软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(11):2610−2624 [doi: 10.3724/SP.J.1001.2011.04081]

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

监控使能的分布式软件系统构造方法*

刘东红[1], 郭长国[1,2+], 王怀民[1], 王 涛[1]

[1](国防科学技术大学 计算机学院,湖南 长沙 410073)

[2](中国电子设备系统工程公司,北京 100039)

Monitoring Enabled Distributed Software Construction Method

LIU Dong-Hong[1], GUO Chang-Guo[1,2+], WANG Huai-Min[1], WANG Tao[1]

[1](College of Computer, National University of Defense Technology, Changsha 410073, China)

[2](China Electric Equiment and Systems Engineering Ltd., Beijing 100039, China)

+ Corresponding author: E-mail: cgguo@163.net, http://www.nudt.edu.cn

Liu DH, Guo CG, Wang HM, Wang T. Monitoring enabled distributed software construction method. *Journal of Software*, 2011,22(11):2610−2624. http://www.jos.org.cn/1000-9825/4081.htm

Abstract: Debugging, performance tuning, and trustworthy evolution are big challenges for the ultra large distributed software system. Aiming at these challenges, a monitoring enabled distributed software construction method is proposed, by which a user can understand the software system better. Based on publish/subscribe model, a runtime architecture is given, which separates the business logic and monitoring logic. Based on AOP, monitoring enabled software construction tools are designed which can reduce the production cost of the monitoring software and enhance the maintainability of the code. Based on the runtime framework, a dynamic customizable deployment method on monitoring system is proposed. Monitoring enabled distributed software construction method can reduce the tanglement of function logic and non-function logic at system development, which reduces the cost of code maintenance; It can ensure loose coupling between monitoring system and monitored object at deployment; Besides, It collects runtime information from multi-part of the system and gather them together on demand, which makes programmer and system operator able to understand system behavior as comprehensively as possible, and affect the business running as little as possible.

Key words: monitoring; distributed software system; software construction; trusted software

摘 要: 针对开放的网络环境中大型分布式软件的调试、调优、维护和可信演化问题,提出了伴随式的监控使能分布式软件构造方法.基于发布/订阅的分布计算模型,提出了被监控对象的业务逻辑和监控逻辑分离的运行时体系结构;基于面向方面编程思想,提出了监控使能的分布式软件开发方法和工具,降低了监控实施代价,增强了代码的可维护性;基于运行时体系结构,提出了监控系统的动态可定制部署方法.监控使能的分布式软件构造方法能够在开发时控制功能代码和非功能代码的纠结,尽可能地降低软件编程人员的代码维护难度;能够在部署时保证监控系统和被监控对象的松耦合;能够在运行时实现监控信息的按需汇聚和按需处理.从而在对系统核心业务的影响尽可能小的前提下.获得对系统运行行为尽可能全面的理解.

* 基金项目: 国家自然科学基金(90818028); 国家高技术研究发展计划(863)(2007AA010301)
收稿时间: 2010-08-13; 修改时间: 2010-11-16; 定稿时间: 2011-06-20

关键词: 监控;分布式软件系统;软件构造;可信软件
中图法分类号: TP311 文献标识码: A

在大型分布式软件系统(例如运行在计算机数量可达数千台、甚至上万台的数据中心或者云计算中心的互联网服务)中,由于资源分布、节点众多、交互复杂,导致这类软件的开发、调试、优化和维护非常困难[1],大型分布式系统的问题检测和诊断成为现实的巨大挑战[2].无论是使用经典的形式化方法还是传统软件工程方法,由于分布式软件系统复杂的交互关系以及巨大的时空复杂性,人们均难以准确地描述和把握系统行为,在开发阶段一劳永逸地解决大型分布式软件系统的正确性、可靠性和性能等可信问题;传统上比较有效的调试工具只针对单机软件,而且调试时要暂停软件正常运行,这对上千节点联合提供在线服务的分布式软件系统来讲不具有可行性,而且传统调试工具也难以调试由于交互带来的 Bug 以及系统的性能问题;大型分布式软件系统的深层次 Bug 多数是由于复杂的并发交互引起的,而且往往是在相当的规模下才会出现,小规模的模拟难以复现这些 Bug 或者性能瓶颈[3,4];由于系统规模巨大,硬件和软件故障成为大型分布式软件系统中的常态行为,“带病运行”成为系统的基本特征之一.因此,在运行时理解和把握系统,识别故障和瓶颈,调整系统成为这类软件系统演化的基本特点之一.

鉴于此,在大型分布式软件系统的构造中(例如,Google 的数据中心服务[5]、阿里巴巴的云中心服务),都将监控(monitoring)作为系统重要的组成部分,通过监控系统获取系统实时运行状态,通过对状态的在线或者离线分析和综合来定位和发现系统问题,支持系统可信演化.与民用航空业类比,尽管民航飞机在需求分析、系统设计、模拟测试、生产管理等各个环节均有严格的质量控制,但是仍然不能保证飞机在飞行过程中万无一失.为此,民航飞机都设置了飞行数据记录仪(flight data recorder,简称 FDR)和驾驶舱话音记录仪(cockpit voice recorder,简称 CVR)[6]:把飞机出现故障甚至失事坠毁前一段时间内的有关技术参数和驾驶舱内的声音记录下来,供飞行实验、事故分析之用.现代民用航空业所有的空难调查都离不开这些数据分析,以确定原因,进而改进系统,从而使得民航飞机的可信性不断提高.尽管安装这些设备最早是个别航空公司的行为,但是现在已经成为通行标准.同样地,监控使能也将成为构造大型分布式软件系统的基本要求.构造监控使能的分布式软件系统的核心问题是如何在对系统核心业务的影响尽可能小的前提下获得对系统运行行为尽可能全面的理解.要做到这一点,需要从以下 3 个方面系统研究监控使能的分布式软件系统构造方法:

- 必须研究运行时体系结构,从运行机制上保证监控系统对被监控对象的行为干扰尽可能小;
- 必须研究合理的软件开发模式和工具,使得对被监控对象的代码改动尽可能小;
- 必须研究被监控对象的运行时动态关联关系,将监控能力部署在关键部位,使得需要监控的对象尽可能地少.

通过运行时、开发时和部署时这 3 个方面的工作,保证监控系统对被监控对象可靠性和性能的影响尽可能地小,同时保证对整个分布式软件系统运行行为的有效把握.

本文提出了“伴随式”监控使能的分布式软件系统构造方法.该方法包括 3 个方面的内容:

(1) 基于发布/订阅的分布计算模型,提出了被监控对象的业务逻辑和监控逻辑分离的运行时体系结构.该结构框架实现监控信息的按需汇聚和按需处理,从而保证监控系统和被监控对象的松耦合,同时可增强监控系统的独立扩展能力和系统可信性的可演化能力.

(2) 基于面向方面(aspect oriented)[7]的编程思想,提出了监控使能的分布式软件开发方法和工具.通过监控需求语言和编织工具实现开发时被监控对象的业务逻辑和监控逻辑分离,通过工具将监控探针自动注入源系统,从而控制传统手工编写监控逻辑而造成的功能代码和非功能代码的纠结,尽可能地降低软件编程人员的代码维护难度.该方法提供的监控需求描述语言使应用开发者能够监控那些与系统交互以及性能密切相关的程序元素,例如,监控并发线程、监控和交互密切关联的程序变量和函数等.

