ConEntity: 一种基于进程的自主并发实体编程模型及其实现

康亮环^{①②}, 曹东刚^{①②*}

① 北京大学信息科学技术学院, 北京 100871

② 北京大学高可信软件技术教育部重点实验室, 北京 100871

* 通信作者. E-mail: caodg@sei.pku.edu.cn

收稿日期: 2012-08-03; 接受日期: 2012-11-20

国家重点基础研究发展计划 (973 计划)(批准号: 2011CB302604) 和国家自然科学基金 (批准号: G61121063, G61272154) 资助项目

摘要 本文在虚拟计算环境之上, 研究支持具有自主能力、高并发的新型互联网应用开发方法, 在已有的基于进程、面向并发的编程模型中引入实体建模机制, 扩展出一种兼具进程和自主并发实体的程序设计模型 ConEntity, 并给出了形式化定义和描述. ConEntity 模型具有表达性、并发性和可伸缩性的特点, 能对虚拟计算环境资源高效、透明访问. 通过扩展 Erlang/OTP 将其实现为 Erlang 语言设施 UniAgent. 本文的模型为在虚拟计算环境上快速直接构建具有自主、高并发能力实体的新型互联网应用提供了模型和语言上的支持.

关键词 编程模型 进程 并发实体 并发编程 Erlang 语言

1 引言

随着多核架构和互联网技术的高速发展, 面对开放、动态、难控的网络环境, 传统的软件技术面临新的需求和挑战^[1]. 为了应对这种挑战, 国内外研究人员从不同角度提出了一些新型计算模式, 如网格计算^[2]、虚拟计算环境^[3]、面向服务的计算^[4,5]和网构软件^[6]等. 其中虚拟计算环境的目标是对网络资源按需聚合和自主协同, 为终端用户或应用系统提供和谐、可信、透明的一体化服务环境实现有效资源共享和便捷合作工作^[3]. 虚拟计算环境的一种重要应用是高并发应用, 因为虚拟计算环境按需聚合资源的特点, 高并发应用研究如何通过大量并发行为来实现对虚拟计算环境所聚合的计算和存储资源高效利用是非常重要的. 在虚拟计算环境之上构建高并发应用需要具有良好表达性、并发性和可伸缩性的编程模型支持, 从而为开发者提供直接、高效、简单的程序设计语言设施. 表达性主要体现在能对问题域直接建模, 同时具有易于描述分布式环境下并发与交互的语言设施; 并发性则是能支持应用的高并发和透明分布, 高效利用虚拟计算环境中的资源而程序员不必关心底层实现细节; 可伸缩性是指提供支持伸缩的计算模型、交互同步策略和透明分布能力, 使得程序员易于写出自动、高效伸缩的应用. 然而具有上述性质的编程模型本身存在着很多技术难题, 如并发元素不是语言的一阶设施, 实现跨机分布非常复杂, 对共享资源存取的不可控性, 程序正确性难以保证, 以及如何对底层资源

引用格式: 康亮环, 曹东刚. ConEntity: 一种基于进程的自主并发实体编程模型及其实现. 中国科学: 信息科学, 2013, 43: 126-142, doi: 10.1360/112012-496

进行高效而透明的接入等. 本文的主要动机是研究如何在虚拟计算环境之上提供支持开发高并发应用的编程模型和语言设施, 增强语言表达能力, 为程序员屏蔽高并发和可伸缩带来的风险, 简化编程难度, 提高编程效率.

本文的研究基于面向并发的编程模型, 其中并发元素是核心概念. 并发元素是作为基本编程单元提供给开发者的一阶语言设施. 如果一阶语言设施不是语言的并发元素, 如面向对象语言中的对象, 则会使得语言在实现并发时变得困难, 语言设施和动态执行单元难以对应. 以 Java 为例, 虽然 Java 语言在近年来通过不断扩展 Concurrent 包为开发者提供更抽象、更方便的并发编程接口, 但它最基本的并发仍采用线程方式. 线程是运行环境为程序设计语言提供的一种系统调用设施, 通常以开发库形式存在, 将底层操作系统的线程管理接口通过封装提供给程序员, 程序员将对象的执行与操作系统的线程绑定以实现并发, 而线程之间的交互和同步都需要程序员通过代码显式进行干预, 实现起来较为复杂. 线程之间共享数据, 建立在线程之上的编程模型通常采用共享内存的方式, 这种数据共享使得线程容易相互影响, 可能造成状态不一致等问题, 通常采用锁机制或事务性内存机制 (STM) 等方式解决, 在单机多核环境中易于实现, 但是在多机分布式环境下就变得非常复杂. 并发元素的另一种形式是进程, 基于进程的编程模型是分布内存的模型, 进程本身具有更好的封装性和独立性, 采用消息通信机制交换数据, 进程之间不共享状态, 因而不会有一致性带来的各种问题, 适用于面向互联网的多机环境. 本文基于进程的编程模型以进程代数为理论基础, 典型的有 CSP^[7], CCS^[8] 和 ACP^[9] 等, 它们以进程作为描述系统行为的核心, 基于进程代数的语言中进程是一阶语言设施. 在面向对象或过程式语言, 如 Java, C 中, 进程是通过语言提供的 API 机制显式进行创建和操作, 这些进程与底层操作系统直接对应, 依赖于操作系统的进程调度机制, 通常比较重量级, 创建和通信代价较大, 在抽象程度、并发能力上都受到一定的限制. 现有主流语言通过进程代数的相关模型扩展, 如在语言库中加入支持进程的设施, 以 CSP 为例, 就有 JCSP^[10], CTJ^[11] 和 C++CSP^[12], 或者加入进程代数模型作为语言设施后得到新的语言, 如 Groovy Parallel^[13], Honeysuckle^[14]. 这些对主流语言的扩展都有一定局限性, 编程受限与所依赖的语言, 而语言中的进程又依赖底层操作系统, 难以面向大规模并发, 因此需要独立于操作系统的轻量级进程来构建真正面向并发的编程模型. Occam^[15] 是一种基于 CSP 的并发编程语言, 其进程不同于操作系统进程, 是一阶语言设施, 用于描述抽象行为^[16]. 但 Occam 进程不具有分布能力, 主要是本地进程, 在网络环境下并发受到限制. Cloud Haskell 和 Erlang 都是函数式语言, 函数是进程执行的基本单元. 函数式语言具有单赋值和无全局变量的特性, 进程之间不会因为修改全局变量而互相影响, 适合大规模并发. Cloud Haskell^[17] 的进程作为 Haskell 的一种领域特定语言扩展, 用来构建活动, 它的进程机制是借鉴了 Erlang. Erlang^[18] 语言面向并发, 它以进程作为一阶语言设施, Erlang 进程具有轻量级、隔离性的特点, 能分布于互联网的各个节点, 进程之间通过消息进程通信, 传输效率高. 由此可得虚拟计算环境之上面向并发的模型具有 4 个重要性质: 1. 能创建大量进程; 2. 进程的创建和销毁操作高效; 3. 进程之间采用高效的通信机制; 4. 进程具有隔离性, 不共享数据和操作.

