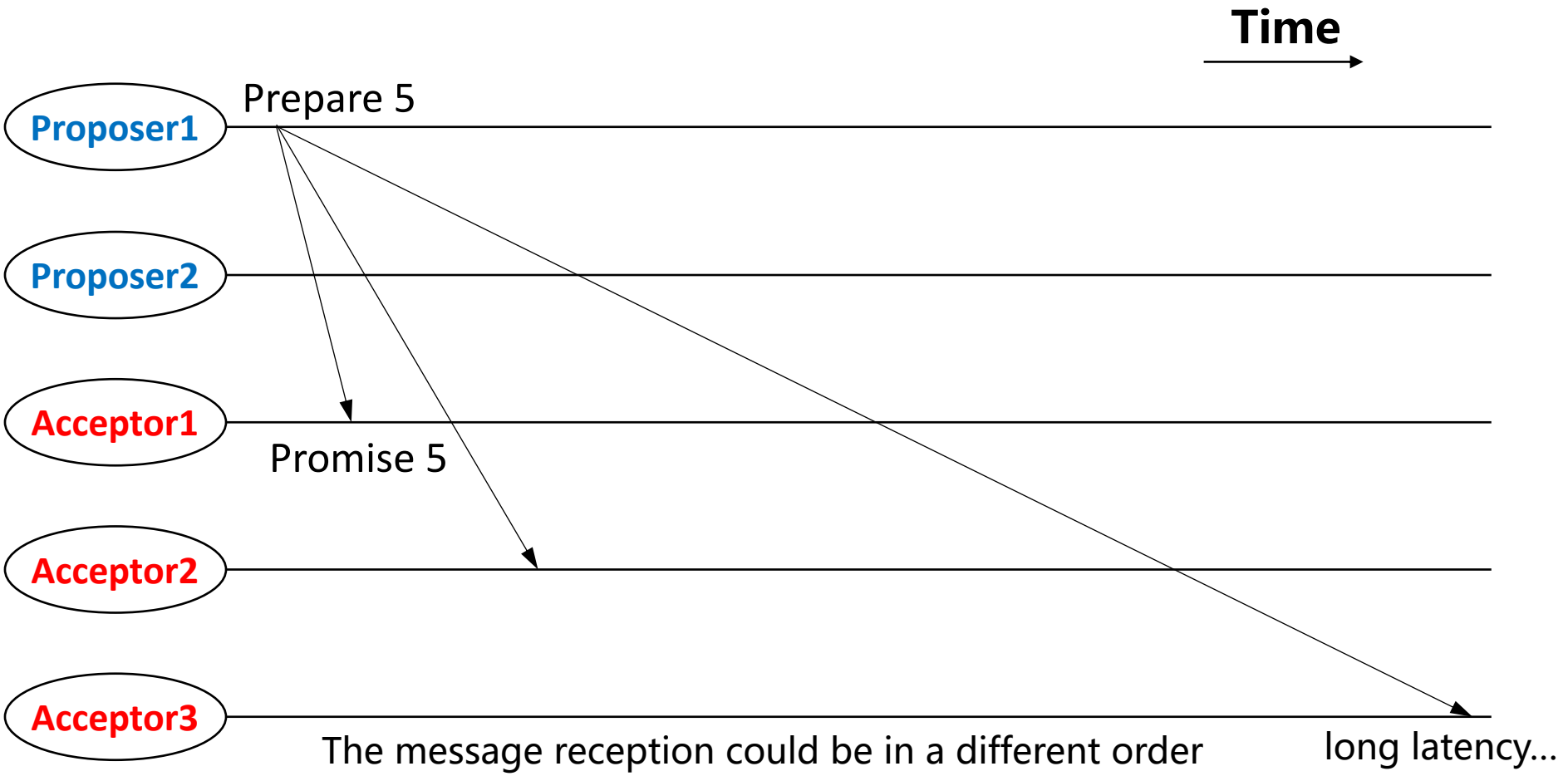
☐Phase 2b: Acceptor (ACCEPT)

Each acceptor receives a PROPOSE(ID, VALUE) message from a proposer. If the ID is the highest number it has processed then accept the proposal and propagate the value to the proposer and to all the learners.

