



第五章

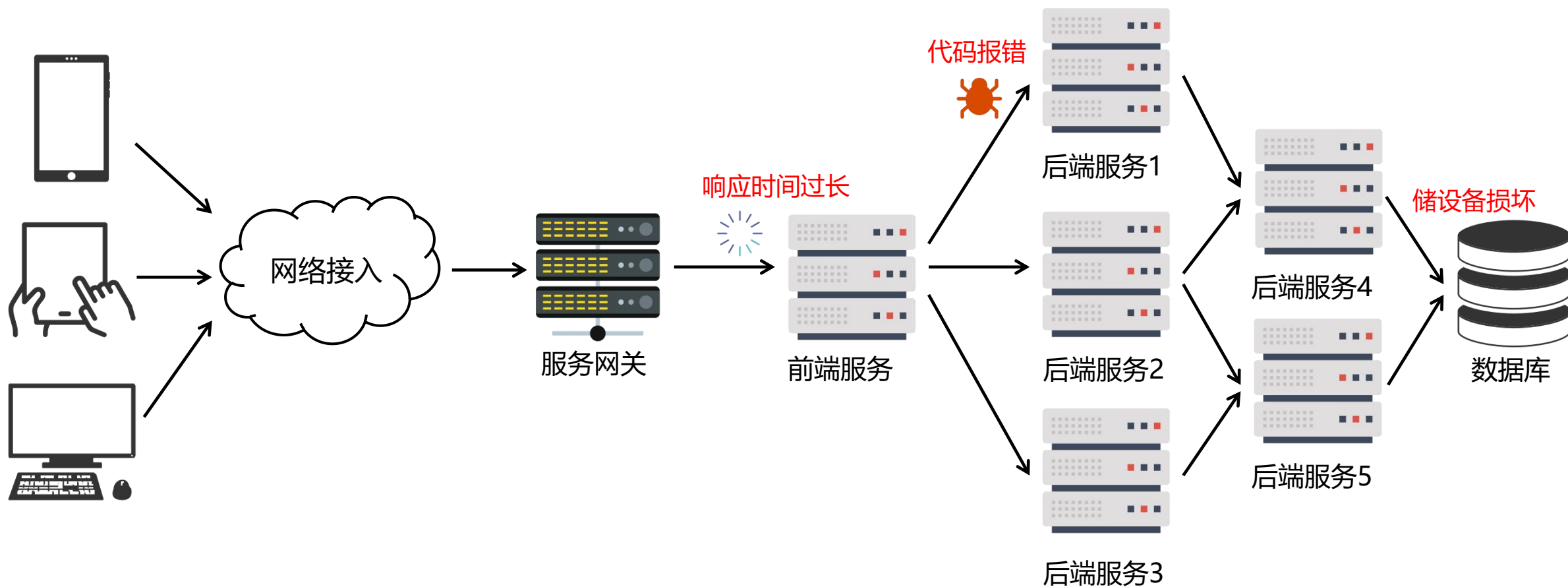
容错与共识

什么是容错？为什么分布式系统需要容错？

分布式环境中，存在许多可能导致节点故障或通信失败的因素，也许是一台服务器的电源故障或磁盘损坏，亦或是由于网络连接中断导致某一节点被隔离在与其他节点不同的网络分区中，因此分布式系统是很不稳定的。

因此，分布式系统设计里的一个重要目标，就是使系统在故障发生后可以自动的修复故障，同时又可以继续提供服务。这种能容忍错误的能力被称为容错性。

什么是容错？为什么分布式系统需要容错？



第五章、容错与共识

01

5.1 故障类型

02

5.2 故障检测与恢复

03

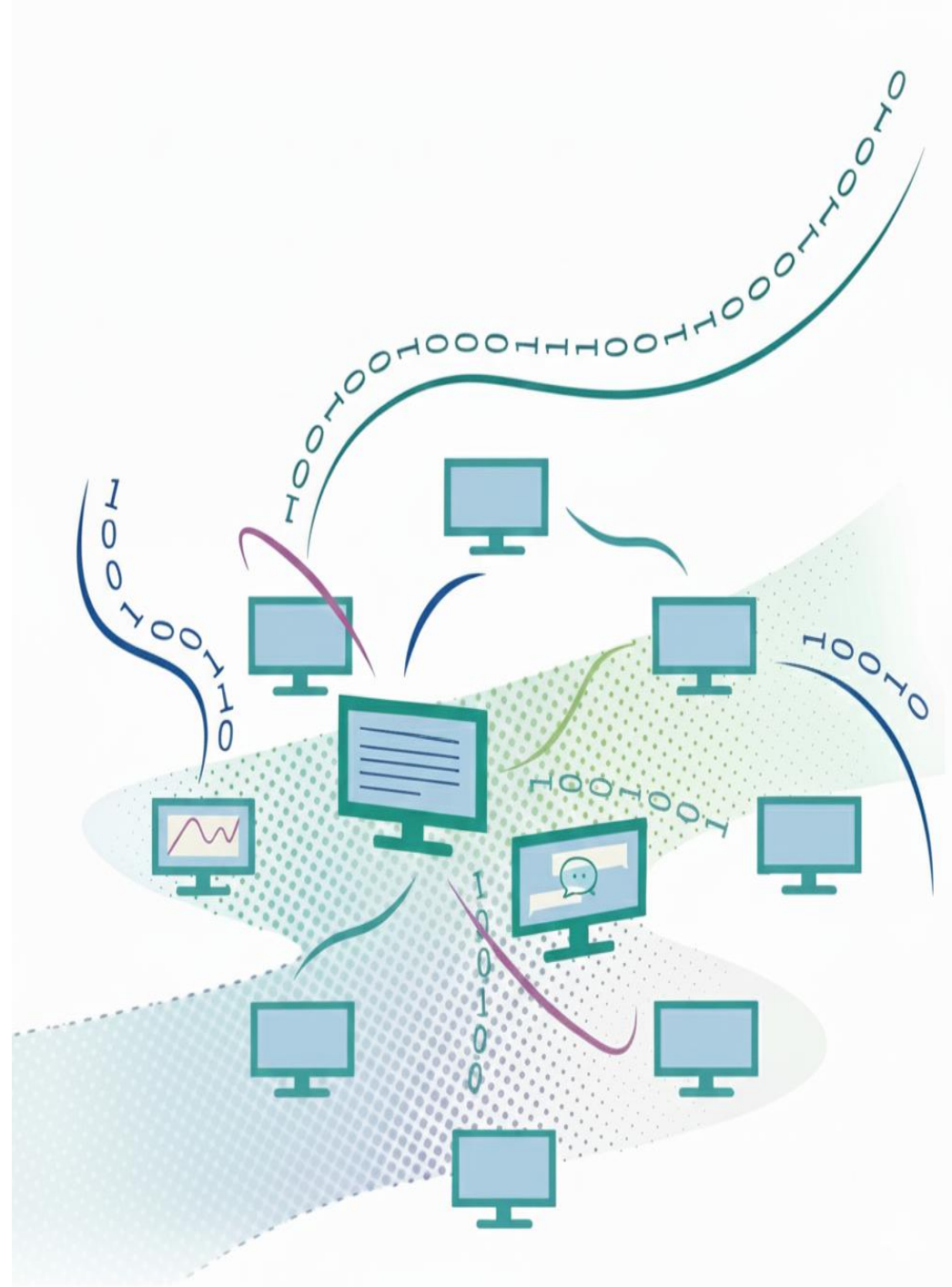
5.3 Paxos算法

04

5.4 Raft算法

05

5.5 PBFT算法*



5.1 故障类型

01

5.1.1 基于影响范围划分

02

5.1.2 基于故障性质划分



5.1.1 基于影响范围划分

—— 单机上的故障（节点故障）

- 软件故障



1. 我的游戏输入空白名字就会报错闪退！



2. 重新启动之后有什么问题吗？



3. 这是游戏的bug，闪退重启会导致你存档之后的进度丢失...

5.1.1 基于影响范围划分

• 软件故障

1、定义

软件故障指由于软件程序的错误、内存泄漏、死锁等原因导致的系统失效。

2、示例

常见的软件故障包括缓冲区溢出、内存泄漏等故障。

3、特点

软件故障仅使正在运行的任务受到影响，并且会丢失内存中的全部数据，但硬件存储本身没有被破坏。

5.1.1 基于影响范围划分

—— 单机上的故障（节点故障）

- 硬件故障



1. 我的固态硬盘坏掉了!



2. 看看有什么影响吗?



3. 存档可能丢失, 严重点有可能会因为损坏无法启动...

5.1.1 基于影响范围划分

• 硬件故障

1、定义

软硬件故障指分布式系统中硬件设备（如服务器、存储设备、网络设备等）的故障。

2、示例

常见的硬件故障包括存储介质故障（如磁盘、SSD）等硬件设备发生的故障。

3、特点

与软件故障相比，硬件故障对节点的影响更严重，不仅使运行中的事务受到影响，还可能会破坏持久化存储导致持久化数据丢失。

5.1.1 基于影响范围划分

——网络中的故障

- 网络故障



1. 我的游戏
怎么卡住了!

This illustration shows three cartoon characters: a woman with black hair on the left, a man with brown hair in the center, and a woman with pink hair on the right. A large, cloud-shaped thought bubble originates from the man, containing the text '1. 我的游戏怎么卡住了!'.



2. 可能由于在
线高峰期网络
拥塞等原因导
致延时、丢包...

This illustration shows a cartoon man with black hair wearing a white jacket over a red shirt. A cloud-shaped thought bubble originates from him, containing the text '2. 可能由于在线高峰期网络拥塞等原因导致延时、丢包...'.

5.1.1 基于影响范围划分

• 网络故障

1、定义

网络故障指分布式系统中节点之间由于网络分区、网络延迟、连接中断等导致的通信层面的故障。

2、示例

例如，推特平台因网络延迟等原因导致某个数据中心无法读取其他数据中心更新的动态，因为请求可能无法到达其他数据中心。

3、特点

由于分布式系统的节点都是通过网络连接的，与前两者相比网络故障属于集群整体层面的故障。（网络故障与节点故障也存在交叉，因为可能由节点故障导致节点间无法通信）

5.1 故障类型

01

5.1.1 基于影响范围划分

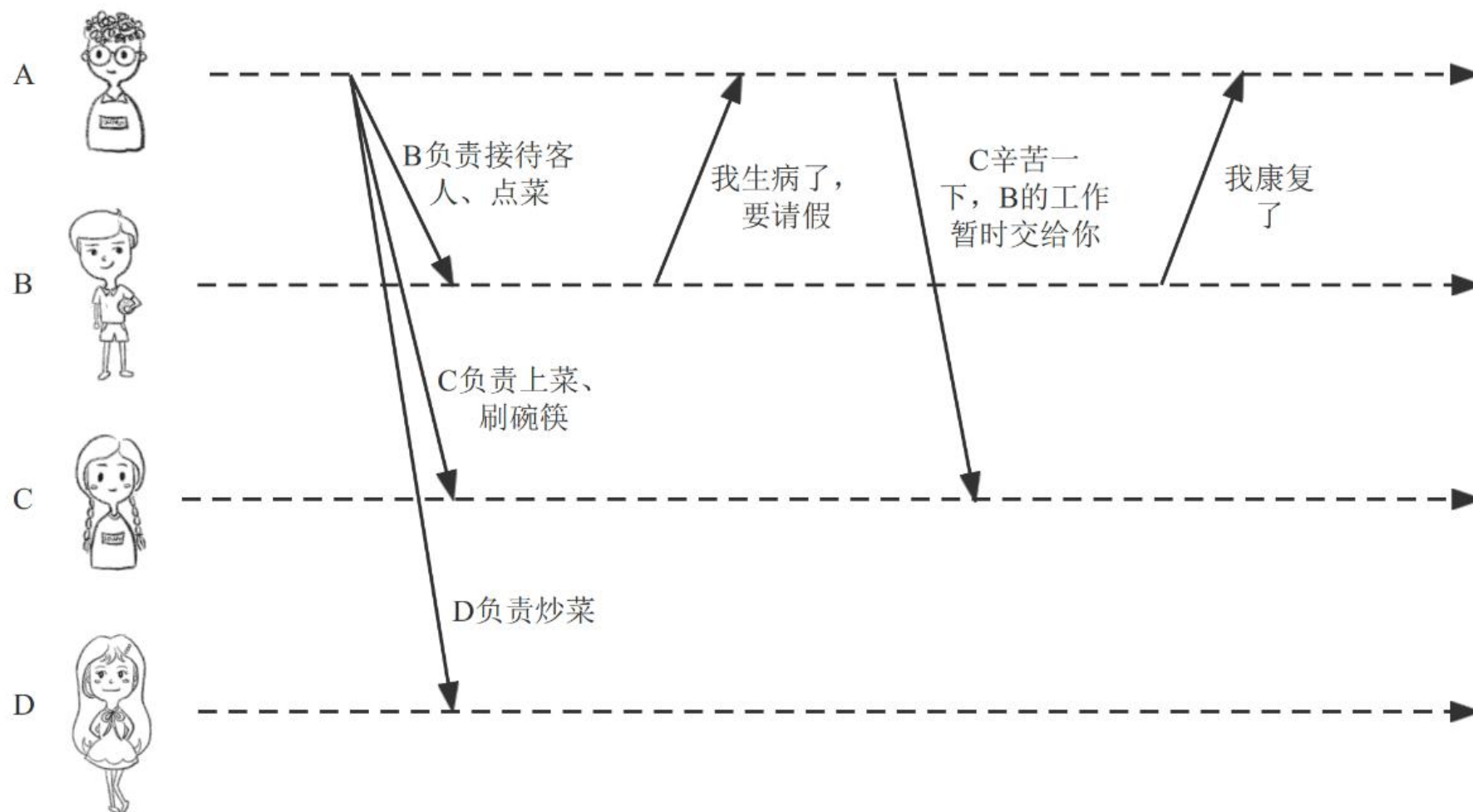
02

5.1.2 基于故障性质划分



5.1.2 基于故障性质划分

• 崩溃故障



5.1.2 基于故障性质划分

• 崩溃故障

1、定义

崩溃故障是指在分布式系统中的节点或进程由于硬件故障、软件错误或其他原因而突然终止运行的情况。

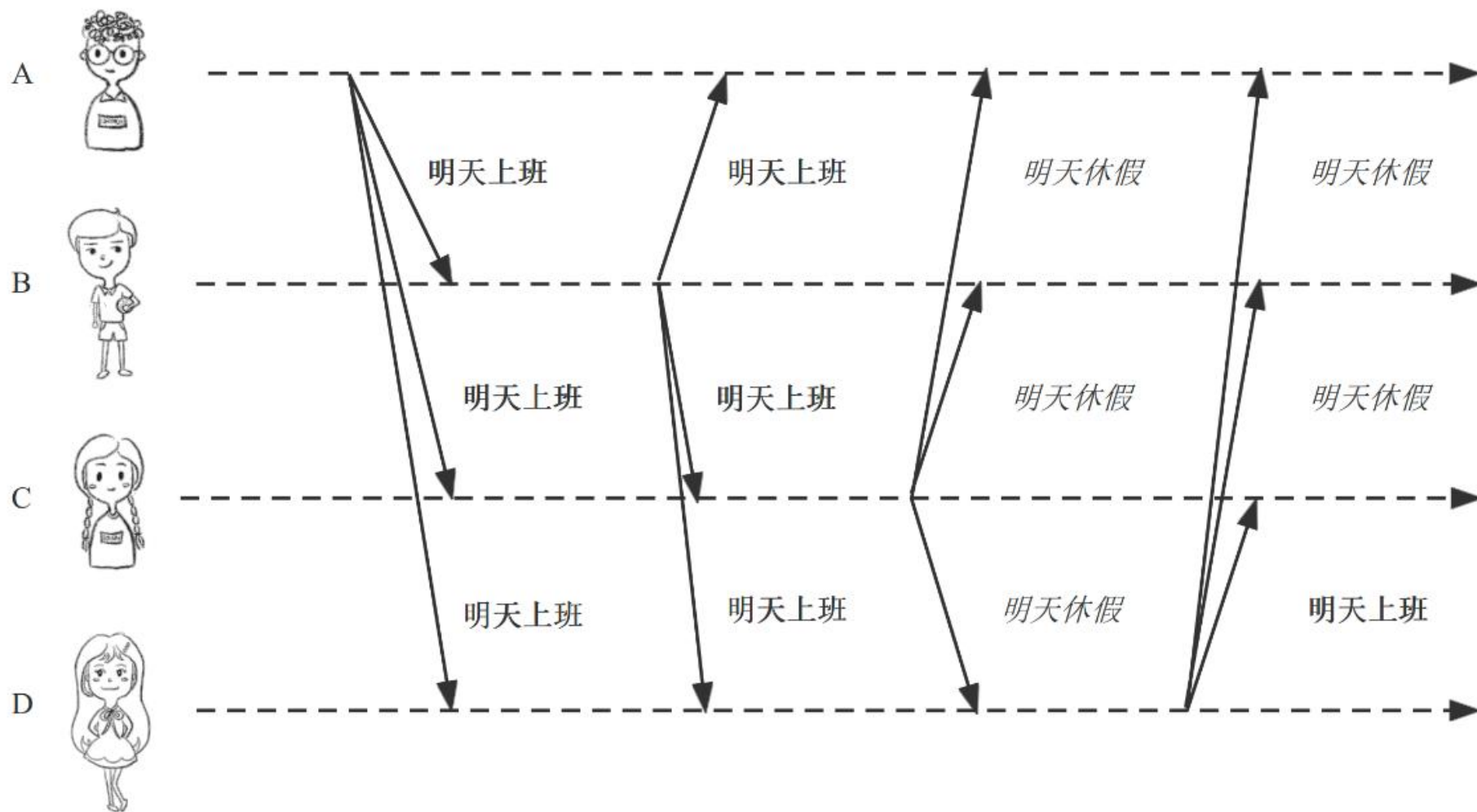
2、特点

崩溃故障的主要特点在于其**局部性**，系统的部分失效仅局限于发生故障的节点。对于此类故障，可以：

- 通过冗余备份提供数据的完整性保护防止数据丢失
- 通过故障转移将任务转交给其他节点来恢复系统的正常服务

5.1.2 基于故障性质划分

• 拜占庭故障



5.1.2 基于故障性质划分

- 拜占庭故障

1、定义

拜占庭故障是是一种更为复杂的故障类型。在拜占庭故障中，节点可能会出现任意的、不可预测的行为，包括发送错误的消息、篡改或伪造消息等。

2、特点

拜占庭故障的主要特点在于传播性，节点发出的错误信息会传播到其他节点，影响整个系统的安全性和正确性。对于此类故障，需要拜占庭容错算法，如PBFT算法。

第五章、容错与共识

01

5.1 故障类型

02

5.2 故障检测与恢复

03

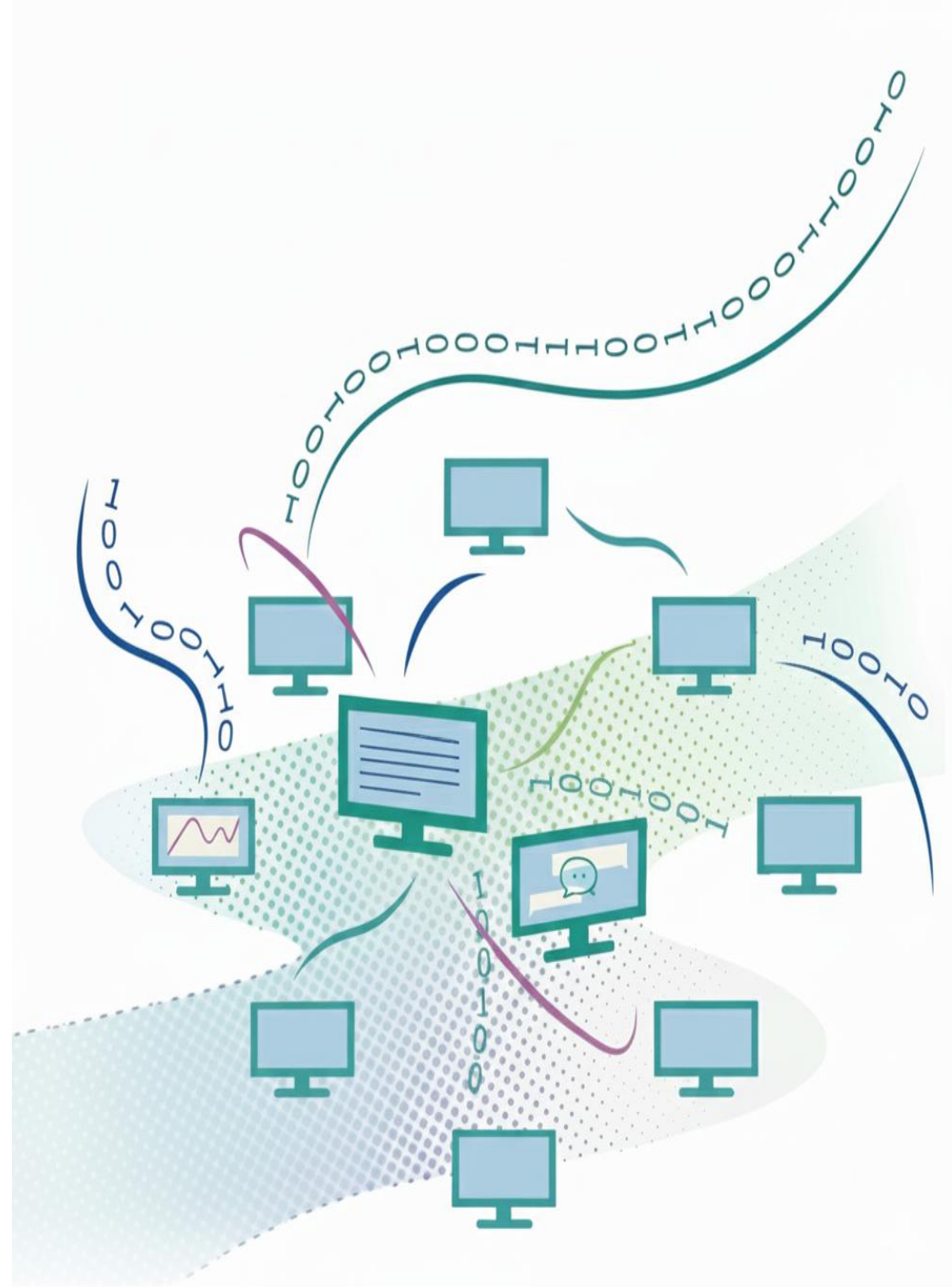
5.3 Paxos算法

04

5.4 Raft算法

05

5.5 PBFT算法*



5.2 故障检测与恢复

01

5.2.1 故障检测

02

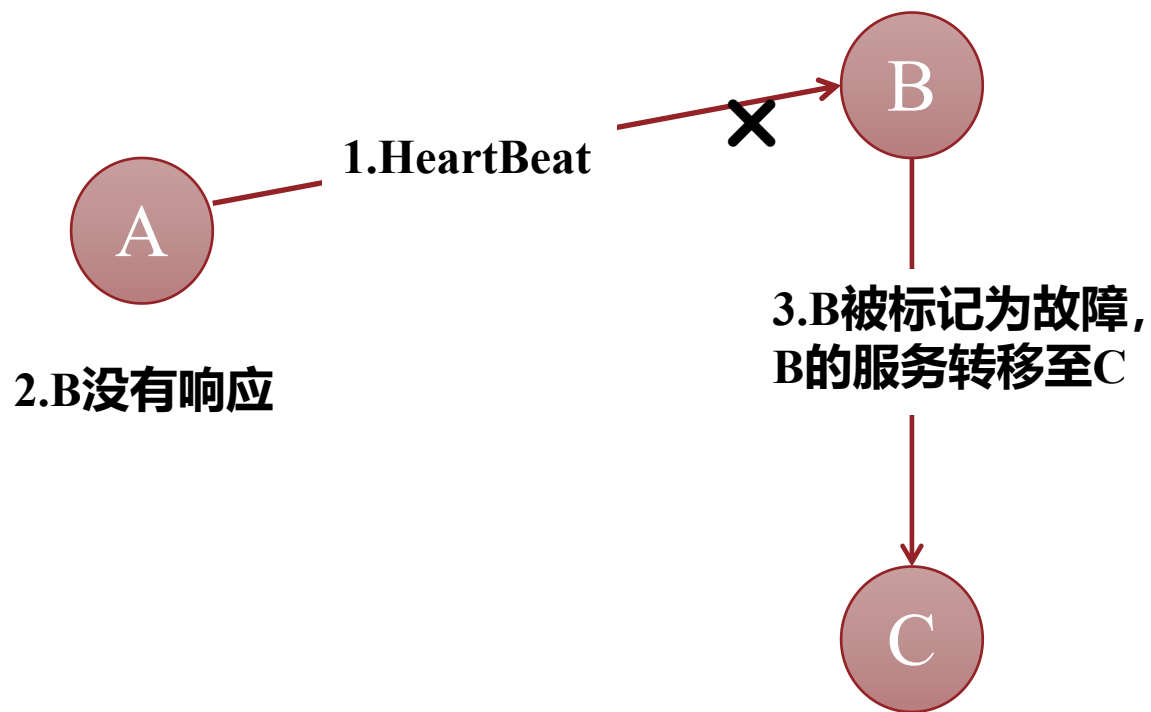
5.2.2 故障恢复



5.2.1 故障检测

• 基于心跳 (Heartbeat) 的故障检测

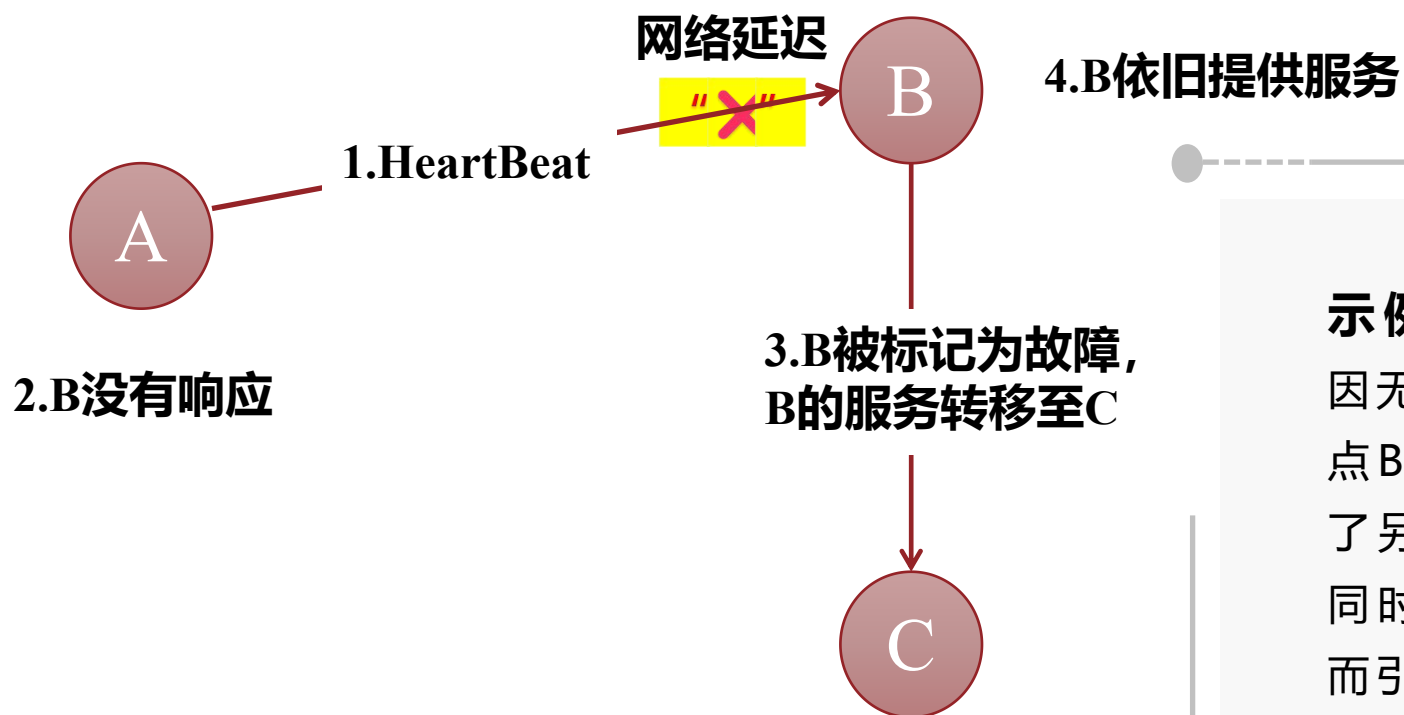
如何检测故障?



问题： 被监控节点B没有响应不一定说明节点B崩溃了，节点B可能处理了请求但是在网络回传过程中反馈信息丢包了，也可能由于节点B的负载压力较大，消息依旧在队列中等待；相应地，也存在节点B无法进行有效计算但仍反馈心跳消息的可能，会将故障节点标记为正常节点。

5.2.1 故障检测

- GFS (Google File System) 租约机制

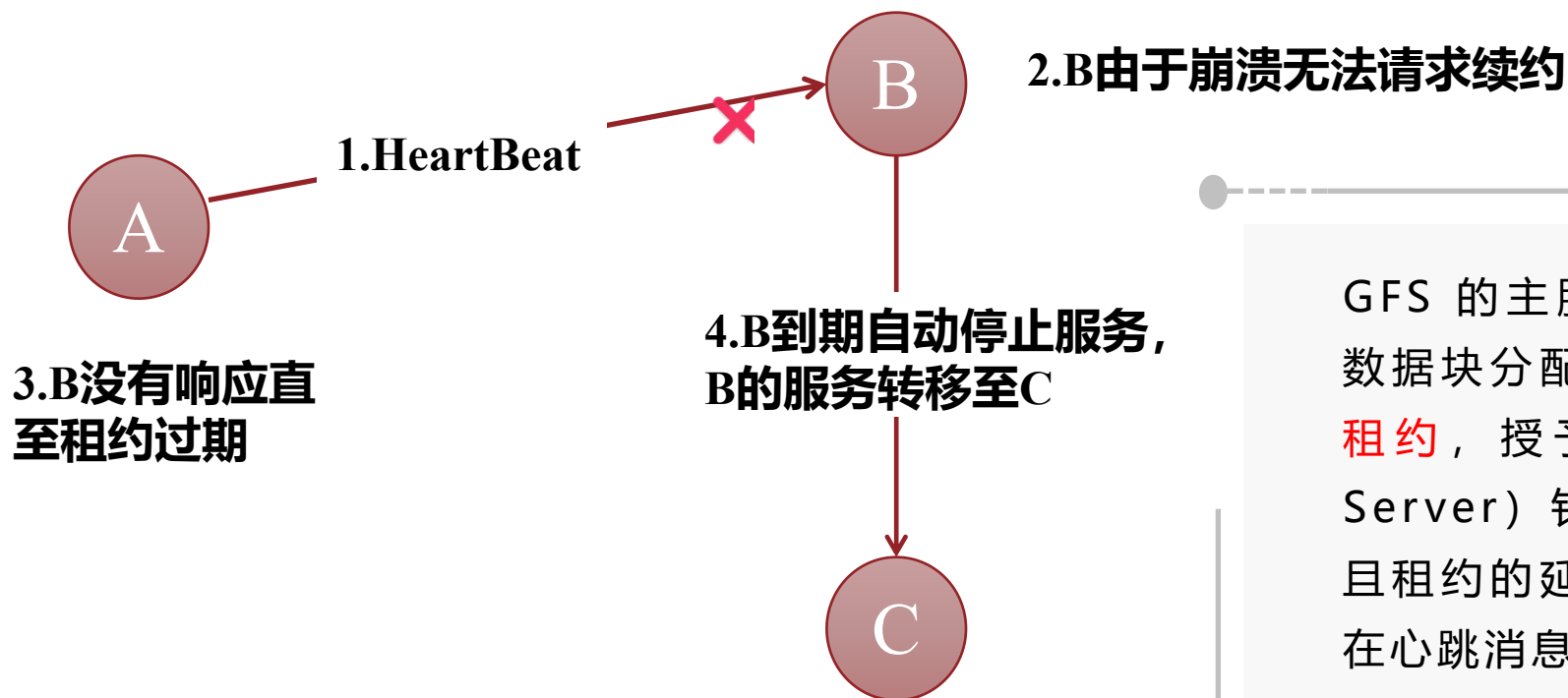


示例： B并不是因为节点崩溃故障的原因无法响应，而监控节点A却单方面将节点B标记为故障，将节点B的服务转移到了另一个节点C，那么会出现节点B和C同时对同一份数据提供服务的情况，从而引发“双主”问题。

5.2.1 故障检测

• GFS (Google File System) 租约机制

如何避免“双主”？



GFS 的主服务器 (Master) 为每个数据块分配一个具有固定时间期限的租约, 授予一个从服务器 (Chunk Server) 针对该数据块的写入权限, 且租约的延长请求与批准信息都附加在心跳消息中