虽然以进程作为并发元素适用于并发编程, 但是相比面向对象、面向 agent 等编程模型, 它在描述复杂系统时仍存在着不足, 主要原因在于它的模型抽象程度不高, 缺少对现实世界中实体的描述机制. 现实世界中除了活动、动作, 一些实体也能并发, 我们将其称为自主并发实体. 本文在基于进程的编程模型中加入自主并发实体的建模机制 ConEntity, 这里的“自主”与 Autonomous Agents^[19] 中“自主”的概念存在差异. 后者的自主表现在对周围环境的感知以及自治性, 其自主性具有普遍含义, 而自主并发实体的自主性是特定应用场景下的自主, 主要表现在并发能力上. ConEntity 模型能根据外部请求和一定策略自动进行并发, 并根据虚拟计算环境提供的聚合资源为进程进行分布, 同时提供了一

定的同步策略, 在交互或状态出现不一致时自动决策处理. 因此 ConEntity 具有自主性且主要体现在并发行为上. 本文的主要贡献为:

- 提出一种支持对自主并发实体建模的 ConEntity 模型, 使得语言既能描述动态行为, 也能描述静态实体, 从而简化并发程序设计.
- 提出 ConEntity 的 3 种行为特征: 自主并发、交互与同步和可伸缩性, 并构建它的内部结构, 从而实现模型的表达性、并发性和伸缩性.
- 利用进程代数中的 CSP 对 ConEntity 模型进行形式化定义和描述.
- 扩展 Erlang 语言, 增加支持 ConEntity 模型的语言设施 UniAgent.

2 编程模型 ConEntity

在基于进程、面向并发的模型中, 进程通常描述动态的动作或活动, 缺乏对具有自主并发能力的实体的描述. 举一个直观的例子, 用进程能自然描述一个人的行为, 如他在处理一个任务, 但要描述这个人本身就不是那么方便, 因为一个人能自主并行执行各种任务, 各任务之间可能存在协作. 在一些进程代数的表述中, 也可以将一个进程直接取名为实体, 如名为自动售货机的进程, 但在描述该进程时仍然是用它所有动态行为的集合来表示. 本文在基于进程、面向并发的模型中引入一种实体建模机制 ConEntity (concurrent entity 的简写) 模型, 在语言层次直接构建自主并发实体.

ConEntity 具有两个主要特征:

首先在模型层次, ConEntity 在动态元素进程化的编程模型中加入了描述静态实体的语言设施, 实现了静态元素的实体化. 它是对进程代数中的进程模型的扩展, 实体化的静态元素本身也是由进程构成, 而并非重新创造的元素, 因此它也是一阶语言设施. 这种把动态行为和静态实体都作为一阶语言设施的方法, 自然地实现了语言设施到执行单元的映射, 在语言中的进程和实体都与虚拟环境中运行的轻量级进程直接对应, 因此以 ConEntity 为并发元素的语言也是面向并发的编程语言. 同时 ConEntity 模型在语言层次将许多分布式编程细节屏蔽, 并能自主产生大量工作进程执行并发行为, 这些工作进程之间通过用户定义或系统默认的策略进行交互, 从而简化应用开发.

其次是在行为层次, 通过为静态实体加入 3 种行为特征, 形成 ConEntity 的行为模型. 三个行为特征分别是主动并发、交互与同步以及可伸缩性. 这 3 种行为特征使得 ConEntity 具有描述自主并发实体以及实体之间各种交互机制的能力, 使得编程模型具有一定表达性、并发性和可伸缩性.

2.1 主动并发

类似于主动对象^[20], ConEntity 是具有自主行为能力的实体, 只是这种主动性主要表现在并发能力上. ConEntity 的并发能力包括内部并发和外部并发两个方面. 内部并发体现在, 每个 ConEntity 内部可以自动产生任意数量的进程来处理外部请求或进行并行计算, 把这种处理请求的进程称为工作进程 worker. ConEntity 具有一个称为主体 body 的核心进程, body 有一个消息队列用来接收外部请求并创建 worker 并发处理 body 执行的行为通过一个行为模块来描述, 这个模块是接收外部请求和实现实体行为逻辑的主体, ConEntity 的内部并发能力由一组 worker 来实现. 对于内部数据并行计算, ConEntity 能自动产生 worker 对数据分段后并行处理. 和其他一些语言提供的数据并行函数不同, worker 进程由 ConEntity 统一管理, 能根据 ConEntity 的内部并发策略实现透明分布, worker 的并发策略通过行为模块进行配置. 如用 Erlang 实现的 ConEntity 并发函数 `distributed_lists:map(Fun, List,`

Strategy), 与 clojure^[21] 中的 pmap 相比, ConEntity 能对产生的 worker 统一管理, 采用 Strategy 策略将其分布到底层环境中, 而 clojure 的 pmap 只能实现本地的单机多线程并发. 在内部并发策略上, ConEntity 实现了两种策略, 一种是 worker 的最大并发数量, 这是考虑到具体平台对进程数量的限制, 凡是超过这一阈值就不再产生新的进程, 而是顺序处理, 默认这一数值是无穷大; 另一种是 worker 的分布策略, 设置两种模式选择: local 和 global, 默认是 local 策略, 表示产生的 worker 都在本地, global 策略则是根据负载均衡策略选择全局节点执行. 外部并发和 actor 模型^[22] 类似, ConEntity 能通过创建新的 ConEntity 增加并发能力, 所不同的是 actor 只有外部并发而没有内部并发.

另外, ConEntity 可以在创建后将立即启动, 称为自发式, 也可以在创建后休眠, 每次接收请求或事件发生时才启动, 称为反应式. 在接收外部请求后, 其反应可以分为显式的和隐式的^[23]. 显式就是接收一切请求然后自动并行处理, 隐式就是它可以根据一定的规则和事先写好的条件接收或拒绝一部分请求, 这与面向对象语言如 POOL, Eiffel 类似. ConEntity 默认是自发的和显式的, 当需要隐式接收请求时, 通过设置黑名单和白名单来确定接收和拒绝的请求.

