软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007623] [CSTR: 32375.14.jos.007623]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于服务器无感知计算的对象存储跨云复制系统*

舒俊宜[1,2], 黄小龙[1,2], 马 郓[2], 李浩源[2,3], 黄 罡[1,2], 刘譞哲[1,2], 金 鑫[1,2], 梅 宏[1,2]

[1](北京大学 计算机学院, 北京 100871)

[2](高可信软件技术教育部重点实验室 (北京大学) , 北京 100871)

[3](北京开元维度科技有限公司, 北京 100080)

通信作者: 刘譞哲, E-mail: liuxuanzhe@pku.edu.cn

摘　要: 对象存储是云计算环境下最常用的存储方案之一. 将对象存储中的数据复制到一个或多个云区域能够提升其可靠性和可用性, 并降低用户的访问延迟. 对象存储复制系统需要保障副本间的一致性并尽可能降低复制时延. 现有的开源跨云区域对象存储复制系统 Skyplane 尽管能支持主流云平台间的对象数据复制, 但无法保障数据一致性, 且在动态负载下复制时延较高. 为了实现动态负载下的低成本、低延迟对象数据复制, 提出基于服务器无感知计算的对象存储跨云复制系统 FastRep. FastRep 通过对象粒度的复制任务锁和乐观的复制机制保障数据一致性, 通过时间有界完全树实现云函数快速调用, 通过自适应函数克隆应对带宽下降问题. 实验结果显示, FastRep 能以分钟级延迟实现大小动态变化的对象数据的低成本复制.

关键词: 跨云数据复制; 服务器无感知计算; 对象存储

中图法分类号: TP311

中文引用格式: 舒俊宜, 黄小龙, 马郓, 李浩源, 黄罡, 刘譞哲, 金鑫, 梅宏. 基于服务器无感知计算的对象存储跨云复制系统. 软件学报. http://www.jos.org.cn/1000-9825/7623.htm

英文引用格式: Shu JY, Huang XL, Ma Y, Li HY, Huang G, Liu XZ, Jin X, Mei H. Cross-cloud Replication System for Object Storage Based on Serverless Computing. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7623.htm

Cross-cloud Replication System for Object Storage Based on Serverless Computing

SHU Jun-Yi[1,2], HUANG Xiao-Long[1,2], MA Yun[2], LI Hao-Yuan[2,3], HUANG Gang[1,2], LIU Xuan-Zhe[1,2], JIN Xin[1,2], MEI Hong[1,2]

[1](School of Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

[3](Alluxio Inc, Beijing 100080, China)

Abstract: Object storage is one of the most widely used storage solutions in cloud computing environments. Replicating data stored in object storage across one or more cloud regions improves reliability and availability while reducing access latency for users. An object storage replication system is required to ensure consistency among replicas while minimizing replication latency. Existing open-source cross-cloud object storage replication systems, such as Skyplane, support object data replication across major cloud platforms but do not guarantee data consistency and exhibit high replication latency under dynamic workloads. To enable low-cost and low-latency object data replication under dynamic workloads, this study proposes a cross-cloud object storage replication system, termed FastRep, based on serverless computing. Data consistency is ensured through object-grained replication task locks and an optimistic replication mechanism. Cloud function invocation latency is reduced via a time-bounded complete tree, while bandwidth degradation is mitigated through adaptive function cloning. Experimental results demonstrate that FastRep achieves low-cost replication of dynamically sized object data with latency at the minute level.

Key words: cross-cloud data replication; serverless computing; object storage

---

* 基金项目: 国家重点研发计划 (2022YFB4500700); 国家自然科学基金 (62325201, 62172008); 中央高校青年教师科研创新能力支持项目 (ZYGXQNJSKYCXNLZCXM-I1)
收稿时间: 2025-06-20; 修改时间: 2025-10-07; 采用时间: 2025-12-29; jos 在线出版时间: 2026-05-20

跨云区域 (cloud region) 的数据复制对于确保数据可用性、可靠性以及降低云应用访问延迟至关重要. 尽管云计算厂商通过在一个云区域内建设多个数据中心并对数据进行冗余存储提升了单云区域存储的可靠性和可用性, 但影响整个云区域的技术故障仍然时有发生. 例如, 阿里云在 2023 年 11 月出现了两次造成多个服务不可用的技术故障[1], 而谷歌云在 2024 年 4 月的一次技术故障导致了基金公司 UniSuper 的数据丢失[2]. 通过在多个不同云区域或云平台存储数据备份, 云用户能够有效防范单个云区域发生故障的风险. 此外, 将数据复制到靠近最终用户的地理位置也能有效降低云应用加载数据的延迟, 提升用户体验.

据多家分析机构估算, 非结构化数据已经占据了企业数据总量的 80%–90%[3]. 亚马逊 S3[4]、微软 Azure Blob Storage[5]等对象存储 (object storage) 常用于存储海量的非结构化数据, 在云计算环境中发挥着重要作用. 以亚马逊 S3 为例, S3 可以支持从 0 byte 到 5 TB 之间任意大小的对象的存储. Satija 等人[6]通过对亚马逊 AWS 上的云架构的分析指出, 对象存储在 68% 的云架构中得到使用, 是云用户最常使用的云计算服务.

然而, 现有对象存储复制方案存在较大的局限. 云计算厂商自身提供的跨云区域对象存储复制服务仅支持在一个云平台内部的不同云区域间进行复制. 同时, 云用户在使用这些服务时还必须开启多版本存储的功能, 从而保障数据的最终一致性, 导致产生大量额外的存储费用. 为了支持在不同云平台的云区域间复制对象数据, Jain 等人[7]构建了开源对象存储复制系统 Skyplane. Skyplane 在不同云平台的云区域中启动虚拟机, 将数据从起点云区域的对象存储中下载下来, 并传输到终点云区域的对象存储中去, 由于虚拟机的启动通常需要至少数十秒, 对于较小的对象数据而言, Skyplane 的复制延迟和成本仍然较高.

服务器无感知计算 (serverless computing)[8]因此成为动态大小对象复制的理想选择. 云函数启动时间通常不足 1 s, 使得较短时间内可以完成的任务不会因为较高的启动时间导致延迟大幅上升. 本文通过实验验证了将服务器无感知计算用于跨云区域对象数据复制的可行性. 总的数据传输速率随函数数量线性增长, 聚合带宽可达数百 Gb/s. 但实验结果也展示了使用服务器无感知计算的两点挑战: 一方面, 并行复制大对象时, 函数启动时间可能成为瓶颈; 另一方面, 函数传输速率存在波动, 系统难以预判带宽何时下降并提前应对.

本文提出了一种低成本且能在分钟级完成复制的对象存储复制系统 FastRep. FastRep 的设计具有 3 点优势: 第一, FastRep 能够对不同大小的对象数据均实现分钟级复制时间; 第二, FastRep 能够实现完全的按需付费, 无复制任务时不产生费用; 第三, FastRep 支持在多个云平台间进行复制且集成简便. 针对基于服务器无感知计算进行数据复制的固有挑战, 系统将复制过程分解为函数调用与数据传输两阶段. 在函数调用阶段, FastRep 通过构建时间有界完全树优化函数调用关系; 在数据传输阶段, FastRep 采用自适应函数克隆机制应对带宽波动.

本文实现了 FastRep 的原型系统, 并在亚马逊 AWS、微软 Azure 和谷歌 GCP 等云平台进行了实验验证. 结果表明, 相较 Skyplane, FastRep 将复制时间最高降低两个数量级, 能够在 1 min 以内完成 100 GB 对象数据的复制, 并将复制成本降低最多 3 个数量级.

本文第 1 节介绍跨云区域对象存储复制的研究背景和相关工作. 第 2 节介绍使用服务器无感知计算传输数据的可行性和挑战. 第 3 节介绍本文设计与实现的基于服务器无感知计算的对象存储跨云复制系统 FastRep. 第 4 节通过实验验证 FastRep 的有效性. 最后总结全文.

1 研究背景和相关工作

1.1 对象存储

对象存储是云计算环境下常用的存储方案, 常用于存储和管理文档、图片、视频等非结构化数据. 与传统的文件系统存储和块存储不同, 对象存储以对象为粒度对数据进行管理. 亚马逊 S3[4]、微软 Azure Blob Storage[5]、谷歌云 Cloud Storage[9]、阿里云 OSS[10]均属于对象存储.

