软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(5):1332−1352 [doi: 10.13328/j.cnki.jos.005956]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

CRDT 协议的 TLA+描述与验证*

纪 业, 魏恒峰, 黄 宇, 吕 建

(计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 魏恒峰, E-mail: hfwei@nju.edu.cn

摘 要: 无冲突复制数据类型(conflict-free replicated data types,简称 CRDT)是一种封装了冲突消解策略的分布式复制数据类型,它能够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.旨在采用模型检验技术验证一系列 CRDT 协议的正确性.具体而言,构建了一个可复用的 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与最终可见性(所有的更新操作最终都会被所有的副本节点接收并处理).使用 TLA+形式化规约语言实现了该框架,然后以 Add-Wins Set 复制数据类型为例,展示了如何使用框架描述具体协议,并使用 TLC 模型检验工具来验证协议的正确性.

关键词: 无冲突复制数据类型;强最终一致性;最终可见性;模型检验;TLA+
中图法分类号: TP301

中文引用格式: 纪业,魏恒峰,黄宇,吕建.CRDT 协议的 TLA+描述与验证.软件学报,2020,31(5):1332−1352. http://www.jos.org.cn/1000-9825/5956.htm

英文引用格式: Ji Y, Wei HF, Huang Y, Lü J. Specifying and verifying CRDT protocols using TLA+. Ruan Jian Xue Bao/Journal of Software, 2020,31(5):1332−1352 (in Chinese). http://www.jos.org.cn/1000-9825/5956.htm

Specifying and Verifying CRDT Protocols Using TLA+

JI Ye, WEI Heng-Feng, HUANG Yu, LÜ Jian

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: Conflict-free replicated data types (CRDT) are replicated data types that encapsulate the mechanisms for resolving concurrent conflicts. They guarantee strong eventual consistency among replicas in distributed systems, which requires replicas that have executed the same set of updates be in the same state. However, CRDT protocols are subtle and it is difficult to ensure their correctness. This study leverages model checking to verify the correctness of CRDT protocols. Specifically, a reusable framework is proposed for modelling and verifying CRDT protocols. The framework consists of four layers, i.e., the communication layer, the interface layer, the protocol layer, and the specification layer. The communication layer models the communication among replicas and implements a variety of communication networks. The interface layer provides a uniform interface for existing CRDT protocols, including both the operation-based protocols and the state-based ones. In the protocol layer, users can choose the appropriate underlying communication network required by a specific protocol. The specification layer specifies strong eventual consistency and the eventual visibility property (i.e., all updates are eventually delivered by all replicas) that every CRDT protocol should satisfy. This framework is implemented using a

* 基金项目: 国家重点研发计划(2017YFB1001801); 国家自然科学基金(61702253, 61772258)

Foundation item: National Key Research and Development Program of China (2017YFB1001801); National Natural Science Foundation of China (61702253, 61772258)

本文由“系统软件构造与验证技术”专题特约编辑赵永望副教授、刘杨教授、王戟教授推荐.

收稿时间: 2019-09-03; 修改时间: 2019-10-24; 采用时间: 2019-12-24; jos 在线出版时间: 2020-04-07

© 中国科学院软件研究所 http://www.jos.org.cn

formal specification language called TLA+. It is also demonstrated that how to model CRDT protocols in this framework and how to verify their correctness via the model checking tool called TLC, taking Add-Wins Set as an example.

Key words: conflict-free replicated data types (CRDT); strong eventual consistency; eventual visibility; model checking; TLA+

为了保证高可用性,分布式系统通常采用数据副本技术,将数据的不同副本存放在不同的物理节点上.为了保证低延迟并且容忍网络分区,用户提交到某个副本节点的操作需要立即返回,而不需要先与其他副本节点进行通信.本地副本节点上的更新操作将以异步的方式发送给其他副本节点[1−3].在这种情况下,多个副本节点上的更新操作将产生冲突,导致数据一致性问题.根据 CAP 定理[4,5],任何一个分布式数据副本系统都无法同时满足(强)数据一致性、可用性及网络分区容忍性这 3 种特性,因此,很多分布式系统都选择提供较弱的数据一致性[6−8],如最终一致性与强最终一致性.最终一致性(eventual consistency)保证了当用户停止发送更新操作后,各个副本节点最终将达到一致的状态[6,9].但是,最终一致性对系统的中间状态没有任何约束.强最终一致性(strong eventual consistency,简称 SEC)则要求执行了相同的更新操作集合(不要求以相同顺序执行)的副本节点具有相同的状态[2,10].强最终一致性反映了分布式系统的安全性(safety),它确保“坏事”一定不会发生.除此之外,一个有用的分布式系统还需要具有活性(liveness),它确保“好事”终将发生[11].最终可见性(eventual visibility,简称 EV)是一种基本的活性性质,它要求所有(更新)操作最终都会被所有的副本节点交付并处理[1,12].

CRDT(conflict-free replicated data type,无冲突复制数据类型)[2,10,12]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(hashtable)、树(tree)与图(graph)等[2,3,13].NoSQL 数据库,如 Redis Labs(Labs:https://redislabs.com/)(Redis(https://redis.io/r)的商用版本)与开源的 Riak(https://riak.com/),提供了对 CRDT 的支持.CRDT 协议分为两类:基于操作的(op-based)协议与基于状态的(state-based)协议.基于操作的 CRDT 协议每次仅广播本地最新执行的更新操作,而基于状态的 CRDT 则将副本节点状态包含在消息中广播给其他节点.CRDT 协议通常需要精巧的设计,正确性难以理解也难以保证[14].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.相对而言,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[15].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 个甚至更少副本节点[16].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 CRDT 协议的正确性,包括安全性(即最终一致性 SEC)与活性(即最终可见性 EV)两方面.具体而言,我们使用 TLA+[17,18]形式化规约语言描述一系列 CRDT 协议,并使用 TLC[19]模型检验工具验证它们的正确性.为了便于验证一系列 CRDT 协议,我们提出了一个可复用的、模块化的 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,包括只提供最基本通信能力的基础(basic)网络、保证消息不丢失不重复的可靠(reliable)网络、满足因果关系的基础网络(basic causal network)与满足因果关系的可靠网络(reliable causal network).
- 协议接口层为 CRDT 协议(包括基于操作的协议与基于状态的协议)提供统一的接口:*IntDo*(*r*)封装了特定 CRDT 所提供的各种操作,*IntSend*(*r*)与 *IntDeliver*(*r*)则封装了与网络通信层之间的交互.
- 在具体协议层,每个协议根据需求选用合适的通信网络并实现上述接口.
- 规约层则描述了所有 CRDT 协议都需要满足的强最终一致性与最终可见性.

为此,我们需要分别针对基于操作的协议与基于状态的协议,为每个副本节点维护其执行过的更新操作集.我们使用 TLA+实现了该框架.然后,我们以 AWSet(Add-Wins set)复制数据类型为例展示如何使用框架描述具体协议.最后,我们使用 TLC 验证 CRDT 协议的正确性.

本文第 1 节先以计数器复制数据类型为例介绍 CRDT 协议的设计与实现,然后介绍 TLA+形式化规约语言并给出基于操作的计数器协议的 TLA+描述.第 2 节介绍 CRDT 协议描述与验证框架,重点介绍网络通信层、协

© 中国科学院软件研究所 http://www.jos.org.cn

议接口层与规约层的设计与实现.第 3 节以 AWSet 复制数据类型为例展示如何使用该框架描述具体协议,包括基于操作的 AWSet 协议与基于状态的 AWSet 协议.第 4 节介绍模型检验实验与验证结果.第 5 节介绍相关工作.第 6 节总结全文,并讨论可能的未来工作.

1 预备知识

1.1 CRDT协议简介

CRDT 协议可以分为两类:基于操作的(op-based)CRDT 协议与基于状态的(state-based)CRDT 协议[1,2,12].在基于操作的协议中,每条消息仅携带自上次广播以来本地节点执行的更新操作.此类协议通常要求底层网络是可靠的,即消息不能丢失、也不能重复交付.有些协议还对消息的交付顺序有额外要求.如果并发的更新操作之间具有可交换性,那么此类协议可以满足强最终一致性.在基于状态的协议中,每条消息会包含发送节点的当前状态.此类协议对底层网络的要求较低,通常能容忍消息的丢失、重复交付与乱序交付.如果节点的状态集合在“合并(merge)”操作下能构成半格(join semilattice),那么此类协议可以满足强最终一致性.下面,我们以仅支持“自增(*inc*)”操作的计数器(counter)复制数据类型为例展示这两类 CRDT 协议的设计理念.

1.1.1 基于操作的计数器协议

在基于操作的计数器协议中[1,2,12],每个副本节点维护一个二元组(c,d):c 表示计数器的当前值,d(称为更新缓冲值)表示自从上次消息广播以来该副本节点执行自增操作的次数.自增操作 *inc* 会同时增加 c 与 d 的值.每个副本节点以异步消息传递的方式将 d 值广播给其他副本节点,同时将 d 置为 0.当副本节点接收到消息 d 时,它会将当前计数器的值 c 更新为 $c+d$.

算法 1. 基于操作的计数器协议(以副本节点 r 为例).

```
▷(c,d): c for counter, d for update buffer
Σ=N0×N0    ▷Σ: replica state
initial(·):(c,d)
    let (c,d)=(0,0)
read(·):v
    let v=c
inc(·)
    c←c+1
    d←d+1
send(·)
    broadcast d to other replicas
    b←0
deliver(d)
    c←c+d
```

该协议要求底层网络是可靠的.由于自增操作具有可交换性,该协议可以保证强最终一致性.

1.1.2 基于状态的计数器协议

在基于状态的计数器协议中[1,2,12],每个副本节点 r 维护一个从副本节点到自然数的映射(也称向量)vv:对于副本节点 s,$vv[s]$表示 r 已接收的来自 s 的自增操作的次数.副本节点 r 上的计数器的当前值就是它的 vv 向量的所有分量之和.自增操作 *inc* 仅增加本地分量 $vv[r]$的值.每个副本节点会以异步消息传递的方式将自己的 vv 向量广播给其他副本节点.当副本节点接收到向量 vv_m 时,它会将 vv_m“合并”到本地向量 vv 中,即将 vv 的每个分量 $vv[s]$更新为 $vv[s]$与 $vv_m[s]$中的较大值.

算法 2. 基于操作的计数器协议(以副本节点 r 为例).