(3) 基于运行时体系结构,提出了监控系统的动态可定制部署方法.该方法支持运行时根据监控需要,动态开启监控信息采集点和监控功能.

本文第 1 节针对运行时,介绍监控使能的软件运行时框架.第 2 节针对开发时,介绍监控需求描述语言和监控探针植入工具.第 3 节针对部署时,介绍监控系统的动态可定制部署策略.第 4 节通过案例介绍该方法的使用,分析其效果.第 5 节分析相关工作.第 6 节给出结论和进一步工作的展望.

1 运行时结构

我们将对系统实施监控的方式分为两类:一类是内嵌式,一类是伴随式.这二者的比较分析及相关工作在第 5 节论述.对大型分布式系统来讲,伴随式是一种更加合适的结构.

对分布式软件系统进行调试和调优,首先要能够掌握全局状态,这需要收集多个运行节点的信息;其次,要对汇聚的全局状态进行多种在线或者离线的分析,才能发现逻辑错误或性能瓶颈.因此从功能上讲,运行时结构要能够:

(1) 监控运行节点,收集业务信息和资源使用信息;
(2) 汇聚局部监控信息,形成全局视图;
(3) 对全局信息进行综合分析处理.

同时,为了降低对系统运行时的干扰,从结构上讲,理想的运行时框架应能够:

(1) 采集尽量少(按需采集,需要什么信息采集什么信息)的运行时信息;
(2) 实现系统运行、监控信息采集和监控信息处理这 3 个环节的解耦,使监控信息的采集和处理尽可能少地影响系统运行.

基于上述功能和结构约束,提出伴随式监控使能的分布式软件运行时框架 MERF(monitoring enabled distributed software runtime framework).MERF 使用基于发布/订阅的监控信息按需汇聚方法,局部监控信息汇聚在独立于任何运行节点的发布/订阅服务器,形成全局视图,基于全局视图进行信息分析.这种结构保证了系统运行、监控信息采集和处理 3 个环节在时间、空间和控制上的解耦:这 3 个环节在运行时间上是并行的,在空间上是分布的,在控制上是异步的,从而保证对系统运行的干扰尽可能地小.

MERF 的组成如图 1 所示.分布式软件系统有多个节点(node)组成,软件设计人员可以根据实际监控需要,在分布式软件系统的多个目标系统节点中部署监控探针(probe).监控探针的作用是运行时获取程序和系统运行状态,状态信息缓存在本地存储介质(例如磁盘)中,经过本地 agent 异步汇集到监控信息发布/订阅服务(P/S service).发布/订阅服务将信息按需派发给处理单元,根据需要,可以配置多个处理单元,例如,运行状态记录仪(recorder)、显示仪(displayer)和运行状态分析器(analyzer).运行状态记录仪根据需要持久化程序运行状态,以备需要时分析之用;显示仪根据需要实现运行状态信息或者运行状态分析结果的可视化;运行状态分析仪则按照部署的业务规则对软件运行状态进行分析和判断,并将分析结果交由显示仪展现,系统管理和维护人员可依据分析结果对系统实施调控.

该运行时框架有 4 个关键环节:

(1) 汇聚.在分布系统中,单个节点的信息不足以形成对全局的判断,所以需要多节点信息的汇集才能进行关联和全局分析.
(2) 最小化探针功能.探针仅负责获取运行状态,而不负责状态的汇聚和处理,一方面是因为探针获得的仅是局部信息,不足以用来判断系统;另一方面是因为探针功能的单一性有助于实现对系统运行时的干扰尽可能地小,无论是出于稳定性还是性能的考虑.
(3) 基于 Agent 的解耦.探针收集的运行信息缓存在本地,由本地节点的 Agent 再按需汇聚到 P/S 服务,这种异步结构解耦了监控系统和被监控系统,对被监控系统的性能和可靠性的影响尽可能地降低.
(4) 扩展性.运行时可根据需要动态配置记录仪和分析器,例如,增加或者修改分析器的业务规则、调整记录仪的过滤策略等,而这些调整并不影响已经部署的探针.这种分离结构保证了对分布式软件系统的支持,同时对软件运行时影响小,而且具有较好的扩展性.

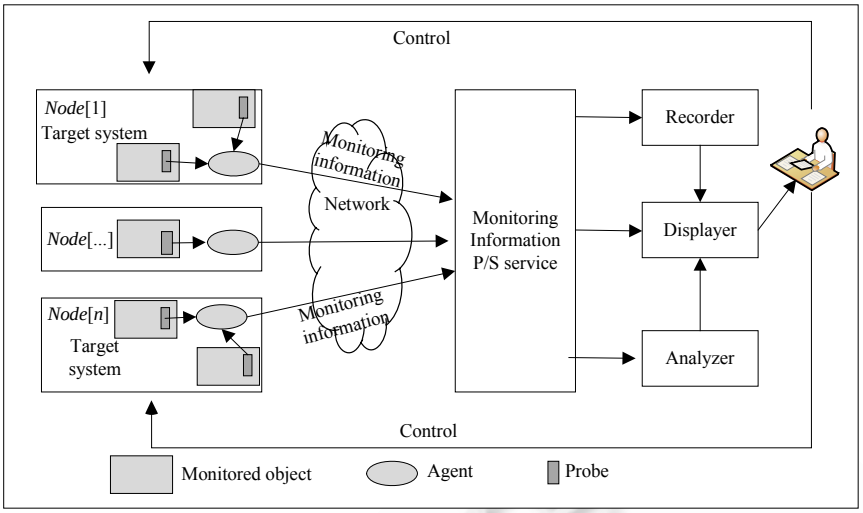

Fig.1 Monitoring enabled distributed software runtime framework MERF

图 1 监控使能的分布式软件运行时框架 MERF

2 开发时工具

基于伴随式 MERF,监控系统的开发涉及监控探针、信息采集 Agent、监控信息汇聚和监控信息分析等 4 个部分,其中,只有监控探针与被监控对象相关;其他 3 个部分是伴随式成分,与被监控对象完全分离.本节仅讨论涉及被监控对象的开发问题.

监控系统运行必然要在被监控对象中部署探针以观察系统,观察系统运行有两种典型的方法:一种是黑盒方式,一种是白盒方式.黑盒方式将系统看作不透明的盒子,无需系统代码,通过外部观察系统的资源消耗(比如 CPU、Mem 和网络带宽等)以及系统底层的消息(例如,网络应用中 TCP 层的报文等)推理系统行为,进而实施分析诊断.这种方式的好处是几乎无须改动被监控系统;缺点是信息不够精准,而且推理一般需要大量的样本.Pinpoint[8]属于该类型.白盒方式则基于对系统行为的预先判断,特别是在程序员的帮助下可以在系统代码中定向性地植入探针,有针对性地观察被监控对象.Pin[4],DBC[9]等均属于这种类型.白盒方式的优点是监控的针对性强,信息准确;问题是探针植入导致程序员额外的工作量,而且监控代码和功能代码纠结后代码维护难度加大.本文提出使用监控描述语言来减少程序员探针植入的工作量,使用基于方面的代码编制工具自动实现监控代码和功能代码的合并,从而不增加代码维护难度.

