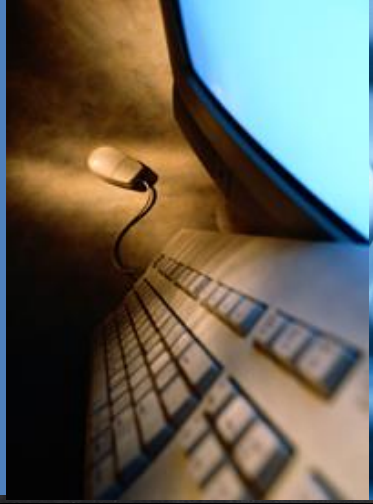
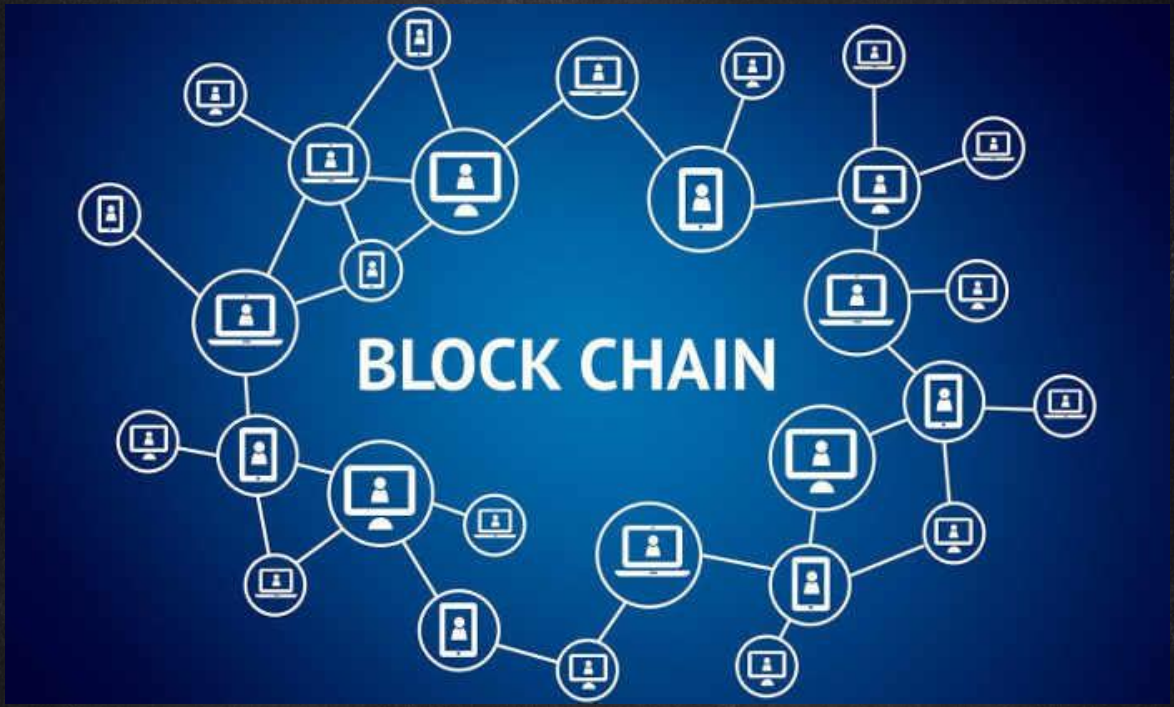



分布式系统
Distributed Systems

陈鹏飞
数据科学与计算机学院
chchenpf7@mail.sysu.edu.cn
办公室：超算5楼529d
主页：http://sdcs.sysu.edu.cn/node/3747

第八讲 — 容错

背景



分布式系统的设计目标之一是允许部分失效。

主要内容

1. 基本概念
2. 进程恢复
3. 可靠的通信方式
4. 分布式提交
3. 系统恢复



1 基本概念

可依赖性（dependability）

➢ 含义

软件组件为用户提供服务。为了提供服务，组件可能需要来自其他组件的服务（服务依赖），即组件可能依赖于其他组件。那么组件是否可依赖？

➢ 与可依赖性相关的需求

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) VS 可用性（Availability）

➢ 组件 C 的可靠性 R(t)

