arXiv:2302.00506v2 [cs.LO] 3 Feb 2023

面向定时异步网络的分布式流运行时验证

Luis Miguel Danielsson[1,2] 和 C´esar S´anchez[1]

[1] IMDEA软件研究所，西班牙[2] 马德里理工大学（UPM），
西班牙{luismiguel.danielsson,cesar.sanchez}
@imdea.org

关键词： 去中心化监控、分布式、运行时验证、流运行时验证、时间异步网络

摘要。 我们研究分布式系统的监控问题。在这类系统中，计算机通过消息传递进行通信，并共享几乎同步的时钟。这对于监控速度足够慢（以人类尺度衡量）、能够实现高效时钟同步的网络而言，是一种现实场景；在这种场景中，时钟偏差相对于监控周期而言很小。当通过广域网、互联网或包括大规模部署在内的网络监控人类系统时，情况正是如此。

更具体地说，我们研究如何在时间异步网络下，监控其监视器以流运行时验证规约表示的去中心化系统。我们的监视器通过网络进行通信，消息可能需要任意长的时间才能到达，但不会被重复或丢失。这种通信设置在许多信息物理系统中很常见，例如智能建筑和环境智能生活。以往的去中心化监控方法仅适用于同步网络，而同步网络由于网络故障，在实践中并不易于实现。即使网络故障并不常见，也可能需要若干监控周期才能修复。

在本文中，我们提出了一种解决定时异步监控问题的方法，并说明该问题是同步情形的推广。我们研究了能够使监控在有界资源下进行、且不受轨迹长度影响的规约和网络行为条件。最后，我们报告了某项实现的实证评估结果，并从有效性和效率方面验证了理论结果。

1 引言

我们研究在定时异步计算模型下，对流运行时验证（SRV）规约进行去中心化运行时验证的问题。在去中心化监控中，规约被分解为一个监控器网络，各监视器通过交换消息进行通信。这些监视器协同求值规约，以判断分布式位置上执行的输入观测轨迹

是否满足该规约。我们针对定时异步计算模型下的去中心化监控问题提出了一种解决方案。在该模型中，进程共享一个足够同步的全局时钟，但消息可能需要任意长的时间才能到达。

运行时验证(RV) 是一种用于软件质量保证的动态技术，其方法是根据形式化规范生成监视器。随后，该监视器会检查所分析系统的一条执行轨迹。与静态验证技术不同，RV 牺牲了完备性，以提供一种易于使用的形式化方法，例如可以轻松地与测试和调试相结合。RV 必须解决的问题之一，是如何根据规约生成监视器。早期的 RV 规约语言方法基于时序逻辑 [Havelund 和 Ro¸su, 2002, Eisner 等人, 2003, Bauer 等人，2011], 正则表达式 [Sen 和 Ro¸su, 2003], 定时正则表达式 [Asarin 等人，2002], 规则 [Barringer 等人，2004], 或重写 [Ro¸su 和 Havelund，2005]。另一种监视器规约方法是流运行时验证（SRV），由 Lola 开创 [D'Angelo 等人，2005]。该方法通过声明方程来定义监视器，以描述结果输出流与观测输入流之间的依赖关系。SRV 是一种比大多数 RV 解决方案更丰富的形式体系，它超越了布尔判定（例如逻辑方法中的布尔判定），允许规约计算并输出更丰富的判定结果。例如，可以对事件进行计数并计算其他统计数据、计算鲁棒性值，或生成错误解释。有关展示 SRV 语言表达能力的示例，请参见 [D'Angelo 等人，2005, Faymonville 等人, 2016, Gorostiaga 和 S´anchez, 2018, Danielsson 和 S´anchez, 2019,Gorostiaga 等人，2020] 。

运行时验证的另一个重要方面是监视器的实际执行：如何收集信息以及如何执行监控任务。本文关注在线监控，即在观察输入轨迹的过程中逐步进行监控。在 [Bauer 和 Falcone，2012,El-Hokayem 和 Falcone, 2017a,Danielsson 和 S´anchez，2019]中，作者考虑了一种集中式规约，该规约被部署为由同步网络连接的分布式监视器网络，其中全局同步时钟既用于通信，也用于周期采样。监视器交换消息并协作完成全局监控任务。这个问题称为去中心化监控（见 [Francalanza etal.，2018]）。我们研究定时异步网络中的通信，以及输入的周期采样，即在异步网络上进行同步计算。我们的解决方案涵盖了此前针对同步计算和同步可靠网络提出的 SRV 解决方案，该网络由 [Danielsson 和 S´anchez,2019]研究。我们将本文研究的这一更一般的问题称为定时异步去中心化监控问题。我们的目标是在每个节点生成局部监视器，使其协作监控规约，在分散计算负载的同时，尽量降低网络带宽消耗以及判定计算的延迟。除了提升求值效率之外，去中心化监控还能提供容错能力，因为即使部分网络失效，系统仍可利用

未失效网络部分提供的信息对规约进行部分求值。出于同样的考虑，如果协作监视器网络的某一部分发生拥塞——也就是说，由于某种原因运行速度变慢——其余部分仍能保持正常吞吐量。以*if-then-else* 规约为例，计算*then* 和*else* 两部分的值都需要耗时较长的计算。考虑一个去中心化部署，其中三个监视器以树形连接：叶监视器计算*then* 和*else* 两部分，而根监视器则使用*if*部分的布尔输入流来计算规约。假设条件有 90% 的时间为真，因此大多数时候使用*then* 的值，而*else* 的值会被丢弃。现在再假设根监视器与计算*else* 部分的叶监视器之间的网络链路较慢，那么在 90% 的情况下，规约根节点的吞吐量仍不受影响，结果也无需等待*else* 部分的长时间计算及其链路延迟即可得出。我们计划在未来的工作中利用去中心化系统的优势，积极引入容错能力。

我们的解决方案。 本文针对 Lola [D'Angelo 等， 2005] 规约，提出了一种适用于任意网络拓扑和本地监控器部署方式的去中心化监视问题解决方案。

我们研究时间异步网络 [Cristian 和 Fetzer， 1999]，其中各节点共享一个全局时钟（通过限制网络同步性延迟和硬件时钟漂移来实现），但监控消息可能需要任意长的时间才能送达。时间异步网络 [Cristian 和 Fetzer， 1999]"……允许实现实际所需的服务，例如时钟同步、成员管理、共识、选举和原子广播"。同步网络是其特殊情形，此外消息还会在一个已知的有界时间内送达。我们利用全局时钟可用这一事实，采用一种按轮次进行监控的计算模型：每一轮先读取输入并处理收到的消息，接着更新本地监视器的内部状态，最后生成输出消息。在许多场景中，这种同步执行模型都很符合实际，例如智能建筑或智慧城市——其中可以使用时间网络协议同步时钟，精度足以支持每轮周期为数十秒的执行。本文还假设系统可靠：节点不会崩溃，消息不会丢失或重复。在我们的解决方案中，规约的不同部分（建模为流）——包括输入读数——会作为本地监视器部署到不同的网络节点上。必要时，本地监视器会与其他监视器通信，以解析分配给它们的流，并尽量减少通信开销。直观地说，数据将从传感器监视器中读取，随后每一层中间监视器都会计算子表达式，并将部分结果发送给负责超表达式的远程监视器，最终计算出根表达式的值流。这种设置的一个退化情形是集中式解决方案：映射了观测数据的节点将传感值发送到固定的中心节点，由该节点负责计算整个规约。

我们考虑的 SRV 语言是 Lola [D'Angelo 等，2005，S´anchez， 2018]。我们将确定哪些规约及网络行为条件能够使监控在有界资源下进行，且不受轨迹长度影响。

动机示例。

示例 *1.* 我们以一栋智能建筑作为贯穿全文的示例。这栋建筑的各个房间都配有传感器，并设有一个中心节点。我们的目标是在存在火灾风险时触发警报。以下规约通过检测某个房间内温度和 CO_2 的急剧升高，来捕捉这种风险。我们将用于判断测量变量是否上升"足够多"的计算，放在进行传感器读数的节点上。这样，中心节点只需计算哪些节点同时出现温度和 CO_2 警报。为简洁明了起见，我们省略了CO_2 计算（因为它与温度计算完全相同）。CO_2 值可用于评估"Building"监视器中的火灾风险。

```
@Room1 {
    输入 数值 t_1  评估 # 初始值_1 是一个具有合理边界的恒定值
# 定义 数值 tlow  =  1.6 * 初始值 _1 定义 数值 thi =  2.0 * 初始值
_1定义 数值 t_spike_q_1  =如果 t_1  <=low 则 0 否则如果 t
_1  >  hi 则 1 否则 ( t_1 - low )/( hi - low ) } @Room2 { 输入 数
值 t_2  评估 定义 数值 t_low  =  1.6 * 初始值_2定义 数值 t_hi
=  2.0 * 初始值_2定义 数值 t_spike_q_2  =如果 t_2  <=  t_low 则
0 否则如果 t_2  >  t_hi 则 1 否则 ( t_2 - t_low )/( t_hi -t_low ) }
@Building { 定义 布尔值 fire_risk_q_1  =  t_spike_q_1  >  0.5
定义 布尔值 fire_risk_q_2  =  t_spike_q_2  >  0.5 }
```

相关工作。 在综述 [Francalanza 等人 2018]中，使用术语去中心化监控 来区分分布式监控：在后者中，进程不共享全局时钟。在分布式监控中，假设网络是完全异步的；而去中心化监控通常假设

网络完全同步，所有样本和通信都同步进行。本文探讨两者之间的情形：网络节点共享一个足够同步的全局时钟（如同步分布式系统），但通信可能耗时任意长（如异步分布式系统）。此外，在 [Francalanza 等人 2018] 的研究中，他们还提出了策略检查等其他概念，并将其称为去中心化监控；这些概念与本文介绍的监控并不相同，因为它们只关注可用于异步计算和异步网络的全局安全属性。

在 [Ganguly 等人的研究中， 2021]他们也研究了由协作监控器组成的定时异步网络，但使用 SMT 求解器来简化线性时态逻辑公式。

分布式流处理已得到广泛研究。在 [Quoc 等人， 2017]的工作中，他们采用了复杂事件处理中的流概念，其中事件可以是结构化数据类型，计算也可能很复杂，即每个事件都需要执行多项操作，例如通过滑动窗口操作对到达的事件进行聚合计算。 [Quoc 等人， 2017] 的目标是将隐私与近似技术相结合，从而实现零知识隐私以及低延迟、高效的分析。 [Carbone 等人， 2015] 介绍了 Apache Flink，它采用流数据流处理来处理连续流和批处理。运行时验证领域也研究了分布式和去中心化监控。Sen 等人 [Sen 等人， 2004]提出了 PT-DTL，这是一种用于监控分布式系统的 LTL 逻辑变体，但他们考虑的是完整的异步分布式系统，并且仅限于布尔判定。 [Francalanza 等人，2018]的工作使用切片，在具有全局时钟的分布式消息传递系统监控中应对节点崩溃和消息错误。Bauer 等人 [Bauer 等人， 2013]提出了一种一阶时序逻辑和一种轨迹长度无关的生成自动机；而在 [Bauer和Falcone， 2012] 的工作中，他们展示了一种使用公式重写来监控同步系统中 LTL_3 的去中心化方案。LTL_3是 LTL 的一种三值变体，其中格中有一个中心值，用来表示某个表达式目前的值未知，因此需要继续处理输入轨迹，才能确定其真值。 [El-Hokayem 和 Falcone，2017a，El-Hokayem 和 Falcone，2017b] 使用执行历史编码（EHE）对此进行了改进。EHE 是一种数据结构，用于存储不同监视器部分求值后的表达式及其部分信息，使去中心化监控器能够推断监控自动机所处的状态。在 [El-Hokayem 和 Falcone， 2020] 的工作中，作者扩展了 EHE，加入分布式和多线程支持，并通过构造保证数据结构的确定性。随后，他们使用 EHE 分析去中心化规范的兼容性和可监控性。然而，判定和值仍然是布尔值，网络假设则是同步性。 [Jaber 等人， 2020] 将全局编排（作为一种主控式协议）综合（包括控制流、同步、通知、确认以及嵌入计算），并应用于分布式系统。此外，他们还提供了到 Promela 的转换，从而可以使用 LTL 规约验证实现

。他们展示的一些方案包括生产者-消费者的变体或两阶段提交，并将其应用于构建微服务，例如购物系统。这项工作着重于综合监视器的流程，但观察结果和判定仍然是布尔值。在 [Kazemlou 和 Bonakdarpour， 2018]的工作中，LTL 监视器构成的同步网络协同工作，为被测系统得出判定，同时也可能遭遇崩溃。在此场景中，文中提出了一种基于 SMT 的算法，用于综合 LTL 监视器的自动机；即使崩溃的监视器永远无法恢复，该算法仍能实现容错并保证可靠性。尽管这项工作考虑了故障（这超出了本文的讨论范围），但其假设通信是同步的。所有这些方法都只考虑布尔判定。相比之下，SRV 能够从任意数据域生成判定。

此前所有 SRV 相关工作，从 Lola [D'Angelo 等， 2005],Lola2.0 [Faymonville 等， 2016], Copilot [Pike 等，2010 年，Pike 等，2013 年，Perez 等，2020] 以及针对定时事件流的扩展（如 TeSSLa [Convent 等， 2018],RTLola [Faymonville 等， 2019] 或 Striver [Gorostiaga 和 S´anchez， 2018] 均假设采用集中式监控设置。 [Gorostiaga 等 2020] 探讨了基于时间（软实时）和基于事件的计算模型之间的关系及其对 SRV 的影响，但仍是在集中式设置下。 [Basin 等 2015] 的工作展示了如何监控分布式系统的度量时序逻辑规约（包括故障和消息重排序）：节点以树状方式通信，由根节点发出最终判定。 [Danielsson 和 S´anchez 2019] 提出了一种同步监控 SRV 规范的解决方案，但假设网络是同步的。我们将 [Danielsson 和 S´anchez 2019] 的工作扩展到定时异步网络。

