第42卷 第12期
2015年12月
计 算 机 科 学
Computer Science
Vol. 42 No. 12
Dec 2015

一种面向 PaaS 的实例级应用动态更新技术

张 婕 曹 春 余东亮
(南京大学计算机科学与技术系 南京 210023)
(南京大学计算机软件新技术国家重点实验室 南京 210023)

摘 要 云计算是当前信息技术的重要技术领域,而平台即服务(PaaS)已成为业界研究的热点之一。PaaS 平台为用户提供高可用、高可扩展的应用开发、部署和运行环境。然而当部署到云端的应用需要不断更新以修复错误、增加功能时,当前主流 PaaS 平台却因缺乏对应用在线更新的有效支持而削弱了其自身的高可用特性。为解决该问题,提出一个面向 PaaS 平台的动态更新技术框架。基于现有软件动态更新技术的研究,通过对 PaaS 平台中应用的事务管理、动态依赖管理、版本管理等机制的扩展,为 PaaS 平台提供运行时实例级的应用动态更新支撑,并在 Cloud Foundry 上进行实现和实验,结果证明了该动态更新技术的有效性。

关键词 PaaS,动态更新,Cloud Foundry

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.12.014

Dynamic Updating Technology for Application Instances on PaaS

ZHANG Jie CAO Chun YU Dong-liang
(Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China)

Abstract PaaS (Platform-as-a-Service) is one of the key services in cloud computing, which provides high availability and scalable development and runtime environment for applications. However, when the applications running on a PaaS platform need to be updated, current PaaS platforms lose their high availability due to the lack of effective support for dynamic updating. To solve the problem, based on current research on software dynamic updating technology, we introduced a PaaS oriented dynamic software updating framework. With extensions of transaction management, dynamic dependence management and version management, we realized instance-level dynamic update for applications on PaaS platforms. We implemented the technical framework on Cloud Foundry to demonstrate the effectiveness of our technology.

Keywords PaaS, Dynamic updating, Cloud Foundry

1 引言

云计算服务包括基础设施即服务(IaaS)、平台即服务(PaaS)、软件即服务(SaaS)3 个层次[1]。其中 PaaS 为用户提供一个可用于开发、部署和维护的综合环境,同时提供可扩展性、可靠性和安全性[2]。PaaS 平台为保障应用的高可用性,通常会扩展多个实例运行,并达到负载均衡的目的。随着应用的不断演化,部署在 PaaS 层的应用需要更新来满足需求。主流 PaaS 平台通常采用离线更新:首先删除已部署的应用,然后部署新版本应用。在此期间,为保证系统的安全性,直接重新部署应用需要等待所有的请求都结束,否则会导致更新过程中请求失效和应用状态丢失。该过程导致 PaaS 在一定程度上失去高可用性[3]。为解决上述问题,我们提出一个面向 PaaS 平台的动态更新技术框架,该框架通过扩展 PaaS 平台来完成面向动态更新的实例管理,主要技术包括扩展应用部署和执行构件、引入事务和动态依赖管理、扩展应用请求的路由规则等。该技术框架支持对已部署的应用进行以实例为单位的动态更新,并保证更新时系统的一致性。

2 问题分析

2.1 主流 PaaS 平台对更新的支持

现有的主流 PaaS 平台包括 Heroku、GAE、CloudFoundry 等,它们仅提供离线式的更新支持,且不考虑更新的安全性和应用系统状态的一致性。Heroku 的架构包括反向代理服务器、动态路由处理层、动态网格层(Dyno Grid)等。Heroku 在动态网格层,使用轻量的 Unix 容器 dyno 运行应用。当用户部署新版本应用时,Heroku 会 kill 当前所有运行的 dyno 并生成新 dyno。但在替换过程中旧请求将被忽视,新请求直到 dyno 启动完成后才被处理。

GAE 架构的前端包括 Front End、Static Files、应用服务器等。Front End 用于负载均衡和请求转发,具体请求由应用服务器处理。GAE 支持应用部署多个版本,默认版本为当前

到稿日期:2015-03-03 返修日期:2015-06-27 本文受国家重点基础研究发展计划(973 计划)(2015CB352202),国家基金中美合作项目(61361120097),国家自然科学青年基金项目(61100037)资助。

