多复制服务器间无阻塞的数据更新*

赵洪彪 周立柱

(清华大学计算机科学与技术系 北京 100084)

摘要 在Client/Server系统中,服务器的可用性是提高整个系统可用性的关键,采用多复制服务器是提高系统可用性的最有希望的手段.但是,复制数据更新过程中的阻塞问题是整个系统性能的一个瓶颈.本文提出一种无阻塞的多服务器独立提交的复制数据更新方法,对于因失效不能完成更新的服务器采用协调机制使其达到相同的最终状态.

关键词 客户服务器,数据复制,数据更新,数据协调,失效恢复.

中图法分类号 TP393

在Client/Server系统中,服务器的可用性是提高整个系统性能的关键,采用多复制服务器则是提高系统可用性最有希望的手段之一,但是,复制数据更新过程中的阻塞问题是整个系统性能的一个瓶颈.

复制数据更新的首要困难是控制多副本的一致性.通常采用的方法包括投票(Voting)和令牌(Token)传递方法[1],但是,这些方法的复杂性给实现带来困难.更新传播的另一个问题是保证事务的原子性,两阶段提交协议是使用最多的保证事务原子性的提交协议,但是它由事务协调程序(Coordinator)决定各个参与者是否可以提交,这样就降低了系统的可用性,并增加了事务提交的延迟时间.网络环境的不健壮性使得所有参与结点同时提交的可能性减小,不可避免地造成能够正常提交的结点等待因失效不能提交的结点,而形成事务提交的阻塞.因此,允许独立提交是解决阻塞问题的关键.在实际的Client/Server系统开发中,并不是所有的复制服务器上的数据都要自始至终维护一个一致的复制数据视图和进行即刻的更新传播[2],而可以采用一种延迟更新的方法.[3,4]在这方面有代表性的工作包括将分布式事务分裂为几个单结点的原子事务模型[5],或者是采用主/辅方式延迟更新,首先在主数据结点提交,然后异步传输到辅数据结点.[6]

本文就可等的复制服务器结点的数据复制提出一种无阻塞的更新方法,对于不能完成更新的服务器结点,利用一种协调机制在随后使数据达到相同的最终状态.

1 数据更新

我们假定在网络上有多个等同的复制服务器结点(文中简称结点),它们或者处于连接状态,或者处于非连接状态.但是在连接状态下通讯是可靠的,网络各个结点之间能够保持消息的顺序,保证消息的内容正确到达,并且提供超时检测.系统出现故障的情况有两种,或者是各个结点之间处于非连接状态而不能通讯,或者某个结点可能会因失效而不能正常工作.在每个结点有一个本地的逻辑时钟,在整个系统内对各个结点操作的定序可以利用Lamport邮戳机制解决.操作对象可以唯一标识,并且数据操作是一元的,它只影响到本地数据,每个事务中的数据操作不跨越多个数据对象.在每个结点都有一个本地日志,日志的格式是(事务标识,邮戳,事务协调服务器编号,数据对象标识,操作标识,操作参数),其中邮戳是指记录在事务协调程序中的事务接收时间.在每个结点建立一个用于数据协调目的的Reco-Info表,元组形式是(数据对象,结点标识),以便在数据协调时参考.

1.1 更新算法

数据的更新首先在某个服务器开始(我们称之为更新起始点),然后传播到其他结点.下面以a_c^d表示以C结点为更新起始点对数据d执行a操作,$time(a_c^d)$表示操作a_c^d的邮戳,H_C是结点C的历史日志,在每个结点,我们引入一个数据结构——一致性矢量$CV_c[1,\ldots,n]$,对应每个结点都有一个分量,利用它反映各个数据副本的一致性.一致性矢

---

* 作者**赵洪彪**,1964年生,博士后,主要研究领域为数据库,软件工程,分布式系统.**周立柱**,1947年生,教授,博士导师,主要研究领域为数据库,信息系统集成,计算机软件.

