第37卷 第19期 计 算 机 工 程 2011年10月
Vol.37 No.19 Computer Engineering October 2011

·工程应用技术与实现· 文章编号：1000—3428(2011)19—0239—04 文献标识码：A 中图分类号：TP393

基于CORBA的自适应系统实现

郑尚书，沈立炜，彭 鑫，赵文耘

(复旦大学计算机科学技术学院，上海 200433)

摘 要：针对自适应系统在运行过程中的状态一致性问题，基于公共对象请求代理体系结构(CORBA)规范，提出一种总线消息型的自适应系统实现方法。该方法以构件组装模型为核心，利用总线对运行时的构件进行状态侦测及重配置操作，使用事务处理机制解决CORBA系统的状态一致性问题，使系统可在运行时修改自身行为。在Java/C++复合构件组装工具上的实现结果验证该方法的正确性，并表明其具备较好的稳定性和自适应能力。

关键词：自适应系统；总线；公共对象请求代理体系结构；状态一致性

Implementation of Self-adaptive System Based on CORBA

ZHENG Shang-shu, SHEN Li-wei, PENG Xin, ZHAO Wen-yun

(School of Computer Science and Technology, Fudan University, Shanghai 200433, China)

【Abstract】Aiming at the runtime state consistency of self-adaptive system and based on Common Object Request Broker Architecture(CORBA) standard, this paper proposes an implementation method of bus based self-adaptive system. The core aspect of the approach is a component assembly model with self-adaptive ability. Bus is used to monitor components' status and do reconfiguration, message transaction scheme is introduced to ensure system state consistency. It makes the system reconfigure itself at runtime. A Java/C++ composite component assembly tool which is developed independently is used to evaluate this method, result shows that it is a stable and it has self-adaptive.

【Key words】self-adaptive system; bus; Common Object Request Broker Architecture(CORBA); state consistency

DOI: 10.3969/j.issn.1000-3428.2011.19.079

1 概述

当前，自适应系统已经在人类生活的方方面面得到广泛应用。由于这类系统可以通过自身的配置和重配置不断适应变化的用户需求、系统入侵或缺陷、变化的运行环境和资源可用性[1]，因此能为需要高可靠性的行业(例如航空航天、医疗控制系统等)提供稳定的7×24的服务，尤其是可以避免在构件失效的情况时严重危害软件安全性[2]。使用基于构件的开发方法设计与实现自适应系统，其设计的核心是系统架构，或称为体系结构，它用来描述整个系统内部各个构件之间关系的模型。在体系结构中，构件表示相对独立的系统功能组成单元，因此基于构件的开发模式使软件系统的架构更清晰，增强软件开发人员的分工协作能力，大幅度提高软件产品的生产速度。基于构件的开发方法发展到现在有2种主要的系统架构设计方式：接口调用型和总线消息型。总线消息型类似于计算机体系结构，所有构件的消息都通过总线传递，总线是消息的传输通道[3]。使用总线消息型的架构可以提高构件的复用水平，更容易并行开发，降低生产周期，提高构件质量，降低开发成本。

OMG(Object Management Group)提出的公共对象请求代理体系结构(Common Object Request Broker Architecture, CORBA)是一种广泛使用的跨语言调用规范，它基于消息总线型，可以使不同语言开发的构件通过总线进行交互，实现系统的功能。

使用CORBA开发总线消息型自适应系统是一个有效的方式。在总线上注册的构件都可以在动态运行时被更改或替换，并且支持构件的分布式部署，而CORBA技术本身就是分布式系统[4]。目前，分布式的自适应系统有不少，但是它们的实现方式与自适应控制都比较复杂，不如总线式灵活，开发不便、实现难度大。

另一方面，自适应系统的状态一致问题不容忽视。在自适应系统中，将状态进行迁移是保证系统自适应前后状态一致的重要策略。在自适应动作前后，容易发生状态不一致从而导致业务系统错误。如网银用户c使用银行B的网上银行转账给用户d，在c已经从账户中划出款项而d未收到时网银系统崩溃，经过自适应后网银系统恢复正常，而此时系统若重新要求c输入转账，则会造成业务错误。可见，保证自适应系统在自适应前后的状态一致对业务正确性非常重要。

本文基于CORBA规范，采用总线消息型的构件组装模式，提出自适应系统的一种实现方法，并解决状态一致问题。

2 相关知识

2.1 自适应系统

自适应系统具有自管理的特性，包括自配置、自治愈、自优化和自保护的能力[5]。在系统架构中，对应于自适应体系结构的3个特征，自适应系统总是存在3个模块：环境侦测、分析决策、执行重配置。

