软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(8):1713−1730 [doi: 10.3724/SP.J.1001.2013.04344]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种支持 Java 应用中计算按需远程执行的方法*

张 颖[1,2], 黄 罡[1,2], 刘譞哲[1,2], 梅 宏[1,2], 李 影[2,3], 杨顺祥[1,2,3]

[1](北京大学 信息科学技术学院 软件研究所,北京 100871)

[2](高可信软件技术教育部重点实验室(北京大学),北京 100871)

[3](IBM 中国研究院,北京 100193)

通讯作者: 黄罡, E-mail: hg@pku.edu.cn, http://www.sei.pku.edu.cn/~huanggang

摘 要: 按需远程执行是软件应用实现对资源按需占有,从而保障性能并提高资源利用率的重要手段.给出了一种通过自动程序转换来支持 Java 应用中计算按需远程执行的方法,其核心是支持计算按需远程执行的设计模式.介绍了将 Java 应用转换成该模式所面临的技术挑战、处理机制以及 DPartner 转换系统.与已有工作相比,DPartner 有两大特色:一是程序转换自动执行;二是转换后应用可实现真正按需的远程执行,使性能和资源利用率得以提升.此外,DPartner 被设计为可对只有 Java 字节码的遗产应用进行转换,更具实用性.

关键词: 按需占有资源;远程执行;程序转换;性能

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 张颖,黄罡,刘譞哲,梅宏,李影,杨顺祥.一种支持 Java 应用中计算按需远程执行的方法.软件学报,2013,24(8):1713−1730. http://www.jos.org.cn/1000-9825/4344.htm

英文引用格式: Zhang Y, Huang G, Liu XZ, Mei H, Li Y, Yang SX. Approach to supporting on-demand remote execution of the computations in a Java application. Ruan Jian Xue Bao/Journal of Software, 2013,24(8):1713−1730 (in Chinese). http://www.jos.org.cn/1000-9825/4344.htm

Approach to Supporting On-Demand Remote Execution of the Computations in a Java Application

ZHANG Ying[1,2], HUANG Gang[1,2], LIU Xuan-Zhe[1,2], MEI Hong[1,2], LI Ying[2,3], YANG Shun-Xiang[1,2,3]

[1](Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

[3](IBM China Research Laboratory, Beijing 100193, China)

Corresponding author: HUANG Gang, E-mail: hg@pku.edu.cn, http://www.sei.pku.edu.cn/~huanggang

Abstract: On-Demand remote executing is an important way to enable an application occupy resource on-demand to guarantee performance as well as improve resource utilization. This paper proposes an automatic program transformation approach for the on-demand remote execution of the computations in a Java application. The core of the approach is a design pattern supporting on-demand remote execution of computations. The research presents the technical challenges of and solutions for transformation, and gives out the DPartner transformation system. Comparing with previous work, DPartner has two major characteristics: first, transformation is carried out automatically; second, the transformed application is able to execute remotely on-demand, so its performance can be improved and the resource utilization can be increased. Additionally, DPartner is designed to be a practicable tool, as it can transform legacy applications with only Java bytecode.

Key words: resource on-demand; remote execution; program transformation; performance

* 基金项目: 国家自然科学基金(61222203, 60933003, 61121063); 国家重点基础研究发展计划(973)(2009CB320703); 国家高技术研究发展计划(863)(2013AA01A208, 2012AA010107); 新世纪优秀人才支持计划(NCET-09-0178); 中国博士后科学基金(2013M530011)

收稿时间: 2012-01-01; 定稿时间: 2012-10-10

对计算资源(如 CPU、内存等)实现按需使用,是软件应用增强性能并提高资源利用率的一种主要手段[1].所谓按需(on-demand),是指当计算资源不足时,应用可以占有并使用额外的资源,从而保障其高效运行;当资源过剩时,又可以释放掉多余的资源,从而减少浪费[2].按需使用计算资源必须同时具备两个条件[3]:一是有灵活可用的额外资源;二是应用可以真正使用到这些资源.在 Internet 环境下,虽然网络单节点的计算资源仍然有限,然而互连起来的网络节点其整体所拥有的计算资源则极为丰富.当应用需要额外资源时,可以通过网络获得其他节点所拥有的资源.因而可以看到,实现资源按需使用的根本仍在于应用自身.

应用对计算资源的使用方式主要有两种[4]:复制式和分割式.复制式的资源使用方式通过对应用进行复制,将副本运行在额外的节点上,并在原应用及其副本前端加上负载平衡器来将用户请求转发到这些应用实例组成的集群上[5].这种方式可使应用有效应对由于用户请求激增而性能急剧下降的情况.然而已有研究指出,这种复制整个应用的方式在较大程度上会造成资源浪费[6].其原因在于,通常情况下,组成应用的模块不全都亟需资源.那些不亟需资源但占有资源的模块会与那些真正亟需资源的模块相竞争,造成隐性浪费.因此,分割式资源使用方式将组成应用的模块分别部署在不同的节点上,让那些真正亟需资源的模块充分占有资源,以此实现对应用整体性能的保障以及对资源的充分利用[7,8].在此基础上,又可将两种方式相结合,在较细的粒度上实现复制式资源使用,以进一步高效地保障应用性能.

由此可见,应用对计算资源的按需使用要以应用模块的按需分布式部署执行为保障,也就是说,需要实现应用中计算的按需远程执行.远程执行主要有两种实现方式:(1) 编写分布式应用,将应用中对资源消耗较大的计算在设计和实现时就安排到拥有较多资源的网络节点上执行.然而,由于分布式应用要求开发者处理远程方法调用、序列化、同步等与程序分布相关的问题,因而编写起来通常费时、费力.此外,由于开发者很难在设计实现时就准确预料到应用对资源的使用,因而所开出的应用程序中的本地/远程调用方式通常固定不变,这就使得应用对资源难以实现真正的按需使用.(2) 通过程序转换器自动将给定应用中的程序代码转换后分为两部分:一部分继续留在原节点上运行,另一部分则被移动到其他网络节点上运行.这两部分之间所需要的跨网络的互操作代码由该转换器以打补丁的方式自动加入到原始应用当中,从而保证了转换后应用的正确运行.这种方式不要求开发者编写与程序分布相关的代码,极大地降低了应用开发难度,因而成为近年来极具潜力的一个研究方向[9].然而现有工作中,转换后应用代码中的本地/远程调用方式仍然固定不变,导致当资源变化而需要调整计算的本地/远程调用关系时必须停机、重转换、重启,因而难以实现应用在运行时对资源的按需使用.

本文以 Internet 上广泛存在的 Java 应用为对象,研究如何基于自动程序转换来实现其计算的按需远程执行.提出了一种支持计算按需远程执行的设计模式,并着重介绍了在将给定应用转换为符合该模式的应用过程中所面临的主要技术挑战,包括:(1) 正确性:自动识别出哪些应用类中的计算能够被转移到远程网络节点,从而保证转移后应用执行的正确性;(2) 有效性:决定哪些可转移的应用类中的计算需要被真正转移到远程网络节点上执行,从而降低网络通信的开销以保障远程执行后应用的整体性能并提高资源利用率.本文实现了名为 DPartner 的自动程序转换系统,将给定的 Java 应用转换为其计算可以按需远程执行的应用.DPartner 在 Java 字节码层次上实现转换,这使得它不但可用于新开发的应用,而且可用于遗产应用.DPartner 的输入为一个给定 Java 应用的字节码以及相关的资源文件(如图像文件、XML 配置文件等).输出为两类制品:一类为转换后的 Java 应用,该应用包含转换后的全部应用类和资源文件,且留在原网络节点上运行;另一类制品包含了从原应用中抽取出来的那些适合在远程网络节点上执行的应用类以及所使用的资源文件.这样,转换后应用中对这些类的调用可直接在本地执行,也可根据需要被转发到在远端的对应类上去执行.本文以 JEE 应用 RUBiS[10]为例对 DPartner 进行了实验,并证明了计算按需远程执行后应用性能的提升.本文主要贡献在于:

(1) 给出了一种支持 Java 应用中计算按需远程执行的设计模式.
(2) 实现了名为 DPartner 的自动程序转换系统,将给定的 Java 应用转换为计算可按需远程执行的应用.解决了转换过程中所面临的正确性、有效性等技术挑战.
(3) 通过实验验证了 DPartner 的有效性,证明了计算按需远程执行能够保障应用性能,并且能够提高资源利用率.

本文第 1 节给出支持 Java 应用中计算按需远程执行的设计模式.第 2 节给出一个对应的例子.第 3 节介绍 DPartner 如何通过字节码级的自动程序转换而将给定的 Java 应用转换为符合该模式从而可以按需远程执行的应用.第 4 节通过一组实验证明 DPartner 在实现计算按需远程执行以保障应用性能上的有效性.第 5 节介绍相关工作.第 6 节总结全文并展望未来的工作.

1 支持 Java 应用中计算按需远程执行的方法概述

1.1 一种支持Java应用中计算按需远程执行的设计模式

本文所给出的程序转换遵循重构的原则[11],即对给定应用程序的内部结构进行修改而不改变其外部功能.转换包含如下 3 个基本要素:

(1) 给定 Java 应用所具有的原始程序结构,称为源结构(source structure);

(2) 转换后 Java 应用所具有的新程序结构,称为目标结构(target structure);

(3) 一系列将源结构变为目标结构的转换操作.

我们首先介绍源结构以及目标结构,并在下一节中给出转换操作的概述.

一个给定的 Java 应用中的程序代码无外乎具有两种结构:一种是直接内存调用结构或称本地调用结构(local invocation),如图 1(a)所示;另一种是远程调用结构(remote invocation),如图 1(b)所示.

