计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2021.20200501
58(3):497-512,2021

集中式集群资源调度框架的可扩展性优化

毛安琪　汤小春　丁　朝　李战怀
(西北工业大学计算机学院　西安　710129)
(工信部大数据存储与管理重点实验室(西北工业大学)　西安　710129)
(maoanqi@mail.nwpu.edu.cn)

Scalability for Monolithic Schedulers of Cluster Resource Management Framework

Mao Anqi, Tang Xiaochun, Ding Zhao, and Li Zhanhuai
(*School of Computer Science, Northwestern Polytechnical University, Xi'an* 710129)
(*Key Laboratory of Big Data Storage and Management (Northwestern Polytechnical University), Ministry of Industry and Information Technology, Xi'an* 710129)

Abstract The significant advantages of monolithic cluster resource management system in ensuring the consistency of global resource status and applying multiple scheduling models make it widely used in actual systems. Howerver, the performance of the monolithic resource manager in a large cluster management environment does not meet expectations, because it uses a single node to maintain the global resource state. When the resource manager is receiving and processing large-scale periodic heartbeat information, the load pressure on the resource manager will increase sharply, which leads to a scalability bottleneck. In order to solve these problems, this paper proposes the idea of "no change, no update" to replace the periodic update mechanism of the resource manager. In our paper, we briefly summarize three main topics. Firstly, we introduce a differential-based heartbeat information processing model in the computing node. When the resource status of the computing node has not changed, it will not send the message to the resource manager, thereby reducing the size and number of messages. Secondly, we propose a ring network monitoring model between computing nodes. By adopting this mode, the periodic monitoring pressure can be transferred to the computing nodes. Finally, we implement these two models on YARN. After experimental verification, we can conclude that when the cluster reaches 10 000 nodes and the heartbeat interval is 3 s, the YARN based on our models increases the heartbeat information processing efficiency and resource update efficiency by about 40%. In addition, the scale of the cluster managed by improved YARN is more than 1.88 times that of the original YARN.

Key words monolithic schedulers; scalability; heartbeat message; differential; ring monitoring

摘　要　集中式集群资源管理系统既能够确保全局资源状态的一致性亦拥有多种调度模型,因此被广泛应用于实际系统中.但是,当集中式资源管理器在接收并处理大规模的周期性心跳信息时,由于其采用单一节点来维护全局资源状态,所以资源管理器的负载压力急剧增加,导致调度能力降低,影响了集群系统的可扩展性.针对上述问题,提出一种"没有变化就不更新"的思想,取代集中资源管理的定时更新

收稿日期:2020-06-19;**修回日期**:2020-12-09
基金项目:国家重点研发计划项目(2018YFB1003400)
This work was supported by the National Key Research and Development Program of China (2018YFB1003400).
通信作者:汤小春(tangxc@nwpu.edu.cn)

机制,改善了集中式资源管理系统的可扩展性.首先,通过计算节点引入基于差分的心跳信息处理模型,使得未发生状态变化的节点不必发送心跳消息,从而减少消息发送的规模和次数;其次,针对节点宕机监测过程,提出基于环形监视的节点监控模型,让各个计算节点之间互相监视对方的宕机状态,从而将周期性监测压力转移到计算节点;最后,给出这2种模型在集中式资源管理系统YARN上的实现,并针对改进前后的系统进行实验测试.通过实验验证,当集群达到1万个节点且心跳时间间隔3 s时,改进后YARN系统的心跳信息处理效率以及资源更新效率相比原YARN系统提高40%左右.另外,改进后YARN系统管理集群节点规模相比原YARN系统扩大1.88倍以上.

关键词 集中式调度;可扩展性;心跳消息;差分;环形监控

中图法分类号 TP311

随着大数据处理技术的发展,数据内在价值的分析与挖掘成为这个时代的研究热点之一.当数据的体量以惊人的速度增长时,单台机器已经无法满足大数据的存储与处理,于是以Hadoop分布式文件系统(Hadoop distributed file system, HDFS)为代表的分布式文件系统应运而生.它以主从架构管理模式,将数据划分到多台机器节点上存储,形成一个集群系统,也称数据中心,推动了大数据计算的发展.在企业或组织内,数据中心上通常会存在各种不同类型的数据处理应用程序.如果为每种数据处理应用程序建立专用的集群基础设施,不但为企业带来巨大的建设成本,也导致数据在各个集群之间来回复制,容易产生错误.因此,越来越多的企业开始在数据中心上部署集群资源管理系统,确保各种各样的数据处理应用程序共享一套硬件基础设施.

集群资源管理系统的主要功能是管理集群的可用计算资源并分配给不同的数据处理应用程序.集群计算资源的分配要按照一定的策略进行,保证各个应用程序独自使用集群的部分资源,避免各个数据处理应用程序之间的资源竞争和相互干涉,实现各个应用程序对资源的公平使用.

现有的集群资源管理系统种类非常丰富,如文献[1-43]都是有关集群资源管理和用户应用程序调度的系统.从集群资源调度框架的观点看,集群资源管理系统分为3个大类,即集中式资源调度框架、分布式资源调度框架以及混合式资源调度框架.集中式资源调度框架又分为单一资源调度和二级资源调度模型,前者采用"拉"的模式,后者采用"推"的模式.文献[1-25]研究单一资源调度模型,文献[26-34]研究二级资源调度模型.文献[35-40]研究分布式调度框架,文献[41-43]研究集中分布混合的资源调度框架.文献[22]是一种典型的单一资源调度模型,分3步完成资源的分配:应用程序向资源管理主节点请求计算资源;资源管理主节点负责资源的分配;应用程序提交任务执行.文献[34]是一种二级调度模型,也分为3步完成资源的分配:应用程序按照次序向资源管理框架请求资源;资源管理器按照各个应用程序使用资源情况,依次为各个应用程序分配计算资源;应用程序使用资源执行任务.文献[38]是分布式资源调度框架,分2步完成资源分配:各个应用程序独立向请求发出"采样请求";根据采样结果提交任务.文献[42]是集中分布混合的资源调度框架,短时间的应用程序使用分布式调度框架,长时间应用程序采用集中式调度框架.

文献[44]指出,集中式资源调度框架的优点是调度实现简单,所以应用广泛,但是集中式调度框架存在着可扩展的瓶颈.相比于集中式调度框架,分布式资源调度框架不存在单一的资源管理节点,所以在可扩展性方面具有较大的优点,但是分布式资源管理缺乏集中管理,增加了资源状态的同步和并发控制的难度,也无法做到最优全局决策与资源控制,所以真正应用的系统较少.混合式资源管理框架结合了2个系统,由于存在集中式资源监视,虽然在一定程度上缓解了分布式调度的不一致问题,但是其集中式调度和分布式调度相互独立,需要维护2套代码,维护代价较大.另外,分布式调度和集中式调度相互预留资源,影响总体资源利用率,因此混合式资源管理框架处于研究状态,很少有成熟的系统广泛使用.

从以上的研究文献看,80%以上的集群资源管理模型采用集中式,如果再加上混合式中的集中式调度模型,集中式资源调度框架可以达到85%左右.一些大的互联网公司,例如微软、腾讯、百度等,目前也使用集中式资源调度框架来处理自己的大数据业务.但是,集中式资源调度框架中,资源管理器为了获得集群可用资源,需要集群上的每个计算节点

周期性地发送健康状态信息以及容器状态信息到集中式调度器,以便监控整个集群计算节点的健康状态以及整个集群节点上容器(任务)的运行情况.当计算节点的数量扩大到一定的规模后,大量心跳信息的存储和处理给资源管理器带来较大的负载压力.另外,网络延迟以及通信次数的快速增长使得集群资源管理中的调度器无法在调度许可的时间内处理全部的心跳信息,造成调度过程的失效.

随着集群规模的增大,集中式资源管理器很快达到心跳信息处理上限,从而限制了集群节点规模的横向扩展.所以,当集群节点的规模较大时,集中式集群资源管理通常采用增加心跳的时间间隔的方法来减少心跳发生的频率,降低心跳信息的产生次数.但是心跳时间间隔的增大,会导致资源状态的变化不能及时地汇报到调度器,一边是资源的空闲,另一边是用户作业需要等待更长的时间才能被调度.文献[45]在备用节点上开启资源监控服务提高大规模心跳信息的处理效率,并借助分布式存储服务(MySQL NDB)对心跳信息一致性进行保证.虽然文献[45]的测试结论中,使用模拟器可以将 YARN 支持的机器数量从 4 000 台扩展到 7 500 台,但是其引入了 C++ 版本的 MySQL 数据库以及 NDB 事件流库,使用 JNI 与 YARN 进行耦合,增加了代码的维护代价,同时 MySQL NDB 需要增加多个资源监视节点,不但增加了硬件代价,也增加了数据一致性检测代价.

集中式资源调度框架的易用性和可扩展性是一对矛盾.一面是大量用户使用集中式资源管理系统,一面是集中式资源管理可扩展的困境.基于这两方面原因,对集中式资源管理系统进行可扩展的优化研究有一定的迫切性,而且也实实在在地符合用户的意愿.因此,本文对集中式资源管理系统的可扩展性进行了优化,改善了集中式资源管理系统的可扩展性.