与传统存储系统相比, 对象存储具有独特的读写特性. 首先, 对象存储提供了简单的 PUT 和 DELETE 写入接口. 对象一旦创建, 不能局部修改. 如需变更, 就必须创建一个新版本覆盖原有对象. 其次, 对象存储的 GET 接口更为灵活, 允许用户从指定偏移量读取对象的连续片段, 从而避免不必要的冗余读取, 还能支持对大对象的并行读

取. 第三, 对象存储支持将大对象分割为更小的部分并行上传. 这种方法通过将大文件分块上传, 提高了上传效率和可靠性. 另一方面, 对象数据的大小最小可为 0 字节, 最大则可以到达数 TB. 根据 IBM 开放的 Cloud Object Storage (IBM COS) 请求日志数据集, 约 80% 的 PUT 请求的对象大小低于 1 MB[11]. 虽然其他主要云服务没有公开具体数据, 但根据公开信息进行估测可以得出类似的结论[12].

1.2 相关工作

数据中心间的数据复制是一个经典的研究问题. 针对数据中心间数据复制的技术方案基于传统的广域网数据复制技术, 叠加考虑了数据中心间网络的多种特性. 随着云计算被广泛采用, 过去的许多系统假设都在云计算的范式下发生了新的变化. 这些变化主要体现在两个方面: 首先, 从用户视角来看, 传统数据中心用户的可用资源是相对固定的, 而云用户视角下的资源则是可以近乎无限扩展的; 其次, 从计费模式上来看, 与商用和家用有线网络按带宽而非实际用量计费不同, 云计算采用了与移动网络类似的按实际使用流量进行计费的模式. 同时, 云计算厂商对不同的网络流量采取了差异化的定价方式. 以亚马逊 AWS 为例, 一个云区域内的多个数据中心间流量是完全免费的, 而不同云区域之间的数据传输价格差异极大, 从美国北弗吉尼亚向其他云区域传输仅需 2 美分/GB, 而从日本大阪向其他云区域传输则需要 9 美分/GB.

近年来工业界和学术界针对云计算的特性对数据复制进行了进一步的优化. 云计算厂商提供了 DataSync[13]、Azure AzCopy[14]等数据复制服务. 尽管这些服务能够让用户在特定云计算平台的云区域之间互相传输数据以及从其他数据来源向云上迁移数据, 但是用户无法通过这些服务自由地向其他云平台复制数据. Skyplane[7]是一个开源的跨云和跨云区域数据复制工具, 目前已支持了亚马逊 AWS、微软 Azure、谷歌 GCP 等主流云平台间的数据复制. Skyplane 根据对象的传输起始、目标位置以及对象大小, 规划出所需虚拟机数量和数据传输路径, 从而达到在目标时间内最小化传输成本或在目标预算内最大化传输带宽的目的. Skyplane 针对单个虚拟机带宽受到云计算厂商限制的特性, 通过启动多个虚拟机的方式提升了单个云区域内的可用带宽. Cloudcast[15]在 Skyplane 的基础上, 进一步将问题从单播的场景泛化到多播的场景. 由于多播问题的复杂度远超单播问题, Cloudcast 采用多种优化手段去降低求解时间. Cloudcast 将传输成本和性能相近的节点进行合并, 并将转发次数限制在 2 跳以内. Cloudcast 将数据切分为若干条带, 每个条带的路径独立求解, 使得规划时间相对于数据量仅线性增加. CloudMPCast[16]与 Cloudcast 的目标和解决方法相近, 但没有充分利用云计算的资源弹性. SPANStore[17]不仅关注数据复制, 还搭建了一个跨越多个云计算平台的对象存储. SPANStore 通过将数据存放在多个云区域, 降低了数据的访问延迟. 同时, 通过感知价格, SPANStore 可以选择数据传输和存储成本最低的方案. SPANStore 支持数据在多个云区域间保持一致, 并且可以容忍各类错误. Park 等人[18]提出的 Macaron 则采取了以存代传的思路, 针对高频跨云区域访问的数据, 利用目标云区域的多级缓存进行加速和成本削减. SkyStore[19]和 Alluxio[20]等系统也采用了与 Macaron 相似的思路. 除了通过网络进行传输外, 云计算厂商针对超大规模的数据迁移, 还提供了将数据存入存储器再物理运输到目标位置的方式[21,22]. 在数据量极大的情况下, 这种方式相比网络传输反而可以更快地完成复制任务.

2 使用服务器无感知计算传输数据的可行性和挑战

考虑到对象存储的大小分布不一, 较小对象对启动时间敏感, AWS Lambda[23]、Azure Functions[24]、Google Cloud Run Functions (后文简称为 Cloud Run)[25]等服务器无感知计算服务成为虚拟机的合理替代方案. 作为新兴的云计算范式, 服务器无感知计算将底层计算资源交由云计算厂商管理, 仅向用户暴露云函数接口. 采用云函数实现对象复制具有多重优势: 相较于虚拟机, 云函数的启动时间经过大幅优化[26−31], 能显著降低小对象复制的额外时延; 其按照更细粒度付费的模式也可带来显著的成本节约. 然而, 尽管已有较多的工作对云函数的性能进行了建模[32−37], 云函数在数据传输任务中的表现仍待探究. 为此, 本文通过实验验证了其可行性, 并分析其带来的新挑战.

2.1 可行性分析

首先, 本文验证单个云函数能提供足够的传输带宽. 与仅有数种固定配置的虚拟机不同, 用户可以更加灵活地调整云函数的 vCPU 数量和内存大小. 本文测试了 AWS Lambda 在不同内存配置下进行同一云区域内下载 (us-

esat-1)、向云内同一大洲的云区域上传 (us-east-1 到 ca-central-1)、向云内其他大洲的云区域上传 (us-east-1 到 eu-west-1)、跨云上传 (us-east-1 到谷歌 GCP europe-west6) 这 4 种不同情况的传输速率, 如图 1 所示. 我们从实验结果中可以得出 3 个结论: 第一, 云区域内下载以及向同一云平台内相近云区域上传的速率可达或略超 600 Mb/s, 这意味着 70 MB 或更小的对象可以在 1 s 内完成数据传输; 第二, 向同一云平台内较远云区域或跨云上传的速率显著下降, 仅能达到 200 Mb/s; 第三, AWS Lambda 在 512 MB 内存以下呈现线性性能提升, 并在 1–2 GB 区间达到峰值, 而使用谷歌云 Python 客户端跨云向谷歌云上传至少需要 512 MB 内存才能正常运行. 这意味着 512 MB 内存下, 云函数能够正确执行且具有较好的性价比.

随后, 由于单个云函数传输速率无法满足在较短时间内复制较大对象的需要, 本文进一步验证云函数传输带宽的可扩展性. 仍然针对同一云区域内下载 (us-esat-1)、向云内同一大洲的云区域上传 (us-east-1 到 ca-central-1)、向云内其他大洲的云区域上传 (us-east-1 到 eu-west-1)、跨云上传 (us-east-1 到谷歌 GCP europe-west6) 这 4 种不同情况, 本文同时启动了多个云函数执行相同的传输任务, 观察总传输带宽随并发云函数数量增加的变化关系. 如图 2 所示, 我们从实验结果中可以得出两个结论: 第一, 在 4 种情况下, 聚合的总传输带宽都可以随函数数量增加呈现接近线性增长的趋势; 第二, 即使是较慢的跨云上传链路, 在 512 个云函数并发的情况下, 也可实现超过 100 Gb/s 总传输带宽, 从而可以在理想情况下实现 GB 级数据的秒级上传.

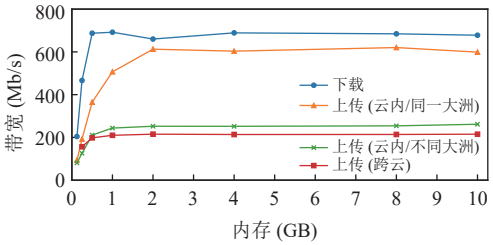


图 1 单个 AWS Lambda 云函数的传输带宽

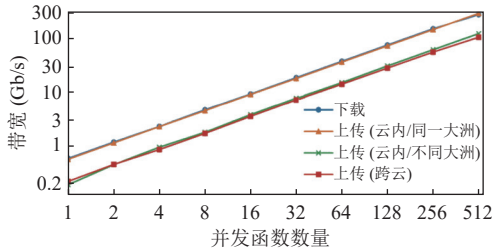


图 2 多个 AWS Lambda 云函数的总传输带宽

2.2 挑战分析

在验证了使用云函数传输数据的可行性后, 本文也发现了两个新的挑战.

首先, 尽管更多的并发云函数能够带来更高的聚合带宽, 但是在实际的数据复制场景下, 启动更多的云函数却可能反而导致复制性能下降. 本文进行了一组将 8 GB 大小的对象分块复制的实验. 如图 3 所示, 当启动的函数数量从 2 个增加到 16 个, 复制时间随函数数量增加而逐渐降低. 而当函数数量超过 32 个时, 复制时间反而随函数数量增加而大幅上升. 我们通过云函数的生命周期分析发现, 云函数每次从调用到开始传输数据会有数百毫秒的响应延迟, 因此通过一个云函数顺序启动其他云函数可以达到的有效并发函数数量是有限的, 过多的顺序启动反而会带来额外的开销. 因此本文的第 1 个挑战在于如何快速启动大量的云函数.