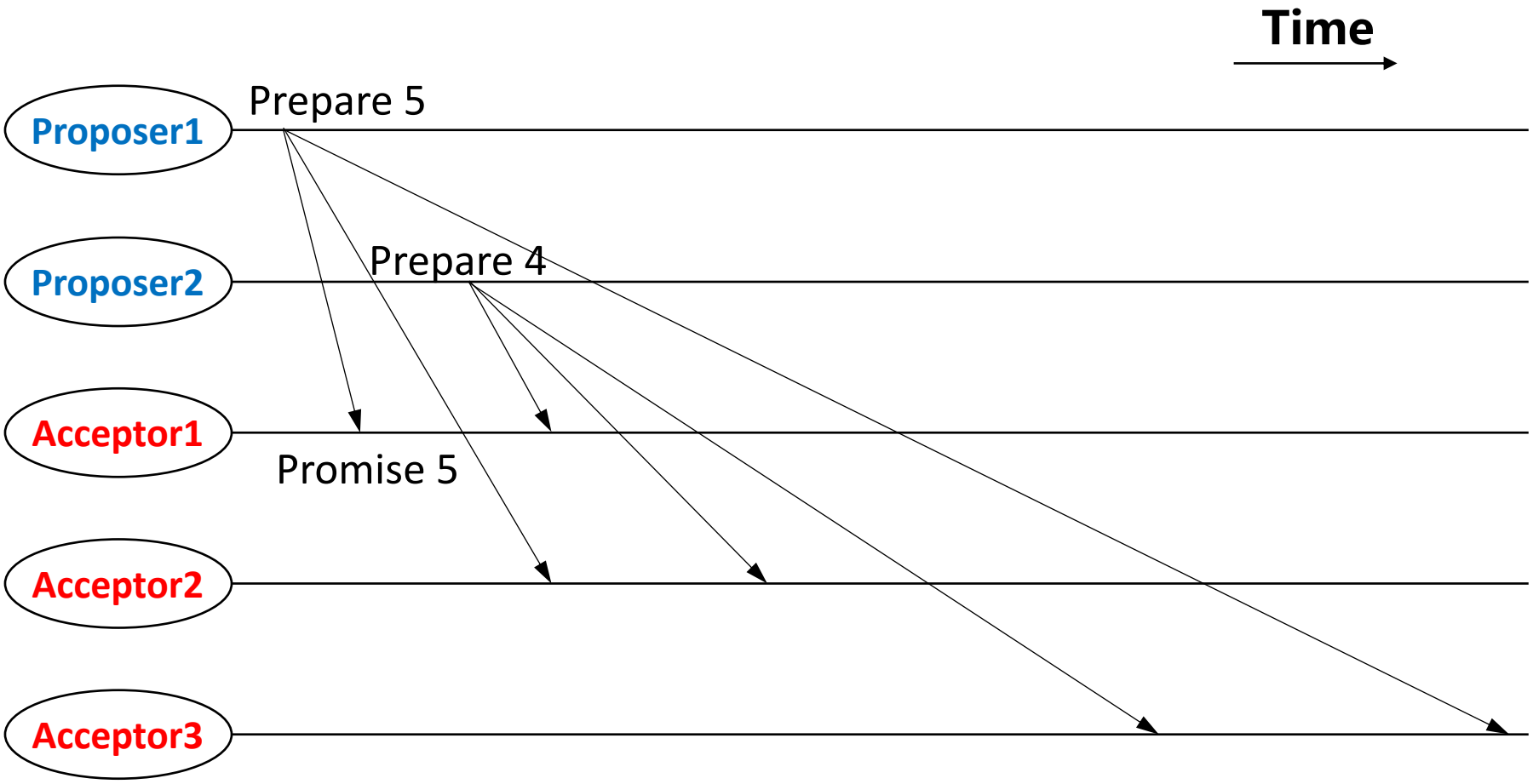
```
if (ID >= max_id) // the ID the largest seen so far?
    proposal_accepted = true // accepted a proposal
    accepted_ID = ID // save the accepted proposal number
    accepted_VALUE = VALUE // save the accepted proposal data
    respond: ACCEPTED(ID, VALUE) to the proposer and all
learners


else
    do not respond (or respond with a "fail" message)
```