2.2 总线型架构的软件系统与CORBA

软件系统中的总线型架构是模拟计算机硬件系统中总线结构的一种构件体系架构。即插即用、开放性、透明性、可

基金项目：国家自然科学基金资助项目(90818009)

作者简介：郑尚书(1983－)，男，硕士，主研方向：自适应系统，动态软件体系结构；沈立炜，博士；彭 鑫，副教授；赵文耘，教授、博士生导师

收稿日期：2011-04-18 **E-mail：**072021128@fudan.edu.cn

靠性和通用性是软件总线设计开发的宗旨[6]。

CORBA 本身是一种远程方法调用模型，其特点是支持不同语言间的方法调用。在总线式的体系结构当中，可以结合 CORBA 技术，将总线作为一个服务提供者，所有总线上的构件是调用总线服务的调用者。在这种模型之下，可以非常方便地实现总线式架构。

2.3 长事务管理

为了在自适应系统重配置过程中保持系统状态的一致性，而重配置可能是一个长生命周期的，或者协调者对参加者不拥有完全的控制权，所以，需要引入一种类似 Web 服务中提出的长事务思想。

首先，自适应系统中的长事务是由一连串的原子事务构成的。原子事务一般对应到某个构件中的一个操作，并且原子事务符合 ACID 特性。长事务的执行跨越多个构件或者应用系统，支持部分回滚。回滚结束后，等待下一个原子事务被恢复或者替换掉，长事务继续执行。

对于某个构件或者应用系统方法所对应的原子事务，可以是传统处理方法，也可以是长事务嵌套。如果是一个简单构件，可以按传统的原子事务处理，比如数据库系统或者分布式系统里面的事务处理机制；如果是复合构件或者对总线上其他构件的调用，则这个原子事务内部可以嵌套一个内部的长事务。通过这个机制，实现整个自适应系统的事务处理机制。

3 总线型 Java/C++构件组装自适应模型

本节主要描述关于如何在总线型架构软件系统中添加自适应特征的一些问题。图 1 中是已添加自适应模块的总线型 Java/C++复合构件组装模型。其中，Web 服务器构件组与客户端构件组是 2 个构件例子；消息处理器、消息过滤器、注册表等是组装模型内部模块；自适应上下文(Context)、计划(Plan)、分析器(Analyzer)、重配置器(Reconfiguration)是自适应功能模块。图 1 中虚线部分为负责自适应处理的扩展部分。模型中采用 CORBA 技术作为总线与构件、构件与构件之间的底层通信协议。使用 CORBA 技术的显著特性之一是支持异构构件，只要构件遵循模型的接口规约。模型支持分布式构件调用，所以，这些构件均为在线构件。构件可以被启动、停止、暂停，存在运行状态。

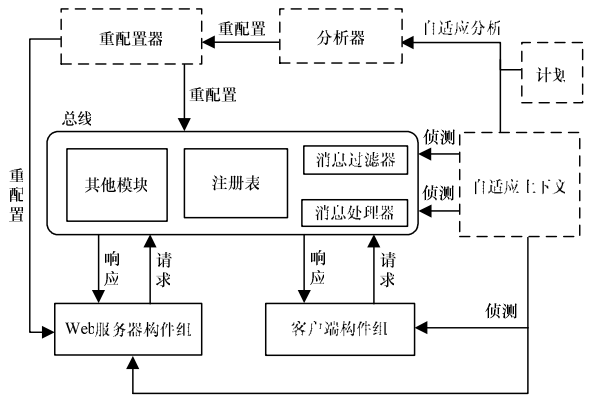

图 1 具备自适应特征的 Java/C++复合构件组装模型

3.1 模型结构

在 Java/C++复合构件组装模型中，总线所具有的功能主要包括构件注册、消息接收、消息发送、消息处理(消息过滤、信息添加等)。

在运行过程中，所有构件之间的联系与调用均通过总线实现。构件必须向总线注册才能成为模型中的在线构件。为了使构件组装模型支持自适应，首先需要对该模型原有的架构进行扩展。本文在总线架构基础上添加自适应的操作是使用构件冗余方法。

在模型中添加构件组的概念，几个不同的构件实例组成一个构件组。模型中原先负责处理请求的构件包装器将从包装一个构件转变到包装一个构件组，构件组中的所有组员构件是一个构件类型的多个不同实例。构件组里存在一个组长领头构件(Foreman)。总线对构件的交互均经包装器定向到 Foreman。构件组内存在同步机制，在处理完一个请求后，所有组员构件的状态始终与 Foreman 保持一致，以备 Foreman 被替换后能及时将状态复制到组内的替换者上去。本文涉及的自适应状态一致性主要针对该替换过程中的状态同步问题。

