

一种基于主动复制的动态容错算法

孟玉明¹, 张修如¹, 刘玲霞²

(1. 中南工业大学 信息科学与工程学院, 湖南 长沙 410083;

2. 空军工程大学 电讯工程学院, 陕西 西安 710077)

摘要:对于诸如 Web 服务这种面向广域环境的分布计算, 服务应答慢等同于不可用。它在不改变可用性的前提下对算法的性能提出了很高的要求。然而目前的容错算法很少致力于提高算法的性能。文中提出一种新的复制算法 RRR。它的主要优点是: 应答由处理速度最快的副本返回, 响应时间短; 节约系统资源; 基于主动复制, 但避免了重复嵌套呼叫问题。最后用理论分析和实验证明了算法的优点。

关键词:容错; 复制算法; 主动复制; 性能

中图分类号: TP302.8

文献标识码: A

文章编号: 1673-629X(2007)12-0048-05

A Dynamic Fault Tolerant Algorithm Based on Active Replication

MENG Yu-ming¹, ZHANG Xiu-ru¹, LIU Ling-xia²

(1. College of Information Sci. and Eng., Central South University of Industry, Changsha 410083, China;

2. Telecommunication Eng. Institute, Air Force Eng. University, Xi'an 710077, China)

Abstract: To the wide area network oriented distributed computing such as Web services, a slow service is equivalent to an unavailable service. It makes demands of effectively improving the performance without damaging availability to the replication algorithms. However, existing replication algorithms seldom address to improve the performance. Propose a new replication algorithm named RRR. Its main advantages are: Its response time is short because the response is returned by the fastest replica; It economizes the system resources; It is based on the active replication algorithm, but it avoids the redundant nested invocation problem. Finally prove the advantages by analyzing and experiments.

Key words: fault tolerance; replication algorithm; active replication; performance

0 引言

容错是提供可靠性和可用性的关键机制, 使得系统在有失效发生时仍然能够继续正确运行^[1]。在分布计算环境中, 容错一般都通过冗余配置实现。

主动复制和被动复制是两个经典的复制算法^[2]。主动复制算法中, 所有冗余副本同时响应客户请求, 当有副本失效时, 其余副本在失效副本缺席的情况下继续正常工作。它失效恢复时间短, 但耗费系统资源, 同时它还要求成员的状态是确定的 (deterministic)^[3], 而且会带来重复嵌套呼叫问题^[4]。被动复制算法中, 冗余副本分成主从两种, 只有主副本接收和处理请求, 主副本还负责更新从副本的状态。当主副本失效时, 将

从副本中的一个升级为主副本, 继续提供服务。它节约系统资源, 不要求服务的状态是否确定, 但失效恢复时间长, 而且还带来了主副本状态一致性问题。尤其是当主副本在处理请求的过程中突然失效, 这时副本的状态一致性很难维护, 处理不当会带来严重后果。

对于诸如 Web 服务^[5]这种面向广域环境的分布计算, 服务应答慢等同于不可用。它在不改变可用性的前提下对算法的性能提出了很高的要求。

然而目前的容错算法大多只是对主动或被动复制算法进行了某些方面的改进, 很少致力于提高算法的性能。例如 Semi Active^[6]解决了主动复制算法中服务状态的确定性问题。文献[7]提出异步主动算法 Asynchronous Active Replication。

经过分析发现, 性能较优的主动复制算法中的结果协调 (Consensus) 过程对算法性能带来很大的影响, 主要体现在: (1) 结果协调过程必须等所有副本都处理完请求后才能进行。应答速度由处理速度最慢的副本决定; (2) 在大部分副本没有失效的情况下需要一个至

收稿日期: 2007-03-12

基金项目: 国家 863 计划项目 (2004AA112020); 空军工程大学电讯工程学院博士启动基金 (KDYBSJ303)