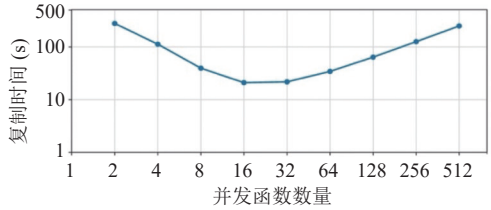


图 3 启动多个云函数复制 8 GB 对象的延迟

其次, 尽管前文验证了单个云函数的传输带宽, 但其随时间变化是否能保持稳定仍然是未知的. 为了探究云函数传输带宽随时间变化情况, 本文从 AWS 的 us-east-1 云区域对多个传输链路进行了连续 24 h 的监控. 如图 4 所示, 与虚拟机能够在较长一段时间内保持传输速率稳定[7]不同, 云函数在多个下载和上传链路上都出现了相对最

大带宽超过 40% 的性能下降, 且性能下降出现的时间并未呈现出显著的规律. 因此, 本文的第 2 个挑战是如何在云函数带宽骤然下降时保持较为稳定的复制时延.

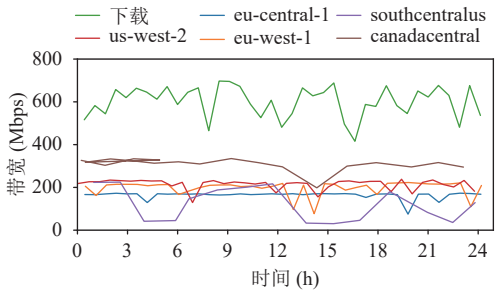


图 4　24 h 内的云函数传输带宽变化

3　基于服务器无感知计算的对象存储跨云复制系统

3.1　系统架构

本文提出了基于服务器无感知计算的对象存储跨云复制系统 FastRep. 通过采用服务器无感知计算的方法复制对象数据以及针对云函数特点加速函数启动和数据传输, FastRep 有效降低了各类不同大小对象数据的跨云区域复制时间和成本. 如图 5 所示, FastRep 是一个完全依赖于服务器无感知计算和云数据库按需付费的系统, 其由 4 部分组成: 复制事件响应模块、函数调用规划模块、复制模块和日志记录模块.

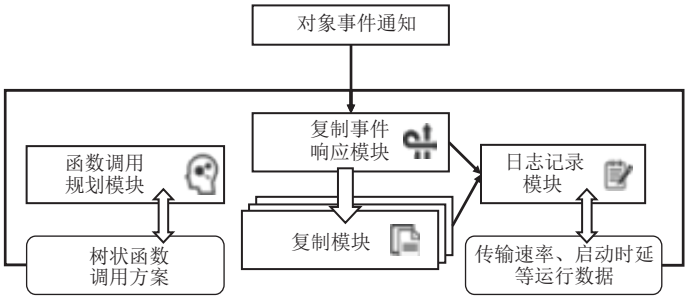


图 5　FastRep 系统架构

● 复制事件响应模块: 当对象存储发生 PUT 或 DELETE 等写入操作时, 云平台会生成通知用于更新后处理. 复制事件响应模块在接收到通知后, 基于系统对复制时间的建模和云数据库记录的数据传输速率决定并行度, 以满足配置的服务质量目标 (service level objective, SLO).

● 函数调用规划模块: 函数调用规划模块离线生成最优的函数调用方案. 虽然该模块可以被集成到复制事件响应模块中在线执行, 但由于函数的调用方案在函数数量不变时总是相同的, 因此可以将该模块进行剥离采用离线处理的方式降低在线部分的响应时延. FastRep 设计了时间有界完全树, 以 $O(n\log n)$ 的时间复杂度生成方案并存储于云数据库中供复制事件响应模块查询.

● 复制模块: 复制模块负责在数据复制任务的发起云区域和目标云区域之间执行实际的数据复制操作. 该模块从发起复制的存储桶中下载对象或对象的一部分内容, 并将其上传至目标存储桶. 当云函数传输速率下降时, 复制模块通过自适应的函数克隆以追赶进度.

● 日志记录模块: 复制事件响应模块与函数调用规划模块依赖传输速率和函数调用时间优化复制时间. 由于传输速率和函数调用时间等关键性能参数可能会随时间波动, 日志记录模块持续跟踪代表性任务的复制时间. 这些数据用于定期更新分析模型的参数, 以确保分析模型保持相对准确.

3.2 数据一致性保障

现有的跨云区域对象复制系统[7]通常假设复制的对象是静态的, 并且在复制时也不存在并发的复制任务. 云平台提供的跨云区域存储桶复制功能则依赖于对象版本控制. 启用了版本控制后, 每个版本的对象都是静态的, 彼此之间不会相互干扰. 尽管启用版本控制可以确保复制的最终一致性 (eventual consistency), 但也会带来显著的存储成本. 例如, 如果每个对象每天更新一次, 启用版本控制至少会使存储成本翻倍, 因为版本的生命周期规则是按天的粒度管理的, 旧版本的对象至少要等待一天才能被删除. 因此, 本文的目标是在不增加显著的额外成本或改变用户行为的情况下实现最终一致性.

● 未启用版本控制时的一致性问题: 在未启用版本控制的情况下, 有两种并发情况可能会破坏最终一致性. 首先, 在图 6 中, 本文展示了起点存储桶中对同一对象的两个 PUT 操作可能会触发对终点存储桶中同一对象的并发 PUT 操作. 由于对象存储在处理同一对象的并发写入时没有确定的行为模式, 因此可能出现两种可能的结果. 一种情况是两次 PUT 请求均成功, 最终对象可能来自其中一次 PUT 操作. 另一种情况是其中一个 PUT 请求失败或是两个 PUT 请求都失败. 因此, 为了确保一致性, 系统必须避免并发复制任务. 其次, 图 7 展示了当一个大对象被拆分成多个部分并由多个云函数并行复制时, 如果在复制过程中发生了另一个 PUT 操作, 终点存储桶中的对象最终可能会由不一致的多个数据块合并而成.

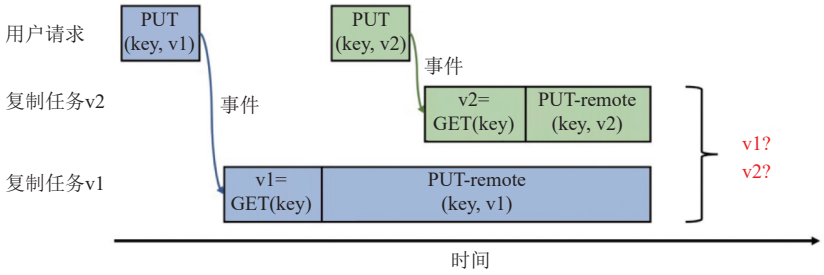


图 6 同一对象不同版本的并发复制导致未定义行为

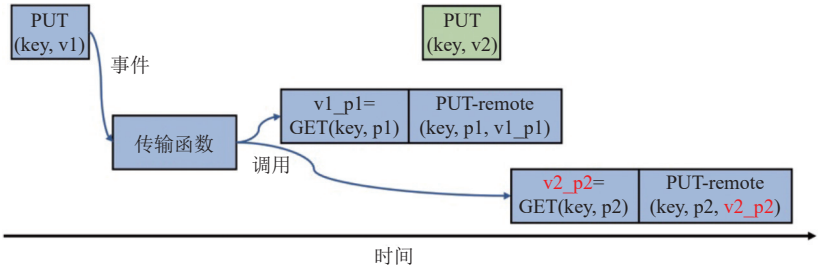


图 7 多个复制函数复制的版本不一致

● 对象粒度的复制任务锁: 为防止多个复制任务同时执行导致并发 PUT 操作, FastRep 引入了一个分布式锁来确保复制任务的串行化, 分布式锁可以通过云数据库实现. 算法 1 展示了复制任务锁的加锁和解锁机制. 当存在正在进行的复制任务时, FastRep 在每次复制任务中跟踪最新版本的 ETag (ETag 基本等同于对象的内容哈希值, 由平台自动生成). 在复制任务结束释放锁之前, FastRep 会将待处理的 ETag 与最近复制的 ETag 进行比较. 如果两者不匹配, 说明最新版本尚未复制, 此时 FastRep 将再次调用复制策略规划函数以确保最新的版本得到复制.

算法 1. 复制任务锁.