张 婕(1990－),女,硕士生,主要研究方向为软件工程、软件中间件,E-mail:ndzj981479673@gmail.com;曹 春(1978－),男,博士,副教授,主要研究方向为软件架构、中间件、软件动态更新技术等;余东亮(1990－),男,硕士生,主要研究方向为软件工程、软件中间件。

活跃版本。更新应用时，用户需要提供待更新应用的版本配置。为避免应用访问正在部署的版本，GAE需要用户保证新版本应用不是当前活跃版本。GAE不考虑应用何时能更新，而由用户决定何时更新，且不处理被依赖应用在更新过程中的访问失效问题。

CloudFoundry(简称CF)的架构主要包括路由器、控制器、执行器等组件。CF可以通过扩展运行在其上的应用平台(如Tomcat[1])来支持应用的动态更新，但这意味着对每个应用平台都要进行扩展，且可能出现不同应用平台间不兼容的问题。CF在PaaS平台层并不支持应用的动态更新，如果有新版本应用需要部署，需要用户手动删除当前正在运行的应用，再重新部署新版本应用。

2.2 多实例应用动态更新一致性问题

如前所述，目前的PaaS平台通常提供离线式的应用更新支持，在这一方式下，如果待更新的应用未被其它应用依赖，则更新时系统的正确性不会受到影响；但当存在依赖关系的应用被更新时，则会产生系统状态不一致的问题。

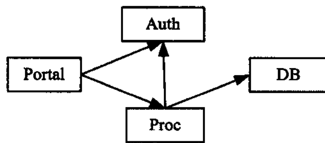

图1 具有依赖关系的应用组件[3]

图1是文献[3]中的实验用例，访问入口为Portal应用，业务流程如下：Portal先从Auth处获取一个token，然后以该token为参数，调用Proc。Proc先向Auth验证token的合法性，验证通过后，Proc应用向DB请求数据。图2是系统的访问时序图。这里假设待更新的应用为Auth，如果更新发生在A点或D点，则不会产生系统不一致问题。但当更新发生在B或C点时，可能会出现Portal使用旧版本Auth，Proc使用新版本Auth，从而导致系统不一致。因此，目前很多研究工作致力于对动态更新的研究：马晓星等[3]提出一致性算法Version-Consistency(简称VC)；Kramer和Magee提出Quiescence算法[4]；Vandewoude提出Tranquility算法[5]。这些算法可以在不同灵活度下保证系统一致性。

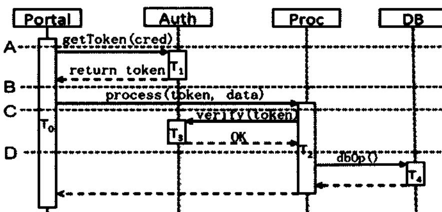

图2 具有依赖关系的应用的访问时序图

如果待更新应用运行在PaaS平台上，则需要解决两个主要问题。首先需要解决更新安全点问题，这个由上面提到的一致性算法来保证。本文选择VC算法进行实现。该算法引入事务概念，并对事务运行时不断变化的动态依赖进行形式化定义，用它来检测更新安全点。其中事务是有限时间内完成的一组操作序列，图2中T_0表示的就是运行的事务。由外部客户端发起的事务称为根事务，而由其它事务发起的事务则称为子事务。图2中T_1是T_0的子事务，T_1的根事务为T_0。动态依赖是指运行过程中应用间的依赖关系，系统运行到图2中A点时，Portal应用依赖Auth和Proc。

其次，在对PaaS中运行多个实例的应用进行更新时，还需保证更新效率。接收到更新请求后，先对空闲的实例进行更新，而被使用的实例延后更新。但是如果更新发生在C点，而此时Auth应用的实例1未更新，实例2已更新，则可能会出现Portal使用的是Auth的实例1(旧版本)，而Proc使用的是Auth的实例2(新版本)，导致系统状态不一致。因此更新多实例应用时，需保证同一个根事务下对同一个应用的请求转发到同一个应用实例。现有PaaS平台下，对于同一个根事务下的请求，路由器将其随机转发给一个实例，而更新过程中可能出现上述请求响应的版本不一致问题。为解决该问题，需扩展路由器的路由规则，添加应用层的路由策略。

