

一个基于事件驱动的面向服务计算平台

刘家红 吴泉源

(国防科技大学计算机学院网络与信息安全研究所 长沙 410073)

摘 要 基于请求/响应调用模型的面向服务体系结构(Service-Oriented Architecture, SOA)的实现存在通信耦合程度高,协同能力不足的问题.事件驱动体系结构特别适合于松耦合通信和应用需要感知支持的环境.在面向服务的计算平台中提供事件驱动支持,可满足计算平台的松耦合通信与协同需求.文中给出了面向服务计算平台中事件驱动的框架,针对需高效处理事件流上复合事件的需求,在框架中设计了基于 SEDA 模型的并发事件处理与基于事件代数的事件流处理机制.在事件代数中给出了上下文语义和相应的检测算法,以实现高效事件流处理.实验表明,设计的事件驱动面向服务计算平台具有松耦合通信、协同计算、高效事件流处理和复合事件处理的特点,适应了目前动态多变的大规模分布式计算环境的需求,有着广阔的应用前景.

关键词 面向服务的计算平台;事件驱动体系结构;事件流处理;事件代数;复合事件处理

中图法分类号 TP311

An Event-Driven Service-Oriented Computing Platform

LIU Jia-Hong WU Quan-Yuan

(Institute of Network Technology & Information Security, School of Computer,
National University of Defense Technology, Changsha 410073)

Abstract SOA implementations based upon Request/Response model have deficiency in tight-coupled communication and poor cooperativity. Event-Driven Architecture (EDA) is one of the choices for implementing loose coupled communication with minimum delay and for responding rapidly and effectively to changing conditions, especially complex events derived from multiple events. By introducing Event-Driven paradigm into Service-Oriented computing platform, we can achieve loose couple communication and cooperativity. This paper presents a unified event-driven framework with design in delicacy for high-performance SEDA-based event concurrency and event algebra based event stream processing. To improve event stream processing performance, context semantic and corresponding detection algorithm are also presented. Experiment results prove that the framework is adapted to the dynamic large-scale distributed computing environment well and will be used widely due to its loose coupled communication, cooperativity and high-performance event stream processing and complex event processing properties.

Keywords service-oriented computing platform; event-driven architecture; event stream processing; event algebra; complex event processing

1 引言

面向服务的计算 (Service-Oriented Computing) 作为一种新的计算范型, 与其它分布计算技术同样面临着网络分布环境带来的挑战, 其中包括: 松耦合方式连接应用以达到互操作^[1]; 应用自身、应用间协作均需要感知, 尤其是对感知的多源信息进行融合和关联分析^[2], 需要分布计算技术提供协同功能. 因此面向服务计算平台在通信模型方面必须具有松耦合、异步特性, 在处理体系结构方面, 必须对输入、异常以及内部/外部改变具有反应性, 同时支持对多源信息的融合和关联. 大部分面向服务计算平台采用请求/响应的处理, 在服务的通信模型上是 RPC 的, 在处理体系结构上是转换式^[3]的. 在基于请求/响应的处理模型中, 服务请求者指明要调用的服务, 然后发送请求; 服务提供者接收请求, 进行处理, 随后返回响应信息. 这种请求/响应的处理模型不能提供反应性, 也不能提供松耦合的通信模型. 事件驱动体系结构上是对反应式系统^[3]的抽象, 通过“推”模式的事件通信^[1]提供并发的反应式处理, 特别适合松耦合通信和支持感知的应用, 是解决 SOA 松耦合通信与协同处理的理想解决方案.

事件驱动体系结构也存在缺点, 例如系统设计复杂、可理解性差. 另外系统设计时面临高并发通信和处理的性能挑战^[1,4], 在对应用感知的多源信息进行融合和关联分析方面, 系统需要支持事件流上的复合事件处理. 论文以事件驱动体系结构为基础, 为应用层支持松耦合通信模型以及反应式处理提供框架; 在框架上对并发事件处理进行优化设计, 并支持事件流上的复合事件处理^[2], 以解决事件驱动体系结构带来的复杂性、可理解性差以及高效处理并发请求问题. 优化的事件驱动框架提供松散耦合、并发处理、流上复合事件处理特性, 适应了目前动态多变的大规模分布式计算环境的需求, 有着广阔的应用前景.

论文首先介绍事件驱动体系结构风格; 随后给出面向服务计算平台中事件驱动体系结构的处理框架, 包括体系结构以及行为模型; 最后针对面向服务计算平台对高速到达的事件流上需高效处理复合事件的需求, 设计了阶段化事件驱动思想的并发处理机制和基于事件代数的事件流处理模型. 关于计算平台的通信模型设计的详细介绍请参见论文前期工作^[5].

2 事件驱动体系结构概述

事件驱动体系结构在很多领域中均有应用, 例如汇编语言中的中断处理、图形界面设计中的 MVC、事件通信如发布/订阅^[6], 主动数据库中的 ECA 规则^[7], 复合事件处理以及事件流处理^[2]等.

事件指系统或环境中发生的活动或状态变化^[2]. Rosenblum 提出了 Internet 环境下事件驱动体系结构的通用设计, 用七个模型对之进行分析和设计^[8], 奠定了大规模分布环境下事件驱动体系结构设计的基础. Mandy 从软件体系结构风格的角度, 提出事件驱动体系结构是以“推”(push)模式的事件通信模型^[1]为中心的应用软件体系结构风格^[9], 由事件感知 agent、事件处理 agent、事件响应 agent 以及事件通信服务四个构件组成. 事件通信在时间、空间、同步三个维度实现了松耦合^[1], 即通信方不必在通信时同时处于激活状态, 通过事件主题/内容等逻辑寻址而非物理寻址进行多元通信, 通信方在通信时不阻塞其控制流. “推”模式指事件产生时由事件源向事件目的发送事件的一种通信模式^[1].

通过使添加侦听事件的新构件变得容易(可扩展性), 鼓励使用通用的事件接口和集成机制(可重用性), 允许构件被替换而不会影响其他构件的接口(可进化性), 事件驱动体系结构为可扩展性、可重用性和可进化性提供了强有力的支持^[4]. 通常事件驱动体系结构通过一个事件总线进行事件通信. 在事件高速突发到和事件总线单点故障的情况下会带来事件总线的性能瓶颈和可伸缩性问题^[1,4], 解决方案一般以降低事件总线简单性为代价, 如提供多个分布事件通信服务和在事件通信服务中支持基于内容的事件过滤和事件组合等. 事件驱动体系结构的另一个缺点是: 难以预料一个动作将会产生什么样的响应(缺乏可理解性)^[4]. 本文通过优化设计框架、在框架中支持复合事件通信和处理来解决这些问题. 优化设计框架中提供阶段化事件处理, 使事件处理构件模块化的同时又提高并发处理能力, 提高系统可理解性和可伸缩性; 针对事件流特性, 基于事件代数的复合事件处理通过事件关联来增强系统的事件表达能力. 事件关联是对系统中多节点处以及系统外部生成的大量事件进行深化处理的过程, 首先对事件进行语法层面的同一化处理, 然后采用规则匹配、推理等多种关联策略对大量事件进行语

义层面的分析,根据事件之间存在的时序关系、因果关系等关联关系,将多个事件关联成为复合事件,从而减少事件的数量,提高事件的质量,从全局的角度描述系统周边态势^[2].

