ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.9, September 2008, pp.2471−2482
DOI: 10.3724/SP.J.1001.2008.02471

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种面向分布式虚拟环境的分层迭代负载平衡算法*

王少峰, 周 忠+, 吴 威

(虚拟现实技术与系统国家重点实验室(北京航空航天大学),北京 100191)
(北京航空航天大学 计算机学院,北京 100191)

A Layered Iterative Load Balancing Algorithm for Distributed Virtual Environment

WANG Shao-Feng, ZHOU Zhong+, WU Wei

(State Key Laboratory of Virtual Reality Technology and Systems, BeiHang University, Beijing 100191, China)
(School of Computer Science and Technology, BeiHang University, Beijing 100191, China)
+ Corresponding author: E-mail: zz@vrlab.buaa.edu.cn

Wang SF, Zhou Z, Wu W. A layered iterative load balancing algorithm for distributed virtual environment. *Journal of Software*, 2008,19(9):2471−2482. http://www.jos.org.cn/1000-9825/19/2471.htm

Abstract: To support large-scale users to share a virtual environment, multi-server architecture is applied to the virtual environment system and each server handles a partition of the virtual environment. The unpredictable movements and interactions of avatars may cause some servers to be overloaded. Existing load balancing algorithms try to redistribute load among servers and incur too much overhead, which may reduce the interactivity of system. In this paper, a layered iterative dynamic load balancing algorithm is proposed. Setting the overloaded region as the center, the algorithm chooses a limited number of neighboring regions as the load adjusting goals and iteratively spreads the overload outward from the center. The load balancing state can be achieved through iterations. Based on the virtual environment wherein avatars are scattered under skewed and clustered distribution, the algorithm is built and comparison with the existing load balancing algorithms are made. The results show that this algorithm can adjust load efficiently and incur less overhead.

Key words: distributed virtual environment; multi-server; dynamic load balancing; layered iteration

摘 要: 为了支持大规模用户共享虚拟环境,多服务器结构被应用到分布式虚拟环境系统中,每个服务器负责虚拟环境的一个区域划分.由于用户不可预知的移动和交互,可能会导致某些服务器负载过重.现有的负载平衡算法注重于将负载在服务器间重分配,但引入开销过大,影响系统交互性能.提出一种分层迭代的动态负载平衡算法,以过载区域为中心,分层地选择周围有限数量的区域作为调整目标,将过载部分由内向外迭代地扩散到各层,多次迭代达到负载平衡状态.针对倾斜和聚簇两种典型用户分布的虚拟环境,对算法进行验证并与现有的 3 种负载平衡算法进行比较.结果表明,该算法可以快速、有效地调整负载并引入较少的开销.

关键词: 分布式虚拟环境;多服务器;动态负载平衡;分层迭代
中图法分类号: TP391 **文献标识码**: A

* Supported by the National Natural Science Foundation of China under Grant No.60603084 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z331 (国家高技术研究发展计划(863))
Received 2006-11-15; Accepted 2008-03-11

大规模分布式虚拟环境需要支撑大量地理分布的用户同时在线交互,并且共享一致的虚拟环境,随着虚拟场景的扩大和并发用户的增加,提高系统可扩展性和负载能力是关键问题.影响分布式虚拟环境系统的可扩展性和负载能力的最基本因素是其基于的网络模型.为了构建一个多用户分布式虚拟环境系统,采用的网络模型主要有:分布式、集中式和层次式结构.分布式结构没有中心服务器,由本地节点负责计算处理和网络通信,早期的一些系统如 NPSNET[1],DIVE[2]等采用的就是分布式结构,优点是低延时,避免了中心节点的瓶颈;但占用了大量带宽,安全性差,并且当规模变大时,整个系统的一致性难以得到保证.集中式结构存在中心服务器,每个节点将网络消息发送给服务器进行计算处理并且转发,由客户端进行响应,MASSIVE-3[3]及著名的网络游戏反恐精英(Counter-Strike)[4]、雷神之锤(Quake)[5]采用了集中式结构,系统一致性和安全性容易控制,服务器易于进行消息过滤;但是随着用户规模的扩大,服务器易成为瓶颈,导致延时变长,并且存在单点失效问题.层次式结构结合分布式和集中式的特点,由多个服务器为客户端提供服务,客户端和服务器以集中式连接,而服务器间采用分布式结构.基于多服务器结构的分布式虚拟环境系统采用的就是层次式结构,用于提高系统的可扩展性和负载能力.

