面向大规模集群的柔性配置更新推送方法

唐震^{1,2}, 王伟^{1,2,3*}, 黄宇^{4*}, 李艳林⁵, 纪树平⁶, 宋傲^{1,2}, 魏峻^{1,2}, 黄涛^{1,2}

1. 计算机科学国家重点实验室 (中国科学院软件研究所), 北京 100190, 中国

2. 中国科学院大学, 北京 100049, 中国

3. 中国科学院软件研究所南京软件技术研究院, 南京 211135, 中国

4. 计算机软件新技术国家重点实验室 (南京大学), 南京 210023, 中国

5. 阿里巴巴集团, 杭州 311121, 中国

6. Electrical & Computer Engineering, University of Toronto, Toronto ON M5S 3G4, Canada

* 通信作者. E-mail: wangwei@otcaix.iscas.ac.cn, yuhuang@nju.edu.cn

收稿日期: 2020-05-07; 修回日期: 2020-08-20; 接受日期: 2020-10-13; 网络出版日期: 2020-11-09

国家重点研发计划 (批准号: 2017YFB1001804) 和阿里巴巴创新研究计划 (Alibaba Innovative Research, AIR) 资助项目

摘要 配置管理是支撑云服务提供商管理大规模容器集群的重要基础设施. 这一规模的集群通常包含百万量级的容器实例, 如何根据不同业务场景的需求, 及时可靠地将配置更新推送至对其感兴趣的容器实例, 是亟待解决的关键问题. 然而, 现有方法仍存在不足之处. 保障顺序一致性的共识算法限制了集群的扩展能力, 难以适用于大规模集群. 反熵算法存在长尾现象, 时延难以保障, 不适用于推送关键的配置更新. 为了应对上述挑战, 本文提出了一种面向大规模集群的柔性配置更新推送方法. 这一方法基于发布/订阅机制, 引入基于完全 N 叉树拓扑的可定义的多层次推送, 并使用订阅者的部分计算资源协助推送, 以提升推送性能; 引入容错机制以应对节点失效和网络分区, 保障网络分区时多分区读写可用. 方法的拓扑参数和策略可根据业务场景对性能、可靠性等维度的不同需求而调整. 实验结果表明, 与现有的方法相比, 我们的方法可以有效降低更新推送的时延, 并可有效应对节点失效和网络分区场景.

关键词 更新推送, 配置管理, 完全 N 叉树

1 引言

配置管理系统是支撑云服务提供商实现大规模集群运维的重要基础设施. 一个典型的场景是基于容器的集群管理. 主流的云服务提供商, 如 Amazon AWS¹⁾、阿里云²⁾、Microsoft Azure³⁾等都提供了

1) Amazon. Amazon web services (AWS) — cloud computing services. <https://aws.amazon.com/>.

2) Alibaba. Alibaba cloud: reliable & secure cloud solutions to empower your global business. <https://www.alibabacloud.com/>.

3) Microsoft. Microsoft azure cloud computing platform & services. <https://azure.microsoft.com/en-us/>.

引用格式: 唐震, 王伟, 黄宇, 等. 面向大规模集群的柔性配置更新推送方法. 中国科学: 信息科学, 2020, 50: 1645–1664, doi: 10.1360/SSI-2020-0126
Tang Z, Wang W, Huang Y, et al. Flexible configuration update delivery for large clusters (in Chinese). Sci Sin Inform, 2020, 50: 1645–1664, doi: 10.1360/SSI-2020-0126

基于容器的实例供用户部署基于微服务的应用. 容器内的应用或容器实例会向配置管理系统订阅相应的配置, 从而及时根据最新配置改变程序的行为. 从数据中心的视角看, 应用以及容器实例都存在需要快速推送的配置以保证服务正常运行. 当用户发起配置更新时, 更新数据应当高效可靠地被推送至订阅它的容器实例. 在主流云服务提供商的容器实例规模已达到百万量级的背景之下, 配置更新推送所面临的性能、可用性和可靠性问题更为严峻, 不同业务场景对这些维度的需求也存在差异. 例如, 当某一服务需要进行扩容, 需要向服务的 IP 列表中添加节点时, 能容忍的最大配置更新推送时延较高, 并希望尽可能降低系统的整体负载; 但当某一服务需要进行缩容或出现节点失效, 需要从服务的 IP 列表中删除节点时, 能容忍的最大配置更新推送时延较低, 并希望尽可能多地利用系统资源以提升推送性能. 如果这些差异化的配置更新推送需求未能得到有效保障, 可能导致服务出现负载不均衡, 或应用的请求被路由到错误的节点而返回超时或错误的结果, 引发严重后果. 因此, 亟需柔性的配置更新推送方法, 针对不同业务场景选择合适的策略以应对不同的性能、可靠性等维度的需求.

考虑大规模集群环境下的一个典型的配置管理场景. 数百台服务器组成的发布者集群负责向数百万客户端组成的订阅者集群推送配置更新, 发布者和订阅者分布在多个数据中心. 典型的配置包括服务的 IP 地址和端口号、服务列表等. 这一场景对配置更新推送提出了新的要求. 首先, 为了保证服务质量, 配置更新需要在限定时间内到达订阅了这一配置的容器实例. 如果更新未能及时送达, 可能导致订阅者集群中存在不一致的数据, 容器实例可能会向已被移除的服务实例发送请求, 导致请求超时或出错. 其次, 容器实例很有可能由于某些原因而不可用, 当其从失效状态中恢复时, 需要将未送达的更新重新推送至该容器实例, 否则也有可能造成状态不一致. 进一步地, 多地多数据中心的部署方式已成为承载大规模集群的有效途径. 电源故障、网络链路故障等小概率事件会变得相对常见, 从而导致网络分区. 在出现网络分区时, 仍然要求每个独立的网络孤岛内的配置更新可以正常推送. 这要求配置更新推送机制可以跟踪推送进度, 识别不可达节点并保证更新数据最终可以送达每个节点.

综上, 随着集群规模的增加, 配置更新推送面临日益严峻的挑战, 具体而言包括如下 3 方面:

(1) **低时延.** 配置更新应当在限定时间内到达指定比例的订阅了这一配置的订阅者节点 (容器实例). 例如, 一个典型的场景是要求服务对应的 IP 列表的更新在 3 s 内到达 99% 的可达订阅者节点.

(2) **容错和多分区可用.** 当节点由于故障或网络分区而出现不可达的情形时, 配置更新应当能够到达可达客户端. 进一步地, 对于网络分区场景, 位于不同网络分区内的节点应当是读写可用的, 即配置更新可以被正确推送到所在网络分区的订阅者节点.

(3) **可跟踪.** 配置管理系统应当可以跟踪配置更新推送的具体过程, 并保证不可达节点从故障或网络分区中恢复后最终可以收到更新.

值得注意的是, 现有的保障顺序一致性的共识算法, 如 Paxos^[1] 及其变种算法和 Raft^[2] 算法, 已广泛应用于分布式键值 (key-value) 存储系统中, 也常用于在集群中推送配置更新. 但顺序一致性对于配置更新推送场景而言相对过强, 因为配置更新推送过程中出现的数据不一致情形可以较为容易地使用重发配置更新的方式解决. 此外, 这类算法在出现网络分区时只能保证拥有半数以上节点的分区读写可用. 因此, 对于大规模集群而言这类算法并不适用. 反熵, 或 Gossip 类算法^[3,4] 等保障最终一致性的算法的初始传播速度较快, 但这类算法存在长尾现象, 导致配置更新推送时延难以确定, 整体时延可能较长, 因此对于大规模集群而言也并不适用. 从配置更新推送的机制来看, 现有的发布/订阅 (Pub/Sub) 方法^[5,6] 主要关注消息的发布/订阅流程, 并未特别关注消息的具体传播过程, 也很少关注如何使用订阅者的资源来协助完成配置更新推送. 因此现有的发布/订阅方法也难以适用于百万订阅者规模的集群的配置更新推送场景.

