计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2017.20151044
54(5):1086-1096,2017

一种基于 Actor 模型的弹性可伸缩的流处理框架

詹杭龙　刘澜涛　康亮环　曹东刚　谢　冰
(高可信软件技术教育部重点实验室(北京大学)　北京　100871)
(北京大学(天津滨海)新一代信息技术研究院　天津　300450)
(zhanhl@pku.edu.cn)

An Elastic Scalable Stream Processing Framework Based on Actor Model

Zhan Hanglong, Liu Lantao, Kang Lianghuan, Cao Donggang, and Xie Bing
(*Key Laboratory of High Confidence Software Technologies* (*Peking University*), *Ministry of Education*, *Beijing* 100871)
(*Peking University Information Technology Institute*(*Tianjin Binhai*), *Tianjin* 300450)

Abstract　In the era of big data, stream processing has been widely applied in financial industry, advertising, Internet of things, social networks and many other fields. In streaming scenarios, the generation speed of stream data tends to be fluctuant and difficult to predict. If the streaming peak is larger than system capacity, the system may run slowly or even crash, which leads to job failure. If excessive resources are provided in case of streaming peak, there can be unnecessary waste under light load. In order to solve the matching problem between stream processing load and resources, stream processing system should be elastically scalable, which means that provided resources can be adjusted automatically according to the real-time change of stream flow. Although some researches have made great progress in stream processing, it is still an open problem that how to design an elastic scalable system. This paper introduces eSault, an elastically scalable stream processing framework based on Actor model. eSault firstly manages the processing units stratified hierarchically based on Actor model, and realizes scalability with two-layer routing mechanism. On this basis, eSault proposes an overload judgment algorithm based on data processing delay and light load judgment algorithm based on the data processing speed to efficiently allocate the resources, and achieve elastically scalable stream processing. Experiments show that eSault has good performance, and can achieve flexible scalability well.

Key words　stream processing; Actor model; cloud computing; elastic scalable; two-layer routing mechanism

摘　要　流处理是一种重要的大数据应用模式,在金融、广告、物联网、社交网络等众多领域得到了广泛应用.在流处理场景中,流数据的产生速度往往变化剧烈且不容易预测.这时,如果数据流量峰值超过处理系统的承载能力,可能使得系统运行缓慢甚至崩溃,导致处理作业失效;如果为了应对数据流量峰值

收稿日期:2015-12-09;**修回日期**:2016-08-08
基金项目:国家"八六三"高技术研究发展计划基金项目(2015AA01A202);国家"九七三"重点基础研究计划基金项目(2011CB302604);国家自然科学基金项目(61272154,61421091);百度云服务开放平台示范项目(2015年)
This work was supported by the National High Technology Research and Development Program of China (863 Program) (2015AA01A202), the National Basic Research Program of China (973 Program)(2011CB302604), the National Natural Science Foundation of China (G61272154, G61421091), and the Baidu Cloud Service Open Platform Demonstration Project (2015).
通信作者:曹东刚(caodg@pku.edu.cn)

而过度配置资源,则可能在系统轻载时产生不必要的浪费.为了解决流处理中负载和资源的匹配问题,流处理系统应该具有弹性可伸缩的能力,一方面以高效的方式组织运算资源;另一方面能根据数据流量的实时变化自动地调整资源使用量.然而,现有的流处理框架对于弹性可伸缩的支持尚很初步.介绍了一种基于 Actor 模型的弹性可伸缩的流处理框架 eSault. eSault 首先基于 Actor 模型将批量的处理单元进行分层管理,通过 2 层路由机制实现了对伸缩性的支持;在此基础上,设计一个基于数据处理延迟的过载判断算法和基于数据处理速度的轻载判断算法来指导系统对资源的有效使用,进而实现弹性可伸缩的流处理.实验结果表明:eSault 具有较好的性能,而且能够很好地实现弹性可伸缩.

关键词 流处理;Actor 模型;云计算;弹性可伸缩;2 层路由机制

中图法分类号 TP391

大数据时代数据规模不断增加,数据产生的速度越来越快.在很多领域,数据的价值随着时间的推移迅速流失[1],应用对数据时效性的要求越来越强,这就要求数据处理系统能够对大量"新鲜"数据进行实时分析.例如社交网络公司可能需要在几分钟内分析话题走向、广告商可能需要实时分析哪些用户点击了广告、服务运营商可能需要在几秒内通过分析日志文件发现系统异常等.因此,流处理作为一种契合上述应用场景的处理模式得到了广泛应用.流处理是指对一连串在时间上连续的消息数据进行实时分析、运算的处理过程[2].在流处理中,一个受到普遍关注的问题是,消息数据往往由外部产生,其流量经常处于变化之中,甚至会突然爆发式增长.例如,亚洲移动电话网络的呼叫记录在峰值时可以达到每秒几十万条记录,而在低谷时只有每秒几千条记录;重大新闻引起新闻网站的访问量突然增大[3].在这种场景下,如果高峰值的消息流量超过了流处理系统的承受能力,可能导致系统运行缓慢甚至崩溃;而如果为了应对消息流量峰值而过度配置资源,则可能在系统轻载时产生不必要的浪费.这些现象的本质其实是流处理中的运算资源无法与负载变化实现动态匹配.为了解决这个问题,一些流处理系统通过在负载过高时,随机丢弃一些消息以应对流量峰值;另一些流处理系统通过重排消息或定义消息优先级,从而在系统负载较高时优先处理一些消息[4];此外,大部分流处理系统通过使用消息队列对消息数据进行缓存[5],从而平滑输入流量,但是这种方式违背了流处理实时性和低延迟性的需求,并没有真正解决此问题.

近年来,云计算的发展为解决流处理中运算资源与负载变化的动态匹配问题提供了新的思路.云计算是一种基于互联网的计算方式,其运算资源是按需聚合与弹性绑定的[6].在云环境中,一个作业在运行过程时具有获取更多资源的能力.对于流处理作业,如果能够在消息流量较大时向云环境申请更多的资源,在负载变低时合理释放部分运算资源,这样便能较好地实现运算资源与负载变化的动态匹配.这样的流处理系统被称为是弹性可伸缩的.弹性是云计算的基本属性之一[7].云环境能够将底层的分布式集群组织起来,通过虚拟机部署设施(如 OpenStack)以及资源调度工具(如 Yarn,Mesos 等)为上层的处理作业提供弹性的运算资源.然而,仅有云环境的弹性支持是不够的,上层的处理作业还需要根据运行时负载的大小实时地调整对运算资源的使用规模,这样才能够实现弹性可伸缩.

