南京大学

本 科 毕 业 论 文

院　　系　计算机科学与技术系

专　　业　计算机科学与技术

题　　目　CRDT 协议的模型检验——使用 TLA+工具

年　　级　2015 级　　学　号　151220117

学生姓名　王芷芙

指导教师　魏恒峰　　职　称　助理研究员

提交日期　2019 年 5 月 24 日

南京大学本科生毕业论文 (设计、作品) 中文摘要

题目：CRDT 协议的模型检验——使用 TLA$^+$ 工具

院系：计算机科学与技术系

专业：计算机科学与技术

本科生姓名：王芷芙

指导教师（姓名、职称）：魏恒峰 助理研究员

摘要：

CRDT（Conflict-free Replicated Data Types, 无冲突复制数据类型）是一种封装了冲突消解策略的分布式复制数据类型，它能保证分布式系统中副本节点间的强最终一致性，即执行了相同更新操作的副本节点具有相同的状态。CRDT 协议设计精巧，不易保证其正确性。相关文献表明，有些公开发表的、甚至是经过证明的 CRDT 协议后来却被发现是错误的。

本文采用模型检验技术验证一系列 CRDT 协议的正确性。具体而言，我们构建了一个可扩展的 CRDT 协议描述与验证框架，包括网络通信层、协议接口层、具体协议层与规约层。网络通信层描述副本节点之间的通信模型，实现了多种类型的通信网络。协议接口层为已知的 CRDT 协议（分为基于操作的协议与基于状态的协议）提供了统一的接口。在具体协议层，用户可以根据协议的需求选用合适的底层通信网络。规约层则描述了所有 CRDT 协议都需要满足的强最终一致性。我们使用 TLA$^+$ 形式化规约语言实现了该框架，然后以 Counter 与 Add-Wins Set 两种 CRDT 为例展示了如何使用框架描述具体协议并使用 TLC 模型检验工具验证协议的正确性。

关键词： 无冲突复制数据类型 CRDT；强最终一致性；模型检验；TLA$^+$

南京大学本科生毕业论文 (设计、作品) 英文摘要

THESIS: Model Checking CRDT Protocols using TLA$^+$
DEPARTMENT: Department of Computer Science and Technology
SPECIALIZATION: Nanjing University
UNDERGRADUATE: Zhifu Wang
MENTOR: Hengfeng Wei (Research Assistant)
ABSTRACT:

Conflict-free Replicated Data Types (CRDT) is a replicated data structure that encapsulates methods to resolve inconsistencies. It guarantees strong eventual consistency of all replicas in distributed systems, that is, any two nodes that have received the same (unordered) set of updates will be in the same state. However, CRDT protocols are subtle and thus it is difficult to ensure their correctness. Relevant literature indicates that several formal, even published "proofs" turned out to be erroneous.

This paper leverages model checking to verify the correctness of a class of CDRT protocols. Concretely, we build a reusable framework that models and verifies CRDT protocols. The framework contains the communication layer, the interface layer, the protocol layer, and the specification layer. The communication layer models the communication between replicas and implements varieties of communication networks. The interface layer provides a uniform interface for existing CRDT protocols (operation-based and state-based protocols). In the protocol layer, users can select the appropriate underlying communication network required by the protocol. The specification layer models strong eventual consistency that every CRDT protocol should satisfy. We use TLA$^+$, a formal specification language to implement this framework. Then we show how to use the framework to model a specific protocol and verify its correctness via the model checking tool, TLC, taking Counter and Add-Wins Set as two examples.

KEY WORDS: CRDT (Conflict-free Replicated Data Types), Strong eventual consistency, Model checking, TLA$^+$

目　　次

第一章　绪论

分布式系统为了满足上层应用对高性能、高可用性与高容错性的需求，通常采用数据副本技术，将数据的多个副本存放在不同的物理节点上。然而，数据副本技术在带来诸多好处的同时，也带来了数据一致性问题：如何保证数据副本之间的一致性？为了保证系统的高可用性, 大多数分布式系统都选择仅提供弱数据一致性[1,2]。最终强一致性 (Strong Eventual Consistency)[3] 是一种典型的弱数据一致性，它要求如果两个副本节点执行了相同的更新操作集合，那么它们具有相同的状态。CRDT（Conflict-free Replicated Data Type，无冲突复制数据类型）[3,4] 将冲突消解策略封装到数据类型中, 从而实现强最终一致性。已实现的 CRDT 包括计数器 (Counter)、读写寄存器 (Read/Write Register)、集合 (Set)、列表 (List)、哈希表 (Hash Table)、树 (Tree) 与图 (Graph) 等[4–6]。本文的主要工作是使用模型检验[7] 形式化方法验证一系列 CRDT 协议的正确性。

1.1　研究背景

分布式系统通常采用数据副本技术，将数据的多个副本存放在不同的物理节点（称为副本节点）上，以满足上层应用对高性能、高可用性与高容错性的需求[4,6,8]。为了保证低延迟，用户提交到某个副本节点的操作需要立即返回，而不需要先与其它副本节点通信。本地副本节点上的更新操作将以异步的方式发送到其它副本节点。在这种情况下，多个副本节点上的并发更新操作将产生冲突，导致数据一致性问题。

数据一致性模型多种多样，且有强弱之分[9]。为了保证系统的高可用性与高性能，很多分布式系统都选择弱数据一致性，如最终一致性与最终强数据一致性。最终一致性[10,11] 保证了客户端停止发送更新请求后，各个副本节点最终达到一致的状态。但是，它对系统的中间状态没有任何约束。强最终一致性[3] 比最终一致性稍强，它要求执行了相同的更新操作的节点具有相同的状态。

CRDT（Conflict-free Replicated Data Types，无冲突复制数据类型)[3,4] 是一种抽象分布式数据类型，它封装了并发冲突的消解策略，向上层应用提供

所需的强最终数据一致性。已实现的 CRDT 包括计数器 (Counter)、读写寄存器 (Read/Write Register)、集合 (Set)、列表 (List)、哈希表 (Hash Table)、树 (Tree) 与图 (Graph) 等[4–6]。NoSQL 分布式数据库，如开源的 Riak [①] 以及 Redis [②]的商用版本都提供了对 CRDT 的支持。

目前，CRDT 在大规模分布式系统中已得到广泛应用。例如，使用 *Elixir* 语言搭建的网络框架 Phoenix，使用 CRDTs 支持实时多节点信息共享 [③]；TomTom 使用 OUR-set (Observed, Updated, Removed) 对 CRDT-set 的更新功能进行扩展 [④]，实现了同一用户多个设备之间的导航数据同步。再如，《英雄联盟》在游戏内聊天系统中使用 Riak 实现的 CRDT，支持 750 万名用户同时使用和每秒 11000 条的消息通信；Bet365 (欧洲最大的线上赌博公司，高峰期达 250 万名用户同时在线) 使用 Riak 实现的 OR-Set 存储上成百上千兆字节的数据。

1.2 研究工作

CRDT 协议通常需要精巧的设计，正确性难以理解也难以保证[12]。定理证明与模型检验这两种形式化方法可以有效地提高 CRDT 协议正确性的可靠性。定理证明的优点是可以得到可靠的结论，缺点在于使用难度大。相比而言，模型检验是一种易于使用的、能够自动化验证有限状态系统的正确性的技术[13]。对于给定的系统模型，模型检验技术遍历所有可能的执行，检查系统是否满足给定的规约。模型检验方法的不足在于它仅能验证有限规模的系统的正确性。然而，经验表明大多数错误在小规模分布式系统中就可以检测出来，比如仅需要 3 个甚至更少个副本节点[14]。因此，模型检验技术可以提供充分的可靠性保障。

本文旨在使用模型检验方法验证 CDRT 协议的正确性。具体而言，我们使用 TLA$^+$[15] 语言描述一系列 CRDT 协议，并使用模型检验工具 TLC 验证它们的正确性。为了能更好地验证一系列 CRDT 协议，我们首先提出一个 CRDT 协议描述与验证框架，它包括网络通信层、协议接口层、具体协议层与规约层。网络通信层描述副本节点之间的通信模型，实现了多种类型的通信网络。上层协议可以根据需求选用合适的通信网络。协议接口层为已知的 CRDT 协