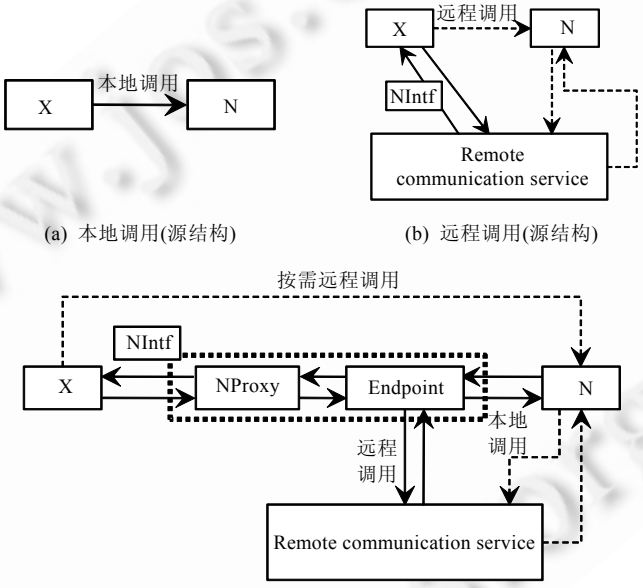


Fig.1 Source structure of a Java application and the target structure it held after transformation

图 1 Java 应用中程序的源结构以及转换后的目标结构

在图 1(a)中,应用类 X 调用应用类 N 的过程首先是获得对 N 的内存引用,然后通过该引用去调用所需要的方法.显然,这种直接内存调用结构并不允许 N 中的计算被按需转移到远程执行.倘若 N 转移到远程网络节点后,X 不能从它所在的地址空间获得对在远端的 N 的引用.在给定的应用中,还可能存在如图 1(b)所示的远程调用结构.X 通过 RCS(remote communication service,远程通信服务)获得运行在远端的 N 的引用,然后使用该引用去远程调用 N 中的方法.在该结构中,由 RCS 负责将 N 的引用与 N 相关联.以 JEE 为例,JEE 的 InitialContext 就相当于 RCS 的关键部分.X 通过 InitialContext 获得了 N 的 stub(即对 N 的远程引用),然后用其调用 N.尽管该结构允许 N 远程执行,却不能支持 N 实现按需远程执行.原因在于,当 N 和 X 在同一地址空间时,X 对 N 的方法调

用仍需要 RCS 发消息给网络栈才能传递到 N,而网络栈中的消息传递相对于直接内存引用来说极为耗时且耗资源,从而导致应用性能下降.这与通过计算按需转移来保障应用性能并提高资源利用率的初衷相违背.此外,在该结构中,X 与 RCS 相绑定,当 RCS 改变时(如从 TCP 变为 HTTP),X 也不得不改变,带来了额外负担.

为了实现计算的按需远程执行,转换后应用的目标结构必须允许 X 能够有效调用 N 中的方法而不管当前 X 与 N 运行在同一地址空间还是在不同的网络节点.我们给出了一种支持 Java 应用中计算按需远程执行的目标程序结构.它主要包含如下两个核心元素:proxy 和 endpoint.如图 1(c)所示,我们将 X 和 N 之间的直接内存调用以及通过 RCS 的远程调用都转换成了经由 proxy 和 endpoint 进行的间接调用.NProxy 的外部行为和 N 完全一致,只是它本身不执行任何实际的计算操作,只负责将方法调用转发到 N 执行.如果 N 的位置改变,比如从 X 所在的地址空间转移到了远程网络节点,或是从某一远程网络节点转移到了另一节点,X 并不会知道 N 位置的变化.Endpoint 负责识别 N 当前的位置并负责 X 和 N 之间的跨网络互操作.倘若 N 运行在远程节点,则 endpoint 会利用给定的 RCS 获得对 N 的远程引用,并把该引用以 NProxy 的形式供 X 使用,使得 X 能够通过该引用而远程调用 N.当 X 和 N 都运行在同一地址空间时,endpoint 会直接获得对 N 的内存引用,并同样以 NProxy 的形式供 X 所用,使得 X 调用 N 时不必经过耗时的网络栈.endpoint 带来了两个好处:首先,它帮助实现了 N 中计算的按需远程执行.当本地资源不足并且网络条件较好时,X 可以通过 endpoint 调用在远端的 N.从而通过 N 中计算的远程执行来实现对额外资源的占有以提高应用性能.当整体资源过剩或是网络条件不佳时,X 则可通过 endpoint 直接调用在同一地址空间中的 N,以保障应用性能并节约资源;其次,通过 endpoint,X 与具体的 RCS 相解耦.即便是 RCS 改变,X 和 NProxy 都不需要做任何变化,提高了应用对环境变化的适应性.

1.2 程序转换步骤

在确定了 Java 应用的源结构以及目标结构以后,我们逐步实施如图 2 所示的程序转换步骤来实现给定 Java 应用中计算的按需远程执行,并保障远程执行后应用整体性能的提升以及资源的节省.

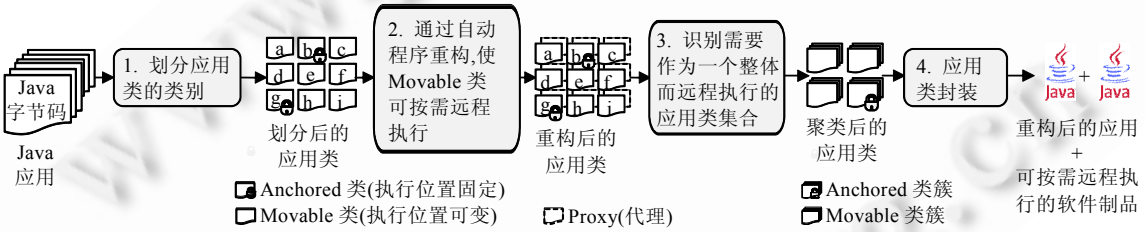


Fig.2 Transformation steps for the on-demand remote execution of a Java application

图 2 Java 应用中实现计算按需远程执行的主要转换步骤

(1) 应用类分类.对于一个给定的 Java 应用,DPartner 自动将其应用类,即字节码文件,分成 anchored 和 movable 两类. anchored 类型的应用类必须留在原网络节点上执行.原因在于,这些类使用到了一些只能在原节点才能获取的特殊资源(比如图形用户界面显示器或特殊的文件).倘若这些类被转移到远端网络节点上执行,它们会因找不到所需要的资源而造成执行错误.这些类之外的其他应用类都被自动归为 movable 类型的类.例如,一个用于数学处理的 Math 类就通常是一个 movable 类型的类.必要时,这些类可以远程执行,从而通过对远程计算资源的占有来提高整个应用的性能.

(2) 应用类转换.在这一步中,我们会将给定 Java 应用从源结构转换为目标结构,使其可按需远程执行.当一个应用类放到远端执行时,与之交互的应用类都需要被转换成可按需远程调用的结构,即生成被调用者的代理类 proxy,重写调用类来使用 proxy.需要注意的是,如果一个 anchored 类被在远端执行的 movable 类所调用,后者同样需要前者的代理类.由于只有根据运行时信息才能最终决定 movable 类的本地/远程执行以提高应用整体性能,因此我们必须对所有的被调用类生成相应的代理,并重写调用类来使用这些代理,除非调用类和被调用类都是 anchored 的情况.在本文所提出的方法中,一个 proxy 与其被代理的应用类在外部表现及功能上完全一致.

也就是说,这两个类继承了同样的父类,实现了同样的业务接口,具有同样的方法签名.这样,才能使那些原本使用被代理类的应用类在使用proxy时不能感觉到差别.在转换过程中,还需要处理Java语言的特性,比如静态类、内部类、数组等.对转换的进一步说明将在第3节中展开.

(3) 应用类聚类.为了使计算远程执行能够真正提高应用的性能,我们必须避免频繁网络调用所带来的负面影响.因为网络调用通常极为耗时,若过于频繁则反而会影响应用性能的提高.正因如此,我们需要把相互间调用频繁的类作为一个整体在必要时放到远端执行.倘若在应用运行才去发现应用类间的调用关系,则通常会带来巨大的开销.因此,本步骤的目的就是要简化对应用中哪些计算需要被真正远程执行的运行时决策.我们基于应用类聚类的方法来实现这种简化.例如,通过聚类我们发现类X和N调用频繁.在运行时,我们可以只监控X的执行轨迹来决定是否需要将X和N一起放到远端执行.这样做不但可以避免X和N被分到网络两端时所需要进行的频繁的跨网络互操作,而且可以简化运行时X和N之间关系的分析以及对N的监控,从而降低开销.

(4) 应用类封装.DPartner的输入是一个给定的Java应用(如EJB应用)的组成字节码文件以及相应的资源文件,如图像文件、XML配置文件等.在经历了以上步骤之后,DPartner的输出包含两类制品:一个是转换后封装为原始应用格式的应用,比如EJB应用的ear/jar/war.该应用留在原网络节点运行;另一类是原始应用中那些转换后的movable类,代理类等所组成的一个可在远程网络节点部署执行的软件制品(集合).该制品的封装格式与原始输入应用的格式相关.比如,对于EJB应用,这第2类制品是多个EJB Bean的jar文件.

2 实 例

图3展示了一个可将计算按需远程执行的Java应用的运行时系统结构.该应用包含了6个类,名字分别为a~i.通过步骤1,DPartner发现类b和g都是anchored类,必须留在原网络节点上执行.其他类都是movable类,它们既可以在原网络节点上执行,又可以在新增的远程网络节点上执行.通过步骤2,每一个应用类都生成了对应的proxy类,并且应用类间的调用都通过proxy和endpoint来完成.步骤3中,DPartner发现a,c,d,e和f是关系紧密的类,因此它们被聚类在一起,以备在必要时作为一个整体运行在远端.步骤4中,DPartner将所有的movable类型的应用类、代理以及endpoint封装为可在远端节点部署执行的软件制品,如EJB jar文件(集合);该制品允许有多个副本运行在不同的网络节点上,以支持本方法所提出的通过计算远程执行来实现的对资源的占有.同样,DPartner会将全部应用类、它们的代理以及endpoint都按照原有应用格式进行封装,比如封装为EJB的jar文件和Servlet的war文件.转换后的应用仍放在原网络节点上进行,不同之处在于,转换后应用中,movable类中的计算可以根据需要而调用在远端的对应movable类执行或是直接在本地执行.