2.2 交互与同步

具有交互和同步能力的 ConEntity 与同步对象 (synchronized object) 非常类似, 可以分为两层: 消息通信层同步和 ConEntity 层同步. 在消息通信层, ConEntity 采用 sender/receiver 模型, 有两种可能的行为, 一种是发送者阻塞直到接收者完成对消息的处理, 称为同步通信, 另一种是发送者和接收者彼此独立, 称为异步通信. 为了将这两种通信方式统一起来, 在 ConEntity 之间建立一种抽象层次更高的通信模型, ConEntity 实现了 promise 机制^[24]. 它将 ConEntity 之间的调用返回值实现为一个 future, 并且能提供 continue 机制, 4.3 小节通过扩展 Erlang 语言对其进行了实现. 在 ConEntity 层, 根据不同的同步类型又可以分为 3 种: ConEntity 内部同步、行为同步以及 ConEntity 间同步. ConEntity 内部同步指的是 ConEntity 能同时处理多条请求消息, 并且能保证其状态的一致性, 如避免出现“读者写者问题”; 行为同步指的是 ConEntity 在同步时会等待直到某种条件得到满足, 从而避免出错; ConEntity 间同步指的是 ConEntity 同时与多个 ConEntity 交互时能实现彼此之间的同步.

为了实现上述同步机制, 类比同步对象的中心型同步模型, 本文得到 ConEntity 的同步模型: 以 body 为核心, 采用行为替代方法, 将 worker 作为 body 的行为替代来处理由 body 委派请求, 并引入了抽象状态.

主体 (body): ConEntity 将 “body” 作为实现主动行为的主体, 既有接收外部消息和调用请求的核心逻辑, 也能描述 ConEntity 的各种行为. ConEntity 的主体实现为 3 个部分: 一个接收者 (receiver)、一个行为模块 (behavior module) 和一个状态管理器 (state manager). 接收者是 body 的核心, 它负责接收外部请求并给 worker 派发任务, 每个接收者都包含一个 mailbox 用来缓存消息队列. Worker 在接收委派任务后的执行逻辑包含在行为模块中, 行为模块可以看成是由应用开发者实现的一个描述行为和配置策略的代码块.

行为替代 (behavior replacement): 这种方法借鉴了 actor 模型的行为替代, 由 worker 来实现的. 每个 worker 都作为 body 处理消息的行为替代者, 由 body 进行指派. 在遇到比较复杂的并发请求时, ConEntity 也可以通过创建新的 ConEntity 来处理, 是否创建新的 ConEntity 依赖于开发者在行为模块中的描述. ConEntity 模型与 actor 模型有类似之处, 它们都具有行为替代能力, 所不同的是 ConEntity 具有内部并发等更多的性质.

抽象状态 (abstract state): 用来描述具有状态的 ConEntity, 由状态管理器进行管理和维护. 在

worker 处理完请求后, 可能会修改原有抽象状态并计算出一个新的抽象状态, 这时状态转移和一致性保证就由状态管理器来完成, 本文在状态管理器中实现了一种处理状态一致性的策略, 在 4.2 小节中详细介绍.

相比一般基于进程的同步和交互模型, ConEntity 的同步机制的优势在于: ConEntity 在通常的 sender/receiver 模型基础上提出了 ConEntity 层交互和同步机制, 其内部同步过程被隐藏, 在状态不一致时采用一定策略自动处理, 并具有 non-block 和 lock-free 的特点, 具有很好的可伸缩性; 在行为层实现多请求同步策略, 使得同步模型具有更强的表达性和适用性; 通过封装进程间的远程消息传输, 将 ConEntity 之间的同步抽象为类似于对象之间的方法调用, 屏蔽了底层分布式环境的复杂性, 使得同步模型具有更好的表达性和易用性.

2.3 可伸缩性

ConEntity 由进程构建, 而进程本身有灵活的分布能力, 因此 ConEntity 构建的应用具备良好的可伸缩性. 本文从自主、透明分布, 内部并发和状态一致性机制这 3 个方面实现 ConEntity 模型的可伸缩性.

ConEntity 是一个能自主分布在多处理器或多机网络节点上的实体, 它的分布自主性需要虚拟计算环境资源管理模型的支持. 虚拟计算环境资源管理模型实现了计算和存储资源的聚合, 包括节点的启动、连接和配置, 同时虚拟计算环境提供了对节点的容错性管理. ConEntity 模型基于 worker 的 local 和 global 策略选取合适的计算节点, 将并发产生的大量轻量级进程分配到对应节点上, 虚拟计算环境资源管理模型则对产生的进程进行监控和容错处理. 这一策略的目标是减少网络通信代价, 提高 ConEntity 的执行效率和吞吐率. 可配置的分布策略也保证了 ConEntity 具有分布透明性, 它的启动方法中不含有节点信息, 程序员在编程时无需关心它在底层计算节点所处的位置. 分布自主性和透明性在系统吞吐率和加速比方面保证 ConEntity 模型具有一定的可伸缩性, 同时降低了程序员的编程复杂度. 第 5.2 小节的案例实验中通过一个典型应用的吞吐率和加速比来对 ConEntity 在 Erlang 语言上实现的可伸缩性进行测试.

在内部并发上, ConEntity 与虚拟计算环境中计算和存储资源的关系, 又有单程序多数据 (SPMD) 和多程序多数据 (MPMD) 两种模型来保证可伸缩性. 在单程序多数据的实现中, 大数据被划分并传送给分布在不同节点上的 worker 并行处理, worker 之间可以交互数据; 在多程序多数据的实现中, 每个 ConEntity 管理自己的数据并与其他 ConEntity 进行数据交互.

在状态一致性机制方面, 本文采用的“两阶段提交”保证状态的一致性, 具有 non-block 和 lock-free 的特点. 这是由于锁机制的阻塞等待特点, 往往会将并行处理转换成串行处理, 降低了程序的可伸缩性, 采用非阻塞的“两阶段提交”则不会有这一问题.

3 并发实体建模的形式化描述

在进程代数中, CSP (communicating sequential processes 顺序进程通信) 是一项非常重要且具有代表性的工作, 它是由 Hoare 在 1976 年最早提出. 本文采用 CSP 中的符号和 BNF (Backus-Naur 范式) 对 ConEntity 进行描述.

在文献 [7] 中, 一个 command 是指执行该命令的设备所表现出来的行为. 可以用 command 来表示进程或实体的动作或活动, 它的 BNF 如下:

```

<command>::=<simple command>|<structured command>
<structured command>::=<null command>|<non-parallel command>|<parallel command>
<parallel command>::=[<process>]{||<process>}}

```

其中我们使用 $\langle \text{non-parallel command} \rangle$ 来代替顺序命令, 因为本文中的模型对行为的描述不局限于顺序的. 对 CSP 中 parallel command 的 BNF 并对其进行扩展, 得到新的 BNF 如下所示:

```

<parallel command>::=[<process>|<conentity>]{||<process>|<conentity>}}
<conentity>::=<entity label>::<body>|<worker list>
<entity label>::=~ | ~<identifier>::|<identifier>(label subscript){, <label subscript>}}::
<label subscript>::=<integer constant>|<range>
<range>::=<bound variable>:<lower bound>..<upper bound>

