中山大學 软件工程学院

SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

SSE316：云计算技术
Cloud Computing Technology

陈壮彬

软件工程学院

https://zbchern.github.io/sse316.html

云数据存储

- 文件共享语义
- 数据一致性
- 数据容错

云数据存储

- 文件共享语义
- 数据一致性
- 数据容错

SSE36课程论文

- 假设课程论文改成团队协作完成

SSE316课程论文

如何管理大家写的部分并最终合并成一篇完整的课程论文？

文件共享语义

多个用户可以同时访问共享文件，因此需要明确定义用户对文件数据的修改何时可以被其他用户观察到。这是由文件系统所采用的文件共享语义（File Sharing Semantics）定义的。

常见的文件共享语义类型

Unix语义
Unix Semantics

会话语义
Session Semantics

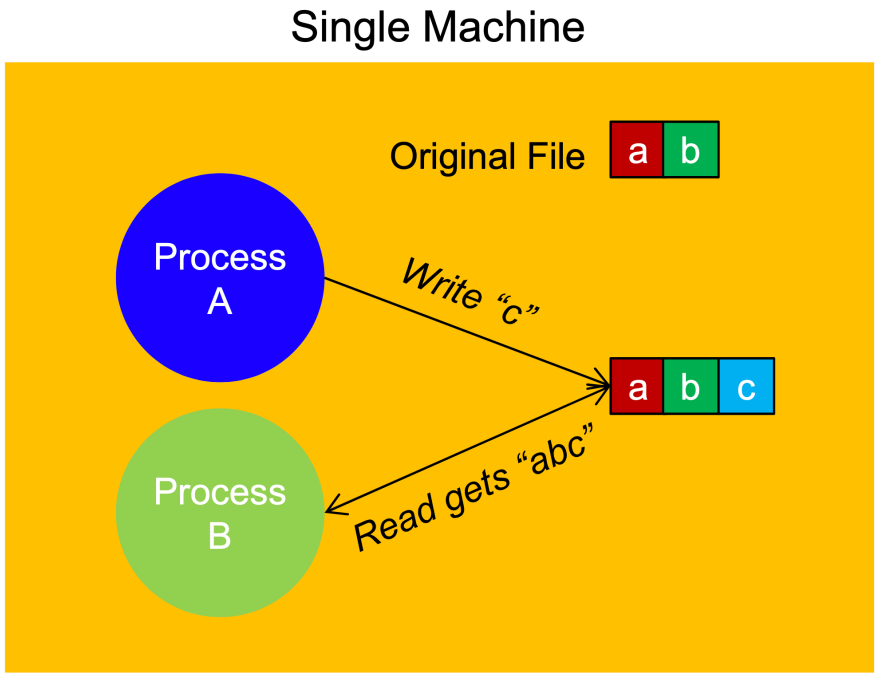
不可变语义
Immutable Semantics

纸质论文共享

- 纸质论文只有一份，一个人写完传给下一个

Unix语义

Unix语义对所有操作强制执行绝对时间顺序，并确保文件上的每个读取操作都能看到以前对该文件执行的所有写入操作的效果



DFS中的Unix语义

- 在DFS中，如果只有一个文件服务器并且客户端不缓存文件，则可以很容易地实现Unix语义

- 因此，所有的读写操作都直接进入文件服务器，由文件服务器严格按顺序进行处理

- 但是，由于所有文件请求都必须发送到单个服务器，性能可能会下降

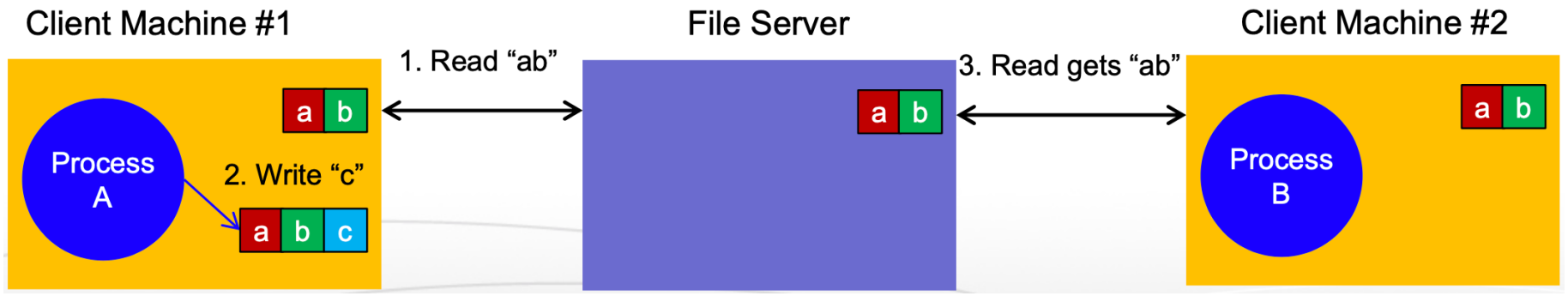
电子文档共享

- 工具升级了，使用word，每个同学可以同时写各自的部分

缓存与Unix语义

- 具有单个文件服务器和Unix语义的DFS的性能可以通过缓存（Caching）来提高

- 但是，如果一个客户端在本地修改缓存文件，并且很快另一个客户端从服务器读取该文件，那么它将得到一个过时的文件



会话语义（1）

- 避免过时文件的一种方法是立即将对缓存文件的更改传播回服务器
- 这种方法非常复杂
- 另一种解决方案是放宽文件共享的语义

Session Semantics

对文件的更改在会话中仅对进程可见。只有当文件关闭时，更改才会对其他进程可见。

会话语义（2）

- 使用会话语义会引发一个问题，即如果两个或多个客户端同时缓存和修改同一文件会发生什么？

- 合并更改
  - ✓ 冲突检测：当多个用户同时修改同一个文件时，如果他们进行重叠或矛盾的更改，可能会出现冲突。系统必须在合并过程中检测到这些冲突。
  - ✓ 冲突解决：系统必须自动或通过用户干预来解决冲突。自动冲突解决策略可能包括选择最新更新，复杂的冲突可能需要手动干预。

