ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.6, June 2006, pp.1328−1337
DOI: 10.1360/jos171328

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

基于容器中间件的组件系统体系结构性能评价*

张 勇[1,2+], 黄 涛[1,2], 魏 峻[1,2], 陈宁江[3]

[1](中国科学院 软件研究所 软件工程技术研发中心,北京 100080)

[2](中国科学院 软件研究所 计算机科学重点实验室,北京 100080)

[3](广西大学 计算机与电子信息学院,广西 南宁 530004)

Architectural Level Performance Modeling of Component System Based on Container Middleware

ZHANG Yong[1,2+], HUANG Tao[1,2], WEI Jun[1,2], CHEN Ning-Jiang[3]

[1](Technology Center of Software Engineering, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

[2](Key Laboratory of Computer Sciences, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

[3](College of Computer, Electronic, and Information, Guangxi University, Nanning 530004, China)

+ Corresponding author: Phn: +86-10-62630989 ext 208, E-mail: yzhang@otcaix.iscas.ac.cn, http://otc.iscas.ac.cn

Zhang Y, Huang T, Wei J, Chen NJ. Architectural level performance modeling of component system based on container middleware. *Journal of Software*, 2006,17(6):1328−1337. http://www.jos.org.cn/1000-9825/17/1328.htm

Abstract: This paper analyzes the effect of Container style middleware on the structure and performance of Component-based system based on architectural patterns, and proposes an approach integrating Container style middleware components and their interaction relation into the application UML (unified modeling language) models. The performance model derived from the integrated UML models can reflect the impact of middleware. So, analysts do not have to know the internal details of middleware at performance modeling. The architectural pattern-based method can be extended to deal with various style middlewares. In the paper, the proposed approach is illustrated by a case study.

Key words: component-based system; performance modeling; container style middleware; architecture pattern; UML

摘 要: 对组件系统性能建模时,需要考虑中间件平台的影响.基于体系结构模式,分析了容器风格中间件对组件系统结构和性能的影响,并提出了一种在组件系统 UML 描述中集成中间件组件及交互关系的方法.从该集成 UML 模型导出的性能模型,能够有效地反映中间件的影响.这样,在对组件系统性能建模时,无须了解中间件内部细节.这种基于体系结构模式的方法可以方便扩展以处理不同风格的中间件,且易于实现自动化.以 EJB 容器中间件为例说明并验证了所提出方法的有效性.

关键词: 组件系统;性能建模;容器中间件;体系结构模式;UML

* Supported by the National Natural Science Foundation of China under Grant No.60573126 (国家自然科学基金); the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312005 (国家重点基础研究发展规划(973))

Received 2006-01-18; Accepted 2006-03-13

中图法分类号: TP311 文献标识码: A

软件性能工程(software performance engineering)从性能的角度对软件的设计进行定量的预测和评价,并用于指导设计过程.其中,体系结构级的性能建模、评价和决策,对保障系统的性能具有重要意义[1,2].

中间件(middleware)解决了分布式组件系统的分布和异构问题,实现了应用逻辑与系统服务关注点的分离,简化了应用开发.同时,也对分布系统的结构和性能产生了影响.在对分布式系统性能建模时,需要考虑中间件平台的影响[3].

目前,从体系结构描述(比如 UML 图)导出性能模型的方法及工具对位于不同层次上的中间件平台和应用系统不能作为一个统一的整体处理,无法直接得到包含中间件在内的系统性能模型[4,5].为反映中间件对分布组件系统性能的影响,目前的一些研究[6−11]或者对整个系统从底层直接建模,或者对应用系统和中间件分别建模,然后再通过特定的方法进行组合.这要求建模者既要熟悉建模方法,又要了解中间件内部细节,增加了性能建模的难度.

本文基于体系结构模式(architectural pattern)分析了容器风格(container style architecture)中间件[12]对分布组件系统结构和性能的影响,提出了一种在组件系统 UML 模型中集成容器中间件组件及交互关系的方法,以获得包含中间件在内的集成体系结构描述.从该描述导出的性能模型可以反映包含中间件的影响,这样就避免了建模时再了解中间件内部细节.这种基于体系结构模式的方法方便扩展,可以处理不同风格的中间件.本文以 EJB 容器中间件为例说明了所提方法,并通过比较模型预测值与实验测量值分析了所提方法的有效性.

本文第 1 节介绍相关背景知识.第 2 节介绍容器风格中间件.第 3 节基于体系结构模式,分析容器中间件对组件系统结构和性能的影响.第 4 节介绍如何在组件系统 UML 模型中集成中间件描述.第 5 节通过例子说明所提方法.第 6 节比较相关工作.最后总结全文.