为了应对上述 3 方面挑战, 我们选择使用完全 N 叉树作为配置更新推送的底层拓扑, 与 Raft 等

算法使用的一对多广播以及 Gossip 类算法使用的随机推送相比,可以减少发布者集群的平均开销.同时,我们引入了“绕开不可达节点”的容错机制以及确认机制,以保证网络分区时的多分区读写可用,并保证更新最终可送达所有订阅者.此外,我们分析了大规模集群的配置管理场景,并得出了两个核心观察以指导我们设计柔性配置更新推送方法:(1)对于不同类型的配置更新,可容忍的最大“一致性窗口”可能不同;(2)订阅了相同配置的容器实例很有可能从属于同一个应用或数据中心.基于核心观察 1,我们选择调整底层拓扑的参数以满足不同配置更新推送的最大时延限制.基于核心观察 2,我们选择引入两阶段的推送机制,将从属于同一个应用或数据中心的容器实例组织在一个拓扑中,使用容器实例的部分计算资源来协助完成配置更新推送以提升性能,并使用不同倾向性的策略以权衡性能、可靠性和节点负载.

我们基于阿里巴巴的开源配置管理系统 Nacos⁴⁾实现了柔性配置更新推送方法.我们在由 200 个服务器节点和 10^6 个客户端节点组成的大规模集群中验证了方法的效果.实验结果表明,与现有方法相比,我们的方法可以实现高效的配置更新推送,在配置管理系统处于高负载时,配置更新推送的时延最多可减少 74.7%.此外,它可以容忍节点不可用的情形,且在出现网络分区时可保证所有网络分区都可正常推送配置更新.

本文的具体贡献如下:

(1) 我们构建了环境模型,并引入了“一致性窗口”的概念,用以刻画不同业务场景对配置更新推送的需求.

(2) 我们提出了基于完全 N 叉树的底层配置更新推送拓扑,以环境模型为指导,使用可调整的多轮次推送以适应不同的业务场景.进一步地,我们提出了两阶段推送机制,使用客户端的部分计算资源协助进行配置更新推送,以提升性能.

(3) 我们提出了“绕开不可达节点”的容错机制和确认机制,从而保证不可达节点最终可以收到配置更新,且在出现网络分区时各分区均读写可用.对于不同的业务场景,可调整返回操作成功时需要收到的确认消息的比例,以权衡性能和可靠性.

本文的后续章节按如下方式组织:第 2 节介绍了大规模集群的配置管理场景,并介绍了两个核心观察.第 3 节提出了系统模型和环境模型,并刻画了研究问题.第 4 节介绍了基于完全 N 叉树的底层拓扑以及相应的容错机制,以实现柔性配置更新推送.第 5 节介绍了系统原型的实现以及实验验证的细节.第 6 节对比了相关工作,第 7 节总结了全文.

2 问题分析

主流云服务提供商通常维护了百万量级的容器实例用于承载服务.对配置管理系统而言,这些容器实例都作为客户端存在.针对大规模集群,如何根据不同业务场景的需求,高效、可靠地将配置更新推送至感兴趣的客户端节点,是亟待解决的关键问题.大规模集群的配置更新推送场景如图 1 所示.由数百台服务器组成的配置管理集群作为发布者,由数百万客户端组成的订阅者集群向服务器订阅感兴趣的配置,并从服务器接受配置更新.对于一个给定的配置更新,应当在指定时间内送达指定比例的订阅了它的客户端节点.

然而,现有的配置管理系统通常基于 Paxos 和 Raft 等保障顺序一致性的共识算法.这些算法提供了很强的一致性保障,但由于它们依赖一个单一的领导者 (leader) 节点,集群规模受到领导者节点

4) Alibaba. Nacos: an easy-to-use dynamic service discovery, configuration and service management platform for building cloud native applications. <https://nacos.io/en-us/>.

图 1 (网络版彩图) 大规模集群中的配置更新推送
Figure 1 (Color online) Overview of configuration update delivery in large-scale clusters

处理能力的限制, 系统的伸缩性受限, 当集群规模增大时配置更新推送时延难以保障, 因此难以适用于百万量级客户端的配置更新推送. 此外, 配置管理场景中的配置数据通常用于管理容器或应用, 它们的更新频率相对较低, 且配置更新的冲突可以相对容易地通过重发配置更新来解决. 例如, 应用可能会将可用服务及其 IP 列表存储在配置管理系统中, 针对 IP 列表的添加、删除等更新操作的冲突不会导致应用产生非预期的行为, 并且在收到最新的配置更新时, 冲突会得到自动解决. 因此, 考虑到上述特征, 在大规模集群配置管理场景下, 保障配置更新按时送达比保障配置更新的顺序一致性相对更重要, 可以适当放松一致性保障以提升性能.

我们分析了大规模集群的配置更新推送场景, 并得出了两个核心观察以支撑我们设计新的柔性配置更新推送方法.

核心观察 1: 对于不同类型的配置更新, 可容忍的最大“一致性窗口”可能不同.

其中, “一致性窗口”指的是客户端收到配置更新的等待时间. 例如, 配置管理的一个典型应用是维护服务列表以支持微服务架构应用的服务发现, 包括向服务列表中添加节点 (服务扩容)、调整服务列表中的服务实例的权重 (负载均衡), 以及从服务列表中删除失效节点 (服务缩容). 这 3 种操作相关的配置更新对“一致性窗口”的容忍程度不同, 服务缩容的容忍程度较低, 因为服务调用请求如果被错误地路由到失效节点会直接导致服务中断. 相比较而言负载均衡的容忍程度较高, 而服务扩容的容忍程度最高. 未能及时应用负载均衡策略可能导致部分节点由于负载过高而响应缓慢或失效, 影响服务的整体可用性; 对于服务扩容而言, 配置更新未送达至多导致一小段时间窗口内新加入节点负载过低. 基于这一核心观察, 我们选择引入基于完全 N 叉树的层次化的底层推送拓扑, 通过为配置更新推送算法设定不同的参数以应对不同的“一致性窗口”容忍程度, 实现供需匹配.

核心观察 2: 订阅了相同配置的容器实例很有可能从属于同一个应用或数据中心.

配置管理系统中通常维护两种维度的配置: 容器的配置以及应用的配置. 位于同一个数据中心的容器可能共享大部分容器配置, 而从属于同一个应用的容器可能共享大部分应用配置. 考虑到一个大

型集群中的典型应用通常包含数百个节点, 而每个节点可订阅数千个配置, 其中包含一部分共享配置和服务, 其局部性非常明显. 基于这一核心观察, 使用客户端节点的部分计算资源来协助进行配置更新推送是合理的. 当发生配置变更时, 可以利用这一特征缩小配置更新推送的范围以降低负载、提升性能. 另一方面, 可以将共享同一配置的客户端组织在一个拓扑结构中, 使用客户端的计算能力协助进行配置更新推送, 以提升性能. 订阅相同配置的客户端都有针对这一配置的读取权限, 因此可以将其组织在一个拓扑中以加速配置更新推送. 对于特定配置, 在拓扑结构中未订阅它的客户端节点可将自身标记为失效, 从而避免配置更新推送过程中的带宽浪费.

3 模型

3.1 基本概念

在这一小节中我们首先将解释本文使用到的一些基本概念. 本文所涉及的配置为键值对形式存在的数据, 使用 key 唯一标识一个 value, value 的大小通常小于 1 MB, 因此可以较快地通过网络传输, 不需要依赖 P2P 等额外机制. 本文所涉及的配置管理系统由服务器和客户端两类节点组成, 每个节点上运行一个进程, 可映射为分布式算法中的处理器. 发起配置更新的角色是用户, 用户会向服务器集群中的某一台服务器发起更新请求, 将具有指定 key 的配置设置为一个新的 value.