2.1 MPL语言

对软件实施监控的前提是在其中植入监控探针,这对软件的生产和维护带来了额外的开销:一方面,编写探针需要额外的设计和编码工作;另一方面,探针散落在程序的功能单元间,导致功能代码和监控代码的纠结,增加了软件代码维护的难度.

使用软件工具是提高生产率和增强可维性的有效方法.本文设计了监控探针定制语言(monitoring probe language,简称 MPL)及对应的编译工具.

软件设计者使用 MPL 描述所需的监控探针,定制获取软件的运行时状态.软件的运行状态体现在两个方面:一是软件的资源消耗;二是软件交互活动,软件交互活动集中体现在程序基本元素的实时属性上.MPL 关键组成的 BNF 见附录 1 中的表 8.MPL 可监视软件运行时的计算资源、存储资源、系统句柄以及网络连接资源的消耗.MPL 可监视的交互活动包括函数、通信与并发线程以及和交互相关的变量.

函数是系统交互中可调用的基本功能体,MPL 可监视函数某次执行的性能、函数开始执行和结束执行的绝对时间、函数执行时的参数以及函数运行时的线程上下文.线程是系统并发可调度的基本单元,MPL 可监视

线程的开始和结束时间以及线程当前正在执行的任务.变量的值则直接反映了软件的交互中的状态,MPL 可监视变量被修改的时间以及变量修改前后的值,还可以监视变量被修改时的函数或者线程上下文.

2.2 MPL编译工具

MPL 编译工具将 MPL 描述的监控探针需求编译成与源系统兼容的监控探针代码,并注入源系统,从而使目标系统具备被监控能力.有两种编译 MPL 的方法.一种方法是将 MPL 直接编译为与源系统相同的代码并植入系统.比如,源系统使用 C++代码开发,那么编译器将 MPL 直接转换为 C++代码,并植入相应的位置.这种方法不仅要对 MPL 进行处理,还需要对源系统代码进行处理,以实现插入.另一种方法是将 MPL 转换成某种中间代码,再利用合适的工具将中间代码植入源系统.这种方式可以充分利用已有的工具,简化编译及注入过程.

面向方面编程(aspect oriented programming,简称 AOP)[7]可独立对软件的非功能方面编程,并和功能方面进行编织(weaver),形成目标系统.监控可以作为软件的重要非功能方面对待.因此,本文将 Aspect 作为 MPL 的中间代码形式,将 MPL 编译为 Aspect 代码,再利用 AOP 编织器来实现方面代码植入,其处理流程如图 2 所示.

首先,开发人员根据需要,利用设计的监控探针定制语言编写监控探针;然后,MPL 编译器实现监控探针到监控方面代码的转换;最后,由 AOP 编织工具将监控方面代码注入源系统中,生成具备被监控能力的系统.其中,MPL 编译器是编译工具的主要组成部分,也是我们的主要工作,AOP 编织器则采用了现有的编织工具.

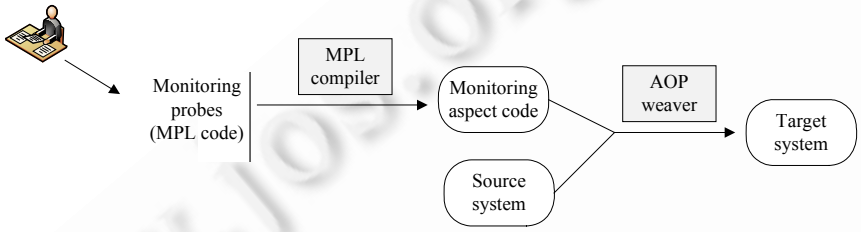

Fig.2 MPL processing flow

图 2 MPL 处理流程

MPL 监控探针文件(MPL_f)由 1 条或多条 MPL 语句(MPL_s)组成,每条 MPL_s 包含了监控类型 M_t、监控元素 M_e 及 1 个或多个监控属性 M_a.MPL 编译处理的基本思想是,通过解析监控语句判断该语句的监控类型、监控元素以及监控属性,从而确定 Aspect 代码的切入点(pointcut)和通知(advice)等,实现监控需求到方面代码的映射.附录 2 中的表 9 是基于 AspectJ[10]的由监控类型和监控元素属性到切入点关键字、通知关键字以及处理动作的映射.

根据映射,MPL 编译器编译算法如下:

算法 1. MPL 编译算法.

输入:MPL 描述的监控探针文件.

输出:Aspect 代码.

1. get the input MPL_f;
2. parse the first MPL_s;
3. do
4. get M_t and M_e;
5. get the first M_a;
5. do
6. lookup the mapping table, get the K_p, K_a and A_p;
7. generate the pointcut by using M_a and K_p;
8. turn A_p into corresponding programming language code;
9. generate the aspect code, add to the aspect file;

10 get the next M_a

11. while (not the end of the MPL_s)

12. parse the next MPL_s;

13. while (not the end of MPL_f)

基于此编译算法,MPL 编译器编译 MPL 探针生成 Aspect 代码.在进行编织时,AOP 编织器通过解析切入点表达式查找连接点,然后将通知中定义的动作插入到核心模块的相应位置,生成最终的系统.同时,为了能够运行时对探针的开关进行配置,在生成的探针逻辑中还包含了相关的判断逻辑.当前,针对主流的编程语言都有了相应的 AOP 编织器,如针对 C++的 AspectC++[11]以及针对 Java 的 AspectJ[10]等,根据需要我们可以选择不同的 AOP 工具来完成最终的编织过程.

2.3 实 例

图 3 是一个“生产者/消费者”的 Java 示例程序的 UML 类图.如图 3 所示的程序包括主函数 main、两个线程类 Producer 和 Consumer、消息类 Message 以及消息队列类 Queue.生产者线程 Producer 生产 Message 对象并添加到 Queue 中;消费者线程消费 Queue 中的 Message 对象;Queue 中的变量 count 表示当前 Queue 中 Message 对象的数量;Message 通过函数 setContent 设置消息的内容.

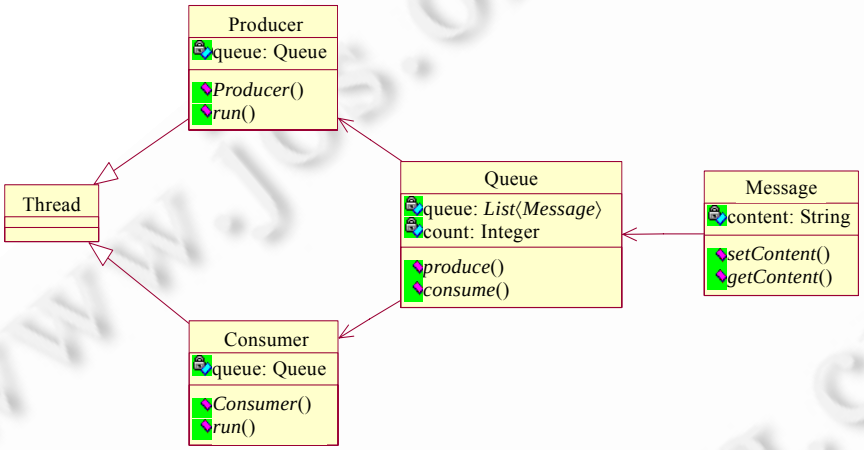

Fig.3 UML of example program

图 3 示例程序的 UML 类图

针对如图 3 所示程序的监控需求包括监视 Producer 和 Consumer 线程、Message 的 setContent 方法、Queue 的变量 count 以及程序运行时所占用的系统资源,其 MPL 描述见表 1.