作者简介: 孟玉明 (1978-), 男, 河南开封人, 工程师, 研究方向为计算机应用; 张修如, 硕士, 副教授, 研究方向为计算机应用。

少是◇S(Eventually Strong)级的失效检测器^[8]的支持,而这样的失效检测器在分布计算环境尤其是异步环境下是很难实现;(3)结果协调过程本身是耗费时间的,因此,改进 Consensus 过程能有效改善算法的性能。

针对主动复制算法中的不足,笔者提出一种新的复制算法 RRR(Rapid Response Replication)。它基于下述设计思想:每个冗余副本都接收请求消息,但应答只由处理速度最快的副本返回。与主动复制算法相比,其主要优点是:(1)应答由处理速度最快的副本返回,不需要对结果协调(Consensus),响应时间短;(2)算法根据副本的负载情况动态改变执行请求副本数的大小,节约了系统资源;(3)算法基于主动复制,但避免了重复嵌套呼叫问题,不需要引入过滤机制。

1 系统模型

RRR 算法基于文献[9]提出的分布式面向对象系统。模型中 client 表示客户,replica₁, ..., replica_n 表示提供服务的对象组 Group(由一个唯一 URN 来标识)的 n 个成员。与客户相对应,将一个对象组称为一个服务。FTC(Fault-Tolerant Communication Layer)层提供容错通信服务,包括可靠的点到点通信及可靠的原子组播服务。FTO(Fault-Tolerant Object Layer)层为系统中的分布对象提供透明的容错支持,包括创建对象组、检测失效,以及设置检查点和完成恢复等。FTC 层和 FTO 层彼此独立,实现了复制协议和通信协议的分离。在该系统中,对象本身并不参与算法。FTO 层捕获对象消息,并代表对象实现算法逻辑以提供容错。

在模型中,假定底层系统能识别各种消息类型,譬如能识别一个请求消息是客户对服务的请求消息还是一个对象对另一个对象的请求消息。这种假设是合理的,因为所有分布对象之间的通讯都由底层设施负责。这里将系统中的消息进行了如下分类:

- * Client-Request: 客户发送给服务的请求消息;
- * Client-Response: 服务发送给客户的应答消息;
- * Object-Request: 一个对象发送给另一个对象的请求消息;
- * Object-Response: 一个对象发送给另一个对象的应答消息;
- * State-Update: 一个对象发送给另一个对象的状态更新消息。

对于组中的每个成员,系统都提供一个消息队列来暂存消息。所有对组成员的请求都按照先进先出的顺序被缓存在队列中。FTO 层负责消息的适配转发以及给消息标识或修改 ID。

2 容错算法

在 RRR 算法中,组中的每个成员都接收由系统组播的请求消息,但应答消息只由处理速度最快的成员返回。该成员在返回应答消息的同时向组中的其它成员发送状态更新消息。这就是算法的核心思想。RRR 算法的一个简单流程如图 1 所示。

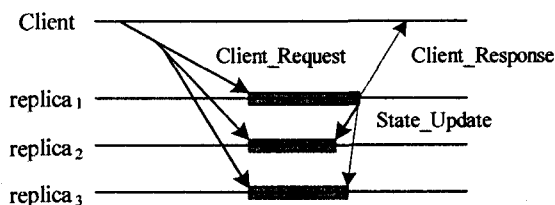


图 1 算法流程

组中的成员根据负载情况和处理速度决定是否执行请求。请求总能由系统中负载最轻、处理速度最快的主机上的副本及时执行。请求不是必须由组的每个成员处理,应答也不必等待所有执行请求的成员执行完毕并对结果做协调之后再返回,这就是它和主动复制算法最主要的区别,也是算法动态性的体现。

算法维护了一个 Hash 表来保证组中处理速度最快成员的唯一性(组中的两个成员同时或几乎同时最快处理完某个相同请求的情况是存在的)。组成员在不确定自己是否处理速度最快的时候,发出询问消息,算法根据 Hash 表中的记录返回确认信息。这样,在系统中又增加了两种类型的消息,即对象发出的询问消息 ObjectF-Query 以及算法返回的应答消息 ObjectF-Response。Hash 表以请求(包括客户和对象请求)消息的 ID 作为 Key 值,数据项格式为(地址列表,处理速度最快的成员地址,时间),其中地址列表是除了处理速度最快成员以外的所有其他发出询问的成员地址列表。