© 中国科学院软件研究所 http://www.jos.org.cn

1: *Replica*: the set of all replicas
2: $\Sigma = Replica \times (Replica \rightarrow N_0)$ ▷Σ: replica state
3: $initial(\cdot):vv$
4: $\forall s \in Replica, vv[s] \leftarrow 0$
5: $read(\cdot):v$
6: **let** $v = \sum_{s \in Replica} vv(s)$
7: $inc(\cdot)$
8: $vv[r] \leftarrow vv[r]+1$
9: $send(\cdot)$
10: broadcast vv_m to other replicas
11: $deliver(vv_m)$
12: $\forall s \in Replica, vv[s] \leftarrow \max(vv[s], vv_m[s])$

该协议可以在任意网络中执行.特别地,它不要求网络是可靠的.由于 *vv* 向量在上述“合并”操作下构成半格,该协议可以保证强最终一致性.

1.2 TLA+简介

TLA+是由 Leslie Lamport 开发的、适于描述并发与分布式系统的形式化规约语言[17,18].TLA+将系统建模为状态机.一个状态机由它可能的初始状态(initial states)与一组动作(actions)来刻画.一个状态是对所有变量的一种赋值.一个动作表达了新、旧状态之间的某种关系(relation),可用包含带撇变量(表示新状态)与不带撇变量(表示旧状态)的公式来描述.例如,$x'=y+42$ 断言新状态中 x 的值比旧状态中 y 的值大 42.系统的一个行为就是一个状态序列,它表示系统的一次可能的执行过程.

TLA+是对时序逻辑 TLA(temporal logic of actions)[20]的扩展.在 TLA+中,一个系统可以表达为 TLA 中的一个形如 $Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge L$ 的时序公式.其中,*Init* 谓词(predicate)刻画了系统所有可能的初始状态,*Next* 定义了系统的次态关系(next-state relation),□是表示“总是(always)”的时序操作符,*vars* 是由系统中的所有变量构成的元组,*L* 表示系统的公平性(fairness)性质.次态关系 *Next* 是由系统的所有动作构成的析取式.$[Next]_{vars}$ 为真当且仅当 *Next* 为真(即某个动作为真,我们称系统执行了该动作)或者 *vars*(即所有变量)保持不变.

Init 和 *Next* 刻画了系统动作发生的可能性,公平性 *L* 则刻画了系统动作发生的必然性.公平性可以根据强弱程度分为弱公平性(weak fairness)与强公平性(strong fairness)[20]:弱公平性要求如果系统的某个动作从某时刻开始是持续可执行的(即该动作的前置条件持续被满足),则该动作终将被执行;强公平性要求如果系统的某个动作是频繁(无穷多次,但不要求持续)可执行的,则该动作终将被执行.TLA+分别使用 *WF* 和 *SF* 操作符表达系统的弱公平性和强公平性.对于动作 *A* 以及由所有变量构成的元组 *vars* 而言,*WF* 与 *SF* 的定义如下.

$$WF_{var}(A) \triangleq \Diamond\Box ENABLED\langle A\rangle_{vars} \Rightarrow \Box\Diamond\langle A\rangle_{vars},$$

$$SF_{vars}(A) \triangleq \Box\Diamond ENABLED\langle A\rangle_{vars} \Rightarrow \Box\Diamond\langle A\rangle_{vars}.$$

其中,$\langle A\rangle_{vars}$ 表示动作 *A* 被执行,定义为 $\langle A\rangle_{vars} \triangleq A \wedge vars' \neq vars$,$ENABLED\langle A\rangle_{vars}$ 则表示动作 *A*(在当前状态下)是可执行的,即它的前置条件被满足.◇是表示“最终(eventually)”的时序操作符.组合操作符 $\Diamond\Box F$ 表示 *F* 从某时刻开始持续(eventually always)成立,即存在时刻 *t*,对于任意时刻 $t' \geqslant t$,*F* 均成立;$\Box\Diamond F$ 表频繁(infinitely often)成立,即对任意时刻 *t*,均存在时刻 $t' \geqslant t$,使得 *F* 在 t' 时刻成立.

TLA+在 TLA 的基础上,加入了一阶谓词逻辑以及 ZF 集合论,从而支持丰富的数据类型与表达式.图 1 总结了本文使用到的逻辑与集合操作符(operator).文献[21]给出了完整的 TLA+操作符列表.

TLA+规约以模块(module)的形式组织在一起.在每个模块中,我们可以声明常量(CONSTANTS)与变量(VARIABLES)、定义操作符(operator)或者提出定理(THEOREM *P*).一个模块 *M* 可以通过扩展(extend)其他模块

© 中国科学院软件研究所 http://www.jos.org.cn

$M_1,\ldots,M_n$ 的方式引入声明、定义与定理;在模块 M 中,写作 EXTENDS $M_1,\ldots,M_n$.模块也可以被实例化(instantiated).考虑 M 模块中的实例化语句:

$$IM_1 \triangleq \text{INSTANCE } M_1 \text{ WITH } p_1 \leftarrow e_1,\ldots,p_n \leftarrow e_n.$$

其中,p_i 包含了 M_1 中的所有常量与变量,e_i 是 M 中的合法表达式.对模块 M_1 中的任一操作符 F 及其定义 d,该语句在模块 M 中引入了操作符 $IM_1!F$,其定义是将 d 中的 p_i 替换为相应的 e_i.此外,TLA+中的隐式替换规则允许我们在 e_j 与 p_j 相同时省略 $p_j \leftarrow e_j$ 子句.

	操作符	含义
逻辑	CHOOSE $x \in S:p$	选择集合S中满足条件p的元素x (通常用于符合条件的x是唯一的情况下)
集合	SUBSET S	S 的幂集
	$\{e:x \in S\}$	将e作用在S中所有元素得到的集合,比如$\{x^2:x \in S\}$
	$\{x \in S:p\}$	S 中满足条件 p 的元素构成的集合
函数	$f[e]$	函数 f 作用在参数 e 上
	$[x \in S \mapsto e]$	对于 $x \in S$,使得 $f(x)=e$ 的函数 f
	$[f \text{ EXCEPT } ![e_1]=e_2]$	函数 $\hat{f}:\hat{f}[e]=\begin{cases} e_2, & \text{if } e=e_1 \\ f[e], & \text{otherwise} \end{cases}$
	$[f \text{ EXCEPT } ![c]=e]$,其中,$e$ 包含符号@	e 中的@表示 $f[c]$
记录	$e.h$	记录 e 的域 h
	$[h_1 \mapsto e_1,\ldots,h_n \mapsto e_n]$	域 h_i 为 e_i 的记录
	$[h_1:S_1,\ldots,h_n:S_n]$	满足域 h_i 属于 S_i 的所有记录构成的集合
	$[r \text{ EXCEPT } !.h=e]$	记录 $\hat{r}:[h_1:S_1,\ldots,h:e,\ldots,h_n:S_n]$
	$[r \text{ EXCEPT } !.h=e]$,其中,$e$ 包含符号@	e 中的@表示 $r.h$
元组	$e[i]$	元组 e 的第 i 个分量
动作操作符	e'	动作的新状态中 e 的值
	UNCHANGED e	e 不变:$e'=e$
	$[A]_e$	动作 A 成立或者 e 不变:$A \vee (e'=e)$
时序操作符	$\Box F$	F 在所有情况下均成立($\Box$表示“always”)
	$\Diamond F$	F 最终成立($\Diamond$表示“eventually”)
	$\Diamond\Box F$	F 从某时刻开始持续(eventually always)成立
	$\Box\Diamond F$	F 频繁(infinitely often)成立
	$F \rightsquigarrow G$	F 导致(leads to)G
	$WF_e(A)$	动作 A 的弱公平性(weak fairness)
	$SF_e(A)$	动作 A 的强公平性(strong fairness)

Fig.1 A summary of TLA+ operators used in this paper

图 1 本文所使用的 TLA+操作符

TLC 是 TLA+的模型检验工具[19],它通过遍历 TLA+规约的有穷实例的状态空间验证规约的正确性.这些有穷实例被称为 TLA+规约的 TLC 模型(models).例如,考虑一个包含多个进程的分布式系统.在规约中,我们使用 CONSTANTS *Proc* 表示可能的进程集合.在对该规约进行验证时,我们需要在 TLC 模型中将 *Proc* 实例化为有穷的进程集合,如 $Proc \triangleq \{1,2,3\}$.为了体现 *Proc* 取值的特殊性,我们也可以用 TLA 模型值(model value)表示进程.在 TLA+,一个模型值与其他任何值都不相等.因此,我们可以将 *Proc* 实例化为一组模型值,如 $Proc \triangleq \{p_1,p_2,p_3\}$.并且,如果对一组模型值进行任意重排(比如将 p_1 替换为 p_2,将 p_2 替换为 p_3,将 p_3 替换为 p_1)都不影响系统行为是否满足给定规约,我们可以进一步将 *Proc* 标记为对称集(symmetry set)[17],从而减少 TLC 需要遍历的状态空间.

模块 *OpBasedCounter* 展示了如何使用 TLA+描述基于操作的计数器协议.在 *OpBasedCounter* 模块中,每个副本节点 r 维护 4 个变量:$c[r]$表示计数器的当前值,$d[r]$表示更新缓冲值,$incoming[r]$表示接收消息的信道,$seq[r]$表示节点 r 已发送消息的数目.*Msg* 表示消息类型,其中包括 d 值和用来确保消息唯一性的 r 和 seq.*Init* 表示每个副本节点 r 的初始状态:$c[r]=0,d[r]=0,incoming[r]=\{\cdot\},seq[r]=0$.*Next* 表示每个副本节点 r 可能的动作:$Do(r)$包