3 PaaS扩展设计框架

当前的PaaS平台具有如图3所示的一般性架构形式，包括部署工具CLI、路由器Router和执行构件Executor。

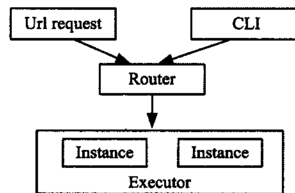

图3 PaaS通用架构

我们针对该架构提出一种应用实例级的动态更新方案，该方案的基本目标是：(1)不破坏现有PaaS平台架构，通过添加更新管理器、事务管理器、依赖维护模块，来扩展路由功能等，支持应用实例级动态更新；(2)保证动态更新时系统一致性。PaaS平台扩展方案如图4所示。

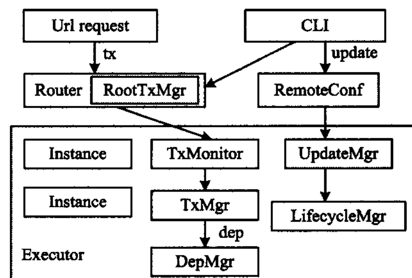

图4 PaaS组件扩展图

VC算法需维护运行时的事务和动态依赖，因此PaaS的执行构件需为每个应用实例维护事务管理器(TxMgr)、动态依赖管理器(DepMgr)和更新管理器(UpdateMgr)。当应用发生远程调用时，需要路由器维护全局事务关系和调用信息；此时事务状态也会变化，通过该变化可以知道事务的依赖是否发生变化。动态依赖管理器根据事务的变化，建立构件间的依赖关系。更新管理器根据更新算法及应用间的动态依赖，寻找更新安全点并执行更新操作。

3.1 事务管理器

事务管理器是维护事务状态的设施，其依赖于应用和路由器提供的事务状态，执行事务状态的转换，完成对实例的事务管理。事务管理器包括监听器和事务生命周期管理。事务

[1] http://tomcat.apache.org

生命周期如图 5 所示。

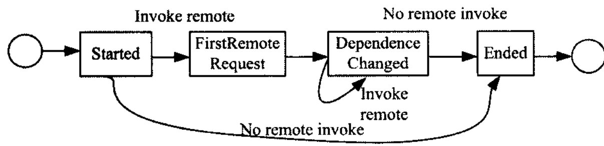

图 5 事务状态转换

通过分析程序运行状态，可以获取事务状态自动机[6]。应用从事务开始时即触发状态自动机，对远程应用的调用以及事务的结束都会改变事务状态。对于扩展多个实例的应用，执行器为每个实例维护一个局部的事务管理器。

3.2 应用生命周期管理

生命周期管理器是维护运行时实例状态的设施，它接收更新管理器的请求，执行实例状态的切换，最终辅助更新管理器完成更新过程。PaaS 平台上应用的生命周期包括启动、运行、停止等，这里我们主要对应用的运行态进行扩展，如图 6 所示。由于收集动态依赖需要耗费时间，因此我们只在接收到更新请求后按需建立动态依赖，所以引入 Ondemand 状态[3]；准备完成后，进入依赖建立过程，并到达依赖收集状态(Valid)；之后判断应用是否可以更新，如果可更新，进入可更新状态(Free)；更新开始后应用进入正在更新状态(Updating)；更新完成后应用进入更新完成状态(Updated)。

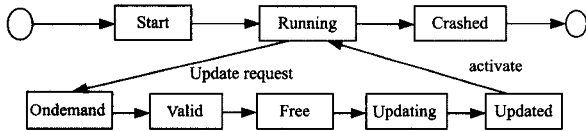

图 6 应用生命周期

3.3 应用更新管理器

应用更新管理器是提供实例更新接口的设施，其依赖于生命周期管理器所提供的实例生命状态，执行依赖建立操作和更新安全点检查操作，最终完成实例的更新过程。更新管理器接收到更新请求后首先判断应用实例的状态，如果不是 Valid 状态，则开始准备建立动态依赖，此时实例进入 Ondemand 状态。依赖建立的准备过程完成后应用实例进入 Valid 状态，开始建立动态依赖并检测该应用实例能否进行动态更新。当应用实例满足可更新的约束条件后，对该应用实例进行更新。对多实例应用按实例进行更新，只有第一个被更新的应用实例需要部署应用，其余应用实例在更新时只需扩展应用实例个数，启动新的实例。