[①] Riak: https://docs.riak.com/riak/kv/2.2.3/developing/data-types/
[②] Redis Enterprise: https://redislabs.com/
[③] https://dockyard.com/blog/2016/03/25/what-makes-phoenix-presence-special-sneak-peek
[④] https://speakerdeck.com/ajantis/practical-demystification-of-crdts

议提供了统一的接口。在具体协议层，我们用 TLA$^+$ 描述了一系列 CRDT 协议。规约层给出了强最终一致性在 CRDT 协议中的具体 TLA$^+$ 表达形式。最后，我们使用 TLC 模型检验工具验证了 CRDT 协议的正确性。

1.3 相关工作

Gomes 等人使用 Isabelle 定理证明器验证了基于操作的一系列 CRDT 协议的正确性[12]。他们开发了一个可重复使用的组合的证明框架。证明框架中包括一个网络模型，用以证明定理在所有可能的网络行为中的正确性。同时，作者还建立了一个一致性定理的抽象，为强最终一致性提供形式化定义。然后作者使用 Isabelle 定理证明器对具体的 CRDT 协议进行定理证明。

Zeller 等人使用 Isabelle 定理证明器验证了基于状态的一系列 CRDT 协议的正确性[16]。他们将每种 CRDT 抽象为四个参数以及四个组成部分，开发了一个形式化的系统模型，并基于该证明框架描述系统不同状态之间的转移关系。

操作转换（Operational Transformation; OT）技术是实现 CRDT 的另一类方法。由于 OT 技术难以理解，有些工作采用形式化方法验证基于 OT 技术的 CRDT 协议的正确性。Chengzheng 等人设计了一系列独立于具体的 OT 算法的验证框架和软件方法，遍历覆盖所有可能的转换情况，从而对 OT 的正确性进行验证[17]。Sinchuk 等人描述了包含几种 OT 算法的基于 SSReflect/Coq 的方法库，并对它们的正确性进行了证明[18]。Yang 等人使用一系列良定义的转换条件和性质，为基于 OT 的协同编辑系统形式化建模，并对包括控制算法和转换函数的 OT 的正确性进行完整的检验[19]。

然而，很多 CRDT 协议的正确性证明后被证明是错误的。有些 CRDT 协议的正确性证明中使用了似是而非的非正式推理 (例如 SOCT2 和 SDT[20])，有些给出了未经证明的断言 (例如 dOPT 和 adOPTed 断言 OT 的 TP2 性质[21,22])，还有些 ‘‘机械的形式化证明’’ 使用了错误的假设[23]。

1.4 论文组织

本文后续内容组织如下:

第 2 章介绍相关预备知识。

第 3 章介绍 CRDT 协议框架的 TLA$^+$ 描述。

第 4 章具体介绍 CRDT 协议中 Counter 和 Set 的相关描述，并对每一个数据类型按照基于操作和状态分别进行讨论。

第 5 章介绍 CRDT 的一致性规约，以及使用 TLC 工具对 CRDT 协议的正确性验证。

第 6 章是对本文工作的总结和展望。

第二章　预备知识

本章介绍预备知识，包括系统模型、CRDT 简介与 TLA$^+$ 简介。

2.1　系统模型

分布式系统通常采用数据副本技术，将数据的多个副本存放在不同的副本节点上。用户提交到某个副本节点的操作无需先与其它副本节点通信，而是立即返回；本地副本节点上的更新操作以异步的方式发送到其它副本节点。

基于该系统模型，我们给出无冲突复制数据类型的形式化表述。具体地，我们将无冲突复制数据类型 τ 形式化为多元组 $D_\tau = (\Sigma, M, init, do, send, deliver)$。其中，

- 集合 Σ 是副本节点状态集。
- 集合 M 是副本节点之间通信的消息集。
- 函数 $init : ReplicaID \to \Sigma$ 定义了每个副本节点的初始状态。
- 函数 $do : \Sigma \times Op_\tau \times Timestamp \to \Sigma \times Val_\tau$ 定义了副本节点本地更新的相关方法。
- 函数 $send : \Sigma \to \Sigma \times M$ 和函数 $deliver : \Sigma \times M \to \Sigma$ 描述了副本节点之间异步更新的原语。

2.2　CRDT 简介

CRDT 是封装了自动冲突消解策略的抽象数据类型，用以保证分布式系统中所有副本节点的强数据一致性[4]。在 CRDT 结构中，各个副本节点可以独立地并发更新，而不与其他副本节点进行协同，它可以保证接收了相同的更新集合的副本节点最终到达相同的确定性状态，也就是说，不同副本节点之间的冲突总是可以得到解决。

CRDT 分为两种类型：基于操作的无冲突复制数据类型（Operation-based CRDTs）和基于状态的无冲突复制数据类型（State-based CRDTs）[4]。在消息传递方面，前者仅将最近一次更新操作发送给其它副本节点，而后者则发送最

新的节点状态。这两种 CRDT 是等价的，可以相互模拟实现，但是二者对通信层的要求和传输的状态规模有一定差异。基于操作的无冲突复制数据类型对通信层有额外的要求，即当某个副本将操作传输给其他副本时，通信层不可以发生丢包或者重传，而传输时不同操作之间的顺序不受约束限制。在并发的更新操作满足可交换性的条件下，此类 CRDT 可以保证数据的强最终一致性。相对地，基于状态的无冲突复制数据类型对通信层没有特殊要求。在状态合并 (Merge) 函数具有格结构 (更具体地讲是 Join-Semilattice 结构) 下，此类 CRDT 可以保证强最终数据一致性。

2.3 TLA+ 简介

TLA+ 是 Lamport 设计的形式化规约语言，用于对并发系统和分布式系统的设计、建模、记录和验证[15]。在 TLA+ 中，系统被建模为状态机，由初始状态（initial state）和动作（action）描述。状态（state）是数值（value）对变量（variable）的赋值，动作（action）是旧状态和新状态之间的关系，用包含不加撇的变量（表示旧状态）和加撇的变量（表示新状态）的公式描述。例如，$x' = y + 42$ 断言了新状态中 x 的值比旧状态中 y 的值大 42。

TLA+ 基于 TLA（the Temporal Logic of Actions）[24]，使用 TLA+ 规约的程序形式上为时序公式 $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$，其中谓词 $Init$ 规约了程序所有可能的初始状态，$Next$ 规约了程序的次状态关系，$\Box$ 是表示“总是”的时序操作符，$vars$ 是程序中使用的所有变量的多元组，L 是一种活跃性性质。

本文中的涉及的 TLA+ 操作如表 2-1。

表 2-1: 本文中使用的 TLA+ 操作

	Operators	Meaning
Logic	CHOOSE $x \in S : p$	An x in S satisfying p
Sets	SUBSET S	Powerset (i.e., set of subsets) of S
	$\{ e : x \in S \}$	Set of elements e such that x is in S
	$\{ x \in S : p \}$	Set of elements x in S satisfying p
Functions	$f[e]$	Function application
	$[x \in S \mapsto e]$	Function f such that $f[x] = e$ for $x \in S$
	$[S \in T]$	Set of functions mapping from S to T
	$[f$ EXCEPT $![e_1] = e_2]$	Function $\hat{f}$ equal to f except that $\hat{f}[e_1] = e_2$
	$[f$ EXCEPT $![c] = e]$, where e contains @	@ in e stands for $f[c]$
Records	$e.h$	The h-field of record e
	$[h_1 \mapsto e_1, \cdots, h_n \mapsto e_n]$	The record whose h_i field is e_i
	$[h_1 : S_1, \cdots, h_n : S_n]$	Set of all records with h_i field in S_i
	$[r$ EXCEPT $!.h = e]$	Record $\hat{r}$ equal to r except that $\hat{r}.h = e$
	$[r$ EXCEPT $!.h = e]$, where e contains @	@ in e stands for $r.h$
Tuples	$e[i]$	The i^{th} component of tuple e
	$\langle e_1, \cdots, \rangle$	The n-tuple whose i^{th} component is e_i
	$S_1 \times \cdots \times S_n$	The set of all n-tuples with i^{th} component in S_i
Sequences	$Seq(S)$	The set of all sequences of elements of the set S
	$Head(s)$	The first element of sequence s
	$Last(s)$	The last element of sequence s
	$Tail(s)$	The tail of sequence s, which consists of s with its head removed
Action Operators	e'	The value of e in the new state of an action
	UNCHANGED e	$e' = e$
	$[A]_e$	$A \vee (e' = e)$
Temporal Operators	$\Box F$	F is always true