基于多服务器结构的分布式虚拟环境系统将整个虚拟环境划分为多个区域,每个区域由一台服务器来负责,整个系统的负载被分配到多个服务器,每个服务器与其负责区域内的用户采用客户端/服务器方式连接,而在服务器之间以高速网络连接.BrickNet[6],RING[7],NetEffect[8],CittaTron[9],CyberWalk[10],ATLAS[11]等分布式虚拟环境系统都采用了这种结构.一些流行的多人在线网络游戏,如 Ultima Online[12],Asheron's Call[13]等也采用了多服务器结构,将整个游戏世界划分为连续的区域,用户可以从一个区域迁移到另一个区域.多服务器结构具备集中控制的优点,并且将整个系统的负载分配到多台服务器,具有更好的扩展性.

对于用户行为不可预知的分布式虚拟环境,随着用户在虚拟环境的不均匀分布,一些区域可能会聚集过多的用户,造成相应的服务器负载过重,严重影响整个系统的负载能力和交互性.所以,基于多服务器的分布式虚拟环境系统涉及的一个关键问题就是运行时服务器间的动态负载平衡,将过载服务器的部分负载调整到其他服务器,以避免负载过重,服务器成为瓶颈,导致系统整体性能下降.动态负载平衡涉及到区域重新划分,用户状态的迁移及服务器间的协调,是一个计算开销大的任务,并且用户在分布式虚拟环境的分布是实时变化的,整个系统的负载也相应地不断变化,若算法在调整负载时动态性差,执行时间过长,则负载平衡的结果是不合适的.

针对上述问题,本文提出一种分层迭代的动态负载平衡算法,分层地选择周围有限数量的区域服务器作为调整目标,尽量减少开销;并且算法可迭代执行,每次迭代获取系统的最新负载状态,提高负载调整动态性.本文第 1 节总结当前多服务器虚拟环境系统中的动态负载平衡算法.第 2 节详细介绍分层迭代动态负载均衡算法,给出基本思想和设计.第 3 节对算法进行验证,选取用户在虚拟环境的两种典型分布,与局部和全局动态负载平衡算法进行对比分析.最后对全文进行总结.

1 相关工作

在服务器之间进行动态负载平衡是多服务器虚拟环境系统的研究热点之一.根据算法选择负载转移目标的策略不同,现有的负载平衡算法主要分为全局和局部两种类型.

全局负载平衡算法在调整负载时,过载服务器转移负载的目标是系统内所有的服务器,达到各服务器的负载平衡[9,14].通过获取全局的服务器的负载信息,当一个服务器负载过重时,选择整个系统中负载最轻的服务器进行负载调整,达到整个系统的负载平衡[15].提出了一种并行递增的图分割算法对虚拟环境的多服务器进行负载分配,是一种全局负载平衡机制.算法将整个虚拟世界映射成图 $G=(V,E)$,V 为用户集合,E 为边集合,任意一条边代表两个用户存在网络通信,将图 G 分成不相交的子集来实现负载在服务器间的均匀分配.这种算法虽然能够有效地解决负载均匀分配问题,但是随着服务器和参与用户数目的增多引入了过多的计算量,执行时间过长,根据文中实验数据,当用户和服务器数目分别为 25 000 和 16 时,算法执行时间将达到秒级,不适合在高实时性的分布式虚拟环境系统中应用.全局负载平衡算法可以处理负载分布很不均匀的虚拟环境系统的负载失衡问题,但是区域不相邻的服务器之间负载迁移代价大,并且一个服务器负载多个不相邻区域易引入过多的服务器

间网络通信.

局部负载平衡算法将过载服务器的负载向其相邻服务器转移,在局部范围内进行负载调整.CyberWalk[10]是一个多服务器结构的分布式虚拟环境系统,整个虚拟环境划分为多个区域,由相应的服务器负责,采用一种局部负载平衡算法来动态调整过载服务器的负载.每个服务器定期监视自己的负载情况,当负载超过一定的阈值时,将部分负载调整到区域相邻的负载最轻的服务器.局部负载平衡算法的负载调整只在区域相邻的服务器之间进行,负载迁移代价比较小,不会引起服务器间大的通信量,适合于规模比较小、过载区域比较分散的分布式虚拟环境系统;但是,对于负载分布很不均匀的大规模的虚拟环境系统,多个负载过重的区域相邻地聚集在一起,局部负载平衡算法不能很好地缓解负载失衡的状态.