5.2 故障检测与恢复

01

5.2.1 故障检测

02

5.2.2 故障恢复

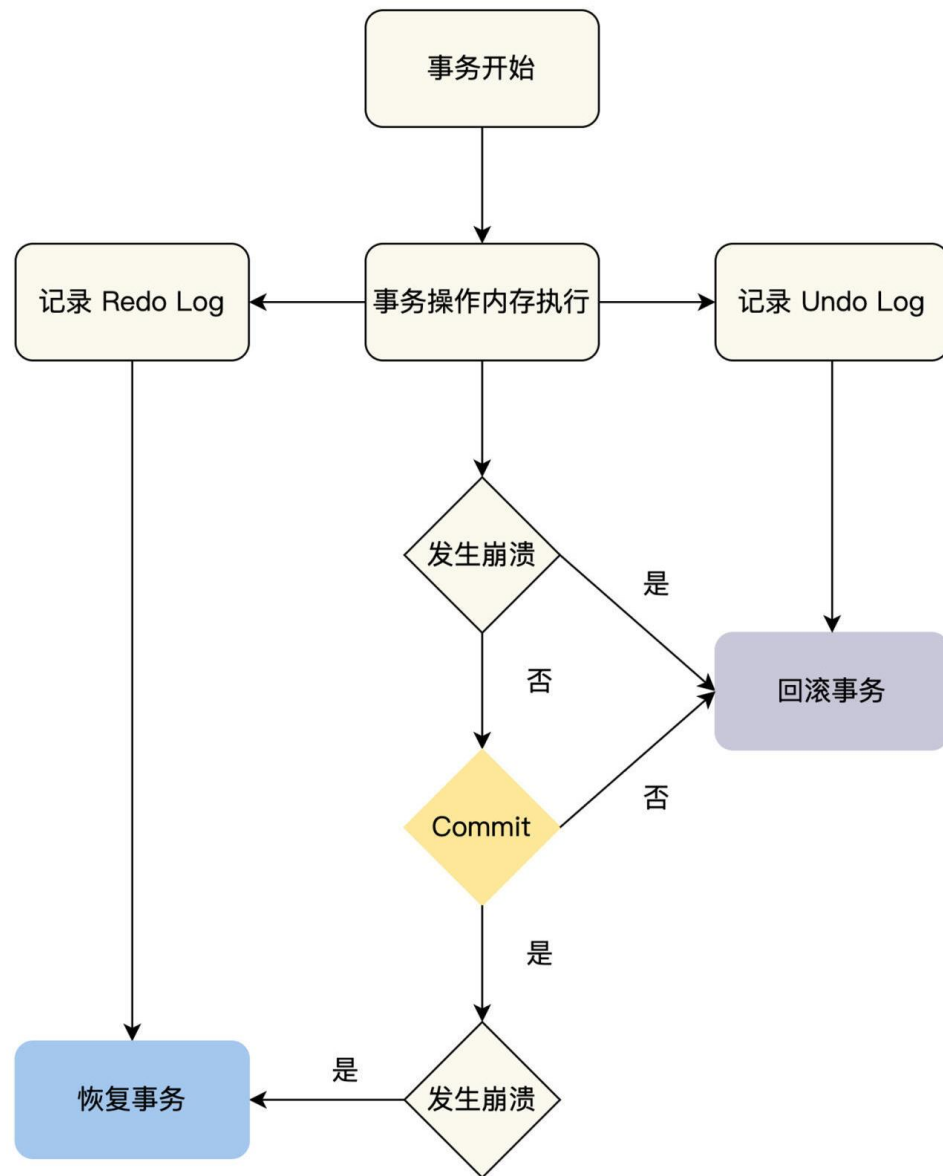


5.2.2 故障恢复

• 本地日志恢复

示例：数据库崩溃后通过 Undo/Redo 恢复数据库某一时刻的状态：

1. 尚未写入 commit 的事务，通过 Undo Log 回滚已执行的操作；
2. 已经写入 Commit 的事务，通过 Redo Log 重做事务；



5.2.2 故障恢复

• 本地日志恢复

1、定义

系统需要在反馈用户请求操作之前将操作记录在**持久化的日志**中，以便在故障发生时通过重放日志来恢复到故障发生前的一致状态。

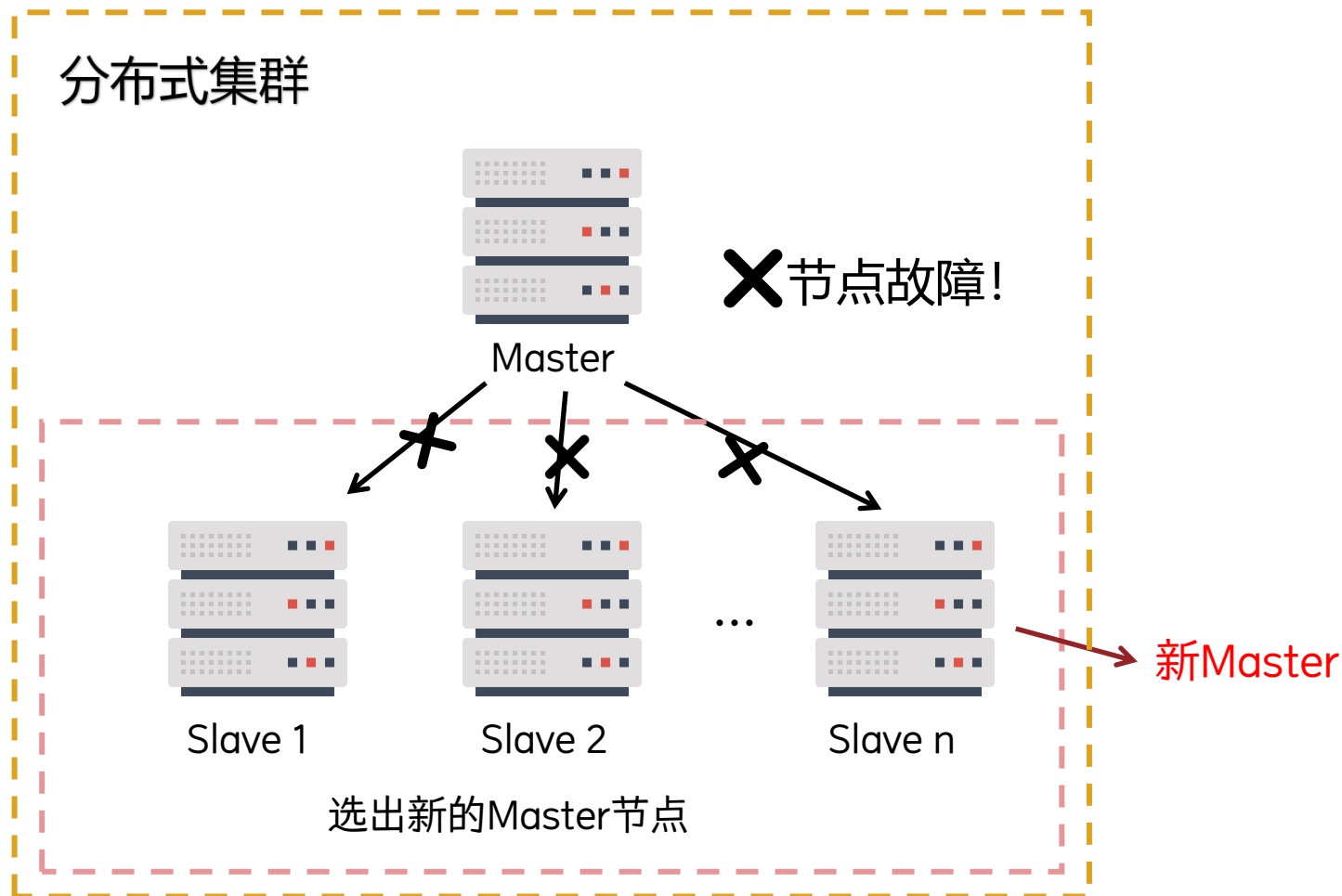
2、特点

日志恢复技术可靠且通用，广泛应用于数据库系统和分布式事务处理系统中，主要用于处理节点自身发生的**软件故障**。但是如果是硬件故障导致持久化存储被破坏，那么本地日志恢复策略将无法进行有效恢复。

5.2.2 故障恢复

• 备机恢复

示例：假设有一个主机 Master 和 n 个从机 Slave1, Slave2, ..., Slaven; 所有数据写入操作都在 Master 上执行, 然后同步到 Slave 上。Master 因为硬件故障宕机, 整个系统不能再处理新的写入请求, 于是选出新的 Master 继续接收新的请求;



5.2.2 故障恢复

• 备机恢复

1、定义

配置一个或多个**备用节点**，在主要节点发生故障时，备用节点会接管服务，确保系统的持续运行。

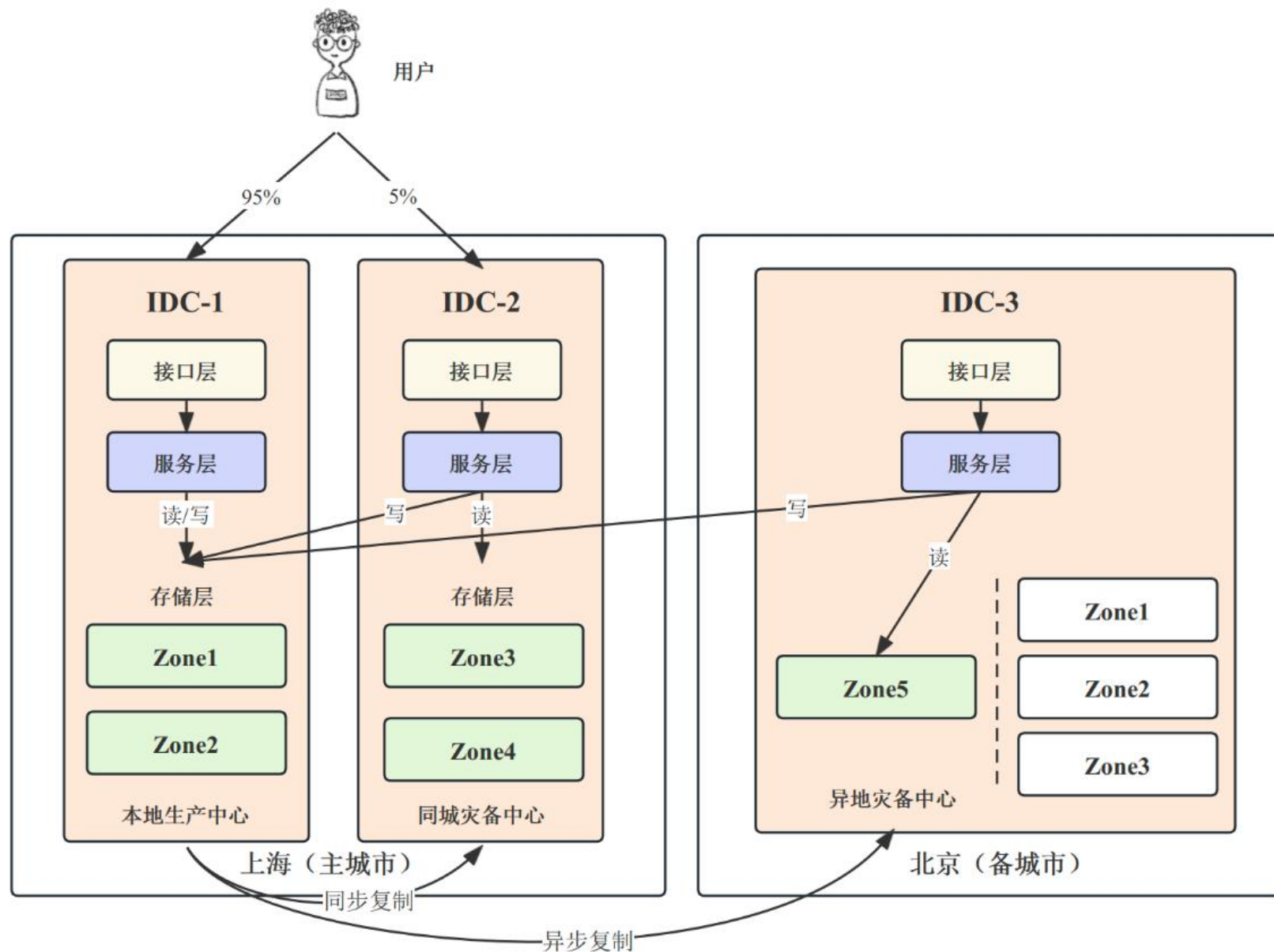
2、特点

备机通常与原始系统具有相同的硬件和软件配置，并且处于就绪状态以应对故障发生时的切换。这种方式可以快速恢复系统，并且在**同一地点场地**内进行，减少了数据传输和网络延迟的开销，但无法提供地域级别的容灾能力。

5.2.2 故障恢复

• 跨域恢复

示例：分布式数据库OceanBase的两地三中心部署架构：该架构由主城市节点和备城市节点共同组成一个五副本的集群。



5.2.2 故障恢复

• 跨域恢复

1.Region: 地区，一般一个数据库集群可能包含多个地区，一个地区包含多个可用区。

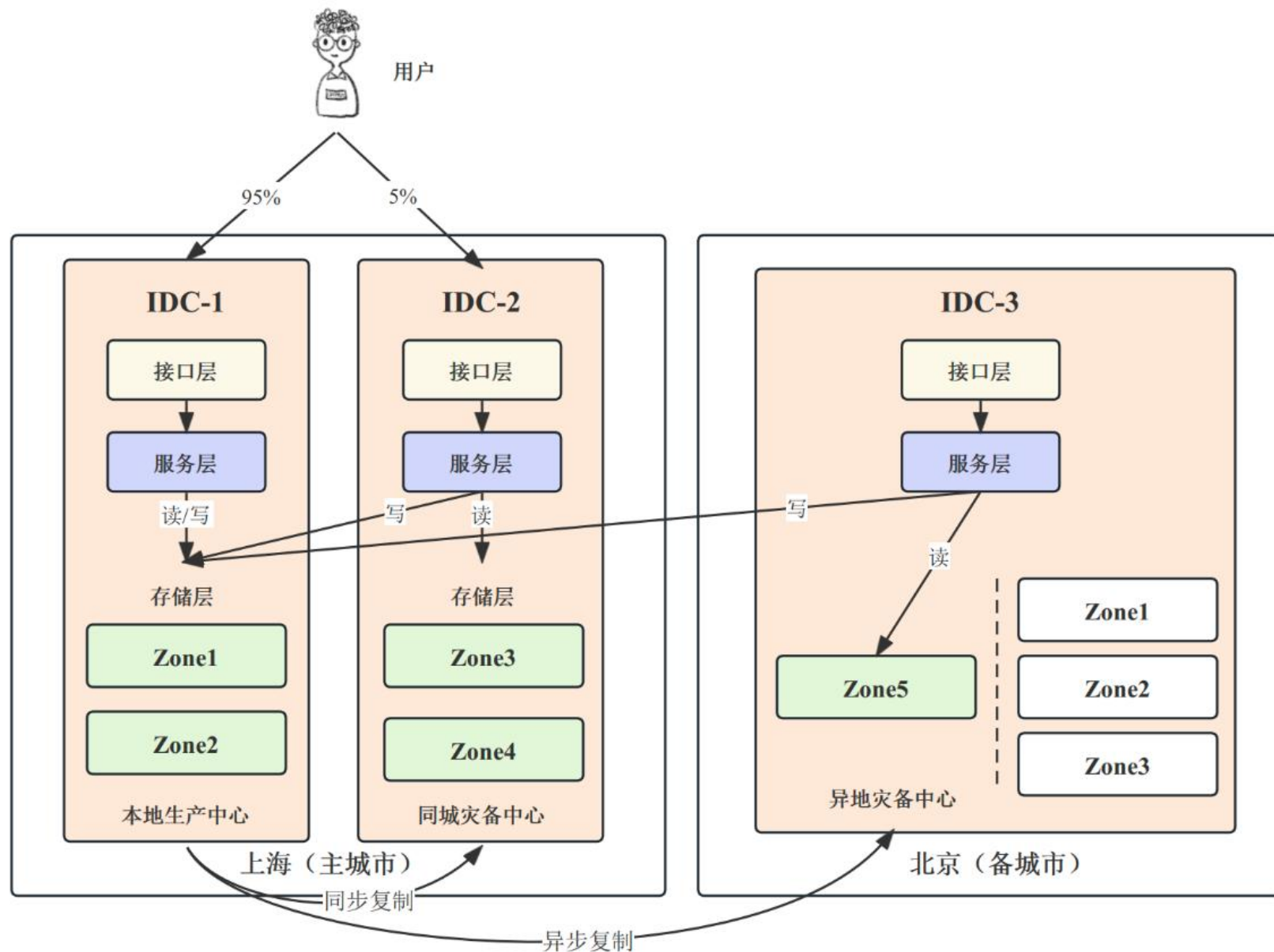
2.IDC: 互联网数据中心，一般对应一个地区中物理上的机房。

3.Zone: 可用区，指逻辑上的数据中心，一般对应一个有独立网络和供电容灾能力的数据中心。(Zone与IDC无绝对的对应关系，既可以一个机房具备两个相互独立的Zone，也可以一个Zone由多个机房维护)

5.2.2 故障恢复

• 跨域恢复

- **Region**: 地区
- **IDC**: 互联网数据中心, 一般对应一个地区中物理上的机房。
- **Zone**: 可用区, 指逻辑上的数据中心



5.2.2 故障恢复

• 跨域恢复

1、定义

跨域恢复在不同地理位置**数据中心**之间进行数据同步和恢复，由此实现高可用性和地域级的容灾能力。

2、特点

在跨域恢复中，数据中心通常被划分为主要数据中心和备用数据中心。主要数据中心是正常运行的主服务器的所在地点，而备用数据中心则是用于容灾和恢复的备用服务器所在的地点，可以提供地域级别的容灾能力。

第五章、容错与共识

01

5.1 故障类型

02

5.2 故障检测与恢复

03

5.3 Paxos算法

04

5.4 Raft算法

05

5.5 PBFT算法*



5.3 Paxos算法

01

5.3.1 驱动案例

02

5.3.2 角色与问题描述

03

5.3.3 算法流程

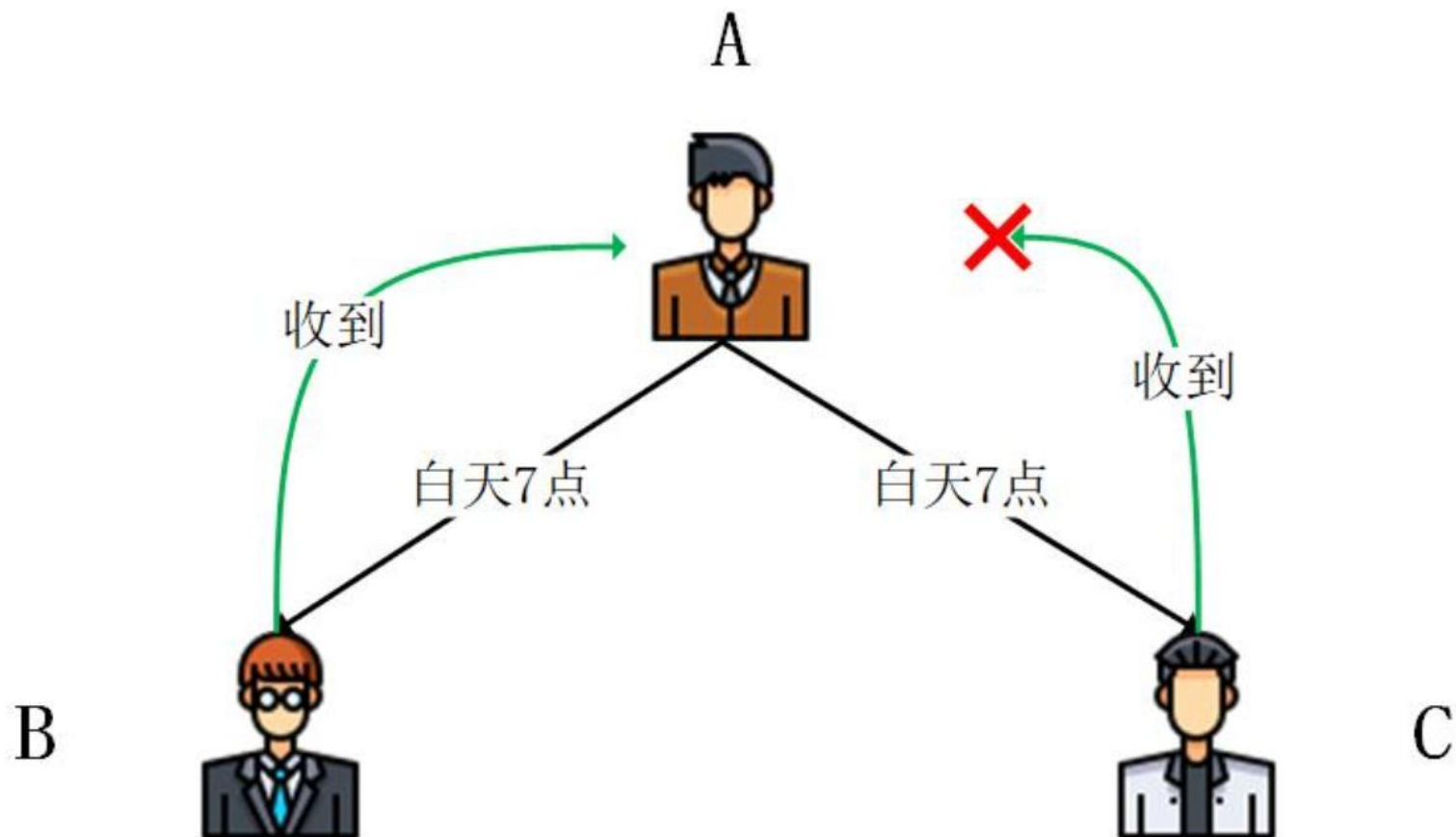


5.3.1 驱动案例



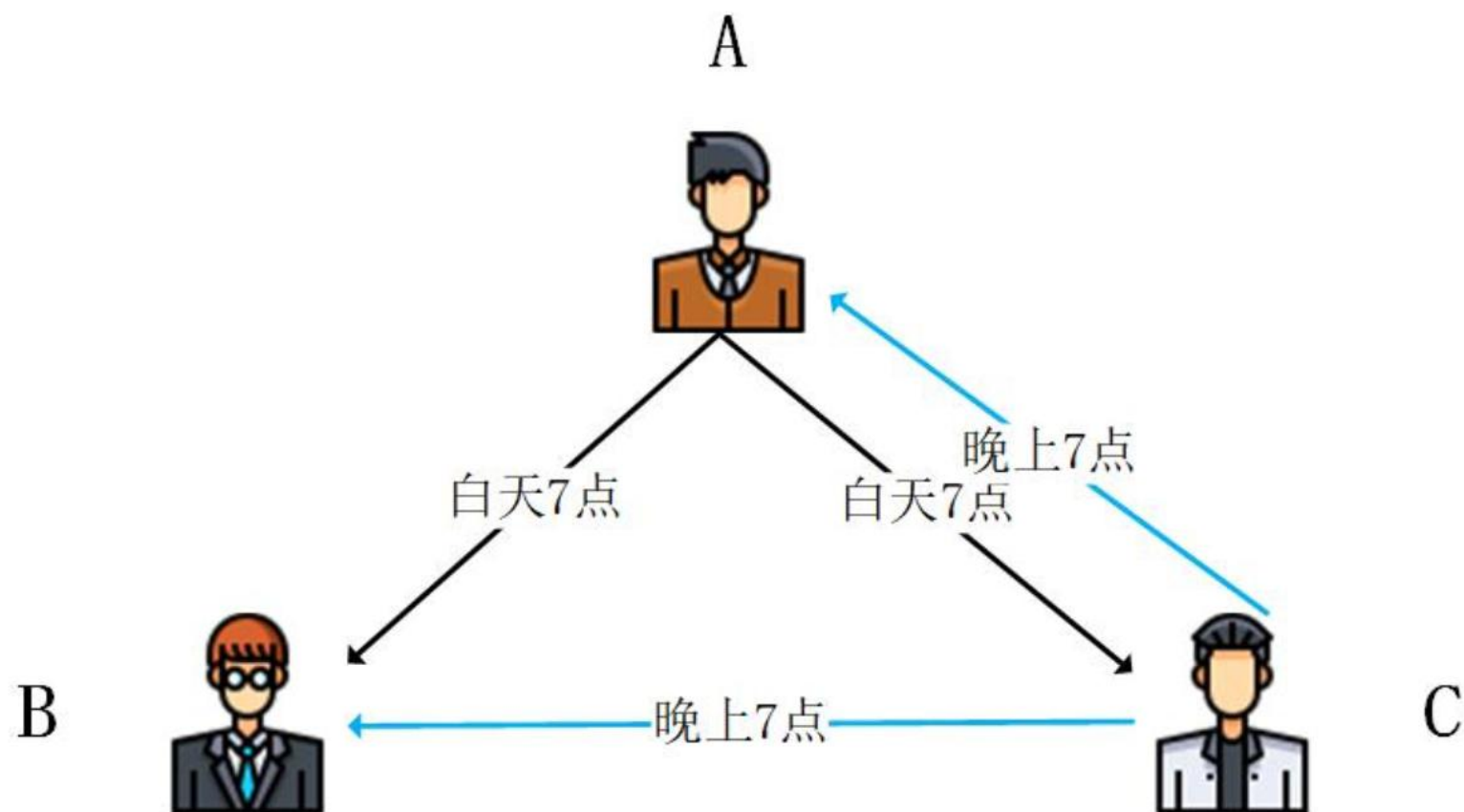
示例： A和他的伙伴B和C共3人好久未见，现在想要找个时间聚一聚，三人通过短信远程交流。

5.3.1 驱动案例



示例：假设A提出了白天7点的聚会提案并将提案发送给B和C，A只收到了B的回复而C的回复丢失了。

5.3.1 驱动案例



示例：另一种情况，没有收到提议消息的C以为目前还没有人提议，那么他也可能提出另外一个聚会提案，并且聚会时间是晚上7点

5.3 Paxos算法

01

5.3.1 驱动案例

02

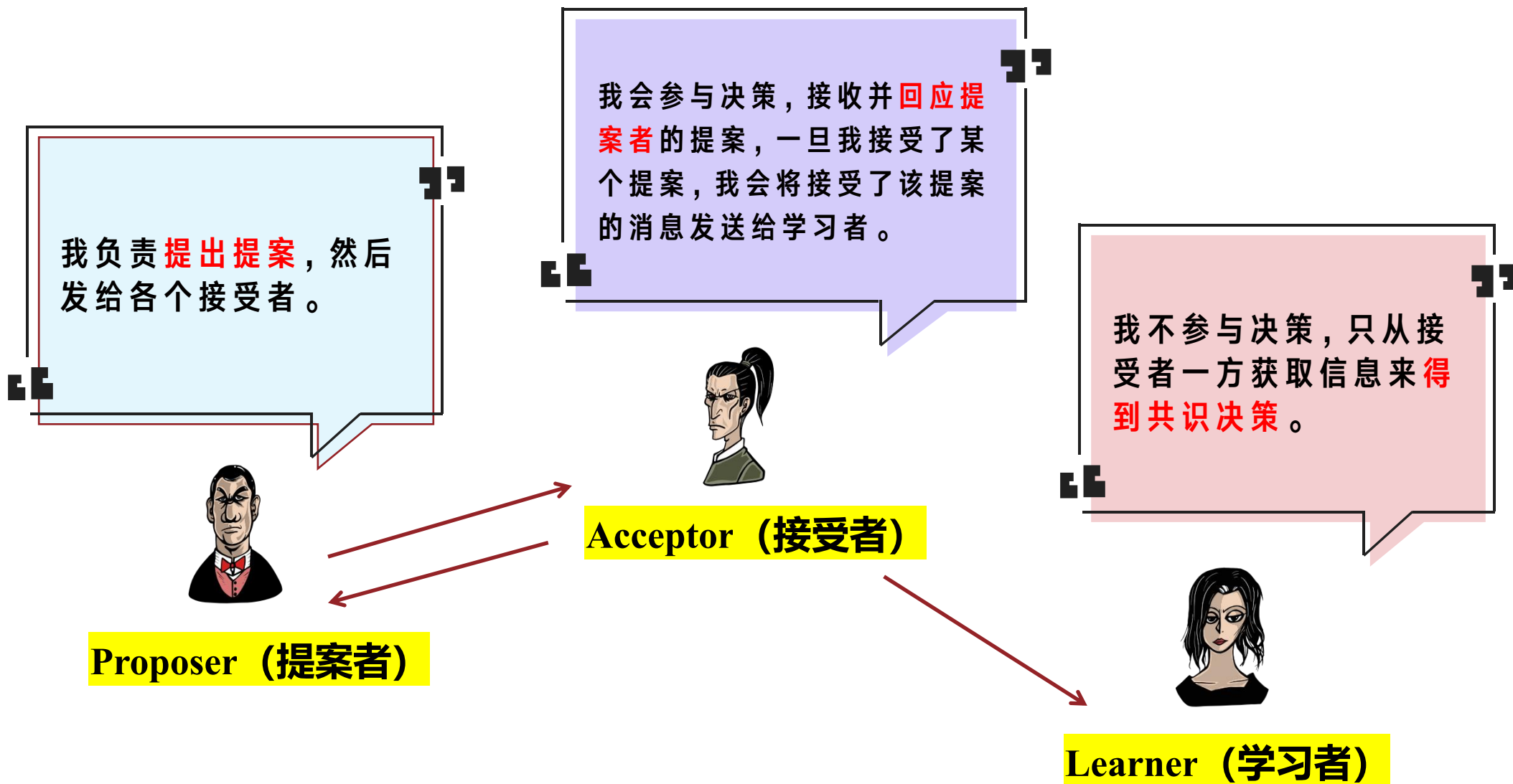
5.3.2 角色与问题描述

03

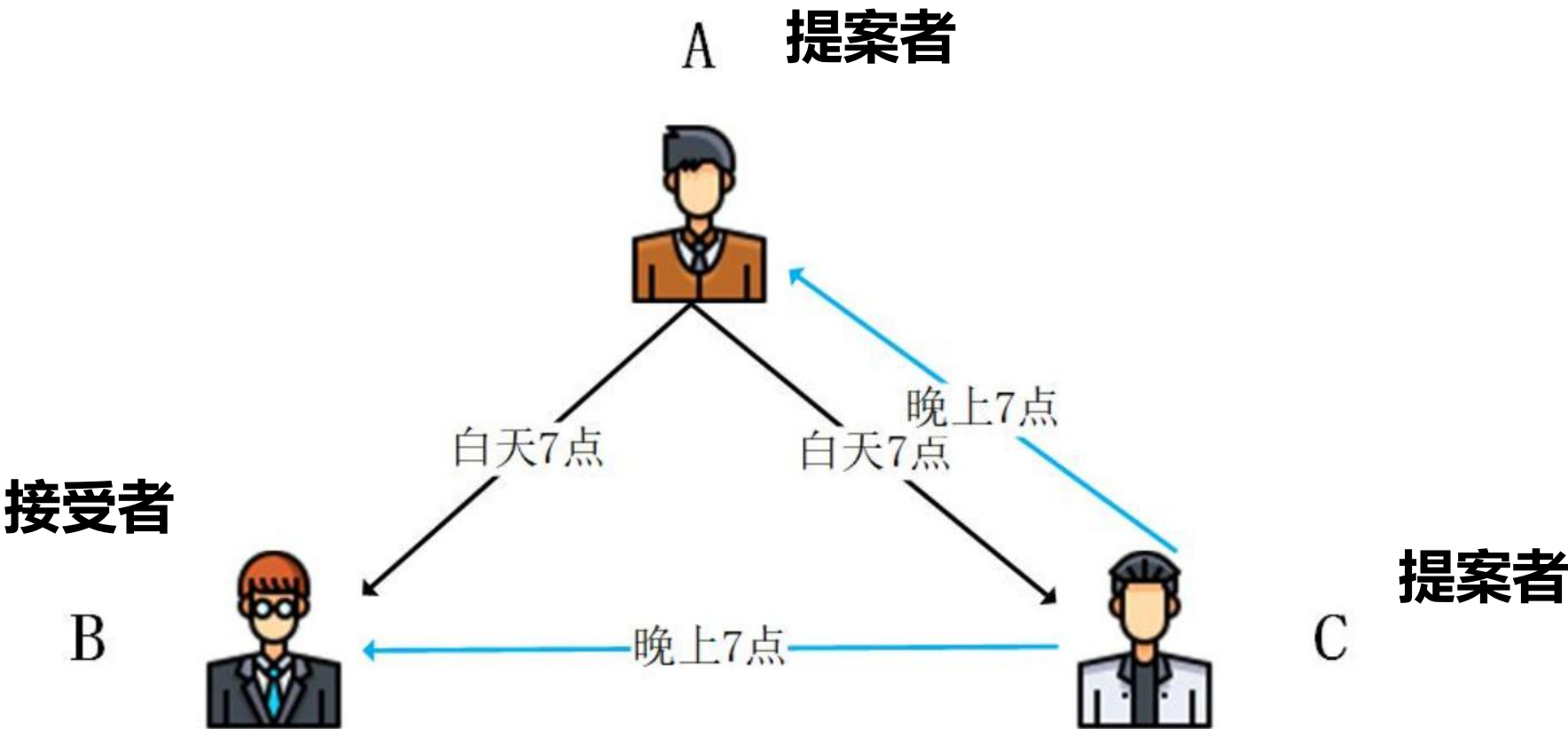
5.3.3 算法流程



5.3.2 角色与问题描述



5.3.2 角色与问题描述



5.3.2 角色与问题描述

共识的安全性要求

1. 只能选定被提案者提议出来的聚会时间；
2. 只有一个聚会时间能够被选定；
3. 只有一个聚会时间真的被选定了，学习者才能够学习到这个聚会时间是共识的结果。（即如果学习者认为这个聚会时间被选定，那这个聚会时间一定是真的被选定且不会被推翻）



5.3 Paxos算法

01

5.3.1 驱动案例

03

5.3.2 角色与问题描述

03

5.3.3 算法流程



5.3.3 算法流程

1. 值的选定

Question1：如何选定一个聚会时间？

首先如果接受者们不接受他所收到的第一个提案，那么共识就无法往下进行，因为第一个提案是必须被接受的，这是共识的启动过程

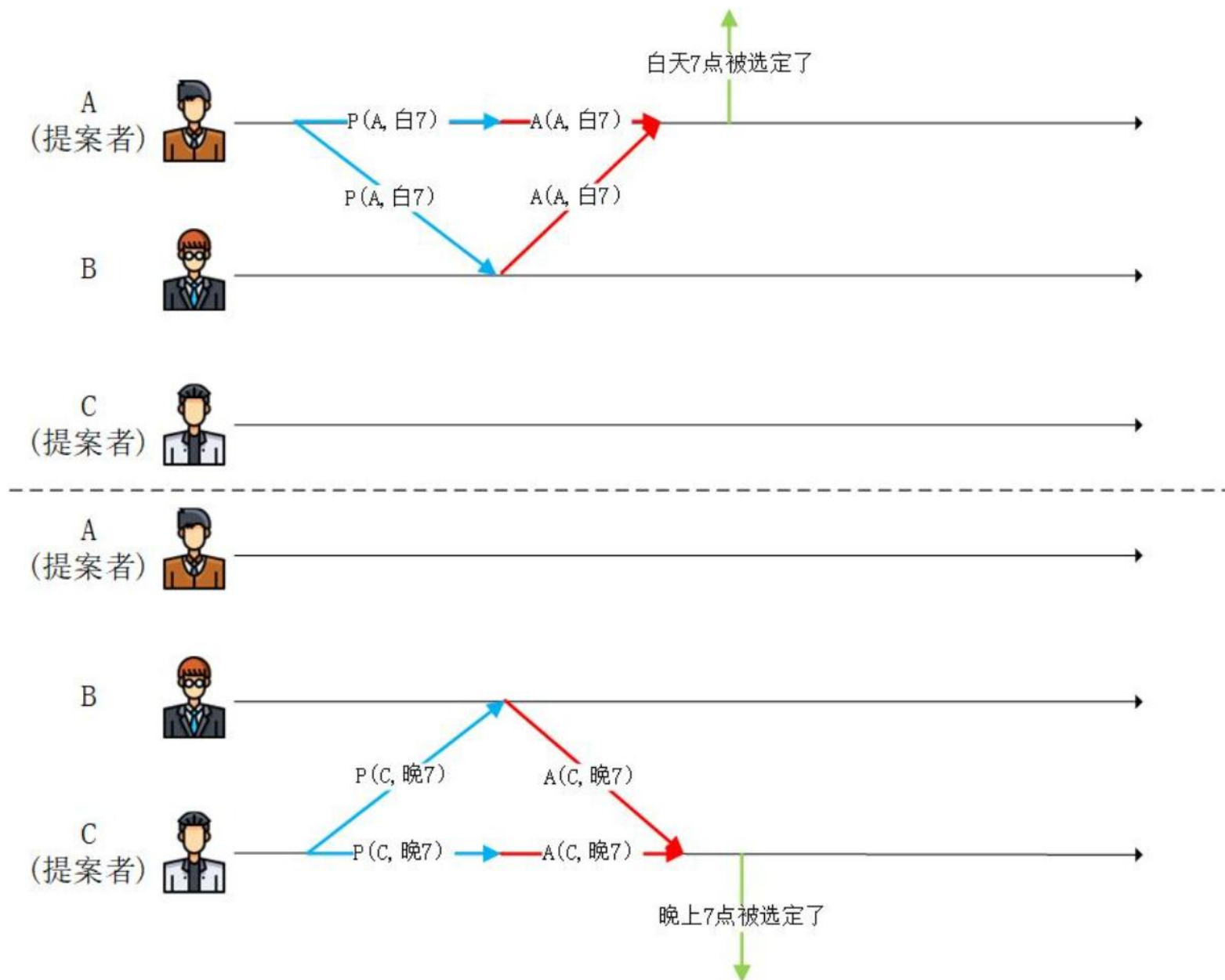


P1：每个接受者必须同意他收到的第一个提案

5.3.3 算法流程

Question2.1: 接受了第一个提案之后, 对后续的提案要如何处理?