表 1 总结了事件驱动体系结构与请求/响应模型的区别.

表 1 请求/响应与事件驱动模型对比		
	请求/响应模型	事件驱动模型
一次服务中参与的构件数	1:1	n:n
耦合程度	时间、空间、同步紧耦合	时间、空间、同步松耦合
业务处理路径	预定义的线性、串行路径	动态(非线性,可回馈处理)、并发路径
反映性	只在闭合环境下有反映	动态,可对外部环境作出反映

3 事件驱动体系结构的框架设计

为了支持事件驱动体系结构,服务平台扩展了 JBI 的正规化路由器^[10]. 总体框架如图 1. 从体系结构上看,事件驱动的正规化路由器由三部分组成:事件代理(Broker)、事件分派策略(Delivery Policy)以及所有在服务容器上注册的服务的运行状态(Shared State). 事件代理(Broker)负责管理事件分发的整个过程,是事件驱动服务通信模型中正规化

路由器的核心. 事件分派策略是把事件分派到事件目的地时使用的策略. 服务状态提供所有服务容器上注册的服务构件的实时运行状态以及构件展现的服务元数据.

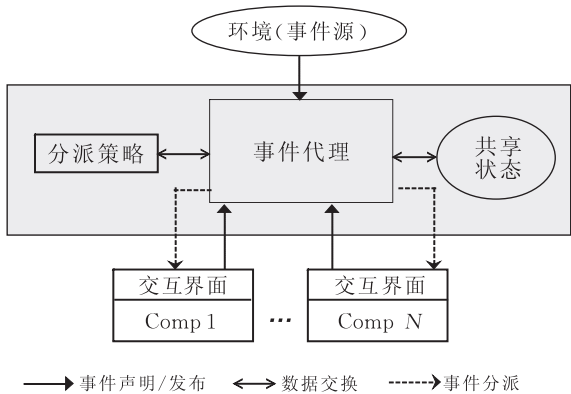


图 1 服务计算平台的事件驱动体系结构

基于图 1 所示的结构,面向服务计算平台中事件驱动体系结构的行为模型如图 2. 其中,感知行为收集事件的状态、上下文(时间以及因果)、其它元数据. 捕获行为负责对收集到的事件进行语法上的同一化,允许为事件添加附加元数据,例如用于度量此事件所需的处理成本属性和唯一标识此事件的 key 等. 收集行为用于对事件状态进行集中,使用内存队列或者持久对象存储到状态库中. 在收集步骤中,事件可根据一些等价类定义进行分类聚合,可以根据

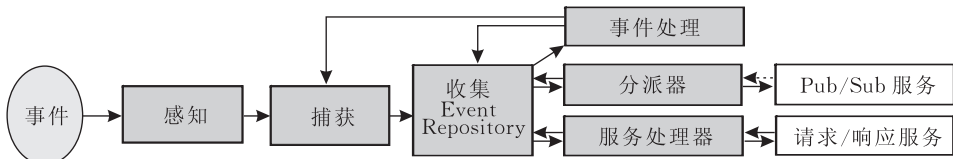


图 2 事件驱动体系结构行为模型

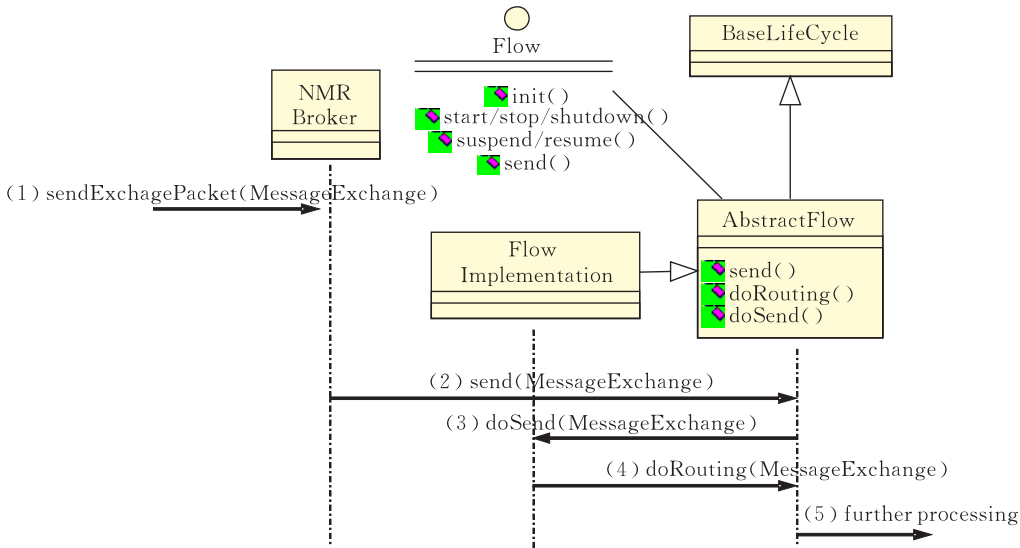


图 3 负责事件分派的流

某些过滤原则进行过滤以及事件排序等. 事件处理用于多源事件深化处理, 生成复合事件. 另外, 事件处理还可以调整捕获步骤的过滤器来改变哪些事件应当被捕获的规则, 复合事件处理在第 5 节作详细讨论.

如图 1 中所示的事件分派策略很大程度上影响服务间的事件通信. 为了支持请求/响应模型, 也支持发布/订阅的事件通信, 采用图 3 所示的事件分派框架. 图 3 中使用了分派流(Flow)的抽象概念. 分派流决定了事件代理分派事件到事件目的地的机制. 抽象流(AbstractFlow)基于模板方法设计模式, 指明事件分派策略基本流程, 其中定义的用来指定事件分派的 doSend 方法是个抽象逻辑, 由具体实现类实现, 分派流的各种具体实现使用自身的事件分派协议来决定分派策略.

4 模块化和高并发：阶段化事件驱动架构

高负载情况下, 多线程并发方法因为使用堆栈,

性能高, 但带来上下文切换开销代价大的问题. 事件驱动由于使用协作式多任务机制, 同步开销小, 另外还有控制流灵活等优点, 其缺点在于设计复杂, 严重依赖动态内存分配. 阶段化事件驱动架构(Staged Event-Driven Architecture, SEDA)的优势在于, 与传统的并发系统体系结构不同, 兼具事件驱动和多线程的优点, 在事件驱动体系结构中引入模块化思想, 通常将系统分解为模块, 使用显式事件队列进行模块间通信. SEDA 的具体思想为: 发送到系统、服务的请求分割为一套链接的阶段(stage). 每个阶段有自己独立的运行参数, 阶段执行请求所需的全部工作负载中自己的部分. 阶段间由请求队列连接, 如果不存在互斥, 阶段间便可以并发执行. 对每个阶段, 通过控制每个阶段管理请求的速度, 可以执行集中的负载管理^[11].

SEDA 的概念模型如图 4, 包括五个部分: 请求事件队列(Incoming Event Queue)、事件处理器(Event Handler)、线程池(Thread Pool)、资源控制器(Resource Controller)、管理控制器(Admission Controller).