论文的主要贡献有 3 个方面:

1) 提出了基于差分模式的资源状态监控模型,减少了心跳信息的大小和规模,减轻了资源调度器的存储和处理压力,使得调度器可以处理更多计算节点的消息;

2) 提出了基于环形监视的节点监控模型,将调度器的监控功能分散到各个计算节点,缓解了调度器的处理压力;

3) 对 YARN[22] 系统的资源管理功能进行了优化,并使用 YARN 提供的模拟器进行了可扩展测试,实验证明我们的方法可以使集群的规模提高 1.88 倍以上.

1 可扩展资源管理系统模型

本节首先针对集中式资源监控模块的流程进行了分析,揭示了可扩展性问题.然后,提出基于差分模型的心跳信息处理流程,在计算节点上对心跳信息进行预处理.最后提出了基于环形监视的节点监控过程.

1.1 资源监控服务的性能瓶颈分析

集中式资源调度框架有 3 个主要的服务:1)应用程序管理服务,用户的资源请求由应用程序管理服务提交到资源调度器,请求分配计算资源来运行任务;2)资源调度器,集群计算资源上运行着许多应用程序,而每个应用程序会包含大量的任务,这些任务会分散在不同的计算节点上并行运行,资源调度器的任务是将不同应用程序的任务映射到集群计算节点上并执行任务;3)资源监控服务,它是调度器与计算节点之间连接的纽带,负责接收集群计算节点的加入和退出消息,同时周期性捕获计算节点的心跳信息,即将各个集群节点的容器(任务)状态信息进行汇总处理后通知调度器,更新集群资源的最新状态.这些更新信息对资源调度器来说非常重要,它们是资源分配的依据.

当集群计算节点注册到资源管理器后,便周期性向集群资源管理的资源监控服务汇报心跳消息.心跳信息包含 2 个重要的内容,即机器的健康状态信息和集群节点上全部任务的运行状态信息.健康状态信息主要用于检测资源的使用情况以及评价系统的性能等.计算节点的全部任务运行状态信息用来维护资源管理器中的全局资源视图.另外,任务状态的变化决定了资源可用状态的变化.因此,周期性的心跳信息使得资源管理器与计算节点之间保持联系,并保证集群资源与任务状态的一致性.

当心跳信息到达资源监控服务后,首先判断计算节点的健康状态.若该计算节点为非健康状态,则通知调度器减少相关计算节点的可用资源配额,防止进一步将该机器的资源分配给其他计算任务.否则,资源监控服务就开始检查该节点上任务运行状态,通过与上次心跳信息的对比,提取出新启动的任务与运行结束的任务相关信息.如果存在新启动任务和结束任务,说明任务状态有变化,资源监控服务产生一个更新事件通知给资源调度器,资源调度器将相关任务运行状态进行更新维护,并对集群资源

全局视图进行更新.另外,对于已经结束的任务,将任务的结束状态保存到日志中,这些信息是资源分配模块进行系统优化时的参考依据.

在计算节点心跳汇报以及资源监控服务处理心跳这2个过程中,资源管理的资源监控服务要承受的通信量以及计算量将会给资源管理器带来较重的负载压力.随着集群规模的增大,该问题将更加明显.究其原因,有2个方面问题:1)当集群节点上无运行的任务,仍会周期性向资源监控服务发送心跳信息,以便记录心跳更新时刻,作为新的心跳信息处理依据.这些信息仅能代表节点处于正常运行中,并没有实质的数据传输.它不但增加了集群节点内部的通信开销,也增加了资源监控服务的处理开销.2)集群计算节点只是按心跳时间间隔,定期将本机所有容器以及任务状态等相关信息汇报给集群资源监控服务.如果任务的运行时间较长,心跳间隔期间状态没有发生变化,无变化信息也会被汇报到资源监控服务,那么这将给资源监控服务增加额外的工作量,浪费计算资源.

综上,当集群计算节点进行大规模扩展后,资源监控服务需要处理大量的无用心跳信息,导致负载过重,无法及时处理应用程序的资源分配请求,延迟了调度器的资源分配.而扩大心跳间隔虽然减少了单位时间内心跳信息的发送和处理,却导致计算节点的有用状态信息无法被及时更新,使得这些计算节点的资源处于闲置状态.因此,减少计算节点与资源监控服务之间的通信次数,降低资源监控服务的数据处理强度,是集中式资源管理横向扩展的主要策略.

1.2 基于差分的心跳信息处理模型

鉴于资源监控服务承担着太多的心跳信息处理,本文提出一种策略,将心跳信息处理功能进行分解,一部分功能由计算节点承担,一部分功能由资源监控服务承担,即:让计算节点承担容器状态的过滤功能以及健康状态的检测,资源监控服务只承担容器状态和健康状态的更新,降低资源监控服务的负荷.图1给出了基于差分的心跳信息处理模型,具体流程如下描述.

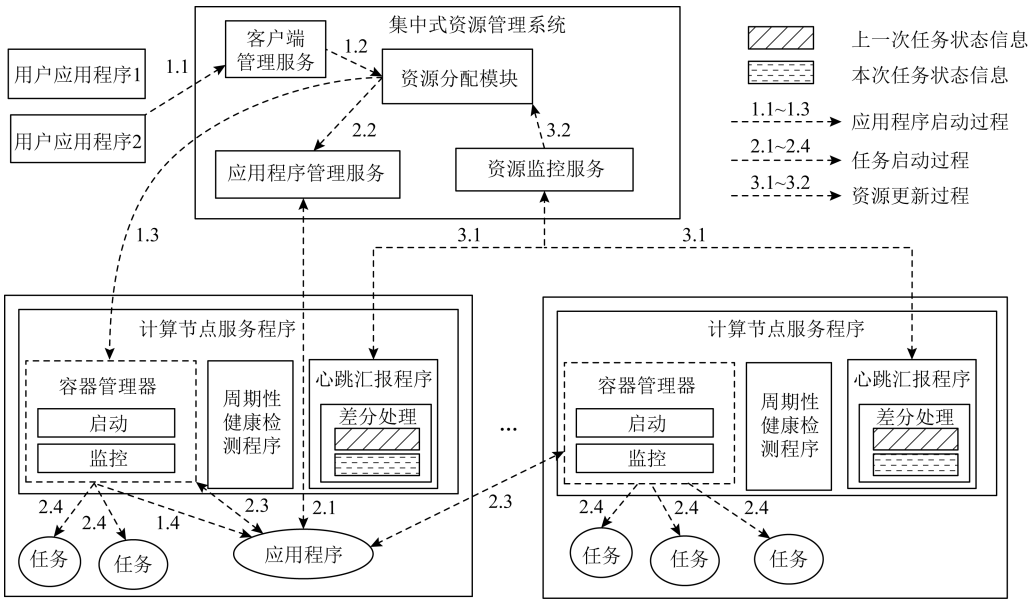


Fig. 1 A differential-based heartbeat information processing model

图1 基于差分的心跳信息处理模型

用户应用程序向集中式资源管理器的客户端管理服务进行注册并提交应用程序,如图1中的1.1.资源分配模块为该计算框架分配第1个容器资源用于运行用户应用程序,并通知计算节点的容器管理服务启动该应用程序,如图1中的1.2~1.4.

应用程序成功启动后,向应用程序管理服务注册并为其包含的任务申请资源,如图1中的2.1.应用程序管理服务从资源分配模块读取资源的分配结果返回给应用程序,如图1中的2.2.应用程序拿到资源后,将资源与任务映射匹配后,通知各计算节点容器管理服务,如图1中的2.3.计算节点容器管理服务接收到启动容器消息后,启动容器并配置任务所需的运行环境,监控与管理任务的生命周期,如图1中的2.4.心跳汇报程序首先从周期性健康检测

程序中读取健康状态进行保存,然后与上次所缓存的健康状态进行对比,获得健康状态的差分值.另外,心跳汇报程序还需要读取当前任务相关状态信息时,并与上一次缓存的任务状态进行对比,得到有任务状态变化的差分值.如果差分值不为空,则向资源监控服务发送心跳信息,如图 1 中的 3.1;否则,省略本次心跳汇报过程.

资源监控服务收到心跳信息后,无需进行过滤处理,直接将本次接收的任务状态信息更新,并通知资源分配模块进行资源更新,如图 1 中 3.2.资源分配模块收到通知后,对相关资源进行更新维护,并通过一定资源分配方法将可用资源分配给应用程序,等待应用程序资源请求并领取资源.

1.3 基于环形监视的节点监控模型

定时发送心跳信息是资源管理器判定计算节点正常运行的依据,如果长时间不发送心跳信息,资源管理器就可以认为节点出现故障,将该节点纳入不可用计算节点列表中.基于差分的心跳信息处理模型中,任务状态或健康状态不发生变化就不产生心跳信息,这可能会导致资源管理器错误判断计算节点的运行状态.由于计算节点在不发生状态变化的情况下取消心跳信息的发送,从而使得资源管理器无法辨别长期不发送心跳信息的节点是宕机还是计算节点无状态变化.例如,在运行过程中某个计算节点宕机后,宕机节点就不能汇报任务的状态导致应用程序无法得到任务的最新状态,必须等待超过设定的超时范围后,应用程序才会终止该任务并重新申请资源,进而影响相关作业的执行效率.