近年来,有些研究将局部和全局两类算法相结合,试图在调整负载的有效程度和开销方面加以折衷[16].以过载服务器为起点,通过相邻关系确定一个数目有限的候选服务器集合,对这个集合采用图分割算法进行全局性的负载调整,实现集合内服务器负载均匀分配.但采用图分割算法进行负载均匀分配引入了过多的计算开销,需要较多的用户迁移,服务器间的通信负载较重[17].根据过载服务器划定一个负载共享的范围,由内到外通过协商方式在相邻服务器间进行负载调整,直至负载过重服务器负载正常,这种算法的优点是负载转移只发生在相邻区域间,负载迁移代价小,但是多次协商引入了服务器间额外的通信开销,可能导致负载过重服务器不能很快地降低负载,影响正常交互[18].基于分割聚合(partition and aggregation)思想,系统优先考虑局部负载调整;在局部负载调整不能进行时,负载过重的服务器会在全局范围内选择一个轻负载服务器(负载低于一个预先定义的阈值)进行负载转移,以达到负载平衡;对于可能产生的一个服务器负责多个区域的情况,算法选择一定时机将一些区域进行聚合,减少服务器间的网络通信,但区域聚合会带来相应开销,如果聚合时机选择不当,会加剧负载调整的发生.上述 3 种算法没有解决负载平衡存在的计算开销大.动态性差的问题,在转移负载时,算法一次性地将过载部分转移出去,负载转移开销大,执行时间较长,并且在整个算法执行过程中没有考虑系统的负载状态实时变化,影响系统的交互能力.

2 分层迭代的动态负载平衡算法

基于负载平衡过程中尽量减少计算开销,提高负载调整动态性的考虑,分层迭代的动态负载平衡算法以过载区域服务器为中心,分层地选择周围有限数量的区域服务器作为调整目标,每次迭代选定可调整负载的相邻两层,在两层间相邻的区域服务器进行负载调整,尽量减少迁移开销,并且算法的每一次迭代都会获取系统的最新负载状态作为负载调整依据,以保证负载调整的动态性.如图 1 所示,算法经过多次迭代将过载部分由内向外地扩散到各层,逐步达到负载平衡状态,用户可以根据实际应用设定迭代的间隔.

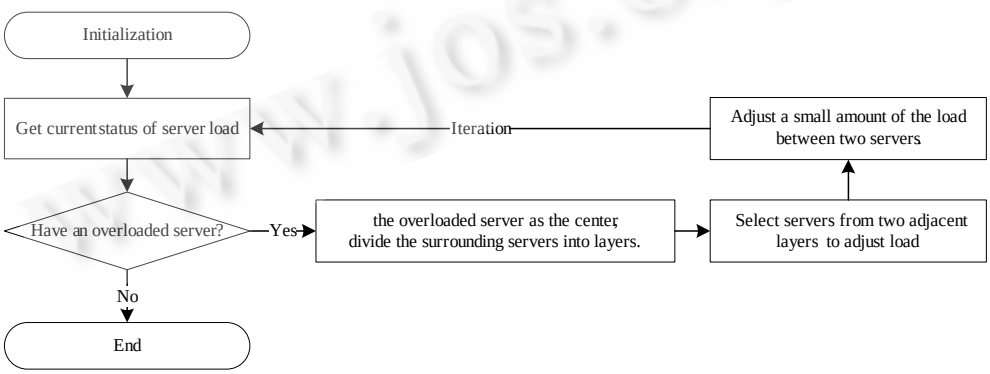


Fig.1 Layered iterative load adjusting process

图 1 分层迭代调整负载的过程

2.1 基本定义

下面给出本文算法中涉及的一些基本定义,算法建立在下述定义的基础之上.