示例: 接受者B既接受了提案者A的白天7点聚会的提案, 即 $P(A, \text{白}7)$ 又接受了C晚上7点的提案, 于是两个值都被选定了, 违背了安全性要求。



5.3.3 算法流程

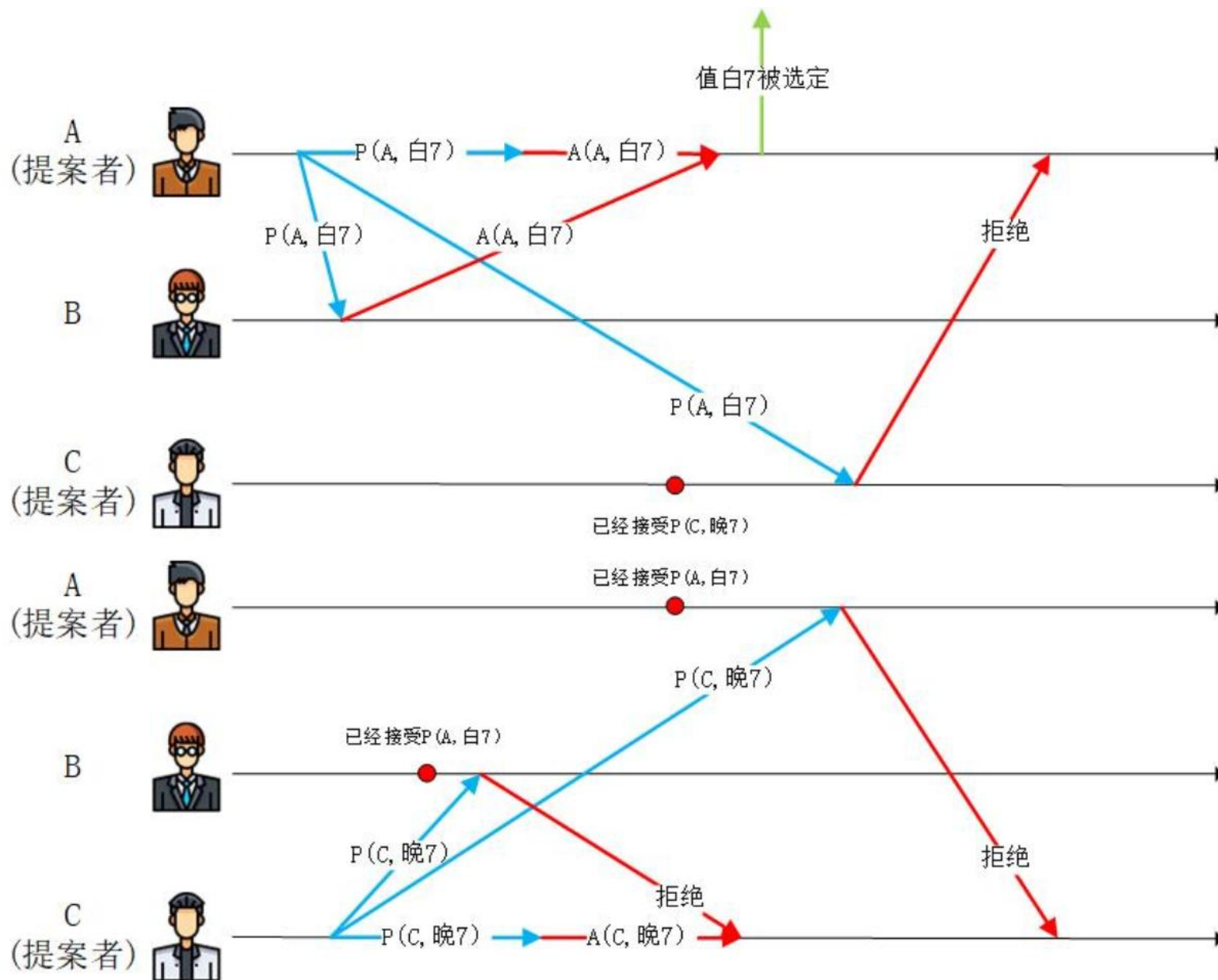
Question2.1：接受了第一个提案之后，对后续的提案要如何处理？

分析：上述情况的“罪魁祸首”是B，如果接受者只能接受自己收到的第一个提案（即不能接受后面的提案），那么只有一个值能够形成多数派，因为每个人只能投一票，那么只会有一项提案的票数超过一半，此时便可以达成共识。

5.3.3 算法流程

Question2.1: 接受了第一个提案之后, 对后续的提案要如何处理?

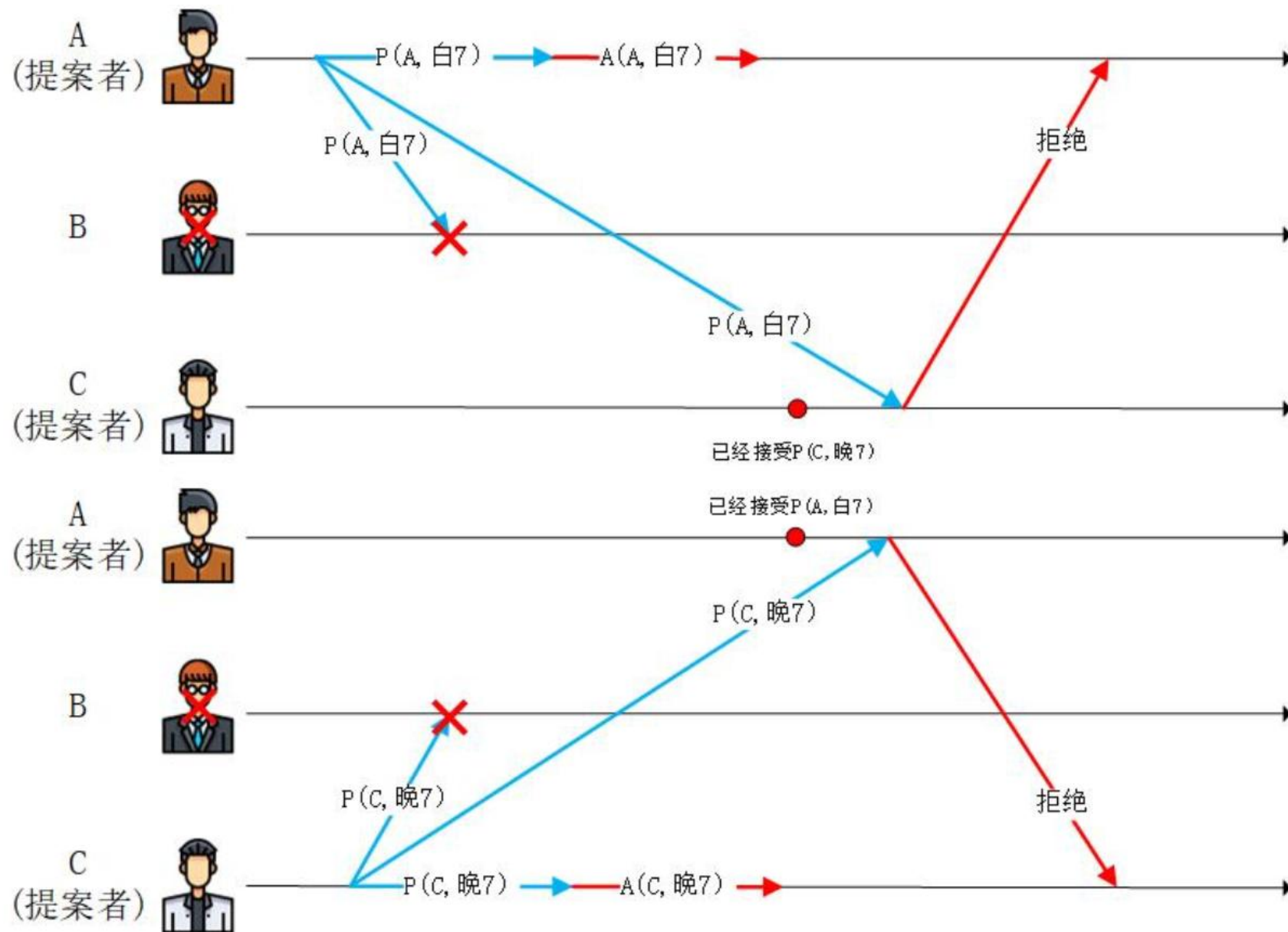
示例: 提案者A率先将自己的提案消息发送给接受者A和接受者B, 并从中获得了两票, 形成了多数派共识。而后面的提案者C发送的提案并不能够被A和B接受。



5.3.3 算法流程

Question2.2: 如果B因为故障而投不了票了呢?

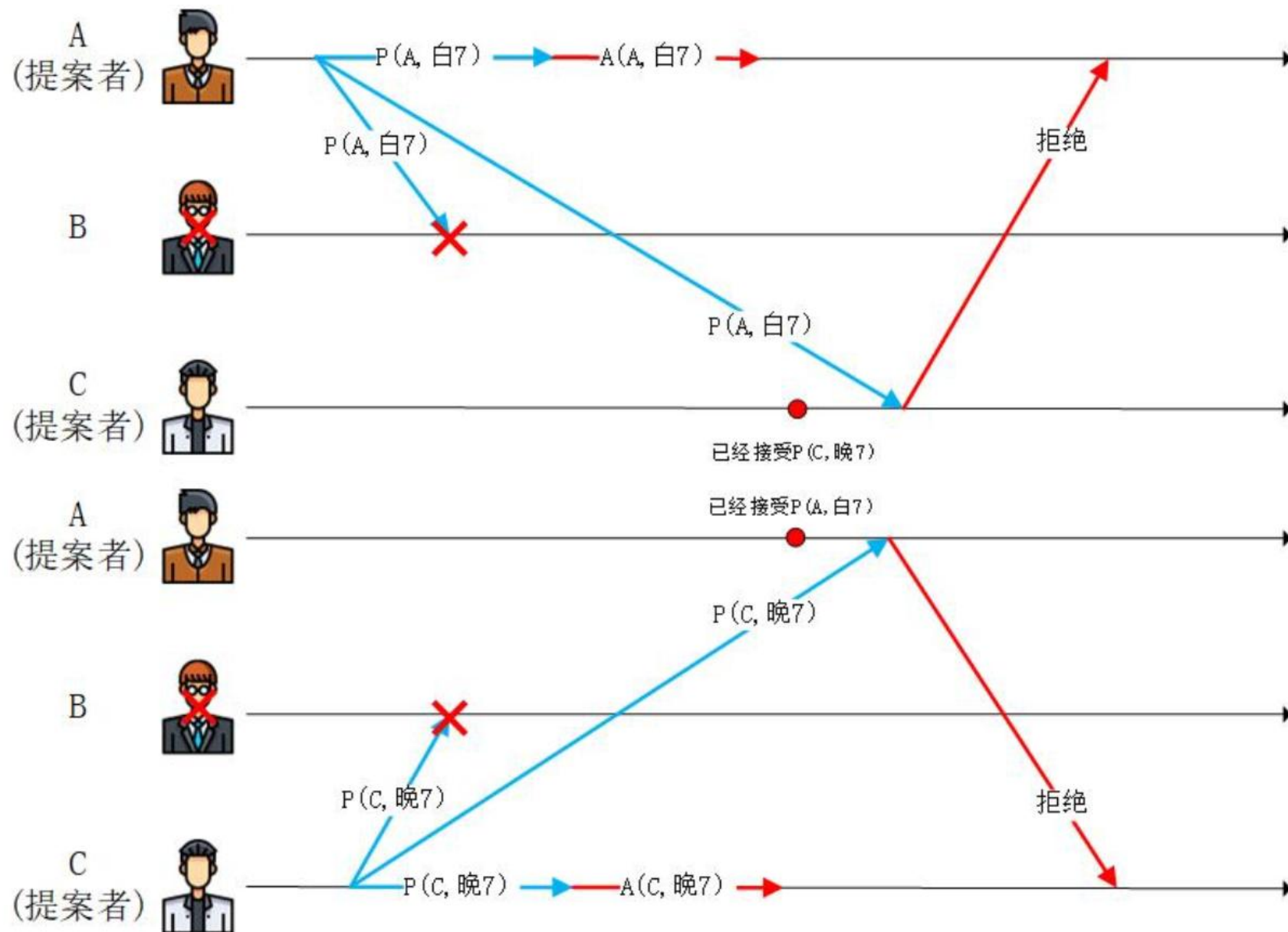
示例： Q2 的解决方案可能对单个接受者产生了过度依赖。A 和 C 只接受了自己的提案，但因为 B 失去联系，A 和 C 都无法从 B 获得投票，整个共识因 B 一人的故障而崩溃。



5.3.3 算法流程

Question2.2: 如果B因为故障而投不了票了呢?

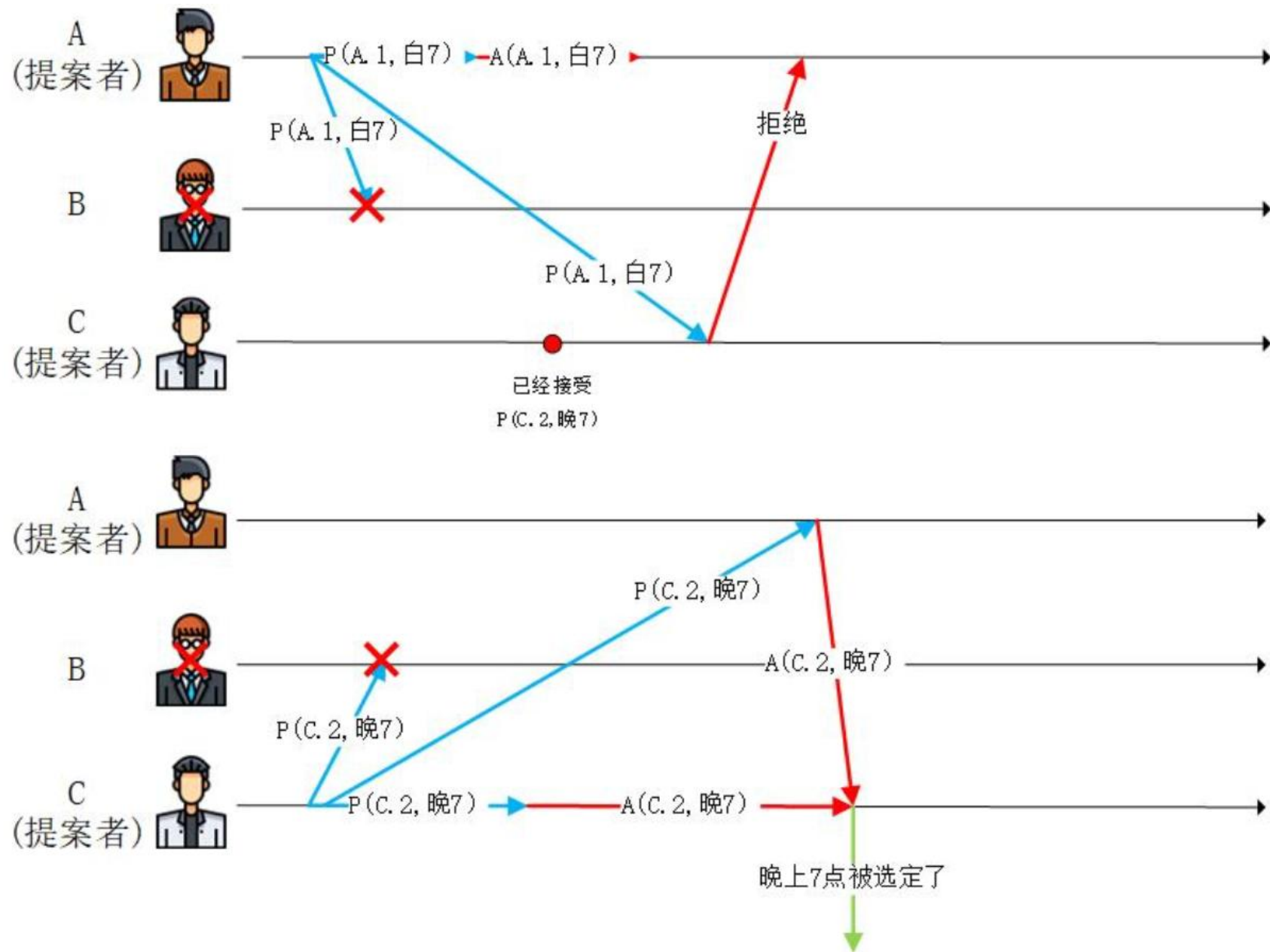
分析：当B发生故障后，如果想要共识能够进行下去，就需要A和C至少其中一个接受更多的提案：如果某个提案者的提案在一段时间内不能形成多数派，那么提案者会提出**更新的提案**。



5.3.3 算法流程

Question2.3: 怎么判断一个提案比另一个提案更新?

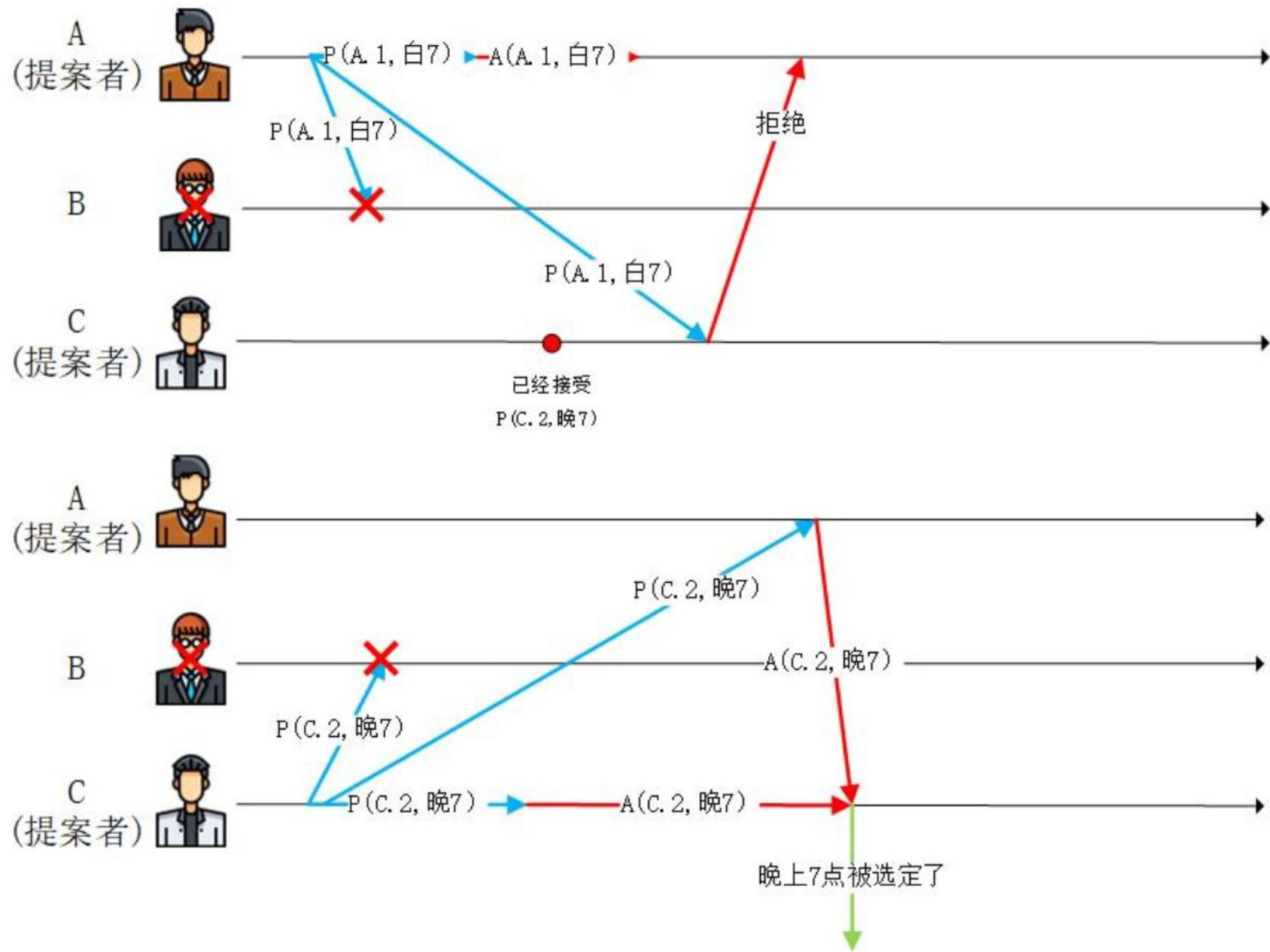
分析：收到提案的顺序没有参考价值，因为消息的传播速度是不确定的，更新的提案并不意味着更晚收到。因此，在发起提案时带上一个**提案编号**，且每个提案者提出的提案编号是互不相同的。



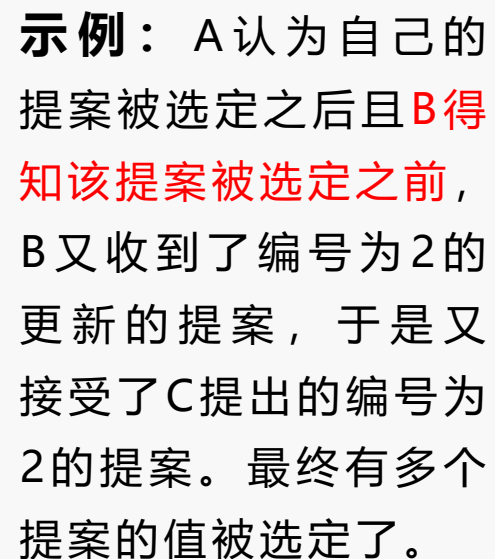
5.3.3 算法流程

Question2.3: 怎么判断一个提案比另一个提案更新?

示例： A将编号为1的提案发送给C，但是由于C接受了自己提出的编号为2的提案，A的提案号比C旧，因此C会拒绝A的提案，并且C发送给A的提案会被A接受，于是两者就晚上7点的提案达成共识。

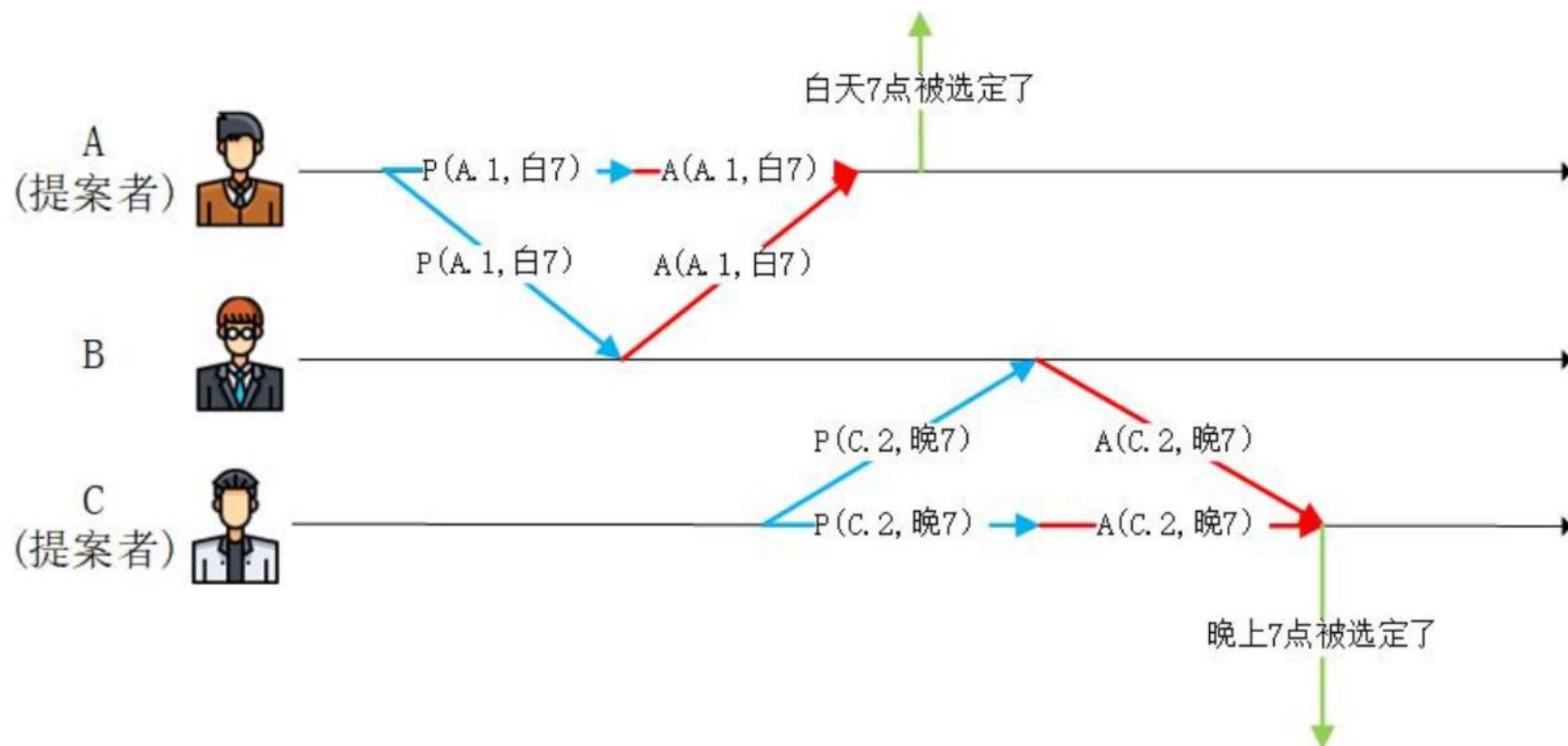


Question2.4: 如何解决多个值被选定的问题?



5.3.3 算法流程

Question2.4: 如何解决多个值被选定的问题?



分析：安全性的要求是：只有一个值可以被选定。如果提案者A和提案者C提出的提案的聚会时间都是白天7点呢？那么不管接受者接受了哪个提案，都等于是就 **同一个值** 达成了共识。

➡ P2: 如果某个提案的值v被选定了，那么更新的提案一定是值为v才能被选定

5.3.3 算法流程

Question3：更新的提案怎么确定被选定的值？

分析： 因为每个人必须同意收到的第一个提案，但每个人收到的第一个提案的值可能不同，而接受者无法预测哪一个值是将来可能会被选定的值。

与其花费大量的开销预测，不如让接受者解决问题：

如果接受者们得知提案者发出编号为 n 的提案，那么接受者：

- 保证之后不再接受编号比 n 小的旧提案。
- 如果之前接受过旧的提案，就把这个消息传递给新提案的提案者。

5.3.3 算法流程

Question3：更新的提案怎么确定被选定的值？

—— Paxos将算法分为了两个阶段：

1. Prepare (准备) 阶段

2. Accect (接受) 阶段



注意区分不同阶段的响应内容：

提案者：

- 准备阶段——**准备请求**
- 接受阶段——**接受请求/提案**

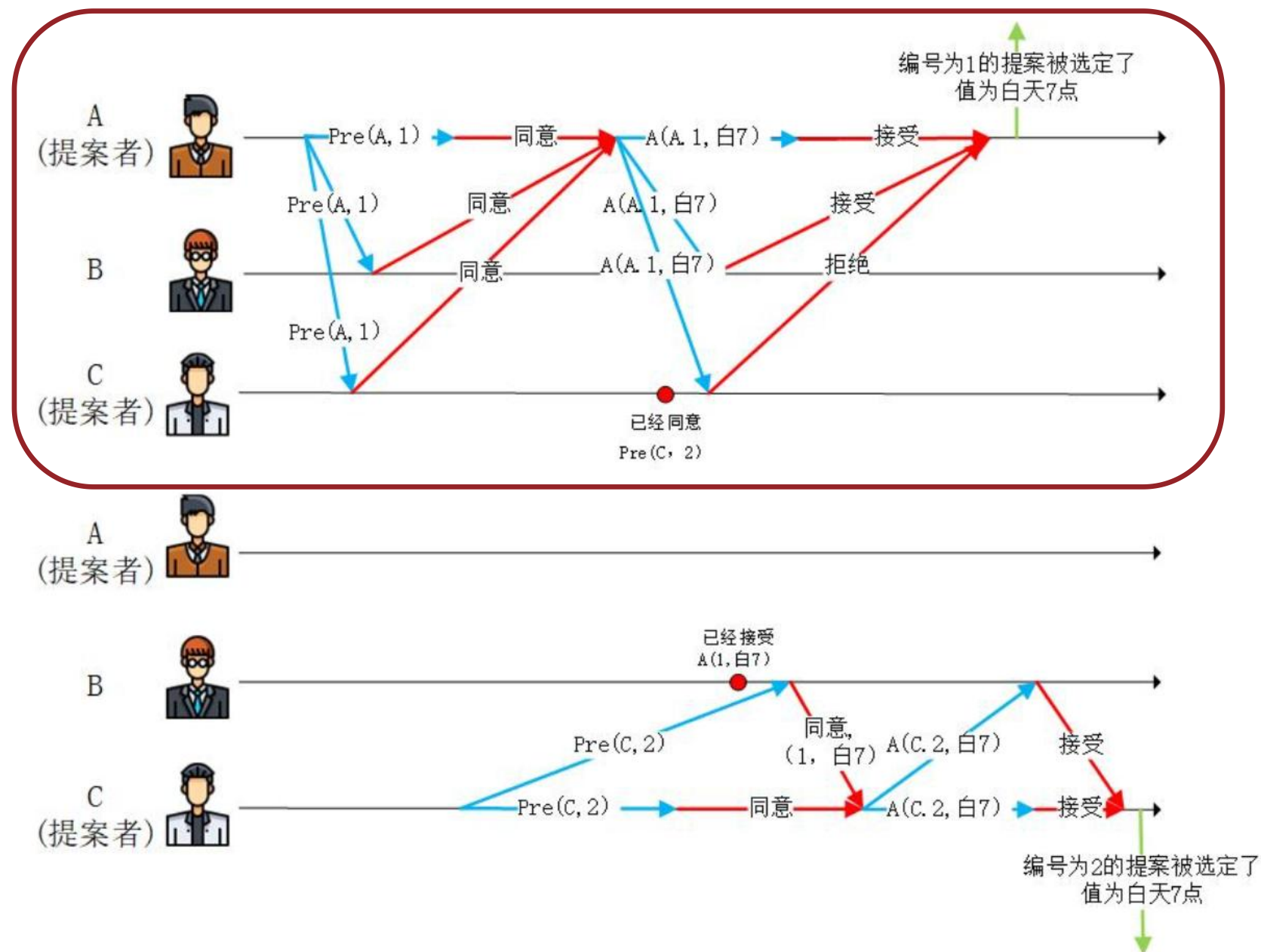
接受者：

- 准备阶段——**同意**请求
- 接受阶段——**接受**提案

5.3.3 算法流程

Question3: 更新的提案怎么确定被选定的值?

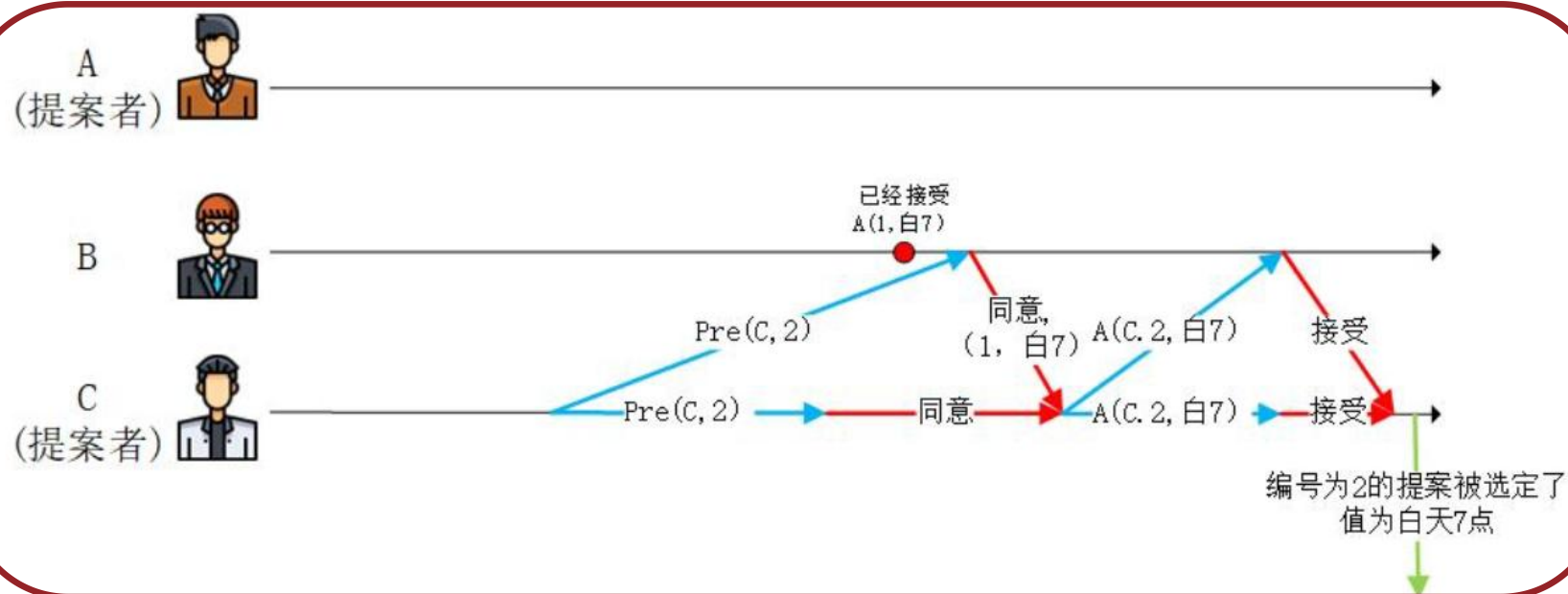
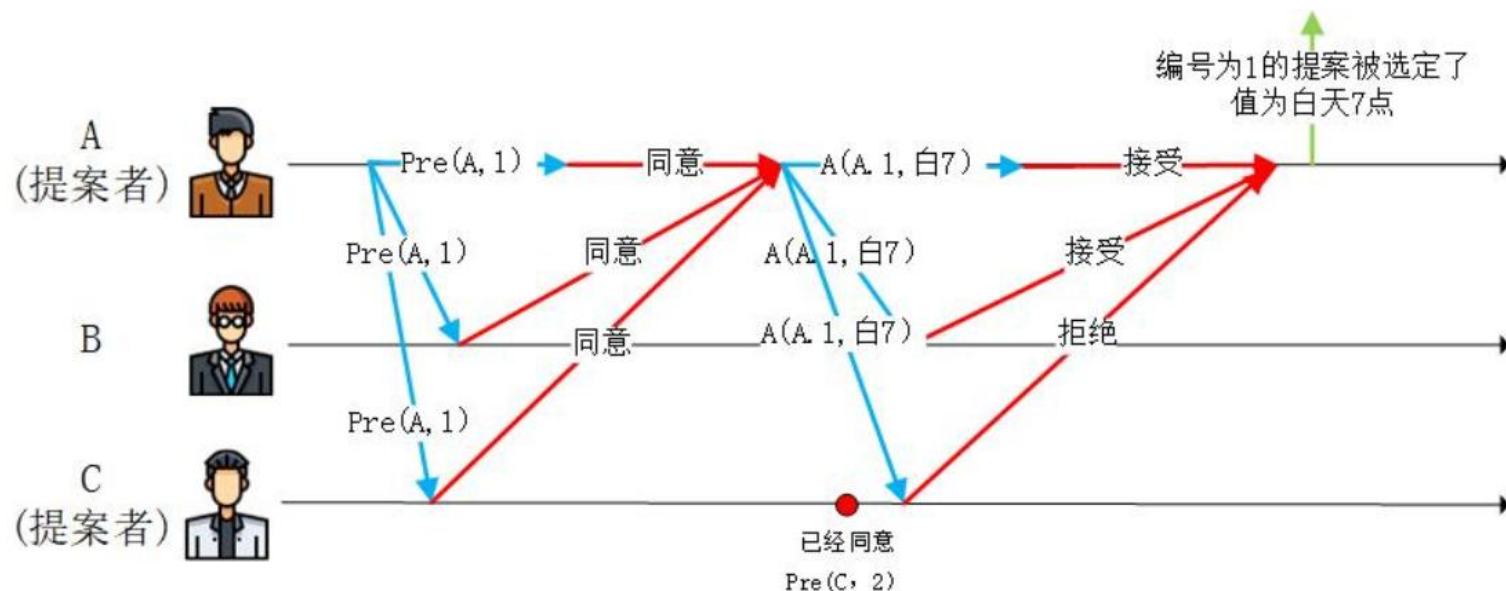
示例：A发送**准备请求**给ABC，因为BC没有**接受**过提案，提案者A按自己喜好选择了白天7点作为提案值**发送提案**。但C由于先收到自己编号为2的准备请求，根据前面的分析，C会作出**承诺**拒绝编号小于2的A的提案。



5.3.3 算法流程

Question3: 更新的提案怎么确定被选定的值?