托管文档

- 最终版本可能由某个同学负责，太忙了想换成第三方托管
- 每个同学建一个branch自由修改自己的部分

不可变语义（1）

- DFS中文件共享语义的另一种方法是使所有文件都不可变（immutable）

- 使用不可变语义，无法打开文件进行写操作，写文件需要创建一个全新的文件

不可变语义（2）

- 文件访问：当用户访问一个文件时，他们可以通过指定其唯一标识符来检索该文件的任何版本。这允许用户同时处理不同版本的文件，而不需要冲突或合并操作。

- 版本管理：为了管理一个文件的多个版本，使用不可变文件的系统通常使用版本控制技术，如版本树或版本向量。这些技术有助于跟踪不同版本之间的关系。

- 存储开销：使用不可变文件的主要缺点是增加了存储开销，因为文件的每个版本都必须单独存储。但是，可以使用delta编码、压缩或垃圾收集等技术来消除未使用的版本。

云数据存储

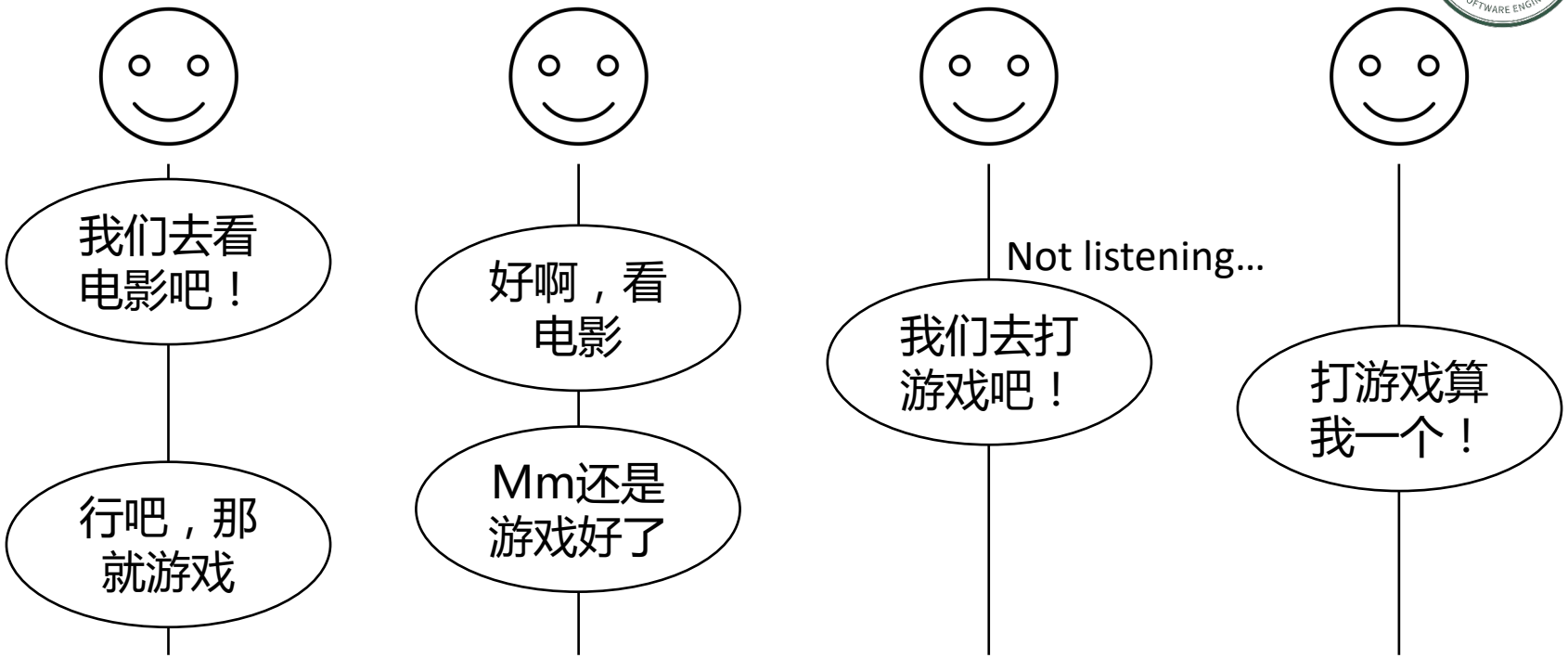
- 文件共享语义
- 数据一致性
- 数据容错

Paxos算法

Paxos算法解决分布式系统的一致性问题，
即一个分布式系统中的各个进程如何就
某个值(决议)达成共识。

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

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，而不仅仅就他们的提议达成一致
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

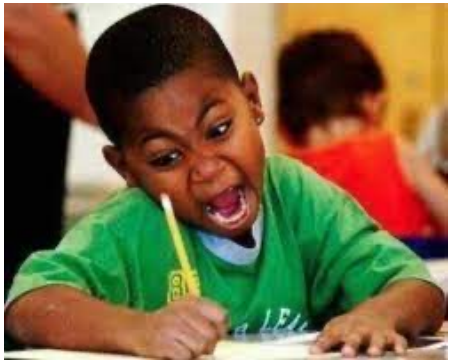
Paxos基础

- Paxos定义了三种角色
  - ✓提议者（proposer）：提出提案让大家讨论
  - ✓接受者（acceptor）：参与提案讨论使共识达成
  - ✓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

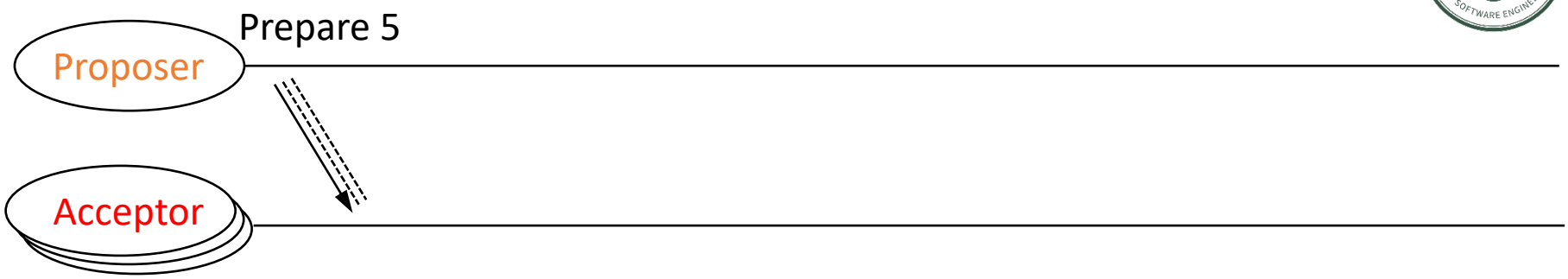
Paxos基础

- Paxos定义了三种角色
  - ✓提议者（proposer）：提出提案让大家讨论
  - ✓接受者（acceptor）：参与提案讨论使共识达成
  - ✓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

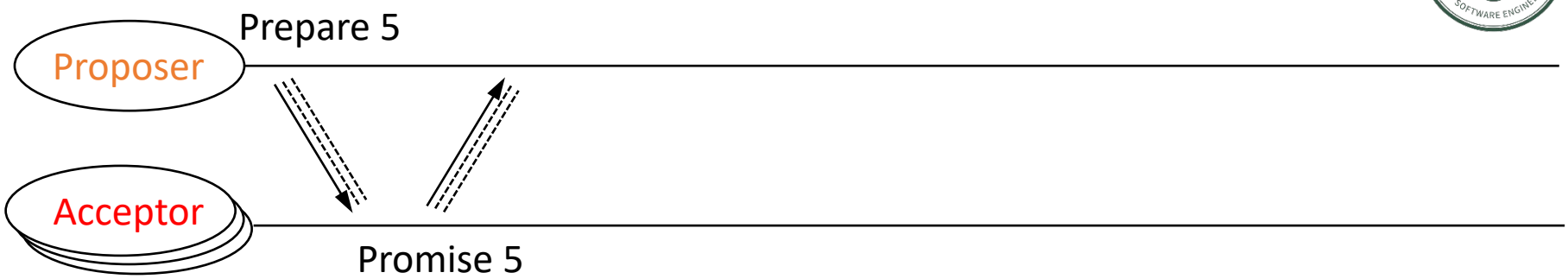
Paxos算法



Proposer发起提案

- 发送**Prepare ID**给多数(或所有)Acceptor
- **ID**在多个Proposer中唯一
  - ✓ 即若有其他Proposer将使用不同的**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

