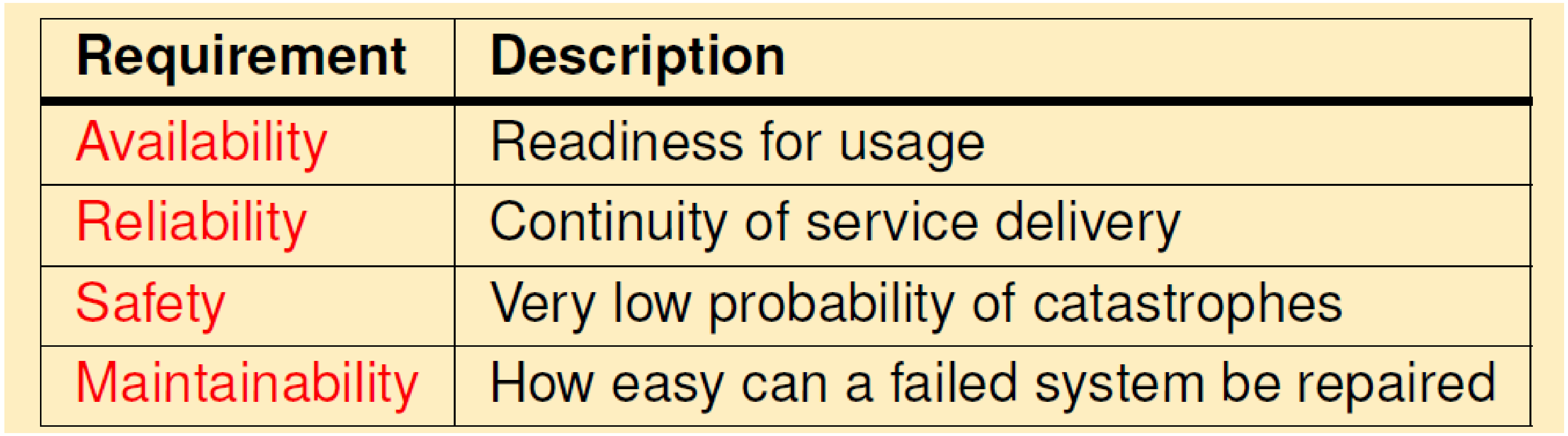
第8讲 容错

§8.1 基本概念

- 容错(Fault Tolerance)
  - 容错，容许系统的某些组件失效，而不会“严重”影响整体性能。故障发生时，系统可以进行恢复，同时可以继续进行任务处理操作。能够容错的系统，是“可依赖的”、“可靠的”（Dependable）。
- Dependability



Requirement	Description
Availability	Readiness for usage
Reliability	Continuity of service delivery
Safety	Very low probability of catastrophes
Maintainability	How easy can a failed system be repaired

可靠性(Reliability)

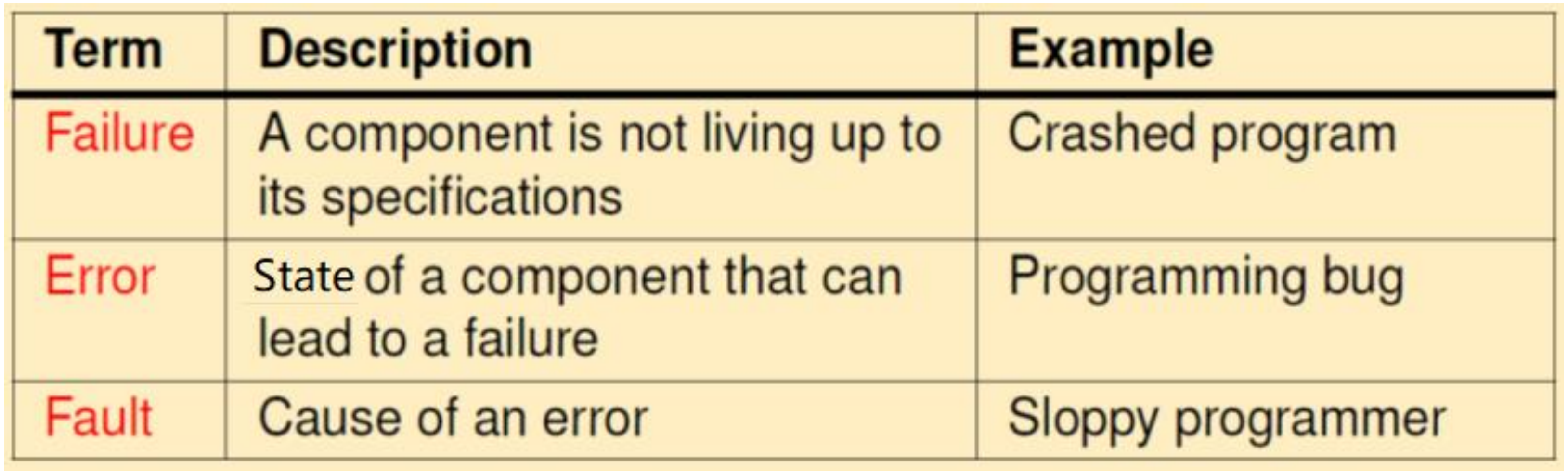
- 组件 C 的可靠性 R(t)
  - 组件 C 能够持续无故障运行的能力。
- 传统的度量
  - 平均失效时间（MTTF）： 直到组件失效的平均时间
  - 平均恢复时间（MTTR）： 恢复一个组件的平均时间
  - 两次失效的平均时间（MTBF）： MTTF + MTTR

可用性（Availability）

- 组件 C 的可用性 A(t)
  - 在 [0, t) 时间段内，组件 C 可用的时间比例；
  - 长期可用表示为：A, $A(\infty)$;
  - 注意 $A = \frac{MTTF}{MTBF} = \frac{MTTF}{(MTTR+MTTE)}$
- 注意
  - 可靠性和可用性只有在定义清楚什么是失效时才有意义；

失效与故障

- 失效（Failure）、错误（Error）、故障（fault）

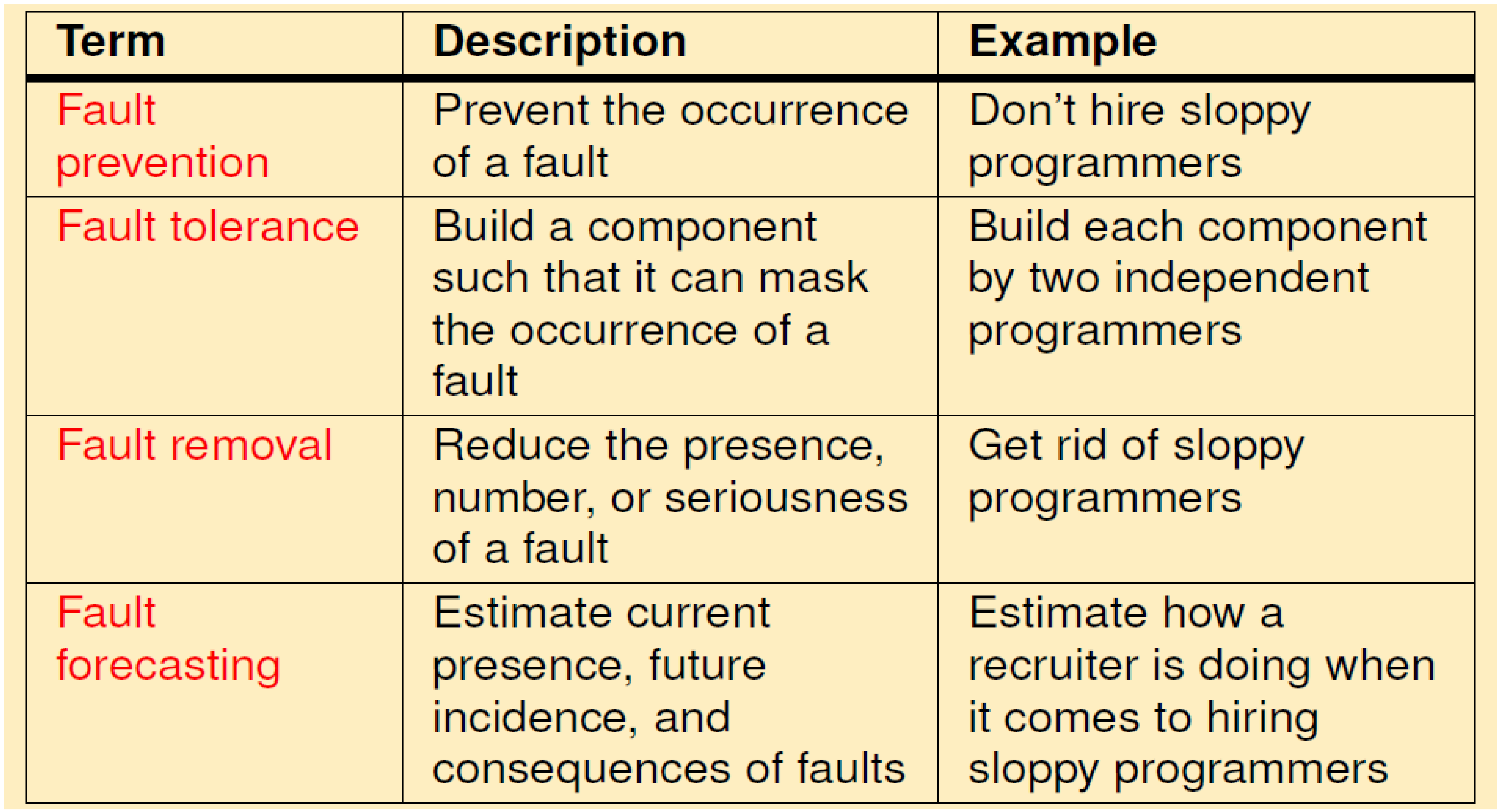


Term	Description	Example
Failure	A component is not living up to its specifications	Crashed program
Error	State of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

  - 暂时（Transient）故障：偶然发生一次，不会重复发生；
  - 间歇（Intermittent）故障：反复周期性发生；
  - 持久（Permanent）故障：故障被修复前持久存在的故障；

故障处理方式

- 如何处理故障



Term	Description	Example
Fault prevention	Prevent the occurrence of a fault	Don’t hire sloppy programmers
Fault tolerance	Build a component such that it can mask the occurrence of a fault	Build each component by two independent programmers
Fault removal	Reduce the presence, number, or seriousness of a fault	Get rid of sloppy programmers
Fault forecasting	Estimate current presence, future incidence, and consequences of faults	Estimate how a recruiter is doing when it comes to hiring sloppy programmers

失效模型（故障模式）

Type	Description of server's behavior
Crash failure	Halts, but is working correctly until it halts
Omission failure	Fails to respond to incoming requests
Receive omission	Fails to receive incoming messages
Send omission	Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure	Response is incorrect
Value failure	The value of the response is wrong
State-transition failure	Deviates from the correct flow of control
Arbitrary failure	May produce arbitrary responses at arbitrary times

Arbitrary Failure （Byzantine Failure）：可以是恶意的，也可以是非恶意的，可能有各种表现。

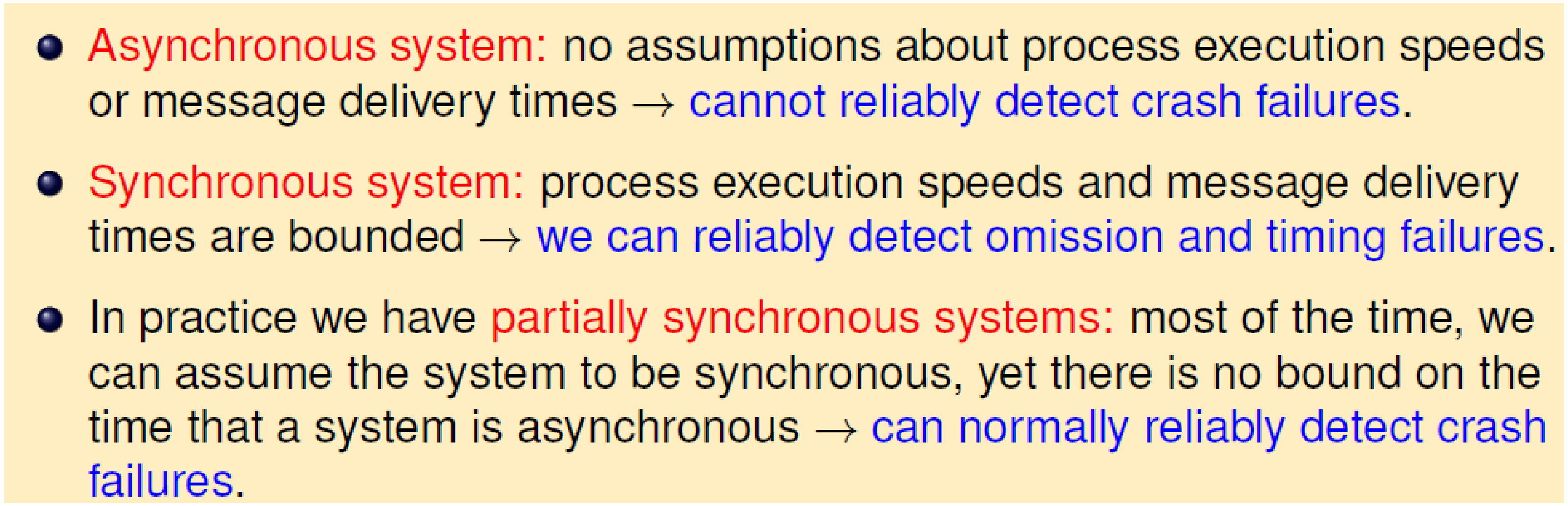
Dependability VS Security

- 遗漏（Omission）VS 执行（Commission）

  任意的失效有时候被认为是恶意的。但是，我们需要区分以下两点：
  - 遗漏性失效：组件由于没有执行应该执行的行为导致的失效；
  - 执行性失效：组件由于执行了它不应该执行的行为导致的失效；
- 观察
  - 故意失效无论是“遗漏性失效”还是“执行性失效”都是典型的安全问题；
  - 判断清楚故意失效和非故意失效实际上是非常困难的。