1 相关背景

UML 是目前常用的体系结构描述方式,并侧重于描述系统的功能性行为.为使 UML 模型能够描述系统性能需求,一种叫做 SPT 性能文档(UML profile for schedulability,performance and time)的扩展语言[13]已经被 OMG 组织采纳并定为规范.SPT 性能文档通过子类型(stereotype)和标记值(tagged value)扩展了 UML 语言,以反映系统的性能需求,为设计时评估系统性能提供了方便.

性能模型可以选用排队网(queueing network)、分层排队网(layered queueing network,简称 LQN)、随机 Petri 网(stochastic Petri net)或者随机进程代数(stochastic process algebra)等进行描述.其中,分层排队网 LQN 能够有效建模软件资源竞争和硬件资源约束,适合描述分布式系统[2,14].

从体系结构描述导出系统性能模型,是体系结构级建模的一个重要环节.目前也提出了一些方法[2].其中,文献[4,5]基于图转换(graph-transformation)提出了一种从 UML 描述(具体包括 UML 协作(collaboration)、UML 活动图(activity diagram)和 UML 部署图(deployment diagram))导出 LQN 性能模型的方法.为给性能模型指定参数,可以利用 SPT 性能文档在 UML 活动图中标记性能信息,比如执行时间、访问频率或资源需求等,然后利用 LQN 模型求解工具[15]对导出的性能模型求解,进而可以根据求解结果评价和指导系统设计.本文中,在获得包含了中间件后的集成体系结构描述后,可以利用上述方法导出系统性能模型.

2 容器风格中间件

对象中间件,如 Web 应用服务器、CORBA 组件环境等,为分布组件系统提供了公共服务,如并发控制、通信、事务、安全及组件生命周期管理等,支持组件系统业务逻辑与系统服务关注点的分离.负责为组件集成系统服务的运行时环境,常被称作容器(container)[12].容器封装了组件,提供了业务逻辑与技术关注点集成的透明性.为集成、扩展和复用这些服务,容器必须足够灵活.为此,主要采用了两种体系结构模式:客户-代理-服务器模式(client-proxy-server)和拦截器链模式(chains of interceptor)[12].代理模式解决了组件通信问题,解耦了组件通

信与业务逻辑、逻辑通信与物理通信,提供了位置透明性;拦截器链模式基于触发器(trigger)提供了一种较为通用的集成和扩展机制.通过这两种模式,可以方便地为组件集成系统服务并对系统行为进行动态修改,增加了容器中间件的灵活性和扩展性.

容器体系结构包含的组件类型主要有容器、派发器(dispatcher)、拦截器和上下文(context).在拦截器中,可以定义事件触发条件.拦截器被注册在派发器中,可以由容器负责,也可以由拦截器自行注册(self-register).容器负责及时通知派发器外部发生的事件,发生符合拦截器注册条件的事件时,派发器将激活相关拦截器,并将包含事件的上下文传递给拦截器(链).拦截器(链)将检查上下文并执行与自身相关的功能.容器内拦截器链的触发过程如图 1 所示.

图 2 给出了一个容器风格的中间件实例:J2EE 应用服务器.中间件服务(比如事务、安全等)被实现为具体拦截器,并可以预先定义触发条件.调用处理器(invocation handler)与代理(stub)一起,负责处理组件间通信.为了提供更进一步的灵活性,客户端也可以定制特定于应用的拦截器.

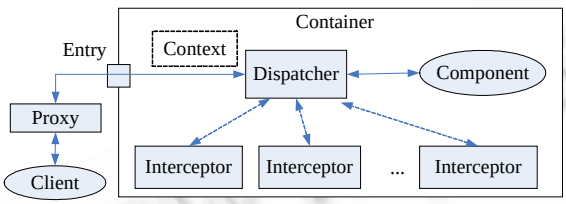

Fig.1 Interaction relation of interceptors in Container middleware

图 1 容器内拦截器链的触发过程

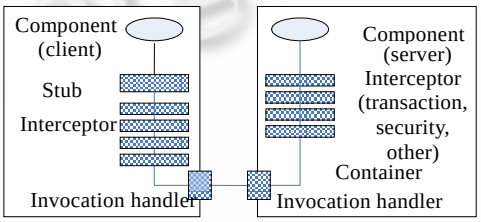

Fig.2 Example container style middleware: Application server

图 2 容器中间件例子:应用服务器

3 容器中间件对组件系统结构和性能的影响

系统性能分析一般针对某种特定的交互场景(scenario)进行[1].为便于分析容器中间件对分布组件系统的影响,我们按照容器中间件中采用的体系结构模式,将分布组件的交互过程分为两个阶段:一是请求从客户组件到达服务器端的容器入口(entry);二是请求在容器内部的处理.

