计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.009818] [CSTR: 32024.14.csa.009818]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

RPKI 依赖方分布式同步系统①

邵　晴, 包　卓, 马　迪

(互联网域名系统北京市工程研究中心有限公司, 北京 100102)
通信作者: 包　卓, E-mail: baozhuo@zdns.cn

摘　要: 近年来互联网码号资源公钥基础设施 (resource public key infrastructure, RPKI) 部署率逐年上升, 这对依赖方软件原有单体同步的架构在性能与效率方面提出了挑战-其架构设计需要进行重新考量, 以适应 RPKI 技术的演进. 本文对 RPKI 同步任务进行了梳理与分析, 并基于此设计了一个 RPKI 依赖方任务同步系统. 相比单体架构, 该系统的分布式架构有着较高的同步性能及节点容错性. 同时, 本文为该系统设计了多种调度算法, 同时, 为进一步优化该系统性能, 本文对这些调度算法及任务调度策略进行了多组对照分析实验. 从实验结果看, 该分布式系统在大作业优先 (large job first, LJF) 任务调度策略下的动态调度算法表现性能最佳.
关键词: 互联网码号资源公钥基础设施; RRDP; Delta; RPSTIR2; 分布式调度

引用格式: 邵晴,包卓,马迪.RPKI 依赖方分布式同步系统.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/9818.html

RPKI Relying Party Distributed Synchronization System

SHAO Qing, BAO Zhuo, MA Di

(Internet Domain Name System Beijing Engineering Research Center Co. Ltd., Beijing 100102, China)

Abstract: In recent years, the deployment rate of resource public key infrastructure (RPKI) has been increasing year by year, which challenges the performance and efficiency of the original monolithic synchronization architecture of the relying party software. Hence, its architectural design needs to be reevaluated to adapt to the evolution of RPKI technology. This study sorts out and analyzes the RPKI synchronization tasks, and then designs an RPKI relying party synchronization system based on the above analysis. Compared with the monolithic architecture, this distributed architecture boasts higher synchronization performance and node fault tolerance. At the same time, this study designs a variety of scheduling algorithms for the system. To further optimize the performance of the system, this study carries out groups of control analysis experiments of these scheduling algorithms and task scheduling strategies. From the experimental results, the dynamic scheduling algorithm under the large job first (LJF) task scheduling strategy has the best performance in this distributed system.
Key words: resource public key infrastructure (RPKI); RRDP; Delta; RPSTIR2; distributed scheduling

1　背景与现状

边界网关协议 (border gateway protocol, BGP) 路由劫持事件频发, 威胁互联网的安全. 针对这个议题, 国际互联网工程任务组 (Internet engineering task force, IETF) 提出互联网码号资源公钥基础设施 (resource public key infrastructure, RPKI) 技术并对其进行了标准化推动.

RPKI 是一种解决前缀劫持和路由泄露的技术解决方案. 该技术能有效增强 BGP 协议的安全性能. 该技术的主要思路是利用公钥基础设施 (public key infra-

① 基金项目: 国家重点研发计划 (2022YFB3104800)
收稿时间: 2024-09-27; 修改时间: 2024-10-23; 采用时间: 2024-11-12; csa 在线出版时间: 2025-02-28

structure, PKI) 技术对路由源信息进行授权. 并应用该授权数据指导现网环境的路由器进行非法前缀验证过滤. 经过多年发展. 该技术已有较高成熟度和可靠度. 目前 RPKI 受到世界各国网络安全战略的关注. 其中美国白宫在 2024 年 9 月发布的“增强互联网路由安全路线图”[1]报告着重概述了 RPKI 推广的行动计划和政策框架.

RPKI 构成组建主要分为数据生产方认证中心 (certificate authority, CA)、消费者依赖方 (relying party, RP) 软件及验证执行者边界路由器[2]3 个部分. 这三者紧密合作, 共同保证路由信息的安全性. 其中, RP 软件与 CA 的发布点保持一定同步周期, 定期进行数据的同步[3]. 随后, RP 软件将有效的数据条目发送到路由器, 路由器将其存储以作前缀通告校验之用.

1.1 研究背景

RP 依赖方软件是 RPKI 体系的中间服务, 向上承接 RPKI 数据的同步、验证逻辑, 向下为边界路由器提供数据分发服务. 作为 RPKI 体系重要一环, RP 软件的可用性, RP 软件与发布点的交互方式与效率, 对于确定 RPKI 是否有效工作显得至关重要. 可知目前 RP 软件缺乏成熟度, 缺乏生产级的韧性, 其性能、可靠性、拓展性直接影响下游路由器侧对非法数据的识别与过滤能力[4], 导致整个 RPKI 系统应对互联网路由安全潜在威胁与风险的防护强度不足.

目前大约有 50% 的 IPv4, 52% 的 IPv6 拥有路由源授权 (route origin authorization, ROA)[5]并且完成 RPKI 有效验证[6]. 可知随着用于完成路由声明校验的 RPKI 数据规模愈来愈大, RP 软件与发布点之间的连接已经被广泛认同是关键的互联网基础设施[2]. 其次, RPKI 签名对象有效期较短且不规律, 为保证数据的一致性与时效性, RP 更新时间间隔需要保证足够及时. 在 RPKI 部署率逐步攀升的背景下, 单体架构的 RP 架构存在设计瓶颈, 导致在大规模数据频繁同步时, 其整体同步性能表现不佳. RPKI 部署率的上升, 势必对 RP 的可用性和拓展性带来考验与挑战.

分布式架构有解决高可用性、高拓展性、容错性等问题的优势, RPKI 同步系统业务场景也有着高可用性、高拓展性、容错性的需求, 适合进行分布式化研究.

1.2 应用背景

RPKI 将所有数据委托区域互联网注册机构 (regional internet registry, RIR) 进行托管与发布, 其仓库的发布点目前有 80 多个, 地理位置跨度分布于全球五大洲, 跨度较广. 依赖方同步系统分布式化, 使用就近节点对发布仓库进行网络同步, 能解决跨区域带来的网络可达性, 减少网络延迟.

单体架构易于部署和推广, 符合 RPKI 推广初期的市场需求. 目前市面流行的 RP 软件包括不限于 Routinator、Validator3、OctoRPKI、Fort、RPKI-Prover、rpki-client[7]. 这些 RP 软件大部分主要仍然采用单体架构, 如表 1 所示. 单体架构的 RP 软件具有单点失效的风险, 软件出现故障会导致整个服务不可用. 分布式架构通过节点冗余设计与分布式同步机制, 能显著提高系统的容错性和可拓展性.