- 组件 C 从开始时刻 T = 0其，经历[0, t)一段事件仍然能够提供正常功能的条件概率;

➢ 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

可靠性(Reliability) VS 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 $A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTR+MTTF}$

➢ 观察发现

- 可靠性和可用性只有在定义清楚什么是失效时才有意义；

相关概念

- 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	Part of a component that can lead to a failure	Programming bug
Fault	Cause of an error	Sloppy programmer

相关概念

- 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

故障分类

- 暂时故障

  只发生一次，不会重复发生；

- 间歇故障

  反复周期性发生；

- 持久故障

  故障被修复前持久存在的故障；

失效模型

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

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

注意，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；

停止性失效（Halting Failures）

➢ 场景

- C 不能感知来自 C*的任何行为 --- 是否发生停止性失效？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 → cannot reliably detect crash failures.
- Synchronous system: process execution speeds and message delivery times are bounded → 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 → can normally reliably detect crash failures.

停止性失效（Halting Failures）

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

➢ 冗余的类型

- 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。

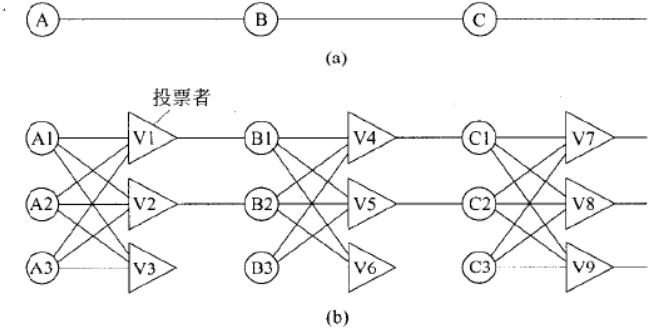
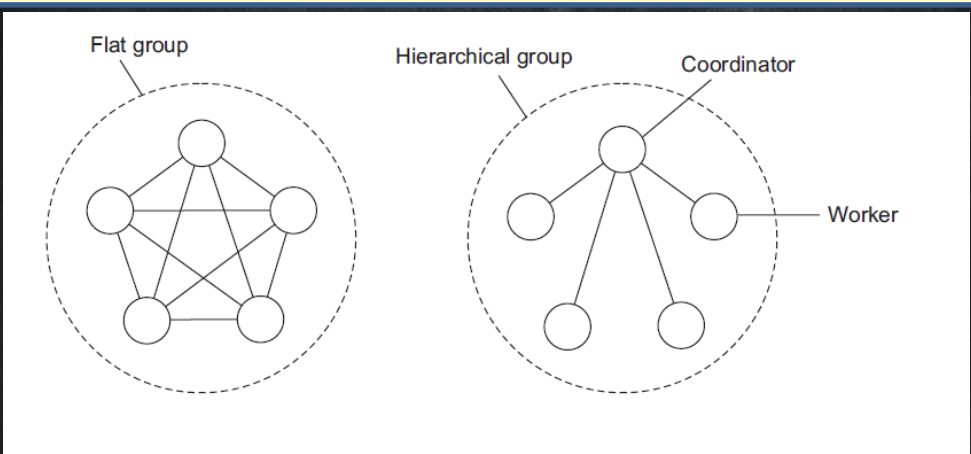


图 8.2　三倍模块冗余

进程恢复

➢ 基本思想

通过进程复制来防止进程出现问题，将多个进程放在一个进程组里面。这里需要区分平等组（flat group）和分级组（hierarchical group）。



组和失效屏蔽

➢ K-容错组

- 当一个组可以屏蔽任何 K 个并发的成员失效（K称为容错度）；

➢ K-容错组的大小

- 如果发生的是停止失效（crash/omission），我们需要 K+1 成员数保证结果的有效输出；
- 但是如果发生的随意失效（拜占庭失效），我们需要 2k + 1个成员才能保证结果的正确输出；