为了支持自适应，需要在模型中添加自适应模块。自适应中的 Context 模块负责侦测系统，发现问题。在本文中，Context 主要负责侦测总线上的系统状态、构件状态、消息传送。Context 交互的对象包括注册表、消息处理器、构件包装器等。

在自适应过程中，模型中使用事件提供自适应的分析决策依据。在 Context 模块侦测到需要处理的系统变动时，会发送事件到分析决策模块，然后分析决策模块进行分析并做出正确的自适应决策，选择出与该事件匹配的 Plan，最后依据 Plan 实行自适应动作。事件内容主要包括 3 个部分：事件源，事件类型，事件内容。事件源表明了事件发生的地点，是构件或者总线上的某个模块；事件类型指定了事件所属范畴；事件内容包含了需要提供给分析决策模块进行分析的数据，比如需要替换掉的构件的 ID 或者目标构件需要满足的特定条件数据。

3.2 自适应侦测

由于整个系统运行时所有构件都是注册在总线上并且所有通信均经过总线，这是一个以总线为中心的系统，因此自适应侦测主要关注总线上的异动。与此同时，构件或者构件组需要自适应的异常情况或者自适应请求是被提交到总线上交由总线进行处理。

自适应侦测主要处理 3 种类型的情况：

(1)由构件内部需求所导致的自适应，比如需要更换处理算法，或者子事务失败。这种情况属于构件主动发出的自适应请求。

(2)构件在执行过程中失效，由构件侦测发现并处理。

(3)总线在系统运行过程中出于特定需求进行自适应，例如进行负载均衡，该情况是由总线主动提出。

3.3 分析决策与重配置

分析器以上下文和事件(Event)作为接收参数，通过决策库分析出对应的决策结果。如果有对应的分析结果存在，则会进行重配置。自适应处理算法如下：

```
Analyzer.analyze(context, event) {
if context is null or event is null then
  logger.info("Sorry, there is no self adapt reconfiguration matches!");
  return
end if
plan←planRepository.searchPlan(context, event);
if  plan is null then
  logger.info("Sorry, there is no self adapt reconfiguration
```

```
matches!");
        return
    end if
    plan.execute(event);
    }
    searchPlan(context, event) {
    //在模型中，目前只是以最简单的按照事件来源以及事件名的匹
//配方式搜索出对应的决策
    for i←1 to entries.length
      entry←entries[i]
    if entry.getContext().getClass() equals context.getClass()) then
     if entry.getEventName() equals event.getName()) then
       return entry.getPlan();
       end if
     end if
    end for
    …
    }
```

重配置的手段在 Java/C++复合构件组装模型上主要是通过改变构件端的 Foreman 或者消息重定向实现。替换策略有 2 种方式：

(1)组内替换，在组内选择一个备用构件实例替换掉当前的 Foreman，该情况一般发生在某个 Foreman 失效的情况下。

(2)组间替换，功能性的重配置，比如某个构件的特性不适合当前系统状态，或者某个构件负载过高需要被替换为具有更大处理能力的构件中去。

4 事务处理与状态一致性

4.1 事务处理架构

在系统运行过程中，模型中的构件(构件组)在一个时刻只会也只能接收并处理一个请求信息。构件在处理一个请求信息的过程中采用传统的事务机制保证该请求过程中 ACID 属性。如此，可以将一个请求认作一个独立的事务。构件在处理请求过程中给其他构件发出的请求称为该请求的子请求，对应为子事务，构件在处理的请求称为该子请求的父请求。父请求(父事务)与子请求(子事务)形成树形的请求(事务)结构。

为了保证状态一致，构件与构件之间的状态是相互独立的，如此在自适应前后被替换构件的状态能够正确地迁移到替换构件中去。构件在接收到请求之后只有 3 个可能结果：返回成功消息；返回失败消息并且内部事务回滚保证构件实例与处理请求前一致；构件失效引发自适应重配置。若某 Foreman 所在构件组中存在组员，则在 Foreman 完成某个请求后，只有所有组员的状态已经被同步到与 Foreman 一致，才最终返回成功消息。

在本文模型中，结合 Java/C++复合构件组装模型的特性，通过消息重发(事务重做)或者消息回滚(事务回滚)的方式实现构件之间的状态一致，在事务处理中心中存在一个状态存储模块，这个模块是一个数据库，负责保存下列数据：