1. **function** *LOCK*(*obj_key*, *etag*, *seq*)
2. *lock* ← *try_lock*(*obj_key*)
3. **if** *lock.status* = success **then**

```
4.      return true
5.   else if lock.seq = NULL or lock.seq < seq then
6.      lock.seq ← seq; lock.etag ← etag
7.      return false
8. function UNLOCK(obj_key, lock)
9.   etag ← lock.etag; seq ← lock.seq
10.  release_lock(lock)
11.  if etag ≠ NULL then
12.     cur_etag ← get_etag(obj_key)
13.     if etag ≠ cur_etag then
14.        event_notify(obj_key, etag, seq)
```

● 乐观的复制与校验机制: 为避免在一个复制任务内复制了不一致的多个部分, 复制模块会验证当前 ETag 是否与复制策略规划模块提供的 ETag 匹配. 如果发现不匹配, 当前的复制任务将被中止. 本文认为, 这种中止情况发生的频率在现实负载下是较低的. 除非遇到极其频繁更新的大对象, 重试操作通常能够成功完成复制. 在极端情况下, 这类对象本身必须被加锁, 以确保 FastRep 能够成功复制它们.

3.3 复制时间建模

复制任务的瓶颈通常由两个主要因素决定: 函数调用时间和数据传输时间. 因此, 假设 T_f 是从通知触发调度器到最后一个复制函数准备好进行数据传输的时间间隔, T_t 是从第 1 个字节开始传输到整个复制任务完成的时间间隔, 则复制时间的上限 T_{total} 为:

$$T_{\text{total}} \leqslant T_f + T_t \tag{1}$$

为了简化问题表示, 本文假设复制任务由两个不重叠的阶段组成: 函数调用阶段和数据传输阶段. 上述不等式变为等式, 从而可以分而治之地解决问题.

函数调用时间优化问题可以表示为公式 (2). 对于给定数量的函数 n, E_i 是从调度器被触发到函数 i 准备好进行数据传输的时间. FastRep 的目标是:

$$Minimize \max\{E_i,\ i = 1, 2, \ldots, n\} \tag{2}$$

其中, E_i 由调用函数 i 的时间以及函数 i 需要调用多少其他函数两个因素决定. 因此, 准备估算 E_i 需要考虑这两个关键参数. 首先, 每次函数调用接口需要数十毫秒来完成. 这里定义接口调用时间为 I. 其次, 在成功响应返回后, 函数将在数百毫秒的延迟后启动. 延迟时间定义为 D. 如果将函数 i 的实际启动时间表示为 S_i, 并且函数 k 是函数 j 启动的第 m 个函数, 那么:

$$S_k = S_j + I \times m + D \tag{3}$$

$$S_1 = 0 \tag{4}$$

如果函数 i 又启动了另外 N_i 个函数, 那么:

$$E_i = S_i + I \times N_i \tag{5}$$

当所有的复制函数准备好进行数据传输时, 假设它们具有相同的传输速率, 每个函数应当获得相等的对象传输份额. 对于每个复制函数, 它首先从起点存储桶下载相应的部分, 然后将这些部分上传到终点的存储桶. 假设给定对象的大小为 L, 则每个函数需要传输的数据总量为 L/n. 这里将从起点存储桶下载的速率表示为 R_d, 将向终点存储桶的上传速率表示为 R_u, 则传输时间为:

$$T_t = \frac{(R_d + R_u) \times L/n}{R_d \times R_u} \tag{6}$$

为了满足不同的延迟目标, 调度器可以通过选择合适的 n 值来调整数据传输时间. 同时, 调度器需要保证每个函数至少处理一个完整的数据单元以避免额外的开销.

3.4 基于时间有界完全树的函数调用优化算法

虽然增加内存可以降低函数调用时间, 但当所有复制函数由一个单独的函数触发时, 复制任务仍然可能在函数调用上出现瓶颈. FastRep 的目标是找到一个不依赖于云函数的配置最优方案. 假设函数调用接口时间 I 是一个时间单位, 在给定配置下函数启动延迟 D 是 I 的倍数. 函数调用规划本质上是一个给定总节点数的树构建问题. 函数调用树的根节点是监听对象事件的入口函数, 完成全部复制函数的调用后, 该函数也可以转换成为复制函数. 树的每个边表示一次函数调用, 其中子节点代表由父节点触发的函数. 每个节点的子节点按调用顺序严格排序. 对于每个拓扑结构, 其中每个节点都有一个对应的时间, 树的时间是所有节点时间的最大值.

构建给定节点数的所有可能的树并找到时间最短的树是一个非确定性多项式困难问题 (NP-hard problem). 即使可以通过剪枝和动态规划来减少搜索时间, 仍然难以在合理时间内计算出数百个节点的最优树拓扑. 为了在 $O(n\log n)$ 时间内生成最优解, 本文定义了一种新的树结构——时间有界完全树, 并基于这一数据结构提出了一个最优的树生成算法. 时间有界完全树的定义为, 对于给定时间 B, 时间有界完全树是一个时间不超过 B 的树, 且无法再对这棵树增加节点使其时间仍不超过 B. 图 8 展示了时间约束 $B=10$ 和启动延迟 $D=3$ 时的时间有界完全树示例. 所有黄色节点 (即最右侧的叶节点) 的时间均为 10.

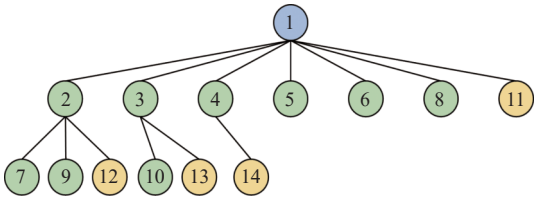


图 8 B=10, D=3 的时间有界完全树

基于时间有界完全树, 本文提出了一种能够为任何节点数 n 生成最优树拓扑的贪心算法. 假设时间约束为 B 的时间有界完全树 $Tree_B$ 有 n_B 个节点, 如果 $n=n_B$, 则 $Tree_B$ 就是时间约束 B 下的最优树拓扑. 接着假设时间约束为 $B+1$ 的时间有界完全树 $Tree_{B+1}$ 有 n_{B+1} 个节点. 如果 $n_B<n<n_{B+1}$, 则可以将 $Tree_{B+1}$ 中存在而 $Tree_B$ 中不存在的 $n-n_B$ 个节点添加到 $Tree_B$ 中从而构建一个时间为 $B+1$ 的树. 算法 2 展示了其工作原理.

算法 2. 函数启动树构建.

```
1. function GENERATE_TREE(n, tree)
2.     B ← 0; num_nodes ← 1
3.     while num_nodes < n do
4.         candidate_parents ← ∅
5.     for node ∈ tree do
6.         if time(node) + D = B then
7.             candidate_parents.append(node)
8.     B += 1
9.     for parent ∈ candidate_parents do
10.        add_child(parent)
11.        num_nodes += 1
12.        if num_nodes = n then
13.            break
```

在算法 2 的第 4–7 行需要找出所有可以添加子节点的位置 (如图 8 中的节点 1–4). 寻找所有这些位置需要 $O(n)$ 的时间. 对于每个时间约束 B, 可以添加的额外节点数量为:

$$1+\sum_{i=2}\left(\begin{array}{c}B-i\times D-1\\ i-1\end{array}\right) \tag{7}$$

因此, 当 D 是常数时, 时间有界完全树中的节点数随 B 指数级增长. 算法 2 会通过 $\mathrm{O}(\log n)$ 次时间有界完全树构建找到节点数为 n 的解, 该算法的总体时间复杂度是 $\mathrm{O}(n\log n)$.

接下来, 本文将证明该算法生成的树对于给定节点数 n 始终是最优的. 首先, 本文将证明最优函数调用时间是节点数 n 的单调递增函数.

假设最优时间不是单调递增的, 则必有:

$$T_{\mathrm{opt}}(i+1)<T_{\mathrm{opt}}(i) \tag{8}$$

由于树的时间被定义为树中所有节点的最大时间, 当从树中移除节点时, 树的时间只可能保持不变或变小. 因此, 一定存在另一个节点数为 i 的树, 其时间 $T'(i)$ 满足:

$$T'(i)\leqslant T_{\mathrm{opt}}(i+1)<T_{\mathrm{opt}}(i) \tag{9}$$

这与 $T_{\mathrm{opt}}(i)$ 最优的假设矛盾.

接着, 本文证明对于给定的调用延迟 D 和时间约束 B, 时间有界完全树是确定的. 通过逐层去构建树, 我们可以证明这一点. 对于深度 1, 时间有界完全树必定有 $B-D$ 个节点. 如果在深度 1 有超过 $B-D$ 个节点, 则显然超过了时间约束 B. 如果深度 1 的节点数少于 $B-D$ 个, 则可以在时间约束内添加更多节点.