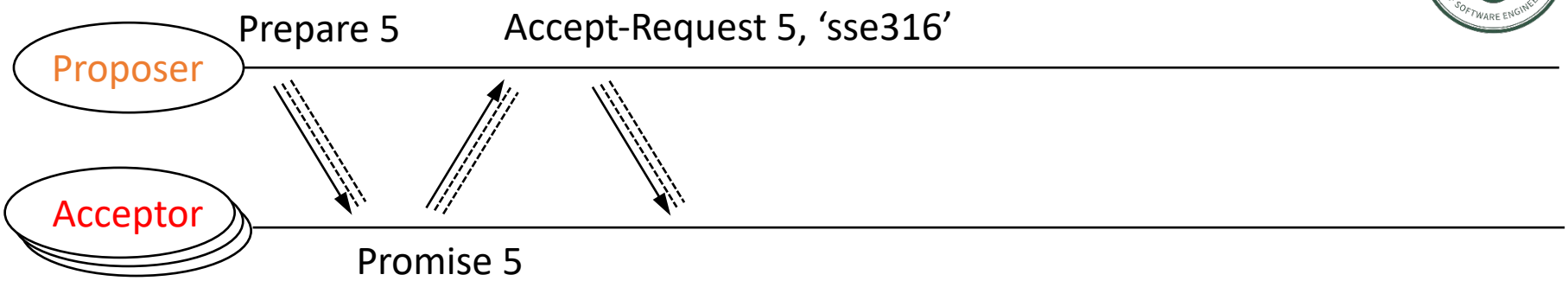


Acceptor接收到提案**Prepare ID**

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

若大多数Acceptor接受了某个**ID**，其他更小的**ID**将不会被接受

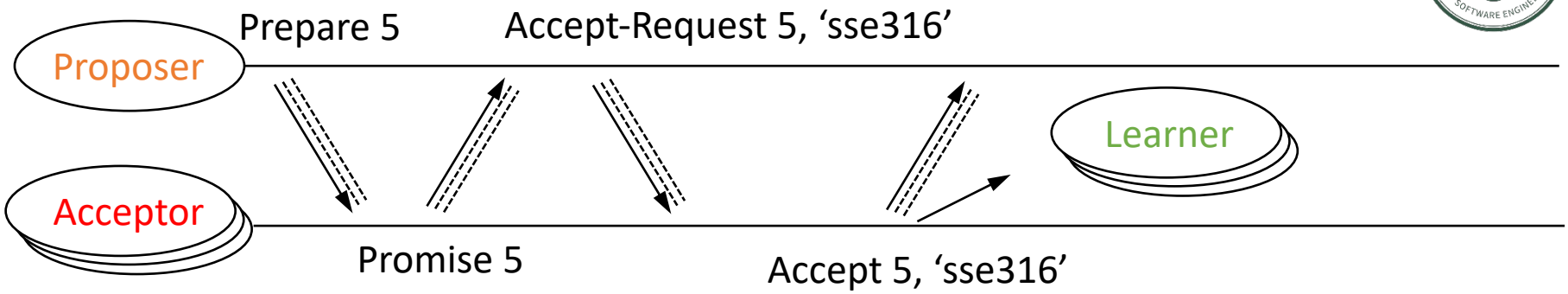
Paxos算法



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

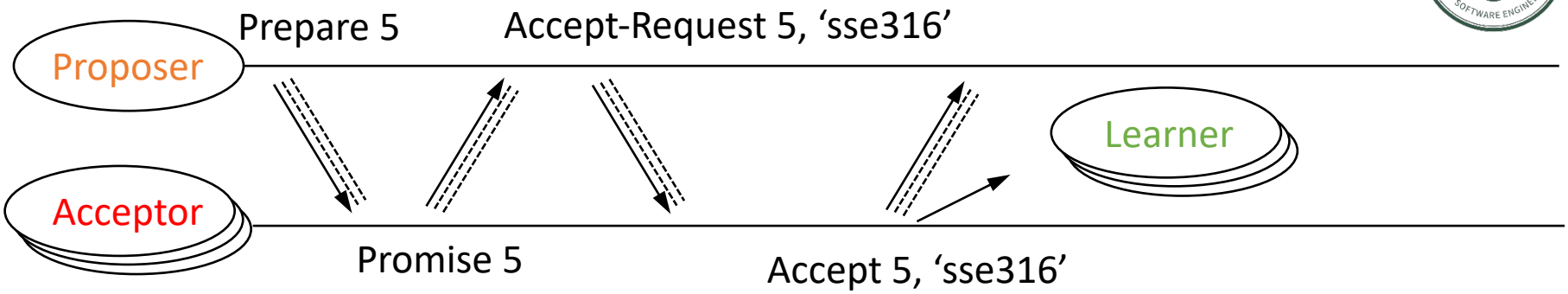


Acceptor收到**Accept-Request ID, value**

- 判断是否承诺忽略此ID
  - ✓ 是 -> 忽略
  - ✓ 否 -> 回复**Accept ID, value**并发送给所有Learner

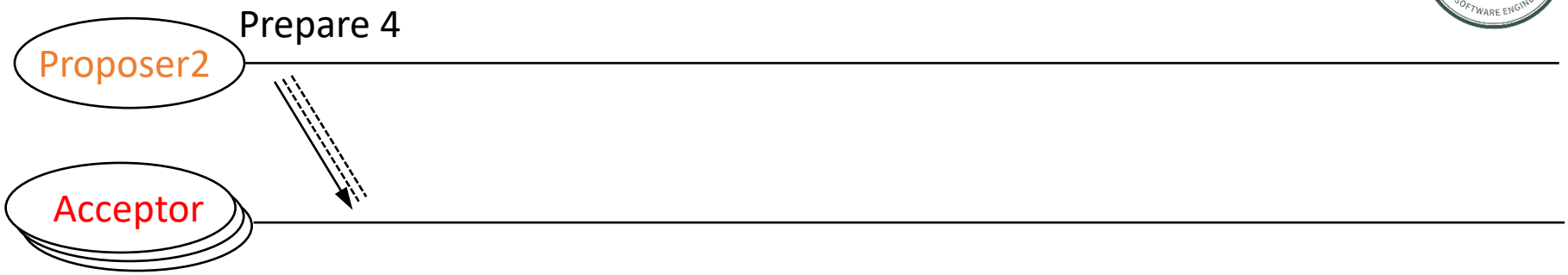
若大多数Acceptor接受了某个**ID, value**，则共识达成！
共识达成的是**value**，而不一定是**ID**，**ID**可能会变

Paxos算法

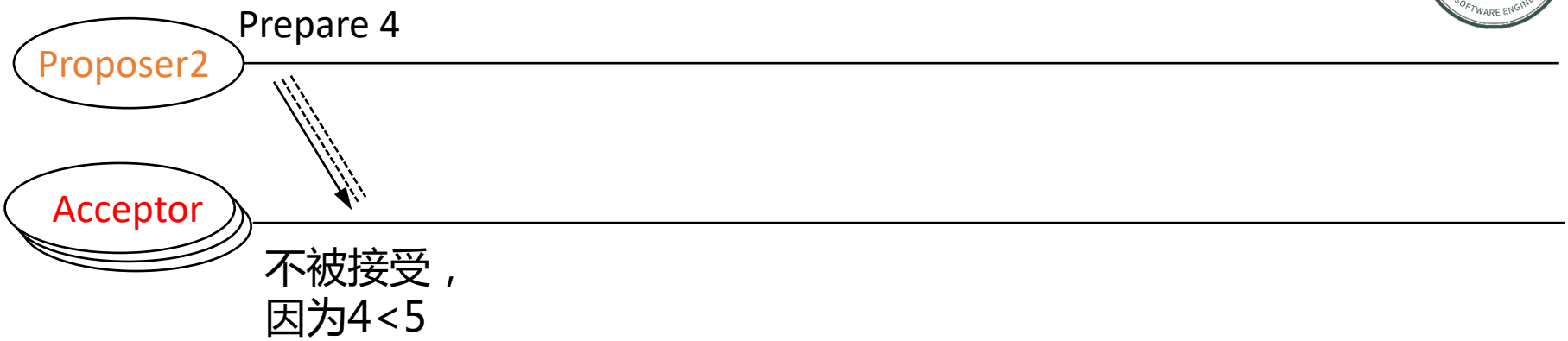


Proposer或者Learner接收到**Accept ID, value**信息
如果一个Proposer或者Learner知道**ID**已被大多数接受，
则他们知道已达成共识**value**（而不是**ID**）

Paxos算法



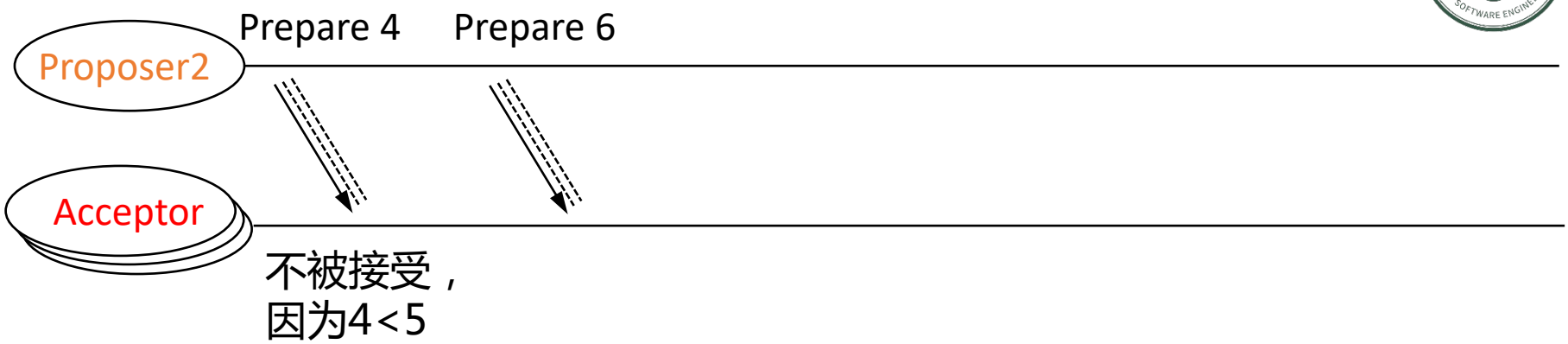
Paxos算法



Acceptor接收到提案**Prepare ID2**

- 判断是否承诺忽略此**ID2**
  - ✓ 是 -> 忽略（已接受5）
  - ✓ 否 -> 将承诺忽略比此**ID2**更小的提案
    (?) 回复**Promise ID**消息