第 1 个阶段——基于客户-代理-服务器模式处理远程调用:客户组件获得服务组件的本地存根,存根作为客户端代理拦截客户请求并经调用处理器序列化(marshaling)后,将请求发送到服务器端.经服务器端的调用处理器解序列化(unmarshaling)后进入容器.在容器内部处理完毕后,从同一路径沿相反的方向传回客户端.通过序列化及反序列化操作,实现了不同语言、不同操作系统平台组件之间的协同.在这个阶段,生成存根、序列化及解序列化等操作,都会造成性能开销.

第 2 个阶段——基于拦截器链模式触发系统服务:请求到达容器后,容器为其生成调用上下文(invocation context),包含事务、安全、调用方法及参数说明等特定于请求的信息.派发器根据上下文调用相关拦截器,并将上下文在拦截器链中传递.被触发的拦截器,除了应用声明使用的中间件服务外,还包括一些隐性服务,如实例处理(instanceHandler)及状态管理(stateHandler)等(这些服务也实现为拦截器),以获得处于就绪(ready)状态的组件实例.最后调用应用组件的业务方法.处理完毕后,响应结果沿相反的方向传回客户组件.

生成调用上下文及容器内各种中间件服务,都会造成性能开销.中间件服务在不同线程控制下,并发为请求提供服务:拦截器完成本环节的处理后,将请求传递给下一拦截器继续进行处理.同时,继续服务下一请求,并不阻塞等待.根据上下文确定拦截器链的开销可以被忽略.确定了具体拦截器后,派发器将不再影响请求性能.容器的初始化(initialization)及销毁(destroy)也会造成开销,但考虑到性能模型反映的是运行时的动态行为,而容器的初始化和销毁一般发生在组件部署和卸载时,因此可以不考虑.我们主要考虑调用过程中各个环节对性能的影响.

上面分析了容器中间件影响系统性能的一般过程.为了对具体应用进行建模,需要明确一些特定于应用的

中间件使用信息,比如:哪些组件在交互时使用了中间件,使用了哪些中间件服务,以及中间件服务的性能开销、资源约束及部署信息等,需要一种合适的方式将这些信息组织起来.本文采用 XML 文件表示这些信息,图 3 以 XML Schema 的形式给出了组织结构的主要部分.

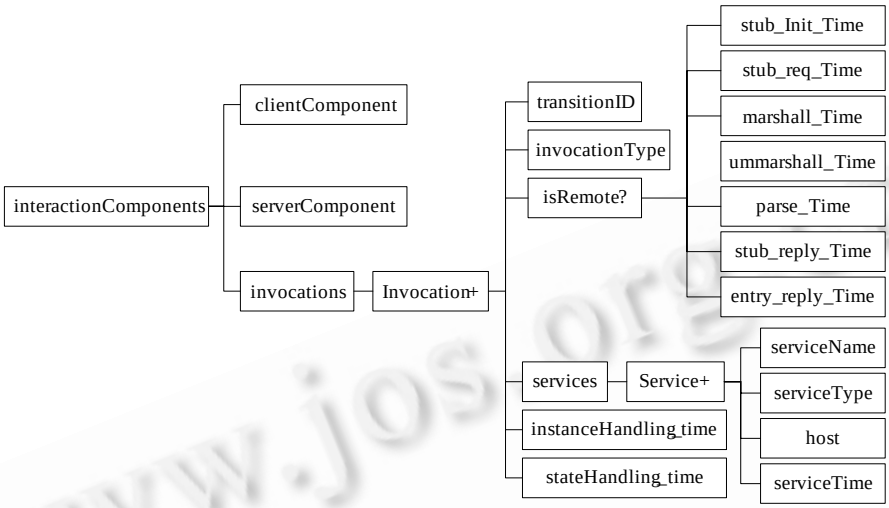

Fig.3 XML schema for middleware information

图 3 与应用相关的中间件使用描述

XML 文件按照使用中间件的交互组件进行描述:元素⟨clientComponent⟩和⟨serverComponent⟩分别表示客户和服务组件;⟨invocations⟩描述它们之间的交互;⟨transitionID⟩对应于应用 UML 活动图中描述该调用的迁移(transition);⟨invocationType⟩表示是否是远程调用,如果是,则需要在⟨isRemote⟩中指定相关属性;⟨services⟩描述使用的中间件服务,包括名称、类型及开销等.本文将一项中间件服务作为一个整体进行建模并指定性能信息,而不再对其内部细节建模.对应用系统性能建模时,需要以 XML 文件的形式提供这些信息.

4 获得集成体系结构描述

根据前面的介绍,性能模型可以从 UML 协作、UML 活动图和 UML 部署图中导出.因此,可以主要考虑如何将中间件信息集成到上述模型中.集成方法的主要思想是:基于前面分析的两种体系结构模式中影响系统性能的一般环节,在特定于应用的中间件使用描述中寻找具体的相关信息;将这些信息增加到应用的 UML 模型中;然后按照体系结构模式内含的组件交互关系,在增加的中间件组件的各个动作间建立调用关系.