```

其中在 parallel command 中新增加了一个元素 conentity 用来描述本文模型中的 ConEntity , 而 conentity 本身是由 process 组成的, 是特殊的 process , 这与 parallel command 在 CSP 中的定义一致. 根据我们在第 2.2 小节中的讨论, ConEntity 有两个主要部分: 一个 body 和若干 worker . Worker list 是一个 worker 组成的列表, 每个 ConEntity 可以创建任意数量的 worker . 每个 workers 对应 ConEntity 的一种行为, 它是 worker 的集合. 在上述 BNF 中, workers 如果是无限集, 则可省略 $\langle \text{range} \rangle$, 如定义为 $\text{publish_message}(1..\text{infinite})$ 的 workers , 可以简写为 publish_message .

```

<body>::=<receiver><state manager>
<receiver>::=<receiver label><receive process>
<receive process>::=<command list>*[<command list> <receive command><command list>]
<state manager>::=<state manager label><state manager process>
<state manager process>::=*[<receive command><commandlist>]
<receive command>::=<process name>??<message>

<process name>??<message> =><process mailbox>?<message>
<process name>!!<message> =><process mailbox>!<message>

```

如上所示, 一个 body 包含一个 receiver 和一个 state manager , 它们都是迭代执行的 process (可以是循环或者递归执行, 这里为了方便, 采用顺序进程中的循环来表示), 每次迭代都检查是否有输入消息. 这里定义了 receive command 来实现消息的接收, 其中引入双问号“??”表示异步接收, 区别于单问号“?”的同步接收. 为了实现异步机制, 引入了一个 process mailbox 作为消息缓冲区, 能接收任意数量的消息并排成队列. Receive process 每次迭代时都从 process mailbox 中取出特定的消息进行处理, 如果没有新消息或者消息不符合特定格式则马上执行后面的 command list , 而不会一直等待消息的输入, 具有异步性. $\text{State manager process}$ 迭代并读取与状态相关的消息, 后面的 command list 主要执行状态管理, 如状态修改.

```

~ chirper(i:1..N):: chirper_body |[add_follower | add_following | follow_user | update_list]
chirper_body::chirper_receiver | chirper_sm
chirper_receiver ::
[c1, c2:integer;
*[chirper_receiver ?? {Process, Args};
  Process == add_follower - >
      add_follower(c1) !! Args; c1 = c1 + 1
□ Process == follow_user - >
      follow_user(c2) !! Args; c2 = c2 + 1]]
chirper_sm::
*[chirper_sm ?? {List, Chirper}; uniagent:update(chirper, List, Chirper)]

add_follower(i) ::
add_follower(i) ?? Subscriber; update_list(i) !! {Subscriber, followerlist}.

add_following(i) ::
add_following(i) ?? Publisher; update_list(i) !! {Publisher, followinglist}.

follow_user(i) ::
follow_user(i) ?? Publisher;
Publisher !! {add_follower, chirper(i)}; chirper_receiver !! {add_following, Publisher}.

update_list(i) ::
update_list(i) ?? {Chirper, List}; chirper_sm !! {List, Chirper}.

```

上面是一个应用实例: chirper 系统, 这是一个类似于 Twitter 的消息订阅和发布系统. 将它里面的每个 chirper 用户定义为一个 conentity, 由一个 body (chirper_body) 和 4 个 workers (add_follower, add_following, follow_user, update_list) 构成. 以 follow_user 为例, 如果用户 A 关注用户 B, 则 A 执行 follow_user 行为, 参数为 B, 它首先调用 B 的 add_follower 行为, 然后调用自己的 add_following 行为. Chirper 的 ConEntity 各部分 process 的具体的描述如下. 在 chirper 系统的 CSP 描述中, 每个行为对应的方法都被定义为一个 process 的无限集合, 如 follow_user(*i*). 每个 receiver 处理两种类型的任务, 一种是异步输入消息, 通过 receive process 和 process mailbox 进行, 另一种是根据消息格式和用户定义的其他规则, 分派任务给相应的 worker 如 add_follower(*c1*)!Args. 状态管理器 chirper_sm 对 ConEntity 状态的转换以及一致性进行管理, 通过一个方法 uniagent:update/3 来简单描述. Chirper 系统详细讨论将在 5.1 小节中进行.

4 Erlang 实现

ConEntity 模型适合于采用基于进程、面向并发的编程语言实现. 本文选择在 Erlang 语言^[18]上

进行扩展, 有以下 3 个主要原因:

- Erlang 是一种以进程作为基本编程和执行单元的语言. 不同于 Java 和 C 等语言, Erlang 进程是一种自包含的虚拟机进程, 不依赖于底层操作系统的进程机制, 能从并发语言设施直接对应到运行时系统.

- Erlang 是一种支持高并发的语言. Erlang 进程轻量级, 能被高效创建和销毁, 在 Erlang 系统中允许同时大量的出现, 进程之间只能通过消息通信, 并且支持进程级的容错. 因此 Erlang 语言被称为面向并发的语言.

- Erlang 的高并发性和容错性使得它在工业界逐渐被广泛应用, 相比 Haskell 等纯函数式语言, Erlang 在构造分布应用上具有一定优势.

Erlang 进程作为基本语言设施存在着一定的限制. 因为 (1) 它缺乏对具有自主并发能力的实体的描述; (2) 不能透明分布于底层计算环境, 需要程序员指定网络节点; (3) 不具有自主并发能力, 必须由编程人员自己创建; (4) Erlang 进程无状态, 描述有状态的实体时受到一定限制. 因此本文对 Erlang 语言进行扩展, 实现一个基于 Erlang 语言的 ConEntity 模型.

4.1 语言设施 UniAgent

Erlang/OTP 是 Erlang 语言为开发者提供的开发环境, 帮助开发者更快地构建大型容错和分布式实时系统. 本文通过扩展 Erlang/OTP 来实现 ConEntity 模型, 在 Erlang 中的实现被称为 UniAgent. UniAgent 的名字与 A&A(agents and artifacts) 元模型^[25]相对应, “agent” 代表的就是具有自主行为的实体. 扩展 Erlang 得到的新模块 uniagent 实现了关于 UniAgent 的所有操作, 包括创建、启动、停止、UniAgent 之间的通信、状态转换等. 下面从 ConEntity 模型的 3 个特征对 UniAgent 进行实现.

- 实现 ConEntity 模型的自主并发. UniAgent 的 body 由 OTP 模块 gen_server 实现. Gen_server 是 Erlang/OTP 中实现的基于 client-server 模式的进程, 把它作为一个接收者, 行为模块作为它的回调模块, 该模块需要声明行为为 gen_uniagent, 并由程序员实现行为模块中定义的方法及各种策略. Gen_server 的状态信息维护在它的循环数据中. 每个 worker 都是一个 Erlang 进程, 由 body 自动创建.

- 实现 ConEntity 模型的交互与同步. UniAgent 的实现主要解决了两个问题: 状态一致性问题 and 实体间的通信问题, 将在后两节具体介绍.