为了实现弹性可伸缩的流处理系统,有 2 个必备条件:1)流处理系统是可伸缩的.伸缩性是指系统可以利用变化的资源以调整负载承受力的能力[8].2)系统有一套自适应算法,能够根据运行时负载的变化来决策如何对运算资源进行伸缩.然而,现有的流处理系统尚未完全支持这 2 方面的条件.典型的流处理框架系统如 Apache S4[9],Storm[10],MillWheel[11],Spark Streaming[12]等尚未完全支持弹性可伸缩.一些学术工作对弹性的流处理技术进行了研究,取得了一定进展.如 Esc[13]通过中心式的负载监控器监控机器负载情况,根据 MAPE loop 自动分析负载情况并触发弹性伸缩.Esc 不支持有状态处理单元的伸缩,并且在处理单元内部消息通过单点转发给所有处理元素,效率较低. StreamCloud[3]将流处理单元划分为子处理单元,并根据机器负载情况动态迁移子处理单元,从而实现弹性伸缩.但 StreamCloud 只提供有限的查询操作,并不支持通用的流数据分析. SEEP[14]实现了状态管理系统,通过中心式的负载监控实现了有状态处理单元的动态扩展和状态容错.然而,SEEP 缺乏自适应的调度机制来实现弹性可伸缩.

本文介绍了一种基于 Actor 模型的支持弹性可伸缩的流处理框架 eSault. eSault 除了实现通用流处理框架的基本功能外,重点是支持了应用的弹性可伸缩运行. eSault 首先基于 Actor 模型将批量的处理单元进行分层管理,通过 2 层路由机制实现了对伸缩性的支持;在此基础上,设计一个基于数据处理延迟的过载判断算法和基于数据处理速度的轻载判断算法来指导系统对资源的有效使用,进而实现弹性可伸缩的流处理.

与现有弹性流处理系统的研究工作相比,本文的主要特点在于:

1) 同时支持弹性扩展和弹性收缩. 现有研究多只关注弹性扩展的实现,而 eSault 基于处理元素动态创建退出机制和 2 层路由机制的弹性实现方式,在统一的解决方案下同时支持了弹性扩展和弹性收缩.

2) 基于消息处理延迟和速度的负载判断算法. 现有研究工作主要基于机器资源使用情况进行负载判断,这种方式的限制在于:①资源使用情况的获得需要底层资源管理系统的支持,这增加了框架层与资源层解耦的难度;②流处理应用需要综合使用网络、CPU、内存甚至磁盘等资源,较难设计一种能够准确反映应用负载情况的综合指标. 而 eSault 设计的基于消息处理延迟的过载判断算法和基于消息处理速度的轻载判断算法,完全在应用层实现负载判断,更加直接地监控应用的性能.

3) 完全基于 Actor 模型的设计与实现. eSault 探索了基于 Actor 模型设计与实现弹性流处理框架的可行性,并得到了较好结果.

1 Actor 模型与弹性伸缩

Actor 模型是一种并发编程模型,由 Hewitt 等人[15]在 1973 年提出. 它把“Actor”作为并发编程的基本元素,Actor 可以根据收到的消息进行本地决策,用于创建更多 Actor,发送更多消息和决定如何响应下一个消息. Actor 模型如今已成为许多计算理论和并发系统的理论基础. Actor 模型具有许多特性,例如无共享状态、简单的高层抽象、异步非阻塞的事件驱动编程模型等,这些特性使其非常适合用来对并发程序建模. 此外,目前大部分 Actor 模型的实现中都将 Actor 实现得非常轻量,可以快速地批量创建和销毁,“几十万甚至上百万进程同时并行运行十分常见,而且经常仅仅占用很少的内存”[16]. 这为实现支持弹性的流处理框架带来了 2 点好处:

1) 简化流处理框架的编程模型. 在流处理应用中,数据流的 *key* 往往数量巨大. 如果使用轻量级 Actor 实现处理元素,我们可以为每个 *key* 标识的数据子流启动一个处理元素,使用户在编写处理元素的处理逻辑时可以直接对数据子流进行处理,而不需进一步进行数据分流,从而简化流处理框架的编程模型.

2) 简化弹性的实现机制. 实现系统弹性的基础是伸缩性,可以通过批量创建和销毁轻量级 Actor,实现处理元素的批量迁移,从而动态调整处理元素在集群中的分布.

因此,基于 Actor 模型对弹性流处理系统建模并予以实现,一方面可以简化流处理系统的设计;另一方面可以充分利用 Actor 模型的特点简化框架的编程模型并高效地实现弹性伸缩. 在现有的基于 Actor 模型设计和实现的流处理框架中,S4 尚未支持弹性;而 Esc 虽然支持了弹性,但一方面其论文中并未表明其弹性收缩支持,另一方面其作为原型系统实现较为初步,性能优化空间较大.

2 eSault 系统设计

2.1 编程模型与系统架构

eSault 的编程模型如图 1(a)所示,将流处理应用的处理单元根据功能的不同分为了 Spout 和 Bolt 2 种类型. 一个流处理应用事实上是 Spout 与 Bolt 拼接成的 DAG 图,Spout 是图的源节点,其他节点为 Bolt,图中的边表示处理单元之间的数据路由. Spout 产生 Tuple 格式的流数据,传递给 Bolt 处理,经过多级 Bolt 处理后生成最终结果从输出端流出. 其中,Tuple 是框架中数据流的传输形式,它事实上是一个键值对(*key*/*value*),框架中的数据流都是由连续不断的 Tuple 组成的. Spout 是流处理应用的数据流来源,它源源不断地生成 Tuple 形式的数据流交由后续的 Bolt 处理. Bolt 是流处理应用负责数据处理的单元,它接收由上游传来的 Tuple 数据,调用用户定义的处理方法对数据进行处理后,将新产生的 Tuple 数据发送给下游 Bolt 进行处理. 流处理应用的主要处理逻辑都在各阶段的 Bolt 中实现.

eSault 是基于 Actor 模型设计的,其各功能模块及其子模块在设计时都严格保证了无共享状态,且只通过发送消息交换数据,每个模块都可以抽象成 1 个 Actor. eSault 的系统架构如图 1(b)所示. 应用驱动运行在用户端,为用户提供编程模型中应用

程序的接口,使用户得以构建、提交和控制流处理应用.框架驱动运行在集群中,框架的所有其他模块都由框架驱动启动并控制.Spout/Bolt 处理单元:Spout 和 Bolt 在集群中的运行实例,其包含分布在集群中的大量处理元素(processing element, PE).框架驱动模块通过控制 Spout 与 Bolt,使得流数据可以依据应用程序所定义的逻辑一步步进行处理.Ack(acknowledgement)服务器保证了所有在规定时间内处理完成的消息会被确认,而其他处理超时的消息将由 Spout 重发,从而保证了至少 1 次(at least once)的消息语义.资源接口封装了资源管理器的管理接口,框架驱动通过调用资源接口申请和释放资源,而不需考虑具体的下层资源管理器类型,从而保证框架与资源管理器解耦.