➢ 重要假设

- 所有的成员都是同质的；
- 所有成员以相同的顺序处理命令；

共识（Consensus）

➢ 定义

- 使所有非故障进程就由谁来执行命令达成一致，而且在有限的步骤内就达成一致。

➢ 前提

- 在一个容错组里面，每一个非故障进程执行的命令以及执行的顺序与其他非故障进程相同；

能够达成共识的分布式协定

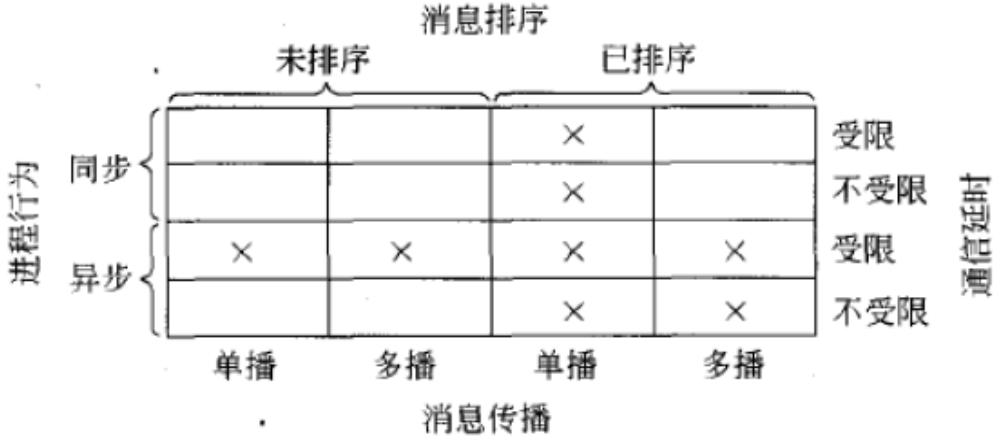


图 8.4 分布式协定下能够达到的环境

基于泛洪的共识

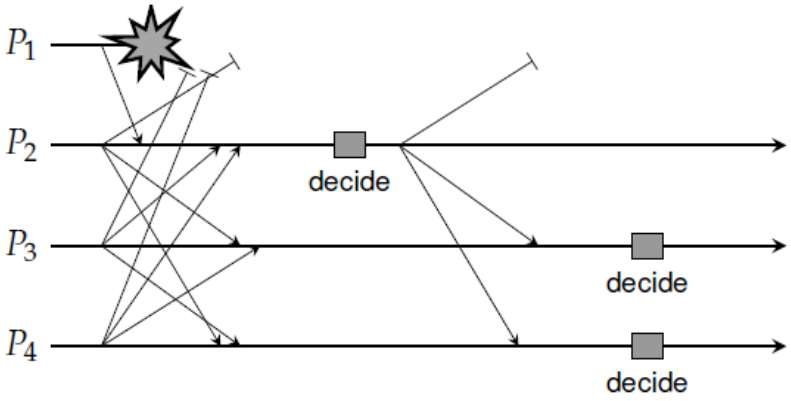
➢ 系统模型

- 给定一个进程组 P={P1, P2, …, Pn}；
- Fail-stop类型的失效语义，也就是可以有准确到系统失效；
- 一个客户端联系 Pi 让其执行命令；
- 每一个进程 Pi 维护一个发出命令的列表；

➢ 基本算法（基于轮数）

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



- 观察发现

- P2 获得所有进程提交的命令 => 作出决定；
- P3 可能已经检测到 P1 失效，但是不知道 P2 是否也检测到了失效，也就是说 P3 不知道自己的信息是否与P2一致 => 不做决定；

Paxos

下次讲解

一致性，可用性，分区容错性

➢ CAP

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.

➢ 结论

- 在会出现通信失效的网络环境下，实现一个具有原子读写的共享内存同时又保证每一个请求都得到响应是非常困难的；

故障检测

➢ 问题

我们如何可靠地检测一个进程是否实际存在宕机？

➢ 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

➢ 对于一个异步系统的观察

一次可疑的crash = crash

实际的失效检测

➢ 实现

- 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;



3 可靠的通信方式

可靠的远程过程调用

➢ 错误类型

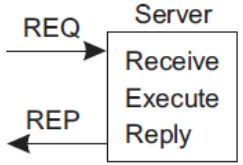
1. The client is unable to locate the server.
2. The request message from the client to the server is lost.
3. The server crashes after receiving a request.
4. The reply message from the server to the client is lost.
5. The client crashes after sending a request.

