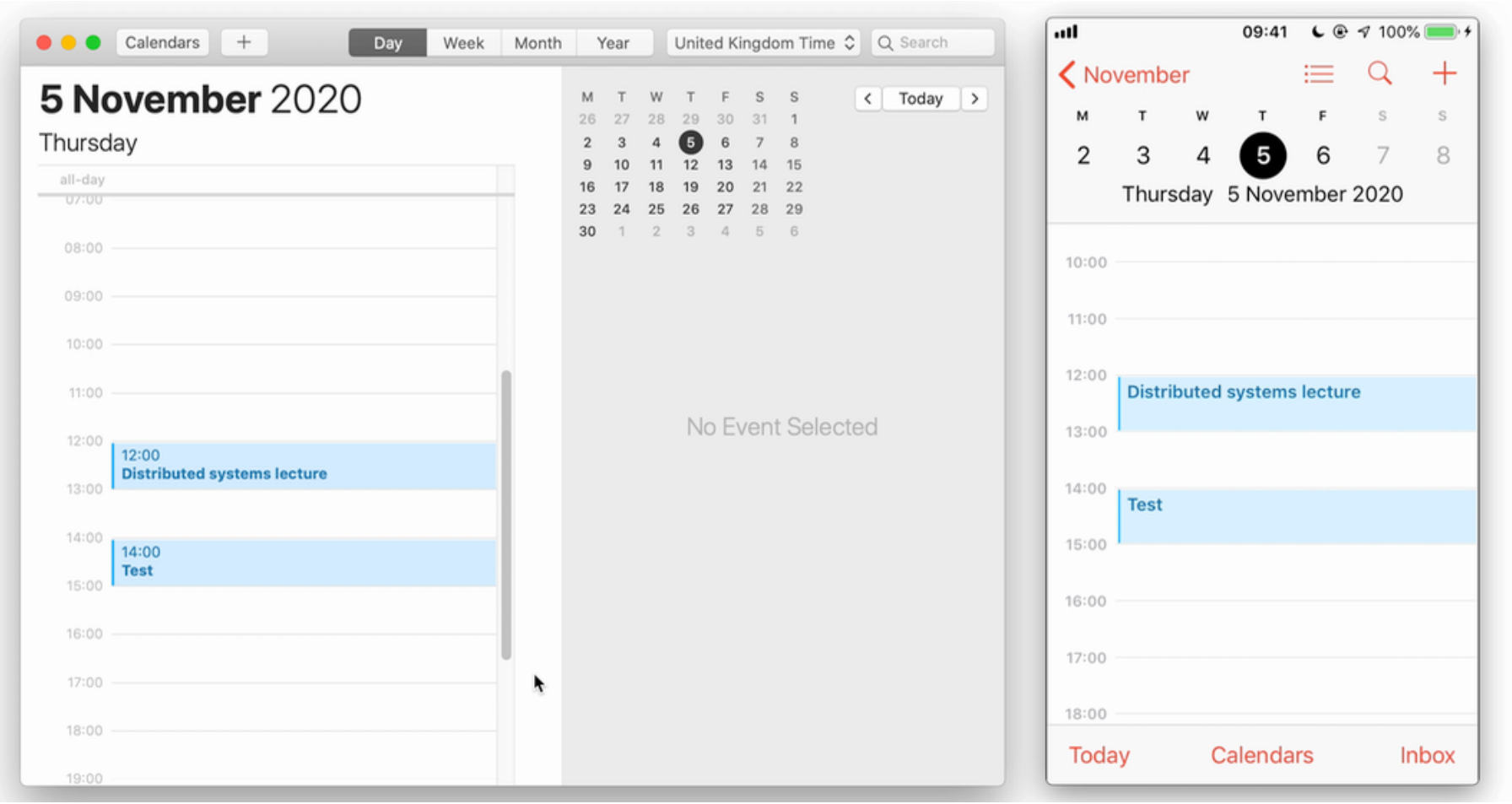
最终一致性

赵来平
天津大学软件学院

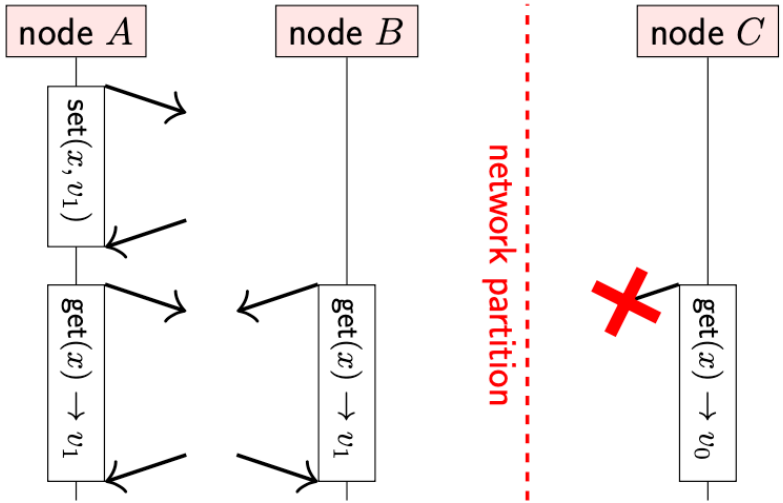
最终一致性

- **线性一致性的优点：**
  - 让一个分布式系统**看起来就像不是分布式的一样**（好像只有一个节点在处理所有操作）。
  - 对应用开发者来说，语义**简单、好理解、好使用。**
- **缺点：**
  - 性能开销大：需要大量消息交互，并且要**等待各方响应。**
  - 可扩展性受限：**leader 容易成为瓶颈。**
  - 可用性问题：如果**无法联系到一个仲裁多数（quorum）节点**，就**无法处理任何操作。**
- **最终一致性（eventual consistency）：是一种比线性一致性弱的模型，在性能、可用性和一致性之间做了不同的权衡选择。**


Calendars
Day
Week
Month
Year
United Kingdom Time
Search
5 November 2020
Thursday
all-day
12:00
Distributed systems lecture
14:00
Test
Today
No Event Selected
09:41
100%
November
Thursday 5 November 2020
Distributed systems lecture
Test
Today
Calendars
Inbox

CAP定理

- 在发生网络分区（Partition）的时候，一个系统要么选择保持强一致性（Consistent，通常指线性一致性），要么选择保持可用性（Available），不可能在网络分区存在时同时完全满足这两者。



- C 要么无限期地等待网络恢复，要么返回一个可能已经过期（陈旧）的值。

最终一致性

- **副本只根据自己的本地状态来处理操作。如果之后不再有新的更新发生，那么所有副本最终都会收敛到相同的状态。（只是无法保证要花多长时间。）**
- **强最终一致性（Strong eventual consistency）：**
  - **最终送达（Eventual delivery）**：对任意一个**没有发生故障的副本**所做的每一次更新，最终都会被**所有未故障的副本**处理到。
  - **收敛（Convergence）**：任意两个副本，只要它们已经处理了**同一组更新**，那么它们就会处于**相同的状态**，
    （即使这些更新在两个副本上的处理顺序不同）。
- **性质：**
  - 不需要为了一致性而**等待网络通信完成。**
  - 可以用**因果广播**（或者更弱的广播方式）来传播更新。
  - 并发更新不可避免会产生**冲突**，需要有相应的**冲突解决机制。**

对系统模型最低要求的总结

问题（Problem）	必须等待的通信范围（Must wait for communication）	所需同步假设（Requires synchrony）
原子提交（atomic commit）	所有参与节点（all participating nodes）	部分同步（partially synchronous）
共识、全序广播、线性一致 CAS（consensus, total order broadcast, linearizable CAS）	仲裁多数（quorum）	部分同步（partially synchronous）
线性一致的 get/set	仲裁多数（quorum）	异步（asynchronous）
最终一致性、因果广播、FIFO 广播（eventual consistency, causal broadcast, FIFO broadcast）	仅本地副本（local replica only）	异步（asynchronous）