© 中国科学院软件研究所 http://www.jos.org.cn

含更新操作 *Inc*(*r*)与查询操作 *Read*(*r*);*Send*(*r*)表示 *r* 将 *d*[*r*]广播给其他副本节点;*Deliver*(*r*)表示 *r* 交付并处理一条消息.在 *OpBasedCounter* 中,我们使用集合表示 *incoming* 信道,*Deliver*(*r*)中交付消息的方式建模了该协议所需的可靠网络(后文第 2.3 节详细描述了如何使用 TLA+建模各种类型的网络).

```
---------------------------- MODULE OpBasedCounter ----------------------------
EXTENDS Naturals
CONSTANTS Replica                    \* the set of replicas
-----------------------------------------------------------------------------
VARIABLES c,                         \* c[r]: current value at r \in Replica
          d,                         \* d[r]: buffer of increments at r \in Replica
          incoming,                  \* incoming[r]: incoming channel at replica r \in Replica
          seq                        \* seq[r]: local sequence number for messages sent by r \in Replica
Msg==[r:Replica,seq:Nat,d:Nat]       \* message type
vars==⟨⟨c,d,incoming,seq⟩⟩
-----------------------------------------------------------------------------
TypeOK==∧c \in [Replica→Nat]
        ∧d \in [Replica→Nat]
        ∧incoming \in [Replica→SUBSET Msg]
        ∧seq \in [Replica→Nat]
-----------------------------------------------------------------------------
Init==∧c=[r \in Replica↦0]
      ∧d=[r \in Replica↦0]
      ∧incoming=[r \in Replica↦{·}]
      ∧seq=[r \in Replica↦0]
-----------------------------------------------------------------------------
Read(r)==c[r]
Inc(r)==∧c'=[c EXCEPT ![r]=@+1]
        ∧d'=[d EXCEPT ![r]=@+1]
        ∧UNCHANGED ⟨⟨incoming,seq⟩⟩
Do(r)==Inc(r)                        \* We ignore Read(r) since it does not modify states.
-----------------------------------------------------------------------------
Send(r)==
    ∧d[r] # 0
    ∧d'=[d EXCEPT ![r]=0]
    ∧seq'=[seq EXCEPT ![r]=@+1]
    ∧LET m==[r↦r,seq↦seq[r],d↦d[r]]
       IN incoming'=[x \in Replica↦IF x=r THEN incoming[x]
                                          ELSE incoming[x] \cup {m}]
    ∧UNCHANGED ⟨⟨c⟩⟩
Deliver(r)==
    ∧incoming[r] # {}
    ∧\E m \in incoming[r]:           \* choose m from incoming[r] non-deterministically
       ∧c'=[c EXCEPT ![r]=@+m.d]
       ∧incoming'=[incoming EXCEPT ![r]=@\{m}]        \* delete m
    ∧UNCHANGED ⟨⟨d,seq⟩⟩
-----------------------------------------------------------------------------
Next==\E r \in Replica:Do(r)∨Send(r)∨Deliver(r)
Spec==Init∧[·][Next]_vars
=============================================================================
```

2 CRDT 协议描述与验证框架

本文旨在验证一系列 CRDT 协议的正确性,包括安全性(强最终一致性 SEC)与活性(最终可见性 EV).为了描述众多的 CRDT 协议,我们构建了一个可供复用、便于扩展的 CRDT 协议描述与验证框架.该框架分为 4 层,包括网络通信层、协议接口层、具体协议层与规约层(如图 2 所示).

- 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,包括只提供最基本通信能力的基础(basic)网络、保证消息不重复不丢失的可靠(reliable)网络、满足因果关系的基础网络(basic

© 中国科学院软件研究所 http://www.jos.org.cn

causal network)与满足因果关系的可靠网络(reliable causal network).

- 协议接口层为两类 CRDT 协议(包括基于操作的协议与基于状态的协议)提供统一的接口:*IntDo*(*r*)封装了特定 CRDT 所提供的各种操作,*IntSend*(*r*)与 *IntDeliver*(*r*)则封装了与网络通信层之间的交互.
- 在具体协议层,每个协议根据需求选用合适的通信网络并实现上述接口.
- 规约层则描述了所有 CRDT 协议都需要满足的强最终一致性与最终可见性.

本节先介绍系统模型,然后分别介绍协议接口层、网络通信层与规约层.后文第 3 节以 AWSet 为例介绍具体协议层.

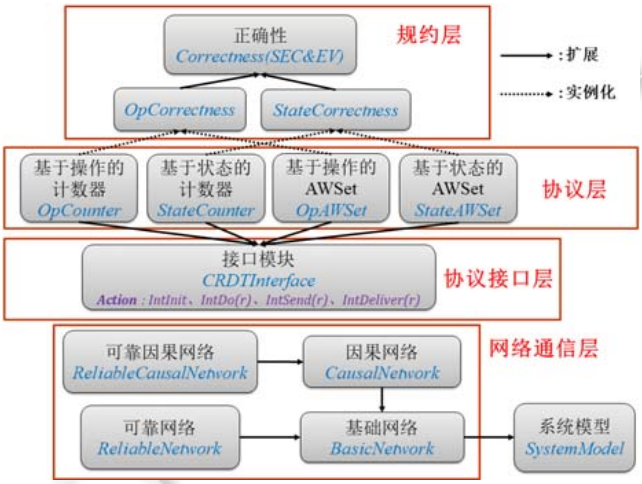

Fig.2 Framework for specifying and verifying CRDT protocols

图 2 CRDT 协议描述与验证框架

2.1 系统模型

在模块 *SystemModel* 中,常量 *Replica* 表示副本节点集合.副本节点之间通过异步消息传递进行通信.*Msg* 表示可能的消息集合,其类型由具体协议决定.每个副本节点 *r* 维护一个用于接收消息的 *incoming*[*r*]信道,用集合表示.

```
---------------------------- MODULE SystemModel ----------------------------
CONSTANTS
    Replica,            \* the set of replicas
    Msg,                \* the set of messages
    InitMsg             \* special message for initialization
----------------------------------------------------------------------------
VARIABLES incoming  \* incoming[r]: incoming channel at replica r \in Replica
----------------------------------------------------------------------------
SMTypeOK==          \* each incoming channel is a set of messages
incoming \in [Replica→SUBSET Msg]
============================================================
```

2.2 协议接口层

协议接口层模块 *CRDTInterface* 为两类 CRDT 协议提供统一的接口:*IntInit*(*r*),*IntDo*(*r*),*IntSend*(*r*)与 *IntDeliver*(*r*).*IntInit* 负责初始化副本节点的状态.*IntDo*(*r*)用以封装特定 CRDT 所提供的各种操作.*IntSend*(*r*)与 *IntDeliver*(*r*)封装了与网络通信层之间的交互.具体的 CRDT 协议将扩展该模块,并实现特定于该 CRDT 的 *Init*, *Do*(*r*),*Send*(*r*)与 *Deliver*(*r*)动作.比如,假设某 CRDT 支持两个操作(在 TLA+实现中对应于两个操作符)Op_1 与 Op_2,则该 CRDT 所实现的 *Do*(*r*)可以定义为 $Do(r) \triangleq IntDo(r) \wedge Op_1 \wedge Op_2$.

协议接口层的主要作用是为 *IntDo*(*r*)与 *IntSend*(*r*)生成唯一的动作标识.每个动作标识 *aid*∈*Aid* 是一个由副本节点 *r*∈*Replica* 与本地单调递增的序号 *seq*[*r*]∈*Nat* 构成的二元组.动作标识主要有 3 个方面的用途.第一,用作消息标识符.在具体协议中,每条被广播的消息都携带 *IntSend*(*r*)动作对应的 *aid*.因此在网络通信层,消息具有唯一性.第二,用作操作标识符.要验证 SEC 和 EV,需要为每个副本节点记录它已经执行的更新操作集.在 *Correctness* 模块中,我们将使用 *IntDo*(*r*)对应的 *aid* 标识这些更新操作.第三,用作数据标识符.例如,为了避免并发冲突,AWSet 协议会为每个新添加的数据分配唯一的标识符(见后文第 3 节).

```
---------------------------- MODULE CRDTInterface ----------------------------
EXTENDS Naturals
CONSTANTS Replica

Aid==[r:Replica,seq:Nat]     \* action id
-----------------------------------------------------------------------------
VARIABLES seq              \* seq[r]: local sequence number for actions at replica r \in Replica
   \* incremented upon IntDo and IntSend

IntTypeOK==seq \in [Replica→Nat]
-----------------------------------------------------------------------------
IntInit==seq=[r \in Replica↦0]

IntDo(r)==seq'=[seq EXCEPT ![r]=@+1]

IntSend(r)==seq'=[seq EXCEPT ![r]=@+1]

IntDeliver(r)==UNCHANGED ⟨⟨d,seq⟩⟩
=============================================================================
```

2.3 网络通信层

不同的 CRDT 协议对底层通信网络有不同的需求.总体而言,基于状态的 CRDT 协议可以容忍消息的丢失、重复交付与乱序交付,可以在任何网络条件下执行.基于操作的协议则要求网络是可靠的(即每条消息恰好被交付一次),如第 1.1.1 节介绍的基于操作的计数器协议.此外,有些基于操作的协议对消息的接收顺序还有特殊要求.例如,第后文 3.1 节介绍的基于操作的 AWSet 协议需要按照因果序交付消息[1,2].

已知的 CRDT 协议所依赖的网络通信类型可分为 4 类[1].

- 基础网络(basic network):基础网络允许消息的丢失(即从未被交付)、重复交付与乱序交付.
- 可靠网络(reliable network):可靠网络保证每条消息都被每个副本节点恰好交付一次.
- 因果网络(causal network):因果网络要求按照因果序交付消息.此处的因果序是由消息之间的“先于”(happen-before)关系[22]决定的.假设消息 m_1 与 m_2 分别由副本节点 r_1 与 r_2 广播,则 m_1 先于 m_2 发生当且仅当:
  - r_1 与 r_2 是同一副本节点且 m_1 在 m_2 之前被广播;或
  - r_2 在广播 m_2 之前已交付过消息 m_1;或
  - 存在消息 m',使得 m_1 先于 m' 发生且 m' 先于 m_2 发生.

  因果网络要求各个副本节点按照消息之间的“先于”关系交付消息.也就是说,只有当先于消息 *m* 发生的所有消息都被交付之后,*m* 才能被交付.
- 可靠因果网络(reliable causal network):可靠因果网络同时满足可靠网络与因果网络的要求.

下面我们分别介绍以上 4 种网络通信类型在 TLA+中的实现.具体 CRDT 协议可以通过 TLA+提供的实例化(INSTANCE)机制引入并使用合适的网络通信类型.