配置管理系统可抽象为维护了配置的多份数据副本的分布式存储系统. 服务器节点的规模通常为数百台, 维护了所有数据副本, 可提供全部计算能力用于配置更新推送. 客户端节点即为容器实例, 规模通常为数百万台, 只维护感兴趣的数据副本, 可提供部分计算能力用于配置更新推送. 对于用户发起的一个给定的配置更新请求而言, 存在配置更新的“一致性窗口”, 本文中我们将这一窗口刻画为对于指定的数据副本, 从服务器接收到用户请求的时间点到所有可达的服务器节点和指定比例的感兴趣的客户端节点 (例如 99.9%) 都收到数据副本更新的时间点之间的窗口. 针对特定场景, 可允许的最大窗口大小即为配置更新的“一致性窗口”的容忍程度.

3.2 系统模型

我们将配置管理系统抽象为一个典型的分布式存储系统, 数据 (即配置) 副本会在多个节点上保持一致. 具体而言, 系统中包含 3 种角色: 客户端节点、服务器节点、用户. 本文以一个典型的大规模容器实例场景为例, 不失一般性, 系统中包含相对固定的 $n \leq 200$ 个服务器节点和可变的 $m \leq 10^6$ 个客户端节点. 因此, 对于一个给定的配置更新, 最多有 $m + n$ 个数据副本. 客户端节点会连接到某一服务器节点并订阅配置以获得配置更新. 每个服务器节点最多维持 $k \leq 10,000$ 个客户端连接. 服务器节点维护了连接到其上的客户端节点订阅配置的相关信息, 以支持配置更新推送. 用户会出于业务需要向服务器集群中的任意一台服务器节点发起配置更新请求.

节点之间的通讯遵循经典的消息传递 (message passing) 模型, 消息会在节点之间采用单播的方式传递. 在本文所述场景中, 消息包括元数据 (对于配置更新的通知) 和数据 (配置的最新值). 消息传递遵循异步的非拜占庭模型, 即消息可能由于节点失效或不可达而出现丢失、延迟、乱序, 但不会因为网络原因出现消息内容错误. 同时, 任意的客户端或服务器节点在任意时刻都有可能因为自身失效或网络原因而不可达.

节点间通讯如图 2 所示. 具体而言, 服务器接收两种类型的消息:

(1) **read(key)**. 客户端向服务器发起 read(key), 请求获得标识符为 key 的配置的最新值, 服务器读取本地存储并向客户端返回标识符为 key 的配置的最新值及对应的时间戳.

图 2 (网络版彩图) 节点间通信概览

Figure 2 (Color online) Communications between nodes

(2) **update(key, value)**. 用户或服务器向服务器发起 $\text{update}(\text{key}, \text{value})$, 请求将标识符为 key 的配置的值更新为 value , 并同步到所有订阅这一配置的客户端. 服务器本地将标识符为 key 的配置的值更新为 value , 并通知连接到它的客户端配置有更新. 特别地, 如果 $\text{update}(\text{key}, \text{value})$ 是由用户发起的, 收到这一请求的服务器会向集群中的其他服务器发出 $\text{update}(\text{key}, \text{value})$ 以更新数据副本.

客户端接收一种类型的消息: **notify(key)**. 当服务器上标识符为 key 的配置更新时, 会向连接到它向客户端发起 $\text{notify}(\text{key})$, 通知客户端标识符为 key 的配置有更新. 客户端收到请求后主动向服务器发起 $\text{read}(\text{key})$ 以获取配置更新.

如前文所述, 本文所涉及的配置更新推送场景不要求保障顺序一致性. 配置更新有可能出现冲突, 这一冲突通常可以在业务层通过重发配置更新的方式解决. 本文所述配置管理系统遵循 Delta 一致性模型, 即用户在 T 时刻发起并收到操作成功响应的一个给定的配置更新请求, 订阅这一配置的绝大部分客户端会在 $T + \delta$ 时刻之前获得最新副本. 在这里配置更新请求操作成功的标志是在集群中传播到了足够多的节点. δ 即为配置更新的最大“一致性窗口”.

3.3 问题定义

我们分析了完整的配置更新流程, 将这一过程解构为 3 个步骤: 配置更新在服务器集群内达成一致; 配置更新从服务器集群推送至客户端集群; 配置更新在客户端集群内达成一致. 相对应地, 我们需要根据给定配置项的“一致性窗口”容忍程度 (δ) 确定服务器集群内的推送算法、服务器集群到客户端集群的推送算法以及客户端集群内的推送算法, 并分别配置不同的参数以实现柔性配置更新推送, 满足用户要求的推送时延并具备容错能力. 如前文所述, 我们关注由数百台服务器和数百万台客户端构成的大规模集群中的配置更新推送. 在这一场景中, 用户会出于业务目的对某些配置进行更改, 并要求在经过一个给定的时间窗口后, 配置更新可以到达订阅这一配置的所有可达的客户端节点. 由于网络分区或节点本身失效导致不可达的节点会在故障恢复之后接受到这一配置更新.

具体而言, 我们使用如下模型描述这一问题: 首先, 用户可能向任意服务器节点发送请求, 更新某一配置, 并由服务器向订阅这一配置的客户端推送更新. 用户的请求发送速率为每秒 t 个, 特定时间段内的请求服从参数为 $\lambda = t \text{ req/s}$ 的泊松分布. 对于不同的配置更新, 存在不同的一致性容忍程度 δ .

图 3 (网络版彩图) 大规模集群配置更新推送示例

Figure 3 (Color online) An example of configuration delivery in the large-scale cluster

用户要求在 T 时刻发起的针对标识符为 k 的配置更新需要在 $T + \delta$ 的时间内到达至少 $X\%$ 的可达客户端, 并最终收集得到配置更新推送的统计信息, 即已推送的节点和不可达的节点列表. 亦即, 对于一个给定的配置更新或给定的客户端而言, 最终只存在两种可能状态: 客户端可达, 配置更新已到达这一客户端; 客户端由于节点故障或网络分区而不可达, 配置更新将在客户端可达时到达这一客户端, 同时不可达客户端的信息将会反馈给用户. 本文的目标是改进这一业务流程, 实现性能优化, 保证配置更新推送的 $T + \delta$ 的时效性, 并根据不同业务场景权衡不同维度, 以实现柔性配置更新推送.

一个具体的例子如图 3 所示. 服务器集群由部署在 3 个数据中心中的服务器节点构成, 若干客户端节点分别连接到每个服务器 (客户端 C_1, C_2, C_3 连接到服务器 S_1 ; 客户端 C_4, C_5, C_6 连接到服务器 S_2 ; 客户端 C_7, C_8, C_9 连接到服务器 S_3). 在这一简单例子中, 由于数据中心间的网络链路故障, 数据中心 1, 2 和数据中心 3 构成了 2 个网络分区 P_1 和 P_2 . 此外, 由于节点失效, 客户端 C_3 不可达. 客户端 $C_1 \sim C_9$ 订阅了标识符 k_1 对应的数据. 此时, 网络分区 P_1 中的用户向服务器集群中的 S_2 发起数据更新请求 $\text{update}(k_1, v_1)$, 即将标识符 k_1 对应的数据修改为 v_1 . 用户对时效的需求为“在 1 s 内将更新推送至 100% 的可达客户端节点”. 为应对这一请求, 配置更新推送方法需要达到如下目标:

(1) **更新按时送达.** k_1 对应的最新值 v_1 需要在 1 s 内到达 100% 的可达客户端 (即图中的 C_1, C_2, C_4, C_5, C_6).

(2) **具备容错机制.** 客户端 C_3 不可达, 以及网络分区 P_2 中的节点不可达不影响协议的正常执行.