贡献与结构。本文的主要贡献是提出一种解决方案（见第3节），用于解决定时异步分布式流运行时验证问题。我们给出了算法的正确性证明，并表明我们的解决方案能够以零开销涵盖同步分布式问题。第二项贡献（见第3.6节）描述了在何种规约和网络行为条件下，监控可以在有界资源下进行，且不受轨迹长度影响。在信息物理系统中，资源受限（内存、带宽乃至计算时间都有限），因此必须及时作出正确响应以适应不断变化的环境；对资源加以限制至关重要。如果信息物理系统不受轨迹长度影响，即使资源受到物理限制，也能无限期运行。第三项贡献（详见第4节）是原型实现和实证评估。第四项贡献（见第5节）是改进后的算法，它允许节点仅在收到请求时才传送流值，从而节省带宽。第2节介绍预备知识，第6节总结全文。

2 预备知识。流运行时验证

下面简要回顾流运行时验证（SRV）。更详细的介绍请参见 [D'Angelo 等, 2005] 以及教程 [S´anchez, 2018]。由 Lola [D'Angelo 等, 2005] 开创的 SRV，其核心思想是通过一组方程以声明式方式描述监视器，这些方程刻画了输出值流与输入值流之间的依赖关系。本文聚焦于在线监控。监视器由规约生成，并在运行时计算输出流的值序列；每当观测到输入流中的一个值后，便尽可能快地计算出相应的输出值。输入值通常取自某个传感器，或从日志文件中读取。

Lola 规约声明了输出流与输入流之间的关系，其中包括未来和过去的时间依赖关系。Lola 语言将时间依赖关系与每个步骤要执行的具体操作清晰地分离开来，从而将逻辑的监控算法推广到对更丰富的值（如数值、字符串或更丰富的数据类型）的计算。

2.1 Lola 语法。

Lola 规约通过声明输出流与事件输入流之间的关系来定义。流表达式是使用一组（经解释的）构造符号构建的项。之所以称符号为经解释的，是因为每个构造符号不仅用于构建项，还关联一个求值函数；该函数根据参数值生成一个具有相应返回类型的值。给定一组 Z 带类型的流变量，流表达式集合由以下内容组成：(1) 来自 Z 的变量；(2) 偏移量 $v[k, d]$，其中 v 是类型为 D 的流变量，k 是自然数，且 d 是来自 D 的值；以及 (3) 项 $f(t_1, \ldots, t_n)$，其中使用理论中的构造符号 f 构建先前定义的项。流变量表示规约中的值序列（流）。偏移表达式 $v[-1,$假$]$ 的预期含义是轨迹中前一个位置上流 v 的值（如果不存在这样的位置，即处于开头，则为 假）。我们用 项$_D(Z)$ 表示由来自 Z 的变量构建的、类型为 D 的流表达式集合（如果上下文清楚，则省略 Z）。给定一个项 t，子(t) 表示 t 的子项集合。

定义 1（规约）。 *Lola* 规约 φ（I, O）由一组$I = \{r_1, \ldots, r_m\}$ 输入流变量、一组 $O = \{s_1, \ldots, s_n\}$ 输出流变量以及一组定义方程组成，$s_i = e_i$（$r_1, \ldots, r_m, s_1, \ldots, s_n$）每个输出变量 $s_i \in O$对应一个定义方程。项 e_i 属于 $Term_D$（$I \cup O$），其中 D 是 s_i的类型。

我们用 $r, r_i, \ldots$ 表示输入流变量；用 $s, s_i, \ldots$ 表示输出流变量；用 u, v 表示任意输入或输出流变量。给定 $\varphi(I, O)$，我们用 $appears(u)$ 表示使用 u 的输出流集合，即

为 $\{s_i \mid u[-k,d] \in sub(e_i)$ 或 $u \in sub(e_i)\}$。此外，$ground(t)$ 表示表达式 t 是否为基表达式（不包含变量或偏移量），因此能否根据常量和函数符号的解释求值为一个值。

示例 *2.* 性质“对输入流 y 中之前的值求和，但如果*reset* 流为真，则重置计数”可以表示如下，其中流变量 根 使用累加器 累加器 和输入 重置 来计算所需的和。流累加器 使用关键字 定义 进行定义，以强调它是一个中间流。

```
输入布尔型 重置输入 数值y定义 整数 累加器 = y  + 根 [-
1 | 0]输出 整数 根  =如果 重置 则 0  否则 累加器
```

2.2 Lola 语义。

现在我们介绍 Lola 的形式语义，它保证每个输入流都有唯一正确的输出流。通过证明算法能够生成所需的输出，这一语义使我们能够证明算法是正确的。在运行时，输入流变量会逐步与值输入流相关联。

给定输入流 σ_I（每个输入流变量对应一个序列）以及输出候选 σ_O（每个输出流对应一个序列），形式语义描述这一对 (σ_I, σ_O) 是否符合规约，记作 $(\sigma_I, \sigma_O) \vDash \varphi$。我们用 σ_r 表示与输入变量 r 相对应的 σ_I 中的流，用 $\sigma_r(k)$ 表示流 σ 在位置 k处的值。要使 $(\sigma_I, \sigma_O) \vDash \varphi$ 成立，所有流都必须是长度相同的序列。

规约 φ 的一个赋值 是一个对 σ：(σ_I, σ_O)，其中每个输入流变量和输出流变量在 φ中都对应一个流（其值的类型适当），且所有这些流的长度相同。给定一个项 t，其求值 $[\![t]\!]_\sigma$ 是一个由 t 类型的值组成的序列，定义如下：

– 如果 t 是一个流变量 u，那么 $[\![u]\!]_\sigma(j) = \sigma_u(j)$。– 如果 $f = f(t_1, \ldots, t_k)$，那么 $[\![f(t_1, \ldots, t_k)]\!]_\sigma(j) = f([\![t_1]\!]_\sigma(j) \ldots, [\![t_k]\!]_\sigma(j))$。– 最后，如果 $t = v[i, c]$ 是一个偏移量，那么当 $j + i$ 是轨迹中的有效点时取 $[\![v\langle j + i\rangle]\!]$，否则取默认值 c。形式化地：如果 $0 \leq j + i$，则 $[\![v[i, c]]\!]_\sigma(j) = [\![v]\!]_\sigma(j + i)$；否则为 c。

当对每个输出变量 s_i，$[\![s_i]\!]_{(\sigma_I,\sigma_O)} = [\![e_i]\!]_{(\sigma_I,\sigma_O)}$. 时，赋值 (σ_I, σ_O) 满足 Lola 规约 φ。在这种情况下，我们称 σ 是 φ 的求值模型，并记作 $(\sigma_I, \sigma_O) \vDash \varphi$。

规约 φ 的目的是根据给定的输入描述唯一的输出；若 φ 没有周期，则可保证这一点，具体定义如下。规约 $\varphi(I \cup O)$ 的依赖图 D_φ 是一个加权多重图 (V, E)，其顶点为流变量 $V = I \cup O$，且 E 包含一条有向

加权边 $u \xrightarrow{w} v$，当且仅当 $v[w,d]$ 是 u的定义方程中的一个子项。若规约 φ 的 D_φ 不含零权重环，则该规约是良构的，这保证了在当前位置，没有流依赖于自身。

考虑示例 2。其依赖图如下：

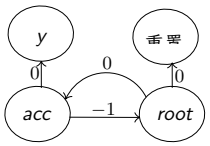


图 1. 示例 2 的依赖图

给定流变量 u 和位置 $i \geq 0$，瞬时流变量（或简称瞬时变量）定义为一对 $u\langle i\rangle$，其中 $u\langle i\rangle$ 是一个与 u 类型相同的新变量。请注意，每个时刻 i 都有一个不同的瞬时变量 $u\langle i\rangle$。求值图 *EG* 是依赖图在所有时刻上的展开。给定 φ（I，O）和轨迹长度 M（对于无限轨迹则为 $M=\omega$），求值图$G_{\varphi,\ M}$ 的顶点集为$u \in I \cup O$ 和 $0 \leq k < M$对应的瞬时变量集合 $\{u\langle k\rangle\}$；若依赖图包含边 $u \xrightarrow{j} v$ 和 $k+j=k'$，则求值图中也有边 $u\langle k\rangle \to v\langle k'\rangle$

$M=5$ 对应的求值图如图 2 所示。

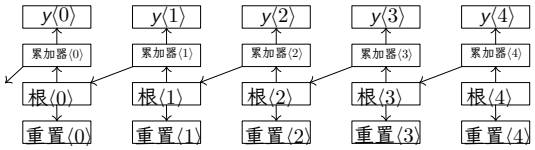


图 2。 示例 2 的求值图

考虑示例 2， $acc\langle 4\rangle$ 在所有具有$M \geq 4$的求值图中都指向 $root\langle 3\rangle$。我们将 $e_s\langle k\rangle$ 表示为这样一个项（其叶节点为瞬时变量）：该项由 e_s 在 k处得到，即将偏移量项替换为经过适当移位修正的相应瞬时变量。再次考虑示例 2。在瞬时 4 处， acc 的瞬时流表达式 e_{acc}为 $acc\langle 4\rangle = y\langle 4\rangle + root\langle 3\rangle$。

依赖图的节点构成一个由极大强连通分量（MSCC）组成的有向无环图。另请注意，依赖图中没有正权环的规约称为可高效监测规约 [D'Angelo 等，2005]。可高效监测规约的求值图中不存在周期，因此我们可以对求值图使用归纳法进行推理，稍后将会这样做。请注意，这些规约可以包含正权边

（对应于未来依赖项），前提是它们不构成正权环。正如 [S´anchez, 2018] 所示，这些规约可以通过中央监视器在线（增量式）求值，且只需有限内存。

示例 *3.* 以下代码片段分别展示了一个难以高效监控的规约、一个可高效监控的规约，以及一个可极高效监控的规约。第一个片段中的规约难以高效监控，因为流 b 依赖于自身的未来值；在求值图（EG）中，所有瞬时变量都将无界地依赖于未来的下一个时间瞬间。因此，在无限轨迹中，所有瞬时流 b 都将永远无法确定。

```
input int a
output int b = b[1|0]
```

下一个规约可高效监控，因为它对未来的引用都是有界的：每个瞬时变量 b 仅依赖于后续两个位置的值，因此，对于每个瞬时变量 $b\langle k\rangle$ ，它将在$k+2$处得到确定。

```
input int a
output int b = a[2|0] + b[-1|0]
```

这是一个可极高效监控的规约，因为它没有对未来的引用，所有偏移量均为负数或零。

```
输入 inta输出 intb =a +b[-1|0]
```

2.3 去中心化同步在线监控

文献 [Danielsson 和 S´anchez, 2019]介绍了一种用于在异步网络中监控 Lola 规约的在线去中心化算法。其核心思想是利用由多个协作节点构成的网络来监控 Lola 规约：先根据规约的语法树对其进行切分，再将每个子表达式（包括输入）映射到一个节点。这要求监视器通过消息共享各子表达式的部分结果。在每个时间瞬间，算法都会读取输入、更新内部表达式，并与相应节点通信结果，从而利用这些部分结果完成规约的计算。因此，对于给定的良构 Lola 规范，文献 [Danielsson 和 S´anchez, 2019]中提出的去中心化在线算法会逐步计算每个输出瞬时变量的值，并假定网络是同步的，且消息不会丢失或重复。本文介绍的算法将文献 [Danielsson 和 S´anchez, 2019]中的算法扩展到了更一般的定时异步网络。

3 定时异步网络的去中心化流运行时验证

本节将介绍我们针对定时异步网络中的去中心化 SRV 问题所提出的解决方案。下面给出的算

法将根据输入读数的值计算出输出瞬时变量的唯一取值。我们在定理 1 中证明该算法的终止性，并在定理2 中证明其正确性，从而验证操作语义与指称语义等价。我们要求 Lola 规约是良构的，并且需要建立流与负责计算这些流的网络节点之间的映射。每个网络节点都将部署一个本地监视器，负责计算规约中的部分流。对于流变量 s，我们用 $\mu(s)$ 表示其本地监视器负责解析流 s 的值的网络节点。为完成全局监控任务，本地监视器会在需要时交换包含部分结果的消息。然而，由于输入所在位置不同以及通信造成的延迟，我们的去中心化算法可能会在不同于集中式版本的时间瞬间计算出某些输出值。我们将在第3.6节从理论上研究这一影响，并在第4节进行实证研究。集中式监视器对应于 [D'Angelo 等人，2005,S´anchez，2018] 中的操作语义；这等价于一种网络映射，即将所有输入流和输出流都分配给单个节点，从而避免通信。

3.1 问题描述

网络。我们假设网络由一组节点 N构成，其中每个节点都可以通过发送消息与其他所有网络节点通信。我们假设在定时异步网络上进行可靠单播通信（不丢失消息，也不重复消息），因此给定消息可能需要任意长的时间才能到达。由于网络节点共享一个全局时钟，计算按周期进行。每个周期中，网络中的所有节点都会并行执行以下操作，直至完成：(1) 读取输入消息，(2) 执行会终止的本地计算，(3) 生成输出消息。我们使用以下类型的消息：$(s\langle k\rangle,\ c,\ n_s,\ n_d)$，其中 $s\langle k\rangle$ 是一个瞬时变量，c 是与 s类型相同的值，n_s 是源节点，n_d 是目标节点。我们使用以下缩写：消息。源 $= n_s$、消息。目的地 $= n_d$、消息。流 $= s\langle k\rangle$和 消息。值 $= c$。这些消息用于告知读取或计算所得的实际值。

