Fault is common: fault, error, failure

Fault can be latent or active

- if active, get wrong data or control signals

Error is the results of active fault

- e.g. violation of assertion or invariant of spec
- discovery of errors is ad hoc (formal specification?)

Failure happens if an error is not detected and masked

- not producing the intended result at an interface

distribute-system

Metrics to measure reliability

- MTTF: mean time to failure
- MTTR: mean time to repair
- MTBF: mean time between failure
- **MTBF = MTTF + MTTR**

cap

c:consistence:一致性

a:available：可用性

p:partition tolerance：分区容错性

file system

kernel 在文件系统启动时读取 superblock

superblock 有

1. **块大小**：文件系统中每个块的大小。
2. **空闲块计数**：文件系统中空闲块的数量。
3. **空闲 inode 计数**：文件系统中空闲 inode 的数量。
4. **总 inode 数量**：文件系统中 inode 的总数量

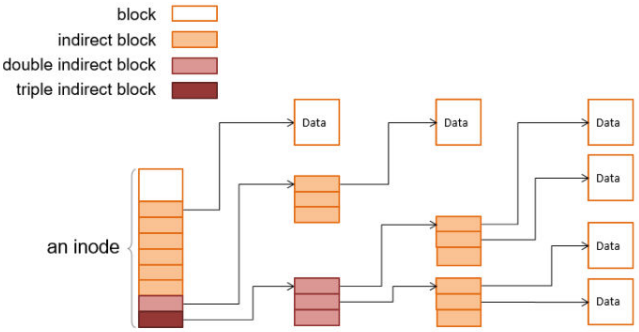
block layer

记录 free block： bitmap

file layer

inode:存储文件的元数据：

```
struct inode
    int blick_nums[N]
    int size
    int type
    int refcnt //for linking
    int userid
    int groupid
    int mode    //权限
    int atime    //最近获取
    int mtime    //最近修改
    int ctime    //创建时间
```



inode number layer

inode table： 位于存储的固定位置

inode num 是 inode table 的 index

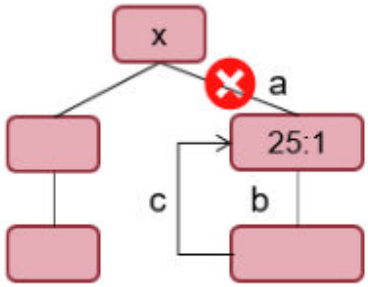
file name layer

mapping between name and inode num

path name layer

link

link 不能有环



renaming:

```
1 UNLINK(to_name)
2 LINK(from_name, to_name)
3 UNLINK(frome_name)
```

What if the computer **fails between 1 & 2**?

- The file `to_name` will be lost, which will surprise the use
- Need **atomic** action (in later lectures)

为了防止第一二步操作时断电产生错误，故须保证原子操作，如下：

Renaming - 2

Path name	File name	Inode num	File (inode)	Block num	Disk Block

```
1 LINK(from_name, to_name)
2 UNLINK(frome_name)
```

Weaker specification without atomic actions

- 1. Changes the inode number in for `to_name` to the inode number of `from_name`
- 2. Removes the directory entry for `from_name`

If fails between 1 & 2

- Must increase reference count of `from_name`'s inode on recovery

If `to_name` already exists

- It will always exist even if the machine fails between 1 & 2

27

absolute path name layer

Why File Descriptor?

Other options

- Option-1: OS returns an inode pointer
- Option-2: OS returns all the block numbers of the file

Reasons and considerations

- Security: user can never access kernel's data structure
- Non-bypassability: all file operations are done by the kernel
  - Aka., *complete mediation*

When writing, which **order** is preferred?

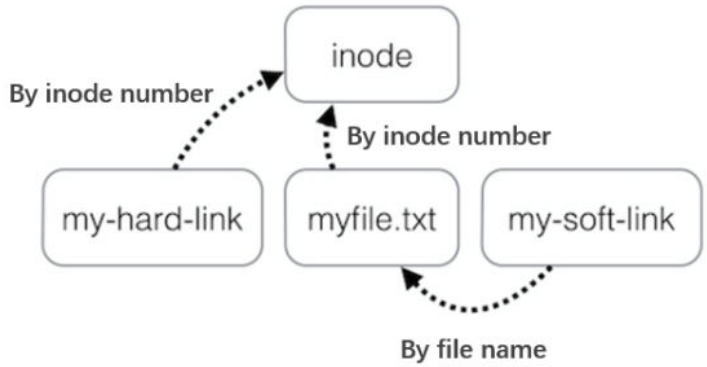
- Update block bitmap, write new data, update inode (size and pointer)
- Update block bitmap, update inode (size and pointer), write new data
- Update inode (size and pointer), update block bitmap, write new data

第一种更好，后两个可能会引起数据泄露（读到旧数据），第一个可以进行全盘扫描恢复block

Symbolic link layer

soft link and hard link:

- `open()` returns an `fd`; `fopen()` return a `FILE*`
- `open()` is a system call of OS; `fopen()` is an API of `libc`




```
hard link:
A_inode_id = lookup(1, "A")
B_inode_id = lookup(1, "B")
a_inode_id = lookup(A_inode_id, "a")
add_dentry(B_inode_id, "a-hard-link", a_inode_id)
symbolic link:
symlink_inode_id = allocate_inode(SYMLINK)
write_file(symlink_inode_id, "/A/a")
add_dentry(B_inode_id, "a-symbolic-link", symlink_inode_id)
```

要点

- 文件名不是文件的一部分
- 目录很小（相对文件），因此叫文件夹是不妥当的
- hard link 间没有差别

fs-api

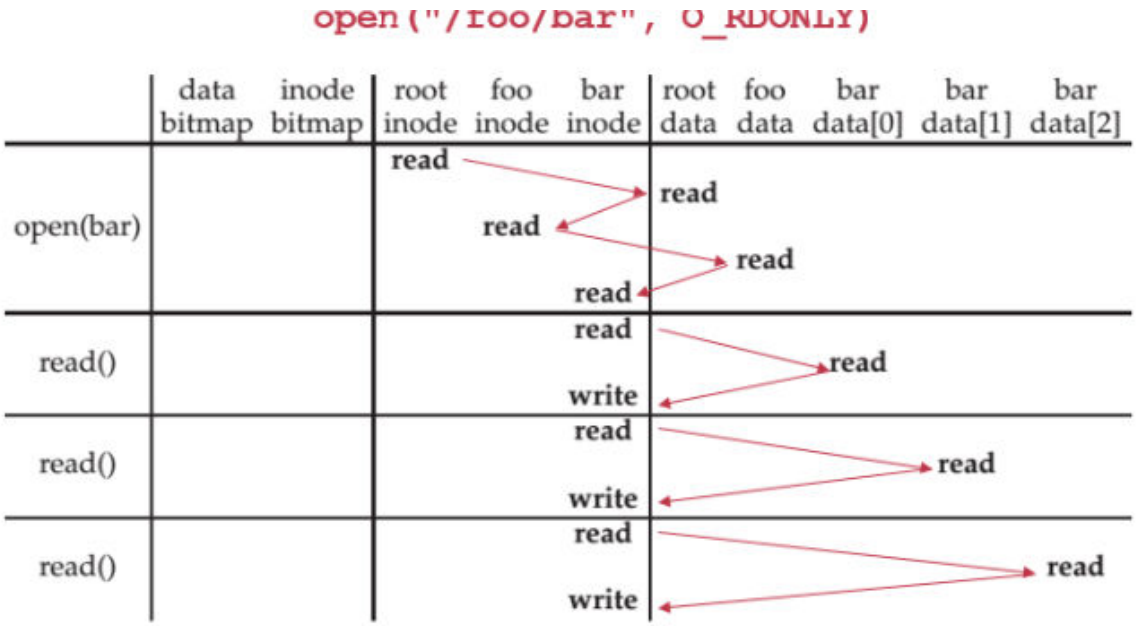
open 和 fopen:**fopen is better**

- `open()` returns an `fd`; `fopen()` return a `FILE*`
- `open()` is a system call of OS; `fopen()` is an API of `libc`

Which one has better performance?

- `fopen()` provides you with buffering I/O that may turn out to be a lot faster than what you're doing with `open()`

文件读写过程



write 是为了修改 atime

File Creation Timeline

处理复杂问题

M: modularity

A: abstract

L: layering

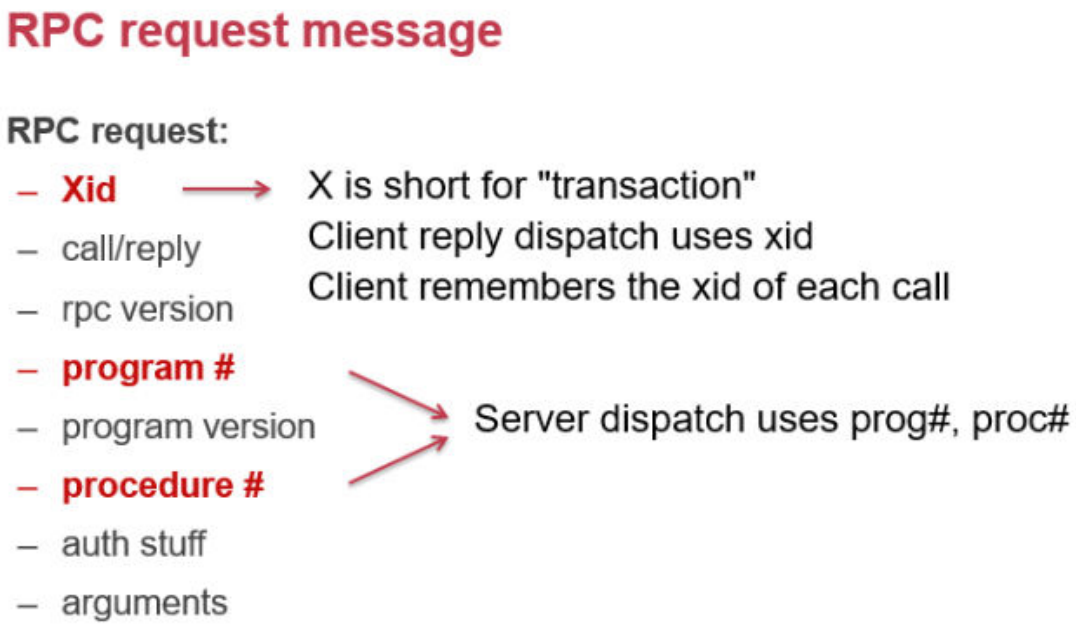
H: hierarchy

rpc

为了使得分布式计算更像集中式应用

使程序能够在不同的地址空间（通常是不同的计算机）上执行代码，就像在本地调用函数一样。RPC 的主要目的是简化分布式计算，使得开发者可以更容易地构建跨网络的应用程序

一个 rpc 信息可能有：service id(function id),参数，using marshal/unmarshal(把内存的结构转换为一串字符流，即序列化)



Xid:表示事务的 id

rpc version:版本

program #:

procedure #:确定调用哪个二进制的哪个函数

auth stuff:权限

RPC reply message

RPC reply:

- **Xid**
- call/reply
- **accepted**? (Yes, or No due to bad RPC version, auth failure, etc.)
- auth stuff
- **success**? (Yes, or No due to bad prog/proc #, etc.)
- **results**

连接举例：

Binding: find the server

Can implement with other network name services

- E.g., 192.168.10.233:8888 + function ID

Example: gRPC

gRPC client

```
grpc::CreateChannel("localhost:50051",
    grpc::InsecureChannelCredentials());
```

gRPC server

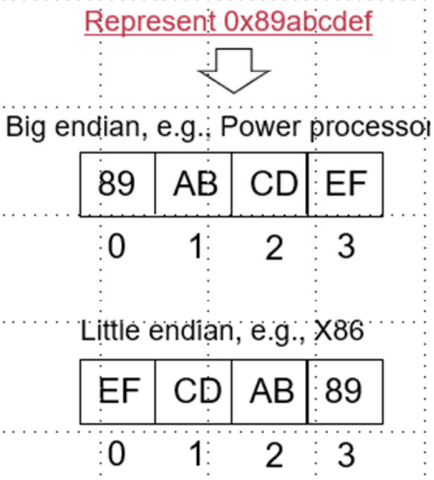
```
std::stringserver_address("0.0.0.0:50051");
...
... AddListeningPort(server_address, ...);
```

rpc 传参有很多挑战：

Parameter passing is challenging across machines

Distributed systems have the **incompatibility** problem
- which does not exist on a single machine

For example, remote machine may have **different**
- byte ordering,
- sizes of integers and other types,
- floating point representations,
- character sets,
- alignment requirements
- etc.



- 远端机器可能有不同的:
  - byte ordering
  - int 等类型字节大小
  - 浮点表示
- 服务器和客户端 rpc 版本或函数版本差异

因此，我们的应用要有**向前兼容和向后兼容(Backward compatibility)**

传参要保证：**正确性，紧凑，**

编码解码要求

- 能通过网络传输，正确编码解码
- 兼容性（多语言，多版本）
- 高效

Why not using language-specific formats

Drawbacks:
- The encoding is tied to a particular programming language
  - E.g., it's challenging to use a Java client to call a python server
- No versioning -> no forward and backward compatibility

传输方式	优点	缺点
文本（json 等）	容易 debug	二义性（数字和字符），慢
二进制	无二义性(Ambiguity)，紧凑	不易 debug

在工业界，主流为二进制传输

rpc 错误处理

rpc 未被服务器回复

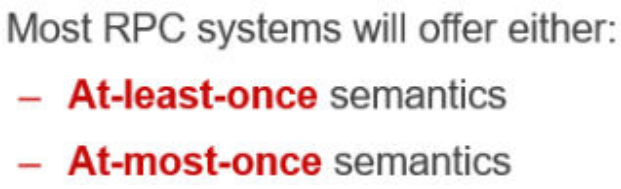
- 请求丢失
- 网络堵塞被丢弃
- 响应丢失
- 服务器崩溃或无法处理

应用应当为失败做好准备：比如 grpc 会返回状态给应用检查

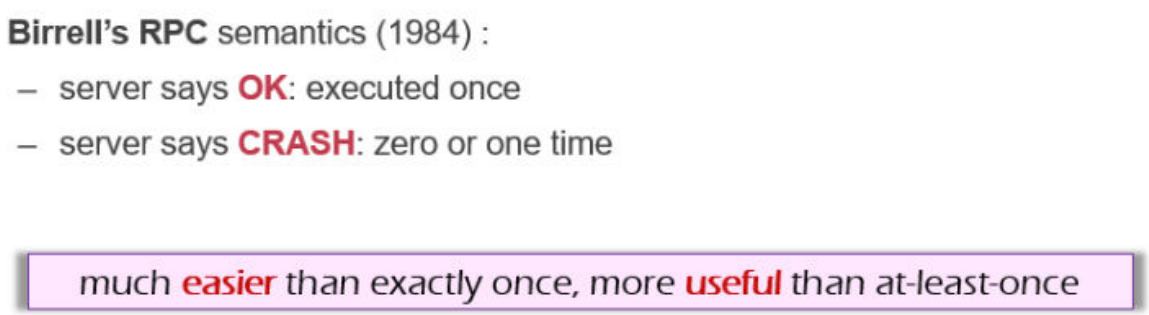
方式：retry

为了保证正确性（执行一次），需要加语义限制，记录 server 执行次数

大多 rpc 会提供



客户端只要 retry，保证 at-least-once,但为了正确性，服务器需要有其他行为



当请求存在**幂等性**或其有特定**检测 duplication**(应用 xid)时，可使用 at-least-once

应用 exactly once

- **检测 duplication**(应用 xid)

DFS(分布式文件系统)

ftp

优点：简单

缺点：

- **Wasteful** “what if client needs small pieces?”
- **Problematic** “what if client does not have enough space?”
- **Consistency** “what if others modify the same file?”