Paxos 算法实例

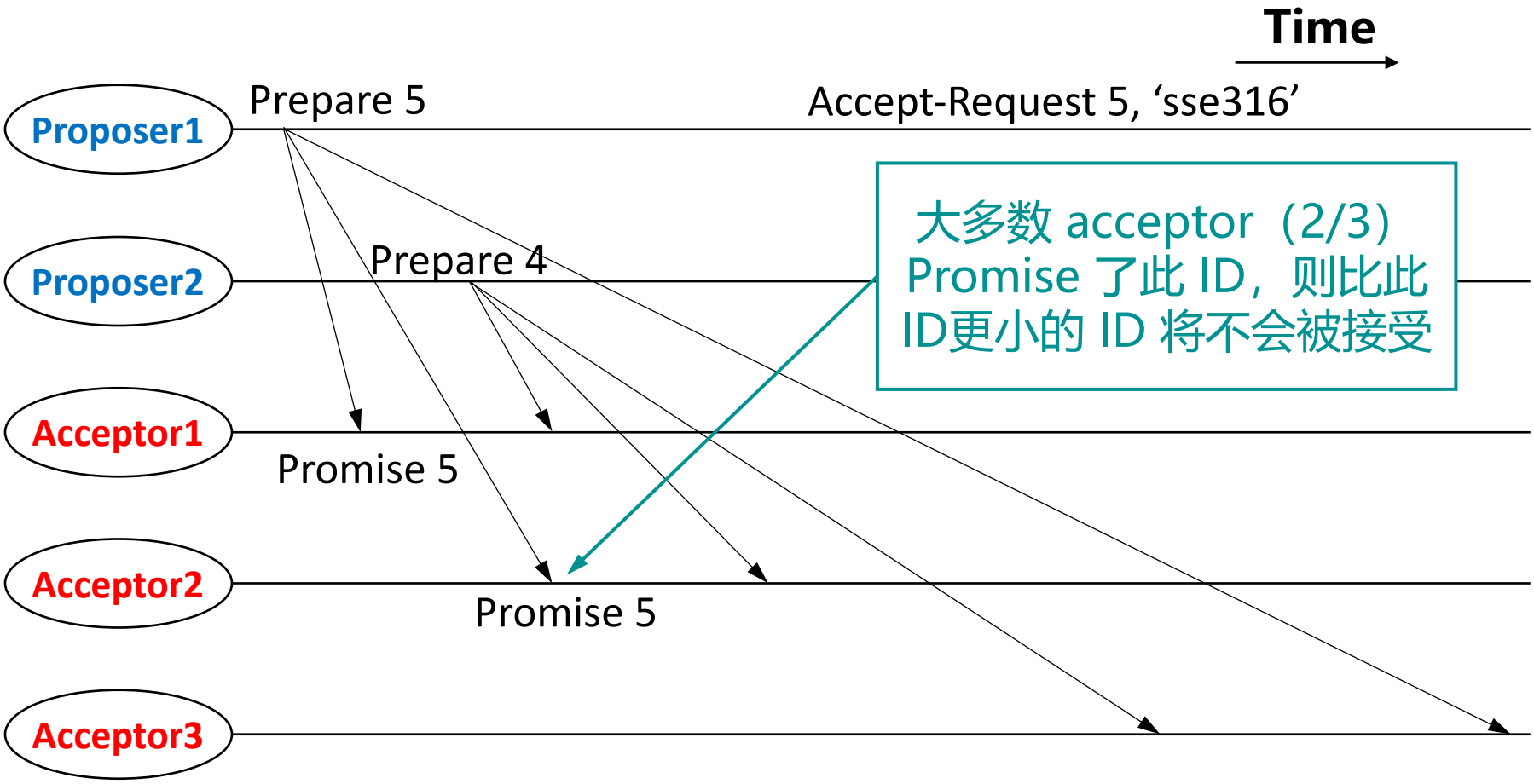
Paxos 算法实例


Time
Proposer1
Prepare 5
Proposer2
Acceptor1
Promise 5
Acceptor2
Acceptor3
The message reception could be in a different order
long latency...

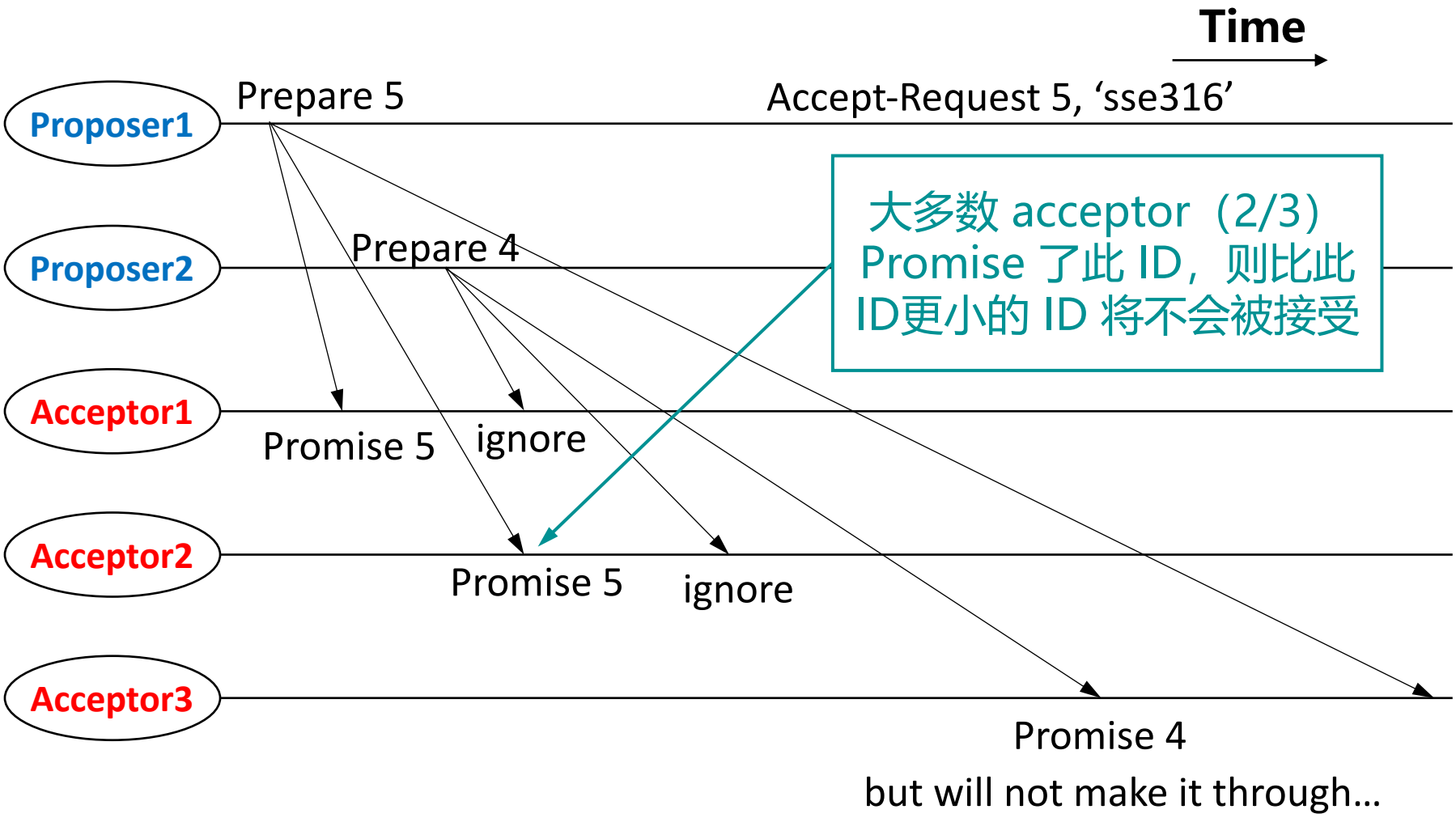
Paxos 算法实例


Time
Proposer1
Prepare 5
Proposer2
Prepare 4
Acceptor1
Promise 5
Acceptor2
Acceptor3