流分配与通信策略 给定一个规约$\varphi(I,\ O)$和一个包含节点 N的网络，流分配 是一个映射 μ：$I \cup O \to N$，它为每个流变量分配一个网络节点。输入流变量 r 对应的节点 $\mu(r)$，是网络中每个时钟周期感知到 r 的位置。在运行时，每当时刻 k 到来时，都会读取 $r\langle k\rangle$ 的新输入值。另一方面，输出流变量 s 对应的节点 $\mu(s)$，是其本地监视器负责确定 s的值的位置。

每当 $v\langle k\rangle$ 的值确定时，瞬时值 $v\langle k\rangle$ 会自动传达给所有可能感兴趣的节点。设 v 和 u 为两个流变量，使得 v 出现在 u 的方程中，并且 $n_v = \mu(v)$ 和 $n_u = \mu(u)$。那么，n_v 会通过以下方式将其确定的每个值 $v\langle k\rangle = c$ 通知 n_u：

发送一条消息（$v\langle k\rangle$， c， n_v， n_u）。至此，我们终于可以定义去中心化 SRV 问题了。

定义 2. 去中心化 *SRV* 问题 $\langle \varphi, N, \mu \rangle$ 由一项规范 φ、一个包含节点 N 的网络，以及为每个流变量指定的流分配 μ 构成。

我们用 DSRV 表示去中心化 SRV 问题。求解一个 DSRV实例，意味着借助由承载本地监视器的互连节点组成的网络，根据输入流的瞬时变量的值，计算与输出流对应的瞬时变量的值。

3.2 通信模型

现在，我们详细描述所假设的定时异步计算模型。网络中插入的每条消息都会按照以下条件到达其目的地：

– 始终延后：每条在 t 插入的消息 m 都会在 t' 到达，并满足 $t' > t$；– 任意延迟：任何消息到达所需的时间都没有先验上界。– 每对节点之间先进先出：设 m_1 和 m_2 是具有相同源节点和目的地的两条消息， m_1.源节点 $= m_2$.源节点 且 m_1.目的地 $= m_2$.目的地。设 m_1在 t_1 插入并在 t'_1 到达，且 m_2 在 t_2 插入并在t'_2到达。如果 $t_1 < t_1$，那么 $t'_1 \leq t'_2$。也就是说， m_1 不可能晚于 m_2到达。

同步模型是定时异步网络的一种特殊情况，其中插入网络的所有消息在每对网络节点之间始终需要相同的时间。在这种情况下，延迟始终是一个常量。形式上，为了分析算法的行为，我们将消息延迟建模为一族函数$arr_{u\to v}$ （每对节点（u, v）对应一个函数），该函数在任意时刻都会给出：在 t 从 u 发送的消息将在瞬时 t' 到达 v。

3.3 定时异步网络的 DSRV：监视器和算法

我们的解决方案由一组本地监视器组成，每个网络节点n对应一个。本地监视器 $\langle Q_n, U_n, R_n \rangle$ 负责 n 维护一个输入队列 Q_n 和两个存储区：

– **已解析** 存储区 R_n，其中 n 存储已解析的瞬时变量（$v\langle k\rangle, c$）。– **未解析** 存储区 U_n，其中 n 存储未解析方程 $v\langle k\rangle = e$，其中e 不是一个值，而是一个包含其他瞬时变量的表达式。

当 n 从远程节点收到消息时，相关信息会被添加到 R_n中，因此，之后本地对该信息的请求可以在本地立即得到满足。在时刻 k的计算周期开始时，节点 n读取分配给本地传感器的输入流值，并实例化

以便 k 负责的所有输出流变量都能得到处理。之后，使用 n迄今获取的知识对所得方程进行简化，这些知识存储在 R_n中。最后，生成新消息并将其插入相应邻居的队列中。

算法 1 节点 n 的本地监视器，使用 $\langle Q_n, U_n, R_n \rangle$

```
1: 过程 MONITOR
2:     Q_n ← ∅; U_n ← ∅; R_n ← ∅; k ← NOW()
3:     当未结束 执行 STEP(k)
4:     M ← k; FINALIZE(M)
5: 过程 STEP(k)
6:     R_old ← MS_n.R_n
7:     PROCESSMESSAGES(MS_n)
8:     R_n.add({r⟨k⟩ ↦ read(r, k) | r ∈ ins_n})
9:     U_n.add({s⟨k⟩ ↦ e_s⟨k⟩ | s ∈ outs_n})
10:    EVALUATE(MS_n)
11:    R_new ← MS.R_n \ R_old
12:    SENDRESPONSES(MS_n)
13:    PRUNE(MS_n)
14: 过程 EVALUATE(MS_n)
15:    done ← false
16:    当 not done 时执行
17:        done ← true
18:        对所有 s⟨k⟩ ↦ e ∈ U_n 执行
19:            e' ← SUBST(e, R_n)
20:            如果 ground(e') 则 done ← false
21:                U_n.del(s⟨k⟩ ↦ e); R_n.add(s⟨k⟩ ↦ e')
22:            否则 U_n.del(s⟨k⟩ ↦ e); U_n.add(s⟨k⟩ ↦ e')
23: 过程 PROCESSMESSAGES(MS_n)
24:    对于所有 msg = ⟨resp s⟨k⟩, c⟩ ← Q_n.pop() 执行
25:        R_n.add(s⟨k⟩ ↦ c)
26: 过程 SENDRESPONSES(MS_n, R_new)
27:    对所有 u⟨k⟩ ↦ c ∈ R_new 执行
28:        send（响应 s⟨k⟩, c, n, n_r）
29: 过程 PRUNE（MS_n, R_new）
30:    对所有 u⟨j⟩ ↦ c使得 now ≥ MTR（u⟨j⟩） 执行
31:        R_n.del（u⟨j_i⟩ ↦ c_i） }                    ▷ 移除
```

具体而言，每个节点 n 都会执行算法 1 所示的过程 MONITOR，该过程在每个时钟周期都会调用 STEP。过程 FINALIZE 用于在轨迹结束时将待处理值解析为其默认值。请注意，如果监控轨迹始终不终止，就不会调用此过程（Monitor 将持续观察并生成输出）。过程 STEP 执行以下步骤:

1. **处理消息**：第 7 行调用第 23–25 行中的过程 ProcessMessages，该过程负责处理到达的响应，并将其添加到 R_n。2. **读取输入并实例化输出：**第 8 行读取当前时间 k 的新输入，第 9 行则实例化 n 负责的每个输出流的方程。3. **求值：**第 10 行调用第 $14-22$ 行中的过程 Evaluate，该过程对未求解方程求值。4. **发送响应：**第 12 行调用第 26–28 行中的过程 SendResponses，为所有新近求解的变量发送消息。5. **剪枝：**第 29–31 行从集合 R 中剪除不再需要的信息。参见第3.6节。

3.4 形式化正确性

现在，我们通过证明计算出的输出与指称语义中的输出相同，并且每个输出最终都会被计算出来，来说明我们的方案是正确的。

定理 1. 以下结论对每个瞬时变量 $u\langle k\rangle$ 都成立：

(1) $u\langle k\rangle$ 的值最终会得到确定。(2) </style当且仅当在某个瞬时，$(u\langle k\rangle, c) \in R$，$u\langle k\rangle$ 的值才是 c。(3) 最终会向所有关注的网络节点发送关于 $u\langle k\rangle$ 的响应消息(所有负责处理流 v （其中出现了 $u \in$ ）的节点(v))。

证明。 证明通过对求值图进行归纳来展开，并在归纳步骤中同时证明 (1)–(3)，因为它们相互依赖于前面的归纳步骤。设 M 为计算的长度（可以是 ω），σ_I 为长度为 M 的输入。注意，上述 (1) 至 (3) 都是关于瞬时变量 $u\langle k\rangle$ 的陈述，而这些变量是求值图 $G_{\varphi,M}$ 的节点。我们对 $G_{\varphi,M}$ 进行归纳（根据假设，由于 D_φ 良构，因此它是无环的）。

– **基本情况**：基本情况是求值图中没有出边的顶点，这些顶点要么是

- 对应于从本地传感器读取的输入的瞬时变量，要么是
- 其瞬时方程不包含其他瞬时变量的已定义变量；当方程是常量，或者时间瞬间使得产生的偏移量超出轨迹范围时，就属于这种情况；此时使用默认值。

对于输入，命题 (1) 显然成立，因为在时刻 k，$u\langle k\rangle$ 会在节点 $\mu(u)$ 处被读取。对于不含变量的输出方程，或其变量的偏移量实例化后变为负数或大于 M 的输出方程，其叶节点的值要么立即确定，要么在计算偏移量时于 M 确定。此时，计算所得的值会插入 R，因此 (2) 在 $\mu(u)$ 处也成立。注意，对于其他节点，(2) 也成立，因为

当且仅当 $(u\langle k\rangle, c) \in R_n$ 时，响应消息中包含 $u\langle k\rangle = c$，其中 $\mu(u) = n$。随后，响应消息会恰好在其得到解析的位置插入，因此 (1) 蕴含 (3)。

– **归纳情形**：考虑求值图中的任意 $u\langle k\rangle$ $G_{\varphi,\ M}$并令 $u_1\langle k_1\rangle$，……，$u_l\langle k_l\rangle$ 为其所依赖的瞬时变量。这些是$G_{\varphi,\ M}$ 中低于 $u\langle k\rangle$ 的节点，因此归纳假设适用，且它们满足 (1)-(3)。设 $n = \mu(u)$。在时刻 k，$u\langle k\rangle$ 被实例化并插入 U_n。瞬时变量的值也会计算并发送（依据 (1) 和 (3)）。在最晚到达时刻，$u\langle k\rangle$ 的方程不再包含变量，并被求值为一个值，因此 (1) 成立，且 (2) 在 n处成立。此时会发送响应消息（因此 (1) 对 $u\langle k\rangle$成立），所以 (1) 也成立。

至此，证明完成。

定理 1 意味着，对于协作监视器网络，在每个时刻，每个已定义流的每个值最终都会得到解析。因此，给定输入流 σ_I，该算法通过（2）计算出唯一的输出流σ_i，每个 s_i 对应一个。元素 $\sigma_i(k)$ 是本地监视器为 $\mu(s_i)$ 中的 $s_i\langle k\rangle$ 解析出的值。下一个定理表明，算法 1 计算出的值是正确的（依据 Lola 的指称语义）；定理 1 则表明所有值最终都会被计算出来。

我们将 $out(\sigma_I)$ 用作从输入流到协作监视器所计算的输出流的函数。我们用 $[s]$ 表示 $out(\sigma_I)$ 中与流变量 s 对应的值流。现在我们证明，计算出的值序列与该规约的语义相对应。

定理2。设 φ 是一个规约，$S = \langle \varphi, \mathcal{T}, \mu\rangle$ 是一个去中心化 *SRV* 问题，σ_I 是一个值输入流。那么 $(\sigma_I, out(\sigma_I)) \vDash \varphi$。

证明。令 σ_O 为唯一的求值模型，使得 $(\sigma_I, \sigma_O) \vDash \varphi$（我们用$\sigma_O(s)$表示流变量 s 的输出流，用 $\sigma_O(s)(k)$表示其在第 k个位置上的值）。我们需要证明，对于每个 s 和 k，都有 $[s](k) = \sigma_O(s)(k)$。我们再次对求值图 $G_{\varphi,M}$进行归纳。

– **基本情况：**对于输入，其值显然成立。另一个基本情况对应于输出变量 s 在不依赖其他变量的时刻的取值（因为偏移量的所有出现位置——若有——都超出了轨迹范围）。该值由网络节点 $\mu(s)$求值，并满足 s的方程 e_s，且不依赖任何其他流的值。因此，它满足 $[s](k) = [\![e_s[k]]\!] = \sigma_O(s)(k)$，正如所需。

– **归纳情形：**令 s 为任意流变量，k 为 0 和 $M-1$ 之间的任意瞬时，并假设求值图中所有可由 $s\langle k\rangle$ 到达的瞬时变量 $u\langle k'\rangle$ 都满足归纳假设。令 n 为负责计算 s 的节点。根据定理 1，所有值最终都会由 n 接收，并出现在 R_n 中；根据归纳假设，这些值与指称语义中的值相同，即 $[u](k') = \sigma_O(u)(k')$。对 $s\langle k\rangle$ 的求值对应于计算 $[\![e_s]\!]$，其使用了语义

（根据第2节）中表达式的语义。对表达式 e_s 做简单的结构归纳可知，求值的结果，即赋给 $s\langle k\rangle$ 的值，是 $[\![e_s]\!]_\sigma(k) = \sigma_O(s)(k)$，正如所需。

至此，证明完成。

3.5 简化器

算法 1 中表达式的求值假设表达式中的所有瞬时变量 e 都已知（即 e 是基项），因此数据理论中的解释函数可以对 e 求值。有时，只需知道表达式涉及的部分瞬时变量（而非全部），就可以对表达式进行部分求值（甚至完全确定其值）。简化器是一个函数 f : 项$_D$ →项$_D$，满足：(1) $f(t)$ 中的变量是 t 中变量的子集；(2) 对 t 中变量进行任意值替换，所得值都与对 $f(t)$ 进行替换所得的值相同。例如，以下是一些典型的简化器:

$$
\begin{array}{ll}
if\ \ true\ \ then\ \ t_1\ \ else\ \ t_2 & \mapsto t_1 \\
\text{若为假}\ \ then\ \ t_1\ \ else\ \ t_2 & \mapsto t_2 \\
\text{真}\ \vee x & \mapsto \text{真} \\
\text{真}\ \wedge x & \mapsto x \\
0 \cdot x & \mapsto 0
\end{array}
$$