第三章　CRDT 协议描述与验证框架

本章介绍 CRDT 协议描述与验证框架，分为网络通信层、协议接口层、具体协议层以及规约层。

3.1　CRDT 协议描述与验证框架

已有的 CRDT 协议涉及多种数据类型，如计数器（Counter）、读写寄存器（Read/Write Register）、集合（Set）、列表（List）、树（Tree）与图（Graph）等。每种复制数据类型又有多种不同语义，并且对于同一种语义，还有基于操作（Op-based）与基于状态（State-based）两类不同的协议实现方式。因此，我们需要抽取并实现各种协议的共性，构建一个具有良好的可复用性与可扩展性的 CRDT 协议验证框架。具体而言，CRDT 协议描述与验证框架分为网络通信层、协议接口层、具体协议层以及规约层，如图 3-1所示。

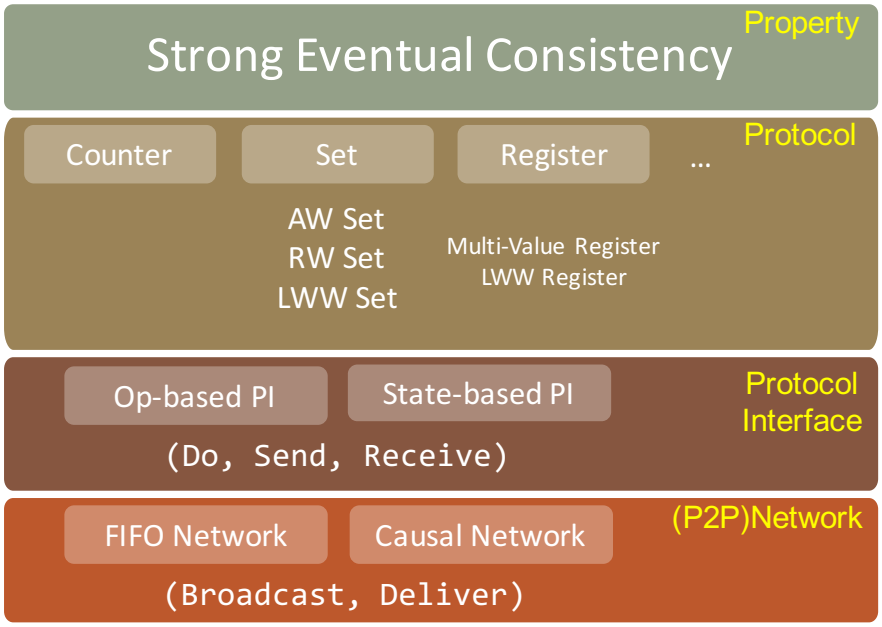

图 3-1: CRDT 协议描述与验证框架

网络通信层建模了副本节点之间的 P2P 网络连接，并提供了 CRDT 协议所依赖的各种类型的通信信道，比如保证 FIFO 性质或者因果关系（Causal）

的网络。协议接口层为对消息传输的可靠性要求相似的 CRDT 协议族提供统一的接口。具体协议层则实现该接口，并根据协议的要求选择所需的通信模块。规约层描述了所有 CRDT 协议都需要满足的强最终一致性。基于上述框架，本文使用 TLA$^+$ 描述了一系列 CRDT 协议，并验证了它们的正确性，即保证了强最终一致性。

3.2 网络通信层

网络通信层描述副本节点之间的 P2P 网络通信。在本文工作中，我们主要关注两类通信网络：满足 FIFO （First In First Out）性质的可靠（Reliable）网络和保证因果关系的（Causal）网络。可靠网络保证每条消息都会被每个节点接收且仅接收一次（exactly once）。FIFO 网络保证每个节点先后发送的消息会被其它节点按序接收。Causal 网络则保证每条消息仅能在它所依赖的消息都被接收之后才能被接收。此处的依赖关系即是 Lamport 所定义的 happens-before 关系[25]。需要注意的是，Causal 网络对可靠性没有要求。也就是说，它允许消息的重传与丢失。

Network 模块

在 *Network* 模块中，常量 *Replica* 表示副本节点；数组类型变量 *incoming* 表示副本节点维护的待处理消息集合，其数组元素类型是多集 *Bag*；数组类型变量 *msg* 表示副本节点当前正在处理的消息。

NBroadcast 描述了副本节点广播消息的行为。当某个节点 r 将一条消息 m 广播给其他节点时，节点 r 的待处理消息队列保持不变，同时使用 *SetToBag* 将消息集合转化为消息队列，添加到其他各个节点的待处理消息队列中。该行为允许发送重复的消息。

NDeliver 描述了副本节点接收消息的行为。节点 r 调用接收消息的事件 *NDeliver* 时，待处理消息队列 $incoming[r]$ 不为空，同时使用 *BagToSet* 将消息队列转化为消息集合，任选其中的一条消息 *msg* 进行接收。该行为丢弃了发生重传的消息，忽略了消息的发送顺序。

MODULE *Network*

EXTENDS *Bags*, *Message*

VARIABLES

incoming, *incoming*[*r*]: incoming messages at replica $r \in Replica$

msg, $msg[r]$: current message at replica $r \in Replica$

$NBroadcast(r,\ m) \triangleq$
$\quad \land\ incoming' = [x \in Replica \mapsto$
$\qquad$ IF $x = r$
$\qquad$ THEN $incoming[x]$
$\qquad$ ELSE $incoming[x] \oplus SetToBag(\{m\})]$
$NDeliver(r) \triangleq$
$\quad \land\ incoming[r] \neq EmptyBag$
$\quad \land \exists\, m \in BagToSet(incoming[r]) : msg' = [msg$ EXCEPT $![r] = m]$

ReliableNetwork 模块

可靠性网络 *ReliableNetwork* 模块是对 *Network* 模块的扩展，该网络保证每条消息都会被每个节点接收且仅接收一次。节点调用 *RDeliver* 事件接收消息时，使用 *IsDeliverMsg* 对消息进行判断，仅接收未被接收过的消息，并将消息序列号标记为已被接收，保证同一消息不会被接收多次。

MODULE *ReliableNetwork*

EXTENDS *Network*

$RDeliver(r) \triangleq$
$\quad \land\ incoming[r] \neq EmptyBag$
$\quad \land \exists\, m \in BagToSet(incoming[r]) : \neg IsDeliverMsg(m,\ r)$
$\qquad \land\ msg' = [msg$ EXCEPT $![r] = m]$

CausalNetwork 模块

保证因果关系的（Causal）网络 *CausalNetwork* 模块是对 *Network* 模块的另一个扩展，该网络保证每条消息仅能在它所依赖的消息都被接收之后才能被接收。我们使用二维数组变量 vc 对系统的向量时钟进行维护，描述消息之间的 happens-before 依赖关系。图 3-2 是一个在分布式系统中使用向量时钟进行通信的实例。

具体地，每条消息 m 维护一个向量 $m.ts$，存储所有副本节点的时间戳。二维数组变量 vc 的所有元素初始化为零。当某个节点 r 发送消息或者接收消息时，首先将该节点维护的自身的逻辑时钟 $vc[r][r]$ 加一。另外，当某个节点接收消息时，还需要判断该节点维护的向量中的每一个时钟是否需要更新，若收到的消息中的对应时间戳较后，则将其更新为对应值，否则不更新。