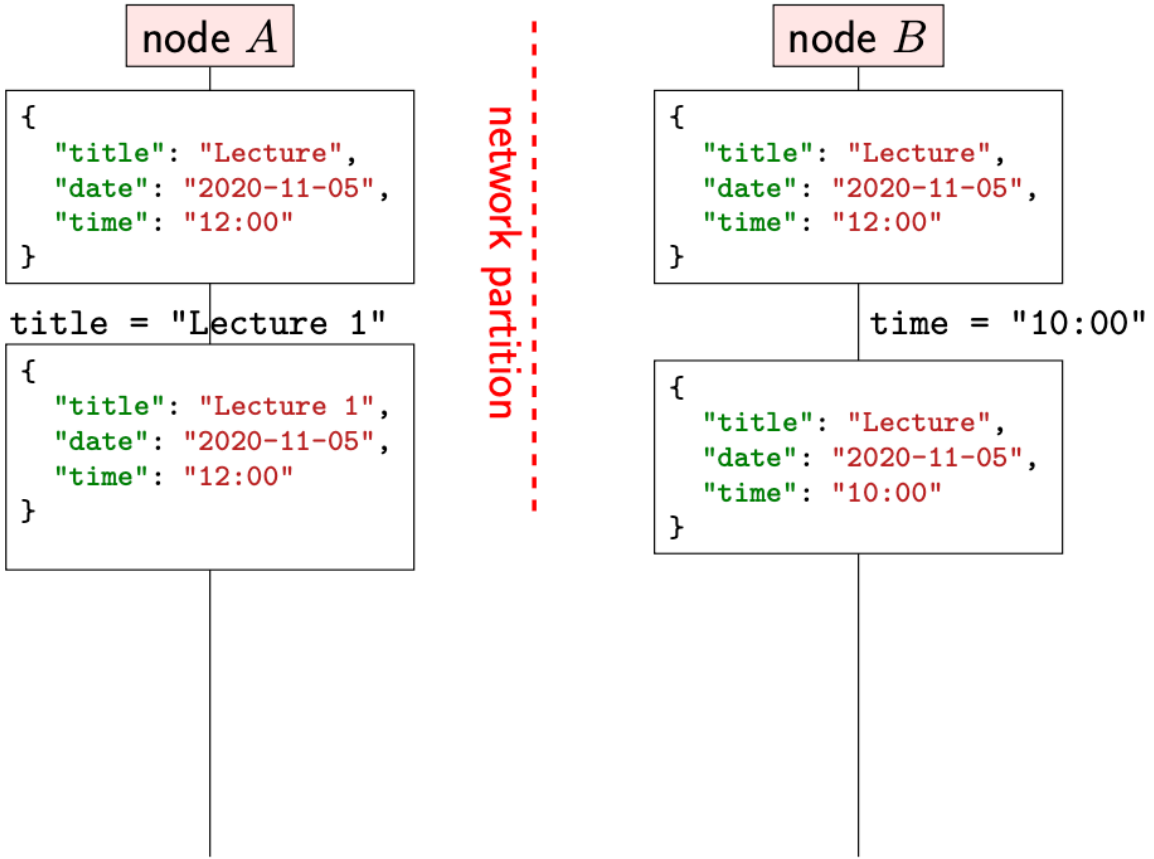
本地优先软件

- **终端用户的设备本身就是一个完整副本，服务器只是作为备份存在。**
- **以一个支持多设备同步的日历应用为例：**
  - 应用在离线时也能工作（既可以读取数据，也可以修改数据）
  - 快速：不需要等待网络往返延迟
  - 在线时与其他设备进行同步
  - 支持与其他用户进行实时协作
  - 具有持久性：即使云服务关闭，你在自己电脑上仍然保留有文件副本
  - 支持端到端加密，从而提供更好的安全性
  - 比基于 RPC 的编程模型更简单
  - 用户对自己的数据拥有更多控制权与自主权