实际上，化简器会显著影响性能，具体体现在瞬时变量被解析的时刻，以及在去中心化监控中，延迟和交换的消息数量上。从本质上说，化简器是一个从项到项的函数，对于原始项中变量的每一种可能赋值，它都不会改变最终得到的值。显然，对于通过实例化定义方程得到的每个项 t，以及每个化简器 f $[\![t]\!]_{\sigma_I,\ \sigma_O} = [\![f(t)]\!]_{(\sigma_I,\sigma_O)}$，这是成立的，因为 t 中的变量值与 $f(t)$ 中的变量值都由相同的值（取自 σ_I 和 σ_O）填入。

考虑在算法 1 第 19 行用于化简表达式的任意化简器$simp$ 。设 U_n 是节点 n 的未解析存储区，并设 $u\langle k\rangle$ 是满足 $\mu(u) = n$ 的瞬时变量。根据算法 1， U_n 将存储的项序列 $(u\langle k\rangle,\ t_0), (u\langle k\rangle,\ t_1) \ldots (u\langle k\rangle,\ t_k)$ 满足以下条件：每个 t_i 都会应用化简器。因此，使用化简器计算出的值与不使用化简器时相同。还容易证明，使用化简器的算法获取每个瞬时变量值的时间，不会晚于不使用化简器的算法。这是因为在最坏情况下，只有当某个瞬时变量所依赖的所有瞬时变量都已知时，该变量才会被解析；而所有响应消息都会在变量解析时发送。

3.6 理论资源利用率

本节旨在定义一些条件，在这些条件下，本地监视器只需使用有界内存即可计算每个输出值。首先需要考虑的是

规约必须能够高效地进行去中心化监控 [Danielsson 和 S´anchez， 2019],这一要求基本上指出， G_φ 中的每个强连通分量都必须映射到同一个网络节点。也就是说，如果 u 传递地出现在 v 的声明中，并且 v 出现在 u 的声明中（带有某些偏移量），那么 $\mu(u) = \mu(v)$。

为了保证节点 n 的本地监视器中的给定存储有界，我们必须为解析分配给 n 的流的瞬时变量所需的时间提供一个上界。我们使用解析时间（ TTR ）来指代给定瞬时变量 $u\langle k\rangle$ 完成解析所需的时间量。该时间是从变量在时间 k 实例化，到其完成解析、离开 U_n 并存储在 R_n中的时刻之间所经过的时间瞬间数。这发生在算法 1 的第 21 行。

解析时间的一般方程现在，我们引入递归方程的一般定义，用于描述瞬时变量 $s\langle k\rangle$ 何时完成解析。为了界定网络节点 n处 Monitor 使用的内存，我们需要界定存储 U_n 和 R_n:

– 的上界：远程需要的已解析值会立即发送到远程节点，因此 R_n 仅包含本地未来所需的已解析值 n。由于可高效监测规约只包含（未来）有界路径，因此规约中使用的未来引用 b存在最大值。该上界限制了已解析值 $v\langle k\rangle$ 可以保留在 R_n中的时长，因为至多经过 b 个步骤后，需要 $v\langle k\rangle$ 的值且存储在 R_n 中的瞬时变量 $u\langle k'\rangle$ 就会被实例化（注意， $k' - k \leq b$）。

在 $t = max(k + b,\ k + TTR(u\langle k\rangle))$ 之后，不再需要 $u\langle k\rangle$。在 t 时，可以从 R_n 中移除 $u\langle k\rangle$ 的值。这保证在每个节点 n 上， R_n 的大小始终有一个常量上界。**– 的上界：** 节点 n 上用于存储 U_n 的内存大小，与负责解析 s （即 $n = \mu(s)$）的已实例化但尚未解析的瞬时变量数量成正比。因此，要界定 U_n ，我们需要计算解析分配给 n 的流的瞬时变量所需时间的上界。

下面给出的通用方程取决于网络中的消息延迟。之后，我们将针对以下网络行为情形实例化这些通用方程:

– 同步网络；– 消息延迟在整个轨迹上有上界的定时异步网络（我们称之为**永恒**情形）；– 消息延迟在给定时间范围内有上界的定时异步网络（我们称之为**暂时**情形）。

请注意，算法的正确性（定理2）证明了输出流 σ_O 仅取决于输入流 σ_I ，但并未说明

每个元素 σ_O 得到确定的时间界限，也未说明消息的延迟。

本节研究消息延迟如何影响瞬时变量的解析时刻，而这又会影响计算节点的内存使用量。我们用 $d(t,\ a,\ b)$ 表示在时间 t 从 a 发送到 b 的消息所需的到达时间。换句话说，到达时间$_{a\to b}(t) = t + d(t,\ a,\ b)$。回顾一下，我们假设消息具有因果性，且队列采用先进先出，这一点已在 3.2 节中说明。因果性意味着消息在发送后才会到达（也就是说，对于任意 n、m 和 t，到达时间$_{a\to b}(t) > t$），而先进先出意味着对于任意 n 和 m，若 $t < t'$，则 到达时间$_{a\to b}(t) \leq$ 到达时间$_{a\to b}(t')$。

现在，我们定义给定瞬时变量 $s\langle t\rangle$ 的解析时刻，记为 $MTR(s\langle t\rangle)$，它表示 Monitor 在负责计算 s 的网络节点 $\mu(n)$ 上能够保证解析 $s\langle t\rangle$ 的时刻。我们的定义考虑两个组成部分：解析 $s\langle t\rangle$ 可能依赖的所有本地瞬时变量所需的时间，以及远程瞬时变量的解析，这也涉及消息延迟。我们使用远程解析时刻这一概念，记为$MTR_{rem}(s\langle t\rangle)$，表示 $s\langle t\rangle$ 直接需要的所有远程值均已到达的时刻（若所有值都在t之前到达，则该时刻为 t）。

$$
\begin{aligned}
&MTR(s\langle t\rangle) \stackrel{\text{def}}{=} \max(\\
&\quad MTR_{rem}(s\langle t\rangle), \{MTR(r\langle t+w\rangle) \mid s \xrightarrow[loc]{w} r\})\\
&MTR_{rem}(s\langle t\rangle) \stackrel{\text{def}}{=} \max(t,\\
&\quad \{arr_{r\to s}(MTR(r\langle t+w\rangle)) \mid s \xrightarrow[rem]{w} r \text{ and } t+w \geq 0\})
\end{aligned}
$$

请注意，对于每个良构规范，这一定义都是明确的，因为求值图是无环的，而关于 $s\langle t\rangle$ 的方程只依赖于求值图中位于较低位置的那些变量；该图是无环的。

示例 *4*。考虑网络节点 1 处的流 i 和 acc，以及在网络节点 2 处计算的重置和根。然后，我们可以替换为

这些方程，以获得 MTR(根$\langle 1\rangle$)。

$$
\begin{aligned}
&MTR(root\langle 1\rangle) = \max(MTR(reset\langle 1\rangle), MTR_{rem}(acc\langle 1\rangle)) = \\
&= \max(1, \max(1, arr_{acc\to root}(MTR(acc\langle 1\rangle)))) = \\
&= \max(1, arr_{acc\to root}(\max(1, \\
&\qquad MTR(i\langle 1\rangle), MTR_{rem}(root\langle 0\rangle)))) = \\
&= \max(1, arr_{acc\to root}(\max(1, MTR_{rem}(root\langle 0\rangle)))) = \\
&= \max(1, arr_{acc\to root}(\max(1, \\
&\qquad \max(MTR(reset\langle 0\rangle) MTR_{rem}(acc\langle 0\rangle))))) = \\
&= \max(1, arr_{acc\to root}(\max(1, \max(0, \\
&\qquad \max(0, arr_{acc\to root}(MTR(acc\langle 0\rangle)))))))) = \\
&= \max(1, arr_{acc\to root}(\max(1, arr_{acc\to root}(\max(0, \\
&\qquad MTR(i\langle 0\rangle), MTR_{rem}(root\langle -1\rangle))))))) = \\
&= \max(1, arr_{acc\to root}(\max(1, arr_{acc\to root}(0))))
\end{aligned}
$$

当瞬时变量 根$\langle 1\rangle$ 收到瞬时变量 累加器$\langle 1\rangle$的响应时，便可保证其已解析——也就是 max(1，$arr_{acc\to root}$(...)) 这一部分。只有收到累加器$\langle 0\rangle$ 的响应后，才能生成该响应；这对应最内层部分：...max(1，$arr_{acc\to root}$(0))。请注意，无需将 $MTR_{rem}(root\langle -1\rangle)$ 纳入计算，因为它会立即解析为默认值。同样，输入也会立即解析，在获取 MTR 的值时不会引入任何延迟。

请注意，对于 $MTR_{rem}(s\langle t\rangle)$，只考虑满足 $t + w \geq 0$ 的远程瞬时变量，否则在实例化 $s\langle t\rangle$ 时将使用默认值。在 $MTR(s\langle t\rangle)$ 的方程中，我们假设基本情况为 $MTR(s\langle t\rangle) = 0$，当 $t < 0$ 时成立，因为此时同样会改用偏移表达式中的默认值，而该值可立即确定。很容易看出，第一个方程等价于：

$$
MTR(s\langle t\rangle) \stackrel{\text{def}}{=} \max(\{MTR_{rem}(r\langle t + w\rangle) \mid s \xrightarrow[loc]{w}{}^{*} r\})
$$

现在我们可以证明，这些定义确实刻画了 $s\langle t\rangle$ 解析完成的时间。

定理 3. 设 φ 为一个规约，μ 为一个网络放置，σ_I 为输入轨迹，arr 为网络行为。每个 $s\langle t\rangle$ 都在 $MTR(s\langle t\rangle)$ 时或更早解析完成。

证明。 该证明通过对由 φ 和 σ_I的长度所诱导的求值图 $G_{\varphi,M}$ 进行归纳法得出。

– **基本情况**：不依赖任何其他瞬时变量的输入和瞬时变量 $s\langle t\rangle$ 。这些是事件图中没有任何

出边的节点。由于 $s\langle t\rangle$ 在 t实例化，因此该值会在 t 处得到精确解析，可以通过读取传感器或实例化为默认值来实现。此外，$MTR(s\langle t\rangle) = MTR_{rem}(s\langle t\rangle) = t$。

– **一般情况**。令 $s\langle t\rangle$ 为任意瞬时变量，并根据归纳假设，假定该定理对事件图中低于 $s\langle t\rangle$ 的所有瞬时变量均成立。在时刻 $MTR_{rem}(s\langle t\rangle)$，来自其所依赖的远程节点的所有瞬时变量 $r\langle t+w\rangle\, s\langle t\rangle$ 均已到达，因为根据归纳假设，$r\langle t+w\rangle$ 将在$MTR(r\langle t\rangle)$ 时得到解析。同样，$s\langle t\rangle$ 所依赖的所有本地元素在依赖图中也都位于较低位置，因此归纳假设同样适用。因此，在时刻

$$\max(MTR_{rem}(s\langle t\rangle), \{MTR(r\langle t+w\rangle) \mid s \xrightarrow[loc]{w} r\})$$

或在此之前，$s\langle t\rangle$ 所依赖的所有元素都将已知，且 $s\langle t\rangle$ 将得到解析。

至此，证明完毕。

以下推论由这样一个事实得出：瞬时变量得到解析后发生的任何事情（无论是 σ_I 中后续的值，还是网络行为）都不会影响计算出的值。因此，计算 $s\langle t\rangle$ 的值及时间不取决于$MTR(s\langle t\rangle)$ 之后的未来。

推论 1. 对于所有 $s\langle t\rangle$，都存在一个 t'，使得 $s\langle t\rangle$ 仅取决于 σ_I 以及截至 t'的 *arr*。

瞬时变量的 MTR 取决于网络节点之间的网络$arr_{\rightarrow}$延迟，这些节点协作计算该瞬时变量。因此，如果这些延迟可能任意长，我们就无法保证 MTR 的界限，也就无法限制内存使用量。由此，在一般的时间异步网络中，监控与轨迹长度无关这一点无法成立。

接下来，我们研究网络行为的不同条件（涉及链路之间的延迟）如何影响 MTR，并在这些条件下确立内存界限，从而恢复与轨迹长度无关的监控。

同步时间实例化 我们首先假设采用同步计算模型。这是定时异步模型的一种特殊情况，其中两个监视器之间的所有消息延迟在整个轨迹中都完全相同。我们使用谓词 $dist_{r\ s}$ 表示每条消息从 $\mu(r)$ 到 $\mu(s)$ 所需的延迟，该延迟不受消息发送时间瞬间的影响。因此，$arr_{r\rightarrow s}(t) = t + dist_{r\ s}$。这一延迟使我们能够将同步网络的$MTR_{rem}$ 简化如下：

$$MTR^{sync}_{rem}(s\langle t\rangle) \stackrel{\text{def}}{=} \max(t, M(s\langle t\rangle))$$

其中 $M(s\langle t\rangle) =$

$$\{MTR^{sync}(r\langle t+w\rangle) + dist_{r_s} \mid s \xrightarrow[rem]{w} r, t+w \geq 0\})$$

回顾一下，解析时间 是指变量被实例化的时刻与其得到解析的时刻之间的时间间隔（即$TTR(s\langle t\rangle) = MTR(s\langle t\rangle) - t$）。在同步情形下，我们得到：

$$\begin{aligned}
&TTR^{sync}(s\langle t\rangle) = MTR^{sync}(s\langle t\rangle) - t = \\
&= \{MTR^{sync}_{rem}(r\langle t+w\rangle) \mid s \xrightarrow[loc]{w} r\} - t = \\
&= \max(t, M(s\langle t\rangle) - t = \\
&= \max(0, \{TTR^{sync}(r\langle t+w\rangle) + dist_{r_s} \mid s \xrightarrow[rem]{w} r\})
\end{aligned}$$

