

一种基于数据相关性的优化数据一致性维护方法

周 婧 王意洁 李思昆

(国防科学技术大学计算机学院并行与分布处理国家重点实验室 长沙 410073)

摘 要 针对数据一致性中的数据相关性问题,提出一种优化的数据一致性维护方法.在该方法中,数据对象按固定大小分块,并以数据块作为数据管理的基本单位;数据更新利用 Bloom filter 技术压缩表示,并进行双路径传播;发起方和协商方在一致性维护过程中,分别调用各自的协商算法检测 and 解决更新冲突;动态数据管理算法调节数据更新过程中的动态数据块变化,对数据块进行合并或分解.模拟测试结果表明,在选取适当范围内的分块大小时,该方法在一致性维护开销、动态性和鲁棒性方面均具有较好的性能.文中给出了选定适当分块大小的指导性方法.

关键词 P2P 分布存储系统;数据复制;数据一致性;数据相关性;更新冲突

中图法分类号 TP393

An Optimistic Data Consistency Maintenance Method Based on Data Dependence

ZHOU Jing WANG Yi-Jie LI Si-Kun

(National Key Laboratory for Parallel and Distributed Processing, School of Computer,
National University of Defense Technology, Changsha 410073)

Abstract This paper proposes an optimistic data consistency method according to the question about data dependence in data consistency. In the method, data object is partitioned into data blocks by fixed size as the basic unit of data management. Updates are compressed by Bloom filter technique and propagated in double-path. Negotiation algorithms detect and reconcile update conflicts, and dynamic data management algorithms accommodate dynamic data processing. The results of the performance evaluation show that it is an efficient method to achieve consistency, good dynamic property, and strong robustness when choosing the size of data block appropriately. At the same time, a feasible way is put forward on how to choose appropriate data block size.

Keywords P2P distributed storage system; data replication; data consistency; data dependence; update conflict

1 引 言

利用 P2P 计算技术构建大规模分布存储系统,是当前 P2P 计算研究和应用的热点. P2P 系统对数据复制进行了广泛的研究^[1-2],先前的研究重点集中在大规模分布式系统的信息分布(副本放置)^[3-4]和

信息发现(副本定位)^[5-6]上.传统的 P2P 系统对信息共享所带来的数据一致性维护涉及的较少,最近的研究才将更多的注意力放在该问题上^[7-11].

弱一致性维护方法中存在数据相关性问题,而这些问题是更新日志中使用语义描述数据更新不可避免的.一致性维护中的数据相关性可能导致以下问题:(1)不必要的更新应用延迟;(2)不能保存不

收稿日期:2006-06-14;最终修改稿收到日期:2008-01-07. 本课题得到国家“九七三”重点基础研究发展规划项目基金(2002CB312105)、高等学校全国优秀博士学位论文作者专项资金项目(200141)、国家自然科学基金创新研究群体科学基金项目“千万亿次高性能计算关键技术”(60621003)、国家自然科学基金(69903011,60503042)资助.周 婧,女,1977 年生,博士,讲师,研究方向为网络计算、虚拟现实. E-mail: jingle77@126.com.王意洁,女,1971 年生,教授,博士生导师,研究领域为网络计算、数据库技术、移动计算.李思昆,男,1941 年生,教授,博士生导师,研究领域为虚拟现实与可视化、嵌入式系统与 SoC 设计方法.

同副本发出的更新之间的依赖关系;(3)对相互没有看到的更新简单地排序、应用,导致某个更新的结果使得另一更新的条件发生变化,违背用户提交更新的原始意图。

更新部分排序给每个更新附加上前提更新的名字^[12-13],更新执行必须等到所有前提更新执行之后,从而解决前两个问题。但是更新的大小随着它依赖的前提更新的增加可能会无限增加,并且不能解决第三个问题,因此并不是一种很有效的方法。为此,针对弱数据一致性中的数据相关性问题,提出一种面向 P2P 分布存储系统的基于数据相关性的优化数据一致性维护方法(Data dependence based optimistic data Consistency method in P2P distributed system, DACP)。

DACP 方法基于的 P2P 分布存储系统具有下列特点:(1)系统采用非结构化全分布式拓扑,资源的放置与系统的拓扑结构无关;(2)每个数据对象存在多个对等的副本,且初始状态时同一数据对象的副本是强一致的;(3)每个数据对象具有唯一的逻辑名(标识),且多个副本的逻辑名相同;(4)每个结点和数据对象的标识都是一个 m 位的二进制串,根据结点的 IP 地址或数据对象的关键字信息通过 SHA-1 算法获得。

DACP 从分析数据一致性中的数据相关性出发,对大型数据对象按固定大小分块并以数据块作为基本的数据管理单位;然后利用一种压缩技术表示数据更新;在结点间通过协商算法发现和解决更新冲突。DACP 可以在一定程度上避免数据一致性中的数据相关性,确保数据一致,并提高数据一致性的性能。

2 数据一致性中的数据相关性

首先给出 DACP 方法中所采用的更新语义描述形式:用户提交的对数据对象的更新往往包含多个对数据对象中不同数据内容的修改操作,将每项修改操作分别用语义描述,则更新为修改操作语义描述的集合;每个更新记为 $\mu(D)$,为二元组 $(\zeta(d), \eta)$ 的集合,其中: D 表示数据对象; $d \subseteq D$, d 为包含于 D 的数据内容; ζ 为更新条件, $\zeta(d)$ 表示 d 满足更新条件 ζ ; η 为更新结果,即将 d 修改后的数据内容。

数据共享性是分布存储系统的一个重要特征,在同一系统中必须允许有多个访问和更新存在且可以并发执行,数据一致性维护正是要协调并发更新

之间的冲突。通过分析发现,数据一致性中的数据相关性主要体现在“伪冲突更新”和“更新依赖”两个方面。

传统意义上的更新相互冲突是指:结点 N_1 和 N_2 对同一数据对象 O 分别发布了更新 u_1 和 u_2 ,更新 u_1 的发布时间早于更新 u_2 ,且结点 N_2 在发布更新 u_2 时没有看到更新 u_1 ,则更新 u_1 和 u_2 相互冲突。根据弱一致性维护方法,更新 u_2 必须等到更新 u_1 被提交之后才可以提交。但是在上述的更新冲突中存在伪冲突现象。针对 DACP 方法中所采用的更新语义描述方法,下面给出伪冲突更新(false-conflict updates)的形式化描述。

数据对象 D 的更新 $\mu(D)$ 和 $\mu'(D)$,如果 $\bigcup_{RGE(i)} (d_i) \cap \bigcup_{RGE(j)} (d'_j) = \emptyset$,则 $\mu(D)$ 和 $\mu'(D)$ 是伪冲突更新。其中:

$$\begin{aligned} \mu(D) &= \{(\zeta_1(d_1), \eta_1), (\zeta_2(d_2), \eta_2), \dots, \\ &\quad (\zeta_n(d_n), \eta_n)\}, \\ d_i &\subseteq D, RGE(i) = \{1, 2, \dots, n\}, \\ \bigcup_{RGE(i)} (d_i) &= \{d_0, d_1, \dots, d_n\}, \\ \mu'(D) &= \{(\zeta'_1(d'_1), \eta'_1), (\zeta'_2(d'_2), \eta'_2), \dots, \\ &\quad (\zeta'_m(d'_m), \eta'_m)\}, \\ d'_j &\subseteq D, RGE(j) = \{1, 2, \dots, m\}, \\ \bigcup_{RGE(j)} (d'_j) &= \{d'_0, d'_1, \dots, d'_m\}. \end{aligned}$$

对于上述描述,有几点需要说明:(1)如果以整个数据对象为基本更新操作单位,则并发的多个更新是相互冲突的更新;(2)降低数据更新操作单位的粒度,则原本冲突的更新实质上并不冲突,即是伪冲突的;(3)伪冲突更新之间不必完全依赖于建立的某种顺序关系,而可以与其它更新并发作用;(4)识别伪冲突更新在一定程度上避免因网络延迟或结点失效等不确定因素带来的某些更新被长时间延迟。

弱一致性维护方法松弛一致性要求确保副本的最终一致。弱一致性维护方法允许任何副本在任意时刻都可以发布更新,然后根据一定的规则(如时间戳)对更新排序,提交的更新按照排定的顺序依次作用到副本上。但是,弱一致性维护方法的设计应遵循“不改变用户提交更新的本意”的设计原则,因此必须解决更新依赖(update dependency)问题。

对数据对象 D 的伪冲突更新 $\mu(D)$ 和 $\mu'(D)$,如果有 $\zeta_i(\eta'_j)$ 或 $\zeta'_j(\eta_i)$ ($i \in \{1, 2, \dots, n\}, j \in \{1, 2, \dots, m\}$),则 $\mu(D)$ 和 $\mu'(D)$ 之间存在更新依赖。其中:

$$\begin{aligned}\mu(D) &= \{(\zeta_1(d_1), \eta_1), (\zeta_2(d_2), \eta_2), \dots, \\ &\quad (\zeta_n(d_n), \eta_n)\}, \\ \mu'(D) &= \{(\zeta'_1(d'_1), \eta'_1), (\zeta'_2(d'_2), \eta'_2), \dots, \\ &\quad (\zeta'_m(d'_m), \eta'_m)\}.\end{aligned}$$

假设更新 $\mu'(D)$ 在发布时刻未看到依赖更新 $\mu(D)$ 的更新结果. 根据更新发布时刻的意图, $\mu(D)$ 和 $\mu'(D)$ 作用到副本的结果应为序列 $\eta_1 \eta_2 \dots \eta_n \eta'_1 \eta'_2 \dots \eta'_m$. 如果单纯地对更新按照 $\mu\mu'$ 的顺序应用, 会导致更新作用后的副本为 $\eta_1 \eta_2 \dots \eta_{i-1} \eta'_j \eta_{i+1} \dots \eta_n \eta'_1 \eta'_2 \dots \eta'_m$ (假设 $\zeta'_j(\eta_i)$ 成立). 因此, 一致性维护方法不能一味地接收更新, 必须拒绝某些相关的更新.