失效探测

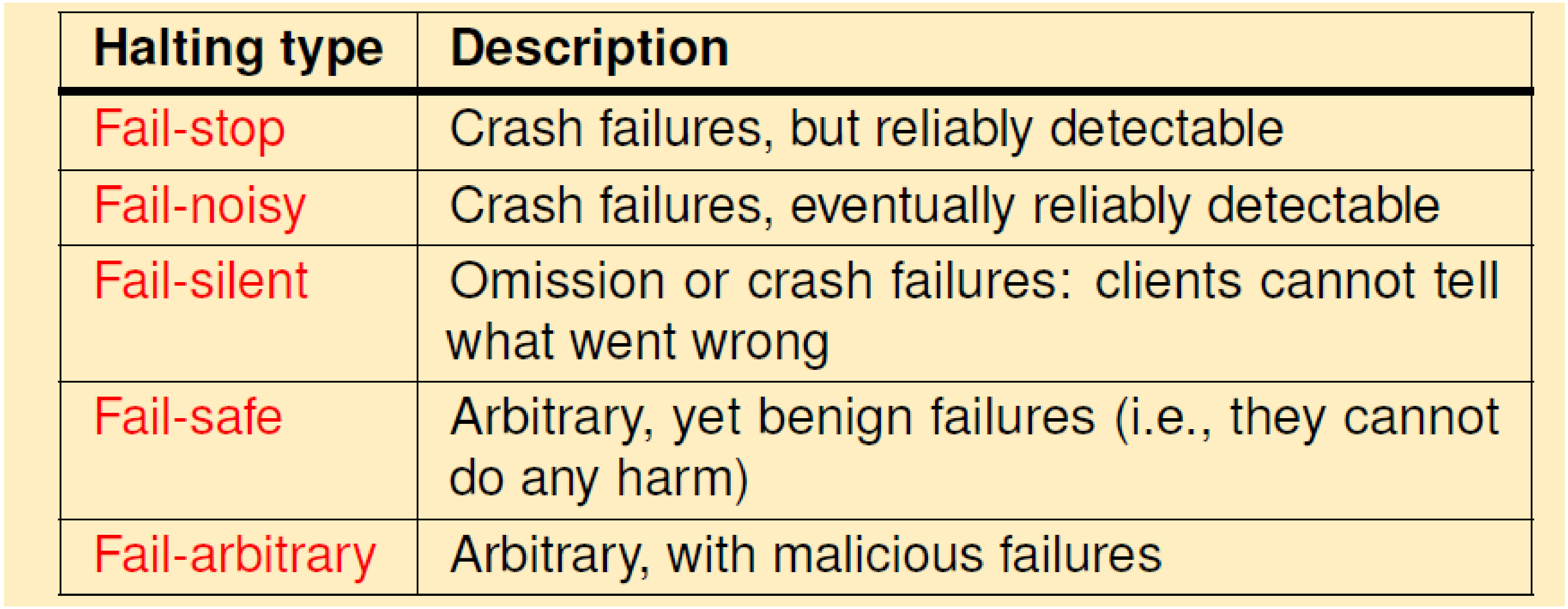
- 准确探测失效的原因和类型是困难的
  - 系统停止（Halting）： 不能感知对方的任何行为
  - 是什么失效？区分停机和遗漏性失效是几乎不可能的。
- 异步vs同步系统



- Asynchronous system: no assumptions about process execution speeds or message delivery times $\rightarrow$ cannot reliably detect crash failures.
- Synchronous system: process execution speeds and message delivery times are bounded $\rightarrow$ we can reliably detect omission and timing failures.
- In practice we have partially synchronous systems: most of the time, we can assume the system to be synchronous, yet there is no bound on the time that a system is asynchronous $\rightarrow$ can normally reliably detect crash failures.

Types of Halting Failures

- 基于Partially synchronous系统
- 从探测者（client）的角度看：



Halting type	Description
Fail-stop	Crash failures, but reliably detectable
Fail-noisy	Crash failures, eventually reliably detectable
Fail-silent	Omission or crash failures: clients cannot tell what went wrong
Fail-safe	Arbitrary, yet benign failures (i.e., they cannot do any harm)
Fail-arbitrary	Arbitrary, with malicious failures

冗余掩盖故障（Failure Masking）

- 冗余掩盖故障（Failure Masking）
  - 信息冗余：在数据单元中添加额外的位数据使错乱的位恢复正常；
  - 时间冗余：如果系统出错，一个动作可以再次执行（事务处理）；
  - 物理冗余：通过添加额外的装备或进程使系统作为一个整体来容忍部分组件的失效或故障。

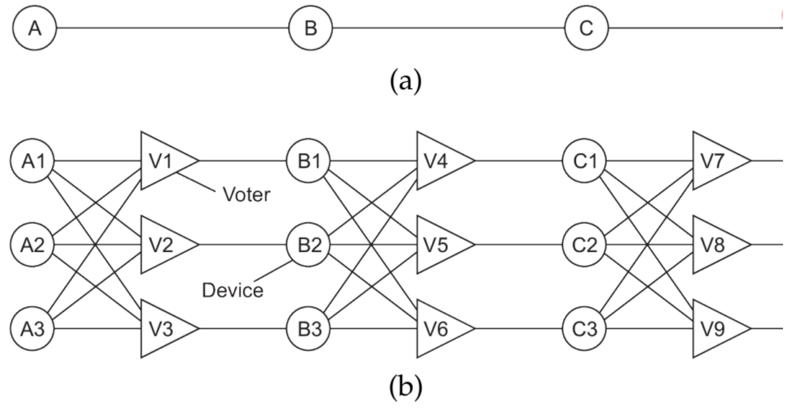
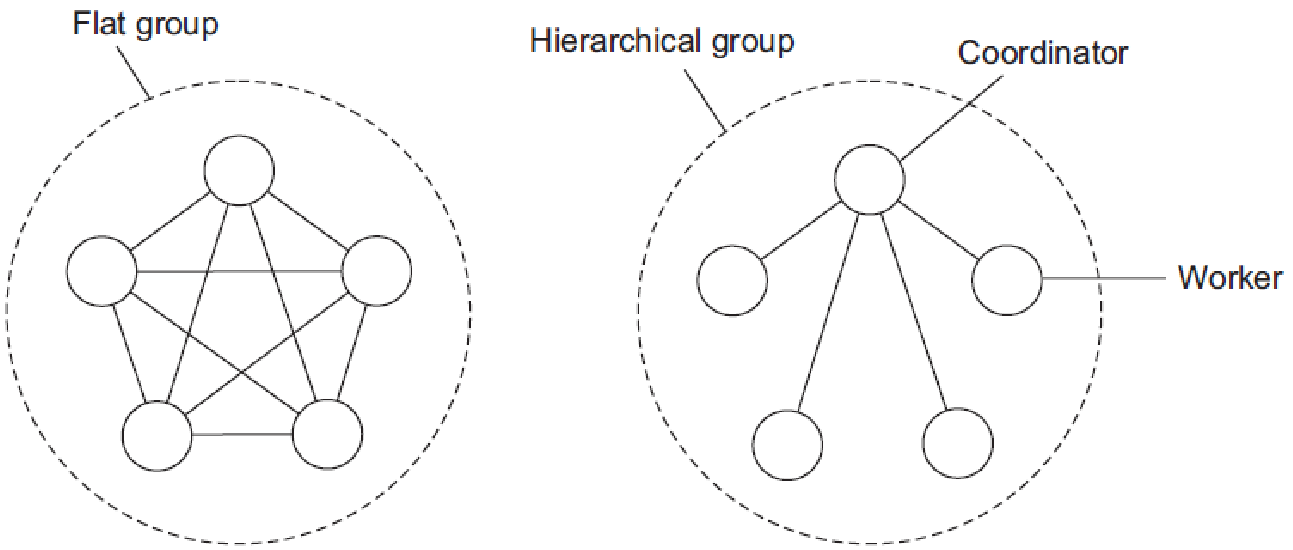


Figure 8.3: Triple modular redundancy.

§8.2 进程容错/恢复

- 容错进程组与进程复制
  - 将多个进程放在一个进程组里
  - 把一个进程组作为一个单一抽象进程来处理：
    - 进程组构成对client透明：消息、交互都是面向“组”
- 平面分组flat vs. 层次化分组hierarchical

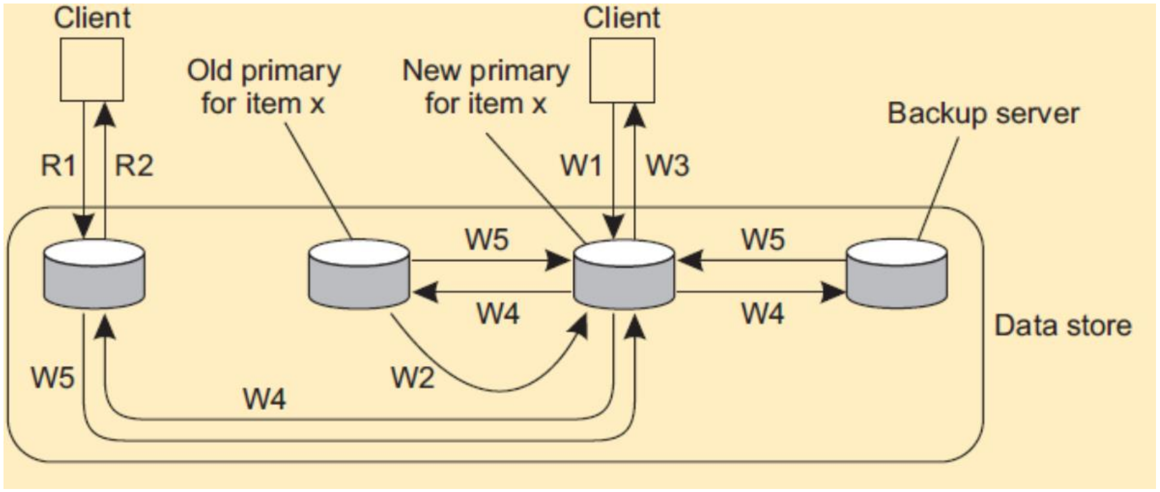


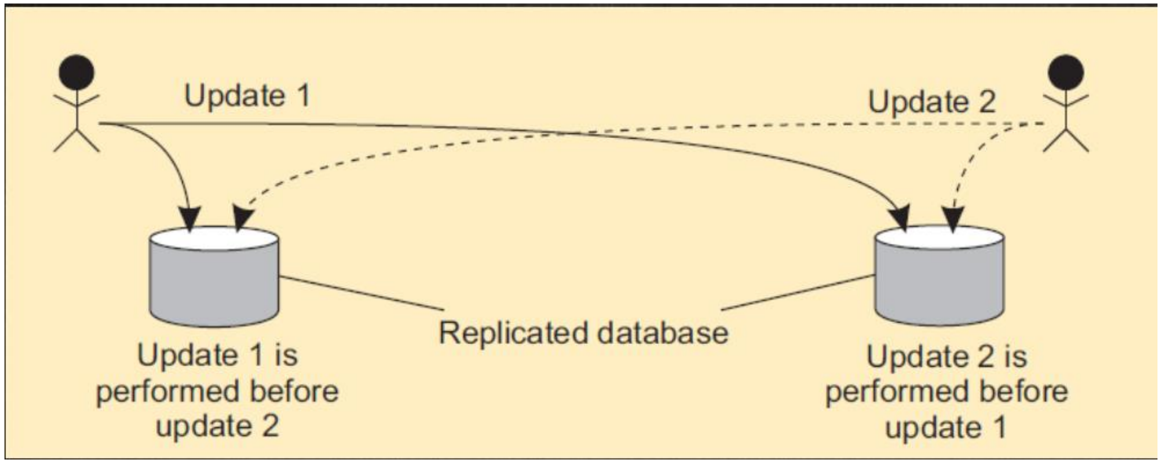
组成员管理

- 管理维护进程组及成员：
  - 组的创建、撤销
  - 成员的加入、退出
- Centralized group server
  - 问题：Single point of failure
- Distributed membership
  - Operation message to all members
  - 问题：节点失效、成员变化与应用消息同步、组失效
  - 需要专门的协议