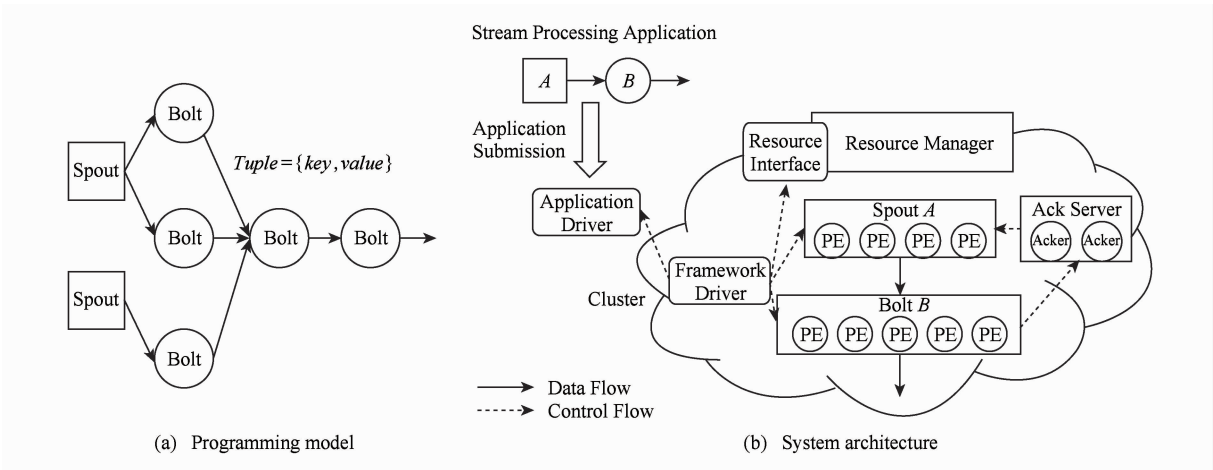


Fig. 1 The architecture and programming model of eSault

图 1 eSault 的系统架构与编程模型

2.2 系统架构与编程模型

流处理单元是 eSault 的数据处理模块,其处理逻辑由用户定义.在编程模型层面一个流处理单元是一个整体,但在实际运行过程中,框架会在集群中启动大量处理元素,使它们并行地执行用户定义的处理逻辑.用户通过应用驱动将实现的流处理单元代码提交给流处理框架,然后通过应用驱动提供的方法创建流处理单元实例并对流处理单元进行动态拼接,从而实现流处理应用.

2.2.1 子处理单元与处理元素

在流处理系统运行过程中,处理单元会在集群中启动并管理数量巨大的处理元素.如果处理单元对这些处理元素进行集中管理,会使处理单元的逻辑变得复杂,运行时负载也较高,很容易导致处理单元运行异常.所以 eSault 将处理单元划分为多个子处理单元,实现分层管理.子处理单元是处理单元的组成部分,与处理单元不同的是,其只运行在 1 台机器上,并且在该机器上启动和管理大量的处理元素.处理单元通过启动和管理多个子处理单元,间接地管理分布在集群中的大量处理元素.图 2 所示 Spout 与 Bolt 中对处理单元进行分层管理的结构图.其中,子处理单元管理器是处理单元的功能模块,其负责启动和管理所有子处理单元;PE 管理器是子处理单元的功能模块,其负责在子处理单元所在的机器上启动和管理大量的处理元素.添加子处理单元后,所有处理元素均由子处理单元管理,处理单元只需管理数量有限的子处理单元即可.这样,处理单元将主要的处理元素管理逻辑下放至子处理单元,从而分散负载并简化了管理逻辑,使系统变得更加稳定,也有利于提高路由效率.

2.2.2 2 层路由机制

一个典型的流处理应用通常由许多处理单元组成,而每个处理单元在同一时间会启动大量的处理元素.在如此大规模的处理元素之间路由消息,保证消息严格按照 *key* 进行分发,并且使这个过程高效、动态、可靠是非常困难的.为了保证消息转发效率,同时又使路由表可以在运行过程中动态进行更改,eSault 提出了 2 层路由转发机制.

如图 3 所示,eSault 的 2 层路由转发机制的主要思想就是结合集中路由和分布路由,在子处理单元之间进行分布路由,在子处理单元内部进行集中路由.源处理单元的所有处理元素均将产生的数据发送给所在子处理单元的输出路由器;输出路由器将数据按照 *key* 值路由给相应的目标子处理单元的输入路由器;目标子处理单元的输入路由器收到数

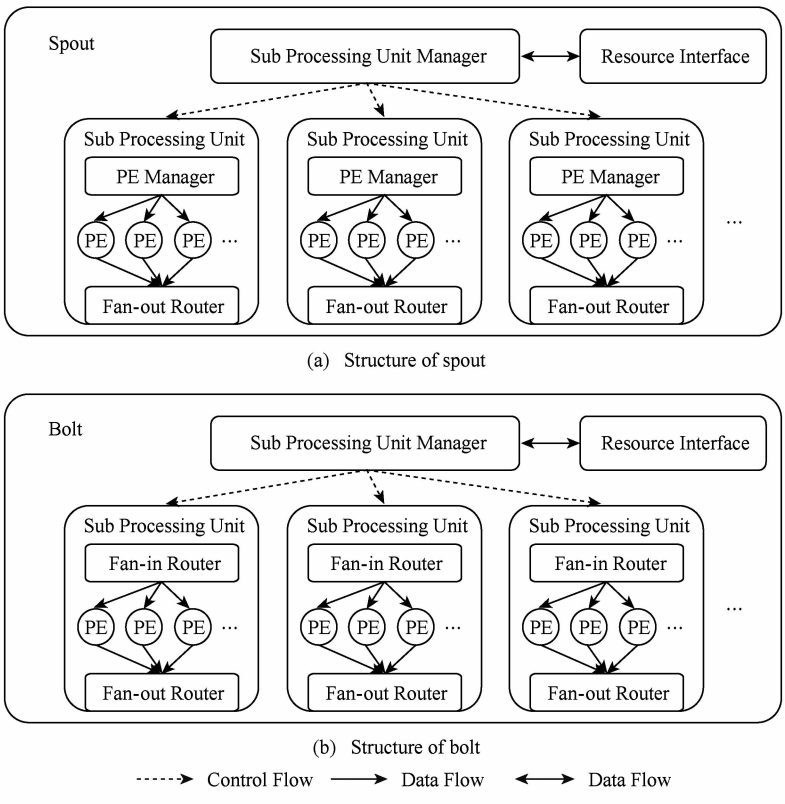


Fig. 2 Hierarchical management of processing unit

图 2 Spout 与 Bolt 中处理单元的分层管理

据后,将数据转发给相应的处理元素.输入路由器和输出路由器是 eSault 的 2 层转发机制的核心构件,两者内部各保存有 1 张路由表用来进行数据路由.这 2 张路由表的设计对于 eSault 的消息转发效率影响巨大,下面分别介绍根据输入路由器和输出路由器各自的功能特点设计和路由表的数据结构.