3 数据分块和前序块更新

通过上述分析可知, 降低更新数据操作单位的粒度, 可以在一定程度上避免更新伪冲突; 另外, 为了解决更新依赖, 必须拒绝某些相关更新. 为此, 我们采用数据分块解决更新伪冲突问题; 引入前序块更新的思想并结合第 4.1 节中的协商算法解决更新依赖, 实现优化的数据一致性维护.

3.1 基于固定大小的数据分块

数据划分是一种将数据分割管理的技术, 主要针对海量数据对象, 被广泛应用于各种分布数据管理系统中^[14-16]. 分布存储系统可以根据数据对象所属资源的特征进行分块, 如地理数据根据经纬度划分, 文档根据原始章节划分等.

DACP 方法不具体地针对某类资源, 采用以固定大小为单位对数据对象分块: (1) 块的数量. 假设分块单位为 $unit$, 则大小为 S 的数据对象被分为 $b = \lceil S/unit \rceil$ 块; (2) 块编号. 被划分后的数据块的顺序编号为 $1, 2, \dots, b$, 且 $2^m \leq b < 2^{m+1}$; (3) 块标识. 每个数据块被分配长度为 $m+1$ 的二进制编码, 与块编号的顺序相对应依次为 $00\dots 0, 00\dots 1, \dots$.

下面的第 5.3 节中将给出根据具体 P2P 系统的要求, 确定最佳数据分块单位的方法.

这里对基于固定大小的数据分块, 有几点需要说明: (1) 假设每个数据对象的更新频率均为 f , 更新对数据对象的整体内容而言均匀分布, 则每个数据块上接收到的更新频率为 f/b . 将大型的数据对象分割成多个数据块, 然后对每个数据块分别进行更新冲突检测, 可以在一定程度上避免更新伪冲突的发生. 随着更新频率和伪冲突发生率的降低, 更新冲突率继而得到降低. (2) 假设每次用户提交的更新所涉及的数据内容相对集中, 即每个被提交更新

$\mu(D) = \{(\zeta_1(d_1), \eta_1), \dots, (\zeta_n(d_n), \eta_n)\}$ 中的 d_i 均属于同一数据块. 这种假设是合理的, 符合事物空间局部性原理, 另外在系统设计时也可控制实现. (3) 如果更新 $\mu(D)$ 中 $\forall_{i=1,2,\dots,n} (d_i \subset D_x) (x \in \{1, 2, \dots, b\})$, 则认为更新 μ 是数据块 D_x 上的更新, 记为 $\mu(D_x)$. (4) 更新标识通过公式 $U = \text{hash}(N \| D \| B \| \text{hash}(T))$ 获取. 其中: N 为更新发布结点标识, D 为数据对象标识, B 为数据块标识, T 为更新发布时钟; $\|$ 表示两个字符串的连接操作, 且 Hash 函数采用 SHA-1 算法.

3.2 前序块更新

DACP 方法中, 用户提交的对数据对象的更新有 3 种状态: 非提交 (uncommitted)、提交 (committed) 和废弃 (aborted). 图 1 给出更新的 3 种状态之间的相互转化过程.

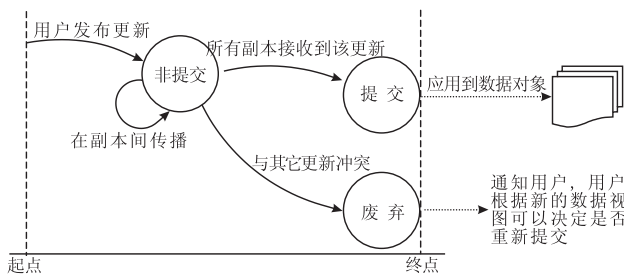


图 1 更新状态转换示意图

在数据一致性维护过程中, 用户不会暂停对数据的访问; 为了提高系统的可用性, DACP 方法不阻止更新提交, 并允许用户可以看到不一致的数据. 用户访问到的数据对象不是系统中存储的物理数据, 而是由物理数据应用本地存储的非提交更新导出的数据视图. 用户在此基础上提交新的更新, 则认为这组非提交更新与新提交更新有一定的关联性.

定义 1. 前序块更新 (Fore-Block-Update). 结点 N 上存储了数据对象 D 的副本和按照时间戳排序的非提交更新集合 $\{\mu_0, \mu_1, \dots, \mu_l\}$, 用户从结点 N 访问到数据对象 D 并提交新的更新 $\mu(D_x)$. 假设 $\{\mu_{i,0}, \mu_{i,1}, \dots, \mu_{i,y}\} \subseteq \{\mu_0, \mu_1, \dots, \mu_l\}$ 且 $\mu_{i,z}(D_x) (z = 0, 1, \dots, y)$, 则 $\{\mu_{i,0}, \mu_{i,1}, \dots, \mu_{i,y}\}$ 为更新 μ 的前序块更新序列, 记为 $FBU(\mu) = \{\mu_{i,0}, \mu_{i,1}, \dots, \mu_{i,y}\}$, 其中每个更新为更新 μ 的前序块更新.

3.3 基于 Bloom filter 技术的更新表示

Bloom filter 技术^[17] 是一种压缩表示技术, 通过具有较少存储开销的位向量来概率性地表示一个集合中的所有元素, 并根据位向量来判断集合是否包含某元素. BF 技术作为一种压缩表示技术得到了广泛应用^[18-20].

假设集合 $S = \{\mu_1, \mu_2, \dots, \mu_n\}$, 其中每个更新 μ_i 用其更新标识表示, 位向量 \mathbf{V} 的长度为 m , 使用 k 个值域为 $[0, m-1]$ 且相互独立的 Hash 函数 $hash_1, hash_2, \dots, hash_k$ 作为 Hash 函数族。

(1) $BF(S)$ ——用 \mathbf{V} 表示 S . 初始化时 \mathbf{V} 各位均为 0, 对 S 中的任意更新 μ_i , 分别计算 $hash_1(\mu_i) = h_1, hash_2(\mu_i) = h_2, \dots, hash_k(\mu_i) = h_k$, 将 \mathbf{V} 的第 h_1, h_2, \dots, h_k 位置为 1。

(2) $Func(\mu, \mathbf{V})$ ——判断集合 S 是否包含 μ . 构造函数 $Func(\mu, \mathbf{V})$ (式(1)), $Func(\mu, \mathbf{V}) = 0$ 表示 $\mu \notin S$; 否则 $\mu \in S$.

$$Func(\mu, \mathbf{V}) = \begin{cases} 0, & \exists h'_i (i \in \{1, 2, \dots, k\}) (\mathbf{V}[h'_i] = 0) \\ 1, & \forall h'_i (i \in \{1, 2, \dots, k\}) (\mathbf{V}[h'_i] = 1) \end{cases},$$

其中,

$$hash_1(\mu) = h'_1, \dots, hash_k(\mu) = h'_k \quad (1)$$

3.4 更新日志中的数据结构

DACP 方法在数据一致性的维护过程中, 跟踪副本和数据块的状态, 记录更新的传播轨迹和数据块的信息, 并组织成更新日志. 更新日志除了用于正常的更新传播中的信息交流之外, 当出现结点失效时, 还可以根据更新日志中记录的信息, 实现最终的数据一致. 更新日志中主要包含 3 个数据结构:

(1) 结点 N 对数据对象 D 的非提交更新表: $T_{ucn}(N, D)$.

结点 N 为本地存储的每个数据对象 D 建立 $T_{ucn}(N, D)$, 记录结点 N 看到且尚未得到提交的更新. $T_{ucn}(N, D)$ 是六元组 $(D_x, U, update, T, P, F)$ 的集合, 其中: $update$ 为更新体; T 为更新发布时钟; P 是更新传播路径上各结点 ID 的集合, $P[0]$ 默认为更新的发布结点; F 是前序块更新序列中的最新更新, 通过该属性可以在整个前序块更新序列中建立顺序关系。

非提交更新表根据属性组合 (D_x, T) 建立索引, 且对数据块 D_x 的更新集合表示为 $T_{ucn}(N, D)(D_x)$.

(2) 结点 N 对数据对象 D 的更新管理表: $T_{mca}(N, D)$.

结点 N 为本地存储的每个数据对象 D 的副本设立更新管理表 $T_{mca}(N, D)$, 记录各数据块上最新提交的更新以及最新废弃的更新序列 (AL), AL 用序列中的更新 $head_{\mu} | \forall \mu' \in AL (head_{\mu} \in FBU(\mu'))$ 表示. $T_{mca}(N, D)$ 是四元组 $(D_x, commit_U, abort_U, T(abort_U))$ 的集合, 其中: $commit_U$ 为数据块 D_x 最新提交更新的标识; $abort_U$ 为最新废弃更新序列中更新 $head_{\mu}$ 的标识, $T(abort_U)$ 为更新 $head_{\mu}$

的发布时钟。

(3) 系统对数据对象 D 的数据块管理表: $T_{dbm}(D)$.