2.3.1 *BasicNetwork* 模块建模基础网络

BasicNetwork 模块通过扩展 *SystemModel* 模块引入表示信道的变量 *incoming*.变量 *lmsg*[*r*]表示在副本节点 *r* 上,网络通信层向协议层交付的最近一次消息.*BNBroadcast*(*r*,*m*)表示副本节点 *r* 将消息 *m* 广播给其他所有副

© 中国科学院软件研究所 http://www.jos.org.cn

本节点.*BNDeliver*(*r*)建模副本节点 *r* 向协议层交付消息的行为:当信道 *incoming*[*r*]不为空时,它非确定性地(建模消息的乱序交付)从 *incoming*[*r*]中选择一条消息 *m*,将其交付给协议层.需要注意的是,*BNDeliver*(*r*)并未删除 *m*(UNCHANGED *incoming*),因此可以建模消息的重复交付以及消息丢失(即存在消息永远不被交付).

```
---------------------------- MODULE BasicNetwork ----------------------------
EXTENDS SystemModel
-----------------------------------------------------------------------------
VARIABLES lmsg        \* lmsg[r]: the last message delivered at r \in Replica to the upper-layer protocol

nVars==⟨⟨incoming,lmsg⟩⟩
-----------------------------------------------------------------------------
BNInit==
    ∧incoming=[r \in Replica↦{·}]
    ∧lmsg=[r \in Replica↦InitMsg]

BNBroadcast(r,m)==
    /\ incoming'=[x \in Replica↦IF x=r THEN incoming[x]
                                       ELSE incoming[x] \cup {m}]
    /\ UNCHANGED ⟨⟨lmsg⟩⟩

BNDeliver(r)==
    ∧incoming[r] # {·}    \* choose m from incoming[r] non-deterministically
    ∧\E m \in incoming[r]: lmsg'=[lmsg EXCEPT ![r]=m]
    ∧UNCHANGED ⟨⟨incoming⟩⟩
=============================================================
```

2.3.2 *ReliableNetwork* 模块建模可靠网络

ReliableNetwork 模块扩展了 *BasicNetwork* 模块,并向协议层提供额外的可靠性保障.

具体而言,*ReliableNetwork* 保持了 *BasicNetwork* 中的消息广播行为.但是与 *BasicNetwork* 中的 *BNDeliver*(*r*)不同,*ReliableNetwork* 中的 *RNDeliver*(*r*)会将已交付的消息从信道中删除,因此消息不会被重复交付.而且在消息有限的情况下,该设计方案可以保证每条消息最终都会被交付(即消息不会丢失).

```
--------------------------- MODULE ReliableNetwork ---------------------------
EXTENDS BasicNetwork
-----------------------------------------------------------------------------
rnVars==⟨⟨incoming,lmsg⟩⟩
-----------------------------------------------------------------------------
RNInit==BNInit

RNBroadcast(r,m)==BNBroadcast(r,m)

RNDeliver(r)==
    ∧incoming[r] # {·}
    ∧\E m \in incoming[r]:
        ∧lmsg'=[lmsg EXCEPT ![r]=m]                \* choose m from incoming[r] non-deterministically
        ∧incoming'=[incoming EXCEPT ![r]=@ \{m}]   \* delete m
=============================================================
```

2.3.3 *CausalNetwork* 模块建模因果网络

CNBroadcast(*r*,*m*)建模因果网络中副本节点 *r* 广播消息 *m* 的行为:*vc*[*r*][*r*]分量加一,并将更新后的本地向量时钟 *vc*′[*r*]作为消息 *m* 的时间戳(保存在 *lvc* 域),构成新的消息 *cm*(可通过 *ts*(*cm*)访问 *lvc*),然后将 *cm* 广播给其他副本节点.*CNDeliver*(*r*)建模因果网络中副本节点 *r* 向协议层交付消息的行为.*CNCausallyReady*(*r*,*cm*)用于判断消息 *cm* 是否可以被副本节点 *r* 交付,即是否所有“先于"*cm* 的消息都已被 *r* 交付.设广播消息 *cm* 的副本节点为 *mr*.条件 *ts*(*cm*)[*mr*]≤*vc*[*r*][*mr*]+1 表示副本节点 *mr* 在广播 *cm* 之前所广播的消息都已被 *r* 交付.条件∀*s*≠*mr*, *ts*(*cm*)[*s*]≤*vc*[*r*][*s*]表示副本节点 *mr* 在广播 *cm* 之前所交付的来自其他副本节点的消息都已被 *r* 交付.

© 中国科学院软件研究所 http://www.jos.org.cn

CNDeliver(*r*)非确定性地从信道 *incoming*[*r*]中选择一条满足 *CNCausallyReady* 的消息 *cm*(实际消息为 *cm.m*),交付给上层协议,并将本地向量时钟 *vc*[*r*]的 *vc*[*r*][*mr*]分量更新为 Max(*vc*[*r*][*mr*],*ts*(*cm*)[*mr*]).

CausalNetwork 模块扩展 *BasicNetwork* 模块,并向协议层提供按因果序交付消息的保障.*CausalNetwork* 模块使用向量时钟刻画消息之间的因果序[1,23].为此,每个副本节点 *r* 维护一个 *n* 维向量 *vc*[*r*](*n* 为副本节点数),其中,*vc*[*r*][*s*]刻画了副本节点 *r* 观察到的来自副本节点 *s* 的最新消息.对任意副本节点 *r*,*s*,*vc*[*r*][*s*]的初始值为 0.

需要注意的是,*CNDeliver*(*r*)没有将 *cm* 从信道中删除,因此允许消息的丢失与重复交付.特别地,为了能在因果网络中建模消息的重复交付,*CNCausallyReady*(*r*,*cm*)使用了条件 *ts*(*cm*)[*mr*]≤*vc*[*r*][*mr*]+1,而不是 *ts*(*cm*)[*mr*]=*vc*[*r*][*mr*]+1.相应地,*CNDeliver*(*r*)将 *vc*[*r*][*mr*]更新为 *vc*[*r*][*mr*]与 *ts*(*cm*)[*mr*]中的较大值,而不是设置为 *ts*(*cm*)[*mr*].

```
---------------------------- MODULE CausalNetwork ----------------------------
EXTENDS BasicNetwork, Naturals
------------------------------------------------------------------------------
VARIABLES
    vc                      \* vc[r][s] denotes the latest message from s \in Replica observed by r \in Replica

cnVars==⟨⟨incoming,lmsg,vc⟩⟩
------------------------------------------------------------------------------
ts(cm)==cm.lvc              \* timestamp (vector clock) for cm
sender(cm)==cm.m.aid.r      \* the replica that sends cm
Max(a,b)==IF a>b THEN a ELSE b
------------------------------------------------------------------------------
CNTypeOK==
    ∧SMTypeOK
    ∧vc=[Replica→[Replica→Nat]]          \* vc[r]: vector clock at r \in Replica
------------------------------------------------------------------------------
CNInit==
    ∧BNInit
    ∧vc=[r \in Replica↦[s \in Replica↦0]]   \* \A r, s, vc[r][s]=0

CNBroadcast(r,m)==
    ∧vc'=[vc EXCEPT ![r][r]=@+1]
    ∧LET cm==[m↦m,lvc↦vc'[r]]            \* assign lvc to m
       IN BNBroadcast(r,cm)

CNCausallyReady(r,cm)==                  \* whether cm is causally ready to be delivered by r \in Replica
    LET mr==sender(cm.m)                 \* cm : message with vector clock
    IN ∧ts(cm)[mr]⇐vc[r][mr]+1
       ∧\A s \in Replica \{mr}: ts(cm)[s]⇐vc[r][s]

CNDeliver(r)==
    ∧incoming[r] # {·}
    ∧\E cm \in incoming[r]:
          ∧CNCausallyReady(r,cm)
          ∧LET mr==sender(cm)
             IN vc'=[vc EXCEPT ![r][mr]=Max(@,ts(cm)[mr])]    \* update vc[r]
          ∧lmsg'=[lmsg EXCEPT ![r]=cm.m]
    ∧UNCHANGED ⟨⟨incoming⟩⟩
==============================================================================
```

2.3.4 *ReliableCausalNetwork* 模块建模可靠因果网络

可靠因果网络 *ReliableCausalNetwork* 同时满足可靠网络对可靠性以及因果网络对因果序的要求.

ReliableCausalNetwork 扩展了 *CausalNetwork*,其消息广播行为 *RCNBroadcast*(*r*,*m*)与 *CausalNetwork* 中的 *CNBroadcast*(*r*,*m*)相同.为了建模 *ReliableCausalNetwork* 的可靠性,*RCNDeliver*(*r*)会将交付给上层协议的消息 *cm* 从信道中删除.特别地,由于该网络不允许消息的重复交付,与 *CausalNetwork* 中的 *CNCausallyReady*(*r*,*cm*)相比,*RCNCausallyReady*(*r*,*cm*)可以改用稍强的 *ts*(*cm*)[*mr*]=*vc*[*r*][*mr*]+1 条件.相应地,在 *RCNDeliver*(*r*)中,可以将

© 中国科学院软件研究所 http://www.jos.org.cn

$vc[r][mr]$更新为 $ts(cm)[mr]$.

```
---------------------- MODULE ReliableCausalNetwork ----------------------
EXTENDS CausalNetwork

rcnVars==⟨⟨incoming,lmsg,vc⟩⟩
---------------------------------------------------------------------------
RCNInit==CNInit

RCNBroadcast(r,m)==CNBroadcast(r,m)

RCNCausallyReady(r,cm)==                    \* whether cm is causally ready to be delivered by r \in Replica
    LET mr==sender(cm)                      \* cm: message with vector clock
    IN ∧ts(cm)[mr]=vc[r][mr]+1              \* no message duplication
       ∧\A s \in Replica \{mr}: ts(cm)[s]≤vc[r][s]

RCNDeliver(r)==
    ∧incoming[r] # {·}
    ∧\E cm \in incoming[r]:
        ∧RCNCausallyReady(r,cm)
        ∧LET mr==sender(cm)
            IN vc'=[vc EXCEPT ![r][mr]=ts(cm)[mr]]      \* update vc[r]
        ∧lmsg'= [lmsg EXCEPT ![r]=cm.m]
        ∧incoming'=[incoming EXCEPT ![r]=@ \{cm}]       \* delete cm
===========================================================================
```

2.4 规约层

规约层给出了所有 CRDT 协议都应满足的强最终一致性(SEC)和最终可见性(EV)的形式化定义.强最终一致性要求(交付)执行了相同更新操作集的副本节点应具有相同的状态.最终可见性要求每个(本地)更新操作最终会被交付给所有副本节点.为此,在模块 *Correctness* 中,每个副本节点 r 维护了两个集合:$doset[r]$是它产生的更新操作构成的集合,$delset[r]$是它交付(执行)过的更新操作构成的集合.另外,模块 *Correctness* 假设每个 CRDT 都提供一个读操作 $Read(r\in Replica)$,用于返回副本节点 r 的当前状态.使用 *Read* 与 *delset*,强最终一致性可以定义为 $SEC\triangleq\forall r_1,r_2\in Replica: delset[r_1]=delset[r_2]\Rightarrow Read(r_1)=Read(r_2)$.使用 *doset* 与 *delset*,最终可见性可以定义为

$$EV\triangleq\forall aid\in Aid,\forall r\in Replica: aid\in doset[r]\rightsquigarrow(\forall s\in Replica: aid\in delset[s]).$$