输入路由表使用散列(Hash)表实现,表的键是输入数据流的 key,表的值是处理该 key 所标记的数据流的处理元素的索引 $PEindex$.当有输入数据时,输入路由器在路由表中查找数据的 key 所对应的路由表项,从而得到该数据对应的处理元素,并将该数据转发给该处理元素.在大规模的数据量下进行快速地增删改查,Hash 表是一个非常理想的选择,因为理想情况下 Hash 表的增删改查的平均时间复杂度都为 $O(1)$,与表项数目无关,所以使用 Hash 表可以高效地实现输入路由表.

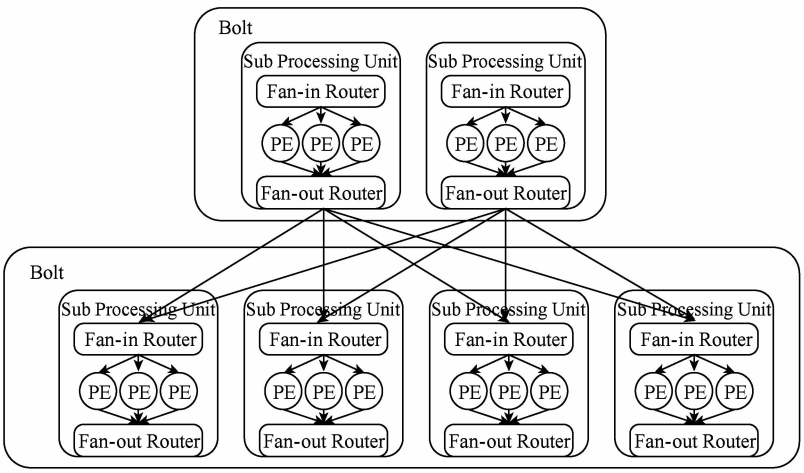


Fig. 3 Two-ayer routing forwarding mechanism

图 3 eSault 的 2 层路由机制

输出路由表的作用是将所有 *key* 尽可能平衡地分给所有目标处理单元的子处理单元,并保证路由效率尽可能的高. eSault 的输出路由表使用线索 2 叉树表示的类似区间树的数据结构实现,2 叉树中的节点由 *key* 和 *PEindex* 组成,其中 *key* 表示[*key*, 后继节点的 *key*)的区间范围,若没有后继节点,则表示[*key*, 最大整数 *INT_MAX*]的区间范围;*PEindex* 则表示该区间对应的子处理单元的索引. 该数据结构的主要特点是可以将在查询某个整数所在的子区间、分裂任意区间和任意相邻区间的时间复杂度控制在 $\log n$ 以内,其中 n 为树中存储的子区间个数.

图 4 展示了区间分裂的过程,初始情况图 4(a)中总的区间范围为[0,*INT_MAX*];图 4(b)中通过插入 *INT_MAX*/2+1 节点,实现了对区间[0,*INT_MAX*]的分裂操作;图 4(c)中插入了 *INT_MAX*/4+1,实现了对区间[0,*INT_MAX*/2]的分裂操作;图 4(d)中进一步插入了 *INT_MAX*×3/4+1,实现了对区间[*INT_MAX*/2+1,*INT_MAX*]的分裂操作. 当有输出数据时,输出路由器首先对数据中的 *key* 在[0,*INT_MAX*]的区间内进行重新散列(rehash),然后在 2 叉树中查找小于等于散列值的最大节点,之后取出该节点对应的子处理单元,即为输出数据的目标子处理单元. 该操作在 $\log n$ 时间内完成,n 为子区间的个数也即子处理单元的个数,因为子处理单元的数量一般与集群规模在同一数量级,最多达到数百数千的级别,所以该时间开销是可以接受的. 使用线索 2 叉树实现输出路由表的最大作用在于配合输入路由表可以非常方便的实现弹性伸缩,这在 2.3 节中会进一步介绍.

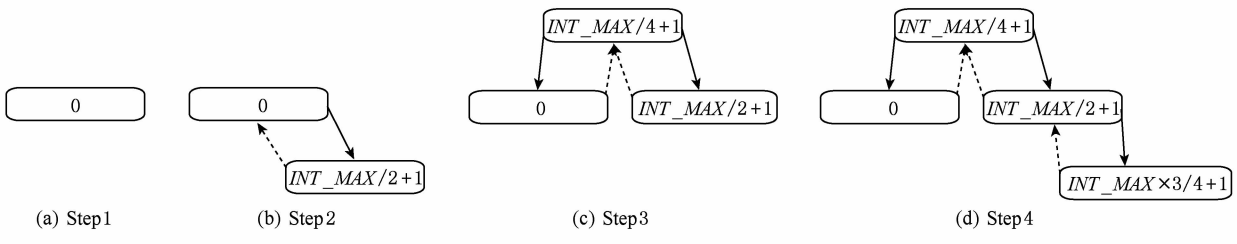


Fig. 4 Interval splitting procedure of fan-out routing table

图 4 eSault 输出路由表的分裂过程

2.3 弹性可伸缩机制

实现伸缩性的关键是能够在有新可用资源时,在新资源上处理任务,从而利用新资源提高系统并行度;在资源减少时,将被减少资源中的任务重新调度到可用资源上处理,从而使系统正常运行[17]. 对于流处理应用来说,也就是在有新资源时,能够将数据流分流至新资源上进行处理;在资源减少时,能够将被减少资源处理的数据流导流至可用资源进行处理.

eSault 伸缩机制的主要设计思想是在处理单元的层面,以子处理单元为单位实现伸缩. 当有新资源时,负载较高的处理单元会在新资源上创建子处理单元,并将部分数据流分流至新的子处理单元,以提高处理能力;当资源减少时,处理单元会将受影响的子处理单元进行迁移,或直接将其输入数据流合流至未受影响的子处理单元.