在算法中每个消息都由一个 ID 标识。系统中消息 ID 的生成规则如下:

- * Client-Request: 分配一个唯一 ID。系统为发送给对象组的 Client-Request 附加一个唯一的 ID。
- * Client-Response: 和相应的 Client-Request 消息的 ID 相同。
- * Object-Request: 在系统中,该类型的消息只有在响应一个客户请求就是请求另外一个服务的情况下才会产生。产生 Object-Request 消息时不修改消息的 ID,此时的 ID 和相应请求消息的 ID 相同。直到发送 Object-Request 消息的时候, ID 才被修改。ID 被修改为相应请求消息(包括 Object-Request 和 Client-Request)的 ID 加请求服务的 URN。
- * Object-Response: 和相应的 Object-Request 消

息的 ID 相同。

* State_Update、ObjectF_Query、ObjectF_Response: 和相应请求消息的 ID 相同。

算法有三个主要模块: Main()、Receive() 和 Process()。Receive() 模块在组成员接收消息时被调用。Process() 在组成员处理完请求后被调用。Main() 模块是算法中的协调者, 在接收到请求、接收到 ObjectF_Query 询问消息或者返回应答时都有可能被调用。下面将详细阐述算法的这三个模块。

2.1 Main() 模块 - 协调者算法

作为算法中的协调者, Main 模块算法如图 2 所示。

1) 接收到底层系统发来的请求消息后, 先在 Hash 表中添加一个记录, 记录值为 (请求 ID, (null, null, null)), 接着将该请求消息组播给组成员。

2) 接收到成员询问消息 ObjectF_Query 后, 根据 ID 查询 Hash 表中此请求处理速度最快的成员地址数据是否为空。

* 如果不为空, 则将该成员地址列表加入地址列表, 返回“否”应答消息给该成员。

* 如果为空, 则算法将该成员地址作为处理速度最快的成员地址写入该记录, 同时将此刻时间记录在时间数据项中, 返回“是”确认消息给该成员。需要说明的是, 如果算法同时接收到几个成员的询问消息时, 算法会挑选出一个成员作为处理速度最快的成员, 记录地址、时间, 同时将其余询问成员的地址记录在地址列表中。

3) 在将某个应答返回之前, 根据 ID 将 Hash 表中的记录删除。

```

Main(msg) {
  if(msg.type=Client-Request) or (msg.type=Object-Request) {
    put a new item to the hash table;
    multicast msg to the group members;
  }
  if(msg.type=ObjectF-Query) {
    if (the fastest member's address != null) {
      put the member's address into address list;
      return a negative ObjectF-Response;
    }
    else { put the member's address as the fastest member's address;
      return a positive ObjectF-Response;
    }
  }
  if(msg.type=Client-Response) or (msg.type=Object-Response) {
    delete the item in the hash table;
  }
}

```

图 2 Main() 模块

考虑到处理速度最快的成员可能会在发送询问消息之后失效, FTO 层在这段时间会通过系统中的失效检测机制来监测该成员的状态。一旦有失效发生, 算法接到 FTO 层的失效通知后从成员地址列表中选择

一个成员, 要求该成员返回应答。

2.2 Receive() 模块 - 消息接收算法

组中的成员可能会接收到五种类型的消息, 即 Client_Request、Object_Request、Object_Response、State_Update、ObjectF_Response。为了不破坏算法处理流程的完整性, 接收处理 ObjectF_Response 消息的流程在消息处理算法中阐述。对于 Object_Response, 算法无需做任何处理。对于其它消息, 消息接收算法如图 3 所示。

```

Receive(msg) {
  1 if(msg.type=Client-Request) or (msg.type=Object-Request)
    put msg into buffer;
    if(msg.type=State-Update) {
  5 search through the buffer and put the msgs into
      collection if buff.msg.ID=msg.ID;
      if(collection != null) { //相应请求已经或还未被处理
        if (there is a message in collection
          and message.type=State-Update) {
  10 discard msg;
          discard all buff-msgs included in collection;
          else replace buff-msg with msg;
        else put msg into buffer; //相应请求正在被处理
      }
    }
}