进程复制与故障掩盖（masking）

- Primary-based and Replicated-write



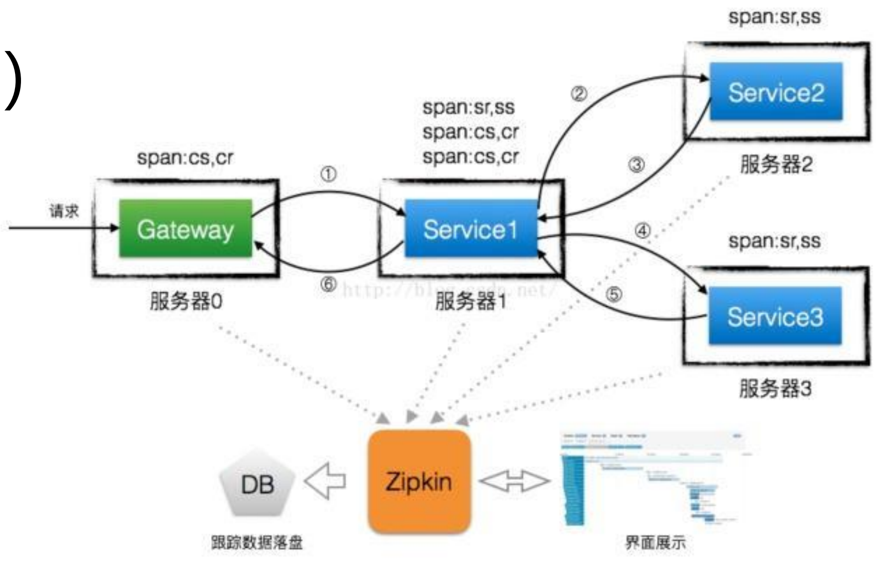

Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

进程复制与故障掩盖（masking）

- 容错度：K-容错组
  - 一个进程组可以屏蔽最多 K 个成员失效
- 复制进程组的容错度:
  - 只是停止失效（crash/omission）：
    - 需要 K+1 成员保证正确输出;
  - 任意行为的失效:
    - 需要 2k + 1个成员才能保证正确输出:
    - 最坏情况：k个进程失效，而且给出相同的错误结果
    - K+1个正确进程可以输出正确结果
      - Client可以通过“相信多数”得到正确值
  - 上述结果的前提:
    - 所有正确进程行动一致（consensus）：相同顺序执行请求
      - 无失效的话：完全有序多播、集中式的排序器等

分布式共识（Consensus）

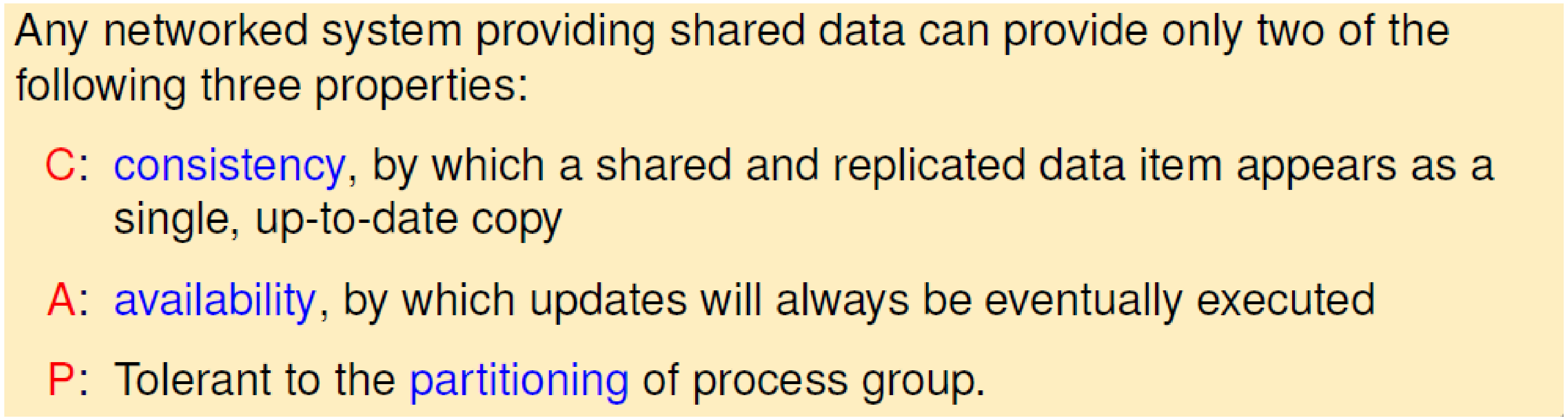
- 定义
  - 有故障场景下，使所有非故障进程在有限步骤内就某个值达成一致（如选举协调者、是否提交事务等）。
- 三个具体要求（正确性属性）
  - 正确进程输出的值相同
  - 输出的都是有效值（valid）
  - 正确进程最终都有输出
- 无节点失效情况下
  - 容易实现：有序多播等



示例场景：交易的一致性

分布式共识（Consensus）

- 有节点失效的情况下是否能达成共识?
  - 取决于系统假设（模型）
- CAP Theory:



Any networked system providing shared data can provide only two of the following three properties:

C: consistency, by which a shared and replicated data item appears as a single, up-to-date copy

A: availability, by which updates will always be eventually executed

P: Tolerant to the partitioning of process group.

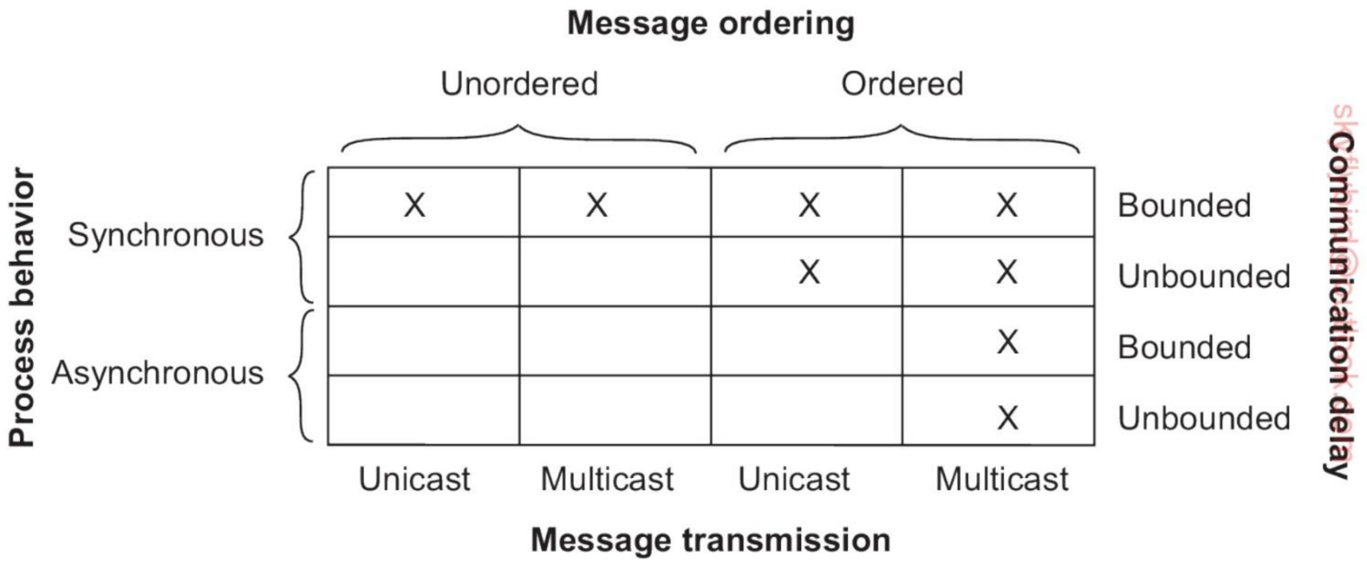
- Consensus
  - 笼统来说，异步的分布式系统环境下无法达成共识。

M.J. Fischer, N.A. Lynch, and M.S. Paterson.
Impossibility of distributed consensus with one faulty process. J. ACM, 32(2):374–382, 1985.

分布式共识（Consensus）

- 什么环境下能容忍节点失效达成共识?

M.J. Fischer, N.A. Lynch, and M.S. Paterson.
Impossibility of distributed consensus with one faulty process. J. ACM, 32(2):374–382, 1985.



Process behavior	Communication delay	Unordered Unicast	Unordered Multicast	Ordered Unicast	Ordered Multicast
Synchronous	Bounded	X	X	X	X
Synchronous	Unbounded			X	X
Asynchronous	Bounded				X
Asynchronous	Unbounded				X

Message ordering
Message transmission

Process synchrony: whether the processes operate in a lock-step mode. (There should be some constant c ≥ 1, such that if any process has taken c + 1 steps, every other process has taken at least 1 step.)

Delay bound: whether messages are delivered with a globally and predetermined maximum time.

Message ordering: whether messages are delivered in the order of sending in real global time.

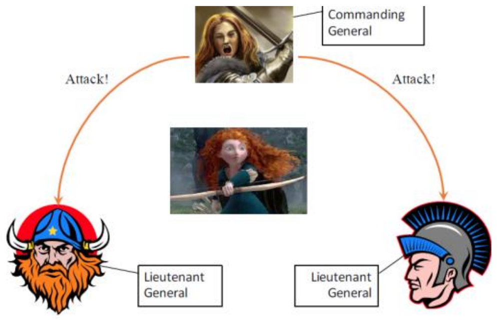
Transmission mode: unicasting or multicasting.

共识协议分类

泛洪　Paxos　Raft　ZAB　BFT　PBFT　PoW

准确检测失效　能最终检测失效　拜占庭容错（有恶意行为）

仅“停止失效”容错



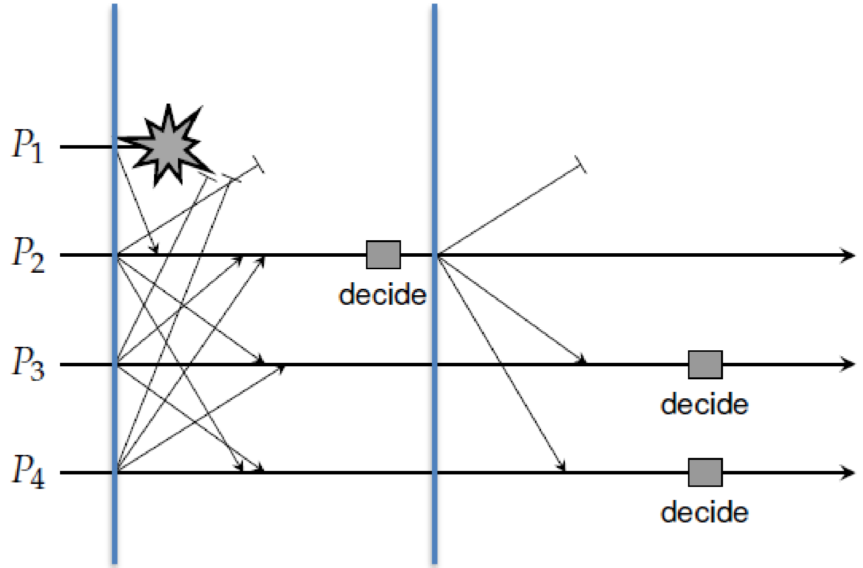
拜占庭共识问题（Byzantine agreement）
--L. Lamport, 1982

基于泛洪的共识

- 系统模型
  - 给定一个进程组 P={P_1, P_2, …, P_n};
  - Fail-stop类型的失效，可以准确探测到系统失效；(synchronous system)
  - 客户端联系 P_i 让其执行命令；
  - 每一个进程 P_i 维护一个发出命令的列表；
- 基本算法（基于轮次）

1. In round *r*, P_i multicasts its known set of commands $\mathbf{C}_i^r$ to all others
2. At the end of *r*, each P_i merges all received commands into a new $\mathbf{C}_i^{r+1}$.
3. Next command cmd_i selected through a globally shared, deterministic function: $cmd_i \leftarrow select(\mathbf{C}_i^{r+1})$.