由于深度 1 的拓扑是固定的, 所有潜在的深度 2 节点的时间也是确定的. 因此, 深度 2 的拓扑对于任何时间有界完全树都是固定的. 以此类推, 直到最大深度. 因此, 时间有界完全树是确定的.

最后, 基于上述引理, 本文可以证明算法 2 生成的树对于给定的节点数 n 始终是最优的. 本文通过数学归纳法证明了该定理. 当 $n=1$ 时, 算法生成一棵只有一个节点的树, 这是唯一可能的拓扑, 因此是最优的. 接下来, 假设算法为 $n=k$ 的情况生成了一棵最优树, 我们将尝试证明算法为 $n=k+1$ 生成的树也是最优的.

情况 1. 假设当 $n=k$ 时, 生成的树不是一棵时间有界完全树. 那么算法将会添加一个节点, 其时间与树中的其他某些节点的时间会完全相同. 设 $T_{\mathrm{tree}}(i)$ 表示树 $Tree_i$ 的时间, $T_{\mathrm{node}}(i)$ 表示节点 $Node_i$ 的时间, 则有:

$$T_{\mathrm{tree}}(k+1)=\max\{T_{\mathrm{tree}}(k),T_{\mathrm{node}}(k+1)\}=\max\{\max\{T_{\mathrm{node}}(i),i\leqslant k\},T_{\mathrm{node}}(k+1)\} \tag{10}$$

$$\exists j\in\{1,2,\ldots,k\},\ T_{\mathrm{node}}(k+1)=T_{\mathrm{node}}(j) \tag{11}$$

因此, $k+1$ 个节点的树的时间为 $T_{\mathrm{tree}}(k+1)=T_{\mathrm{tree}}(k)$.

情况 2. 接下来假设当 $n=k$ 时, 算法生成的树是一棵时间有界完全树. 添加一个节点会导致树的时间变为 $T_{\mathrm{tree}}(k+1)=T_{\mathrm{tree}}(k)+1$. 如果这棵树不是最优的, 那么必有一棵 $k+1$ 个节点的最优树, 其时间为 $T'_{\mathrm{tree}}(k+1)=T_{\mathrm{tree}}(k)$. 因为 $T_{\mathrm{tree}}(k)$ 是 k 个节点的最优时间, 并且在添加节点时时间只会保持不变或增加, 所以必须存在一棵不是时间有界完全树的最优树 $Tree'_k$. 根据时间有界完全树的定义, 可以向 $Tree'_k$ 添加更多节点, 使其成为时间约束为 $T_{\mathrm{tree}}(k)$ 的时间有界完全树, 其节点数为 $k+\delta$, 与前述引理矛盾.

3.5 基于自适应函数克隆的传输时间优化机制

接下来, 本文继续优化传输阶段的时间. 影响传输时间的关键参数是函数的内存配置以及函数并行度. 在图 1 中, 我们可以观察到超过 512 MB 内存配置后, 数据传输速率的边际效益递减. 为了更好的成本效益, 本文使亚马逊 AWS 所有函数使用固定的 512 MB 内存配置, 并依赖函数并行来提供足够的整体传输速率. 数据传输时间是对象内容长度 L、下载速率 R_d、上传速率 R_u 和函数实例数量 n 的函数. 当入口函数收到对象事件通知时, L 可以从对象的元数据中获取, R_d 和 R_u 则可以从云数据库中存储的历史日志信息中查询. 复制事件响应模块会选择 n 的值以满足以下约束条件: 第一, n 的数量不超过数据块的总数, 使得每个函数至少接收一个数据块; 第二, 复制时间在目标的时间范围内.

然而, 日志模块收集到的传输速率并不能完全准确表示运行时的传输速率. 如图 4 所示, 云函数的带宽可能会出现显著下降. 为应对可能的带宽下降, 一个合理的假设是不管每个单独函数的性能如何变化, 总带宽仍然可以随着函数数量的增加而增加. 本文采用自适应函数克隆的方式来克服函数实例的性能差异. 每当一个复制函数过载时, 它会创建更多的副本来协助复制. FastRep 将数据划分成多个数据块, 并在传输时跟踪进度. 每当它落后于预定

完成度时, 复制函数会主动触发更多的函数, 并让它们并行传输剩余的数据块以赶上进度. 需要注意的是, 函数调用存在延迟. 因此, 过于激进地应用自适应函数克隆有时会增加复制时间. FastRep 会给每个函数足够的时间缓冲, 尽可能保持在目标复制时间内, 只有当传输速率出现严重偏差时, 才将自适应函数克隆作为最后的手段. 在图 9 中, 复制函数发现进度落后并异步调用两个额外的函数来负责传输第 3 和 4 个数据块.

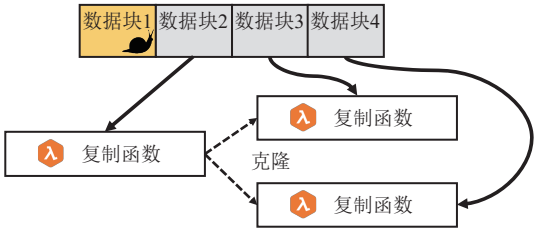


图 9 自适应函数克隆机制

在复制任务开始前, FastRep 会根据要传输的数据块总数 N_{total} 和目标完成时间 T_{target}, 为每个复制函数制定一个预期的进度规划 $P_{\text{expected}}(t)$. 在任意时间点, 预期的已完成数据块比例为 L.

$$P_{\text{expected}}(t) = \frac{t}{T_{\text{target}}} \tag{12}$$

每个复制函数在执行过程中会持续追踪其实际完成的数据块数量 $N_{\text{actual}}(t)$, 从而计算出实际进度 $P_{\text{actual}}(t)$:

$$P_{\text{actual}}(t) = \frac{N_{\text{actual}}(t)}{N_{\text{total}}} \tag{13}$$

系统通过引入一个可配置的性能偏差阈值 δ 来量化感知进度滞后的严重程度. 当一个函数的实际进度与预期进度的差距超过该阈值时, 系统便会触发克隆决策:

$$P_{\text{expected}}(t) - P_{\text{actual}}(t) > \delta \tag{14}$$

4 实验验证

本节从 2 个方面评估 FastRep 原型系统的效果: (1) 相较于开源对象存储复制系统和云计算厂商专有服务的复制时间和成本; (2) 函数调用优化算法和传输时间优化机制的有效性.

4.1 实验设置

● 实验平台. 本文分别在亚马逊 AWS、微软 Azure、谷歌 GCP 上进行了实验. FastRep 是完全服务器无感知的, 不使用虚拟机等需要预配置的资源. AWS Lambda 的内存大小默认配置为 512 MB. Azure Functions 内存大小仅有 2 048 MB 和 4 096 MB 两种默认配置, 实验默认使用 2 048 MB 的内存配置. 在谷歌 GCP 上, 因为 Cloud Run 在 512 MB 内存配置下偶尔会遇到内存不足的情况, 因此 Cloud Run 的内存默认配置为 1 024 MB, vCPU 默认配置为 1 个. 导致内存不足的一个可能原因是 Cloud Run 会将文件系统挂载到其内存空间中.

● 比较基线. 本文将 FastRep 与开源的跨云地域复制工具 Skyplane v0.3.2 版本[38]和 AWS 提供的专有复制服务 S3 RTC (S3 replication time control)[39]进行了比较. Skyplane 借助虚拟机在多个云区域之间进行对象复制. 本文没有将微软 Azure 的跨区域复制[40]和谷歌的 Turbo Replication[41]作为基线进行比较. 微软 Azure 没有提供任何时延保障, 特别是在大对象复制时要比 S3 RTC 时延显著更高. 谷歌云的 Turbo Replication 无法实现任意两个存储桶之间的复制, 这使得本文无法测量其复制时间.

● 评估指标. 本文主要关注的评估指标是复制时间和成本. 为了公平比较各个系统, 本文定义复制时间为从 PUT 请求完成到成功在终点存储桶中查询到该版本之间的时间. Skyplane 的虚拟机关机时间不计入在内. 复制的成本则根据相应云产品的定价和日志中的使用量进行估算.

4.2 复制时间和成本

本文首先比较了 FastRep 与基线系统在不同对象大小下的复制时间和成本. 本文的实验开启了对象的一致性

验证, 即当复制任务完成时, 系统会校验对象的哈希值是否符合预期. 因为用户通常更倾向于在可能的情况下选择将数据复制到邻近的区域, 以避免高昂的数据传输费用, 因此本文将实验中的起点云区域和终点云区域设置为位于同一大洲 (同为北美洲或欧洲). 复制成本被拆分为两个类别: 传输成本和虚拟机或云函数产生的计算成本. 传输成本指的是数据传输费用, 对同一大小的对象该费用是一致的. 计算成本则主要为虚拟机、云函数产生的费用以及少量云平台接口调用的费用. 对于 Skyplane, 在 100 GB 对象复制的实验中在每个云区域使用了 8 个虚拟机, 而其他大小的实验则在每个云区域使用了 1 个虚拟机.