表 1 开源 RP 软件

项目	组织	架构
Routinator	NLnet Labs	单体
Validator3	RIPE NCC	单体
OctoRPKI	Cloudflare	单体
Fort	NIC.MX	单体
RPKI-Prover	Mikhail Puzanov	单体

RPKI 是一个复杂的系统, 需要网络运营机构的协调与合作, RP 作为“桥梁”, 为不同角色的运营上提供同步服务, 因此 RP 需要进行分区化与协同化改造. 可知 RPKI 分布式化架构升级, 是实际业务场景的需求带来的必要结果, 具有极高的应用价值.

1.3 相关工作

目前, 对于 RPKI 依赖方架构优化方向的系统性研究工作有限, 只有少量工作涉及该方向. 文献[8]提出了依赖方系统的解耦机制与编排机制的理论框架, 其中提到了分布式数据同步模块的构想. 文献[9]基于“森林形状信任模型”设计了 RP 同步机制, 将同步任务进行分拆, 通过负载均衡算法将同步任务分摊到其余分布式从节点. 其实现较为简单, 缺少节点容错性与任务调度可预测性上的考虑.

2 系统任务分析与系统模型设计

基于开源软件 RPSTIR2, 本文设计并实现了一套分布式架构的 RP 同步系统, 目标在于降低原有单点依赖的风险, 提高依赖方软件的同步效率, 提升系统可拓展性. 可知在该分布式同步系统中, 同步任务的调度分配是系统的核心问题. 为明确在同步系统中进行任务分发的决策依据, 本节对同步任务进行了深入分析, 随即构建了适用的系统模型.

2.1 任务特征分析

在 RPKI 的同步协议部分, 主要包括远程同步 (remote sync, Rsync) 协议和互联网码号资源公钥基础设施仓库增量协议 (RPKI repository delta protocol, RRDP). 与 Rsync 协议相比, RRDP 协议表现得更加健壮与稳定, 因此 IETF 主推 RRDP 协议作为 RPKI 资料库的默认协议. 本文工作也主要围绕 RRDP 协议进行展开. 在 RRDP 协议中, RPKI 资源发布点会根据自己的发布策略, 定期发布 Notify 文件[10,11]. RRDP Notify 文件中包含仓库的快照文件和增量记录, 通过自增的版本号码对资源进行版本隔离.

如图 1, 在 RRDP 的 Notify 文件中, 包含全量同步任务与增量同步任务的信息描述. 可知全量同步的文件集合表示为 $F_{\rm snapshot}=\{f_{repo,{\rm latest_serial}}\mid repo\in\{jpnic, cnnic\ldots\}\}$, 即各大 RIR 的 Notify 文件中最新版本号的快照文件. 一般的, 在 RP 进行冷启动时, 通常会遍历下各 RIR 的 TAL 锚定文件, 在获取远端发布点地址后, 依次对 RRDP 仓库地址发起全量同步请求. 除此之外, RP 软件在进行增量同步时, 一旦发现大批量 Delta 文件不可用, 也会直接发起全量同步[9].

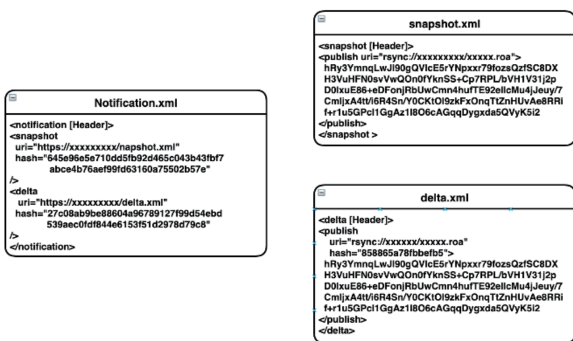

图 1 通知文件

增量同步任务的文件集合表示为 $F_{\rm delta}=\{f_{repo,serial}\mid repo\in\{jpnic,cnnic\cdots\}, serical_{\min}<serial<serical_{\max}\}$, 可知当 RP 本地版本落后于远端最新版本, 会触发增量同步, 去获取落后的版本区间的增量数据. 在落后的本地版本基础上新增或者删减 X509 签名对象与证书文件.

可知同步任务存在以下约束.

(1) 同步任务以 Notify 地址的粒度进行分发. 可知一个 Notify 文件对应一个全量更新任务.

(2) 由于增量任务存在版本号的隔离机制, 为保证数据的一致性, 需把同一 Notify 地址的增量子任务进行封装, 作为下发任务的最小单元. 可知 Notify 地址与同步子任务存在一对多的关系.

增量同步和全量同步均可定义若干子任务的同步任务序列, 具体的任务表示为 $(t_{\rm download}, t_{\rm validate}, t_{\rm parse}, t_{\rm extra}, t_{\rm sync})$. 如图 2 所示, 即包括文件下载、文件解析、签名验证、数据提取及数据的同步与存储.

具体的, RRDP 文件以 XML 格式进行发布, RP 需要通过 HTTP 请求获取 XML 格式的 Notify 文件. 下载完毕之后, 再通过本地解析分别获得快照文件和增量文件的发布地址. 结合本地缓存情况, RP 选择需要的全量或者增量版本, 再次发起 HTTP 请求, 获取包含有签名对象数据的 XML 文件. 在任务的最后, RP 进行本地密码学验证用于完成签名验证, 最终根据数据所需进行数据的提取与存储[10]. 与快照文件全量同步不同的是, 多了一步数据规整的逻辑, 增量同步请求多个文件之后, 需要根据版本进行排序, 以保证数据的一致性.

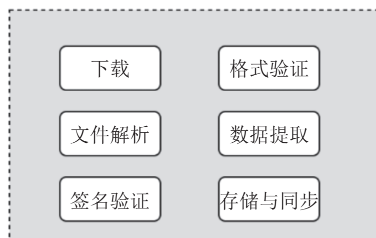

图 2 同步任务

执行 RPKI 数据的同步任务, 需要消耗相应计算资源, 包括 CPU 资源 C, 内存资源 M 及其网络带宽资源 N. 一次同步所需的资源消耗可形式化表达为:

$$P_i=\{p\ \exists C_i,M_i,N_i: p=f(C_i,M_i,N_i)\} \tag{1}$$

其中, CPU 的消耗可以继续拆分为:

$$C_i=C_i^{\rm net}+C_i^{\rm crypto}+C_i^{\rm other} \tag{2}$$

其中, $C^{\rm net}$ 表示为网络行为中 CPU 的消耗, $C^{\rm crypto}$ 表示密码学运算中对 CPU 资源的消耗.