3.4 扩展的路由器

路由器原有的策略主要是转发请求，这里对路由策略和行为进行扩展，添加应用层路由策略。(1)当存在多个实例时，路由器保证属于同一个根事务的请求被转发给同一个应用实例。我们在请求头中加入事务信息，路由器维护根事务与转发实例的对应关系。(2)路由器记录应用实例与远程请求的对应关系。当应用发起远程访问时，会将远程请求信息存储在请求头中，路由器需要记录远程请求信息。事务监听器需要根据该信息，判断需要开启的事务是根事务还是子事务。(3)路由器在更新过程中拦截请求，并判断是阻塞请求还是转发请求。

为验证本文提出的动态更新技术的正确性，我们在 CF 平台上进行实验。通过扩展其执行组件、路由组件、添加系统管理模块等，实现 CF 上应用实例级的动态更新。

4 Cloud Foundry 扩展实现

CF 中 Router 负责转发用户的访问请求以及平台的管理请求，Cloud Controller 负责响应用户请求，DEA 用于部署和运行应用，同时管理应用的生命周期。UAA 用于登录验证，Health Manager 负责检查应用的运行状态。当部署的应用需要更新时，CF 并不提供平台层的动态更新支持，而需要用户先删除已部署应用，再部署新版本应用。CF 的扩展主要包括在执行构件 DEA 上添加事务管理器、应用生命周期管理器、更新管理器；扩展路由器，添加路由规则，并维护全局事务信息；扩展 CLI 等。扩展后的 CF 架构如图 7 所示。

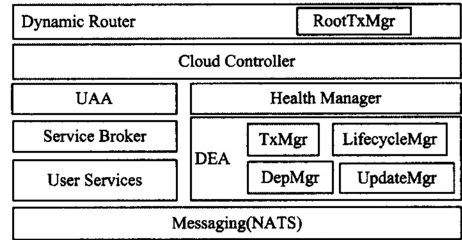

图 7 扩展的 Cloud Foundry

4.1 事务管理器的实现

应用开发人员需要在事务方法前添加@Transaction 注解，扩展后的 CF 在部署应用前可使用 DDET[6] 对添加注解的方法体进行控制流分析，通过分析方法的对外调用信息及插入相应指令来通知实例事务运行时的依赖变化。应用部署成功后启动 Messaging Server，用于接收应用发送的事务状态和 Router 发送的事务信息。应用在运行事务方法时向 Messaging Server 发送事务状态，由 Messaging Server 将事务状态转发给事务监听器。事务监听器向路由器验证该事务是否是根事务，并相应开启根事务或者开启子事务。

4.2 生命周期管理器的实现

生命周期管理器维护应用的状态(status)，默认为 Normal。该管理器在应用实例运行成功后创建。在更新过程中更新管理器，通知生命周期管理器触发状态转换方法，将应用实例转变为相应的状态。例如在准备建立构件依赖时，更新管理器通知生命周期管理器触发 ToOndemand 方法，status 转为 Ondemand。应用实例会定期向路由器注册自己，在更新过程中应用实例状态的变化将触发应用实例的注册事件，即 register_instance 方法，在路由器中记录应用实例的状态和旧版本请求的根事务。

4.3 更新管理器的实现

应用实例启动成功后创建更新管理器，更新管理器包括动态依赖管理器、更新算法以及更新执行器等。更新管理器中定义 3 种消息类型：DepMsg 、RemoteConfMsg、OndemandMsg。MessagingServer 接收到更新消息后通知更新管理器，更新管理器根据消息的类型执行相应的操作：如果是 RemoteConfMsg，则开始进行依赖建立过程，同时通知依赖当前实例的应用(parent)也进行依赖建立过程，并等待回复；如果是 OndemandMsg，说明是接收到当前实例所依赖的应用(child)发来的依赖建立消息，同样开始依赖建立过程，依赖建立完成后通知 child；如果是 DepMsg，则交由依赖管理器维护动态依赖，依赖管理器将接收到的消息进行解析后，调用更新