(3) **多网络分区可用.** 网络分区 P_1 和 P_2 可独立响应读写请求, 即, 尽管网络分区 P_2 中的节点未过半, P_2 中的用户向服务器 S_3 发起的数据更新请求也应当被正常响应, 且在规定时间内到达客户端 C_7, C_8, C_9 中特定比例的节点.

(4) **具备确认机制.** 推送过程中可以识别出 C_3, S_3, C_7, C_8, C_9 不可达, 并会定期尝试重发, 当 C_3 从故障中恢复后重新上线, 或网络分区恢复后 S_3, C_7, C_8, C_9 重新可达时, 这些节点会收到 k_1 对应的最新值 v_1 .

3.4 环境模型

在第 2 节中我们介绍了两个核心观察并引入了配置更新的“一致性窗口”的容忍程度的概念, 本小节我们将进一步总结得到环境模型, 用于刻画系统所处状态, 包括环境特征、网络特征和节点特征, 以此作为算法的输入来保障 Delta 一致性并确保更新能够按时送达.

具体而言, 我们使用如下 3 个维度刻画环境特征:

(1) **服务器集群规模**. 可提供全部计算能力的节点规模. 由于服务器节点需要保存所有配置的副本, 较大的服务器集群规模会带来较高的集群内通讯代价, 但与此同时每个服务器节点所连接的客户端节点数目相对较少, 向客户端推送更新的开销较低.

(2) **以应用和数据中心粒度划分的客户端集群规模**. 可提供部分计算能力的节点规模. 客户端集群规模直接影响了整体负载, 并进一步影响对 Delta 一致性的保障能力. 位于同一个应用或数据中心的客户端可以被组织在一个树型拓扑中, 从而支持两阶段配置更新推送. 此外, 同样需要权衡应用或数据中心内的客户端节点规模以平衡配置更新推送的开销.

(3) **“一致性窗口”的容忍程度**. 对于特定的配置 (即数据副本) 的同步性能的最低预期, 这一关键维度反映了配置更新推送的性能需求. 特别地, 此维度由系统当前处理的所有配置类型中对推送性能要求最高的配置决定.

我们使用如下维度刻画网络特征和节点特征:

(1) **网络时延**. 指节点间交换数据时耗费在网络传输上的时间. 在本文所述场景中, 服务器集群内的节点间网络时延小于 1 ms, 从属于同一个应用的客户端集群内的节点间网络时延小于 1 ms, 位于同一个数据中心的节点间网络时延小于 3 ms, 跨数据中心的节点间网络时延小于 10 ms.

(2) **节点失效的概率模型**. 节点失效的概率模型可用于确定树型拓扑的具体参数. 在本文所述场景中, 由于故障导致的节点失效服从两点分布, 即对单个节点而言, 在特定时间会有 p_{nf} 的概率失效, $1 - p_{nf}$ 的概率不失效. 此外, 节点失效是独立事件, 因此由 k 个节点组成的集群中的节点失效的概率服从参数为 k, p_{nf} 的二项分布. 进一步地, 网络链路故障会将集群分为多个网络分区, 由于网络链路故障也可视为服从两点分布, 在本文所述场景中, 由 n_c 个网络链路组成的数据中心中由于链路故障导致网络分区的事件服从参数为 n_c, p_{cf} 的二项分布, 其中 p_{cf} 为单个网络链路出现故障的概率.

4 基于完全 N 叉树拓扑的配置更新推送

我们将配置更新推送方法解构为两个维度: 底层推送拓扑和容错机制. 在底层推送拓扑方面, 我们将服务器集群和客户端集群抽象为有向图, 与现有的使用一对 N 广播作为底层拓扑的方法不同, 我们选择使用完全 N 叉树作为底层推送拓扑. 在容错机制方面, 我们选择使用“绕开不可达节点”的容错策略保证节点不可达时推送过程仍能正确进行, 从而可以应对节点失效情形并在出现网络分区时保证多分区读写可用. 此外, 我们选择使用确认 (ACK) 机制来跟踪推送进度, 从而确保节点从失效状态恢复或网络分区恢复后最终可以收到更新. 与现有的基于广播的方法相比, 使用树型拓扑可以降低每个节点的平均负载, 即每个节点需要发送接收的消息量, 从而对于大规模集群或当配置管理系统处于高负载状态时可以有效降低配置更新推送的时延. 对于不同的业务场景, 我们通过调整完全 N 叉树的拓扑参数以及容错机制的具体策略来权衡性能、可靠性和节点负载.

Nacos 现有的配置更新推送流程如下所示, 我们将改进这一流程, 将其中第 2, 3, 5 步的推送底层拓扑替换为完全 N 叉树, 并添加容错和确认机制.

图 4 (网络版彩图) 基于完全 N 叉树的拓扑Figure 4 (Color online) The complete N -ary tree-based overlay

- (1) 用户 U 在 T 时刻向服务器集群中的一台服务器 (S_A) 提交修改标识符为 k 的配置项. S_A 更新 k 的本地数据副本.
- (2) S_A 将更新的数据副本以广播的方式通知其他服务器节点.
- (3) 收到配置更新的服务器将首先更新本地副本, 之后通知连接到它的所有对这一配置感兴趣的客户端.
- (4) 客户端会选择唯一一台服务器建立连接.
- (5) 每当有配置更新, 服务器只关心和它建立连接的客户端, 向这些客户端发送通知 (元数据).
- (6) 客户端接收到更新通知之后, 会建立一个新的连接用来传输具体的配置更新 (数据).
- (7) 当集群中指定比例服务器节点收到更新之后, 服务器 S_A 会向用户返回操作结果.
- (8) 所有客户端节点最终都会收到更新的配置.

4.1 基于完全 N 叉树的推送拓扑

本小节我们将介绍完全 N 叉树拓扑的构建以及为了保障按时推送进行的一系列优化.

4.1.1 配置更新推送的基本流程

基于完全 N 叉树的推送拓扑如图 4 所示. 我们将已排序的节点列表转换成生成树, 从而实现基于完全 N 叉树的拓扑构建. 对于完全 N 叉树而言, 节点的子节点的编号可由 N 导出, 在列表中第 i 个位置的节点的第 1 个子节点 (如果存在的话) 的位置为 $(i-1) \cdot N + 2$. 由此, 我们不需要在集群中的每个节点上保存拓扑结构. 此外, 修改拓扑结构时也只需要修改 N , 对应的父子节点坐标可以较为容易地计算得到. 进一步地, 我们通过调整完全 N 叉树中树的分叉数 N 来控制拓扑结构. N 直接影响了完全 N 叉树的树高, 即决定了配置更新推送的轮次. 通过调整 N 可以权衡配置更新推送的时延和节点的平均负载. 算法的具体步骤描述如下:

- (1) 构建生成树. 使用如下规则将集群列表转化为一棵完全 N 叉树:
  - (a) 由环境模型指导, 从当前的配置集合和访问模式导出对应的“一致性窗口”上限 (δ).
  - (b) 由 δ 导出完全 N 叉树中的拓扑结构参数, 即树的分叉数 N .

(c) 对于给定的服务器集群或从属于同一个应用或数据中心的客户端集群中的 m 个节点, 将节点遵照一定规则排序 (例如将节点的网卡 MAC 或 IP 地址映射为可排序的数值), 依次编号为 $1 \sim m$, 各自拥有唯一标识符, 构建拓扑时给定树的分叉数 N , 使用堆结构将集群节点列表转化为一棵完全 N 叉树.

(2) 以更新请求从用户到达客户端为例, 用户的请求以均匀等概率的方式发出, 到达特定的服务器, 速率为每秒最多 t 个请求, 服从参数为 $\lambda \leq t \text{ req/s}$ 的泊松分布.

(3) 接收到请求的服务器进行如下操作:

(a) 服务器将配置更新写入本地存储, 并应用 4.1.2 小节所述冲突解决机制以尝试解决并发更新冲突, 对于无法解决的冲突, 交由业务层通过重发配置更新的方式解决.