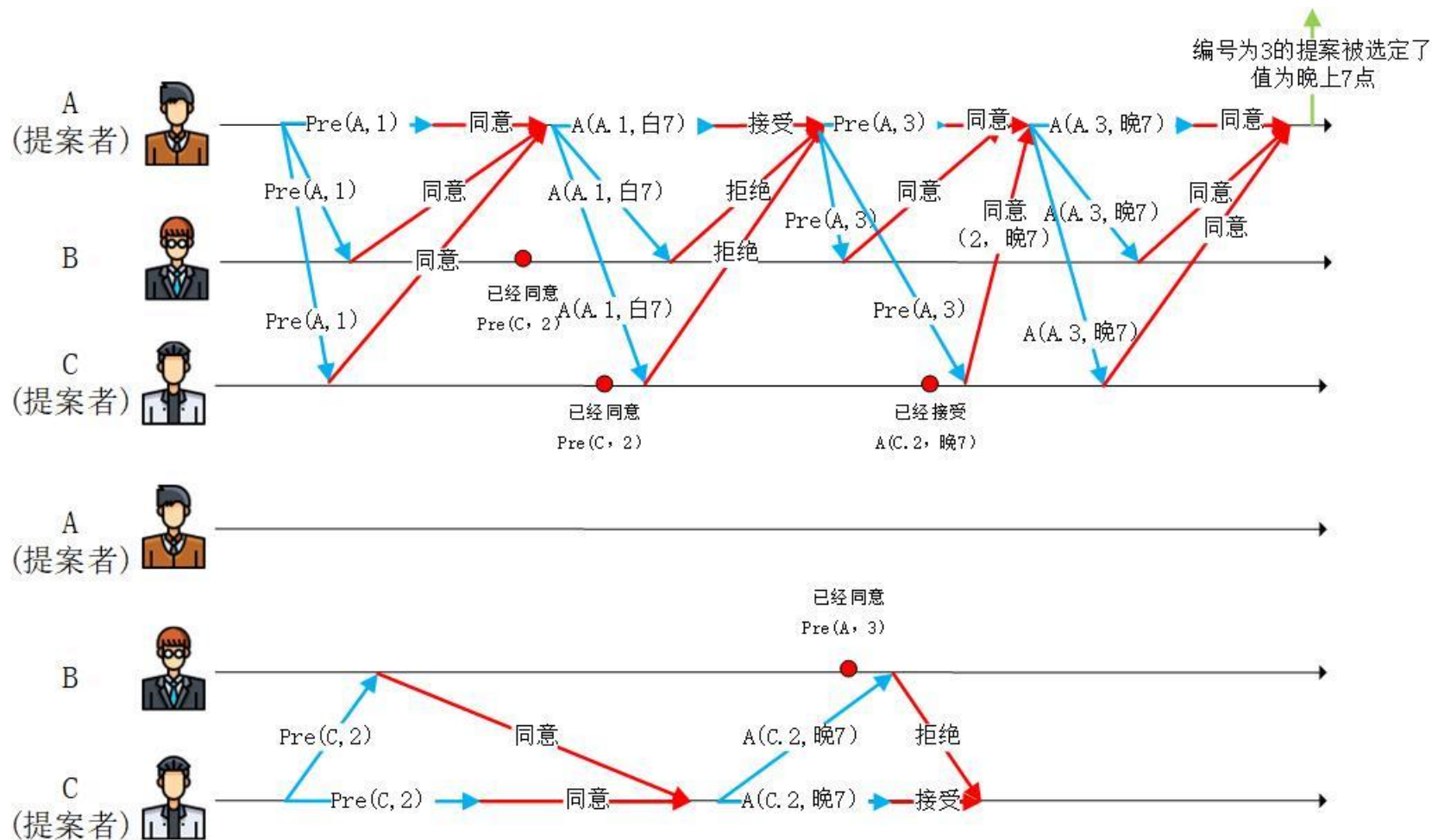
示例：即使A和B接受了编号1的提案，编号为2的提案的选定依然能够达成共识。同样根据前面的分析，C会得知B曾接受了提案为1，并且C会同步提案1的值，所以C发送的提案值也是白天7点，解决了多值选定的问题。



5.3.3 算法流程

Question3: 更新的提案怎么确定被选定的值?

示例：即使A和B接受了编号1的提案，编号为2的提案的选定依然能够达成共识。同样根据前面的分析，C会得知B曾接受了提案为1，并且C会同步提案1的值，所以C发送的提案值也是白天7点，解决了多值选定的问题。



5.3.3 算法流程

两阶段规则：

1. Prepare (准备) 阶段

提案者： 选择一个提案编号 n ，然后向所有的接受者发送这个编号为 n 的**准备请求**（不附带值，即示例中的聚餐时间）

接受者： 收到编号 n 的**准备请求**，如果 n 比之前收到的任何准备请求的编号小，则拒绝；如果比之前收到的编号都大，则向提案者作出**承诺**：

- 之后不再会**接受**编号小于 n 的提案
- 之前已**接受过**某项提案 m ，把提案 m 发给提案者

5.3.3 算法流程

两阶段规则：

2. Accept (接受) 阶段

提案者： 收到半数以上接受者对之前准备请求的**同意回复**（编号为 n ）后才发送提案值，如果发现接受者之前接受过某项提案：

- 从这些提案中选出**编号最大的提案 m** 的值作为要发送的值

接受者： 收到编号 n 的**提案**，如果之前没有收到并响应编号大于 n 的准备请求（即对大于 n 的提案作出承诺）：

- 接受这个编号为 n 的提案

否则拒绝

5.3.3 算法流程

Question3: 更新的提案怎么确定被选定的值?



根据规则P2, 更新的提案需要确定之前被选定的值, Paxos的两阶段方法中提案者选择了接受者之前接受的提案编号最大的提案值 v 作为更新提案的值, 但该值一定就是之前唯一有可能被选定的提案值吗?

(即接受者可能已经接受了多个提案, 怎么确定提案编号最大的提案的值是唯一被选定的那个值)

回顾 — P2: 如果某个提案的值 v 被选定了, 那么更新的提案一定是值为 v 才能被选定

5.3.3 算法流程

Question3: 更新的提案怎么确定被选定的值?

证明: 让我们通过具体案例考虑如下情形, 假设**编号1**和**2**的提案都被选定了, 那么一定有一个**接受者B**, 既接受了**提案1**, 也接受了**提案2**, 那么它在收到**编号3**的准备请求时会告诉提案者自己接受了这两个提案。根据规则, 这个提案者会选择编号最大的 (**提案2**) 作为更新的提案 (**提案3**) 的同步值。

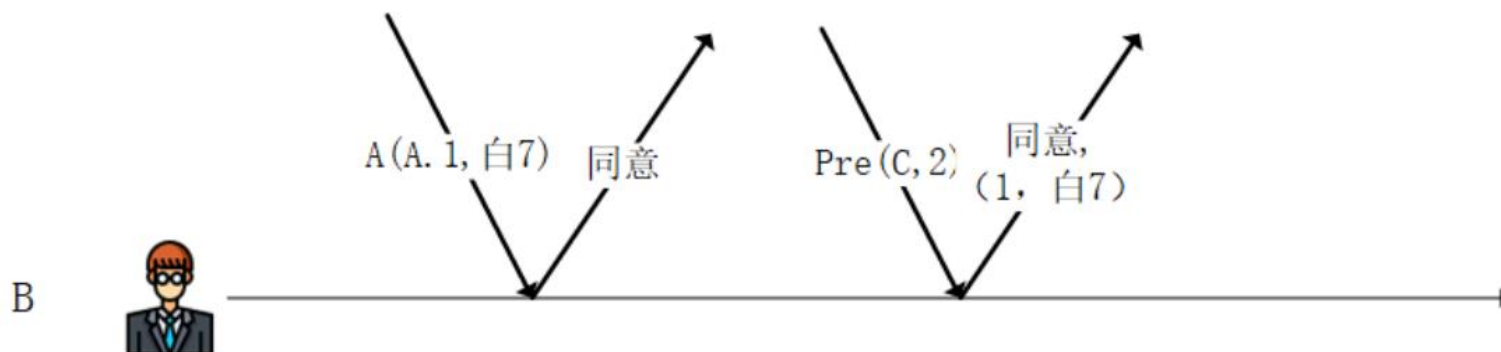
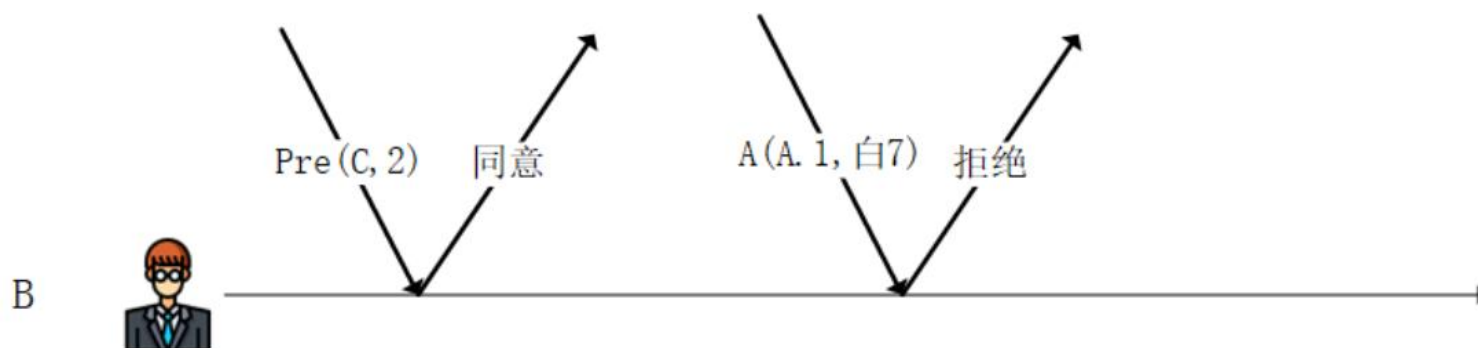
因此, 要保证提案者收到的编号最大的那个提案值是之前**唯一**有可能被选定了的提案值, 只要**提案1**和**提案2**有相同的值就可以了。 (只要能证明该点, 即使有更多被选定的值, 都可以从 $v(n-1)=v(n)$ 开始递归证明**1,2,...,n**号提案具有相同的值)

5.3.3 算法流程

Question3: 更新的提案怎么确定被选定的值?

证明： 提案1和提案2被选定意味着他们一定都被多数接受者同意了接受请求和准备请求，那么也一定有一个接受者（假设为B），既同意了提案1的接受请求，也同意了提案2的准备请求，并且成功响应给提案者
考虑接受者B：

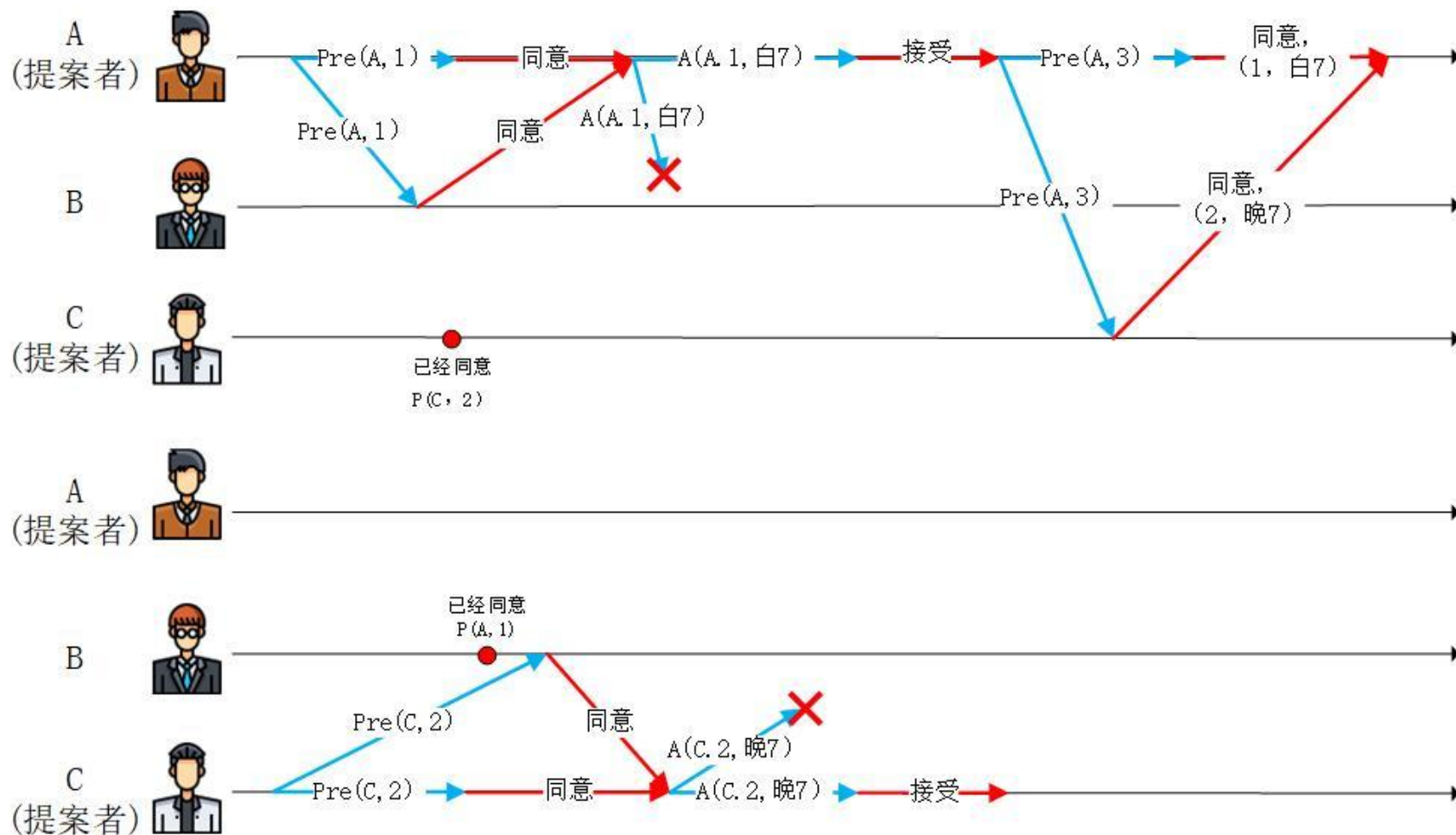
1. 如果B先收到了编号为2的**准备请求**：根据承诺，B一定不会接受编号为1的提案
2. 如果B先收到了编号为1的**接受请求**：那么该提案者就会选择白天7点作为提案2的值，保证了提案1和2的值是一致的



怎么确定提案1和提案2有相同的值?

5.3.3 算法流程

Question3: 更新的提案怎么确定被选定的值?

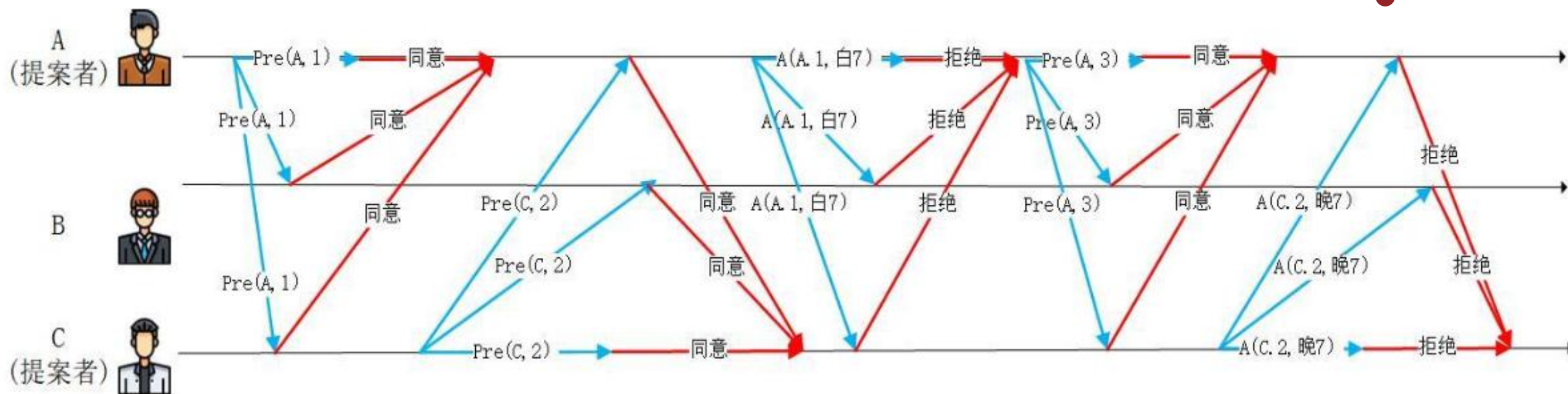


示例： A 因为长时间等待，提出了更新的提案3发送给C，C返回给A自己之前接受的编号为2的提案，A自己之前也接受了自己编号为1的提案，于是A从两个提案中选择了编号为2的晚上7点作为“**被选定值**”。

5.3.3 算法流程

2. 活锁处理

示例： A发送提案1的接受请求，但是因为ABC收到了提案2的准备请求而作出承诺，提案1的接受请求被拒绝，于是A又提出了新的提案3，在C发送提案2接受请求之前将提案3的准备请求发给了ABC，于是ABC又作出承诺拒绝了提案2的接受请求，以此类推不断循环。



5.3.3 算法流程

2. 活锁处理

方法1： 随机化提案者提案的时间，只要接受请求能在更大编号的准备请求到达之前送达到大多数接受者，那么就可以避免活锁，但这种方法很不稳定，不确定什么时候能够跳出循环。

方法2： 在众多的提案者中选取一个Leader(领导者)，只有领导者可以提出提案，如果没有其他提案者竞争，那么这个领导者肯定能提出一个编号最大的提案。但如果Leader挂掉了，那么还要重新选举另一个Leader，并且选举Leader本身也可能锁住。

5.3.3 算法流程

3. 获知被选定的值

Question: 学习者怎么获知接受的提案值被选定?

学习者在此时发挥自己的职能，学习者需要找出某个已经被大多数接受者接受了提案。一种方法是让接受者在接受某个值后将这个消息发送给某个特定的学习者，由于没有拜占庭错误，所以这个学习者可以在得知某个值被选定了以后发送给其他的学习者。

一般来说，学习者是数据备份节点，比如主从架构中的从节点，被动地接受数据，容灾备份。

第五章、容错与共识

01

5.1 故障类型

02

5.2 故障检测与恢复

03

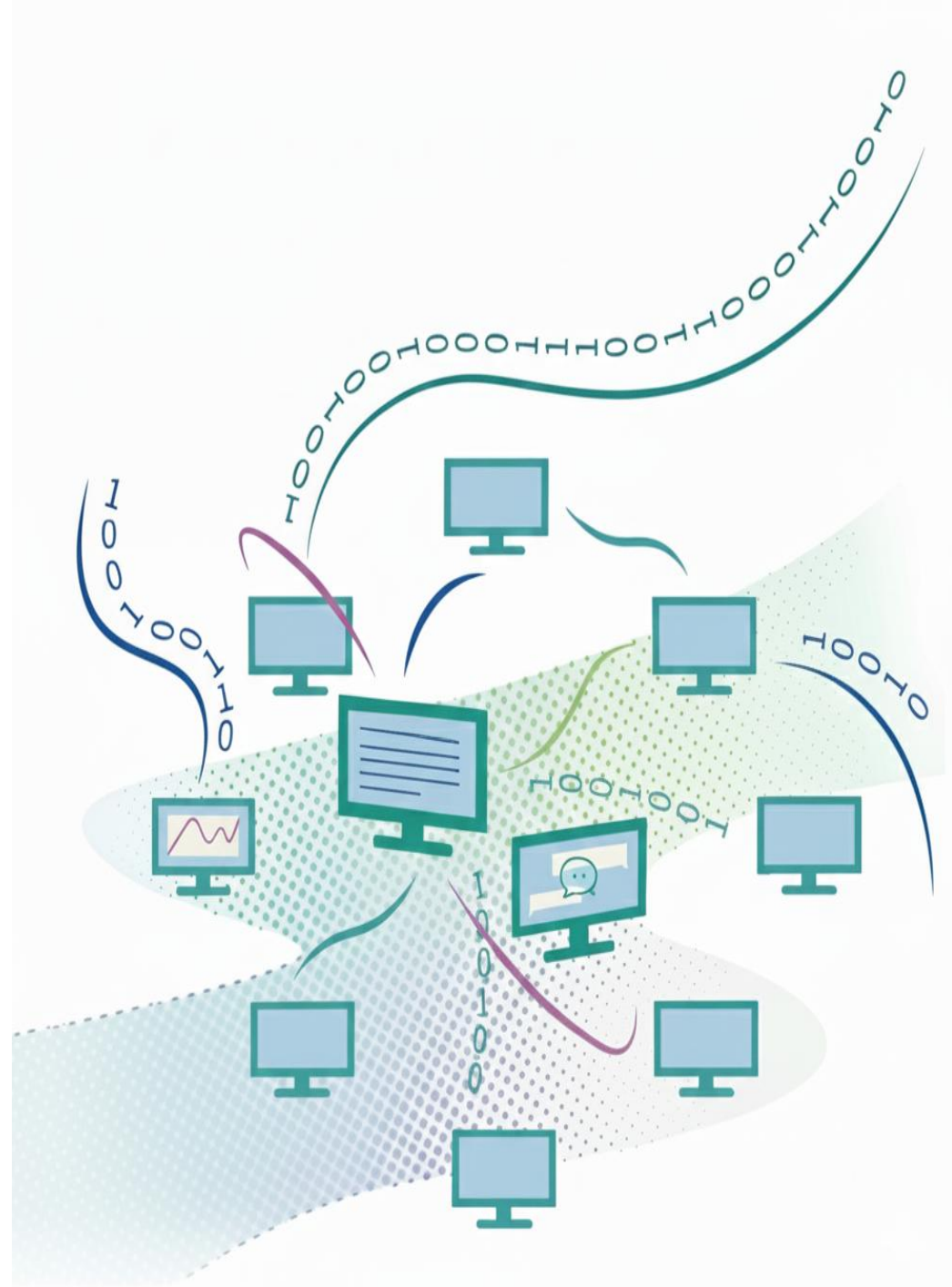
5.3 Paxos算法

04

5.4 Raft算法

05

5.5 PBFT算法*



5.4 Raft算法

01

5.4.1 算法概述

02

5.4.2 Leader选举

03

5.4.3 日志同步

04

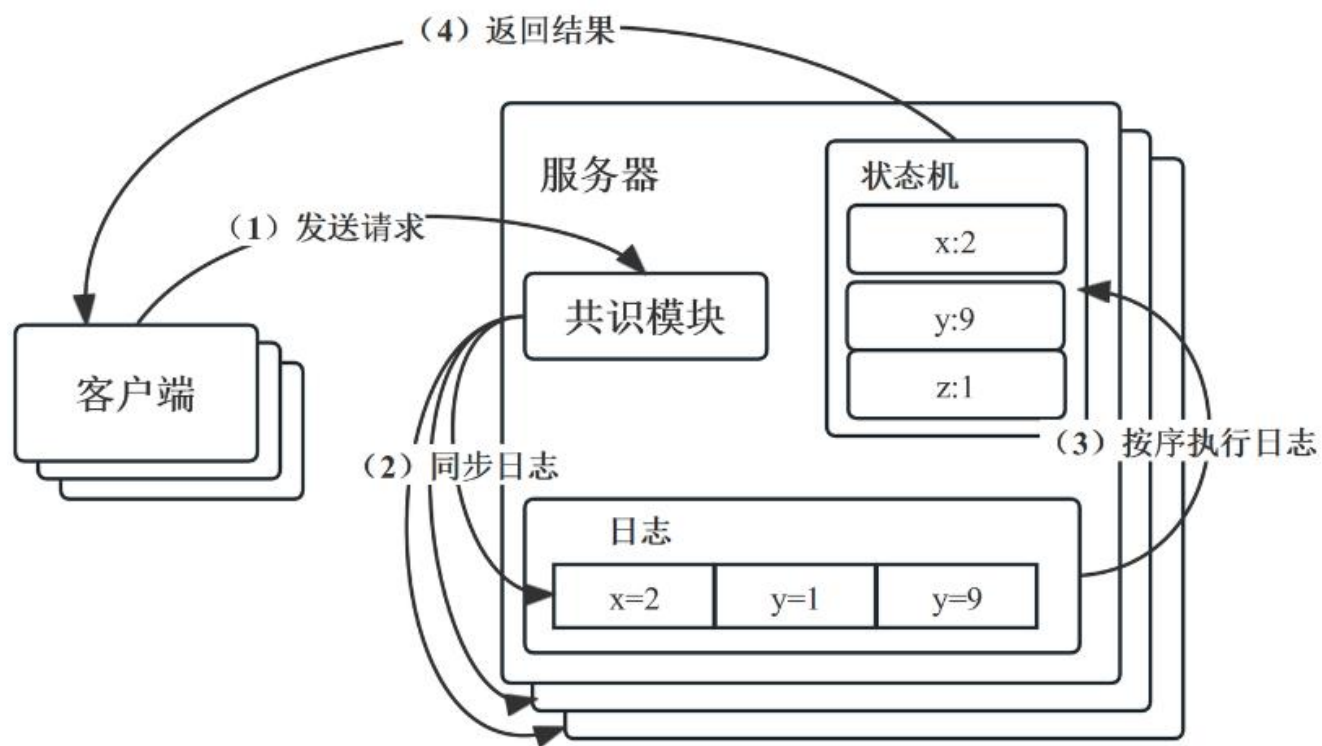
5.4.4 安全性



5.4.1 算法概述

Raft 算法是为了实现高效、可靠的复制状态机而设计的一种共识算法，保证不同的服务器上的日志副本的一致性：

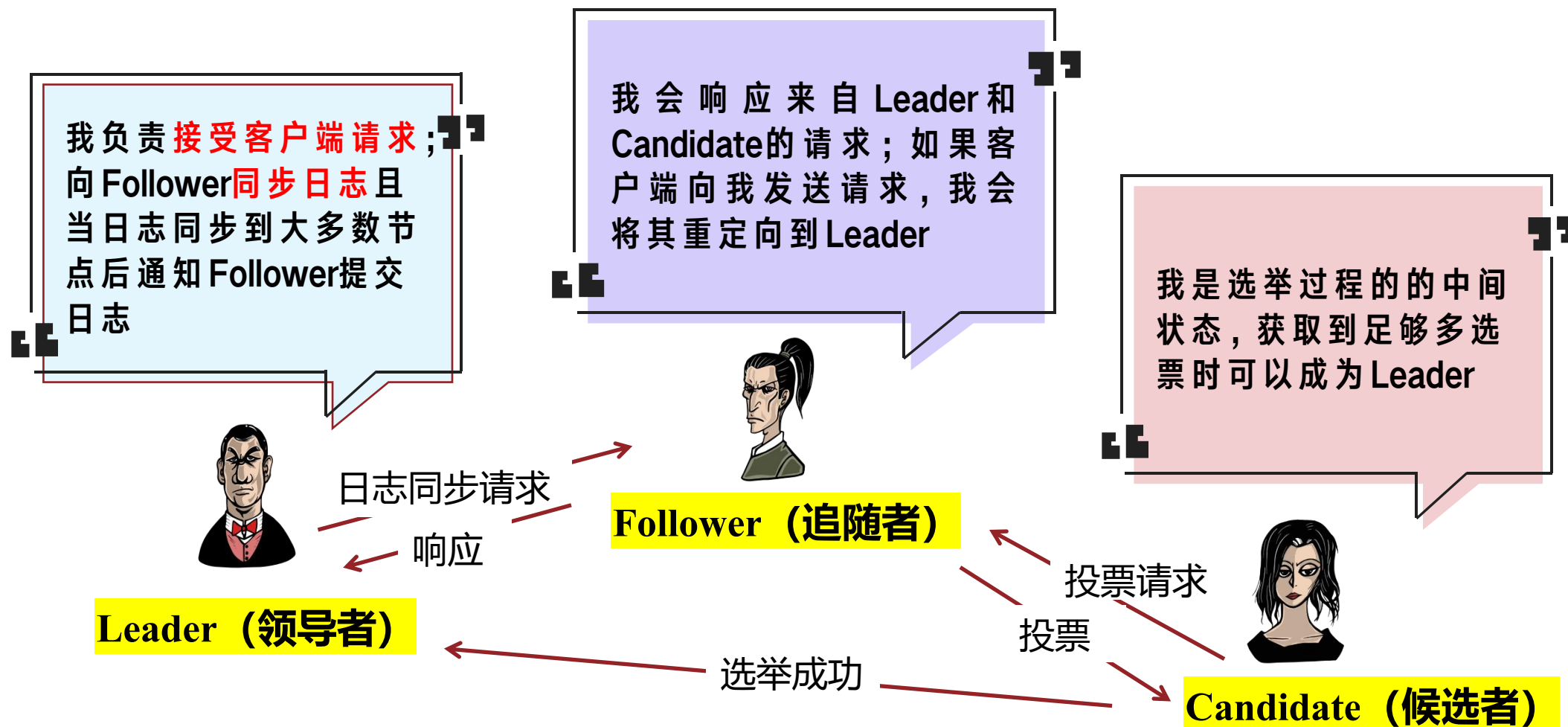
复制状态机：



1. 客户端发送多个命令给服务器；
2. 服务器的共识模块将多个命令记录在日志中，并使不同服务器上的日志保持一致；
3. 服务器按序执行日志的命令记录即可得到相同的输出；
4. 服务器将一致的结果返回给客户端；

5.4.1 算法概述

节点状态：领导者 (Leader) 追随者 (Follower) 候选者 (Candidate)



5.4.1 算法概述

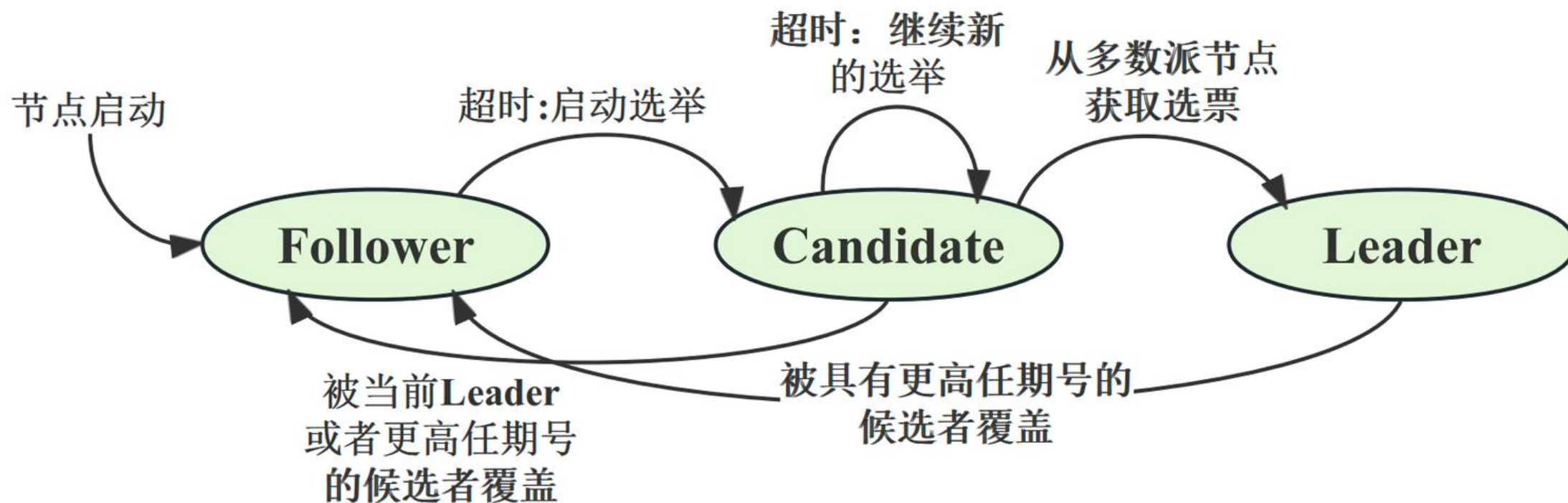
任期 (Term) :

提案者： Raft算法中将节点的时间推进划分成一个个**任期**，且每个任期并非固定的时间间隔，任期号是随着“**计时器超时启动选举**”这一事件递增的。任期的存在是为了辅助Leader选举：