DFS

优点：

- 客户按需获取
- 服务器保证一致性

待解决问题：

- 文件访问期间来了新请求
- 同样的数据被请求多次

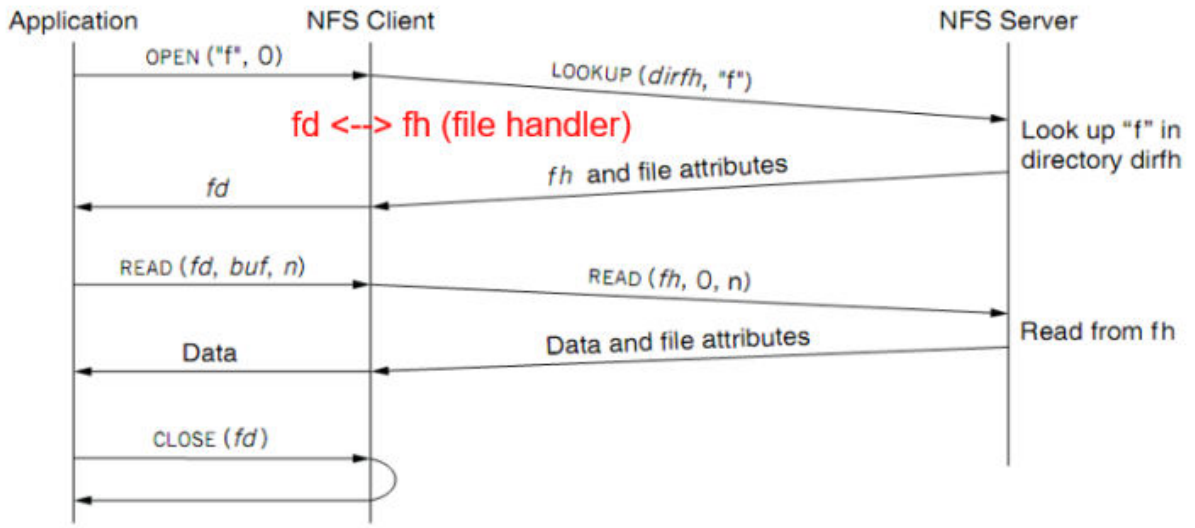
fh:file handler,与 fd 一一映射

- 不能用 file name,防止重命名错误

- 不能用 inode

- inode+generation number(即版本号，每当对应的 inode 被新分配给一个文件，generation number++)：

例子：读取文件



为什么 NFS 不提供 open：

- 保证无状态 stateless，提高容错
- 提高可扩展性（让更多用户访问）

为了写的幂等性：需要保证 at most once，

Solution: each RPC is tagged with a transaction number, and server maintains some "soft" state: reply cache (recall, at-most-once RPC)

对每一个 rpc 标记一个事务 id。

性能

cache

类型

- 在客户端存储文件数据
- 在客户端存储文件属性
- cache 容易破坏一致性

存在问题

close-to-open consistency

一个用户保存文件后如何保证另一个用户收到

Type-1: Close-to-open consistency
- Higher data rate
- **GETATTR** when **OPEN**, to get last modification time
- Compare the time with its cache
- When **CLOSE**, send cached writes to the server

read/write coherence

Type-2: Read/write coherence
- On local file system, **READ** gets newest data
- On NFS, client has cache
- NFS could guarantee read/write coherence for every operation, or just for certain operation

提升读能力

- 提高 chunk 的大小
- read ahead

DFS 限制

1. **Capacity**
   - Can only disks on a single server, which has a limited capacity

2. **Reliability**
   - If the server crashes, the remote files are unavailable

3. **Performance**
   - The file performance is limited to a single file (and a single network bandwidth

如何改进文件系统以适应分布式环境？

- distributed block layer
  - 把 block_id 变为<mac_id,block_id>
  - 选定其中一个 master,记录 free block(存内存以提高性能),
- distributed inode Number layer
  - 将 inode_table 存在 master

其余 layer 不用修改

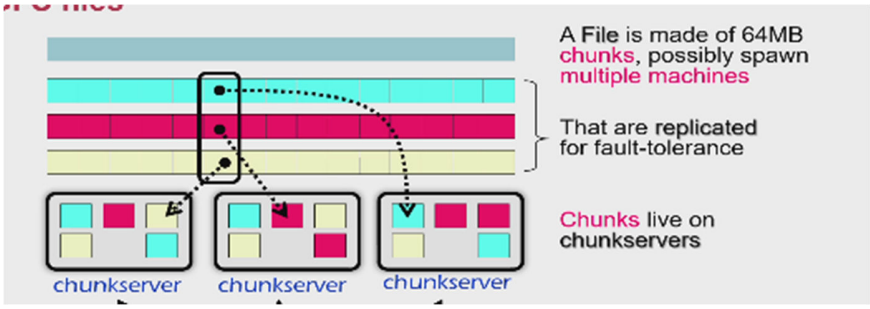
GFS

目标

- scalable
  - 没有 link,symlink,rename
- performance
- fault-tolerant

*A GFS cluster=A master+N chunk**

将同一个文件进行 replica 存在不同机器，提高容错



default chunk size=64M

为什么大 chunk

- 减少频繁与 master 交互
- 确保 tcp 持续连接
- master 可以存储所有元数据

GFS vs NFS

- GFS:master 存储 chunk(block)的当前位置,NFS:block 存在 inode block
- GFS 有非常大的 chunks
- chunks 有备份以提高容错和性能

GFS 只有一个 master

- 简单

GFS 没有 cache

但 client 会存储 chunk 的信息

每台机器记录自己的 chunk,以便 master 获取，因此**不需要在 master 持久化 chunk 信息**

读取文件过程

- 连接 master
- 获取文件元数据：chunk handles
- 获取每个 chunk handle 位置
- 连接任何可用的 chunk server

写文件过程

由于延迟，可能导致无法读到才写完的内容

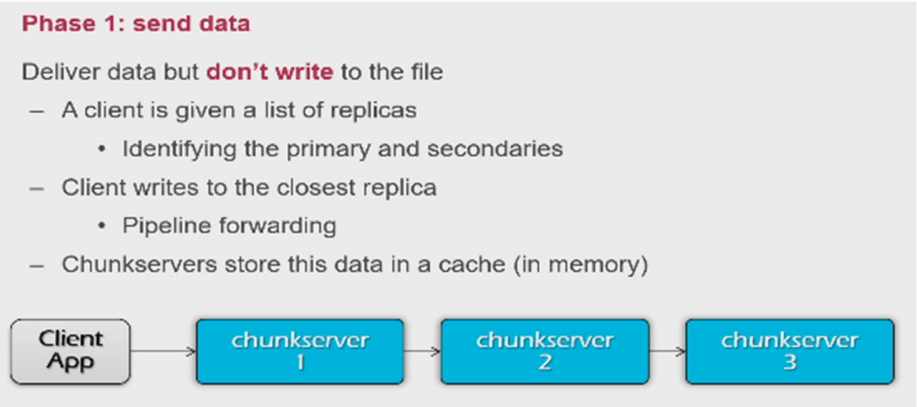
GFS 保证最终一致性

为了防止写-写冲突，会指定一个 chunk server（primary），确定写副本的顺序

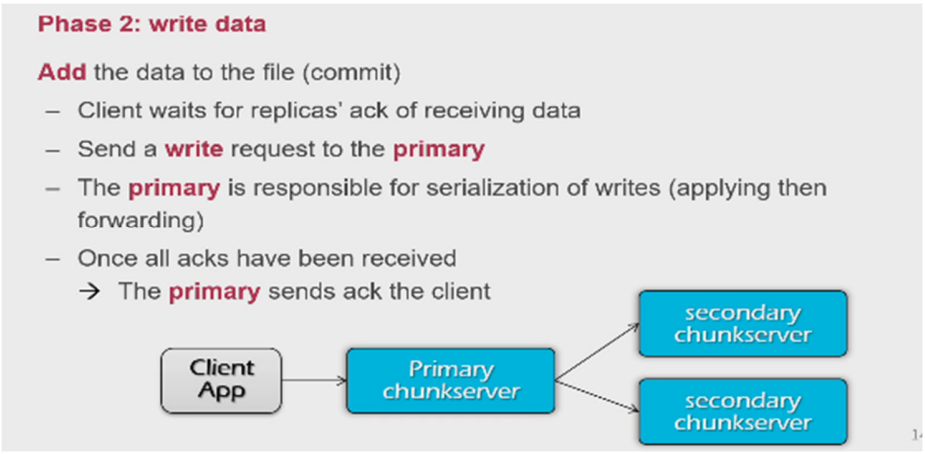
master 在一段时间给其中一个 replica 一个 lease

优化：

- client 先获取副本，给副本分发（pipeline 方式）
- replica 先将数据存在内存



- 当所有副本接受文件后，client 向 primary 发送确认
- primary 确定写排序，向 chunkserver 发送请求，待写完后，回应 client



- chunk 版本，以确定 chunk 是正确的数据

Chunk version numbers are used to detect if any replica has stale data

- Is maintained by the primary chunkserver
- If a replica has stale data, it shall be replaced

Please explain the benefits of splitting the writing process into two phases (data flow and control flow). (3')

data flow makes use of the network bandwidth, the single primary server will not be the bottolneck of transmitting data

control flow ensures the consistency of the chunk among all replicas

GFS 不保证写-写冲突最终谁会写成功，因为 GFS 更多的是 append 操作

append 都会成功

GFS or NFS are not Perfect

NFS

- Can **not scale**
- Is **not fault-tolerant**
- But is well-enough for many workloads, e.g., sharing the data for experiments in our lab ☺

GFS

- **Relaxed consistency model**: the results of concurrent mutations (except append) are undefined
- Single-node master: single point of failures (the next-generation of GFS refines this, with more advanced techniques developed later)
- Work well in Google's datacenter workloads

KFS(key-value)

- key:文件名（可能包含路径）
- value：文件内容

优点

- 减少系统调用
- 节约空间

为了性能，更改文件选择 append,后续进行 merge

删除就插入 null

查询

为数据添加索引

举例:b+树，哈希表

将索引存入内存以提高性能，但插入太多会使内存爆满

选择合适的索引是很难的

- 读性能 vs 写性能
- index 是否易于存储
- 等等

当文件逐渐变大，需要 compation+segmentation

范围查询：b+树，跳表.....

b+树缺点

- get,insert ,update 慢（O(logn)）
- 不适合大量写，随机写，速度慢

Method	Update/Insert efficient	Get efficient	Range Query	Support large dataset
Log	Yes	No	No	Yes
Log + in-memory hash index	Yes	Yes	No	No
Log + on-disk hash	No	Medium	No	Yes
B+Tree	No	Medium	Yes	Yes
LSM tree	Yes	Yes	Medium	Yes

LSM Tree

sstable:

sstable 由 memtable 刷盘而来

- 固定大小
- 当满了后，新建一个 sstable
- 内部 kv 根据 key 排好序
- 不可更改，但后续可 compact

优点

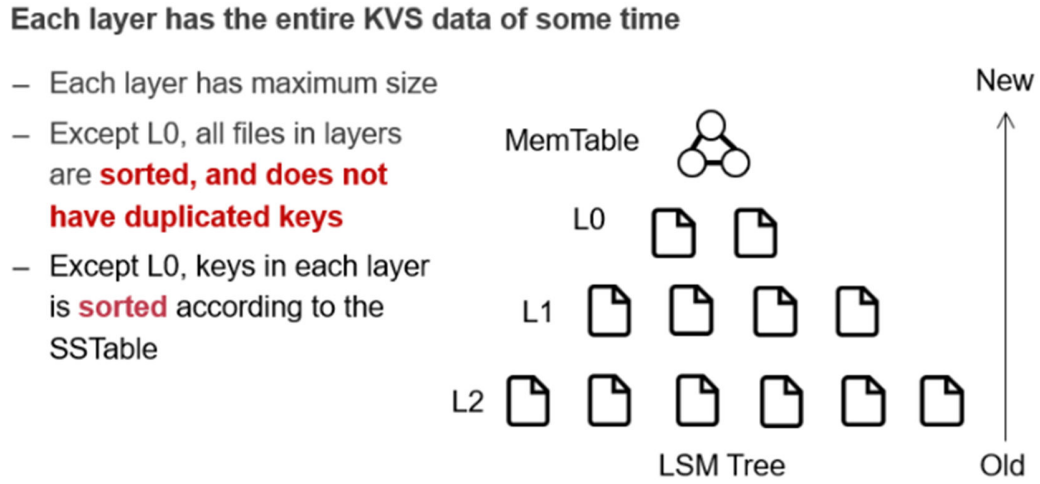
- merge 高效

- search 高效（二分查找）
- 支持范围查找

缺点

- 查找 old value 慢
- 去重困难
- 范围查找不够快

改进：LSM Tree



- 每一层排好序，且没有重复值

B+ vs LSM

LSM:

- 更好的写性能
- 缺点
  - 需要额外的 compaction 开销，
  - 可能更慢的范围查找，
  - 更慢的查找不存在的值，
  - compact 时更慢的写

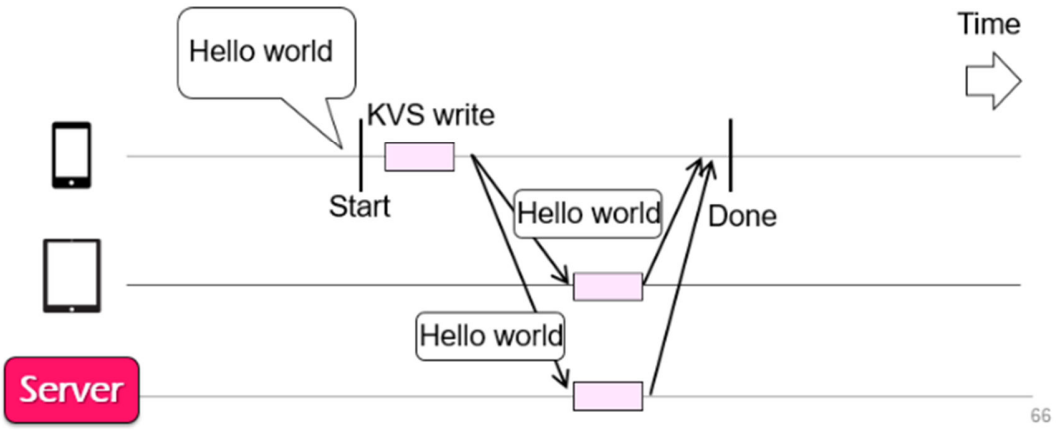
consistency model

如何实现 KVS

- 将 kvs 存在一个中心服务器
  - 低效
  - 不能离线运行
- KVS 存在一个中心服务器和每一个设备
  - 读 good,写低效，需要等待全部完成
  - 容错（需要网络）