图 5(a)完整地描述处理单元的动态扩展过程:

① 处理单元在新资源上创建新的子处理单元;

② 处理单元修改子处理单元路由表,将被分流子处理单元对应的区间进行分裂;

③ 处理单元将新的子处理单元路由表发送给所有源处理单元;

④ 源处理单元收到路由表后用其更新所有子处理单元的输出路由表;

⑤ 输出路由表的变化使一部分数据被导流至新建的子处理单元;

⑥ 子处理单元动态创建新的处理元素处理数据流;

⑦ 被分流的子处理单元中的处理元素因为超时退出.

至此,处理单元的动态扩展过程完成. 处理单元的动态收缩过程与动态扩展过程相似,唯一的区别是需要将路由表中受影响子处理单元对应的区间与其相邻区间进行合并,从而实现将其输入数据合流入其相邻区间对应的子处理单元,故在此不再赘述.

在伸缩性的基础上,如果系统能够自适应地根据负载情况申请和释放资源,并自动地触发伸缩,则实现了弹性伸缩. eSault 通过消息延迟监控器监控消息的处理延迟和处理速度,并根据基于消息处理延迟的过载判断算法和基于消息处理速度的轻载判断算法,实现了自动根据负载情况申请和释放资源并触发伸缩机制,从而最终实现了弹性可伸缩.

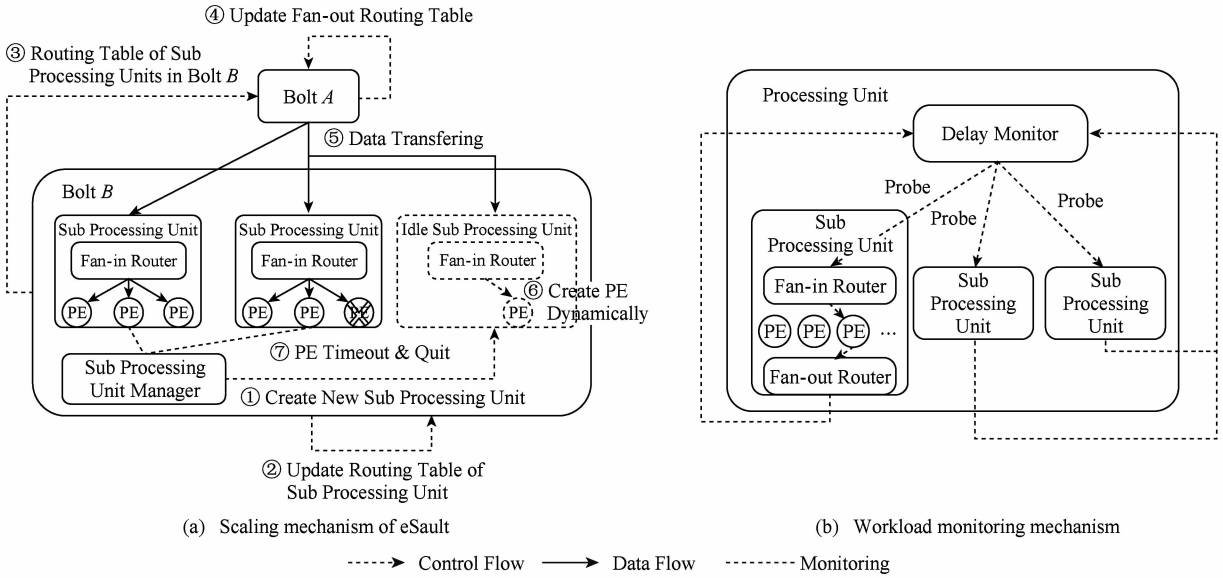


Fig. 5 Workload monitoring and scaling of eSault

图 5 eSault 的伸缩机制与延迟监控器

2.3.1 消息处理延迟监控

在对系统负载进行动态监控时,资源使用情况是最直接的衡量指标,一些研究弹性的流处理系统,例如 SEEP,Esc 等,也都使用这一指标. 但是,使用其作为负载衡量指标也存在一些缺陷:一方面资源使用情况的获得需要底层资源管理系统的支持,这使框架难以与资源管理系统解耦;另一方面,流处理应用需要综合使用网络、CPU、内存甚至磁盘等资源,很难设计一种能够准确反映应用负载情况的综合指标. 因此,eSault 选择在应用层通过性能监测负载,因为无论底层任何资源成为瓶颈,最终表现都是应用无法达到性能要求,而且这也有助于实现框架与资源层解耦. eSault 认为消息处理延迟是非常理想的负载衡量指标,因为其他指标都只能部分地反映负载情况. 例如,输入消息数量很大时,如果每条消息实际处理时间很短,则处理负载并不一定会高,也就很难确定一个消息数量作为负载限额;同理,使用输入数据吞吐量和消息队列长度也会出现类似的情况. 然而,消息处理延迟综合反映了网络传输时间、排队时间和处理时间,而且因为流处理应用的核心价值就在于在线实时处理从而降低处理延迟,所以消息处理延迟是流处理应用的非常理想负载衡量指标.

eSault 的负载监控是由每个处理单元独立进行的,即每个处理单元监控自身各子处理单元的负载,并根据负载情况作出弹性伸缩的决策. eSault 在 Bolt 中添加了延迟监控器模块. 如图 5(b)所示,延迟监控器周期性地给 Bolt 的所有子处理单元发送探针(probe),探针流经输入路由器、最近一次处理数据的处理元素和输出路由器后,返回延迟监控器,之后延迟监控器即可通过探针发出时间和返回时间,判断对应子处理单元的消息处理延迟.

2.3.2 基于消息处理延迟的过载判断算法

算法的思路如下:延迟监控器每隔 *PROBE_PERIOD* 向所有子处理单元发送探针,并监控探针的处理时间是否超过 *MAX_LATENCY*,如果有一个子处理单元在 *OVERLOAD_REACTION_TIME* 个采样周期内的总超时次数所占比例超过 *OVER_LOAD_FACTOR*,则认为该子处理单元过载,需要申请新的资源,并触发伸缩机制将该子处理单元分裂至新资源上.

算法中的 4 个关键变量:*MAX_LATENCY*, *PROBE_PERIOD*, *OVERLOAD_REACTION_TIME* 和 *OVERLOAD_REACTION_FACTOR* 需要应用配置指定,以改变算法的额外开销、灵敏度等属性.