请注意，决定结果的值是最慢远程依赖的$TTR^{sync}(s\langle t\rangle)$，其中既包括解析时间，也包括消息穿过网络所需的时间。此外，我们可以很容易地通过对依赖图进行归纳证明，对于每个流变量 s ，都存在一个常量 k ，使得$TTR^{sync}(s\langle t\rangle) \leq k$，也就是说， s 总是在少于 k个时刻内得到解析。因此，所有可去中心化高效监控的规约都可以在每个本地监视器中以常量空间进行监控；换言之，对可去中心化高效监控规约的同步去中心化监控与轨迹长度无关。

具有永恒有界延迟的定时异步情形 现在我们假设每条消息的延迟时间都有一个全局上界，我们称之为**永恒**有界延迟。形式上，这一假设表示存在一个 d ，使得对于每一对流 r， s 以及每个时间 t，$arr_{r\to s}(t) \leq t + d$。将上界值 d 代入MTR的方程，我们得到MTR的一个常量上界：

$$MTR^{\mathbf{g}}_{rem}(s\langle t\rangle) \leq \max(t, M(s\langle t\rangle))$$

其中

$$M(s\langle t\rangle) = MTR^{\mathbf{g}}(r\langle t+w\rangle) + d \mid s \xrightarrow[rem]{w} r, t+w \geq 0\}$$

请注意，在某些情况下， $s\langle t\rangle$ 可以在$MTR^{\mathbf{g}}(s\langle t\rangle)$之前得到解决，因为 d 是一个上界。在这种情况下，我们也可以界定每个节点为执行监控过程所需存储的内存，但大多数时候所需的内存会更少。图3中给出了一个**永久**上界的示例。

具有暂时有界延迟的定时异步情形 现在，我们仔细考察这些方程，以便更准确地界定解析给定瞬时变量 $s\langle t\rangle$ 所需的时间；我们不假设计算历史中的所有消息都具有上界，只考虑可能影响 $s\langle t\rangle$ 的必要消息。界定 $MTR(s\langle t\rangle)$ 的主要思路，是考察用于计算 $s\langle t\rangle$ 的相关消息发送所处的时间间隔。

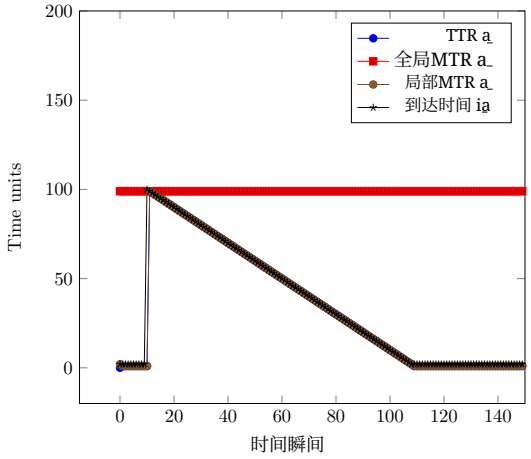


图3. TTR，**永恒** 和**临时** 峰值延迟界限

我们首先定义一个辅助概念。若流变量 r 以延迟 w 直接远程影响 s，我们就称之为直接远程影响，并记作 $s \xrightarrow[drem]{w} r$，当且仅当存在一个路径 $s \xrightarrow[loc]{w_1} s_1 \xrightarrow[loc]{w_2} s_2 \ldots \xrightarrow[loc]{w_k} s_k \xrightarrow[rem]{w_{k+1}} r$ 满足：

– 没有两个节点 s_i 和 s_j 重复（如果 $i \neq j$，则 $s_i \neq s_j$），并且

– $w = w_1 + \ldots + w_k + w_{k+1}$。

注意，$s \xrightarrow[drem]{w} r$ 意味着 $s\langle t\rangle$ 可能会受到远程变量$r\langle t+w\rangle$的影响。我们将 $s\langle t\rangle$ 的关注窗口定义为：

$$win(s\langle t\rangle) = [\min S, \max S] \text{ where } S \text{ is defined as}$$
$$S = \{t, MTR_{rem}(r\langle t+w\rangle) \mid s \xrightarrow[drem]{w} r \text{ and } t + w > 0\}$$

注意，S 是发送会影响$s\langle t\rangle$ 的远程瞬时变量的时刻集合。

示例*5.* 考察示例 2中的规约，并查看图2中的求值图，我们可以发现，任意时刻的任意瞬时变量的关注窗口都包含其在求值图中的依赖项的关注窗口。因此，其关注窗口将包含最早依赖项得到解析的最小时间，以及最后一个依赖项得到解析的最大时间。在此示例中，$root\langle 1\rangle$ 的时间窗口将包含 $acc\langle 1\rangle$、$root\langle 0\rangle$ 和 $acc\langle 0\rangle$ 的时间窗口，以及响应消息从源传输到目的地所需的时间。请注意，输入不会影响MTR。

因此，$win(s\langle t\rangle)$ 包含与 $s\langle t\rangle$ 相关的远程信息发送时刻。该时间窗口最晚总是在 $MTR(s\langle t\rangle)$ 处结束。我们

接着，我们将为计算 $s\langle t\rangle$ 而发送给 s 的最坏消息定义如下：

$$d_{worst}(s\langle t\rangle) = \max\{t' - t \mid \\ t' = arr_{r\to s}(t) \text{ for } s \xrightarrow[drem]{w} r \text{ and } t \in win(s\langle t\rangle)\}.$$

请注意， d_{worst} 仍然是为计算 $s\langle t\rangle$ 而发送的消息的过近似，但在这种情况下，界限考虑了所有这些消息，并且只关注一个有界时间区间。由于所有影响 $s\langle t\rangle$ 的值都在$win(s\langle t\rangle)$内发送，因此我们可以按如下方式界定$MTR(s\langle t\rangle)$：

$$MTR^{\mathbf{temp}}_{rem}(s\langle t\rangle) \leq \max(t, M(s\langle t\rangle))$$

其中 $M(s\langle t\rangle) =$

$$\{MTR(r\langle t + w\rangle) + d_{worst}(win(s\langle t\rangle)) \mid s \xrightarrow[drem]{w} r\}.$$

我们终于得到了期望的结果：一个有限时间窗口，其中包含计算瞬时变量所需的相关消息的发送和接收。这意味着，只有有限数量的网络延迟会影响任何瞬时变量 $s\langle t\rangle$ 的求值。由于我们总能在该时间窗口内找到最大延迟，因此可以为解析任何瞬时变量所需的时间设定上界，并且能够知道这些瞬时变量在 U_n 和 R_n 中存储了多长时间。进而，这使我们能够确定某些瞬时变量何时不再需要，以及何时可以将其剪枝并释放占用的内存。

图3展示了网络峰值行为以及 TTR 如何相应地进行调整。我们可以看到**临时**界限与**永久**界限之间的差异，其中**永久**界限在整个执行过程中都很高且保持恒定，而当网络延迟较小时，**临时**界限会降低。

裁剪已解析存储 R_n。我们现在终于可以剪枝 R_n，因为我们已经知道每个瞬时变量何时会被解析。

推论2。 $s\langle t\rangle$ 中 U_n 的每个未解析瞬时变量最迟都会在 $MTR(s\langle t\rangle)$时解析。

一旦 $MTR(s\langle t\rangle$ 达到（或在此之前）， $s\langle t\rangle$ 的值就会在 $\mu(s)$ 的本地监视器中变得已知，并将发送给需要它的远程监视器。此后，可以从 U_n 中剪枝掉 $s\langle t\rangle$。通过这一机制，我们可以确保每个瞬时变量在内存（U_n 或 R_n）中的驻留时间都有界。推论2表明，当网络行为存在某种界限时，无论是同步界限、**永久**界限还是**暂时**界限，都可以使用有界资源监控定时异步网络中可高效监控的分散式规约。该内存界限仅随规约大小以及网络节点之间的延迟线性增长。这一结果也可以从相反的角度来理解：给定固定的可用内存量，我们可以计算出网络中能够确保正确执行监控的最大延迟。

4 实证评估

我们已将解决方案实现为原型工具 tadLola，该工具使用 Go 编程语言编写（可访问 http://github.com/imdea-software/ dLola）。现在我们介绍：

– (1) 使用从四个不同的真实公共数据集中提取的真实数据，实证研究 tadLola 在不同场景下的能力。– (2) 网络行为（以延迟衡量）对内存以及输出解析时间的影响。

我们的实验设置旨在实证确定异步网络的行为，以及故障如何影响流的解析时间。

4.1 数据集与网络故障。

我们在此次实证评估中使用了四个不同的数据集，分别是：Smart- Politech [Pajuelo-Holguera 等人， 2020]，Tomsk Heating [Zorin 和 Stukach，2020]，Orange4Home [Cumin 等人， 2017] 以及 Context [Kaupp 等人， 2021]。除 Context 与工业4.0有关外，其余数据集均与智能建筑相关。前两个数据集分别关注建筑气候控制，并使用不同房间或建筑中的传感器。Orange4Home 数据集侧重于活动识别，住户可以在公寓内自由活动，目标是推断其进行的活动。最后，Context 是一个智能工厂数据集，其中会出现一种新型故障，即情境故障：当没有特定传感器或采集到的数据能够直接指示错误时，需要根据情境知识推断错误及其根本原因。

	恒定			peak		
	最小值	中位值	最大值	最小值	中位值	max
smartP	4	4	4	4	4	102
smartPShort	4	4	4	4	4	102
托木斯克	2	2	2	2	2	100
contextAct	2	2	2	2	2	100
orange4H	2	2	2	2	2	100

(a) 恒定和峰值网络行为

	正常			normalPeak		
	最小值	中位值	最大值	最小值	中位值	max
smartP	3	10	18	4	10	105
smartPShort	6	11	18	7	12	105
托木斯克	1	5	10	1	5	100
contextAct	1	5	9	1	5	100
orange4H	1	5	10	1	5	100

(b) 正常和 normalPeak 网络行为

图 4. 不同网络行为下 tadLola 的 TTR分析

及其根本原因。针对每个数据集，我们都创建了一份合成规范，以展示工具的功能。我们还注入了合成延迟，以模拟网络拥塞和故障。

– $constant$ 行为建模为每对监视器之间的全局恒定延迟，因此每条消息都恰好需要固定的时间才能到达目的地网络节点。这对应于同步监控中可观察到的网络行为。– $constantPeak$ 由恒定延迟和一次较高的网络延迟构成，用于模拟网络故障及恢复：所有消息都会延迟，直到问题解决，之后网络开始逐步恢复，直至再次恢复正常运行。– $Normal$ 行为遵循以平均延迟为给定值的延迟正态分布。– $normalPeak$ 与 constantPeak 类似，但以正常行为作为基线。

请注意，所有这些行为都具有**永续** 和**临时** 有界性，因为对于每种行为，我们都能找到整个轨迹的上界，以及每个瞬时变量在感兴趣时间窗口内的上界。

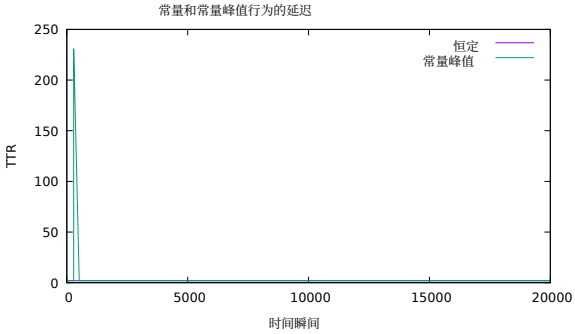
图4展示了在这些网络行为下解析流所需 TTR 的最小值、中位值和最大值。

图5展示了这些行为下观测到的延迟示例。被观测的系统按周期采样，从而获得每个测量变量的输入轨迹。因此，已知轨迹长度和采样周期后，我们就能确定整个实验期间受到监控的系统运行时长。例如，长度为 $200k$ 、采样周期为 30 秒的轨迹，对应于对系统进行为期 ≈ 2.31 个月的监控。在部分实验中，数据集中可用的真实数据轨迹不够长，因此我们通过重复样本来延长这些轨迹，直至达到所需的轨迹长度。此外，部分轨迹需要进行插值，以便为所有事件采用公共时钟节拍，因为其中一些轨迹基于事件，而非对变量进行周期性采样。我们在需要时进行了插值。

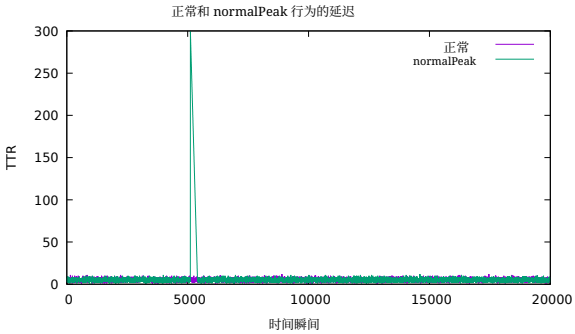
4.2 假设

对于本文的实证评估，我们拟对以下假设进行求值：

– (H1) 当网络呈现同步行为时，我们的时间异步算法表现不逊于 [Danielsson 和 S´anchez 提出的同步算法。– (H2) 同步 SRV 可以通过软件层模拟对时间异步网络的监控，该软件层营造出同步性的假象，但代价是延迟和内存使用量大幅增加。



(a) 恒定和恒定峰值



(b) 正常和 NormalPeak

图 5. 网络行为示例

– (H3) 第3.6节中的理论结果适用于实验的执行。– (H4) 根监视器的本地内存有上限，因此监控器的轨迹长度与轨迹长度无关，这与我们的理论结果预测一致。– (H5) 我们的算法在监视器数量和网络使用量方面具有良好的扩展性。我们预计，内存使用量会随“网络使用量”线性增长，但增加本地监控器数量时将保持恒定。这里所说的本地监控器，是指在同一网络节点上计算的非空流集合。– (H6) 我们可以利用冗余规约和冗余拓扑（借助化简器），通过避开缓慢或故障链路造成的延迟来减少 *TTR* 。