在进行 RPKI 数据同步时, 不同子任务对系统不同资源有不同的需求, 每项资源的权重系数根据任务的特性会有所差异. 例如, 下载子任务为网络 IO 密集型系统任务, 最主要消耗的是网络带宽资源, 而校验子任务则主要为计算密集型任务, 则主要消耗的是 CPU 资源. 上述两种情况消耗可简单表示为:

$$p_{\rm download}\approx N_i \tag{3}$$

$$p_{\rm validate}\approx C_i \tag{4}$$

在 RP 的一次完整同步流程中, 完整的资源消耗可表示为:

$$P_{\text{total}} = \sum_{i=1}^{n}\left(\alpha C_i^{\text{net}} + \beta C_i^{\text{crypto}} + \gamma C_i^{\text{other}} + \delta M_i + \epsilon N_i\right) \tag{5}$$

通过对同步任务的分析, 可以得知, 在同步任务中, 网络带宽, 存储资源与 CPU 资源的资源瓶颈很大程度影响着系统性能. 为解决可能遇到的单点性能瓶颈, 将同步任务平摊给分布式节点, 能够充分利用节点的资源, 加快整体同步速率, 提高系统整体的响应能力.

RP 运行过程中, 同步任务发生错误会使局部数据同步失败, 导致上游供给侧与下游需求侧数据的一致性问题, 降低 RPKI 数据对网络攻击的防御能力[12,13]. 其中, 同步任务的错误可大致分为如下 3 种.

(1) 单点故障. 导致 RP 单点故障的原因众多, 包括且不限于网络不可达导致的服务不可用、服务配置错误或者服务存在缺陷导致的服务启动失败、节点硬件资源耗尽导致的服务崩溃等. 在缺乏 Failover 机制的情况下, 该错误对系统的运行来说是致命的, 会直接导致服务不可用.

(2) 网络传输错误. RRDP 主要通过搭建 Web 应用对外提供服务, 依赖方软件通过 HTTP 进行同步请求. DNS[14]解析超时, 请求文件过大, RRDP 仓库服务端过载、服务资源限制、RP 客户端的网络带宽限制、错误的信任锚点, 均会导致客户端 HTTP 请求的失败. 除此之外, RRDP 会定期清理旧版本文件, 依赖方本地版本过低, 也会导致旧版本的请求失败.

(3) 文件验证错误. 此错误主要是由于 RRDP 服务提供的内容完整性校验失败而引入的错误, RP 自身无法解决. 包括 XML 文件格式错误、签名对象 Hash 完整性校验失败、X509 校验失败等.

2.2　系统模型设计

在分布式系统中, 常根据机器性能等多种特征进行机器选择, 建模与算法较为复杂, 同时对机器性能参数精准程度要求较高. 本文简化了模型, 基于队列模型提出了按需消费的 RPKI 同步模型. 该模型根据机器实例的性能参数, 计算出合适的消费速率, 进行同步任务的消费, 能够提高 RPKI 同步场景下的稳定性与可预测性. 该系统的模型设计如图 3.

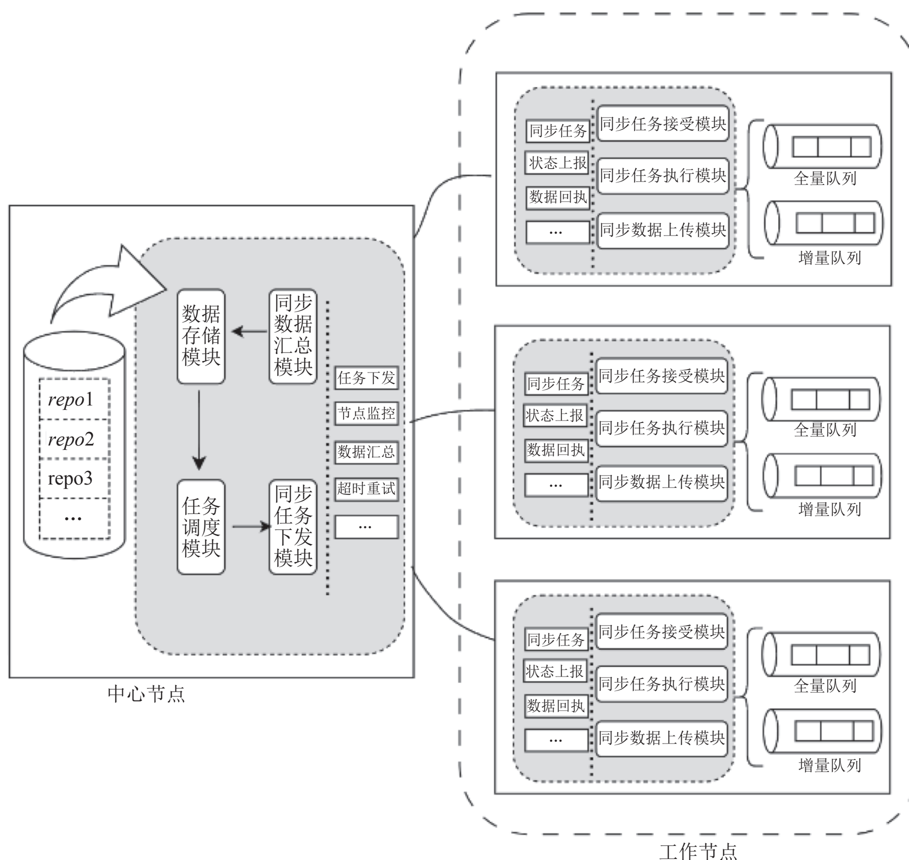

图 3　系统架构

2.2.1 系统功能组件

在中心节点上, 包括数据存储模块、任务调度模块、同步任务下发模块和同步数据汇总模块. 其中, 数据存储模块主要负责同步节点信息和资料库信息的存储; 任务调度模块负责根据调度算法对具体同步任务序列的计算; 同步任务下发模块主要负责任务序列的具体下发; 同步数据汇总模块负责将工作子节点回传的同步数据继续汇总, 并将汇总数据传递给数据存储模块.

而在各个工作节点上, 包括同步任务接收模块、同步任务执行模块和同步数据上传模块. 同步任务接收模块主要负责同步任务的接收; 同步任务执行模块主要都负责实际的 RPKI 数据同步与更新业务逻辑的处理; 同步数据上传模块负责将同步完成的数据回传给中心节点.