Read: return the latest copy on the local KVS

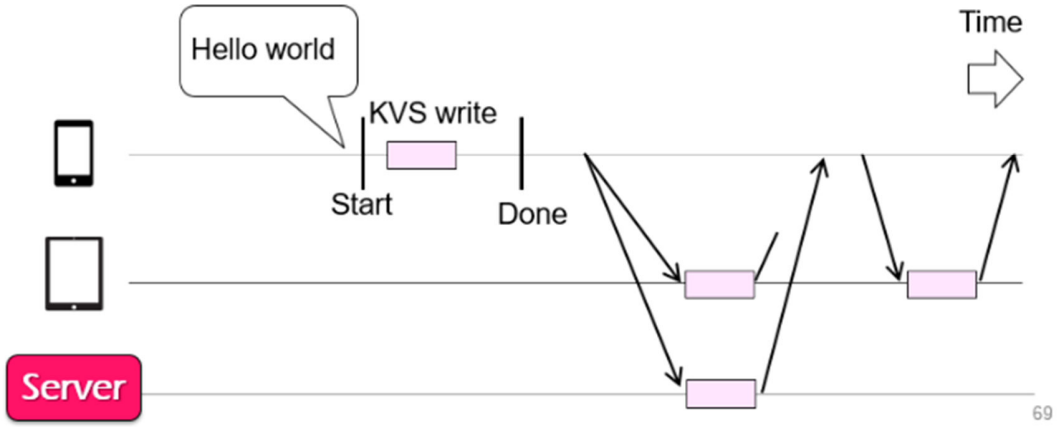
Write: update the local KVS, sync with other KVS, then return to client



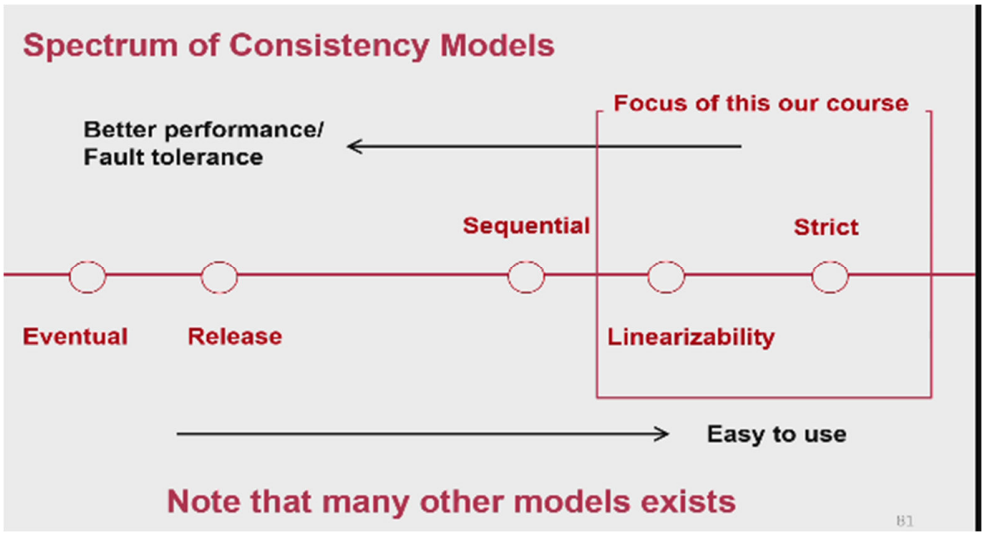
- 改进 **异步但不等待**

  Read: return the latest copy on the local KVS

  Write: update the local KVS, sync with the others in background & return



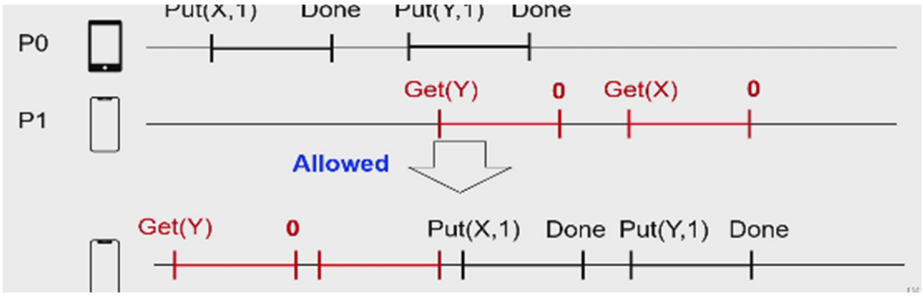
  - 无法读到最新消息（微信）
  - 一致性错误
- 没有对和错的 consistency model，
  - 需要在编码难度与性能间权衡



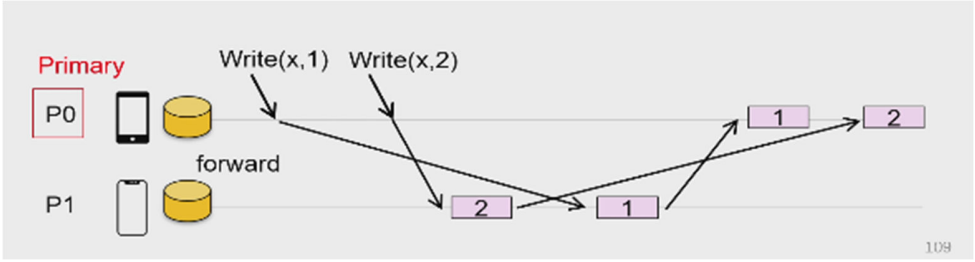
强的 consistency model

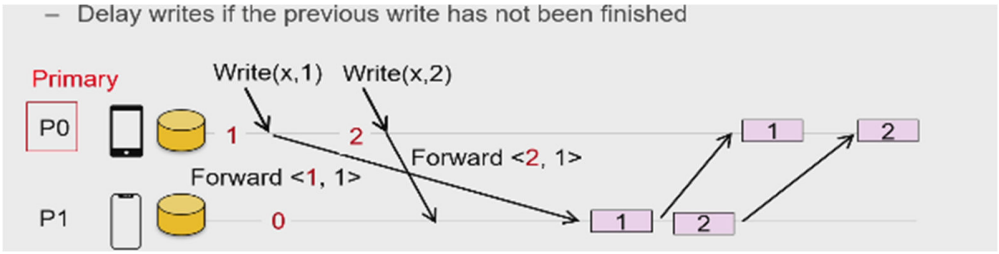
- 尽管有备份，但在用户看来只有一份
- 多线程的执行顺序类似单线程执行
- 整体行为类似不会崩溃的系统

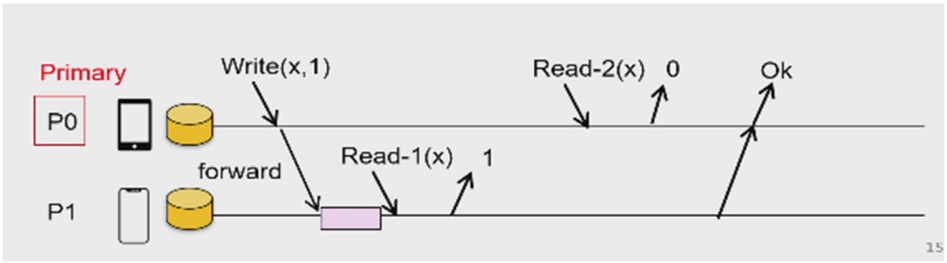
一致性类型

- strict consistency（最强一致性）
  - 难以实现
- sequential consistency（）
  - 需要保证每一个机器的执行是线性的
  - 

- linearizability
  - 若一个 A 操作结束时间在另一个 B 操作开始时间前，则在线性序必须 A 先于 B
  - 如果每一个 X 的操作是 linearizability 的，则所有操作是 linearizability 的
  - 实现细节
    - primary(消息都发给 primary，由 primary 决定顺序)

- 

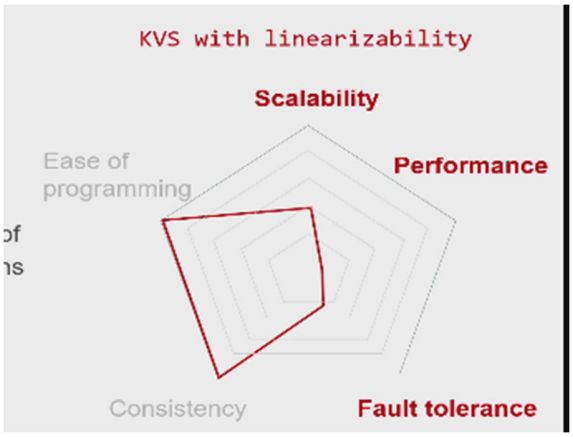
- 全局 counter，每接收一个消息+1，避免上图情况
- 

- 但读取也要向 primary 请求,原因如下
- 

- primary 缺点
  - 读需要额外的 rtt
  - 写需要额外的 rtt
  - primary 成为瓶颈:
    - 不同 key 设置不同 primary
  - primary 可能会 crash

在微信中缺点

- 每次读写需要指定设备处理
- 写入要等待所有副本确认
- 对聊天应用不实际
  - 性能：需要同步所有设备
  - 可用性：如果一些设备离线怎么办
- very slow
- 难以提供容错



Eventual consistency

把性能拉满，降低正确性

所有的 server 最终会收到所有的写是一致的，所有的写副本都一样

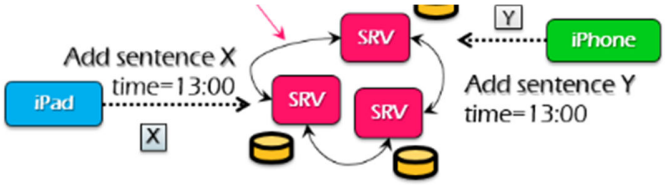
读取：本地最新副本

写：本地写好，直接返回，异步将写发给其他服务器

方案一：直接写入最近服务器，而后传给其他服务器

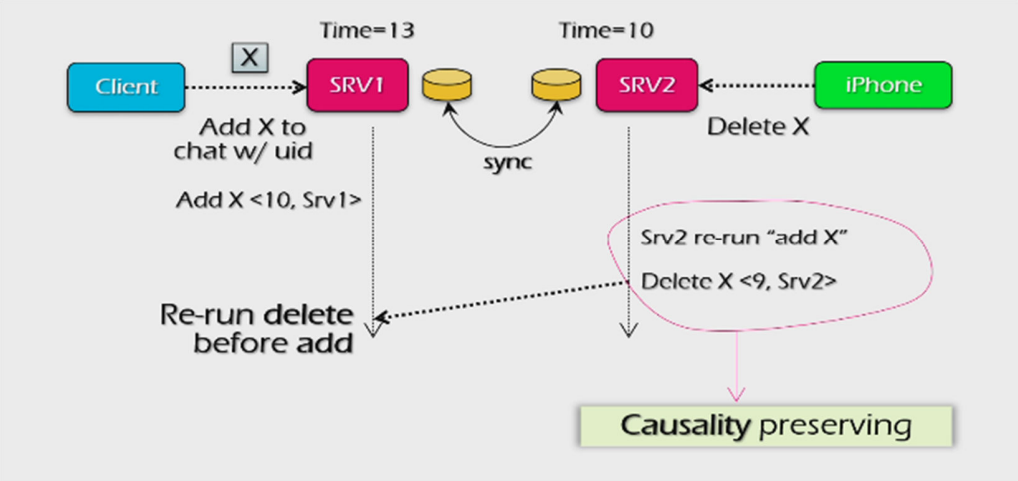
问题：

- 写写冲突



  -
  - 使得 X,Y 数据不一致（顺序颠倒）
  - 与 linearizability 先序列化再更新不同，Eventual consistency 先更新再序列化
  - 方法：update function
    - 更新是一个方法，不是一个值
    - 本地设置一个 update log,使用确定性的方式排序
      - 时间戳<time T,node ID>,当 x,y 时间戳一致，比较二者 id
      - 写数据库同时写 log
      - 当 log 排好序后，数据库回滚然后 replay

- 两个机器时钟不一致，使得可能后发生的事情位于前面



  - 按因果关系排序 X-->Y 当且仅当：
    - 同样机器 x 先于 y
    - x,y 机器不同，但 X causes Y
  - 解决方法：lamport clock
    - 时间戳 T=max(T,T '+1),T '为接收到的上一个消息
    - 但由于 lamport 只有本地时间戳，无法确定两事件是否有因果（时序）关系

      If TS(event #1) < TS(event #2), what does it say about event #1 (create on node #1) and event #2 (created on node #2)?