基于泛洪的共识：举例



- 要点
  - P2 获得所有进程提交的命令：作出决定，广播决定给其他进程；
  - P3/P4 检测到 P1 失效，但是不知道其他进程是否收到P1消息：
    - 不做决定；进入下一轮；
  - 下一轮： P3/P4 收到P2的决定消息后做出决定。

Paxos共识协议

- 相对实际的系统假设（无恶意行为）
  - 不完全同步系统（实际上，也可以是异步系统）
  - 进程间的通信不可靠，消息可能丢失、重复和乱序；
  - 损坏的消息能够被检测到（在后续处理中可以忽略）；
  - 所有操作都是确定的，即操作一旦执行，其结果确定；
  - 进程可能会出现崩溃失效，但不是任意型的失效；
  - 进程之间不会相互串通、欺骗。

L. Lamport. Paxos made simple. ACM SIGACTNews (Distributed Computing Column), 32(4):51–58,December 2001.

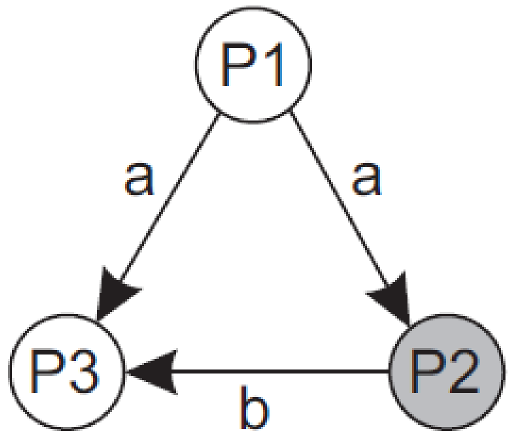
Paxos的详细过程，下一章讲解。

拜占庭容错（共识）

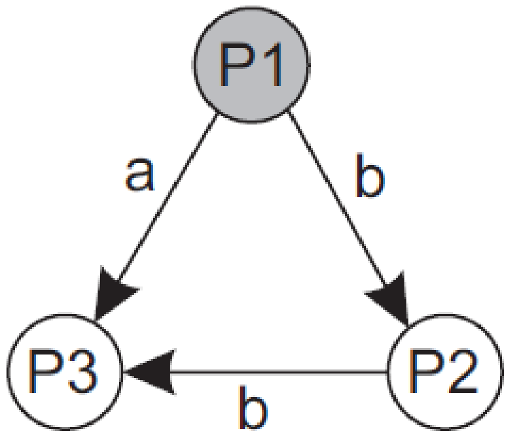
- 任意行为类型的失效（存在恶意行为）

  我们考虑这样的进程组，进程之间的通信是非一致的：

  (a) 不正确的消息传递；（b）告诉不同的进程不同的信息；



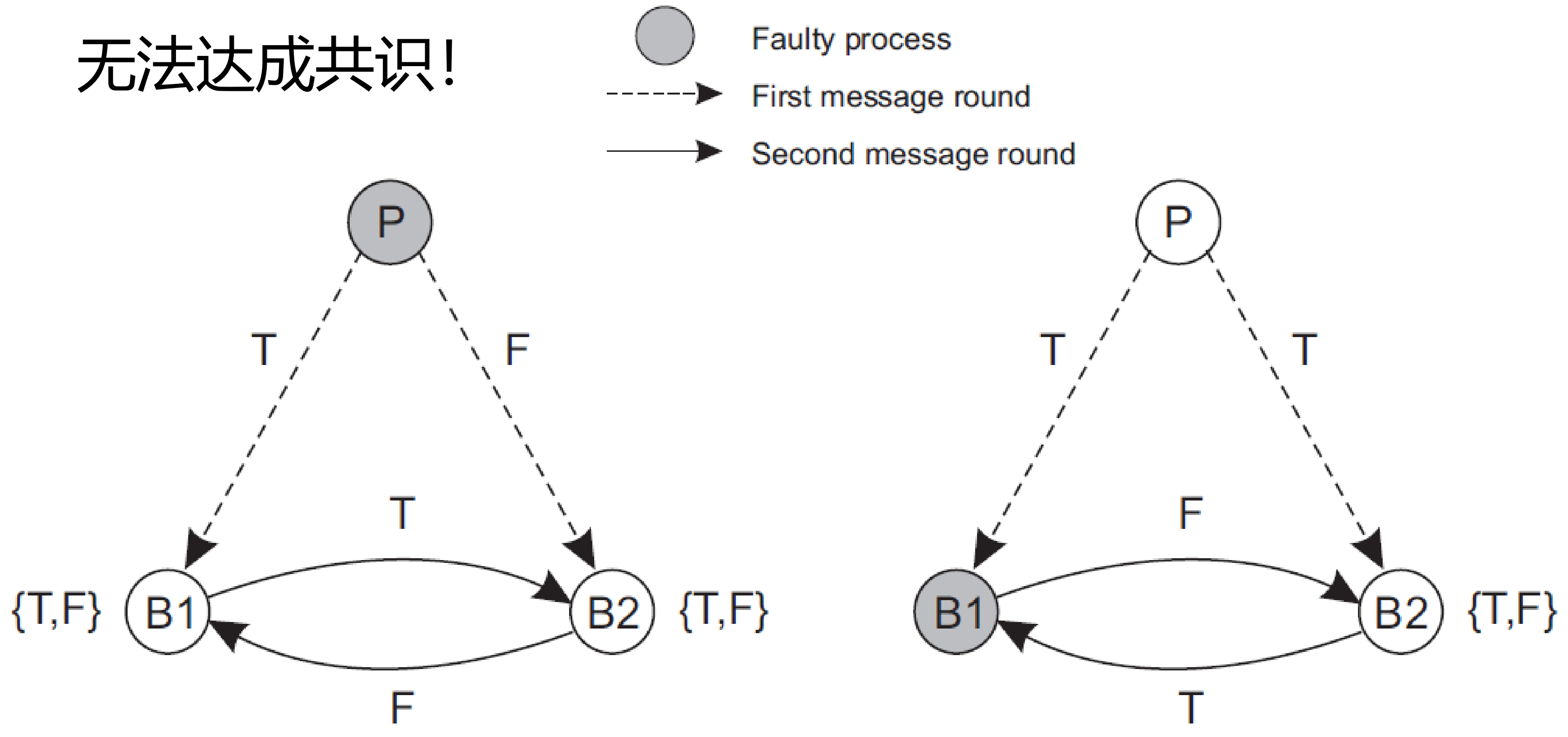
(a)



(b)

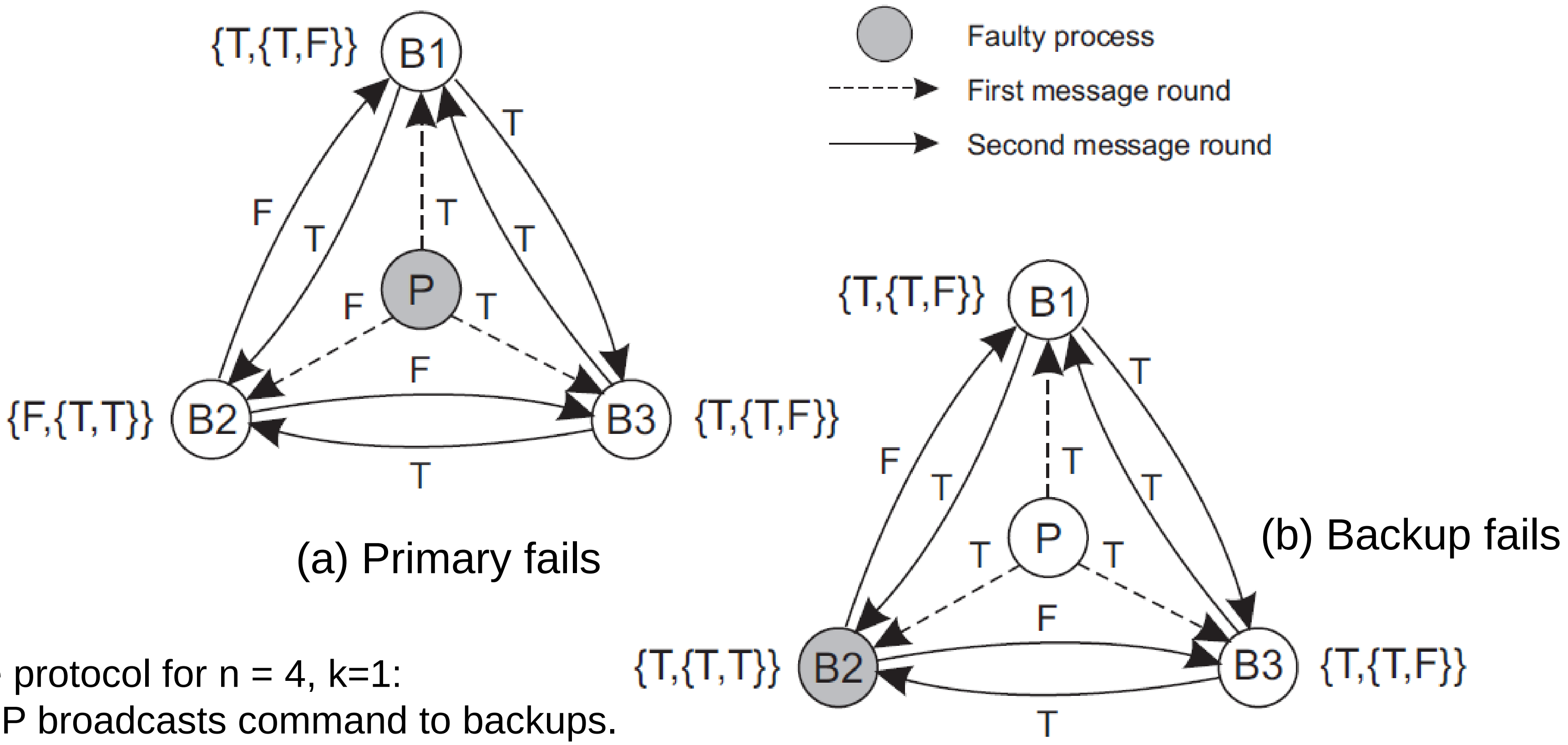
拜占庭容错：3k进程无法k容错

无法达成共识！



- Round 1: the faulty primary P is sending two different values to the backups B1 and B2, respectively.
- Round 2: in order to reach consensus, both backup processes forward the received value to the other.
- B1 and B2 each have received the set of values {T, F}, impossible to draw a conclusion.

拜占庭容错：3k+1进程



The protocol for n = 4, k=1:

1. P broadcasts command to backups.
2. Each backup rebroadcasts command from P to one another.
3. When all three messages arrive, each subordinate takes the majority decision to be the final decision.

拜占庭共识协议BFT

System model

- We consider a primary P and $n-1$ backups $B_1, \ldots, B_{n-1}$.
- A client sends $v \in \{T, F\}$ to P
- Messages may be lost, but this can be detected.
- Messages cannot be corrupted beyond detection.
- A receiver of a message can reliably detect its sender.

同步系统！

Byzantine agreement: requirements

BA1: Every nonfaulty backup process stores the same value.
BA2: If the primary is nonfaulty then every nonfaulty backup process stores exactly what the primary had sent.

Observation

- Primary faulty $\Rightarrow$ BA1 says that backups may store the same, but different (and thus wrong) value than originally sent by the client.
- Primary not faulty $\Rightarrow$ satisfying BA2 implies that BA1 is satisfied.

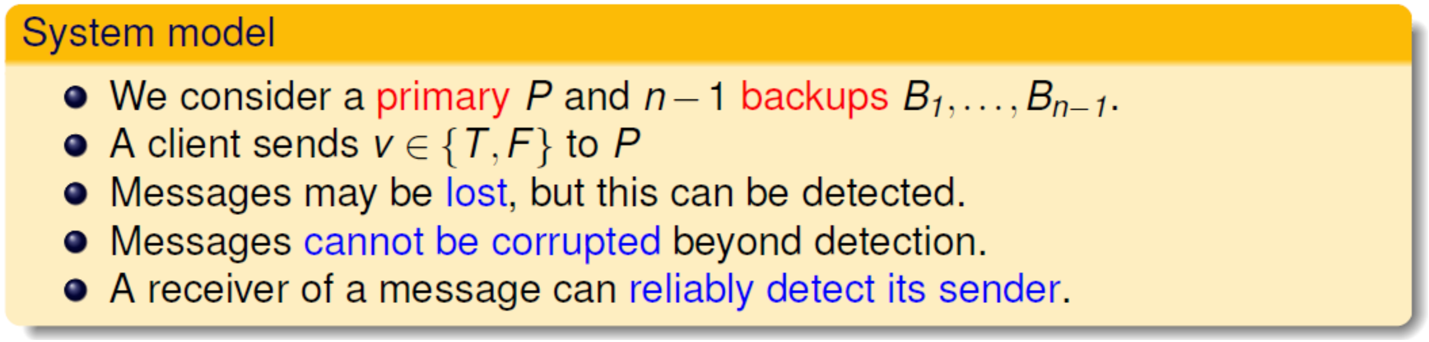
BFT的详细过程，下一章讲解。

拜占庭共识协议PBFT

Miguel Castro, Barbara Liskov: Practical Byzantine Fault Tolerance. OSDI 1999

- Practical的系统假设和协议性能
  - A faulty server is assumed to exhibit arbitrary behavior;
  - Messages may be lost, delayed, and received out of order;
  - A message's sender is identifiable (having messages signed).