➢ 两个简单的解决方案

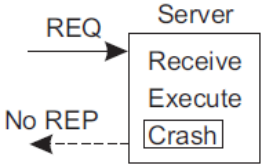
1：(不能定位服务器)：只是将错误信息反馈给客户端

2: （请求丢失）：重新传输请求；

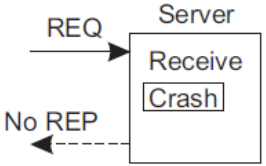
可靠的RPC：服务器宕机



(a)



(b)



(c)

➢ 问题

Where (a) is the normal case, situations (b) and (c) require different solutions. However, we don't know what happened. Two approaches:

- At-least-once-semantics: The server guarantees it will carry out an operation at least once, no matter what.
- At-most-once-semantics: The server guarantees it will carry out an operation at most once.

完全透明的服务器恢复是不可能的

- **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

完全透明的服务器恢复是不可能的

	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事件顺序下正确工作

可靠的RPC：丢失应答信息

➢ 存在的问题

客户端感知到的是没有接收到应答。但是他不能断定原因是丢失请求，服务器宕机还是丢失响应。

➢ 部分解决方案

- 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；

可靠的RPC：客户端崩溃

➢ 问题

- 服务器还在工作，并且持有资源但是没有客户端需要结果（称为孤儿计算）；

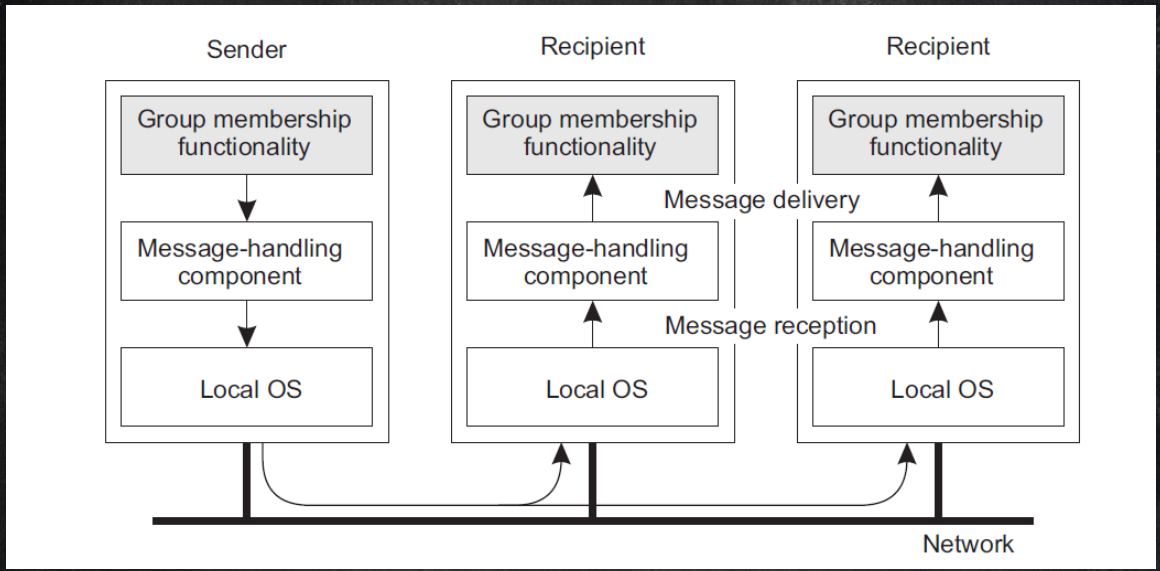
➢ 解决方案

- **孤儿消灭（extermination）**：当客户端恢复时杀死孤儿；
- **再生**：把时间分为顺序编号的时期，客户端恢复后，向所有的服务器广播新时期的开始，由服务器杀死与客户端相关的“孤儿”；
- **优雅再生**：服务器检查远程调用，如果找不到拥有者则杀死计算；
- **到期**：每个RPC都被给定一个标准的时间量 T 来进行工作。抛弃原有的过时的计算。