      ① Event #1 occurred at a physical time earlier than event #2 **No.**

      ② Node #1 must have communicated with Node #2 **No.**

      ③ If event #1 has been synced to node #2, then event #1 must have occurred at a physical time earlier than event #2 **No.**
    -
  - vector clock [1,2,....]记录不同设备的时间戳

    对所有 v1,v2 属于 vector,v1>v2 可比代表有因果关系 不可比代表没有关系

  - 与所有服务器进行同步，若所有机器的时间戳大于 log 的部分值，则该部分 log 可删除
    - 缺点：不能有机器离线
  - 加一层中间件，一个服务器充当 primary
    - 每一个写从 primary 获取新的 CSN,CSN 递增
    - complete timestamp<CSN,local-TS,sevID>递增
    - 当 CSN 是完整的（不存在 1，2，3，5..的情况），则 log 可以删除该部分

Atomicity

一个事务要么都发生，要么都不发生

方法

备份

先拷贝一个副本，修改成功后进行 rename

缺点

- rename 崩溃产生破坏一致性

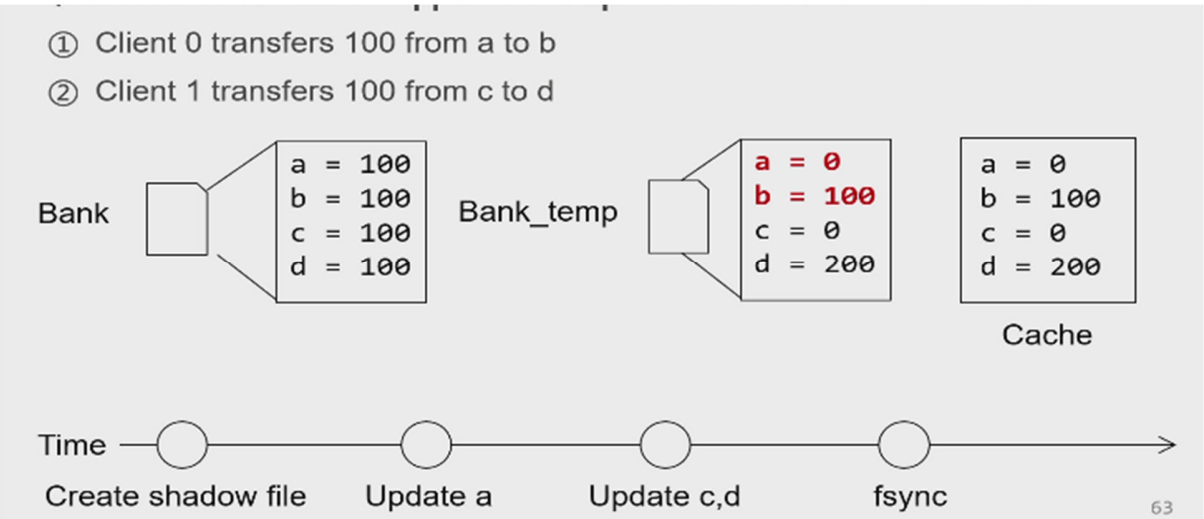
解决方法：record before update

- 步骤
  - record changes in journal
  - commit journal
  - update
- 缺点
  - 每次操作都要在磁盘写两次
  - 文件很大会很慢

为了高效，可以给数据分配优先级如下

- journal
  - 如上
- ordered
  - 数据先修改但不存 journal
  - 再写元数据和 journal
- writeback
  - 不使用 journal

shadow copy



如上，两个客户端 share 同一个 temp-copy,当 client0 执行一半，client1 执行完后进行 async 操作后系统崩溃，导致事务 1 违反一致性

解决方法

- 一个人做完后才允许下一个人操作 log

logging for atomicity

interface

- TX.begin()开始事务
- TX.commit()提交事务

commit logging (redo-only logging)

每次写磁盘之前先写 log

- 为确保 log 正确，可以使用 checksum 进行校验,或写入 commit 代表完成

```
new_a = records[a] - amt
new_b = records[b] + amt
```

```
commit_log = "log start: a:" + new_a + "\b:" + new_b
log.append(commit_log).sync()

record[a] = new_a
record[b] = new_b
```

commit point

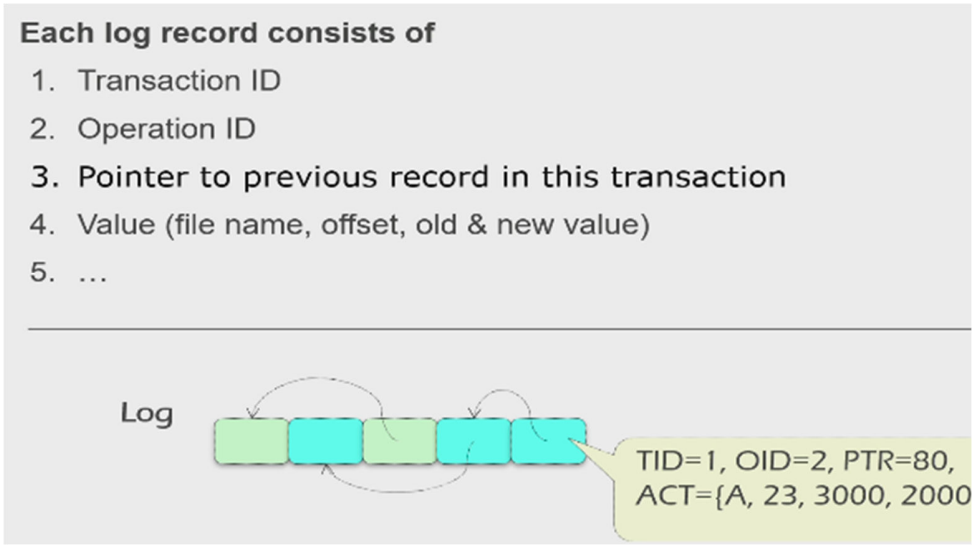
- 过了这个点，代表前面的操作都成功（可以从日志恢复）（如上图第四句为一个 commit point）

缺点：当修改数据太多会占据太多内存

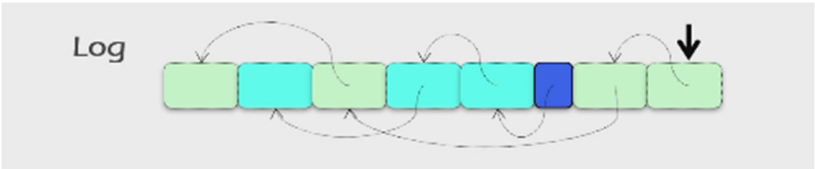
undo-redo logging

当事务操作较多后，将修改内容刷盘，但要 log,以防崩溃

vs commit logging

- commit logging 操作为一个完整的操作
- undo-redo logging 速度较慢
- 

不同颜色代表不同事务



从后往前查找没有 commit 的事务，而后进行回滚

删除不需要的 log

Naïve solution

- Run the recovery process. If it is done, then we can discard all the log file
- Problem: too slow

Observation

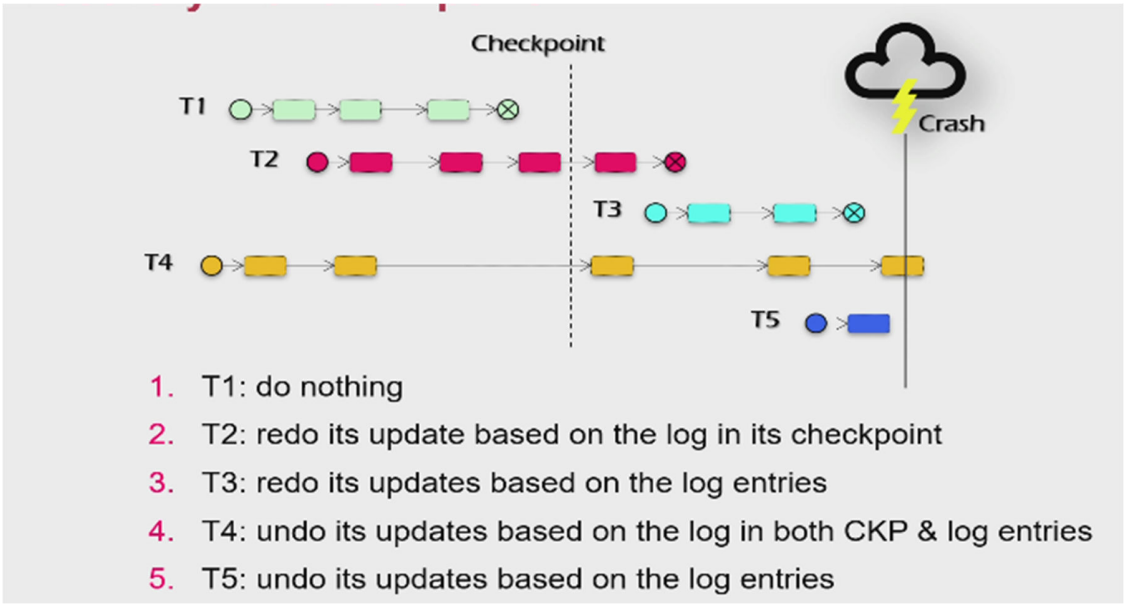
- For redo logging, we only need to flush the page caches so we can discard all the logs of committed TXs
- For undo logging, we only need to wait for TXs to finish to discard all its log entries

简单的 checkpoint

- 步骤
  - 等待没有事务执行
  - flash page cache
  - 丢弃 log
- 缺点
  - 当事务太大，时间会很长

改进：

- 不用等事务完成，等待没有操作执行
- 写一个正在做的但没有完成的事务的 logs 的 checkpoint
- 刷新 page cache
- 丢弃除了 checkpoint 的所有 log



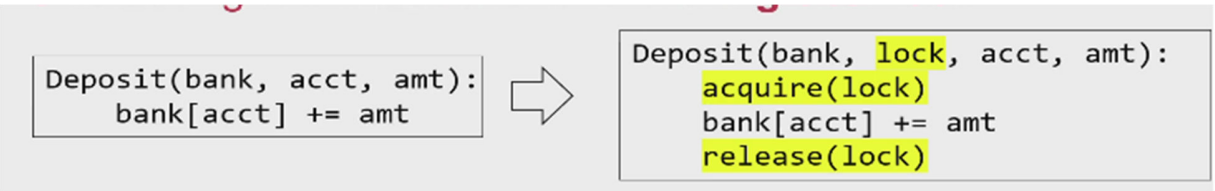
before-or-after-atomicity

race condition:数据竞争

使用锁实现 before-or-after-atomicity

global lock

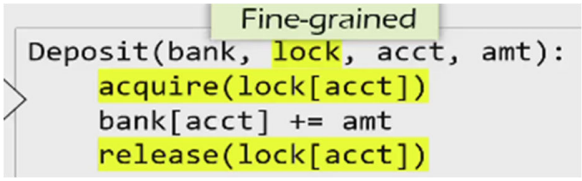
使用全局锁



缺点：过于串行化，效率低

fine-grained locking

只对对应数据加锁



问题：

```
Transfer(bank, locks, a, b, amt):
    Acquire(lock[b])
    bank[b] += amt
    Release(lock[b])
    Acquire(lock[a])
    bank[a] -= amt
    Release(lock[a])
```

```
Audit(bank):
    sum = 0
    for acct in bank:
        Acquire(locks[acct])
        sum += bank[acct]
        Release(locks[acct])
    print(sum)
```

Thread 0 (Transfer)	Thread 1 (Audit)	Bank[Alice]	Bank[bob]	Sum
		10	10	0
Acquire(lock[b])	Acquire(lock[b])	10	10	0
Read(b) = 10	Acquire(lock[b])	10	10	0
Write(b) = 20	Acquire(lock[b])	10	20	0
Release(lock[b])	Acquire(lock[b])	10	20	0
	Read(b) = 20	10	20	20
	Release(lock[b])	10	20	20
	Acquire(lock[a])	10	20	20
	Read(a) = 10	10	20	20
	Release(lock[a])	10	20	30
Acquire(lock[a])		10	20	30
Read(a) = 10		10	20	30
Write(a) = 0		0	20	30
Release(lock[a])		0	20	30

ne

解决方法：遇到数据前先拿锁，j 即 2 phase lock（2PL），而不是提前拿相应数据所有锁

```
Transfer(bank, locks, a, b, amt):
    Acquire(lock[b]
    bank[b] += amt

    Acquire(lock[a])
    bank[a] -= amt
    Release(lock[a])
    Release(lock[b])
```

太多锁会占据过多内存，因此可以限制锁的数量，比如 hash 表，大小代表锁的数量

Serializability has many types

Final-state serializability

– A schedule is final-state serializable if its final written state is equivalent to that of some serial schedule

Conflict serializability ⇦ Most widely used

View serializability

Final-state serializability

```
T1: read(x)              x=0
T2: write(x, 20)
T2: write(y, 30)
T1: tmp = read(y)        y=30
T1: write(y, tmp+10)

At end: x=20, y=40
```

conflict serializability(冲突可串行化)

两个操作 conflict，则

- 他们操作同一数据
- 至少有一个写
- 属于不同 transaction

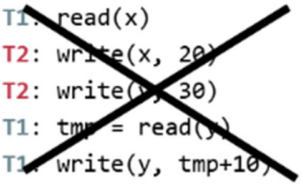
conflict serializability

串行化执行中没有产生环形依赖

T1 -> T2 表示 T2 依赖 T1

```
T2: write(x, 20)
T1: read(x)
T2: write(y, 30)
T1: tmp = read(y)
T1: write(y, tmp+10)
```

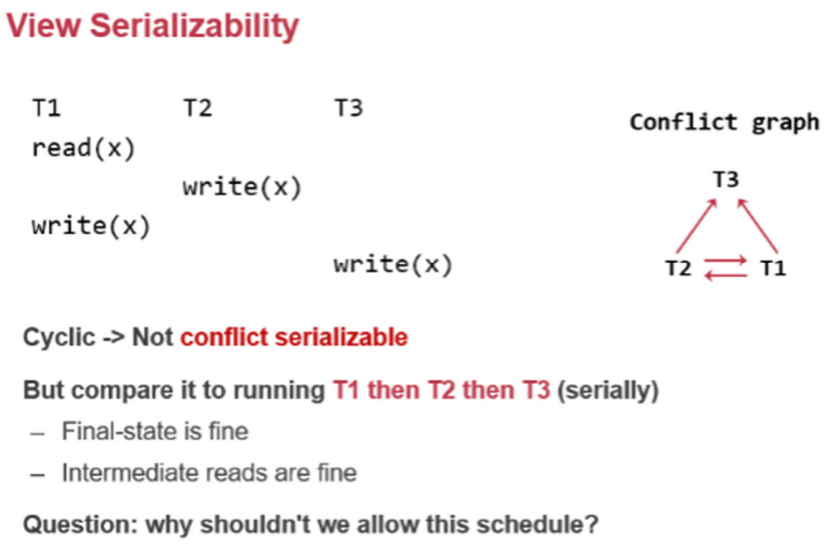
```
T2.1 -> T1.1
T2.2 -> T1.2
T2.2 -> T1.3
```




```
T1.1 -> T2.1
T2.2 -> T1.2
T2.2 -> T1.3
```