首先从 UML 协作开始.UML 协作描述了系统的整体视图,指出了系统的体系结构模式[16].因此,要将容器中间件采用的模式反映到应用 UML 协作中,并修改客户、服务组件最初的协作关系.对于代理模式,需要增加组件 stub,invocationHandler_client,invocationHandler_server 以及它们相互间的“client-server”协作关系;对于拦截器链模式,容器组件、各种中间件服务组件、实例处理器(instanceHandler)、状态处理器(stateHandler)以及它们之间的协作关系(chains of interceptor)也需要增加.

完成 UML 协作的集成后,接下来处理 UML 活动图.活动图反映了组件交互的细节,可用于描述系统的场景[4,5].活动图采用泳道(swimlane)来描述并发软件组件的行为,并可采用组件的名称命名.UML 协作为理解活动图提供了依据.增加到 UML 协作中的每个组件,对应在活动图要增加一个泳道,要将该组件影响性能的动作(action)增加到该泳道中.本文对每个中间件服务组件,只为其增加一个服务动作,并以“服务组件名”+“handling”的形式命名.比如,将事务服务组件“TxService”的服务动作命名为“TxService_handling”.

在活动图中增加完组件及动作后,要对客户、服务组件原来的调用关系重定向.假如是同步调用,则对响应过程也要重定向,并将中间件对响应的影响表示出来.另外,在客户组件向远程服务组件发出第 1 个调用之前,需

要通过名字服务获得远程组件的引用,并生成本地存根.我们在 stub 中增加一个初始化动作 stub_init 来表示这个开销,而不再对名字服务单独建模.集成过程最后处理 UML 部署图,以反映中间件组件在处理节点的分配及其资源约束情况.下面以伪码形式描述上述集成过程的细节:

Parsing XML-based middleware usage describing file;
Obtaining integrated UML Collaboration {
For each *interactionComponents* element {
find original collaboration between *clientComponent* and *serverComponent* and delete it;
For each *service* element {create service component according to *serviceName* attribute;}
create components *instanceHandler* and *stateHandler*;
If *invocationType* between *clientComponent* and *serverComponent* is remote {
create components *stub*, *invocationHandler_client*, *invocationHandler_server* and *Container*;
create *client-server* collaboration relation in turn, among *clientComponent*, *stub*, *invocationHandler_client*, *invocationHandler_server* and *Container*;
create *chain of interceptor* collaboration relation among *Container*, $serviceName_{(1)}$,…,$serviceName_{(n-1)}$, *instanceHandler*, *stateHandler*, *serverComponent*;
} else {/*if invocation is local, only service components are added*/
create *client-server* relation between *clientComponent* and *Container*, *chain of interceptor* relations among *Container*,$serviceName_{(1)}$,…,$serviceName_{(n)}$,*instanceHandler*,*stateHandler* and *serverComponent*;
};
}
}
Obtaining integrated UML Activity diagram {
For each invocation between *clientComponent* and *serverComponent* {
find transition referenced by this invocation (say *original transition*) and delete it;
find source and target partition, source and target action of this transition;
If *invocationType* is remote {
If *stub* not added {/*before the first invocation, adding stub initializing*/
create partition *stub* and action *stub_init* in *stub*;
find starting state *s* of partition *clientComponent*, and target of transition starting in state *s*; (say *s*′)
create transition $s \rightarrow stub\_init \rightarrow s'$;
};
create action *stub_req* in *stub*, partition *invocationHandler_client* and its action *marshaling*, partition *invocationHandler_server* and its action *unmarshaling*, partition *Container* and its action *parse*;
create *transition source*$\rightarrow$*stub_req*$\rightarrow$*marshaling*$\rightarrow$*unmarshaling*$\rightarrow$*parse*; /*redirect the call*/
};
For each *service* sub-element of *services* element
create partition $service_{(i)}$, and actions "*waiting*" and "$service_{(i)}$"+"handling";
/*adding interaction relations among these service partitions*/
create one join in $service_{(1)}$, one input coming from action "*waiting*" of $service_{(1)}$, another input coming from action *parse* of *entry*, and output pointing to action $service_{(1)}$*handling*;
For *i*=2 to *n* {
create one fork in $service_{(i-1)}$, input coming from action $service_{(i-1)}$*handling* of $service_{(i-1)}$,one output pointing to *waiting* of $service_{(i-1)}$, another pointing to *waiting* of $service_{(i-1)}$;
create one join in partition $service_{(i)}$, one input coming from *waiting* of $service_{(i)}$, another input coming from fork in $service_{(i-1)}$, and output pointing to action $service_{(i)}$*handling*;
};
create partition *instanceHandler* and its actions *waiting and instanceHandling*, partition *stateHandler* and its actions *waiting and stateShift*;
/*adding interaction between the two partitions resembles that of above service partitions, here omitted*/