针对该问题,本文提出将所有的集群计算节点组成对等的环形监视网络,处于环形网络的每个计算节点都有一个前驱节点和后继节点.当前计算节点向自己的后继节点定时发送自己的心跳消息.对于后继节点,发送消息的就是它的前驱节点.如果后继节点在设置的时间间隔内没有收到前驱节点的心跳消息,就认为前驱节点出现故障,立即向资源监控服务发送该节点宕机的信息描述,以便资源管理器做出相应的处理.当存在新的节点加入或者一个节点出现故障后,系统需要重组对等环形监控网络,保证集群节点的正常运行,实现节点监控心跳信息的及时收集和发送.

针对对等环形监控网络,资源管理器对集群中的计算节点分配一个唯一的机器编号,编号按照从小到大的顺序产生.这些信息被各个计算节点维护,当新加入计算节点或者节点故障退出时,需要同步这些信息.节点的生存状态发送过程按照节点的机器编号大小来进行,即编号为 N 的节点接收编号为 $N-1$ 的节点的心跳消息(第 1 个节点接收最后一个节点的心跳消息).当节点 $N-1$ 出现故障时,它既不能接收消息,也无法发送消息.当心跳周期到达期间,节点 N 未收到节点 $N-1$ 的生存状态汇报消息,则立即将节点 $N-1$ 故障的消息通知资源管理器,资源管理器则将计算节点设置为不可用状态,并通知各个集群节点更新缓存的机器编号信息.当新节点加入或者故障节点从失效状态恢复正常,这些节点向资源管理器发送注册消息,资源管理器从机器编号缓冲中获得一个编号分配给当前节点,同时通知各个集群节点更新缓存的机器编号信息.计算节点接收到这些消息后,则重新设置自己接收心跳消息和发送心跳消息的节点,维持一个完整的对等环形监控网络.

当计算节点宕机时,环形监控处理流程如图 2 所示.图 2 中 1.1 表示集群有 5 个节点组成为环形监控网络.如果节点 4 宕机,则节点 5 在当前心跳时间内未收到节点 4 的生存状态汇报信息,则默认该计算节点不可用,并向资源监控服务汇报监视节点的故障信息,如图 2 中 2.1.资源监控服务收到节点 5 汇报信息后,将节点 4 从监视列表中删除,并释放节点 4 编号,同时通知集群节点更新监视列表,如图 2 中 2.2~2.5.当计算节点收到监视列表更新通知后,重新设置监视节点与汇报节点.因此在下一次心跳周期到达后,节点 3 向节点 5 汇报心跳消息,同时,节点 5 等待节点 3 的心跳汇报,如图 2 中 2.6.最终节点 1、节点 2、节点 3 以及节点 5 组成了新环形监控结构.

当新的计算节点加入时,环形监控处理流程如图 3 所示.图 3 中 1.1 表示有 4 个节点组成了环形监控网络,即节点 1、节点 2、节点 3 以及节点 5.当一个节点向资源监控服务注册节点信息,如图 3 中 2.1.资源监控服务检查节点管理列表,找到 1 个可用编号 4 号为该节点分配,然后更新监视列表,并通知集群节点更新监视列表,如图 3 中 2.2~2.5.当计算节点收到节点列表更新通知后,重新设置监视节点与汇报节点.因此在下一次心跳周期到达后,节点 3 向节点 4 发送心跳信息,且节点 4 向节点 5 发送心跳信息,如图 3 中 2.6.最终这 5 个节点重新组成了环形监控结构.

在传统的资源调度框架中,资源管理器需要通过接收集群计算节点的周期性心跳汇报来判定节点

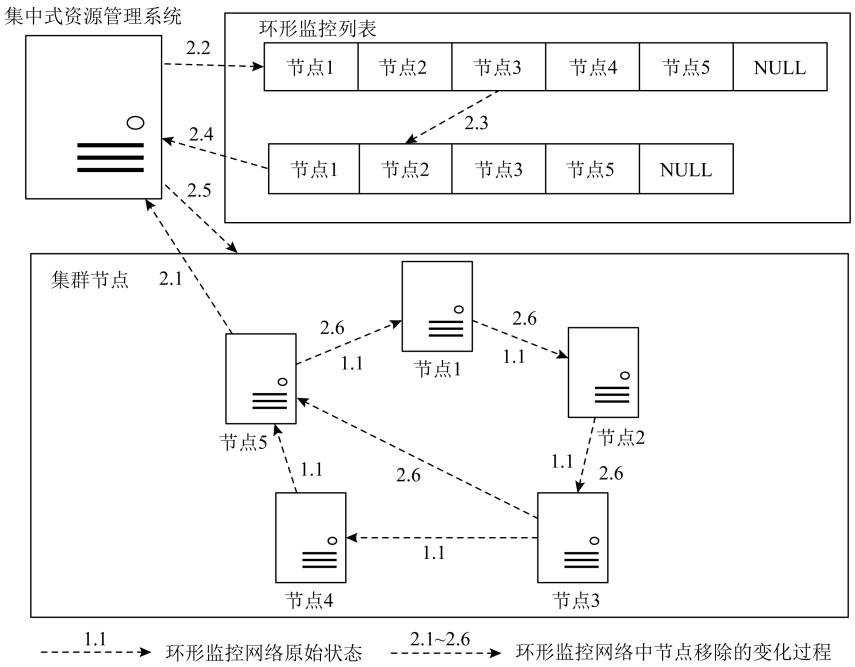


Fig. 2 The node is removed from the ring monitoring network

图 2 节点被移除环形监控网络

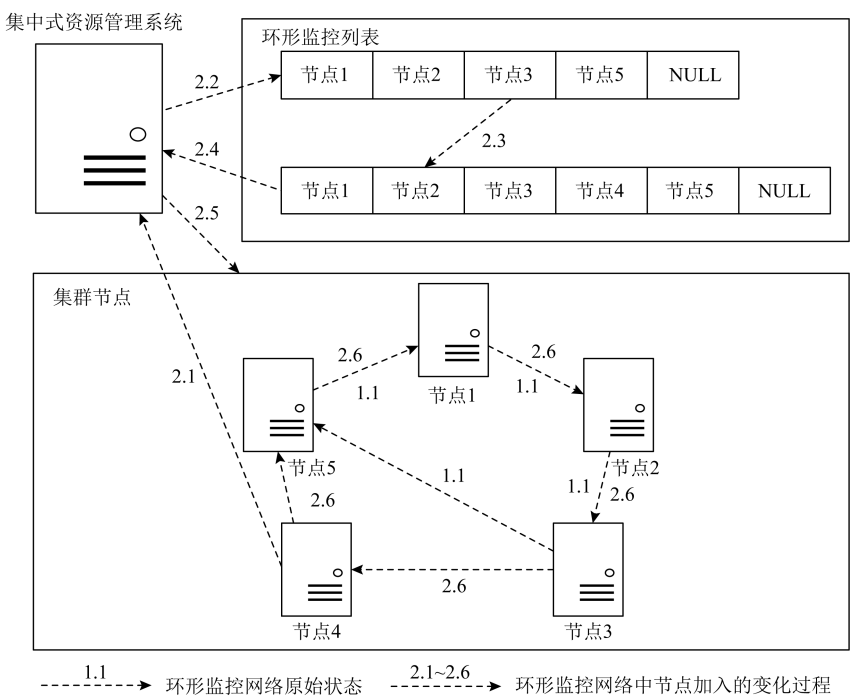


Fig. 3 The node is added to the ring monitoring network

图 3 节点加入环形监控网络

是否宕机.通过图 2 和图 3 的改进,宕机的监视由对等的计算节点来完成,不再由资源管理器来实施,只有当某个计算节点出现故障时,才向资源管理器汇报,从而降低了资源管理器的压力.虽然资源监控服务增加了监视列表的管理功能,但是在通常运行过程中,宕机的概率较小,发生的频度也较低,对系统的整体调度的影响有限.

总之,集群的计算节点通过增加有状态变化的心跳汇报以及环形监控功能,改变资源监控服务严格地接收周期性心跳信息的机制,加快资源监控模块心跳信息处理效率,降低资源状态更新延迟,改善资源管理器的可扩展性问题.

2 可扩展资源管理系统的实现

YARN的资源管理器称为ResourceManager,其资源监控服务称为ResourceTrackerService.计算节点管理程序称为NodeManager,该程序的NodeStatusUpdater组件负责与ResourceTrackerService建立RPC通信机制,完成相应的节点注册以及心跳的汇报功能.YARN同样存在心跳信息周期性发送问题,因此本文按照基于差分的心跳信息处理模型以及基于环形监视的节点监控模型对YARN进行了优化.主要针对计算节点组件NodeStatusUpdater以及资源管理器的资源监控服务ResourceTrackerService进行相关修改.

2.1 集群计算节点的修改

为了完成差分心跳信息的汇报以及宕机监控功能,计算节点需要在计算节点的注册过程以及心跳汇报过程进行相应修改,并增加环形列表的异步更新,保证计算节点组成完整的环形监控网络.

1) 计算节点注册过程修改

计算节点上的服务启动后,NodeStatusUpdater组件首先调用内部函数 *registerWithRM*()完成节点向资源监控服务发起注册请求的信息.为了保证环形监控列表的建立,每个计算节点在注册成功后,应保存由资源监控服务分配的唯一标识.因此,在函数 *registerWithRM*()流程中增加机器序列号的保存工作.