其中,$\forall aid\in Aid$ 枚举所有可能的更新操作.时序操作符$\rightsquigarrow$表示“导致”(leads to),定义为 $F\rightsquigarrow G\triangleq\Box(F\Rightarrow\Diamond G)$[20].

```
---------------------------- MODULE Correctness ----------------------------
EXTENDS Naturals, CRDTInterface
---------------------------------------------------------------------------
CONSTANTS Read(_)     \* Read(r \in Replica): the read operation at r

UMsg==[aid: Aid, update: SUBSET Aid]     \* update message type
---------------------------------------------------------------------------
VARIABLES
    doset,              \* doset[r]: the set of updates generated by replica r \in Replica
    delset,             \* delset[r]: the set of updates delivered by replica r \in Replica
    uincoming           \* uincoming[r]: incoming channel for broadcasting/delivering updates at r \in Replica

CTypeOK==
    ∧doset \in [Replica→SUBSET Aid]
    ∧delset \in [Replica→SUBSET Aid]
    ∧uincoming \in [Replica→SUBSET UMsg]
---------------------------------------------------------------------------
CInit==
    ∧doset=[r \in Replica ↦{·}]
    ∧delset=[r \in Replica ↦{·}]
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
    ∧uincoming=[r \in Replica ↦{·}]

CDo(r)==
    ∧doset′=[doset EXCEPT ![r]=@ \cup {[r↦r,seq↦seq[r]]}]
    ∧delset′=[delset EXCEPT ![r]=@ \cup {[r↦r,seq↦seq[r]]}]
    ∧UNCHANGED ⟨⟨uincoming⟩⟩

CSend(r)==UNCHANGED ⟨⟨delset, doset⟩⟩                \* implemented by OpCorrectness and StateCorrectness

CDeliver(r,aid)==                                    \* choose the update message um according to aid
    ∧LET um==CHOOSE m \in uincoming[r]: m.aid=aid    \* um is unique
        IN delset′=[delset EXCEPT ![r]=@ \cup um.update]
    ∧UNCHANGED ⟨⟨uincoming,doset⟩⟩
-------------------------------------------------------------------------------
SEC==\A r1, r2 \in Replica: delset[r1]=delset[r2]⇒Read(r1)=Read(r2)

EV==\A aid \in Aid, r \in Replica: aid \in doset[r]⇝(\A s \in Replica: aid \in delset[s])
===============================================================================
```

如第 2.2 节所述,我们使用动作标识符 *aid* 代表更新操作.在 *CDo*(*r*)中,新产生的本地更新操作(的动作标识符)被添加到 *doset*[*r*]和 *delset*[*r*](本地操作立即交付执行)中.副本节点之间需要通过消息广播交换它们已执行过的更新操作集.*UMsg* 定义了此类消息的类型:*aid* 用于唯一标识每条消息,*update* 是需要广播的更新操作集.为了将协议的实现与验证解耦,我们使用与 *incoming* 独立的 *uincoming* 信道发送、交付 *UMsg* 消息.也就是说,在具体协议的 *Send*(*r*)动作中,验证正确性所需的 *UMsg* 消息(使用 *uincoming*[*r*]信道)会伴随协议发送的消息(使用 *incoming*[*r*]信道)一同广播给其他副本节点.它们具有相同的 *aid*(均由 *Send*(*r*)决定).具体协议的 *Deliver*(*r*)在交付了某条消息 *m* 后,可以使用 *CDeliver*(*r*,*m.aid*)取出与 *m* 一同被广播的 *UMsg* 消息 *um*,并将 *um* 携带的更新操作集 *um.update* 合并到副本节点 *r* 的更新操作集 *delset*[*r*]中.

```
---------------------------- MODULE OpCorrectness ----------------------------
EXTENDS Correctness
-----------------------------------------------------------------------------
VARIABLES buset    \* buset[r]: the buffer of local updates generated by r \in Replica since the last broadcast
-----------------------------------------------------------------------------
OpCInit==
    ∧CInit
    ∧buset=[r \in Replica↦{·}]

OpCDo(r)==
    ∧CDo(r)
    ∧buset′=[buset EXCEPT ![r]=@ \cup {[r↦r,seq↦seq[r]]}]    \*collect a new update

OpCSend(r)==
    ∧CSend(r)
    ∧uincoming′=[x \in Replica↦            \* broadcast buset[r]
                IF x=r THEN uincoming[x]
                       ELSE uincoming[x] \cup {[aid↦[r↦r,seq↦seq[r]],update↦buset[r]]}]
    ∧buset′=[buset EXCEPT ![r]={·}]      \* clear buset[r]

OpCDeliver(r,aid)==
    ∧CDeliver(r,aid)
    ∧UNCHANGED ⟨⟨buset⟩⟩
=============================================================================
```

在 *CSend*(*r*)中,哪些更新操作需要被广播是由协议类型决定的.模块 *OpCorrectness* 与 *StateCorrectness* 扩展了 *Correctness*,分别针对基于操作的协议与基于状态的协议实现了广播 *UMsg* 消息的行为.具体协议可以依据协议的类型选用 *OpCorrectness* 或者 *StateCorrectnes*.在 *OpCorrectness* 中,*buset*[*r*]表示最近一次广播之后副本节

© 中国科学院软件研究所 http://www.jos.org.cn

点 r 产生的更新操作集.*OpCSend*(r)每次仅广播 *buset*[r](而非完整的 *delset*[r]),并清空 *buset*[r].在 *StateCorrectnes* 中,*StateCSend*(r)则每次都将完整的 *delset*[r]广播给其他副本节点.

```
------------------------- MODULE StateCorrectness -------------------------
EXTENDS Correctness
---------------------------------------------------------------------------
StateCInit==CInit

StateCDo(r)==CDo(r)

StateCSend(r)==
    ∧CSend(r)
    ∧uincoming'=[x \in Replica↦     \* broadcast delset[r]
                IF x=r THEN uincoming[x]
                        ELSE uincoming[x] \cup {[aid↦[r↦r,seq↦seq[r]],update↦delset[r]]}]

StateCDeliver(r,aid)==CDeliver(r,aid)
===========================================================
```

3 CRDT 协议的描述

本节以 AWSet(Add-Wins set)(在参考文献[2]中,AWSet 也被称为 OR-Set(observed-remove set))为例,展示如何使用框架描述具体 CRDT 协议.基于该框架,AWSet 协议只需选择合适的底层通信网络并依照协议接口实现基本的数据添加 Add 与删除 Remove 操作.在集合数据类型上,如果一个添加操作与另一个删除操作并发(即它们之间没有 happens-before 关系,类似消息之间的 happens-before 关系,同样可定义操作之间的 happens-before 关系)且作用在同一数据上,则称它们产生了并发冲突.AWSet 采用“添加操作胜出”的策略消解并发冲突[1,2,24].因此,某数据在 AWSet 中当且仅当存在某个添加操作添加了该数据,并且所有删除该数据的操作要么“先于”(在操作之间的 happens-before 关系下)该添加操作要么与该添加操作并发.

```
------------------------------- MODULE AWSet -------------------------------
EXTENDS CRDTInterface
-----------------------------------------------------------------------------
CONSTANTS Data     \* the set of data
Element==[aid: Aid, d: Data]  \* the set of elements
===========================================================
```

本节实现两个已有的 AWSet 协议:基于操作的 AWSet 协议(*OpAWSet*)[1,2]与基于状态的 AWSet 协议(*StateAWSet*)[1,2].为了消解并发冲突,AWSet 协议为每个被添加的数据(包括重复添加的数据)分配唯一的标识(使其成为“元素”).删除数据 d 则转化为删除当前副本节点上所有数据域为 d 的元素.因此,原本存在并发冲突的数据添加与删除操作被转化为没有并发冲突的元素添加与删除操作.当添加操作和删除操作并发作用在同一数据上时,删除操作未观察到添加操作所添加的元素标识,因此该元素被保留,即添加操作胜出.这两种 AWSet 协议的不同之处在于:*OpAWSet* 每次仅广播自上次广播以来本地节点添加与删除的元素,而 *StateAWSet* 则需要广播本地节点当前的完整状态.在 *AWSet* 模块中,*Element* 定义了元素的类型:d 表示数据,即实际被添加到 AWSet 中的数据;(r,k)表示元素的标识,由副本节点 r 本身与自然数 k 构成.*OpAWSet* 与 *StateAWSet* 扩展了 *AWSet*,并各自实现 *Init*,*Do*(r)(包括 *Add*(d,r)与 *Remove*(d,r)),*Send*(r)与 *Deliver*(r)接口.

3.1 基于操作的AWSet协议

在基于操作的 AWSet 协议[1,2](*OpAWSet* 模块)中,每个副本节点 r 维护 3 个集合:活跃(active)元素集 *aset*[r]保存 r 观察的被添加过且未被删除过的元素,添加缓冲区 *abuf*[r]保存自上次消息广播以来 r 本地添加的元素,删除缓冲区 *rbuf*[r]则保存自上次消息广播以来 r 本地删除的元素.此外,基于操作的 *OpAWSet* 协议需要在可靠因果网络中执行.因此,我们使用 TLA+提供的 INSTANCE 机制引入 *ReliableCausalNetwork* 网络通信模块.下面,

© 中国科学院软件研究所 http://www.jos.org.cn

我们分别描述该协议的初始状态 *Init* 与可能的动作 *Do*(*r*),*Send*(*r*)与 *Deliver*(*r*).