任期号更小的Leader会被认为是过时的，且接收节点接收到任期号更小的节点的请求时也会拒绝请求，以此避免系统中出现多个同时活动的Leader

5.4.1 算法概述

节点状态：领导者（Leader） 追随者（Follower） 候选者（Candidate）



5.4 Raft算法

01

5.4.1 算法概述

02

5.4.2 Leader选举

03

5.4.3 日志同步

04

5.4.4 安全性



5.4.2 Leader选举

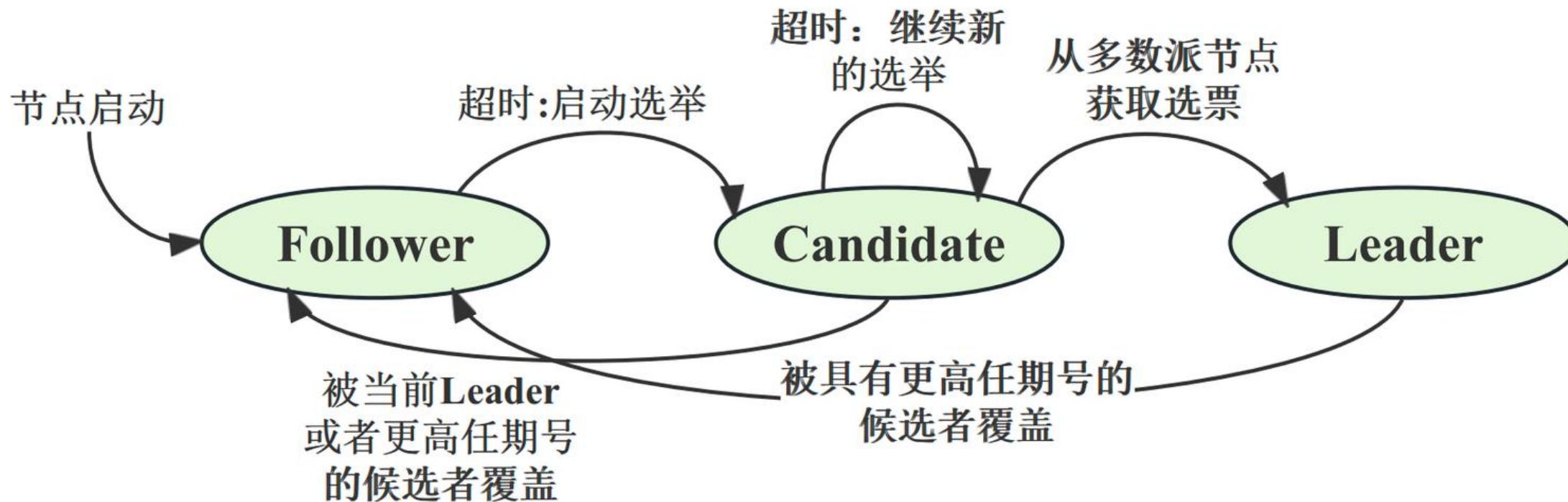
Question1：选择哪一个节点作为Leader节点？

Leader角色负责向Follower同步日志，根据复制状态机协议，相同的操作日志顺序和内容对系统的一致性保证起到关键作用。那么分布式系统中无数节点，该选择哪一个节点作为Leader节点？如果选择的Leader节点发生故障，又该如何保证系统继续正常提供服务？

→ 子问题1：Leader选举

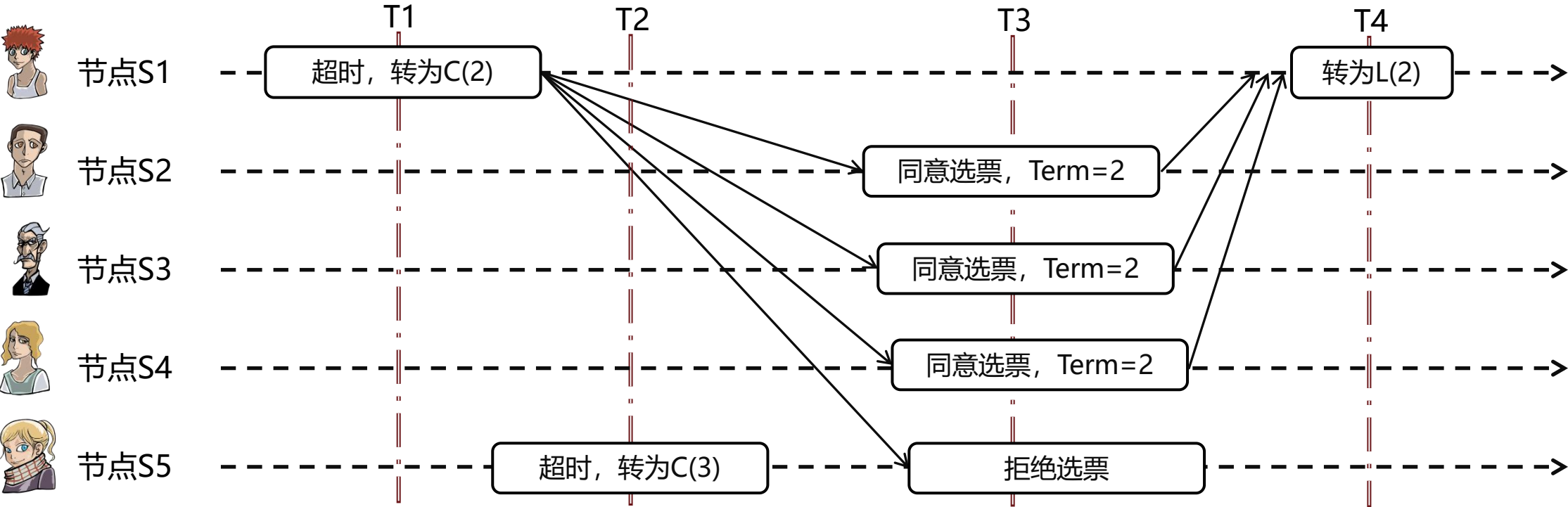
5.4.2 Leader选举

1. 系统中每个节点初始状态都为Follower且都有一个随机的超时时间，超时后节点会转为Candidate的中间状态触发选举过程，向其他节点发送**选举请求（RequestVote RPC）**并附带自己的任期号。



5.4.2 Leader选举

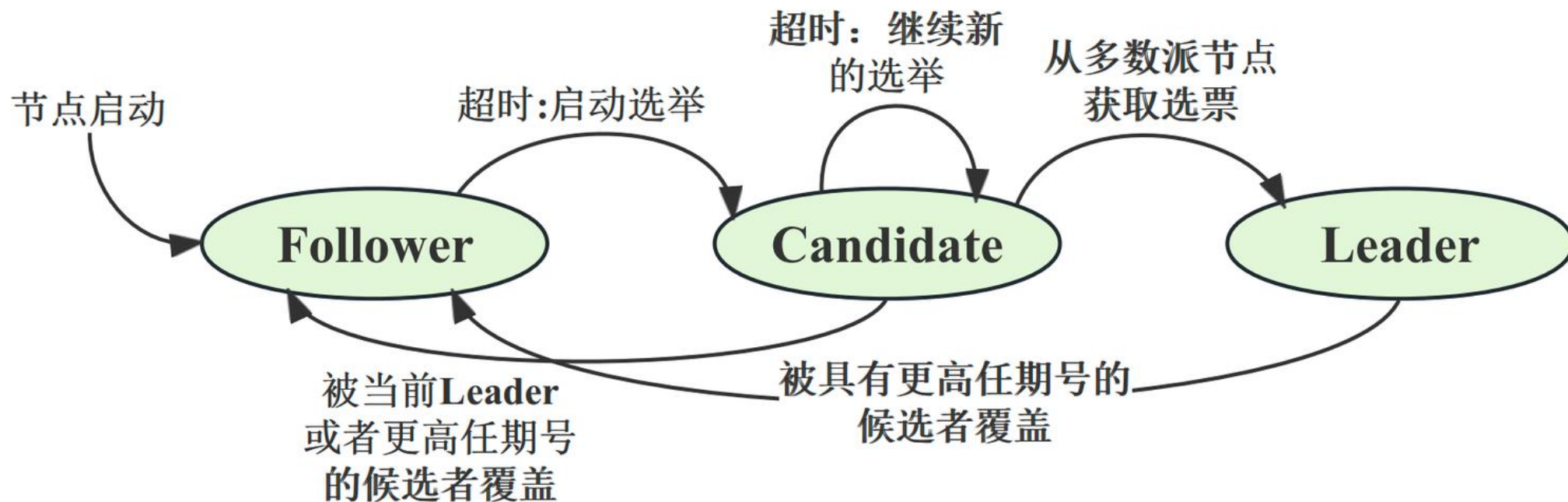
示例： 一个5节点集群，节点S1在T1时刻首先超时转为Candidate启动选举，C(2)表示其Candidate身份以及任期号Term为2；于是节点S1附带自己的任期号2向其他节点发送了**选举投票请求**；



5.4.2 Leader选举

2. 对于接收到选举请求的节点，若Candidate具有大于当前节点的任期号和更完整更新的日志信息（涉及日志同步，会在下一节展开说明）：

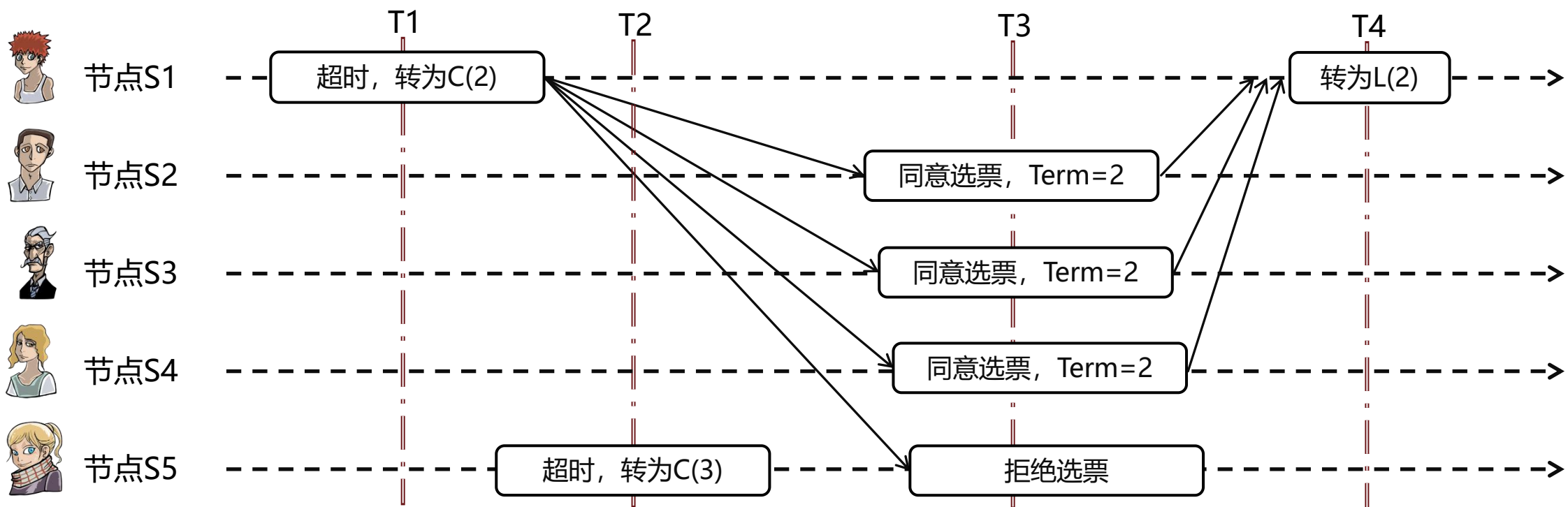
- 响应当前的选举请求，即投票给发起当前请求的节点
- 重置计时器并更新自己的任期号



5.4.2 Leader选举

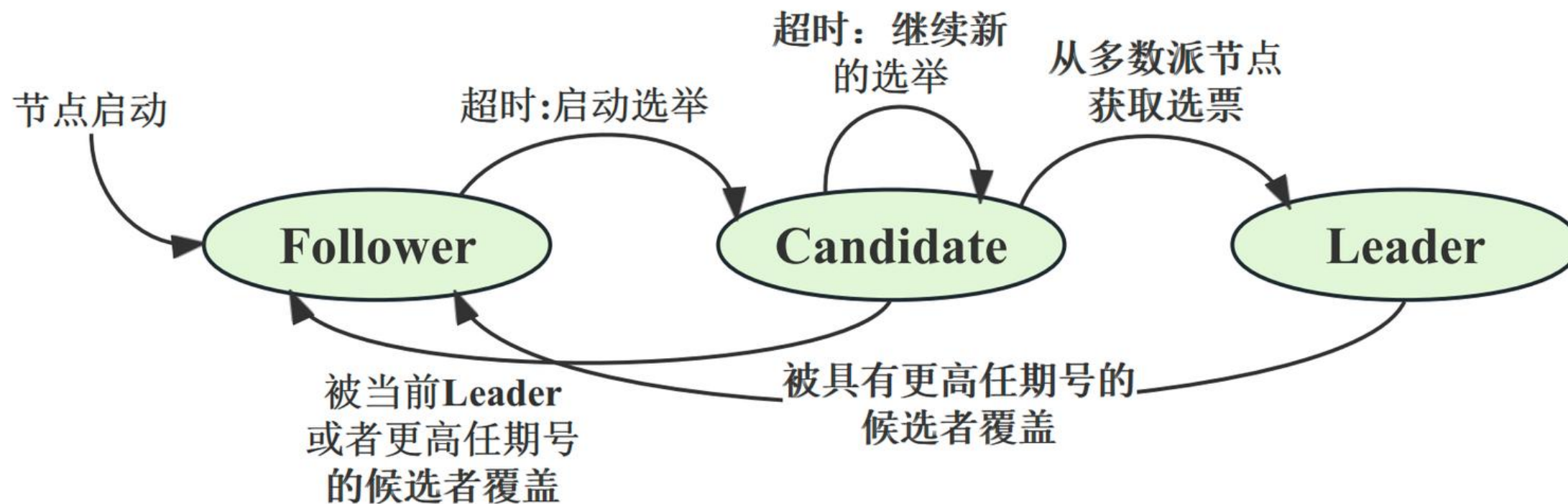
示例：从T3时刻开始，节点S2、S3、S4、S5都陆续收到了节点S1的**选举投票请求**：

- 节点S2、S3、S4原本任期号为1，低于节点S1的任期号2，因此会给节点S1投票并将任期号更新为2；
- 而节点S5因为拥有更高的任期号3而拒绝给节点S1投票；



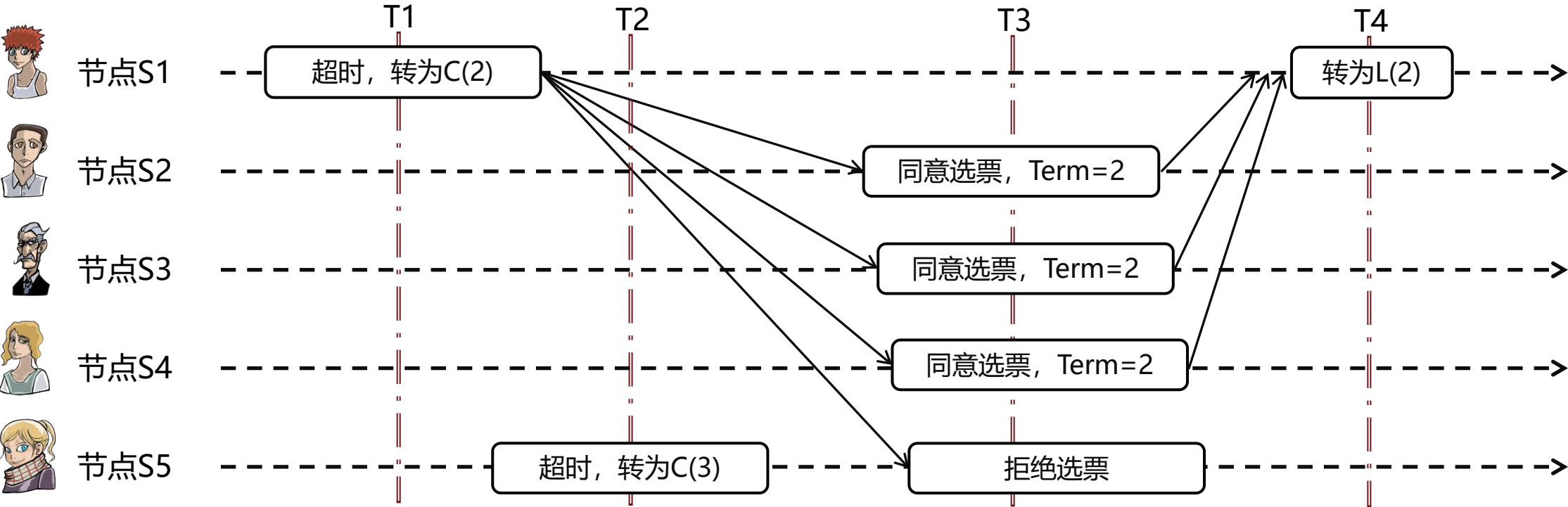
5.4.2 Leader选举

3. 若某一Candidate节点收到了超过半数的选票，该节点将成为Leader节点。



5.4.2 Leader选举

示例：节点S1在T3到T4时刻期间，陆续收到了来自S2、S3、S4的选票，并在T4时刻收到了超半数的选票，成为Leader，L(2)表示其Leader身份以及任期号Term为2；



5.4.2 Leader选举

分析： 通过比较任期号和日志信息，Raft确保了只有拥有更完整、更新日志的Candidate节点才能获得其他节点的支持。这是为了维护系统的一致性，避免过期的或缺乏完整日志的Candidate节点成为Leader节点。

Question1.1：如果Leader节点在之后过期，该怎么处理？

5.4.2 Leader选举

Question1.1：如果Leader节点在之后过期，该怎么处理？

分析：如果Leader过期，说明一定存在某一节点A的任期号大于当前的Leader节点B，因此要考虑怎么做才能让A覆盖Leader节点B成为新的Leader：

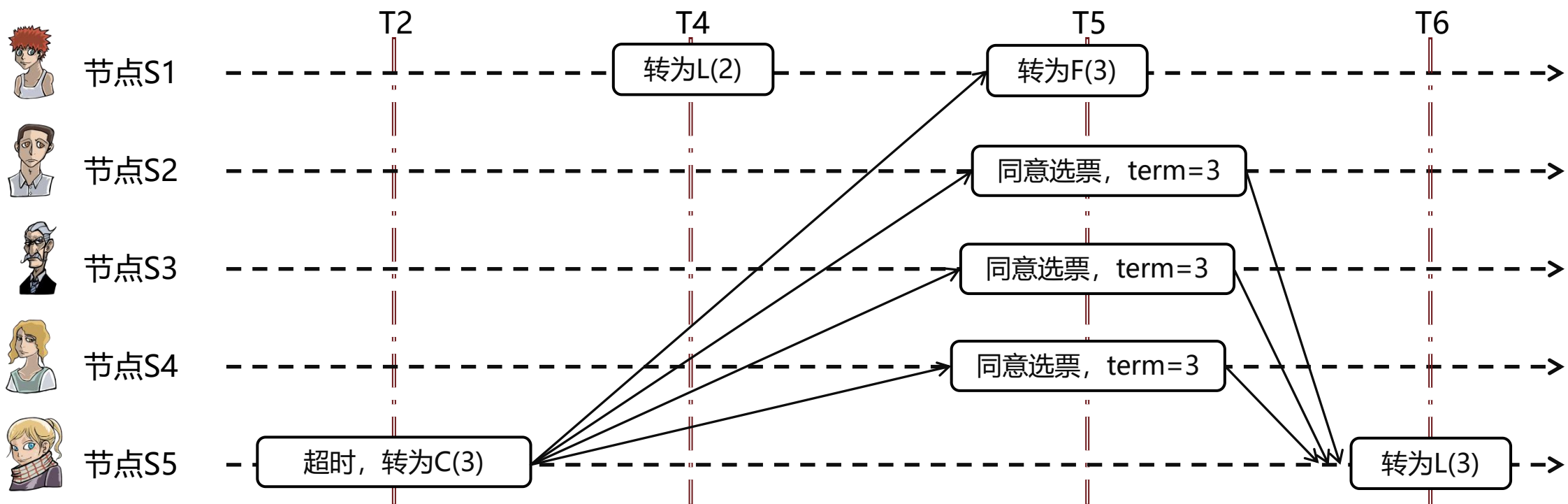
- A**要能够超时选举**，因此要防止计时器重置，所以高任期节点要拒绝来自Leader节点B的请求；
- B**要知道自己不应该再维持Leader**，所以在A向B发起选举请求或B收到A的拒绝响应后应该当转为Follower节点；

实际上即使B还没有收到选举请求或拒绝响应，只要A成为新的Leader节点说明多数派节点都和A同步了任期号，B自然地失去了Leader的职能

5.4.2 Leader选举

示例：在T2时刻节点S5也超时转为Candidate启动选举，并拥有比S1更高的任期3：

- 前面的图例中，首先节点S5会拒绝S1发来的请求；
- 节点S5会发起**选举投票请求**，在T5时刻覆盖节点S1，在T6时刻成为新的Leader；
(S1收到S5的请求后发现自己已经过期，F(3)表示Follower状态以及任期号Term为3)



5.4.2 Leader选举

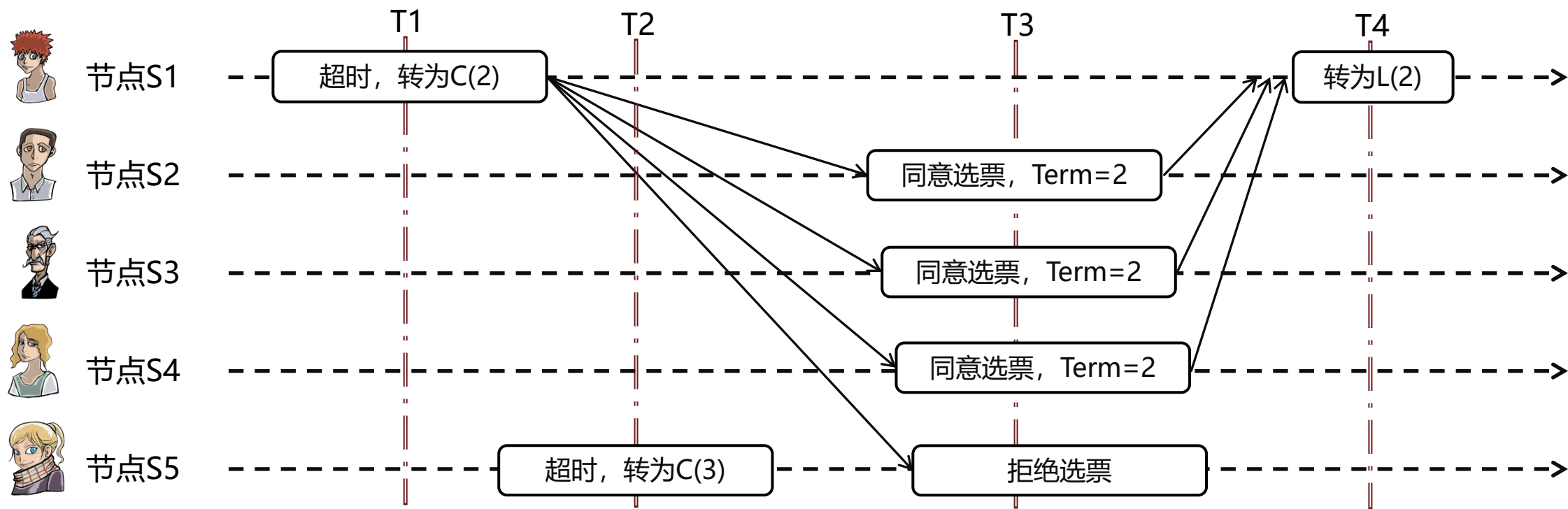
Question1.2: 如果节点满足最高的任期但是缺少日志条目, 请求会被如何响应?

分析: 因为要避免过期的或缺乏完整日志的Candidate节点成为Leader节点, Question1.1演示了选举过程如何保证选举出的Leader节点不是过期节点, 那么如果节点A满足最高的任期但是缺少日志条目时, 发出的请求会被如何响应?

- 如果高任期节点A能够覆盖当前Leader节点B, 那么由于节点A的日志不完整, 造成系统的不一致;
- 如果当前Leader节点B的请求被A一直拒绝, A一直无法同步日志;

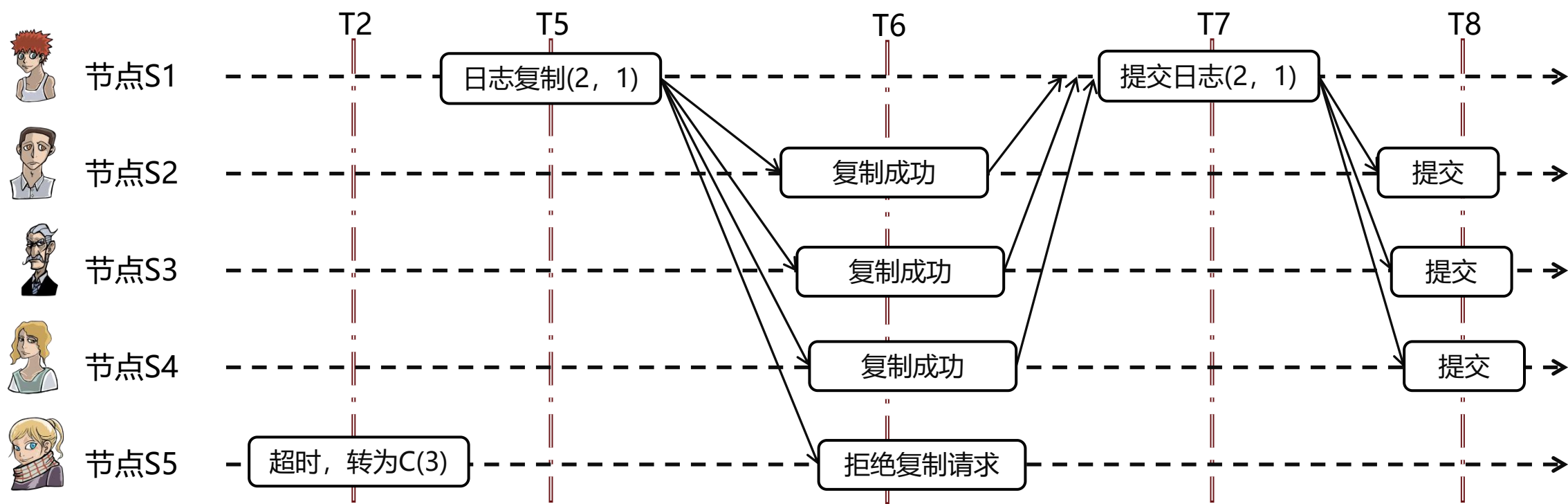
5.4.2 Leader选举

示例：节点S1在T1时刻超时成为Candidate，任期号为2，并在T4时刻收到了多数派节点的选票而成为Leader；



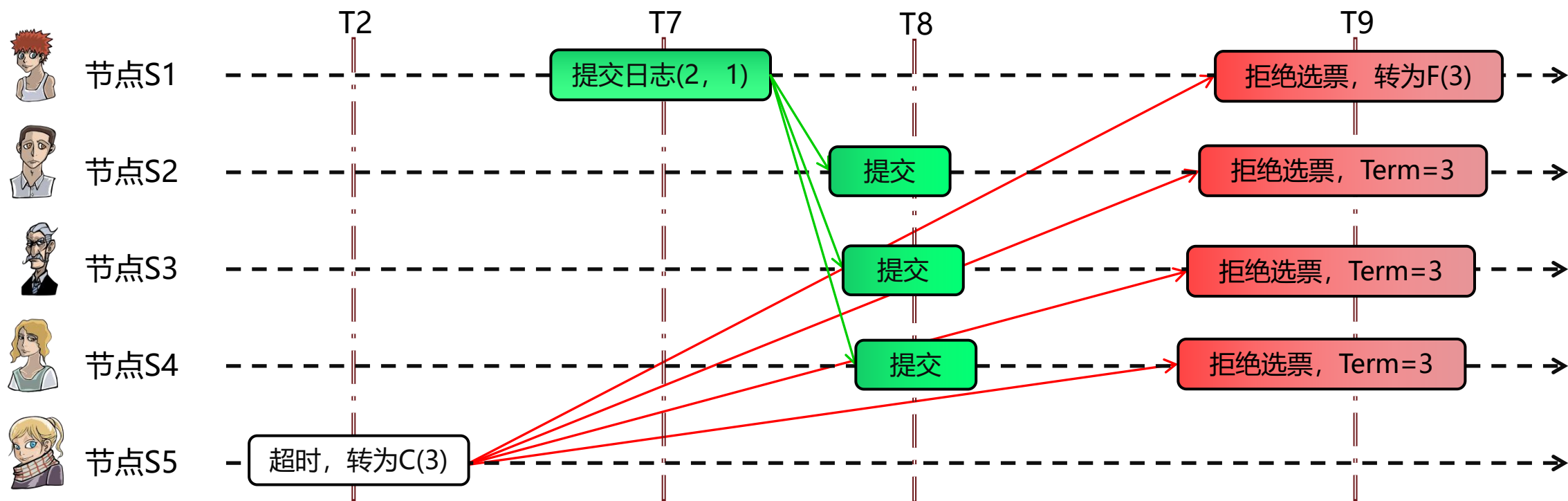
5.4.2 Leader选举

示例：节点S1成为Leader后，收到了客户端请求，将操作命令封装成日志同步到其他节点，(2, 1)表示该日志由任期号2的Leader复制，并且索引号为1；T6时刻S2、S3、S4陆续复制日志成功，而S5因为具有更高的任期号3而拒绝复制请求；T7时刻S1因为收到了多数派节点的复制成功响应，将提交日志(2, 1)；



5.4.2 Leader选举

示例：节点S1会通知其他节点提交日志，在T8时刻，S2、S3、S4都陆续提交了日志(2, 1)；而在T2时刻转为Candidate的节点S5发送的请求终于在T9时刻陆续发送到了其他节点，但其他节点都会因为S5不具备**最新最完整**的日志条目而拒绝选票，同时任期号更新为3，且节点S1失去Leader身份，转为Follower；最终所有节点都为Follower，等待它们中的一员超时再次发起新的选举。



5.4.2 Leader选举



什么叫具有**最新最完整**的日志条目？

分析：节点的日志条目是按时序追加的，日志的索引号代表着日志的先后顺序，而日志的任期号代表着同步该日志的Leader节点的新旧

对比两个节点最新追加的日志条目**LastLog**：

- 若节点A与节点B的LastLog具有**相同的任期号**，LastLog(A)具有**更大的索引号**，说明A节点的日志比B节点**更多更完整**；
- 若节点A与节点B的LastLog具有**相同的索引号**，LastLog(A)具有**更大的任期号**，说明A节点和B节点的日志同样多，但是A节点的日志是由更新的Leader同步的，因此具有**更新**的日志条目；

5.4.2 Leader选举

Question1.3: Leader节点如果发生故障怎么办?

心跳机制 (HeartBeat) : Raft算法的选举过程包含了基于心跳机制的故障检测以及故障转移:

Leader通过定期向所有节点发送**心跳信息**(一条不包含日志条目的RPC条目, 可以理解为只包含元数据的空白条目)维持其Leader状态, Follower在接收到心跳信息后会刷新超时时间。所以当Leader节点发生故障后, Follower会因为未收到心跳信息而超时成为Candidate发起新一轮选举。

5.4 Raft算法

01

5.4.1 算法概述

02

5.4.2 Leader选举

03

5.4.3 日志同步

04

5.4.4 安全性



5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

日志附加请求AppendEntries RPC

Leader节点把来自客户端的请求中用于执行操作的指令作为日志条目加入到它的日志中, 然后向其他节点发起**日志附加请求 AppendEntries RPC** (即请求将Leader节点的日志复制到其他节点)

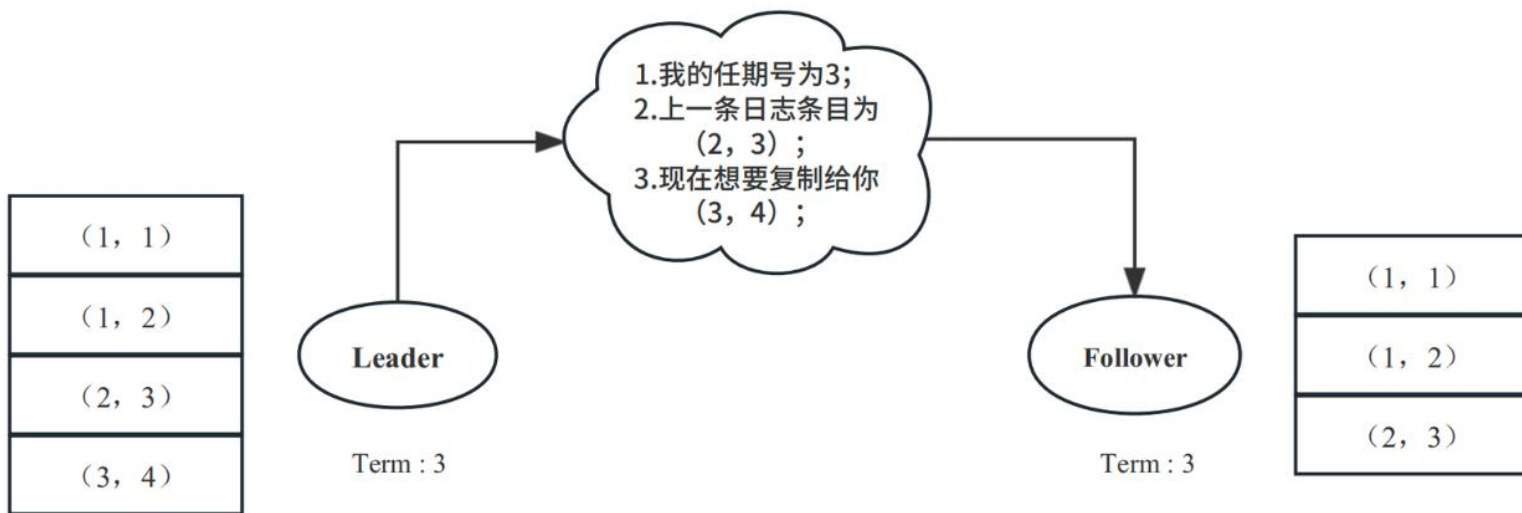
1. **term**: Leader任期号
2. **leaderId**: Leader节点id, 可用于Follower响应请求时定向到Leader节点
3. **preLogIndex**: Leader节点**上一条日志条目的索引**
4. **preLogTerm**: Leader节点**上一条日志条目的任期号**, 如日志条目的任期号为2表示该日志条目是被某一任期号为2的Leader节点同步的
5. **leaderCommit**: Leader节点**提交的日志条目的索引**, 表示在此索引位置前的日志条目都是已被提交的, 用于算法的安全性保证

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

1. Leader周期性地通过心跳信息发送日志附加请求给所有Follower

示例： Leader发送了一个日志附加请求，其任期号为3，希望将任期号为3，索引为4的日志条目(3, 4)同步给Follower，且上一条日志条目的任期号为2，索引为3



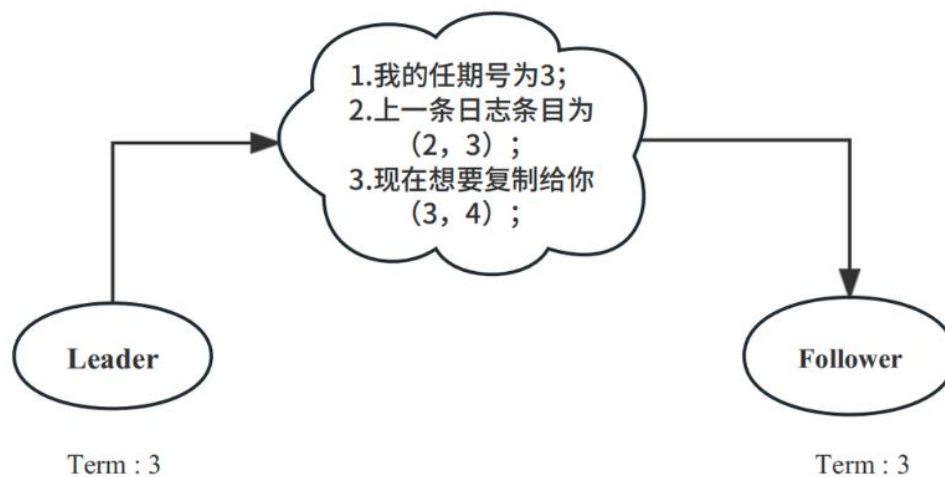
5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