本文通讯联系人:周立柱,北京 100084,清华大学计算机科学与技术系

本文1997-03-07收到原稿,1997-05-09收到修改稿

量的语义由下面的不变式定义.

不变式 1. 对任意结点 X 和 Y，数据对象 d 和操作 a_X^d，有

$$a_X^d \in H_x \leftrightarrow time(a_X^d) \leqslant CV_x^d[y]$$

这个不变式的含义是：如果结点 X 执行了更新起始结点 Y 对数据 d 的某些操作 a_Y^d，则结点 X 也执行了以 Y 为更新起始点在 d 上的所有操作.这个不变式保证结点 X 执行了到 $CV_x^d[y]$这一时刻为止，以 Y 为更新起始结点的对数据 d 的所有操作，所以，$CV_x^d[y]$表示以结点 Y 为更新起始点，在结点 X 执行对数据对象 d 的最近操作的邮戳. 最初所有的一致性矢量的所有分量的初值置为 t_0.

算法 1. 更新起始点 C

```
Begin
    写 Begin-Transaction 到日志缓冲区;
    发送 Begin Transaction 到各个分布结点;
    执行读操作 R(d);
    Repeat
        记录写操作 W(d)到日志缓冲区;
        广播 Write(CV_C^d[C], W(d), t_w)到各个分布结点;
        CV_C^d[C]:=t_w;
    Until 所有写操作结束
        以原子写方式将提交本更新事务;
        向各个结点广播 Commit 消息;
        等待各个结点的确认;
          If 等待超时
          Then 对于所有未收到确认的结点 P,插入(d,P)到恢复文件 Reco-Info;
End
```

各个数据复制结点的动作由 Begin-Commit 消息触发.

算法 2. 各个复制结点 P

```
Begin
    写 Begin-Transaction 到日志缓冲区;
    Repeat
        接收 Write(CV_P^d[C], W(d), t_w);
        If CV_P^d[C]==CV_C^d[C];
          Then 执行此写操作并且写入日志缓冲区;
               CV_P^d[C]:=t_w;
          Else 对此事务执行本地 abort,并且 exit;
    Until 所有写操作结束;
    If 在超时前收到 Commit 消息
      Then 在本地日志中以原子写方式加入此事务的提交记录;
           向结点 C 发出确认;
      Else 对此事务执行本地 abort;
End
```

1.2 算法分析

更新起始点 C 在每一次写操作时，将写操作连同一致性矢量 $CV_C^d[C]$传送给所有的分布结点，由参与者 P 比较 $CV_P^d[C]$和 $CV_C^d[C]$，如果相等，则表明 P 已经执行了以 C 为更新起始点在数据 d 上的所有操作，P 将执行本次写操作，并且更新一致性矢量.否则，P 将流产这一事务并且退出，不再接收关于这个事务的任何操作.在事务执行结束时，C 以原子写方式提交更新事务和其他日志操作记录，然后向各个分布结点发出提交决定.对于各个分布结点，可能由于失效未参与更新或者执行了本地流产操作.这样会导致协调者不能收到相应的参与结点对提交消息的确认.对于这种数据的不一致性，需要通过随后的协调机制来取得复制数据的一致.当更新起始点发现某个参与者 P 没有对它的提交消息给予确认后，对应此事务更新的数据 O，在 Reco-Info 中加入一个元组(O,P).