(1)当前所有构件组的状态。

(2)当前系统中所有消息(事务)的状态。

模块内包含构件消息队列、消息池与消息日志等。其中，最重要的是构件消息队列，保存构件发出到其他构件的所有请求消息。构件消息队列的队列头称为最终请求。

4.2 未自适应中的状态同步

未自适应时有以下 2 个场景需要保持构件状态一致：

(1)Foreman 成功完成一个请求和。

(2)Foreman 完成一个请求，但是返回的是失败消息并且未触发自适应。

场景(1)是最正常且最多出现的场景，在场景(1)中，维护状态一致的主要目的主要是为了在经过一个请求之后，Foreman 与组员即自适应替换构件之间的状态同步，成功完成时的同步算法如下：

```
Bus.synchronizeNormal(lastRequest){
  id←lastRequest.getReceiver().getID();
  com←searchComponentByID(id);
  teammates←com.getGroup().allTeammates();
  //表明以同步方式执行这个请求消息
  lastRequest.setInSyncMode(true);
  for i←1 to teammates.length
    com←teammates[i]
   if com.getID() equals id then
      continue;
   end if
   com.handle (lastRequest);
  end for
  lastRequest.setIsSynchronized(true); //设置同步完成
}
Component.handle(request){
//正常情况下，所有消息都会被处理并且转发到相应的目标构件
//中，但是如果是某个正在被同步的消息的子消息，那么该子消息不
//会被发送到目标构件，而是从消息池中找到过去同个子消息的对应
//返回消息
  if request is in synchronize work mode and request is sent out then
      //如果这是一个外发的子请求消息且是同步模式
      parent←request.getParent(); //获得父请求
      subRequests←bus.getSubMessagesCompleted(request);
      //以 request 为参数，获得父请求的所有已经完成的子消息
//中与本 request 对应的那条
      oldRequest←subRequests[0]; //数列长度只可能为 0 或者 1
      response←bus.getResponse(oldRequest);
      process(response); //返回处理结果
  end if
  …//其他处理分支
}
```

上述算法的核心思想为：当 Foreman 成功完成一个请求后，将该请求发送给每个组员构件(同步模式)，组员构件在处理该请求时所发出的所有子请求消息不发送到目标构件，而是从总线消息池中找到 Foreman 在处理时已经获得的返回消息，直接使用。如此，将整个构件组作为一个整体时，构件组在同步保持一致性时，不会重复发出相同的子请求消息。

在场景(2)中，如果 Foreman 完成的请求消息不存在父请求，那么不需要同步，否则，该父请求需要进行内部回滚。返回失败消息时的同步算法如下：

```
Bus.synchronizeFail(lastRequest){
  parent←lastRequest.getParent();
  if parent is null then return end if //忽略
  com←searchComponentByID(parent.getSender().getID());
  rollback(parent);
}
```

4.3 自适应中的状态一致

在 3.2 节列举 3 种自适应侦测情况，分别对应为场景(3)构件失效、场景(4)构件在执行过程中触发自适应、场景(5)系统运行中进行自适应。所有保持状态一致的操作均在构件替换之前进行。

场景(3)需要存在多种不同情况，针对构件消息队列中的最终请求是否被目标构件收到与否或者已经被收到但是是否有回应。构件失效时保持状态一致算法具体如下：

```
Bus.adaptSynchronizeFail(lastRequest){
  …
  confirmed←lastRequest.isConfirmed(); //是否被确认收到
  if not confirmed then
    return //如果消息未被收到，则直接自适应，替换构件
  end if
  response←searchResponse(lastRequest); //在消息池中寻找返
//回消息
if response is not null then
  return  //已经有返回消息，直接自适应替换构件
end if
  //最终请求已经被收到(目标构件已经对该请求进行处理)，但
//却未发回返回消息(该请求正在被处理中)的情况，需要迁移状态
subRequests←searchSubMessages(lastRequest); //搜索到最终请
//求的所有子请求
sortMessage(subRequests); //按时间先后顺序排序
index ←subRequests.length–1;
while index>0 do
//从后往前，回滚子请求
   m←subRequests[index];
   rollBack(m);
end while
//回滚后，所有与该最终请求相关的状态改变已经通过回滚被取
//消，从头开始
lastRequest.setIsConfirmed(false); //准备重发
clearSubMessages(lastRequest); //清空子请求消息，包括子请求
//的确认消息和返回消息
  …
}
```