2. Follower接收并处理日志附加请求:

- a. 检查日志附加请求中的任期号是否大于等于自己的当前任期号, 如果小于说明**当前Leader已经过期**, 则拒绝请求;
- b. 检查日志附加请求中的上一条日志条目是否与自己的日志匹配, 如果不匹配说明Follower上存在过期的日志条目, **需要重新同步之前的日志**, 则拒绝请求;
- c. 否则, 说明满足日志附加的条件, 会将日志附加请求中的新日志条目追加到Follower自己的日志列表中。

(1, 1)
(1, 2)
(2, 3)
(3, 4)



(1, 1)
(1, 2)
(2, 3)

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

2. Follower接收并处理日志附加请求:

- 检查日志附加请求中的任期号是否大于等于自己的当前任期号, 如果小于说明**当前Leader已经过期**, 则拒绝请求;
- 检查日志附加请求中的上一条日志条目是否与自己的日志匹配, 如果不匹配说明Follower上存在过期的日志条目, **需要重新同步之前的日志**, 则拒绝请求;
- 否则, 说明满足日志附加的条件, 会将日志附加请求中的新日志条目追加到Follower自己的日志列表中;

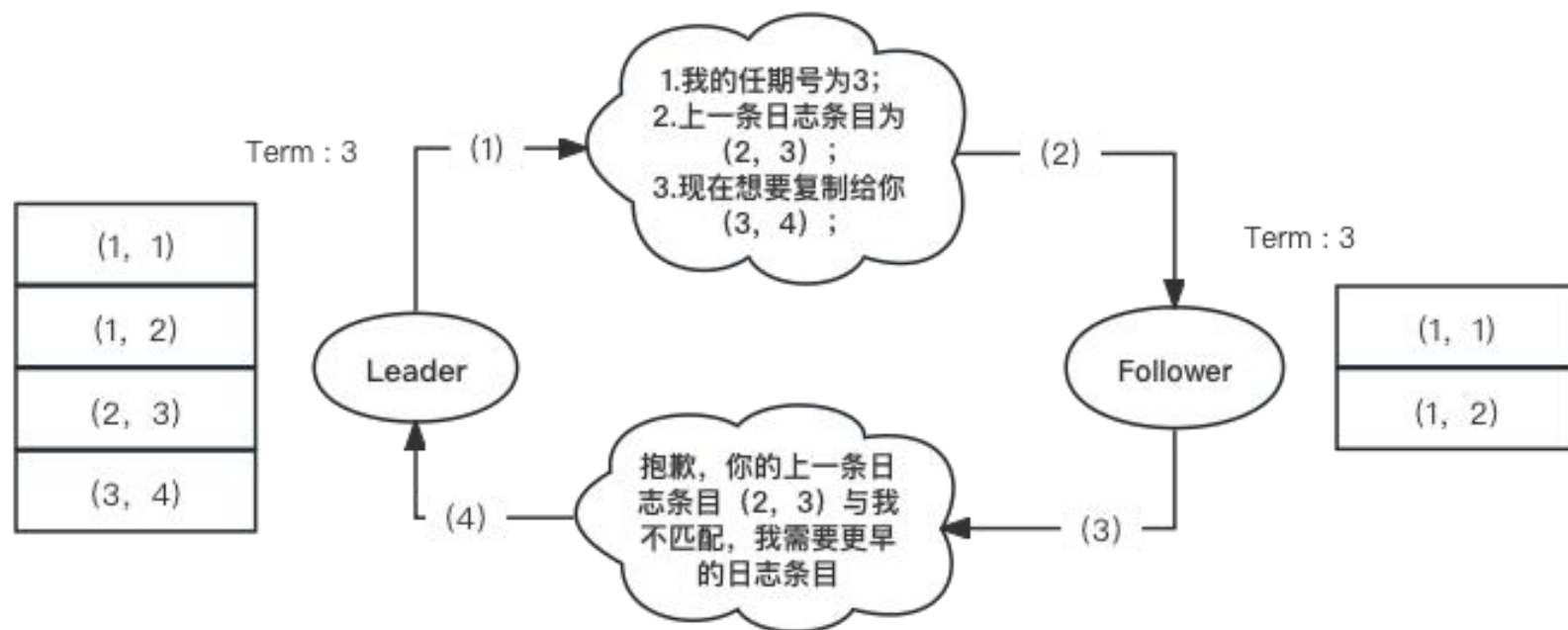
根据以上规则, Follower返回**响应日志附加请求**:
响应中包含:

- 当前Follower节点的任期号
- 处理结果 (成功或失败)
- 已同步的日志条目索引信息

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

示例： Follower节点接收到请求后，经检查发现日志条目（2，3）无法与自己存储的最新日志条目（1，2）匹配，说明在这之前还有尚未同步的日志，不可以直接附加日志条目（3，4），因此Follower给Leader节点返回响应告知Leader附加失败。

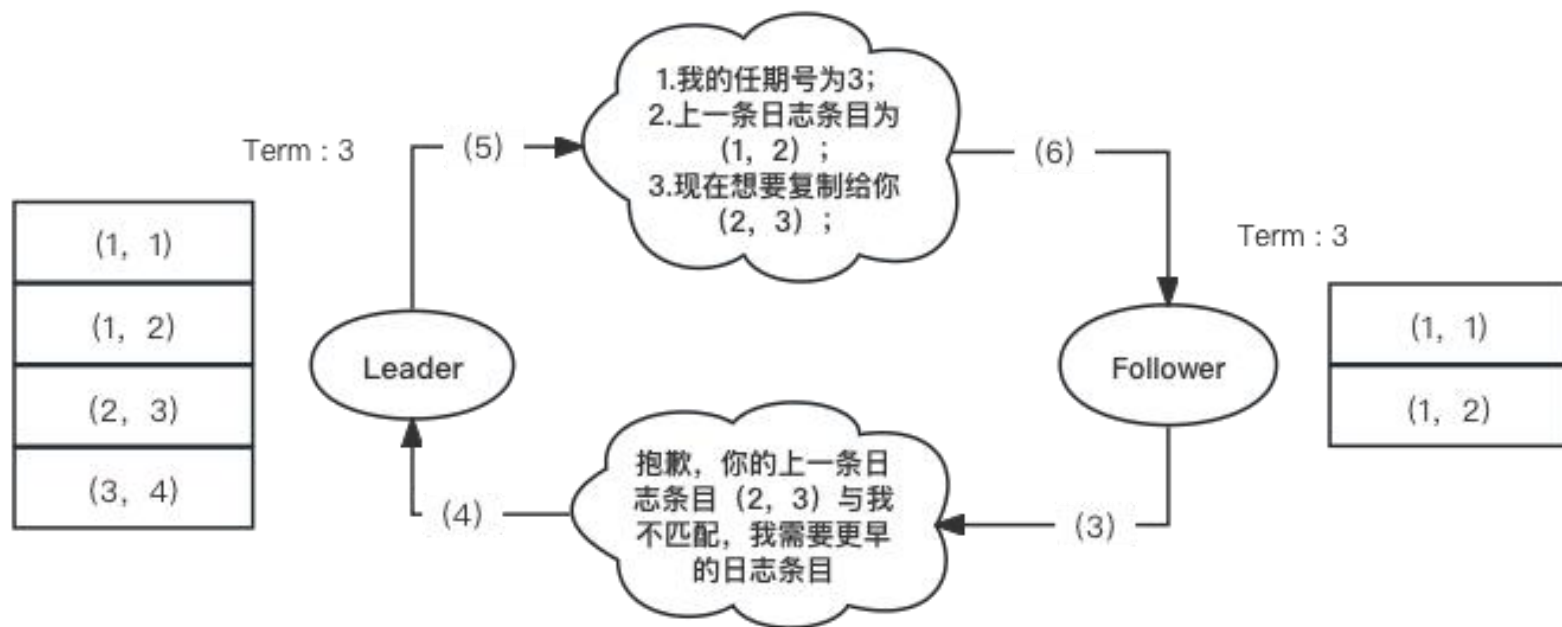


5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

3. Leader处理响应:

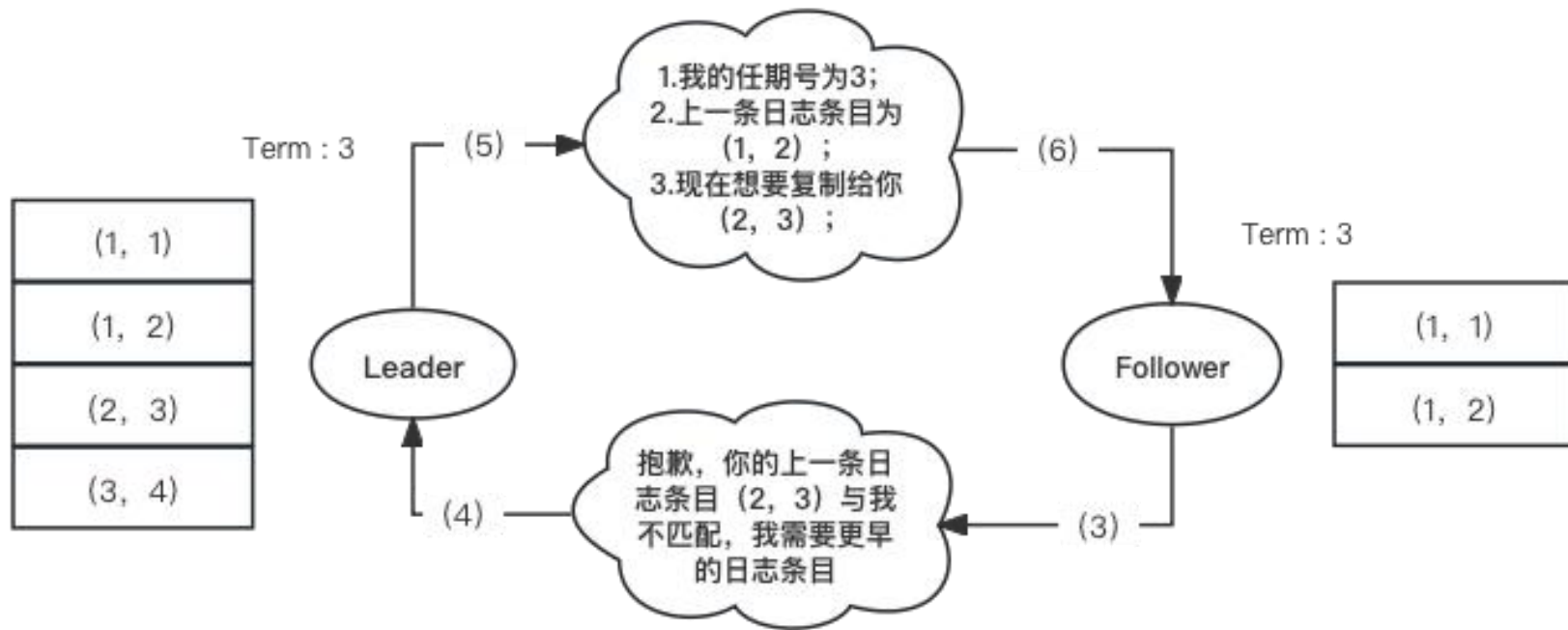
- 如果响应的任期号大于Leader的任期号, Leader将自己**转为Follower**状态;
- 如果响应中的处理结果为**失败**, Leader会**递减日志索引**, 尝试将日志复制到Follower直到成功为止;
- 如果响应中的处理结果为**成功**, Leader更新Follower的已复制日志索引, 并继续发送下一个日志附加请求; 并且当超过半数的日志被成功复制后, Leader会更新已提交日志的索引



5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

示例： Leader 节点收到 Follower 的失败响应后将索引从4递减到3复制前一个日志 (2, 3) 给 Follower 节点；因为 (2, 3) 的上一条日志条目 (1, 2) 与 Follower 匹配，则可以将日志条目 (2, 3) 追加到 Follower 中。

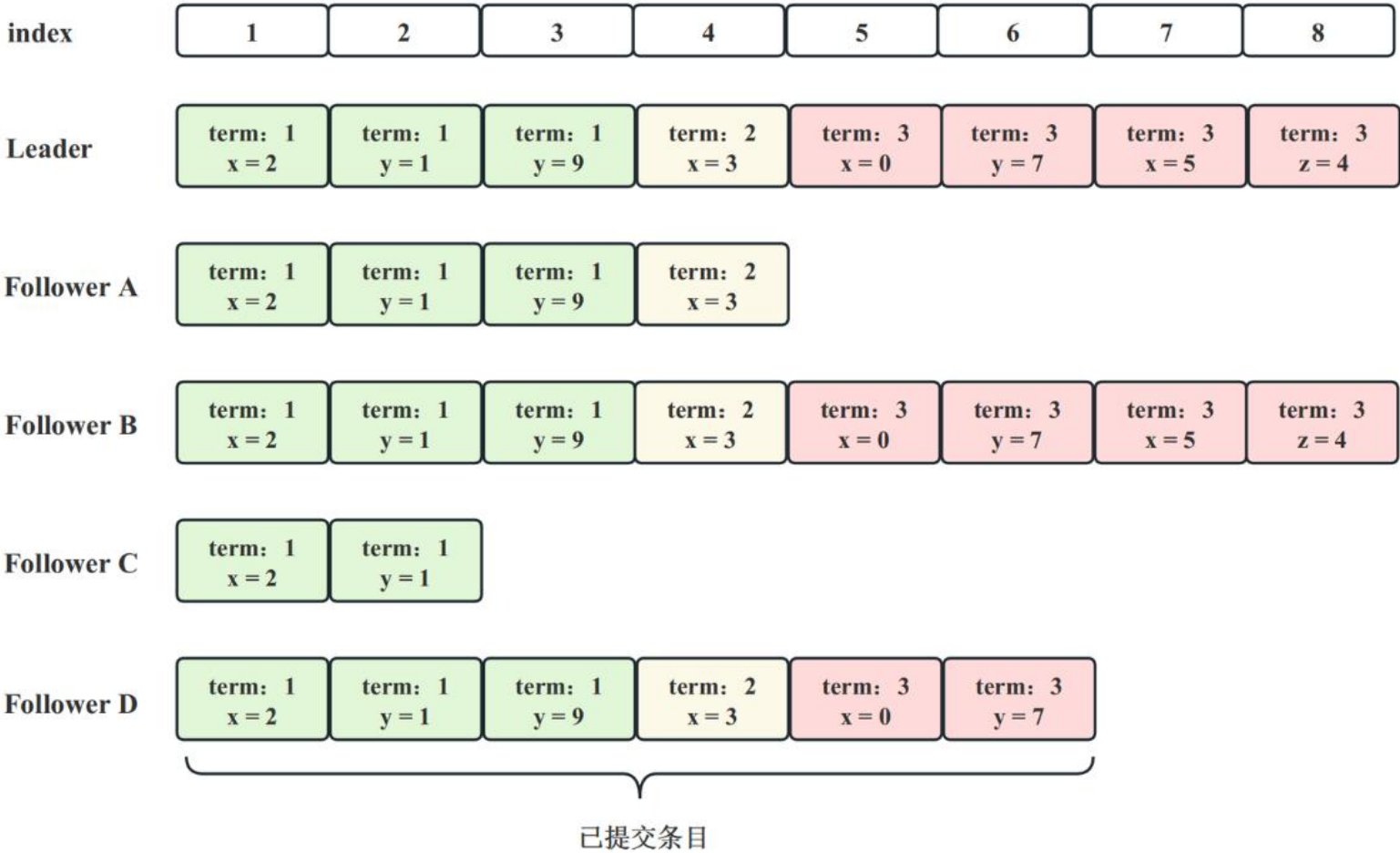


5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

示例：索引为6之前的日志条目已经被成功同步到了FollowerB以及FollowerD，满足多数派副本的条件，因此Leader可以更新已提交索引号为6。

——已提交的日志可以保证最终一定会被状态机应用，即日志存储的命令最后一定会被按序执行



5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?



为什么 AppendEntries RPC 需要 preLogIndex 和 preLogTerm 两个参数来匹配 Follower 中的最新日志条目?

↓

Leader 依据 preLogIndex 和 preLogTerm 判定是否可以进行日志同步, 那么 preLogIndex 和 preLogTerm 匹配成功能否说明节点的日志已经具有了相同的内容和顺序?

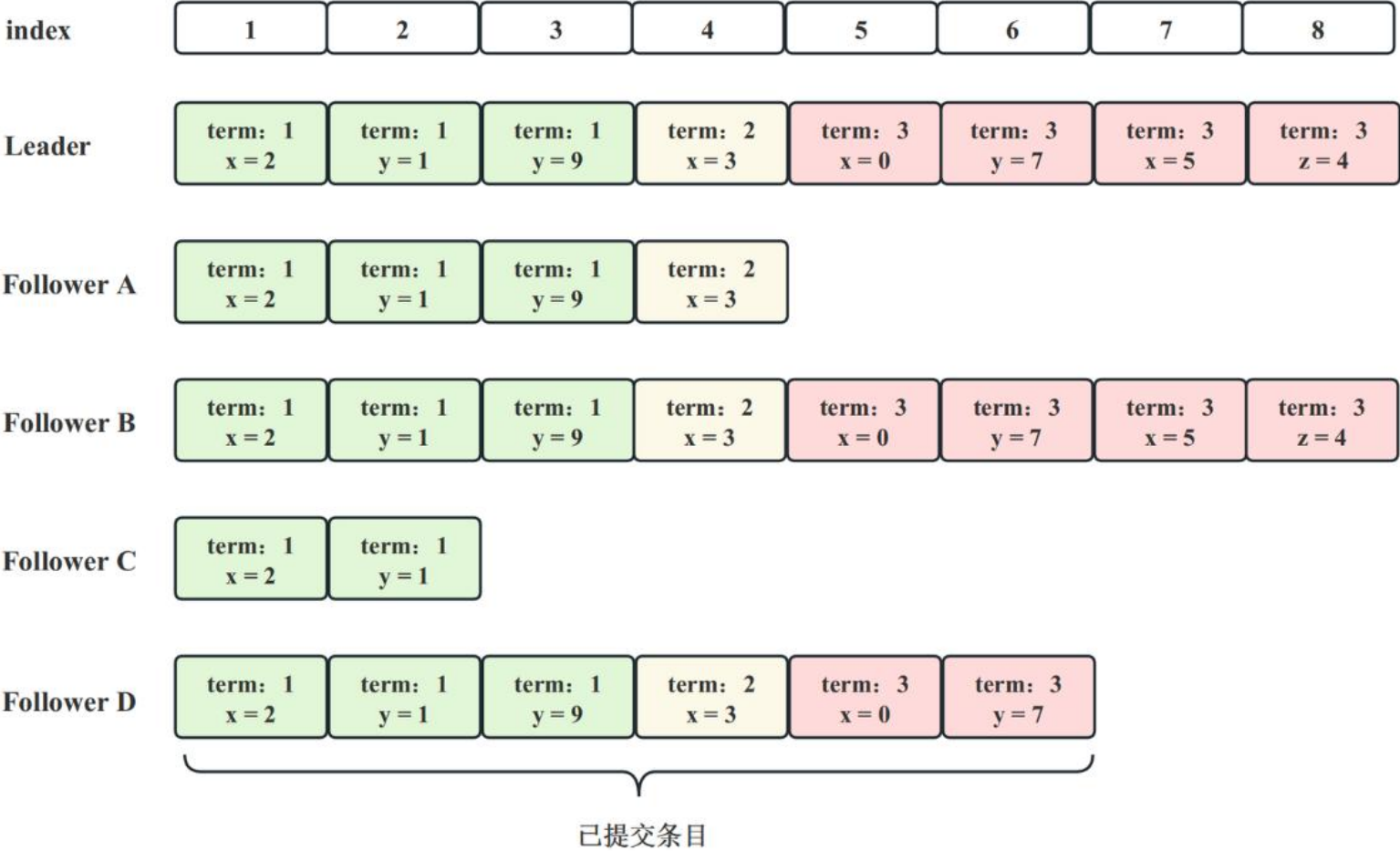
Raft 的日志同步机制可以保证如下两点:

- ① 如果不同日志中的两个条目有相同的索引和任期号, 则它们所存储的命令是相同的
- ② 如果不同日志中的两个条目有相同的索引和任期号, 则它们之前的所有条目存储的命令也都是完全相同的

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

证明 ① : Follower A 与 Follower B 的日志条目 (2, 4) 具有相同的任期号 2, 因为在同一任期内, Leader 节点是唯一的, 因此日志条目 (2, 4) 一定是由同一个 Leader 同步的。又因为两个日志条目 (2, 4) 的索引都为 4, 说明该日志条目都对应 Leader 节点的日志列表中索引为 4 的日志条目

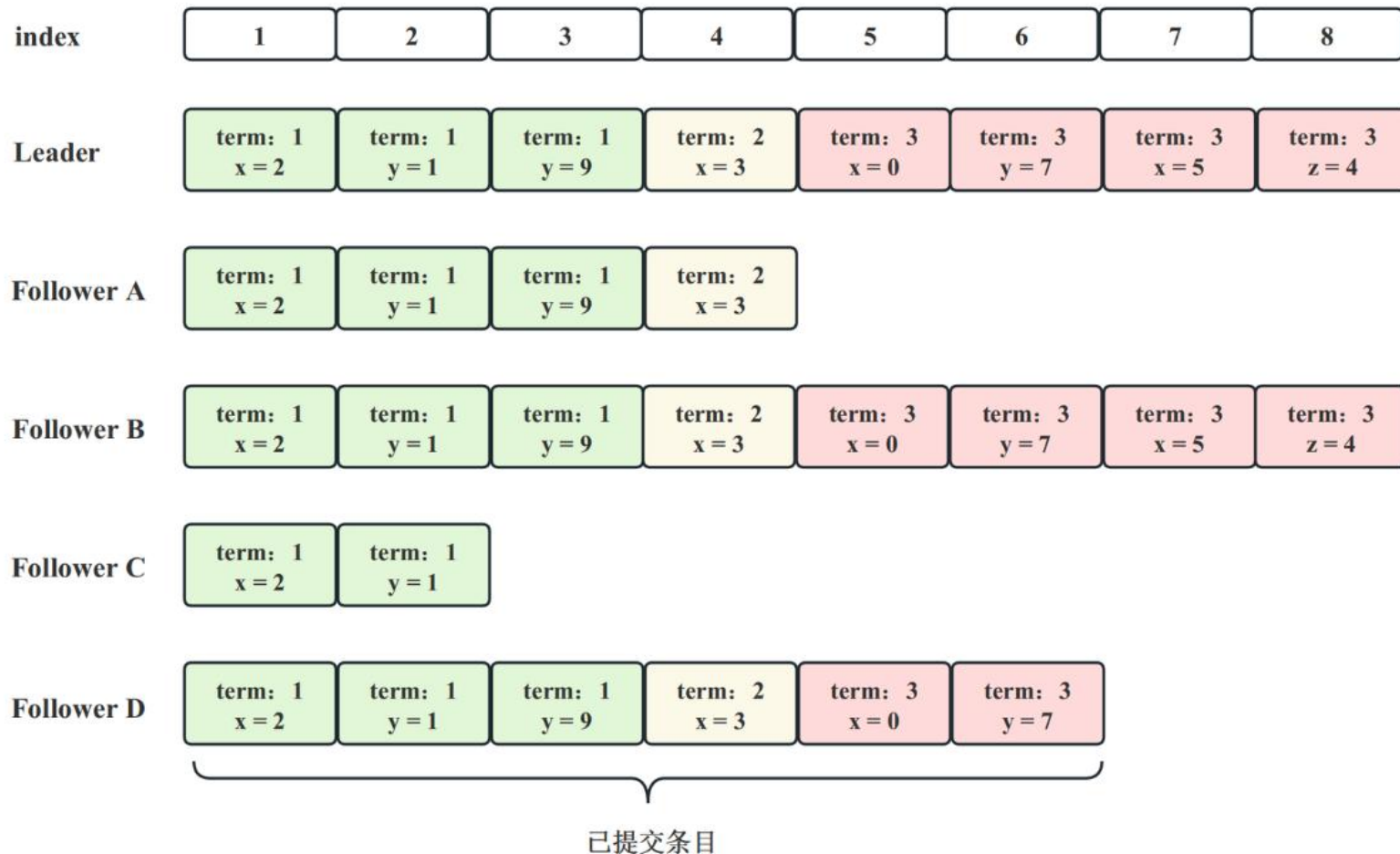


5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

证明 ② : Follower B 与 Follower D 的日志列表中都有日志条目 (3, 6) 说明 A 和 B 都被某一任期为 3 的 Leader 节点同步日志到了索引 6 的位置。

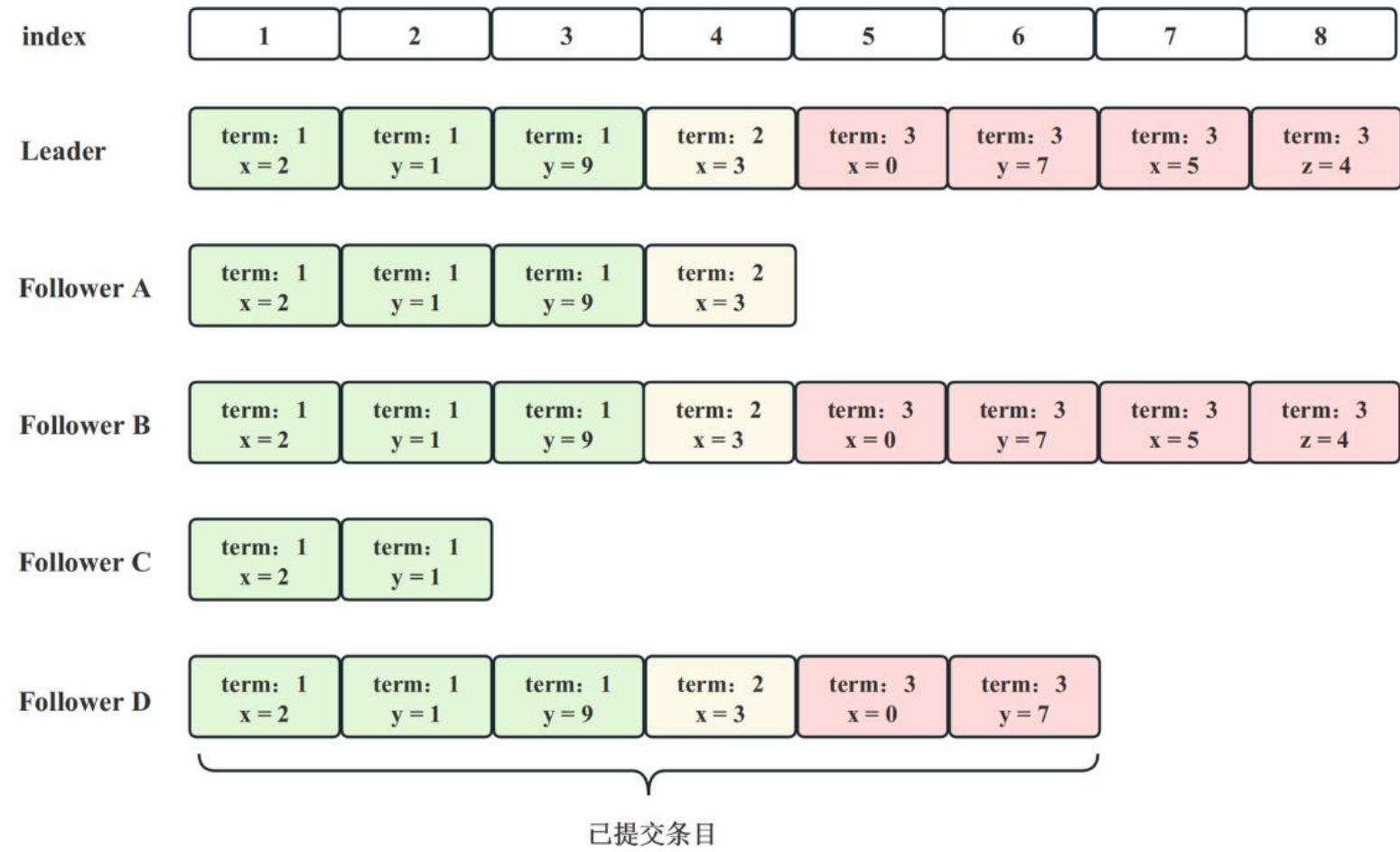
由于每次 Leader 发出日志附加请求都会对比上一条日志是否与 Follower 匹配, 借助递归思想, 索引 6 之前的日志条目都应该与任期为 3 的那个 Leader 节点保持一致



5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

反证法 ②：假设有 Follower B的日志列表在索引6前有日志条目 (2, 5)，那么Leader在同步 (3, 6) 时一定会因为 (2, 5) 与 Leader上一条日志条目 (2, 6) 不匹配而同步失败，与 Follower B具有日志条目 (3, 6) 矛盾。



5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?



一般情况下，Leader和Follower的日志保持一致，因此附加日志的一致性检查通常不会失败。然而Leader崩溃可能会导致日志不一致，即旧的Leader可能没有完全同步完日志中的所有条目

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

示例:

- 1.Follower节点A-B缺少一定日志条目:
 - 如Follower A缺少Leader中的日志条目 (6, 10) ;

index	1	2	3	4	5	6	7	8	9	10	11	12
Leader (Term=8)	1	1	1	4	4	5	5	6	6	6		
Follower A	1	1	1	4	4	5	5	6	6			
Follower B	1	1	1	4	4							
Follower C	1	1	1	4	4	5	5	6	6	6	6	
Follower D	1	1	1	4	4	5	5	6	6	6	7	7
Follower E	1	1	1	4	4	4	4					
Follower F	1	1	1	2	2	2	3	3	3	3	3	3

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

示例:

2.Follower C-D 存在
额外的未提交条目:

- 以Follower D为例,
D在某一时刻超时选
举 成 为 了 新 的
Leader, 任 期 为 7
但尚未同步日志 (7,
11)、(7, 12)
就崩溃了;

index	1	2	3	4	5	6	7	8	9	10	11	12
Leader (Term=8)	1	1	1	4	4	5	5	6	6	6		
Follower A	1	1	1	4	4	5	5	6	6			
Follower B	1	1	1	4	4							
Follower C	1	1	1	4	4	5	5	6	6	6	6	
Follower D	1	1	1	4	4	5	5	6	6	6	7	7
Follower E	1	1	1	4	4	4	4					
Follower F	1	1	1	2	2	2	3	3	3	3	3	3

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

示例:

3.Follower E-F既缺少条目又有额外的未提交条目:

- 以F为例, 节点F是任期2的Leader节点, 在其任期收到了客户端请求, 并附加日志条目到了日志列表索引4-6的位置中, 但还未同步其日志(2, 4)、(2, 5)、(2, 6)就发生故障崩溃, 恢复后再次成为任期3的Leader节点, 但再次发生故障崩溃, 最终既存在其作为旧Leader尚未同步的日志条目如(2, 4)、(3, 7)等又缺少当前新的Leader中的日志条目如(4, 4)、(5, 6)、(6, 8)等

index	1	2	3	4	5	6	7	8	9	10	11	12
Leader (Term=8)	1	1	1	4	4	5	5	6	6	6		
Follower A	1	1	1	4	4	5	5	6	6			
Follower B	1	1	1	4	4							
Follower C	1	1	1	4	4	5	5	6	6	6	6	
Follower D	1	1	1	4	4	5	5	6	6	6	7	7
Follower E	1	1	1	4	4	4	4					
Follower F	1	1	1	2	2	2	3	3	3	3	3	3

5.4.3 日志同步

Question2: Leader节点怎么保证各节点日志具有相同的顺序和内容?