流程 1. 节点注册.

输入:当前节点信息 *curMac*;

输出:当前被更新的节点信息 *curMac*.

① 初始化通信接口 *resourceTracker*;

② 发送 *RegisterNodeManagerRequest*(*curMac*);

③ 接收 *RegisterNodeManagerResponse*;

④ if *RegisterNodeManagerResponse* 是正常状态

⑤ 更新 *curMac.Mid*;

⑥ end if

行①初始化建立 *resourceTracker* 通信接口.行②表示计算节点与资源监控服务的通信注册过程,*RegisterNodeManagerRequest* 将节点IP地址、对外端口号以及总资源的描述信息进行封装,通过 *resourceTracker* 通信接口,调用远程 *RegisterNodeManager* 函数向资源监控服务注册节点信息.行③表示资源监控服务向计算节点返回注册成功消息,其消息类型为 *RegisterNodeManagerResponse*.例如当返回消息标识为shutdown时,可能的原因是当前注册节点资源不符合集群最小分配资源,因此拒绝该节点注册,该节点的服务应该关闭.而在正常情况下被接受注册的节点会收到信息标识为normal.另外,我们对该消息格式增加变量描述,即资源监控服务为计算节点返回1个机器编号.行④~⑥表示计算节点收到正常执行消息后保存机器编号,使得节点知道自身所在环形网络的位置,便于找到其前驱以及后继节点.

2) 心跳信息汇报的修改

在心跳信息汇报过程中,目的是减少不必要的心跳信息发送,并完成计算节点的监控.其过程分为2个步骤:第1步,当心跳时刻到达,计算节点对比本次与上一次缓存的容器状态和资源健康状态,根据对比结果有选择地汇报心跳信息.第2步,增加各个计算节点环形网络的监控功能,保证宕机节点能够被监测到.

流程 2. 节点心跳汇报.

输入:当前节点信息 *curMac*;

输出:当前被更新的节点信息 *curMac*.

① 初始化 T_{new},T_{old},$Health_{new}$,$Health_{old}$,*TaskInfo*;

② 创建1个线程;

③ while !isstopped

④ 读取消息 T_{new},$Health_{new}$;

⑤ if $T_{new}-T_{old}\not\subset\varnothing$或 $Health_{new}\neq Health_{old}$;
/*判断是否有需要向中心节点汇报心跳*/

⑥ $TaskInfo=T_{new}-T_{old}$;
/*任务状态的差分计算*/

⑦ 发送 *NodeHeartbeatRequest*(*TaskInfo*, $Health_{new}$);

⑧ 接收 *NodeHeartbeatResponse*;

⑨ if 检测 *NodeHeartbeatResponse* 为shutdow或resync

⑩ 通知 *NodeManager* 关闭计算节点服务或重启NodeStatusUpdater服务;

⑪ end if

⑫ 更新 *curMac.HeatbeatID*;

⑬ $T_{old}=T_{new}$, $Health_{old}=Health_{new}$;

⑭ end if
/*基于P2P的环形状态检查*/

⑮ 发送 *Liveliness*;

⑯ if 未收到前驱结点消息

⑰ 发送 *UnLivelinessRequest*(*curMac.Prev*);

⑱ 接收 *UnLivelinessResponse*;
⑲ if *UnLivelinessResponse* 的 NodeList 内容不为空
⑳ 更新 *curMac.NodeList*;
㉑ 更新 *curMac.next*,*curMac.prev*;
㉒ end if
㉓ end if
㉔ 等待 *heartbeatTime*;
㉕ end while

行①为变量初始化,行④周期性获取节点容器状态以及健康状态.行⑤～⑧表示若有任务状态或健康状态变化,则先将任务进行过滤处理后,再向资源监控服务汇报心跳信息.行⑨～⑪表示如果资源监控服务返回心跳回复信息执行状态为 shutdown 或 resync,则根据相应行为状态通知 NodeManager 关闭计算节点全部服务或仅重启 NodeStatusUpdater 服务,并影响 NodeStatusUpdater 服务的 isstopped 值的变化,防止当前心跳汇报函数进入下一次循环.行⑫⑬表示按照心跳回复消息内容更新心跳序号,保证计算节点有序汇报心跳信息,同时更新上一次的容器状态以及健康状态信息.行⑮表示计算节点向后继节点发送生存状态信息.行⑯～⑱表示如果当前节点检测前驱节点未发送生存状态消息,则认为节点宕机,并向资源监控服务汇报该节点不可用.否则,不用汇报.行⑲～㉒表示如果节点监控关系不符合当前资源监控服务所维护的环形监控列表,则计算节点需要重新更新前驱与后继节点.

3) 环形监控列表的更新

由于节点的加入或退出会引起计算节点环形监控列表的变化,因此计算节点需要及时更新环形监控列表.为此,在计算节点上增加了环形监控列表更新功能,该功能由资源监控服务触发.针对环形监控列表的更新功能,计算节点相当于服务端,资源监控服务相当于客户端.通过 YARN 提供的类 YarnRPC,在计算节点上构建 RPC 协议,实现了与环形监控列表更新功能相关的函数接口,并在 NodeStatusUpdater 组件初始化时启动该通信服务.同时,对该函数的参数与返回消息进行相关定义与封装.

流程 3. 处理环型列表更新消息.

输入:计算节点环形列表更新消息请求;

输出:计算节点环形列表更新消息回复.

① if *request* 不是来源于中心节点
② return *NodeListUpdateResponse*(reject);
③ end if
④ 更新 *curMac.NodeList*;
⑤ 更新 *curMac.next*,*curMac.prev*;
⑥ return *NodeListUpdateResponse*(accept).

函数参数为 *NodeListUpdateRequest* 消息,内容主要包含节点来源信息以及环形监控列表信息.函数返回为 *NodeListUpdateResponse* 消息,内容主要包含 1 个标志信息,即是否接受处理该信息.行①～③检测该请求来源信息,如果该请求不是资源管理器发来的消息,则拒绝该消息.行④表示读取该消息内容,并更新环形节点列表.行⑤表示计算节点根据自身机器编号以及环形列表,更新前驱节点以及后继节点,即计算节点形成新的环形监控网络.行⑥表示计算节点接受并完成本次列表更新事件,并返回消息.

2.2 资源监控服务处理的实现

利用计算节点差分心跳信息汇报机制,减轻了集中式资源管理器的负载.然而,计算节点的存活状态监控需要资源管理器进行一致性保证.因此,资源管理器的资源监控服务端需要增加环形监控列表的维护功能,相应实现如下.

1) RPC 注册函数的修改

当计算节点发来注册请求消息时,资源监控服务需要对环形监控列表进行更新构建.

流程 4. 处理节点注册消息.

输入:注册请求消息;

输出:注册回复消息.

① if *request* 不是来自有效节点
② return *RegisterNodeManagerResponse*(shutdown);
③ end if
④ *rmnode*=new *RMNode*(*request*);
⑤ *NodeList*+=*rmnode.Mid*;
⑥ 排序 *NodeList*;
⑦ 异步通知各节点更新 *NodeList*;
⑧ return *RegisterNodeManagerResponse*(*normal*,*rmnode.Mid*).

行①～③是对该消息相关内容进行安全检测.行④表示根据消息中节点信息为新注册节点创建状态机,用来管理该节点的生命周期,并为该节点分配 1 个可用机器编号.行⑤表示将该节点编号加入环形监控列表中.行⑥⑦表示将环形列表按照节点序号进行排序,并异步地通知各个计算节点更新环形监控列表,保证计算节点收到节点列表更新消息后,根据自身节点机器编号找到对应的前驱与后继节点,

组成完整环形监控网络.行⑧表示对节点注册信息的回复,为计算节点额外返回1个节点编号,用于标识该计算节点在环形网络的序列关系.

2) RPC心跳响应函数的修改

RPC心跳响应函数中,删除了心跳信息的过滤步骤,直接发送更新通知.下面描述了该函数的正常执行流程.

流程5. 处理节点心跳汇报消息.

输入:心跳汇报请求消息;

输出:心跳汇报回复消息.

① if *request* 不是来自有效节点
② return *NodeHeartbeatResponse* (shutdown);
③ end if
④ if 管理列表中无 *request.Node*
⑤ return *NodeHeartbeatResponse*(resync);
⑥ end if
⑦ if *request.heartbeatID* 不是有序的
⑧ return *NodeHeartbeatResponse*(resync);
⑨ end if
/* 资源监控将当前管理节点的状态相应更新,并引发资源变更事件通知调度器 */
⑩ if *request.NodeStatus* 是健康状态
⑪ 状态转化 *request.Node*→Health;
⑫ else
⑬ 状态转化 *request.Node*→UnHealth;
⑭ end if
⑮ 更新 *HearbeatID*;
⑯ return *NodeHeartbeatResponse*(normal, *HearbeatID*).