```

图 3 Receive() 模块

1) 直接将 Client_Request 和 Object_Request 类型的消息存入消息队列中, 等待算法调度。

2) 接收到 State_Update 消息后, 算法会搜索消息队列, 查找出所有和该消息 ID 相同的消息集合。

* 如果消息集合不为空 (该成员目前尚未处理或已经处理过相应的请求), 则判断消息集合中消息的类型。如果消息集合中有 State_Update 类型的消息 (该成员已经处理过相应的请求, 并且处理完毕时还没有接收到处理速度最快的成员发来的状态更新消息), 则丢弃接收到的 State_Update 消息以及队列中相同 ID 的消息。否则 (该成员目前尚未处理相应的请求), 用 State_Update 消息替换集合中的消息 (根据算法逻辑此时集合中必然只有一个消息, 且消息类型必为请求消息)。

* 如果消息集合为空 (该成员目前正在处理相应的请求), 则将 State_Update 消息存入消息队列中。

2.3 Process() 模块 - 消息处理算法

接收到的消息都存放在消息队列中, 由组成员依次处理。Process() 模块在组成员处理完请求消息后被调用, 流程如图 4 所示。

执行请求消息后, 算法会判断消息队列中是否有消息的 ID 和该请求消息的 ID 相同。

* 如果有 (该成员处理该请求的速度不是最快的), 则丢弃消息队列中相同 ID 的消息以及执行请求后产生的消息。

* 如果没有 (该成员处理该请求的速度可能是最

快的),则向协调者发送 ObjectF_Query 消息,根据 ObjectF_Response 消息中的返回值作不同的处理。如果返回值是“是”(该成员处理速度是最快的),则发送执行请求后可能产生的应答消息或 Object_Request(发送之前先修改 ID)消息,发送 State_Update 给组中的其它成员(这儿将发送消息作为一个原子操作)。如果返回值是“否”(该成员处理速度不是最快的,并且处理完毕时还没有接收到处理速度最快的成员发来的状态更新消息),则将产生的应答消息或 Object_Request 以及 State_Update 消息存入消息队列中。

```

Process(msg){
    //该成员处理该请求的速度不是最快的
    1  if(there are buff-msgs and buff-msg.ID=msg.ID){
        delete all such buff-msgs;
        discard msg;
    }
    5  else{//该成员处理该请求的速度可能是最快的
        send out ObjectF_Query and receive ObjectF_Response;
        if(ObjectF_Response is positive){
            if(msg.type=Object_Request)
                modify msg.ID;
        }
    }
    10 send out msg and send State_Update to other copies;
    else put msg and State_Update into buffer;
}

```

图 4 Process() 模块

2.4 组成员失效

组成员发生失效的时机很多,但对算法流程有影响的只有一个,即处理速度最快的成员在发送消息的时候失效。需要说明的是失效会对算法的流程有影响但不会影响算法的正确性。这种情况下,至少还有一个成员处理了请求并发送了 ObjectF_Query 消息(因为其它成员都没有接收到状态更新消息)。FTO 层通过失效检测机制检测到失效后,会迅速通知另外一个已经处理完该请求的成员发送应答和状态更新消息,该请求还是能被正确执行。

3 算法分析

3.1 算法正确性分析

下面简要验证 RRR 算法能够满足下述特性:

1) 组中的每个成员能保持状态一致。

成员的状态由它处理过的请求决定。在算法中请求通过可靠组播发送给每个成员。每个成员按照相同的顺序接收相同的请求。只要该成员没有失效,它会按照顺序处理请求。处理速度最快的成员会发送状态更新消息给其它成员。此时其它成员可能处于三种状态:正在、已经或尚未处理该请求。对于正在处理请求的情况,算法将状态更新消息加入消息队列中。请求执行完毕后,算法在消息队列中发现存在和请求消息 ID 相同的状态更新消息,算法会将该状态更新消息丢

弃(Process() module line 2,3)。此时该成员和处理速度最快的成员达到状态一致。对于尚未处理请求的情况,算法直接将状态更新消息替换请求消息(Receive() module line 12),当该成员执行此状态更新消息后,也和处理速度最快的成员达到状态一致。对于已经处理完该请求的情况,算法将保存在消息队列中的应答和状态更新消息以及接收到的状态消息丢弃(Receive() module line 10,11),该成员和处理速度最快的成员依旧保持状态一致。从以上分析可知,该算法能保证组中的每个成员保持状态一致。

2) 算法不会产生重复嵌套呼叫问题。

主动复制会带来重复嵌套呼叫问题。RRR 算法基于主动复制,但算法不会产生重复嵌套呼叫问题。在算法中,组中的每个成员都接收了相同的请求,但只有处理速度最快的服务会发送 Object_Request。对于处理速度相对较快的成员,在处理请求时也产生了 Object_Request,但按照算法流程,该消息会被丢弃。被请求的服务只会接收到了一个 Object_Request。

3.2 性能分析

这里主要对 RRR 算法和主动复制算法的响应时间进行比较。

在此不考虑从客户发出请求到组成员接收到请求的这段时间和返回应答所需时间,因为它们所需的时间对于这两种算法是相同的。也不考虑组成员失效处理的时间,因为组成员的失效处理对算法的请求响应时间是没有影响的。

主要分析这两种算法处理请求所需的时间。对于主动复制算法,处理请求的时间是所有成员处理请求所需时间的最大值,因为结果协调过程必须等所有的成员都处理完请求以后才能进行。对于 RRR 算法,处理请求的时间是所有成员处理请求所需时间的最小值,只要组中有成员处理完请求就会直接返回结果给客户,而不必等所有成员都处理完请求。因此,RRR 算法比主动复制算法有更短的请求响应时间。负载越大,RRR 算法的优势越明显,这点在随后的实验中可以得到充分说明。

另一方面,RRR 算法中不是每个成员都必须处理请求,比主动复制算法更节约系统资源。

3.3 算法实现及性能测试

采用 StarWebService^[10](自主研发的 Web 服务平台)作为算法的实现平台。SOAP^[11]引擎是一个调度器,是服务与服务、服务与客户之间交互的唯一通道。它负责接受客户请求消息,解析请求消息,根据请求定位服务,并负责返回结果。算法所需的消息调度、消息类型判断以及为消息附加 ID 等功能都由 SOAP 引擎

负责实现。

对主动复制算法和 RRR 算法进行了性能对比测试。测试在由若干台相同配置的主机组成的局域网上进行。主机配置为 Intel Pentium4 2.0GHz 主频、512M 内存、安装了 Windows 2000 操作系统以及 StarWeb-Services2.0 修改版。

首先测试了在服务冗余度为 3 且无失效的情况下,两个算法的平均请求响应时间随请求频度变化的变化情况。测试结果如图 5 所示。从图中可以看出,随着请求频度的增加,每种算法的请求响应时间都相应变长,但 RRR 算法比主动复制算法有更短的响应时间,这和上面的理论分析结果是一致的。

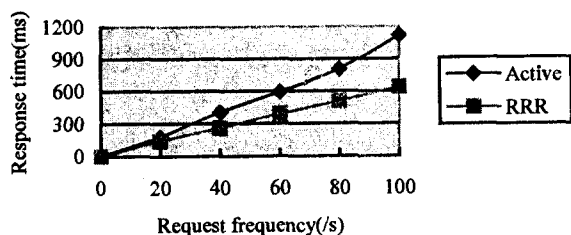


图 5 响应时间随请求频率变化的曲线

对在请求频率为 20/s 且无失效的情况下,各个算法的平均请求响应时间随服务组规模变化的情况进行了测试。测试结果是,随着服务组规模的增加,主动复制算法平均请求响应时间的增加幅度比 RRR 算法大很多。这是因为 RRR 算法总是由处理速度最快的服务返回应答,不需要等待其它成员结束执行请求并对结果做协调后才返回应答。