/*finally, adding invocation to business method of serverComponent*/
create transition from action *stateShift* of *instanceHandler* to target action of *original transition*;
if invocation is synchronous {**/*** the response also need to redirect***/**
find transition representing response, *source action* (say *rs*) and *target action* (say *rt*) of the transition;
create actions *marshaling* in *invocationHandler_server*, *unmarshaling* in *invocationHandler_client*, *entry_reply in Container*, and *stub_reply in stub*;
create transition *rs*→*entry_reply*→*marshaling*→*stub_reply*→*unmarshaling*→*rt*;
}
}
}
Obtaining integrated UML Deployment diagram {
find node containing *clientComponent* and *serverComponent* in deployment diagram;
if *invocationType* is remote {
add components *stub* and *invocationHandler_client* to node containing *clientComponent*;
add components *invocationHandler_server* and *entry* to node containing *serverComponent*;
};
add components *instanceHandler* and *stateHandler* to node containing *serverComponent*;
For each *service* sub-element of *services* element {*add* component $service_{(i)}$ to node designated by *host*;}
}

获得集成的体系结构描述后,可以采用 SPT 性能文档[13]在活动图中为中间件组件的相关动作标记性能信息.为活动图标记性能信息的方法可以参见相关的文献[17].标记性能信息后,就可以利用前面背景部分中介绍的从 UML 到 LQN 的方法,得出相应的性能模型.

5 以 EJB 容器中间件为例说明建模过程

下面,我们以基于 EJB 容器中间件的组件系统为例说明上述集成方法.client 组件远程访问 CustomerControlBean 组件,通过该组件从数据库中读取顾客信息,然后再通过该组件修改顾客的 E-mail 信息.读顾客信息时,需要使用安全服务进行验证;修改信息时,需要事务服务支持.EJB 应用的 UML 模型如图 4~图 6 所示.

按照图 3 的格式,下面给出了该应用的中间件使用描述:

⟨componentInteractions clientComponent=Client serverComponent=CustomerControlBean⟩
⟨invocations⟩
⟨invocation transitionID=t01 invocationType=remote instanceHandling_time=0.8ms
stateHandling_time=0.2ms⟩
⟨isRemote stub_Init_Time=3.2ms stub_req_Time=0.1ms marshall_Time=2.1ms unmarshall_Time=2.3ms
parse_time=0.5ms stub_reply_Time=0.1ms entry_reply_Time=0.15ms/⟩
⟨services⟩
⟨service serviceName=SecService serviceType=Security host=serverNode serviceTime=3.3ms/⟩
⟨/services⟩
⟨/invocation⟩
⟨invocation transitionID=t2 invocationType=remote instanceHandling_time=0.8ms
stateHandling_time=0.2ms⟩
⟨isRemote stub_req_Time=0.1ms marshall_Time=2.1ms unmarshall_Time=2.3ms parse_time=0.5ms
stub_reply_Time=0.1ms entry_reply_Time=0.15ms/⟩
⟨services⟩
⟨service serviceName=TxService serviceType=Transaction host=serverNode serviceTime=6.5ms/⟩
⟨/services⟩
⟨/invocation⟩
⟨/invocations⟩

〈/componentInteractions〉

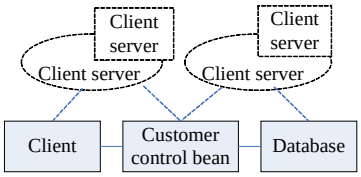

Fig.4 High-Level architecture

图 4 高层体系结构描述

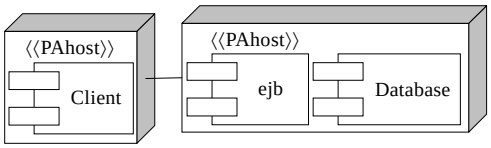

Fig.5 Deploy diagram for the case

图 5 UML 部署图

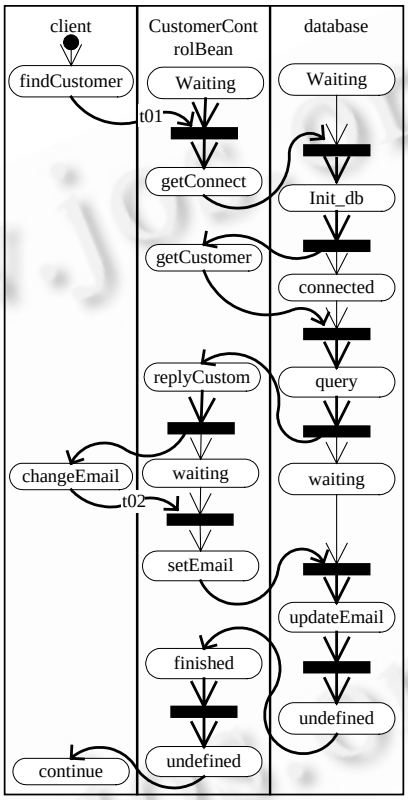

Fig.6 Activity diagram for case

图 6 UML 活动图

按照前述集成方法,得到的包含 EJB 容器中间件在内的集成 UML 模型如图 7~图 9 所示.