行①~⑨表示收到计算节点的心跳汇报请求后,对心跳汇报的节点信息进行安全性检测、判断是否为注册的节点以及是否严格符合消息汇报顺序,如果不符合要求则返回相应计算节点下一步执行行为.由于计算节点状态的变化,会引起资源的变化,因此当资源监控服务收到心跳后进入行⑩~⑭过程,对相应计算节点状态进行更新同步.行⑪表示计算节点从健康状态或非健康状态转化为健康状态的过程.如果该计算节点处于健康状态,则说明容器状态有变化,把计算节点上的变化信息直接通知调度器,完成资源的更新与释放.如果计算节点之前处于非健康状态,则在状态转化过程中,向调度器通知计算节点资源可用消息.行⑬表示计算节点从健康状态到非健康状态的转化,在状态转化过程中,通知调度器当前计算节点资源不可用.行⑮~⑯更新节点心跳序号,并返回心跳回复信息,控制计算节点心跳信息的发送顺序.

3) 宕机节点监控消息的处理

为保证集群计算节点环形一致性监控,资源监控服务需要额外增加功能函数,即计算节点发生宕机时的响应与处理.

流程6. 处理节点宕机汇报消息.

输入:宕机消息请求;

输出:宕机消息回复.

① if *request* 不是来自有效节点
② return *UnLivelinessResponse*(shutdown);
③ end if
④ if 管理列表中无 *request.Node*
⑤ return *UnLivelinessResponse*(resync);
⑥ end if
⑦ if *request.Node*,*request.Pre* 不符合 *NodeList*
⑧ return *UnLivenessResponse*(*NodeList*);
⑨ end if
⑩ 状态转化 *request.Pre*→Lost;
⑪ *NodeList* −=*request.Pre.Mid*;
⑫ 排序 *NodeList*;
⑬ 异步通知各节点更新 *NodeList*;
⑭ return *UnLivelinessResponse*(normal).

在当前通信协议中增加了1个RPC函数,定义为 *UnLivenessNodeManager*,该函数包含1个参数 *UnLivelinessRequest* 消息结构和1个返回值 *UnLivelinessResponse* 消息结构.*UnLivelinessRequest* 主要包含当前计算节点的信息,以及被监控的计算节点的信息.*UnLivelinessResponse* 主要包含当前环形监控列表信息,以及节点下一步执行行为.

行①~⑥首先进行计算节点的安全性检测.行⑦~⑨检查当前计算节点以及它的前驱节点之间的监控关系,判断是否符合当前的环形监控列表,如果不符合则返回最新环形监控列表,通知该计算节点更新前驱与后继的监控关系;否则,接受本次汇报.行⑩将该节点所汇报的前驱节点的状态转为丢失状态,同时向调度器发送节点资源减少消息.行⑪~⑭表示将宕机节点从环形监控列表中移除,重新将环形监控列表排序并异步通知各个计算节点.

另外,YARN资源管理器的NMLivelinessMonitor服务,会根据上一次心跳时间与当前时间之间的差值,判断节点是否宕机.若2个时间差值超过规定时间,则认为该节点宕机,从计算节点管理列表中移除

当前节点信息.采用基于差分的心跳信息处理模型后,NMLivelinessMonitor 服务就不再承担这项功能,因此该服务被删除.

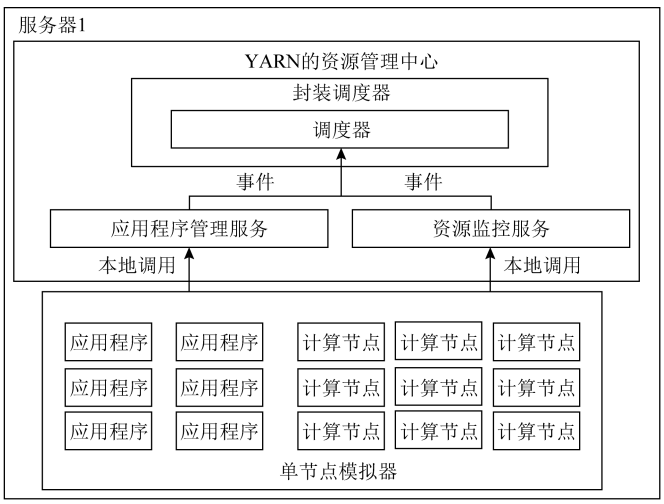
由于 YARN 采用基于事件驱动的资源调度,为避免心跳数量减少而影响资源调度过程,我们开启 YARN 的持续调度机制.

3 实验与结果

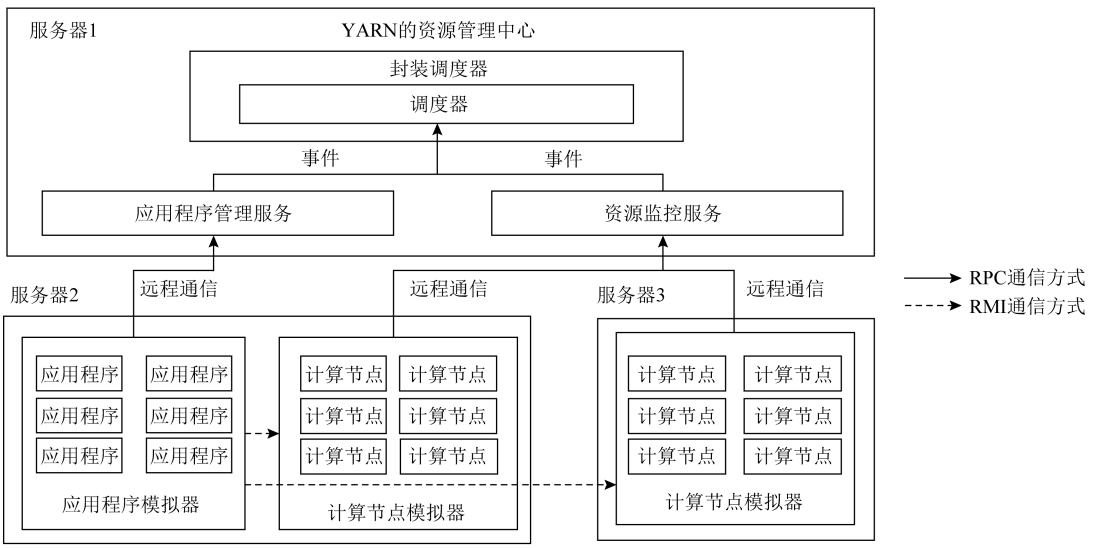
为了评价 YARN 调度器的性能,YARN 官方针对 YARN 资源管理架构提供了一套性能模型工具 SLS(scheduler load simulator),它可以在单台机器上模拟大规模集群,然后根据历史日志信息,获得应用程序负载、资源分配、任务调度和资源回收过程的信息.但是随着模拟的计算节点数量增多以及模拟负载的加大,SLS 中的线程数也相应增加,线程数量的增加导致线程上下文切换开销加大,致使模拟的性能数据受到影响.经过测试,单台机器的最大模拟节点数量为 5 000 个.

为了验证优化后的资源管理框架的可扩展性,提高模拟节点的数量,本文将 SLS 模拟器的应用程序模拟部分以及节点运行模拟部分进行了分离,使其可以运行在多个物理机器上,改进了 SLS 模拟器的运行结构,改进后的 SLS 如图 4(b)所示,称为分布式模拟器.

实验评价过程中,使用了 3 台服务器,每台机器均为 NF5468M5 服务器,包含 2 颗 Xeon2.1 处理器,每个处理器包含 8 个核、32 GB DDR4 内存、2 块



(a) 原模拟器



(b) 分布式模拟器

Fig. 4 Architecture of YARN SLS

图 4 YARN SLS 运行结构图

RTX2080TI GPU 卡、10 GB 显存.3 台服务器中的 1 台服务器作为资源管理器使用,另外 2 台作为集群计算节点使用.

SLS 的系统结构如图 4(a),YARN 的资源管理器的主要服务有 Scheduler,ResourceTrackerService,ApplicationMasterService,它们分别为调度器、资源监控服务、应用程序管理服务.在模拟过程中,应用程序主要模拟作业申请与任务调度.计算节点主要模拟接受应用程序发布的任务,维护其运行状态并周期性向资源监控服务心跳汇报.单节点模拟器发送的心跳方式是通过本地调用的方法来实现,应用程序以及计算节点不通过网络与资源管理系统进行交互,因此无法真实地模拟通信过程.分布式模拟器的资源管理器运行在单独的节点上,计算节点与应用程序运行在其他物理机上.它们之间需要通过 RPC 协议与资源管理系统的各个模块进行信息的交互.另外,应用程序获得资源后,将任务封装成容器描述信息,通过 RMI 通信方式来调用本地或远程机器的对象方法,将容器信息添加到相应模拟计算节点的任务运行管理队列中,保证调度器分配的资源与任务的映射关系在模拟计算节点上得到正确的体现.

为了测试调度器的可扩展性,分布式模拟器的资源管理系统则使用优化后 YARN 的实际代码,集群计算节点使用模拟代码,模拟代码上添加了任务状态与健康状态差分过滤功能,即有变化时汇报心跳消息.