Table 1 Monitoring requirements using MRL

表 1 使用 MRL 语言表达的监控需求

```
//Thread monitoring requirements
Monitor_Thread(“Producer”,CREATE_TIME|START_TIME|END_TIME);
Monitor_Thread(“Consumer”,CREATE_TIME|START_TIME|END_TIME);
//Function monitoring requirements
Monitor_Function(“Message.setContent(String)”,ENTER_TIME|PARAMETER|THREAD_CONTEXT);
//Variable monitoring requirements
Monitor_Variable(“Queue.count”,CHANGE_TIME|BEFORE_CHANGE_VALUE|AFTER_CHANGE_VALUE|
                 THREAD_CONTEXT|FUNCTION_CONTEXT);
//Resource monitoring requirements
Monitor_Resource(CPU_TIME|MEM_USAGE|START_TIME|HANDLE_COUNT|NET_CONNECTION);
```

监控逻辑的生成方式与具体编程语言无关.针对不同的语言,MPL 编译器产生不同的 Aspect 代码,再使用对

应的 AOP 编织器进行编织.表 2 是根据 MPL 编译算法对表 1 所示的监控需求编译后生成的 AspectJ 代码框架.

Table 2 Compiled result of monitoring requirements

表 2 编译后的监控需求

```
public aspect Aspect {
        pointcut producerCreateTime(): within(Producer) && execution(new(...));
        pointcut producerRun(): execution(void Producer.run());
        pointcut consumerCreateTime(): within(Consumer) && execution(new(...));
        pointcut consumerRun(): execution(void Consumer.run());
        pointcut queueSizeChange(): set(int Queue.count);
        pointcut methodCall(): call(void Message.setContent(String));
        pointcut process(): execution(void Test.main(...));

        before(): process(){
        //start a threaf to detect system resource usage and send information to the monitor service
        }
       before(String msg): methodCall() && args(msg){
        //record the time while starting to call "Message.setContent()", get the parameter of it,
        //acquire the thread context, and send these information to the monitor service
        }
        before(): producerCreateTime(){
        //record the time when a Producer thread be created and
        //send these information to the monitor service
        }
        before(): consumerCreateTime(){…}
        before(): producerRun(){
        //record the time while starting to run a Producer thread and
        //send these information to the monitor service
        }
        before(): consumerRun(){…}
        after(): producerRun(){
        //record the time when a Producer thread stopped and
        //send these information to the monitor service
        }
        after(): consumerRun(){…}
        before(): queueSizeChange(){
        //record the value of "Queue.count" before change, acquire the thread and function context
        //and send these information to the monitor service
        }
        after(): queueSizeChange(){
        //record the time when "Queue.count" be changed and
        //record the value after change, send these information to the monitor service
        }
}
```

针对 Java/C++语言,本文分别实现了软件构造环境 BlackBoxJ 和 BlackBoxC++.它们是对典型程序开发环境(Eclipse 和 Visual Studio)的扩展.

3 部署时策略

MERF 的部署包括部署结构、配置关系以及策略选择,这些都是为了能够在对系统的影响尽可能小的前提下获得对系统运行行为尽可能全面的理解.部署策略可以概括为"全面部署、动态配置",即部署时将所有的基础架构一次性部署完毕,运行时再根据需要对各个环节进行动态配置,实现对系统运行干扰尽可能小.

3.1 部署结构

部署监控系统涉及探针、Agent、P/S 服务以及各种监控信息处理单元的部署:

(1) 探针部署:在构造业务系统时,在编译阶段,探针被植入被监控对象中.因此,探针作为业务系统的一部分进行部署.

(2) Agent 部署:Agent 部署在分布式软件系统的每一个节点上,它负责:

(A) 根据监控需求,动态控制被监控对象的探针是否工作;

(B) 根据监控需求,将该节点上被监控对象的状态信息提交到指定的 P/S 服务中.

(3) P/S 服务部署:P/S 服务一般独立于业务系统部署在单独的节点上.根据系统规模和业务相关性需要,可以部署一个或者多个 P/S 服务,多个 P/S 服务可以互相独立,也可以互相级联.

(4) 监控信息处理单元部署:每一个 P/S 服务都部署若干监控信息处理单元.监控信息处理单元靠近所属的 P/S 服务进行部署.监控信息处理单元的信息处理结果可以作为另一个 P/S 服务的信息源.监控信息处理单元一般情况下应包括 Recorder,Displayer 和 Analyzer.

3.2 配置关系

配置关系包括探针配置、输出配置、Agent 配置以及监控信息处理单元配置:

(1) 探针配置:尽管系统构造完成后探针即被植入,但是可以根据需要通过配置文件的方式动态开启或者关闭探针,不需要时可关闭探针,尽可能地降低对系统运行的干扰.

(2) 输出配置:探针获取监控信息后,可以根据配置将信息缓存在内存中,也可以写入本地磁盘,或者直接存入数据库系统,也可以直接交给本地 Agent,根据需要可以选择最恰当的配置关系.存储在内存中的效率最高,但是容量有限;本地磁盘存储容量大,但是速度会有一定的损失;数据库存储在容量和性能上都有较好的保证,但是会带来部署复杂性.通过 Agent 存储容易造成性能瓶颈,但是可以对监控信息进行细粒度控制,比如可以根据需要动态丢弃认为不必要的监控信息.

(3) Agent 配置:一个 Agent 负责一个节点上的全部被监控对象,因此需要配置 Agent 和被监控对象的对应关系;同时,Agent 将监控信息汇聚到 P/S 服务,因此需要配置 Agent 和 P/S 服务之间的关系;

(4) 监控信息处理单元配置:配置监控信息处理单元的信息源(P/S 服务)位置.

3.3 动态配置策略

在运行时动态选择被监控对象,应遵循精和准的原则:

- 所谓精,就是要研究被监控对象的运行时动态关联关系,将监控能力部署在关键部位,尽量监控那些关系全局的关键对象,使得需要监控的对象尽可能地少.
- 所谓准,就是要研究被监控对象的业务逻辑,监控那些关系全局交互的和处理全局任务相关的方法、变量和线程.

4 实验验证

本节通过已经部署的一个服务质量可跟踪的分布式数据可靠传输系统从生产率、可维性和运行效率等方面来分析构造工具的应用效果.所选取的系统由多个管理域组成,且部署在广域网,对其实施监控具有实际意义,系统既需要域内监控又需要域间监控.本文研究的实际需求之一就来源于该系统,以此为案例对研究成果进行实验验证具有代表性.