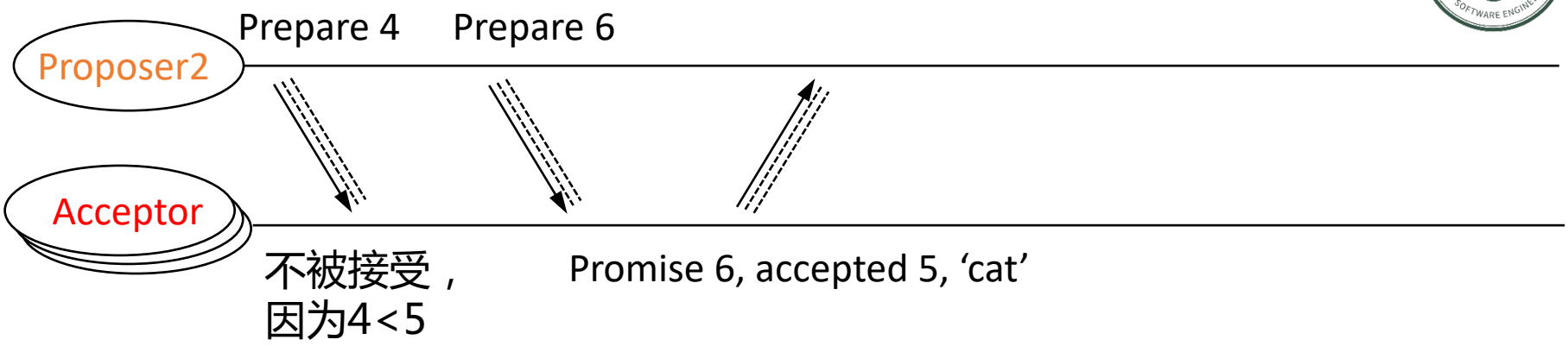
Paxos算法



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

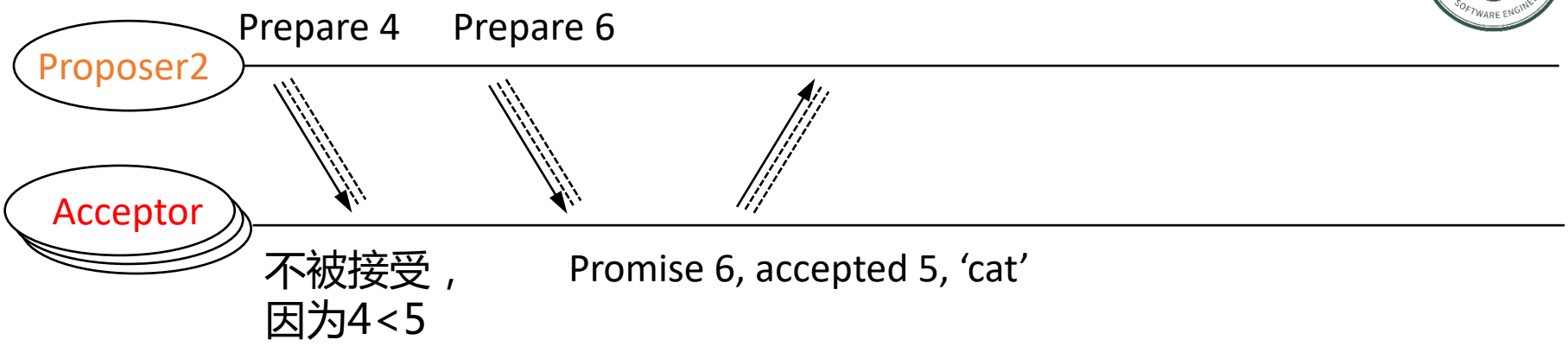


Acceptor接收到提案**Prepare ID2**

- 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

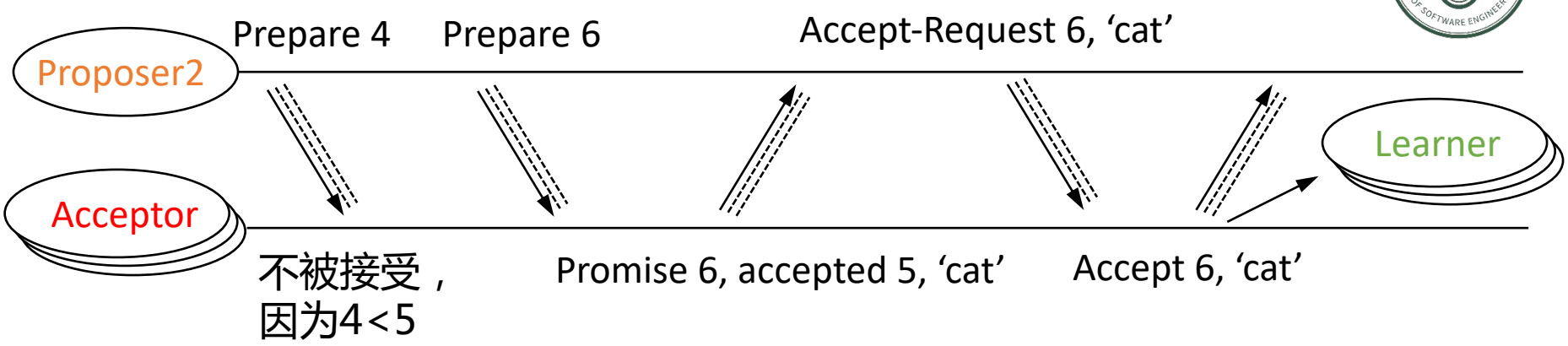
Paxos算法



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



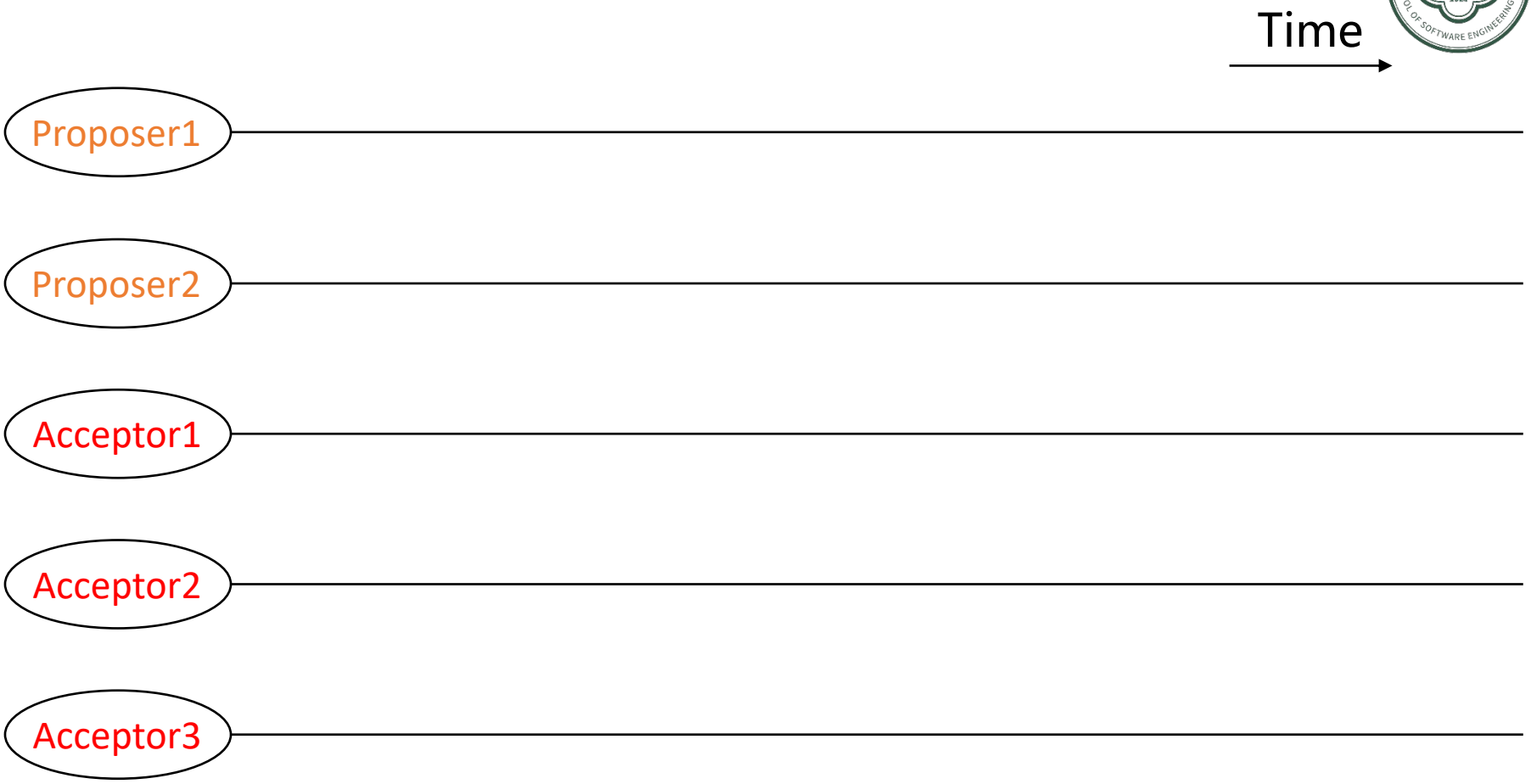
Proposer收到的大多数Promise回复均为某个**ID**

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

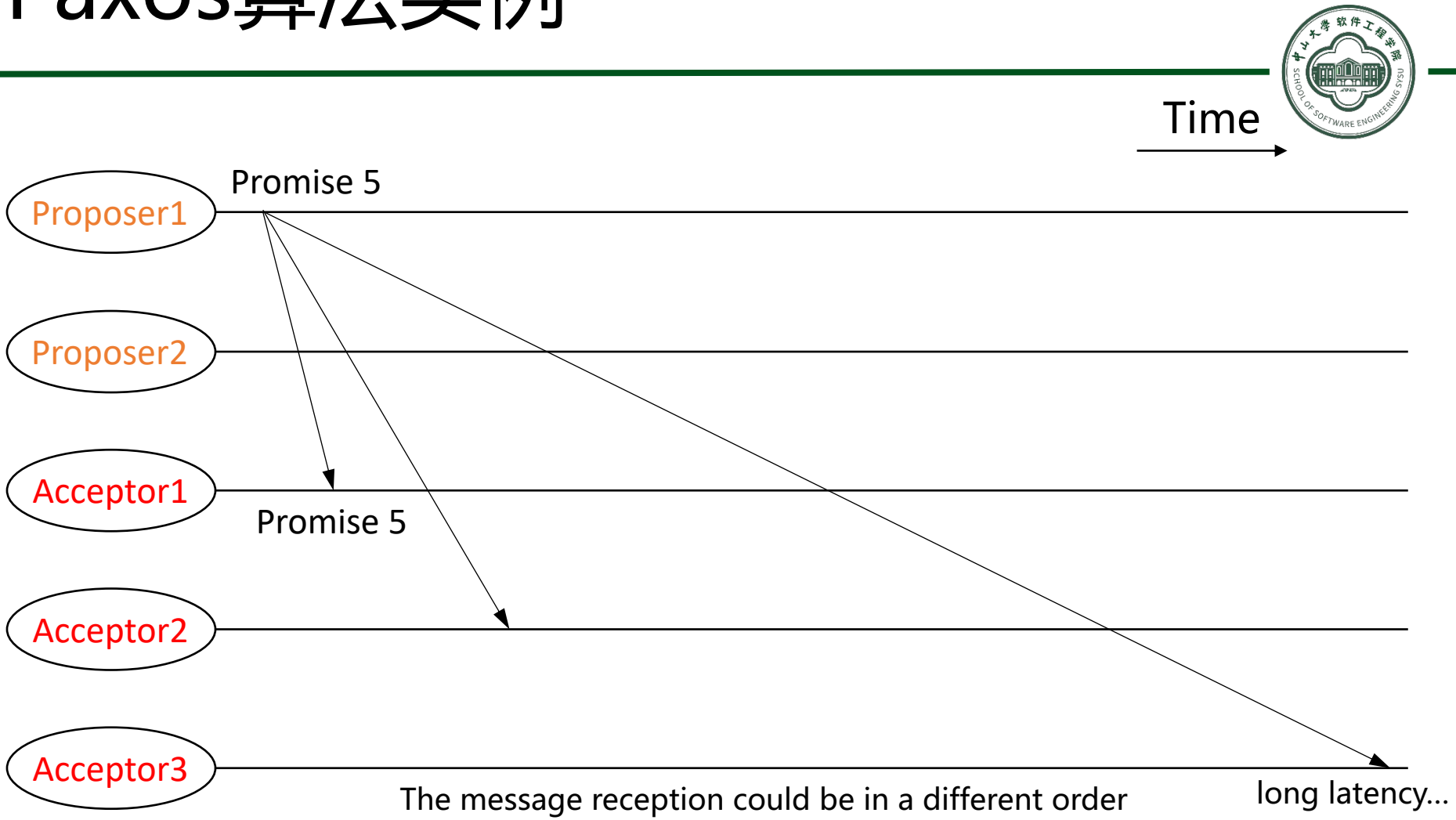
- 判断**Promise**消息中是否包含已接受的**value**
  - ✓ 是 -> 从最大的**ID2**中选择**value**
  - ✓ 否 -> 选择任意的**value**