(1) 区域(region)

整个虚拟环境进行划分后,每一块称为一个区域,初始时各个区域大小相等.每个区域由一个区域服务器RegionServer(RS)负责,不同的 RS 以下标 *i*,*j*,*t*,*v*,…进行区分.整个虚拟环境划分的区域数目为 *N*,相应的,区域服务器数目也为 *N*,图 2 中就是一个划分为 4 个 Region 的虚拟环境.

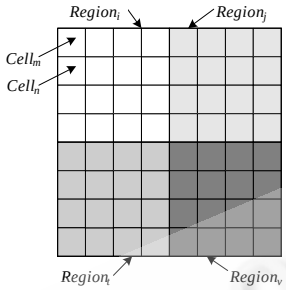


Fig.2 Region and Cell

图 2 Region 和 Cell 示意图

(2) Cell

每个区域细分成多个相邻的规则正方形小单元称为 *Cell*,*Cell* 是空间结构中最小的单位,不同的 *Cell* 以下标 *m*,*n*,…进行区分.图 2 中的虚拟环境中每个区域划分为 16 个 *Cell*.

(3) 用户兴趣区域 AOI(area of interest)

用户 AOI 代表用户的感知范围,一般以用户为中心的圆形区域来表示.一个用户位于另一个用户的 AOI 区域,则表示两个用户可见,如果这两个用户处于不同的 RS,则会引起两个 RS 的服务器之间的通信.

2.2 分层迭代调整负载

我们先给出一些定义,以便阐述分层扩散的动态负载平衡算法.

令 $NB(RS_i)$为所有与 RS_i 相邻的 RS 构成集合,两个区域服务器 RS_i 和 RS_j 相邻,当且仅当 RS_i 和 RS_j 对应于虚拟环境划分的两个 Region 有一条公共边;

令 $DiffRS(RS_i,k)$为区域服务器 RS_i 的第 k 层扩散区域,可由下式计算,且满足条件:任意 RS_t 属于 $DiffRS(RS_i,l)(l<k)$,则 RS_t 不属于 $DiffRS(RS_i,k)$.

$$DiffRS(RS_i,k)=\begin{cases}NB(RS_i), & \text{if } k=1\\ NB(RS_j),\ RS_j\in DiffRS(RS_i,k-1), & \text{if } k>1\end{cases}.$$

令 $OL(RS_i)$为 RS_i 的过载阈值,表示 RS_i 能够处理的负载临界值,超过此值,RS_i 需要转移负载;

令 $SL(RS_i)$为 RS_i 的安全负载阈值,表示 RS_i 负载安全的临界值,$SL(RS_i)=\alpha\times OL(RS_i)$,$\alpha$取值由用户决定,一般可取 0.8~0.9 的值;

令 $L(RS_i)$为区域服务器 RS_i 的负载,用 RS_i 管理的用户数量来衡量.约定:

$L(RS_i)>OL(RS_i)$,负载过重,需要转移负载;

$L(RS_i)>=SL(RS_i)$ && $L(RS_i)=<OL(RS_i)$,临界状态,不能转移负载也不能接受负载;

$L(RS_i)<SL(RS_i)$,负载较轻,能够接受负载.

分层迭代动态负载平衡算法以负载过重的区域服务器为中心,分层地选择周围有限数量的区域服务器作为调整目标,设定一个负载调整的最大层次 k,对于一个负载过重的 RS_i,如果不能向其第 l 层($l<=k$)扩散区域迁移负载,则第 l 层扩散区域内的 RS 先向 RS_i 的第 l+1 层扩散区域迁移负载,这样,RS_i 就可以向第 l 层扩散区域迁移负载,经过多次迭代,将过载部分扩散到 RS_i 的 k 层扩散区域,达到负载平衡状态.对于负载过重的区域服务器 RS_i:$L(RS_i)>OL(RS_i)$,算法的迭代处理流程如下:

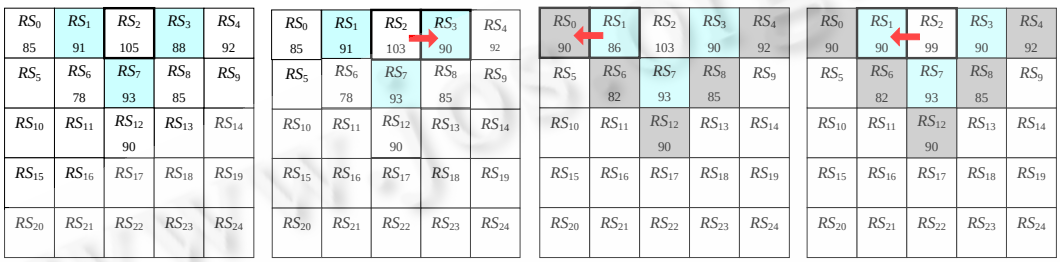
(1) 初始层数 l=1,向第 1 层进行扩散,即 RS_i 向 $DiffRS(RS_i,1)$进行负载迁移.获取 $DiffRS(RS_i,1)$,对任意 RS_j 属

于 $DiffRS(RS_i,1)$检查是否满足 $L(RS_j)<SL(RS_j)$,如果满足,则从 RS_i 负责区域中选择一个 $Cell_m$ 将其迁移给 RS_j;

(2) 如果不能向第 l 层扩散,增加 1 层,$l=l+1$.获取 $DiffRS(RS_i,l)$,任意 RS_t 属于 $DiffRS(RS_i,l-1)$,若 $DiffRS(RS_i,l)$中存在 RS_v,$L(RS_v)<SL(RS_v)$,并且 RS_v 属于 $NB(RS_t)$,则从 RS_t 负载区域中选择一个 $Cell_n$ 将其迁移给 RS_v;

(3) 按层进行局部性的负载扩散,直至 RS_i 能够在第 l 层($l<k$)迁移负载或者超出第 k 层.

设定最大层数 k=2,每个 RS 的过载阈值为 100,安全负载阈值为 0.9×100=90,一个简单的分层迭代负载调整过程示例如图 3 所示.图 3(a)所示 RS_2 负载过重,首先获取其第 1 层 $DiffRS(RS_2,1)=\{RS_1,RS_3,RS_7\}$,其中,$RS_3$ 的负载为 88 小于安全负载,则可以接收 RS_2 的负载,完成第 1 次迭代,调整后的状态如图 3(b)所示;调整后,RS_2 负载为 103,仍负载过重,进行第 2 次迭代,因为第 1 层 $DiffRS(RS_2,1)$已经不能接收负载,则再扩散一层 $DiffRS(RS_2,2)=\{RS_0,RS_4,RS_6,RS_8,RS_{12}\}$,对 $DiffRS(RS_2,1)$中的 RS_1 可以将负载迁移给 $DiffRS(RS_2,2)$中其相邻的 RS_0,负载调整后状态如图 3(c)所示;最后进行第 3 次迭代,RS_2 可将负载再调整给 RS_1,达到负载平衡状态,如图 3(d)所示.这样,对于负载过重的区域服务器 RS_i,通过算法的 3 次迭代,将使整个系统趋向于负载平衡状态.



(a) (b) (c) (d)

Fig.3 An example of layered iterative load adjusting

图 3 分层迭代负载调整过程示例

算法以负载过重的区域服务器 *HeavyRS* 为起始点进行负载扩散,采用一个队列 *nbRSQueue* 来记录扩散中每一层需要处理的 RS,用 *numRSofCurLayer* 和 *numRSofNextLayer* 记录当前层未处理和下一层待处理的区域服务器数目,保证负载调整的过程是从内层到外层,当某一层的 RS 可以进行负载迁移或者超出最大层数 k 时,算法的一次处理过程结束,伪代码实现见算法 1.