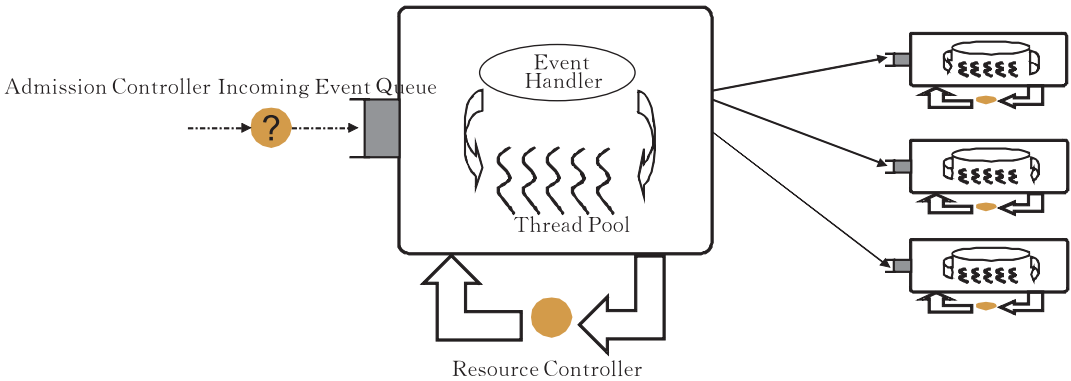


图 4 SEDA 并发模型

结合图 2 中事件驱动体系结构的行为模型, 在面向服务计算平台中设计了如下的 SEDA 模型:

- (1) 每个服务构件接收事件与处理对应于一个阶段, 一次服务调用包括多个服务构件间的事件通信与处理. 各阶段包括事件感知、事件捕获、事件收集、事件分派以及事件处理. 此为阶段化事件驱动的整体结构.
- (2) 资源控制器通过动态调整阶段的参数, 设置阶段为最佳运行状态. 例如, 资源控制器基于阶段提供的载入与性能调整每个阶段中的线程执行数.
- (3) 管理控制器是负载管理的中心, 包括对构件的动态监控与静态监控结合. 静态监控是对构件

接收消息指定某一固定阈值, 根据此阈值来控制请求事件队列. 构件的动态监控表现为统计输入、输出消息, 当(动态感知到的)性能阈值到达时触发事件拒绝逻辑. 另外, 我们还在消息元数据中设计了一个表示事件执行成本的属性字段, 用来表示此事件消息将耗费的吞吐量等成本, 在构件中通过对这些成本的评估进行优先级分类, 依据优先级进行事件排队入列.

- (4) 资源控制器以及管理控制器都提供请求事件队列以及资源状态的信息, 包括队列何时达到上限, 资源何时用尽等信息. 这些信息可以提供给事件处理器, 系统可采取策略来管理负载. 服务计算平

台采用服务质量降级策略来管理负载,例如可靠传输保证由确保发送降级为尽最大可能发送。

(5)由上述的管理控制器把阶段运行状态关联起来,每个下游阶段的运行状态都通过链接传递给上游阶段。上游阶段通过获知下游的运行状态,便可利用(2),(3),(4)中提到的三种方法来管理负载。

5 事件流上的复合事件处理

复合事件处理通过对事件关联关系进行组合,减少事件数量,提高事件质量。事件的质量是应用相关的,因此先给出基于事件代数的复合事件语法和通用语义定义,随后给出复合事件的上下文语义。

5.1 事件模型

为了便于后文讨论,给出事件模型等相关概念定义。

定义 1(时间). 现实世界中时间是连续的,但在计算机世界中一般认为时间是离散而有序、等长单位、与自然数同态的域。因此时间就类似一条具有起点的射线轴,物理时间可映射到时间轴。设 $dom(Time)$ 表示时间的取值范围,则可表示为 $dom(Time) = \mathbb{N}$ 。

定义 2(事件). 事件语义上指发生的活动或者状态变化,但同时还包含其在计算机中的表示。事件由事件类型来指明其结构,每次具体事件类型的发生称为事件实例。发送到复合事件处理方的外部事件称为原子事件,为瞬时的、原子的。复合事件是由原子事件或者复合事件通过事件操作子组合的事件。

形式地,事件 e 表示为一个三元组 $\langle E, V, T \rangle$ 的集合。其中 E 为事件的标识符, V 是一个属性(集),每个元素称作 e 的一个属性值,为了讨论简单,不失一般性,设事件只有一个属性。 T 如定义 1。一个原子事件实例 p 是一个单元素的三元组集合 $\{\langle p, v, t \rangle\}$ 。一个事件实例 e 是 n 个原子事件实例的并,其中 $n > 0$ 。

定义 3(复合构造函数 \oplus). 设 $D = \{e | e \text{ 是事件实例}\}$ 。 \oplus 是 $D \times D \rightarrow D$ 的抽象函数,满足交换律和结合率。对任何 $p_1, p_2 \in D$, $E = p_1 \oplus p_2$, p_1 和 p_2 称为 E 的成分事件。

定义 4(start 函数). $D = \{e | e \text{ 是事件实例}\}$, $start$ 是 $D \rightarrow Time$ 的函数。对任何 $p_1, p_2 \in D$, 满足 $start(p_1 \oplus p_2) = \min(start(p_1), start(p_2))$ 。

定义 5(end 函数). $D = \{e | e \text{ 是事件实例}\}$,

end 是 $D \rightarrow Time$ 的函数。对任何 $p_1, p_2 \in D$, 满足 $end(p_1 \oplus p_2) = \max(end(p_1), end(p_2))$ 。

事件 e 的发生时间定义为 $\langle start(e), end(e) \rangle$ 。因此复合事件是持续的事件,用间隔表示发生时间。

定义 6(事件流). 事件流是事件实例的集合。

特别地,原子事件流 S 是满足如下条件的事件流

(1) $\forall e(e \in S \Rightarrow start(e) = end(e))$;

(2) $\forall e \forall e'(e \in S \wedge e' \in S \wedge end(e) = end(e') \Rightarrow e = e')$ 。

事件流是系统接收的不断到达的事件序列的抽象,是事件实例的集合,即原子事件集合的集合。

定义 7(事件表达式). 如果 $A \in P$, 其中 P 是系统能识别的原子事件的标识符的有穷集合,称 A 为事件表达式。如果 A, B 是事件表达式, $A+B, A|B, AB, A;B, A^*, A-B, A \parallel B, A^N, A^G, A_T$ 也是事件表达式。

5.2 语义

下面的事件操作子定义了复合事件的语义。

5.2.1 原子事件语义

定义 8(事件解释 I). I 是 $P \rightarrow S$ 函数。其中 P 是系统能识别的原子事件的标识符的有穷集合, S 是原子事件流。

定义 9. 对原子事件表达式 $A \in P$, 在事件解释 I 下的含义为 $A^I = I(A)$ 。

5.2.2 复合事件语义

定义 10. 对事件流 M, N , 时间区间 $T \in \mathbb{N}$ 表示事件流持续时间的约束,定义如下函数:

(1) $conjunction(M, N) \equiv \{m \cup n | m \in M \wedge n \in N\}$;

(2) $disjunction(M, N) \equiv M \cup N$;