Paxos算法实例

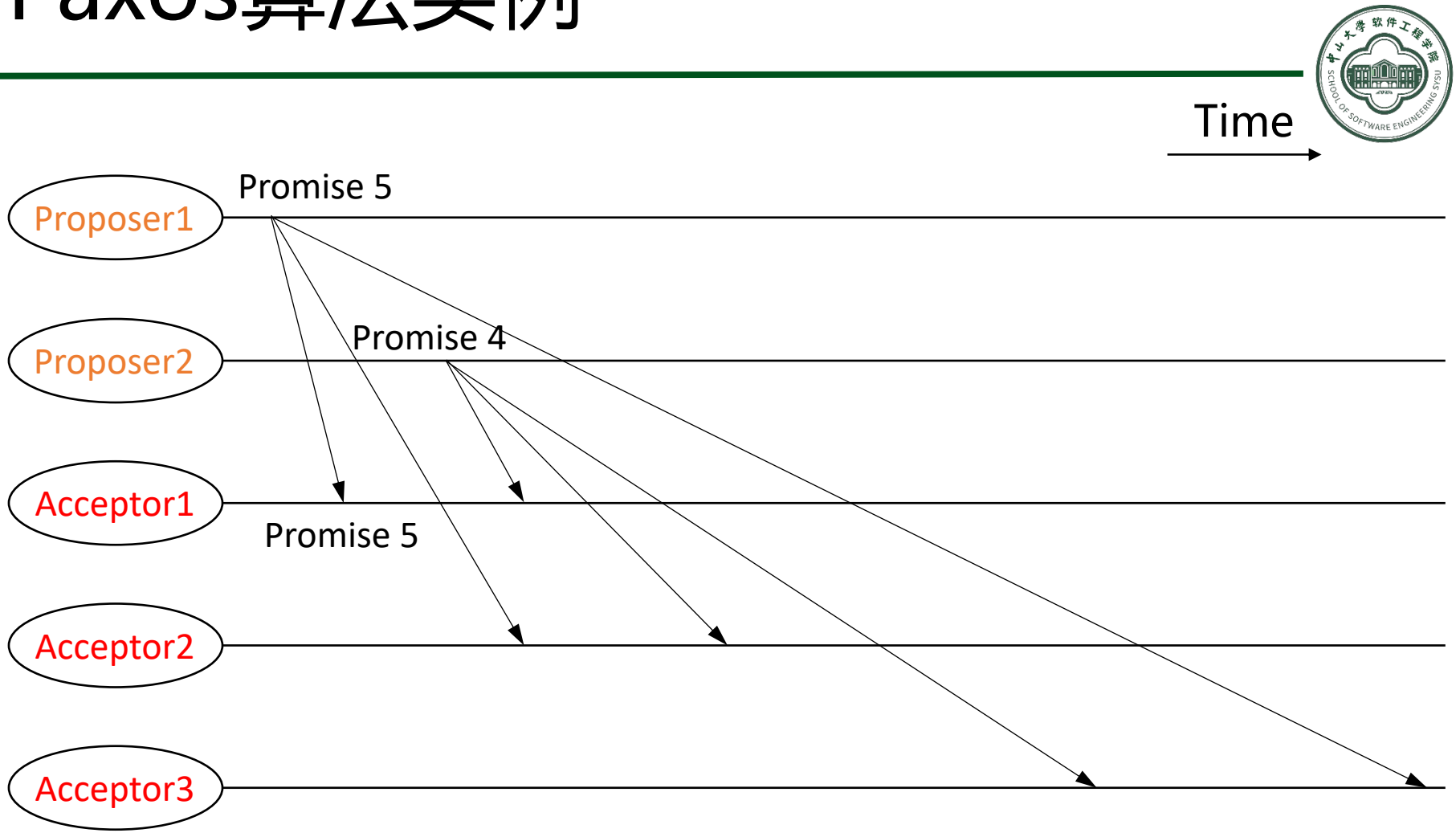

SCHOOL OF SOFTWARE ENGINEERING SYSU


Time
Proposer1
Proposer2
Acceptor1
Acceptor2
Acceptor3

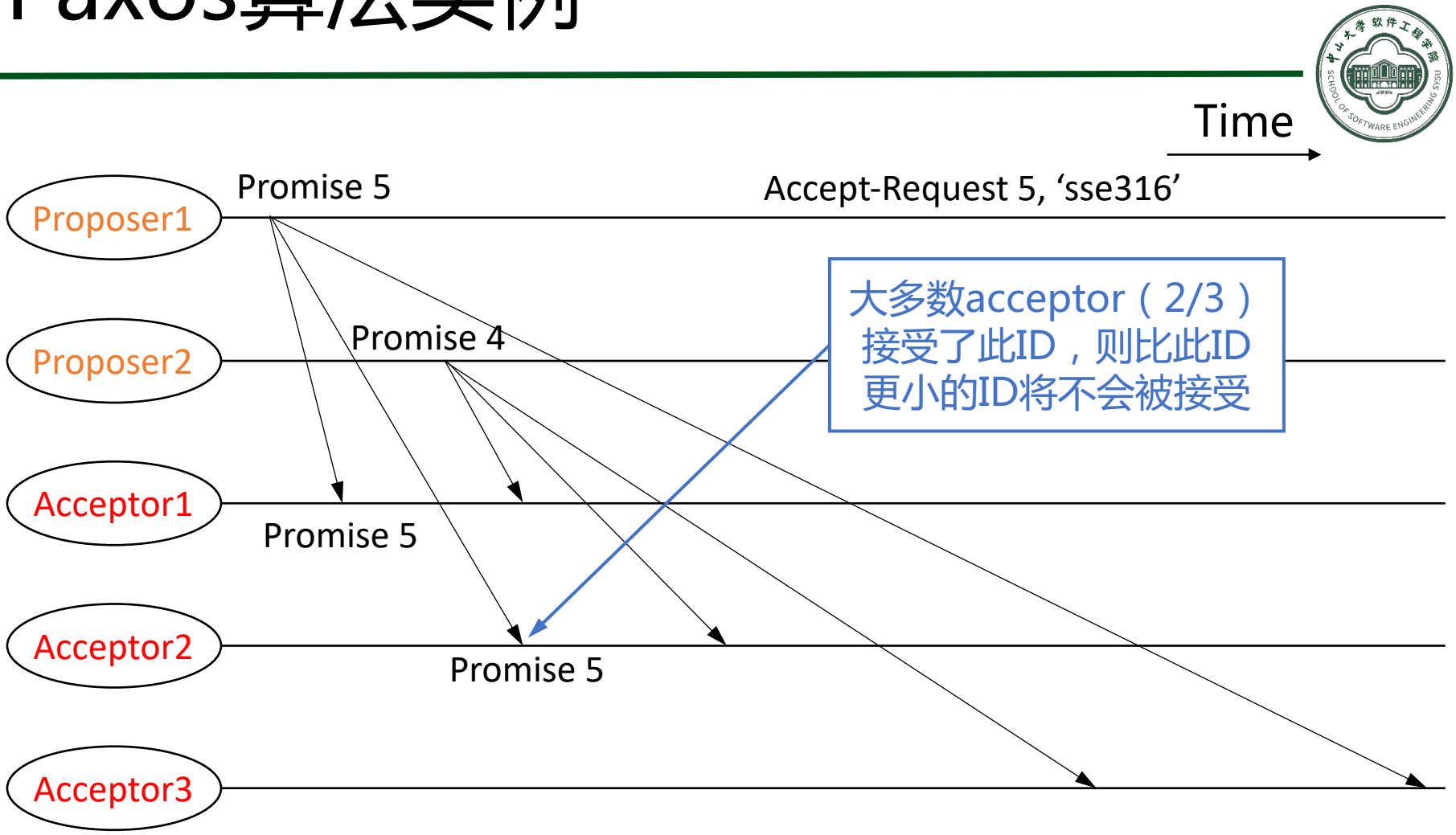
Paxos算法实例


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

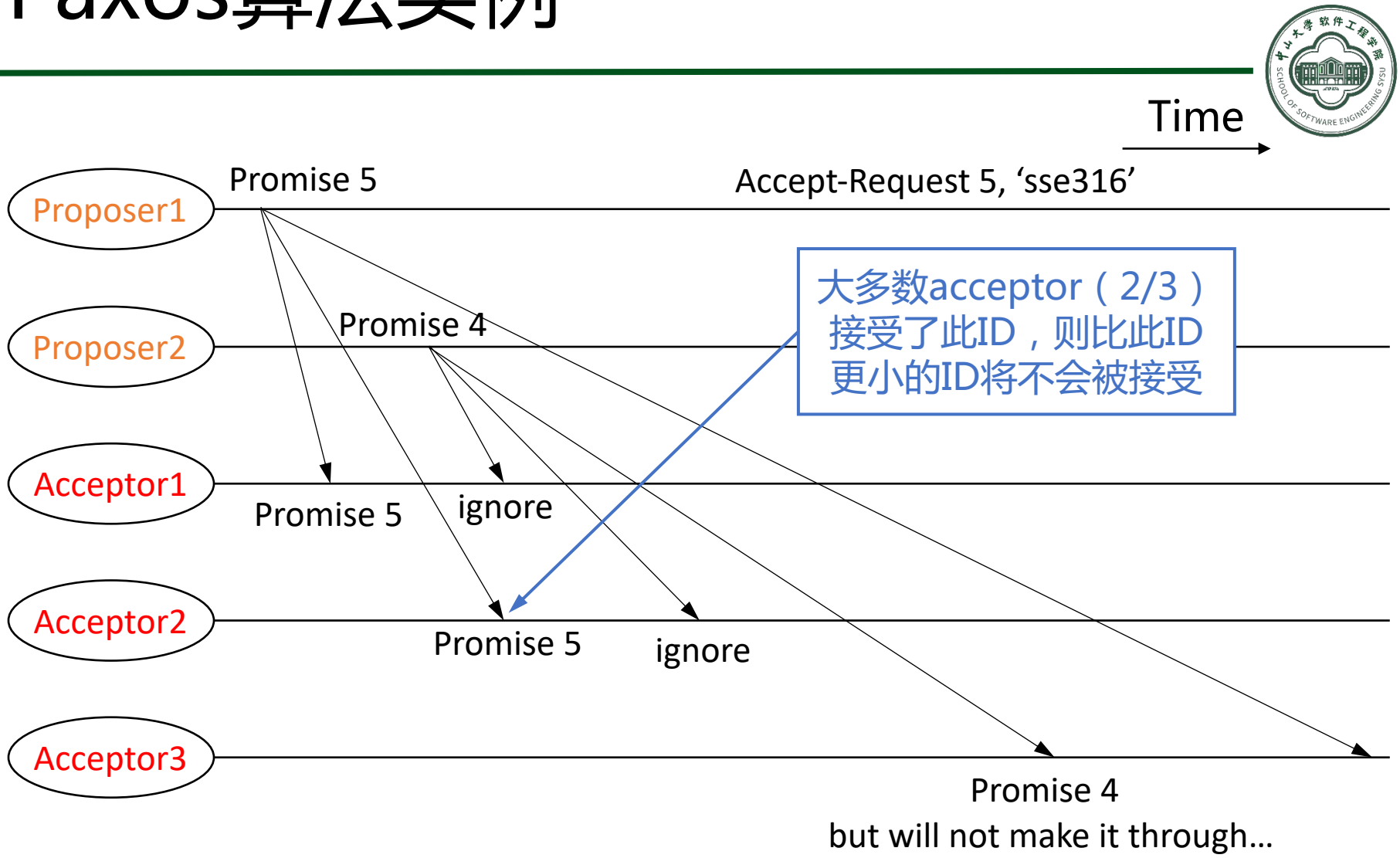
Paxos算法实例


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

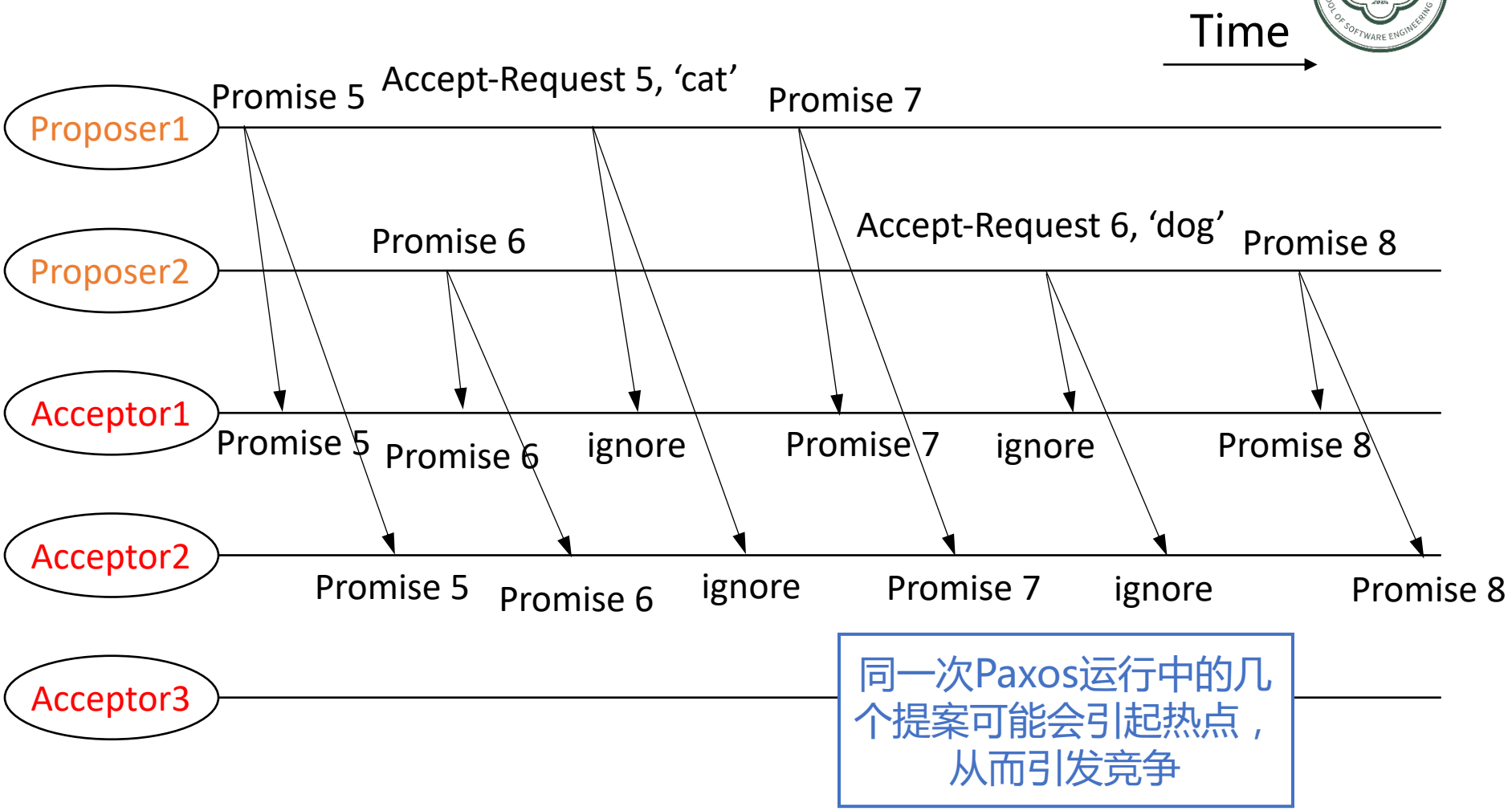