算法完成依赖管理。更新算法负责动态依赖边的创建和删除，并判断何时进行更新。

当扩展多个实例的应用要求更新时，第一个接收到请求的应用实例会进行单实例应用的部署，执行“cf push”命令。之后的实例接收到更新请求时，只增加新版本的应用实例数，并启动新实例，执行“cf increase”命令。扩展后的 DEA 如图 8 所示。

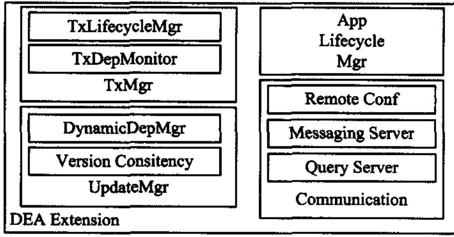

图 8 DEA 扩展图

4.4 路由器扩展的实现

我们在路由器中维护两个映射表(map)：txMap 用于记录根事务和转发实例的对应关系，callMap 用于记录应用实例和远程访问请求的对应关系。路由器对请求进行解析，如果请求头有事务信息，则在 txMap 记录该信息，key 为实例和根事务 ID，value 为实际转发的实例。当同一个根事务第二次对应用进行请求时，路由器根据 txMap 中的记录，将请求转给相同的应用实例。2.1 节的实验用例中，由 Portal 应用发起的根事务 T_0 对应的子事务 T_1、T_3 上的 Auth 请求都属于根事务 T_0，因此请求将转发给同一个 Auth 实例。此外，路由器以应用实例 ID 为 key，以远程请求信息为 value，存储一个 callMap，用于记录应用实例和远程访问请求的对应关系。应用发起远程请求时，会在请求头中加入远程调用信息。事务监听器在接收到事务信息时，会向路由器查询请求是否由其它事务发起；路由器查询 callMap 记录，如果由根事务发起，返回根事务信息和调用信息。路由器还增加拦截器，默认拦截策略是多版本共存策略。更新过程中请求到达时，路由器查询 txMap，如果请求的根事务在 txMap 中，将请求转发给旧版本应用实例；否则转发给新版本实例。

5 实验结果与评价

5.1 实验过程

根据图 1 的实例应用和表 1 的实验环境，我们部署了 4 个 Tuscany[1] 应用进行实验。首先在 CF 平台上部署 4 个应用：Portal、DB、Proc、Auth，并扩展多个 Auth 实例。用户首先发起对 Portal 的连续访问，并在访问过程中请求更新 Auth。

表 1 实验配置环境

操作系统	Ubuntu 12.04
虚拟机版本	Virtual Box-4.3，分配 5GB 内存，双核 CPU
CloudFoundry 版本	Release-149
应用版本	Tuscany 2.0

5.2 实验结果与评价

具体的实验运行过程如下：对 Portal 应用进行连续请求，在 15s 内每隔 100ms 发送一次访问请求，并在第 15s 发送更新 Auth 请求，之后继续发送访问请求，请求的时间间隔为 100ms。

```
index:31
    http://portal.192.168.12.34.xip.io/PaPaComponent/PaPa?method=doPaPa
    {"id":1,"result":"Portal.Proc.Auth.v1 Auth.v1 World"}
index:32
    http://portal.192.168.12.34.xip.io/PaPaComponent/PaPa?method=doPaPa
    {"id":1,"result":"Portal.Proc.Auth.v1 Auth.v1 World"}
index:33
    http://portal.192.168.12.34.xip.io/PaPaComponent/PaPa?method=doPaPa
    {"id":1,"result":"Portal.Proc.Auth.v2 Auth.v2 World"}
index:34
    http://portal.192.168.12.34.xip.io/PaPaComponent/PaPa?method=doPaPa
    {"id":1,"result":"Portal.Proc.Auth.v2 Auth.v2 World"}
```