图 4 是系统的简化部署示意图.抽象地讲,全系统由若干个(>200)管理域组成,每个管理域由 5 种服务组成.其中,SD 服务维护本管理域的资源和用户目录,同时与其他域的 SD 服务交互,形成全系统目录.每个域中的其他服务或用户将需要可靠传输的数据(可以抽象的表示为⟨*username*,*data*⟩)统一递交给 SA 服务,SA 服务负责对数据进行分拣和标识(结果可抽象表示为⟨*glabalusername*,*targetSHM*,*data*⟩),并提交给 SHM 服务进行传输. SHM 服务对数据进行进一步处理(结果可抽象表示为⟨*targetip*,*data*⟩),并将数据提交给 STR 传输;STR 同时负责数据传输过程中的跟踪,它向 SA 递交数据当前已确认到达的环节,这些环节包括本管理域 STR(*localSTR*)、目标 STR(*targetSTR*)、目标 SHM(*targetSHM*)、目标 SA(*targetSA*)以及最终用户(*targetUser*).STR 则根据实际系统和网络状态将数据分包并发地传输给目标 STR;目标 STR 进行重组后将完整的数据提交给目标 SHM,同时给源 SHM 发送回执.目标 SHM 对数据分析后递交给某个合适的 SA,同时发送回执.目标 SA 将最终数据提交给用户,而后再发送回执,从而完成一次数据传输,并实现整个传输过程的可跟踪.在一个管理域中,可能存在多个 SA,SHM 和 STR,但是一般只有 1 个 SD.

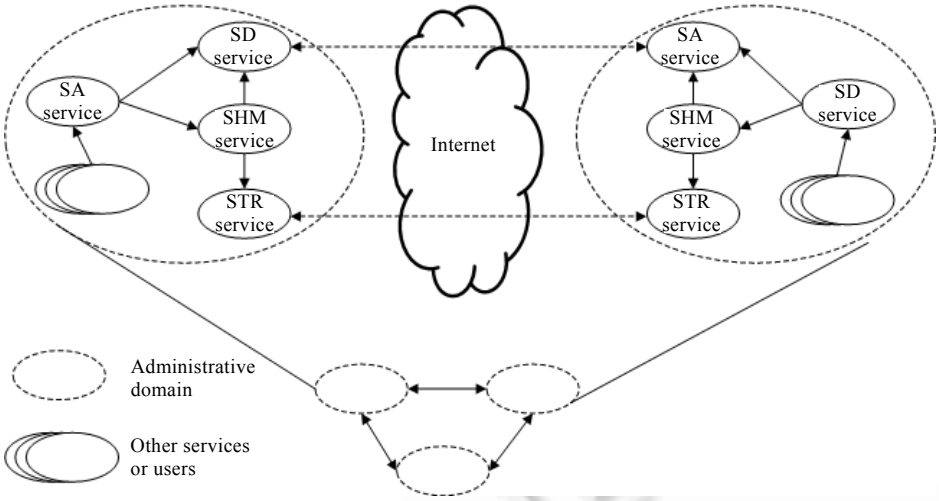

Fig.4 System's deployment view

图 4 系统部署视图

在上述分布式软件系统中,每个管理域中的 SA,SD,SHM 和 STR 构成了一个相对完整的小系统,它们之间的交互对其他管理域的影响较小.因此,在部署监控系统时,在每一个管理域中都部署一个 P/S 服务和若干监控处理单元,以实现对小系统的监控;同时,不同管理域之间通过 SD 和 STR 又构成了一个大系统,因此在全系统中同时部署一个 P/S 服务和若干监控信息处理单元,以实现对全系统的监控.每个独立管理域的监控信息处理单元可以将监控结果作为全系统 P/S 服务的信息源,如图 5 所示.

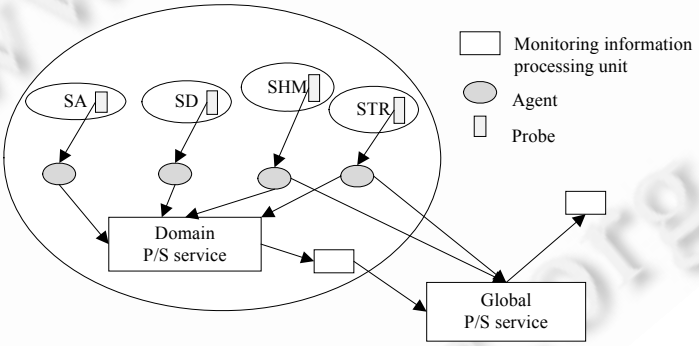

Fig.5 Monitoring information processing unit

图 5 监控系统部署视图

这 4 种服务在没有使用该监控使能软件构造工具之前,其监视功能由不同的程序员手工编写.这导致了复用率低、功能代码和监控代码纠结(tangling)以及不利于代码维护等问题.

表 3 是对这 4 个软件监控代码的统计情况,表中列出了在每个软件中,程序员手工编写的总代码行数,以及该软件中程序员手工编写的获取软件运行状态逻辑,即探针的代码行数.实际统计结果表明,这些手工编写的代码平均占据了手工编写代码的 11.01%,而且大部分代码具有类似的功能,说明代码复用率较低.表中还列出了在每个软件中,手工编写的文件个数以及探针代码所分布的文件个数,可以看到,手工编写的探针代码分布在平均 32.64%的文件中.这反映了监控代码中存在较高纠结度,不利于软件维护.

Table 3 Statistics about monitoring code

表 3 监控代码统计

Software	Total written lines of code	Total written lines of monitor code	Percent (%)	Num of files	Num of files with monitor code	Percent (%)
SD service	23 677	2 631	11.11	43	21	48.83
STR service	53 384	5 976	11.19	114	30	26.32
SHM service	50 394	4 105	8.15	62	24	38.7
SA service	23 143	3 873	16.74	69	19	27.54
—	150 598	16 585	11.01	288	94	32.64

4.1 生产率和可维护性方面的分析

使用本文的工具对系统进行改造后得到表 4 的数据,对比了改造前和改造后软件监控代码的统计情况.

为衡量生产率,表 4 对比列出了改造前使用 C++手工编写的探针代码行,以及改造后使用 MPL 手工编写的探针代码行,二者实现几乎相同的功能.可以看到,为完成相同功能,设计者所需编写的代码行数大幅度减少(平均是原来所写代码的 3.71%).

为了衡量软件维护性,表 4 对比列出了改造前每个软件中探针代码在源文件中的分布情况,以及改造后探针代码在源文件中的分布情况.可以看到,改造后的探针代码集中在一个独立的文件中,不再有代码纠结情况,较好地实现了方面分割,降低了软件维护的代价(尽管这些代码编译后分布到了多个文件,但这些都是由构造工具完成的,不会增加设计者的维护代价).从实际的监控能力来看,改造后的软件可以非常方便地完成原有系统中对资源、变量、函数以及线程的监控.

Table 4 Statistics about monitoring code in modified software

表 4 修改后软件中的监控代码统计

Software	Total written lines of monitor code in original system	Total written lines of monitor code in modified system	Percent (%)	Num of files with monitor code in original system	Num of files with monitor code in modified system
SD service	2 631	51	1.94	21	1
STR service	5 976	204	3.41	30	1
SHM service	4 105	138	3.36	24	1
SA service	3 873	223	5.76	19	1
—	16 585	616	3.71	94	1

在为期 13 个月的软件运行维护中,超过 75%的严重故障是由并发、软件运行中的交互以及软件和环境交互所造成的.这些场景很难在测试中发现,也很难在实验中重现.监控获得的场景信息成为分析故障的主要依据,缩短了故障诊断时间;同时,由于超出预期的并发规模,监控也及时发现了资源过度消耗等额外问题.