Paxos算法实例


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

云数据存储

- 文件共享语义
- 数据一致性
- 数据容错

Google集群首年的故障情况

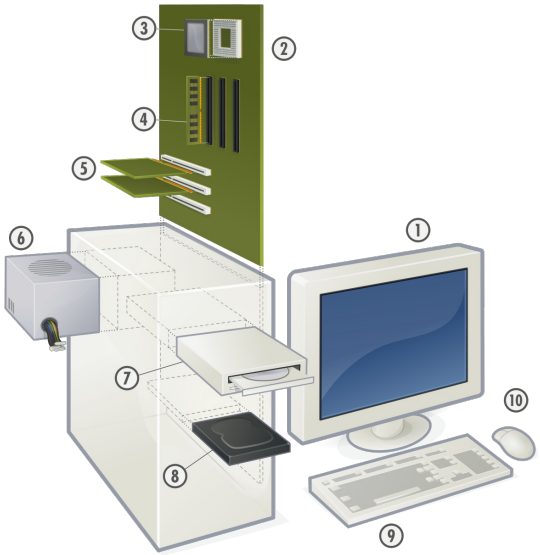
Typical first year for a new cluster:

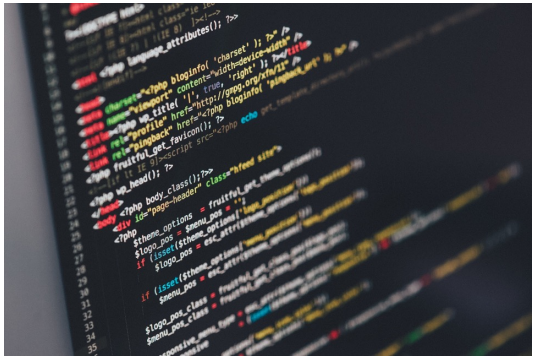
~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 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？

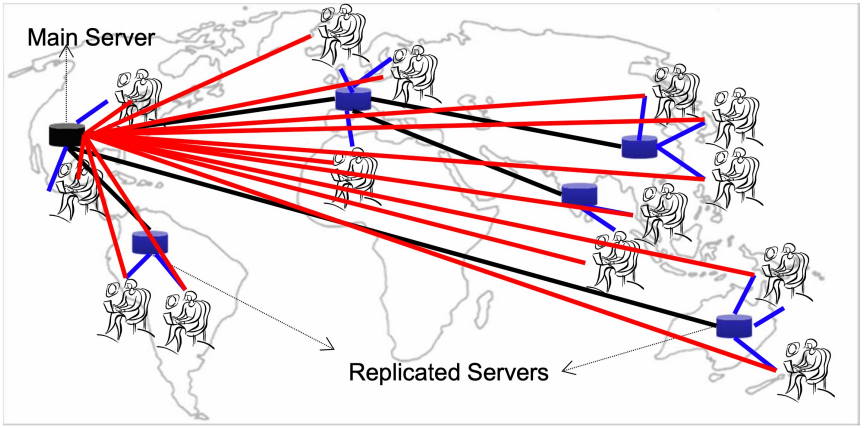