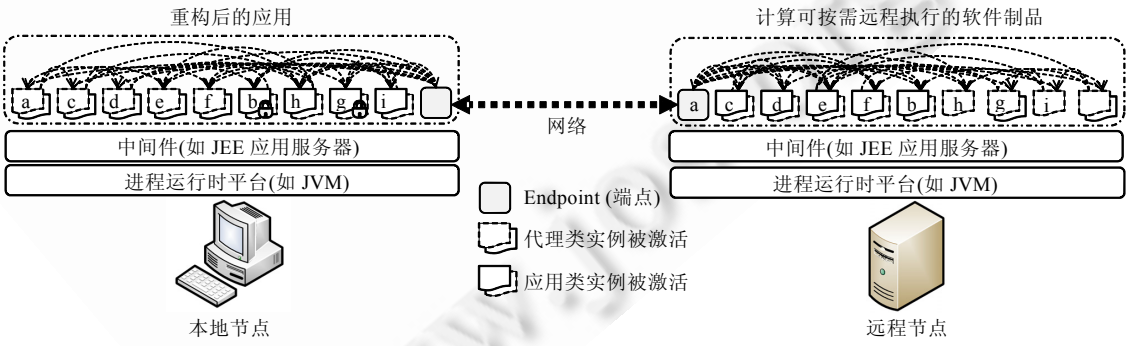


Fig.3 Runtime architecture of a Java application that can carry out computation remotely on-demand

图3 可将计算按需远程执行的Java应用的运行时系统结构

在运行时,endpoint若预测到应用类d远程执行时可以提高应用性能,则会首先激活在远端的d,并且把本地d的状态通过endpoint注入到远端的d,实现状态同步,接着将本地的d停用.此后,d的proxy将会通过endpoint把对原节点d的调用请求转发到远端的d上执行.由于d,c,e和f在第3步时已经被聚类为一个逻辑整体,因此

当 d 被转移到远端执行时,该聚类中的其他类也跟着被移到远端执行,以减少频繁的跨网络调用.该过程也一样要经历激活、状态同步、停用等阶段,因此我们可以从图中看到,在原节点,a,c,d,e 和 f 的 proxy 在起作用,而这些类中计算的真正执行是在远端的网络节点上.当网络条件不佳或是发现仅原节点的资源就足以保障应用性能时,可以通过停用在远端的a,c,d,e和f并重新激活在原节点的对应类实例来将远程执行的计算移回,以节约资源并保障应用正常运行.所有对这些类的调用都将通过 proxy 和 endpoint 转发回原节点,从而以一种按需的方式实现了应用中计算的远程执行.

3 计算按需远程执行的具体实现

3.1 应用类类别划分

DPartner 采用自动程序静态分析来识别一个给定的 Java 应用类的类别.它将符合如下两个条件之一的应用类自动归为 anchored:(1) 类的方法含有 native 关键字.方法含有 native 关键字表明该类需要调用本地动态链接库才能执行,因此其计算不能被转移到远程执行.(2) 该类调用 anchored 系统类.例如,对于本文的实验对象 EJB 应用来说,我们将用于 Web 处理的类如“javax.servlet.Servlet”认为是 anchored 系统类.因为我们认为,该 Web 类必须留在原网络节点运行才能正确接收用户请求.因此,任何继承该类的应用类都是 anchored 类.若一个应用类不满足以上两个特征,则该类被归为 movable 类.DPartner 提供了一个配置文件来描述需要被 anchored 的应用类的名字特征.开发者可以使用该配置文件来确定某 Java 应用中哪些类必须被留在原节点执行或是放到远端执行,从而使分类步骤适应于每一个特殊的 Java 应用,以进一步保证分类结果的正确性.

例如,“cn.edu.pku.password”是一个 movable 类.然而,倘若开发者基于私密性的考虑而不希望将该类被放到远端执行,则可以将其写入配置文件中,DPartner 会自动将该类归为 anchored 类.

3.2 代理生成

生成 proxy 所面临的最大挑战就是使得 proxy 和被代理的对象具有完全相同的外部表现.例如,在应用类 X 的实现体中,它在调用另一个应用类 N 中的方法时将 N 造型成了 N 的父类 NP.倘若我们只在 X 中把对 N 的调用换成对 NProxy 的调用,而没有让 NProxy 继承 NP,则转换后的造型操作会出错.因此,DPartner 所生成的 proxy 必须与被代理的类具有相同的程序结构.特别地,这些 proxy 之间也必须与被代理的类一样维持同样的继承层次结构.例如,如果 N 继承了 NP,那么 NProxy 必须继承 NPProxy.这使得任何对 N 从 NP 所继承的实例方法或是构造函数的调用都能从 NProxy 转发到 NPProxy,并最终转发到 NP.

在 Java 语言中,接口被用来分离一个类的外部表现和内部功能实现.从接口的角度来说,如果一个类及其 proxy 都实现了相同的接口,则这两个类是完全一样的.因此,DPartner 会自动抽取接口来表示一个类及其 proxy.例如,对于应用类 N 来说,DPartner 会:1) 抽取 N 的全部实例方法签名来组成一个 NIntf 类;2) 让 NIntf“继承”NPIntf 接口,以方便造型操作.NPIntf 是对应 N 的父类 NP 的接口;3) 让 N 实现 NIntf 接口;4) 让 NProxy 也实现 NIntf 接口;5) 让所有调用 N 的应用类都改为调用以 NIntf 表示的 NProxy.然而对于 N 中的静态方法来说,由于 Java 语言中不允许静态方法出现在接口中,因此,DPartner 会直接使用 NProxy 的静态方法来进行方法调用转发.

3.3 应用类转换

应用类需要经过转换才能满足按需远程执行的要求.下面列举 DPartner 提供的主要转换器加以说明.

3.3.1 字段-方法转换器

一个应用类的非私有字段(如以 public,protected 以及空修饰符所标识的字段)可以被其他类所直接使用.然而,若该应用类被转移到远程网络节点执行,则原本使用这些字段的类就不能正常获取它们.为了解决这个问题,本转换器自动地为非私有字段生成对应的 getter/setter 方法,然后将这些字段都变成私有字段,最后将对原非私有字段的直接使用转换为通过 getter/setter 方法进行的调用.此外,本转换器还为一个应用类中的私有字段自动地生成对应的 getter/setter 方法.这样可以通过 getter/setter 方法来获取并设置一个应用类的内部状态,以便于

应用中计算按需转移时所需要进行的对象状态同步工作.

3.3.2 服务对象转换器

为了使一个应用类与其proxy相关联,DPartner提供了该转换器以便确认一个proxy究竟应该将方法转发到哪一个具体的应用类实例上.其功能主要是让每一个应用类实现一个特殊的ServerObject接口中的如下两种方法:getID和getProxyForSerializing.其中,getID会返回一个整型的数值,用于表明该应用类实例的身份.一个Proxy对象通过该ID和一个应用类实例相关联,这样能够保证方法调用转发到正确的应用类实例上.此外,通过ID的关联可以实现分布式垃圾回收,也就是说,当发现一个proxy被JVM回收后,可以通知其所关联的应用类实例.该应用类实例在没有其他proxy与之相关联的情况下也可以实现回收.getProxyForSerializing用于处理回调过程中应用类实例的序列化问题,详见第3.5节.

3.3.3 其他转换器

除了以上关键转换器之外,DPartner还实现了若干转换器来处理Java应用中一些特殊的程序结构.这些转换器包括:Array Transformer,用来使对数组的存取操作在网络同步,以保证网络两端对应数组状态的一致性;Add Anonymous Constructor Transformer,用来为一个应用类增加匿名构造函数,以方便其按需远程执行时的激活和状态同步.其他转换器如Remove Abstract Transformer,Remove Final Transformer,Inner Class Transformer,Interface Generator,详见DPartner项目网站以及文献[8],这里不再赘述.

3.4 应用类聚类

如前所述,若随机选择某一应用类来远程执行,虽然占有了额外的资源,但并不能保证应用整体性能的提升.原因在于,应用中留在原网络节点的部分和远程执行的部分需要进行跨网络的交互.若交互过于频繁,则大量的计算时间消耗在网络通信而非业务处理上,因此会导致应用性能低下.应用类聚类的目的就是把那些相互间调用频繁的movable类作为一个逻辑单元(称为cluster),并将它们一起移动到远程网络节点执行,通过避免它们之间高额的网络调用来保障计算远程执行的有效性.DPartner对应用类的聚类是通过程序分析建立调用关系图来实现的,并且为了弥补静态分析的不足,DPartner还基于已有研究中已证明的关于关系紧密的类之间具有语义相似性的结论[12],利用类中的变量以及方法名的文本相似度信息来校准前面建立的调用关系图,从而把那些关系真正紧密的类聚在一起.我们为DPartner设计了一种应用类聚类算法来完成聚类任务.本文中不再详细讨论算法实现和原理证明,可以参见文献[8].此外,聚类信息还可以通过DPartner调用程序动态分析软件进行分析后得到,如执行相关测试用例.DPartner会把聚类信息存储在转换后的应用制品中.在运行时,endpoint将会使用该信息来辅助决定哪些类对应的计算应该被转移到远端执行,详见第3.6节.

3.5 Endpoint的实现

如前所述,DPartner会为转换后的应用提供一个称为endpoint的通信基础设施,用于负责应用中计算远程执行时,留在原节点上执行的部分与在新节点上执行的部分间跨网络的互操作.从应用类X到远端的应用类N的方法调用会首先由表示为NIntf的NProxy转发到endpoint,并最终由endpoint转发到远端的N,如图4所示.