2.2.2 任务调度

同步任务的调度模块是系统核心, 任务选择和节点选择二者构成了调度的关键部分, 如图 4. 调度问题可定义成 $seq: Task \times Node \rightarrow Seq$, 任务选择决定任务队列的入队与出队策略及其任务的调度序列 $Task = \{task_1, task_2, task_3, \cdots\}$. 节点选择器则根据各种选择算法对分布式工作节点进行挑选, 产生节点调度序列 $Node = \{node_1, node_2, node_3, \cdots\}$. 整个同步任务调度序列表示为式 (6):

$$\forall sq \in SchedSeq, \exists Task_i, \exists Node_k | sq = SchedSeq(Task_i, Node_k) \tag{6}$$

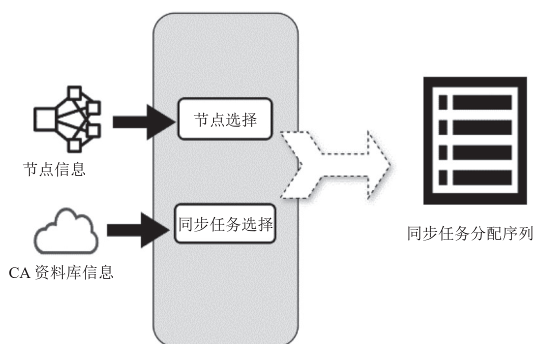

图 4 任务调度

调度流程中, 中心节点定期探测 CA 资料库, 获取最新版本, 计算可得同步任务序列. 根据不同的调度算法策略产生节点序列, 选择节点 $node$ 进行任务下发. 调度容错方面, 下发过程中一旦出现错误, 将任务加入失败任务队列 Q, 重新进行任务分配.

根据工作节点性能特征与同步任务特征, 简单的在工作节点中抽象出两个相互隔离的本地局部队列, 分别负责增量与全量任务, 表示为队列 q_{snapshot} 和 q_{delta}, 分别负责全量同步任务序列 T_{snapshot} 与增量同步任务 T_{delta} 的处理. 为防止其中某个队列把资源消耗完, 需要对消费者的并发度进行限制. 分别设置并发限额, 对两个队列进行资源隔离, 确保 $q_{\text{snapshot}} \cap q_{\text{delta}} = \varnothing$.

2.2.3 同步完成

工作节点从队列完成任务的消费, 产生签名对象的数据集合 O, 通过 RPC 接口将数据回传给中心节点, 完成资料库的数据同步任务. 中心节点接收到回传的数据序列 $\{o_1, o_2, \cdots\}$, 完成校验, 通知同步任务计时器, 告知本数据同步任务已经完成, 其次, 将所有数据传递给系统的数据存储模块, 将本次更新的数据写入到本地缓存文件.

同步任务的完成状态可以形式化描述为式 (7), 来判定同步任务已完成.

$$\begin{aligned} &\text{if}(\forall Queue^{i}_{\text{snapshot}} = \varnothing) \cap \left(\forall Queue^{i}_{\text{delta}} = \varnothing\right) \\ &\cap (Queue_{\text{retry}} = \varnothing) \cap (Task = \varnothing) \end{aligned} \tag{7}$$

若状态如式 (8)，表示系统还处于同步中.

$$\begin{aligned} &\text{if}(\exists |Queue^{i}_{\text{snapshot}}| > 0) \cup \left(\exists \left|Queue^{j}_{\text{delta}}\right| > 0\right) \\ &\cup \left(\left|Queue_{\text{retry}}\right| > 0\right) \cup (|Task| > 0) \end{aligned} \tag{8}$$

2.3 效能评估

在分布式同步系统中, 工作节点定期将自身静态信息与动态信息同步给中心节点.具体的, 调度模型中的动态信息主要为机器的任务负载信息, 包括目前增量任务和全量任务的处理状态 $P = ((P_{\text{all}}, P_{\text{wait}}, P_{\text{done}}) \cdots)$. 静态信息主要为机器配置信息, 包括 CPU 资源 C, 内存资源 M, 网络带宽资源 B. 可知机器的吞吐受限于系统的处理能力, 其中 C 和 M 共同为机器吞吐做正向作用, 参与的内存资源和计算 CPU 资源越多, 机器的吞吐性能越高;机器带宽 B 越高, 越能传递更大规模的数据, 在网络资源 IO 密集的任务中, 能提高机器的吞吐性能. 理想的机器吞吐性能因子可简化表示为:

$$Factor = \frac{\eta CM}{\zeta \cdot (P_t + \xi P_s + \tau P_w) + \kappa \cdot \dfrac{1}{B}} \tag{9}$$

假设在下载后, 两个队列均无等待时间, 直接进入

解析逻辑, 则在非空闲阶段, 根据服务队列理论, 同步队列的服务率表示为 $\mu_{\text{syn}} = \min(pt_{\text{download}} \times r_{\text{download}}, pt_{\text{parse}} \times r_{\text{parse}}) \cdot Factor$. 其中, pt_{download} 和 pt_{parse} 分别是下载任务和校验任务的并发度, r_{download} 和 r_{parse} 分别为处理单个下载和解析任务的速率. 可知同步的整体速率取决于下载与解析中的最慢速率.

直接通过对实际服务时长进行观测, 对系统响应性能进行评估. 可知节点需要增量和全量任务全都执行完成, 才标记为结束. 即 $T_i^{\text{total}} = \max(T_i^{\text{snapshot}}, T_i^{\text{delta}})$ 可知其同步任务的响应时长主要取决于处理最慢队列. 则可得节点服务时长表示为:

$$T_{\text{node_service}} = T_{\text{node_end}}^{\text{total}} - T \tag{10}$$

整个同步任务完成时长:

$$T_{\text{sys_service}} = T_{\text{sys_end}} - T_{\text{sys_start}} \tag{11}$$

节点利用率表示为:

$$U_{node_i} = \frac{\lambda}{\mu} = \frac{T_{\text{node_end}}^{\text{total}} - T_{\text{node_start}}^{\text{total}}}{T_{\text{sys_end}} - T_{\text{sys_start}}} \tag{12}$$

系统利用率为节点利用率的期望值, 表示为:

$$U_{\text{sys}} = \frac{1}{n} \sum_{i=1}^{n} U_{node_i} \tag{13}$$

3 调度算法

分布式数据同步涉及多节点的任务分配与调度, 为了避免节点间任务分配不合理而导致同步速度降级的情况, 本文结合实际情况设计了多种任务调度算法, 包括顺序任务调度 (sequence) 算法、随机任务调度 (random) 算法、动态权重任务调度 (performance) 算法、网速调度 (repoState) 算法、RIR 地理位置调度 (assignedRir) 算法和指定地理位置调度 (locationRir) 算法等. 可根据待同步的任务数量和同步节点的性能, 选择合适的算法, 从而提高整个系统的可用性、健壮性和性能.