示例：对于存在额外未提交条目的情况，Leader节点会先找到与Follower节点能匹配的日志条目如示例中F的日志条目（1，3），**删除**Follower F在该位置之后的全部日志条目，然后再同步日志。

index	1	2	3	4	5	6	7	8	9	10	11	12
Leader (Term=8)	1	1	1	4	4	5	5	6	6	6		
Follower A	1	1	1	4	4	5	5	6	6			
Follower B	1	1	1	4	4							
Follower C	1	1	1	4	4	5	5	6	6	6	6	
Follower D	1	1	1	4	4	5	5	6	6	6	7	7
Follower E	1	1	1	4	4	4	4					
Follower F	1	1	1	2	2	2	3	3	3	3	3	3

5.4 Raft算法

01

5.4.1 算法概述

02

5.4.2 Leader选举

03

5.4.3 日志同步

04

5.4.4 安全性



5.4.4 安全性



安全性 (safety) : 强调系统在某一时刻的正确性, 即系统不会返回错误的结果

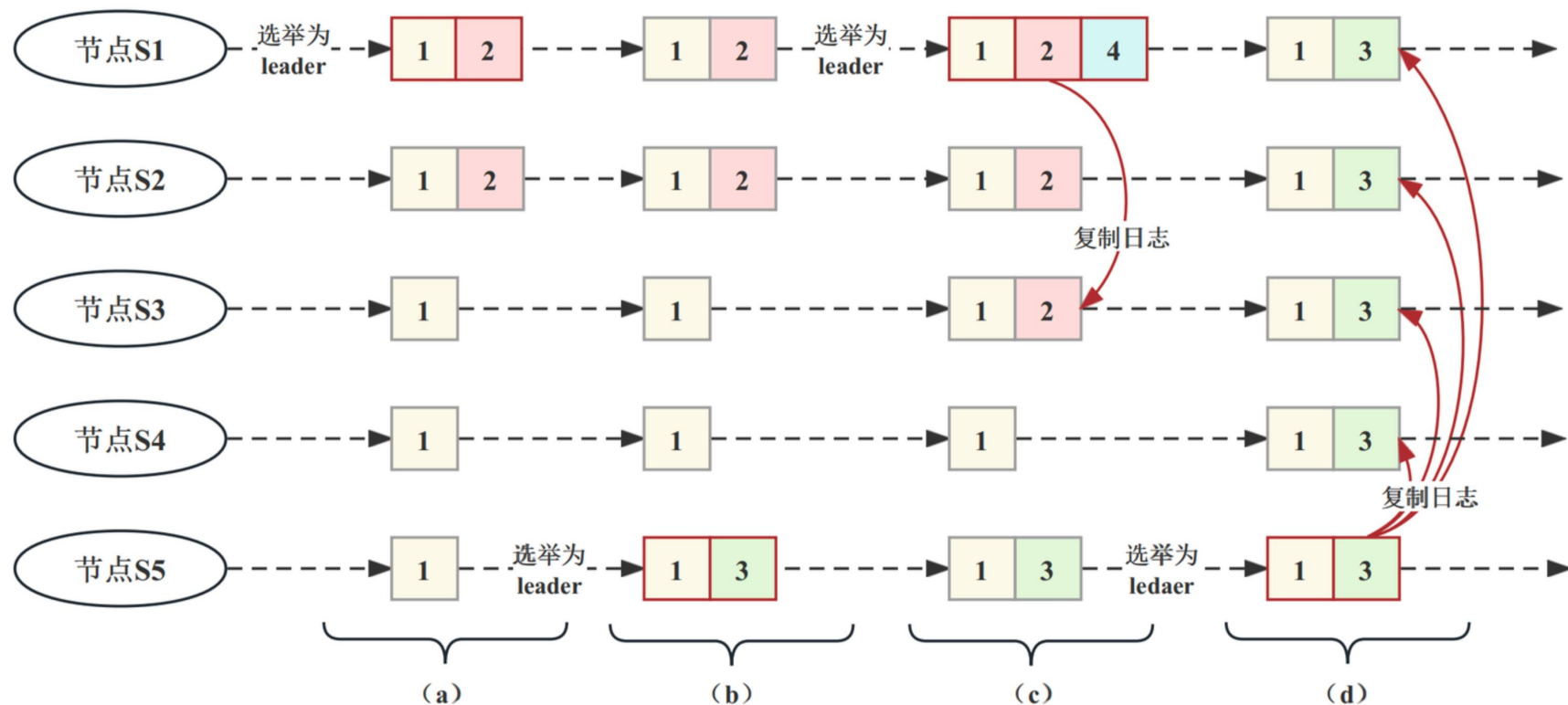
活性 (liveness) : 强调系统的响应能力, 即在一定时间内能够达到预期的结果

5.4.4 安全性

驱动案例

示例：

- a) S1在任期2中选举为Leader部分的节点复制了index为2的日志条目。
- b) S1崩溃，而节点S5在任期3中赢得了选举，然后从客户端接收了一条新的日志条目存储在索引为2的条目中

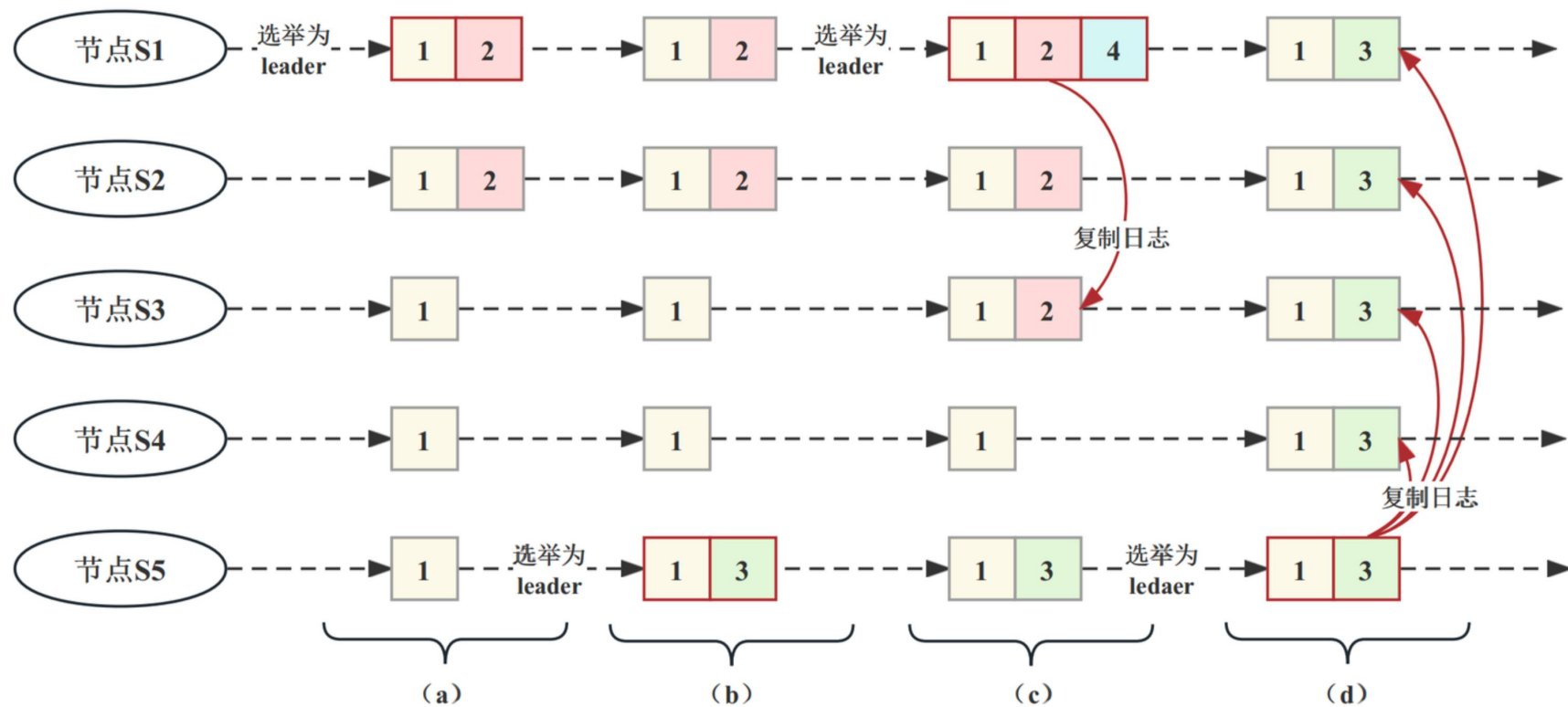


5.4.4 安全性

驱动案例

示例：

c) S5又崩溃了，S1重新启动，由于任期为4再次选举成功，开始复制日志。在这时，来自任期2的那条日志已经被复制到了集群中的大多数节点上，如果没有增加日志提交的限制，该日志条目已经可以被提交了。

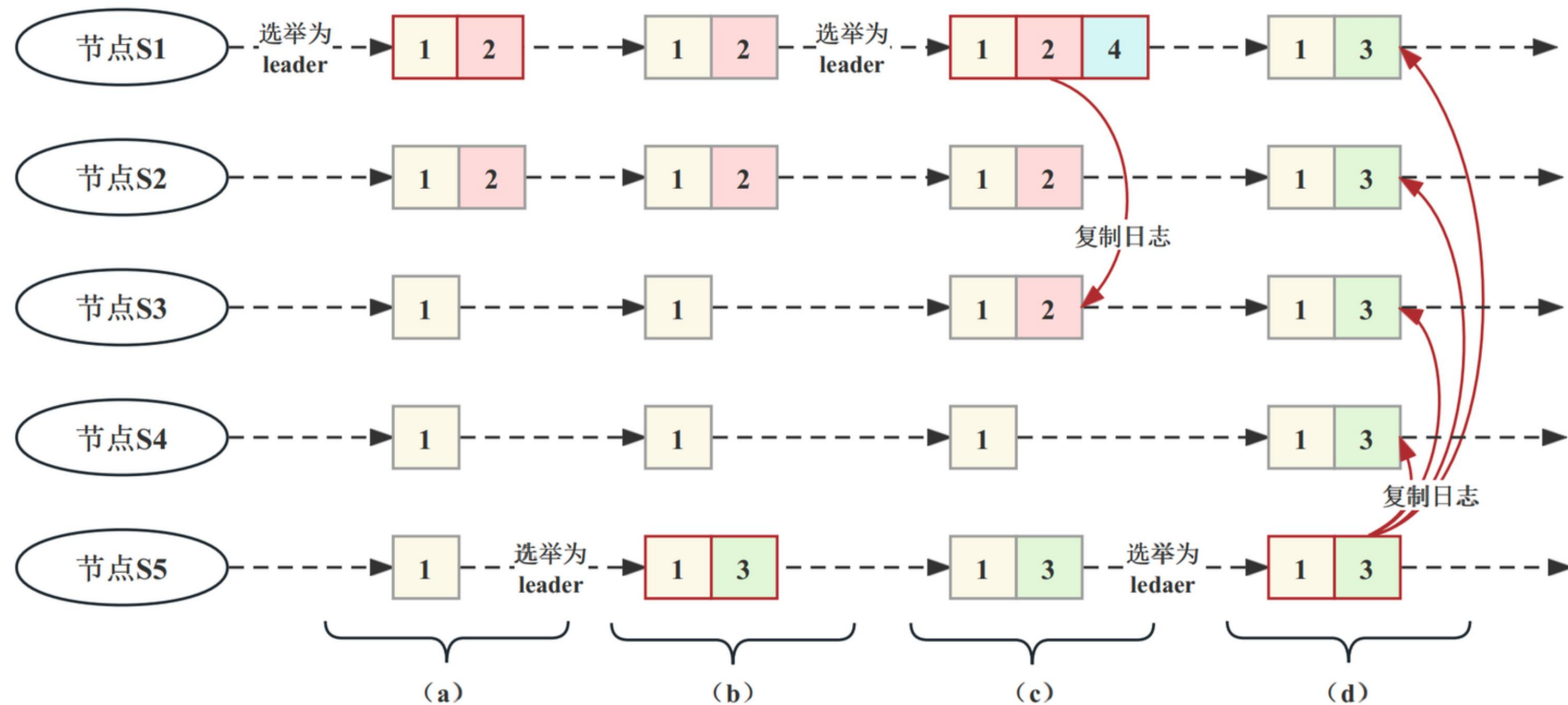


5.4.4 安全性

驱动案例

示例：

d) S1 又崩溃了，S5重新启动且由于任期为5再次选举成功，然后覆盖了他们在index为2处的日志，从这里可以看出，对于节点S1-S3，已提交的日志条目2代表了一个状态或数据的更改，由于节点S4和S5丢失了该日志条目，导致了S1-S3与S4-S5的状态不一致。



5.4.4 安全性

驱动案例

分析：

选举和日志复制的机制还不能充分地保证每一个状态机都按照相同的顺序执行相同的指令。例如一个Follower可能处于不可用状态，同时Leader已经提交了若干的日志条目，当这个 Follower 恢复（尚未与Leader达成一致）时Leader发生故障且该Follower被选举为Leader，便会覆盖曾提交的日志条目，导致了错误的结果。

Raft增加两点限制以保证安全性：

1. 选举限制：拥有最新的已提交日志条目的Follower才有资格成为Leader
2. 提交当前任期内的日志条目：Raft 算法要求节点只能提交来自当前任期的日志条目

5.4.4 安全性

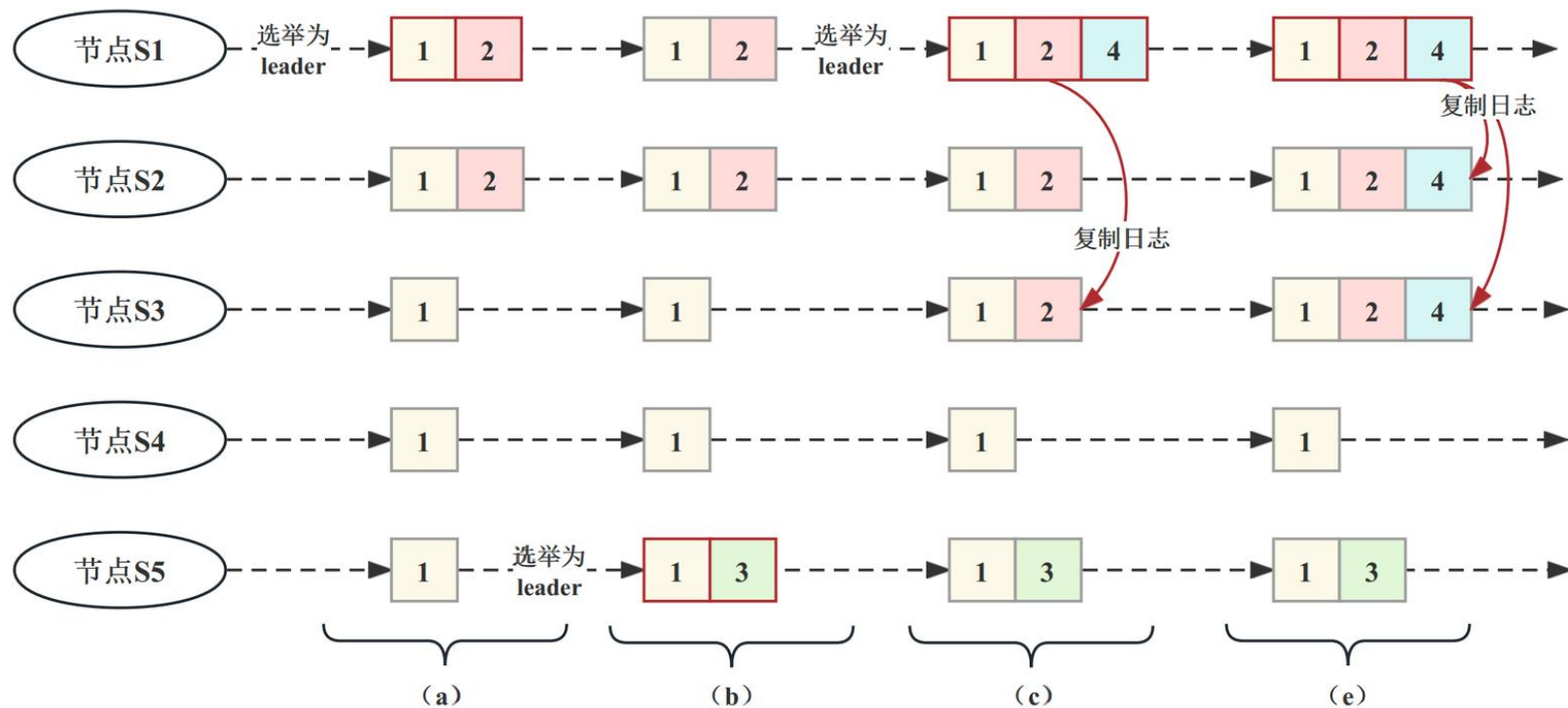
增加安全性限制

示例：

c) 日志条目2因为不是最新的日志条目而无法提交

d) 不存在S5节点的日志复制覆盖了已提交日志的情况

e) 节点S1崩溃之前，把自己新任期接收的日志条目4复制到了多数派节点上，因此日志条目4就可以被提交，并且日志条目4的任期高与日志条目3，此时的S5因为无法获得多数派节点的选票而选举失败





<https://thesecretlivesofdata.com/raft/>



第五章、容错与共识

01

5.1 故障类型

02

5.2 故障检测与恢复

03

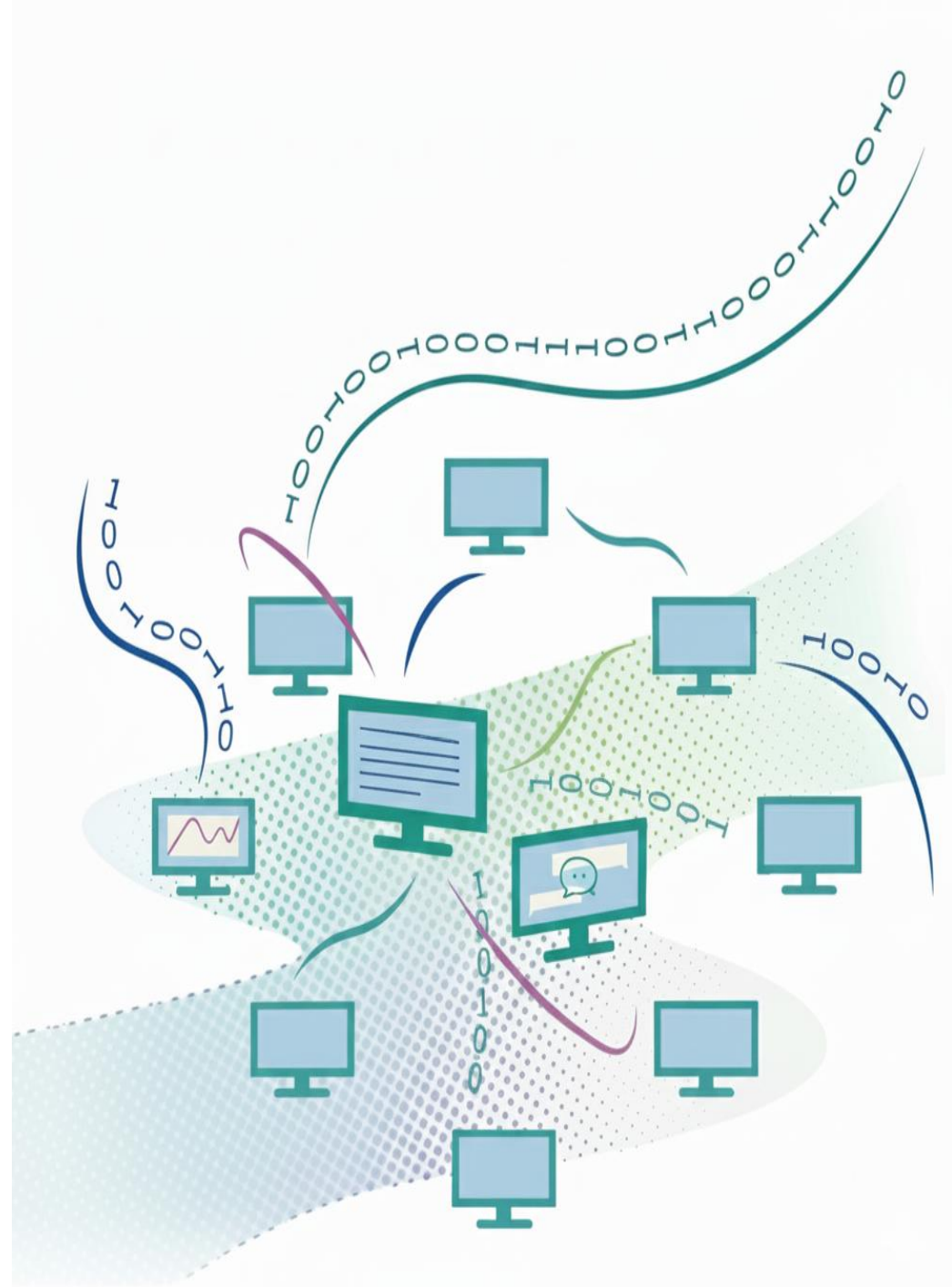
5.3 Paxos算法

04

5.4 Raft算法

05

5.5 PBFT算法*



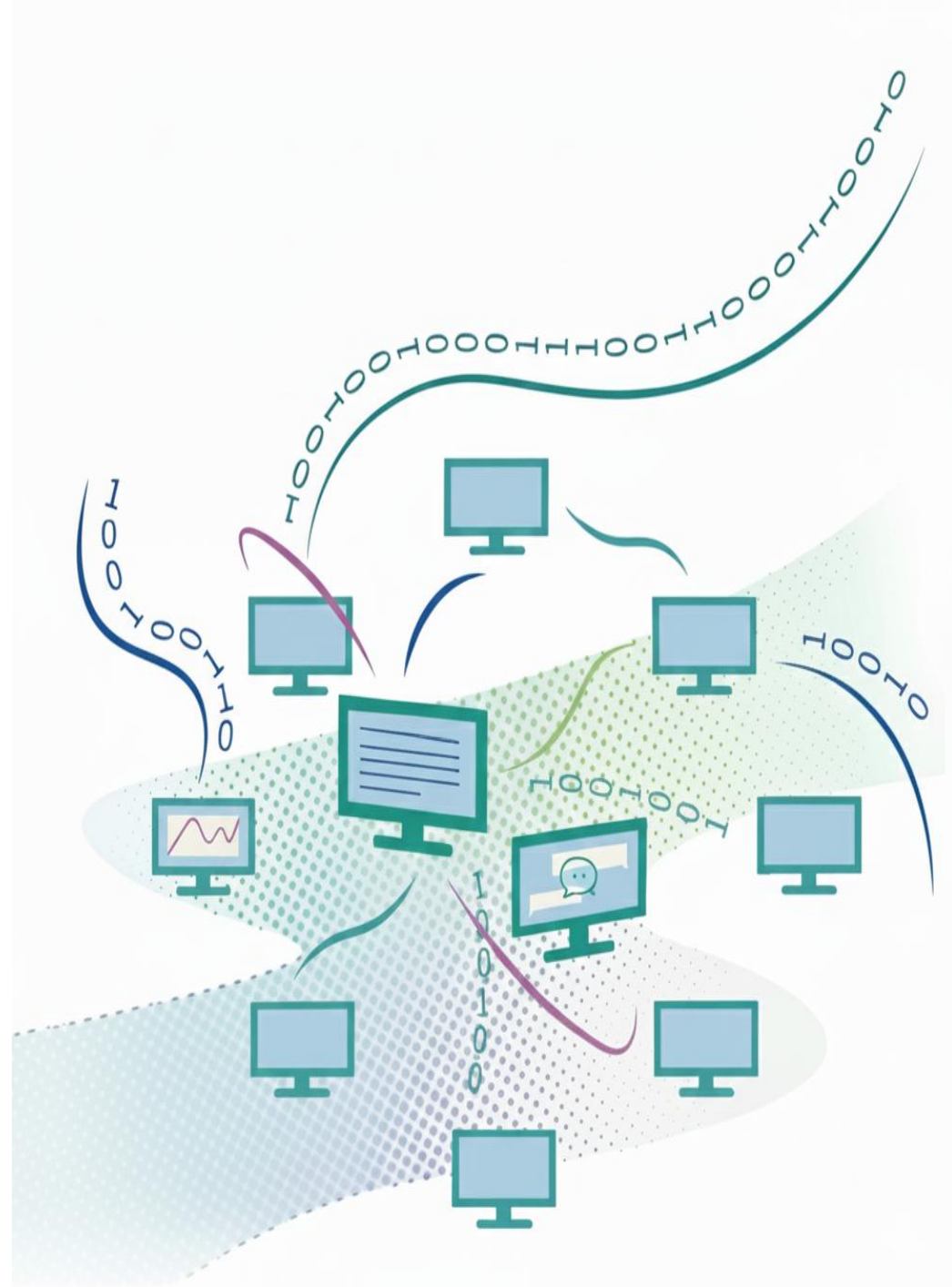
5.4 PBFT算法*

01

5.5.1 问题描述

02

5.5.2 算法流程



5.5.1 问题描述

拜占庭故障

- Paxos算法 ✕
- Raft算法 ✕

⇒ PBFT算法 ✓

示例：

以Raft为例，Leader通过发送日志复制消息让其他的Followers同步日志，但如果Leader本身是恶意节点呢？他可能会让不同的Follower复制不同的日志，而Follower也不会判断Leader消息的正确性，这时共识的安全性就无法保证。

5.5.1 问题描述

分析：

如果所有的节点都是恶意节点，那么任何算法都无法保证安全性，这种情况本身就是无解的。但如果能够保证节点在做出判断或决策的时候**真消息一定比假消息多**，且在诚实的节点中**真消息的数量可以形成多数派**，那么就可以利用人数优势判断消息是否为真，并且可以认为真消息达成了多数派认同。

- 节点总数： n
 - 错误节点数（故障节点或者是恶意节点）： f
1. 如果 f 个错误节点全是故障节点，相当于无拜占庭错误，此时正常的节点只要比 f 多一个即可达成多数共识（即 $f+1$ 个），此时需要 $n-f > f$ ，即 $n > 2f$ 。
 2. 如果 f 个错误节点都是恶意节点，在收到 $n-f$ 条响应消息后进行决策时，这 $n-f$ 条消息中会掺杂 f 条假消息。要保证真消息要比假消息多一个才能保证真消息占据多数，也就是说 $n-f-f > f$ ，即 $n > 3f$ ，所以 PBFT 支持的最大容错数量为 $f = (n-1)/3$

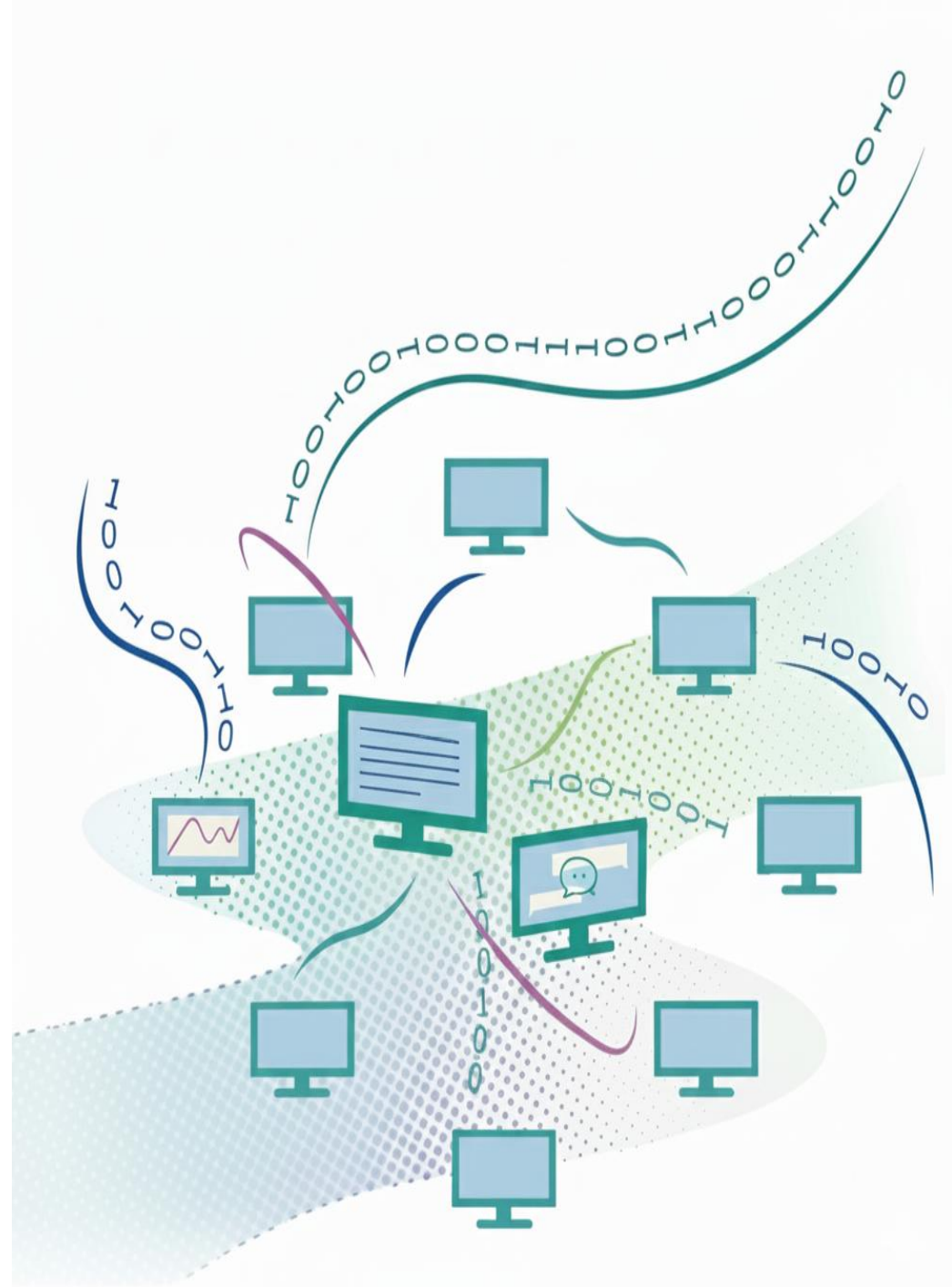
5.4 PBFT算法*

01

5.5.1 问题描述

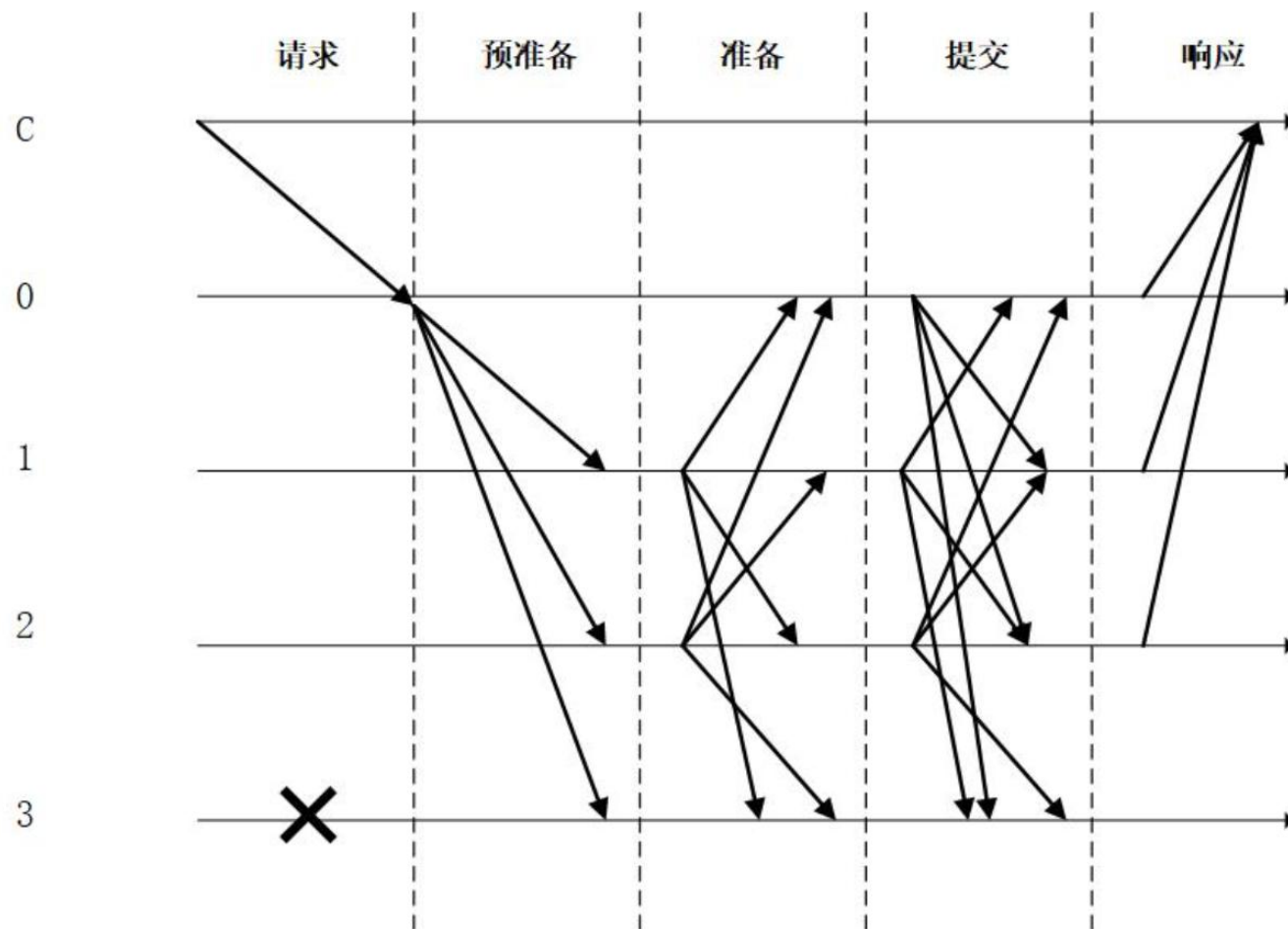
02

5.5.2 算法流程



5.5.2 算法流程

—— PBFT的准备阶段和提交阶段是“全民参与的”，因为领导者或者提议者本身可能就是恶意节点，所以不能只让某个节点“发号施令”



5.5.2 算法流程

客户端请求<REQUEST,o,t,c>

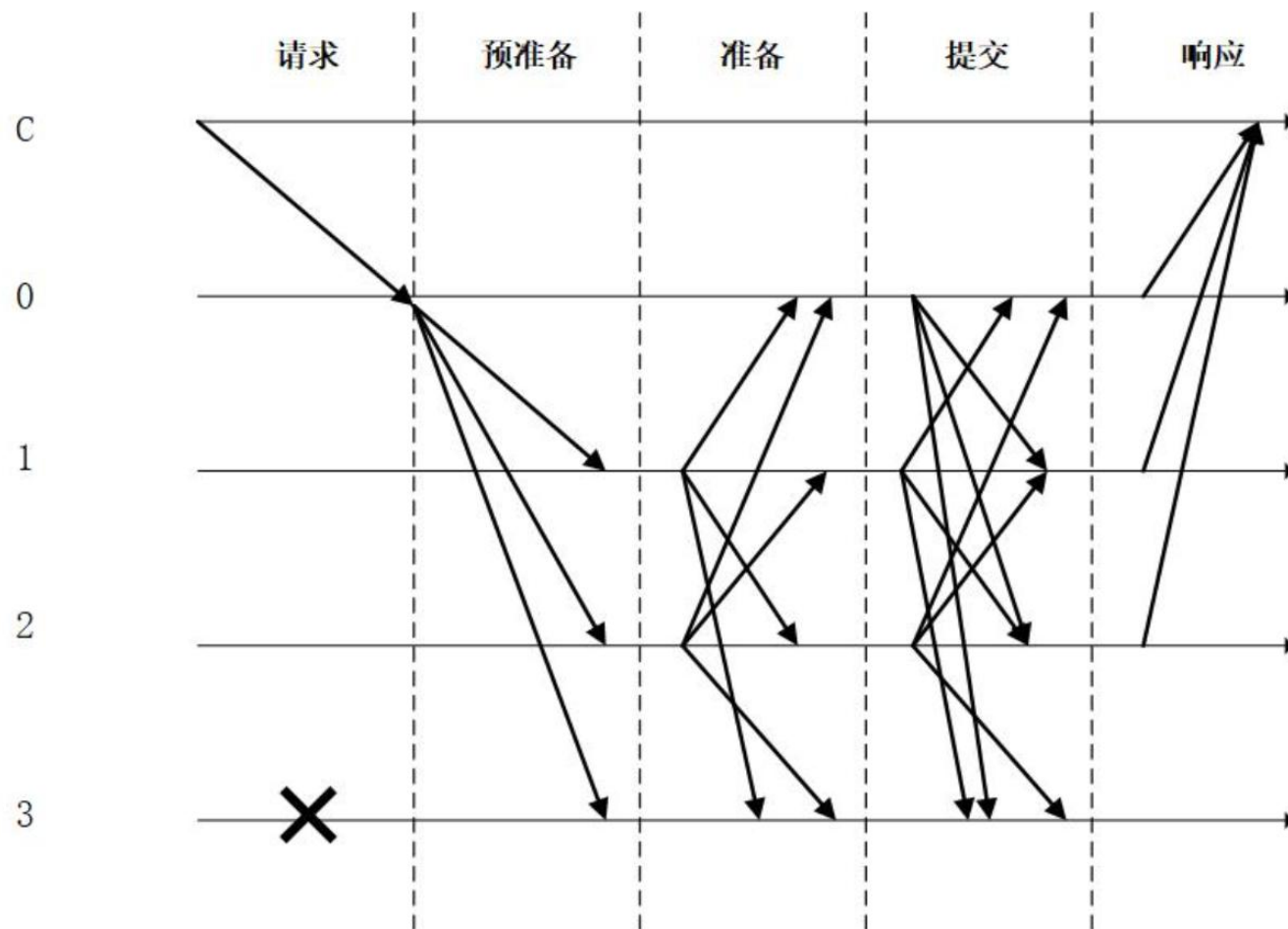
1. **o**: 客户端想要状态机执行的操作
2. **t**: 时间戳, 请求的时间戳是完全有序的, 这样能够保证后面的请求的时间戳一定比前面的大
3. **c**: 代表客户端client

预准备消息<<PRE_PREPARE,v,n,d>,m>

1. **v**: 代表正在发送消息的视图
2. **n**: 请求序号, 在预准备阶段, 主节点会给客户端传来的每个请求分配一个序号n
3. **d**: 消息摘要
4. **m**: 请求的原始消息数据

5.5.2 算法流程

—— PBFT的准备阶段和提交阶段是“全民参与的”，因为领导者或者提议者本身可能就是恶意节点，所以不能只让某个节点“发号施令”