在NProxy的一个方法体中,endpoint的“invokeMethod”方法被用来实现方法调用转发.InvokeMethod的参数如下:(1) 应用类N的全名.(2) 应用类N的实例ID.每一个proxy都拥有它所对应的被代理应用类的实例ID.当proxy被创建时会被赋予该ID来与被代理类实例相关联.(3) 字节码级别的方法签名.在Java字节码中,所有方法签名都表示为如图4所示的“Internal name”.(4) N的“methodM”方法执行时所需要的参数.Endpoint可以通过以上信息来唯一定位一个N对象,并调用其methodM执行,最后返回结果.

当方法调用基于RCS(见第1.1节)发送到网络栈来进行时,方法的参数需要被序列化才能在网络上传输.我们要特别注意回调情况下调用对象的序列化和反序列化问题.例如,X运行在原网络节点,N运行在新分配的远端网络节点.X调用了N的“handleCaller(XIntf)”方法(实际上调用的是以NIntf表示的NProxy对象的handleCaller方法).该方法中,X将自己作为参数传递到N,而后N通过传递过来的X引用来回调X的方法.可以看到,X需要被序列化才能进行网络传递.值得注意的是,当X在N所在的地址空间被反序列化后,整个应用会同

时存在两个 ID 相同的 X 实例,一个在原网络节点,另一个在 N 所在节点.这时,如果 N 对其所在端的 X 进行了处理,则对应操作不能被转发到原节点的 X,从而导致两个 X 之间内部状态不一致,产生运行时错误.此外,倘若 X 是一个 anchored 类型的应用类,它本身就不能被传到远端.因此,为了解决上述问题,当 X 需要被序列化时,应该序列化它的 proxy.如图 5 所示,endpoint 会自动调用 X 实现的 ServerObject 接口中的 getProxyForSerializing 方法来获得一个 XProxy(见第 3.3.2 节),并传给远端的 N.这样,N 可以用该 XProxy 来完成对 X 的远程回调.

```
public class NProxy extends NPProxy
        implements NIntf {
    public SIntf methodM(SIntf s,TIntf t){
        return (SIntf)
            ProxyFacade.getEndpoint().invokeMethod(
                "foo.bar.N",    //remote class
                this.serverobjID,    //class instance ID
                //bytecode-level method signature
                "methodM(Lfoo/bar/intf/SIntf;    Lfoo/bar/intf/TIntf;)➥
    Lfoo/bar/intf/SIntf;",
                new Object[] {s,t}    //parameters
            );
        }    //end methodM
}    //end NProxy
```

Fig.4 Method forwarding chain from proxy to endpoint

图 4 从 proxy 到 endpoint 的方法调用转发链

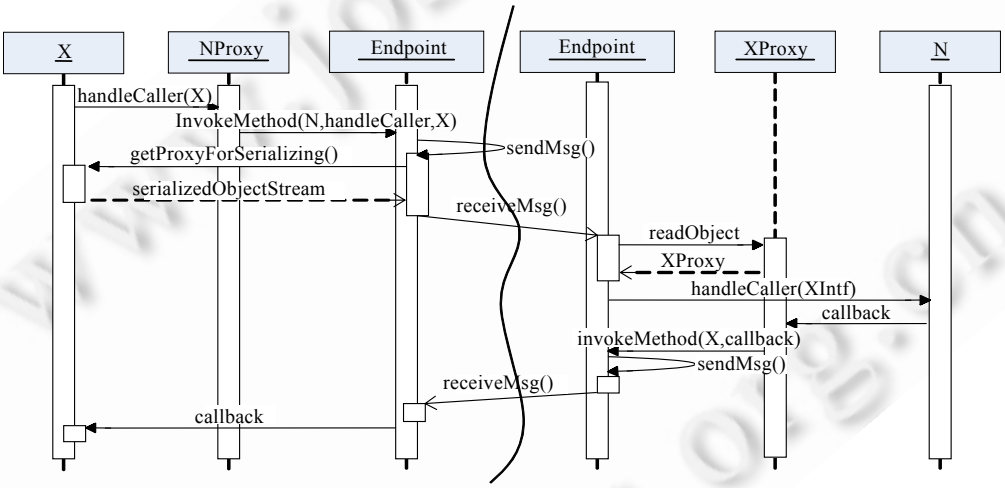


Fig.5 Caller de/serialization in callback-style remote method invocation

图 5 方法调用者在回调风格的远程调用中被序列化/反序列化

通过 endpoint,两个应用类之间的互操作可以较为容易地实现优化.例如,当 X 和 N 都在同一个 JVM 中时,endpoint 会直接使用内存引用来进行方法调用转发.这样,方法调用时就不需要经过耗时的网络栈,从而可以保障应用的性能.此外,将对网络通信的处理集中在 endpoint 而不是分散在各个 proxy 也带来了如下 3 个好处:

1) 使得 proxy 尽可能地小,以方便在网络上的传输.
2) 帮助实现调用者和被调用者的解耦.若用于识别本地/远程通信的代码存在于 proxy 中,则 X 对 NProxy 的直接单向引用将会变成直接双向引用.也就是说,NProxy 必须知道 X 的具体位置才能实现 N 和 X 之间的互操作优化.
3) 其他与通信相关的代码,如远程调用重试、移除分布式死锁[13]等,能够容易地加入 endpoint 中,从而以方法调用的截取器链的形式来保障远程通信的质量.

3.6 计算按需远程执行的决策

如前所述,endpoint 会在运行时决定应用中的计算是否需要远程执行.从整个应用中计算执行的分布角度来看,此项决策的实质就是决定应用中哪部分计算需要执行在哪个节点才能保障应用性能并提高资源利用率.因此,在不考虑应用自身优化的情况下,该项决策转换为求解应用类放在哪个节点才能充分且足够地享有所需的资源以保障执行效率,并且尽量减少与其他应用类之间由于进行耗时的跨网络调用而带来的执行效率损失.我们采用更形式化的方法来描述以上问题.在以下的描述中,N 代表自然数,R 代表实数.

3.6.1 应用结构

我们设应用 $C=\{c_1,c_2,\ldots,c_n\}$,表示应用由 n 个类组成.需要注意的是,这里的 c_k 可以表示为 $c_k=\{c_{k1},c_{k2},\ldots,c_{km}\}$ 这 m 个类,其属性为对应各个类的属性和.这样做的目的是利用第 3.4 节已经计算出的类聚类信息将多个关系紧密的类看成是一个类参与计算,以减小问题域的规模,加快求解过程.应用结构的相关函数参数如下:

- 交互频率(frequency of interaction,简称 foi): $foi:C\times C\to R$;
- 传输数据量大小(transmission data size,简称 tds): $tds:C\times C\to R$;
- 工作负载(workload,简称 wl): $wl:C\to N$;
- 内存消耗(memory consumption,简称 mc): $mc:C\to N$.

3.6.2 节点配置

给定节点集合 $VM=\{vm_1,vm_2,\ldots,vm_k\}$,表示有 k 个节点(虚拟机 VM),相关函数参数如下:

- 处理机能力(processing speed,简称 ps): $ps:VM\to N$;
- 网络带宽(network bandwidth,简称 nb): $nb:VM\times VM\to N$;
- 网络延迟(network delay,简称 nd): $nd:VM\times VM\to N$;
- 内存拥有量(memory possessed,简称 mp): $mp:VM\to N$.

3.6.3 部署结构

应用中计算的按需远程执行实质上就是将应用类部署在网络节点上被并调用执行的过程.可以用下式来表示:$D=\{d|d:C\to VM\}$,其中,d 是将应用类 C 部署在节点 VM 上的一种拓扑结构.

$D^{-1}=\{c\in C|d(c)=vm\}$ 表示确定了部署结构以后,可明确知道一个节点上执行的应用类.

3.6.4 约束条件

合理的部署结构需要满足一定的约束条件,针对本文的工作,我们给出如下两个约束条件:

(a) 资源约束,这里特定为内存资源约束:$\theta_{mem}:D\to\{true,false\}$,即

$$\theta_{mem}(d)=\forall vm\in VM:\sum_{c\in D^{-1}(vm)}mc(c)\leqslant mp(vm).$$

(b) 位置约束:$\varphi_{loc}:D\to\{true,false\}$,即

$$\varphi_{loc}(d)=\forall c_1,c_2\in C:(colloc(c_1,c_2)=1)\to d(c_1)=d(c_2);(colloc(c_1,c_2)\neq 1)\to d(c_1)\neq d(c_2),$$

其中,$colloc(c_1,c_2)=1$ 表示应用类 c_1 和 c_2 必须位于相同的位置;否则,必须位于不同的位置.我们在第 3.1 节介绍的应用类分类信息可以作为 colloc 函数取值的参考输入.

3.6.5 目 标

如前所述,计算按需远程执行的目标是要在充分利用资源的情况下,尽量提高应用的性能.也就是说,要使 $Q:D\to R$ 尽可能地大,即求解使应用质量属性 Q 尽可能大的一种应用部署结构.本文所关注的 Q 为应用的性能与资源消耗的比值:$Q=P/Res$.其中,性能 P 可被看作应用类的工作负载被处理的效率以及网络开销组成的一个函数值,表示如下:

$$P=\alpha\times P_{workload}+\beta\times P_{network},\alpha,\beta\in(0,1),\alpha+\beta=1 \tag{1}$$

$$P_{workload}=\frac{1}{\sum_{j=1}^{k}\sum_{c\in D^{-1}(vm_j)}\frac{wl(c)}{ps(vm_j)}} \tag{2}$$

$$P_{network}=\frac{1}{\sum_{i=1}^{n}\sum_{j=1}^{n}foi(c_i,c_j)\times nd(d(c_i),d(c_j))+\sum_{i=1}^{n}\sum_{j=1}^{n}\frac{foi(c_i,c_j)\times tds(c_i,c_j)}{nb(d(c_i),d(c_j))}} \tag{3}$$

而资源则表示为

$$Res=\sum_{j=1}^{k}ps(vm_j)\times mp(vm_j) \tag{4}$$

因此,最终的目标是要让公式(1)与公式(4)的比值最大化.可以看到:公式(2)本质上是一个背包问题[14],在复杂性上是 NP 难的;公式(3)本质上是一个图分割求最小割集问题[15],当节点数目大于 3 时,也是一个 NP 难问题.然而,这两个问题目前已有很多优化的近似解法,endpoint 缺省采用其中经典且较快的解法——动态规划以及 MinCut[15]来进行求解.考虑到还可能存在如贪心法等其他多种近似求解方法,因此,endpoint 提供了一个 $Q=P/Res$ 值求解扩展机制,可以将运行时得到应用和节点的相关参数信息作为其他方法的输入,取得返回结果 $D=\{d|d:C\rightarrow VM\}$,以决定最终应用中计算远程执行对应的部署方案,从而完成决策过程.Endpoint 也允许由管理员给出的高层命令(如通过 JMX[16]命令)来实施计算转移,以提供灵活性并增强应用对环境的适应能力.