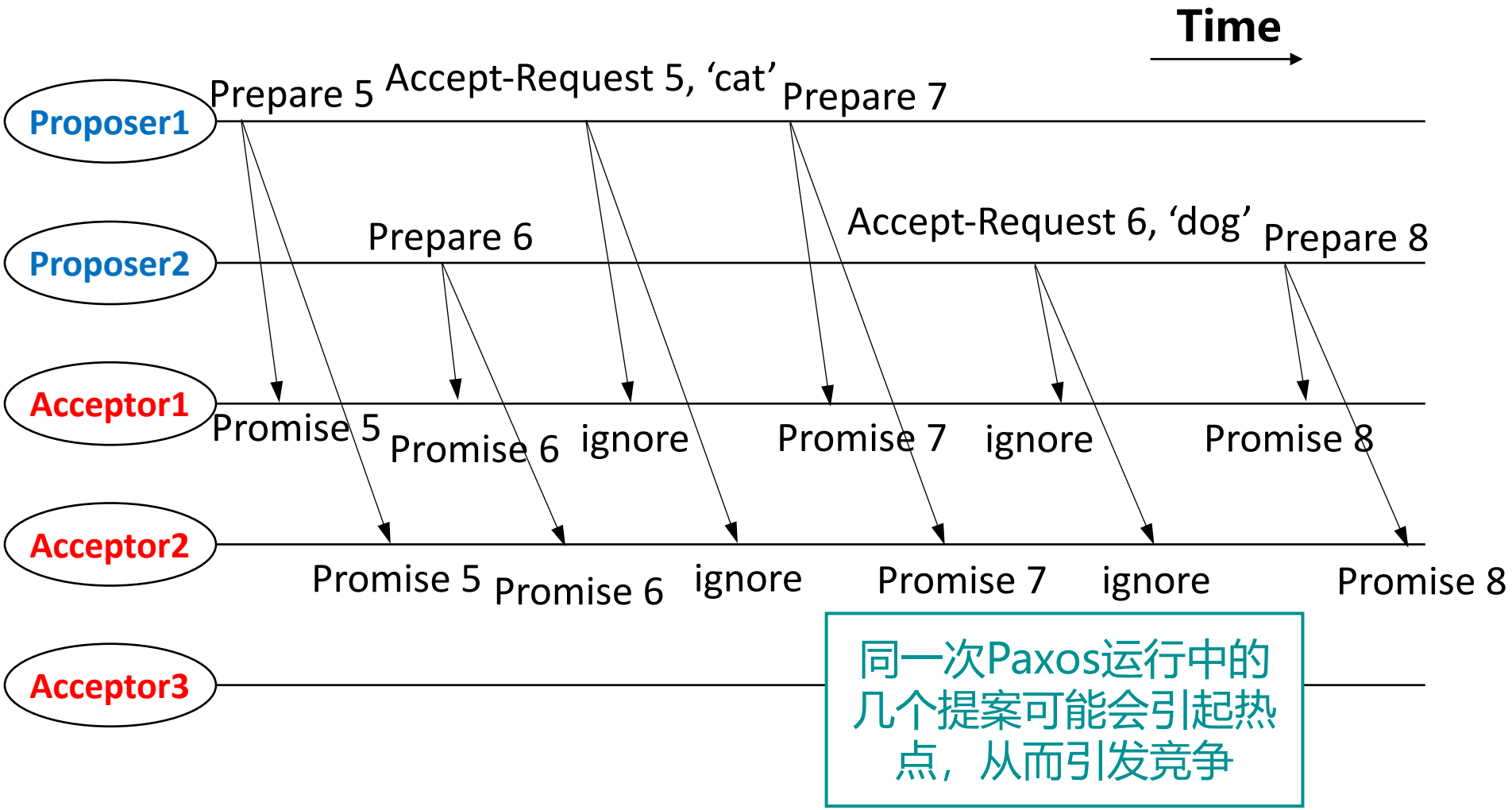
Paxos 算法实例


Time
Proposer1
Prepare 5
Accept-Request 5, ‘sse316’
Proposer2
Prepare 4
大多数 acceptor（2/3）Promise 了此 ID，则比此ID更小的 ID 将不会被接受
Acceptor1
Promise 5
Acceptor2
Promise 5
Acceptor3

Paxos 算法实例


Time
Proposer1
Proposer2
Acceptor1
Acceptor2
Acceptor3
Prepare 5
Accept-Request 5, ‘sse316’
Prepare 4
大多数 acceptor（2/3）
Promise 了此 ID，则比此
ID更小的 ID 将不会被接受
Promise 5
ignore
Promise 5
ignore
Promise 4
but will not make it through…