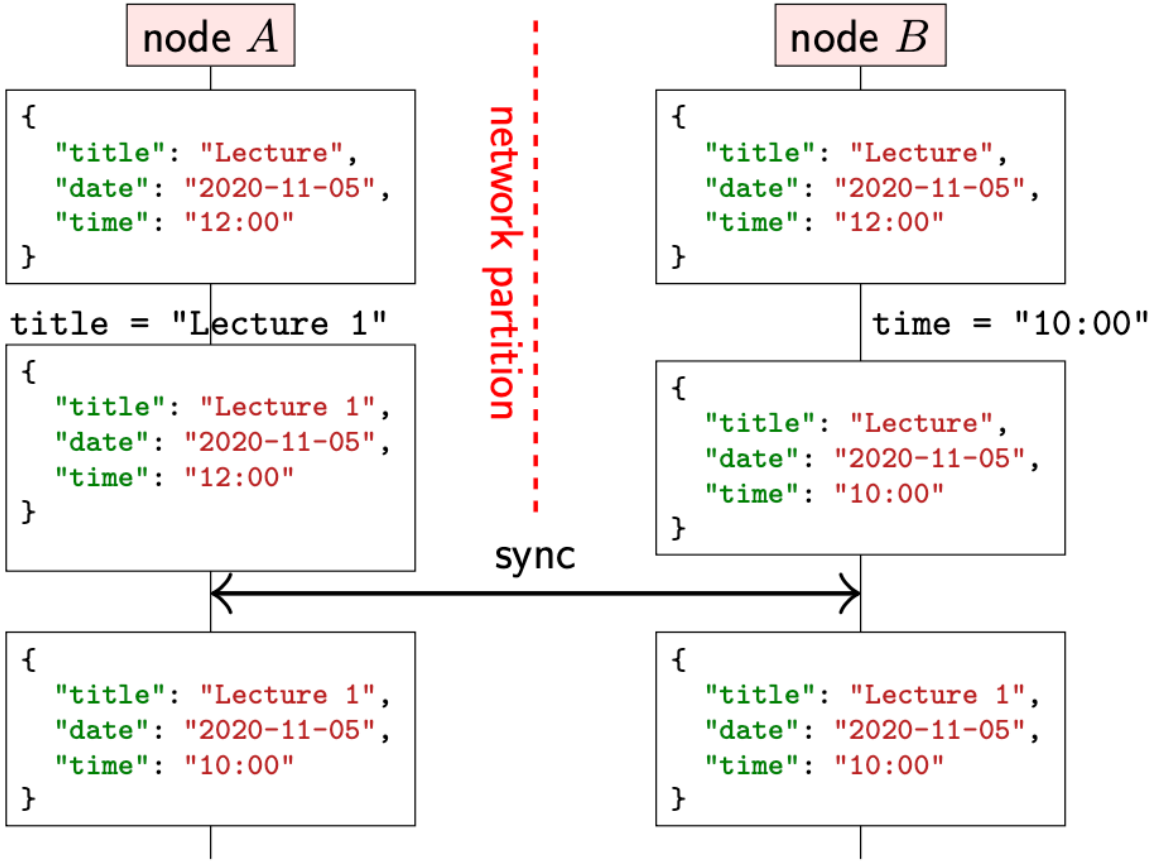
协作与冲突解决

- **现在我们使用了大量的协作类软件：**
  - 例如：日历同步、文本文档编辑器（Google Docs）、表格、演示文稿、图形编辑应用等等。
  - 多个用户 / 多台设备共同编辑同一个共享文件或文档。
  - 每个用户的设备上都保存有一份本地数据副本。
  - 对本地副本进行**乐观更新**，然后再**异步**与其他副本进行同步（因为等网络往返太慢）。
  - 难点：**如何调和并发更新之间的冲突？**
- **主要的算法家族包括：**
  - **CRDT（无冲突复制数据类型，Conflict-free Replicated Data Types）**
    - **基于操作的（operation-based）方法**
    - **基于状态的（state-based）方法**
  - **操作变换（Operational Transformation, OT）**

并发更新导致的冲突


node A
{
"title": "Lecture",
"date": "2020-11-05",
"time": "12:00"
}
title = "Lecture 1"
{
"title": "Lecture 1",
"date": "2020-11-05",
"time": "12:00"
}
network partition
node B
{
"title": "Lecture",
"date": "2020-11-05",
"time": "12:00"
}
time = "10:00"
{
"title": "Lecture",
"date": "2020-11-05",
"time": "10:00"
}

并发更新导致的冲突


node A
node B
network partition
{
"title": "Lecture",
"date": "2020-11-05",
"time": "12:00"
}
title = "Lecture 1"
time = "10:00"
{
"title": "Lecture 1",
"date": "2020-11-05",
"time": "12:00"
}
{
"title": "Lecture",
"date": "2020-11-05",
"time": "10:00"
}
sync
{
"title": "Lecture 1",
"date": "2020-11-05",
"time": "10:00"
}
{
"title": "Lecture 1",
"date": "2020-11-05",
"time": "10:00"
}

基于操作的 Map 型 CRDT

on initialisation **do**
    $values := \{\}$
end on

on request to read value for key k **do**
    if $\exists t, v.\ (t, k, v) \in values$ **then return** v **else return** null