(b) 若当前节点不是根节点, 则将请求转发到拓扑结构中的根节点.

(c) 遍历连接的客户端, 向订阅这一配置的客户端发送 `notify(key)`, 通知客户端配置有更新.

(d) 在收到根节点及下层节点的确认消息后, 向用户返回操作成功.

(4) 由根节点开始递归向子节点传播, 收到更新的服务器遵照第 3 步的规则进行传播.

(5) 最终服务器集群中的所有节点都可收到用户请求.

(6) 收到更新请求的客户端更新本地数据副本, 并使用第 2 步类似的方式向自身所在应用或数据中心的根节点传播更新.

(7) 更新最终传播到所有可达客户端, 确认消息会逐层向上回收, 最终由服务器收集并返回给用户.

(8) 若目标节点不可达, 将由上层节点接管不可达节点的传播任务, 即, 替代不可达节点将配置更新发送到它的子节点.

(9) 服务器将定期回收节点的响应信息, 并向未发送确认消息的节点重发配置更新.

发送消息的具体过程的伪代码如算法 1 所示. 假设位于服务器节点列表中第 i 位的服务器 S_i 接收到了来自用户的请求 `Req`, 节点 S_i 上运行的消息处理器 P_i 的具体流程如算法 2 所示.

算法 1 DoSend(p)

Require: Processor p .

Ensure: Message delivered to the all sub-trees of p .

```

1: Initialization;
2: for all child  $c$  of  $p$  do
3:   if  $c$  is alive then
4:     send Req to  $c$ ;
5:   else
6:     DoSend( $c$ );
7:   end if
8: end for
```

4.1.2 冲突解决机制

在保障顺序一致性的协议中, 通常会引入版本号或全局时钟来保障操作以相同的顺序应用到所有数据副本上, 以此保障数据副本的一致性. 由于我们的方法放松了一致性保障以换取更新推送性能并保证网络分区时的读写可用, 对于并发的更新, 需要引入冲突解决机制, 对于无法通过这一机制解决的更新冲突将交由业务层通过重发配置更新的方式解决. 冲突解决机制的具体步骤如下:

算法 2 Processors(i)**Require:** Request Req.**Ensure:** Request Req has been handled.

```

1: Initialization;
2: if Req is sent from user then
3:   send Req to root;
4: else if Req is sent from  $P_k$  then
5:   update local state from Req and resolve conflicts;
6:   DoSend( $P_i$ );
7:   send Req to all connected clients;
8:   send Ack to  $P_k$ ;
9: else if Req is Ack sent from  $P_j (j \neq i)$  then
10:  send Ack to parent;
11: end if

```

(1) 在本文所述场景中, 针对特定配置项 (key) 的更新是相对低频率的操作, 对于不同 key 的更新操作可以安全地并发进行. 因此, 我们首先选择在每个发起的更新请求中附上来源方服务器的物理时间戳, 并通过 NTP 协议等机制同步集群中服务器的物理时间, 从而粗略地确定更新的先后顺序. 当针对同一个 key 的并发更新请求的时间戳差值大于集群间的最大时间差, 则该并发更新可以安全地应用到数据副本上, 反之则无法确定先后顺序, 需要按后续步骤处理.

(2) 配置管理和服务发现场景中的配置项具有不同的特征: 配置管理场景中的配置更新通常是为其指定的 key 设置一个新的 value (例如更新微服务的端口号等), 我们无法合并两个有冲突的 value; 与之相对地, 服务发现场景中的配置更新通常为将一个原子操作应用到指定 key 对应的 value 上 (例如向 IP 列表中添加或删除 IP), 此时对于某些操作组合我们可以计算出合并后的 value 以解决冲突. 因此, 我们针对这两类配置更新采用了不同的策略来确定更新是否可以正确应用在数据副本上:

(a) 对于配置管理场景, 我们对每个 key 引入混合逻辑时钟 (hybrid logical clock, HLC) ^[7,8] 来确定更新的先后顺序, 对于仍然无法确定先后顺序的并发配置更新, 会交由业务层通过重发的方式解决冲突.

(b) 对于服务发现场景, 由于每个 key 对应的是由一系列 IP 组成的服务列表, 我们同样先使用混合逻辑时钟进行判定, 对于无法判定先后顺序的并发更新, 我们使用简单的集合合并策略来处理并发的添加 IP 请求和删除 IP 请求. 对于添加和删除操作并发的情况, 交由业务层通过重发的方式解决冲突.

对于服务发现场景, 未来我们将尝试使用优化的机制以尽可能解决更新冲突, 避免交由业务层重发.

4.1.3 成员管理

在配置管理系统运行过程中可能会出现节点加入、删除的情形, 此外, 柔性配置更新推送方法也允许运行时动态调整完全 N 叉树的拓扑参数 N . 为应对这一需求, 我们基于前文所述的配置更新推送协议设计了成员管理协议. 由于成员管理协议需要改动推送拓扑, 为了保证消息可以正确送达, 在新旧拓扑的所有非叶子节点收到更新之前更新会遵照两份拓扑传播. 成员管理协议主要包括如下 3 个原子操作:

(1) **节点加入.** 编号为 i 的节点会向已知的拓扑中的根节点发送一个特殊的更新请求, 内容为 “Join(i)”. 根据配置更新推送协议, 如果收到这一消息的节点不是根节点, 它会递归将其发送至拓扑中

的根节点.

(2) **节点退出.** 节点退出时会向集群中发送一个特殊的更新请求, 内容为 “Leave(i)”.

(3) **参数修改.** 节点请求修改拓扑参数 N 时会向集群中发送一个特殊的更新请求, 内容为 “Change(N)”.

由于更新请求最终都会被递归转发至拓扑中的根节点, 并发的成员管理请求最终可以被正确处理.

4.1.4 负载均衡

完全 N 叉树拓扑可以很容易地从一个已排序的节点列表转换而来. 出于负载均衡的需要, 我们可以针对节点列表中的不同节点生成不同的完全 N 叉树拓扑, 以降低非叶子节点的负载. 目前, 对于节点 i , 我们选择将节点列表进行循环移位操作, 直到 i 变为列表的第 1 位, 之后基于这一新列表构建完全 N 叉树拓扑. 由此对于不同的节点, 完全 N 叉树拓扑的根节点都不同, 从而实现负载均衡. 在使用负载均衡策略后, 当一台服务器节点收到其他服务器传来的请求时, 会根据原始消息来源服务器的下标计算对应的节点列表, 并对应地构建消息来源服务器所属的完全 N 叉树以完成消息传递.

4.1.5 参数选择

对于不同的业务场景, 可能更倾向于提升性能或降低负载. 为解决这一问题, 我们选择引入排队论模型来确定服务器集群和连接到每个服务器的客户端集群所构建的完全 N 叉树拓扑的合适的参数 N . 由于服务器集群规模为 n , 我们可以构建 n 个 $M/M/1/\infty/FCFS$ 队列, 即请求的到达服从泊松过程, 服务时间服从负指数分布, 服务线程个数为 1, 请求队列长度无限, 系统的处理能力无限, 请求的排队规则为先来先服务. 由环境模型指导, 可计算得到集群处理请求的平均速率, 由此得到请求从服务器到客户端的最大和最小传播轮次. 进一步地, 我们采用保守和激进两种策略来确定参数. 保守策略倾向于为每一个完全 N 叉树拓扑选择最小的 N , 在保障更新按时送达的前提下尽可能降低服务器和客户端集群的整体负载. 激进策略倾向于为每一个完全 N 叉树拓扑选择最大的 N , 在可用资源限制条件下尽可能降低配置更新推送的时延.