4.3 实证结果

为验证假设 (H1)，我们设计了以下实验：

– SmartPolitechDistr：我们通过分析大学建筑不同房间内的温度、CO_2 和空气湿度水平来检测火灾隐患。我们使用定量鲁棒规范。– tomskHeating：我们检查供暖系统是否按预期运行（根据数据提取）。同样，这是一种定量鲁棒规范。– orange4Home：我们通过分析住户在公寓中进行的活动来检测火灾隐患。– contextAct：我们通过分析智能公寓不同房间内的温度、CO_2 和空气湿度水平来检测火灾隐患。这也是一种定量规范。

图4展示了在不同网络行为下分析的各个数据集的规约根节点延迟指标。这从实证上证明，正如我们的理论证明所预期的那样，TADSRV 涵盖了 DSRV，且不会造成额外的性能损失。因此，(H1) 成立。所有这些不同的网络行为都表明，TADSRV 比 DSRV 更通用，这也符合我们的预期。

为验证 (H2)，我们设计了一项实验，规约要求同时求得输入的最大值和总和。我们将其置于图6所示的拓扑中。

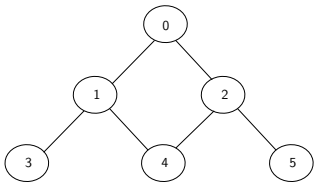


图6。用于 (H2) 实验的监控器拓扑

我们在已有的 normalPeak 轨迹中找出最大延迟，并将该时长用作同步场景下每对监视器之间的全局延迟。我们测量了两种设置：模拟同步性，以及执行定时异步算法。结果如图7所示。图中显示，我们可以用 DSRV 模拟 TADSRV，但内存使用量会大幅增加（比最坏时刻高出 $+200\%$），同时还会产生 $\text{最坏延迟} * \text{拓扑深度}$的延迟；在本例中，这相当于 558 个时刻。这使延迟增加到定时异步算法的约 30 倍。因此，(H2) 也成立。结果清楚地表明，在时间异步场景中，实际上不可行使用 DSRV（即使配有模拟同步的层）；而本工作的贡献正适用于此类场景，且性能要好得多。

此外，我们可以看到，通过经验获得的 TTR 均小于或等于根据第3.6节所述方程预先计算的估计界限。因此，(H3) 成立。

为了验证 (H4)——研究轨迹长度方面的可扩展性——我们使用 smartPolitechDistr 数据集，并以 normalPeak 行为运行了一条包含 $200k$ 个时刻的轨迹。在下方所示的摘录中，我们计算布尔流和定量流，以查找温度升高。

```
定义布尔值 temp_up eval  =temp  >  1.1 * tempini 且 temp
  <=  1.6 * tempini 定义数值 temp_up_q eval  =如果 temp  <=  1.
1* tempini 则为 0，否则如果 temp  >  1.6* tempini 则为 1，否
则为 ( temp - 1.1* tempini )/(1.6* tempini -1.1* tempini ) 定义布
尔值 temp_spike eval  =temp  >  1.6 * tempini 定义数值 temp_
spike_q eval  =如果 temp  <=  1.6* tempini 则为 0，否则如果
temp  >  2  * tempini 则为 1，否则为 ( temp - 1.6* tempini )/
(2* tempini -1.6* tempini )
```

图 8 显示，本实验中根监视器使用的内存保持有界。内存中的尖峰对应于节点之间网络链路上的较高延迟。这迫使监视器将记录保留在内存中，直到所需的消息到达，使监视器能够解析流并剪枝其内存。这一结果表明，采用可高效监视的去中心化规约的算法可以不受轨迹长度影响地运行，从而验证了假设（H4）。

图 9 显示，单个监视器的内存使用量不取决于网络中其他监视器的数量，而取决于其规约在网络中传输时的最大深度。在这项实验中，部署在网络中的规约深度保持恒定（5），同时改变二叉树拓扑中的监视器数量（保留其中一个分支的深度）。直观来说，影响内存使用量的变量并非监视器的数量，而是网络节点的数量以及它们之间影响监控性能的链路数量。这是因为链路越多，网络故障（建模为延迟）影响运行过程的概率就越高。这些结果证明假设（H5）成立。

冗余与延迟。在本小节中，我们将更仔细地考察假设（H6），因此构建拓扑和规约，以尽量降低瞬时变量的*TTR*。我们希望通过使用化简器，尽量减小网络延迟对计算瞬时变量所需消息的影响。因此，我们打算利用从读取输入的节点到计算规约根的节点之间，沿网络中最快路径传输的消息。在拥塞程度最低的部署中，中间结果生成得更快，消息也会沿输入与规约根之间权重最低的路径（按累计延迟计）传输，从而使瞬时变量的 TTR 达到最小。由于使用化简器后不再需要来自较慢监视器的中间结果，因此能够实现这一改进；

而且引擎无需等待根监视器得出最终结果。针对 smartPolitech 中的数据，我们构建了以下规约片段，其中流 C3_fire_risk_q 和 C3_fire_risk_q_red 彼此冗余，并将它们部署在不同的监视器上，使其受到不同延迟的影响。我们为整个网络采用正常延迟，但在与监视器 3 相连的监视器之间引入故障，使延迟出现峰值。这会使经过监视器 2 的路径更快。我们可以从图10中观察到，获取规约根的值所需的延迟达到了可能的最佳水平。由于我们使用或运算来利用化简器，在最佳情况下，判定（结果为真）会有所提升；但在最差情况下，判定为假时，冗余方案并不能提升速度，因为引擎需要等待所有值到齐后才能计算或运算。

@0 { define bool C3_alarm eval =(C3_fire_risk 或 C3_fire_risk_red) and (C3_fire_risk_q > 0.5 或 C3_fire_risk_q_red > 0.5) } @2 { define bool C3_fire_risk_red eval =AND(C3_temp_spike , C3_co2_spike , C3_humid_down) define num C3_fire_risk_q_red eval = AVG(C3_temp_spike_q , C3_co2_spike_q , C3_humid_down_q) } @3 { define bool C3_fire_risk eval =AND(C3_temp_spike , C3_co2_spike , C3_humid_down) define num C3_fire_risk_q eval =AVG(C3_temp_spike_q , C3_co2_spike_q , C3_humid_down_q) }

图10展示了使用冗余拓扑下的冗余规约与完全不使用冗余之间的差异。尽管在网络中利用冗余路径的一般性研究超出了本文的范围，但此案例研究说明冗余部署如何改进去中心化监控。

5 惰性算法

现在，我们介绍算法 2 的一个变体，其中某些流只有在其值被请求时才会发送。当这些值很少需要时，这种做法很有益。我们将这些称为惰性数据流。

为了介绍修改后的算法，我们需要引入一种新的消息类型：请求消息。我们还将包含瞬时变量值的消息称为响应消息。

– **响应**消息：**(resp** $s\langle k\rangle, c, n_s, n_d$**)**，其中 $s\langle k\rangle$ 是一个瞬时变量，c 是与 $s\langle k\rangle$具有相同数据类型的常量，n_s 是消息的源节点，n_d 是消息的目标节点。– **请求**消息：**(req** $s\langle k\rangle, n_s, n_d$**)**，其中 $s\langle k\rangle$ 是一个瞬时变量，n_s是消息的源节点，n_d 是消息的目标节点。

同样，如果 消息 $=$**(请求** $s\langle k\rangle$，n_s，n_d**)**，那么 消息。源 $=n_s$、消息。目的地 $=n_d$、消息。类型 $=$**请求**、消息。流 $=s\langle k\rangle$。类似地，对于响应消息，形式相同，唯一的区别是增加了消息。值 $=c$。

每个流变量 v 都可以采用以下两种通信策略 之一，用来表示瞬时值 $v\langle k\rangle$ 是否会自动通信给所有可能感兴趣的节点，还是仅在收到请求时才提供其值。设 v 和 u 是两个流变量，其中 v 出现在 u 的方程中，并且设 $n_v = \mu(v)$ 和 $n_u = \mu(u)$。

– **主动通信**：节点 n_v 通过发送消息（**响应** $v\langle k\rangle$，c，n_v，n_u），将其确定的每个值 $v\langle k\rangle = c$告知 n_u。这是我们在本文前文中采用的方式。– **惰性通信**：节点 n_u 通过发送消息（**请求** $v\langle k\rangle$，n_u，n_v），请求 n_v 提供 $v\langle k\rangle$ 的值（以防 n_u需要它为某个 k'确定 $u\langle k'\rangle$）。当 n_u 收到此消息并将 $v\langle k\rangle$ 确定为值 c时，n_u 将以（**响应** $v\langle k\rangle$，c，n_v，n_u）作出响应。

每个流变量都可以独立声明为急切或惰性。我们使用两个谓词 急切 (u) 和 惰性 (u)（定义为 ¬急切(u)），来表示流变量 u 的通信策略。请注意，惰性策略涉及两条消息，而急切策略只涉及一条；但急切策略会发送每个已解析的瞬时变量，而惰性策略只发送被请求的变量。如果几乎总是需要这些值，则急切策略更可取；如果较少需要这些值，则惰性策略更可取。现在，我们需要将通信策略加入去中心化 SRV 问题的定义。去中心化 SRV 问题 $\langle \varphi, \mathcal{T}, \mu, \text{急切} \rangle$ 现在由规约 φ、拓扑 $\mathcal{T}$、流分配 μ以及每个流变量对应的通信策略共同刻画。

5.1 定时异步网络的惰性 DSRV 算法

我们扩展本地监视器，$\langle Q_n, U_n, R_n, P_n, W_n\rangle$ 添加以下两种存储：

– **待处理的**请求 P_n，其中 n 记录了其他监视器向 n 请求的瞬时变量，但 n 尚未解析这些变量。– **等待**响应 W_n，其中 n 记录了 n 向其他节点请求、但尚未收到响应的瞬时变量。

存储 W_n 用于防止 n 在等待首次请求得到响应期间，对同一个值重复发起请求。收到该值后，会从 W_n 中删除相应条目，因为随后将直接从

R_n 获取该值，而不再通过网络请求。存储 P_n 用于记录： n 负责的某个值已被请求，但 n 尚不知道答案。当 n 计算出答案后， n 将发送相应的响应消息，并从 P_n中删除该条目。最后，为尚未解析的惰性瞬时变量生成请求消息，并将其插入相应邻居的队列中。

更具体地说，每个节点 n 都会执行算法 2 中所示的过程 MONITOR，该过程会在每个时钟周期调用 STEP，直至输入终止或无限期运行。若轨迹结束，过程 FINALIZE 会将轨迹末尾的待处理值解析为其默认值。过程 STEP 现在会执行一些经过修改的过程和额外步骤：

1. **处理消息**：第 26 行会标记 P_n中的请求，这些请求之后将得到解析并予以响应。第 27–28 行处理响应到达，将其添加到 R_n 中，并从 W_n中移除。 2. **发送响应：**第 33–36 行处理待处理的惰性变量。如果某个待处理的瞬时变量此时已得到解析，则会发送响应消息，并从 P_n中移除相应条目。 3. **发送新请求：**第 37–41 行会为当前所需的所有惰性瞬时流发送新的请求消息。 4. **剪枝：**第 42–44 行会从集合 R 中剪除不再需要的信息。请参见第 5.4 节。

5.2 形式化正确性

现在，我们再次证明该方案的正确性：证明计算得到的输出与指称语义中的输出相同，并且每个输出最终都会被计算出来。

定理 4. 以下结论对每个瞬时变量 $u\langle k\rangle$ 都成立：
(1) $u\langle k\rangle$ 的值最终会得到确定。(2) </style当且仅当在某个瞬时，($u\langle k\rangle$、c) $\in R$，$u\langle k\rangle$ 的值才是 c。(3) 如果 *eager*(u)，那么最终会发送针对 $u\langle k\rangle$ 的响应消息。(4) 如果 *lazy*(u)，那么最终会对针对 $u\langle k\rangle$ 的所有请求消息作出响应。

证明。 证明通过对求值图进行归纳来完成；由于这些结论彼此依赖（依赖于前面的归纳步骤），因此在归纳步骤中同时证明 (1)–(4)。设 M 是计算的长度，σ_I 是长度为 M 的输入。注意，上述 (1) 到 (4) 都是关于瞬时变量 $u\langle k\rangle$ 的陈述，而这些变量是求值图 $G_{\varphi,M}$ 的节点。我们对 $G_{\varphi,M}$ 进行归纳（由于 D_φ 是良构的，因此它是无环的）。

– **基本情况**：基本情况是指求值图中没有出边的顶点，它们要么是对应于输入的瞬时变量，要么是其瞬时方程不包含其他瞬时变量的已定义变量。对于输入，命题 (1) 显然成立，因为在时刻 k,$s\langle k\rangle$，会在节点 μ(k) 读取输入。对于不含变量的输出方程，

或者对于其变量的偏移量在实例化后变为负数或大于 M 的输出方程，其叶节点的值要么立即确定，要么在计算偏移量时于 M 确定。此时，计算出的值会插入 R，因此 (2) 在 $\mu(u)$ 也成立。注意，对于其他节点，(2) 也成立，因为响应消息包含 $u\langle k\rangle = c$ 当且仅当 $(u\langle k\rangle,\ c) \in R_n$，其中 $\mu(u) = n$。随后，响应消息恰好在其解析时插入，因此 (1) 蕴含 (3)。最后，在接收请求消息或解析 $u\langle k\rangle$ 时（以较晚发生者为准），(4) 也成立。