end on

on request to set key k to value v **do**
    $t := \text{newTimestamp}()$ ▷ globally unique, e.g. Lamport timestamp
    broadcast (set, t, k, v) by reliable broadcast (including to self)
end on

on delivering (set, t, k, v) by reliable broadcast **do**
    $previous := \{(t', k', v') \in values \mid k' = k\}$
    if $previous = \{\} \ \vee\ \forall (t', k', v') \in previous.\ t' < t$ **then**
        $values := (values \setminus previous) \cup \{(t, k, v)\}$
    end if
end on

基于操作的 CRDT

- **可靠广播可以以任意顺序交付更新操作：**
  - broadcast (set, t1, "title", "Lecture 1")
  - broadcast (set, t2, "time", "10:00")
- **回顾强最终一致性（strong eventual consistency）：**
  - **最终送达（Eventual delivery）**：对任意一个未故障副本所做的每一次更新，最终都会被每一个未故障副本处理到。
  - **收敛（Convergence）**：任意两个副本，只要它们已经处理了**同一组更新操作**，它们的状态就是相同的。
- **CRDT 算法是这样实现这些性质的：**
  - 可靠广播确保每个操作最终都会被交付到每一个（未崩溃的）副本。
  - 应用这些操作是**可交换的（commutative）**，所以交付顺序**无关紧要。**

基于状态的 Map 型 CRDT

- 运算符 $\sqcup$ 用如下方式将两个状态 s1 和 s2 进行合并：

$$s_1 \sqcup s_2 = \{(t,k,v) \in (s_1 \cup s_2) \mid \nexists (t',k',v') \in (s_1 \cup s_2).\ k' = k \wedge t' > t\}$$

on initialisation **do**
    $values := \{\}$
end on

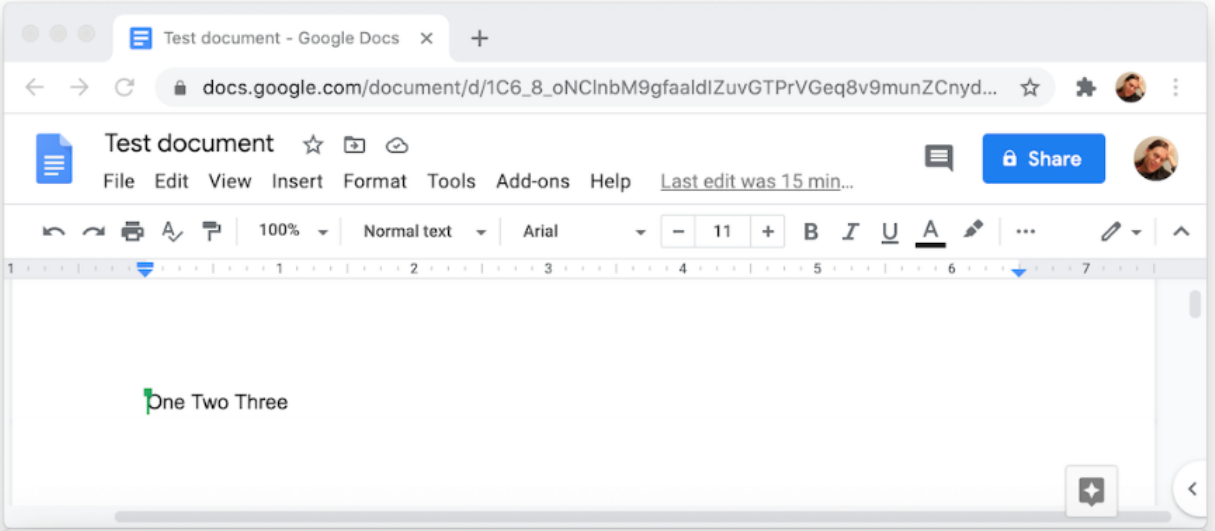
on request to read value for key k **do**
    if $\exists t, v.\ (t,k,v) \in values$ **then return** v **else return** null
end on