算法 1 和算法 2 保证了网络和结点失效时数据更新的原子性.由于网络连接失效对于更新起始点来说等价于一个或者多个结点失效，因此，我们只分析结点失效的情况.(1) 如果更新提交前更新起始点失效，则在此结点失效恢复时，该事务的操作在更新起始点被撤销，而所有的参与者则因超时而执行流产操作；(2) 如果更新提交后更新起始点失效，则该事务的本地操作有效.在失效恢复时，更新事务的操作被重新执行，同时一致性矢量也被更新，起始点假定所有的参与结点流产，并且在 Reco-Info 表中记录.对于已经提交的结点，在数据协调时并不会被执行两次(见 2.3

节);(3) 如果参与者在提交前失效,则在失效恢复时,该事务的操作被撤销,同时更新起始点在等待确认消息时超时,然后假定该参与者失效并在 Reco-Info 中记录;(4) 如果参与者提交后失效,则在失效恢复时,该事务的操作被重新执行,并且修改一致性矢量.如果在发出确认消息前失效,则起始点记录了协调信息,操作在失效结点仍然不会被执行两次.总之在各种失效的情况下,参与者都不会被阻塞,也不必执行远程恢复,而是由更新起始点记录未正常确认的结点信息以后通过协调操作来达到一致.

1.3 正确性证明

我们通过证明下面的定理来证明更新操作事务的正确性.

定理 1. 更新事务保持了不变式 1.

证明:假定在某次更新事务提交之前保持了不变式 1,则在所有未提交更新事务的结点,与更新相关的数据值、历史日志和一致性矢量都保持不变,而与更新事务无关的数据值、历史日志和一致性矢量都保持不变.

假定某个结点 P 提交了一个更新事务,该事务涉及数据对象 O.由于在事务执行过程中,数据被锁定,除了这一更新事务,没有其他操作在此数据上面执行,由于一致性矢量中非起始点的分量未被改变,我们只需要证明:$a_c^o \in H_p \Leftrightarrow time(a_c^o) \leqslant CV_p^o[C]$成立即可(其中 C 是更新起始点).

在更新事务提交时,$CV_p^o[C]$等于该事务在 O 上的最后一次写操作的邮戳,结点 P 执行了这个更新事务,并且在执行第 1 个写操作之前,$CV_p^o[C]=CV_c^o[C]$,根据假设,在更新事务执行前不变式成立,$CV_c^o[C]$是在结点 C 本事务的最后一次操作时间,由于 P 提交了该事务,所以该事务的所有操作都包括在 H_p 中,除此事务之外,C 不再广播其他写操作,更新事务执行时它又锁定了数据,所以⇐满足.

由于 $CV_p^o[C]$的更新值就是在 O 上的最后操作的时间.在该操作加入日志之后,⇒显然成立. □

2 数据协调

各个复制服务器结点之间可能会由于失效而造成复制数据的不一致,数据协调的目的就是使得复制服务器的数据副本达到一致.协调在两个结点之间针对某个数据进行,其中一个结点在它的 Reco-Info 中有元组(O,P),是协调的发起点,另一个结点则是参与结点.协调算法的数据更新同样需要保证其事务特性.

2.1 协调算法

设结点 S 中有的 Reco-Info 中有元组(O,P),是协调发起点.

算法 3. 协调发起点 S 的协调算法

```
Begin
    从 Reco-Info 中删除(O,P);
    向结点 P 发送消息 ReconMess(S,CV_s^o,O);
    等待结点 P 返回 CV_p^o;
    调用 Reconcile(S,P,O,CV_p^o);
    Commit;
End
```

算法 4. 协调参与结点 P

```
Begin
    接收来自结点 S 的消息 ReconMess (S,CV_s^o,O);
    If P 结点的 Reco-Info 表中有记录(O,S)
        Then 删除之;
             向结点 S 发送 CV_p^o;
             调用 Reconcile(P,S,O,CV_s^o);
        Commit;
End
```

S 和 P 结点都需要调用协调过程来达到数据副本的一致,协调过程如下:

```
协调过程. Reconcile(X,Y,O,CV_x^o)
Begin
  Δ_{x,y}^o := {a_q^o ∈ H_x | time(a_q^o) > CV_x^o[y]};   /* 取出所有在 X 结点执行但未在 Y 结点执行的操作 */
  向结点 Y 发送 Δ_{x,y}^o;
  等待来自结点 Y 的 Δ_{y,x}^o;
  t_1 := Min({time(a_q^o) | a_q^o ∈ Δ_{y,x}^o});   /* 取 Δ_{y,x}^o 中最早的操作时间 */
  U_x^o := {a_q^o ∈ H_x | time(a_q^o) > t};   /* 需要撤销的操作 */
```