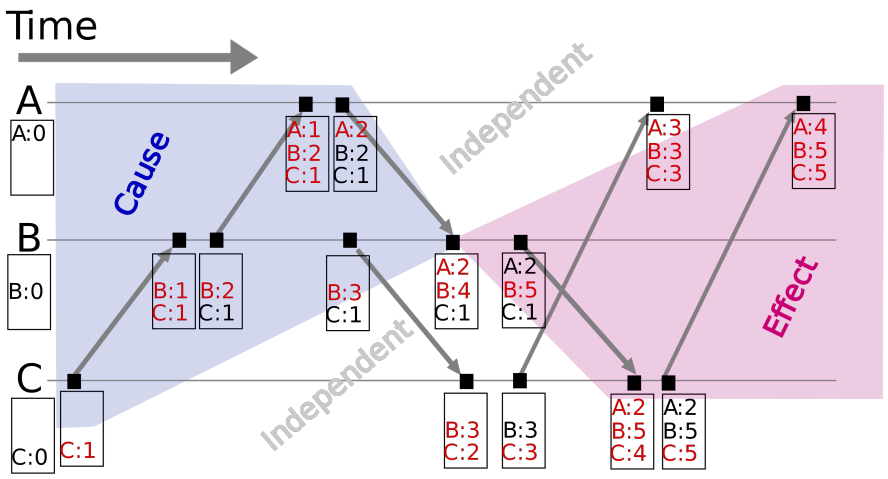

图 3-2: 向量时钟系统的实例。蓝色区域中的事件是事件 B4 的原因。红色区域中的事件是事件 B5 的结果。

MODULE *CausalNetwork*

EXTENDS *Network*, *Naturals*

VARIABLES

vc — *vc*[*r*][*s*]: vector clock of replica $s \in Replica$ at replica $r \in Replica$

$$
\begin{aligned}
&CBroadcast(r,\ m) \triangleq \\
&\quad \land\ NBroadcast(r,\ m) \\
&\quad \land\ vc' = [vc \text{ EXCEPT } ![r][r] = @ + 1] \\
&\\
&CDeliver(r) \triangleq \\
&\quad \land\ incoming[r] \neq EmptyBag \\
&\quad \land\ vc' = [vc \text{ EXCEPT } ![r][r] = @ + 1] \\
&\quad \land\ \exists\, m \in BagToSet(incoming[r]) : \\
&\qquad \land\ \forall\, s \in Replica : \\
&\qquad\quad \lor\ m.ts[s] \leq vc[r][s] \\
&\qquad\quad \lor\ vc' = [vc \text{ EXCEPT } ![r][s] = SetMax(@,\ m.ts[s])] \\
&\qquad \land\ msg' = [msg \text{ EXCEPT } ![r] = m]
\end{aligned}
$$

我们可以看到，FIFO Reliable 网络和 Causal 网络的差别主要体现在可靠性要求和顺序性要求的不同。FIFO Reliable 网络在接收消息时，会对消息进行判断和标记，保证每条消息都会被每个节点接收且仅接收一次；而 Causal 网络对可靠性没有要求，它允许消息的重传与丢失。FIFO Reliable 网络保证每个节点发送的消息会被其它节点按序接收；而 Causal 网络则保证每条消息仅能在它所依赖的（happens-before）消息都被接收之后才能被接收。

3.3 协议接口层

如第 2.2 节所述，CRDT 协议可以分为两类：基于操作的协议与基于状态的协议。在消息传递方面，前者仅将最近一次更新操作发送给其它副本节点，而后者则发送最新的节点状态。协议接口层为这两类协议提供了统一的接口，包括 *Do*、*Send* 与 *Receive*[26]。其中，*Send* 与 *Receive* 建模消息发送与消息接收事件，它们使用底层的网络通信层进行通信。*Do* 则是对具体数据类型所提供的操作的总称。例如，对于计数器（Counter）而言，*Do* 是对更新操作 *Inc* 以及查询操作 *Read* 的总称；对于集合（Set）而言，*Do* 则是对更新操作 *Add* 与 *Remove* 以及查询操作 *Contains* 的总称。

更新操作

Inc 描述了计数器（Counter）类型协议中，副本节点发出加一请求的操作；*Add* 和 *Remove* 分别描述了集合（Set）类型协议中，向集合中添加和删除元素的操作。当节点发出更新操作时，该副本节点维护的消息序列标识号 *seq* 加一，调用 *SEC* 模块的 *SECUpdate* 事件，对更新集的状态进行更新。

查询操作

为了验证 CRDT 协议的正确性，使用查询操作对各个副本节点维护的数据进行读取。例如，在计数器（Counter）类型协议中，使用查询操作 *Read* 获取副本节点维护的计数器状态；在集合（Set）类型协议中，使用查询操作 *Contain* 读取当前副本节点维护的集合中的所有实例所包含的数据。

消息通信

在消息的定义 *Msg* 中，*Instance* 与具体的 CRDT 类型相关，不同的 CRDT 对 *Instance* 有不同的定义。

与消息通信相关的事件可以分为消息发送和消息接收两类。消息发送事件 *Send* 的实现调用了网络模块的广播事件，将消息广播至其他所有副本节点。

Receive 事件对消息接收进行了描述。当某个节点接收到来自其他节点的消息时，调用网络模块的接收事件，对接收的消息进行读取解析，根据具体的数据类型对消息进行处理。

MODULE *SomeCRDT*

$Msg \triangleq [r : Replica, ins : \text{SUBSET } Instance, seq : Nat, update : \text{SUBSET } Update]$

$Inc(r)/Add(d,\, r)/Remove(d,\, r)/\ldots \triangleq$
$\quad \land\, seq' = [seq \text{ EXCEPT } ![r] = @ + 1]$
$\quad \land\, SECUpdate(r,\, seq[r])$

$Read(r) \triangleq counter[r]$
$Contain(r) \triangleq \{ins.d : ins \in xSet[r]\}$

$Send(r) \triangleq$
$\quad \land\, SomeNetwork!SomeBroadcast(r,\, [r \mapsto r,\, Ins \mapsto someIns[r],\, seq \mapsto seq[r],$
$\qquad update \mapsto OpUpdate(r)/StateUpdate(r)])$
$Receive(r) \triangleq$
$\quad \land\ SomeNetwork!SomeDeliver(r)$
$\quad \land\ ...$

3.4 规约层

我们的目标是验证 CRDT 协议满足强最终一致性，它要求执行了相同更新操作集合的节点具有相同的状态。为此，我们使用两个数组类型变量 *updateSet* 和 *prev_updateSet* 对副本节点的更新集状态进行描述。其中，*updateSet* 表示副本节点当前状态的更新集，*prev_updateSet* 表示上一个状态的更新集。

我们定义这两个数组元素的类型为 *Update*，将其作为操作更新的表示，每一个操作更新包括两个域：发出操作请求的副本节点号和操作请求序列号。对于基于操作的无冲突复制数据类型，操作更新的定义 *OpUpdate* 是当前状态的更新集和上一个状态的更新集之间的差集；对于基于状态的无冲突复制数据类型，操作更新的定义 *StateUpdate* 是当前状态的更新集。

当副本节点调用事件，带来操作更新时，规约层调用 *SECUpdate* 事件，将当前操作更新加入当前状态的更新集; 当副本节点发送消息时，规约层调用 *SECSend* 事件，将上一个状态的更新集替换为当前状态的更新集; 当副本节点接收消息时，规约层调用 *SECReceive* 事件，将接收到的消息中的操作更新加入当前状态的更新集和上一个状态的更新集。