4 实 验

在本节,我们要验证计算可按需远程执行的应用在性能上的提升和对资源的有效利用.实验采用了单机应用和分布式应用.文献[8]中已经详细介绍了 DPartner 如何将单机应用通过自动字节码重构而转换为可按需远程执行的应用.接下来,本文主要介绍 DPartner 在分布式应用 RUBiS[17]上的实验.

RUBiS 是美国 Rice University 所研发的 EJB 基准测试集.它模拟了类似于 eBay[18]的商品买卖竞标系统.该应用包含 17 个 ejb 以及对应的若干 servlet 等 Web 界面显示类.它的测试驱动器模拟用户浏览 Web 界面的操作,从而使 ejb 中计算得以执行.该测试驱动器所产生的工作负载主要由如下 3 个参数来控制:request transition table,client number 和 session runtime.RUBiS 的 read-write 类型的 request transition table 表示模拟出的客户访问请求除了进行读操作,如浏览商品项目、浏览别人对该商品的评价等以外,还要进行写操作,如添加商品信息、对商品评价等,从而会改变数据库中的内容.Client number 表示同时有多少用户来访问 RUBiS 应用.访问量越大,系统的负载也就越大.Session runtime 表示一个用户访问 RUBiS 应用的时长.时间越长,该应用在该时间段内所做的操作就越多,因此系统的累积负载量也就越大.

RUBiS 应用虽然是分布式应用,但其中的计算并非能按需远程执行.在原有的 RUBiS 应用中,servlet 对 ejb 的调用以及 ejb 之间的调用都是通过 naming service 来完成的.例如,SB_AboutMeBean 这个 SessionBean 用于显示注册用户的信息.它在执行操作之前必须调用 SB_AuthBean 来完成对输入的用户名和密码的验证工作.然而,SB_AboutMeBean 却直接使用了 new InitialContext()语句来从本地 JVM 中获得 EJB 的上下文,进而用来查找 SB_AuthBean.这也就意味着,只有当 SB_AuthBean 和 SB_AboutMeBean 在同一个 JVM 中执行时,从该上下文中对 SB_AuthBean 的 Lookup 操作才会成功.倘若 SB_AuthBean 被放到网络上的其他 JVM 中,在 SB_AboutMeBean 所在的 JVM 中就没有该 EJB 的信息,从而会导致后续 Lookup 的失效.因此我们可以看到,当 SB_AuthBean 改变了部署的网络节点位置时,调用它的 SB_AboutMeBean 也需要进行修改才能适应这种变化,比如需要修改为

```
Properties env=new Properties();
env.setProperty(Context.INITIAL_CONTEXT_FACTORY,
    “org.ow2.carol.jndi.spi.JRMPContextWrapperFactory”);
env.setProperty(Context.PROVIDER_URL,“rmi://192.168.0.189:1099”);
//192.168.0.189 表示 SB_AuthBean 目前所在的 VM 的地址
env.put(Context.URL_PKG_PREFIXES,“org.objectweb.jonas.naming”);
InitialContext ctx=new InitialContext(env);
```

这样,才能从所获的 InitialContext 查找到 SB_AuthBean 并加以使用.然而如前所述,由于开发者很难预料到应用

对资源的准确使用,因此所开发出的 EJB 应用中的本地/远程调用关系固定不变.倘若由于节点 192.168.0. 189 宕机而将 SB_AuthBean 放到另一个网络节点上执行,则对应的 SB_AboutMeBean 又需要改变.因此,原有的 RUBiS 应用虽然是分布式应用,然而却不能做到对计算资源的按需占有和使用.此外,一个 EJB 应用中除了 EJB 类以外,还有很多普通的 POJO(plain old Java object)类,如 RUBiS 应用中的 *edu.rice.rubis.beans.TimeManagement* 类.这些类通常会被 EJB 类在处理业务请求时作为辅助类被调用.也就是说,EJB 和这些 POJO 类之间形成了第 1.1 节所示的直接内存调用结构,当这些类也亟需计算资源而无法满足时,也需要实施计算转移.然而,已有的结构不允许这些类脱离调用它的 EJB 而被转移到其他节点执行.

4.1 DPartner的可用性

如图 2 所示,DPartner 通过字节码级别的程序转换而自动将原始 RUBiS 重构为计算可按需远程执行的 RUBiS.在 RUBiS 所拥有的 59 个类中,所有的 20 个 servlet 类由于继承了被 DPartner 认为是 anchored 类的系统类:javax.servlet.http.HttpServlet 而被自动归为 anchored 类.在其他类中,由于 *edu.rice.rubis.servlets.ServletPrinter* 使用到了一些本地节点才有的文件系统资源,因此也被归为 anchored 类.剩下的 38 个类(普通 POJO 类以及 EJB 的 Bean 类)都被视作 movable 类型的类.经过自动聚类发现,*edu.rice.rubis.servlets.Config* 被 servlet 类频繁调用,因此,它和 servlet 聚类在一起,始终留在原网络节点执行.转换后的各个 ejb 被封装成可以独立部署的 EJB jar 文件,其中包含了转换后的 EJB 实现体 Bean 文件、interface 文件、proxy 文件等.Endpoint 也被封装为一个 EJB jar 文件以部署在 JOnAS[19] JEE 应用服务器中.转换前后的 RUBiS 相关信息对比见表 1.可以看到,转换后,RUBiS 的大小会有所增加,原因在于,DPartner 对字节码进行了重写,生成了接口、proxy 等新的类,并且增加了 endpoint 等基础设施代码.我们也可以看到,转换时间开销大概在 2min.由于转换发生在应用运行前,因此开销是可以接受的.

Table 1 Refactoring performance of DPartner on RUBiS

表 1 DPartner 对 RUBiS 进行重构的性能

度量	RUBiS
原始应用对应的 ear 文件的大小(KB)	596
重构后应用对应的 ear,jar,war 文件的大小(KB)	1437
大小增长(KB)	841
Movable 类型的应用类的数量和百分比	38 (38/59=64.4%)
重构时间开销(s)	112.3

转换后,在 RUBiS 应用中,servlet 对 ejb 的调用以及 ejb 之间的调用实现了按需远程化.例如,如图 3 所示,servlet B 需要调用 ejb E 所提供的功能.Endpoint 通过监控并分析前一段时间内 B 和 E 之间的调用序列信息,包括消息收发时间、数据量、成功次数等,得到历史上 B 和 E 执行的情况概览(profile).若通过分析预测发现 B 和 E 在同一个 JVM 中执行就能维持下一段时间的性能,就直接调用无参数的 new InitialContext()来使 B 获得对本地 E 的引用,从而直接调用 E;而当发现本地资源难以支持 E 执行时,endpoint 会向运行环境中的虚拟机管理器发送一个启动特定资源量的新虚拟机的命令.该虚拟机已经部署好了转换后 RUBiS 应用中的 ejb 副本.接着,本地的 endpoint 使用带参数的 new InitialContext(env)来获得新虚拟机上的 ebj E 对应的上下文,从而实现 B 对在远端执行的 E 的调用.运行在本地以及远端的 E 的激活、状态同步、停用等操作也由 endpoint 负责.需要注意的是,在远程调用时,InitialContext 参数中 E 所在的 URL 地址信息是由虚拟机管理器返回的结果来设定的.此外,endpoint 也提供了配置文件以及 JMX 服务来实现相关参数的动态调整,以保障应用中计算按需远程执行的灵活性.当通过监控发现,随着应用负载量的减少,系统资源变得相对过剩时,servlet 所在的本地 endpoint 就会逐渐回收 E 的执行.也就是说,该 endpoint 会在其所在虚拟机启动本地 E,并进行本地 E 和远程 E 之间的状态同步工作.然后,该 endpoint 将 B 对 E 的请求转发回本地的 E 来处理,即,使得 B 又改为使用无参数的 new InitialContext()来获得对本地 E 的引用.接着,本地的 endpoint 向远端的 endpoint 发送停止远端 E 的请求.若远端的 E 没有被其他类所调用,它就会进入钝化阶段,释放资源.若远端已没有 RUBiS 应用中的计算在执行,则一段

时间后,其 endpoint 就会向虚拟机管理器发送停止自身所在虚拟机运行的请求,以在保障 RUBiS 应用性能的前提下实现对资源的节省.

如前所述,DPartner 在对 RUBiS 进行静态程序分析的基础上,还通过执行测试用例来预先发现 RUBiS 中究竟是哪个 EJB 最耗资源,并且发现 Servlet 和 EJB 之间调用的紧密程度,以在运行时加快应用中计算按需远程执行的决策.在读写情况下,RUBiS 中的 SearchItemsByCategoryBean 的实例所占的计算资源最大,且计算最耗时,并常与 BrowseRegionsBean,BrowseCategoriesBean,SearchItemsByRegionBean,ViewItemBean,ViewUserInfoBean 以及 StoreBidBean 一起使用.这是因为测试用例所模拟的用户在读写情况下,通常会根据自己所在的位置(BrowseRegionsBean)来查看商品的种类(BrowseCategoriesBean)或搜索商品(SearchItemsByCategoryBean/SearchItemsByRegionBean),查看商品的详细信息(ViewItemBean),在决定要竞买商品时,先要查看商品的卖家(ViewUserInfoBean),最后下单子竞买商品(StoreBidBean).因此,上述几个 EJB 会被 DPartner 聚类在一起,在必要时被一起转移到或留在拥有较大计算资源的节点上运行.