(3) $concatenation(M, N) \equiv$

$\{m \oplus n | m \in M \wedge n \in N \wedge (end(m) = start(n)) \vee ((start(m) < start(n)) \wedge (end(m) > start(n))) \vee ((start(m) < start(n)) \wedge (end(m) = end(n))) \vee ((start(m) = start(n)) \wedge (end(m) < end(n))) \vee ((start(m) < start(n)) \wedge (end(m) > end(n)))\}$;

(4) $sequence(M, N) \equiv$

$\{m \oplus n | m \in M \wedge m \in N \wedge end(m) < start(n)\}$;

(5) $concurrency(M, N) \equiv$

$\{m \oplus n | m \in M \wedge n \in N \wedge start(m) = start(n) \wedge end(m) = end(n)\}$;

(6) $iteration(M) \equiv M \cup conjunction(M, M) \cup$

$conjunction(M, conjunction(M, M)) \cup \dots$;

(7) $\text{negation}(M, N) \equiv$

$$\{m \mid m \in M \wedge \neg \exists n(n \in N \wedge \text{start}(m) \leq \text{start}(n) \wedge \text{end}(n) \leq \text{end}(m))\};$$

(8) $\text{selection}(M, \text{funcSelection}) \equiv$

$$\{m \mid m \in M \wedge \text{funcSelection}(m)\};$$

(9) $\text{aggregation}(M, \text{funcAggregation}) \equiv$

$$\{m \mid m \in M \wedge \text{funcAggregation}(m)\};$$

(10) $\text{tim_restriction}(M, T) \equiv$

$$\{m \mid m \in M \wedge \text{end}(m) - \text{start}(m) \leq T\}.$$

定义 11. 对事件表达式 A, B , 定义其在事件解释 I 下的含义为

$$(1) (A + B)^I = \text{conjunction}(A^I, B^I);$$

$$(2) (A | B)^I = \text{disjunction}(A^I, B^I);$$

$$(3) (AB)^I = \text{concatenation}(A^I, B^I);$$

$$(4) (A; B)^I = \text{sequence}(A^I, B^I);$$

$$(5) (A \parallel B)^I = \text{concurrency}(A^I, B^I);$$

$$(6) (A^*)^I = \text{iteration}(A^I, B^I);$$

$$(7) (A - B)^I = \text{negation}(A^I, B^I);$$

$$(8) (A^N)^I = \text{selection}(A^I, \text{funcSelection});$$

$$(9) (A^G)^I = \text{aggregation}(A^I, \text{funcAggregation});$$

$$(10) (A_T)^I = \text{tim_restriction}(A^I, T).$$

通过对事件表达式中的每个操作子递归地应用定义 10 中的函数, 得到事件表达式所代表的复合事件的语义。

5.3 事件上下文

用户提交了复合事件表达式后, 参与事件关联的所有满足事件操作子语义的事件实例都会输出。当事件流到达速率快时, 在线复合事件处理需要耗用的内存和时间是巨大的, 同时也很难满足个性化定制。因此需要对参与事件关联的事件实例进行剪枝, 使之在满足应用需求输出足够的复合事件结果的同时, 耗用内存和时间资源越少。

从 5.2 节定义的事件操作子语义定义中可看出, 只有时间约束(A_T)、选择操作(A^N)可以剪枝参与事件关联的事件实例, 但这些操作子对于事件的剪枝是不够的。针对此问题, 扩展通用复合事件语义, 给出适合大部分事件关联需求的上下文语义。

基本的思路是: 如果事件流包含具有同时结束($\text{end}(e)$ 相同)的事件实例, 上下文语义只选取具有最近开始的事件实例($\text{start}(e)$ 最大)中的一个。形式上, 定义事件流上的 context 性质:

定义 12(上下文 context)。对于任何两个事件流 S, S' , 如果下列条件满足, 则称 S 与 S' 满足上下文性质, 记为 $\text{context}(S, S')$ 。应用 context 性质后得

到的事件流称为上下文事件流。

$$(1) S' \subseteq S;$$

$$(2) \forall s(s \in S \Rightarrow \exists s'(s' \in S' \wedge \text{start}(s) \leq \text{start}(s') \wedge \text{end}(s) = \text{end}(s')));$$

$$(3) \forall s, s'((s \in S' \wedge s' \in S' \wedge \text{end}(s) = \text{end}(s')) \Rightarrow s = s').$$

定义中条件(2)说明在同时结束的实例中选择最近开始的事件实例, 条件(3)保证只选择其中一个。这说明只要在已发生的事件序列中有此表达式相应的出现, 则肯定能给出一个输出。虽然这种定义对有些应用并不满足要求, 对大多数应用来说, 这种上下文可以递归地应用到子表达式中, 是有效的剪枝策略。

需要注意的问题是, 上下文性质在不同子表达式中的应用是否会产生冲突, 是否会影响操作子的组合。应该保证上下文性质递归地应用到子表达式时与只应用到顶层表达式时产生的结果是一样有效的。定理 1 可以保证在事件关联用户规约事件时只使用一次上下文性质约束, 而在事件检测算法中可自由地应用到各层子表达式中。

定理 1. 如果 $\text{context}(S, S')$ 与 $\text{context}(T, T')$ 成立, 对任何事件流 U 和 $t \in \text{dom}(\text{Time})$, 下列结论成立:

$$(1) \text{context}(\text{conjunction}(S', T'), U) \Rightarrow \text{context}(\text{conjunction}(S, T), U);$$

$$(2) \text{context}(\text{disjunction}(S', T'), U) \Rightarrow \text{context}(\text{disjunction}(S, T), U);$$

$$(3) \text{context}(\text{concatenation}(S', T'), U) \Rightarrow \text{context}(\text{concatenation}(S, T), U);$$

$$(4) \text{context}(\text{sequence}(S', T'), U) \Rightarrow \text{context}(\text{sequence}(S, T), U);$$

$$(5) \text{context}(\text{concurrency}(S', T'), U) \Rightarrow \text{context}(\text{concurrency}(S, T), U);$$

$$(6) \text{context}(\text{iteration}(S'), U) \Rightarrow \text{context}(\text{iteration}(S), U);$$

$$(7) \text{context}(\text{negation}(S', T'), U) \Rightarrow \text{context}(\text{negation}(S, T), U);$$

$$(8) \text{context}(\text{selection}(S', \text{funcSelection}), U) \Rightarrow \text{context}(\text{selection}(S, \text{funcSelection}), U);$$

$$(9) \text{context}(\text{aggregation}(S', \text{funcAggregation}), U) \Rightarrow \text{context}(\text{aggregation}(S, \text{funcAggregation}), U);$$

$$(10) \text{context}(\text{tim_restriction}(S', t), U) \Rightarrow \text{context}(\text{tim_restriction}(S, t), U).$$

证明。对于 conjunction 来说, 假设 $\text{context}(\text{con-}$

$junction(S', T'), U$ 成立. 因此, 对任意 $u \in U$ 有 $u \in context(S', T')$, 而且对 $s \in S', t \in T'$ 有 $u = s \oplus t$. 由 $context$ 定义条件(1)可知, $s \in S, t \in T$. 得 $u \in context(S, T)$, 故 $U \subseteq context(S, T)$.