系统为每个数据对象 D 设立数据块管理表 $T_{dbm}(D)$, 记录数据对象被划分为多个数据块的标识及其大小, 结点复制数据对象到本地的同时复制该表信息. $T_{dbm}(D)$ 是三元组 $(i, B(D_x), S_x)$ 的集合, 其中: i 为数据块编号, $B(D_x)$ 为数据块 D_x 的标识, S_x 为数据块 D_x 的体积。

4 基于数据相关性的优化数据一致性维护

优化数据一致性维护方法 DACP 的提出主要是为了解决弱一致性维护方法存在的数据相关性问题, 方法所基于的 P2P 分布存储系统采用非结构化全分布式拓扑, 数据资源的放置通常与 P2P 系统的拓扑结构无关, 且所有结点都是完全平等的, 系统中没有任何目录服务器。

DACP 方法中一致性维护的过程为

(1) 结点 N 从用户或其它结点接收到更新集合 $S(\forall \mu' \in S(\mu'(D_x)))$, 将 S 写入本地更新日志中的非提交更新表;

(2) 获取 S 中的更新 $\mu(D_x) | \forall \mu' \in S(T(\mu(D_x)) < T(\mu'(D_x)))$; 从 $T_{ucn}(N, D)(D_x)$ 获取 $\mu(D_x)$ 的前序块更新序列 $FBU(\mu) = \{\mu_1, \mu_2, \dots, \mu_i\}$, 计算 $BF(FBU(\mu))$;

(3) 假设结点 N 已知的数据对象 D 的副本集合为 RS , 结点 N 获取与集合 $RS-Path(\mu)$ 中结点之间的网络延迟, 得到与结点 N 之间网络延迟最小 (min_node) 以及最大的结点 (max_node);

(4) 将非提交更新表中更新 $\mu(D_x)$ 的属性 P 的值修改为 $P \cup \{min_node, max_node\}$;

(5) 将 $U(\mu)$, $BF(FBU(\mu))$ 以及 $T_{mca}(N, D)(D_x)$ 等信息通过消息 $SendTentMess$ 主动推 (push) 给结点 min_node 和 max_node , 以期获得较好的更新传播速度及传播的广域性;

(6) 结点 N 和结点 min_node (或 max_node) 之间协商解决更新冲突, 具体见 4.1 节;

(7) 如果任意结点 (结点 N , min_node 或 max_node) 接收到新的更新, 继续按照 (1)~(6) 的步骤传播更新。

下面对 DACP 一致性维护过程中的协商解决算法、数据块动态维护算法以及异常处理等关键技术

术进行具体阐述。

4.1 更新冲突的协商解决算法

结点 N 将 $U(\mu)$ 、 $BF(U(FBU(\mu)))$ 以及 $T_{mca}(N, D)(D_x)$ 信息传播给其它结点(假设为结点 N')，则在两个结点之间建立起一种协商(negotiation)关系, 其中结点 N 为发起方(initiator), N' 为协调方(coordinator). 更新冲突需要在发起方和协调方之间协商解决。

一旦接收到发起方发送的消息, 协调方调用“协调方更新冲突解决算法”(图 2)进行处理, 并根据不同的情况返回相应的响应; 接收到协调方响应的发起方调用“发起方更新冲突解决算法”(图 3)针对不同的响应分别进行处理; 其间为了检测相关更新以及维护副本一致, 需要在发起方和协调方之间进行多次交互. 协商处理的情况分为以下 3 种:

(1) 协调方已经接收到更新 μ , 这是因为更新沿着多条不同的路径传播, 因此协调方合并更新传播路径, 并返回发起方; 发起方相应地修改更新传播路径。

(2) 协调方和发起方之间存在相关更新 ($\exists \mu' \in T_{ucc}(N', d_{id})(b_{id}) (Func(\mu', bf) = 0)$), 协调方返回第一个被怀疑的相关更新. 根据发起方冲突解决算法, 如果废弃协调方上的更新, 则发起方向协调方发送其未看到的更新; 如果废弃发起方上的更新, 则协调方在接收到该废弃消息时, 同样向协调方发送其未看到的更新。

(3) 协调方上的所有更新都已经被发起方接收到 ($\forall \mu' \in T_{ucc}(N', d_{id})(b_{id}) (Func(\mu', bf) = 1)$), 协调方利用 Bloom filter 技术表示本地的更新, 并返回发起方; 发起方判断并传播本地存储的且协调方未接收到的更新。

```

ProcessTentatMess(DataID  $d_{id}$ , BlockID  $b_{id}$ , UpdateID  $u$ , UpdatePath  $p$ , BF  $bf$ , CommitUpdateID  $c$ , AbortUpdateID  $a$ )
// 结点  $N'$  (协调方) 从结点  $N$  (发起方) 接收到 SendTentatMess 消息
for  $c\mu, a\mu$  where  $(b_{id}, c\mu, a\mu, t) \in T_{mca}(N', d_{id})$  // 根据式(1)判断以  $c\mu$  或  $a\mu$  为标识的更新是否在  $bf$  所表示的集合中
do if  $c\mu \neq c(a\mu \neq a)$ 
    then if  $Func(c\mu, bf) = 1 (Func(a\mu, bf) = 1)$  // 结点  $N$  没有接收到最新的提交(或废弃消息)
        then 通知结点  $N$  提交更新  $c\mu$  (或废弃更新  $a\mu$ );
        else 结点  $N'$  提交更新  $c\mu$  (或废弃更新  $a\mu$ );
if  $\exists \mu' \in T_{ucc}(N', d_{id}) (U(\mu') = u)$  // 结点  $N'$  接收到复制的更新消息
then  $P(\mu') \leftarrow P(\mu') \cup p$ ;  $unit\_P \leftarrow P(\mu')$ ; // 合并更新传播路径
if  $P(\mu')$  包含所有的副本
then 更新  $\mu'$  被提交, 并通知所有副本提交更新  $\mu'$ ;
else RespondTentatMess(1,  $unit\_P$ ); // I 型消息应答
if  $\exists \mu'' \in T_{ucc}(N', d) (\mu' \in FBU(\mu''))$  then SendUpdates( $N', N, d_{id}, b_{id}, U(\mu')$ );
return;
flag  $\leftarrow 0$ ;
for each  $\mu' \in T_{ucc}(N', d_{id})(b_{id})$ 
do if  $Func(\mu', bf) = 0$  // 判断更新  $\mu'$  是否已经被结点  $N$  接收到
then flag  $\leftarrow 1$ ;  $corre\_mu \leftarrow \mu'$ ;  $fu \leftarrow U(\mu')$ ; break;
if flag = 1
then RespondTentatMess(2,  $U(corre\_mu)$ ,  $T(corre\_mu)$ ,  $F(corre\_mu)$ ); // II 型消息应答
if 结点  $N'$  接收到结点  $N$  发送的 AbortUpdates( $abort\_mu$ ) 消息 then SendUpdates( $N', N, d_{id}, b_{id}, fu$ ); // 结点  $N$  接收到结点  $N'$  上所有的更新
if flag = 0 // 结点  $N$  接收到结点  $N'$  上所有的更新
then  $SS = \{\mu' | \mu' \in T_{ucc}(N', d_{id})(b_{id})\}$ ; RespondTentatMess(3,  $BF(SS)$ ); // III 型消息应答
SendUpdates(NodeID  $N_1$ , NodeID  $N_2$ , DataID  $d$ , BblockID  $b$ , UpdateID  $id$ ) // 结点  $N_1$  向结点  $N_2$  发送更新集合
for  $T_{ucc}(N_1, d)(b)$ 
do  $S \leftarrow \{id\} \cup \{\mu | id \in FBU(\mu)\}$ ;
for each  $\mu \in S$  do  $P(\mu) \leftarrow P(\mu) \cup \{N_2\}$ ; 将集合  $S$  中的更新发送给结点  $N_2$ ;

```

图 2 协调方更新冲突解决算法

```

ProcessRespond(Int  $respondtype$ ) // 结点  $N$  接收到结点  $N'$  返回的 RespondTentatMess
switch( $respondtype$ ) {
case 1:  $P(\mu) \leftarrow unit\_P$ ; break; // 结点  $N'$  返回  $unit\_P$ 
case 2:  $\exists \mu'' \in T_{ucc}(N, d) (F(\mu'') = F(corre\_mu))$  // 更新  $corre\_mu$  与更新  $\mu''$  怀疑为相关更新, 且  $\mu''$  一定存在
    if  $T(\mu'') < T(corre\_mu)$ 
    then 向所有副本发送 AbortUpdates( $corre\_mu$ ) 的消息; // 废弃较晚发布的更新
        SendUpdates( $N, N', d_{id}, b_{id}, U(\mu'')$ ); // 向结点  $N'$  发送  $\mu''$  以后的更新
    else  $S \leftarrow \{\mu | \mu \in T_{ucc}(N, d) \wedge \mu'' \in FBU(\mu)\}$ ; //  $S$  中包含了所有以更新  $\mu''$  为前序块更新的更新
        废弃集合  $S$  中的所有更新;
         $abort\_mu \leftarrow \mu''$ ; 向所有的副本发送废弃更新 AbortUpdates( $abort\_mu$ ) 的消息;
case 3: for each  $\mu' \in T_{ucc}(N, d_{id})(b_{id})$ 
    do if  $Func(\mu', BF(SS)) = 0$  then  $fu \leftarrow U(\mu')$ ; break;
    SendUpdates( $N, N', d_{id}, b_{id}, fu$ );
}

```

图 3 发起方更新冲突解决算法

对于更新冲突的协商解决算法,有以下几点说明:(1)数据对象 D 同一数据块 D_i 上的更新 $\mu(D_i)$ 和 $\mu'(D_i)$, $FBU(\mu) = \{\mu_0, \mu_1, \dots, \mu_t\}$, $FBU(\mu') = \{\mu'_0, \mu'_1, \dots, \mu'_t\}$. 假设 $t < t'$, 如果 $\{\mu_0, \mu_1, \dots, \mu_t, \mu\} \subseteq \{\mu'_0, \mu'_1, \dots, \mu'_t\}$ 且 $\mu_0 = \mu'_0, \mu_1 = \mu'_1, \dots, \mu_t = \mu'_t, \mu = \mu'_{t+1}$, 则 DACP 认为更新 μ 和 μ' 不冲突, 否则冲突;(2)在协商过程中,常常会向其它副本发送更新废弃消息或更新提交消息. 这两类消息只包含废弃或提交更新的更新标识,不会占用过多的网络资源,却可以加速副本之间的一致性过程并避免用户提交无效的更新. 这是因为:已经决定被废弃的更新仍然存在于某副本的非提交更新表中,导致用户访问到不正确的数据视图.(3)如果结点 N'' 接收到废弃更新 $\mu'(D_x)$ 的消息,则废弃集合 $\{\mu | \mu \in T_{\text{uc}}(N'', D) \wedge \mu' \in FBU(\mu)\}$ 中的更新,并修改 $T_{\text{mc}}(N'', D)$ 中的数据项 $(D_x, c\mu, a\mu, T(a\mu))$ 为 $(D_x, c\mu, U(\mu'), T(U(\mu')))$.(4)如果结点 N'' 接收到提交更新 $\mu'(D_x)$ 的消息,则将更新 μ' 应用到本地副本;将本地更新 μ'' 的属性 $F(\mu'')$ 设置为空 ($\mu'' \in T_{\text{uc}}(N'', D) \wedge \mu' = F(\mu'')$);修改 $T_{\text{mc}}(N'', D)$ 中的数据项 $(D_x, c\mu, a\mu, T)$ 为 $(D_x, U(\mu'), a\mu, T)$.(5)协商过程结束之后,废弃“疑似”相关更新,两个副本结点上的非提交更新表 $T_{\text{uc}}(N, D)$ 和更新管理表 $T_{\text{mc}}(N, D)$ 中关于数

据块 D_x 的所有信息一致.