1) *MAX_LATENCY*. 算法允许的最大探针处理延迟. 改变该变量,可以改变算法可容忍的最大消息处理延迟,应根据处理单元的任务类型和应用对消息处理处理延迟的要求合理配置该值.

2) *PROBE_PERIOD*. 发射探针的采样周期. 改变该变量,可以调整算法的额外开销和灵敏度. 增

① https://github.com/pkusei/Sault

大该值,会使采样周期变长,采样次数变少,从而使发送和处理探针带来的额外开销减小,但也会使算法对过载的反应时间变长,算法灵敏度下降;反之,会使算法的额外开销增大,反应时间变短,灵敏度提高.

3) *OVERLOAD_REACTION_TIME*. 过载判断的反应时间. 改变该变量可以调整算法的反应时间,从而调整算法灵敏度. 增大该值,算法需要更长的时间才能确定过载,因而灵敏度下降;反之,算法灵敏度上升.

4) *OVERLOAD_REACTION_FACTOR*. 允许的 *OVERLOAD_REACTION_TIME* 内超时记录占总记录数的比例. 改变该变量,可以调整算法判断条件的严格程度,从而调整算法灵敏度. 增大该值,算法允许的超时次数增大,过载判断条件更为严格,算法灵敏度下降;反之,算法灵敏度上升.

2.3.3 基于消息处理速度的轻载判断算法

消息处理延迟在判断过载时非常有效,但在判断轻载时却无法显著反映负载情况. 消息处理延迟主要由网络传输时间、排队时间和处理时间组成,动态扩展的主要目的是降低排队时间和处理时间. 在高负载情况下排队时间和处理时间成为消息处理延迟的主要部分,所以其可以显著反应排队时间和处理时间,从而反映系统负载;而在轻载情况下网络传输时间成为消息处理延迟的主要部分,其不再显著反映排队时间和处理时间,从而无法显著反映系统负载.

为了解决这个问题,eSault 设计了基于消息处理速度的轻载判断算法,通过消息处理速度是否显著低于峰值,判断子处理单元是否处于轻载状态. 算法的主要思想是:延迟监控器依然每隔 *PROBE_PERIOD* 向所有子处理单元发送探针;子处理单元的输入路由器会统计 2 次探针之间,子处理单元处理的消息总数,并在收到探针时将该结果存入探针中;延迟监控器根据探针中的消息总数是否低于 *LOW_WATERMARK* × 消息处理峰值判断子处理单元是否轻载;如果有 1 个子处理单元在 *UNDERLOAD_REACTION_TIME* 个采样周期内的轻载次数所占比例超过 *UNDERLOAD_REACTION_FACTOR* 且在 *OVERLOAD_REACTION_TIME* 个采样周期内超时次数为 0,则认为该子处理单元轻载,需要触发伸缩机制将该子处理单元与相邻单元合并,并释放资源.

在该算法中,当消息处理速度显著提高时,算法会将所有峰值信息重置,因为其认为这意味着系统的工作负载发生了显著变化,算法应当使用最新的峰值来做决策. 在基于消息处理速度的轻载判断算法中,当探针超时的时候,说明系统进入了新一轮高负载运行状态,算法通过使历史峰值信息无效化来适应新的工作负载. 算法中的 4 个关键变量:*LOW_WATERMARK*, *PROBE_PERIOD*, *UNDERLOAD_REACTION_TIME* 和 *UNDERLOAD_REACTION_FACTOR* 同样需要应用配置指定,4 个变量的作用与基于消息处理延迟的过载判断算法基本一一对应.

至此,eSault 通过在处理单元中增加延迟监控器监控各子处理单元的消息处理延迟和处理速度,并通过基于消息处理延迟的过载判断算法和基于消息处理速度的轻载判断算法自动分析子处理单元负载,从而实现了根据负载自适应地分配和释放资源,并自动地触发伸缩,最终实现了弹性可伸缩.

3 系统实现与实验分析

3.1 系统实现

本文使用编程库 Akka 实现了 eSault 的原型系统[①]. Akka 是一个运行在 JVM 上的基于 Actor 模型的开源工具包和运行时. Akka 具有轻量级 Actor,Actor 位置透明、消息分发高效等特点,非常适合构建高效的分布式并发应用. 在实现 eSault 的过程中,所有的功能模块均使用 Akka 的 Actor 进行刻画,这使得 eSault 的结构非常简单直观.

3.2 弹性可伸缩效果验证

本实验的主要目的是验证 eSault 弹性可伸缩的能力,即证明 eSault 上运行的流处理应用可以随输入流量的变化自动调整资源使用量,并保证处理延迟的稳定. 实验的主要思路是:阶段性地调整 Emitter 生成单词的速度,并在此过程中监测单词生成速度、单词平均处理延迟和 Counter 的子处理单元数目的变化情况. 其中输入流量对应单词生成速度,并行度对应 Counter 的子处理单元数,延迟对应单词平均处理延迟.

如图 6(a)所示实验过程中,输入流量共经历了 2 次上升和下降的变化周期,每个周期为时约 250 s. 在每个周期内,输入流量在前 30 s 内,每 10 s 提升约 50 000 tuples/s 的单词生成速度,并在达到峰值后稳定约 70 s;此后每 20 s 下降约 50 000 tuple/s 的单词生成速度,并在达到谷值后稳定约 70 s.

通过分析输入流量、处理延迟和并行度三者的变化关系,可以发现如下符合实验预期的现象:

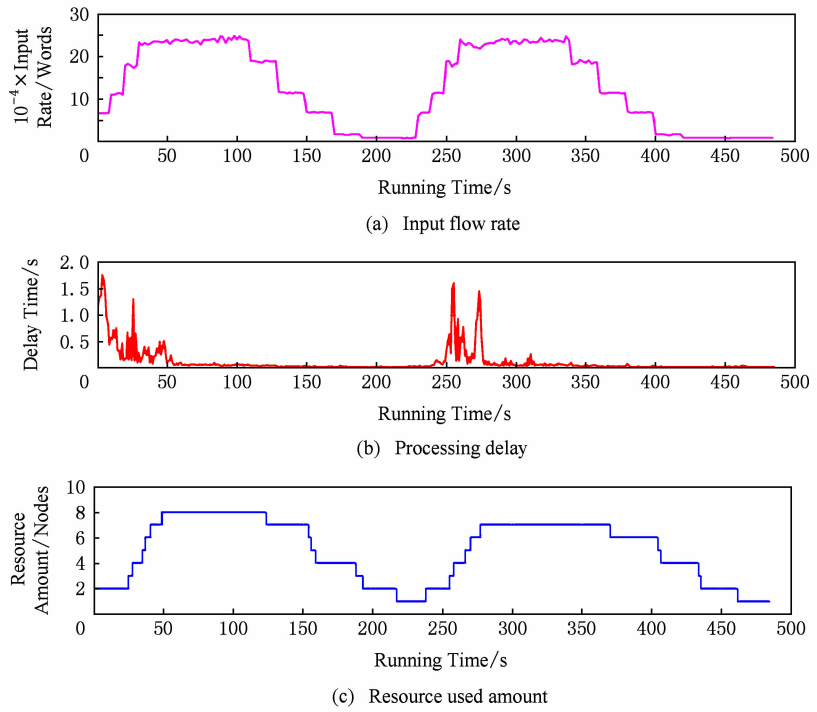


Fig. 6 Verification of elastic effect

图 6 弹性效果验证

1) 实验过程中的大部分时间,消息处理延迟基本稳定在较低水平. 如图 6(b)所示,在整个实验过程中,应用的消息处理延迟基本稳定在 100 ms 以下,即使在输入流量达到峰值后,消息处理延迟在大部分时间也稳定在 100 ms 以下.

2) 并行度随输入流量的变化趋势明显. 图 6(c)中可以观察到 Counter 的初始并行度为 2,随着输入流量提高,其并行度快速增加以适应负载增加,最终达到 8;随着输入流量降低,其并行度逐渐减少以释放多余资源,最终达到 1. 上述实验结果基本满足实验预期,可以证明 eSault 基本支持了流处理应用的弹性可伸缩.

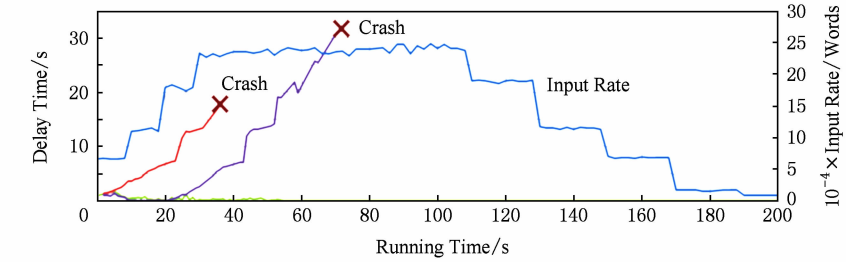
3.3 弹性可伸缩的必要性验证

本实验的主要目的是验证上述弹性效果实验中,系统能够根据数据流负载自动作出资源调整,从而保证系统不会因突然的数据流量高峰崩溃,也不会在数据流量较低时浪费资源. 实验的主要思路是:使用与弹性效果实验中相同的 Emitter,测试不同并行度的 Counter,观察单词平均处理延迟的变化情况,并与弹性效果实验中的延迟变化情况进行比较,从而验证弹性伸缩的必要性.

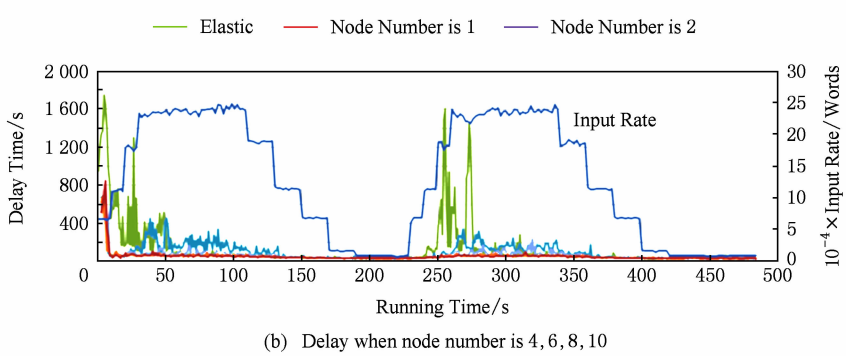
图 7 展示了在预设为不同并行度以及弹性可伸缩执行的情况下,处理延迟随着输入流量而变化的情况. 由于并行度为 1,2 的情况较为特殊,单独在图 7(a)中展示;其余并行度以及弹性执行的情况在图 7(b)中展示;图 7(c)详细展示了输入流量达到峰值且弹性伸缩延迟稳定后(60～110 s)不同并行度情况下的延迟比较.

通过观察实验结果可以发现,当并行度固定为 1 和 2 时,系统在输入流量上升后产生了严重的消息堆积,最终导致底层通信机制因来不及处理心跳信息而出错,使系统无法正常运行. 而弹性伸缩情况下,虽然 Counter 的初始并行度也为 2,但其通过自动提高并行度度过了流量高峰,并保持了延迟的基本稳定. 这说明较低的并行度无法在规定延迟内处理数据流量高峰,而弹性伸缩机制可以通过增加资源应对流量高峰,此现象符合实验预期.

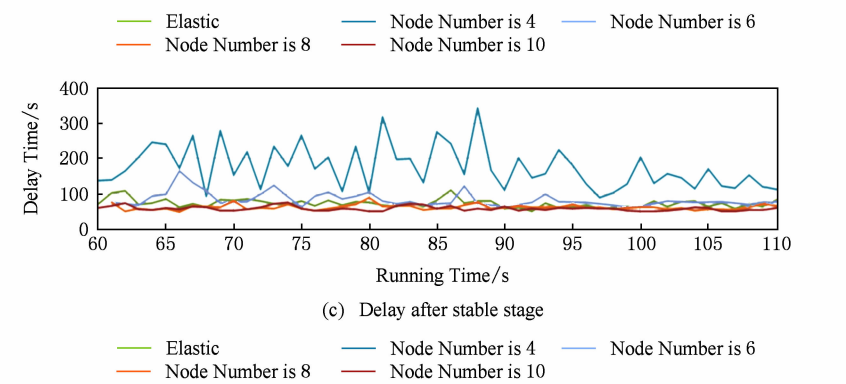
如图 7(b)所示,当并行度固定为 4 和 6 时,系统可以承受数据流量高峰,且没有弹性伸缩的延迟波动期. 但如图 7(c)所示,在弹性伸缩的延迟稳定后,并行度为 4 和 6 的情况下,处理延迟会高于弹性伸缩的情况. 并行度为 8 的情况下,延迟均值、方差和最大值均显著低于并行度为 4 和 6 的情况,而仅略高于并行度为 10 的情况. 这说明 8 是该流量峰值下合适的并行度,而 eSault 的弹性机制确定的并行度恰为 8,这说明 eSault 的弹性伸缩机制准确地找到了最合适的并行度. 不过因为弹性伸缩机制本身有一定开销,所以最终弹性情况下的延迟略高于并行度为 8 的情况.