Paxos 算法 - 竞争



设置指数回退 exponential backoff 避免冲突

Paxos 算法优缺点

□优点：

- **容错性强：** 只要超过半数的 Acceptor 正常工作，就能达成共识
- **异步系统中的正确性：** Paxos 假定消息可能延迟、丢失、重复，但不假定消息顺序或时钟同步（除了提案编号的唯一性和单调递增性）
- **理论基础坚实：** 是许多后续共识算法（如 Raft, Zab）的灵感来源或变种

□缺点

- **复杂性：** 原始的 Paxos 算法难以理解和正确实现
- **活锁可能性：** 在没有 Leader 的情况下，多个 Proposer 竞争可能导致没有提案被接受
- **性能：** 两阶段提交的通信开销相对较大
- **非拜占庭容错：** 不处理恶意节点发送错误信息的情况

从 Paxos 到 Raft：共识算法的演进

□Paxos 的工程困境

- Paxos 虽然理论正确，但极难理解和实现
- Google Chubby 的作者曾说："世界上只有一种共识协议，就是 Paxos"

□Raft 的诞生 (2014)

- Diego Ongaro 设计 Raft 时的首要目标：可理解性
- 将共识分解为：领导者选举 + 日志复制 + 安全性
- 广泛应用于 etcd、Consul、TiKV 等系统

□课后探索：

- **https://raft.github.io/ 可视化演示 Raft 算法**

数据可靠性

Google集群首年的故障情况

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures
slow disks, bad memory, misconfigured machines, flaky machines, etc.

Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

数据丢失的代价

□2017 年 Amazon S3 宕机事件

- 一名工程师的误操作导致 S3 服务中断近 4 小时
- 影响了数千个网站和服务，直接经济损失约 1.5 亿美元

□2021 年 OVHcloud 数据中心火灾

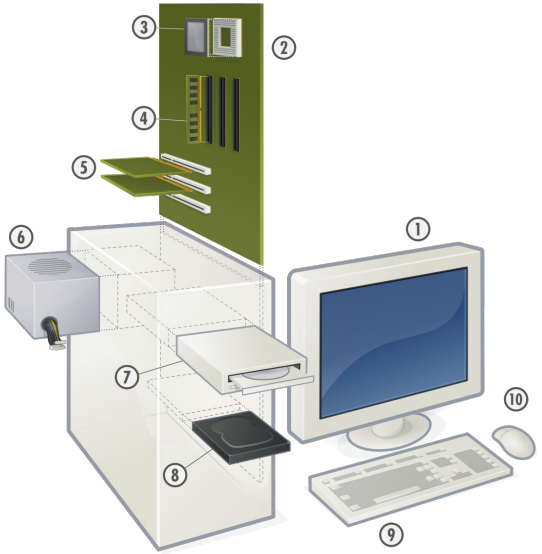
- 法国斯特拉斯堡数据中心起火，约 360 万个网站受影响
- 未做异地备份的客户数据永久丢失

□教训：

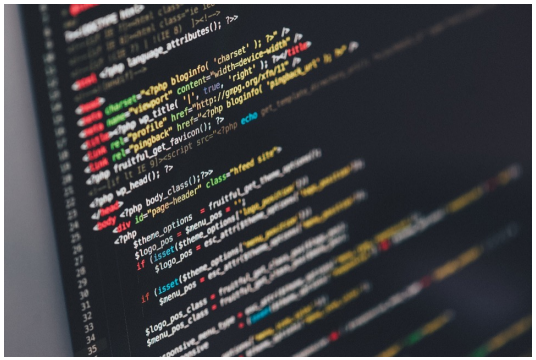
- 任何单一设备/数据中心都不可靠
- 数据可靠性需要系统级的设计保障

数据可靠性

硬件故障难以避免，需要软硬件共同提高数据可靠性



+



如何避免损失数据?

数据备份
Data Replication

纠删码
Erasure Coding

为什么需要数据备份?

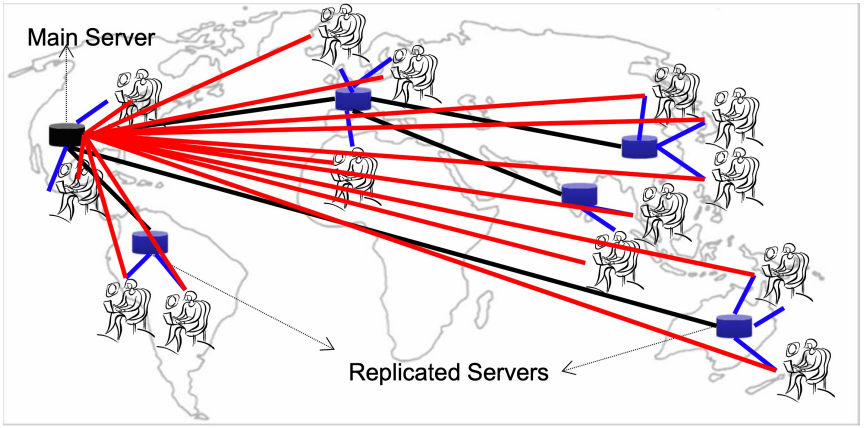
数据备份是在多台计算机上维护多份相同数据的过程

提升性能

客户端可以就近访问附近的数据副本

增强系统可扩展性

对数据的请求可以分发到多个包含数据副本的服务器



为什么需要数据备份?