图 9 在请求过程中更新 Auth 应用

通过打印的请求日志(见图 9)可以看到，Auth 应用被正确更新。日志中返回 Auth 两次调用结果，第一个是 Portal 对 Auth 应用的调用，返回响应的 Auth 实例的版本号；第二个是 Proc 对 Auth 的调用，同样返回响应的 Auth 实例的版本号。可以看到，在更新前后同一个根事务下请求的 Auth 版本总是相同的。更新在第 33 次请求时已经完成，之后的请求都是使用新版本的应用进行响应。第一个实例完成更新的时间为 100s，所有实例完成更新的时间为 111s。更新前请求的平均响应时间为 2.4s；更新过程中请求平均响应时间为 5.85s；更新后平均响应时间为 2.53s。

为进一步证明更新的正确性，我们对 Auth 应用扩展不同实例个数进行实验，更新结果如表 2 所列。运行过程同上：在 15s 内，每隔 100ms 发送一次 Portal 访问请求，并在第 15s 发送更新 Auth 请求，之后继续发送访问请求，请求的间隔为 100ms。通过对 2.2 节中案例进行实验，证明了在线更新被依赖的多实例应用的正确性。

表 2 不同实例数更新实验结果

实例个数	首个实例更新时间(s)	所有实例更新时间(s)
1	86	86
2	100	111
3	101	118
4	105	165
5	124	201

结束语 本文的创新点在于提出了一种基于 PaaS 的应用实例级动态更新技术。通过该技术提供应用的在线更新支持，并保证系统的一致性，增强了 PaaS 的高可用性。我们通过在 CF 平台上进行实验，展示了扩展后的 CF 对应用实例级动态更新的支持，保证了动态更新过程的安全性。下一步将进一步完善实验设计，加入对比实验，并将该动态更新封装成 CF 自带的服务实例，以供用户进行绑定使用，并在更为复杂的环境中验证该动态更新技术的有效性。

参考文献

[1] Mell P, Grance T. The NIST Definition of Cloud Computing [R]. National Institute of Standards and Technology, 2011

[1] http://tuscany.apache.org

[2] Scott D. Assessing the costs of application downtime[OL]. http://citeseerx. ist. pus. edu/showciting?cid=3757589
[3] Ma X, Baresi L, Ghezzi C, et al. Version-consistent dynamic reconfiguration of component-based distributed systems[C]// Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011: 245-255
[4] Kramer J, Magee J. The evolving philosophers problem: Dynamic change management[J]. IEEE Transactions on Software Engineering, 1990, 16(11): 1293-1306
[5] Vandewoude Y, Ebraert P, Berbers Y, et al. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates[J]. IEEE Transactions on Software Engineering, 2007, 33(12): 856-868
[6] Su Ping, Cao Chun, Ma Xiao-xing, et al. Automated Management of Dynamic Component Dependency for Runtime System Reconfiguration[C] // Software Engineering Conference (APSEC 2013). IEEE, 2013: 450-458

(上接第 55 页)
图反向地址解析 API 发起 HTTP 请求将经纬度转化为地址信息的组件实现。用户信息服务由能够获取本机号码的服务组件实现。购书服务由两种与部署在不同 Tomcat 中的购书 Web 服务 A、B 进行 SOAP 通信的服务组件实现，其中 Web 服务 A 可用性较差，B 较好。第三方支付服务由一个与 Web 服务通信的组件实现。

实验过程从两个维度考虑：1)附近有无蓝牙定位热点，2)服务 A 所在的 Tomcat 是否开启。因此就有了 4 组对比实验，系统日志如图 4 所示，运行环境(a)为有蓝牙定位热点且购书服务 A 可用，(b)为附近无蓝牙定位热点但购书服务 A 可用，(c)为附近有蓝牙定位热点但购书服务 A 不可用，(d)为附近无蓝牙定位热点且购书服务 A 不可用。从框出部分可以看出，ASOF 能够实现在不同的移动环境下替换失效的服务绑定，在一定程度上实现系统的自修复，从而提升组合服务整体上的可用性。