对 U_x^o 以邮戳为码执行逆排序；
撤销 U_x^o 中的所有操作；
对 $\Delta_{y,x}^o$ 以邮戳为码进行排序；
对 U_x^o 和 $\Delta_{y,x}^o$ 进行合并排序，将结果赋予 R_x^o；
按顺序执行 R_x^o 中的操作；
$H_x := H_x \cup \Delta_{y,x}^o$；　/* 将操作加入历史日志 */
$\forall q, CV_x^o[q] := Max(CV_x^o[q], CV_y^o[q])$；　/* 更新一致性矢量 */
End

2.2 算法分析

协调开始点 S 首先向参与协调的结点 P 发送关于数据 O 的一致性矢量，然后接收对方发回的一致性矢量．每个结点都用收到的一致性矢量扫描本地的日志，取出在其他结点未执行的操作，发送到另一个结点．当另一个结点收到这些操作时，则执行之并且加入本地日志．在数据协调的最后，更新一致性矢量并且提交．在一个结点向另一个结点传送了相关操作后，它们之间无需进一步通讯，可以是一个结点提交而另一个结点流产．

协调结果不受结点失效的影响．如果结点 X 流产了与结点 Y 的在数据 O 上的协调，则在失效恢复时，它在 X 结点的 Reco-Info 中加入元组 (O,Y)，只有当两个结点都进行提交后，与协调有关的信息才被删除．结点的失效如果在协调提交前发生，在失效恢复时，结点对本地协调执行流产，撤销所有的写操作；结点的失效如果在协调提交后发生，则失效恢复时，结点提交本地的协调操作，重新执行所有的写操作，这样，在失效环境下仍然能够正确执行数据协调．

2.3 正确性证明

下面我们证明算法的正确性．

定理2. 如果结点 X 提交了起始点 Y 关于数据 O 的协调，协调后在 X 结点的 O 值是在提交前 X 和 Y 两结点在 O 上执行的所有操作按邮戳顺序执行的结果．

证明：首先证明结点 Y 执行，而结点 X 未执行的操作在协调开始时被从 Y 结点送到 X 结点，即

$$a_q^o \in H_y \wedge a_q^o \notin H_x \Rightarrow a_q^o \in \Delta_{y,x}^o$$

由于在协调开始时满足不变式，因此考虑一个操作 a_q^o，满足 $a_q^o \in H_y \wedge a_q^o \notin H_x$，由于在协调开始时满足不变式1，所以 $CV_x^o[q] < time(a_q^o) \leqslant CV_y^o[q]$，于是 $a_q^o \in \Delta_{y,x}^o$．

下面证明在结点 X 已经执行的操作在协调过程中不会被 Y 结点传送过来再次执行，即在协调前，$a_q^o \in \Delta_{y,x}^o \Rightarrow a_q^o \notin H_x$．

由于 $a_q^o \in \Delta_{y,x}^o$，在协调前满足不变式，$CV_x^o[q] < time(a_q^o) \leqslant CV_y^o[q]$，于是有 $a_q^o \in H_y$，并且 $a_q^o \notin H_x$，由于在协调中，通过确定协调双方的所有动作的最小邮戳，然后以逆邮戳顺序撤销了所有邮戳比最小邮戳大的操作，对撤销和转来的操作以邮戳为码合并排序，并执行之，因而尽管在协调前各个结点的操作的执行顺序可能不同，但是在协调后具有相同的顺序．