4.2 数据动态合并和分解算法

DACP 方法中,数据对象在初始状态一致的情况下根据固定大小被分块,之后各个数据块相对独立,用户提交的更新被分配到各个数据块上执行. 经过反复的更新操作,有些数据块的体积可能小于预先设定值,而有些可能严重超过预先设定值. 这时如果仍然将其作为一个数据块管理,则失去数据分块带来的效益,因此必须进行数据块的合并或分解.

数据块的动态划分(合并或分解)需要调整或重新分配数据块标识,维护数据块编号以便整个数据对象的恢复,另外还需要相应地修改更新管理表和调整非提交更新表.

数据块合并. 如果数据块 D_x 的体积小于阈值 (DACP 中设定为 $\text{unit}/2$), 则启动数据块合并算法 MergeBlock(图 4). 当 D_x 不是块编号为 1 的数据块时,总是试图与相邻的块编号小的其它数据块合并,直到合并后的数据块的体积 sum 满足 $\text{sum} \geq \text{unit}$; 否则再尝试与相邻的块编号大的数据块合并. 如果 $\text{unit} \leq \text{sum} < 2 \times \text{unit}$, 则将所有参与 MergeBlock 算法的数据块合并为一个数据块; 否则, $\text{sum} \geq 2 \times \text{unit}$, 则将合并后的数据块继而分解为两个数据块.

```

MergeBlock(DataBlock  $D_x$ )
   $sn \leftarrow \text{GetBlockSN}(T_{\text{dbm}}(D), D_x);$  // 获取  $D_x$  的编号
   $sum \leftarrow \text{GetBlockSize}(T_{\text{dbm}}(D), D_x);$  // 获取数据块  $D_x$  的体积
   $number \leftarrow \text{GetBlockNumber}(T_{\text{dbm}}(D));$  // 获取整个数据对象被划分成的数据块的数量
   $\text{unite}[number-1] \leftarrow 0; k \leftarrow 0;$  // 数组 unite 中记录与数据块  $D_x$  合并的数据块的顺序编号
  if  $sn \neq 1$  // 数据块  $D_x$  不是数据对象被划分后的第 1 块数据块
    then  $j \leftarrow 1;$ 
    do  $sum \leftarrow sum + \text{GetBlockSize}(T_{\text{dbm}}(D), sn-j);$  // GetBlockSize 获取顺序编号为  $sn-j$  的数据块的体积
       $j \leftarrow j+1; \text{unite}[k] \leftarrow sn-j; k \leftarrow k+1;$ 
    until  $sum \geq \text{unit}$  or  $sn-j=0;$  // 数据块  $D_x$  试图依次与其相邻的块编号小的数据块合并
  if  $sum < \text{unit}$  or  $sn=1$ 
    then  $j \leftarrow 1;$ 
    do  $sum \leftarrow sum + \text{GetBlockSize}(T_{\text{dbm}}(D), sn+j);$  // 数据块  $D_x$  试图依次与其相邻的块编号大的数据块合并
       $j \leftarrow j+1; \text{unite}[k] \leftarrow sn+j; k \leftarrow k+1;$ 
    until  $sum \geq \text{unit}$  or  $sn+j > number;$ 
  if  $sum < \text{unit}$  // 整个数据对象的体积都不足 unit
    then UpdateBlock( $T_{\text{dbm}}(D), (1, B, S), (1, B, sum)$ )
    // 整个数据对象为一个数据块, B 和 S 分别为  $T_{\text{dbm}}(D)$  中后两个属性上的值
  for each  $(i, B, S) \in T_{\text{dbm}}(D) \wedge i \geq 2$ 
    do UpdateBlock( $T_{\text{dbm}}(D), (i, B, S), (0, B, 0)$ ); // 将数据块管理表中其它项目均设置为闲置状态
  if  $\text{unit} \leq sum < 2 \times \text{unit}$  // 数据块  $D_x$  与 unite[number-1] 中的数据块合并为一个数据块
    then  $\text{min\_sn} \leftarrow \text{MIN}(\text{unite}[k]); \text{max\_sn} \leftarrow \text{MAX}(\text{unite}[k]);$  // 与  $D_x$  合并的最小块编号和最大块编号的数据块
    UpdateBlock( $T_{\text{dbm}}(D), (\text{min\_sn}, B, S), (\text{min\_sn}, B, sum)$ ); // 将最小块编号对应的编码分配给合并后的数据块
    for each  $tsn \leftarrow (sn, \text{unite}[k]) / \text{min\_sn}$  // tsn 的取值为 sn 以及 unite 数组中除去 min_sn 的所有非零值
      do UpdateBlock( $T_{\text{dbm}}(D), (tsn, B, S), (0, B, 0)$ ); // 将块编号为 tsn 的编码设置为闲置状态
    for  $j = \text{max\_sn} + 1$  to  $number$ 
      do UpdateBlock( $T_{\text{dbm}}(D), (j, B, S), (j - (\text{max\_sn} - \text{min\_sn}), B, S)$ ) // 维护数据块的顺序编号
  if  $sum \geq 2 \times \text{unit}$  // 需要将合并后的数据块分解为两个数据块
    then UpdateBlock( $T_{\text{dbm}}(D), (\text{min\_sn}, B, S), (\text{min\_sn}, B, sum/2)$ ); // 分解后的数据块编号为 min_sn 和 min_sn+1
    UpdateBlock( $T_{\text{dbm}}(D), (\text{min\_sn}+1, B, S), (\text{min\_sn}+1, B, sum/2)$ );
    for each  $tsn \leftarrow (sn, \text{unite}[k]) / (\text{min\_sn}, \text{min\_sn}+1)$  do UpdateBlock( $T_{\text{dbm}}(D), (tsn, B, S), (0, B, 0)$ );

```

图 4 数据块动态合并算法

对于数据块动态合并算法,有以下几点需要说明:(1)数据块动态合并算法将由于合并而闲置的数据块编码在数据块管理表中进行标记——将其块编号位设置为 $i=0$,以便于后续数据块分解时子数据块编码的获取。(2)数据块的合并最多在 3 个数据块之间进行,数据块的合并是可控制的,不会带来过量的计算开销。这是因为:假设需要合并不止 3 个数据块 D_x, D_j 和 D_k ,则有 D_x, D_j 和 D_k 的体积之和 $sum' < unit$,因而必有 D_j 或 D_k 的体积小于 $unit/2$ 。如果 D_j 或 D_k 的体积小于 $unit/2$,则 D_j 或 D_k 已经先于 D_x 启动 MergeBlock 算法。(3)假设数据块合并是由于数据块 D_x 提交应用了更新 μ 所引发的,则无论是合并为一个数据块还是合并且分解为两个数据

块,其在更新管理表中的 $commit_U$ 属性均设置为 $U(\mu)$, $abort_U$ 属性设置为参与合并的所有数据块在该属性对应的更新发布时间较早的更新标识;参与合并的全部数据块上的未提交更新代入之后的数据块。

数据块分解. 如果数据块 D_x 体积超过了阈值 (DACP 中设定为 $2 \times unit$),则启动数据块分解算法 DivideBlock (图 5),将数据块 D_x 分解为两个数据块 $D_{x,0}$ 和 $D_{x,1}$ 。假设数据块 D_x 的块标识为 $z_0 z_1 z_2 \dots z_t$,分解后各数据块的体积均为 ds ,算法首先在数据块管理表中查找有没有闲置的数据块编码,否则在原编码长度的编码范围内顺序分配编码;如果原编码长度可产生的编码被分配完毕,从而导致数据块的编码增加一位。

```

DivideBlock (DataBlock  $D_x$ )
 $B(D_{x,0}) \leftarrow B(D_x); flag \leftarrow 0;$  // 分裂后的前半部分数据所构成的数据块沿用原数据块的编码
 $sn \leftarrow GetBlockSN(T_{dbm}(D), D_x); number \leftarrow GetBlockNumber(T_{dbm}(D));$ 
for  $j = sn+1$  to  $number$ 
do UpdateBlock( $T_{dbm}(D), (j, B, S), (j+1, B, S)$ ); // 将编号在数据块  $D_i$  之后的所有数据块编号加 1
for each  $(i, B, S) \in T_{dbm}(D)$ 
do if  $i=0$  then  $flag \leftarrow 1; TB \leftarrow B$ ; break;
if  $flag=1$ 
then  $B(D_{x,1}) \leftarrow TB;$  // 存在空闲编码,并将该编码分配给分裂后的后半部分数据所构成的数据块
else if  $number=2^t+1$  // 不存在空闲编码且现有编码长度内的编码已经全部被分配给数据块
then  $B(D_{x,1}) \leftarrow 10 \dots 0;$  //  $B(D_{x,1})$  的编码长度为  $t+2$ 
for  $j=1$  to  $number$  do UpdateBlock( $T_{dbm}(D), (j, B, S), (j, 0B, S)$ ); // 将全部已有数据块的编码增加 1 位
else  $B(D_{x,1}) \leftarrow (number+1)_2;$  // 顺序获取编码,块编码为  $number+1$  对应的二进制串
DeleteBlock( $T_{dbm}(D), (sn, B(D_x), 2 \times ds)$ ); // 删除原数据块
AddBlock( $T_{dbm}(D), (sn, B(D_{x,0}), ds), (sn+1, B(D_{x,1}), ds)$ ); // 添加两个新分裂出的数据块