简单可靠的组通信

➢ 直观理解

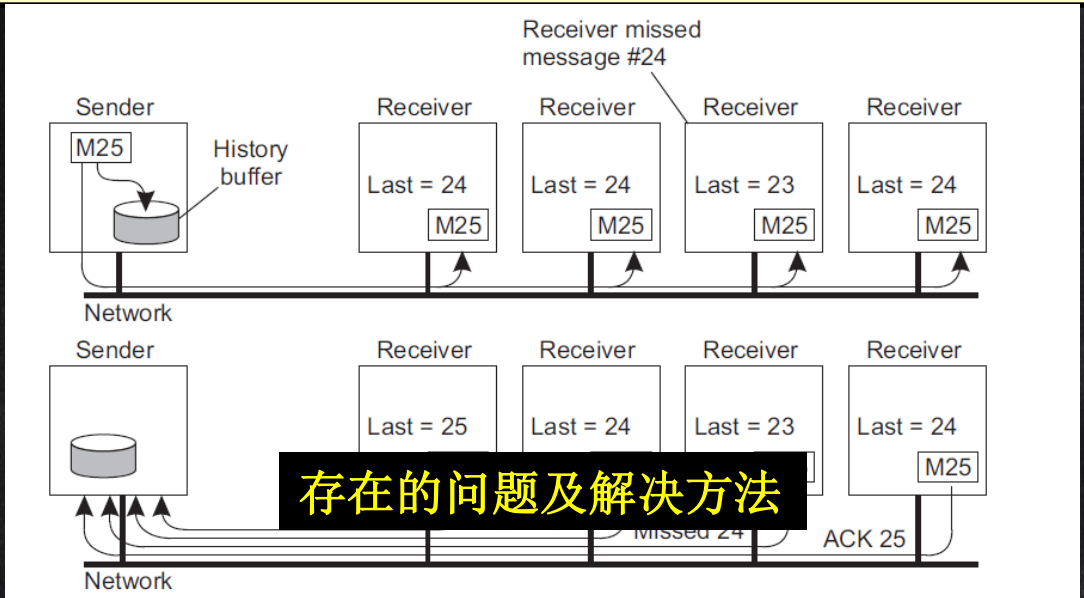
- 一个消息可靠地发送到一个进程组 G, 使 G 中的每个进程都能够接收到消息。 重要的是区分接收和发送消息；