3.1 sequence 算法

顺序分配任务调度算法是最简单的任务调度算法, 该算法从可用服务器列表中按顺序选择一个同步节点去执行任务. 其排序的依据可以是 IP 地址, 也可以内部的序列号. 该算法优点是实现简单, 并且能大致平均分配任务, 但缺点是没有考虑同步节点间性能的差异, 及同步任务的差异, 会导致系统整体的同步性能潜力无法得到充分发挥.

算法 1. sequence 算法

1) 根据工作节点的注册信息, 获取节点列表;
2) 根据获得的节点列表, 获取列表第 1 个元素的索引 *index*;
3) 取出任务列表的第 1 个任务, 对当前索引节点进行任务下发;
4) 对 *index* 依次递增, 选择当前索引的节点. 如果 *index* 等于节点长度–1, 则将 *index* 重新置为 0;
5) 循环步骤 3) 和 4), 直到任务列表为空终止循环.

3.2 random 算法

随机分配任务调度算法是比较常见的任务调度算法, 作为该分布式同步系统的默认调度算法, 该调度算法也作为其他算法失效时的降级兜底算法, 以保障系统的基本运行, 提高系统的鲁棒性, 避免因算法失效导致的任务堆或者任务分发中断. 该算法在每次需要进行任务调度时, 从可用同步节点列表中随机选择一个去执行任务. 该算法优点是实现简单, 缺点同样是没有考虑节点当前的负载情况, 可能会将同步任务分配在已经过载的同步节点上, 导致系统性能下降.

算法 2. random 算法

1) 根据工作节点的注册信息, 获取节点列表;
2) 根据获得的节点列表, 获取列表的长度 *N*;
3) 在 0 到 *N*–1 的范围内, 生成一个随机整数, 该数可当作索引数 *index*;
4) 以 *index* 获取对应节点信息;
5) 取出任务列表的第 1 个任务, 给当前索引节点进行任务下发;
6) 循环步骤 3)–5), 直到任务列表为空终止循环.

3.3 performance 算法

动态权重任务调度算法能够动态地根据同步节点的性能指标 (CPU 内核数和可用率、内存总量和可用率、网络带宽等), 及当前正在执行的同步任务数量等, 进行动态加权计算得到所有同步节点的分值, 根据分值高低选择执行任务的同步节点. 该调度算法应用于节点数量较多, 且机器性能存在性能差异, 或者节点资源利用率不稳定的场景. 可知此场景下需要根据实时参数来调整任务的具体分配, 从而实现节点系统资源的充分利用, 提高系统对复杂环境的适应能力.

根据物理资源和 RP 同步任务的特点, 针对节点, 其得分计算需要考虑的参数有: 系统可用资源 (CPU 资源 (包括 CPU 频率和内核数))、内存资源 (内存大小)、硬盘资源 (硬盘容量) 和网络带宽等资源信息, 并且做正则标准化处理. 同时, 还需考虑当前正在执行的

任务数量, 分为增量同步任务和全量同步任务. 得分计算中, 节点计算得到的得分值会随着分配的情况而动态改变. 得分最多的同步节点获得此次同步任务的执行权.

$$Score = \frac{\bar{w} \cdot \bar{s}}{1 + \delta \times T_i} \tag{14}$$

其中, $\bar{w} = \left[weight_{\rm cpu}, weight_{\rm mem}, weight_{\rm disk}, weight_{\rm net}\right]$, $\bar{s} = \left[score_{\rm cpu}^{\rm norm}, score_{\rm mem}^{\rm norm}, score_{\rm disk}^{\rm norm}, score_{\rm net}^{\rm norm}\right]$.

此算法的优点是充分考虑了同步节点间性能的差异, 以及当前的负载情况, 从而使整个系统达到更好的负载均衡效果. 但缺点是算法比较复杂, 需要事先反复测试调整各种加权参数, 才能达到比较合适的均衡效果.

算法 3. performance 算法

1) 根据工作节点的注册信息, 获取节点列表, 同时获取 CPU 资源, 存储资源, 硬盘资源, 网络带宽资源;
2) 对上述资源向量做最大标准化处理, 并与权重向量相乘;
3) 从节点列表中获取每一个节点中的队列执行情况, 获取等待数量, 其中根据历史值获取对应任务的权重, 计算任务总权重;
4) 计算总得分, 并获取得分最高的节点索引;
5) 如果获取节点失败, 则降级到随机索引算法, 获取随机的节点索引;
6) 根据节点索引获取节点, 并获取任务队列中的任务, 进行任务下发;
7) 循环步骤 2)–6), 直到任务列表为空终止循环.

3.4 repoState 算法

网速调度算法事先将每个同步节点对所有资料库下载地址发起网络测速, 并对下载速度进行记录. 当执行任务分配时, 选择网速最快的节点执行同步.

此算法优点是完全根据实际的网速来分配任务, 更加贴合实际业务. 但此算法缺点在于忽略了节点的实际负载情况. 测速时速度快的同步节点由于被大量分配了同步任务, 反而导致降低了同步速度. 并且如果出现了新的下载地址, 由于尚未进行测速, 导致无法分配同步节点, 需要其他算法进行补充.

算法 4. repoState 算法

1) 根据工作节点的注册信息, 获取节点列表;
2) 依次获取节点的网速数据, 并进行排序;
3) 获取网速排序最高的节点;
4) 如果节点选择出错, 则降级到随机算法, 获取随机的节点索引;
5) 取出任务列表的第 1 个任务, 对当前索引节点进行任务下发;
6) 循环步骤 2)–5), 直到任务列表为空终止循环.

3.5 assignedRir 算法

根据每个同步节点所在的地理位置, 确定其所属的 RIR. 在执行任务分配时然后按照同步节点和下载地址是同一个 RIR 的方式进行分配. 比如同步节点在美国, 则所属的 RIR 为 ARIN, 而如果某个下载地址也是美国来自 ARIN, 则将此任务分配给此同步节点. 对于 RIR 地理位置调度算法, 当按 RIR 分布没有匹配到合适同步节点时, 默认采用随机分配任务调度算法, 确保至少能够找到一个同步节点.