任取 $u \in context(S, T)$, 对 $s \in S, t \in T$ 有 $u = s \oplus t$. 由 $context$ 定义条件(2), 存在 $s' \in S', t' \in T'$, $start(s) \leq start(s')$ 且 $end(s) = end(s')$, $start(t) \leq start(t')$ 且 $end(t) = end(t')$. 设 $u' = s' \oplus t'$. 得 $u' \in context(S', T')$, $start(u) \leq start(u')$ 且 $end(u) = end(u')$. 即存在 $u'' \in U$, $start(u) \leq start(u'')$ 且 $end(u) = end(u'')$. 满足 $context$ 定义的第 2 个条件.

由假设 $context(conjunction(S', T'), U)$ 可知, U 中所有实例的 end 时间是不同的. 因此满足 $context$ 的第 3 个条件.

由 $context$ 定义, 可知对于 $conjunction$, 可推出 $context(conjunction(S, T), U)$. 对其它各函数, 同理可证. 定理 1 得证. 证毕.

定理 2. 如果 $context(S, T)$ 与 $context(S, T')$ 成立, 则对任何 $t \in T$, 存在 $t' \in T'$, 满足 $start(t) = start(t')$ 且 $end(t) = end(t')$.

证明. 因为 $T \subseteq S, t \in S$. 由 $context$ 的第 2 个条件定义, 存在某个 $t' \in T'$, 使得 $start(t) \leq start(t')$ 且 $end(t) = end(t')$. 对 $t' \in S$, 存在某个 $t'' \in T$, 使得 $start(t') \leq start(t'')$ 且 $end(t') = end(t'')$. 根据 $context$ 的第 3 个条件定义, $t = t''$, 有 $start(t) \leq start(t') \leq start(t)$, 因此 $start(t') = start(t)$. 对 end , 同理可证. 证毕.

定理 2 说明一个事件流可能有多个可能的上下文事件流匹配, 但是上下文事件流之间在时间间隔上是相同的.

定义 13(事件表达式等价). 对于任何两个事件表达式 A, B , 如果对任何解释 I 有 $A^I = B^I$, 则称 A 与 B 等价, 记为 $A \equiv B$.

定理 3. 如果 $A \equiv A', B \equiv B'$, 则 $A + B \equiv A' + B', A | B \equiv A' | B', AB \equiv A'B', A; B \equiv A'; B', A^* \equiv A'^*, A - B \equiv A' - B', A \parallel B \equiv A' \parallel B', A^N \equiv A'^N, A^G \equiv A'^G, A_T \equiv A'_T$.

证明. 由定义 11 可直接推出. 证毕.

显然, “ \equiv ”关系是一个等价关系. 从定理 3 来看, \equiv 满足替换条件, 因此 \equiv 定义了事件表达式上的结构一致性.

定理 4. 如果 $A \equiv A', context(A^I, S)$ 成立, 则 $context(A'^I, S)$ 也成立.

证明. 由 \equiv 的定义, $A \equiv A'$ 蕴涵 $A^I = A'^I$. 定

理 4 成立.

证毕.

定理 4 说明, 任何满足 $context$ 性质的具体实现, 只要 $A \equiv A'$ 成立, 检测 A 发生的实现同样对 A' 有效. 此定理保证对一个事件表达式 A 来说, 为了降低时间空间复杂性, 可以为其计算满足 $context(A^I, S)$ 成立的事件流 S , 而不用直接计算 A^I . 这种带限制的复合事件语义可以保证上述复合事件操作子良好的代数性质, 既方便事件关联应用对事件表达式在时间维上的推理, 确保其复合事件表达满足应用需求, 又能降低检测的时间和空间复杂性.

5.4 检测算法

下面给出相应的事件检测算法. 设要检测的复合事件表达式为 $expression$, 函数 $getSubExpressionCount(expression)$ 返回按照括号优先原则, 从左自右、从底向上, 其所有可分解的子表达式个数, $getSubExpressionIndex(expression, i)$ 返回相应的第 i 个子表达式, 并赋值到集合 E 中第 i 个元素 E^i , $getSubExpressionCount(expression) = k$, 第 k 个子表达式就是 $expression$, 而起始元素 $E^1 \in P$, 即原子事件类型集合. 算法以计时系统计时精度单位为周期触发循环迭代, 动态选择子表达式, 计算满足上下文性质 $context(E^i, v_n)$ 成立的 v_n , 其中集合 v 用来存储输出复合事件的实例, 最后一个输出 v_k 即是算法检测得到的复合事件实例. 算法每次迭代操作保存相关状态信息, 包括事件历史、时间信息, 相应的这些信息按序号递增保存在不同集合中, 集合大小均为 k . 集合 x 用于保存树形结构操作左边的表达式的过往事件实例, 集合 y 保存右边表达式的过往事件实例, t 为时间常量, 集合 Q 用于表示顺序型操作子左边表达式的多个实例.

算法 1. 事件检测算法.

1. PROCEDURE *detect_event* (*expression*: String)
2. VAR $i, k, slength, qlength$: integer;
3. v, x, y, Q : $ETypes[]$; Q' : $Etypes$; t, S : $Time[]$.
4. BEGIN
5. $n = getSubExpressionCount(expression)$
6. FOR $i = 1$ to n
7. $E^i = getSubExpressionIndex(expression, i)$
8. CASE E^i OF
9. [$E^i \in P$]: $v_i = E^i \cup \varphi$
10. [$E^x + E^y$]:
11. CASE timestamp OF
12. [$start(x_i) < start(v_x)$]: $x_i = v_x$
13. [$start(y_i) < start(v_y)$]: $y_i = v_y$

```

14.  $[x_i = \varphi \vee y_i = \varphi \vee (x_i = \varphi \wedge y_i = \varphi)]: v_i = \varphi$ 
15.  $[start(v_y) \leq start(v_x)]: v_i = x_i \oplus v_y$ 
16.  $[OTHERS]: v_i = v_x \oplus v_y$ 
17.  $S_i = S_x \cup S_y \cup \{start(v_x), start(v_y)\} \setminus \{-1\}$ 
18.  $[E^x | E^y]:$ 
19. CASE timestamp OF
20.  $[start(v_x) < start(v_y)]: v_i = v_x$ 
21.  $[OTHERS]: v_i = v_x$ 
22.  $S_i = S_x \cup S_y$ 
23.  $[E^x E^y]:$ 
24.  $e' = \varphi$ 
25. REPEAT  $e$  in  $Q_i \cup \{x_i\}$ 
26. IF  $end(e) = start(v_y) \wedge start(e') < start(e)$  THEN
     $e' = e$ 
27. IF  $e' \neq \varphi$  THEN  $v_i = v_y \oplus e'$  ELSE  $v_i = \varphi$ 
28.  $Q' = \varphi$ 
29.  $Q_i = Q_i \cup \{x_i\}$ 
30.  $slength = length(S_y)$ 
31.  $qlength = length(Q_i)$ 
32. REPEAT  $slength > 0 \wedge qlength > 0$ 
33.  $tp = S_y[slength]$ 
34.  $e = Q_i[qlength]$ 
35. IF  $end(e) < tp$  THEN
36.  $Q' = Q' \cup \{e\}$ 
37.  $slength = slength - 1$ 
38. ELSE  $qlength = qlength - 1$ 
39.  $Q_i = Q'$ 
40. IF  $start(x_i) < start(v_x)$  THEN  $x_i = v_x$ 
41.  $S_i = S_x \cup \{start(e) | e \in Q_i \cup \{x_i\}\} \setminus \{-1\} [E^x; E^y]:$ 
42.  $[E^x; E^y]:$ 
43.  $e' = \varphi$ 
44. REPEAT  $e$  in  $Q_i \cup \{x_i\}$ 
45. IF  $end(e) < start(v_y) \wedge start(e') < start(e)$  THEN
     $e' = e$ 
46. IF  $e' \neq \varphi$  THEN  $v_i = v_y \oplus e'$  ELSE  $v_i = \varphi$ 
47.  $Q' = \varphi$ 
48.  $Q_i = Q_i \cup \{x_i\}$ 
49.  $slength = length(S_y)$ 
50.  $qlength = length(Q_i)$ 
51. REPEAT  $slength > 0 \wedge qlength > 0$ 
52.  $tp = S_y[slength]$ 
53.  $e = Q_i[qlength]$ 
54. IF  $end(e) < tp$  THEN
55.  $Q' = Q' \cup \{e\}$ 
56.  $slength = slength - 1$ 
57. ELSE  $qlength = qlength - 1$ 
58.  $Q_i = Q'$ 
59. IF  $start(x_i) < start(v_x)$  THEN  $x_i = v_x$ 