简单可靠的组通信

- 可靠通信，但是假定存在故障进程

可靠组通信变为可靠多播：消息发送和接收按照发送者发送的顺序进行；



存在的问题及解决方法

可靠多播的可扩展性

➢ 存在的问题

- 如果存在N的接收方，会导致大量返回 N 个确认信息，如果数量很大，发送方会被反馈消息淹没，形成反馈拥塞。

➢ 简单方案

- 接收方不对消息进行反馈，而只是在消息丢失时才返回一个反馈消息；
- 存在的问题是：需要缓存大量的陈旧的信息；

无等级的反馈控制

➢ 本质

当接收方发现丢失一条消息时，向组中的其他成员多播它的反馈。多播反馈可以使组中的其他成员抑制自己的反馈，只要有一个重发请求到达发送者就足够了。

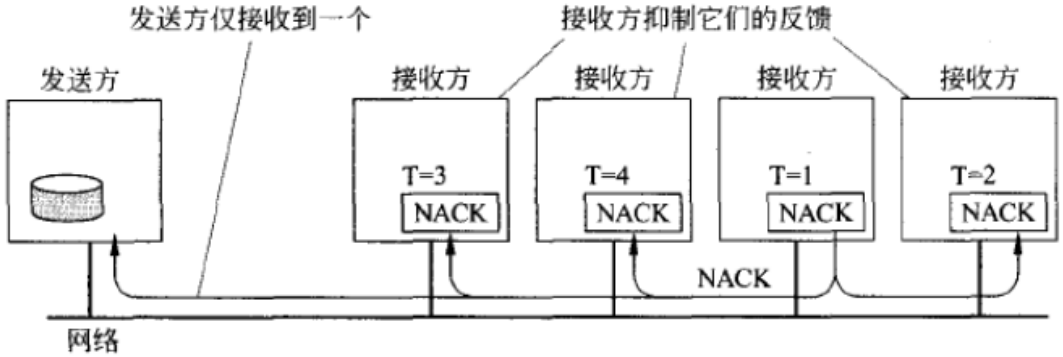


图 8.10　几个接收方要发送重发请求，但是第一个重发请求抑制了其他的请求

分等级的反馈控制

➢ 原理

在非常大的接受组中获得可扩展性，需要采用分等级的方法。接收方分为很多子组，组织成树的形式，包含发送方的子组构成了树的根。

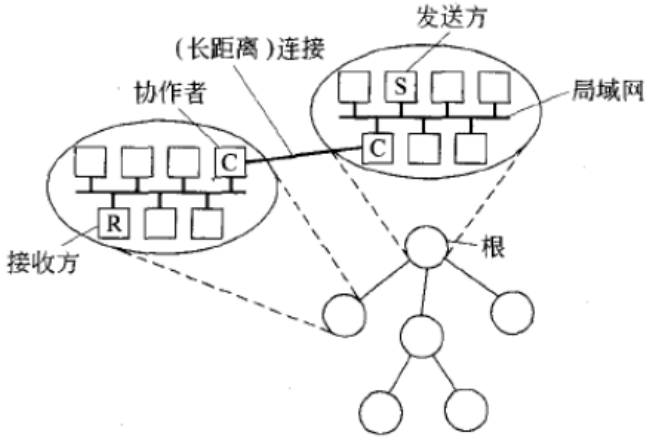


图 8.11 分等级的可靠多播。每个本地协作者都把消息转发给它的孩子然后再处理重发请求

原子多播

➢ 原理

分布式系统中的消息要么发送给所有进程，要么不向任何进程发送。而且需要所有的消息都按照相同的顺序发送给所有的进程。

➢ 虚拟同步

如果消息的发送方在多播期间崩溃，那么消息要么投递给剩余的进程，要么被每个进程忽略，具有这种属性的可靠多播被称为虚拟同步。

➢ 消息排序

将消息排序，分为：不排序多播、FIFO顺序的多播，按因果关系排序的多播和全序多播。

分布式提交协议

- **问题**

  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；

  - 可靠多播： 一个消息传递到所有的参与者；
  - 分布式事务处理：每一个局部事务都必须成功；

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：参与者失效

- **分析：参与者在状态 S 失效，并且恢复到状态 S**

  - INIT：没有问题：参与者还不知道发送的信息；
  - READY：参与者正在等待提交或者终止的信息。恢复后，参与者需要知道它要采取的动作 => 利用日志记录协作者的决定。
  - ABORT: 此时状态是幂等的，重新执行一遍；
  - COMMIT: 此时状态也是幂等的，重新执行一遍；

- **观察发现**

当需要分布式提交时，让参与者使用临时的工作空间保存结果，这样当出现失效时能够比较方便地恢复结果；

两阶段提交协议：参与者失效

- 另一种情况

当参与者需要恢复到READY状态时，检查其他参与者的状态 => 没有必要再去记录协作者的决策；

- 当参与者**P**在恢复时需要联系另外一个协作者**Q**的情况

State of *Q*	Action by *P*
COMMIT	Make transition to *COMMIT*
ABORT	Make transition to *ABORT*
INIT	Make transition to *ABORT*
READY	Contact another participant

如果所有的参与者都处于**READY**状态，整个协议会阻塞。很明显，这说明协作者失效了。

两阶段提交协议：协作者失效

➢ 观察

问题本质是协作者的最后决定不能到达参与者或者丢失了。

➢ 恢复方案

- 开始2PC时，记录进入WAIT状态，恢复之后重新发送VOTE_REQUEST;
- 在第二阶段做出决定后，只要记录表决结果就可以了，恢复时重发这个决定；

协作者失效，整个协议执行过程有可能阻塞，因此2PC称为阻塞协议。

Coordinator in Python

```
1  class Coordinator:
2
3    def run(self):
4      yetToReceive = list(participants)
5      self.log.info('WAIT')
6      self.chan.sendTo(participants, VOTE_REQUEST)
7      while len(yetToReceive) > 0:
8        msg = self.chan.recvFrom(participants, TIMEOUT)
9        if (not msg) or (msg[1] == VOTE_ABORT):
10         self.log.info('ABORT')
11         self.chan.sendTo(participants, GLOBAL_ABORT)
12         return
13       else: # msg[1] == VOTE_COMMIT
14         yetToReceive.remove(msg[0])
15     self.log.info('COMMIT')
16     self.chan.sendTo(participants, GLOBAL_COMMIT)
```