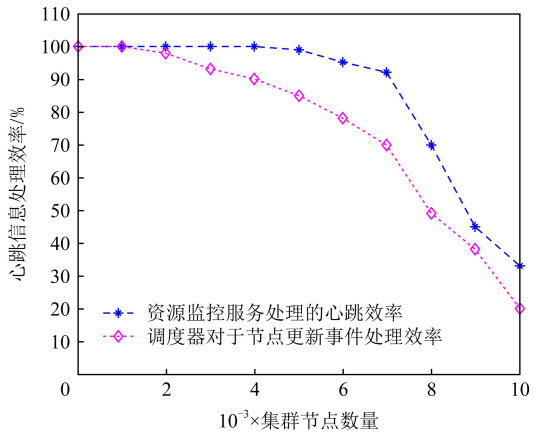
3.1 可扩展性测试

对于计算节点上的负载变化,使用正在运行的容器数量作为负载单位.把计算节点正在运行的容器数与其总容器数的比值称为负载压力.为了合理模拟不同集群节点的负载压力,应保证作业的负载量与集群资源总量成正比.每个作业需要运行的任务数量与计算节点数量和最大运行任务数有关.作业数量的计算公式为 $t=(n\times c\times l)/a$,其中 n 为计算节点模拟数量,c 为每个计算节点最大运行任务数量,$c=20$,l 为负载因子,在实验中规定 $l=0.95$,a 为作业的数量,$a=30$.每个作业相继每隔 10 s 启动一个,每个作业预计执行时间为 400 s,其任务的运行时间随机.预计总运行时长为 $T=a\times T_{\text{offset}}+(T_{\text{job}}-T_{\text{offset}})$,其中 T_{offset} 为每个作业启动的时间间隔,T_{job} 为每个作业的预计执行时间,代入公式得到最终总的预计完成时间为 690 s.随着作业逐个启动,处于运行状态的容器数量逐渐增加,这些信息被周期性汇报给资源监控服务,资源监控服务负载开始上升,当时间到达 400 s 后,资源监控服务的负载开始下降.从第 1 个作业开始到全部作业结束,负载类似正态分布,并在 300～400 s 之间时,负载压力会达到最大,即 40%～50%之间.另外,对于集群节点数量,模拟数量从 1 000 个逐渐增加到 10 000 个进行实验测试.计算节点心跳汇报时间间隔至少设为 3 s,并记录每个心跳周期内发送的消息数量以及接收到心跳回复消息数量.

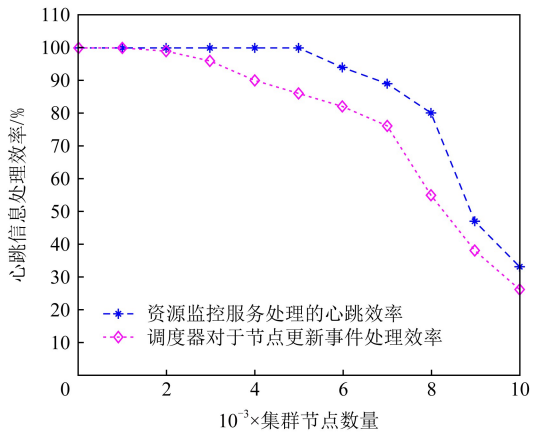
资源监控服务接收到计算节点的心跳信息后,完成 1 次交互.资源监控服务将心跳信息进一步过滤处理后,产生节点更新事件并通知调度器,调度器更新相应的资源状态.随着计算节点上正在运行的容器数量的增加,每次心跳信息的数据量也相应增大,资源监控服务接收到的数据量也增大,资源监控服务将使用更多的 CPU 资源以及内存资源来处理这些数据,并且产生更多的调度器更新事件.由于调度器必须为每个更新事件进行对应的处理,所以调度负载加大.因此集群节点的管理规模取决于调度器对更新事件的处理效率.对于每一次心跳信息的处理,既包括资源监控服务器的数据过滤也包括调度器过滤后的数据的处理.而资源监控服务的处理效率以及调度器的处理效率,决定了整个集群资源管理的能力.资源监控服务的心跳信息处理效率定义为 $h_1=\min(m_i/n_i)$,而调度器对于节点资源更新事件的处理效率为 $h_2=\min(s_i/n_i)$.其中 n_i 为集群模拟节点在心跳时间间隔发送的心跳数量,m_i 为集群模拟节点心跳时间间隔内收到来自资源监控服务的心跳信息回复的数量,s_i 为调度器心跳时间间隔内处理的节点资源更新事件数量.

为了对比改进前后的效果,使用了 YARN 发布的 2 个版本与我们改进后的版本进行测试.图 5 是 YARN 在不同集群规模下的心跳信息处理效率.图 5(a)是使用 YARN 的 Hadoop-2.10.0 版本获得的数据;图 5(b)是 YARN 的 Hadoop-3.1.0 版本获得的数据,Hadoop-3.1.0 对 Hadoop-2.10.0 中的资源监视等服务进行了一定的优化;图 5(c)是本文对 YARN 的 Hadoop-3.1.0 使用差分模型和环形监控优化后的资源管理系统.对比看出,改进后的 YARN 资源管理系统对于心跳信息的处理效率明显高于原系统.YARN 原系统的心跳监控服务大约在 4 500 个计算节点时,就出现性能下降;在 4 000 个节点时,调度器的更新事件处理只能达到 90%左右.而本文优化后的系统,心跳信息处理能够达到 8 000 个

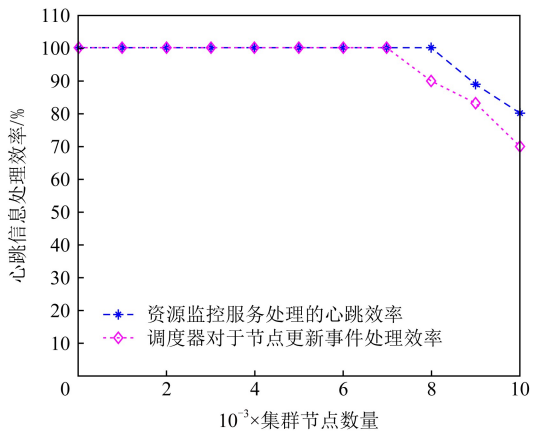
计算节点,在 7 500 个计算节点处,才出现调度器更新事件的处理能力下降.我们知道,资源监控服务以及调度器在同一节点运行,心跳信息处理开销增大,必然影响调度处理开销,反之亦然.YARN 的调度器每接收 1 个更新事件,便进行 1 次资源的回收与分配操作.然而,当资源监控服务对于心跳信息处理出现过载时,更新事件就不能及时发送给调度器,这



(a) YARN(Hadoop-2.10.0)可扩展性测试



(b) YARN(Hadoop-3.1.0)可扩展性测试



(c) 改进后YARN可扩展性测试

Fig. 5 Scalability bottleneck test

图 5 可扩展性瓶颈测试

严重限制了调度器资源回收与分配的性能,因此在大规模集群管理下,更新事件驱动的资源分配方式不太适用.而改进后的系统,在计算节点通过容器状态差分方式对心跳信息进行预处理,减少了心跳数量以及心跳信息所带来的数据量,进而减少通信开销,提高了资源监控服务对于心跳信息的处理效率,使得资源监控服务可以快速地将更新事件发送到调度器,使得调度器的处理性能有较大提升.另外,资源监控服务对于心跳信息的处理效率影响调度器对于资源分配的公平性以及可用资源回收的及时性.总之,提升资源监控服务性能可以进一步提升调度器的性能,进而提升集群管理规模.

当集群管理规模为 8 000 个节点以上时,由于大规模的任务在模拟节点上随机运行,新启动的任务以及结束任务的数量不断出现,导致大量的集群计算节点的心跳汇报信息不能被响应,改进后 YARN 系统的资源监控服务达到了处理瓶颈.同时随着集群节点规模的增大,YARN 内部多种状态机制的转化与管理使得各模块交互事件增多,导致资源管理中的异步事件分派器下发能力限制,调度器对于节点更新事件的处理达到了瓶颈.因此,在心跳时间间隔为 3 s 时,改进后 YARN 的管理集群规模可以扩大到 7 500 个节点,系统亦可以正常地调度任务.修改前的 YARN 在集群节点数量超过 4 000 个时,开始出现调度延迟的现象.改进后的 YARN 资源管理系统的集群规模可以扩大到原来的 1.88 倍.

3.2 调度器对于事件的处理时间测试

YARN 调度器需要处理资源分配请求以及节点更新事件,这些操作分别由应用程序管理服务与资源监控服务引发.在 YARN 的 SLS 结构中,为了获得调度器的各方面性能,SLS 将调度器进行了一次封装,即添加了一个构件 Scheduler wrapper.当资源监控服务处理完心跳信息后,调度器处理开始,同时通过异步事件分派队列,将计算节点更新事件传入 Scheduler wrapper,开始计时.当调度处理结束后,再通过异步事件分派队列,将调度结束消息传入 Scheduler wrapper,停止计时.从处理开始到处理结束的时间段就是处理一次事件所消耗的时间.通过 YARN 的 Metric 性能统计工具,计算出处理相应事件类型的平均耗时.

图 6 和图 7 分别记录了 Hadoop-2.10.0,Hadoop-3.1.0 以及改进后的 YARN 对于节点更新事件的处理时间,以及响应资源请求更新与分配的时间.通过记录节点更新事件的处理时间,可以分析资源监控

服务对调度器处理性能的影响.同时,通过分析应用程序管理服务的资源分配处理过程,可以得到调度器对应用程序的资源分配请求的响应时间.这些数据可以使我们进一步分析应用管理程序、调度器以及资源监控服务之间的性能影响关系.