```

```

60.  $S_i = S_x \cup \{start(e) | e \in Q_i \cup \{x_i\}\} \setminus \{-1\}$ 
61.  $[E^x \parallel E^k]:$ 
62.  $e' = \varphi$ 
63. REPEAT  $e$  in  $Q_i \cup \{x_i\}$ 
64. IF  $(start(e) = start(v_y)) \wedge (end(e) =$ 
     $end(v_y) \wedge start(e') < start(e))$  THEN  $e' = e$ 
65. IF  $e' \neq \varphi$  THEN  $v_i = v_y \oplus e'$  ELSE  $v_i = \varphi$ 
66.  $Q' = \varphi$ 
67.  $Q_i = Q_i \cup \{x_i\}$ 
68.  $slength = length(S_y)$ 
69.  $qlength = length(Q_i)$ 
70. REPEAT  $slength > 0 \wedge qlength > 0$ 
71.  $tp = S_y[slength]$ 
72.  $e = Q_i[qlength]$ 
73. IF  $end(e) < tp$  THEN
74.  $Q' = Q' \cup \{e\}$ 
75.  $slength = slength - 1$ 
76. ELSE  $qlength = qlength - 1$ 
77.  $Q_i = Q'$ 
78. IF  $start(x_i) < start(v_x)$  THEN  $x_i = v_x$ 
79.  $S_i = S_x \cup \{start(e) | e \in Q_i \cup \{x_i\}\} \setminus \{-1\}$ 
80.  $[E^{x*}]:$ 
81.  $v_i = v_x \cup y_i; x_i = v_i$ 
82.  $S_i = S_x \cup S_y; S_x = S_i$ 
83.  $[E^x - E^y]:$ 
84. CASE timestamp OF
85.  $[t_i < start(v_y)]: t_i = start(v_y)$ 
86.  $[t_i < start(v_x)]: v_i = v_x$ 
87.  $[OTHERS]: v_i = \varphi$ 
88.  $S_i = S_x$ 
89.  $[E^{xN}]: v_i = funcSelection(x_i, v_x); x_i = v_i; S_i = S_x$ 
90.  $[E^{xG}]: v_i = funcAgregation(x_i, v_x); x_i = v_i; S_i = S_x$ 
91.  $[E_i^x]:$ 
92. CASE timestamp OF
93.  $[(end(v_x) - start(v_x)) \leq t]: v_i = v_x; S_i = S_x$ 
94.  $[OTHERS]: v_i = \varphi; S_i = \varphi$ 
95. ENDFOR
96. END

```

5.4.1 时间复杂性分析

事件检测时从左自右、从底向上分解子表达式。应用上下文语义进行检测，只保存最大启动时间事件在变量中，其中连接(AB)和并发($A \parallel B$)可看作顺序操作子($A; B$)的特例。顺序操作子检测中第一个内循环执行 $|Q_i| + 1$ 次，第二个循环执行次数为 $|Q_i| + 1 + |S_y|$ ，最后的赋值操作执行需 $O(|S_i|c)$ ，其中 $O(c)$ 为 \oplus 函数的时间复杂度。可以证明， $|Q_i|$ ， $|S_i|$ ， $|S_y|$ 均小于 k ，因此检测算法的时间复杂度为

$O(k^2c)$.

6 实 验

计算平台的综合性能实验结果参见文献[5],这里给出并发处理和事件流上复合事件处理的性能实验结果.复合事件处理需要很高的计算能力,本实验与文献[5]中不同,配置的客户端和服务计算平台的硬件均增强为多核配置,具体如表 2.用于测试的事件流数据集为模拟的安全事件数据库,每个数据库都包含 1000 个不同源 IP 伪造 VPN 登录实施 Telnet 操作的攻击场景(含三个事件),其余为非相关事件,分别保存在每个客户端.客户端从自己的数据库中读取事件元组,根据事件表达式需要,模式为(eventID varchar(4), timestamp long, attackIP long),以 100000 事件/s 的速率,通过网络并发地发送到服务计算平台,服务计算平台中注册事件表达式,对到达的事件流进行处理.测试均运行 5min 后计算实验结果,以保证性能优化发挥作用.

表 2 实验环境配置	
项	配置情况
服务计算平台和客户端操作系统	Red Hat 64 位 Enterprise Linux AS release 4
服务计算平台硬件	2 x Intel Xeon 5130 2GHz(共四核), 16GB RAM
客户端硬件	2 x Intel Xeon 5130 2GHz(共四核), 4GB RAM 客户端均在一个机柜
网络	100Mbit/s
服务计算平台和客户端 JVM 版本	BEA JRockit(R)R27.3
服务计算平台 JVM 参数	-Xms2g -Xmx2g -Xns128m -Xgc:gencon

实验分别测试 4 种长度的表达式的处理性能. $T=60000$, $funcSelection=\{attackIP=x\}$, 分别注册 1000 个不同长度的事件表达式,依次为

$$A=VPNLogin,$$
$$B=VPNLogin; telnetLogin,$$
$$C=(VPNLogin; telnetLogin)_T^N,$$
$$D=((VPNLogin; telnetLogin)-$$
$$DomainLogin)_T^N.$$

表达式 D 表示防止来自源 IP 为 x ,通过 VPN 登录系统后 1min(60000ms)内未经域登录的 Telnet 操作,这表明攻击试图通过 VPN 伪造合法用户跳过域登录直接进行 Telnet 操作.4 个表达式之间具有相关性,方便对事件表达式长度影响事件处理

的性能进行比较.