最后，我们定义 *SameUpdate* 对两个更新集是否相同进行判断。

MODULE *SEC*

VARIABLES
updateSet, *updateSet*[*r*]: set of *Update* in the current state at replica $r \in Replica$
prev_updateSet *prev_updateSet*[*r*]: set of *Update* in the previous state at replica $r \in Replica$

$$SECvars \triangleq \langle updateSet,\ prev\_updateSet \rangle$$

$$Update \triangleq [r : Replica,\ seq : Nat]$$
$$OpUpdate(r) \triangleq updateSet[r] \setminus prev\_updateSet[r] \quad \text{for op-based } CRDT$$
$$StateUpdate(r) \triangleq updateSet[r] \quad \text{for state-based } CRDT$$

$$SECUpdate(r,\ seq) \triangleq$$
$$\land\ updateSet' = [updateSet \ \text{EXCEPT}\ ![r] = @ \cup \{[r \mapsto r,\ seq \mapsto seq]\}]$$

$$SECSend(r) \triangleq$$
$$\land\ prev\_updateSet' = [prev\_updateSet \ \text{EXCEPT}\ ![r] = updateSet[r]]$$

$$SECDeliver(r,\ m) \triangleq$$
$$\land\ updateSet' = [updateSet \ \text{EXCEPT}\ ![r] = @ \cup m.update]$$
$$\land\ prev\_updateSet' = [prev\_updateSet \ \text{EXCEPT}\ ![r] = @ \cup m.update]$$

$$SameUpdate(r1,\ r2) \triangleq updateSet[r1] = updateSet[r2]$$

第四章　使用 TLA$^+$ 描述 CRDT 协议

我们以基于操作和基于状态的 Counter 和 Set 为例，使用 TLA$^+$ 语言对 CRDT 协议进行具体描述。

4.1　使用 TLA$^+$ 描述 Counter 协议

我们使用 TLA$^+$ 描述 Counter 协议。这里提及的 Counter 特指 Grow-only Counter，而不考虑 Positive-Negative Counter 的情形。

4.1.1　使用 TLA$^+$ 描述 Op-based Counter 协议

我们使用 TLA$^+$ 描述 Operation-based Counter 协议[26]。

我们声明 *counter* 和 *incVal* 两个数组类型变量对 Operation-based Counter 的本地节点进行维护。*counter* 表示每个副本节点的计数器的当前值；*incVal* 表示自上次广播之后，每个副本节点的计数器的增加值。在声明变量的时候我们没有对其变量的类型进行限制，因而设计 *TypeOK* 规约对变量的类型进行检查，*counter* 和 *incVal* 两个变量均为一组由 *Replica* 到自然数的映射。

我们定义 *Spec* 规约描述状态之间的转移。在描述初始状态的规约 *Init* 中，对所有的变量进行初始化赋值，将 *counter* 和 *incVal* 置零。在描述次状态的规约 *Next* 中，对 Operation-based Counter 的相关事件进行了抽象。对于行为的下一个状态，调用的事件可能是 $Inc(r)$，$Send(r)$ 或 $Receive(r)$ 事件中的一种。

$Inc(r)$ 事件

$Inc(r)$ 事件描述了副本节点发出计数器加一请求时的状态。当加一请求被提出时，该副本节点维护的各个变量的变化如下：计数器 *counter* 的值加一，累积增加数 *incVal* 的值加一。

Send(*r*) 事件

Send(*r*) 事件描述了某个节点向其他节点发送消息时的状态。此时该节点的计数器累积值不为空，即存在待发送的请求，调用 *Send*(*r*) 事件之后，将其重置为零。

Receive(*r*) 事件

Receive(*r*) 事件描述了某个节点接收消息时的状态。当某个节点接收来自其他节点的消息时，对接收的消息进行解析，更新全局变量 *counter* 的值。

MODULE *OpCounter*

VARIABLES
counter, *counter*[*r*]: current value of the counter at replica $r \in Replica$
incVal, *incVal*[*r*]: number of increments performed since the last broadcast at replica $r \in Replica$

$$
\begin{aligned}
&TypeOK \triangleq \\
&\quad \land counter \in [Replica \to Nat] \\
&\quad \land incVal \in [Replica \to Nat]
\end{aligned}
$$

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land counter = [r \in Replica \mapsto 0] \\
&\quad \land incVal = [r \in Replica \mapsto 0] \\
&Inc(r) \triangleq \\
&\quad \land counter' = [counter \text{ EXCEPT } ![r] = @ + 1] \\
&\quad \land incVal' = [incVal \text{ EXCEPT } ![r] = @ + 1] \\
&Send(r) \triangleq \\
&\quad \land incVal[r] \neq 0 \\
&\quad \land incVal' = [incVal \text{ EXCEPT } ![r] = 0] \\
&Receive(r) \triangleq \\
&\quad \land counter' = [counter \text{ EXCEPT } ![r] = @ + msg'[r].inc] \\
&Next \triangleq \exists\, r \in Replica : Inc(r) \lor Send(r) \lor Receive(r) \\
&Spec \triangleq Init \land \Box[Next]_{vars}
\end{aligned}
$$

4.1.2 使用 TLA+ 描述 State-based Counter 协议

我们使用 TLA+ 描述 State-based Counter 协议[26]。

在 State-based Counter 中，每个节点均需要维护所有节点的数据信息。我们定义向量数据类型 *Vector*，作为对每个节点维护的数据类型的封装。我们声明 *counter* 和 *sendAllowed* 两个数组类型变量对 State-based Counter 的本地节点进行维护。其中，*counter* 是一组由 *Replica* 到 *Vector* 的映射；*sendAllowed*

是一组由 *Replica* 到 $\{0,1\}$ 的映射，作为每个节点是否处于可以发送消息的状态的标识。*TypeOK* 规约的设计和 Operation-based Counter 类似。

我们定义 *Spec* 规约描述状态之间的转移。在描述初始状态的规约 *Init* 中，对所有的变量进行初始化赋值，将 *counter* 数组中的所有元素置零，*sendAllowed* 变量置零。在描述次状态的规约 *Next* 中，对 State-based Counter 的相关事件进行了抽象。对于行为的下一个状态，调用的事件可能是 $Inc(r)$，$Send(r)$ 或 $Receive(r)$ 事件中的一种。

$Inc(r)$ 事件

$Inc(r)$ 事件描述了副本节点发出计数器加一请求的状态，该副本节点的对应的各个变量变化如下：当前节点的计数器 *counter* 加一，*sendAllowed* 置为一，表示允许该节点向其他节点发送消息。

$Send$ 事件

Send 事件描述了某个节点向其他节点发送消息时的状态。某个节点向其他节点发送消息后，不再允许该节点发送消息，即将 *sendAllowed* 变量置零。

$Receive(r)$ 事件

$Receive(r)$ 事件描述了某个节点接收消息时的状态。当某个节点接收到其他节点发送的消息时，会将其维护的向量中的所有节点的值同接收到的消息中对应节点的值进行比较，取较大者作为更新后的值，更新 *counter* 变量。

MODULE *StateCounter*

VARIABLES
counter, *counter*[*r*][*s*]: current value of the *Replica*[*s*] at *replica*[*r*]
sendAllowed, *sendAllowed*[*r*]: is the replica $r \in Replica$ allowed to send a message

$Vector \triangleq [Replica \to Nat]$
$InitVector \triangleq [r \in Replica \mapsto 0]$

$TypeOK \triangleq$
$\quad \land \quad counter \in [Replica \to Vector]$
$\quad \land \quad sendAllowed \in [Replica \to \{0, 1\}]$

$Init \triangleq$
$\quad \land\, counter = [r \in Replica \mapsto InitVector]$
$\quad \land\, sendAllowed = [r \in Replica \mapsto 0]$
$Inc(r) \triangleq$
$\quad \land\, counter' = [counter \text{ EXCEPT } ![r][r] = @ + 1]$
$\quad \land\, sendAllowed' = [sendAllowed \text{ EXCEPT } ![r] = 1]$

$Send(r) \triangleq$
$\quad \land sendAllowed' = [sendAllowed \text{ EXCEPT } ![r] = 0]$
$Receive(r) \triangleq$
$\quad \land \ \forall s \in Replica :$
$\qquad counter' = [counter \text{ EXCEPT } ![r][s] = SetMax(@, msg'[r].buf[s])]$
$Next \triangleq \ \land \exists r \in Replica : Inc(r) \lor Send(r) \lor Receive(r)$
$Spec \triangleq Init \land \Box[Next]_{vars}$