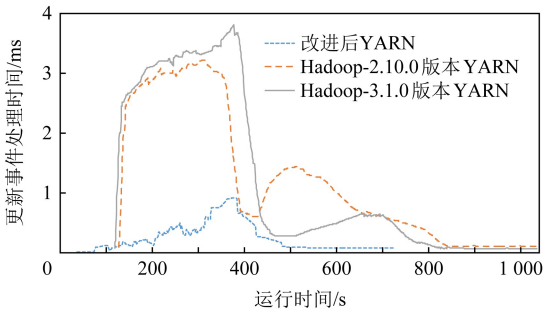


Fig. 6 Scheduler cost time on node update event

图 6 调度器对于节点更新事件处理时间

图 6 为 7 500 节点下调度器对于节点更新事件的处理时间记录,可以看出改进后 YARN 系统的调度器运行时间与预计时间相符合,即在 690 s 内完成整个模拟过程.而 Hadoop-2.10.0,Hadoop-3.1.0 版本的 YARN 系统,在模拟过程中调度器的运行时间均出现一定程度的延迟.另外,在模拟运行过程中,调度器对于事件处理的平均时间明显高于改进后的 YARN 系统.其最主要原因是,改进后的 YARN 系统的资源监控服务可以及时响应集群计算节点的心跳信息并高效地处理,系统未出现过载现象,改善了调度器的更新事件的处理能力.

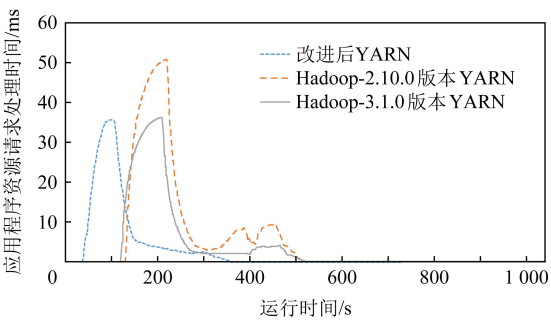


Fig. 7 Scheduler cost time on processing update requests from application master service

图 7 调度器对应用程序管理服务的更新请求处理时间

从图 7 可以看出,改进后的系统能较快地更新资源,大大提高了资源分配请求的更新速度.其主要原因是,当 YARN 的资源监控服务对于心跳信息的处理变得缓慢时,会导致节点资源更新的延迟,使得应用程序对于本次资源分配请求所获取资源不能达到要求,导致应用程序不断地向应用程序管理服务发送资源分配请求,增大了资源请求更新的频率.在 2 次调度之间数据本地化,优先级等因素可能发生变化,调度器需要重新进行资源分配,进一步加重调度器负载压力.因此,资源监控服务心跳信息的处理效率直接影响了资源的回收,进一步影响了资源分配效率,导致调度器在性能方面受到了一定的影响.

3.3 系统性能整体测试

为了测试系统繁忙程度,本文将 CPU 使用率作为参考指标.本节 CPU 使用率特指在用户态下的 CPU 使用率.图 8 给出了系统在维护不同集群规模期间的 CPU 平均使用率.从图 8 可以看出,在 6 000 节点之前,各系统的 CPU 平均使用率随着集群管理规模扩大呈现升高趋势.由于改进后的 YARN 系统的资源监控服务通过接收少量的关键心跳信息来管理大规模集群,使得 CPU 平均使用率相比原系统会低一些.对于改进前的 YARN 系统,当集群规模进一步扩大以及负载压力的提高,集群节点持续地发送大批心跳信息导致资源监控服务过载,因而处理效率更加缓慢.由于心跳信息无法被及时处理,模拟计算节点会长时间等待心跳信息的回复消息,从而导致模拟节点无法正常发送心跳消息,因此其模拟节点上的 CPU 平均使用率从 7 000 节点开始明显下降.相反,由于改进后的 YARN 系统的资源监控模块对于心跳信息的处理效率提高,因此可以响应更多的心跳,其模拟节点不会出现长时间等待心跳回复事件,相比之下 CPU 平均使用率较高,但当节点数量超过 8 000 后其处理能力达到瓶颈.

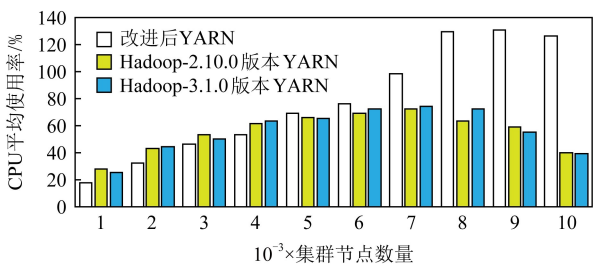


Fig. 8 CPU average usage under different cluster sizes

图 8 不同集群规模下 CPU 平均使用率对比图

3.4 节点宕机情况汇报性能测试

为了验证改进后资源管理系统对于节点宕机情况的处理性能,本文在节点模拟过程中增加基于环形监视的节点状态监控功能.对于节点自身以及其后继节点在同一台物理机器的场合,则直接调用函数来更新状态事件;对于节点自身与后继节点不在

同一台物理机器的场合,则需要与另一台机器建立通信连接,然后进行状态事件的发送与更新.图 9 给出宕机时的测试结果.测试过程中,模拟节点数量为 7 500 个,从整个模拟集群节中随机选择 5%的模拟节点,在运行了 400 s 后停止汇报心跳功能,同时停止向监控节点汇报自己的生存状态,用来模拟节点宕机情况.在该运行过程中,通过 Metric 性能统计工具,记录了节点移除事件.

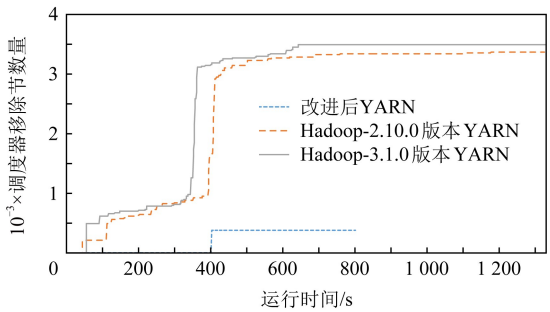


Fig.9 Comparison of the numbers of removed nodes

图 9 节点移除数量对比

从图 9 结果可以看出,对于 Hadoop-2.10.0 与 Hadoop-3.1.0 版本的 YARN 系统,发现大量的额外节点被移除.分析原因后发现,由于节点存活检测时间设置过小,随着在后续过程中心跳信息带来的负载量增大,心跳信息处理的延时增加,导致越来越多的节点被误认为宕机状态而被移除.由于大量的节点被移除,使得调度器需要对该部分资源回收,同时对移除节点上的任务需要重新分配资源,进一步加大了调度器负载,使得任务的运行被延迟.然而设置时间间隔太大,其宕机监控效果将会不敏感.

对于改进后的 YARN 系统,集群计算节点的存活状态监控由计算节点来实施,每个节点只监控它的前驱节点,只有当计算节点向资源监控服务汇报某个节点宕机时,资源监控服务才会将相应节点移除,并通知调度器进行资源的更新与任务的回收,因此上述情况不存在.同时,由于计算节点提前对心跳数据过滤,只有计算节点真正出现宕机时,才向资源监控服务汇报,从而减少了资源监控服务处理消息的数量,使得集群管理器的资源监控服务负荷降低.因此,改进后的 YARN 能够改善集群系统的可扩展性.

4 总 结

资源管理系统处理心跳信息的效率影响系统的可扩展性,因此提高系统运行效率,加快资源状态视图的更新,可以缓解调度器不能及时处理更新事件而导致的延迟等问题.通过资源管理器的资源监控服务功能的分解,将心跳信息的过滤功能转移到计算节点的方式,改变严格的周期性心跳汇报机制,最终减轻资源管理器的负载压力,提高了集中式资源管理系统的可扩展性.事实上,集中式的心跳处理依旧会成为瓶颈.随着集群规模的扩展和状态的规模进一步扩大,集群的资源状态的存储必须使用分布式数据储存机制来保证可用性和低延迟.