4.2 转换后RUBiS应用中计算的按需远程执行

我们使用北京大学自主研发的网构软件测试床 Internetware Testbed[20]中的虚拟机作为实验平台,一共使用了 5 台虚拟机节点,见表 2.其中,$VM_{testdriver}$ 用来放置 RUBiS 的测试驱动器;VM_{apache} 放置 Apache 服务器,用来将来自 RUBiS test driver 的访问请求转发给 VM_{jonas1} 或 VM_{jonas2},后者根据需要用来放置 RUBiS 的 servlet 和 ejb;VM_{db} 用来放置 MySQL 数据库,其中包含有 RUBiS 应用所需的 4 万条商品信息以及 10 万个注册用户信息.这 5 台虚拟机的操作系统版本都是 Ubuntu8.04 server,并由 100M 带宽的网络相连.需要注意的是,VM_{jonas1} 和 VM_{jonas2} 的虚拟 CPU 数和内存拥有量都小于其他节点,只有这样,才能在接下来的实验中保障 RUBiS 应用的性能由其自身所获得的计算资源量来决定.换句话说,我们这样做的目的是要避免 test driver,Apache,MySQL 早于 RUBiS 面临到计算资源不足而成为瓶颈,避免导致影响测试准确性的情况发生**.

Table 2 VMs in the experiment

表 2 实验所用 VM

节点	CPU (vCPU)	内存(MB)	软件系统配置
$VM_{testdriver}$	2	1 024	JDK 1.6, RUBiS test dirver
VM_{apache}	2	1 024	Apache/2.2.8 (Ubuntu)
VM_{jonas1}	1	384	JDK 1.6; JOnAS JEE Server v5.2.0
VM_{jonas2}	1	384	JDK 1.6; JOnAS JEE Server v5.2.0
VM_{db}	2	1 024	Mysql-5.0.51a-3ubuntu5.7-log (Ubuntu)

实验的目的是要对比原始 RUBiS 和转换后的 RUBiS 在性能上的差异(以吞吐量和响应时间为指标),以验证计算按需远程执行的有效性.实验中,RUBiS 的测试驱动器的参数都设为一样:采用读写模式的 request transition table;client number 从 100 逐渐增大到 1 000;每一组 client number 的 session runtime 为 10min.

我们设计了如表 3 所示的对比实验场景.在图 6 中展示了表 3 场景下 RUBiS 应用的吞吐量和响应时间的对比结果.

从图 6 中我们可以得到如下 3 个主要结论:

(1) 计算资源受限是影响应用性能提升的重要因素.

与其他场景相比,Local 场景下 RUBiS 应用所占有的资源是最少的,其性能总体上也是最差的.当 client number 超过 600 时,由于计算资源受限,Local 中 RUBiS 的性能反而急剧下降.当 client number 增大到 800 以上时,整个 RUBiS 应用甚至出现了崩溃现象.

(2) 计算资源应该按需使用以提高利用率.

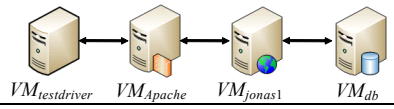
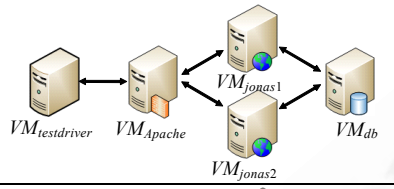
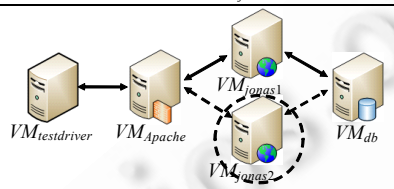
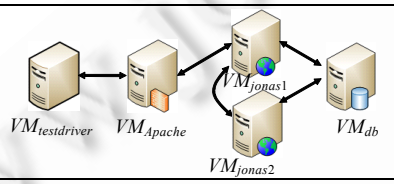
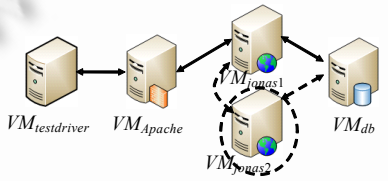
我们也可以发现,Local 场景下 RUBiS 性能并不总是不如其他场景.如当 client number 在 350 以下时,其测

** 我们将 VM_{jonas1} 以及 VM_{jonas2} 的内存都设为 512 及以上容量时,整个 RUBiS 应用的性能瓶颈出现在 MySQL 数据库上.因此,这里我们将这两个 *VM* 的内存值设定得比较小,使其成为真正的系统性能瓶颈.

得的吞吐量甚至是最高的.这反映了其他场景中的 RUBiS 虽然占有了更多的计算资源,但没有充分利用,实际上产生了浪费.对于 FixedCluster 以及 FixedOffload,造成的浪费则是最大的,因为没有必要在用户请求负载量不大的情况下就比 Local 场景多消耗一个 VM_{jonas2} 节点的资源.

Table 3 Testing scenarios

表 3 实验场景

实验场景	系统结构	描述
Local		原始 RUBiS 被完全部署在 VM_{jonas1}.也就是说,所有的 servlet 被封装成了 war 文件,所有的 ejb 都被封装为了 jar 文件.然后,这些文件都被部署在 VM_{jonas1}.
FixedCluster		原始 RUBiS 的两个副本被分别部署在 VM_{jonas1} 和 VM_{jonas2} 上,它们一起形成了 RUBiS 集群.Apache 服务器负责将客户请求按照轮询(round-robin)的方式转发给这两个 RUBiS 所组成的集群.
DynamicCluster		原始 RUBiS 的两个副本被分别部署在了 VM_{jonas1} 和 VM_{jonas2} 上,它们形成了 RUBiS 集群.Apache 服务器负责将客户请求转发给 VM_{jonas1} 所在的 RUBiS.当客户请求量增大到 400 时,VM_{jonas2} 以及其上运行的 RUBiS 被启动.在此之后,Apache 负责将客户请求按照轮询(round-robin)的方式发送给这两个 RUBiS 所组成的集群.
FixedOffload		原始 RUBiS 中的 ejb 被分成两部分,分别运行在 VM_{jonas1} 和 VM_{jonas2} 上.在 VM_{jonas2} 的 ejb 是 *ViewUserInfoBean* 和 *SearchItemsByCategoryBean*.其余 ejb 以及 servlets 都运行在 VM_{jonas1}.Apache 服务器将客户请求转发给 VM_{jonas1}.当 servlets 收到请求后,它会根据请求内容而调用在不同 *VM* 上的 ejb.
On-Demand Offload		重构后的 RUBiS 被首先部署并激活运行在 VM_{jonas1} 上.当 RUBiS 应用系统未能达到既定性能目标时,其中的一些 ejb 就被自动转移到一个新分配的 *VM* 上运行.该 *VM* 已事先部署上了转换后 RUBiS 的 ejb,因而 ejb 转移执行也就是在 VM_{jonas1} 上停止执行某个 ejb 而在新分配的 *VM* 上激活执行对应的 ejb.在本实验中,这个新分配的 *VM* 是 VM_{jonas2}.

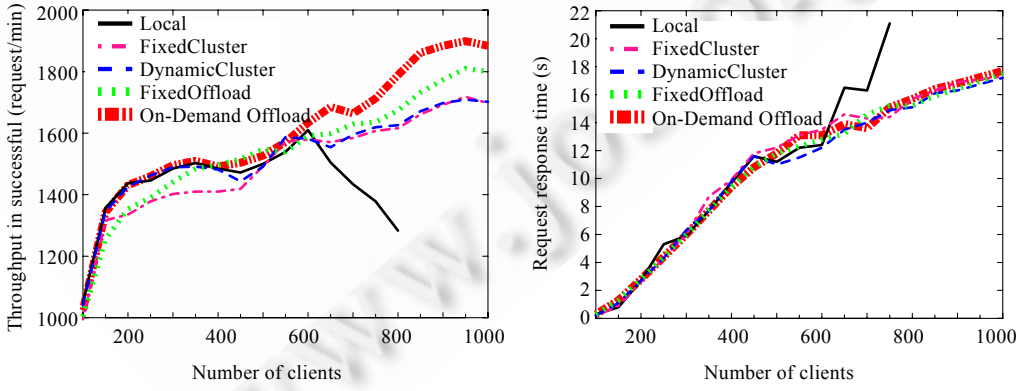


Fig.6 Throughputs and response time of RUBiS in the five experimental scenarios

图 6 5 种场景下 RUBiS 的吞吐量和响应时间

对比 FixedCluster 和 FixedOffload 这两个场景下 RUBiS 的性能可以看到,虽然 RUBiS 占有的资源量是一样的,但 FixedCluster 对应的性能却相对较差.其原因在于,FixedOffload 中的 VM_{jonas2} 只承担 RUBiS 中的 *ViewUserInfoBean* 和 *SearchItemsByCategoryBean* 这两个最亟需资源的 ejb 运行,而 FixedCluster 中的 VM_{jonas2} 却要承担了整个 RUBiS 副本的运行,造成了资源竞争,反而影响了整体性能的提高.同时我们还需要注意,在 FixedOffload 场景中,Servlet 对 *ViewUserInfoBean* 和 *SearchItemsByCategoryBean* 这两个 EJB 的远程调用是固定不变的,这就导致了即便是存在这两个 ejb 的副本运行在 servlet 所属的 JOnAS 应用服务器中,serlvet 也不会调用它们,即也不会进行本地调用.也就是说,当用户请求负载较小(小于 350)而本地资源充足时,这两个 ejb 的计算也还要在远端节点上执行,造成了浪费.