– **归纳情形**：考虑求值图中的任意一个 $u\langle k\rangle$ $G_{\varphi,\ M}$并令 $u_1\langle k_1\rangle \dots u_l\langle k_l\rangle$ 为其所依赖的瞬时变量。这些都是G_φ中的节点，$_M$ 其层级低于 $u\langle k\rangle$，因此归纳假设适用，且 (1)-(4) 对这些瞬时变量均成立。设 $n = \mu(u)$。在瞬时 k，$u\langle k\rangle$被实例化并插入 U_n。在周期 k结束时，会请求$u_1\langle k_1\rangle \dots u_l\langle k_l\rangle$ 中的惰性变量。根据归纳假设，在某个瞬时，所有这些请求都会得到响应，且 (1) 和 (4) 成立。类似地，所有急切变量的值也会被计算并发送（根据归纳假设，(1) 和 (3) 成立）。在最晚到达时，$u\langle k\rangle$ 的方程不再包含变量，并求值得到一个值，因此 (1) 成立，且 (2) 对 $u\langle k\rangle$ 在 n时成立。此时，如果 急切(u)，则会发送响应消息（因此 (1) 对 $u\langle k\rangle$成立）；如果惰性(u)，则会答复所有请求（之前在 P_n 收到的请求以及之后收到的请求），因此 (1) 也成立。

证明完成。

5.3 惰性情形的资源

分析惰性情形需要进行修改。在定时异步网络中，我们需要引入一种新的消息，以提供确认，这些确认仅用于通知接收节点某些瞬时变量已不再需要，因此可以对其进行剪枝。这种新消息的形式如下：

– **确认**消息：(**confirm** $s\langle k\rangle,\ n_s,\ n_d$)，其中 $s\langle k\rangle$ 是瞬时变量，n_s 是消息的源节点，n_d 是消息的目标节点。

该消息将被解释为：源节点 n_s 已解析截至 k的瞬时变量 s 。此信息使目标节点能够得出结论：对于已解析节点，远程节点所需的瞬时变量已不再必要。我们修改MTR_{rem}，使其包含以下规定：响应会在请求到达时或远程瞬时变量得到解析时发出，取两者中较晚者。

$$MTR^{lazy}_{rem}(s\langle t\rangle) \stackrel{\text{def}}{=} \max(t, M(s\langle t\rangle))$$

其中

$$M(s\langle t\rangle) = \{arr_{r\to s}(t') \text{ s.t.} s \xrightarrow[rem]{w} r \text{ and } t + w \geq 0\})$$

and

$$t' = \max(arr_{s\to r}(t), MTR^{lazy}(r\langle t+w\rangle))$$

这里，$arr_{s\to r}(t)$ 表示发送请求的时间，也就是瞬时变量 s 被实例化并存储在 U 中的时间。$MTR^{lazy}_{rem}(r\langle t+w\rangle)$ 表示远程瞬时流得到解析的时间。最后，$arr_{r\to s}(t')$ 表示惰性瞬时流变量的响应到达请求节点的时刻。

同步化实例化 同样，我们首先考虑这样一种情况：执行期间任意链路的延迟均为常量。这个常量有助于简化方程，但我们现在需要考虑，对于每个瞬时变量，都需要先发送请求，再接收响应，以获取远程值。同样，$dist_{r\ s}$ 用于表示每条消息从 $\mu(r)$ 到 $\mu(s)$ 所需的延迟，且不受消息发送时间瞬间的影响。我们利用这一信息来简化同步网络的 MTR^{lazy}_{rem}，如下所示

$$MTR^{sync\ lazy}_{rem}(s\langle t\rangle) \stackrel{\text{def}}{=} \max(t, t') \text{ s.t.} s \xrightarrow[rem]{w} r; t+w \geq 0\})$$

其中

$$t' = \{dist_{r_s} + \max(t + dist_{s_r}, MTR^{sync\ lazy}(r\langle t+w\rangle))$$

远程瞬时变量的值会在响应消息到达时到达 $dist_{r\ s}$。该响应消息在请求到达 $t+dist_{s\ r}$ 时发出，或在远程值确定 $MTR^{sync}(r\langle t+w\rangle)$ 时发出，取两者中较晚者。

永恒有界延迟 现在考虑这样一种情况：我们知道网络中的一个最大延迟，它是网络行为中所有其他延迟的上界。将上界值 d 代入 MTR 的方程后，我们得到 MTR 的一个恒定上界（尽管该值可能是一个粗略的过近似）：

$$MTR^{\mathbf{aeternal}\ lazy}_{rem}(s\langle t\rangle) \stackrel{\text{def}}{=} \max(t, M(s\langle t\rangle))$$

其中

$$M(s\langle t\rangle = \{d + \max(t+d, t') \text{ s.t. } s \xrightarrow[rem]{w} r; t+w \geq 0\})$$

and

$$t' = MTR^{\mathbf{aeternal}\ lazy}(r\langle t+w\rangle)$$

暂时有界延迟 最后，我们不假设网络延迟存在永久上界。相反，我们只需考察哪些因素会影响瞬时变量的计算，也就是说，它所依赖的其他瞬时变量，以及影响用于计算这些瞬时变量的消息的网络延迟。我们再次考虑时间窗口 $win(s\langle t\rangle)$，它包含了一个区间，该区间涵盖了所有影响 $s\langle t\rangle$的值被确定并发送的时刻。此时间窗口的结束时刻最迟为$MTR(s\langle t\rangle)$。在此窗口内，我们可以找到为计算该瞬时变量而发送的消息的最坏延迟： $d_{worst}(s\langle t\rangle)$。那么，对于惰性情形，$MTR(s\langle t\rangle)$可以界定如下：

$$MTR^{\mathbf{temp}\ lazy}_{rem}(s\langle t\rangle) \stackrel{\text{def}}{=} \max(t, \{t' \text{ s.t.} s \xrightarrow[rem]{w} r; t + w \geq 0\})$$

where

$$t' = d_{worst}(s\langle t\rangle) + \max \begin{pmatrix} d_{worst}(s\langle t\rangle) \\ MTR^{\mathbf{temp}\ lazy}(r\langle t + w\rangle) \end{pmatrix}$$

这里， $d_{worst}(s\langle t\rangle)$ 是影响 $s\langle t\rangle$ 计算的最坏消息所需的时间，因此获取该值的时间窗口同时考虑请求消息和响应消息。我们使用此值来限定请求和响应。首先，我们通过 $\max(d_{worst}(s\langle t\rangle), MTR^{\mathbf{temp}\ lazy}(r\langle t + w\rangle))$ 得到请求到达或远程依赖已解析两者中较晚的时刻，然后将 $d_{worst}(s\langle t\rangle)$ 中的值所对应的响应消息到达时间加上去。由此得到我们能够确定远程依赖必定已解析，且其值已到达请求方网络节点的时刻。

5.4 剪枝已解析存储。

现在，我们终于可以针对惰性算法的情况剪枝 R_n ，因为我们现在知道每个瞬时变量何时会被解析。

定理5。 每个在 U_n 中为惰性的未解析瞬时变量 $s\langle t\rangle$ ，最迟会在 $MTR^{lazy}(s\langle t\rangle)$ 时解析。

一旦到达这一时刻，考虑到网络延迟有界，就会向那些已计算出作为已解析瞬时变量依赖项的惰性瞬时变量的监视器发送确认消息，并且该消息会在有界时间内到达。随后，接收节点可以从其内存中剪枝相应的瞬时变量。现在，我们需要在该定理中加入 $tconf$ ，表示确认消息到达所需的时间：

每个尚未解析的 $s\langle k\rangle = e \text{ in } U_n$ 最多会在 $max(\{MTR^{lazy}(u\langle k-w\rangle) + tconf_u\})$ 时被剪枝。其中， $u\langle k-w\rangle$ 是一个远程瞬时变量，其方程中包含 $s\langle k\rangle$ ，而 $tconf_u$ 表示确认消息从在 $MTR^{lazy}(u\langle k-w\rangle)$ 时发送的 $\mu(u)$ 传输到 $\mu(s)$ 所需的时间。该消息会在有界时间内到达目的地，随后瞬时变量便会被剪枝。因为此时

接收节点知道该瞬时变量已不再需要，因此即使它尚未解析，也可以将其剪枝。通过这种机制，我们可以确保每个瞬时变量在内存中存在的时间（U_n， R_n）是有界的。这意味着，在定时异步网络中，可去中心化高效监控的规范能够使用有界资源进行监控。该上界仅与规约的大小、网络直径以及网络各节点之间的延迟呈线性关系。

6 结论与未来工作

我们研究了定时异步网络中的分布式流运行时验证问题，其中消息可能需要任意长的时间才能到达。该问题以一份规约和一个网络为输入。我们的解决方案包括对输出流进行放置，并在每个节点上运行一种在线局部监控算法。我们证明了所提算法的终止性和正确性。我们对规约和网络假设（同步、**永恒**和**临时**边界）进行了建模，这些假设能够保证监控所需的内存为常量，且与轨迹长度无关；这表明我们的解决方案涵盖了先前的同步算法。我们报告了原型工具 tadLola 的实证评估结果。评估结果表明，放置对性能至关重要，并且在大多数情况下，经过仔细的放置可以实现有界的开销和延迟。未来，我们计划将解决方案扩展到灾难场景，在这种场景下，某些链路可能出现无限延迟，导致消息无法通过该链路。我们的直觉是，可以利用规约和网络拓扑中的冗余来增强对故障网络链路的弹性，同时提供比仅仅复制时间异步算法、让各副本彼此隔离并行运行更好的性能。

参考文献

Asarin 等，2002。Asarin, E., Caspi, P. 和 Maler, O.（2002）。定时正则表达式。 *J. ACM*, 49(2):172–206。Barringer 等，2004。Barringer, H., Goldberg, A., Havelund, K. 和 Se n, K.（2004）。基于规则的运行时验证。载于 *Proc. of the 5th Int'l Conf. on Verification, ModelChecking and Abstract Interpretation (VMCAI'04)*， *LNCS*第 2937 卷，第 44–57 页。Springer。Basin 等，2015。Basin, D., Klaedtke 和 Zalinescu, E.（2015）。分布式系统的感知故障的运行时验证。载于 *Proc. of the 35th IARCS Ann. Conf. on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'15)* 45 *LIPIcs* 590 603 ，第 卷，页码 –。达格施图尔–莱布尼茨信息学中心。Bauer 等，2013。Bauer, A., K¨uster, J. 和 Vegliach, G.（2013）。从命题监控到一阶监控。载于 Legay, A. 和 Bensalem, S. 编， *Runtime Verifi-cation - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings* 8174 *Lecture Notes in Computer Science* 59 75 ，第 卷，页码 –。Springer。

Bauer 等，2011。Bauer, A., Leucker 和 Schallhart, C.（2011）。线性时态逻辑和时序时态逻辑的运行时验证。 *ACM Transactions on Software Engineering and Methodology* 20 4 14 , ( ): 。

Bauer 和 Falcone，2012。Bauer, A. K. 和 Falcone, Y.（2012）。去中心化LTL监控。载于 *Proc. of the 18th Int'l Symp. on Formal Methods (FM'12)*， *LNCS*第 7436 卷，第 85–100 页。Springer。Carbone 等，2015。Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S. 和 Tzoumas, K.（2015）。Apache Flink：单一引擎中的流处理与批处理。 *IEEE Data Eng. Bull.*, 38(4):28–38。Convent 等，2018。Convent, L., Hungerecker, S., Leucker, M., Scheffel, T., Schmitz, M. 和 Thoma, D.（2018）。TeSSLa：基于时态流的规范语言。载于 *Proc. of the 21th Brazilian Symp. on Formal Methods (SBMF'18)*， *LNCS*第 11254 卷，第 144–162 页。Springer。Cristian 和 Fetzer，1999。Cristian, F. 和 Fetzer, C.（1999）。定时异步分布式系统模型。 *IEEE Transactions on Parallel and Distributed Systems*, 10 6 642 657( ): –。Cumin 等，2017。Cumin, J., Lefebvre, G., Ramparany, F. 和 Crowley, J.（2017）。仪器化住宅中的日常活动数据集。第 413–425 页。D'Angelo 等人，2005。D'Angelo, B., Sankaranarayanan, S., S´anchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra 和 Manna, Z.（2005）。LOLA：同步系统的运行时监控。载于 *Proc. of the 12th Int'l Symp. of Temporal Representation and Reasoning (TIME'05)*，页码 –。 166 174 IEEE计算机学会出版社。Danielsson 和 S´anchez，2019。Danielsson, L. M. 和 S´ánchez, C.（2019）。去中心化流运行时验证。载于 Finkbeiner, B. 和 Mariani, L. 编， *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*， *Lecture Notes in Computer Science*第 11757 卷，第 185–201 页。Springer。Eisner 等，2003。Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac 和 Campenhout, D. V.（2003）。在截断路径上运用时态逻辑进行推理。载于 *Proc. of the 15th Int'l Conf. on Computer Aided Verification (CAV'03)*， *LNCS*第 2725 卷，第 27–39 页。Springer。El-Hokayem 和 Falcone，2017a。El-Hokayem, A. 和 Falcone, Y.（2017a）。去中心化规范监控。载于 *Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA'17)*，页码 –。 125 135 ACM。El-Hokayem 和 Falcone，2017b。El-Hokayem, A. 和 Falcone, Y.（2017b）。THEMIS：去中心化监控算法工具。载于 *Proc. of the 26th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA'17)*，页码 –。 125 135 ACM。El-Hokayem 和 Falcone，2020。El-Hokayem, A. 和 Falcone, Y.（2020）。关于去中心化规范的监控：语义、性质、分析与仿真。*ACM Trans. Softw. Eng. Methodol.*, 29(1)。Faymonville 等，2016。Faymonville, P., Finkbeiner, B., Schirmer, S. 和 Torfah, H.（2016 *Proc. of the 16th Int'l Conf. on Runtime Verification (RV'16)*）。用于网络监控的基于流的规范语言。载于 , 第 卷，第 152–168 页。Springer。Faymonville 等，2019。Faymonville, P., Finkbeiner, B., Schledjewski, M., Schwenger, M., Stenger, M., Tentrup, L. 和 Hazem, T.（2019）。StreamLAB：信息物理系统的基于流的监控。载于 *Proc. of the 31st Int'l Conf. on Computer-Aided Verification (CAV'19)*， 11561 *LNCS* 421 431 , 第 卷，页码 –。Springer。Francalanza 等，2018。Francalanza, A., P´érez, J. A. 和 S´ánchez, C.（2018）。去中心化与分布式系统的运行时验证。载于 Bartocci, E. 和 Fal-