4.2 使用 TLA+ 描述 Add-Wins Set 协议

我们使用 TLA+ 描述 Add-Wins Set 协议。Add-Wins Set 是添加操作优先级高于移除操作的集合。这意味着如果并发地添加和移除一个元素，添加元素请求不会被处理，该元素会被添加到集合中。

4.2.1 使用 TLA+ 描述 Op-based Add-Wins Set 协议

我们使用 TLA+ 描述 Operation-based Add-Wins Set 协议[4]。

首先对常量 *Data* 进行声明，*Data* 表示数据的集合。接下来对 record 类型消息实例 *Instance* 进行定义，指定每个域的名称及其具体类型：数据域 d 属于 *Data*，副本节点号域 r 属于 *Replica*，消息序列号域 k 属于自然数。其中，消息序列号就是消息实例对应的标识符。

然后对变量进行声明。协议中每个副本节点维护一个 S 集合 *sSet*，其包含的元素是该副本节点的实例。*seq* 是消息序列标识数组，用以对消息进行唯一性标识。当一个元素被从集合中移除后，如果它再次被添加到集合中，那么需要对消息序列号进行更新，以标记操作的唯一性。*incoming*，*msg* 和 *messageSet* 是由每个副本节点各自维护的网络通信层变量，*SECvars* 是从 *SEC* 模块引入的变量组。使用 *TypeOK* 规约对变量进行类型检查。*sSet* 数组中的每个元素均为副本节点 *Replica* 到 *Instance* 实例的集合的映射。

我们定义 *Spec* 规约描述状态之间的转移。在描述初始状态的规约 *Init* 中，将 *sSet* 数组中的每个元素初始化为空集。对于行为的下一个状态，次状态的规约 *Next* 调用的事件可能是添加元素 $Add(d, r)$，移除元素 $Remove(d, r)$，发送消息 $Send(r)$ 或接收消息 $Receive(r)$ 四种事件中的一种。

Add(d, r) 事件

调用 *Add*(d, r) 事件时，会把当前副本节点 r 维护的消息序号 *seq* 加一，向该节点维护的 S 集合中添加一个 *Instance* 实例元素，它的三个域分别为 d，r 和修改后的消息序号 $seq'[r]$。

Remove(d, r) 事件

调用 *Remove*(d, r) 事件时，我们需要在节点维护的集合 S 中删除包含数据 d 的消息实例。具体地，我们定义一个辅助变量 D 集合进行操作，D 集合是 S 集合中所有数据为 d 的实例组成的集合，将 S 集合与 D 集合作差，结果作为更新之后的 S 集合。

Send(r) 事件和 *Receive*(r) 事件

对于消息通信，当某个节点调用 *Send*(r) 事件发送消息时，无需对 S 集合进行处理；而当其接收到来自其他节点的消息时，会将消息中的 S 集合加入该副本节点维护的 S 集合中。

MODULE *ORSet*

CONSTANTS
$Data$ the set of data

$Instance \triangleq [d : Data,\ r : Replica,\ k : Nat]$

VARIABLES
$sSet$ $sSet[r]$: set of active *Instance*(s) maintained by $r \in Replica$

$vars \triangleq \langle sSet,\ seq,\ incoming,\ msg,\ messageSet,\ SECvars \rangle$

$TypeOK \triangleq sSet \in [Replica \to \text{SUBSET } Instance]$

$Init \triangleq sSet = [r \in Replica \mapsto \{\}]$
$Add(d,\ r) \triangleq$
$\quad \land\ seq' = [seq \text{ EXCEPT } ![r] = @ + 1]$
$\quad \land\ sSet' = [sSet \text{ EXCEPT } ![r] = @ \cup \{[d \mapsto d,\ r \mapsto r,\ k \mapsto seq'[r]]\}]$
$Remove(d,\ r) \triangleq$
$\quad \land \text{LET } D \triangleq \{ins \in sSet[r] : ins.d = d\}$
$\quad\ \text{IN } sSet' = [sSet \text{ EXCEPT } ![r] = @ \setminus D]$
$\quad \land\ seq' = [seq \text{ EXCEPT } ![r] = @ + 1]$
$Send(r) \triangleq$
$\quad \land\ Network!RBroadcast(r,\ [r \mapsto r,\ S \mapsto sSet[r],\ seq \mapsto seq[r],$
$\qquad update \mapsto OpUpdate(r)])$
$\quad \land\ SECSend(r)$
$Receive(r) \triangleq$
$\quad \land\ Network!RDeliver(r)$

$\quad \land \ SECDeliver(r,\ msg'[r])$
$\quad \land \ sSet' = [sSet \ \text{EXCEPT} \ ![r] = @ \cup msg'[r].S]$
$Next \triangleq$
$\quad \lor \exists\, r \in Replica : \exists\, a \in Data : Add(a,\ r) \lor Remove(a,\ r)$
$\quad \lor \exists\, r \in Replica : Send(r) \lor Receive(r)$
$Spec \triangleq Init \land \Box[Next]_{vars}$

4.2.2 使用 TLA+ 描述 State-based Add-Wins Set 协议

我们使用 TLA+ 描述 State-based Add-Wins Set 协议[8]。

协议中每个副本节点维护一个 A 集合变量 *aSet* 和一个 T 集合变量 *tSet*，A 集合（Active Set）维护该副本节点当前活跃的实例集合，T 集合（Tombstone [①] Set）维护该副本节点已经失活的实例集合，对读操作不可见，仅用来记录元素移除的相关信息。A 集合和 T 集合不相交。

常量 *Data* 的声明，网络通信和性质检验相关变量的声明，record 类型消息实例 *Instance* 的定义，类型检查规约 *TypeOK* 的定义，状态转移规约 *Spec* 的定义，均与第 4.2.1 节中的声明和定义类似。

在初始状态规约 *Init* 中，将 *aSet* 和 *tSet* 数组中的每个元素初始化为空集。在描述次状态的规约 *Next* 中，对行为的下一个状态可能调用的四种事件进行描述。

$Add(d, r)$ 事件

调用 $Add(d, r)$ 事件时，会把当前副本节点 r 维护的消息序号 *seq* 加一，向该节点维护的 A 集合中添加一个 *Instance* 实例元素，它的三个域分别为 d，r 和修改后的消息序号 $seq'[r]$。

$Remove(d, r)$ 事件

调用 $Remove(d, r)$ 事件时，我们需要在节点维护的活跃集合 A 中删除包含数据 d 的消息实例，并将其加入非活跃集合 T 中。具体地，我们定义一个辅助变量 D 集合进行操作，D 集合是 A 集合中所有数据为 d 的消息实例元素组成的集合，将 D 集合的元素从活跃集合 A 中作差，将 D 集合中的元素加入非活跃集合 T 集合中。

[①] https://en.wikipedia.org/wiki/Tombstone_(data_store)