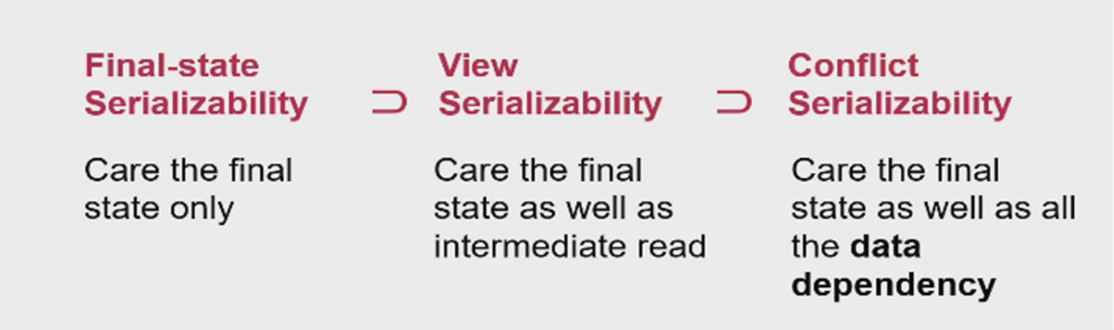
view serializability

难以实现

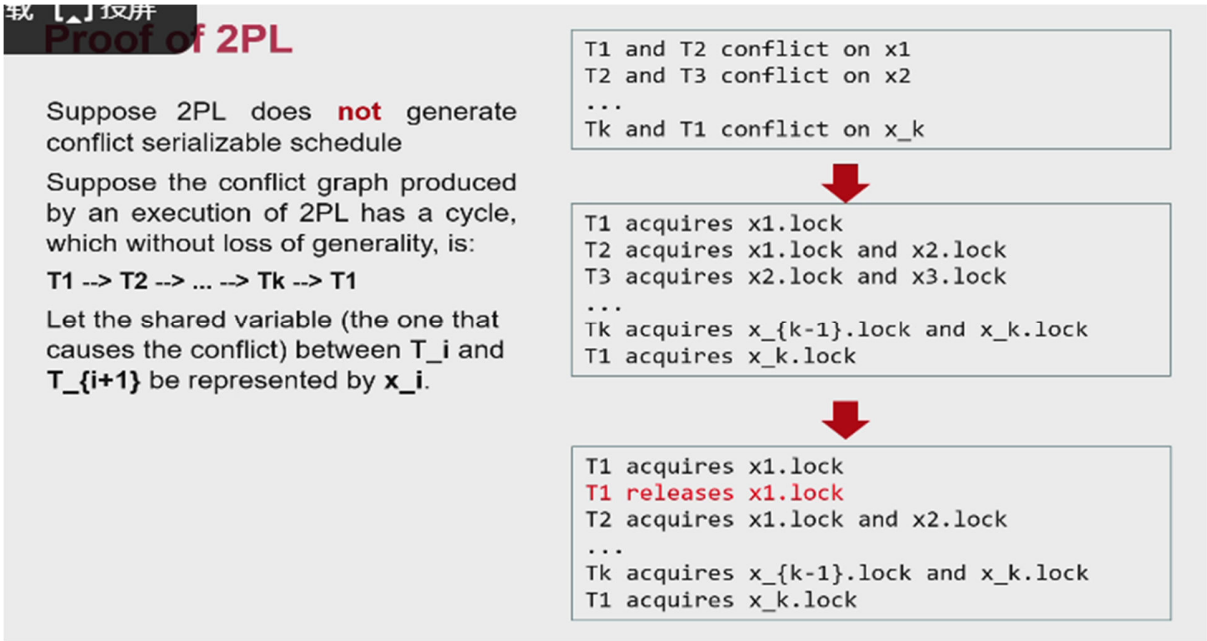


上图虽然看似有环，但写操作与 t3->t2->t1 没区别

因此，即使使用 conflict serializability 无法判断是否串行化，但其仍可能是串行化的

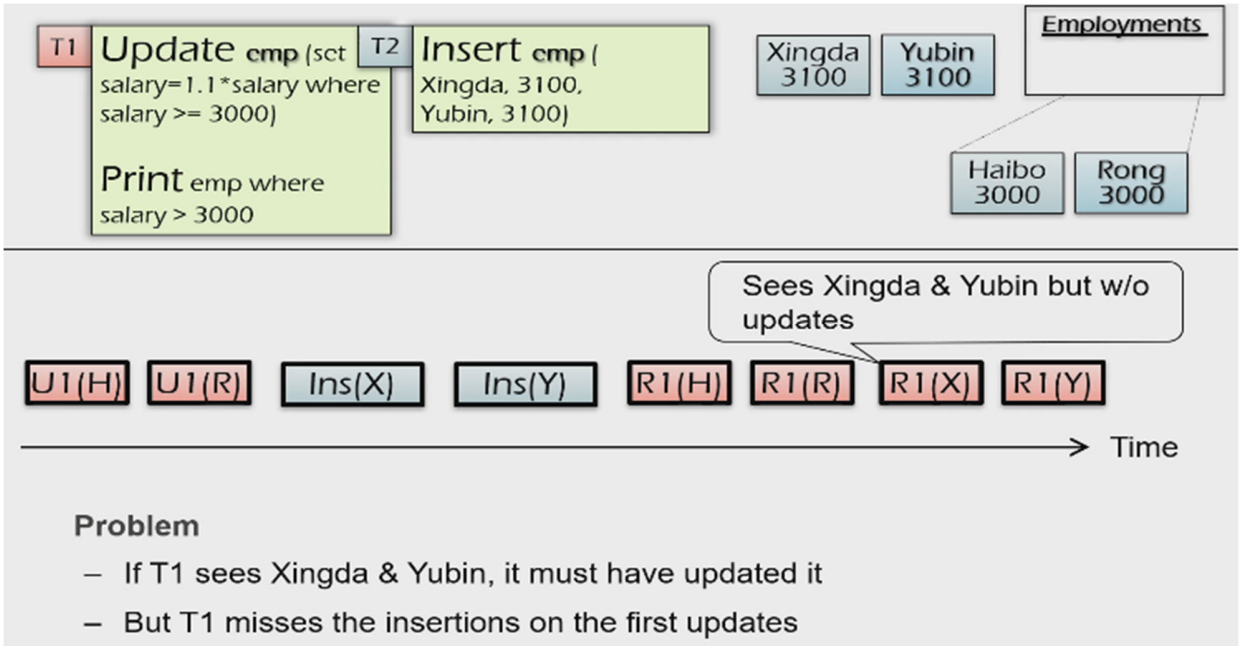


conflict serializability 是最严格的，但很容易判断出来



2PL 不一定总能保证 conflict serializability

- 幻读问题



所以要保证没有新数据插入

解决方法

- Predicate locking
- range lock (类似对 B 树的 index)或整个表
- 有时候可以忽略，手动解决

deadlock

解决方法：

- 按顺序拿锁
- 检测死锁，然后终止其中一个 TX(transaction),但检测很耗资源
- 使用启发的方法（比如一个事务超时则终止），但可能出错

乐观锁（OCC）

乐观锁假设并发冲突很少发生，因此在访问资源时不加锁，而是在提交数据时检查是否发生了冲突。如果检测到冲突，则回滚事务并重试

```
...
tx.begin();
...
tx.read(A)
...
tx.commit();
```
```
Init read_set
Init write_set
```

- 读阶段
  - 事务在本地缓存中处理数据
    - read data into a read set

```
...
tx.read(A)
...
```
```
val_a = read(A)
read_set.add(val_a)
```

    - buffers writes into a write set

```
tx.write(A)
tx.commit();
```
```
Write_set[A] = ..
if A in read_set:
   read_set[A] = .
```

    - 写完还要更新读集合，原因如下

```
tx.write(A)
tx.read(A)
tx.commit()
```
```
if A in read_set:
   return read_set[A
... // normal read
```

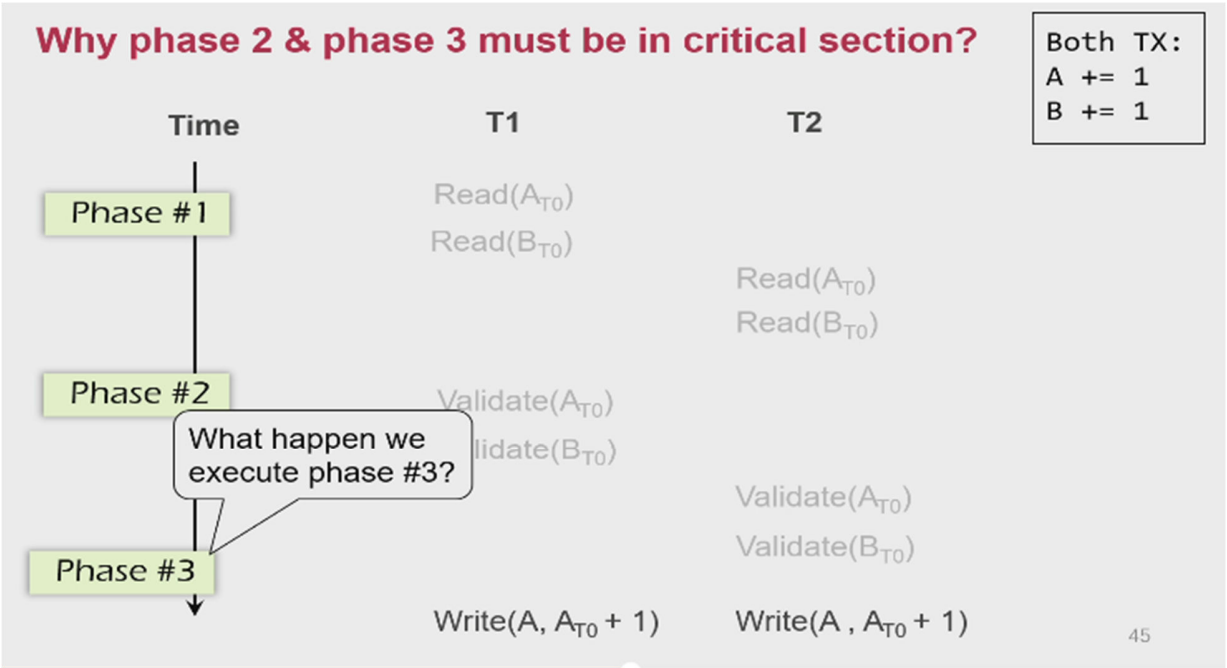
- 验证阶段（validation）
  - 检测是否满足串行化
  - 看第一阶段 read set 的数据与现在是否一致

```
...
tx.commit();
...
```
```
for d in read_set:
   if d has changed:
      abort()
```

- commit 阶段
  - 若 validation failed,则 abort（比对版本号，防止一个数据被改动过两次）
  - install write set,提交事务

```
();
```
```
Critical section
for d in read_set:
   if d has changed:
      abort()
for d in write_set:
   write(d)
```

2,3 阶段必须是原子操作（in a critical section）,防止 validation 阶段时一个值改变了



- 全局锁（因为此阶段时间很短，冲突不会很多）

```
def validate_and_commit() // phase 2 & 3
    global_lock.lock()
    for d in read-set:
        if d has changed:
           abort()
    for d in write-set:
        write(d)
    global_lock.unlock()
```

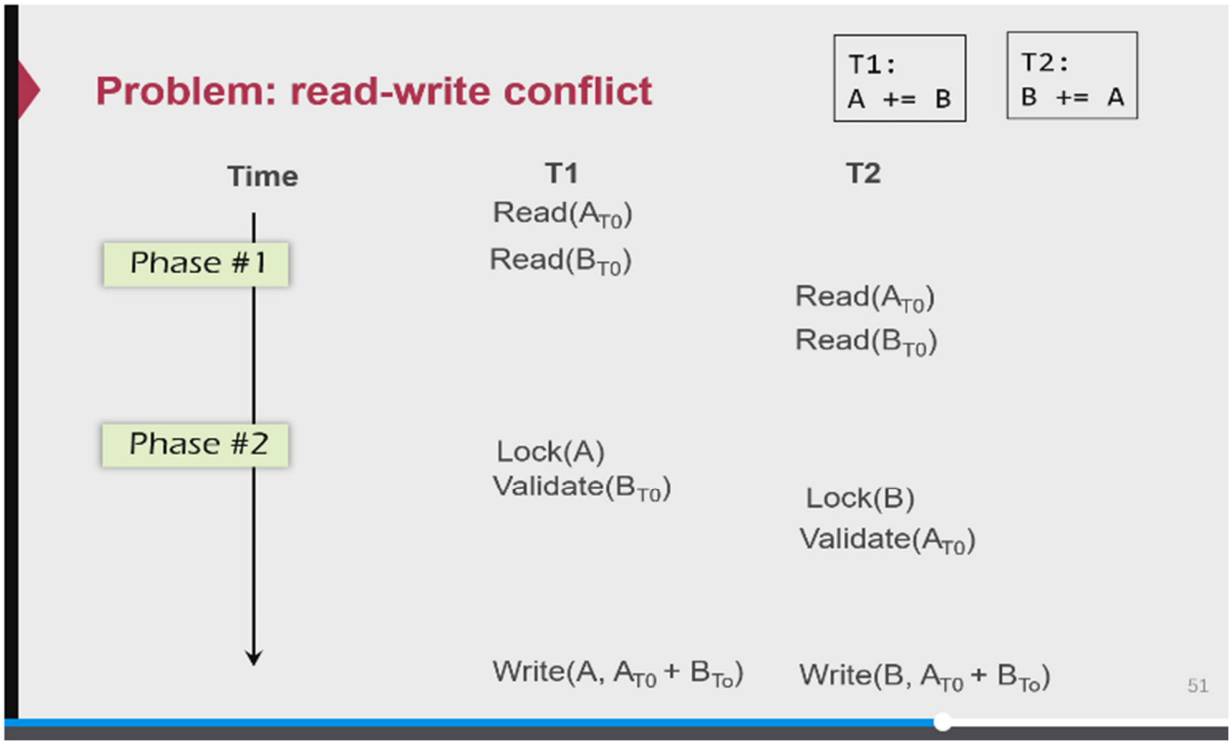
- 提高并发性，同时进行排序防止死锁（因为数据已知，便于排序）

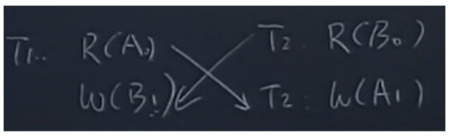
Using two-phase locking

- Allow more concurrency
- What about the problem of deadlock?

```
def validate_and_commit() // phase 2 & 3
    for d in read-set:
        d.lock()
        if d has changed:
           abort() // abort will release all the held lock
    for d in write-set:
        d.lock()
        write(d)
    // release the locks
    ...
```

occ 可以不拿读锁，因为 occ 通过 validation 等价于拿了一把隐形的读锁?不对！原因如下





为了不上读锁，validation 还要检测读数据是否有锁

```
def validate_and_commit() // phase 2 & 3 with before-or-after
    for d in sorted(write-set):
        d.lock()
    for d in read-set:
        if d has changed or d has been locked:
           abort()
    for d in write-set:
        write(d)
    // release the locks
```

OCC 优点

在读远多于写的情景下 ，读集合不需要拿锁

- OCC (in the optimal case, i.e., no abort)
  - 1 read to read the data value
  - 1 read to validate whether the value has been changed or not (as well as locked)
- 2PL
  - 1 operation to acquire the lock (typically an atomic CAS)
  - 1 read to read the data value
  - 1 write to release the lock
    - A single CPU write is atomic, no need to do the atomic CAS

上锁的开销非常大

锁

硬件保证

Hardware primitive to guarantee before-or-after atomicity

- Compare-and-swap (on SPARC)
- Compare-and-exchange (on x86)
- `Lock prefix` to ensure an instruction is atomically executed on a memory address