● 亚马逊 AWS 实验结果. 图 10(a) 展示了在 AWS 的 us-east-1 和 us-east-2 之间的 3 个系统的复制时间. 总体而言, 与最佳的基线系统相比, FastRep 减少了 72%–98% 的复制时间. 对于小于 100 MB 的对象, Skyplane 的大部分复制时间都花费在虚拟机创建和容器启动上. S3 RTC 的复制时间从 20–30 s 不等, 其复制时间与对象大小无明显正相关关系. 相比之下, FastRep 只使用一个云函数进行复制, 复制时间始终保持在 3 s 以下. 对于大于 1 GB 的对象, FastRep 会并行启动多个函数进行复制. 尽管复制时间随对象大小的增大而增加, FastRep 仍然能将时间控制在 25 s 以内. 尽管 Skyplane 在每个云区域使用了 8 个虚拟机来复制 100 GB 的对象, 但它仍需要超过 180 s 才能完成. 实验显示, S3 RTC 在复制较大对象时时间显著增加. 对于 1–100 GB 的对象, 复制时间大约为 80 s, 如果长时间未进行复制, 还会存在一定的冷启动时间.

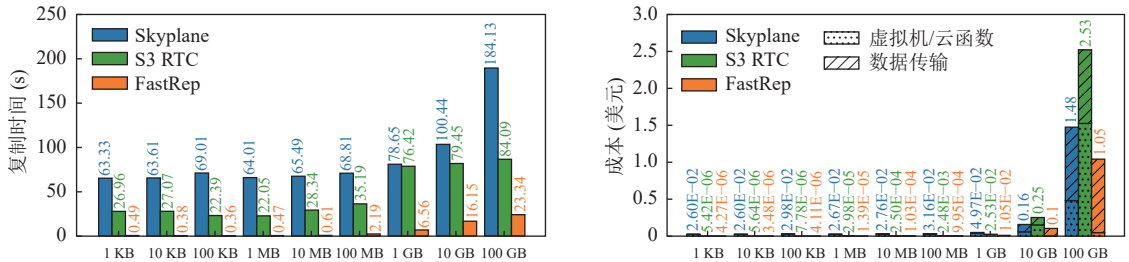
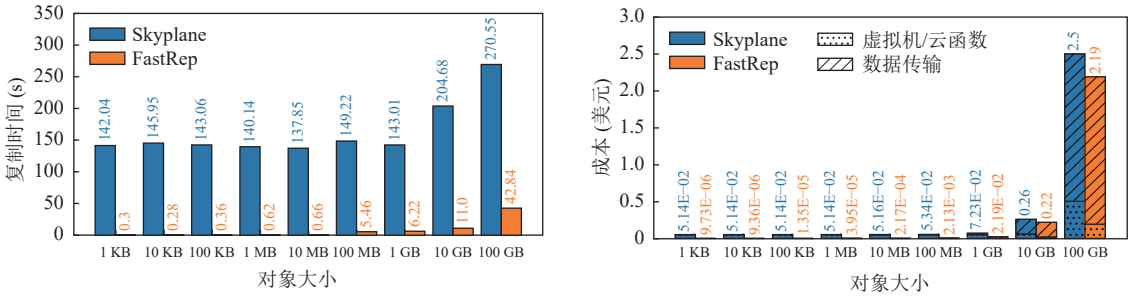


图 10　AWS 云区域之间 (us-east-1 到 us-east-2) 的复制时间和成本

成本方面, FastRep 也优于两个基线系统. us-east-1 和 us-east-2 之间的数据传输价格为 0.01 美元/GB. 在图 10(b) 中, 对于 1 GB 以下的对象, 由于每次启动新虚拟机时用户至少需要支付 60 s 的费用, 因此 Skyplane 的每字节传输成本是最高的. 因为 S3 RTC 的定价基于复制对象的大小 (0.015 美元/GB), 所以 S3 RTC 相比 Skyplane 在小对象上更具成本效益. 而对于 1 GB 以下的对象, FastRep 相较于 S3 RTC 可以进一步降低 21%–59% 的成本. S3 RTC 在复制大对象时变得更加昂贵. 除此之外, S3 RTC 还要求用户启用对象版本控制, 其生命周期管理策略只能以天为粒度删除历史版本. 而这可能会导致额外的存储成本, 图 10(b) 的实验中并未覆盖这一部分费用. 随着对象大小的增大, Skyplane 和 FastRep 之间的成本差异逐渐缩小, 这是因为数据传输成本开始占主导地位.

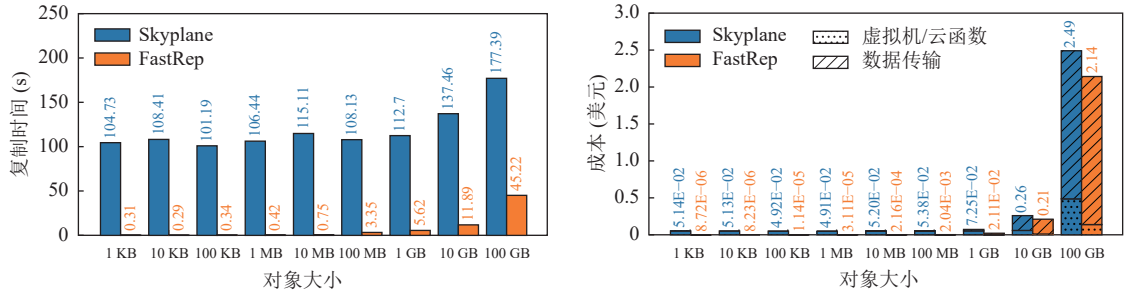
● 微软 Azure 实验结果. 图 11(a) 展示了在 Azure 的 UK South 和 Sweden Central 之间的复制时间. 尽管 Skyplane 和 FastRep 在 Azure 上的性能相较在 AWS 上都有所下降, 但 FastRep 仍然优于 Skyplane. 与 AWS 的实验结果相比, Skyplane 在 Azure 上启动虚拟机的时间增加了约 30 s. 对于 FastRep, 小对象的复制依然很快, 小于 1 GB 的对象可以在 7 s 内完成复制. 然而, FastRep 复制 100 GB 对象需要 42 s. 导致 FastRep 复制较大对象时间变长的原因之一是 FastRep 已经达到了 Azure 存储账户的带宽限制. Azure 限制每个存储账户的入站带宽为 25–60 Gb/s, 这一限制导致在实验过程中出现了“Ingress is over the account limit”错误. 我们预计通过发起工单使 Azure 存储账户提高入站带宽的上限, 进一步改善 FastRep 的性能表现. 在图 11(b) 中, Skyplane 和 FastRep 的复制成本均高于 AWS. 在传输成本方面, Azure 的 UK South 和 Sweden Central 之间的数据传输定价为 0.02 美元/GB, 较 AWS 的 us-east-1 和 us-east-2 之间提升至 2 倍. 计算成本方面, 由于相对较长的运行时间, 虚拟机和云函数产生的费用也更高. Azure Functions 的粗粒度内存配置也为 FastRep 增加了额外的费用.



(a) 复制时间 (b) 复制成本

图 11 Azure 云区域之间 (UK South 到 Sweden Central) 的复制时间和成本

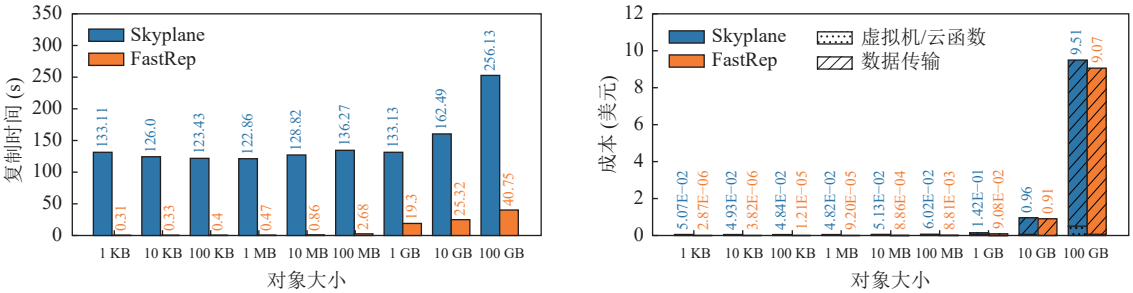
● 谷歌 GCP 实验结果. 图 12 展示了在 GCP 的 us-east1 和 us-east4 之间的复制时间和成本 (GCP 的云区域格式与 AWS 不同). 从该实验可以得出类似结论. 由于谷歌云虚拟机启动较慢, Skyplane 的复制时间超过了 100 s, 且虚拟机产生的费用也较高. 对于 FastRep, 复制 100 GB 的对象需要 45 s. Cloud Run 有时在接收到函数调用时不会立即启动新的实例, 而是等到正在运行中的云函数执行完成后复用实例. 这会导致复制任务产生更高的延迟. 与 AWS 相比, FastRep 在大对象复制上的成本也更高. 较高的成本主要来源于 0.02 美元/GB 的数据传输费用和更长的函数执行时间.