Send(*r*) 事件和 *Receive*(*r*) 事件

对于消息通信，当某个节点调用 *Send*(*r*) 事件发送消息时，无需对 A 集合和 T 集合进行处理；而当其接收到来自其他节点的消息时，会将消息中的 S 集合加入该副本节点维护的 S 集合中。会将消息中的 A 集合和 T 集合中的消息实例元素分别加入该副本节点维护的 A 集合和 T 集合中，并从该节点维护的 A 集合中删除其 T 集合包含的元素。

```
MODULE AWSet
CONSTANTS
    Data        the set of data

VARIABLES
    aSet,       aSet[r]: set of active Instance(s) maintained by r ∈ Replica
    tSet        tSet[r]: set of tombstone Instance(s) maintained by r ∈ Replica

vars ≜ ⟨aSet, tSet, seq, incoming, msg, messageSet, SECvars⟩
----------------------------------------------------------------
Instance ≜ [d : Data, r : Replica, k : Nat]
----------------------------------------------------------------
TypeOK ≜
    ∧  aSet ∈ [Replica → SUBSET Instance]
    ∧  tSet ∈ [Replica → SUBSET Instance]
----------------------------------------------------------------
Init ≜
    ∧ aSet = [r ∈ Replica ↦ {}]
    ∧ tSet = [r ∈ Replica ↦ {}]
Add(d, r) ≜
     ∧ seq′  = [seq EXCEPT ![r] = @ + 1]
     ∧ aSet′ = [aSet EXCEPT ![r] = @ ∪ {[d ↦ d, r ↦ r, k ↦ seq′[r]]}]
Remove(d, r) ≜
   LET D ≜ {ins ∈ aSet[r] : ins.d = d}
   IN   ∧ aSet′ = [aSet EXCEPT ![r] = @ \ D]
        ∧ tSet′ = [tSet EXCEPT ![r] = @ ∪ D]
Send(r) ≜
     ∧ SomeNetwork!SomeBroadcast(r, [r ↦ r, Ins ↦ someIns[r], seq ↦ seq[r],
       update ↦ OpUpdate(r)/StateUpdate(r)])
Receive(r) ≜
    ∧  tSet′ = [tSet EXCEPT ![r] = @ ∪ msg′[r].T]
    ∧  aSet′ = [aSet EXCEPT ![r] = (@ ∪ msg′[r].A) \ tSet′[r]]
    ∧  seq′ = [seq EXCEPT ![r] = @ + 1]
Next ≜
    ∨ ∃ r ∈ Replica : ∃ a ∈ Data :
       Add(a, r) ∨ Remove(a, r)
    ∨ ∃ r ∈ Replica :
```

$$
\begin{aligned}
&\quad Send(r) \lor Receive(r) \\
Spec &\triangleq Init \land \Box[Next]_{vars}
\end{aligned}
$$

第五章　使用 TLC 验证 CRDT 协议的正确性

我们使用模型检验工具 TLC [27] 对 CRDT 协议的正确性进行检验。

5.1　CRDT 的一致性规约

在第 1.1 节中我们提到，分布式系统中提供的数据一致性模型主要分为以下三类：强一致性，最终一致性和强最终一致性。

定义 5-1　强一致性　接收了相同更新的节点到达相同的状态。

定义 5-2　最终一致性　如果客户端停止提交更新，副本节点最终到达相同的状态。

定义 5-3　强最终一致性　执行了相同的更新操作的节点具有相同的状态

我们主要关注 CRDT 的强最终一致性，它要求各个节点以相同的方式解决操作更新带来的冲突，减少资源的浪费，提高系统性能，同时对系统效率有一定的保证。强最终一致性要求对于任意两个副本节点，使用查询操作读取它们维护的数据集合，使用 TLA$^+$ 的描述如下（以查询操作 *Read* 为例），

$$SEC \triangleq \forall r1, r2 \in Replica : SameUpdate(r1, r2) \Rightarrow Read(r1) = Read(r2)$$

5.2　TLC 模型检验实验

TLC 能对可执行的 TLA$^+$ 规约进行检查和模拟，检验系统的安全性和活跃性等性质。对于 TLA$^+$ 描述的模型，TLC 生成满足规约的初始状态，然后对所有初始状态进行广度优先搜索，对状态空间进行遍历。另外，TLC 对多线程和分布式模式的运行机制的支持，使得模型检验的运行效率能随着 CPU 核心数提升而得到近似线性的提升。

5.2.1 实验设计

在模型检查实验中，我们变换副本节点和数据的数目，使用多个工作线程对不同的 CRDT 协议的强最终一致性进行验证。同时，由于我们没有对消息序列号 *seq* 的上限作出限制，模型检验不会终止，因而我们对状态数进行限制，将不同状态数的上限设置为 100000000。

```
TLCSet("exit", TLCGet("distinct") > 100000000)
```

我们使用的硬件配置为 6 核 12 线程 2.40GHz 处理器，64GB 内存。TLC 的工作线程数设置为 12。

5.2.2 实验结果

我们对 TLC 检查的状态数，不同的状态数和检验时间（以 hh : mm : ss 为单位）进行统计报告。

对于基于操作和基于状态的计数器类型，统计结果如表 5-1 和表 5-2。我们设置的副本节点 *Replica* 的数目为 2，3，4，5，6。

表 5-1: Op-based Counter 满足强最终一致性的模型检验结果

TLC Model *#Replicas*	Diameter	# States	# Distinct States	Checking Time (*hh* : *mm* : *ss*)
2	27	323734223	100000014	0 : 19 : 09
3	21	397112964	100000010	0 : 22 : 39
4	19	427889983	100000036	0 : 24 : 32
5	17	439576601	100000030	0 : 25 : 20
6	16	443697850	100000031	0 : 26 : 30

对于基于操作和基于状态的集合类型，统计结果如表 5-3 和表 5-4。我们设置的副本节点 *Replica* 和 *Data* 的数目为 (2, 1)，(2, 2)，(2, 3)，(3, 1)，(3, 2)，(3, 3)。

表 5-2: State-based Counter 满足强最终一致性的模型检验结果

TLC Model ($\#Replicas$)	Diameter	# States	# Distinct States	Checking Time ($hh:mm:ss$)
2	22	238659029	100000024	0 : 16 : 18
3	18	408288667	100000020	0 : 26 : 42
4	15	543915501	100000025	0 : 35 : 12
5	13	572815237	100000053	0 : 40 : 21
6	11	546284115	100000033	0 : 43 : 18

表 5-3: Op-based AWSet 满足强最终一致性的模型检验结果

TLC Model ($\#Replicas, \#Data$)	Diameter	# States	# Distinct States	Checking Time ($hh:mm:ss$)
$(2,1)$	17	227990476	100000023	0 : 15 : 19
$(2,2)$	15	228232406	100000031	0 : 16 : 48
$(2,3)$	14	239089058	100000048	0 : 28 : 10
$(3,1)$	15	297094435	100000045	0 : 33 : 31
$(3,2)$	13	305691249	100000054	0 : 36 : 25
$(3,3)$	13	326937514	100000052	0 : 51 : 59

表 5-4: State-based AWSet 满足强最终一致性的模型检验结果

TLC Model ($\#Replicas, \#Data$)	Diameter	# States	# Distinct States	Checking Time ($hh:mm:ss$)
$(2,1)$	15	170480475	100000051	0 : 12 : 48
$(2,2)$	14	182729246	100000050	0 : 13 : 48
$(2,3)$	13	199395208	100000051	0 : 20 : 38
$(3,1)$	15	224573987	100000049	0 : 23 : 30
$(3,2)$	14	241277578	100000054	0 : 30 : 31
$(3,3)$	12	266384279	100000063	0 : 40 : 29

第六章　结论

在分布式系统中，强最终一致性是 CRDT 的重要一致性规约，本文使用形式化规约语言 TLA$^+$ 语言和模型检验工具 TLC 对 CRDT 的相关协议及其满足性质分别进行了描述和验证。

6.1　工作总结

本文首先对系统模型进行了描述，然后介绍了三种强弱不同的数据一致性模型，而后我们使用形式化规约语言 TLA$^+$ 描述 CRDT 协议，并使用模型检验工具 TLC 验证它们的正确性。

本文的主要贡献包括：首先，对 CRDT 协议的验证框架进行抽象，将其划分为网络通信层、协议接口层、具体协议层以及规约层等四个层次。其中，网络通信层对副本节点之间的 P2P 网络连接进行了建模，并提供了 CRDT 协议所依赖的各种类型的通信信道；协议接口层为对通信信道要求相似的 CRDT 协议族提供了统一的接口；具体协议层是对协议接口层的实现，不同的 CRDT 协议可能要求选择不同的通信模块；规约层描述了所有 CRDT 协议都满足强最终一致性。

其次，我们以 Counter 和 Add-wins Set 为例，使用 TLA$^+$ 语言实现了这两种 CRDT 基于操作和基于状态的描述。

最后，我们使用 TLC 模型检验工具，对 CRDT 协议的正确性，即强最终一致性进行了验证，根据模型检验实验，我们有很大的把握认为，我们验证的 CRDT 协议满足强最终一致性规约。

6.2　研究展望

在我们的实验中，验证的节点数目和数据规模有限，我们不能对 CRDT 协议相关的正确性作出过于绝对的判断，因而为增强模型检验的可靠性，我们还需要设置更大规模的实验参数进行实验。

此外，我们目前完成的工作局限于几种 CRDT 协议，其他 CRDT 协议的正确性验证亟待完成，相关工作主要集中在具体协议层。

参考文献

[1] BREWER E A. Towards Robust Distributed Systems (Abstract)[C/OL] // PODC '00 : Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. New York, NY, USA : ACM, 2000 : 7 – . http://doi.acm.org/10.1145/343477.343502.