Participant in Python

```
class Participant:
  def run(self):
    msg = self.chan.recvFrom(coordinator, TIMEOUT)
    if (not msg):  # Crashed coordinator - give up entirely
      decision = LOCAL_ABORT
    else: # Coordinator will have sent VOTE_REQUEST
      decision = self.do_work()
      if decision == LOCAL_ABORT:
        self.chan.sendTo(coordinator, VOTE_ABORT)
      else: # Ready to commit, enter READY state
        self.chan.sendTo(coordinator, VOTE_COMMIT)
        msg = self.chan.recvFrom(coordinator, TIMEOUT)
        if (not msg): # Crashed coordinator - check the others
          self.chan.sendTo(all_participants, NEED_DECISION)
          while True:
            msg = self.chan.recvFromAny()
            if msg[1] in [GLOBAL_COMMIT, GLOBAL_ABORT, LOCAL_ABORT]:
              decision = msg[1]
              break
        else: # Coordinator came to a decision
          decision = msg[1]

    while True: # Help any other participant when coordinator crashed
      msg = self.chan.recvFrom(all_participants)
      if msg[1] == NEED_DECISION:
        self.chan.sendTo([msg[0]], decision)
```

三阶段提交协议

➢ 动机

两节点提交的一个问题在于当协作者崩溃时，参与者不能做出最后的决定，处于阻塞状态，为了解决这一问题，提出了三阶段提交协议。