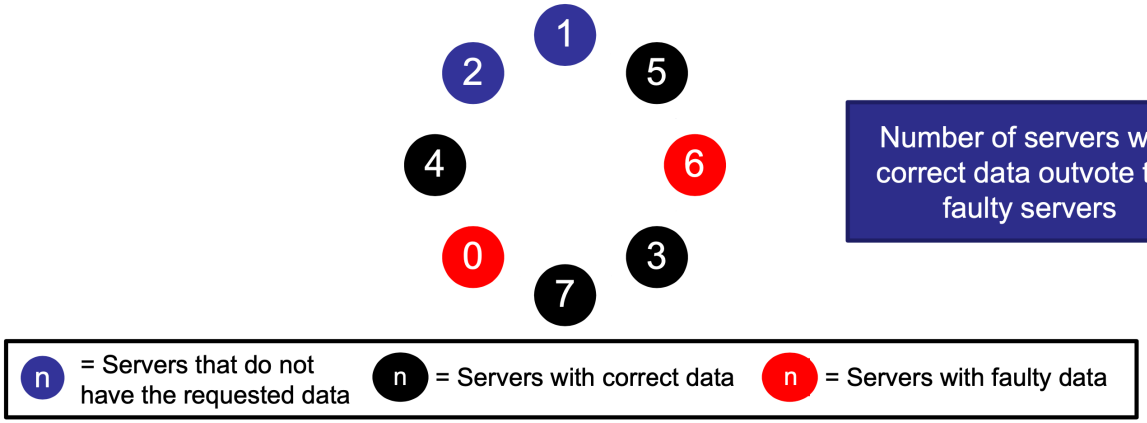
增加服务的可用性

备份可以掩盖诸如服务器崩溃和网络断开连接之类的故障

防止恶意攻击

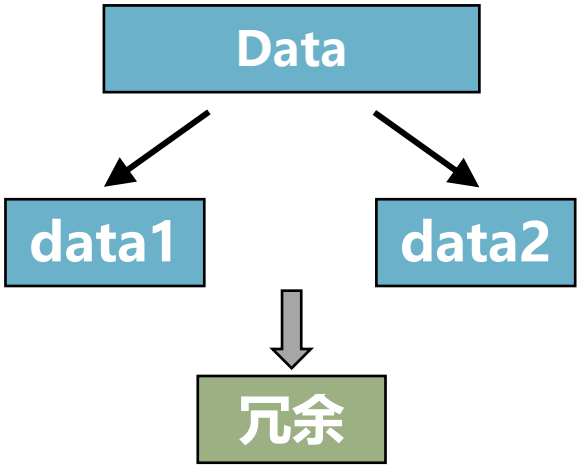
即使某些副本是恶意的，也可以依赖未受损服务器上的副本来保证数据安全

数据备份虽然简单，但需要消耗大量存储资源



如果持有数据的服务器中有少数是恶意的，那么非恶意服务器可以投票超过恶意服务器，从而提供安全性

纠删码



纠删码将数据分解成多个片段

通过数学算法生成额外的冗余片段

将片段分布在多个存储节点或设备上

即使其中一些片段丢失或不可用，也可以利用其他片段重建原始数据

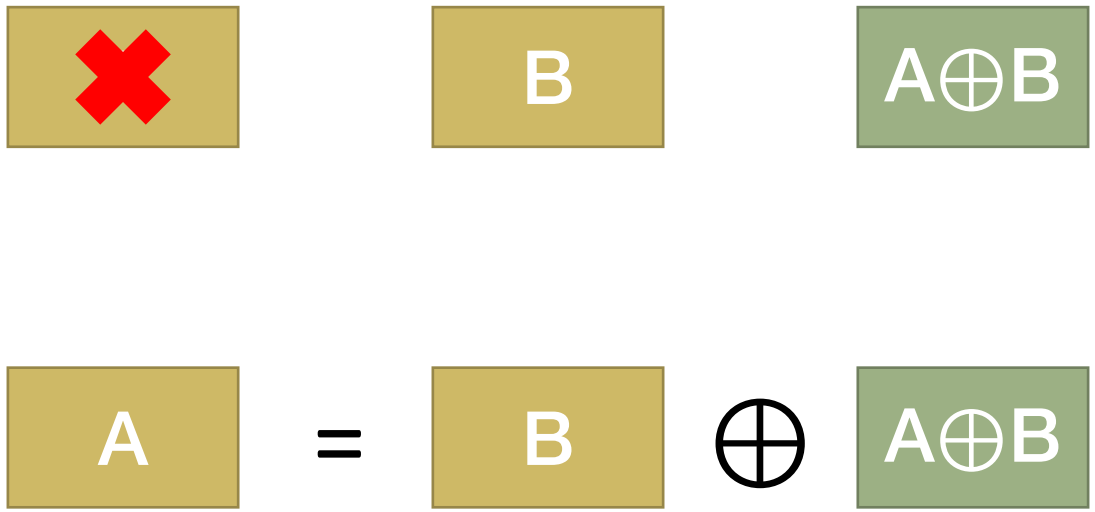
一个简单的纠删码例子（1）

异或

$$0 \oplus 0 = 0$$
$$1 \oplus 1 = 0$$
$$1 \oplus 0 = 1$$
$$0 \oplus 1 = 1$$

一个简单的纠删码例子（2）

一个简单的纠删码例子（3）

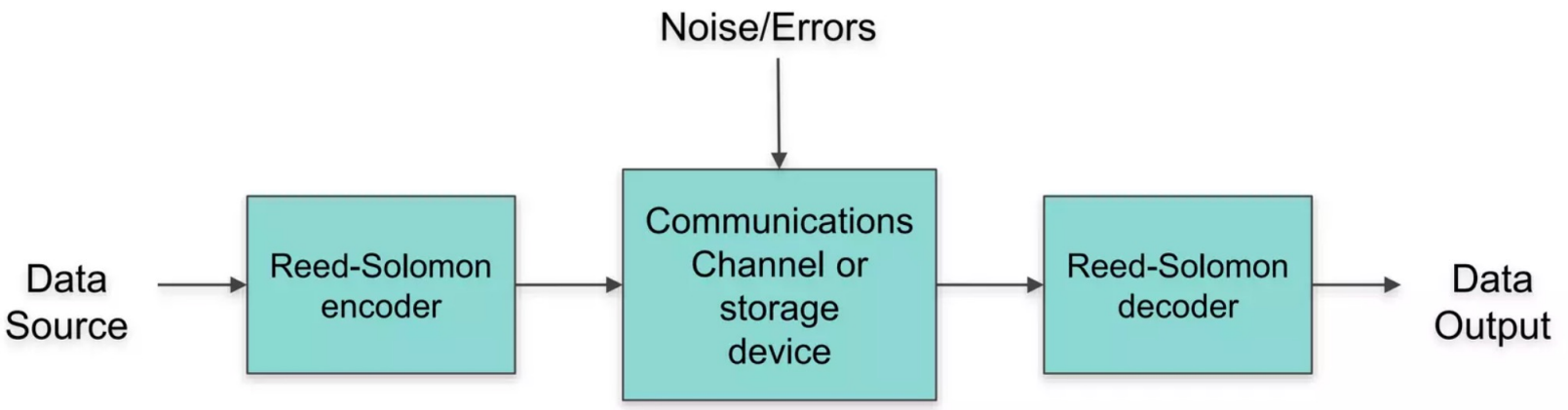


里德所罗门码

里德所罗门码（Reed-Solomon Codes, RSC）是基于块的纠错码，在存储和数字通信中有大量应用：

- ✓ 存储设备（包括磁带、光盘、DVD、条形码等）
- ✓ 无线或移动通信（包括手机、微波连接等）
- ✓ 卫星通信
- ✓ 高速调制解调器，如 ADSL、xDSL 等
- ✓ …

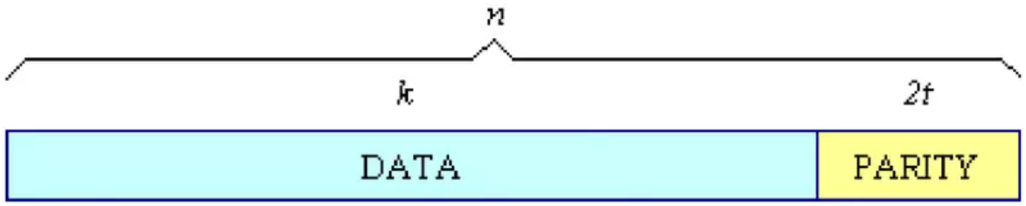
RSC 结构



- ✓ RSC 编码器（Reed-Solomon encoder）接收一块数据并添加额外的冗余位
- ✓ 数据在传输或存储过程中可能出现错误（如噪声或干扰，CD 上的划痕）