(a) 复制时间 (b) 复制成本

图 12 GCP 云区域之间 (us-east1 到 us-east4) 复制时间和成本

● 跨云实验结果. 图 13 展示了从 AWS 的 us-east-1 到 Azure 的 West US 之间的跨云复制的时间和成本. 图 2 的实验显示, 云函数即使在跨云链路上也能够实现线性的性能扩展. 对于 FastRep, 100 GB 的对象复制时间为 40 s, 而小于 100 MB 的对象则仅需 3 s. 相比之下, Skyplane 在 Azure 侧的虚拟机启动较慢. 100 GB 对象的复制时间与 Azure 的复制实验结果非常接近. 产生这一现象的一个主要原因是这两个实验都受到了 Azure 存储账户入站带宽限制的影响. 对于大对象而言, Skyplane 和 FastRep 数据传输费用相同, 均为 0.09 美元/GB. 对于 Skyplane 和 FastRep 而言, 复制 100 GB 对象的总费用中, 数据传输费用占比超过了 94%.

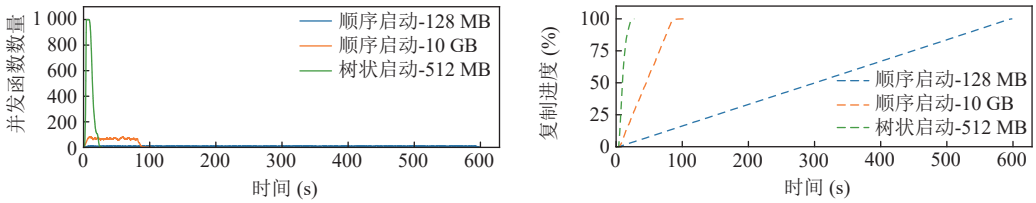


(a) 复制时间 (b) 复制成本

图 13 跨云平台 (us-east-1 到 West US) 的复制时间和成本

4.3 技术有效性验证

● 函数调用优化算法的有效性. 本文首先评估基于时间有界完全树的函数调用优化算法的有效性. 如图 14 所示, 实验使用最多 1 000 个复制函数 (AWS 默认支持的并发上限) 复制 1 个 100 GB 的对象. 实验首先在 128 MB 和 10 GB 两种不同内存配置下通过单个入口函数去触发其他复制函数. 虽然增加内存可以改善函数调用时间, 但其最大并发数最多仅能达到 83 个云函数, 完成复制仍然需要超过 100 s. 相比之下, 本文提出的函数调用优化算法对内存配置不敏感. 根据经验观察, 本文将算法中 D 的值配置为 10, 该算法在 5 s 内启动了 1 000 个云函数, 并在 30 s 内完成了复制任务.



(a) 并发函数数量 (b) 复制进度

图 14 函数调用优化算法的效果

● 自适应函数克隆的有效性. 本文接下来评估自适应函数克隆在带宽下降时是否能够实现进度的追赶. 为了可靠地复现实验, 实验将复制函数的带宽限制在对应水平来模拟带宽下降. 实验使用 100 个云函数复制一个 10 GB 的对象, 并在不同的带宽条件下进行测试. 当带宽下降不足 20% 时, 因为额外的开销可能会进一步影响性能, 系统不启用函数克隆. 如图 15 所示, 当启用自适应函数克隆时, 复制时间能够保持在 30 s 以内, 而当关闭自适应函数克隆且带宽下降 80% 时, 复制时间会达到 90 s.

● 应对请求突发的能力. 为了进一步验证 FastRep 在不同负载密度下的稳定性和可扩展性, 本文同时触发多个 10 MB 对象的复制任务, 并测量在并发任务数从 1 增加到 256 的情况下, 单个小尺寸对象的平均端到端复制延迟. 实验结果如图 16 所示, FastRep 展现出了较高的稳定性. 在突发的 256 个并发复制任务下, 系统的平均复制延迟仍然稳定地维持在 1 s 以内, 与单个任务在无负载情况下独立执行的延迟相比没有出现显著增长.

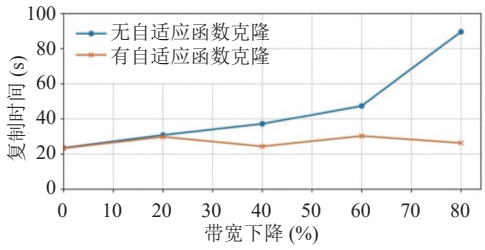


图 15 自适应函数克隆的效果

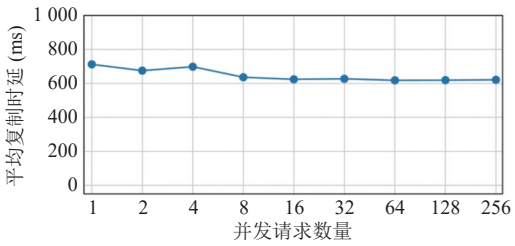


图 16 应对请求突发的能力

5 讨 论

FastRep 的整体性能与成本受一系列核心参数的影响. 这些参数共同决定了系统在面对不同大小的对象和多变的云环境下, 如何在复制延迟和运行成本之间取得最优平衡.

并行函数数量 n 是最直接的性能调节杠杆. 对于较大对象, 增加 n 能够有效提升聚合传输带宽. 然而, n 的选择并非越大越好, 它构成了系统中最核心的权衡点. 一方面, 过多的并发函数会使函数调用的开销成为新的性能瓶颈, 反而导致总复制时间大幅上升. 另一方面, 计算成本与 n 的规模和总执行时间正相关, 过多的函数会直接推高成本. 因此, FastRep 以达到 SLO 为目标, 尽可能使用较小的 n.

与并行度相比, 云函数的内存配置则展现了成本与效益的边际递减关系. 实验表明, 虽然增加内存可以在一定

程度上提升单个函数的传输速率, 但其性能增益在超过特定阈值后便不再显著. 考虑到更高的内存配置会带来线性增长的单位时间成本, FastRep 为实现成本效益最大化, 选择采用固定的、性价比较高的内存配置 (512 MB), 并主要依赖并行扩展来满足不同场景下的性能需求.

最后, 系统的动态适应能力也依赖于关键算法参数的设定. 函数启动延迟 D 是函数调用优化算法的基石, 准确地对 D 进行建模是生成最优调用树、实现快速启动的前提. 而在数据传输阶段, 自适应函数克隆机制的触发阈值 δ 决定了系统的反应灵敏度. 过于敏感的阈值可能会因暂时的网络抖动而触发不必要的函数克隆, 从而增加额外开销; 而过于迟钝则可能无法及时应对带宽骤降, 导致违反复制时间目标. FastRep 对此采取了折中策略: 系统会容忍一定程度的性能下降 (<20%), 仅在传输速率出现严重偏差时才启动克隆, 以此作为保障性能的最后手段, 避免了在轻微波动下的过度反应和资源浪费.

6 总 结

跨云对象存储复制对提升云存储的可用性和可靠性具有重要意义. 本文提出了基于服务器无感知计算的对象存储跨云复制系统 FastRep. FastRep 通过对象粒度的复制任务锁和乐观的复制与校验机制保障了跨云区域对象复制的数据一致性. 本文对跨云区域的对象存储复制任务所需时间进行了建模. 基于该模型, FastRep 通过基于时间有界完全树的函数调用优化算法和基于自适应函数克隆的传输时间优化机制加速了数据复制任务. 本文对 FastRep 的原型系统进行了全面的评估. 实验结果显示, FastRep 能够以分钟级延迟实现动态大小对象的复制, 与现有跨云区域对象存储复制系统相比, 同时降低了复制时间和成本.

References

[1] Alibaba cloud suffers second service outage in a month. 2023. https://www.reuters.com/technology/alibaba-cloud-suffers-second-service-outage-month-2023-11-28/

[2] UniSuper. A joint statement from UniSuper CEO Peter Chun, and Google Cloud CEO, Thomas Kurian. 2024. https://www.unisuper.com.au/about-us/media-centre/2024/a-joint-statement-from-unisuper-and-google-cloud

[3] Harbert T. Tapping the power of unstructured data. 2021. https://mitsloan.mit.edu/ideas-made-to-matter/tapping-power-unstructured-data

[4] Amazon S3. 2025. https://aws.amazon.com/s3/

[5] Azure Blob Storage. 2025. https://azure.microsoft.com/en-us/products/storage/blobs

[6] Satija S, Ye CH, Kosgi R, Jain A, Kankaria R, Chen YW, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Srinivasan K. Cloudscape: A study of storage services in modern cloud architectures. In: Proc. of the 23rd USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2025. 7.