对比 DynamicCluster 和 FixedOffload 这两个场景可以看到,由于动态集群(DynamicCluster)只在需要时(这里为 client number 大于 400 时)才启动新的副本以承载用户请求,因此当负载较小而本地资源充足时,本实验中的 DynamicCluster 就等同于 Local,而这时 DynamicCluster 所得到的性能要优于 FixedOffload,且占用的资源要少.而当用户请求负载量增大时,DynamicCluster 等同于 FixedCluster.根据前面所分析的资源竞争的原因可以看到,这时 DynamicCluster 虽然与 FixedOffload 占用等量的计算资源,然而所得到的性能却不如 FixedOffload.

综合上述分析可以看到,本文所提出的通过计算按需远程执行来保障应用性能并提高资源利用率是必要的,但原始的 RUBiS 应用在各种场景下(即 Local,FixedCluster,DynamicCluster 以及 FixedOffload)都难以达到此目标,因此我们接下来通过 On-Demand Offload 场景来验证转换后的 RUBiS 在实现计算可按需远程执行上的有效性.

(3) On-Demand Offload 能够有效地保障应用性能并提高资源利用率.

经 DPartner 转换后的 RUBiS 应用中计算按需远程执行的场景为 On-Demand Offload.在该场景中,我们参考图 6 的测试结果,为 RUBiS 设定了相应的性能目标:在 client number 小于 350 的情况下,其吞吐量以 Local 中的吞吐量为基准,上下浮动 50 requests/s;当 client number 超过 350 时,其吞吐量以 DynamicCluster 的吞吐量为基准,不设上限,但下限设为 DynamicCluster 吞吐量+(100~200) request/s(同时参考了 FixedOffload 的吞吐量).

分别以 Local 和 DynamicCluster 作为基准性能目标的原因在于,我们想验证 On-Demand Offload 场景中的 RUBiS 是否能通过按需使用资源来保障性能,并且验证其是否在拥有相同资源的情况能够达到更优的性能.我们将性能目标写入转换后 RUBiS 的 endpoint 配置文档中.在运行时,endpoint 会监控 RUBiS 的 servlet 的执行,从而验证目标是否达到.若现有计算资源不能支持目标实现,则 endpoint 会实施计算转移,即选择 RUBiS 中的一部分 ejb,将其计算转移到新申请的 *VM* 上,并不断调整在新 *VM* 上执行的 EJB,从而实现在保障应用性能的前提下,充分地利用计算资源.

在实验过程中,VM_{jonas1} 和 VM_{jonas2} 上运行的 ejb 见表 4.从图 6 和表 4 中我们可以看到:当 client number 小于 400 时,VM_{jonas1} 所拥有的计算资源足以支撑 RUBiS 达到所设定的性能目标,因此在此阶段,RUBiS 中所有的 servlets 和 ejbs 都留在 VM_{jonas1} 上执行;当 client number 增大到 400 时,情况却发生了变化,现有的 VM_{jonas1} 的资源难以支撑 RUBiS 达到设定的目标,因此在 endpoint 的预测和控制下,整个应用获得了一台新的虚拟机 VM_{jonas2},RUBiS 开始将自己的一部分 ejb 转移到这台虚拟机上执行.随着 client number 的不断增大,转移到 VM_{jonas2} 上执行的 ejb 也越来越多.刚开始只有 *Remote*1={StoreCommentBean,PutCommentBean,BuyNowBean} 这 3 个 ejb;待 client number 增大到 800 以上时,已有 *Remote*3 对应的 9 个 ejb 在 VM_{jonas2} 上执行.正如第 3.6 节所述,endpoint 会计算出如何分配 RUBiS 的 ejb 才能最大限度地提高 Q=性能/资源的比值.这里,由于为了减少计算转移时因状态同步引起的网络通信开销,因此 endpoint 选择留下那些对计算资源消耗较大的 ejb 运行在 VM_{jonas1} 上,而将一些不很亟需资源但仍然占有并竞争资源的 ejb(如,*Remote*1 中所包含的 ejb)转移到 VM_{jonas2} 来执行,servlets 对这些 ejbs 的调用也由本地调用自动变为远程调用.如第 4.1 节所分析的,不难从表 4 中看到,资源消耗最大的 SearchItemsByCategoryBean 留在 VM_{jonas1} 上,与它在一起的还有 BrowseCategoriesBean,BrowseRegionsBean,SearchItemsByRegionBean,ViewItemBean,ViewUserInfoBean 以及 StoreBidBean 等 ejb.原因在于这些 ejb 常常被一起调用(见第 4.1 节的分析),而且资源消耗都很大,倘若只将其中某些 ejb 转移到远端执行,

则会增加网络通信开销,导致 client 在模拟 RUBiS 的购物操作过程中,在 session state 转换的时延增大,降低了应用的整体性能.

Table 4 Ejbs that are executed remotely during the "On-Demand Offload" experimental scenario

表 4 On-Demand Offload 实验场景下远程执行的 ejb

客户数量	在 VM_{jonas1} 上运行的 EJB	在 VM_{jonas2} 上运行的 EJB	资源量相同的参考实验场景	性能提升(%)
100~400	全部 servlet 和 ejb	–	Local	–
400~500	全部 servlet 以及除了 *Remote*1 集合以外的 ejb	*Remote*1={StoreCommentBean, PutCommentBean,BuyNowBean}	DynamicCluster FixedOffload	2.4 −1.3
500~600	全部 servlets 以及除了 *Remote*2 集合以外的 ejb	*Remote*2=*Remote*1+ {BuyNowBean StoreBuyNowBean,AboutMe}	DynamicCluster FixedOffload	3.2 2.7
600~700	全部 servlets 以及除了 *Remote*2 集合以外的 ejb	*Remote*2	DynamicCluster FixedOffload	5.3 2.1
700~800	全部 servlets 以及除了 *Remote*3 集合以外的 ejb	*Remote*3=*Remote*2+{RegisterUserBean, RegisterItemBean,ViewBidHistoryBean}	DynamicCluster FixedOffload	10.1 6.8
800~900	全部 servlets 以及除了 *Remote*3 集合以外的 ejb	*Remote*3	DynamicCluster FixedOffload	11.3 7.4
900~1 000	全部 servlets 以及除了 *Remote*3 集合以外的 ejb	*Remote*3	DynamicCluster FixedOffload	10.7 4.6

当 client number 小于 400 时,On-Demand Offload 场景下 RUBiS 的执行等同于 Local 场景下的执行.其性能与 FixedCluster 以及 FixedOffload 场景中 RUBiS 的性能相比平均要高 5.2%,且后两个场景中 RUBiS 还占用了更多的资源,即,多占用了 VM_{jonas2};当 client number 增大到 400 以上时,On-Demand Offload 相对于占用等量资源的 DynamicCluster 和 FixedOffload 来说,性能仍然要高.比如,当 client number 为 900 时,On-Demand Offload 场景下 RUBiS 的性能要比在 DynamicCluster 场景下高出 11.3%,比 FixedOffload 场景也要高出 7.4%.这充分说明了转换后的 RUBiS 在保障应用性能的同时提高了计算资源利用率.

通过以上的一系列实验,我们可以得到如下主要结论:(1) 通过计算按需远程执行来占用充足且足够的计算资源是保障应用性能并节约成本的重要手段;(2) DPartner 能够有效实现应用中计算的按需远程执行,并且经其转换后的应用可以在保障性能的前提下提高对资源的利用率,也更能适应复杂多变的计算环境.因此,实现计算按需远程执行可以作为一种对传统的粗粒度 cluster 进行精细优化的重要方法.

5 相关工作

计算的远程执行是近年来较为热门的研究之一,并且随着云计算的兴起,该研究也进入了一个新的发展时期[21].究其原因,在 Internet 环境下,网络上单节点(比如云中的一台虚拟机)所拥有的资源总是有限的,而多个网络节点所组成的整体却拥有大量的资源.实现计算的远程执行,正是一种通过对远程资源的占有来解决本地资源不足的有效手段.

基于分布式中间件或分布式应用开发框架实现远程执行的方式,要求开发者必须按照框架所规定的编程模型进行开发,并且只适用于新的应用.此外,由于开发者难以预料到应用对资源的使用,因此所开发出的应用的远程/本地调用方式也固定不变,导致应用仍然难以实现按需的计算远程执行.即便是采用集群的方法来实现计算转移,比如使用 JOnAS JEE 应用服务器所提供的 CMI(cluster method invocation)服务[22],按照 Round-Robin 等方式将对某个 ejb 的调用转发到拥有该 ejb 副本的 JOnAS 集群中的某一个成员来处理,开发者仍然需要编写与 CMI 相关的代码并进行相应的配置才能达到目标.并且,现有的按照 Round-Robin 实现的远程调用方案也难以达到按需使用资源的目的.一些计算远程执行的工作是基于 DSM(分布式共享内存系统)来实现的,DSM 可被看成是一类特殊的分布式中间件,提供一个本地和远程所共享的内存存储环境.COMET[23]是基于 DSM 来实现计算远程执行的代表性工作之一,它修改了移动应用的运行支撑虚拟机,使其与服务器虚拟机一起联合提供对于移动应用来说一致的共享内存,促使应用中的计算任务可按需在本地或远程服务器执行.然而,由于修改环境

来构造 DSM 的代价很大,该类工作也仅限于特定的 DSM 环境,难以实用化.