算法执行完毕后，由失效构件发出的最终请求所造成的其他构件的改动均已通过回滚恢复到接收该请求相关的子请求前，自适应模块可以将后备构件替换然后重新发送最终请求。

场景(4)具体考虑在自适应时该构件正在处理某个请求，在处理过程中有若干个子请求，而触发自适应的原因即可能是因为某个子请求失败，也可能是直接发出自适应中断，这两情况可以统一考虑，策略是：返回当前正在被处理的请求的所有已成功子请求，然后返回表明失败的返回消息。构件正常执行时触发自适应的保持状态一致算法具体如下：

```
Bus.adaptSynchronizeNormal(lastRequest){
//这里的 lastRequest 参数不是构件消息队列里的最终请求，而
//是当前构件中正在处理的请求的最后一个子请求
  if lastRequest is null or lastRequest is not confirmed then
    return //为空或者未被收到
  end if
  subRequests←searchSubMessages(lastRequest); //搜索到最终
//请求的所有子请求
  if subRequests.length equals 0 then
   //没有子请求，直接回滚后退出
    rollback(lastRequest);
    return;
  end if
  sortMessage(subRequests); //按时间先后顺序排序
  index←subRequests.length–1;
  while index>0 do
   //从后往前，回滚子请求
   m←subRequests[index];
   adaptSynchronizeNormal(m); //递归处理所有子请求
end while //构件中正在处理的请求的最后一个子请求已经回滚
rollback(lastRequest); /回滚自身
}
```

上述算法是一个递归调用结构，因为在本文模型中的父消息子消息的结构是一个递归树状结构，所以用递归处理比较简洁方便。

场景(5)与场景(3)如上文所述，均属于系统“主动”操作构件进行自适应，所以，场景(5)可以使用场景(3)中相同的状态一致算法。

以上通过对 3 种发生自适应时的场景进行区分处理，并应用不同的状态一致算法，能够保证在严格规范下该系统的自适应前后状态一致性。

5 工具实现与案例研究

本文采用复旦大学软件工程实验室开发的基于 CORBA 技术的总线架构框架 Java/C++复合构件组装工具，是本文模型的一个具体工具实现。工具采用 CORBA 技术实现总线型体系结构，构件只要符合模型的规约，即可无缝“插入”挂接到总线之上，并调用总线上已有的其他构件上的方法。该工具目前支持 Java 和 C++构件，其他语言实现仍在开发当中。

为验证本文方法的有效性，使用了一个简易在线电脑组装平台进行实验，该平台是运行在组装工具上的一个具体应用。该系统类似目前互联网上主流电子商务网站，用户在这个平台上通过自己的偏好进行搜索然后平台提供推荐结果，用户选中组合后可以进行支付并且输入收货地址接收货物。

在实验中，首先在不具备自适应能力的平台(称为 Source)上进行测试记录，随后在具备自适应但不包括状态一致能力的平台(称为 Target)以及包括状态一致能力的平台(称为 Advanced)上进行测试，将对这 3 种测试结果进行比较。

实验首先测试自适应能力，结果见表 1。

表 1 3 个平台的自适应能力测试结果比较

平台	客户数	成功数	成功率/(%)
Source	100	0	0
Target	100	86	86
Advanced	100	81	81

经分析，有自适应能力平台上的总共 33(14+19)个失败，是由于系统自适应造成的系统暂停和恢复引起的。其中，Advanced 因为有状态一致处理，所花费的时间更多。

其次，测试状态一致。将平台中支付接口构件分别放在不同硬件资源的主机上。在 Target 和 Advanced 中的自适应规则中添加“当系统同时访问量超过 50 时就变化账户资金管理构件”。实验开始后，逐渐将并发访问量从 1 增加到 100。限于篇幅，仅罗列部分数据。结果经分析，得到 Target 上一致率为 26%；Advanced 上一致率为 94%。可见，添加了事务处理和一致性规则之后的 Advanced 平台具有很大优势。

从实验结果可以看到，在 Java/C++复合构件组装模型上添加自适应能力后，系统的可靠性提高了很大一个幅度。在添加了消息事务处理机制后，系统状态的一致率从 26%明显提高到 94%，而在其中的不一致部分(Target 的 74%和 Advanced 的 6%)，仍然是因为系统暂停或者延时造成的缺陷。所以，还需要在这个方面继续改进与优化。总体上来说，文中所应用的方法是成功的，正确率也比较高。

6 结束语

目前，对自适应系统中的状态一致问题的研究还不是很

(下转第 257 页)