前面BFT协议的假设



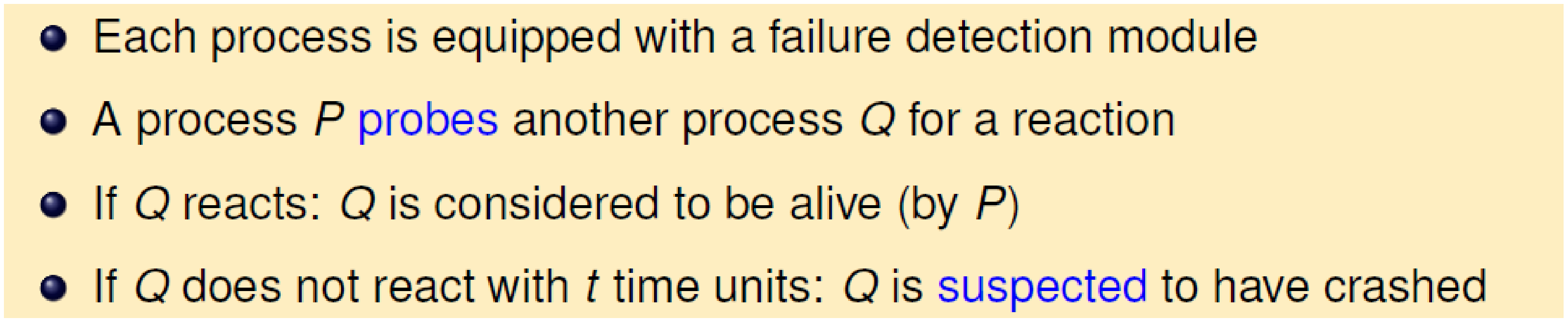
System model

- We consider a primary P and $n-1$ backups $B_1, \ldots, B_{n-1}$.
- A client sends $v \in \{T, F\}$ to P
- Messages may be lost, but this can be detected.
- Messages cannot be corrupted beyond detection.
- A receiver of a message can reliably detect its sender.

PBFT的详细过程，下一章讲解。

故障检测 Failure Detection

- 问题
  - 我们如何可靠地检测一个进程是否失效了？
  - 基本方法：timeout机制（主动查询 vs. 被动等待）
- 通用模型



- Each process is equipped with a failure detection module
- A process *P* probes another process *Q* for a reaction
- If *Q* reacts: *Q* is considered to be alive (by *P*)
- If *Q* does not react with *t* time units: *Q* is suspected to have crashed

- 同步系统 vs. 异步系统

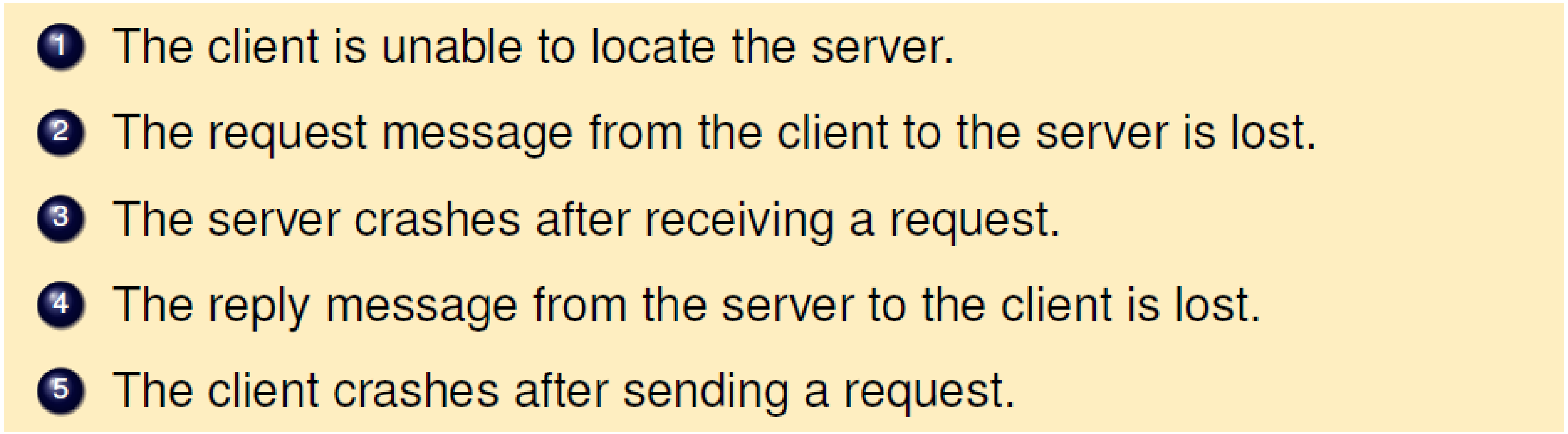
实际的失效检测

- 最终准确性
  - 如果 P 没有在规定的时间 t 内收到来自 Q 的心跳信息：P 怀疑 Q失效；
  - 如果 Q 稍后发出消息：
    - P 停止怀疑 Q；
    - P 增加timeout的时间；
  - 如果 Q 确实宕机， P 会一直怀疑 Q;

Timeout不够好，但很难做得更好！

§8.3 可靠C-S通信

- 可靠RPC
  - 错误类型



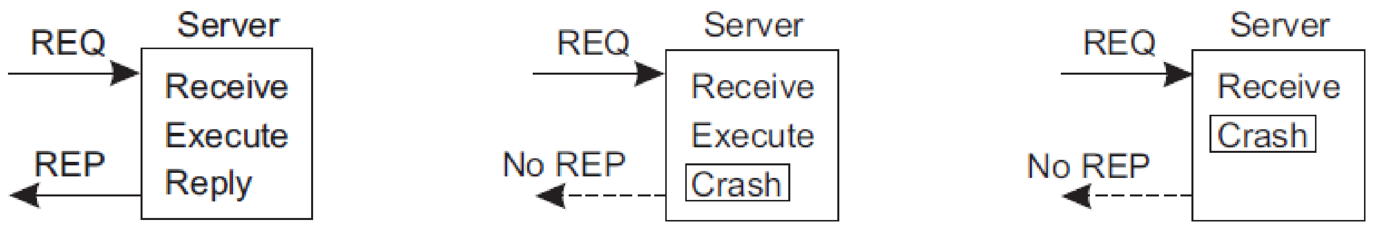
1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

可靠RPC

- 错误1：不能定位服务器
  - 原因：服务器关闭、服务端版本变化（参数变化）等
  - 解决办法：生成并发送报错信息（如抛出exception）（一定程度上破坏透明性）
- 错误2：请求消息丢失
  - 原因：网络问题或其他原因
  - 解决办法：设置timeout，然后重发（如果总是超时，可以认为是“不能定位服务器”或“服务器宕机”）

可靠RPC

- 错误3：服务器宕机



  – 两种情形：执行后宕机、执行前宕机
  – 解决办法：
    - at-least-once semantics：keep try until getting reply
    - at-most-once semantics: give up and report error
    - guarantee nothing: anything may happen
    - exactly-once semantics: ? impossible to guarantee

Exactly-once不可能

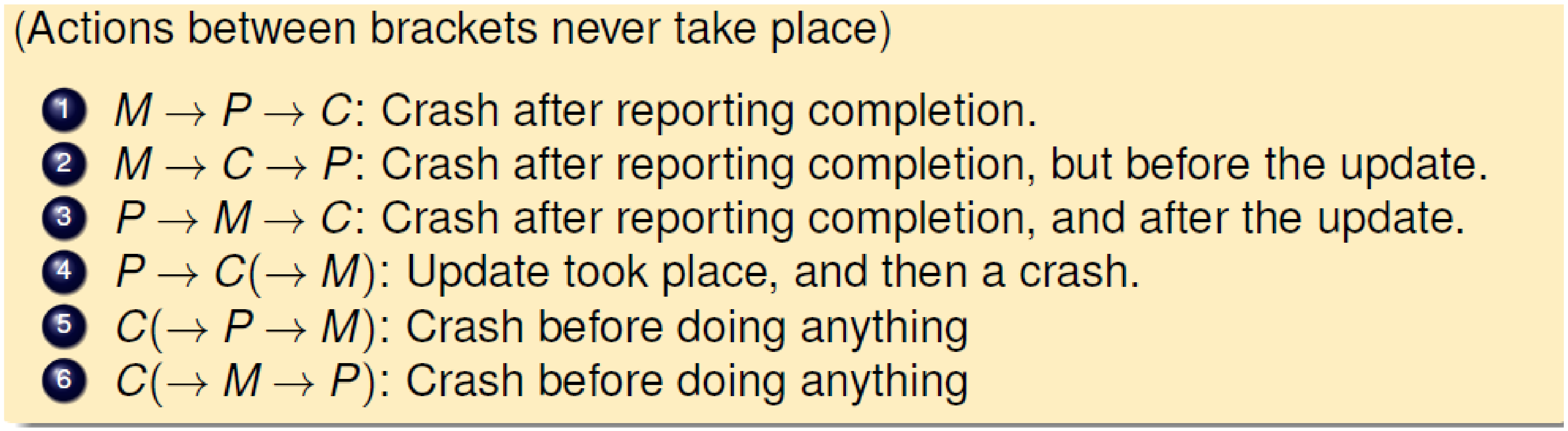
- 服务器端的三种不同的事件

  场景假定：请求服务更新文档;

  M: 发送完成信息;

  P：完成文档处理;

  C：Crash
- 6种不同的顺序



(Actions between brackets never take place)

1. $M \rightarrow P \rightarrow C$: Crash after reporting completion.
2. $M \rightarrow C \rightarrow P$: Crash after reporting completion, but before the update.
3. $P \rightarrow M \rightarrow C$: Crash after reporting completion, and after the update.
4. $P \rightarrow C (\rightarrow M)$: Update took place, and then a crash.
5. $C (\rightarrow P \rightarrow M)$: Crash before doing anything
6. $C (\rightarrow M \rightarrow P)$: Crash before doing anything

Exactly-once不可能

	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK
Client	Server			Server		

OK	=	Document updated once
DUP	=	Document updated twice
ZERO	=	Document not update at all

没有一种客户策略和服务器策略的结合可以在所有可能的失效事件顺序下正确工作

可靠RPC

- 错误4：丢失应答信息

  原因：无法断定原因是请求/应答消息丢失，还是服务器宕机。

  – 解决办法：设计服务器时，让其操作都是幂等的（idempotent）即重复多次执行与执行一次的结果是相同的：
    - 纯粹的读操作
    - 严格的写覆盖操作

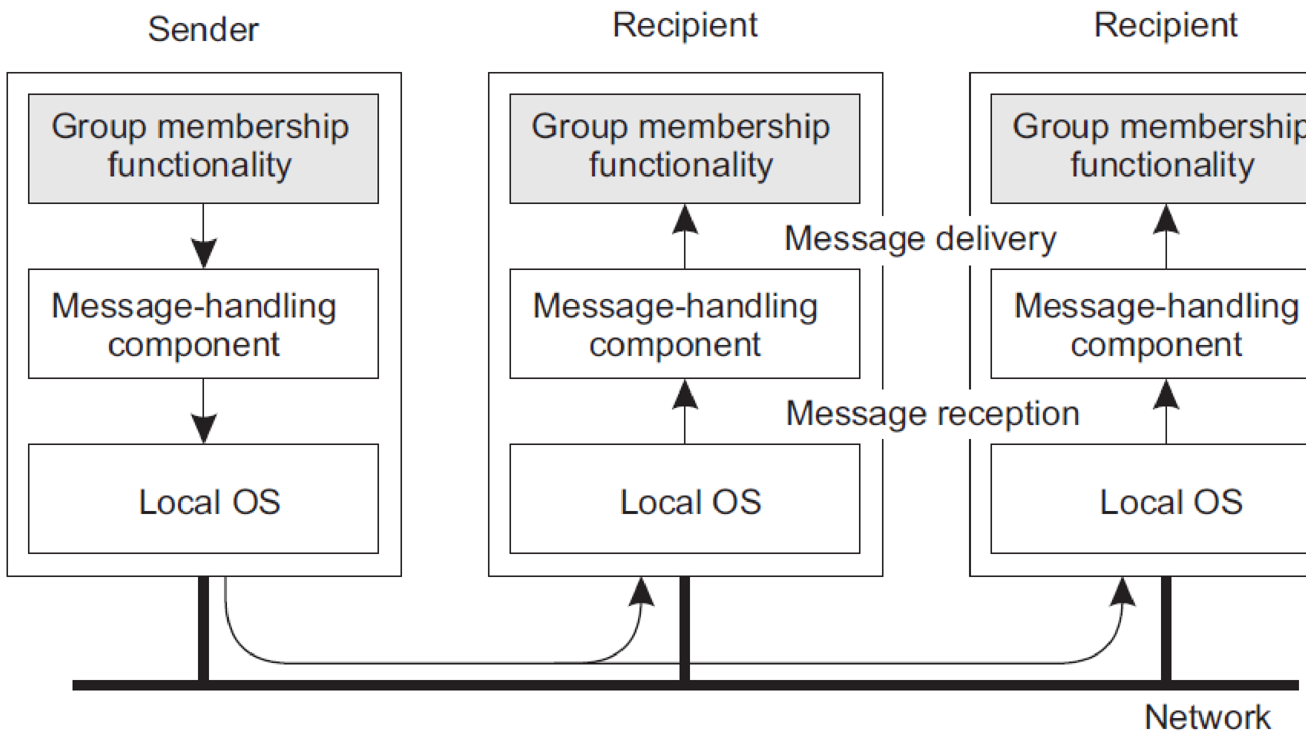
  – 但是实际上很多操作天然就不是幂等的，例如银行事务系统。

可靠RPC

- 错误5：客户端宕机
  服务器的计算结果没有用户了，成为孤儿计算（Orphan）。
  - 消灭：当客户端恢复时杀死孤儿;
    (弊端：客户端需要成本来记录状态、grandorphan无法处理等)
  - 再生：按时间周期（epoch），客户端恢复后，向通知所有服务器开始新周期，杀死与客户端相关的孤儿;
  - 优雅再生：新周期开始后，服务器尝试找回client，如果找不到拥有者则杀死孤儿;
  - 到期：每个RPC都被给定一个标准的时间量 T 来进行工作，超时后需要申请延期。重启后等待T，所有孤儿会被终止。