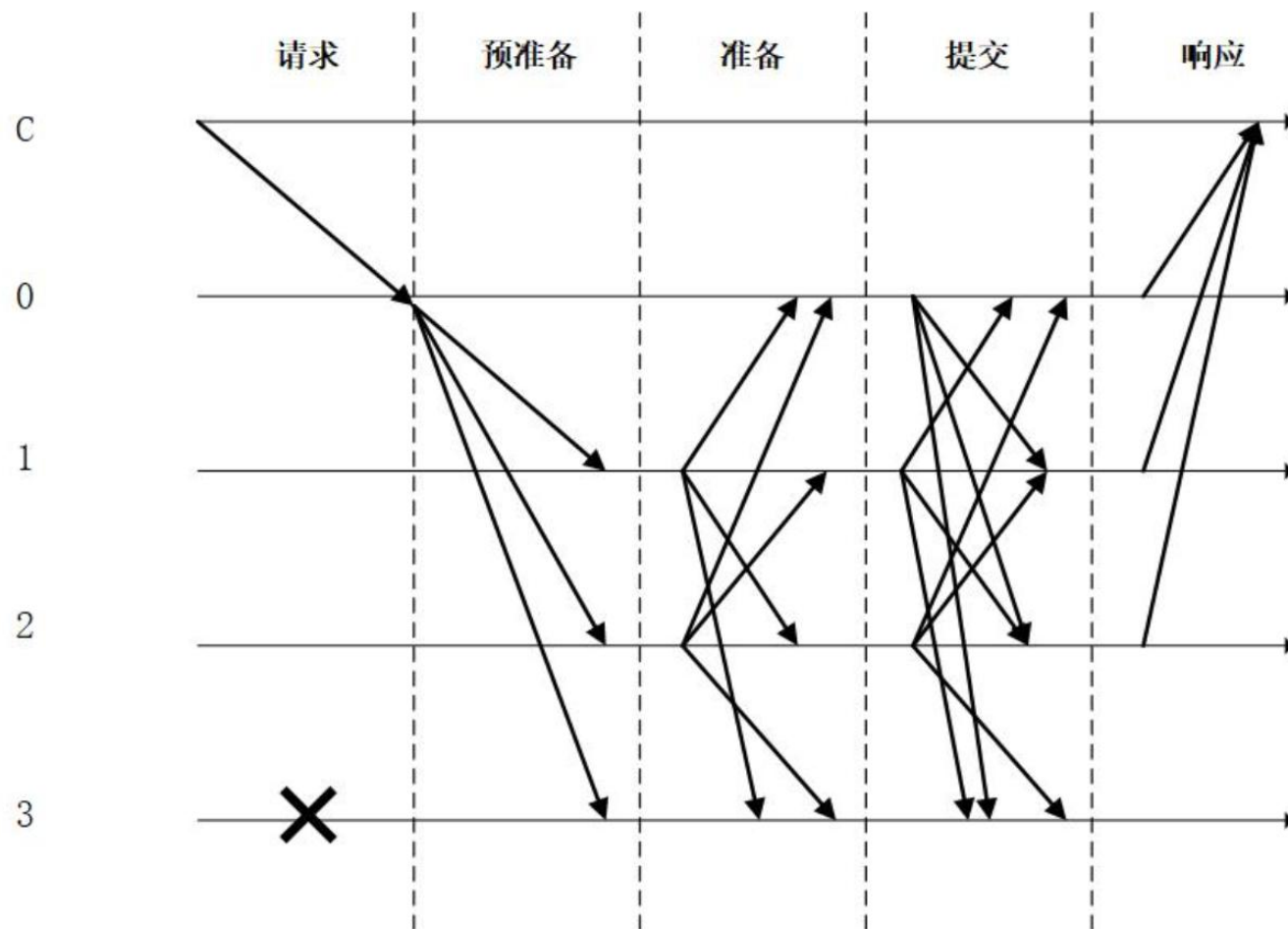


示例：

客户端将请求阶段的请求发送给主节点0，主节点0发送**预准备消息**给其他节点（ v 代表正在发送消息的视图，如果主节点变成节点1，那么 v 的值也会变化）

5.5.2 算法流程

预准备阶段



示例:

其他节点在收到预准备消息后, 进行如下验证:

- 该请求和预准备消息的签名是正确的, 且 d 确实是 m 的摘要
- 该节点正处于视图 v 中
- 该节点之前没有接受过视图 v 和请求编号 n 相同但是摘要 d 不同的预准备消息
- 该消息的请求序号在低水位 h 和高水位 H 之间

5.5.2 算法流程

准备消息<PREPARE,v,n,d,i>

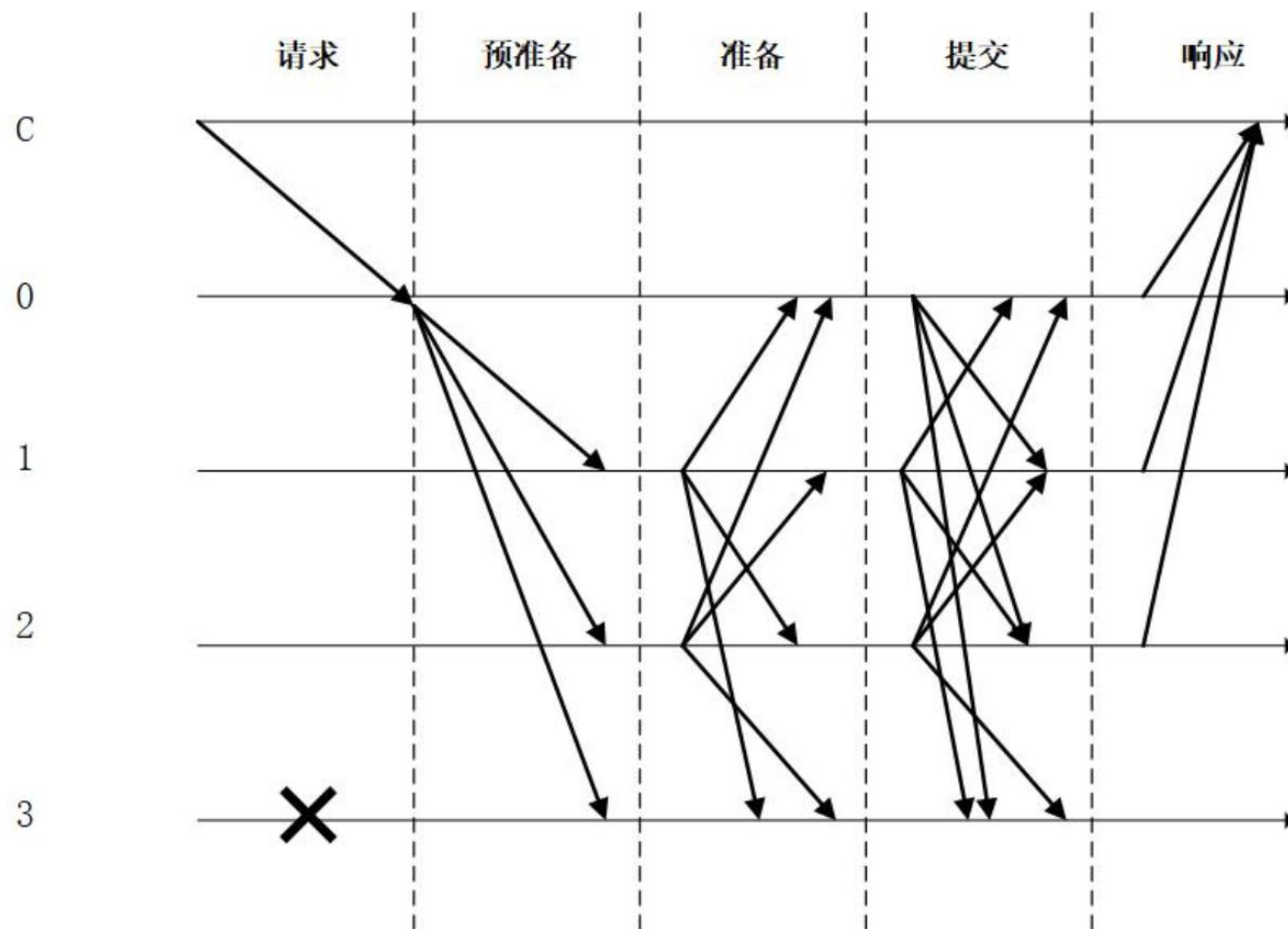
1. **v**: 正在发送消息的视图
2. **n**: 请求序号
3. **d**: 消息摘要
4. **i**: 发送该消息的节点编号

提交消息<COMMIT,v,n,D(m),i>

1. **v**: 正在发送消息的视图
2. **n**: 请求序号
3. **D (m)**: 消息m的摘要
4. **i**: 发送该消息的节点编号

5.5.2 算法流程

准备阶段



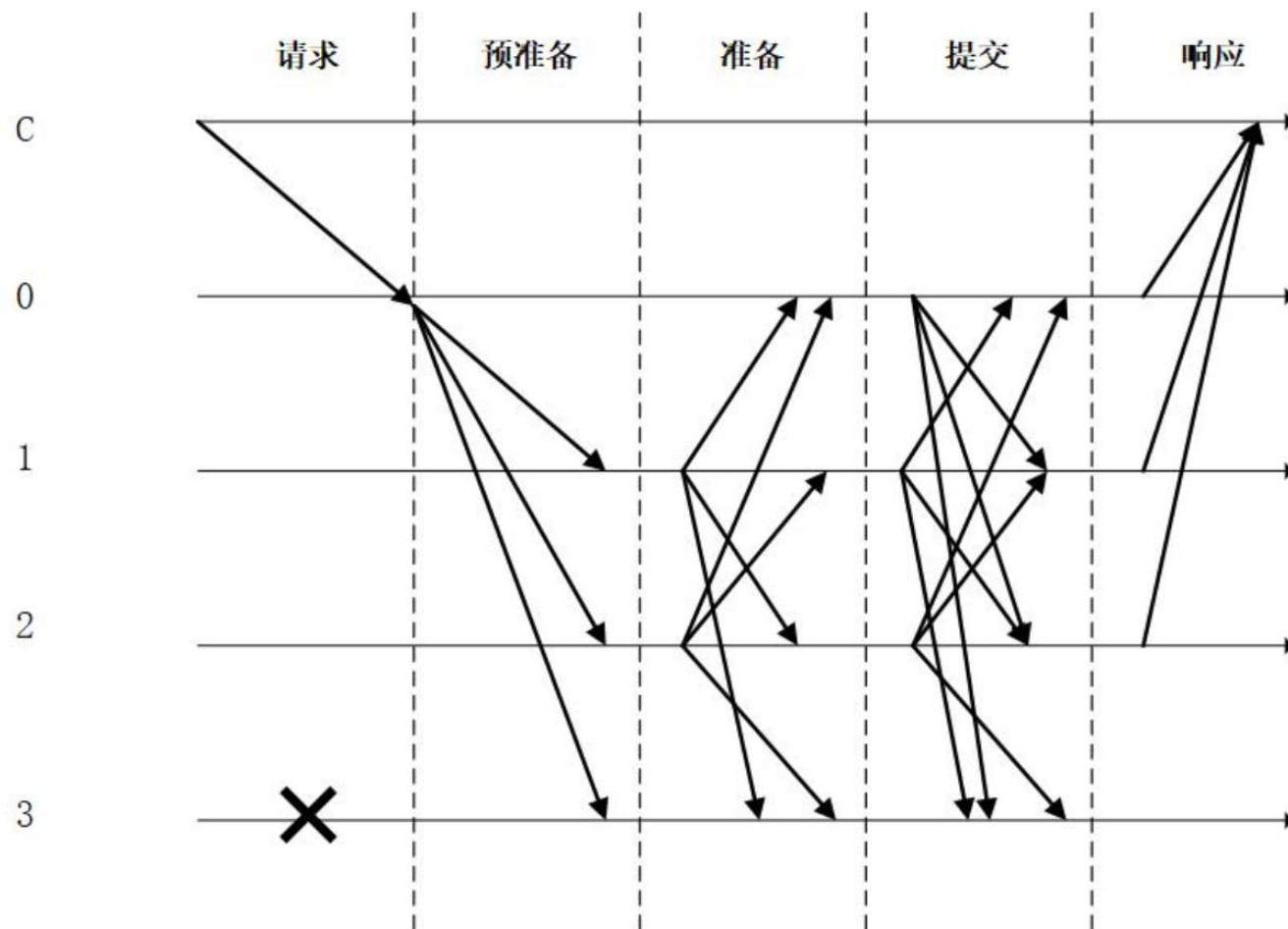
示例:

当某个节点*i*对收到的预准备验证通过后，会进入准备阶段：

- 向其他节点发送 **准备消息** $\langle \text{PREPARE}, v, n, d, i \rangle$ ，同时节点会把预准备消息和该准备消息写入自己的日志中；
- 该节点也会收到其他节点发送的准备消息，该节点会验证其他节点发送的准备消息中的 v 、 n 以及 d 是否是一致，验证通过后把该消息记录到日志中；

5.5.2 算法流程

准备阶段

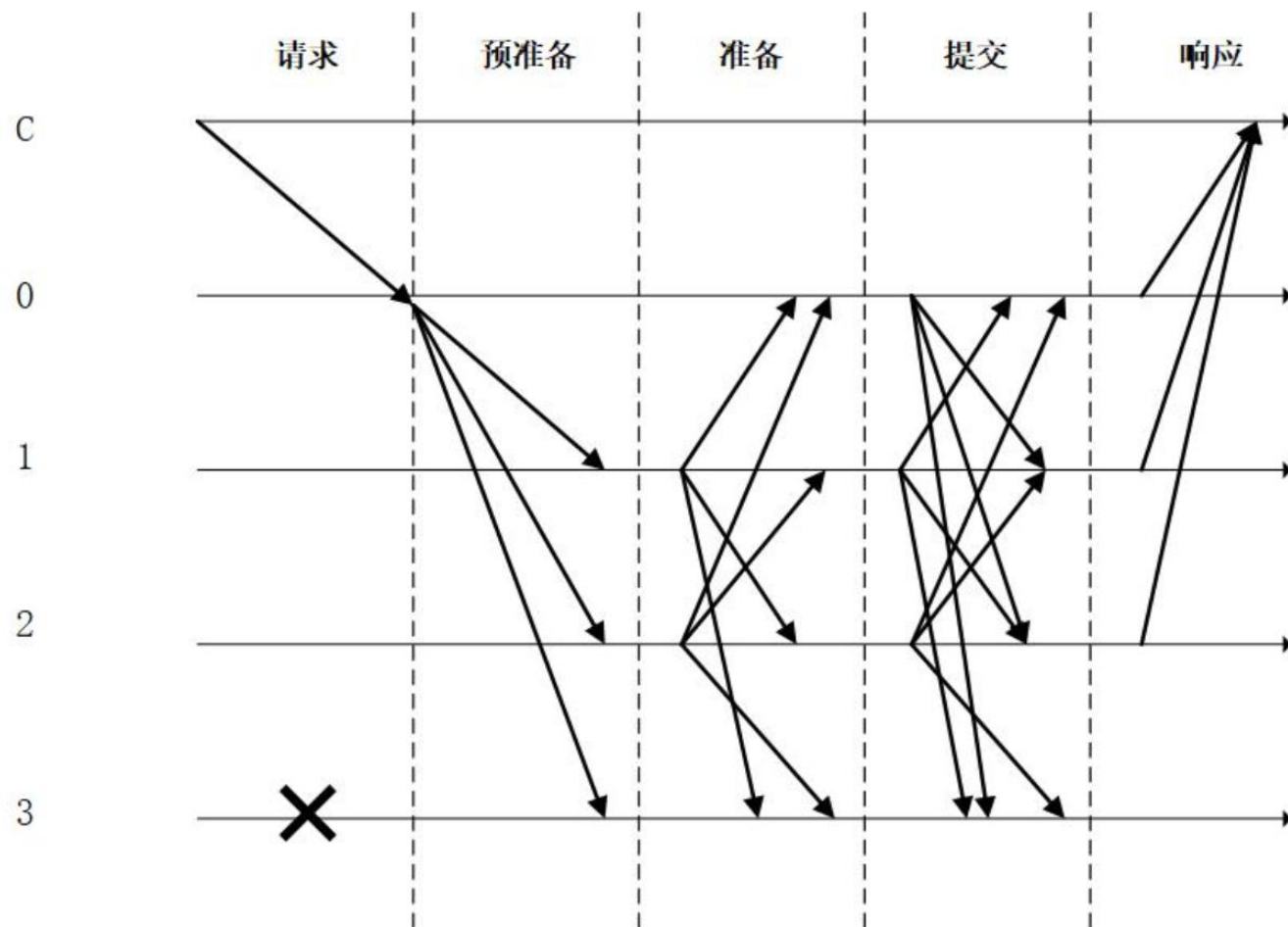


示例：

节点0、1、2都收到了除节点3之外的其他节点的准备消息并且验证通过，它们都发送了提交消息并进入了提交阶段。

5.5.2 算法流程

提交阶段



示例：

PBFT 的提交阶段需要收到 $2f+1$ 个一致消息后才能够进行提交，每个节点只会选择多数认同的消息，而不是听从某个 Leader：

节点 0、1、2 都收到了 3 个不同节点的请一致的提交消息（满足 $2f+1$ ），于是它们都会执行该请求。

5.5.2 算法流程

Question: 如果主节点是恶意节点会不会影响共识的安全性?

分析：如果共识出现问题，表明一个提案出现了两种结果，在PBFT算法中表现为有两种消息在提交阶段被执行了。为了更好地区分，将一种消息称为消息 m ，另一种消息称为消息 m' ：

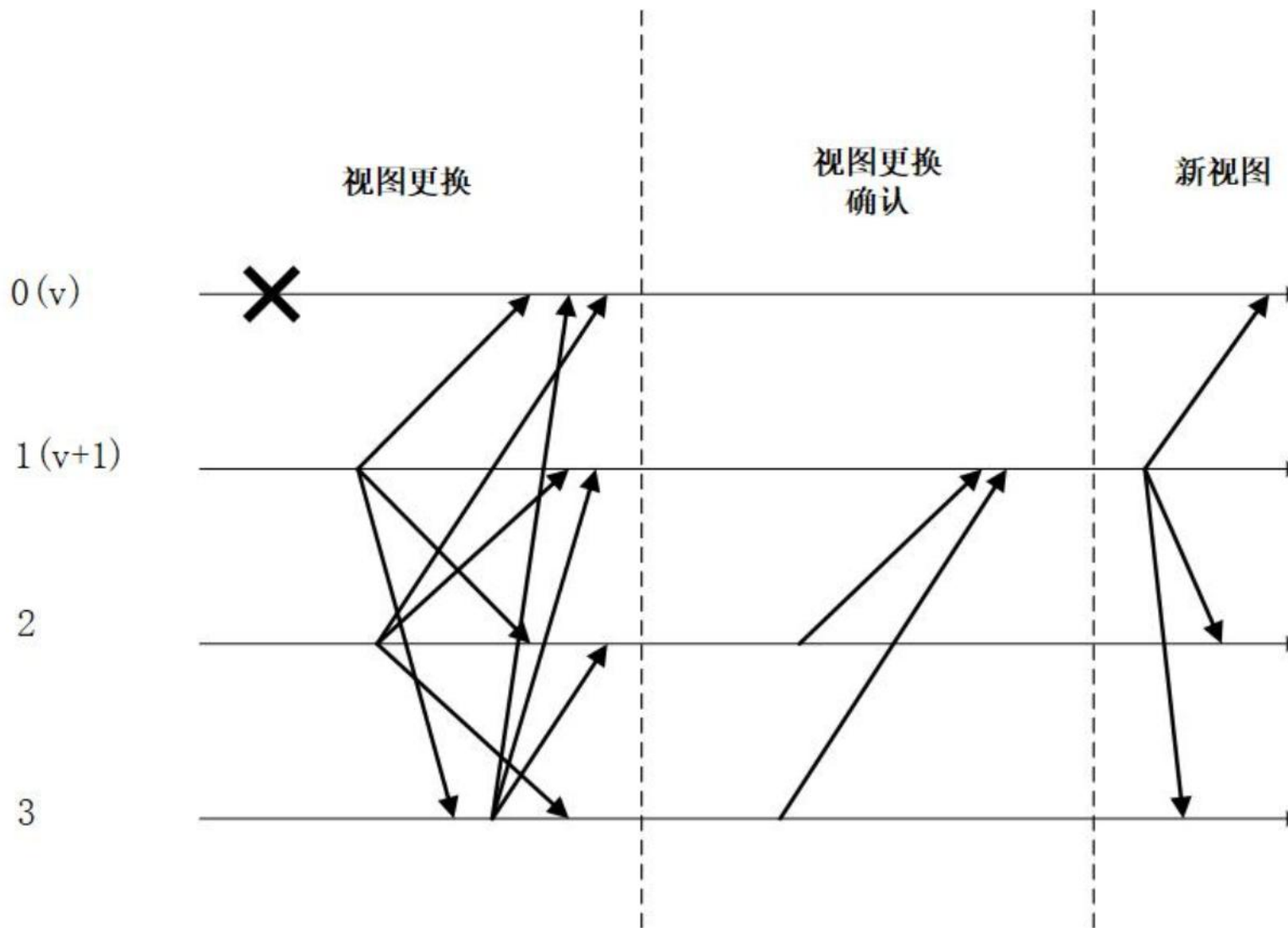
如果已经有一个诚实节点提交了消息 m ，那么消息 m' 是会被任何诚实节点提交的。因为如果消息 m' 也被提交了，那么相当于消息 m 和消息 m' 都通过了准备阶段，即有 $2f+1$ 个节点收到了消息 m ，有 $2f+1$ 个节点收到了消息 m' 。那么一定有 $f+1$ 个节点既收到了消息 m 也收到了消息 m' （多数派相交），而这 $f+1$ 个节点中至少有一个是诚实的节点，这个诚实的节点会认为这两个消息冲突了，也就不可能同时认同两个准备请求，所以这就产生了矛盾。

5.5.2 算法流程

视图更换

尽管恶意主节点不会破坏共识安全，恶意的主节点还是可能使共识过程停滞：

如果其他节点迟迟未等到共识完成，此时其他节点就会怀疑当前主节点有问题，会商议更换主节点，即更换视图。



5.5.2 算法流程

视图更换

视图更换消息<VIEW-CHANGE,v+1,n,C,P,i>

1. **v**: 上一个主节点的视图
2. **n**: 节点i的最新的stable-checkpoint的序号
3. **C**: $2f+1$ 个节点的有效checkpoint的集合
4. **P**: 节点i在上一个视图v中大于n的已经设置prepared为true的请求消息的集合

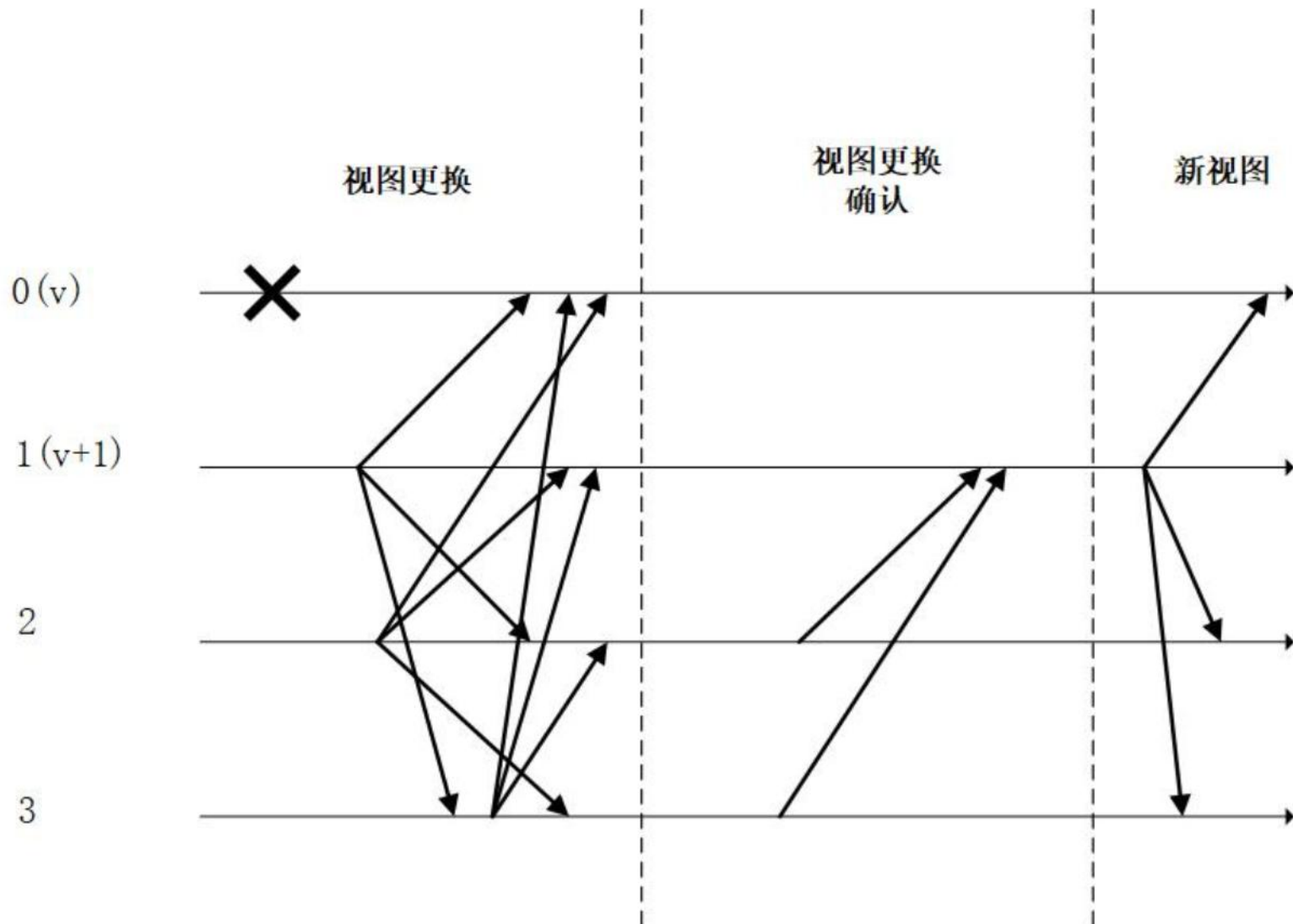
5.5.2 算法流程

视图更换

示例：

当其他节点1、2、3检测到主节点0有问题后，分别向其他节点发出了视图更换消息：

假设他们商定当前存活的最小编号的节点会在新的视图中成为新节点，新主节点1收到其他节点发送的视图更换消息，但无法确认视图更换消息是否是诚实节点发出的，主节点1让其他节点对自己收到的视图更换消息进行验证，然后将验证的结果发给主节点1，即 **视图更换确认**（view-change-ack）阶段。



5.5.2 算法流程

视图更换确认

视图更换确认消息<VIEW-CHANGE-ACK,v+1,i,j,d>

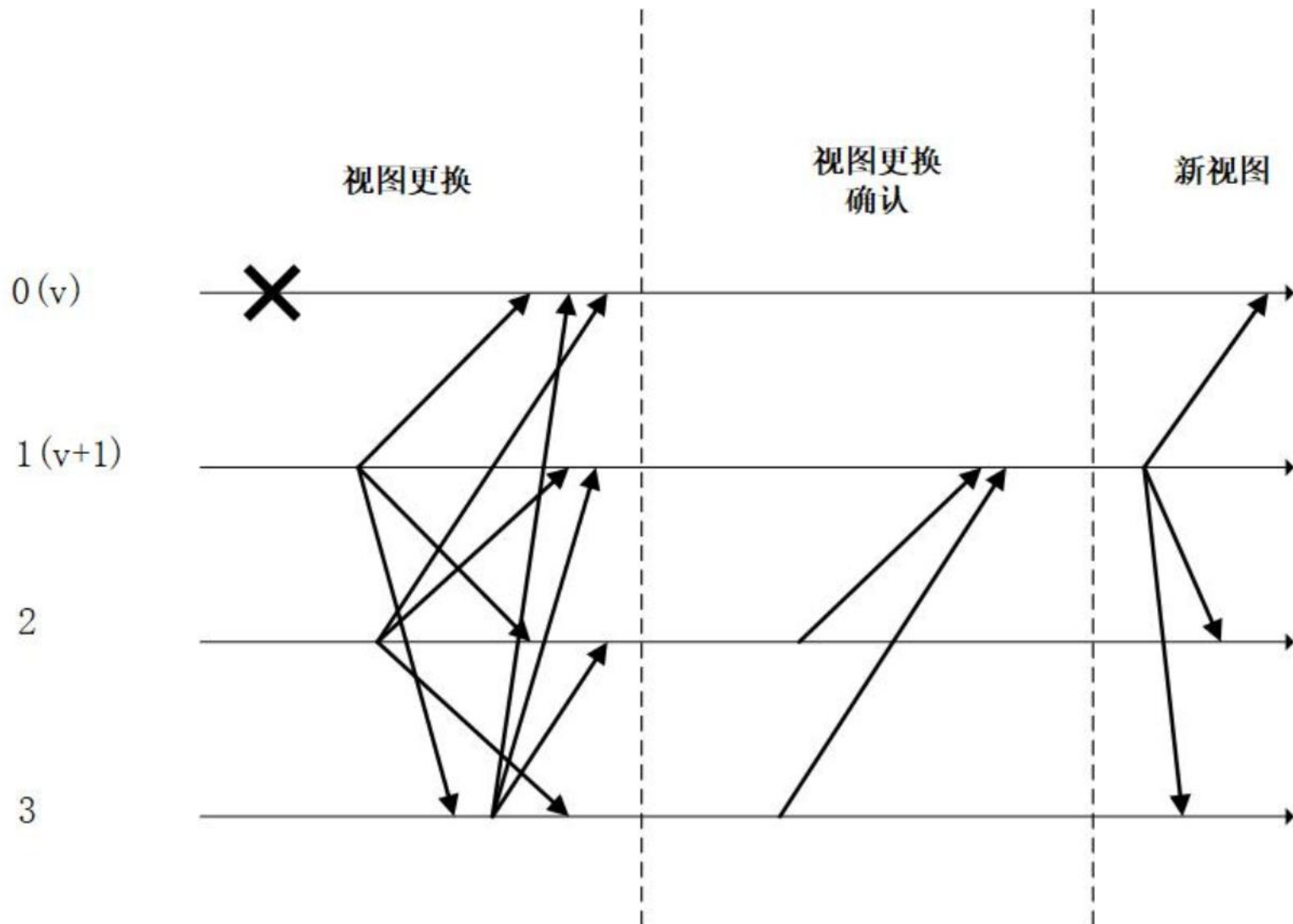
1. **v+1**: 主节点的视图
2. **i**: 发送ack消息的节点
3. **j**: 节点i要确认的视图更换消息的发送方
4. **d**: 节点i要确认的视图更换消息的摘要

5.5.2 算法流程

视图更换确认

示例：

主节点1维护了一个集合S，用来存放正确的视图更换消息。只有与某个视图更换消息相关的视图更换确认消息达到 $2f-1$ 个时，该视图更换消息才能加入到集合S，但视图更换确认消息不加入集合S。当S的数量达到 $2f+1$ 时，主节点1认为有足够多的诚实节点认为需要进行视图变更

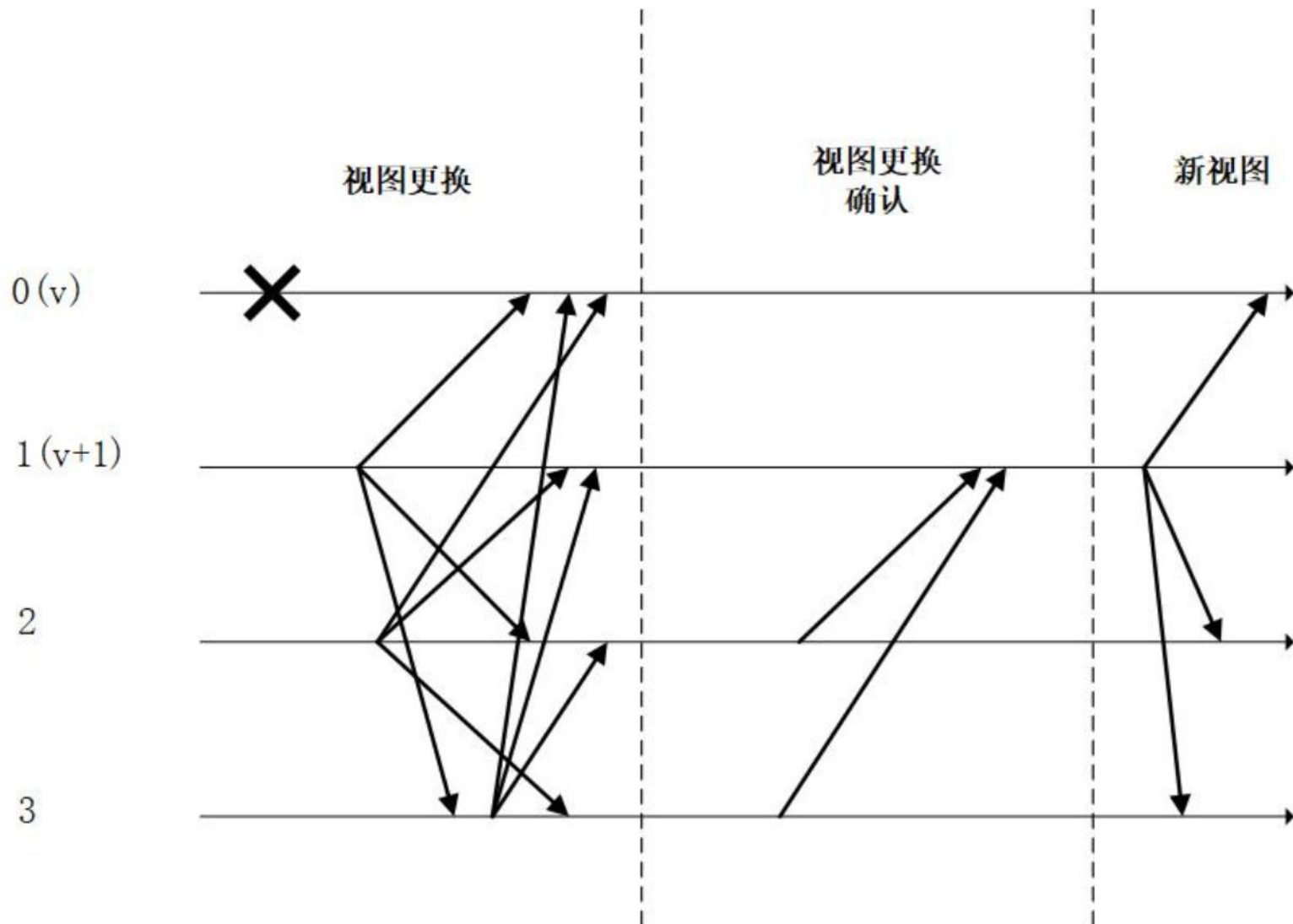


5.5.2 算法流程

新视图

示例：

其他节点在收到新视图消息后，会验证新视图消息的正确性，如果是正确的，那么会根据新视图消息中的预准备集合广播准备消息给其他节点，这样又回到了PBFT正常的三阶段流程，如果是错误的，那么该节点会发起下一轮的视图更改（即 $v+2$ ）。



本章小结

- 5.1节：从分布式系统面临的诸多挑战出发，探讨了分布式环境中可能发生各种故障。
- 5.2节：为了应对各类故障，引出了容错机制，从故障检测到故障恢复，使得系统具备了应对各类故障的能力。
- 5.3节：介绍了Paxos共识算法。
- 5.4节：介绍了RAFT共识算法。
- 5.5节：介绍了PBFT共识算法。

第五章 完