4.2 两阶段配置更新推送

作为一种改进的发布/订阅机制, 与现有方法不同, 我们使用订阅者 (客户端) 作为中继节点, 使用客户端的部分计算资源加速配置更新推送, 从而提高性能. 如第 2 节中的第 2 个核心观察所述, 我们选择将从属于同一应用或同一数据中心的客户端节点分配到同一台服务器, 从而加速配置更新推送. 这些客户端节点很有可能共享部分配置, 而它们都拥有对这些共享配置的读写权限, 因此可以使用部分计算资源协助推送配置更新. 由此, 我们将这些客户端节点组织成完全 N 叉树, 并遵循前文所述的配置更新推送协议执行. 如图 4 所示, 为了应对不同业务场景的需求, 我们可以为不同的服务器和客户端集群设置不同的拓扑参数. 进一步地, 在拓扑结构中位于上层的客户端节点相比下层节点而言会消耗较多的资源用于配置更新推送, 出于负载均衡的需要, 服务器节点会随机选择拓扑中的客户端节点推送更新. 此外, 由于客户端节点已按照应用和数据中心的粒度来组织, 共享的配置信息只会在特定的完全 N 叉树拓扑中传递, 天然限制了配置更新的推送范围. 特别地, 客户端不会参与未订阅的配置信息的推送. 对于不关心的配置信息, 将会直接在拓扑中将自身标记为不可达, 这一机制避免了带宽浪费.

4.3 容错机制

如前文所述, 拓扑结构在特定节点上的视图即为一棵生成树, 在此基础上我们选择应对节点失效

并解决多分区可用的问题的主要思路是“绕开不可达节点”,即将不可达节点的推送任务交由上层节点完成.如算法 1 所示,基本原理是当某一节点检测到下一跳待推送的节点列表中存在不可达节点时,将不可达节点的子节点加入到这一列表中,递归执行这一操作最终代替失效节点完成推送任务.这一容错机制除了可以应对节点失效之外,也可以保障网络分区时的多分区可用.在出现网络分区时,多分区内的节点的视图不同,但都可以借助“绕开不可达节点”的机制将不可达节点的推送任务交由可达节点完成,最终实现更新在特定网络分区内的推送.

在默认情况下,当数据更新的源节点完成一轮递归的消息传播或者确认消息传播到拓扑中的第 2 层之后向用户返回操作成功.对于不同的业务场景,可调整确认的消息比例以权衡性能和可靠性.例如,对于可靠性较高的场景,可要求消息传播到拓扑中所有非叶子节点之后再向用户返回操作成功.

此外,我们引入了一个简单的心跳检查机制以快速检测不可达节点,避免在配置更新推送时通过超时判断不可达而影响性能.为实现心跳检查机制,每个节点需要维护一个节点健康位图,并配置一个检查间隔 t 和最大退让次数 maxRetry .每个节点每隔 t 时刻向本地拓扑中的邻居节点(即生成树的第 2 层节点)发送心跳探测请求并交换心跳位图,如果收到正确响应,则更新本地位图,并记录心跳信息的来源.如果节点不可达,则将对应节点以及由该节点探测的节点标记为不健康.进一步地,每个节点会持续向本地心跳位图中的不可达节点发送心跳探测请求,并采用指数退让机制,第 1 次间隔为 t ,探测到不可达后间隔变为 $2 \cdot t$,最多退让 maxRetry 次之后维持 $2^{\text{maxRetry}} \cdot t$ 的探测间隔直到节点恢复健康.

4.4 确认和重发机制

我们引入了确认消息位图和对应的重发机制,以实现推送进度跟踪并解决失效和不可达节点恢复后的数据更新重发问题.对于特定的数据更新,每个节点维护该更新在源节点对应的生成树下以该节点为根的子树中的所有节点的确认消息.特别地,在重发更新时,需要根据当前节点与不可达节点间的距离决定重发时间间隔,生成树中距离不可达节点越近(跳数越少)的节点的重发时间间隔越短.例如,假设最短重发间隔为 t ,在拓扑中相隔 1 跳的节点的重发间隔为 t ,相隔 2 跳的节点的重发间隔为 $2t$.

4.5 讨论

本节我们将讨论柔性配置更新推送方法具备的两个性质:

(1) 对于给定的配置更新,所有节点处于以下两种状态之一:已收到更新;即将收到更新.对于给定的配置更新,只要接收到它的服务器节点向用户返回操作成功,即意味着集群中始终维护了对应这一配置更新的确认消息列表,列表中的节点只会处于已确认或未确认的状态.此外,由于确认消息是递归收集的,任意节点的失效都不会影响这一确认消息列表的状态,至多导致节点由于未同步到最新的确认消息列表而向子节点发送重复的配置更新,而这一极端场景只在父节点失效恢复,且重发配置更新的事件发生在子节点重发确认消息的事件之前,带来的性能损耗极为有限.

(2) 所有节点在有限时间内都会收到给定的配置更新.如上文所述,任意一个成功提交的配置更新都会对应一个确认消息列表,由于重发机制的存在,节点只要从故障或网络分区导致的不可达情形中恢复,就会很快收到父节点重发的配置更新.由于系统模型中约定了节点只会出现不可达而不会出现无限期离线的情形,因此可以断定所有节点在有限时间内都会收到给定的配置更新.

5 实验验证

我们修改了阿里巴巴集团研发的开源配置管理系统 Nacos, 添加了基于完全 N 叉树拓扑的配置更新推送机制并从如下维度验证我们提出的方法的效果:

- (1) 与现有的基于一对多广播及反熵算法相比, 我们的方法是否能在大规模集群上实现按时推送并提升推送性能?
- (2) 对于不同的树型拓扑构建策略, 我们的方法是否可以提升性能并降低节点负载?
- (3) 我们的方法是否具备应对节点失效的能力? 是否可以保障网络分区时的多分区可用?

5.1 原型实现

我们基于 Nacos 的内核实现了原型系统, 修改了其中负责服务发现的 `nacos-naming` 模块中的一致性协议相关部分并抽取相关核心代码作为服务器端。此外, 我们构建了一个轻量级的客户端用于订阅配置更新。在实验过程中, 我们将一个服务器节点和多个客户端节点放置在同一个阿里云 ECS 实例中。每一个服务器节点使用独立线程处理请求。此外, 为了限制客户端节点的最大资源使用率, 我们使用一个线程分时复用的方式承载 20 个客户端节点。在通信机制方面, 我们部署了轻量级的 HTTP 服务器, 节点间通讯使用 HTTP 协议实现。此外, 在实验过程中我们为每一个服务器和客户端节点预先设置了唯一标识符。

尽管我们的方法和目前的原型实现主要关注百万量级容器实例集群场景下的柔性配置更新推送, 这一方法同样可以较容易地推广到其他类似场景, 例如由虚拟机组成的集群, 或是推送其他类型的小规模数据。

5.2 实验设计

我们使用 100 个阿里云 ECS 实例以构建测试床。每个实例具有 4 个虚拟 CPU, 8 GB 内存和 40 GB 的高效云盘。我们使用 20 个 ECS 实例构建了由 200 个服务器节点和 2×10^5 个客户端节点组成的规模相对较小的集群, 使用 100 个 ECS 实例构建了由 200 个服务器节点和 10^6 个客户端节点组成的规模较大的集群。每个 ECS 实例上最多承载了 10 个服务器端线程和 500 个客户端线程, 即 10 个服务器节点和 10000 个客户端节点。

实验场景的具体配置如下: 我们选择模拟配置管理的一个典型业务场景, 即用户出于业务需要对某一服务进行扩容, 向该服务的 IP 列表中添加节点, 并希望这一更新可以推送至所有订阅了这一服务 IP 列表的客户端。在这一场景中, 请求的平均大小为 2 KB。我们在服务器节点间通讯时注入 1 ms 的网络时延, 在服务器节点和客户端节点通讯时注入 3 ms 的网络时延。考虑到我们在 1 个 ECS 实例上承载了 10 个服务器节点和 10000 个客户端节点, 我们将最大“一致性窗口”设置为 30 s, 且更新需要被推送至 99% 的可达节点。此外, 用户请求到来服从参数 $\lambda = 10$ req/s 的泊松分布。