此算法优点在于能够根据同步节点和下载地址的地理位置就近分配, 加快网络下载速度. 其缺点有如下几点, 首先是分配依据粗略, 可能导致某些同步节点由于所属的 RIR 的 RPKI 数据较多, 而被分配了大量的任务; 其次如果有多个同步节点属于同一个 RIR, 就还需要其他算法进行补充; 最后是同步协议 RRDP 可能采用了 CDN 缓存机制, 这会导致同步节点可能和 CDN 的下载地址其实并不是就近下载, 无法提高网络下载速度.

算法 5. assignedRir 算法

1) 根据工作节点的注册信息, 获取节点列表;
2) 遍历节点列表, 建立 RIR 到节点信息的索引;
3) 下发任务时, 获取同步任务的仓库信息, 索引到相应的节点;
4) 要是没有相应节点被索引到, 则降级为随机算法, 随机对节点选择索引;
5) 取出任务列表的第 1 个任务, 对当前索引节点进行任务下发;
6) 循环步骤 2)–5), 直到任务列表为空终止循环.

3.6 locationRir 算法

locationRir 算法和 assignedRir 算法类似, 也是实现得到同步节点所属的 RIR, 以及下载地址所属的 RIR, 在执行分配时, 是根据输入的参数进行指定分配, 比如下载地址在 ARIN 的任务, 通过参数指定由属于 RIPE NCC 的同步节点执行下载. 对于指定地理位置调度算法, 当按 RIR 分布没有匹配到合适同步节点时, 处理方式和 RIR 地理位置调度算法类似, 也是默认采用随机分配任务调度算法, 确保至少能够找到一个同步节点.

此种算法主要目的是用于研究, 通过分析指定不同 RIR 下载时同步速度的差异, 从而优化下载地址和优化同步节点的地址位置.

算法 6. locationRir 算法

1) 根据工作节点的注册信息, 获取节点列表;
2) 获取每一个节点的 RIR 信息, 并建立 RIR 信息到节点信息的索引;
3) 取出任务列表的任务, 根据任务的 RIR 信息, 再根据步骤 2) 建立的索引, 获取对应的节点信息. 进行任务的下发;
4) 要是没有相关的节点能被索引到, 则降级使用随机算法进行节点选择索引;
5) 取出任务列表的第 1 个任务, 对当前索引节点进行任务下发;
6) 循环步骤 2)–5), 直到任务列表为空终止循环.

4 实验分析

4.1 实验环境

目前在 RPKI 同步系统分布式扩展的研究领域, 相对研究工作较少, 有少量的研究对该系统的分布式协作性能提升进行了实验与分析. 詹子林[9]的“基于森林形状信任模型的 RPKI 数据并发同步机制”中, 有对 RPKI 同步系统分布式架构实验的设计, 该实验搭建了分布式同步系统架构和单点同步架构, 将两者进行对比试验. 选择了其中一个节点作为中心节点, 其他作为同步节点, 均部署在北京测试机房. 测试的数据为当前 RPKI 资料库全部数据执行增量更新. 由于分布式节点均在北京测试机房, 彼此共享带宽, 同时没有把 RPKI 数据发布点的地理位置特征考虑进去, 其性能数据可能未能完全反映理想条件下的结果. 此外该实验也只简要对算法性能做了实验与分析, 实验与分析覆盖面有限. 基于此, 本文力求在研究的基础上对实验环境与设计进一步改进优化, 以便获得更加广泛的实验结果.

为检验分布式同步架构中不同调度算法的性能差异, 本文利用国内外多个节点搭建了高性能可扩展 RPKI 数据同步试验床 (亚洲的中国北京和新加坡, 北美洲的美国加利福尼亚, 欧洲的德国法兰克福), 从而能够更加深入对分布式系统架构、任务分配调度算法等进行实验验证, 为 RPKI 的研究提供了优质的实验环境.

参与测试的服务器共 7 个节点, 节点 1 作为中心节点, 其他作为同步节点. 中心节点和 1 个同步节点部署在北京测试机房, 其他同步节点均部署在国外, 详细配置如表 2 所示. 节点的地理位置囊括了全球五大洲, 并且节点服务托管机构包含了科研机构、运营商、云服务提供商等多个类型. 对于 RPKI 分布式同步系统的高性能及其容错性等特性的验证, 该试验床的节点规模已经能够模拟实际应用的经典环境. 对试验床中节点进行充分配置, 通过对任务分配, 响应时长, 节点容错等功能表现进行观察, 可得到具有代表性的实验结果.

表 2 实验环境

位置	角色	操作系统	配置	IP地址分配区域
亚洲中国北京	中心节点	Ubuntu 18.04.6 LTS	6核Intel(R) Westmere E56xx/L56xx/X56xx 30 GB内存 300 GB 硬盘 200 Mb/s 带宽 (共享)	APNIC
亚洲中国北京	同步节点	Rocky Linux 8.10	8核Intel(R) Xeon(R) CPU E5-2640 @ 2.50 GHz 16 GB 内存 300 GB 硬盘 200 Mb/s带宽 (共享)	APNIC
亚洲新加坡	同步节点	Ubuntu 22.04 LTS	8核Intel(R) Xeon(R) CPU E5-26xx v4 16 GB 内存 300 GB 硬盘 18 Mb/s 带宽	APNIC
南美洲巴西圣保罗	同步节点	Ubuntu 22.04 LTS	8核Intel(R) Xeon(R) Platinum 8255C CPU @ 2.50 GHz 16 GB 内存 300 GB 硬盘 18 Mb/s 带宽	LACNIC
北美洲美国加利福尼亚	同步节点	Ubuntu 22.04.3 LTS	8核Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30 GHz	ARIN
欧洲德国法兰克福	同步节点	Ubuntu 22.04 LTS	8核AMD EPYC 7K62 48-Core Processor	RIPE NCC
非洲埃及开罗	同步节点	Ubuntu 22.04.03 LTS	8核Intel(R) Xeon(R) Gold 6348 CPU @ 2.60 GHz	AFRINIC

4.2 系统吞吐与节点数量

为比较分布式系统中, 分布式节点数量与总响应时长的关系, 本文设计对照实验, 反复发起全局同步任务, 并即时清空本地缓存, 在不同节点数 (控制节点数在 1–6) 的情况下对系统响应时长做了观测, 从实验结果 (图 5) 来看, 随着分布式节点的增多, 能显著缩短完整全量同步任务的响应时长. 并且同时验证了分布式架构比单体架构相比, 带来更加高效的吞吐性能优势.