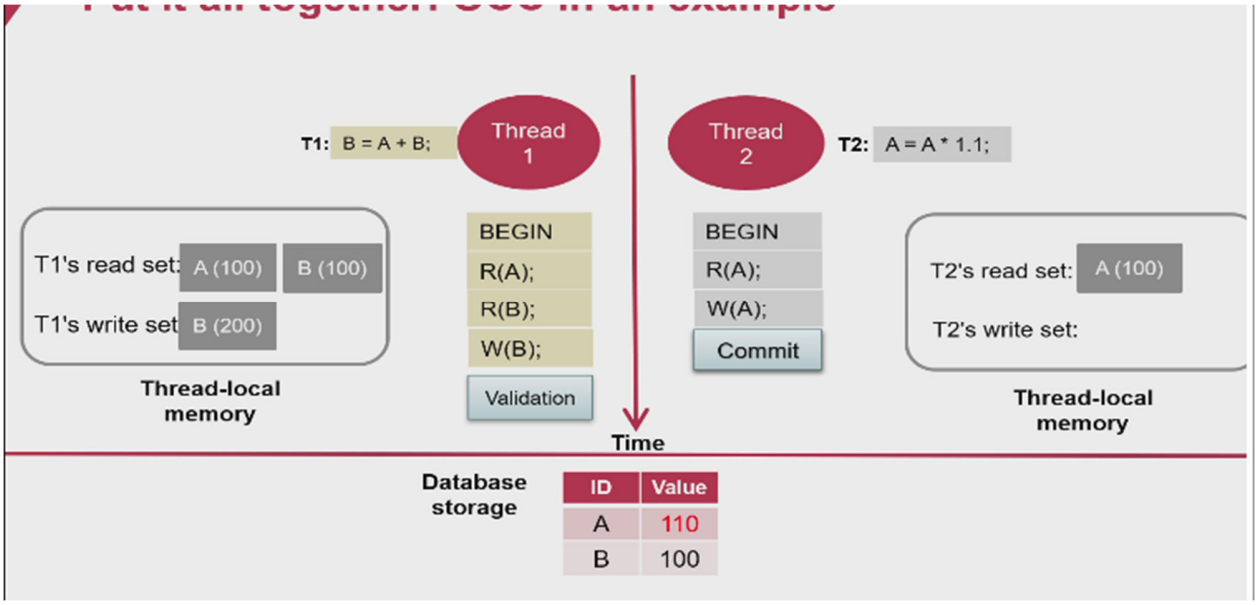
```
1 int CompareAndSwap(int *ptr, int expected, int new) {// with the
lock prefix
2     int actual = *ptr;
3     if (actual == expected)
4         *ptr = new;
5     return actual;
6 } // hardware ensures atomicity with the lock prefix
```

57

自旋锁

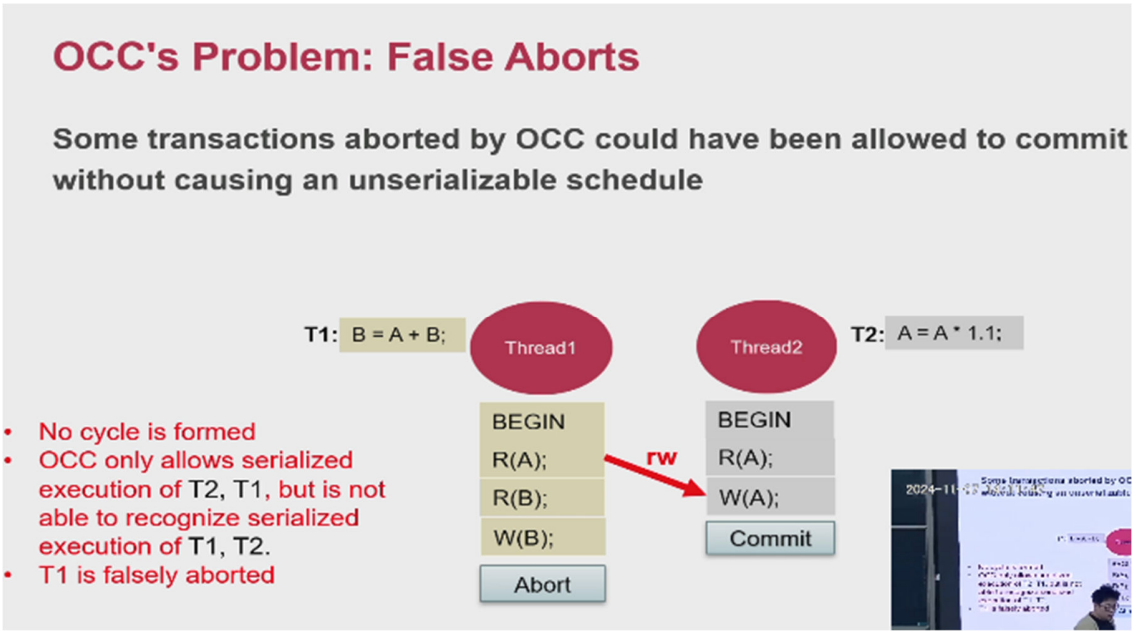
```
1 typedef struct __lock_t {
2     int flag;
3 } lock_t;
4
5 void init(lock_t *lock) {
6     // 0 indicates that lock is available, 1 that it is held
7     lock->flag = 0;
8 }
9
10 void lock(lock_t *lock) {
11     while (CompareAndSwap(&lock->flag, 0, 1) == 1)
12     ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

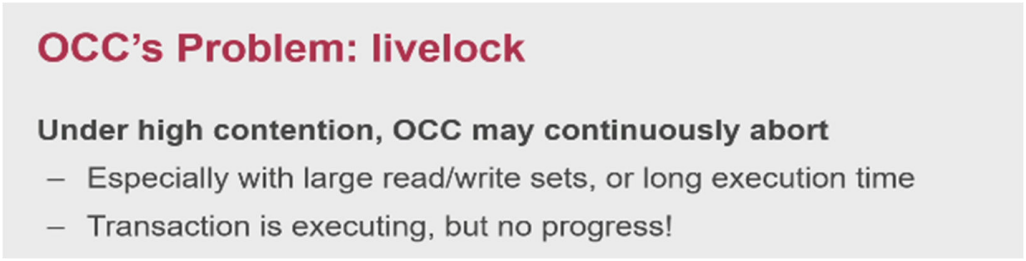
OCC 流程：最终 T2 会成功，T1 会失败，因为 A 不一致

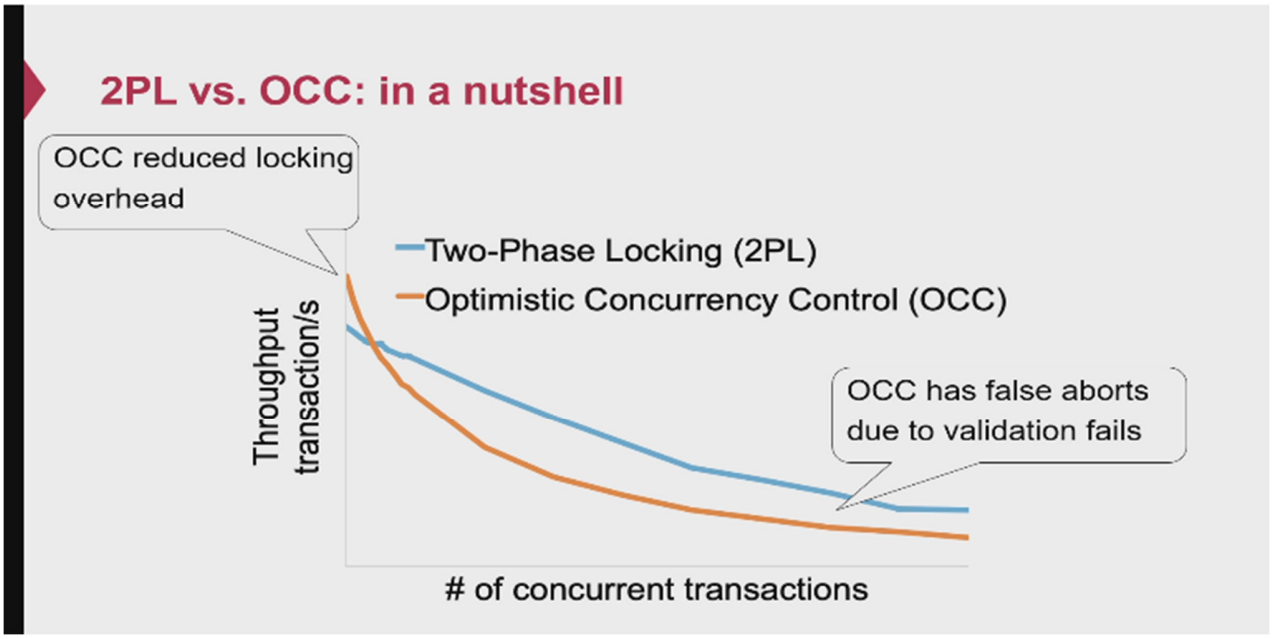


但 OCC 相对于 2PL 更保守，因为可能没有出现依赖的时候依然会被 abort

当一个事务执行越长，越可能被 abort（造成 livelock）





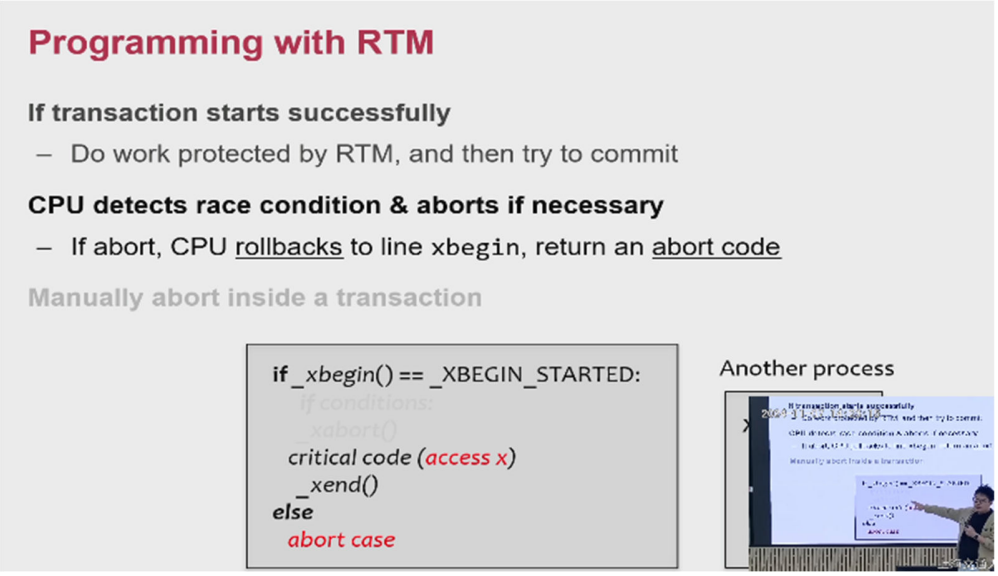


Hardware Transactional Memory(HTM&RTM)

基于 OCC 实现

提供指令级命令：

- 使用 xbegin 表示 RTM 开始
- 使用 xend 表示 RTM 结束



由于使用 OCC 实现，为了保证成功，需要软件层面保证：

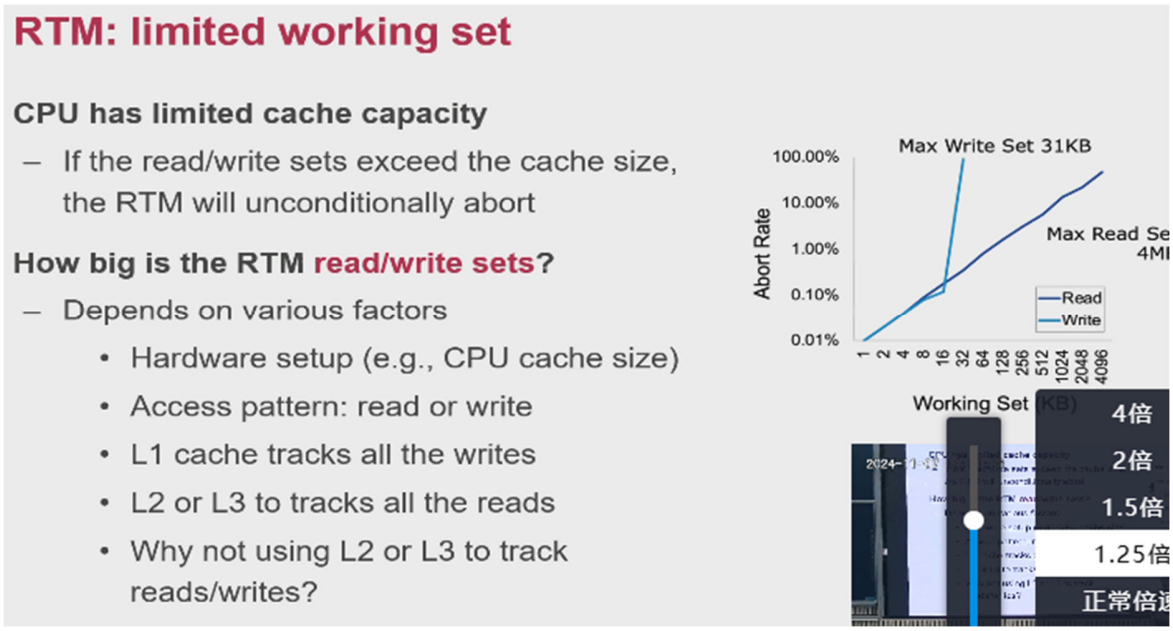
```
Beginning:
        if _xbegin() == _XBEGIN_STARTED
            /* do some critical work */
            _xend()
        else
            goto beginning
```

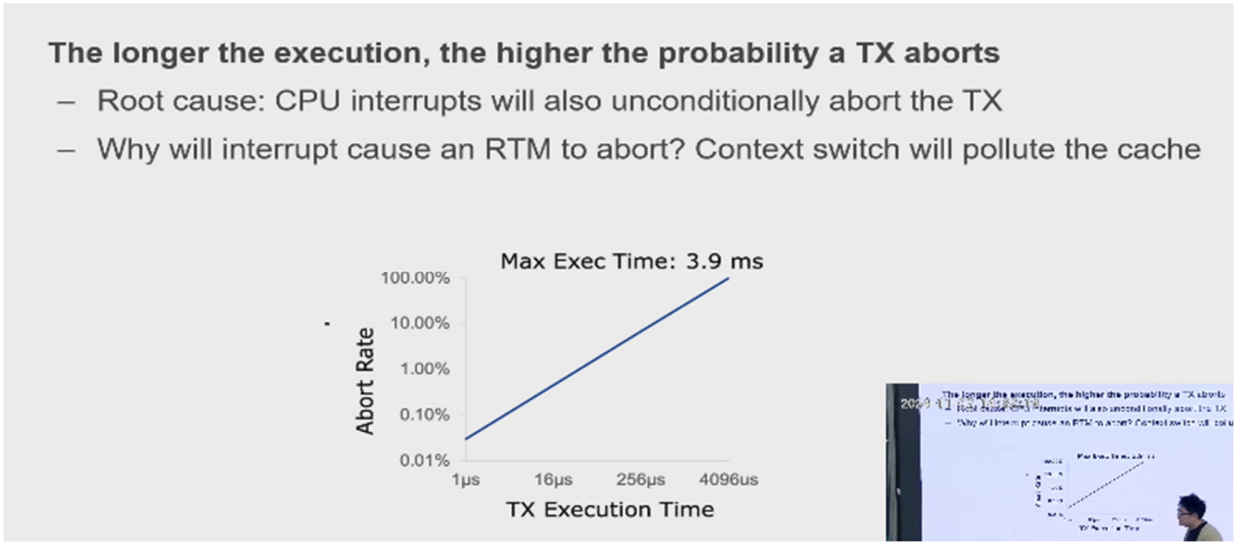
为了保存中间结果（cache），硬件使用 L1，L2 层进行保存

硬件使用 OCC 非常容易，因为一旦一个核改变了一个数据，会广播给其他核

RTM 限制

- cpu 的 cache 容量不够大

- 

- 运行时间越长，越容易 abort
- 

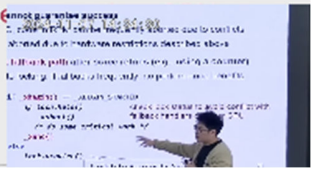
为了解决问题，当多次 abort(counter 计数)，可以使用一个回调路径（fallback path）

Must switch to a fallback path after *some* retries (e.g., using a counter)

– E.g., fallback to locking; if fallbacks frequently, no performance benefits