每个事件元组大小为 24Bytes,因为 TCP 包头还占用空间,100Mbit/s 的网络最多允许 5 个客户端同时发送事件.面向服务计算平台的事件处理时间分布如图 5,平均处理时间如图 6 所示.结果表明,使用 2 个双核 CPU 的配置,500000 事件/s 的负载,1000 个表达式,事件的平均处理时间不超过 1800ms,且 90%的事件的处理时间小于 4ms. A,B,C,D 的表达式长度分别为 1,2,4,6,如图 6,在 1,3,4,5 个客户端发送事件的负载下各表达式平均处理时间之比为 18:75:295:659, 26:113:435:1005, 33:137:539:1237, 47:205:809:1798,可见在网络带宽允许条件下,事件的平均处理时间与表达式长度的平方近似成正比.

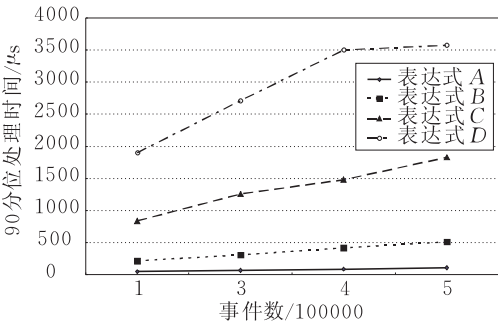


图 5 服务计算平台事件处理时间的分布

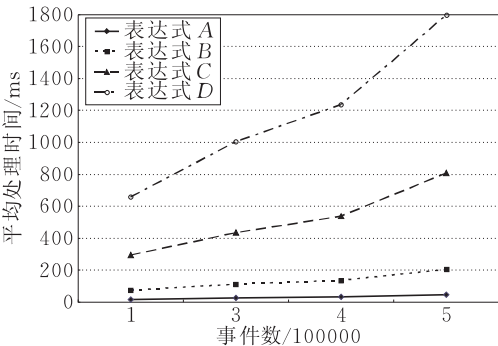


图 6 服务计算平台事件平均处理时间

7 相关工作

在应用领域松耦合应用集成和协同计算需求的驱动下,分布计算平台中支持事件驱动体系结构风格的设计在学术界和工业界近年来成为研究热点^[12],如 IBM Research 的 AMIT^[13]、Microsoft Research 的 CEDR^[14]、StreamBase^[15] 以及开源项目 Esper^[16],研究性项目有 UMass 的 SASE^[17]、Berkeley 的 TelegraphCQ 扩展^[18] 以及 Cornell 的 Cayuga^[19] 等.

事件驱动体系结构在带来控制灵活、松耦合通信以及反应性的同时,也存在事件总线性能瓶颈、可伸缩性问题以及系统设计复杂、系统难理解等问题。事件驱动体系结构中事件的表达能力与系统的可伸缩性是一对矛盾,相关工作从不同角度针对这些问题均提出了相应的设计,以求在事件的表达能力和系统的可伸缩性方面寻求权衡。从事件驱动体系结构的应用领域看,大体可分为反应式系统^[13]、事件通信^[19]以及系统监控^[15-18,20]等领域。

7.1 反应式系统

反应式系统(reactive system)是指与环境有着持续不断交互的系统^[3],典型的例子有嵌入式系统、操作系统和 Web 服务器等。它是与转换式系统(transformational system)相对而言的。主动数据库^[7]可归为此类应用。主动数据库是相对传统的、被动的数据库而言,通过定义在数据库内部和外部发生有意义的事件时其应该作出的反应实现反应性。这种反应通常都由所谓的 ECA 规则来表示。为了增强灵巧性,很多主动数据库允许规则可由复合事件触发。AMIT 基于主动数据库概念,定义了与本文中复合事件类似的情势(situation)概念,通过规则触发来检测复合事件^[13]。另外,部分主动数据库也提供类似本文中的事件上下文来剪枝参与事件关联的事件实例^[21], Zimmer 给出了主动数据库中支持事件驱动体系结构风格和事件关联的形式化定义和综合比较^[22]。

在体系结构上,反应式系统因为事件到达速率不高,因此其定义的事件语言的表达能力都很强大,但可伸缩性不高。在事件模型上反应式系统一般是特定领域的,例如主流的主动数据库一方面用点时间来表示复合事件的发生^[13],因此带来 Galton 指出的检测语义错误^[23];一方面事件概念局限于数据库操作或系统时钟事件范畴,因此限制了其使用的领域。针对这些问题近年来出现了使用间隔语义来表示复合事件的主动数据库^[24]。另外,本文提出的上下文性质与 Snoop 中的 Recent 消费策略^[22]类似,独特之处在于上下文性质一次应用到表达式后在事件检测过程中可重复应用到子表达式中,且检测的效果是有效的。

7.2 事件通信

基于事件通信的系统通常要处理高并发的事件到达,体系结构上事件通信围绕事件订阅和匹配进行研究,提供基于内容的订阅和匹配,事件模型上不太关注复合事件的处理^[1]。Cayuga 声称其事件的表

达能力强大、可伸缩性高,使用非确定性有限自动机检测复合事件,事件检测时基于状态合并和多查询优化技术来解决系统的伸缩性问题^[19]。与本文相比,其使用的事件模型中的复合事件是基于点时间的,导致在检测时出现与 AMIT 类似的语义错误;另外其采用的用于事件关联时剪枝事件实例的设计,是非形式定义的,固化在系统代码中,不利于对事件表达式进行静态优化和推理,也限制了事件关联用户的灵活应用^[19],其事件检测的时间复杂度也没有明确给出。

7.3 系统监控

在系统监控应用中,事件被用于对复杂系统(包括软件系统和网络,以及现实世界系统如金融交易、RFID、传感器网络)进行管理、监控和分析。数据流管理系统^[14-17]、事件流处理^[17-18]、网络安全监控^[25]等都属于此范畴。这些系统一部分是通过扩展数据流模型,即通过扩展原来的流查询语言,增加流上的事件操作予以支持事件驱动的体系结构风格^[14-17],另一部分是通过时态逻辑等形式化方法表达事件模型,使用算法来检测事件^[25]。Rizvi 通过扩展 TelegraphCQ,提出语义窗口的概念,流的时间窗口不是由时间长度或元组长度结构化定义,而是由事件的发生定义的,并给出基于树的方法来检测复合事件的机制^[18]。TelegraphCQ 扩展、SASE、StreamBase 均使用点时间模型。TelegraphCQ 扩展没有考虑系统性能优化,StreamBase 中具体的事件规约和事件检测方法不明确。SASE 使用扩展数据流的连续查询语言以支持事件的表达和检测,提出了基于查询计划的优化方法,提供高性能的事件流处理,但其同样存在点时间模型的问题,与本文相比,其事件模型中不支持复合事件的递归组合以及事件上的聚合操作,没有事件实例剪枝的解决方案,时间复杂度也未明确给出。使用时态逻辑表达事件模型的网络安全监控^[25],其事件模型表达能力丰富,与本文系统相比,其采用过程性的方法表达事件,用户对事件检测过程控制能力差,不提供事件实例剪枝,对一个长度为 k 的事件表达公式,其检测算法的时间复杂度高于本文,为 $O(2^k)$ ^[25]。