- 在初始状态,每个副本节点 *r* 上的 *aset*[*r*],*abuf*[*r*]与 *rbuf*[*r*]都为空.
- *Do*(*r*)包含两个更新操作和一个查询操作:
  - 更新操作 *Add*(*d*,*r*)表示副本节点 *r* 向 AWSet 添加数据 *d*.该协议为数据 *d* 分配唯一标识 *aid*,将元素(*aid*,*d*)加入到元素集 *aset*[*r*]中.此外,它还将该元素加入到添加缓冲区 *abuf*[*r*]中.
  - 更新操作 *Remove*(*d*,*r*)表示副本节点 *r* 从 AWSet 中删除数据 *d*.该操作删除活跃元素集 *aset*[*r*]中所有数据域为 *d* 的元素(该集合记为 *E*,*E* 可能为空).此外,它还将 *E* 中元素加入到删除缓冲区 *rbuf*[*r*]中.
  - 查询操作 *ReadOpAWSet*(*r*)(用于实例化 *Correctness* 模块所要求的 *Read*(*r*)操作,见第 4.1 节)返回副本节点 *r* 上的 AWSet 中的所有数据.数据 *d* 在 *r* 上的 AWSet 中当且仅当元素集 *aset*[*r*]中存在数据域为 *d* 的元素.
- *Send*(*r*)表示副本节点 *r* 向其他副本节点广播消息.消息类型为 *Msg*≜[*aid*:*Aid*,*abuf*:SUBSET Element, *rbuf*:SUBSET Element],其中,*aid* 用作该消息的标识符,*abuf* 与 *rbuf* 分别为添加缓冲区 *abuf*[*r*]与删除缓冲区 *rbuf*[*r*].该动作将清空本地缓冲区 *abuf*[*r*]与 *rbuf*[*r*].
- *Deliver*(*r*)表示副本节点 *r* 向上层协议交付一条消息.要交付的消息是由底层通信网络 *ReliableCausalNetwork* 从信道 *incoming*[*r*]中选择的消息 *lmsg′*[*r*].首先,副本节点 *r* 将消息中的添加元素集 *lmsg′*[*r*].*abuf* 合并到本地的 *set*[*r*]中;然后,从其中移除在消息发送方中被删除的元素集合 *lmsg′*[*r*].*rbuf*.

```
---------------------------- MODULE OpAWSet ----------------------------
EXTENDS AWSet, FiniteSets
CONSTANTS Read(_), InitMsg
-----------------------------------------------------------------------------
VARIABLES
    aset,           \* aset[r]: the set of active elements maintained by r \in Replica
    abuf,           \* abuf[r]: the buffer for elements added by r \in Replica since the last broadcast
    rbuf,           \* rbuf[r]: the buffer for elements removed by r \in Replica since the last broadcast
(* variables for network: *)
    incoming,       \* incoming[r]: incoming channel at replica r \in Replica
    lmsg,           \* lmsg[r]: the last message delivered at r \in Replica to the upper-layer protocol
    vc,             \* vc[r][s] denotes the latest message from s \in Replica observed by r \in Replica
(* variables for correctness: *)
    doset,          \* doset[r]: the set of updates generated by replica r \in Replica
    delset,             \* uset[r]: the set of updates delivered by replica r \in Replica
    uincoming,      \* uincoming[r]: incoming channel for broadcasting/delivering updates at r \in Replica
    buset           \* buset[r]: the buffer of local updates made by r \in Replica since the last broadcast

nVars==⟨⟨incoming,lmsg,vc⟩⟩
cVars==⟨⟨doset,delset,uincoming,buset⟩⟩
vars==⟨⟨aset,abuf,rbuf,seq,nVars,cVars⟩⟩
-----------------------------------------------------------------------------
Msg==[aid: Aid, abuf: SUBSET Element, rbuf: SUBSET Element]
Network==INSTANCE ReliableCausalNetwork          \* WITH incoming←incoming, lmsg←lmsg, vc←vc

ReadOpAWSet(r)==={ele.d: ele \in aset[r]}        \* read the state of r \in Replica
Correctness==INSTANCE OpCorrectness
                \* WITH doset←doset, delset←delset, uincoming←uincoming, buset←buset
-----------------------------------------------------------------------------
TypeOK==
    ∧aset \in [Replica→SUBSET Element]
    ∧abuf \in [Replica→SUBSET Element]
    ∧rbuf \in [Replica→SUBSET Element]
    ∧IntTypeOK
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
    ∧Correctness!CTypeOK
----------------------------------------------------------------------------------------
Init==
    ∧aset=[r \in Replica ↦{·}]
    ∧abuf=[r \in Replica ↦{·}]
    ∧rbuf=[r \in Replica ↦{·}]
    ∧IntInit
    ∧Network!RCNInit
    ∧Correctness!OpCInit
----------------------------------------------------------------------------------------
Add(d,r)==                                                  \* r \in Replica adds d \in Data
    ∧LET e==[aid↦[r↦r,seq↦seq[r]],d↦d]
        IN ∧aset'=[aset EXCEPT ![r]=@ \union {e}]
           ∧abuf'=[abuf EXCEPT ![r]=@ \union {e}]
    ∧IntDo(r)
    ∧Correctness!OpCDo(r)
    ∧UNCHANGED ⟨⟨rbuf,nVars⟩⟩

Remove(d,r)==                                       \* r\in Replica removes d \in Data
    ∧LET E==\{ele \in aset[r]: ele.d=d\}            \* E may be empty
        IN ∧aset'=[aset EXCEPT ![r]=@ \E]
           ∧rbuf'=[rbuf EXCEPT ![r]=@ \cup E]
    ∧IntDo(r)
    ∧Correctness!OpCDo(r)
    ∧UNCHANGED ⟨⟨abuf,nVars⟩⟩

Do(r)==                                     \* We ignore ReadOpAWSet(r) since it does not modify states.
    \E d \in Data: Add(d,r)∨Remove(d,r)
----------------------------------------------------------------------------------------
Send(r)==                                   \* r \in Replica sends a message
    ∧buset[r] # {·}
    ∧abuf'=[abuf EXCEPT ![r]={·}]
    ∧rbuf'=[rbuf EXCEPT ![r]={·}]
    ∧Network!RCNBroadcast(r,[aid↦[r↦r,seq↦seq[r]],abuf↦abuf[r],rbuf↦rbuf[r]])
    ∧IntSend(r)
    ∧Correctness!OpCSend(r)
    ∧UNCHANGED ⟨⟨aset⟩⟩

Deliver(r)==                                \* r \in Replica delivers a message (lmsg'[r])
    ∧IntDeliver(r)
    ∧Network!RCNDeliver(r)
    ∧Correctness!OpCDeliver(r,lmsg'[r].aid)
    ∧aset'=[aset EXCEPT ![r]=(@ \cup lmsg'[r].abuf) \lmsg'[r].rbuf]
    ∧UNCHANGED ⟨⟨abuf,rbuf⟩⟩
----------------------------------------------------------------------------------------
Next==\E r \in Replica: Do(r)∨Send(r)∨Deliver(r)

Fairness==\A r \in Replica: WF_vars(Send(r))∧WF_vars(Deliver(r))

Spec==Init∧[·][Next]_vars∧Fairness
========================================================================
```

3.2 基于状态的AWSet协议

在基于状态的AWSet协议[1,2](*StateAWSet*模块)中,每个副本节点*r*维护两个集合:活跃(active)元素集*aset*[*r*]保存 *r* 观察到的被添加过且未被删除过的元素;tombstone 集 *tset*[*r*]保存 *r* 观察到的已被删除的元素.由于*StateAWSet*协议可以在任意网络中执行,因此我们使用TLA+提供的INSTANCE机制引入*BasicNetwork*网络通信模块.下面,我们分别描述该协议的初始状态*Init*与可能的动作*Do*(*r*),*Send*(*r*)与*Deliver*(*r*).

- 在初始状态,每个副本节点*r*的*aset*[*r*]与*tset*[*r*]都为空.

© 中国科学院软件研究所 http://www.jos.org.cn

- *Do*(*r*)包含两个更新操作和一个查询操作:
  - 更新操作 *Add*(*d*,*r*)表示副本节点 *r* 向 AWSet 添加数据 *d*.该协议为数据 *d* 分配唯一的标识 *aid*,将元素(*aid*,*d*)加入到活跃元素集 *aset*[*r*]中.
  - 更新操作 *Remove*(*d*,*r*)表示副本节点 *r* 从 AWSet 中删除数据 *d*.该操作删除活跃元素集 *aset*[*r*]中所有数据域为 *d* 的元素(该集合记为 *E*,*E* 可能为空).此外,它还将 *E* 中元素加入到 tombstone 集 *tset*[*r*]中.
  - 查询操作 *ReadStateAWSet*(*r*)(用于实例化 *Correctness* 模块所要求的 *Read*(*r*)操作,见第 4.1 节)返回副本节点 *r* 上的 AWSet 中的所有数据.数据 *d* 在 *r* 上的 AWSet 中当且仅当 *aset*[*r*]中存在数据域为 *d* 的元素.
- *Send*(*r*)表示副本节点 *r* 向其他节点广播消息.消息类型为

  $$Msg \triangleq [aid{:}Aid, A{:}\text{SUBSET Element}, T{:}\text{SUBSET Element}].$$

  其中,*aid* 用作该消息的标识符,*A* 和 *T* 分别为活跃元素集 *aset*[*r*]与 tombstone 集 *tset*[*r*].
- *Deliver*(*r*)表示副本节点 *r* 向上层协议交付一条消息.要交付的消息是由底层通信网络 *BasicNetwork* 从信道 *incoming*[*r*]中选择的消息 *lmsg*′[*r*].首先,副本节点 *r* 将消息中的 tombstone 集 *lmsg*′[*r*],*T* 合并到本地的 tombstone 集 *tset*[*r*]中,表示副本节点 *r* 观察到 *lmsg*′[*r*],*T* 的元素已被删除;然后,副本节点 *r* 将消息中的活跃元素集 *lmsg*′[*r*].*A* 与本地的活跃元素集 *aset*[*r*]合并,并从中移除已确认被删除的元素(即更新后的 tombstone 集 *tset*′[*r*]).

```
---------------------------- MODULE StateAWSet ----------------------------
EXTENDS AWSet, FiniteSets
CONSTANTS Read(_), InitMsg
-------------------------------------------------------------------------------
VARIABLES
    aset,           \* aset[r]: the set of active elements maintained by r \in Replica
    tset,           \* tset[r]: the set of tombstone elements maintained by r \in Replica
    (* variables for network: *)
    incoming,       \* incoming[r]: incoming channel at replica r \in Replica
    lmsg,           \* lmsg[r]: the last message delivered at r \in Replica to the upper-layer protocol
    (* variables for correctness: *)
    doset,          \* doset[r]: the set of updates generated by replica r \in Replica
    delset,         \* delset[r]: the set of updates delivered by replica r \in Replica
    uincoming       \* uincoming[r]: incoming channel for broadcasting/delivering updates at r \in Replica

nVars==⟨⟨incoming,lmsg⟩⟩
cVars==⟨⟨doset,delset,uincoming⟩⟩
vars==⟨⟨aset,tset,seq,nVars,cVars⟩⟩
-------------------------------------------------------------------------------
Msg==[aid: Aid, A: SUBSET Element, T: SUBSET Element]
Network==INSTANCE BasicNetwork                  \* WITH incoming←incoming,lmsg←lmsg

ReadStateAWSet(r)==={ele.d: ele \in aset[r]}      \* read the state of r \in Replica
Correctness==INSTANCE StateCorrectness
                    \* WITH doset←doset, delset←delset, uincoming←uincoming
-------------------------------------------------------------------------------
TypeOK==
    ∧aset \in [Replica→SUBSET Element]
    ∧tset \in [Replica→SUBSET Element]
    ∧IntTypeOK
    ∧Correctness!CTypeOK
-------------------------------------------------------------------------------
Init==
    ∧aset=[r \in Replica ↦{·}]
    ∧tset=[r \in Replica↦{·}]
```