4.3 系统吞吐与算法关系

本文为该系统设计了诸多调度算法, 为测试系统吞吐与算法的关系, 本文针对调度算法这一变量设计对照组实验. 即分别采用了顺序任务调度算法、随机任务调度算法、动态权重任务调度算法、网速调度算法、RIR 地理位置调度算法和指定地理位置调度算法

对全局同步任务进行分发. 具体的, 分别随机选取 10、50、80 个发布点进行多轮次的增量与全量的同步.

实验效果结果如图 6 所示, 在同一任务序列的条件下, 不同调度算法对系统的响应性能不一致. 其中, 动态调度算法在调度的过程中会根据节点性能及其节点中任务负载信息, 做出合理的节点动态选择, 可知此算法能够合理地利用机器资源, 在此算法下系统有较高的同步性能.

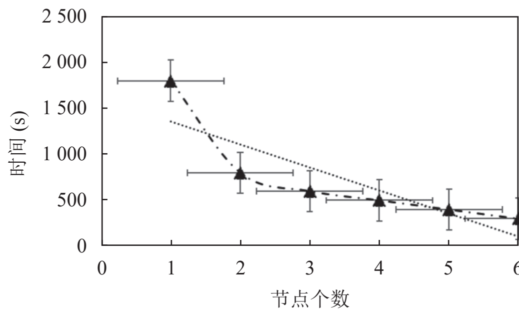

图 5　系统响应时间与节点数量

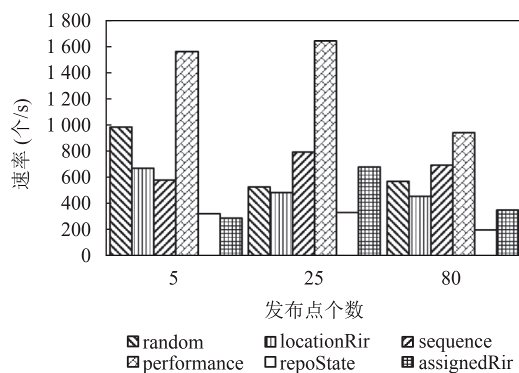

图 6　系统响应时间与调度算法

顺序调度算法也有不错的性能表现. 与动态分配相比, 顺序调度算法将任务较为平均地分摊给所有的节点, 分配的粒度是以仓库地址为最小单位来进行分配, 粒度较粗, 其性能稍弱于动态分配算法.

随机调度算法选择节点较为随机, 可知节点分配到的任务较为随机, 可能会导致性能好的节点分配到小的任务, 性能差的机器分到了大的任务, 无法充分利用机器资源, 所以性能低于顺序调度算法和动态调度算法.

网速调节算法根据实际网速来进行任务分配, 由于无法动态调整, 导致会把任务都分配给网速快的机器, 同步任务都堆积在某几台机器上, 降低了系统的资源利用率, 拉低分布式系统同步速率.

同时, 在发布点数量这一变量影响下, 响应速率也有所差异. 发布点数量越少, 系统负载的同步任务越少, 系统的响应速度越快, 当发布点数量逐渐增多时, 其响应速率也随之降低, 此处存在明显的长尾效应.

除此之外, 本文基于所有发布点的同步任务, 计算并统计了在实际同步过程中每个节点的节点利用率, 通过加权平均对系统利用率进行了计算, 如图 7 所示, 其中图注为节点 IP 地址. 从结果上看, 动态调度算法的系统利用率最高. 并且节点利用率最为平均, 该观测结果也与系统响应时间与调度算法实验结论 (图 6) 相验证, 可知动态调度算法利用节点性能进行调度选择, 对系统计算资源的利用最为合理.

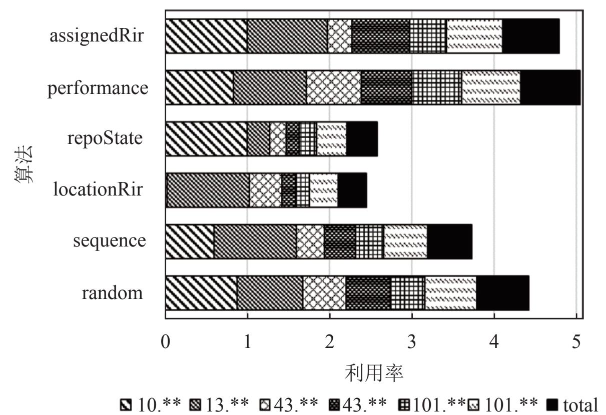

图 7　系统利用率与调度算法

4.4 系统吞吐与任务分配分析

为对比不同的任务调度序列对总的系统响应的影响, 本文控制任务调度这一变量 (FIFO 先来先服务, LJF 大作业优先, SJF 小作业优先), 做了多组全量同步重复实验, 并对响应速率做平均值处理, 便于进行量化观察. 如图 8 可知在先来先服务, 大作业优先和小作业优先 3 种基础调度策略中, 大作业优先调度策略下的系统, 其系统响应时长最短, 且同步效率最高.

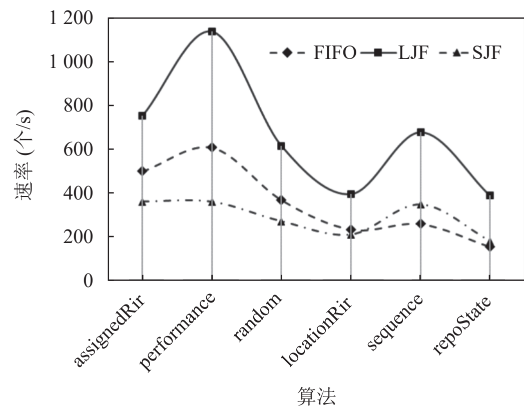

图 8　响应速率与任务分配策略

根据分析可知, 在本文所设计调度模型中, 子队列均为“多服务台”模式, 即存在多个并行单元同时对任务进行消费与处理, 因此不存在大作业阻塞导致“饿死”的情况. 大作业同步任务时间长, 是整个系统的瓶颈, 在此基础上, 优先处理大作业任务, 能加快系统整体的同步速度.

4.5 系统容错性实验

为降低节点故障带来的影响, 提高系统容错性. 系统主要采用了以下机制: (1) 心跳检测与健康检查: 通过定期对节点的心跳进行检查, 及时发现节点故障, 进行任务的重新分配; (2) 任务的冗余与共享管理: 在所有节点共享一个任务池, 当节点失效, 其余健康节点能够及时获取任务信息, 进而对任务进行接管, 保障任务的连续性; (3) 重试机制: 分布式同步系统会对失败任务进行一定次数的重试, 保证因节点失效的导致中断的任务能够重新被调度和分配; (4) 故障隔离: 发生节点故障时, 系统会自动将故障节点摘除, 避免故障节点再次进入任务流, 进而对故障进行传播.