数据备份是在多台计算机上维护多份相同数据的过程

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？

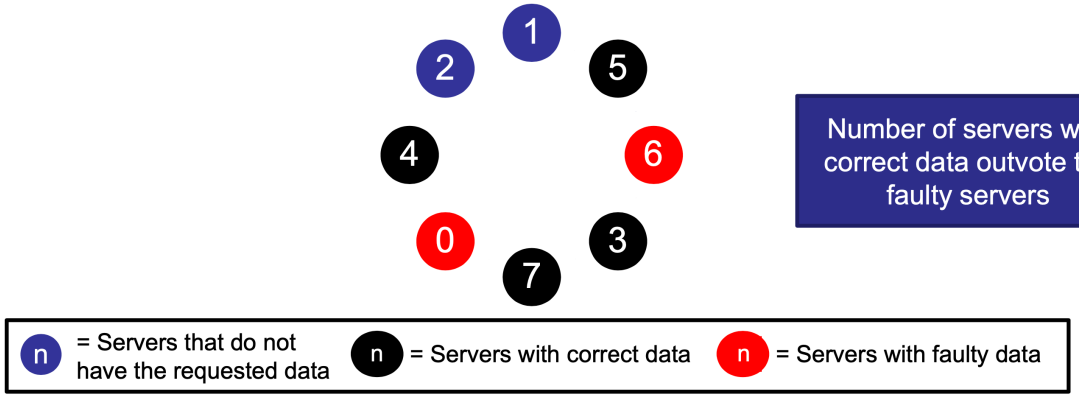
增加服务的可用性

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

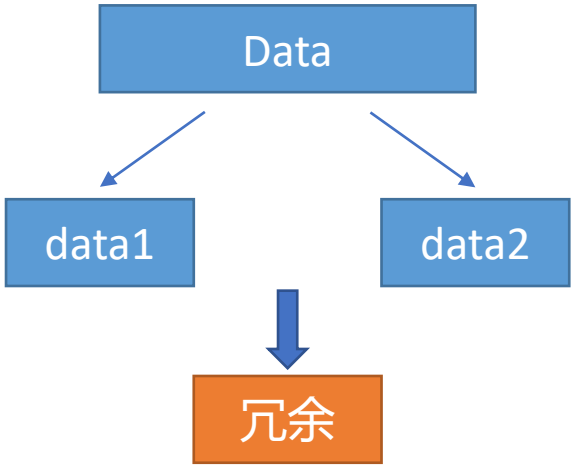
即使某些副本是恶意的，也可以依赖未受损服务器上的副本来保证数据安全

数据备份虽然简单，但需要消耗大量存储资源



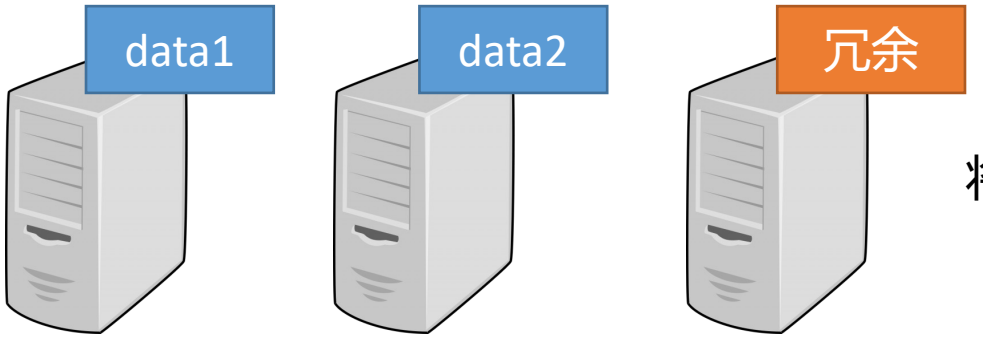
如果持有数据的服务器中有少数是恶意的，那么非恶意服务器可以投票超过恶意服务器，从而提供安全性。