[2] GILBERT S, LYNCH N. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services[J/OL]. SIGACT News, 2002, 33(2) : 51 – 59.
http://doi.acm.org/10.1145/564585.564601.

[3] SHAPIRO M, PREGUIÇA N, BAQUERO C, et al. Conflict-free Replicated Data Types[C/OL] // SSS'11 : Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems. Berlin, Heidelberg : Springer-Verlag, 2011 : 386 – 400.
http://dl.acm.org/citation.cfm?id=2050613.2050642.

[4] SHAPIRO M, PREGUIÇA N, BAQUERO C, et al. A comprehensive study of Convergent and Commutative Replicated Data Types : RR-7506[R/OL]. [S.l.] : Inria – Centre Paris-Rocquencourt ; INRIA, 2011 : 50.
https://hal.inria.fr/inria-00555588.

[5] ROH H-G, JEON M, KIM J-S, et al. Replicated abstract data types: Building blocks for collaborative applications[J]. Journal of Parallel and Distributed Computing, 2011, 71(3) : 354 – 368.

[6] ATTIYA H, BURCKHARDT S, GOTSMAN A, et al. Specification and Complexity of Collaborative Text Editing[C/OL] // PODC '16 : Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing. New York, NY, USA : ACM, 2016 : 259 – 268.
http://doi.acm.org/10.1145/2933057.2933090.

[7] Jr CLARKE E M, GRUMBERG O, PELED D A. Model Checking[M]. Cambridge, MA, USA : MIT Press, 1999.

[8] ZAWIRSKI M. Dependable Eventual Consistency with Replicated Data Types[D/OL]. [S.l.] : Universite Pierre et Marie Curie, 2015.
https://tel.archives-ouvertes.fr/tel-01248051.

[9] STEINKE R C, NUTT G J. A Unified Theory of Shared Memory Consistency[J/OL]. J. ACM, 2004, 51(5) : 800 – 849.
http://doi.acm.org/10.1145/1017460.1017464.

[10] TERRY D B, THEIMER M M, PETERSEN K, et al. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System[C/OL] // SOSP '95 : Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles. New York, NY, USA : ACM, 1995 : 172 – 182.
http://doi.acm.org/10.1145/224056.224070.

[11] VOGELS W. Eventually Consistent[J/OL]. Commun. ACM, 2009, 52(1) : 40 – 44.
http://doi.acm.org/10.1145/1435417.1435432.

[12] GOMES V B F, KLEPPMANN M, MULLIGAN D P, et al. Verifying Strong Eventual Consistency in Distributed Systems[J/OL]. Proc. ACM Program. Lang., 2017, 1(OOPSLA) : 109:1 – 109:28.
http://doi.acm.org/10.1145/3133933.

[13] CLARKE E M. 25 Years of Model Checking[G/OL] // GRUMBERG O, VEITH H. . Berlin, Heidelberg : Springer-Verlag, 2008 : 1 – 26.
http://dx.doi.org/10.1007/978-3-540-69850-0_1.

[14] YUAN D, LUO Y, ZHUANG X, et al. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems[C/OL] // OSDI'14 : Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation. Berkeley, CA, USA : USENIX Association, 2014 : 249 – 265.
http://dl.acm.org/citation.cfm?id=2685048.2685068.

[15] LAMPORT L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers[M]. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002.

[16] ZELLER P, BIENIUSA A, POETZSCH-HEFFTER A. Formal Specification and Verification of CRDTs[C] // ÁBRAHÁM E, PALAMIDESSI C. Formal Techniques for Distributed Objects, Components, and Systems. Berlin, Heidelberg : Springer Berlin Heidelberg, 2014 : 33 – 48.

[17] SUN C, XU Y, AGUSTINA A. Exhaustive Search of Puzzles in Operational Transformation[C/OL] // CSCW '14 : Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing. New York, NY, USA : ACM, 2014 : 519 – 529.
http://doi.acm.org/10.1145/2531602.2531630.

[18] SINCHUK S, CHUPRIKOV P, SOLOMATOV K. Verified Operational Transformation for Trees[C] // BLANCHETTE J C, MERZ S. Interactive Theorem Proving. Cham : Springer International Publishing, 2016 : 358 – 373.

[19] LIU Y, XU Y, ZHANG S J, et al. Formal Verification of Operational Transformation[C] // JONES C, PIHLAJASAARI P, SUN J. FM 2014: Formal Methods. Cham : Springer International Publishing, 2014 : 432 – 448.

[20] Randolph A, Boucheneb H, Imine A, et al. On Synthesizing a Consistent Operational Transformation Approach[J/OL]. IEEE Transactions on Computers, 2015, 64(4) : 1074 – 1089.
http://dx.doi.org/10.1109/TC.2014.2308203.

[21] "IMINE A, MOLLI P, OSTER G, et al. [C] // . Dordrecht : Springer Netherlands, 2003 : 277 – 293.

[22] IMINE A, RUSINOWITCH M, OSTER G, et al. Formal design and verification of operational transformation algorithms for copies convergence[J/OL]. Theoretical Computer Science, 2006, 351(2) : 167 – 183.
http://www.sciencedirect.com/science/article/pii/S030439750500616X.

[23] OSTER G, URSO P, MOLLI P, et al. Proving correctness of transformation functions in collaborative editing systems: RR-5795[R/OL]. [S.l.]: INRIA, 2005: 45.
https://hal.inria.fr/inria-00071213.

[24] LAMPORT L. The Temporal Logic of Actions[J/OL]. ACM Trans. Program. Lang. Syst., 1994, 16(3): 872–923.
http://doi.acm.org/10.1145/177492.177726.

[25] LAMPORT L. Time, Clocks, and the Ordering of Events in a Distributed System[J/OL]. Commun. ACM, 1978, 21(7): 558–565.
http://doi.acm.org/10.1145/359545.359563.

[26] BURCKHARDT S, GOTSMAN A, YANG H, et al. Replicated Data Types: Specification, Verification, Optimality[C/OL] // POPL '14: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. New York, NY, USA: ACM, 2014: 271–284.
http://doi.acm.org/10.1145/2535838.2535848.

[27] YU Y, MANOLIOS P, LAMPORT L. Model Checking TLA+ Specifications[C/OL] // CHARME '99: Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods. London, UK, UK: Springer-Verlag, 1999: 54–66.
http://dl.acm.org/citation.cfm?id=646704.702012.

致　　谢

本文的工作是在魏恒峰老师的悉心指导下完成的。从毕业设计的选题，相关知识的学习到实验的正式开展，各个阶段的答辩考核和论文的撰写排版，魏老师在各个阶段都给予了我及其宝贵的指导和热情无私的帮助。魏老师一丝不苟的治学风格和精益求精的生活态度亦使我受益匪浅。

感谢分布式算法组的所有老师和同学，很幸运能与你们相遇相知。特别感谢黄宇老师在我迷茫时为我指明了前行的方向，为我日后的人生规划提供了真诚宝贵的指点。

感谢我的父母和其他所有家人，你们在我的成长过程中亦师亦友，你们无条件的爱是我顺利完成学业的坚实后盾。