5.3 推送性能验证

在底层推送拓扑方面, 我们将柔性配置更新推送方法与现有的一对多广播以及反熵算法的随机拓扑进行对比, 以验证我们的方法的性能表现。我们将配置更新推送的时延定义为用户发起更新请求到集群中至少 99% 的可达客户端收到更新的时间。在这一实验过程中, 我们使用激进策略来选择 N , 并且使用两阶段推送策略。

我们首先在由 2×10^5 个客户端组成的相对小的集群中验证柔性配置更新推送方法的性能。结果

图 5 (网络版彩图) 2×10^5 个客户端节点构成的大规模集群的推送性能Figure 5 (Color online) Performance evaluation in a large-scale cluster with 2×10^5 clients 图 6 (网络版彩图) 10^6 个客户端节点构成的大规模集群的推送性能Figure 6 (Color online) Performance evaluation in a large-scale cluster with 10^6 clients

如图 5 所示. 实验结果表明, 与反熵算法相比, 柔性配置更新推送方法的平均时延降低了 24.5%, 99% 分位的时延降低了 27.9%. 与一对多广播拓扑相比, 柔性配置更新推送方法的平均时延降低了 43.1%, 99% 分位的时延降低了 45.8%. 考虑到前文对最大“一致性窗口”的限制, 可以发现在相对小规模集群场景下, 一对多广播的底层拓扑难以保障更新按时送达, 而与反熵算法相比, 柔性配置更新推送方法的性能更优.

我们之后在由 10^6 客户端组成的大规模集群中验证柔性配置更新推送方法的性能. 结果如图 6 所示. 实验结果表明, 与反熵算法相比, 柔性配置更新推送方法的平均时延降低了 41.3%, 99% 分位的时延降低了 41.7%. 与一对多广播拓扑相比, 柔性配置更新推送方法的平均时延降低了 74.7%, 99% 分位的时延降低了 76.5%. 可以发现在这一大规模集群场景下, 一对多广播和反熵算法的底层拓扑都难以保障更新按时送达, 柔性配置更新推送方法的性能表现更优. 此外, 集群规模越大, 柔性配置更新推送方法的性能优势越明显.

5.4 容错能力验证

本小节我们将在 2×10^5 个客户端组成的集群中验证柔性配置更新推送方法的容错能力. 本小节所述的场景中有 20% 的服务器和客户端节点不可达, 以此模拟节点失效和网络分区. 尝试连接到不可达服务器的客户端节点会随机选取一个新的服务器节点连接并接收更新. 在这一实验过程中, 我们使用保守策略来选择 N , 并且不使用两阶段推送策略. 实验结果如图 7 所示. 实验结果表明, 与节点全部可达的场景相比, 99% 分位的推送时延增加了 34.1%, 平均时延增加了 28%. 特别地, 10% 分位的时延增加了 2 倍. 这一开销可能是由于客户端节点重新选择新的可达服务器节点而导致的. 但在这一场

图 7 (网络版彩图) 存在 20% 不可达节点时的配置更新推送性能

Figure 7 (Color online) Update delivery in a cluster with 20% nodes unreachable

图 8 (网络版彩图) 使用不同参数的性能表现

Figure 8 (Color online) Performance of using different parameters

景下, 柔性配置更新推送方法仍然保障了更新按时到达, 说明我们的方法可以较好地应对节点失效和网络分区场景.

5.5 使用不同策略时的性能表现

本小节我们将改变柔性配置更新推送方法中的不同维度的参数并验证我们的方法是否可以保障配置更新按时送达.

首先我们将验证选取不同底层拓扑参数时的表现, 主要分为保守 (选取尽可能小的 N , 在满足推送时间限制的条件下降低节点负载) 和激进 (选取尽可能大的 N , 在节点资源限制的条件下尽可能提升推送性能) 两种参数选择策略. 实验结果如图 8 所示. 可以发现, 使用激进策略时配置更新推送的 99% 分位的时延降低了 7.9%, 但两者的平均推送时延类似. 进一步的分析结果表明, 采用保守策略和激进策略时配置更新推送的平均跳数类似, 最大跳数相差不大, 这也导致两者拓扑中的叶子节点数目差距不大, 因此采用激进策略时整体性能提升有限. 然而, 对于更大规模的集群和拓扑, 减少一跳可以带来相对可观的性能提升.

进一步地, 我们在由 10^6 客户端节点的集群中验证使用客户端的部分计算资源进行两阶段推送的性能表现. 实验结果如图 9 所示. 实验结果表明, 采用两阶段推送相比只使用服务器资源推送而言, 99% 分位的推送时延降低了 21.9%, 平均时延降低了 25.6%. 可以发现采用两阶段推送, 将连接到同一台服务器的客户端节点组织成树型拓扑可以有效降低服务器负载并提升性能.

图 9 (网络版彩图) 两阶段推送的性能表现

Figure 9 (Color online) Performance of using different phases

6 相关工作

以 Paxos^[1] 变种和 Raft^[2] 为代表的共识算法可以保障顺序一致性, 但对于大规模集群的配置更新推送而言这类保障机制过强. 这类算法需要依赖单一领导者 (leader) 节点来唯一确定更新操作的执行顺序, 且依赖领导者节点获得最新的数据副本, 因此所有读写请求都会经过领导者节点. 此外, 在出现网络分区时, 只有节点数过半的网络分区可以正常完成选举操作, 其他网络分区中的节点不可用. 从底层拓扑角度看, 这类算法默认使用一对 N 的广播, 当集群规模增大时领导者节点的负载也会急剧升高. 因此, 这类算法的领导者节点存在单点瓶颈, 约束了集群的横向扩展能力, 无法适用于百万乃至千万量级的集群.

以 Gossip^[3,4,9~11] 等反熵算法为代表的最终一致性的算法初始传播速度较快, 但由于每轮传播的目标节点是随机选取的, 算法本身存在较为严重长尾现象, 在大规模集群场景下收敛速度难以控制. 此外, Gossip 本身只关心随机分发, 并不关心消息是否被可靠接收, 并发的更新是否被合理处理, 因此通常用于分布式数据库管理^[3,12], 或同步集群内的健康检查信息或节点的状态信息⁵⁾⁶⁾, 难以用于大规模集群的配置更新推送.

Facebook 提出了一种新的发布/订阅方法, LAD⁷⁾⁸⁾, 同样使用了树作为底层推送拓扑. 与我们提出的柔性配置更新推送方法不同, 它使用 ZooKeeper 作为数据存储, 使用配置分发代理将订阅相同配置的客户端节点组织在树型拓扑结构中. 此外, 它的节点健康管理机制与容错机制也与我们的方法有所不同.

现有的发布/订阅系统, 如 Wormhole^[5] 等通常关注如何将发布者与订阅者解耦合, 并针对发布/订阅流程本身进行优化. 它并未使用订阅者的计算资源用于推送配置更新. 为了提升发布/订阅性能和可靠性, 反熵算法也被经常用于基于内容的发布/订阅系统^[13]. 但这类方法本身并不保证更新可以及时送达. 一些方法使用多播^[14,15] 的方式消减不必要的消息传递开销, 并关注如何高效地将节点组织成多播组. 此外, 基于过滤器^[16,17] 和基于地址^[6] 的路由也经常用于消减发布/订阅系统中的冗余消息以提升性能. 与这些方法不同, 我们的方法使用改变服务器集群和客户端集群的拓扑参数 N 的方式控制拓扑结构, 以保证更新按时送达.

5) Apache. Apache Cassandra. <https://cassandra.apache.org/>.

6) HashiCorp. Consul by HashiCorp. <https://www.consul.io/>.

7) Facebook. Location-aware distribution: configuring servers at scale. <https://engineering.fb.com/data-infrastructure/location-aware-distribution-configuring-servers-at-scale>.