```

图 5 数据块动态分解算法

对于数据块动态分解算法,有以下几点需要说明:(1)在数据块分解结束之后,各数据块的编码长度都保持一致;并且按照数据块编号可以还原整个数据对象。(2)假设分解之前数据块 D_x 在更新管理表中的项目为 $(D_x, cu, au, T(au))$,则分解之后删除该项目并增加两个新项目 $(D_{x,0}, cu, au, T(au))$ 和 $(D_{x,1}, cu, au, T(au))$;原数据块上的非提交更新将同时作为两个子数据块的非提交更新。

4.3 结点失效和恢复的处理

由于 DACP 方法中,全部副本接收到更新之后更新才被提交,结点失效将导致:(1)更新较长时间地被延迟提交;(2)非提交更新表体积增大;(3)利用 Bloom filter 技术表示结点上非提交更新时计算开销增加;(4)增加协调方或发起方上更新冲突检测的计算开销(增加了进行比对的非提交更新的数量)。

当发起方结点 N 在获取各副本结点的网络延迟时,如果没有从远程结点 N_k 接收到响应,则通过

周期性的心跳检测 N_k 的存活状态;如果发现 N_k 超过一定时间阈值 T_0 一直没有响应,则认为该结点已经失效。失效结点在更新提交时不被考虑在内,即该结点被直接加入更新的“传播路径”属性中,“伪装”成该结点已经接收到更新。

失效结点 N_k 恢复时,定位到附近某个活动的副本结点 N_m ,通过比较两个结点上的更新管理表和数据块管理信息,发现并复制在结点失效期间发生变化的数据块,避免结点失效恢复时复制整个数据对象所带来大量开销。

(1) 比较 $T_{mca}(N_m, D)$ 与 $T_{mca}(N_k, D)$,获取块集合 C_1 , C_1 中包含的数据块在结点 N_k 失效期间提交了新的更新;另外,数据块合并是由数据对象提交了新的更新所引发,因此块合并而导致变化的数据块包含在集合 C_1 中。

$$C_1 \leftarrow \{D_j \mid (D_j, cu, au, T(au)) \in T_{mca}(N_m, D) \wedge (D_j, cu, au', T(au')) \notin T_{mca}(N_k, D)\}.$$

(2) 比较结点 N_m 和 N_k 的 $T_{dbm}(D)$,获取块集合

C_2 . C_2 中包含在 N_k 失效期间进行块分解之后的数据块.

$$C_2 \leftarrow \{D_j \mid (D_j \in T_{\text{dbm}}(D)(N_m) \wedge D_j \notin T_{\text{dbm}}(D)(N_k))\}.$$

(3) $C \leftarrow C_1 \cup C_2$, 结点 N_k 复制集合 C 中的数据块以及对应的非提交更新. 结点 N_k 根据与定位到的其它副本结点之间的网络延迟, 从网络延迟较小的多个副本结点并行复制各个数据块, 从而缩短结点 N_k 的恢复过程.

5 模拟测试

OptorSim^[21]是一个用于测试复制算法的模拟器. OptorSim 模拟的分布存储系统由一个资源代理器和多个存储结点构成. 资源代理器负责接收外部数据请求, 并将数据请求发送到各个存储结点. 每个存储结点包括计算单元、存储单元和副本管理器, 具有一定的计算能力和存储能力, 副本管理器负责副本的选择和动态调整.

我们基于该模拟器进行性能测试. 在我们的性能测试中模拟了 1000 个结点的 P2P 分布存储系统, 系统中共有 1000 个数据对象, 根据数据对象的大小对其进行分类, 见表 1. 我们从 DACP 方法的动态特性、一致性开销、分块粒度以及方法的鲁棒性 4 个方面进行模拟测试.

表 1 数据对象分类		
类别	大小/MB	概率/%
DT_1	1	20
DT_2	10	30
DT_3	100	30
DT_4	1000	20

5.1 更新的动态特性

我们在下面实验中假设数据对象以 B_x 为单位进行分块管理, 并假设当数据块小于 $\lceil B_x/2 \rceil$ 时启动数据块合并算法 MergeBlock, 当数据块大于 $2B_x$ 时启动数据块分解算法 DivideBlock. 这里将更新按其操作类型分为数据增量 (increment)、数据修改 (update) 和数据减量 (decrement) 三类, 每类更新又根据其所涉及数据内容的大小进一步地细分, 如表 2 所示.

我们针对各种分块大小 B_x 以及数据对象具有不同数量副本的多种情况下, 进行模拟测试. 在每组模拟实验的初始时刻, 每个数据对象产生固定数量的副本, 并将这些副本分散到随机选择的结点上; 在每组实验进行过程中, 提交 5000 个更新, 模拟运行结束条

件为完成 5000 个更新的传播, 所有副本一致, 且所有的数据块大小在 $\lceil B_x/2 \rceil \sim 2B_x$ 之间. 针对 5000 个更新只包含修改操作类型的静态情况 (static- B_x) 和包含所有操作类型的动态情况 (dynamic- B_x), 对 B_x 取值 5KB, 10KB, 20KB, 50KB, 100KB 和 1MB 分别进行模拟测试, 比较平均更新传播时间.

表 2 更新分类

类别	操作类型	大小/KB	概率/%
$UT_{1,1}$	增量	$\lceil B_x/10 \rceil$	11.1
$UT_{1,2}$	增量	$\lceil B_x/2 \rceil - 1$	11.1
$UT_{1,3}$	增量	$B_x - 1$	11.1
$UT_{2,1}$	修改	$\lceil B_x/10 \rceil$	11.1
$UT_{2,2}$	修改	$\lceil B_x/2 \rceil - 1$	11.2
$UT_{2,3}$	修改	$B_x - 1$	11.1
$UT_{3,1}$	减量	$\lceil B_x/10 \rceil$	11.1
$UT_{3,2}$	减量	$\lceil B_x/2 \rceil - 1$	11.1
$UT_{3,3}$	减量	$B_x - 1$	11.1

从实验结果 (图 6) 可以看出: 动态情况下比静态情况下的平均更新传播时间只有稍许增加, 说明数据块的合并操作和分解操作的计算量并不会影响系统性能, 这里只给出了 $B_x=20\text{KB}$ 时的测试数据, 其它分块大小下结论相同, 故给予忽略. 因此, 在下面的实验中将不再区分更新操作类型.

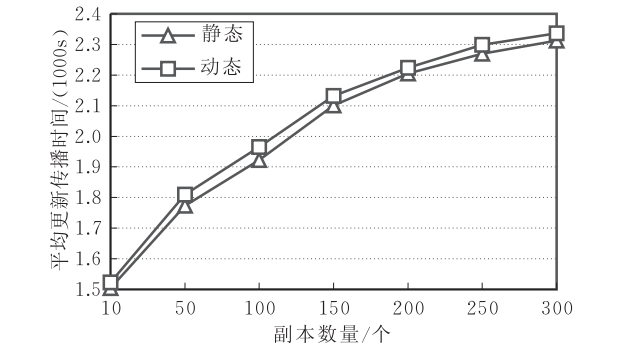


图 6 数据动态划分对更新传播时间的影响 ($B_x=20\text{KB}$)

5.2 一致性开销

我们模拟测试了在副本数量固定的情况下, DACP 方法和 Bayou^[13]、UPTReC 算法^[11]以及我们设计的 CWLM 方法^[22]在一致性维护方面的开销, 并且针对不同的分块大小 $B_x=5\text{KB}$, $B_x=10\text{KB}$, $B_x=20\text{KB}$, $B_x=50\text{KB}$ 和 $B_x=100\text{MB}$, 测试 DACP 方法的性能.

Bayou 和 CWLM 均采用宿主确认方法, 通过反熵 (anti-entropy) 传播更新. 反熵是一种一致性协调机制, 每个结点阶段性地与选取的另一结点进行一致性协调, 直到两个结点上的数据对象一致. 根据反熵中会话对象的选择方法, 我们将 Bayou 分别采用

随机选择(random)、基于延迟选择(latency-biased)、CWLM 以及 UPTReC 算法四种方法与 DACP 进行对比测试。UPTReC 算法用于维护全分布非结构化 P2P 系统中的文件一致性,每个文件的所有副本组成逻辑副本链,每个副本维护其 k 个最近的副本信息(包括副本 ID 和 IP 地址);CWLM 方法基于更新提交的顺序或本地更新日志信息选择会话结点。

初始状态时,每个数据对象产生固定数量的副本,并将其分散到随机选择的结点上。

(1) 副本数量固定情况下的一致性维护开销

我们对每种方法在不同的副本数量下分别进行一组模拟实验,每组模拟实验中提交 5000 个更新,模拟运行结束条件同 5.2.1 节。

从实验结果(图 7)可以看出:在副本数量固定且状态稳定的情况下,UPTReC 算法的性能最佳,因为 UPTReC 算法中每个副本维护其 k 个最近的副本信息,通过副本链 push 更新给所有在线的其它副本,但是 UPTReC 算法没有考虑副本链的维护代价,并且没有解决更新冲突。当副本数量较小时,CWLM 方法性能最佳,随机方法次之。随着副本数量的增多,DACP 方法由于在更新冲突检测中废弃了部分更新,也就相应地减少了传播的更新数量,因此其更新传播开销相对较少;对于 $B_x=10\text{KB}$, $B_x=20\text{KB}$ 和 $B_x=50\text{KB}$ 的 3 种情况下,随着副本数量的增多,其更新传播开销的变化趋势与其它情况相比趋于平稳,性能与其它方法相当或优于其它方法;但是 $B_x=5\text{KB}$ 和 $B_x=100\text{KB}$ 的情况下,性能较差,这是因为数据对象分块的粒度小,系统中所要管理的子块太多,对每个子块都要分别维护,带来了大量