可靠的组通信

- 直观理解
  - 一个消息可靠地发送到一个进程组 G,
  - G 中的每个进程都能够收到（区分接收和交付）。

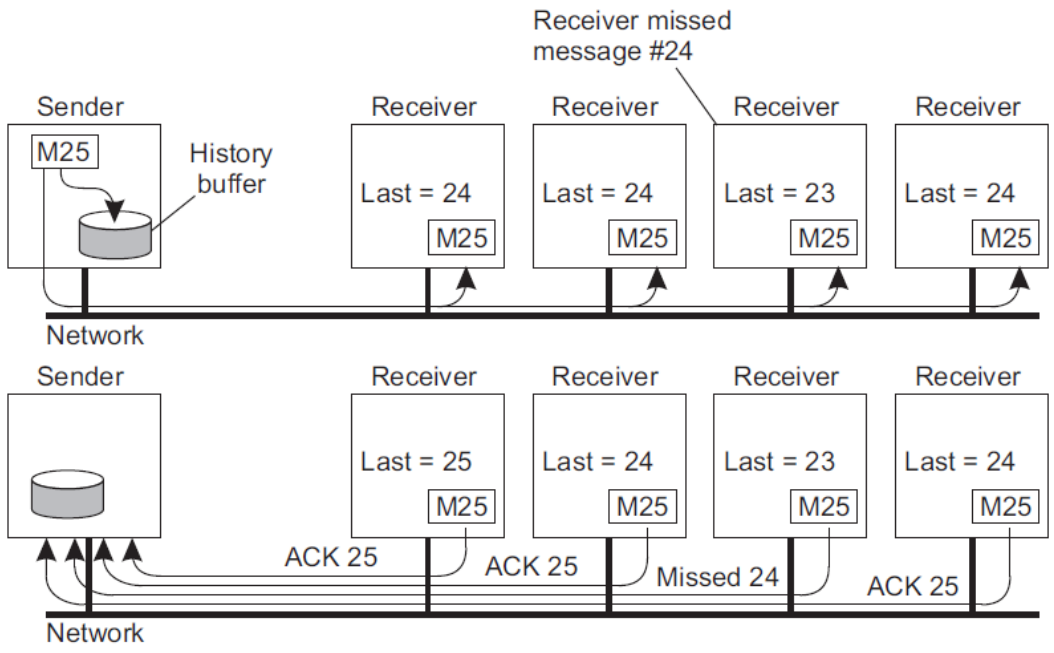


Sender
Recipient
Recipient
Group membership functionality
Group membership functionality
Group membership functionality
Message delivery
Message-handling component
Message-handling component
Message-handling component
Message reception
Local OS
Local OS
Local OS
Network

简单的可靠组通信

- 不考虑组成员变化（组成员变化会涉及consensus问题）
- 无故障情形：可以用可靠点播协议，如TCP，实现
- 有故障情况：消息丢失

  简单方案：消息编号后发送，基于编号缺失重发。



具体细节改变?
Ack piggyback、
point retransmit等

局限性：扩展性差!
(ack implosion)

可靠多播的可扩展性

- 如何避免Ack implosion?
  - 简单方案：接收方只在消息丢失时才发一个反馈消息；
    - 存在的问题是：需要缓存大量的陈旧的信息；
  - 进一步改进：反馈抑制，即：消息丢失反馈被多播给全组，避免重复反馈。

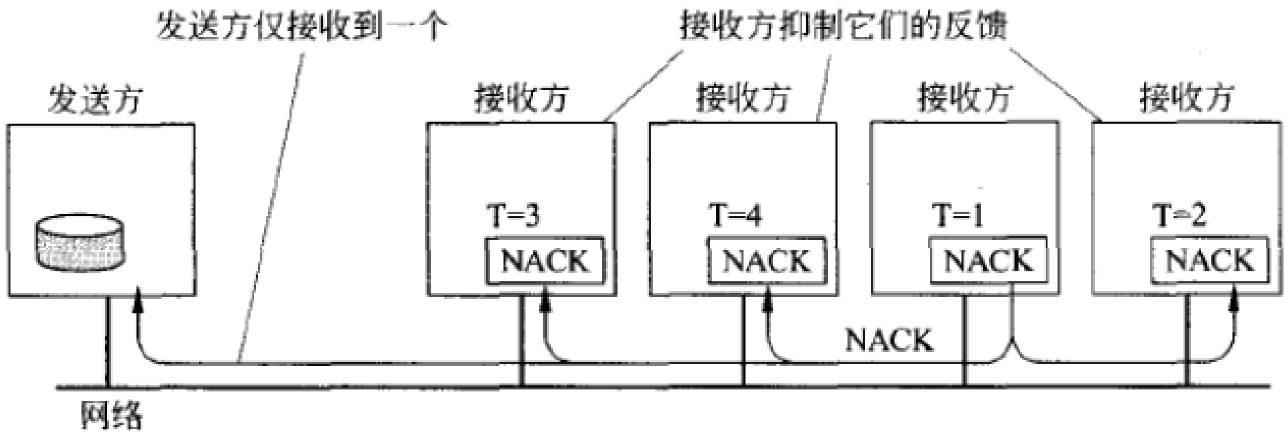
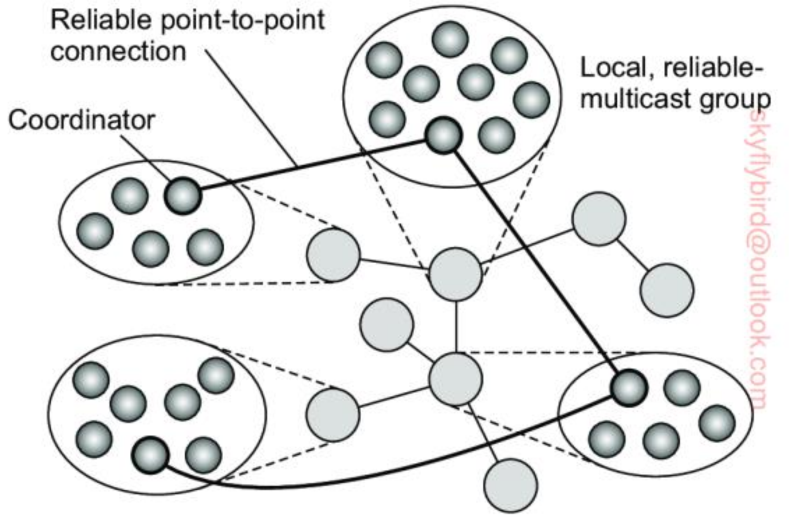


图 8.10 几个接收方要发送重发请求，但是第一个重发请求抑制了其他的请求

仍然很多问题：
同时发送的反馈
正常接收者被打断等

层次化多播及反馈控制

- 进程被分成小组，每组有coordinator
- Coordinator构成树结构



Reliable point-to-point connection
Coordinator
Local, reliable-multicast group

Figure 8.24: The essence of hierarchical reliable multicasting. Each local coordinator forwards the message to its neighboring coordinators in the tree and later handles retransmission requests.

树结构实现ACK的聚合，降低开销。

Gossip多播

- 节点随机选择另一个节点进行消息交换；
- Eventually，消息会发布给所有节点。
- 优点：简单、高效
- 缺点：速度慢、概率性可靠

原子多播：有进程失效的情形

- 原子性
  - 消息要么交付给组中所有进程，要么不给任何进程。
- 如：副本数据库系统
  - 所有正常节点应该确保收到全部消息
- 关键问题：
  - 组视图/成员变化（宕机或进出）：虚拟同步
  - 消息顺序处理：有序无序

虚拟同步

- 消息交付与组成员变化同步
  - 消息的发送者在多播期间崩溃，那么消息要么交付给所有其他进程，要么被每个进程忽略。
  - 如果组视图发生变化（进程加入或退出），多播消息要么按变化前视图交付要么按之后的视图交付。

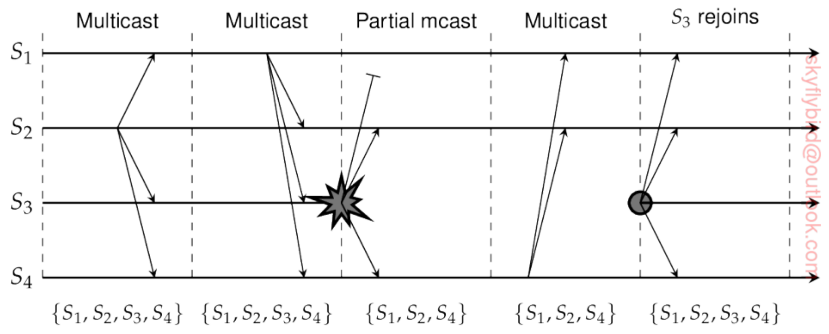


Figure 8.25: The principle of virtual synchronous multicast.

- 基本方法
  - 可靠多播协议+组成员信息（view）消息多播

(教材Note 8.11，P484)

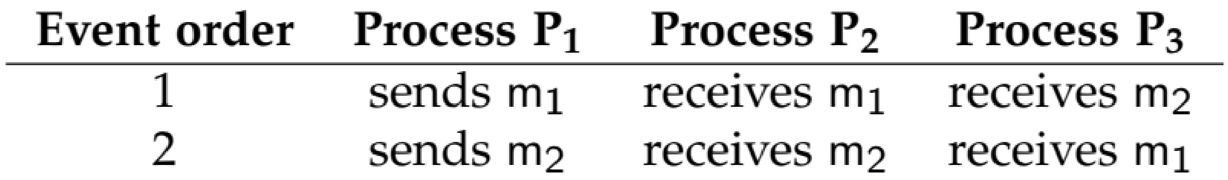
消息顺序

- 消息交付是否“遵循”某种顺序
- 单个接收者视角看
  - 遵循某种“客观存在/要求”的顺序
- 全局视角看
  - 不同接收者是否按相同的顺序
- FIFO顺序（单接收者视角）
  - 同一进程发来的消息要按“发送顺序”交付
  - 不同进程发来的消息可以按不同顺序交付

Event order	Process P_1	Process P_2	Process P_3	Process P_4
1	sends m_1	receives m_1	receives m_3	sends m_3
2	sends m_2	receives m_3	receives m_1	sends m_4
3		receives m_2	receives m_2	
4		receives m_4	receives m_4	

消息顺序

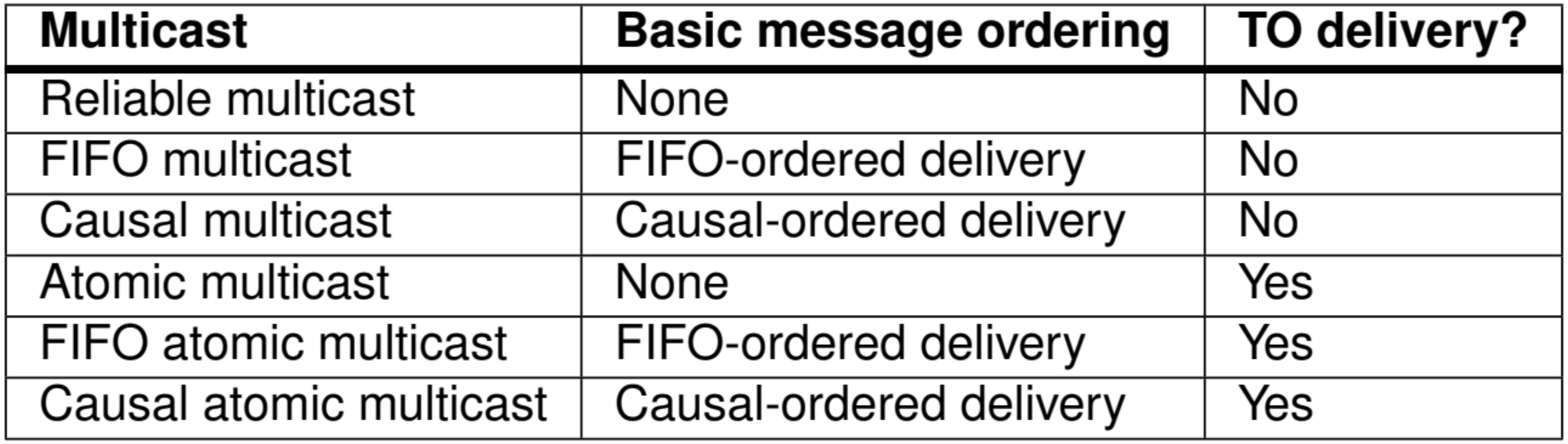
- 因果顺序（单接收者视角）：
  - 有因果关系的消息要按“因果顺序”交付
  - 可以基于Vector Clock实现
- 无序（单接收者视角）：
  - 只要求消息“可靠”交付，无顺序要求



Event order	Process P_1	Process P_2	Process P_3
1	sends m_1	receives m_1	receives m_2
2	sends m_2	receives m_2	receives m_1

消息顺序

- 完全有序（全局视角）：
  - 所有进程按相同顺序交付所有消息
- 可靠性+单进程顺序+全局顺序



Multicast	Basic message ordering	TO delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

§8.4 分布式提交

- 问题定义

  一个操作要么被进程组中的每一个成员执行，要么一个都不执行。（原子多播的更一般化抽象）

- 例子：
  - 原子多播： “操作”具体为“消息的交付”
  - 分布式事务： “操作”具体为“事务提交”

- 系统模型：
  - 一个Coordinator，多个Participants
  - 节点/进程会失效
  - 消息不丢失（如果丢失可以通过重发机制处理）

两阶段提交协议

Essence

The client who initiated the computation acts as coordinator; processes required to commit are the participants.