- 实现 ConEntity 模型的可伸缩性. 实现需要有底层虚拟计算环境的支持, UniAS 是一种基于 Erlang/OTP 的面向企业内网络资源的虚拟计算环境. UniAgent 运行在 UniAS 上, 由 UniAS 为其提供计算和存储节点. UniAgent 通过向 UniAS 申请空闲节点来获得计算资源. UniAgent 实现了基于 body 和 worker 的自主和透明分布. 在 Erlang/OTP 中, 进程的创建方法是: spawn(Node, Module, Method, Args) 程序员需要指定进程所在的网络节点 Node, 而 UniAgent 的 start/1 方法中则不需要. 当 UniAgent 启动之后, UniAS 能根据事先定义好的分布策略和负载均衡策略, 自动在集群中选择一个合适的节点并启动 body, 然后通过一个新定义好的进程创建方法: uni_spawn(Module, Method, Args) 创建 worker. 则 UniAgent 为 worker 实现了一个本地性策略, 如果分布策略是 local 则尽量将 worker 与其所属的 body 创建在一个网络节点上, 如果是 global 则尽量分布在距离 body 近的节点上, 这样可以减少网络通信代价, 提高效率.

UniAgent 的结构如图 1 所示.

图 1 UniAgent 结构

Figure 1 Structure of UniAgent

4.2 状态管理

Erlang 是一种函数式语言, 具有无变量和单赋值的特点. 这使得 Erlang 程序无状态和无副作用, 适用于并发编程. 但是在描述有状态并且状态可变的实体时, Erlang 语言存在不足. 为了给实体加入属性, 简化程序员的编程复杂度, 我们在 uniagent 模块中提供了两种方法: set_property/3 和 get_property/2, 来给程序员保存和获得 UniAgent 的相关属性. 示例代码如下:

```
set_userName(Username) ->
    uniagent:set_property(THIS, username, Username).
get_userName() ->
    uniagent:get_property(THIS, username).
```

UniAgent 采用 gen_server 的循环数据保存状态. 虽然 Erlang 语言本身也提供了进程字典实现状态, 但是进程字典对应的是进程的状态, 程序员修改时不受控制, 也不能保证状态一致性, 因而 Erlang 不推荐使用, 而 UniAgent 的状态由后台进程管理, 则不会有这样的问题.

为了解决并发时的状态冲突, UniAgent 限制状态不能被其他 UniAgent 直接访问, 只能通过自己的 worker 访问. 利用函数式语言单赋值、无状态的特点, 可以将状态修改由语言层交给下层平台统一管理. 因此在 UniAgent 的状态不一致情况只可能通过 set_property/3 和 get_property/2 产生, 而且只能通过 UniAgent 所属 worker 调用, 只需要解决这些 worker 之间对状态修改的不一致即可. 传统的解决方法是给 UniAgent 的内部数据加锁, 或者给每个 worker 增加进程级事务. 这两种方法都存在一定的效率和伸缩性问题, 本文参考文献 [26], 提出了一种“两阶段状态”的方法来保证 UniAgent 的一致性. 此方法具有 non-block 和 lock-free 的特点, 能保证良好的效率和可伸缩性.

Worker 先在 UniAgent 的一个本地副本状态上进行修改. 本地副本具有两个阶段的状态: 临时状态和提交状态. 临时状态是 worker 在本地副本上修改后得到的状态, 它保存在 worker 所在节点的内

图 2 “两阶段状态法”解决状态一致性问题

Figure 2 Resolve the inconsistency in the state manager of UniAgent. (a) Commit condition is false; (b) merge function method; (c) first accept method

存中. 如果临时状态不符合应用中提交条件则被丢弃, worker 重新执行, 如图 2(a) 所示. 否则临时状态转换为提交状态. 在提交状态提交更新时有可能与其他 worker 的提交出现冲突, 采用两种方法解决, 如图 2(b) 和 (c) 所示. 图 2(b) 参考文献 [27] 采用合并函数法, 引入一个合并函数来将两个冲突状态合并为一个一致的状态, 此方法高效但是适用范围有限, 合并函数需要用户自己定义, 我们提供了一些简单的合并函数供开发者选择, 如累加函数、累积函数、字符串合并函数等; 若找不到合适的合并函数, 则采用图 2(c) 的先提交法, 先提交状态先更新, 如果后提交的状态与它产生冲突, 则被丢弃并重新执行, 此方法虽然效率较低, 但是能保证正确执行.

4.3 通信

Erlang 语言采用 sender/receiver 保证进程之间通信的独立性和隔离性. 在本文的 UniAgent 实现中, 进程之间以及 UniAgent 与进程之间仍然可以使用 Erlang 消息机制进行交互, 同时实现了 UniAgent 之间的互操作机制.

本文将 promise 通信机制^[24]引入 ConEntity 模型, 在 Erlang 上进行实现, 为 UniAgent 提供了同步与异步调用相结合的方式, 简化互操作, 提高抽象层次. 在 uniagent 模块中实现了两种基本操作: asyn.invoke/3 和 syn.invoke/3. syn.invoke/3 提供直接同步调用, asyn.invoke/3 是异步调用, 返回值称为 future, 有 3 种可能的状态: 未完成、成功完成和出错. 当 future 未完成而需要它的返回值或者需要等待返回值时, future 就会自动进行同步并返回结果, 因此 asyn.invoke/3 具有同步的能力; 当 future 成功完成时, 结果会自动保存起来; 当 future 失败时, 返回错误信息, 使得它本身具有容错性. 我们将 future 实现为一个无参数的 Erlang 函数, 函数内部实现了同步操作, 只需直接调用即可完成同步. 通过 chirper 系统的一个操作 follow_user/1 来说明, 代码如下, 其中 THIS 是 chirper 对应的 UniAgent, 它先异步调用 Publisher 的 add_follower/1 操作, 在执行完返回结果后马上调用自己的 add_following 操作. 最后设置一个同步点, 等待所有操作执行完成后再结束.

```

follow_user(Publisher) - >
  Invoker = uniagent:asyn_invoke(Publisher, add_follower, [THIS]),
  Fun = fun() - > add_following(Publisher) end,
  uniagent:continue_with(Invoker, [Fun]),
  Invoker().

```

Invoker 就是 future, 调用 Invoker() 后程序会等待直到返回计算结果. Promise 的另一个特点是能实现 continue 机制, 指派后续执行操作这里实现为一个 continue_with/2 方法, 它能指派一系列函数, 在 future 的结果返回时立即执行. 如 add_following/1 操作, 将在 Publisher 的 add_follower/1 操作执行完后立即执行, 并通过 get_continue_result/1 方法获取结果. 将 Erlang 函数作为 future 比其他语言如文献 [28] 中的实现更方便, 省去了 wait 等显示同步的函数.