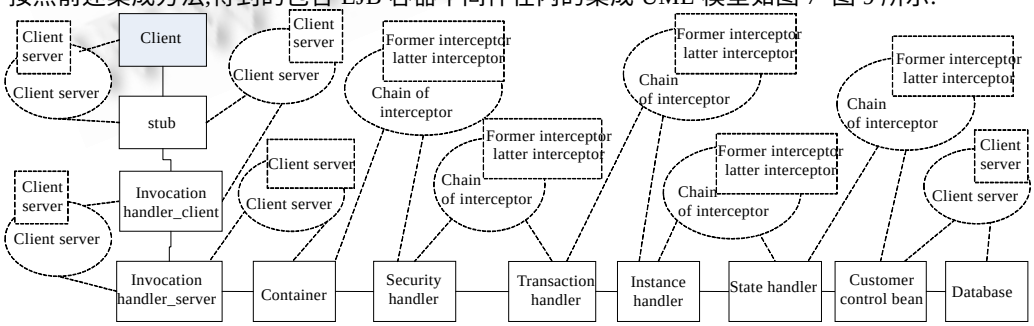

Fig.7 Structural and behavioral view of the collaboration including middleware

图 7 集成了中间件的高层体系结构描述

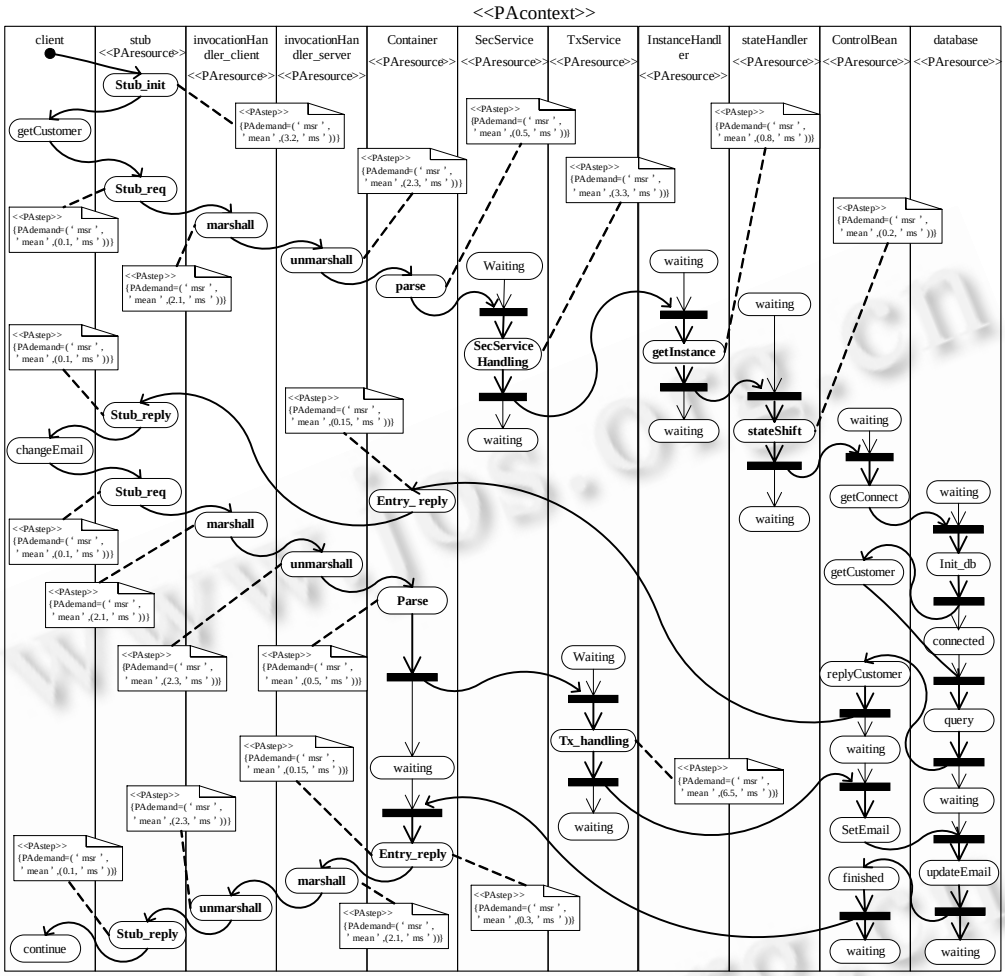

Fig.8 Integrated activity diagram annotated with performance information

图 8 集成了中间件的 UML 活动图

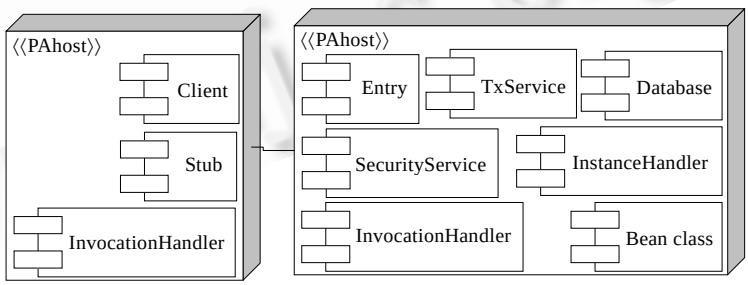

Fig.9 Integrated deploy diagram including middleware

图 9 集成了中间件的 UML 部署图

在活动图(图 8)中,加粗的迁移表示了调用重定向后的路径.采用 SPT 文档给中间件组件动作标记了性能信息,本文以响应时间为例进行了说明.这些动作的执行时间主要基于原型系统,通过实验方法测试得到.实验时,EJB 容器中间件选用 OnceAS2.0 应用服务器[18],操作系统选用 RedHat Linux7.2.为了表达清晰,图 8 中省略了应用组件的性能信息.

利用背景部分介绍的图转换方法[4,5],可以从图 7~图 9 的 UML 模型导出 LQN 模型,并可以利用 LQN 求解

工具[15]来求解模型.由于篇幅有限,此处略去了得到的 LQN 性能模型.为了验证所提出方法的有效性,本文将模型预测值与实验测量值进行了比较.图 10 给出了不断增加客户数量时(从 10~200),请求响应时间与客户数之间的函数关系.对比表明,模型预测与实验测量的误差最大为 12%.分析造成误差的原因,主要有两方面:一是在考察容器中间件影响性能的环节时,只关注了其主要因素,对细微部分考虑不够;二是对涉及后端数据库的操作进行了简化处理,考虑不够充分.

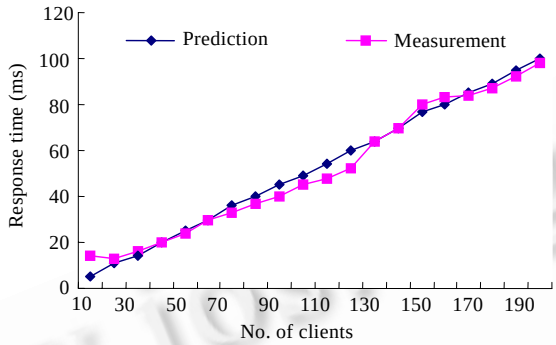