的开销;数据对象分块的粒度大,对于每个数据块来讲,计算和通信开销大,性能并不高,并且数据块太大,会拒绝很多“伪冲突”更新。

(2) 副本数量动态变化情况下的一致性维护开销

我们模拟了在副本数量动态变化的情况下,DACP 方法和具有可比性的基于延迟选择方法、CWLM 方法以及 UPTReC 算法在系统维护上的开销,这里的系统维护开销包含一致性维护和创建新副本两个部分。

为了模拟副本数量的动态变化,我们采用根据访问频率增加和撤消副本的复制策略,且 DACP 在增加副本时从多个副本并行复制数据块。在每组模拟实验的初始时刻,每个数据对象产生固定数量的副本,并将这些副本分散到随机选择的结点上。在实验进行过程中,除了提交 5000 个更新之外,还另外提交 10000 次随机访问请求,模拟运行结束条件同上。

从实验结果(图 8)可以看出:在副本数量动态变化时,DACP 方法在 $B_x=20\text{KB}$ 时的性能要优于基于延迟选择方法和 CWLM 方法,因为相对于更新传播而言,这时创建新副本所带来的系统开销成为主要矛盾,DACP 方法中因为数据对象按照数据块划分,使得多个副本并行复制变得非常简单,副本增加时不会因为某两个结点间大量数据的传播而导致网络拥挤。另外,DACP 方法在副本数量较少的情况下性能低于 UPTReC 算法,因为这时由于复制而带来的数据传输不会导致局部网络状况变差;但是,

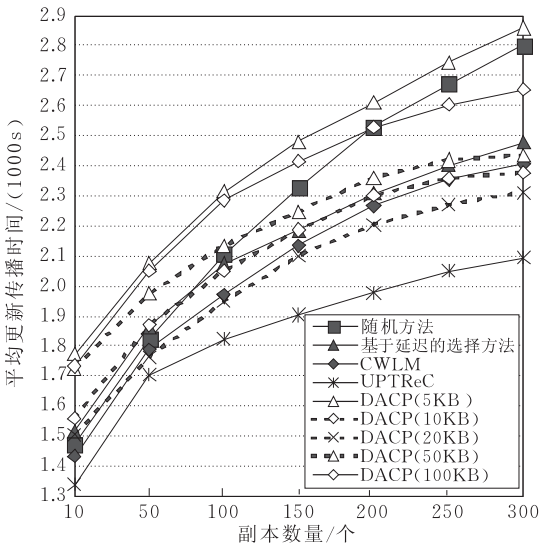


图 7 副本数量固定情况下的一致性维护开销对比

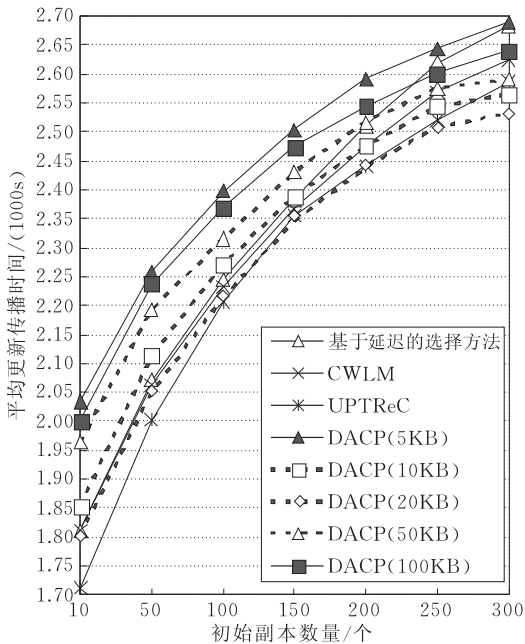


图 8 副本数量动态变化情况下的系统维护开销对比

随着副本数量的增多,DACP 方法在 $B_x=10\text{KB}$, $B_x=20\text{KB}$ 和 $B_x=50\text{KB}$ 时的性能与 UPTReC 算法相当或优于 UPTReC 算法,从而体现出分块复制带来的好处;但是 $B_x=5\text{KB}$ 和 $B_x=100\text{KB}$ 的情况下的性能并不理想.

(3) 副本数量动态变化情况下多播对系统维护开销的影响

我们模拟对比了副本增加时采用多播通知其它副本和不采用多播通知其它副本对性能的影响. 由于 CWLM 方法是基于更新提交的顺序或本地更新日志信息选择会话结点,与 DACP 方法不具有可比性,因此这里只与基于延迟的选择方法进行比较. 实验方法同上.

从实验结果(图 9)可以看出:采用多播通知副本增加对基于延迟的选择方法和 UPTReC 算法几乎没有影响,而对 DACP 方法(这里只测试 $B_x=20\text{KB}$ 的情况)带来了性能上的提高,因为尽快地获取新增加副本的存在,有利于更新的广域传播. 对于基于延迟选择方法,复制一般是读取“最近”的副本来创建新副本,更新传播时也是选择与“最近”副本交互,因此远程副本对新增副本的存在并不关心,采用多播对性能几乎没有影响.

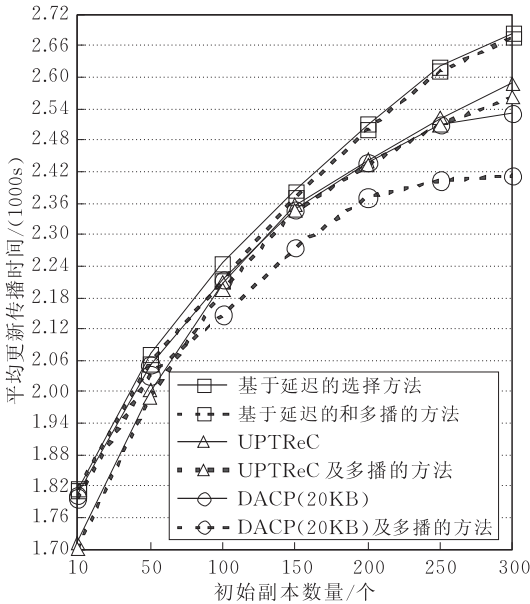


图 9 副本数量动态变化情况下多播对系统维护开销的影响

5.3 分块粒度的确定

通过上面的分析以及模拟测试,我们可以发现数据对象分块大小是 DACP 方法的基础,DACP 方法中选取适当的分块大小将会获得较优的数据一致性维护性能. 适当的分块大小可以通过一定的模拟测试分析获取,也可能随着系统的调整进行相应的

修改,下面我们给出如何选取适当分块的方法.

数据一致性维护方法的设计将直接影响到 P2P 系统中更新传播开销、系统实际接受更新数量、副本的“老化”程度等多个方面,由于不同的 P2P 应用系统所关注的被影响因素(系统参数)不同,并且每个因素的重要程度不同,因此需要对其所采用的数据一致性维护方法中的参数进行相应的调整.

根据具体 P2P 系统的需求,由设计人员选定系统参数并确定其影响因子;每个参数的影响因子的大小反映该参数在所有系统考虑到的系统参数中的重要程度,并且可以根据系统不同阶段的具体需求来进行相应的调整. 假设针对某 P2P 系统,在数据一致性维护方法设计时需要考虑到 n 个系统参数: P_1, P_2, \dots, P_n ; 第 j 个参数 P_j 的影响因子为 $f_j (0 < f_j \leq 1)$, 且 $\sum_{j=1}^n f_j = 1$.

系统试运行阶段,选定一组分块大小 B_1, B_2, \dots, B_m 分别测试参数 P_1, P_2, \dots, P_n , 获取各个参数的统计信息,其中右下区域内第 i 行 j 列上的数据表示: 当选定分块大小为 B_i 时,测试得到的参数 P_j 的统计信息.

	P_1	P_2	\dots	P_n	
B_1	$d_{1,1}$	$d_{1,2}$	\dots	$d_{1,n}$	(*)
B_2	$d_{2,1}$	$d_{2,2}$	\dots	$d_{2,n}$	
\vdots	\vdots	\vdots	\dots	\vdots	
B_m	$d_{m,1}$	$d_{m,2}$	\dots	$d_{m,n}$	

下面对获取到的参数的统计信息进行处理,处理主要分为 3 个方面:

(1) 影响方向的调整. 从实验数据上来看,每个参数对系统性能的影响的方向是不同的;有的参数的影响是正向的,即实验数据越大,系统性能越佳;而有的参数的影响是负向的,即实验数据越小,系统性能越佳;因此需将所有参数的实验数据调整到同一方向上,假设为函数 $A_Force(P_j) (j \in [1, n])$. 具体操作中,可以采用取倒数的简单方法更改参数的影响方向.

(2) 数量级的调整. 调整完实验数据的影响方向之后,获取到的实验数据所处的数量级可能不同,因此为了系统参数的可比性,将所有参数的统计信息调至可比的量级上,假设为函数 $A_Magnitude(P_j) (j \in [1, n])$.

(3) 相对大小的调整. 虽然经过 $A_Force(P_j)$ 和 $A_Magnitude(P_j) (j \in [1, n])$ 的处理之后,实验数据已经非常的规整,但是同一系统参数的所有实验

数据的可比性并不强,因此选取其中的某个数据作为基准,其它数据都是与其相比的相对值,假设为函数 $A_Relative(P_j)(j \in [1, n])$.