4.4 UniAgent 的行为 gen_uniagent

UniAgent 的行为用 Erlang 中的 behavior 定义为 gen_uniagent. 根据 ConEntity 模型的行为特征将 UniAgent 行为分为两类, 一类是基本行为, 包括创建 (start)、停止 (stop) 和初始化 (init); 另一类是策略, 包括并发策略: 最大并发数 (concurrency_limit) 和隐式接收消息条件的黑名单 (black_list) 和白名单 (white_list), 分布策略: worker 分布的 local/global 策略 (stratigy), 同步策略: 状态冲突合并函数 (merge) 和行为同步策略 (join pattern).

```

-module(gen_uniagent).
-export([behaviour_info/1]).
behaviour_info(callbacks) - >
  [{start, 2}, {init, 1}, {stop, 0}, {concurrency_limit, 1}, {stratigy, 0}, {black_list, 0},
  {white_list, 0}, {merge_fun, 1}, {join_pattern, 1},
  behaviour_info(_Other) - > undefined.

```

5 案例研究

本文通过一个应用实例来展示 UniAgent 作为语言设施的表达能力, 并利用实验来测试 UniAgent 构建的应用的可伸缩性.

5.1 案例实现 chirper 系统

Chirper 系统是类似于 Twitter 的消息订阅和发布系统. 它属于通信密集型应用, 单个消息是大小受限的短消息, 因此计算量不大. 每个 chirper 用户可以在网络中发布自己的消息和订阅他人的消息. 任何关注了某个用户的用户都能及时收到该用户发布的最新消息, 任何用户发布的新消息也会被所有关注它的用户及时收到. 每一个 chirper 用户都实现为一个 UniAgent, 并且有一个管理用户 chirper_manager 用来创建 chirper, 它也是一个 UniAgent.

Chirper 行为模块的实现方法如下所示, 省略策略函数, 采用默认策略.

```

-module (chirper, [Chirper_id]).
-export([start/1, stop/1]).
-export([add_follower/1, remove_follower/1, add_following/1, remove_following/1, follow_user/1,
publish_message/1, get_published_message/1, [get_follower_list/1, get_following_list/1, get_newslist/0]).
-export([init/1]).
-behavior(gen_uniagent).

start(Args) - >...
init(Args) - >...
get_userId() - >...
add_follower(Subscriber) - >...
remove_follower(Follower) - >...
add_following(Publisher) - >...
remove_following() - >...
follow_user(Publisher) - >...
publish_message(Message) - >...
get_published_message(Publisher) - >...
get_follower_list() - >...
get_following_list() - >...
get_news_list() - >...

```

下面以一个重要的方法 `publish_message/1` 为例说明. 这是一个发布消息的方法, 实现代码如下.

```

publish_message(Message) - >
    Mymessagelist = get_mymessage_list(),
    NewMymessagelist = Mymessagelist ++ [Message],
    set_mymessage_list(NewMymessagelist),
    Num = get_messageNum() + 1,
    set_messageNum(Num),
    update_news(THIS, Num, Message, add),
    Followerlist = get_follower_list(),
    Fun = fun(Follower) - >
        uniagent:async_invoke(Follower, update_news, [THIS, Num, Message, add])
    end,
    ResultList = distributed_lists:map(Fun, Followerlist),
    uniagent:wait(ResultList).

```

在上述实现中, 先更新 UniAgent 的消息列表, 之后异步调用订阅者列表 Followerlist 中的每一个订阅者 (包括自己) 的 `update_news/4` 方法, 来更新它们的消息列表, 保证它们能收到该 UniAgent 的最新消息. 在方法的最后设置一个同步点 `uniagent:wait/1`, 保证所有的订阅者都成功更新消息后该方法才结束. 其中有一个特殊的模块 `distributed_lists`, 这是扩展了 Erlang/OTP 中的 `list` 模块, 利用函数式

图 3 UniAgent 本地和远程调用延时

Figure 3 Local and remote invocation latency

语言的结构性特征, distributed.lists 能自动将列表划分为若干部分并在分布式计算环境中并行处理, 最后形成新的列表. 从上述实现可以看出, 无需程序员自己创建进程, 应用就能最大程度的并发执行.

5.2 案例实验

在 UniAS 上对 chirper 系统的 UniAgent 实现进行可伸缩性测试. 测试用例为 chirper 系统创建 1000 个用户, 根据 Twitter 提供的统计数据, 平均每个用户被 27 个用户关注, 平均更新 20 次消息. 测试的物理环境是由一台服务器主机上的 10 个虚拟机组成的集群. 机器配置为: 24 Intel®Xeon®CPU X5650, 主频 2.67 GHz, 内存 24 GB RAM, 操作系统 64 bit Linux 2.6.32; 10 台虚拟机 KVM 每台双核 2.67 GHz, 内存 1 GB 或 2 GB RAM, 操作系统 Debian GNU/Linux wheezy.

5.2.1 微基准程序

图 3 显示了处在同一台服务器和不同服务器的两个 UniAgent, 它们之间一次调用的往返延迟时间. 该调用含有一个参数, 大小在 0.1 KB 到 50 KB 之间. 处在不同服务器的 UniAgent 远程调用平均时间为 4.4 ms, 同一服务器的本地调用平均时间为 1.6 ms. 本地调用时间主要对消息体的拷贝以及进程的调度上, 而远程调用还包括了对消息序列化、解析、响应, 以及 socket 读写、网络延时等时间, 消息越大, 时间花费越大. 另外对比 Erlang 消息通信, 采用 promise 需要额外消耗 3~5 ms, 但相比消息本身的延时可以忽略. 而创建一个新的 UniAgent 并进行全局注册和名字同步需要花费 5 ms 左右时间.

5.2.2 系统吞吐量

我们创建 1000 个用户并在最多 10 台节点上运行, 每个用户随机发布大小不超过 255 个字节的消息并更新其关注者. 采用用户数除以响应时间来估计系统吞吐量. 图 4 显示了实际和理想吞吐率的情况. 10 台服务器时 chirper 系统的吞吐率大约是 1900 chirper/s, Orlean 约在 2200 chirper/s, Twitter 吞吐率最高纪录是 35 台服务器 6839 tweets/s^[28].

图 4 chirper 系统运行 1000 个用户时的加速比

Figure 4 Chirper: speedup in 1000 users

图 5 chirper 系统运行 1000 个用户的吞吐率

Figure 5 Chirper: throughput in 2000 users

UniAgent 效率与 Orlean 的差距主要有两个原因: 一是 Orlean 系统在 Microsoft.Net 平台上用 C# 实现的, 其性能比 Erlang 平台高. 这可以通过对 C# 和 Erlang 平台进行多基准程序测试发现; 另一个原因是集群由虚拟机构成, 对其运行效率也产生影响.

5.2.3 加速比

在吞吐率测试的基础上, 将多机运行结果与单机运行结果相除得到加速比的估计值, 结果如图 5. 结果具有良好的线性增长趋势, 但是与理想线性加速比相比较, 仍然存在一定差距, 这是因为 chirper 系统是交互密集型应用, 增加运行节点则会增加消息交互和网络通信的频率, 造成效率上的损失.