4.2 运行效率方面的分析

监控一定会影响软件的运行,尤其是效率.运行框架对软件运行效率的影响表现在如下几个方面(其中的测试数据均在 WindowsXP,CPU:Intel®Core™2Duo CPUT8300@2.40GHz,1.0GB RAM,Intel®82566MM,Gigabit Network Connection 的配置下测得):

(1) 在获取线程、函数和变量的状态时,会在线程构造函数或者函数入口和出口处植入探针,会相应地增加运行时开销:一部分是获取状态的开销,一部分是将状态信息经网络传输到发布/订阅服务时的开销.在运行平台一定的情况下,这些探针所消耗的时间基本是常量,因此不会改变原有程序的计算复杂度.

测试获取函数运行状态的探针开销见表 5,第 1 行是植入探针的函数运行时间,第 2 行没有植入探针的函数运行时间.分别测试 10 组(每一组都对应一次软件的单独运行),测得的附加开销为 28.38ms.

测试获取线程状态的探针开销见表 6,第 1 行是植入探针后线程构造函数的运行时间,第 2 行是没有植入探针时线程构造函数的运行时间.分别测试 10 组(每一组都对应一次软件的单独运行),测得的附加开销为 10.29ms.

Table 5 Function monitoring time

表 5 监控函数的开销

Test number	1	2	3	4	5	6	7	8	9	10	AVG
Time (ms) with monitoring	630.31	613.18	622.35	600.70	614.10	618.75	606.99	591.19	622.36	599.92	611.99
Time (ms) without monitoring	559.80	595.68	575.53	577.52	593.97	581.80	591.83	582.47	578.47	598.99	583.61
Time (ms)	Time with monitoring-time without monitoring										28.38

Table 6 Thread monitoring time

表 6 监控线程的开销

Test number	1	2	3	4	5	6	7	8	9	10	AVG
Time (ms) with monitoring	10.92	10.65	9.57	10.93	11.05	10.61	10.75	9.86	11.44	11.00	10.68
Time (ms) without monitoring	0.41	0.34	0.33	0.33	0.48	0.39	0.33	0.47	0.44	0.40	0.39
Time (ms)	Time with monitoring-time without monitoring										10.29

(2) 一般而言,监控的程序元素越多,植入的探针也越多,运行时的附加开销也就越大.被监控的程序元素本身运行时间越小,附加运行开销所占总运行时间的比例越大.例如,假设对函数 f 实施监控,f 运行一次的时间为 t^f,对 f 实施一次监控所需要的时间为 m,则相对未监控情况,监控所带来的额外开销百分比为

$$(m/(t^f+m))\times 100\%.$$

t^f 越小,额外开销的百分比越高.在情况(1)的第 1 个测试用例中,m=28.38,t^f=583.61,$(m/(t^f+m))\times 100\%$=4.64%;在第 2 个测试用例中,m=10.29,t^f=0.39,$(m/(t^f+m))\times 100\%$=96.35%.

(3) 探针获取监控信息后,传输给发布/订阅服务.这部分处理可以在不同的处理器上完成,并且处理逻辑和软件的运行逻辑没有关联性,所以这部分处理会带来额外的平台开销(例如,需要为系统增加一台专用的计算机),但是不会给程序功能逻辑的运行带来额外开销.

为了便于用户控制运行时开销,在工程实现上,通过配置的方式允许用户关闭某类探针.所以,即使探针已经植入运行程序,也可以通过配置使其运行时不工作,从而为实施监控提供更好的灵活性.

通过选择合理的监控元素以及采取适当的配置策略,在已经部署的 4 个软件中,其监控带来的运行时额外开销可以控制在 10%以内,对系统的整体运行不会造成影响.

5 相关工作比较

从作用上来讲,监控主要有正确性保证和系统调试、调优及可信演化两个方面.

传统上,监控主要用在软件的正确性保证方面,通过实时监控软件的行为,基于某种形式化规约判断行为的正确性,这些系统包括 MaC(monitoring and checking)、面向监控编程(monitoring-oriented programming,简称 MOP)、契约式设计(design by contract,简称 DBC)等.MaC[12,13]依据形式化的需求规约,通过运行时对目标程序进行监视和检查,以保证程序执行的正确性.MOP[14,15]提供了一个通用框架,将系统的需求用形式化规约方法表示,并插入在程序的特定位置,用于实现实时验证程序状态变迁的需求.DBC[9,16]允许开发人员在程序中插入断言,对函数调用的前置条件、后验条件和循环不变式等进行安全和可靠性监控.它们主要应用于集中式系统或小规模的分布式系统.

随着大型分布式软件系统,特别是云计算系统的广泛应用,监控成为这些系统调试、调优以及可信演化的重要手段,监控设施也成为构造这类系统的必不可少的基础设施.X-Trace[17]和 Pip[4]都是利用监控对分布式软件系统进行调试和调优的设施.本文的工作属于这种类型.

从实现方式上来讲,我们将监控分为两类:内嵌式和伴随式.

内嵌式是紧耦合的同步方式.在这种方式下,系统运行、监控和检查是绑定的:系统运行的同时进行监控,监控的同时进行检查,只有检查无故障,系统才可以继续运行,否则立即停止系统运行进行故障排除.这 3 个环节在运行时间上是串行的,在部署空间上是一体的,在控制逻辑上是同步的.这种结构的好处是对系统运行故障的及

时发现和处理;但是对大型分布式软件系统来讲,这种结构的缺点也非常明显:一是对分布式软件系统进行调试和调优需要全局信息,每一个运行的节点都获得全局信息显然代价太高;二是紧耦合的结构对系统性能和可靠性的干扰过大,进行运行时检查的时候实质上是暂停了系统的运行;三是这种结构的检查逻辑必须是在开发前就确定的,运行时是和运行逻辑绑定的,这对具有动态演化特征的分布式软件系统来讲几乎是不可能的.因此,这种结构多用在集中式系统或者小规模的分布式软件系统中,而且多数是应用在控制系统等具有明确的可形式化描述的行为规范的系统中.MaC,MOP 和 DBC 均属于内嵌式.MaC 和 MOP 可以独立于业务代码表达检查约束逻辑,因此对业务逻辑代码的干扰较小;DBC 则需要手工在源程序中插入检查逻辑,因此对系统的业务逻辑代码干扰较大.

伴随式是松耦合的异步方式.在这种方式下,系统运行、监控和检查是分离的:在运行时间上是并行的,部署空间上是分布的,逻辑控制上是异步的.这种方式的缺点是分析滞后于运行,难以实时阻断系统故障.但这种结构对系统性能和可靠性干扰少,运行、检查和分析互相独立,可扩展性强,比较适合于大规模的分布式软件系统.

X-Trace 是一个通用的对分布式程序进行路径跟踪的平台.它通过还原请求在多个网络和设备层的路径来对分布式应用进行调试.X-Trace 需要修改各层相关的协议实现或者网络设备,并通过 API 的方式允许在应用程序中插入跟踪语句,因此其对源代码的影响较大,部署代价相对较高.X-Trace 通过在各层协议消息(例如,IP 和 TCP 可选项、HTTP 协议头)中插入元数据实现对路径的真实跟踪,因此会在一定程度上带来应用吞吐量下降(文献中的实验结果表明,吞吐量降低了 15%).执行路径的收集、汇聚和还原分析则和业务逻辑的运行是分离的,因此其属于伴随式的一种.X-Trace 的目标是通过还原真实执行路径来对分布式软件系统进行调试,由于其 API 中允许携带一定的应用相关扩展信息,因此有一定的监控业务逻辑相关性.X-Trace 可以跨域进行执行路径的收集和汇聚,因此对分布式软件系统的支持较好.