对上述(*)中实验数据的第 $i(i \in [1, m])$ 行,根据式(2)进行计算:

$$\sum_{j=1}^n f_j \times A_Relative(A_Magnitude(A_Force(d_{i,j}))) \quad (2)$$

计算后矩阵变形为

	D
B_1	D_1
B_2	D_2
\vdots	\vdots
B_m	D_m

比较数据 $D_i(i \in [1, m])$ 的大小,获取 $D_k = \min\{D_1, D_2, \dots, D_m\}$,则 B_k 选定为对系统适当的分块大小.

在我们的模拟实验中,选定更新传播时间(P_1)和更新接收率(P_2)两个系统参数,更新传播时间是每个数据一致性维护方法都必考虑的基本系统参数.另外,在广域分布的环境中,提交的更新必然存在冲突,分布式系统不可能接收所有的更新,但是如果系统过度地拒绝用户提交的更新,则降低了系统的可用性.因此我们这里定义更新接受率(UAR): $UAR = \text{提交更新数量} / \text{发布更新数量}$,将其作为系统参数之一,并简单地约定参数 P_1 和 P_2 的影响因子分别为 0.6 和 0.4.

正式的模拟测试前,我们选定一组经验值作为拟分块大小,为 1KB, 5KB, 10KB, 20KB, 100KB, 1MB, 10MB, 从中确定在我们的模拟环境下将依据的最适当分块大小.

我们对每种数据块大小分别进行一组模拟实验,在每组模拟实验的初始时刻,每个数据对象产生 100 个副本,并将这些副本分散到随机选择的结点上.在实验进行过程中,提交 5000 个更新,模拟运行结束条件为完成 5000 个更新的传播,所有副本一致,比较更新的平均传播时间,实验数据如图 10 所示.

我们在上述实验的同时,测量了在不同更新频率(F_u)的情况下,不同的数据分块大小对 UAR 的影响,实验数据如图 11 所示.从实验结果看出:当数据对象分块的粒度小时,UAR 很高;随着数据对象分块的粒度增大,UAR 降低.这是因为,数据块越

小,伪冲突更新越容易被识别.但是数据块太小,增加了将提交的更新控制在同一数据块内的难度.

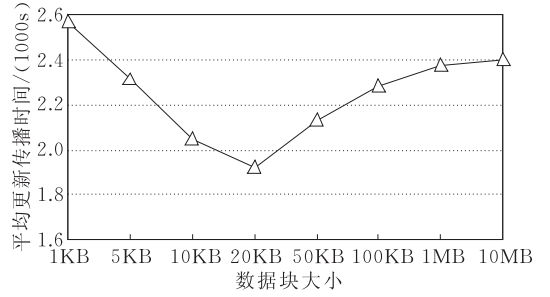


图 10 数据分块大小对平均更新传播时间的影响

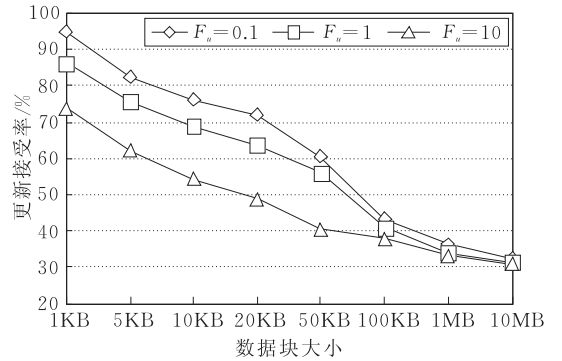


图 11 数据分块大小对更新接受率的影响

表 3 给出了对模拟测试结果进行计算的过程,其中:参数 P_1 对系统性能的影响是负向的,因此取 $A_Force(P_1) = 1/P_1$,将其对系统的影响调整为正向;由于 $1/P_1$ 和 P_2 对应的数据均属于 $(0, 1)$ 范围之内,因此 $A_Magnitude(P_j) = P_j(j = 1, 2)$;这里设定

$$A_Relative(P_j) = P'_j = \frac{d_{i,j}}{\max\{d_{1,j}, d_{2,j}, \dots, d_{m,j}\}}, \quad i \in [1, m]; j = 1, 2.$$

通过式(2)计算得到结果 D ,我们可以看到当 $B_x = 20KB$ 时对应的 D 值最大,当 $B_x = 10KB$ 时对应的 D 值次之.另外,虽然 $B_x = 1KB$ 时计算得到的 D 值在所有的 D 值中仅次于 $B_x = 10KB$ 和 $B_x = 20KB$,但是在实际选择时一般不应选取 $B_x = 1KB$ 作为数据分块大小,因为其在参数 P_1 上的性能是最差的.因此,数据分块大小的选择应综合考虑各个系统参数,在将式(2)的计算结果作为参考之外,还应该注意所选取的分块大小在每个参数上的实验结果是否在系统可以接受的范围之内;否则,可以选择另一相对权衡到各个因素的分块大小.

另外,我们模拟了副本数量对 UAR 的影响.模拟实验中,数据对象以 20KB 为单位进行分块管理;在实验进行过程中,提交 5000 个更新,运行结束条件同上,比较 UAR.

表 3 数据分块大小对系统参数的影响对比

数据块大小		P_1	P_2	$1/P_1$	P_2	P'_1	P'_2	D
$F_u=0.1$	1KB	2.570	0.9478	0.3891	0.9478	0.7478	1	0.8487
	5KB	2.313	0.8232	0.4323	0.8232	0.8307	0.8685	0.8458
	10KB	2.048	0.7604	0.4883	0.7604	0.9385	0.8023	0.8840
	20KB	1.922	0.7197	0.5203	0.7197	1	0.7593	0.9037
	50KB	2.134	0.6046	0.4686	0.6046	0.9006	0.6379	0.7955
	100KB	2.285	0.4328	0.4376	0.4328	0.8411	0.4566	0.6873
	1MB	2.377	0.3637	0.4207	0.3637	0.8086	0.3837	0.6386
	10MB	2.402	0.3240	0.4163	0.3240	0.8001	0.3418	0.6168
$F_u=1$	1KB	2.570	0.8614	0.3891	0.8614	0.7478	1	0.8487
	5KB	2.313	0.7568	0.4323	0.7568	0.8307	0.8786	0.8497
	10KB	2.048	0.6893	0.4883	0.6893	0.9385	0.8002	0.8831
	20KB	1.922	0.6347	0.5203	0.6347	1	0.7368	0.8947
	50KB	2.134	0.5612	0.4686	0.5612	0.9006	0.6515	0.8010
	100KB	2.285	0.4095	0.4376	0.4095	0.8411	0.4754	0.6948
	1MB	2.377	0.3401	0.4207	0.3401	0.8086	0.3948	0.6431
	10MB	2.402	0.3136	0.4163	0.3136	0.8001	0.3641	0.6257
$F_u=10$	1KB	2.570	0.7363	0.3891	0.7363	0.7478	1	0.8487
	5KB	2.313	0.6210	0.4323	0.6210	0.8307	0.8434	0.8358
	10KB	2.048	0.5425	0.4883	0.5425	0.9385	0.7368	0.8578
	20KB	1.922	0.4884	0.5203	0.4884	1	0.6633	0.8653
	50KB	2.134	0.4052	0.4686	0.4052	0.9006	0.5503	0.7605
	100KB	2.285	0.3776	0.4376	0.3776	0.8411	0.5128	0.7098
	1MB	2.377	0.3319	0.4207	0.3319	0.8086	0.4508	0.6655
	10MB	2.402	0.3083	0.4163	0.3083	0.8001	0.4187	0.6475

从实验结果(图 12)可以看出:副本越多,相互冲突的更新也随之增多,从而导致 UAR 降低. 随着系统的运行,系统中各个数据的副本数量随之增加,因此可以降低数据分块大小,从而获得较为满意的系统更新接受率,同时提高了系统的可用性.

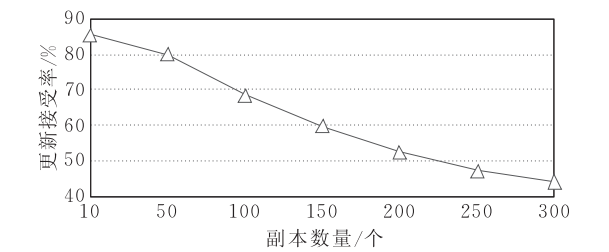


图 12 副本数量对更新接受率的影响

5.4 鲁棒性

我们模拟测试了 DACP 方法与 Bayou 以及 UPTReC 算法在不同的结点失效率时一致性的性能. 这里的结点失效为随机地从整个系统中选择固定数量的结点使其失效,考察各种方法在一致性维护、结点失效处理和结点恢复方面的系统维护开销.

模拟初始化时,每个结点上存储了随机选择的 100 个数据对象. 每组模拟实验中,随机地从系统中选择数量为 $(1000 \times \text{结点失效率})$ 个结点,使其处于失效状态,处于失效状态的结点在随机的时间段后恢复;DACP 方法($B_x=20\text{KB}$)按照 4.3 节中的方法处理失效恢复,Bayou 和 CWLM 根据会话选择方法选择某个副本处理失效恢复,UPTReC 算法通过 pull 某个在线的副本来同步文件状态和副本链上的

信息. 模拟器提交并完成 5000 个更新的传播,模拟运行结束条件同上.