通过自动程序转换来实现计算远程执行的方法不但可用于新的应用,也可以用于遗产应用.并且,转换的自动执行也能有效减少人力开销,因而逐渐成为一个极具潜力的研究方向.Goign[24]是最早通过自动程序转换来将给定的 Windows COM 应用中的计算远程执行的工作之一.由于 COM 构件间的方法调用必须经过虚函数表(virtual function table,简称 VTBL)来查找对应方法,因而 Coign 通过修改指向 VTBL 的指针位置以及 VTBL 中每一项所指向的方法的位置将对某一方法的调用截取后,转发给远端 COM 构件执行.JavaParty[25]是针对 Java 应用来实现其计算远程执行的最早的工作,它在应用源代码上完成程序转换任务,应用最终被转换成符合 Java RMI 的应用.JavaParty 在实施转换前需要开发者在应用源代码中对他们认为需要远程执行的类上标注“Remote”关键字.那些标注后的应用类在转换后增加了 java.rmi.Remote 接口等 RMI 所规定程序元素,并且实现了向 NamingService 注册的工作,从而得以在 RMI 平台的支撑下远程执行.J-Orchestra[9]也是通过自动程序转换而将给定的 Java 应用转换为符合 RMI 规范的应用.与 JavaParty 不同的是,J-Orchestra 所实施的转换是在 Java 的字节码层次上进行的.在转换之前,J-Orchestra 给出了一个 GUI 界面,将该应用中的类(字节码文件)以列表的形式展现出来,然后由开发者指定这些类中有哪些需要被移动到远端网络节点执行.而后,这些类将被 RMI 化.应用留在原节点的部分通过 RMI stub 来调用那些移动到远端执行的类.与以上的研究工作相比,本文所提出的方法以及 DPartner 系统实现具有两个最大的特色:

(1) 转换过程对开发者保持透明.DPartner 不需要开发者在应用源代码上进行标注或是在字节码中进行选择,而会通过自动程序分析来判断哪些应用类是否适合于远程执行,并自动完成转换工作,极大地降低了转换过程给开发者带来的负担.

(2) 转换后所得应用中的计算是可以按需远程执行的.在如 JavaParty,J-Orchestra 等已有工作中,转换后的应用的本地/远程调用结构固定不变,这导致应用难以适应变化的环境,降低了转换方法的实用性.

6 结束语

计算的按需远程执行,是从软件应用自身角度实现对资源按需占有,以保障性能并提高资源利用率的重要手段.本文给出一种通过自动程序转换来实现 Java 应用中计算按需远程执行的方法.该方法提出了一种支持计算按需远程执行的程序结构,并实现了名为 DPartner 的自动程序转换系统,在字节码层次上将原始 Java 应用转换为符合该程序结构的应用.在转换过程中,处理了正确性和有效性等带来的技术挑战.本文以及前驱工作还在 Java 单机应用和分布式应用上进行了一系列实验.结果表明了 DPartner 在实现应用中计算按需远程执行上的有效性,并证明了转换后应用在性能上的提升以及对资源的充分使用.未来的工作是继续改进 DPartner 并将用更多真实的应用来对其进一步验证.DPartner 可以在 http://code.google.com/p/dpartner/上访问到.

致谢 在此,我们向对本文的工作给予支持和建议的同行,尤其是北京大学信息科学技术学院软件研究所网构软件小组的老师和同学表示感谢.

References:

[1] Yang FQ, Mei H, Lü J, Jin Z. Some discussion on the development of software technology. Acta Electronica Sinica, 2002,30(12A): 1901−1906 (in Chinese with English abstract).

[2] Zhang Y, Huang G, Liu XZ, Mei H. Integrating resource consumption and allocation for infrastructure resources on-demand. In: Proc. of the 3rd IEEE Int'l Conf. on Cloud Computing (Cloud). IEEE Press, 2010. 75−82. [doi: 10.1109/CLOUD.2010.11]

[3] Yang FQ, Lü J, Mei H. Technical framework for internetware: An architecture centric approach. Science in China Series E: Technological Sciences, 2008,38(6):818−828 (in Chinese with English abstract). [doi: 10.1007/s11432-008-0051-z]

[4] Mei H, Huang G, Lan L, Li JG. A software architecture centric self-adaptation approach for internetware. Science in China Series E: Technological Sciences, 2008,38(6):901−920 (in Chinese with English abstract). [doi: 10.1007/s11432-008-0052-y]

[5] Weng CL, Li ML, Wang ZG, Lu XD. Automatic performance tuning for the virtualized cluster system. In: Proc. of the 29th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS). IEEE Press, 2009. 183−190. [doi: 10.1109/ICDCS.2009.45]

[6] Johnston-Watt D. Get smart: The case for intelligent application mobility in the cloud. The Cloud Computing Journal, 2010. http://cloudcomputing.sys-con.com/node/1625129

[7] Ma LY, Huang G, Lan L, Liu TC, Wang M, Fan G, Mei H. Towards software architecture based application deployment. In: Proc. of the National Software Application Conf. 2004 (in Chinese with English abstract).

[8] Zhang Y, Huang G, Zhang W, Liu XZ, Mei H, Yang SX. Refactoring android java code for on-demand computation offloading. In: Proc. of the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA). ACM Press, 2012. 233−247. [doi: 10.1145/2384616.2384634]

[9] Tilevich E, Smaragdakis Y. J-Orchestra: Enhancing Java programs with distribution capabilities. ACM Trans. on Software Engineering and Methodology (TOSEM), 2009,19(1):1−40. [doi: 10.1145/1555392.1555394]

[10] Cecchet E, Marguerite J, Zwaenepoel W. Performance and scalability of EJB applications. In: Proc. of the Object-Oriented Programming, Systems, Languages & Applications (OOPSLA). ACM Press, 2002. 246−261. [doi: 10.1145/582419.5824 43]

[11] Fowler M, Beck K, Brant J, Opdyke W, Roberts D. Refactoring: Improving the Design of Existing Code. Addison- Wesley, 1999.

[12] Maletic JI, Marcus A. Supporting program comprehension using semantic and structural information. In: Proc. of the Int'l Conf. on Software Engineering (ICSE). IEEE Press, 2001. 103−112. [doi: 10.1109/ICSE.2001.919085]

[13] Tilevich E, Smaragdakis Y. Portable and efficient distributed threads for Java. In: Proc. of the ACM/IFIP/USENIX Int'l Middleware Conf. (Middleware). LNCS Press, 2004. 478−492. [doi: 10.1007/978-3-540-30229-2_25]

[14] Knapsack problem. http://en.wikipedia.org/wiki/Knapsack_problem

[15] Stoer M, Wagner F. A simple min-cut algorithm. Journal of the ACM, 1997,44(4):585−591. [doi: 10.1145/263867.263872]

[16] JMX. http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

[17] RUBiS. http://rubis.ow2.org/

[18] EBay. http://www.ebay.com/

[19] JonAS. http://jonas.ow2.org

[20] InternetWare TestBed. http://edu-icloud.internetware.org/

[21] Chen W, Wei J, Huang T. W^{4}H: An analytical framework for software deployment technologies. Ruan Jian Xue Bao/Journal of Software, 2012,23(7):1669−1687 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4105.htm [doi: 10.3724/SP.J.1001.2012.04105]

[22] Sicard S, De Palma N, Hagimont N. J2EE server scalability through EJB replication. In: Proc. of the 21st Annual ACM Symp. on Applied Computing (SAC). New York: ACM Press, 2006. 778−785. [doi: 10.1145/1141277.1141455]

[23] Gordon MS, Jamshidi DA, Mahlke S, Mao ZM, Chen X. COMET: Code offload by migrating execution transparently. In: Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI). USENIX Press, 2012. 93−106.

[24] Hunt GC, Scott ML. The Coign automatic distributed partitioning system. In: Proc. of the USENIX Symp. on Operating Systems Design and Implementation (OSDI). ACM Press, 1999. 187−200.

[25] Philippsen M, Zenger M. JavaParty: Transparent remote objects in Java. Concurrency: Practice and Experience, 1997,9(11): 1225−1242. [doi: 10.1002/(SICI)1096-9128(199711)9:11<1225::AID-CPE332>3.0.CO;2-F]

附中文参考文献:

[1] 杨芙清,梅宏,吕建,金芝.浅论软件技术发展.电子学报,2002,12(12A):1901−1906.

[3] 杨芙清,吕建,梅宏.网构软件技术体系:一种以体系结构为中心的途径.中国科学(E 辑:信息科学),2008,38(6):818−828. [doi: 10.1007/s11432-008-0051-z]

[4] 梅宏,黄罡,兰灵,李军国.基于体系结构的网构软件自适应方法.中国科学(E 辑:信息科学),2008,38(6):901−920. [doi: 10.1007/s11432-008-0052-y]

[7] 马丽雅,黄罡,兰灵,刘天成,汪萌,范刚,梅宏.基于软件体系结构的应用部署方法初探.见:全国软件与应用学术会议.北京,2004.

[21] 陈伟,魏峻,黄涛.W4H:一个面向软件部署的技术分析框架.软件学报,2012,23(7):1669−1687. http://www.jos.org.cn/1000-9825/4105.htm [doi: 10.3724/SP.J.1001.2012.04105]

张颖(1983－),男,四川乐山人,博士,讲师,CCF 会员,主要研究领域为分布式系统,软件中间件,软件工程.
E-mail: zhang.ying@pku.edu.cn

黄罡(1975－),男,博士,教授,博士生导师,CCF 会员,主要研究领域为分布式系统,软件中间件,软件工程.
E-mail: hg@pku.edu.cn

刘儇哲(1980－),男,博士,副教授,主要研究领域为软件工程,服务计算.
E-mail: liuxzh@sei.pku.edu.cn

梅宏(1963－),男,博士,教授,博士生导师,中国科学院院士,CCF 高级会员,主要研究领域为软件工程,软件中间件,软件体系结构.
E-mail: meih@pku.edu.cn

李影(1975－),女,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布式系统,软件中间件,软件工程.
E-mail: li.ying@pku.edu.cn

杨顺祥(1975－),男,博士生,主要研究领域为软件工程,软件测试技术.
E-mail: yangsx07@sei.pku.edu.cn