8) Facebook. Systems @scale 2018: scaling data distribution at Facebook using LAD. <https://atscaleconference.com/videos/systems-scale-2018-scaling-data-distribution-at-facebook-using-lad/>.

现有的分布式事件分发^[18,19]的相关工作通常关注事件分发过程中的消息模型、路由机制, 以及QoS保障等方面, 未能充分考虑数据副本的一致性保障, 即如何确保事件最终可以按时、可靠送达. 我们的方法与这类工作的最大区别在于侧重于保障最终一致性, 确保网络分区时的多分区读写可用, 并保障配置更新可以最终送达.

7 结论与未来工作

我们提出了适用于百万量级客户端集群的柔性配置更新推送方法, 引入了柔性的两阶段配置更新推送机制, 结合完全 N 叉树拓扑以降低配置更新推送的时延及节点负载. 此外, 柔性配置更新推送方法可应对节点失效场景, 并且在网络分区时保证多分区读写可用. 拓扑参数和具体策略可根据业务场景不同进行调整, 以满足性能和可靠性等维度的需求.

未来我们将探索如何引入自适应机制以应对负载的动态变化, 并自动选择合适的拓扑参数和策略, 从而使方法适用于更广泛的业务场景. 此外, 我们也将针对客户端订阅配置的具体情况, 考虑对客户端拓扑结构进行优化, 以进一步降低带宽开销并减少不必要的容错操作以提升性能. 在冲突解决机制方面, 我们也将尝试优化现有方法以尽可能少地通过交由业务层重发的方式解决冲突.

参考文献

- 1 Lamport L. The part-time parliament. *ACM Trans Comput Syst*, 1998, 16: 133–169
- 2 Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: *Proceedings of USENIX Annual Technical Conference (USENIX ATC'14)*, 2014. 305–319
- 3 Demers A, Greene D, Hauser C, et al. Epidemic algorithms for replicated database maintenance. In: *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*, 1987
- 4 Birman K P, Hayden M, Ozkasap O, et al. Bimodal multicast. *ACM Trans Comput Syst*, 1999, 17: 41–88
- 5 Sharma Y, Ajoux P, Ang P, et al. Wormhole: reliable pub-sub to support geo-replicated Internet services. In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, 2015. 351–366
- 6 Ji S P, Ye C Y, Wei J, et al. MERC: match at edge and route intra-cluster for content-based publish/subscribe systems. In: *Proceedings of the 16th Annual Middleware Conference (Middleware'15)*, 2015. 13–24
- 7 Kulkarni S S, Demirbas M, Madappa D, et al. Logical physical clocks. In: *Proceedings of International Conference on Principles of Distributed Systems (OPODIS'14)*, 2014. 17–32
- 8 Demirbas M, Leone M, Avva B, et al. Logical physical clocks and consistent snapshots in globally distributed databases. 2014. <https://cse.buffalo.edu/tech-reports/2014-04.pdf>
- 9 Lin M J, Marzullo K. Directional gossip: gossip in a wide area network. In: *Proceedings of the 3rd European Dependable Computing Conference on Dependable Computing (EDCC-3)*, 1999. 364–379
- 10 Eugster P T, Guerraoui R, Handurukande S B, et al. Lightweight probabilistic broadcast. In: *Proceedings of International Conference on Dependable Systems and Networks (DSN'01)*, 2001. 443–452
- 11 Eugster P T, Guerraoui R, Handurukande S B, et al. Lightweight probabilistic broadcast. *ACM Trans Comput Syst*, 2003, 21: 341–374
- 12 Demers A, Greene D, Houser C, et al. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Syst Rev*, 1988, 22: 8–32
- 13 Costa P, Migliavacca M, Picco G P, et al. Introducing reliability in content-based publish-subscribe through epidemic algorithms. In: *Proceedings of the 2nd International Workshop on Distributed Event-based Systems (DEBS'03)*, 2003
- 14 Riabov A, Liu Z, Wolf J L, et al. Clustering algorithms for content-based publication-subscription systems. In: *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, 2002. 133–142
- 15 Opyrchal L, Astley M, Auerbach J, et al. Exploiting IP multicast in content-based publish-subscribe systems. In: *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'00)*, 2000. 185–207

- 16 Li G L, Hou S, Jacobsen H-A. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05), 2005. 447–457
- 17 Li G L, Muthusamy V, Jacobsen H-A. Adaptive content-based routing in general overlay topologies. In: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware'08), 2008
- 18 Mahambre S P, Kumar M, Bellur U. A taxonomy of QoS-aware, adaptive event-dissemination middleware. IEEE Int Comput, 2007, 11: 35–44
- 19 D'Angelo G, Ferretti S, Marzolla M. Adaptive event dissemination for peer-to-peer multi-player online games. In: Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools'11), 2011. 312–319

Flexible configuration update delivery for large clusters

Zhen TANG^{1,2}, Wei WANG^{1,2,3*}, Yu HUANG^{4*}, Yanlin LI⁵, Shuping JI⁶, Ao SONG^{1,2}, Jun WEI^{1,2} & Tao HUANG^{1,2}

1. State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China;

2. University of Chinese Academy of Sciences, Beijing 100049, China;

3. Institute of Software Technology, Chinese Academy of Sciences, Nanjing 211135, China;

4. State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China;

5. Alibaba Group, Hangzhou 311121, China;

6. Electrical & Computer Engineering, University of Toronto, Toronto ON M5S 3G4, Canada

* Corresponding author. E-mail: wangwei@otcaix.iscas.ac.cn, yuhuang@nju.edu.cn

Abstract Efficient configuration management plays an important role in managing large-scale clusters in public cloud services. Because a cluster may contain millions of containers, it is challenging to guarantee that configuration updates will be delivered to interested containers reliably and in time to meet the requirements of different scenarios. Existing solutions to this problem have limitations: consensus algorithms limit the scalability of the cluster and may not work for large clusters; epidemic algorithms face the challenge of long-tail latency, which means the overall response time is too long to deliver critical configuration updates effectively. To overcome the limitations of existing solutions, in this paper, we present a novel flexible approach for delivering configuration updates for large-scale clusters. This pub/sub approach uses a configurable, complete N -ary tree as the overlay and introduces flexible, two-phase configuration update delivery. This method of update delivery uses a portion of subscribers' resources to improve its performance. Furthermore, it is fault-tolerant when it encounters node failures and network partitions. The strategies and the parameters of the overlay can be changed to meet performance and reliability requirements for different scenarios. Evaluations show that our approach significantly reduces the latency of update delivery compared to existing solutions. It also performs well in cases of node failures and network partitions.

Keywords update delivery, configuration management, complete N -ary tree

Zhen TANG was born in 1990. He received his Ph.D. degree from Institute of Software, Chinese Academy of Sciences in 2019, and his B.E. degree from Nanjing University of Science and Technology in 2012. Currently he is an assistant research fellow in Institute of Software, Chinese Academy of Sciences. His research interests include distributed systems and storage virtualization.

Wei WANG was born in 1982. He received his M.S. and Ph.D. degrees from Institute of Software, Chinese Academy of Sciences in 2007 and 2010, respectively. Currently he is a professor in Institute of Software, Chinese Academy of Sciences. His research interests include distributed computing and performance management.

Yu HUANG was born in 1982. He received his B.Sc. and Ph.D. degrees in computer science from the University of Science and Technology of China in 2002 and 2007, respectively. He is currently a professor with the Department of Computer Science and Technology at Nanjing University. His research interests include distributed algorithms, distributed systems, formal methods, and system reliability.

Yanlin LI was born in 1989. He is a staff engineer at Alibaba Group. He is a member of Nacos Project Management Committee and is in charge of Alibaba Cloud's Application Configuration Management and Microservice Engine products. He is focusing on the evolution of cloud-native architecture to support millions of instances for Alibaba.