参 考 文 献

[1] Apache Software Foundation. Fair scheduler [EB/OL]. (2019-08-23) [2020-02-19]. https://hadoop.apache.org/docs/r1.2.1/fair_scheduler.html

[2] Zaharia M, Konwinski A, Joseph A D, et al. Improving MapReduce performance in heterogeneous environments [C] //Proc of the 8th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2008: 29-42

[3] Mao Hongzi, Schwarzkopf M, Venkatakrishnan S B, et al. Learning scheduling algorithms for data processing clusters [C] //Proc of the ACM SIGCOMM'19. New York: ACM, 2019: 270-288

[4] Zaharia M, Borthakur D, Sarma J S, et al. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling [C] //Proc of the 5th ACM European Conf on Computer Systems. New York: ACM, 2010: 265-278

[5] Le T N, Sun Xiao, Chowdhury M, et al. BoPF: Mitigating the burstiness-fairness tradeoff in multi-resource clusters [J]. ACM SIGMETRICS Performance Evaluation Review, 2018, 46(2): 77-78

[6] Henzinger T A, Singh V, Wies T, et al. Scheduling large jobs by abstraction refinement [C] //Proc of the 6th ACM European Conf on Computer Systems. New York: ACM, 2011: 329-342

[7] Ananthanarayanan G, Agarwal S, Kandula S, et al. Scarlett: Coping with skewed content popularity in MapReduce clusters [C] //Proc of the 6th ACM European Conf on Computer Systems. New York: ACM, 2011: 287-300

[8] Mao Hongzi, Venkatakrishnan S B, Schwarzkopf M, et al. Variance reduction for reinforcement learning in input-driven environments [C] //Proc of the 7th Int Conf on Learning Representations. New Orleans: ICLR Association, 2019: 1-20

[9] Ferguson A D, Bodik P, Kandula S, et al. Jockey: Guaranteed job latency in data parallel clusters [C] //Proc of the 7th ACM European Conf on Computer Systems. New York: ACM, 2012: 99-112

[10] Tumanov A, Cipar J, Ganger G R, et al. Alsched: Algebraic scheduling of mixed workloads in heterogeneous clouds [C] // Proc of the 3rd ACM Symp on Cloud Computing. New York: ACM, 2012: 308-315

[11] Mars J, Tang Lingjia. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers [C] //Proc of ACM SIGARCH Computer Architecture News. New York: ACM, 2013: 619-630

[12] Ghodsi A, Zaharia M, Shenker S, et al. Choosy: Max-min fair sharing for datacenter jobs with constraints [C] //Proc of the 8th ACM European Conf on Computer Systems. New York: ACM, 2013: 365-378

[13] Delimitrou C, Kozyrakis C. QoS-aware scheduling in heterogeneous datacenters with paragon [J]. ACM Transactions on Computer Systems, 2013, 31(4): 12-25

[14] Ousterhout K, Panda A, Rosen J, et al. The case for tiny tasks in compute clusters [C] //Proc of the 14th Workshop on Hot Topics in Operating Systems. Berkeley, CA: USENIX Association, 2013: 12-18

[15] Ananthanarayanan G, Ghodsi A, Shenker S, et al. Effective straggler mitigation: Attack of the clones [C] //Proc of the 10th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2013: 185-198

[16] Delimitrou C, Kozyrakis C. Quasar: Resource-efficient and QoS-aware cluster management [C] //Proc of the 19th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2014: 127-144

[17] Gu Juncheng, Chowdhury M, Shin K, et al. Tiresias: A GPU cluster manager for distributed deep Learning [C] // Proc of the 16th USENIX Conf on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2019: 485-500

[18] Yadwadkar N J, Ananthanarayanan G, Katz R. Wrangler: Predictable and faster jobs using fewer resources [C] //Proc of the 5th ACM Symp on Cloud Computing. New York: ACM, 2014: 311-325

[19] Zhang Qi, Zhani M F, Yang Yuke, et al. PRISM: Fine-grained resource-aware scheduling for MapReduce [J]. IEEE Transactions on Cloud Computing, 2015, 3(2): 182-194

[20] Yao Yi, Tai Jianzhe, Sheng Bo, et al. LsPS: A job size-based scheduler for efficient task assignments in Hadoop [J]. IEEE Transactions on Cloud Computing, 2015, 3(4): 411-424

[21] Verma A, Pedrosa L, Korupolu M, et al. Large-scale cluster management at Google with Borg [C] //Proc of the 10th ACM European Conf on Computer Systems. New York: ACM, 2015: 18-34

[22] Vavilapalli V K, Murthy A C, Douglas C, et al. Apache Hadoop YARN: Yet another resource negotiator [C] //Proc of the 4th ACM Symp on Cloud Computing. New York: ACM, 2013: 5-20

[23] Rasley J, Karanasos K, Kandula S, et al. Efficient queue management for cluster scheduling [C] //Proc of the 11th European Conf on Computer Systems. Berkeley, CA: USENIX Association, 2016: 36-48

[24] Gog I, Schwarzkopf M, Gleave A, et al. Firmament: Fast, centralized cluster scheduling at scale [C] //Proc of the 12th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2016: 99-115

[25] Delgado P, Didona D, Dinu F, et al. Job-aware scheduling in Eagle: Divide and stick to your probes [C] //Proc of the 7th ACM Symp on Cloud Computing. New York: ACM, 2016: 497-509

[26] Wang Wei, Li Baochun, Liang Ben. Dominant resource fairness in cloud computing systems with heterogeneous servers [C] //Proc of the 33rd IEEE Int Conf on Computer Communications. Piscataway, NJ: IEEE, 2014: 583-591

[27] Cho B, Rahman M, Chajed T, et al. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in MapReduce clusters [C] //Proc of the 4th ACM Symp on Cloud Computing. New York: ACM, 2013: 6-17

[28] Ahmad F, Chakradhar S T, Raghunathan A, et al. Shufflewatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters [C] //Proc of the 2014 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2014: 1-13

[29] Curino C, Difallah D E, Douglas C, et al. Reservation-based scheduling: If you're late don't blame us! [C] //Proc of the 5th ACM Symp on Cloud Computing. New York: ACM, 2014: 12-14

[30] Coppa E, Finocchi I. On data skewness, stragglers, and MapReduce progress indicators [C] //Proc of the 6th ACM Symp on Cloud Computing. New York: ACM, 2015: 139-152

[31] Grandl R, Kandula S, Rao S, et al. Graphene: Packing and dependency-aware scheduling for data-parallel clusters [C] // Proc of the 12th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2016: 81-97

[32] Grandl R, Chowdhury M, Akella A, et al. Altruistic scheduling in multi-resource clusters [C] //Proc of the 12th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2016: 65-70

[33] Hindman B, Konwinski A, Zaharia M, et al. Mesos: A platform for fine-grained resource sharing in the data center [C] //Proc of the 8th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2011: 22-35

[34] Apache Software Foundation. Hadoop on demand [EB/OL]. (2019-08-23) [2020-05-20]. http://hadoop.apache.org/docs/r1.0.4/cn/hod.html

[35] Schwarzkopf M, Konwinski A, Abd-El-Malek M, et al. Omega: Flexible, scalable schedulers for large compute clusters [C] //Proc of the 8th ACM European Conf on Computer Systems. New York: ACM, 2013: 351-364

[36] Ousterhout K, Wendell P, Zaharia M, et al. Sparrow: Distributed, low latency scheduling [C] //Proc of the 24th ACM Symp on Operating Systems Principles. New York: ACM, 2013: 69-84

[37] Goder A, Spiridonov A, Wang Yin. Bistro: Scheduling data-parallel jobs against live production systems [C] //Proc of the 2015 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2015: 459-471

[38] Boutin E, Ekanayake J, Lin Wei, et al. Apollo: Scalable and coordinated scheduling for cloud-scale computing [C] //Proc of the 11th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2014: 285-300

[39] Delimitrou C, Sanchez D, Kozyrakis C. Tarcil: Reconciling scheduling speed and quality in large shared clusters [C] // Proc of the 6th ACM Symp on Cloud Computing. New York: ACM, 2015: 97-110

[40] Wang Ke, Liu Ning, Sadooghi I, et al. Overcoming Hadoop scaling limitations through distributed task execution [C] // Proc of 2015 IEEE Int Conf on Cluster Computing. Piscataway, NJ: IEEE, 2015: 236-245

[41] Delgado P, Dinu F, Kermarrec A M, et al. Hawk: Hybrid datacenter scheduling [C] //Proc of the 2015 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2015: 499-510

[42] Karanasos K, Rao S, Curino C, et al. Mercury: Hybrid centralized and distributed scheduling in large shared clusters [C] //Proc of the 2015 USENIX Annual Technical Conf. Berkeley, CA: USENIX Association, 2015: 485-497

[43] Le T N, Sun Xiao, Chowdhury M, et al. AlloX: Compute allocation in hybrid clusters [C] //Proc of the 15th European Conf on Computer Systems. New York: ACM, 2020: 466-480

[44] Hao Chunliang, Shen Jie, Zhang Heng, et al. Structures and state-of-art research of cluster scheduling in big data background [J].Journal of Computer Research and Development, 2018, 55(1): 53-70 (in Chinese)
(郝春亮, 沈捷, 张珩, 等. 大数据背景下集群调度结构与研究进展[J]. 计算机研究与发展, 2018, 55(1): 53-70)

[45] Srijeyanthan K. Distributed resource management for YARN [D]. Stockholm, Sweden: KTH Royal Institute of Technology School of Information and Communication Technology, 2015

Mao Anqi, born in 1996. Master candidate. Her main research interest is cluster resource management.

毛安琪,1996 年生.硕士研究生.主要研究方向为集群资源管理.

Tang Xiaochun, born in 1969. PhD, associate professor. Senior member of CCF. His main research interests include big graph data mining, big data computing and cluster resource management.

汤小春,1969 年生.博士,副教授,CCF 高级会员.主要研究方向为大图数据挖掘、大数据计算、集群资源管理等.

Ding Zhao, born in 1995. Master candidate. His main research interest is cluster resource management.

丁 朝,1995 年生.硕士研究生.主要研究方向为集群资源管理.

Li Zhanhuai, born in 1961. PhD, professor, PhD supervisor. Senior member of CCF. His main research interests include database theory, massive data storage and cluster resource management.

李战怀,1961 年生.博士,教授,博士生导师,CCF 高级会员.主要研究方向为数据库系统理论、海量数据管理、集群资源管理.