- Phase 1a: Coordinator sends VOTE-REQUEST to participants (also called a pre-write)
- Phase 1b: When participant receives VOTE-REQUEST it returns either VOTE-COMMIT or VOTE-ABORT to coordinator. If it sends VOTE-ABORT, it aborts its local computation
- Phase 2a: Coordinator collects all votes; if all are VOTE-COMMIT, it sends GLOBAL-COMMIT to all participants, otherwise it sends GLOBAL-ABORT
- Phase 2b: Each participant waits for GLOBAL-COMMIT or GLOBAL-ABORT and handles accordingly.

两阶段提交协议

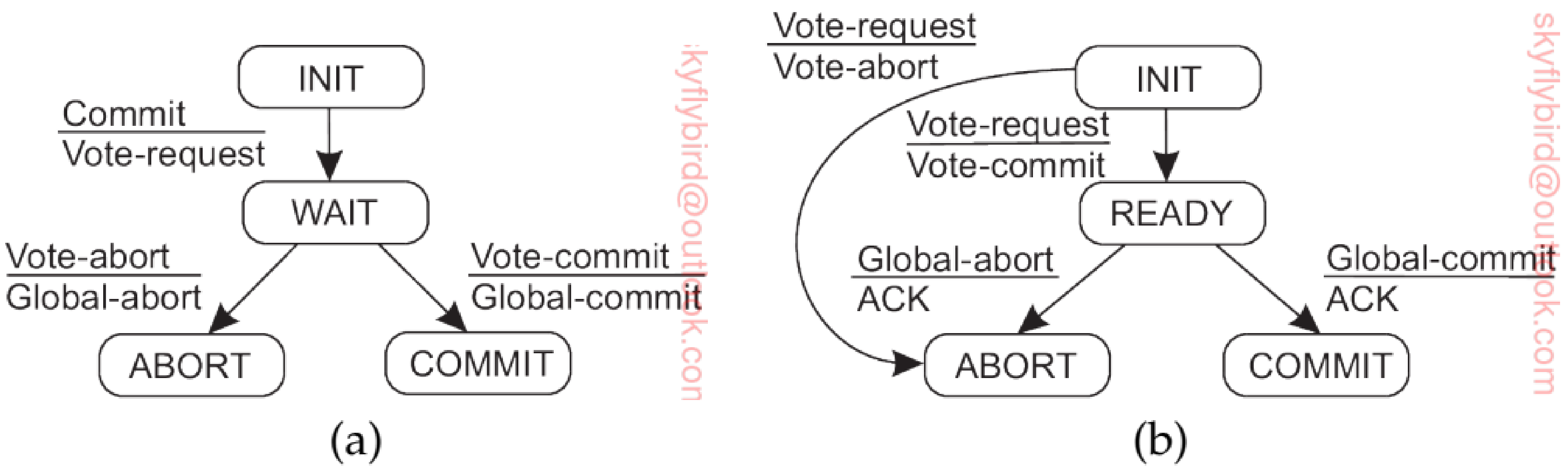

INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Global-commit
ABORT
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Global-commit / ACK
ABORT
COMMIT
(b)

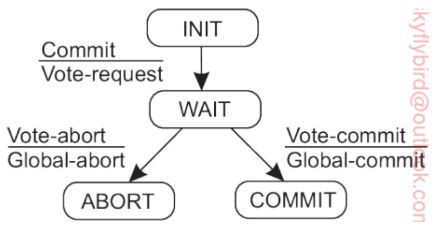
图 8.18　2PC 中协作者参与者的有限状态机

(a) 2PC 中协作者的有限状态机；(b) 参与者的有限状态机

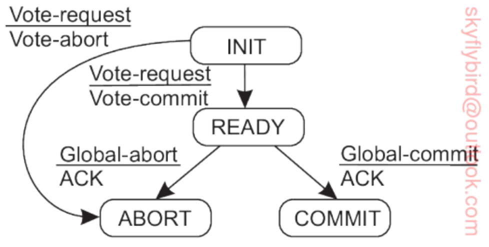
两阶段提交协议

- 失效处理：正常进程被Block后的处理（timeout后预期消息未收到）

角色	Block的状态	Block措施
P	INIT	Send Vote-abort，Abort
C	WAIT	Send Global-abort，Abort
P	READY	a) Wait until C recover, or b) Contact other participant Q to get info.: i) Q COMMIT/ABORT: do the same ii) Q INIT: Abort iii) READY: contact another, if all "REDAY": Cannot decide, wait for C to recover



(a)



(b)

如果所有的参与者都被Block在READY状态，整个协议Block。

两阶段提交协议

- 失效处理：失效进程恢复后的处理

角色	失效时的状态	Recover措施
P	INIT	Send Vote-abort，Abort
P	COMMIT/ABORT	Send Global-commit/abort，Commit/Abort
P	READY	Contact other participant Q to get info.: i) Q COMMIT/ABORT: do the same ii) Q INIT: Abort iii) READY: contact another, if all “REDAY”: Cannot decide, wait for C to recover
C	WAIT/COMMIT/ ABORT	Resend previous message

- 观察发现
  – 让参与者保存消息日志，当出现失效时能够比较方便地恢复结果。

两阶段提交协议

- 2PC可能被阻塞
  - 关键点：无法判断协作者的最后决定是否到达所有达参与者。
- 解决办法
  - 办法1：利用多播原语：消息被收到后立即多播给其他参与者；（参与者可以达成一致决定）
  - 办法2：将2PC扩展为3PC。

三阶段提交协议

- 增加预提交阶段

If all correct participants blocked in “READY”: go to Abort.

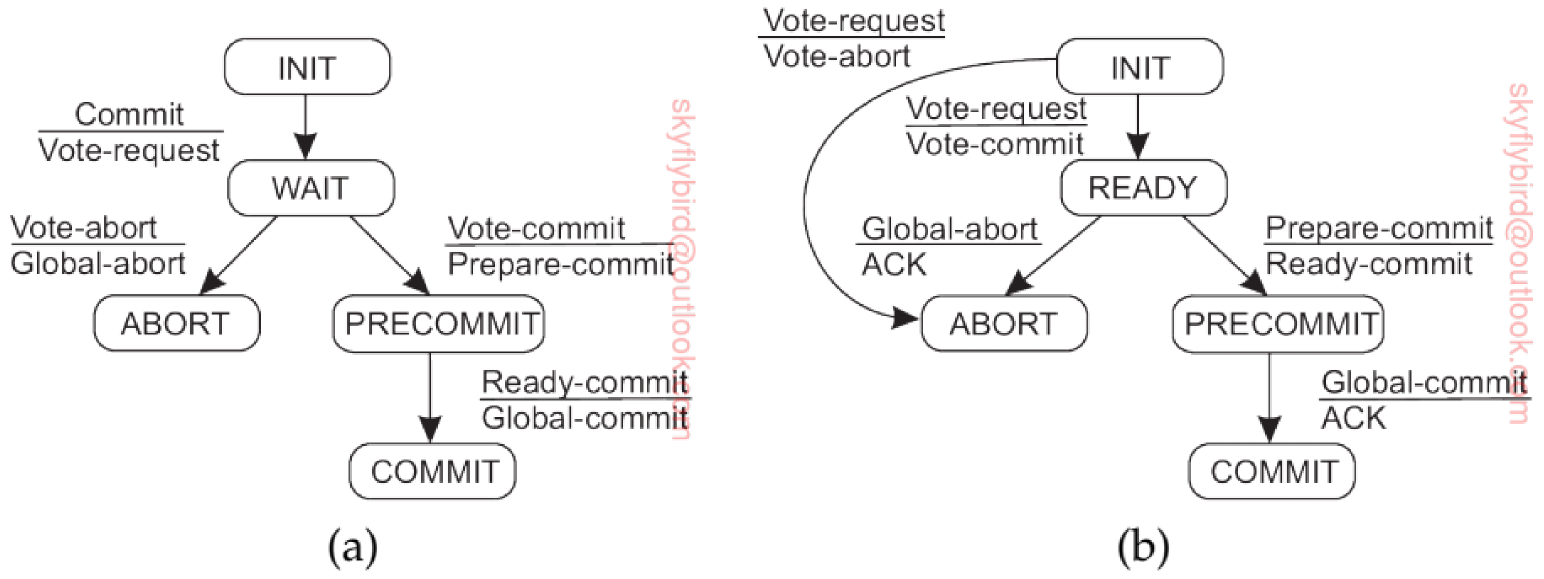

INIT
Commit / Vote-request
WAIT
Vote-abort / Global-abort
Vote-commit / Prepare-commit
ABORT
PRECOMMIT
Ready-commit / Global-commit
COMMIT
(a)
Vote-request / Vote-abort
INIT
Vote-request / Vote-commit
READY
Global-abort / ACK
Prepare-commit / Ready-commit
ABORT
PRECOMMIT
Global-commit / ACK
COMMIT
(b)
skyflybird@outlook.com

图 8.22 3PC 中协作者和参与者的有限状态机

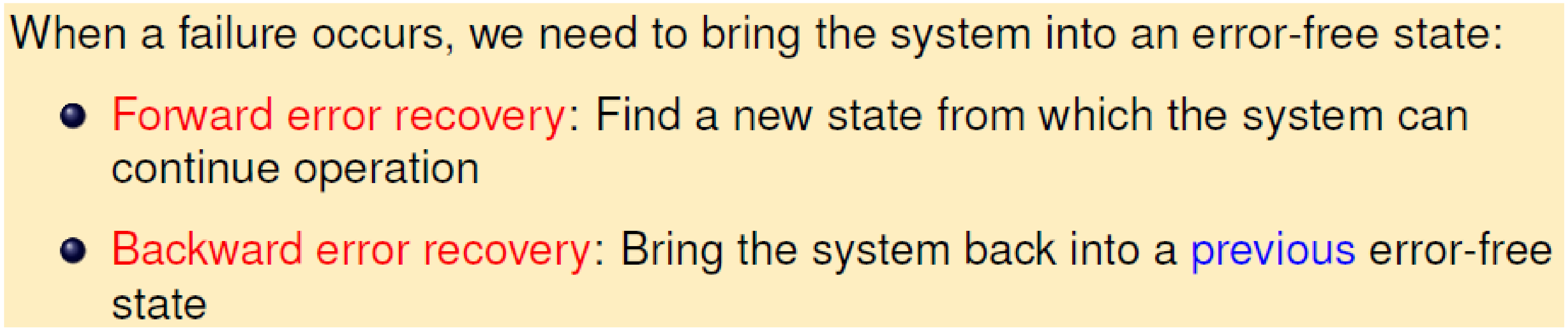
(a) 3PC 中协作者的有限状态机；(b) 参与者的有限状态机

角色	Block的状态	Block措施

P	READY	Contact other participant Q,
		if some INIT or all “REDAY”: Abort
		if all “PRECOMMIT”: Commit

§8.5 系统恢复

- 概念

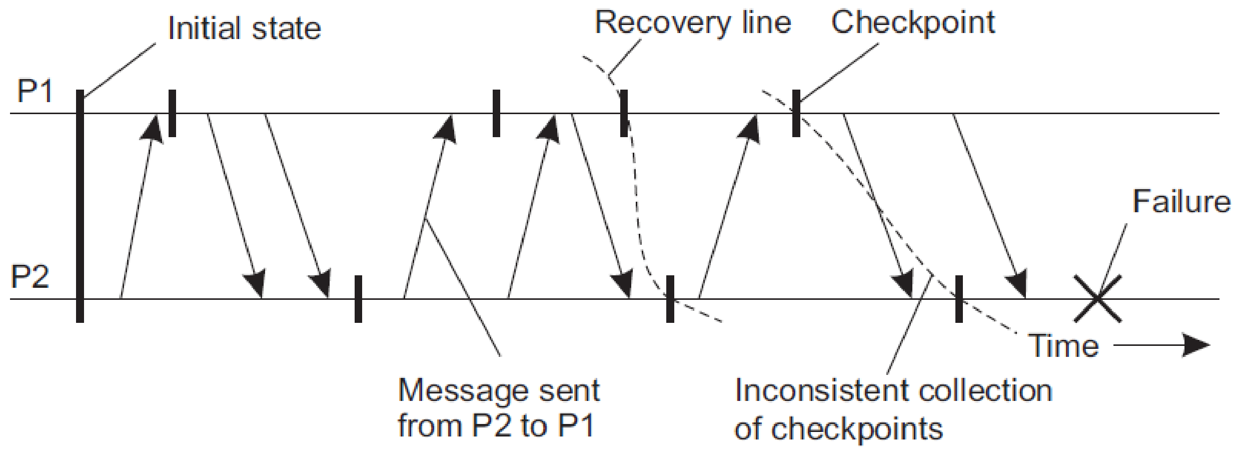

When a failure occurs, we need to bring the system into an error-free state:

- Forward error recovery: Find a new state from which the system can continue operation
- Backward error recovery: Bring the system back into a previous error-free state

- 关键点
  - 后向恢复：需要记录历史状态：checkpointing、message logging
  - 前向恢复：需要知道错误内容
- 例子
  - 后向：消息重发；前向：纠删码(Erasure code)消息

检查点Checkpointing

- 概念
  - 分布式系统的状态记录，即分布式快照（Snapshot），如消息收发。
  - 进程的本地记录构成全局快照。
- 恢复线路（Recovery line）
  - 最近一次的全局一致的检查点。


Initial state
Recovery line
Checkpoint
P1
P2
Failure
Time
Message sent from P2 to P1
Inconsistent collection of checkpoints

检查点机制：协调检查点

Essence

Each process takes a checkpoint after a globally coordinated action.