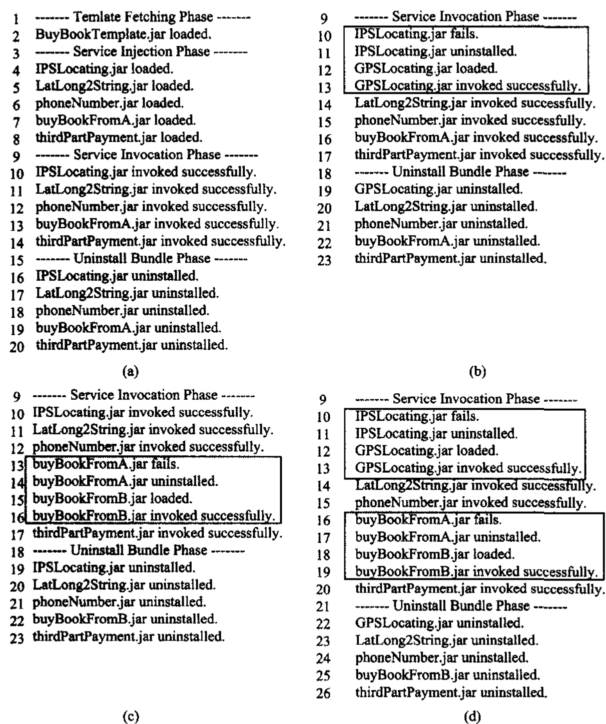

图 4　4 组对比实验系统日志

结束语　本文提出了一套 Android 移动终端服务编排框架 ASOF，该框架借助 OSGi 规范，实现了普适计算环境下 Android 平台上不同类型服务的混合组装。在服务失效时，该框架能够自主替换失效的服务组件，实现复合服务的自修复。由于模板和服务匹配都是在服务器端进行的，因此客户端的资源消耗得以降低，服务组合的过程也就更加轻量。同时，本文也给出了一套标准的 ASOF 实现，并为 ASOF 的两个重要参与者(模板定义者和服务开发者)提供了一套开发工具包。基于工具包，模板定义者可以定义符合自己业务逻辑需求的组合服务模板，服务开发者可以实现具有一定功能的具体服务组件，以对应资源库作为平台发布。最后以一个具体案例验证了框架在不同的普适计算环境下复合服务的自修复能力。

参 考 文 献

[1] Satyanarayanan M. Pervasive computing: Vision and challenges [J]. Personal Communications, IEEE, 2001, 8(4): 10-17
[2] OSGi Homepage[OL]. http://www. osgi. org
[3] Knoernschild K. Java Application Architecture: Modularity Patterns with Examples Using OSGi[M]. Prentice Hall Press, 2012
[4] Mokhtar S B, Preuveneers D, Georgantas N, et al. EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support[J]. Journal of Systems and Software, 2008, 81(5): 785-808
[5] 唐磊，淮晓永，李明树. 一种基于上下文协商的动态服务组合方法[J]. 计算机研究与发展，2008，45(11)：1902-1910
Tang Lei, Huai Xiao-yong, Li Ming-shu. An approach to Dynamic Service Composition Based on Context Negotiation[J]. Journal of Computer Research and Development, 2008, 45(11): 1902-1910
[6] Kalasapur S, Kumar M, Shirazi B. Dynamic service composition in pervasive computing[J]. IEEE Transactions on Parallel and Distributed Systems, 2007, 18(7): 907-918
[7] Rouvoy R, Barone P, Ding Y, et al. Music: Middleware support for self-adaptation in ubiquitous and service-oriented environments [M] // Software engineering for self-adaptive systems. Springer Berlin Heidelberg, 2009: 164-182
[8] Groba C, Clarke S. Opportunistic composition of sequentially-connected services in mobile computing environments[C]// 2011 IEEE International Conference on International Conference on Web Services (ICWS). IEEE, 2011: 17-24
[9] Guinard D, Trifa V, Karnouskos S, et al. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services[J]. IEEE Transactions on Services Computing, 2010, 3(3): 223-235
[10] 张威，史殿习. OSGi4HSI：普适计算环境下的异构服务集成框架[OL]. http://cpfd. cnki. com. cn/Article/CPFDTOTAL-JDMT-201010001006. htm