2.4 协调时机

由于数据协调的开销很大，在执行数据协调时会影响整个系统的其他操作的处理．因此，复制服务器之间数据协调的时机应该仔细考虑，应该尽量避开系统正常服务的高峰期，利用夜间或者其他空闲时间进行．数据协调的调用时机可以分成即刻、阶段性、根据要求和全连接等几种方式：(1) 即刻：在比较一致性矢量发现不一致时，就执行协调机制；(2) 阶段性：在某个预定的时间点或者固定的时间间隔，对所有的可通讯的结点上的数据执行协调机制；(3) 根据要求，在被系统管理者或者用户要求使用时，执行协调机制；(4) 全连接：当所有结点都处在完全连接的系统状态时，执行对所有的结点和数据进行数据协调．

适当选择系统数据协调的时机和结点之间协调的顺序，是降低数据协调对系统性能影响的主要方法．

3 结束语

本文就 Client/Server 系统中多个复制服务器之间的数据更新问题，介绍了一种独立更新的无阻塞的方法，分析了此方法在失效情况下的性能．这种数据更新算法只要更新起始点的服务器不失效，它的数据更新就可以首先在此结点完成，向用户作出反应，并且同时在连接状态和结点状态正常的其他复制服务器完成复制更新．各个服务器在决定是否提交时可以进行单方面的选择，不必被迫等待，因而避免了阻塞．此方法适用于对于更新操作反应时间要求短，并允许延迟更新传播的 Client/Server 系统设计，它能够在结点失效时保证系统的服务响应时间的同时，通过数据协调来保证数据副本的一致性，因此，对于构造多服务器的高可用性的 Client/Server 系统具有实用意义．

参考文献

1 Sidell J, Aoki P, Sah A *et al*. Data replication in Mariposa. In: Su S Y W eds. Proceedings of the 12th International Conference on Data Engineering. New Orleans, USA: IEEE Computer Society, 1996

2 Gray J, Helland P, Neil P O *et al*. The dangers of replication and a solution. In: Jagadish H, Mumick I eds. Proceedings of ACM SIGMOD International Conference'96 on Management of Data. Montreal, Quebec, Canada. ACM SIGMOD, June 1996

3 Bratsberg S, Hvasshovd S, Torbjurnsen O. Location and replication independent recovery in a highly available database. In: Small C, Douglas P, Johnson R *et al* eds. Advances in Databases Proceedings of the 15th British National Conference on Databases. London, United Kingdom. Lecture Notes in Computer Science, Vol 1271, Springer, 1991

4 Barbard D, Garcia-Molina H. The demarcation protocol: a technique for maintaining linear arithmetic constraints in distributed database systems. In: Pirotte A, Delobel C, Gottlob G eds. Advances in Database Technology, Proceedings of the 3rd International Conference on Extending Database Technology. Vienna, Austria. Lecture Notes in Computer Science, Vol 580, Springer, 1992

5 Wai-Chee Fu A, Wai-Lok Cheung D. A transaction replication scheme for a replicated database with node autonomy. In: Bocca J, Jarke M, Zaniolo C eds. Proceedings of the 20th International Conference on Very Large Databases. Santiago de Chile, Chile: Morgan Kaufman Publishers, 1994

6 Rangarajan S, Setia S, Tripathi S K. A fault-tolerant algorithm for replicated data management. In: Golshani F ed. Proceedings of the 8th International Conference on Data Engineering. Tempe, Arizona: IEEE Computer Society, 1992

Non-blocking Data Updates Among Multiple Replicated Servers

ZHAO Hong-biao ZHOU Li-zhu

(*Department of Computer Science and Technology Tsinghua University Beijing* 100084)

Abstract The availability of server is a key factor in improving client/server system performance. The most promising approach in this regard is to use multiple replicated servers. However, the blocking problem is a bottleneck in replicated data update for this approach. In this paper, a non blocking independently replicated data update method and a reconciliation mechanism to reach agreement at those servers not committed because of the failure are presented.

Key words Client/server, data replication, data update, data reconciliation, failure recovery.