[7] Jain P, Kumar S, Wooders S, Patil SG, Gonzalez JE, Stoica I. Skyplane: Optimizing transfer cost and throughput using cloud-aware overlays. In: Proc. of the 20th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2023. 1375–1389.

[8] Schleier-Smith J, Sreekanti V, Khandelwal A, Carreira J, Yadwadkar NJ, Popa RA, Gonzalez JE, Stoica I, Patterson DA. What serverless computing is and should become: The next phase of cloud computing. Communications of the ACM, 2021, 64(5): 76–84. [doi: 10.1145/3406011]

[9] Google Cloud Storage. 2025. https://cloud.google.com/storage

[10] Aliyun. Aliyun object storage OSS. 2025 (in Chinese). https://www.aliyun.com/product/oss

[11] Eytan O, Harnik D, Ofer E, Friedman R, Kat R. It's time to revisit LRU vs. FIFO. In: Proc. of the 12th USENIX Conf. on Hot Topics in Storage and File Systems. USENIX Association, 2020. 12.

[12] Building for durability in Amazon S3 and Glacier. 2018. https://www.youtube.com/watch?v=nLyppihvhpQ

[13] AWS. AWS DataSync. 2025. https://aws.amazon.com/datasync/

[14] Azure AzCopy. 2025. https://learn.microsoft.com/en-us/azure/storage/common/storage-ref-azcopy

[15] Wooders S, Liu S, Jain P, Mo XX, Gonzalez JE, Liu V, Stoica I. Cloudcast: High-throughput, cost-aware overlay multicast in the cloud. In: Proc. of the 21st USENIX Symp. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2024. 17.

[16] García-Dorado JL, Rao SG. Cost-aware multi data-center bulk transfers in the cloud from a customer-side perspective. IEEE Trans. on Cloud Computing, 2019, 7(1): 34–47. [doi: 10.1109/TCC.2015.2469666]

[17] Wu Z, Butkiewicz M, Perkins D, Katz-Bassett E, Madhyastha HV. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In: Proc. of the 24th ACM Symp. on Operating Systems Principles. Farminton: ACM, 2013. 292–308. [doi: 10.1145/2517349.2522730]

[18] Park H, Qiu, ZY, Ganger GR, Amvrosiadis G. Reducing cross-cloud/region costs with the auto-configuring Macaron cache. In: Proc. of the 30th ACM Symp. on Operating Systems Principles. Austin: ACM, 2024. 347–368. [doi: 10.1145/3694715.3695972]

[19] Liu S, Mo XX, Hershcovitch M, Zhang H, Cheng A, Girmonsky G, Vernik G, Factor M, Bang T, Ponnapalli S, Crooks N, Gonzalez JE, Harnik D, Stoica I. SkyStore: Cost-optimized object storage across regions and clouds. Proc. of the VLDB Endowment, 2025, 18(7): 2084–2096. [doi: 10.14778/3734839.3734846]

[20] Alluxio. 2025. https://www.alluxio.io/

[21] AWS. AWS Snowball. 2025. https://aws.amazon.com/snowball/

[22] Azure Data Box. 2025. https://azure.microsoft.com/en-us/products/databox

[23] AWS. AWS Lambda. 2025. https://aws.amazon.com/lambda

[24] Azure Functions. 2025. https://azure.microsoft.com/en-us/products/functions

[25] Google Cloud Run Functions. 2025. https://cloud.google.com/functions?hl=en

[26] Agache A, Brooker M, Florescu A, Iordache A, Liguori A, Neugebauer R, Piwonka P, Popa DM. Firecracker: Lightweight virtualization for serverless applications. In: Proc. of the 17th USENIX Conf. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2020. 419–434.

[27] Du D, Yu TY, Xia YB, Zang BY, Yan GL, Qin CG, Wu QX, Chen HB. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In: Proc. of the 25th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2020. 467–481. [doi: 10.1145/3373376.3378512]

[28] Roy RB, Patel T, Tiwari D. IceBreaker: Warming serverless functions better with heterogeneity. In: Proc. of the 27th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Lausanne: ACM, 2022. 753–767. [doi: 10.1145/3503222.3507750]

[29] Oakes E, Yang L, Zhou D, Houck K, Harter T, Arpaci-Dusseau AC, Arpaci-Dusseau RH. SOCK: Rapid task provisioning with serverless-optimized containers. In: Proc. of the 2018 USENIX Conf. on Usenix Annual Technical Conf. Boston: USENIX Association, 2018. 57–69.

[30] Akkus IE, Chen RC, Rimac I, Stein M, Satzke K, Beck A, Aditya P, Hilt V. SAND: Towards high-performance serverless computing. In: Proc. of the 2018 USENIX Conf. on Usenix Annual Technical Conf. Boston: USENIX Association, 2018. 923–935.

[31] Yang G, Liu J, Qu MZ, Wang S, Ye D, Zhong H. Review on performance optimization technology in serverless computing system. Ruan Jian Xue Bao/Journal of Software, 2025, 36(1): 47–78 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7190.htm [doi: 10.13328/j.cnki.jos.007190]

[32] Zhang ZL, Jin C, Jin X. Jolteon: Unleashing the promise of serverless for serverless workflows. In: Proc. of the 21st USENIX Symp. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2024. 10.

[33] Jin C, Zhang ZL, Xiang XY, Zou SY, Huang G, Liu XZ, Jin X. Ditto: Efficient serverless analytics with elastic parallelism. In: Proc. of the 2023 ACM SIGCOMM Conf. New York: ACM, 2023. 406–419. [doi: 10.1145/3603269.3604816]

[34] Mahgoub A, Yi EB, Shankar K, Elnikety S, Chaterji S, Bagchi S. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In: Proc. of the 16th USENIX Symp. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2022. 303–320.

[35] Zhang H, Tang YP, Khandelwal A, Chen JR, Stoica I. Caerus: NIMBLE task scheduling for serverless analytics. In: Proc. of the 18th USENIX Symp. on Networked Systems Design and Implementation. USENIX Association, 2021. 653–669.

[36] Shahrad M, Fonseca R, Goiri Í, Chaudhry G, Batum P, Cooke J, Laureano E, Tresness C, Russinovich M, Bianchini R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In: Proc. of the 2020 USENIX Conf. on Usenix Annual Technical Conf. USENIX Association, 2020. 14.

[37] Wen JF, Chen ZP, Liu Y, Liu XZ. Measurement study on performance of serverless platforms. Ruan Jian Xue Bao/Journal of Software, 2025, 36(5): 1974–2005 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7191.htm [doi: 10.13328/j.cnki.jos.007191]

[38] Skyplane v0.3.2. 2025. https://github.com/skyplane-project/skyplane/tree/0.3.2

[39] AWS. Meeting compliance requirements with S3 replication time control. 2025. https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication-time-control.html

[40] Object replication for block blobs. 2025. https://learn.microsoft.com/en-us/azure/storage/blobs/object-replication-overview

[41] Google Cloud Storage—Turbo Replication. 2025. https://cloud.google.com/storage/docs/availability-durability#turbo-replication

附中文参考文献

[10] 阿里云. 阿里云对象存储 OSS. 2025. https://www.aliyun.com/product/oss

[31] 杨光, 刘杰, 曲慕子, 王帅, 叶丹, 钟华. 服务器无感知计算系统性能优化技术研究综述. 软件学报, 2025, 36(1): 47–78. http://www.jos.org.cn/1000-9825/7190.htm [doi: 10.13328/j.cnki.jos.007190]

[37] 温金凤, 陈震鹏, 柳熠, 刘譞哲. 服务器无感知平台性能度量研究. 软件学报, 2025, 36(5): 1974–2005. http://www.jos.org.cn/1000-9825/7191.htm [doi: 10.13328/j.cnki.jos.007191]

作者简介

舒俊宜, 博士生, CCF 学生会员, 主要研究领域为系统软件, 云计算.

黄小龙, 博士生, 主要研究领域为系统软件, 数联网.

马郓, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为智能化系统软件, Web 系统.

李浩源, 博士, 主要研究领域为分布式系统, 云计算.

黄罡, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为系统软件, 数联网, 数据空间.

刘譞哲, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为系统软件, 服务计算.

金鑫, 博士, 副教授, 博士生导师, CCF 高级会员, 主要研究领域为系统软件, 云计算.

梅宏, 博士, 教授, 博士生导师, CCF 会士, 主要研究领域为软件工程, 系统软件.