```
if _xbegin() == _XBEGIN_STARTED
    if lock.held()
        xabort()
    /* do some critical work */
    _xend()
else
    lock.acquire()
```

check lock status to avoid conflict with fallback handle



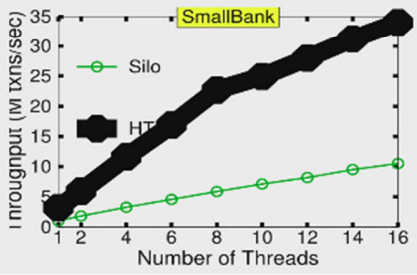
HTM 在不同复杂度场景下的处理速率（如下可以看出 HTM 适合简单事务）

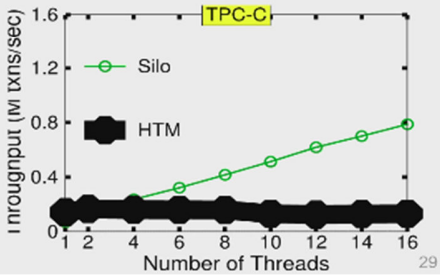
Naïve: using HTM to execute the in-memory transaction

– Smallbank: transfer &. Audit

– TPC-C: much complex, insert 10+ orders and update 10+ stocks

Silo@SOSP'13: the fastest in-memory OCC implementation so far





29

总结

Short summary of OCC & RTM

OCC is yet another classic protocol for before-or-after atomicity

– Whose idea has even been adopted by hardware designers

Hardware support for transactional memory

– Easy programming model for the programmer (no need for locking)
  - As long as the programmer do the in-memory computations, e.g., in-memory database

– Good performance if using properly
  - No need for locking & atomic operations

– However, the programmer should handle its pitfalls

MVCC(多版本并发控制)

Get the start and commit time

Requirement: the counter reflects TX's serial execution order

- E.g., if T1 finishes before T2, it will have a smaller start & commit timestamp
- Not necessary for read-only TX, but is critical for the write

Simplest (& most widely used) solution: global counter

- Using atomic fetch and add (FAA) to get at the TX's begin & commit time
- TX Begin: use FAA to get the start time
- TX Commit: use FAA to get the commit time

We will introduce more advanced timestamps in later lectures

- Not using a global counter is challenging because de-centralized time (e.g., physical time) is unsynchronized

try1：优化 OCC（有问题）

黄色部分为区别

- phase1:读取距离事务开始前最新的数据

  Acquire the start time

  Phase 1: Concurrent local processing

  - Reads data belongs to the snapshot closest to the start time
  - Buffers writes into a write set

  Acquire the commit time

  Phase 2: Commit the results in critical section

  - Commits: installs the write set with the commit time
- phase2

以下仅对于读操作

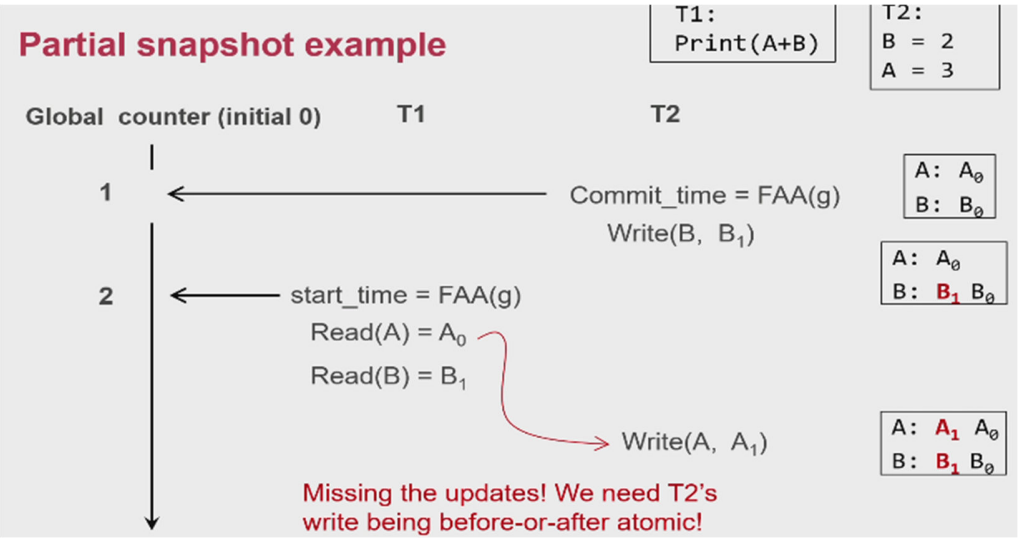
Compared to the OCC, no validation is need!

```
Commit(tx):
   for record in tx.write_set:
       lock(record)
   let commit_ts = FAA(global_counter)
   for record in tx.write_set:
       record.insert_new_version(commit_ts, ...)
       unlock(record)


Get(tx, record):


   for version,value in record.sort_version_in_decreasing():
       if version <= tx.start_time:
          return value
```

问题：写不是原子的，当写 write_set 写到一半，另一个新开事务



写操作需要原子操作

加锁

```
Commit(tx):
   for record in tx.write_set:
       lock(record)
   let commit_ts = FAA(global_counter)
   for record in tx.write_set:
       record.insert_new_version(commit_ts, ...)
       unlock(record)