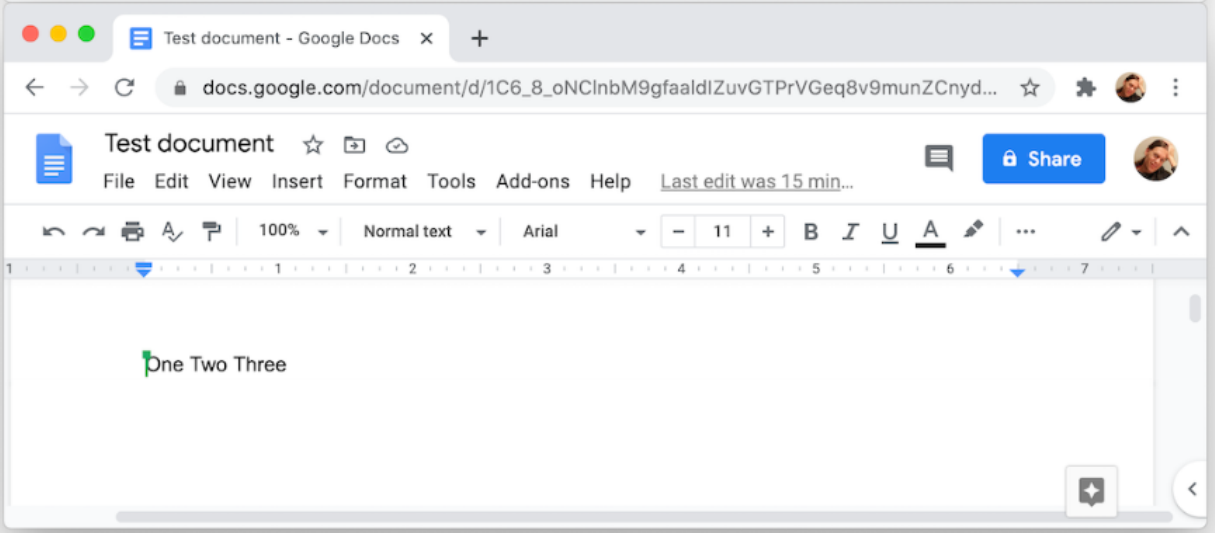
on request to set key k to value v **do**
    $t := \text{newTimestamp}()$ ▷ globally unique, e.g. Lamport timestamp
    $values := \{(t',k',v') \in values \mid k' \neq k\} \cup \{(t,k,v)\}$
    broadcast $values$ by best-effort broadcast
end on

on delivering V by best-effort broadcast **do**
    $values := values \sqcup V$
end on