© 中国科学院软件研究所 http://www.jos.org.cn

```
    ∧IntInit
    ∧Network!BNInit
    ∧Correctness!StateCInit
-----------------------------------------------------------------------------------------
Add(d,r)==                                  \* r\in Replica adds d \in Data
    ∧aset'=[aset EXCEPT ![r]=@ \union {[aid↦[r↦r,seq↦seq[r]],d ↦d]}]
    ∧IntDo(r)
    ∧Correctness!StateCDo(r)
    ∧UNCHANGED ⟨⟨tset,nVars⟩⟩

Remove(d,r)==                               \* r \in Replica removes d \in Data
    ∧LET E=={ele \in aset[r]: ele.d=d}      \* E may be empty
       IN ∧aset'=[aset EXCEPT ![r]=@ \E]
          ∧tset'=[tset EXCEPT ![r]=@ \cup E]
    ∧IntDo(r)
    ∧Correctness!StateCDo(r)
    ∧UNCHANGED ⟨⟨nVars⟩⟩

Do(r)==                                     \* We ignore ReadStateAWSet(r) since it does not modify states.
    \E a \in Data: Add(a,r)∨Remove(a,r)
-----------------------------------------------------------------------------------------
Send(r)==                                   \* r \in Replica sends a message
    ∧Network!BNBroadcast(r,[aid↦[r↦r,seq↦seq[r]],A↦aset[r],T↦tset[r]])
    ∧IntSend(r)
    ∧Correctness!StateCSend(r)
    ∧UNCHANGED ⟨⟨aset, tset⟩⟩

Deliver(r)==                                \* r \in Replica delivers a message (lmsg'[r])
    ∧IntDeliver(r)
    ∧Network!BNDeliver(r)
    ∧Correctness!StateCDeliver(r,lmsg'[r].aid)
    ∧tset'=[tset EXCEPT ![r]=@ \cup lmsg'[r].T]
    ∧aset'=[aset EXCEPT ![r]=(@ \cup lmsg'[r].A) \tset'[r]]
    ∧UNCHANGED ⟨⟨·⟩⟩
-----------------------------------------------------------------------------------------
Next==\E r \in Replica: Do(r)∨Send(r)∨Deliver(r)

Fairness==\A r \in Replica: WF_vars(Send(r))∧WF_vars(Deliver(r))

Spec==Init∧[·][Next]_vars∧Fairness
=========================================================
```

4 CRDT 协议的验证

4.1 实验设置

本节使用 TLC[19]模型检验工具验证 *OpAWSet* 与 *StateAWSet* 协议的正确性,即是否满足强最终一致性与最终可见性.实验所用的机器配置为:2.40 GHz GPU,6 核以及 64GB 内存(TLA+源代码与实验脚本 GitHub 仓库: https://github.com/JYwellin/CRDT-TLA).

在验证强最终一致性 SEC 的每组实验中,我们调整副本节点集合 *Replica* 与数据集 *Data* 的大小,并将它们设置为对称集[17](第 1.2 节),以提高 TLC 验证效率.另外,在 *OpAWSet*(相应地,*StateAWSet*)的 TLC 模型中,我们需要使用 *ReadOpAWSet*(*r*)(相应地,*ReadStateAWSet*(*r*))实例化 *Correctness* 模块中定义的常量 *Read*(*r*).我们使用 10 个线程进行实验,并报告如下统计数据:已遍历(以 BFS 方式遍历)的系统状态图的直径,TLC 已检验的所有状态的数量,TLC 已检验的不同状态的数量以及检验时间(格式为 hh:mm:ss).由于 AWSet 允许重复加入或者删除元素,*OpAWSet* 与 *StateAWSet* 的每个行为都是(潜在)无穷的.当 TLC 已检验的不同状态的数量超过某个阈值(设置为 1 亿)时,我们就人为地终止该次检验(2019 年 01 月 28 日构建的 TLC 版本支持该功能).

© 中国科学院软件研究所 http://www.jos.org.cn

在验证最终可见性 EV 的每组实验中,我们调整副本节点集合 *Replica* 与数据集 *Data* 的大小(TLC 暂不支持在检验活性性质时使用对称集技术[25]),并限制 *doset* 集合的大小(即限制每个副本节点允许产生的更新操作的数量).

4.2 验证结果

图 3 与图 4 分别给出了多种配置下验证 *OpAWSet* 与 *StateAWSet* 满足强最终一致性所需的时间(在给定配置下,验证时间可能受多种因素影响,包括 TLC 工具本身的实现技术,比如 BFS 队列的并发访问策略、磁盘读写等).总体而言,在相同配置下,验证 *OpAWSet* 协议比验证 *StateAWSet* 协议要消耗更多的时间.这是因为基于操作的 *OpAWSet* 协议使用的是可靠因果网络 *ReliableCausalNetwork*,它的 TLA+实现比 *StateAWSet* 协议所使用的基础网络 *BasicNetwork* 更为复杂模型.

TLC模型(副本节点数,数据集大小)	状态图直径	状态数	不同状态数	检验时间(hh:mm:ss)
(2,2)	14	216 353 094	100 000 035	0:23:39
(2,3)	13	221 536 999	100 000 045	0:28:38
(2,4)	13	236 154 405	100 000 067	0:52:51
(2,5)	13	269 575 089	100 000 048	2:53:00
(3,2)	14	288 385 868	100 000 044	0:48:36
(3,3)	13	298 212 738	100 000 048	1:10:50
(3,4)	12	322 324 714	100 000 048	2:33:14
(4,2)	13	353 773 546	100 000 050	2:23:04
(4,3)	12	374 991 027	100 000 037	3:59:51
(4,4)	12	407 327 244	100 000 072	10:45:35

Fig.3 Model checking results of verifying that *OpAWSet* satisfies SEC

图 3 “*OpAWSet* 满足 SEC”的模型检验验证结果

TLC模型(副本节点数,数据集大小)	状态图直径	状态数	不同状态数	检验时间(hh:mm:ss)
(2,2)	14	215 691 981	100 000 043	0:18:10
(2,3)	13	218 832 293	100 000 054	0:22:29
(2,4)	12	238 758 933	100 000 062	0:44:08
(2,5)	13	275 918 637	100 000 067	2:40:24
(3,2)	13	290 875 497	100 000 069	0:39:03
(3,3)	12	302 058 508	100 000 060	0:57:46
(3,4)	12	331 012 172	100 000 064	2:16:21
(4,2)	12	368 615 463	100 000 048	2:00:48
(4,3)	12	383 242 663	100 000 045	3:34:57
(4,4)	12	422 421 793	100 000 059	10:00:50

Fig.4 Model checking results of verifying that *StateAWSet* satisfies SEC

图 4 “*StateAWSet* 满足 SEC”的模型检验验证结果

由于规约 *StateAWSet* 中的动作 *Send*(*r*)在任意时刻都是可以执行的,因此即使在限制 *doset* 集合大小的情况下,*StateAWSet* 的每个行为也是(潜在)无穷的.所以,本节仅验证 *OpAWSet* 的最终可见性(在满足最终可见性方面,*StateAWSet* 与 *OpAWSet* 的行为是类似的).图 5 给出了多种配置下验证 *OpAWSet* 满足最终可见性所需的时间.

TLC模型(副本节点数,数据集大小,*doset*大小)	状态图直径	状态数	不同状态数	检验时间(hh:mm:ss)
(2,2,2)	13	210 425	22 031	0:03:02
(2,3,2)	13	843 893	62 205	0:07:56
(2,4,2)	13	2 430 313	138 267	0:17:45
(3,2,1)	13	245 323	17 287	0:07:38
(3,3,1)	13	783 517	38 764	0:16:10
(3,4,1)	13	1 906 531	72 691	0:31:22

Fig.5 Model checking results of verifying that *OpAWSet* satisfies EV

图 5 “*OpAWSet* 满足 EV”的模型检验验证结果

实验结果表明,最终可见性的验证规模显著小于强最终一致性的验证规模.这主要有两个方面的原因.

- 第一,SEC 是一种定义在单个状态上的安全性(safety)性质,针对此类性质的验证算法只需检查单个状

© 中国科学院软件研究所 http://www.jos.org.cn

态;而 EV 是定义在行为上的活性(liveness)性质,验证此类性质的算法更为复杂,复杂度也较高.

- 第二,在本工作中,我们发现 *doset* 集合的大小与副本节点数对模型的规模都有较大影响.在目前实验结果的基础上增大 *doset* 集合或增加副本节点数,都会将需要遍历的状态图的直径从 13 提高到 19 或以上,从而极大地增加状态数量,检验时间也变得无法接受.

由于最终可见性较为直观,其正确性并不直接依赖于系统的规模,因此我们认为如图 5 所示的小规模验证结果仍然有助于增强我们对 *OpAWSet* 满足最终可见性的信心.

5 相关工作

模型检验与(自动/辅助)定理证明是提高分布式协议可靠性的两种常用的形式化方法.本文使用模型检验方法验证 CRDT 协议的正确性.下面我们介绍使用定理证明方法验证 CRDT 协议正确性的相关工作.与模型检验方法相比,该类方法的优点是验证结果不受模型规模的限制,缺点则是使用门槛较高,难以复用与扩展.此外,本文不仅验证了 CRDT 协议的安全性(即满足强最终一致性),而且验证了它们的活性(即满足最终可见性).下述使用定理证明方法的工作均未涉及 CRDT 协议的活性[14,26,27].

Gomes 等人使用 Isabelle/HOL 定理证明器验证了一系列基于操作的 CRDT 协议的正确性[14],如计数器协议、ORSet(即 AWSet)集合协议以及 RGA 列表协议[3,13]等.他们构建了一个可重用的、模块化的 CRDT 协议描述与定理证明框架,该框架给出了真实环境下网络的公理语义,克服了之前工作中由于对网络作了错误假设而导致的证明失效问题.此外,框架还基于抽象收敛定理(abstract convergence theorem)给出了强最终一致性的形式化定义.在网络通信模块,Gomes 等人使用公理化方法刻画了异步不可靠因果网络.在本文,我们则实现了更多类型的通信网络.

Nagar 等人同样考虑基于操作的 CRDT 协议的自动验证问题[26].他们首先论证了强最终一致性对操作之间的可交换性的要求是与系统所能提供的一致性模型(如最终一致性、因果一致性、RedBlue 一致性等)相关的.在此基础上,他们提出了以一致性模型为参数的强最终一致性证明规则,并以集合、列表与图等 CRDT 为例展示了如何使用该规则自动验证 CRDT 协议.