Simple solution

Use a two-phase blocking protocol:

- A coordinator multicasts a checkpoint request message
- When a participant receives such a message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
- When all checkpoints have been confirmed at the coordinator, the latter broadcasts a checkpoint done message to allow all processes to continue

Observation

It is possible to consider only those processes that depend on the recovery of the coordinator, and ignore the rest

检查点机制：独立的检查点

- 每个进程的检查点是以去中心化的方式来记录本地状态。
- 如何确定：恢复线路，即一致性的全局快照？

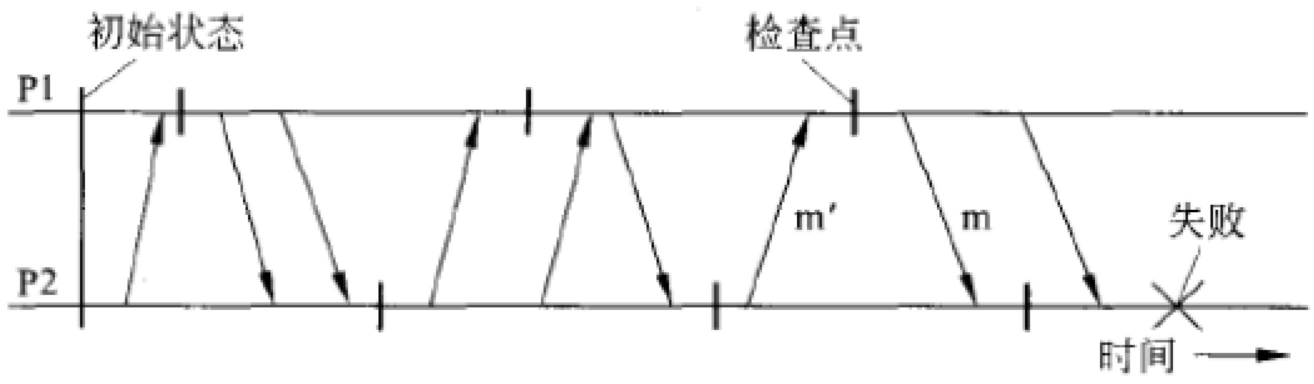


图 8.25　多米诺效应

独立的检查点

Essence

Each process independently takes checkpoints, with the risk of a cascaded rollback to system startup.

- Let $CP_i(m)$ denote m^{th} checkpoint of process P_i and $INT_i(m)$ the interval between $CP_i(m-1)$ and $CP_i(m)$.
- When process P_i sends a message in interval $INT_i(m)$, it piggybacks (i,m)
- When process P_j receives a message in interval $INT_j(n)$, it records the dependency $INT_i(m) \rightarrow INT_j(n)$.
- The dependency $INT_i(m) \rightarrow INT_j(n)$ is saved to storage when taking checkpoint $CP_j(n)$.

Observation

If process P_i rolls back to $CP_i(m-1)$, P_j must roll back to $CP_j(n-1)$.

检查点实现

- 保存进程的运行状态，存储到内存或者磁盘，用于对进程进行迁移或者故障恢复。

APP

Container Engine

OS

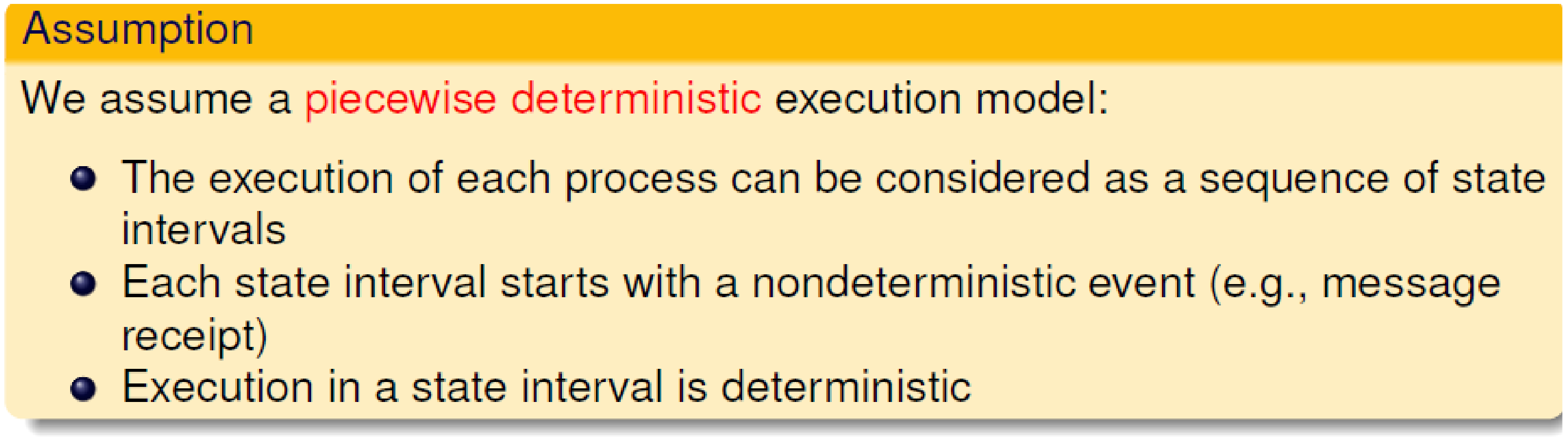
虚拟机监控器（KVM、Qemu、Xen、Vmware）

硬件

不同层次检查点

消息日志

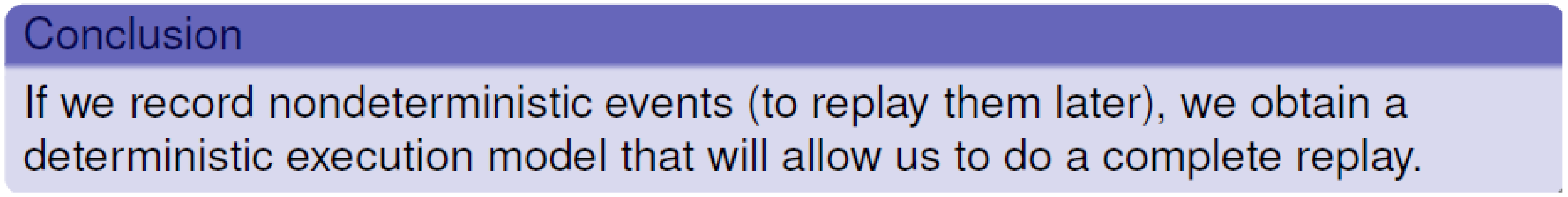
- 需求
  - 检查点的代价偏高；可以通过“重放（replay）”操作达到一致状态。
  - 方法：在日志中记录消息，用于重放。



Assumption

We assume a piecewise deterministic execution model:

- The execution of each process can be considered as a sequence of state intervals
- Each state interval starts with a nondeterministic event (e.g., message receipt)
- Execution in a state interval is deterministic



Conclusion

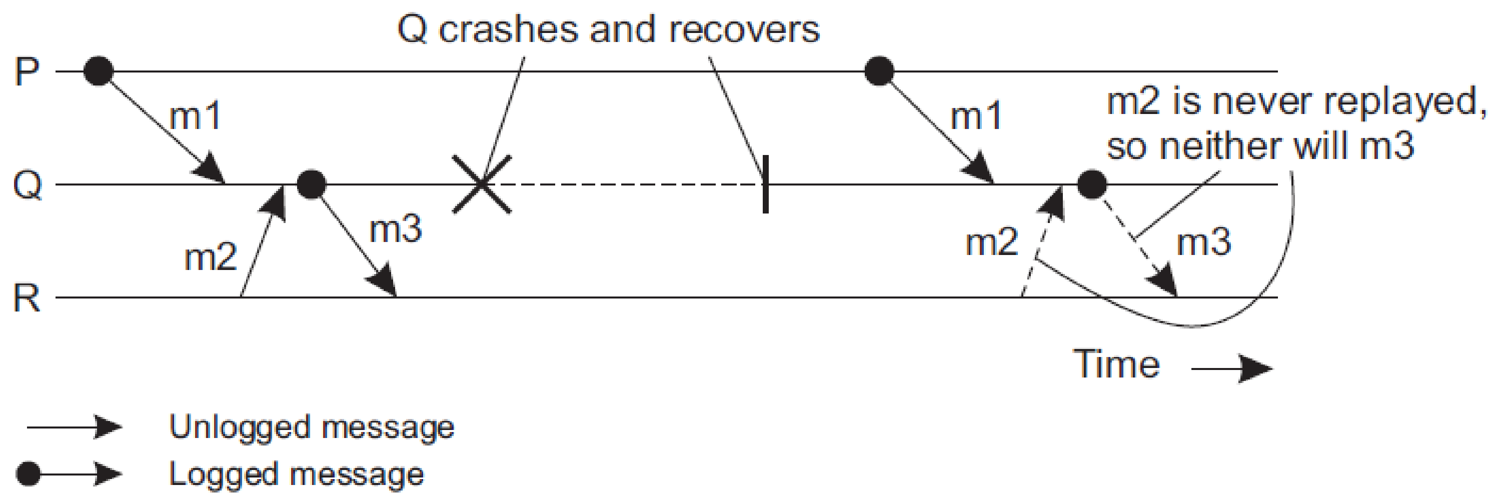
If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

消息日志一致性

When should we actually log messages?

Avoid orphan processes:

- Process Q has just received and delivered messages m_1 and m_2
- Assume that m_2 is never logged.
- After delivering m_1 and m_2, Q sends message m_3 to process R
- Process R receives and subsequently delivers m_3: it is an orphan.



消息记录方式

Notations

- **DEP**(m): processes to which m has been delivered. If message m^* is causally dependent on the delivery of m, and m^* has been delivered to Q, then $Q \in$ **DEP**(m).
- **COPY**(m): processes that have a copy of m, but have not (yet) reliably stored it.
- **FAIL**: the collection of crashed processes.

Characterization

Q is orphaned $\Leftrightarrow \exists m : Q \in$ **DEP**(m) and **COPY**(m) $\subseteq$ **FAIL**

消息记录方式

Pessimistic protocol

For each nonstable message m, there is at most one process dependent on m, that is $|\mathbf{DEP}(m)| \leq 1$.

Consequence

An unstable message in a pessimistic protocol must be made stable before sending a next message.

Optimistic protocol

For each unstable message m, we ensure that if $\mathbf{COPY}(m) \subseteq \mathbf{FAIL}$, then eventually also $\mathbf{DEP}(m) \subseteq \mathbf{FAIL}$.

Consequence

To guarantee that $\mathbf{DEP}(m) \subseteq \mathbf{FAIL}$, we generally rollback each orphan process Q until $Q \notin \mathbf{DEP}(m)$.

面向恢复的计算（RoC）

- UC Berkeley/Stanford 提出的一种计算模式;
- 只需要重启一部分系统;
- 构成系统的组件尽可能分离，组件和组件之间的关联尽可能简单;
- 可以使用检查点和恢复技术而不影响系统的继续运行。

Summary

- 容错：分布式系统部分失效后，仍然可以正确（不是正常）运行
- 失效模型：Crash、Omission、Timing、Response、Byzantine
- 冗余容错（最基本方法）：容错进程组
- 共识Consensus
  - 系统模型假设：同步、异步；故障模型：Crash、Byzantine
  - 共识协议：Paxos、BFT等
- 可靠C-S通信
  - 可靠RPC
  - 可靠组通信（多播）：Scalability、Atomicity、Group membership
- 系统恢复
  - 检查点机制、消息日志

Homework Questions

1. Can the model of triple modular redundancy (三倍模块冗余）handle Byzantine failures? Why?
2. In reliable multicasting, is it always necessary that the communication layer keeps a copy of a message for retransmission purposes?
3. In the two-phase commit protocol, why can blocking never be completely eliminated, even when the participants elect a new coordinator?
4. Does a stateless server need to take checkpoints?

谢谢！

wuweig@mail.sysu.edu.cn