将上述实验结果与文献 [28] 中 Orlean 的实验可以进行对比. UniAgent 实现了与 Orlean 具有相同功能接口的 chirper, Orlean 的代码量在 200 行左右, 而 UniAgent 的代码量为 150 行. UniAgent 与 Orlean 一样具有良好的线性加速比, 吞吐率比 Orlean 稍差, 但是两者在一个数量级. 因此, UniAgent 为程序员开发高并发应用提供了简单高效的编程模型, 用 UniAgent 直接对问题域中的自主并发实体进行构建, 不仅使得编程语言具有良好的表达性, 而且通过实验说明其能为应用提供一定的可伸缩性.

6 相关工作

ConEntity 模型主要是面向虚拟计算环境, 目前基于虚拟计算环境的编程语言主要有 Owlet^[29], 它主要用于构建基于角色交互的应用, 而非高并发应用. 除了虚拟计算环境, 当今互联网环境下的软件开发的另一个重要概念是网构软件. 网构软件是在开放、动态和多变环境下软件系统基本形态的一种抽象, 具有自主性、协同性、反应性、演化性和多态性的特点^[30]. 本文的 ConEntity 模型也是一种网构软件的特殊形态, 主要关注的是网构软件在互联网环境下的分布性和并发性. ConEntity 模型可以用来构建网构软件中具有并发能力的单个软件实体, 这些软件实体内部包含一些自主并发实体, ConEntity 模型为其提供内部并发能力、可伸缩性、交互与同步、通信和内部状态一致性.

Orlean^[28] 是基于 actor 模型进行扩展的编程模型. Actor 是一种经典的并发编程模型, Erlang 语言本身也可以看作是一种 actor 模型的实现. 因此 UniAgent 与 Orlean 在并发元素和基本编程单元上有相似之处. Orlean 增加了对事务和一致性的支持, 并且同样采用 promise 机制进行通信. 不同的是, Orleans 在语言层次使用称为 grain 的基本编程单元来对行为和实体进行描述. Grain 是单线程的,

通过创建若干 activation 来实现高并发, 而 UniAgent 是基于 ConEntity 模型, 兼具静态实体和动态行为的描述机制, 实体通过自身的固有并发能力实现自主并发. Grain 的并发可以认为是实体间的并发, 而 UniAgent 则是实体内部并发和实体间并发兼具. 一个 Grain 的 activation 是顺序处理外部请求, 而 UniAgent 能同时处理. 另外, Orleans 通过实现事务来处理状态不一致性, UniAgent 则实现了一种“两阶段状态法”来处理. Orleans 用命令式语言 (C#) 描述实体行为, 而 UniAgent 使用函数式语言描述.

Map/Reduce^[31] 和数据流 (dataflow)^[32] 框架都是现在流行的构建大规模分布式运算的模型. Map/Reduce 模型非常适用于在大规模数据集合之上的离线运算. 通过两个高度抽象的数据并行处理函数 Map 和 Reduce 为程序员屏蔽了底层复杂的数据并行处理逻辑, 上层编程模型非常简单. 但是 Map/Reduce 模型难以解决现实世界的很多复杂问题, 比如获得相关数据项集的交互式请求问题, 通信密集的信息发布订阅系统等. 本文的模型具有更广泛的表达性和应用性, 能适用于更多的应用类型. 但是在数据密集型的应用上, Map/Reduce 模型具有更好的表达性和效率, 如何在 ConEntity 模型和 UniAgent 实现中加入 Map/Reduce 模型, 是将来需要研究的工作.

MPI(message passing interface) 是一种高效的并行计算框架. MPI 的进程通常是直接对应底层操作系统, 它不局限于某种并发编程模型, 如 Map/Reduce 模型, 而是可以给开发者进行灵活的设计. 但是 MPI 的抽象层次低, 代码较复杂, 消息的传输以及同步机制都没有很好的抽象机制来封装, 导致 MPI 的编程复杂度较高. MPI 同样没有对并发实体的描述机制, 没有控制状态一致性的机制, 也没有异步通信的模型支持.

7 结论

虚拟计算环境是支撑互联网新型软件开发的重要机制, 为了在虚拟计算环境之上高效地进行高并发、可伸缩的互联网新型应用开发, 需要在程序设计模型和语言设施上提供支撑. 本文从这一需求出发, 在已有的基于进程、面向并发的编程模型和语言中引入并发实体建模机制, 目的是为开发者提供一套能同时支持动态行为和静态并发实体的并发编程模型, 以支持程序员在虚拟计算环境之上开发具有自主能力、高并发和可伸缩的应用. 进程天然具有描述并发的能力, 作为一阶语言设施能很好地支持高并发和可伸缩性, 已有的一些编程语言已经具有基于进程、面向并发的特点, 如 Erlang, Occam 等. 但其编程模型和语言在描述具有自主并发能力的实体时存在不足. 本文的主要工作是在已有基于进程、面向并发的编程模型和语言的基础上, 提出一个基于进程的自主并发实体模型 ConEntity, 以它作为编程基本单元, 降低开发高并发应用的复杂度. 我们从自主并发、相互与同步以及可伸缩 3 个行为特征来分析, 从而得到 ConEntity 的内部结构以及在虚拟计算环境中的特性, 然后通过进程的特殊模式对其进行构建, 利用进程代数 CSP 进行形式化定义和描述. 本文的另一个主要工作是用 Erlang 语言实现 ConEntity 模型, 称为 UniAgent. 主要方式是扩展 Erlang/OTP, 为上层 Erlang 语言设施提供一套库函数支撑, 并得到 UniAgent 的行为 `gen_uniagent`. UniAgent 具有透明分布的特点, 并能自主并发. 通信采用 promise 机制. 为了解决 UniAgent 的状态不一致问题, 采用“两阶段状态法”. 本文的模型存在两个方面的局限, 一方面需要虚拟计算环境提供资源管理模型支持, 以便底层计算和存储资源, 如果要持久化存储则需要接入分布式存储系统; 另一方面它的高并发与可伸缩性不是强保证, 进程与实体需要有好的设计模式支持. 这些都在将来的工作中需要研究和改进.