Fig.10 Comparison of prediction with measurement

图 10 模型预测与实验测量的比较

6 相关工作比较

在对组件系统进行性能建模时,为反映中间件平台的影响,一种比较简单的处理方法是调整应用系统性能模型的参数,使其包含中间件的开销.比如建模分布调用时,将生成存根、框架以及序列化等的开销都直接包含在调用的性能参数中.这种粗粒度的处理方法简单、易行,不会增加性能模型的复杂度,但建立的性能模型精确性不够,而且不容易识别性能瓶颈发生在中间件内部时的情况.

另一类处理方法是对包含中间件平台在内的整个系统直接进行性能建模.文献[11]基于应用服务器模块化结构(modular structure)的特点及应用系统的体系结构,采用 LQN 模型对 EJB 组件系统进行了性能预测;文献[6–9]采用 LQN/QN 模型对基于 CORBA 中间件的系统进行了性能建模.与第 1 种方法相比,这种方法能够更有效地表达系统的性能特征,但需要建模人员熟悉中间件内部实现细节,而且无法实现自动建模,必须手工建模.

同样,为了预测基于中间件的组件系统的性能,文献[10]采用了经验测试和数学建模相结合的方法.数学模型反映了包含中间件在内的组件系统的通用行为,其中涉及中间件的部分为待定参数.通过测试套件在特定中间件平台上的定量测试,确定与中间件相关的参数值.这种方法存在两个问题:一是难以将特定于应用的行为包含进性能模型中;二是中间件参数与具体平台密切相关,不同的平台需要不同的测试套件,测试代价比较大.

文献[19]采用了与本文类似的思想,将中间件的影响自动包含进体系结构描述中,提出了一种基于模型驱动(model driven architecture)的方法,将与中间件无关的 UML 模型转换为与中间件有关的 UML 模型,并以 CORBA 中间件为例进行了研究.该文采用的方法与我们的不同,相比之下,我们采用的基于体系结构模式的方法更易于扩展处理不同风格的中间件平台.此外,该文主要解决远程调用对性能的影响,对中间件服务的影响考虑得不够.而本文除了解决远程调用的影响以外,还重点解决了中间件服务的影响.