为验证系统容错处理机制, 现设计子节点故障转移实验. 在系统同步状态中, 对分布式子节点逐步注入故障, 同时对日志打点记录, 观察节点失效时间和任务转移时间. 如图 9 关键日志可知, 注入节点故障后, 中心节点感知到节点故障, 即刻进行了任务的重分配.

```
)24/09/11 16:26:00.826  [D]  [NodeHealthMonitor] updateNodeHealth Fault count
exceeds threshold, node: &{3 13.56.160.133 node https://13.56.160.133:8075
'mem": {"memory": 16758857728, "freeMemory": 14391132160}, "net": {"mbps": 16
 "cpus": {"cpuPerformanceModels": [{"Mhz": 2300, "coresCount": 1}, {"Mhz": 2
)0, "coresCount": 1}, {"Mhz": 2300, "coresCount": 1}, {"Mhz": 2300, "coresCou
:": 1}]}, "disk": {"totalSpace": 520120602624}, "task": {"deltaOkCount": 0, "
eltaFailCount": 0, "snapshotOkCount": 0, "deltaHistoryCount": 2288, "snapshot
ilCount": 0, "deltaHistoryOkCount": 459113, "deltaInProgressCount": 0, "snap
otHistoryCount": 38, "deltaHistoryFailCount": 3089, "snapshotHistoryOkCount"
```

图 9　错误节点故障日志

根据日志信息, 提取每个节点的任务数特征, 从任务时序图 10 可知, 在发生错误时, 原有错误节点的未完成任务重分配给了其余健康节点, 健康节点的未完成任务数明显上涨, 随着重分配任务的消费, 整个系统完成收敛, 总未完成数降为 0. 从未完成任务数的转移可知, 该系统具有单节点故障转移能力.

其次, 当节点请求出现超时错误, 并且重试到阈值后, 该系统会将该任务转移到其余合适节点进行再次重试. 可知该机制能够解决部分因网络传输错误导致的任务失败问题. 但是对于文件完整性错误, 例如数据格式错误, 非网络传输导致的数据残缺等数据质量问题, 会导致依赖方系统无法正确解析数据. 出于安全考虑, 需要对这部分数据进行忽略与丢弃. 同时也就对此类错误无容错处理机制. 表 3 对机制的容错性进行了总结.

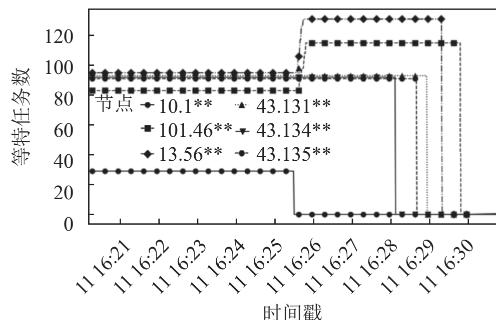

图 10　节点队列任务时序图

表 3　系统容错性

错误	容错支持	策略
单节点故障	支持	• 心跳检测与健康检查 • 任务的冗余与共享: 便于任务重分配 • 重试机制 • 故障隔离
网络传输错误	部分支持	• 重试机制: 单重试失败后, 转移就近节点进行重试
文件错误	—	—

5 结论与展望

基于相关研发工作, 本文设计并实现了一种 RPKI 依赖方分布式同步系统, 并实现了支持多节点同步下载的分布式同步结构, 并设计和采用了顺序任务调度算法、随机任务调度算法、动态权重任务调度算法、网速调度算法、RIR 地理位置调度算法和指定地理位置调度算法等多种任务调度算法, 从而使多个节点之间可以密切协同高效完成同步任务.

当前分布式同步架构尚需要持续优化, 进一步压缩同步时间, 提高同步效率. 首先, 高性能可扩展 RPKI 数据同步试验床可以继续增加更多的测试节点, 通过完善同步节点的分布, 优化下载. 其次, 调度算法还可进一步优化, 比如动态权重任务调度算法可以参考其他算法, 引入更多的其他参数, 从而进一步提高性能.

参考文献

1 White House Office of the National Cyber Director. Roadmap to enhancing internet routing security. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/09/Roadmap-to-Enhancing-Internet-Routing-Security.pdf.

[2024-09-24].
2 Kristoff J, Bush R, Kanich C, *et al*. On measuring RPKI relying parties. Proceedings of the 2020 ACM Internet Measurement Conference. 2020. 484–491.
3 Friedemann PH, Rodday N, Rodosek GD. Assessing the RPKI validator ecosystem. Proceedings of 13th International Conference on Ubiquitous and Future Networks. Barcelona: IEEE, 2022. 295–300.
4 Mirdita D, Shulman H, Vogel N, Waidner M. The CURE to vulnerabilities in RPKI validation. arXiv:2312.01872v1, 2024.
5 Lepinski M, Kent S, Kong D. A profile for route origin authorizations (ROAs). rfc-editor.org, 2012.
6 Snijders J, Madory D. The latest RPKI ROV deployment metrics. NANOG. https://nanog.org/stories/articles/rpki-rov-deployment-reaches-major-milestone/, (2024-05-01).
7 秦超逸, 张宇, 方滨兴. RPKI 去中心化安全增强技术综述. 通信学报, 2024, 45(7): 196–205.
8 马迪. 构建可扩展的 RPKI 依赖方系统部署机制. 中兴通讯技术, 2023, 29(1): 40–44.
9 詹子林. RPKI 数据同步及验证优化机制研究与实现 [硕士学位论文]. 北京: 中国科学院大学, 2023.
10 Bruijnzeels T, Muravskiy O, Weber B, *et al*. RFC 8182: The RPKI repository delta protocol (RRDP). 2017.
11 Rodday N, Cunha Í, Bush R, *et al*. The resource public key infrastructure (RPKI): A survey on measurements and future prospects. IEEE Transactions on Network and Service Management, 2024, 21(2): 2353–2373. [doi: 10.1109/TNSM.2023.3327455]
12 van Hove K, van der Ham J, van Rijswijk-Deij R. Rpkiller: Threat analysis from an RPKI relying party perspective. arXiv:2203.00993, 2022.
13 Mirdita D, Schulmann H, Vogel N, *et al*. The CURE to vulnerabilities in RPKI validation. arXiv:2312.01872, 2023.
14 Mirdita D, Schulmann H, Waidner M. SoK: An introspective analysis of RPKI security. arXiv:2408.12359, 2024.

(校对责编: 王欣欣)