Pip 结合了路径跟踪和期望检查来发现分布式软件系统的结构 Bug 和性能瓶颈.Pip 通过监控执行路径获得软件的真实行为,进而与用户期望进行比较.Pip 使用声明式语言表达对程序通信结构、资源以及时间消耗的期望,因此对业务逻辑代码干扰较小,而业务逻辑监控的针对性较强.在 Pip 中,业务系统的运行、执行路径获取以及行为约束检查是互相分离的,因此它也属于伴随式的一种,对分布式系统的支持较强,其部署代价也较小.

Pinpoint 通过修改应用的底层运行时库或设施(例如 J2EE 服务器)来获取应用请求和失败信息,从而判断故障和构件的相关性信息.Pinpoint 不需要修改应用,因此对应用代码无影响,部署时对业务系统干扰较小,但是对请求的监控针对性较弱.另外,Pinpoint 尚不能支持大型分布式系统,其实验结果表明对性能的影响小于 10%.

随着云计算平台的普及以及虚拟机的广泛应用,基于虚拟机对应用进行监控成为一种重要的手段[3].在这种方式下,由于软件在虚拟机中执行,因此通过虚拟机可以监控到软件的执行行为.但是,通过虚拟机也只是能够监控一个节点的运行,要对整个分布式软件系统进行调试、调优和可信演化仍然需要伴随的采集、汇聚和处理设施.基于虚拟机的监控必然带来虚拟机的执行开销,对性能和可靠性影响较大.由于将软件看作黑盒子,所以对代码没有干扰,部署代价也较低,但是这种监控没有业务逻辑的针对性.

本文所采用的 MERF 伴随式方法强调的是为程序员和管理者提供监控系统的方法和设施,与已有系统相比有两个显著区别.一个区别是监控内容的可定制性.X-Trace,Pip 和 Pinpoint 监控执行路径,DBC 主要监控关系表达式,MERF 则可以细粒度地监控程序的各种执行元素和资源消耗,程序员可以根据不同的需要定制监控内容.另一个区别是监控信息处理的可定制性.MERF 不假定采用的对监控信息进行分析处理的具体手段(例如基于执行路径),程序员可以根据需要定制监控信息处理单元.因此,MERF 在可定制性上具有特色,致力于为程序员和管理者提供监控平台和基础设施,为软件调试、调优和可信演化提供支撑,而不限于基于某种具体手段监控和诊断分布式系统.MERF 使用 MPL 语言描述监控需求,使用编译器进行探针植入,因此在开发时对业务逻辑代码的干扰很小.MPL 不仅能够表达对资源消耗的监控,而且能够表达对程序元素的监控.相对于路径跟踪来说,这种监控单元粒度更细,监控信息具有明确的语义信息,因此业务逻辑的监控针对性较强.部署时,只有监控探针和监控 Agent 随业务系统一起部署,监控系统的其他部分独立部署,因此部署时对业务系统的干扰较小.

表 7 对各种典型监控系统进行了概括比较.其中,Q 表示内嵌式;B 表示伴随式;C 表示正确性保证;D 表示系

统调试、调优和可信演化.

Table 7 Comparison of different monitoring systems

表 7 监控系统比较

System	Type	Function	Support for large distributed system	Interference			Monitoring pertinence to business logic
				Interference to performance and reliability at runtime	Interference to business logic code at development	Interference to business system at deployment	
MaC	Q	C	Weak	Great	Little	Medium	Strong
MOP	Q	C	Weak	Great	Little	Medium	Strong
DBC	Q	C	Weak	Great	Great	Little	Strong
X-Trace	B	D	Medium	Medium	Great	Great	Medium
Pip	B	D	Strong	Little	Little	Medium	Strong
Pinpoint	B	D	Weak	Little	No	Little	No
Monitoring based on VM	B	D	Medium	Great	No	Little	No
MERF	B	D	Strong	Little	Little	Medium	Strong

6 结论和进一步的工作

针对开放的网络环境中分布式软件的调试、调优和可信演化等问题,本文提出了将监控纳入软件构造和运行时的方法,实现了 BlackBoxJ/BlackBoxC++系统.案例分析和实验结果表明,系统达到了如下目标:

(1) 通过语言和工具的支持,以较低的软件生产成本使软件系统具有可监控能力.对分布式软件,特别是开放环境下的大规模分布式应用,监控已经成为提高其可信的重要手段之一.与手工编写监控探针代码相比,根据 MPL 代码自动产生探针并进行植入,降低了软件生产成本.

(2) 在为软件增加可监控能力的同时,保持了较好的代码可维护性.相比手工编写监控代码,工具使得开发人员只需维护一个 MPL 代码文件,减少了同一文件中出现多个方面代码的问题,使得代码修改等维护代价显著降低.

(3) 通过为软件系统增加可监控能力,提高系统可用性.MERF 可辅助设计者进行测试和故障诊断,特别是针对那些在实验室中难以模拟的场景,MERF 能够在软件发生故障时记录场景信息,做到发现问题及时定位故障,及时进行故障恢复,减少了故障诊断时间,特别是减少手工调试时间,提高了系统的可用性.

本文提出的运行框架和工具能够监控的基本程序元素包括变量、函数和线程.下一步工作主要包括:对更细粒度的程序单元实施监控,比如,循环或者程序的某条语句;对更高粒度的软件单元实施监控,比如以服务为粒度进行监控.

致谢 在此,我们向对本文的工作给予支持和建议的同行,尤其是向国防科学技术大学计算机学院 613 教研室软件监控课题组的朱俊、朱锐、王晓兵、李小玲、殷越鹏表示感谢.

References:

[1] Aguilera MK, Mogul JC, Wiener JL, Reynolds R. Muthitacharoen A. Performance debugging for distributed systems of black boxes. In: Proc. of the 19th ACM Symp. on Operating Systems Principles. New York: ACM Press, 2003. 74−89. [doi: 10.1145/1165389.945454]

[2] Gao J, Jiang GF, Chen HF, Han JW. Modeling probabilistic measurement correlations for problem determination in large-scale distributed systems. In: Proc. of the 29th IEEE Int'l Conf. on Distributed Computing Systems. Washington: IEEE Computer Society, 2009. 623−630. [doi: 10.1109/ICDCS.2009.56]

[3] Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M. A view of cloud computing. Communications of the ACM, 2010,53(4):50−58. [doi: 10.1145/1721654.1721672]

[4] Reynolds P, Killian C, Wiener JL, Mogul JC, Shah MA, Vahdat A. Pip: Detecting the unexpected in distributed systems. In: Proc. of the 3rd Symp. on Networked Systems Design and Implementation. California: USENIX, 2006. 115−128.

[5] Barroso LA, Hölzle U. The datacenter as a computer: An introduction to the design of warehouse-scale machines. In: Proc. of the Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2009. 1−108. [doi: 10.2200/S00193ED1V01Y200905CAC006]

[6] Roberts CA. Flight recorders and aircraft safety. In: Proc. of the Annual Conf. Houston: United States, 1976. 265−269. [doi: 10.1145/800191.805594]

[7] Elrad T, Filman RE, Bader A. Aspect-Oriented programming: Introduction. Communications of the ACM, 2001,44(10):29−32.