从模拟结果(图 13)可以看出:随机方法和 UPTReC 算法受结点失效的影响最大,因为随机方法在传播更新时,盲目地将更新发送给远程副本结点,对结点的状态并不关心,而 UPTReC 算法由于每个副本结点只维护少量的其它副本信息,当失效的副本增多时,更新传播可能停止在某个结点上;CWLM 方法在选择会话结点时可能将更新传播给失效的结点,因此其性能不佳;DACP 方法在传播更新时避免了盲目性和冗余性,且结点恢复时利用

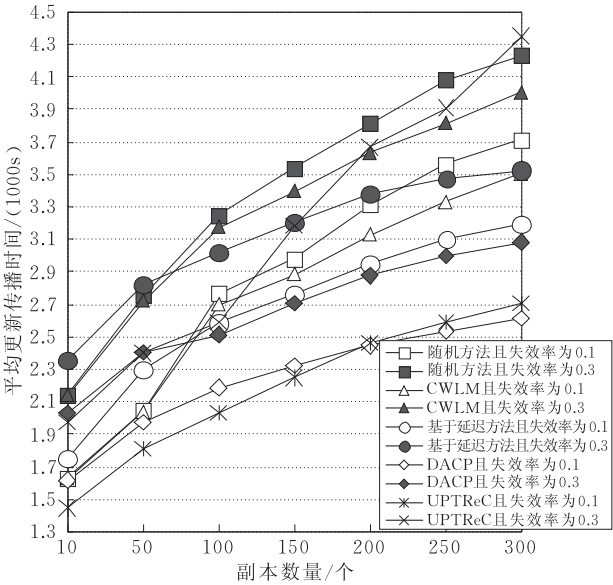


图 13 不同结点失效率下系统性能对比

分块的特征并行恢复,具有较强的鲁棒性.

6 结 论

P2P 分布存储系统在多个结点上复制数据对象,数据复制在提高系统可用性和系统性能的同时却不可避免地带来数据一致性维护问题.数据一致性中的数据相关性问题表现为伪冲突更新和更新依赖两个方面,这些问题是在弱一致的复制算法中必须要解决的.DACP 是一种面向 P2P 分布存储系统的基于数据相关性的优化数据一致性维护方法,具有以下特点:

(1) 利用数据分块技术将数据对象按照固定大小分块,从而在一定程度上消除伪冲突更新;增加副本或者结点失效恢复时,通过多副本并行复制各个数据块,提高了系统的性能;

(2) 利用 Bloom filter 技术压缩表示非提交更新,在无需建立状态估计机制的情况下避免更新的冗余传输,减少了一致性维护过程中的通信开销;

(3) 采用双路径更新传播,避免更新传播的盲目性和冗余性,获得了较好的更新传播速度及传播的广域性;

(4) 采用协商算法发现相关更新,解决更新冲突,实现了副本一致;

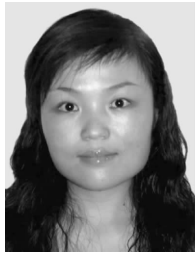
(5) 采用动态数据管理算法适应更新过程中数据大小的动态变化.

理论证明 DACP 方法可以确保数据副本一致.模拟实验中对更新的动态特性、分块大小对 DACP 方法的影响、DACP 方法与其它方法在一致性维护开销、和鲁棒性等方面的性能进行了对比测试,并给出了选定适当分块大小的指导性方法.测试结果表明 DACP 方法在选取适当范围内的分块大小情况下具有较好的性能.

参 考 文 献

- [1] Dahlin M, Gao L, Nayate A, Venkataramani A, Yalagandula P, Zheng J. PRACTI replication for large-scale systems. University of Texas at Austin, Austin: Technical Report TR-04-28, 2004
- [2] Byung Brent, Kang Hoon. S2D2: A framework for scalable and secure optimistic replication [Ph. D. dissertation]. University of California, Berkeley, 2004
- [3] van Renesse R, Schneider F B. Chain replication for supporting high throughput and availability//Proceedings of 6th Symposium on Operating Systems Design & Implementation. San Francisco, CA, 2004: 91-104
- [4] Ranganathan K, Iamnitchi A, Foster I. Improving data availability through dynamic model-driven replication in large Peer-to-Peer communities//Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid. Berlin, Germany, 2002: 376-381
- [5] Hildrum K, Kubiawicz J D, Rao S, Zhao B Y. Distributed object location in a dynamic network//Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures. Winnipeg, Manitoba, Canada, 2004: 41-52
- [6] Zhao B Y, Huang L, Stribling J, Rhea S C, Joseph A D. Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications, 2004, 22(1): 41-53
- [7] Leontiadis E, Dimakopoulos V V, Pitoura E. Creating and maintaining replicas in unstructured Peer-to-Peer systems//Proceedings of the 12th International Euro-Par Conference on Parallel Processing. Dresden, Germany, 2006: 1015-1025
- [8] Wang Zhijun, Kumar Mohan, Das Sajal K, Shen Huaping. File consistency maintenance through virtual servers in P2P systems//Proceedings of the 11th IEEE Symposium on Computers and Communications. Pula-Cagliari, Sardinia, Italy, 2006: 435-441
- [9] Yin L, Cao G. DUP: Dynamic-tree based update propagation in Peer-to-Peer networks//Proceedings of the 21st IEEE International Conference on Data Engineering. Tokyo, Japan, 2005: 258-259
- [10] Aberer K, Datta A, Hauswirth M. P-Grid: Dynamics of self-organizing processes in structured P2P systems//Proceedings of the Peer-to-Peer Systems and Applications. LNCS 3485. Berlin/Heidelberg: Springer, 2005: 137-153
- [11] Wang Zhijun, Das Sajal K, Kumar Mohan, Shen Huaping. Update propagation through replica chain in decentralized and unstructured P2P systems//Proceedings of the 4th IEEE International Conference on Peer-to-Peer Computing. Zurich, Switzerland, 2004: 64-71
- [12] Terry D B, Theimer M M, Petersen K, Demers A J, Spreitzer M J, Hauser C H. Managing update conflicts in Bayou, a weakly connected replicated storage system//Proceedings of the 15th ACM Symposium on Operating Systems Principles. Copper Mountain, Co, 1995: 172-182
- [13] Petersen K, Spreitzer M J, Terry D B. Flexible update propagation for weakly consistent replication//Proceedings of the 16th ACM Symposium on Operating Systems Principles. Saint Malo, France, 1997: 288-301
- [14] Wang Yi-Jie, Qin Yong-Jin, Li Si-Kun. Research of massive data oriented replication algorithm. Journal of Computer Research and Development, 2004, 41 (Supplement): 6-10 (in Chinese)
(王意洁,秦永进,李思昆.面向海量数据的复制算法研究.计算机研究与发展, 2004, 41(增刊): 6-10)
- [15] Kubiawicz J, Bindel D, Chen Y, Czerwinski S, Eaton P, Geels D, Gummadi R, Rhea S, Weath-Erspoon H, Weimer W, Wells C, Zhao B. OceanStore: An architecture for global-scale persistent storage. ACM SIGPLAN Notices, 2000, 35(11): 190-201
- [16] Wang Yi-Jie, Zhang Xiao-Ming. A prediction-based parallel replication algorithm in distributed storage system//Proceed-

- ings of the 4th Grid and Cooperative Computing International Conference. Beijing, China, 2005: 690-700
- [17] Bloom B. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970, 13(7): 422-426
- [18] Fan Li, Cao Pei, Almeida Jussara, Broder Andrei Z. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 2000, 8(3): 281-293
- [19] Rhea Sean C, Kubiatowicz John. Probabilistic location and routing//*Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*. New York, USA, 2002: 1248-1257
- [20] Gribble Steven D, Welsh Matt, von Behren Robert J et al. The ninja architecture for robust internet-scale systems and services. *Computer Networks*, 2001, 35(4): 473-497
- [21] Bell W H, Cameron D G, Capozza L, Millar P, Stockinger K, Zini F. Optorsim: A grid simulator for studying dynamic data replication strategies. *International Journal of High Performance Computing Applications*, 2003, 17(4): 403-416
- [22] Zhou Jing, Wang Yi-Jie, Li Si-Kun. An optimistic replication algorithm to improve consistency for massive data//*Proceedings of the 4th Grid and Cooperative Computing International Conference*. Beijing, China, 2005: 713-718



ZHOU Jing, born in 1977, Ph. D., lecturer. Her research interests include network computing and virtual reality.

WANG Yi-Jie, born in 1971, professor and Ph. D. supervisor. Her research interests include network computing, database technology and mobile computing.

LI Si-Kun, born in 1941, professor and Ph. D. supervisor. His research interests include virtual reality and visualization, embedded system and SoC design methodologies.

Background

Data replication is an important technique to improve the performance of distributed systems. P2P systems replicate objects on multiple nodes to improve availability and performance. Data replication has many advantages such as avoiding single server failure, reducing access response time, reducing communication overhead and balancing load, but it unavoidably introduces well-known consistency issues.

Data updates may be issued simultaneously on different replicas in optimistic consistent systems and they may reach other replicas in an arbitrary order. While being intuitive, two updates conflict when a replica issues an update before it receives another update that is already circulating among replicas.

However, the two updates do not conflict radically if each update operates just on different segment of data object. Additionally, if the outcome of one of the two updates changes the condition of another, eventual data consistency by sequentially applying them breaches user view virtually. All the above questions are brought by data dependence in data consistency, which is inevitable for describing updates by semantic and often needs to be resolved manually.

In this paper, based on data dependence in data consistency in P2P distributed storage system, the optimistic data consistency method is proposed. It partitions large object into data blocks, uses Bloom filter compressed representation

technique to summarily represent updates, and treats blocks as management unit to detect and reconcile conflicts.

In recent years, Peer-to-Peer (P2P) computing has become a popular network computing paradigm. Researches on and applications of P2P computing have spread into many fields. P2P distributed storage systems, constructed by P2P computing technique, can offer data sharing and storage services for massive users and massive data. Data replication, one of the crucial technologies for managing massive data, can improve data availability and data access performance, but bring about problems of maintaining data consistency. Compared with other distributed systems, Peer-to-Peer systems exhibit some special characteristics, such as large-scale, dynamic, and heterogeneity, which has brought many challenges for data consistency maintenance technique in P2P distributed storage systems. Based on characteristics of massive data and P2P systems, an intensive study is conducted of data consistency maintenance technique for P2P distributed storage systems, including a multi-replica clustering management method based on limited-coding, an optimistic data consistency maintenance method to improve update propagation and reduce the space overhead of write-log and an optimistic data consistency maintenance method based on key-attributes. The relevant papers of study have been published.