与本文相比,上述支持事件驱动体系结构的分布计算平台体系结构关注反应性与松耦合事件通信的结合不够,大部分只关注其中的一方面而忽略另一方面;在解决系统可伸缩性问题时仅考虑性能提高,未考虑系统可理解性问题;事件模型均没有考虑事件语言上的相关性质,不利于事件表达式检测时

的静态优化(编译时优化);动态优化中对参与事件关联的事件实例剪枝又未给出形式化的定义,大部分固化在系统设计中。

8 结 论

本文针对动态多变的大规模分布式计算环境对面向服务计算平台的松耦合通信的需求,引入事件驱动的体系结构风格设计,针对事件驱动体系结构设计时面临的不可理解性不好、事件总线性能瓶颈、可伸缩性问题以及在事件流上进行复合事件处理的需求,设计了阶段化事件驱动体系结构和支持复合事件的事件流处理机制。实验结果表明,引入事件驱动体系结构风格的面向服务计算平台在优化的设计下具有较好的处理性能和更高的可伸缩性。下一步的工作将主要从以下两方面展开:(1)原子事件目前建模为瞬时事件,在有些应用中不够。例如为了考虑网络传输延迟的影响而把原子事件认为是某个间隔内发生的模型。这需要改进事件的语义和检测算法;(2)针对流处理环境进行进一步优化,例如让多个复合事件的检测能共享其中的中间计算结果。

参 考 文 献

- [1] Eugster P T et al. The many faces of publish/subscribe. *ACM Journal of Computing*, 2003, 35(2): 114-131
- [2] Luckham D C. The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Boston: Addison-Wesley, 2001
- [3] Tombros D, Geppert A, Dittrich K. Semantics of reactive components in event-driven workflow execution//Proceedings of the International Conference on Advanced Information Systems Engineering. Barcelona, Spain, 1997: 409-422
- [4] Fielding R T. Architectural styles and the design of network-based software architectures [Ph. D. dissertation]. Owen Branch, UC, 2000
- [5] Liu Jia-Hong, Wu Quan-Yuan. An event-driven service-oriented computing platform//Proceedings of the CNCC. Suzhou, China, 2007. Beijing: Tsinghua University Press, 2008(in Chinese)
(刘家红,吴泉源. 一个基于事件驱动的面向服务计算平台//全国计算机大会. 苏州, 中国, 2007. 北京: 清华大学出版社, 2008)
- [6] Huang Y, Gannon D. A comparative study of Web services-based event notification specifications//Proceedings of the International Conference on Parallel Processing Workshops (ICPP). 2006: 7-14
- [7] Gatzu S, Dittrich K R. Events in an active object-oriented database system//Proceedings of the International Conference on Rules in Database Systems. Edinburgh, UK, 1993: 23-39
- [8] Rosenblum D S, Wolf A L. A design framework for Internet-scale event observation and notification//Proceedings of the European Software Engineering Conference/ACM SIGSOFT Symposium on the Foundations of Software Engineering. Zurich, Switzerland. Springer-Verlag, 1997: 344-360
- [9] Chandy K M. Sense and respond systems//Proceedings of the 31st Annual International Conference of the Association of System Performance Professionals. Orlando, 2005: 59-66
- [10] Vinoski S. Java business integration. *IEEE Internet Computing*, 2005, 9(4): 89-91
- [11] Welsh M, Culler D, Brewer E. SEDA: An architecture for well-conditioned, scalable Internet services//Proceedings of the Symposium on Operating Systems Principles (SOSP). Chateau Lake Louise, Canada, 2001: 230-240
- [12] Chandy K M. Towards a theory of events//Proceedings of the DEBS. Toronto, Ontario, Canada, 2007: 180-187
- [13] Adi A, Etzion O, Amit — The situation manager. *VLDB Journal*, 2004, 13(2): 177-203
- [14] Barga R S et al. Consistent streaming through time: A vision for event stream processing//Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR). Asilomar, California, USA, 2007: 363-374
- [15] Stonebraker M, Çintemel U, Zdonik S. The 8 Requirements of Real-Time Stream Processing. 2006
- [16] Esper. Esper reference documentation. Esper Team, Wayne, NJ: Technical Report: Esper Reference Documentation Version 1.10.0, 2007
- [17] Wu E, Diao Y, Rizvi S. High-performance complex event processing over streams//Proceedings of the SIGMOD. 2006: 407-418
- [18] Rizvi S. Complex event processing beyond active databases: Streams and uncertainties [Master Thesis]. UCB/EECS-2005-26, 2005: 46
- [19] Demers A et al. Cayuga: A general purpose event monitoring system//Proceedings of the Biennial Conference on Innovative Data Systems Research(CIDR). Asilomar, 2007: 412-422
- [20] Progress Software. The need for speed and agility: Event stream processing for event-driven business. Progress Software, 2005: 1-8
- [21] Adaikkalavan R. Snoop event specification: Formalization, algorithms, and implementation using interval-base semantics [Ph. D. dissertation]. Arlington: University of Texas at Arlington, 2002
- [22] Zimmer D, Unland R. On the semantics of complex events in active database management systems//Proceedings of the International Conference on Data Engineering. New South Wales, Australia, 1999: 392-399
- [23] Galton A, Augusto J C. Two approaches to event definition//Proceedings of the International Conference of Database and Expert Systems Applications. Aix-en-Provence, France, 2002: 547-556

[24] Adaikkalavan R, Chakravarthy S. SnoopIB: Interval-based event specification and detection for active databases//Proceedings of the Conference of Advances in Databases and Information Systems. Dresden, 2003: 190-204

[25] Naldurg P, Sen K, Thati P. A temporal logic based frame-

work for intrusion detection//Proceedings of the IFIP WG International Conference on Formal Techniques for Networked and Distributed Systems (FORTE). Madrid, Spain, 2004: 359-376



LIU Jia-Hong, born in 1980, Ph. D. candidate. His research interests include distributed computing, service-oriented computing and event stream processing.

WU Quan-Yuan, born in 1941, professor, Ph. D. supervisor. His research interests include distributed computing, artificial intelligence.

Background

Many Request/Response based SOA implementations lack flexibility for communication and rapid response for changing conditions. Event-Driven Architecture(EDA) is the choice for implementing multistage processes that deliver information with minimum delay and for responding rapidly and effectively to changing conditions, especially complex events derived from multiple events.

To get a high-performance event-driven computing platform for event concurrency and complex event processing, a unified framework for event processing, a concurrency management service and event correlation service including an expressive event language and a rapid detection mechanism on top should be designed with delicacy. Unfortunately the expressiveness of event language and scalability of framework is contradicting. Reactive systems, event notification systems and system monitoring applications have much appropriate solutions for separate aspects but lack a unified framework.

Complex event can be application's logic, like popular event notification systems' suggest. Systems capable of processing complex events over stream in common use expressive declarative languages to model complex events and get them detected using Petri-Nets or automata based mechanism, some of which initiate auxiliary mechanisms to reduce event instances during event correlation, but their focus is to satisfy applications' semantic, rather to get high-performance event stream processing. Data stream management systems are used for processing data of high arrival rate, but they lack capability to process time-series data in efficiency. This paper presents a unified framework to get high performance by a SEDA based model for processing event concurrency, a context policy based event algebra with well defined algebra properties for processing complex events over stream. The authors justify performance by experiment results.