Get(tx, record):
   while record.is_locked():
         pass
   for version,value in record.sort_version_in_decreasing()
       if version <= tx.start_time:
          return value
```

必须先上锁在拿数据，因为要确定好数据的序列

MVCC 只保证没有读竞争，但不保证写

Race conditions of reads are isolated via snapshots

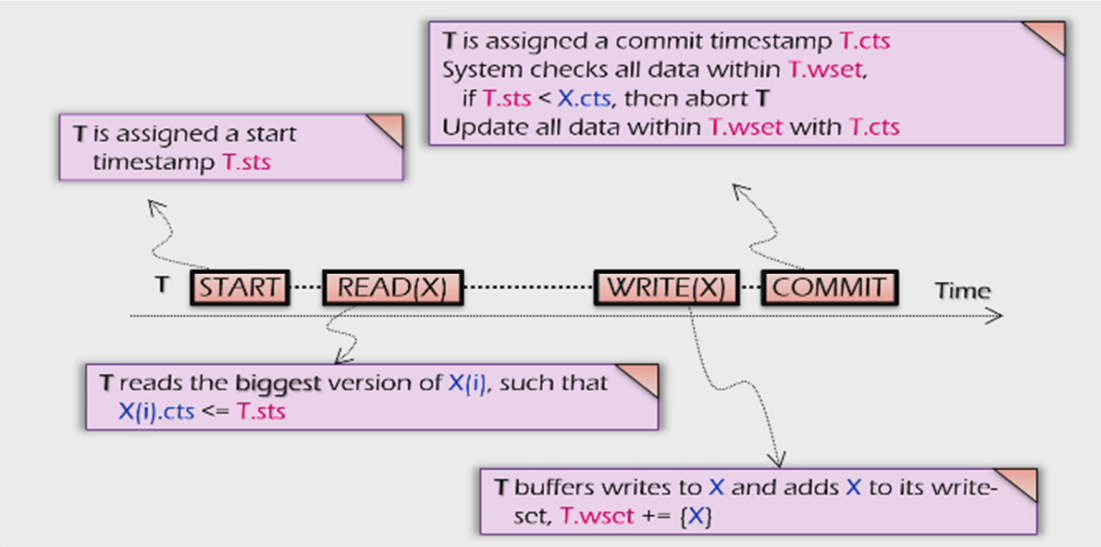
But we still needs to rule out race conditions between reads + writes

- We do so by validating the writes at the commit time

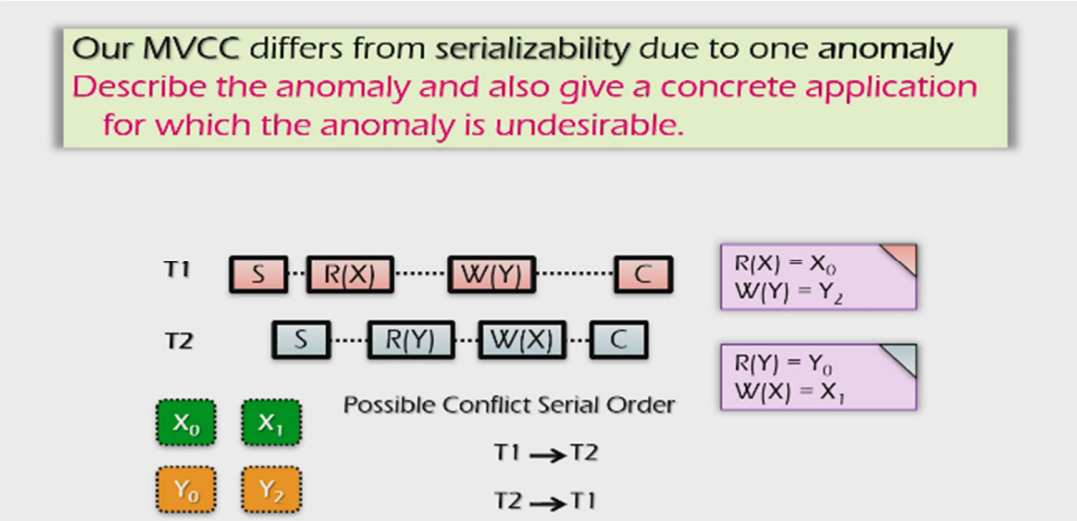
The validation is simple:

- During commit time, check whether another TX has installed a new snapshot after the committing TX's start time

因此 MVCC 的写操作需要 validate



MVCC 不一定支持线性化



若要支持，则读也要 validation

Fixing the anomaly

The simplest way is to validate the read-set in read-write TX

- Essentially fallbacks to OCC for read-write TX
- But read-only TX can still enjoy the benefits from MVCC
  - Never aborts & no validations

Usually being ignored in practice (Snapshot isolation)

- The MVCC without the read validation is also called **snapshot isolation (SI)**

Though the idea of MVCC is first proposed in SI, its usage is not restricted to it, e.g., we can have MV-2PL and MV-OCC; & we will see it later

事务

特性

- A Atomisity（原子性）
- C consistency（一致性）
- I isolation（隔离性）
- D durability(持久化)

一致性是由应用程序自己保证的

Why is isolation typically related to consistency?

Because it simplifies the programmer to enforce consistency

- Recall: consistency depends on how the programmer writes the program

Consistency

Transaction must change the data from a consistent state to another

- What is consistent is **defined by the applications**
- E.g., transfer should leave the `sum(bank[a] + bank[b])` **unchanged**

线性化是理想化的

Why serializability is ideal?

Assumption: programmers are pro at writing single-thread programs

- Specially, $\forall_{tx}\ Ci \rightarrow tx \rightarrow Cj$, in a single-thread context, tx can move data from a consistent state (C_i) to another consistent state (C_j)

Then, if transactions guarantee serializability, then the final state of concurrent execution is consistent

- i.e., the concurrent execution can reduce to $C_0 \rightarrow tx_0 \rightarrow tx_1 \rightarrow \ldots \rightarrow Cn$ If C_0 is consistent, then C_n must be consistent

Why we need to use a TX?

Programmers can manually add logging & locking to their program

- Can guarantee all-or-nothing and before-of-after

However, this is a typically bad idea

- Question: what if the programmer falsely releases the lock (like the fine-grained lock example in our previous lecture?)
- What if the programmer write the wrong logging & recovery mechanism?

总结

Summary

Transactions provide ACID properties

OCC & 2PL are basic protocols to provide serializability

- Problem of 2PL: locking overhead & deadlock
- Problem of OCC: False abort & livelock

Hardware transactional memory (HTM)

- Advanced CPU features inspired by ACID properties of TXs
- Commercial implementation of HTM uses OCC
  - Leads to several drawbacks

The cost of concurrency control can be reduced w/ relaxed isolation level

- Snapshot Isolation (require global counter, not serializable)

Multi-site autonomy

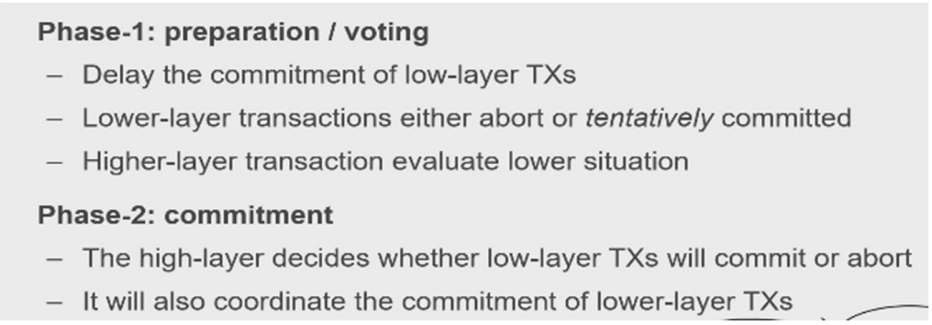
two-phase commit

- phase1:preparation/voting

  在准备阶段，协调器向所有参与者发送准备请求（Prepare Request），询问它们是否可以提交事务。每个参与者执行事务操作，但不提交，并将结果记录到日志中。然后，参与者向协调器发送准备响应（Prepare Response），表示是否可以提交事务。

- 提交阶段（Commit Phase）

  在提交阶段，协调器根据所有参与者的响应决定是否提交事务。如果所有参与者都同意提交事务，协调器向所有参与者发送提交请求（Commit Request），参与者提交事务并释放资源。如果有任何参与者不同意提交事务，协调器向所有参与者发送回滚请求（Rollback Request），参与者回滚事务并释放资源。



缺点

- **阻塞问题**：在提交阶段，如果协调器发生故障，参与者将一直等待提交或回滚请求，导致系统阻塞。
- **性能开销**：两阶段提交协议需要多次网络通信和日志记录，增加了系统的性能开销。
- **单点故障**：协调器是两阶段提交协议中的单点故障，如果协调器发生故障，整个事务将无法继续。

logging rule

low-layer transaction

- redo-undo log entry: like nolmal
- commit log entry->tentative commit log entry(添加 prepare 状态)
  - 包含指向 high-layer tx 的地址，
  - in case of failure:以询问能否恢复状态
- high-layer transaction:
  - 只是记录最终是提交还是 abort 状态

原则

- high level 与 low level 都有 log
- 是否 commit 只看 coordinator 的决定
- low-level 只能收到 high-level 的决定做决定
- 网络丢包，retry
- 若 prepare 一直等不到响应，则直接 abort

Summary: the logging strategy of high TX

What must be logged?

- Whether a TX is committed (or aborted)

Do we have to log the prepare? E.g., whether a TX has acked the prepare

- Not necessary. We can just abort upon recovery

Recall: minimize logging is critical to performance

- E.g., redo vs. undo-redo logging

Remaining question: what about checkpoint?

- Easy to do with the low TX, but needs some additional work on the high TX

2PL and OCC in 2PL

Nearly identical

- 2PL: each low-layer TX cannot release its lock until the high-layer TX decides to commit
- OCC: the validation & commit phases are done by the coordinator

2PL and CAP

为了防止 coordinator 挂，需要进行 replicate

replication

- 增强性能
- fault tolerance

Optimistic Replication (e.g., eventual consistency)

- Tolerate inconsistency, and fix things up later
- Works well when out-of-sync replicas are acceptable

Pessimistic Replication (e.g., linearizability)

- Ensure strong consistency between replicas
- Needed when out-of-sync replicas can cause serious problems

replicated state machines(RSM)

复制状态机

- 使分布式机器像一个单机
- 使用 view server 告诉所有人谁是 primary
    - 当 primary fail,可以

RSMs provide single-copy consistency

- Operations complete as if there is a single copy of the data
- Though internally there are replicas

RSMs can use a primary-backup mechanism for replication

- Using **view server** to ensure only one replica acts as the primary
- It can also recruit new backups after servers fail

- 使用 primary-backup
    - 保证有一个服务器对接受的请求排序
- backup：当 primary 崩溃，可以从 backup 恢复

Implementing RSM w/ Primary Backup model

RSMs can use a primary-backup mechanism for replication

- Ensure only **one server** for ordering inputs received from clients

Primary does important stuff

- Ensures that it sends all inputs to the backup before ACKing the ops
- Chooses an ordering for all operations, so that the primary and backup agree (i.e., one writer)
- Decides all non-deterministic values (e.g., `random()`, `time()`)

It can also recruit new backups after servers fail, but needs to take care (see later)

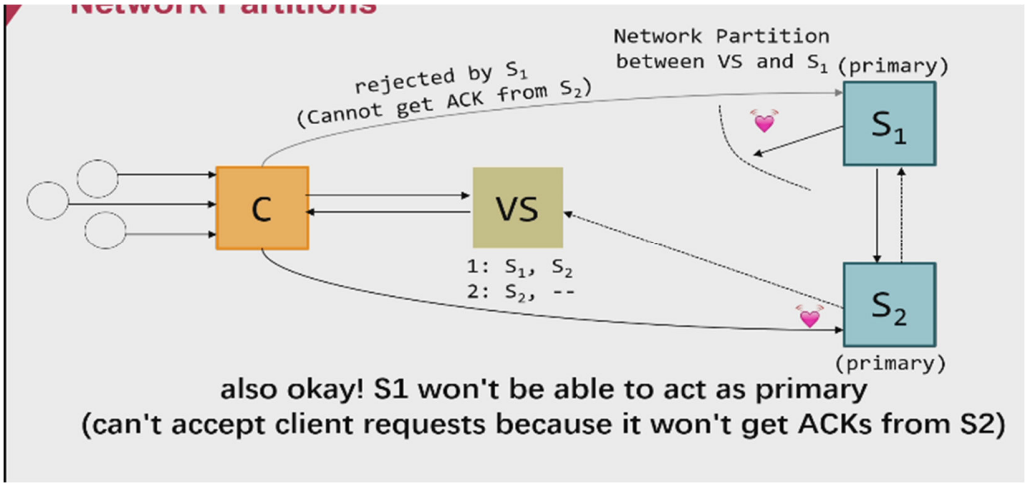
Primary-backup can trivially support single-copy, like what we did in linearizability, but what is the problem of our previous protocol ?

5

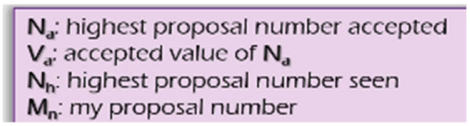
view server

- 为了发现挂了的机器，需要听心跳，如果 miss n 次 ping,则认为该服务器挂了
- 当 primary failed,view server 指定 S2 为 primary
- backup 会拒绝来自客户端的请求
- view server 会重新选择 backup

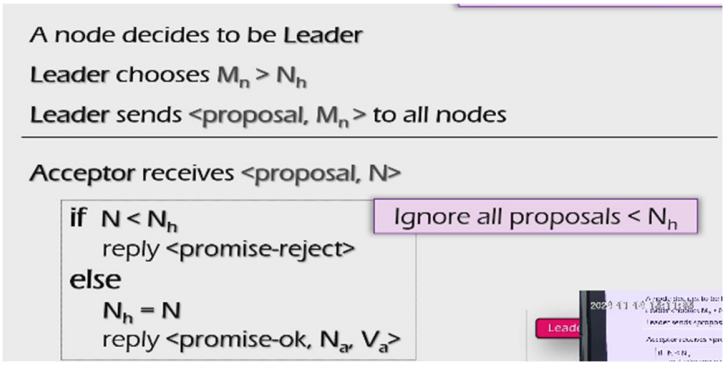
假设一种情况：s1 与 vs 断联，vs 指定 s2 为 primary,这时候 c 向 s1 请求会失败，因为 s1 执行前会先向 backup-s2 请求，但 s2 此时认为自己是 primary，拒绝 s1 的请求，而后 s1 拒绝 c 的请求：



paxos



- 一个 leader 选择一个比见过的 N 都打得值 M 并发给 accepter
- receiver：如果 M<Nr,拒绝，否则接受



- 若接受的 V！=null,选取 V 为最大 Na 的 V(若 N=3->v=bar,但 N=4->v=null，则设 V 为 bar),若接受的 V=null,可选取任意的 V,然后发给所有节点
-

If Leader gets promise-ok from a majority

if V != null, V = the value of the highest N_a received
if V = null, then Leader can pick any V
send <accept, M_n, V> to all nodes

If Leader fails to get majority promise-ok

delay and restart Paxos

- receiver

Upon receiving <accept, N, V>

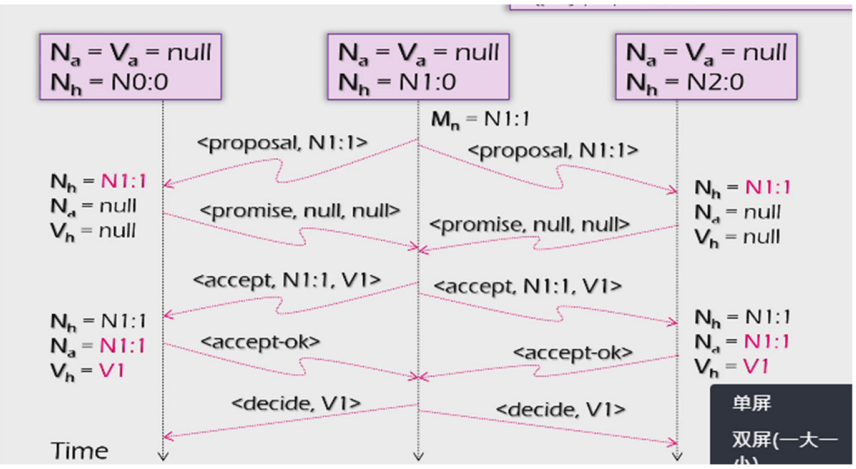
if $N < N_h$
reply <accept-reject>
else
$N_a = N$; $V_a = V$; $N_h = N$;
reply <accept-ok>

If Leader gets accept-ok from a majority

send <decide, V_a> to all nodes

If Leader fails to get majority accept-ok

delay and restart Paxos



情况三：提案未提交，Proposer 不可见 当然，另外一种可能的结果是 S5 提案时 Promise 应答中并未包含批准过 X 的决策节点，譬如应答 S5 提案时，节点 S1 已经批准了 X，节点 S2 、S3 未批准但返回了 Promise 应答，此时 S5 以更大的提案 ID

获得了 S3、S4、S5 的 Promise，这 3 个节点均未批准过任何值，那么 S3 将不会再接收来自 S1 的 Accept 请求，因为它的提案 ID 已经不是最大的了，这 3 个节点将批准 Y 的取值，整个系统最终会对“取值为 Y”达成一致，如图下图所示。

3. **Previous value not chosen, new proposer doesn't see it:**
   - New proposer chooses its own value
   - Older proposal blocked

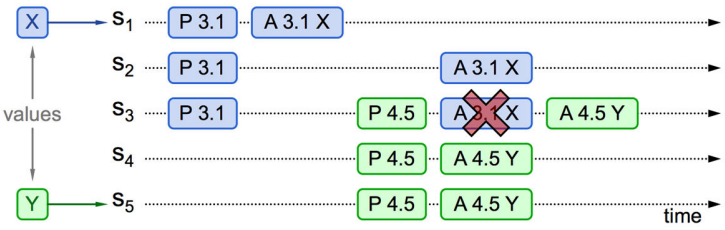
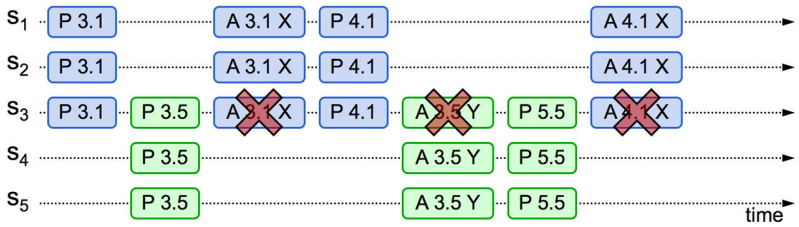


图 6-11 提案未提交，Proposer 不可见

情况四：出现活锁

从情况三可以推导出另一种极端的情况，如果 2 个提案节点交替使用更大的提案 ID 使得准备阶段成功，但是批准阶段失败的话，这个过程理论上可以无限持续下去，形成活锁（Live Lock）。例如 S3、S4、S5 拿着更高的提交号导致 S1、S2、S3 的 accept 被拒绝重新进行提交，又把 S3、S4、S5 给拒绝了，提议者没有看到先前提议的情况下，当 S1 发现自己的提议没有通过，就会发起新一轮 Prepare RPC，然后就有可能又封锁了 S5 的提议，S5 又会回到 Prepare 阶段，有概率双方都轮流封锁对方的协议，导致无法达成共识。

- **Competing proposers can livelock:**



- **One solution: randomized delay before restarting**
  - Give other proposers a chance to finish choosing

解决这个问题的办法就是把重试时间进行一些随机化，减少这种巧合发生，或者把重试的时间指数增长等等。

提问：

Why setups multiple acceptors?

- Failure of the single acceptor halts decision

Why not accepts the first proposal and rejects the rest?

- Multiple leaders result in no majority accepting
- Leader dies

What if more than one leader is active?

- Can both leaders see a majority of promises?

第三个答案：不会

When is the value V chosen?

- Leader receives a majority <promise, …>
- A majority acceptors receive <accept, N, V>
- Leader receives a majority <accepted, …>

What if acceptor fails after sending promise?

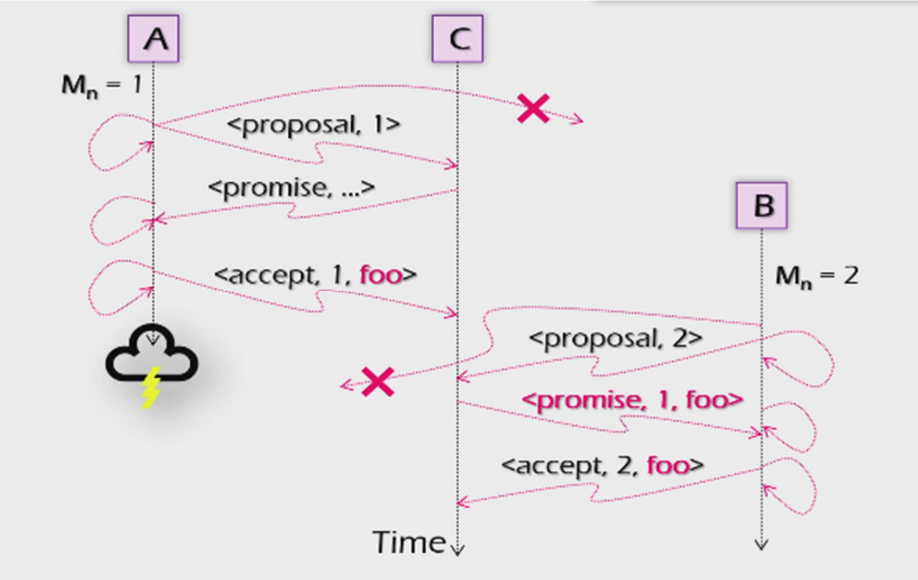
- Must remember $\mathbf{N_h}$

What if acceptor fails after receiving accept?

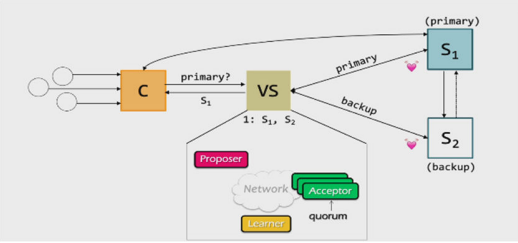
- Must remember $\mathbf{N_h}$ and $\mathbf{N_a}$ $\mathbf{V_a}$

What if leader fails while sending accept?

- Propose $\mathbf{M_n}$ again



在 2PC 的应用



Paxos Summary

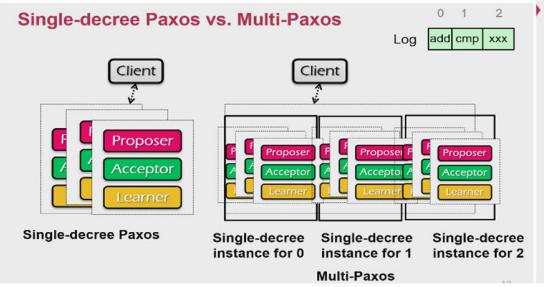
Paxos allows us to ensure consistent (total) ordering over a set of events in a group of nodes

- Events = commands / actions / state updates

Each machine will have the latest state or a previous version of the state

Multi-paxos

构建多个 paxos 以决定多个值



Summary of Multi-Paxos

Based on single-decree Paxos, realizing RSM

- i.e., agree on a sequence of values

Needs non-trivial application efforts to make it work

- E.g., should use a leader for better performance
- Should fill the holes of the log entries
- Paxos itself is not friendly to the programmers