Zeller 等人使用 Isabelle/HOL 定理证明器验证了一系列基于状态的 CRDT 协议的正确性[27].他们首先给出了基于状态的 CRDT 协议的参数化操作语义.在此基础上,他们给出了多种基于状态的 CRDT 协议的形式化规约,如多值寄存器(multi-value register)[2]、PN 计数器协议[2]、2P 集合协议[2]以及 ORSet(即 AWSet)集合协议等.由于 TLA+规约描述了协议的初始状态以及所有可能的动作,因此本文中的 *StateCorrectness*(以及 *Correctness*)规约也可以看作基于状态的 CRDT 协议的操作语义.除了验证所有 CRDT 协议都需要满足的强最终一致性之外,Zeller 等人还验证了多种 CRDT 协议相对于各自规约的正确性.比如,如第 3 节所述,AWSet 的规约是“某数据在 AWSet 中当且仅当存在某个添加操作添加了该数据,并且所有删除该数据的操作要么先于该添加操作,要么与该添加操作并发”.本文关注于 CRDT 协议的强最终一致性与最终可见性,并未验证它们相对于各自规约的正确性.

使用模型检验方法验证协议的正确性是提高协议可靠性的有效手段.张玉清等人[28]使用模型检测工具 SMV(symbolic model verifier)分析了著名的 Needham-Schroeder(NS)公钥协议.骆翔宇等人[29]使用时间自动机验证工具 UPPAAL 检测组合 Web 服务的安全性(如死锁)与活性等性质.Leslie Lamport[30]使用 TLA+/TLC 描述并验证了 Paxos 协议及其变体的正确性.本文则使用 TLA+/TLC 描述并验证了一系列 CRDT 协议的正确性.

6 总结与未来工作

本文采用模型检验技术验证了一系列 CRDT 协议的正确性.我们首先构建了一个可复用的、模块化的 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.我们使用 TLA+形式化规约语言实现了该框架,然后以 Add-Wins Set 复制数据类型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.我们计划扩展上述框架,使其可以描述并验证实现同一 CRDT 的不同协议之间

© 中国科学院软件研究所 http://www.jos.org.cn

的精化关系[12,31].此外,我们还将研究基于操作的 CRDT 协议与基于状态的 CRDT 协议之间的等价性[2],并使用 TLAPS 定理证明器[18,32,33]给出严格的形式化证明.

只能验证有限模型的正确性,是模型检验方法的不足之处.有文献表明,某些协议具有“分界点(cutoff bound)”性质:如果协议在某个特定大小的有限模型上是正确的,那么它在任意规模的模型上都是正确的.例如,Marić 等人[34]证明了多种以进程数为参数的共识算法具有该性质.具体而言,如果它们在 5 个或 7 个进程上是正确的,那么它们在任意多个进程上都是正确的.我们计划考察 CRDT 协议是否具有此类性质.

References:

[1] Zawirski M. Dependable eventual consistency with replicated data types [Ph.D. Thesis]. Universite Pierre et Marie Curie, 2015.

[2] Shapiro M, Preguica N, Baquero C, Zawirski M. A comprehensive study of convergent and commutative replicated data types. Research Report, RR-7506, Centre Paris-Rocquencourt, INRIA, 2011. https://hal.inria.fr/inria-00555588

[3] Attiya H, Burckhardt S, Gotsman A, Morrison A, Yang H, Zawirski M. Specification and complexity of collaborative text editing. In: Proc. of the 2016 ACM Symp. on Principles of Distributed Computing (PODC 2016). New York: Association for Computing Machinery, 2016. 259−268. https://doi.org/10.1145/2933057.2933090

[4] Gilbert S, Lynch NA. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant Web services. Sigact News, 2002,33(2):51−59. https://doi.org/10.1145/564585.564601

[5] Brewer EA. Towards robust distributed systems (abstract). In: Proc. of the 19th Annual ACM Symp. on Principles of Distributed Computing (PODC 2000). New York: ACM, 2000. 7. http://dx.doi.org/10.1145/343477.343502

[6] Decandia G, Hastorun D, Jampani MM, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon’s highly available key-value store. In: Proc. of the 21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP 2007). New York: Association for Computing Machinery, 2007. 205−220. https://doi.org/10.1145/1294261.1294281

[7] Lakshman A, Malik P. Cassandra: A decentralized structured storage system. Operating Systems Review, 2010,44(2):35−40. https://doi.org/10.1145/1773912.1773922

[8] Terry DB, Theimer MM, Petersen K, Demers AJ, Spreitzer M, Hauser C. Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP’95). New York: Association for Computing Machinery, 1995. 172−182. https://doi.org/10.1145/224056.224070

[9] Vogels W. Eventually consistent. Communications of the ACM, 2009,52(1):40−44. https://doi.org/10.1145/1435417.1435432

[10] Shapiro M, Preguica N, Baquero C, Zawirski M. Conflict-free replicated data types. In: Proc. of the 13th Int’l Conf. on Stabilization, Safety, and Security of Distributed Systems (SSS 2011). Berlin, Heidelberg: Springer-Verlag, 2011. 386−400. http://dl.acm.org/citation.cfm?id=2050613.2050642

[11] Lamport L. Proving the correctness of multiprocess programs. IEEE Trans. on Software Engineering, 1977,3(2):125−143. https://doi.org/10.1109/TSE.1977.229904

[12] Burckhardt S, Gotsman A, Yang H, Zawirski M. Replicated data types: Specification, verification, optimality. In: Proc. of the 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2014). New York: Association for Computing Machinery, 2014. 271−284. https://doi.org/10.1145/2535838.2535848

[13] Roh H, Jeon M, Kim J, Lee J. Replicated abstract data types: Building blocks for collaborative applications. Journal of Parallel and Distributed Computing, 2011,71(3):354−368. https://doi.org/10.1016/j.jpdc.2010.12.006

[14] Gomes VB, Kleppmann M, Mulligan DP, Beresford AR. Verifying strong eventual consistency in distributed systems. In: Proc. of the ACM Program. 2017. Article No.109. https://doi.org/10.1145/3133933

[15] Clarke Jr EM, Grumberg O, Long D. Model checking. In: Proc. of the NATO Advanced Study Institute on Deductive Program Design. Berlin, Heidelberg: Springer-Verlag, 1996. 305−349.

[16] Yuan D, Luo Y, Zhuang X, Rodrigues GR, Zhao X, Zhang Y, Jain PU, Stumm M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation (OSDI 2014). USENIX Association, 2014. 249−265.

[17] Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Boston: Addison-Wesley Longman Publishing Co., Inc., 2002.

© 中国科学院软件研究所 http://www.jos.org.cn

[18] Lamport L. The TLA+ hyperbook. 2019. http://lamport.azurewebsites.net/tla/hyperbook.html

[19] Yu Y, Manolios P, Lamport L. Model checking TLA+ specifications. In: Pierre L, Kropf T, eds. Correct Hardware Design and Verification Methods (CHARME 1999). Berlin, Heidelberg: Springer-Verlag, 1999. 54−66. [doi: 10.1007/3-540-48153-2_6]

[20] Lamport L. The temporal logic of actions. ACM Trans. on Programming Languages and Systems (TOPLAS), 1994,16(3):872−923. https://doi.org/10.1145/177492.177726

[21] Lamport L. Summary of TLA+. 2019. http://lamport.azurewebsites.net/tla/summary-standalone.pdf

[22] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978,21(7):558−565. https://doi.org/10.1145/359545.359563

[23] Fidge CJ. Timestamps in message-passing systems that preserve the partial ordering [Ph.D. Thesis]. Department of Computer Science, Australian National University, 1987.

[24] Burckhardt S. Principles of eventual consistency. Foundations and Trends in Programming Languages, 2014,1(1-2):1−150. https://doi.org/10.1561/2500000011

[25] Model-values and symmetry. 2019. http://tla.msr-inria.inria.fr/tlatoolbox/doc/model/model-values.html

[26] Nagar K, Jagannathan S. Automated parameterized verification of CRDTs. arXiv preprint arXiv:1905.05684, 2019.

[27] Zeller P, Bieniusa A, Poetzsch-Heffter A. Formal specification and verification of CRDTs. In: Proc. of the Int'l Conf. on Formal Techniques for Distributed Objects, Components, and Systems. Berlin, Heidelberg: Springer-Verlag, 2014. 33−48.

[28] Zhang YQ, Wang L, Xiao GZ, Wu JP. Model checking analysis of needham-schroeder public-key protocol. Ruan Jian Xue Bao/Journal of Software, 2000,11(10):1348−1352 (in Chinese with English abstract). http://www.jos.org.cn/jos/ch/reader/create_pdf.aspx?file_no=20001013&journal_id=jos

[29] Luo XY, Xuan AC, Sha ZL. Model checking Web services based on timed automata. Computer Science, 2010,37(8):139−142,197 (in Chinese with English abstract).

[30] Lamport L. Fast Paxos. Distributed Computing, 2006,19(2):79−103.

[31] Mukund M, Shenoy RG, Suresh SP. Optimized OR-sets without ordering constraints. In: Proc. of the Int'l Conf. on Distributed Computing and Networking. Berlin, Heidelberg: Springer-Verlag, 2014. 227−241.

[32] TLAPS Website. 2019. http://tla.msr-inria.inria.fr/tlaps/content/Home.html

[33] Chaudhuri K, Doligez D, Lamport L, Merz S. A TLA+ proof system. arXiv preprint arXiv:0811.1914, 2008.

[34] Marić O, Sprenger C, Basin D. Cutoff bounds for consensus algorithms. In: Proc. of the Int'l Conf. on Computer Aided Verification. Springer-Verlag, Cham, 2017. 217−237. https://doi.org/10.1007/978-3-319-63390-9_12

附中文参考文献:

[28] 张玉清,王磊,肖国镇,吴建平.Needham-Schroeder 公钥协议的模型检测分析.软件学报,2000,11(10):1348−1352. http://www.jos.org.cn/jos/ch/reader/create_pdf.aspx?file_no=20001013&journal_id=jos

[29] 骆翔宇,轩爱成,沙宗鲁.基于时间自动机的 Web 服务模型检测.计算机科学,2010,37(8):139−142,197.

纪业(1995－),男,江苏淮安人,硕士生,主要研究领域为分布数据一致性,形式化方法.

黄宇(1982－),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为分布式算法,分布式系统,网络化软件系统.

魏恒峰(1986－),男,博士,CCF 专业会员,主要研究领域为分布数据一致性,形式化方法.

吕建(1960－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为软件方法学.

© 中国科学院软件研究所 http://www.jos.org.cn