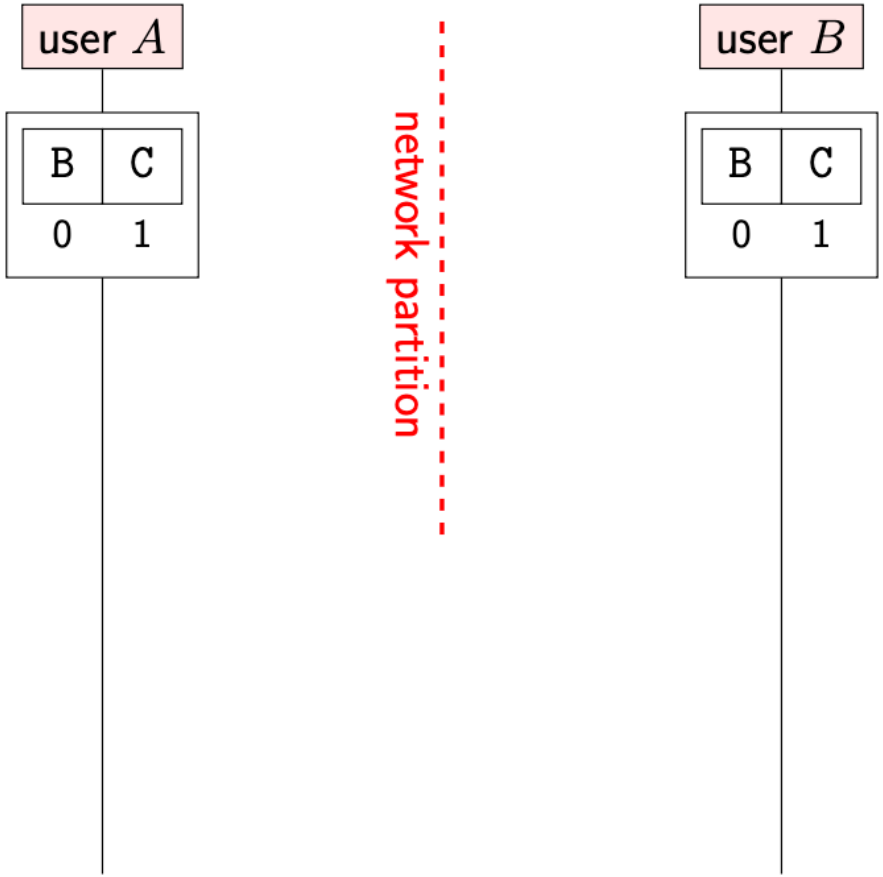
基于状态的CRDT

- **合并运算符 ⊔ 必须满足：对所有状态 (s_1, s_2, s_3) …**
  - **交换律（Commutative）**：(s_1 ⊔ s_2 = s_2 ⊔ s_1)。
  - **结合律（Associative）**：((s_1 ⊔ s_2) ⊔ s_3 = s_1 ⊔ (s_2 ⊔ s_3))。
  - **幂等性（Idempotent）**：(s_1 ⊔ s_1 = s_1)。
- **基于状态 vs 基于操作：**
  - 基于操作的 CRDT 通常**消息更小**；
  - 基于状态的 CRDT 则能**容忍消息丢失 / 重复**。
- **并不一定必须使用广播：**
  - 也可以在其他场景下合并并发更新的副本，例如：仲裁复制（quorum replication）、反熵协议（anti-entropy）等。

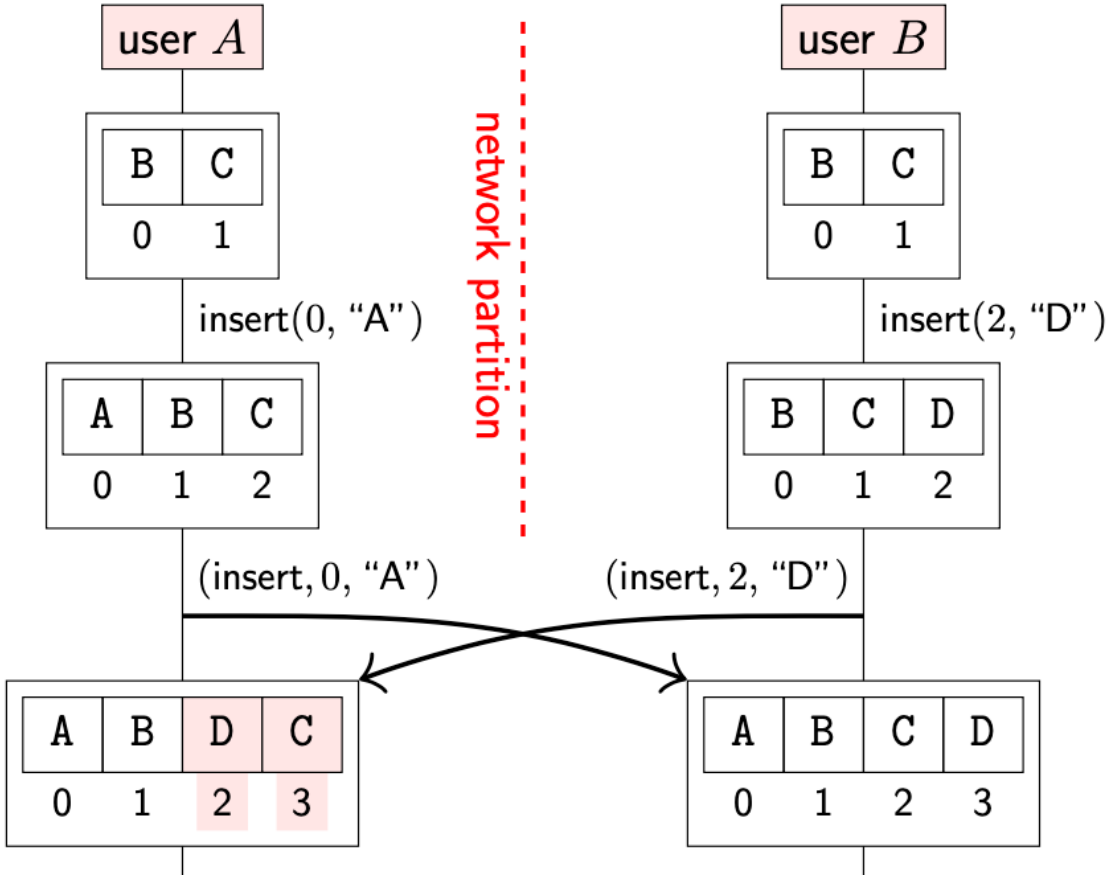

Test document - Google Docs
docs.google.com/document/d/1C6_8_oNClnbM9gfaaldIZuvGTPrVGeq8v9munZCnyd...
Test document
File Edit View Insert Format Tools Add-ons Help
Last edit was 15 min...
Share
100%
Normal text
Arial
11
One Two Three