最后对算法在失效情况下的性能进行了测试,结果表明 RRR 算法的平均请求响应时间几乎不随组成成员失效个数的变化而变化。

4 结束语

介绍了一种基于主动复制的复制算法 RRR。它根据服务的负载情况动态改变执行请求副本数的大

小,请求结果只由处理速度最快的副本返回,在不浪费系统资源的情况下,保证了最短的请求响应时间。理论分析和实验表明,RRR 算法提供较快的请求响应时间。

参考文献:

- [1] Flavin C. Understanding fault-tolerant distributed systems [J]. Comm. of ACM, 1991(2): 57-58.
- [2] Addison W. Distributed Systems [M]. New York: ACM Press, 1993: 169-197, 199-216.
- [3] Chandy M, Lamport L. Distributed snapshots: Determining global states of distributed systems [J]. ACM Transactions on Computing Systems, 1985(1): 63-75.
- [4] Kenichi H, Tomoya E. Nested Invocation Protocol for Object-based Systems [C] // Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. Japan: [s. n.], 2003.
- [5] Web Services Architecture [EB/OL]. 2003. <http://www.w3.org/TR/ws-arch/>.
- [6] Powell D. Delta4: a generic architecture for dependable distributed computing [M]. New York: Springer Verlag, 1991.
- [7] Baldoni R, Marchetti C, Tucci Piergiovanni S. Asynchronous Active Replication in Three-tier Distributed Systems [C] // Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing. Japan: [s. n.], 2002.
- [8] Chandra T, Toueg S. Unreliable Failure Detectors for Reliable Distributed Systems [J]. Journal of the ACM, 1996, 43(2): 225-267.
- [9] Ganesha B, Anish K, Anil G. Fault tolerant objects in distributed systems using hot replication [R]. Technical Report TR-95-023, Department of Computer Science, Texas AM University, 1996: 89-95.
- [10] StarWebServices2.0 [EB/OL]. 2004. <http://www.starmiddleware.net/ws>.
- [11] SOAP1.2 [EB/OL]. 2003. <http://www.w3.org/TR/soap12>.

(上接第 47 页)

- [21] Liu Dongsheng, Wang Jianmin, Chan S C F, et al. Modeling workflow processes with colored Petri nets [J]. Computers in Industry, 2002, 49: 267-281.
- [22] 陈翔, 夏国平. 基于着色 Petri 网的工作流建模和合理性分析 [J]. 计算机集成制造系统 - CIMS, 2004, 10(4): 381-387.
- [23] 李炜, 曾广周, 王晓琳. 一种基于时间 Petri 网的工作流模型 [J]. 软件学报, 2002, 13(8): 1666-1671.
- [24] 周丹晨, 殷国富. 基于 Web 服务面向虚拟企业的柔性工作流管理技术研究 [J]. 计算机辅助设计与图形学学报, 2004, 16(4): 427-433.
- [25] Workflow Management Coalition. Workflow Process Definition Interface - XML Process Definition Language Version 1.0 [S]. WfMC - TC1025, 2002.
- [26] 邢建川, 李志蜀, 陈黎. 基于多 Agent 的企业过程建模和仿真技术研究 [J]. 系统仿真学报, 2006, 18(z1): 242-244.
- [27] 赵卫东, 黄丽华. 面向角色的多 agent 工作流模型研究 [J]. 管理科学学报, 2004, 7(2): 55-61.
- [28] 钟凌燕, 高济. 基于 XML 和 Agent 联邦的工作流建模方法 [J]. 计算机辅助设计与图形学学报, 2003, 15(3): 355-361.