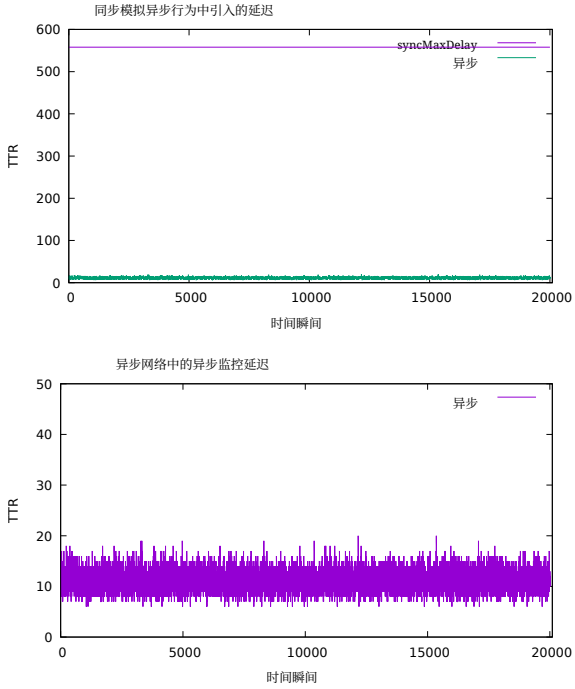
cone, Y., 编者. *Lectures on Runtime Verification - Introductory and Advanced Topics* 10457 *Lecture Notes in Computer Science* 176 210 , 卷, 页码 –。

Springer。Ganguly 等, 2021。Ganguly, R., Momtaz, A., and Bonakdarpour, B. (2021). 部分同步条件下的分布式运行时验证。载于 Bramas, Q., Oshman, R., and Romano, P., 编者, *24th International Conference on Principles of Distributed Systems (OPODIS 2020)* 184 *Leibniz International Proceedings* , 卷, 页码: –: *in Informatics (LIPIcs)* 20 1 20 17, 达格施图尔, 德国。Schloss Dagstuhl– Leibniz-Zentrum f¨ur Informatik。Gorostiaga 等, 2020。Gorostiaga, F., Danielsson, L. M., and S´anchez, C. (2020). 统一流运行时验证的时间—事件谱。载于 Deshmukh, J. and Nickovic, D., 编者, *Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9*, 2020, *Proceedings*, *Lecture Notes in Computer Science* 第12399卷, 页码462–481。Springer。Gorostiaga 和 S´anchez, 2018。Gorostiaga, F. and S´anchez, C. (2018). Striver: 实时事件流的流运行时验证。载于 *Proc. of the 18th Int'l Conf. on* , 卷, 页码 –。 *Runtime Verification (RV'18)* 11237 *LNCS* 282 298 Springer。Havelund 和 Ro.su, 2002。Havelund, K. and Ro.su, G. (2002). 综合生成安全属性监视器。载于 *Proc. of the 8th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, *LNCS*第2280卷, 页码342–356。Springer-Verlag。Jaber 等, 2020。Jaber, M., Falcone, Y., Attie, P., Khalil, A.-A., Hallal, R., and El-Hokayem, A. (2020). 从全局编排到可验证的高效分布式实现。 *Journal of Logical and Algebraic Methods in Programming*, 115 100577:。Kaupp 等, 2021。Kaupp, L., Webert, H., Nazemi, K., Humm, B., and Simons, S. (2021). Context: 智能工厂中上下文故障的工业4.0数据集。*Procedia Computer Science*, 180:492–501。第二届工业4.0与智能制造国际会议（ISM 2020）论文集。Kazemlou 和 Bonakdarpour, 2018。Kazemlou, S. and Bonakdarpour, B. (2018). 抗崩溃的去中心化同步运行时验证。载于 *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)* 207 212 , 页码 –。

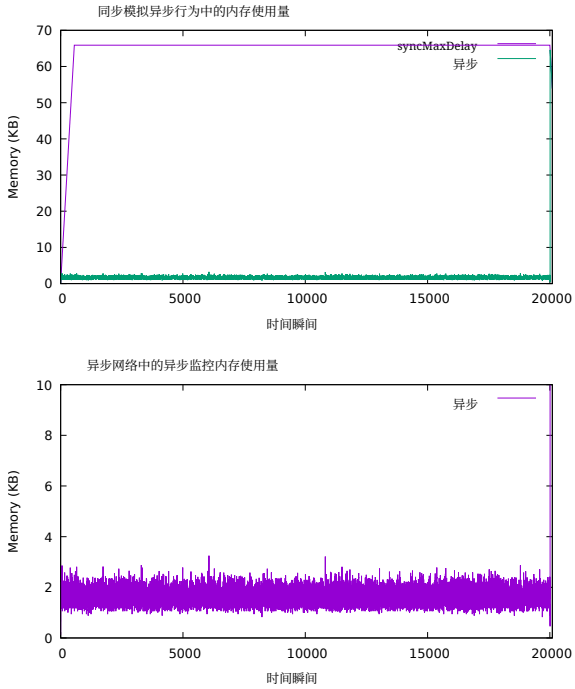
Pajuelo-Holguera 等, 2020。Pajuelo-Holguera, F., G´omez-Pulido, J. A., and Ortega, F. (2020). 学术设施中用于基于传感器的环境控制的推荐系统。 *Engineering Applications of Artificial Intelligence*, 96:103993。Perez 等, 2020。Perez, I., Dedden, F., and Goodloe, A. (2020). Copilot 3。技术报告 NASA/TM–2020–220587, NASA兰利研究中心。Pike 等, 2010。Pike, L., Goodloe, A., Morisset, R., and Niller, S. (2010). Copilot: 硬实时运行时监控器。载于 *Proc. of the 1st Int'l Conf. on Runtime Verification (RV'10)* , 卷, 页码 –。 6418 *LNCS* 345 359 Springer。Pike 等, 2013。Pike, L., Wegmann, N., Niller, S., and Goodloe, A. (2013). Copilot: 嵌入式系统监控。 *Innovations in Systems and Software Engineering*, 9 4 235 255( ): –。Quoc 等, 2017。Quoc, D. L., Beck, M., Bhatotia, P., Chen, R., Fetzer, C., and Strufe, T. (2017 *2017 USENIX Annual Technical Conference (USENIX ATC 17)* 659 672, 隐私保护流分析。载于, 页码 –, 圣克拉拉, 加利福尼亚州。USENIX协会。Ro.su 和 Havelund, 2005。Ro.su, G. and Havelund, K. (2005). 基于重写的运行时验证技术。 *Automated Software Engineering*, 12(2):151–197。S´anchez, 2018。S´anchez, C. (2018). 同步系统的在线与离线流运行时验证。载于 *Proc. of the 18th Int'l Conf. on Runtime Verification (RV'18)* , 卷, 页码 –。 11237 *LNCS* 138 163 Springer。

Sen 和 Ro.su，2003。Sen, K. and Ro.su, G. (2003). 为扩展正则表达式生成最优监测器。载于 Sokolsky, O. and Viswanathan, M., 编者，*Electronic Notes in Theoretical Computer Science* 89，卷。爱思唯尔。

Sen 等，2004。Sen, K., Vardhan, A., Agha, G., and Rosu, G. (2004). 高效去中心化监控分布式系统中的安全性。载于 *Proc. of the 26th Int'l Conf. on Software Engineering (ICSE'04)* 418 427 页码 –。IEEE计算机学会出版社。Zorin 和 Stukach，2020。Zorin, P. and Stukach, O. (2020). 来自俄罗斯托木斯克住宅建筑热量表的数据，用于建筑热特性的统计建模。



(a) 同步与异步的 TTR



(b) 同步与异步根监视器的内存

图 7. 异步网络中的同步与异步：异步的详细信息

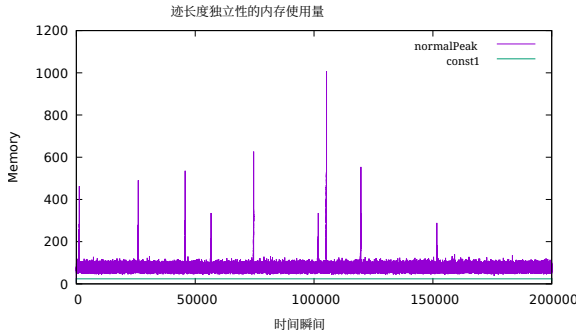


图8。 最近10000个时刻根监视器的最大内存

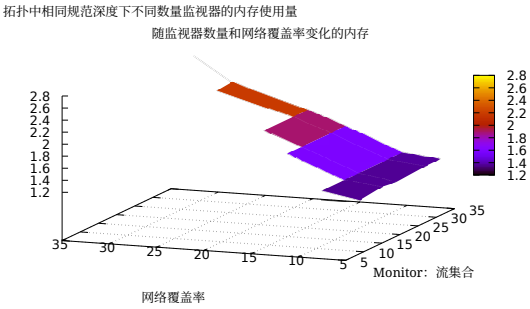


图9。 最近1000个时刻中内存使用量最大的监视器的平均内存使用量，tlen 20k，不同网络覆盖率下

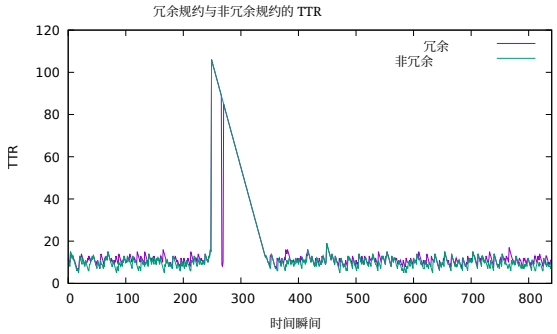


图10。 使用冗余在累计延迟方面的优势

算法 **2**节点 n 处的局部监控算法，使用 $MS_n = \langle Q_n, U_n, R_n, P_n, W_n \rangle$

过程 MONITOR
  $MS_n \leftarrow \emptyset$; $k \leftarrow$ NOW()
  当未结束时 **执行** STEP(k)
  $M \leftarrow k$; FINALIZE(M)

过程 STEP(k)
  $R_{old} \leftarrow MS_n.R_n$
  PROCESSMESSAGES(MS_n)
  $R_n.add(\{r\langle k\rangle \mapsto read(r, k) \mid r \in ins_n\})$
  $U_n.add(\{s\langle k\rangle \mapsto e_s\langle k\rangle \mid s \in outs_n\})$
  EVALUATE(MS_n)
  SENDRESPONSES(MS_n)
  SENDREQUESTS(MS_n)
  PRUNE(MS_n)

过程 EVALUATE(MS_n)
  $done \leftarrow false$
  当 $not\ done$ **时执行**
    $done \leftarrow true$
    对所有 $s\langle k\rangle \mapsto e \in U_n$ **执行**
      $e' \leftarrow$ SUBST(e, R_n)
      如果 $ground(e')$ **则** $done \leftarrow false$
        $U_n.del(s\langle k\rangle \mapsto e)$; $R_n.add(s\langle k\rangle \mapsto e')$
      否则 $U_n.del(s\langle k\rangle \mapsto e)$; $U_n.add(s\langle k\rangle \mapsto e')$

过程 PROCESSMESSAGES(MS_n)
  对所有 $msg \leftarrow Q_n.pop()$ **执行**
    switch msg **do**
      case $\langle$**req** $s\langle k\rangle\rangle$ $P_n.add(s\langle k\rangle)$
      case $\langle$**resp** $s\langle k\rangle, c\rangle$
        $R_n.add(s\langle k\rangle \mapsto c)$; $W_n.del(s\langle k\rangle)$

过程 SENDRESPONSES(MS_n, R_{old})
  $R_{new} \leftarrow MS.R_n \setminus R_{old}$
  对所有 $u\langle k\rangle \mapsto c \in R_{new}$**执行** ▷ 急切的新知识
    $send$ (**响应** $s\langle k\rangle$, c, n, n_r)
  对所有 $\langle$**请求** $s\langle k\rangle$, n_r, $n\rangle \in P_n$ **执行** ▷惰性请求
    如果 $s\langle k\rangle \mapsto c \in R_n$**则**
      $send($**响应** $s\langle k\rangle, c, n, n_r)$
      $P_n.del(\langle$**req** $s\langle k\rangle, n_r, n\rangle)$

过程 SENDREQUESTS(MS_n)
  对于所有$(\ e)\ \in U_n$ **do**
    对于所有 $u\langle k'\rangle \in sub(e)$ **执行**
      如果 $u\langle k'\rangle \notin W_n \wedge \mu(u) \neq n$ **则**
        $send($**req** $u\langle k'\rangle, n, \mu(u))$; $W_n.add(u\langle k'\rangle)$

过程 PRUNE(MS_n) ▷ 远程瞬时变量 v 已解析，请求如果发送，本来会到达 ▷ 如果 $u\langle j\rangle$ 不再需要
  对于所有 $u\langle j\rangle \mapsto c \in R_n \mid k \geq j + 2 * d + MTR(v)$ **执行**
    $R_n.del(u\langle j_i\rangle \mapsto c_i)\}$ ▷ 移除