Test document - Google Docs
docs.google.com/document/d/1C6_8_oNClnbM9gfaaldIZuvGTPrVGeq8v9munZCnyd...
Test document
File Edit View Insert Format Tools Add-ons Help
Last edit was 15 min...
Share
100%
Normal text
Arial
11
One Two Three

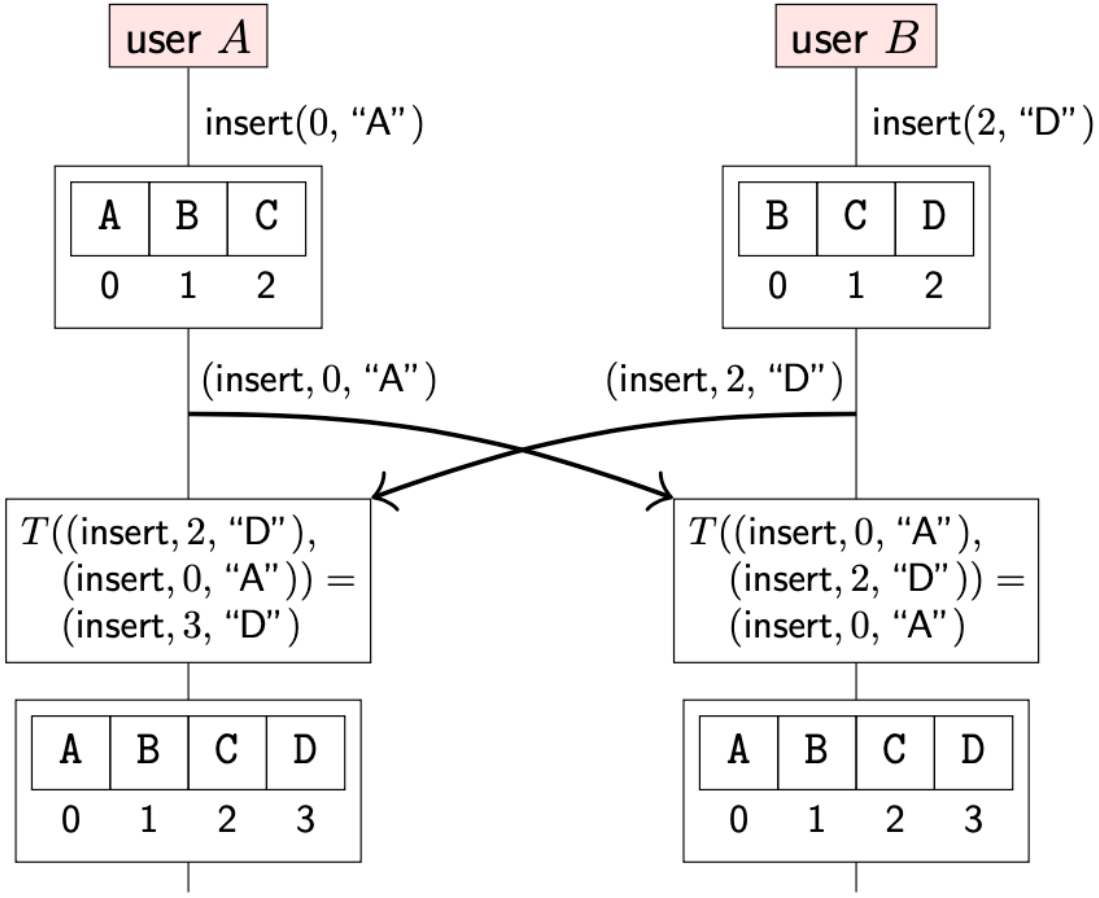
协同文本编辑：要解决的问题



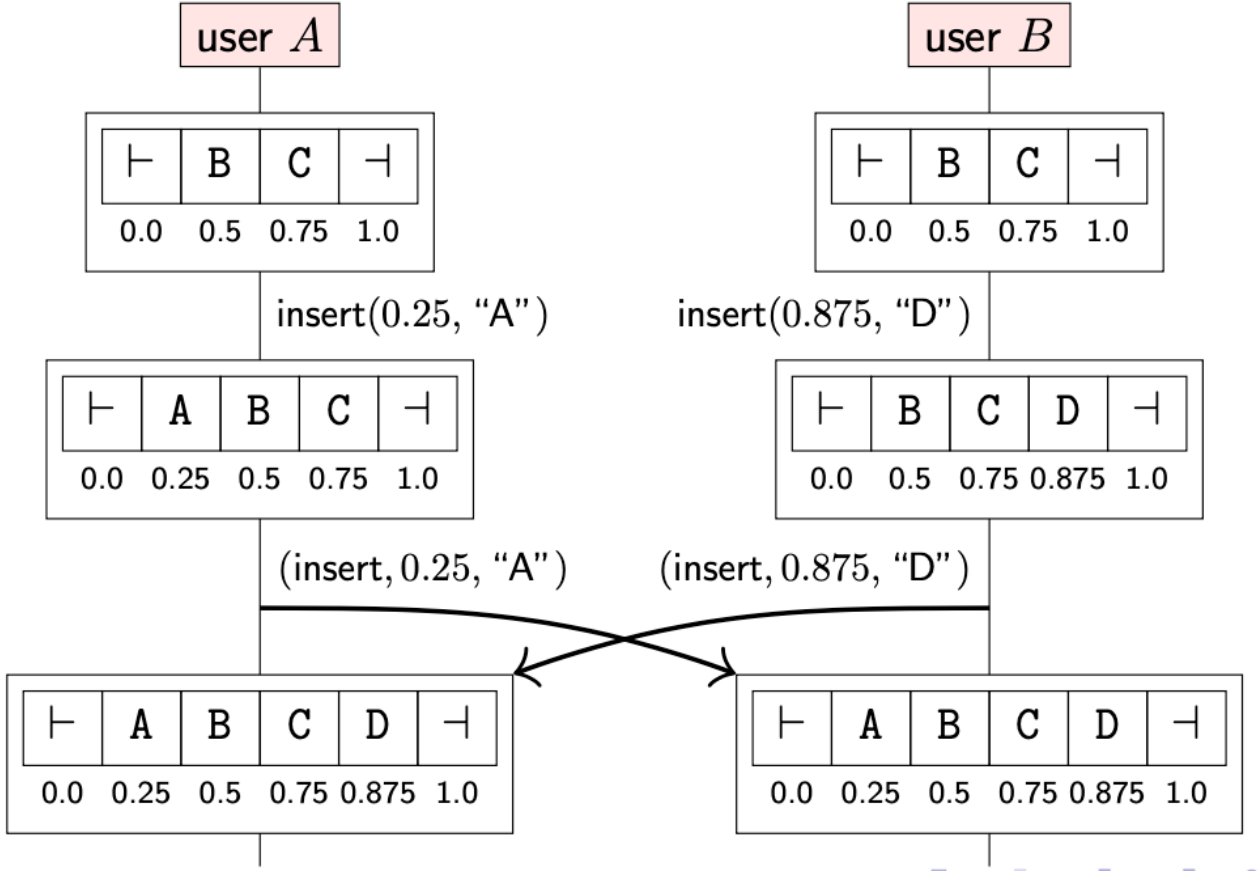
协同文本编辑：要解决的问题



基于操作变换的协同编辑



文本编辑型 CRDT


user A
user B
⊢ B C ⊣
0.0 0.5 0.75 1.0
insert(0.25, "A")
insert(0.875, "D")
⊢ A B C ⊣
0.0 0.25 0.5 0.75 1.0
⊢ B C D ⊣
0.0 0.5 0.75 0.875 1.0
(insert, 0.25, "A")
(insert, 0.875, "D")
⊢ A B C D ⊣
0.0 0.25 0.5 0.75 0.875 1.0

基于操作的文本 CRDT

function $\textsc{ElementAt}(chars, index)$
  $min =$ the unique triple $(p, n, v) \in chars$ such that
    $\nexists (p', n', v') \in chars.\ p' < p \vee (p' = p \wedge n' < n)\}$
  if $index = 0$ **then return** min
  else return $\textsc{ElementAt}(chars \setminus \{min\}, index - 1)$
end function

on initialisation **do**
  $chars := \{(0, \text{null}, \vdash), (1, \text{null}, \dashv)\}$