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



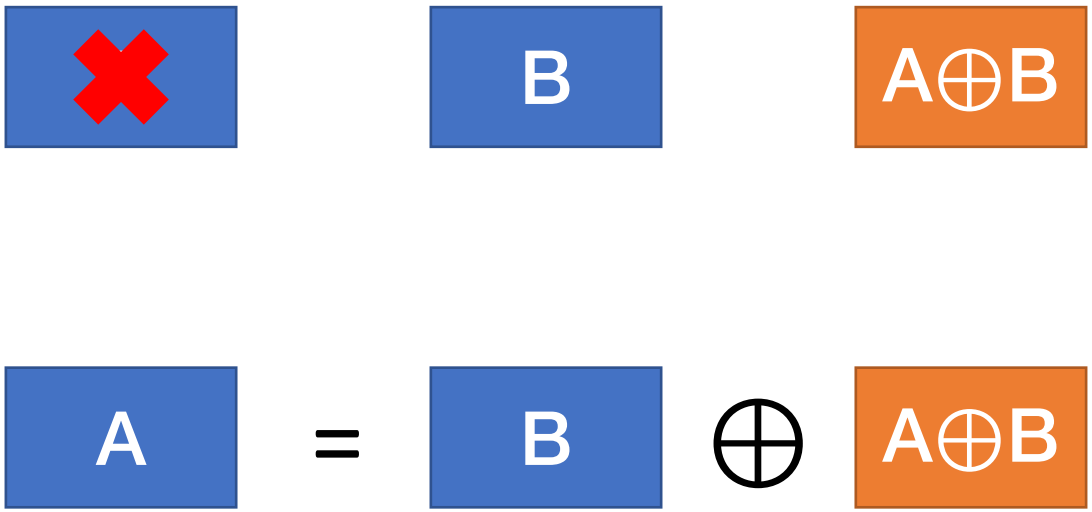
将片段分布在多个存储节点或设备上

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。

一个简单的就删码例子（1）

一个简单的就删码例子（2）

一个简单的就删码例子（3）

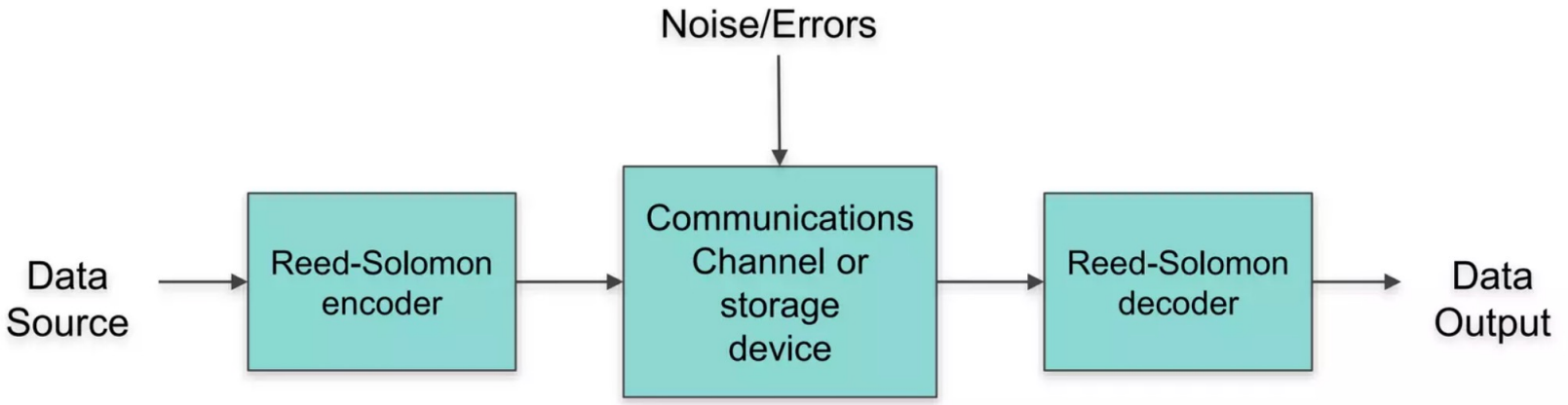


里德所罗门码

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



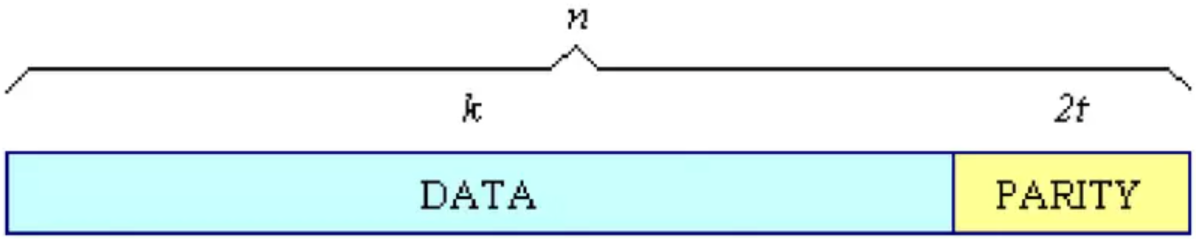
- ✓ RSC编码器（Reed-Solomon encoder）接收一块数据并添加额外的冗余位
- ✓ 数据在传输或存储过程中可能出现错误（如噪声或干扰，CD上的划痕）
- ✓ RSC解码器（Reed-Solomon decoder）处理每个块，试图纠正错误，回复原始数据

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，其中$2t = n - k$
  - ✓ 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。

$$n = 255, k = 223, s = 8$$

$$2t = 32, t = 16$$

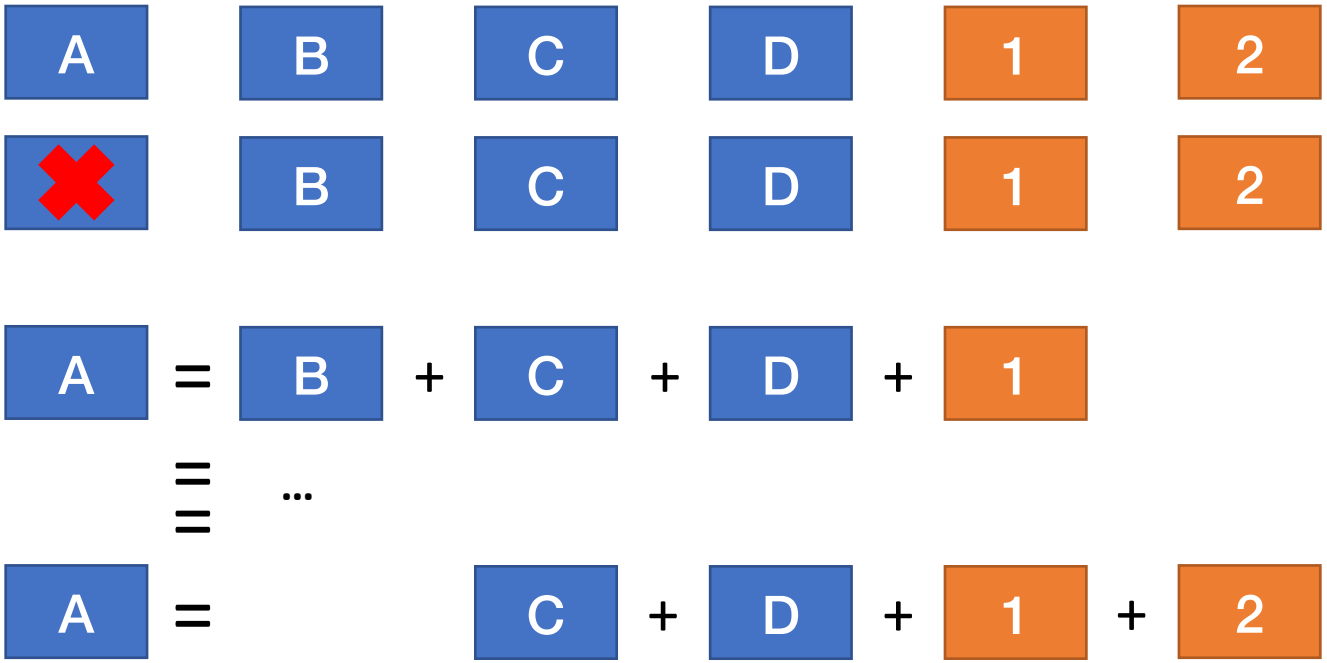
解码器可以更正16个symbol errors

- ✓ 最差的情况：16个bit errors，分别发生在不同的symbol，可纠正个16×1个bit errors
- ✓ 最好的情况：16个symbol内的bit全错了，可纠正16×8个bit errors

RSC解码（1）

当故障发生时，只要有至少任意k个正确的data 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



RSC解码（2）

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
- ✓ 若$2s + r < 2t$，即s个error，t个erasure，则总是可以恢复原始数据
- ✓ 解码器发现其无法恢复原始数据，并进行说明
- ✓ 解码器在无任何说明对情况下进行了错误解码，即恢复了错误数据

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。

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
  - ✓简单的拷贝数据

- 纠删码
  - ✓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

- 纠删码
  - ✓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

- 纠删码
  - ✓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

- 纠删码
  - ✓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

https://zbchern.github.io/sse316.html