参考文献

- 1 吕建, 马晓星, 陶先平, 等. 网构软件的研究与进展. 中国科学 E 辑: 信息科学, 2006, 36: 1037–1080
- 2 Foster I, Kesselman C, Tuecke S. The anatomy of the grid: Enabling scalable virtual organizations. *Intl J High Perform Comput Appl*, 2001, 15: 200–222
- 3 Lu X C, Wang H M, Wang J. Internet-based virtual computing environment (iVCE): Concepts and architecture. *Sci China Ser F-Inf Sci*, 2006, 36: 1081–1098 [卢锡城, 王怀民, 王戟. 虚拟计算环境 iVCE: 概念与体系结构. 中国科学 E 辑: 信息科学, 2006, 36: 1081–1098]
- 4 Papazoglou M P. Service-oriented computing: concepts, characteristics and directions. In: *Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003)*, Rome, 2003. 3–12
- 5 Foster I. Service-oriented science. *Science*, 2005, 308: 814–817
- 6 Mei H, Huang G, Zhao H Y, et al. A software architecture centric engineering approach for Internetware. *Sci China Ser F-Inf Sci*, 2006, 36: 1100–1126 [梅宏, 黄罡, 赵海燕, 等. 一种以软件体系结构为中心的网构软件开发方法. 中国科学 E 辑: 信息科学, 2006, 36: 1100–1126]
- 7 Hoare C A R. Communicating sequential processes. *Commun ACM*, 1978, 21: 666–677
- 8 Milner R. A calculus of communicating systems. In: *Lecture Notes in Computer Science*. Berlin: Springer Verlag, 1980
- 9 Baeten J C M. A brief history of process algebra. Rapport CSR 04-02, Vakgroep Informatica, Technische Universiteit Eindhoven, 2004
- 10 Welch P, Martin J. Formal analysis of concurrent Java systems. *Commun Process Archit*, 2000, 2000: 275–301
- 11 Broenink J F, Bakkens A W P, Hilderink G H. Communicating threads for Java. In: Cook B M, ed. *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, 1999. 243
- 12 Brown N C C, Welch P H. An introduction to the Kent C++CSP library. In: Broenink J F, Hilderink G H, eds. *Communicating Process Architectures*, 2003. 139
- 13 Kerridge J M, Parallel G. A return to the spirit of occam? In: Broenink J, Roebbers H, Sunter J, et al., eds. *Communicating Process Architectures*, 2005. 1–13
- 14 East I R, Wood D. *Communicating Process Architectures*. Amsterdam: IOS Press, 2005
- 15 Roscoe A W, Hoare C A R. The laws of Occam programming. *Theor Comput Sci*, 1988, 60: 177–229
- 16 Hyde D C. Introduction to the programming language Occam. Course material for CSCI 366 Parallel Computation, Bucknell University, 1995
- 17 Epstein J, Black A, Peyton-Jones S. Towards Haskell in the cloud. In: *Proceedings of the Haskell Symposium*, Tokyo, 2011
- 18 Armstrong J. Programming Erlang software for a concurrent world. The Pragmatic Bookshelf Version. 2007-7-17, 2007
- 19 Franklin S, Graesser A. Is it an agent, or just a program? In: Müller J P, Wooldridge M, Jennings N R, eds. *Intelligent Agents III (LNAI Volume 1193)*. Berlin: Springer-Verlag, 1997. 21–36
- 20 Wegner P. Concept and Paradigms of Object-Oriented Programming. *ACM Comput Surv*, 1990, 22:
- 21 Rathore A. *Closure in Action*. Manning: Manning Publications, 2011
- 22 Agha G A. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence, 1986. 2–3
- 23 Compton M J. SCOOP: An investigation of concurrency in Eiffel. Master These. Canberra: The Australian National University, 2000
- 24 Shriram L B, Promises L. Linguistic support for efficient asynchronous procedure calls in distributed systems. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, Atlanta, 1988. 260–267
- 25 Omicini A, Ricci A, Viroli M. Artifacts in the A&A meta-model for multi-agent systems. *Auton Agents Multi-Agent Syst*, 2008, 17: 432–456
- 26 Kaushik S R, Shashank Y G. A programming model for collaborative distributed systems. In: *ACM SIGPLAN, Programming Language Design and Implementation*, Madrid, 2010
- 27 Burckhardt S. Concurrent programming with revisions and isolation types. In: *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Reno/Tahoe, 2010
- 28 Bykov S, Geller A, Kliot G, et al. Orleans: cloud computing for everyone. In: *Proceedings of the ACM Symposium on Cloud Computing (SOCC)*, Cascais, 2011
- 29 Wang J, Shen R, Wang H M. A programming language approach to Internet-based virtual computing environment. *J Comput Sci Technol*, 2011, 26: 600–615

- 30 Yang F Q, Mei H, lv J, et al. Some discussion on the development of software technology. *Acta Electron Sin*, 2002, 30: 1104–1115 [杨芙清, 梅宏, 吕建, 等. 浅论软件技术发展. *电子学报*, 2003, 26: 1104–1115]
- 31 Dean J Ghemawat S. MapReduce: a Flexible Data Processing Tool. *Commun ACM*, 2010, 53: 72–77
- 32 Isard M, Budiu M, Yu Y, et al. Dryad: Distributed data-parallel programs from sequential building blocks. In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* Lisbon, Nice, 2007. 59–72

ConEntity : an autonomous concurrent entity model to process-based programming and its implementation

KANG LiangHuan^{1,2} & CAO DongGang^{1,2*}

1 *School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China;*

2 *Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing 100871, China*

*E-mail: caodg@sei.pku.edu.cn

Abstract In the age of the Internet, lots of computing and storage resources are integrated by the Internet. Several new types of computation models are provided to deal with complexity of those distributed resources. Consequently new types of applications and new styles of programming models are required above them. To address this need, a new concurrent programming model with both process and concurrent entity is proposed in this paper, to develop new Internet applications with autonomy, high-concurrency and scalability. The autonomic concurrent entity called “ConEntity” is introduced to process-oriented programming for enhancing expressiveness of language, keeping consistency of shared states and implementing transparent access of resources. In addition, the formal definition and description are also given for this model using CSP (communicating sequential process). This paper describes an implementation of ConEntity model in Erlang as a language infrastructure named “UniAgent”. Several problems are solved including autonomic concurrency, automatic synchronization, scalability, communication and consistency. The goal of this paper is to provide programming model and language infrastructure for quickly and directly building distributed applications with autonomic concurrency. In the end of this paper, a typical application is implemented using “UniAgent” model in our distributed computing platform called “UniAS”, varying from a communication-intensive social networking application to a computation-intensive algebra application.

Keywords programming model, process, concurrent entity, concurrent programming, Erlang

KANG LiangHuan was born in 1986. He received his B.S. from School of Physics and a double degree from School of Electronics Engineering and Computer Science, Peking University, in 2005. Currently, he is a Ph.D. student at School of Electronics Engineering and Computer Science in Peking University. His research interests include distributed systems, concurrent programming structures and languages.

receiving Ph.D. degree. His research focuses on software engineering, distributed and concurrent systems, web engineering.

CAO DongGang was born in 1975. He received his Ph.D. degree from School of Electronics Engineering and Computer Science, Peking University. Currently, he is an Associate Professor of computer science at Institute of Software, School of Electronics Engineering and Computer Science, Peking University. He joined the faculty in 2005 after