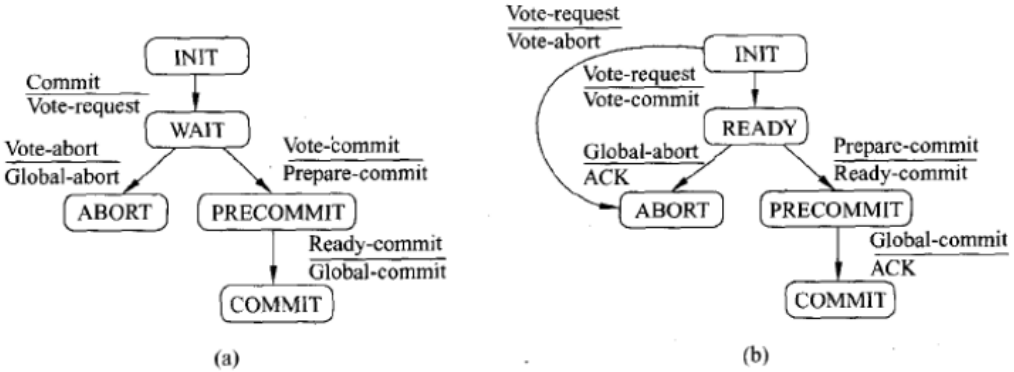


图 8.22 3PC 中协作者和参与者的有限状态机

(a) 3PC 中协作者的有限状态机；(b) 参与者的有限状态机

恢复：背景

➢ 本质

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

➢ 实践

利用后向恢复，需要我们设置恢复点（recovery point）

➢ 难点

分布式系统中的恢复更加复杂，因为需要多个进程协作找出通过恢复可以达到一致的状态的地方；

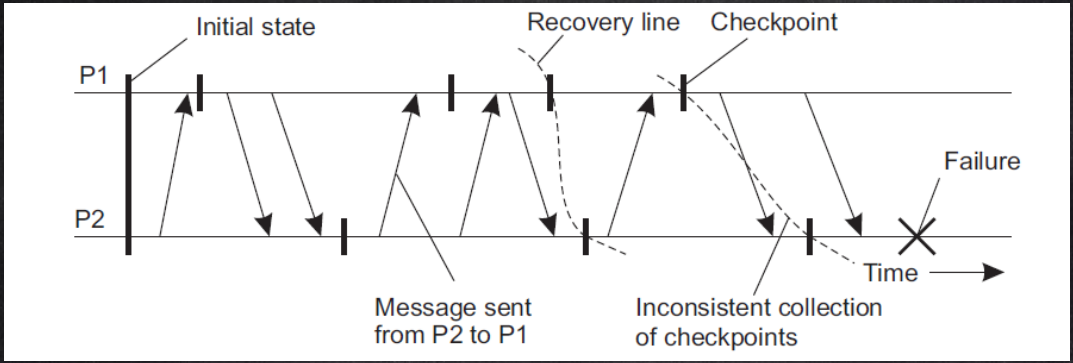
一致的恢复状态

- **需求**

接收的每一个消息都应该有一个发送者记录；

- **恢复线路（Recovery line）**

假定每一个进程都会周期性记录检查点，最近一次的全局一致的检查点就是恢复线路；



独立的检查点

每个进程的检查点是以一种不协调的方式来按时记录本地状态，这种分布式特性使得找到一个恢复线路非常困难，可能会导致多米诺效应。

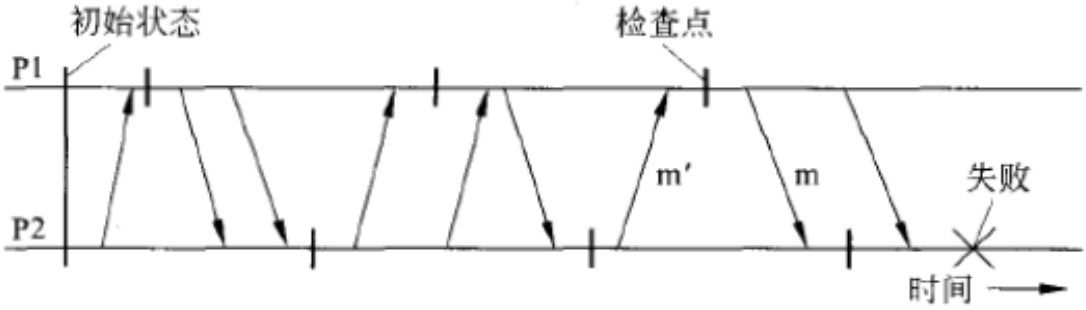


图 8.25 多米诺效应

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 $CP_j(n-1)$.

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

日志消息

- 后备选择

检查点的代价过高，通过“重放（replay）”的方式达到一个全局一致的状态而不需要从持久存储中恢复该状态=>在日志中持久化消息；

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

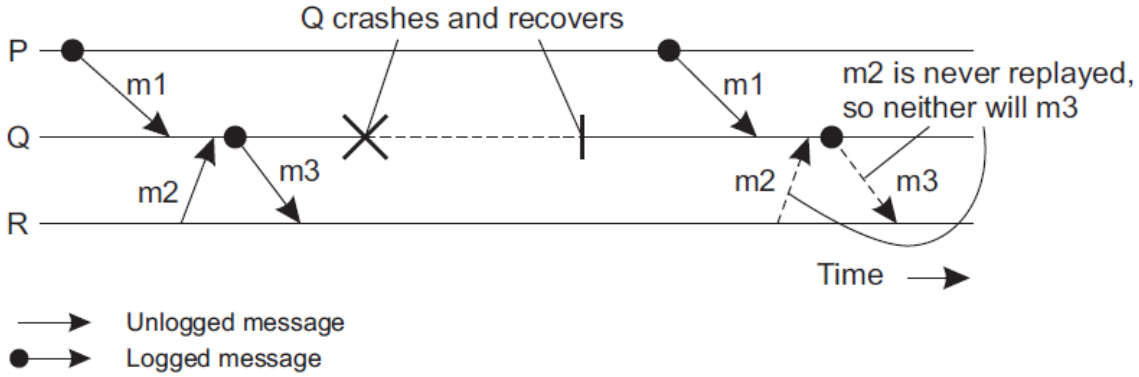
If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

消息记录和一致性

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.



消息记录的模式

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

$$Q \text{ is orphaned} \Leftrightarrow \exists m : Q \in \textbf{DEP}(m) \text{ and } \textbf{COPY}(m) \subseteq \textbf{FAIL}$$

消息记录的模式

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；



谢谢！