- ✓ RSC 解码器（Reed-Solomon decoder）处理每个块，试图纠正错误，回复原始数据

RSC 算法

- Reed-Solomon 码定义为 $RS(n,k)\ with\ s\ bit\ symbols$
- 意味着
  - ✓RSC 编码器接收 k 个 data symbols，每个 symbol 有 s 个 bits
  - ✓添加 parity symbols，获得长度为 n 的 symbol codeword



  - ✓RSC 解码器可纠正 codeword 中最多 t 个 symbol 错误（errors, 位置未知），其中 $2t \le n-k$
  - ✓解码所需的算力与 codeword 内的 parity symbols 数量有关，更大的 t 可以纠正更多的错误，也需要更多的算力
  - ✓RSC 的最大 codeword 长度为 $n = 2^s - 1$

RSC 例子

$$RS(255, 223)\ with\ 8\ bit\ symbols$$

即每个 codeword 包含 255 个 word bytes，其中 223 个 bytes 为 data，其余 32 个 bytes 为 parity

$$n = 255, k = 223, s = 8$$
$$2t = 32, t = 16$$

解码器可以更正 16 个 symbol errors

- ✓ 最少的情况：16 个 bit errors，分别发生在不同的 symbol，可纠正个 16×1 个 bit errors
- ✓ 最多的情况：16 个 symbol 内的 bit 全错了，可纠正 16×8 个 bit errors

RSC 解码

Reed Solomon 代数解码的过程中，可以纠正
✓ t 个错误（error）：不知道错误 symbol 的位置
✓ $2t$ 个擦除（erasure）：知道错误 symbol 的位置