7 结束语

为反映容器风格中间件对组件系统性能建模的影响,本文基于体系结构模式,提出了一种将中间件组件及其交互关系集成进应用 UML 模型的方法.这样,从集成体系结构描述获得的性能模型能够反映中间件平台的影响,避免了建模时再了解中间件平台的内部细节.文中以 EJB 容器中间件为例说明了所提出的方法,并通过比较模型预测值与实验测量值说明了所提出方法的有效性.这种基于体系结构模式的方法方便扩展,可以处理不同风格的中间件平台,有助于实现整个性能建模过程的自动化.未来的工作主要包括建立自动化的工具支持、考

察在实际大型系统中的应用能力以及简化复杂度等.

References:

[1] Smith CU, Williams LG. Performance Solutions. New York: Addison-Wesley Publishing Co., 2002.

[2] Balsamo S, Marco AD, Inverardi P, Simeoni M. Model-Based performance prediction in software development: A survey. IEEE Trans. on Software Engineering, 2004,30(5):295−310.

[3] Emmerich W. Software engineering and middleware: A roadmap. In: Finkelstein A, ed. Proc. of the 22nd Int'l Conf. on Software Engineering. New York: ACM Press, 2000. 117−129.

[4] Petriu DC, Wang X. From UML description of high-level software architectures to LQN performance models. In: Nagl M, Schuerr A, Muench M, eds. Proc. of the Applications of Graph Trans. with Industrial Relevance Workshop. LNCS 1779, Netherlands: Springer-Verlag, 1999. 47−62.

[5] Petriu DC, Shen H. Applying the UML performance profile: Graph grammar-based derivation of LQN models from UML specifications. In: Field T, Harrison PG, Bradley JT, Harder U, eds. Proc. of the 12th Int'l Conf. Computer Performance Evaluation. Modeling Techniques and Tools. Berlin: Springer-Verlag, 2002. 159−177.

[6] Khkipuro P. Performance modeling framework for CORBA based distributed systems [Ph.D. Thesis]. Finland: Helsinki University, 2000.

[7] Petriu D, Amer H, Majumdar S, Abdul-Fatah I. Using analytic models for predicting middleware performance. In: Woodside M, Gomaa H, Menasce D, eds. Proc. of the 2nd Int'l Workshop Software and Performance. New York: ACM Press 2000. 189−194.

[8] Verdickt T, Dhoedt B, Gielen F, Demeester P. Modeling the performance of CORBA using layered queueing networks. In: Proc. of the 29th Euromicro Conf. New York: IEEE Computer Society Press, 2003. 117−123.

[9] Smith CU, Williams LG. Performance engineering models of CORBA-based distributed-object systems. In: Proc. of the Computer Measurement Group Conf. Computer Measurement Group, 1998. 886−898.

[10] Liu Y, Fekete A, Gorton I. Design-Level performance prediction of component-based applications. IEEE Trans. on Software Engineering, 2005,31(11):928−941.

[11] Xu J, Oufimtsev A, Woodside M, Murphy L. Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates. In: Tracz W, ed. Proc. of the Workshop on Specification and Verification of Component-Based Systems. New York: ACM Press. 2005.

[12] Schmidt D, Stal M, Rohnert H, Buschmann F. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. Vol. 2. John Wiley & Sons, 2000.

[13] Object Management Group. UML profile for schedulability, performance, and time. OMG document ptc/2002-03-02.

[14] Woodside M, Tutorial Introduction to Layered Modeling of Software Performance. Carleton University,. http://sce.carlton.ca/ rads, 2005.

[15] Franks G, Hubbard A, Majumdar S, Petriu DC, Rolia J, Woodside CM. A toolset for performance engineering and software design of client-server systems. Performance Evaluation, 1995,24(1-2):117−135.

[16] Object Management Group. UML specification version 1.4. http://www.omg.org/technology/documents/

[17] Xu J, Woodside M, Petriu D. Performance analysis of a software design using the UML profile for schedulability, performance and time. In: Kemper P, Sanders WH, eds. Proc. of the Computer Performance Evaluation, Modeling Techniques and Tools. Springer-Verlag, 2003. 291−310.

[18] 2005. http://www.once.com.cn

[19] Tom V, Bart D, Frank G, Piet D. Automatic inclusion of middleware performance attributes into architectural UML software models. IEEE Trans. on Software Engineering, 2005,31(8):695−711.

张勇(1975－),男,山东宁阳人,博士生,主要研究领域为网络分布计算,软件工程.

魏峻(1970－),男,博士,研究员,主要研究领域为软件工程,网络分布式计算.

黄涛(1965－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为软件工程,网络分布式计算.

陈宁江(1975－),男,博士,讲师,主要研究领域为软件工程,网络分布式计算.