[8] Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic Internet services. In: Proc. of the DSN. 2002. 595−604. [doi: 10.1109/DSN.2002.1029005]

[9] Meyer B. Applying "Design by Contract". IEEE Computer, 1992,25(10):40−51. [doi: 10.1109/2.161279]

[10] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In: Proc. of the European Conf. on Object-Oriented Programming (ECOOP 2001). LNCS 2072, Berlin: Springer-Verlag, 2001.

[11] Spinczyk O, Gal A, Schroder-Preikschat W. AspectC++: An aspect-oriented extension to the C++ programming language. In: Proc. of the 40th Int'l Conf. on Tools Pacific Objects for Internet, Mobile, and Embedded Applications, Vol.10. 2002. 53−60.

[12] Kim M, Viswanathan M, Kannan S, Lee I, Sokolsky O. Java-MaC: A run-time assurance approach for Java programs. Formal Methods in System Design, 2004,24(2):129−155. [doi: 10.1023/B:FORM.0000017719.43755.7c]

[13] Sammapun U, Lee I, Sokolsky O. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In: Proc. of the 11th IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA 2005). Hong Kong, 2005. 147−153. [doi: 10.1109/RTCSA.2005.84]

[14] Chen F, Roşu G. Towards monitoring-oriented programming: A paradigm combining specification and implementation. In: Proc. of the 3rd Int'l Workshop on Runtime Verification (RV 2003). Vol. 89 of ENTCS, Colorado: Elsevier Science, 2003. 106−125. [doi: 10.1016/S1571-0661(04)81045-4]

[15] Chen F, Roşu G. MOP: An efficient and generic runtime verification framework. In: Proc. of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA). 2007. 569−588. [doi: 10.1145/1297105.1297069]

[16] Cheon Y, Leavens G, Sitaraman M, Edwards S. Model variables: Cleanly supporting abstraction in design by contract. Software: Practice and Experience, 2005,35(6):583−599. [doi: 10.1002/spe.v35:6]

[17] Fonseca R, Porter G, Katz RH, Shenker S, Stoica I. X-Trace: A pervasive network tracing framework. In: Proc. of 4th USENIX Symp. on Networked Systems Design & Implementation. 2007. 271−284.

附录 1. MPL 的 BNF 表示

MPL 由监视语句组成,用于描述监视资源消耗和程序元素的需求.MPL 可监视软件运行时的计算资源、存储资源、系统句柄以及网络连接资源的消耗,由表 8 中的 *resource_monitor_statement* 定义.

Table 8 BNF for the MPL

表 8 MPL 的 BNF 表示

monitor_statement::=*resource_monitor_statement*|*program_element_monitor_statement*;

resource_monitor_statement::="*Monitor_Resource*""("*resource_attr*{"|"*resource_attr*}")";
resource_attr::="*CPU_TIME*"|"*MEM_USAGE*"|"*START_TIME*"|"*HANDLE_COUNT*"|"*NET_CONNECTION*";

program_element_monitor_statement::=*function_monitor_statement*|*thread_monitor_statement*|*variable_monitor_statement*;

function_monitor_statement::="*Monitor_Function*""("*function_name*","*function_attr*{"|"*function_attr*}")";
function_name::=*string*;
function_attr::="*PERFORMANCE*"|"*ENTER_TIME*"|"*LEAVE_TIME*"|"*PARAMETER*"|"*THREAD_CONTEXT*";

thread_monitor_statement::="*Monitor_Thread*""("*thread_name*","*thread_attr*{"|"*thread_attr*}")";
thread_name::=*string*;
thread_attr::="*THREAD_ID*"|"*CREATE_TIME*"|"*START_TIME*"|"*END_TIME*";

variable_monitor_statement::="*Monitor_Variable*""("*variable_name*","*variable_attr*{"|"*variable_attr*}")";
variable_name::=*string*;
variable_attr::="*CHANGE_TIME*"|"*BEFORE_CHANGE_VALUE*"|"*AFTER_CHANGE_VALUE*"|
"*THREAD_CONTEXT*"|"*FUNCTION_CONTEXT*";

MPL 可监视的交互活动包括包括函数、通信与并发线程以及和交互相关的变量,分别由表 8 中的 *function_monitor_statement*,*thread_monitor_statement* 和 *variable_monitor_statement* 定义.每条 MPL 监视语句由监控关键字、监控元素名称以及监控属性组成,其中,监控关键字定义了监控类型,包括函数、线程和变量,分别由 *Monitor_Resource*,*Monitor_Function*,*Montor_Thread* 和 *Monitor_Variable* 表示;监控元素名称定义了具体的监控对象,如 *function_name*,*thread_name* 等;监控属性则定义了针对某监控对象需要监控的属性,如线程 ID、线程创建时间等,在表 8 中分别由 *resource_attr*,*function_attr*,*thread_attr* 以及 *variable_attr* 定义.这几部分结合,最终准确描述监控需求.

附录 2. 监控类型和监控元素属性到切入点和通知关键字的映射

Table 9 Mapping from monitoring type and attribution of monitoring element to key words of pointcut and advice

表 9 监控类型和监控元素属性到切入点和通知关键字的映射

Monitoring type (M_t)	Monitoring attribution (M_a)	Pointcut key word (K_p)	Advice key word (K_a)	Processing action (A_p)
Monitor_Resource	*CPU_TIME*	Execution	Before	Get the CPU usage
	MEM_USAGE	Execution	Before	Get the memory usage
	START_TIME	Execution	Before	Record the start time
	HANDLE_COUNT	Execution	Before	Get the handle count
	NET_CONNECTION	Execution	Before	Get the number of net connection
Monitor_Function	*PERFORMANCE*	Execution	Before	Record the function's performance
	ENTER_TIME	Execution	Before	Record the function's enter time
	LEAVE_TIME	Execution	After	Get the function's leave time
	PARAMETER	Call	Before	Acquire the function's parameter
	THREAD_CONTEXT	Execution	Before	Acquire the function's thread context
Monitor_Thread	*THREAD_ID*	Within, execution	Before	Record the thread's ID
	CREATE_TIME	Within, call	Before	Record the thread's create time
	START_TIME	Within, execution	Before	Record the thread's start time
	END_TIME	Within, execution	After	Record the thread's end time
Monitor_Variable	*CHANGE_TIME*	Set	After	Record the variable's change tim
	BEFORE_CHANGE_VALUE	Set	Before	Get the value before change
	AFTER_CHANGE_VALUE	Set	After	Get the value after change
	THREAD_CONTEXT	Set	After	Acquire the variable's thread context
	FUNCTION_CONTEXT	Set	After	Acquire the variable's function context

刘东红(1966－),女,陕西澄城人,高级工程师,主要研究领域为分布对象计算,可信计算,系统集成技术.

王怀民(1962－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布式计算,虚拟计算环境.

郭长国(1973－),男,博士,副研究员,CCF 会员,主要研究领域为分布计算,面向方面编程.

王涛(1984－),男,博士生,CCF 学生会员,主要研究领域为分布式计算,可信软件.