end on

on request to read character at index $index$ **do**
  let $(p, n, v) := \textsc{ElementAt}(chars, index + 1)$; **return** v
end on

on request to insert character v at index $index$ at node $nodeId$ **do**
  let $(p_1, n_1, v_1) := \textsc{ElementAt}(chars, index)$
  let $(p_2, n_2, v_2) := \textsc{ElementAt}(chars, index + 1)$
  broadcast $(\text{insert}, (p_1 + p_2)/2, nodeId, v)$ by causal broadcast
end on

基于操作的文本 CRDT

on delivering (insert, p, n, v) by causal broadcast **do**
    $chars := chars \cup \{(p, n, v)\}$
end on

on request to delete character at index $index$ **do**
    let $(p, n, v) := \textsc{ElementAt}(chars, index + 1)$
    broadcast (delete, p, n) by causal broadcast
end on

on delivering (delete, p, n) by causal broadcast **do**
    $chars := \{(p', n', v') \in chars \mid \neg(p' = p \wedge n' = n)\}$
end on

- **使用因果广播，保证某个字符的插入操作，一定会在它的删除操作之前被投递。**
- **对于不同字符的插入和删除操作，它们是可交换的（即执行顺序可以互换，结果不变）。**

- PPT部分内容来自于剑桥大学和UIUC cst.cam.ac.uk/teaching/2526/ConcDisSys/