(a) Delay when node number is 1, 2



(b) Delay when node number is 4, 6, 8, 10



(c) Delay after stable stage

Fig. 7 Necessity validation of elasticity

图 7 弹性必要性验证

上述实验结果基本满足实验预期,可以证明 eSault 的弹性支持可以准确地根据输入流量找到最佳并行度,既能应对流量高峰,又能在流量低谷节约资源.

4 结束语

本文基于 Actor 模型设计与实现的弹性流处理框架 eSault,除了实现其他通用流处理框架的基本功能外,还重点支持了应用的弹性伸缩.实验证明了 eSault 可以准确根据输入流量决定资源使用量,既能在流量高峰时保持延迟稳定,又能在流量低谷时节约资源,达到了预期效果.

未来的工作包括 2 个方面:

1) 更加智能的自适应算法,使得参数配置可以根据数据流历史情况挖掘流量变化规律自动调整,对于流量波动较少但幅度较大的数据流,采用激进的参数配置;对于流量波动幅度较小但频繁的数据流,采用稳健的参数配置.

2) 有状态处理单元的弹性伸缩,进一步研究如何将状态管理原语以尽可能透明地方式纳入 eSault 的编程模型,并在框架内支持弹性伸缩过程中的状态迁移,从而实现对有状态处理单元弹性伸缩的原生支持.

参 考 文 献

[1] Cheng Xueqi, Jin Xiaolong, Wang Yuanzhuo, et al. Survey on big data system and analytic technology [J]. Journal of Software, 2014, 25(9): 1889-1908 (in Chinese)
(程学旗, 靳小龙, 王元卓, 等. 大数据系统和分析技术综述 [J]. 软件学报, 2014, 25(9): 1889-1908)

[2] Cui Xingcan, Yu Xiaohui, Liu Yang, et al. Distributed stream processing: A survey [J]. Journal of Computer Research and Development, 2015, 52(2): 318-332 (in Chinese)

(崔星灿, 禹晓辉, 刘洋, 等. 分布式流处理技术综述[J]. 计算机研究与发展, 2015, 52(2): 318-332)

[3] Gulisano V, Jimenez-Peris R, Patino-Martinez M, et al. Streamcloud: An elastic and scalable data streaming system [J]. IEEE Trans on Parallel and Distributed Systems, 2012, 23(12): 2351-2365

[4] Hummer W, Satzger B, Dustdar S. Elastic stream processing in the cloud [J]. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2013, 3(5): 333-345

[5] Qi Kaiyuan, Han Yanbo, Zhao Zhuofeng, et al. MapReduce intermediate result cache for concurrent data stream processing [J]. Journal of Computer Research and Development, 2013, 50(1): 111-121 (in Chinese)
(亓开元, 韩燕波, 赵卓峰, 等. 支持高并发数据流处理的 MapReduce 中间结果缓存[J]. 计算机研究与发展, 2013, 50(1): 111-121)

[6] Lu Xicheng, Wang Huaimin, Wang Ji. Internet-based virtual computing environment (iVCE): Concepts and architecture [J]. Scientia Sinica: Informationis, 2006, 49(6): 681-701

[7] Buyya R, Broberg J, Goscinski A. Cloud Computing: Principles and Paradigms [M]. New York: John Wiley & Sons, 2011: 457-490

[8] Herbst N R, Kounev S, Reussner R. Elasticity in cloud computing: What it is, and what it is not [C] //Proc of the 10th Int Conf on Autonomic Computing. Berkeley, CA: USENIX Association, 2013: 23-27

[9] Neumeyer L, Robbins B, Nair A, et al. S4: Distributed stream computing platform [C] //Proc of the 13th Int Conf on Data Mining Workshops. Piscataway, NJ: IEEE, 2010: 170-177

[10] Toshniwal A, Taneja S, Shukla A, et al. Storm@ twitter [C] //Proc of the 2014 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2014: 147-156

[11] Akidau T, Balikov A, Bekiroğlu K, et al. MillWheel: Fault-tolerant stream processing at internet scale [J]. Proceedings of the VLDB Endowment, 2013, 6(11): 1033-1044

[12] Zaharia M, Das T, Li H, et al. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters [C] //Proc of the 4th USENIX Conf on Hot Topics in Cloud Computing. Berkeley, CA: USENIX Association, 2012: 10

[13] Satzger B, Hummer W, Leitner P, et al. Esc: Towards an elastic stream computing platform for the cloud [C] //Proc of 2011 Int Conf on Cloud Computing. Piscataway, NJ: IEEE, 2011: 348-355

[14] Fernandez R, Migliavacca M, Kalyvianaki E, et al. Integrating scale out and fault tolerance in stream processing using operator state management [C] //Proc of the 2013 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2013: 725-736

[15] Hewitt C, Bishop P, Steiger R. A universal modular actor formalism for artificial intelligence [C] //Proc of the 3rd Int Joint Conf on Artificial Intelligence. San Francisco, CA: Morgan Kaufmann, 1973: 235-245

[16] Cesarini F, Thompson S. Erlang Programming [M]. Sebastopol, CA: O'Reilly Media, Inc, 2009

[17] Zhan Hanglong, Kang Lianghuan, Cao Donggang. DETS: A dynamic and elastic task scheduler supporting multiple parallel schemes [C] //Proc of the 8th Int Symp on Service Oriented System Engineering. Piscataway, NJ: IEEE, 2014: 278-283

Zhan Hanglong, born in 1989. Received his PhD degree from the School of Electronics Engineering and Computer Science, Peking University in 2016. His main research interests include big data, system software, parallel and distributed computing, etc.

Liu Lantao, born in 1990. Received his MSc degree from the School of Electronics Engineering and Computer Science, Peking University in 2015. His main research interests include big data, system software, parallel and distributed computing, etc.

Kang Lianghuan, born in 1986. Received his PhD degree from the School of Electronics Engineering and Computer Science, Peking University in 2015. His main research interests include distributed systems, concurrent programming structures and languages, etc.

Cao Donggang, born in 1975. Received his PhD degree from the School of Electronics Engineering and Computer Science, Peking University in 2004. Currently associate professor at Peking University. His main research interests include system software, parallel and distributed computing, etc.

Xie Bing, born in 1970. Received his PhD degree from the School of Computer, National University of Defense Technology in 1998. Currently professor and PhD supervisor at Peking University. His main research interests include software engineering, formal methods and software reuse, etc (xiebing@pku.edu.cn).