对 codeword 解码时，有三种可能

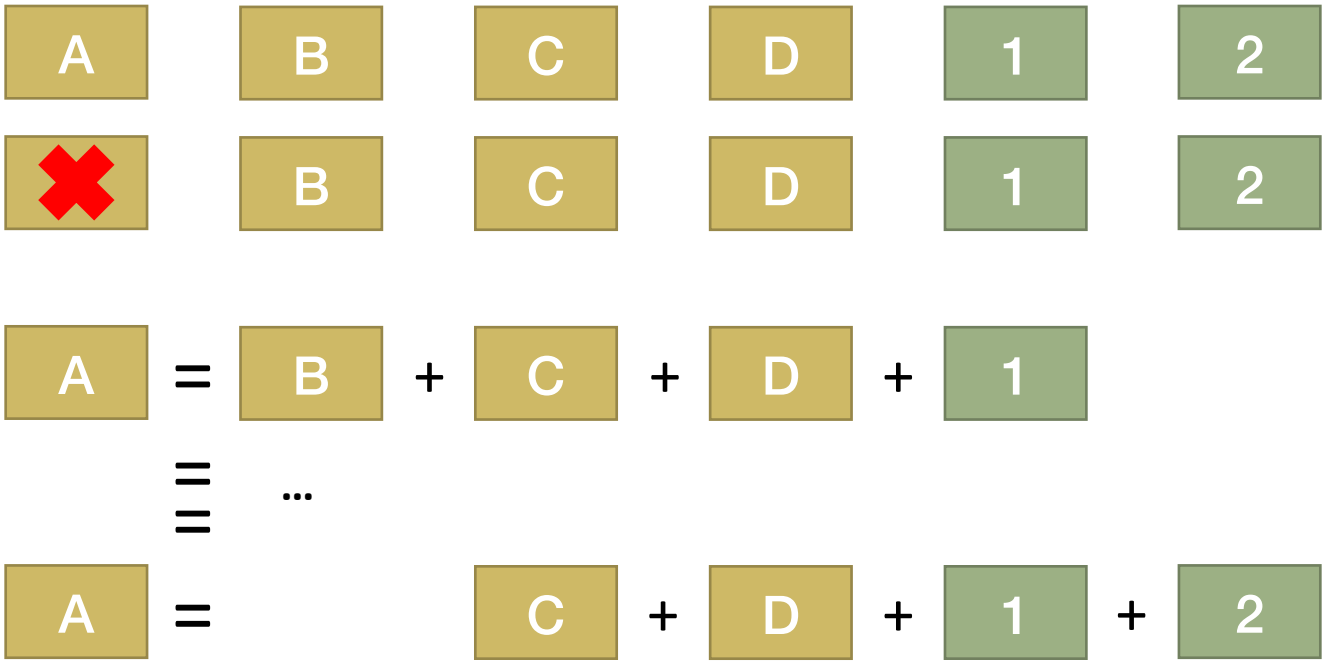
1. 若 $2e + r \leq 2t$，即 e 个error，r 个erasure，则总可以恢复原始数据
2. 解码器发现其无法恢复原始数据，并进行说明
3. 当错误数量超过纠错/擦除能力时，解码器可能不报错，而是错误地将接收数据解码为另一个合法 codeword，最终输出错误结果

以上 3 种情况发生的概率取决于具体的 Reed-Solomon 码、error 数量以及 error 分布情况。

RSC 解码

当擦除故障发生时，只要有至少任意 k 个正确的 data symbol（来自编码后的 n 个 symbol），即可恢复原来的数据

以 $RS(6,4)$ 为例，任意 4 个正确的 data symbol 即可恢复数据



编码增益

使用 Reed-Solomon 码的优势在于，解码后数据内包含错误的概率，通常远低于未使用 Reed-Solomon 码的错误发生概率

数字系统中设计的 Bit Error Ratio (BER) 为 10^{-9}，即每接收 10^{9} 个bits，最多有 1 个 bit 出错

✓ 可通过增加发射器的功率实现，或
✓ 使用 Reed-Solomon（或其他前向纠错码）实现

使用纠错码能使发射器以较低的功率实现 BER 目标，由此节约的 power 称为 coding gain

存储开销

- 容忍 2 个存储故障
  - ✓需要__个数据备份，存储开销为__倍
  - ✓RS(6, 4) 的存储开销为__倍
- 容忍 4 个存储故障
  - ✓需要 __个数据备份，存储开销为__倍
  - ✓RS(14, 10) 的存储开销为__倍
  - ✓RS(104, 100) 的存储开销为__倍

编码开销

- 数据备份
  - ✓简单的数据读取
- 里德-所罗门码
  - ✓每一个 parity symbol (一共 $2t$ 个) 均需要基于 k 个 data symbol 计算编码

解码开销

- 数据备份
  - ✓简单的数据读取
- 里德-所罗门码
  - ✓多数情况下没有故障，只需要读取相应的 data symbol
  - ✓如果出现故障
    - 通过网络从存储中读取 k 个 data symbol
    - 基于 k 个 data symbol 恢复故障 symbol

更新开销

- 数据备份
  - ✓更新每一个备份中的数据
- 里德-所罗门码
  - ✓更新相应 data symbol 中的数据
  - ✓更新所有的 parity symbol

删除开销

- 数据备份
  - ✓在所有备份中删除数据
- 里德-所罗门码
  - ✓删除相应 data symbol 中的数据
  - ✓更新所有的 parity symbol

RSC 实现

RSC 数学原理：https://www.diva-portal.org/smash/get/diva2:1455935/FULLTEXT01.pdf

RSC 算法实现：https://github.com/tomerfiliba/reedsolomon

中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

谢谢

陈壮彬

软件工程学院

chenzhb36@mail.sysu.edu.cn