算法 1. 分层迭代动态负载平衡算法的伪代码.

```
int curLayer=0;                          //当前层数,初始为 0
Queue nbRSQuene;                         //待处理的 RS 队列
int numRSofCurLayer=1;                   //当前层未处理的 RS 数目
int numRSofNextLayer=0;                  //下一层需处理的 RS 数目
bool canLoadTransfer=false;              //在 k 层内能否转移负载
nbRSQuene.push(HeavyRS);
while(curLayer<=k) {
    RSi=nbRSQuene.pop();
    NB(RSi)=getNBSet(RSi);               //获得 RSi 的 NB(RSi);
    for(each RSj in NB(RSi)){
        if(RSj is NOT VISITED){
            标记 RSj 为 VISITED;
            if(L(RSj)<SL(RSj)) {
                canLoadTransfer=true;    //可以转移负载
                break;                   //exit for
            }else {
```

```
                nbRSQuene.push(RS_j);
                numRSofNextLayer++;                    //下一层处理 RS 数目增加
            }
        }
    }                                                  //end for
    if(canLoadTransfer){                               //如果可以转移负载
                                                       //选择一个待转移 Cell 和目标 RS_j
        SelectCellAndRS(RS_i,RS_j,Cell_m);
                                                       //将 Cell_m 由 RS_i 转移给 RS_j
        TransferOneCell(RS_i,RS_j,Cell_m);
        break;                                         //exit while
    }
    numRSofCurLayer--;                                 //当前层需要处理的 RS 数目减 1
    if(numRSofCurLayer==0)  {                          //这一层的 RS 处理完了
        numRSofCurLayer=numRSofNextLayer;
        numRSofNextLayer=0;                            //下一层处理 RS 数目清零
        curLayer++;                                    //增加一层
    }
} //end while
```

2.3 区域划分调整

在选定一个需要负载调整的区域服务器 RS_i 后,需要在其负责区域中选择一个 $Cell_m$,调整给其他区域服务器 RS_j,完成区域划分的调整,为下一步负载的迁移做准备.这一过程是由子算法 $SelectCellAndRS(RS_i,RS_j,Cell_m)$ 完成的,该算法选择合适的目标 RS_j 和待迁移 $Cell_m$,尽量减少迁移开销.

我们定义 $BufferCell(RS_i)$为 RS_i 的缓冲区 *Cell* 集合,由 RS_i 负责区域与其他区域交界处的 Cell 构成,一个属于 RS_i 的用户进入到属于 $BufferCell(RS_i)$的 *Cell*,会引起区域服务器 RS_i 和 RS_j 之间的网络通信.

对于一个区域服务器 RS_i,当需要转移出一个 $Cell_m$ 给 RS_j 时,基于如下几点考虑:

(1) 目标 RS_j 属于 $NB(RS_i)$并且 $L(RS_j)<SL(RS_j)$;

(2) 转移的 $Cell_m$ 属于 $BufferCell(RS_i)$;

(3) $Cell_m$ 的转移应该使受影响 Region 的周长尽量短.

选择 $NB(RS_i)$中的低于安全负载阈值的 RS,迁移代价较小并且不容易引起“抖动”;Cell 从 $BufferCell(RS_i)$中选出可以尽量减少 RS 间的网络通信;(3)的主要原因是:Region 的交界处会引起 RS 间的网络通信,受影响 Region 的周长尽量短可以使靠近边界的用户的 AOI 覆盖 Region 数目尽量少,使引入服务器间通信的 RS 数目尽量少,减少服务器间的网络通信.

针对上面的 3 点考虑,对 $BufferCell(RS_i)$内的 $Cell_m$ 和 RS_i 相邻的 RS_j,我们定义 $weight(Cell_m,RS_i,RS_j)$来衡量 $Cell_m$ 和 RS_j 被选择的权值:

$$weight(Cell_m,RS_i,RS_j)=NBCellNum(Cell_m,RS_j)/NBCellNum(Cell_m,RS_i)/Degree(Cell_m).$$

$NBCellNum(Cell_m,RS_i)$为 $Cell_m$ 直接相邻的 Cell 集合中属于 RS_i 的 Cell 个数;

$Degree(Cell_m)$为 $Cell_m$ 在虚拟环境中的位置因子,划分为 $L\times K$ 个 Cell 的虚拟环境,$Degree(Cell_m)$计算如下:

$$Degree(Cell_m)=\begin{cases}0.5 & m\in(1,1),(1,K),(L,1),(L,K) \quad 4个角上的Cell\\ 0.75 & m\in(1,y),(L,y),(x,1),(x,K) \quad 4个边上的Cell\\ & y\neq 1,K \text{ 并且 } x\neq 1,L\\ 1.0 & 其他\end{cases}.$$

在选择目标 $Cell_m$ 和 RS_j 时,选择权值最大的 $Cell_m$ 和 RS_j;若权值相同,则选择 $L(RS_j)$较小的 RS_j 及其对应的 $Cell_m$,*SelectCellAndRS*(RS_i,RS_j,$Cell_m$)子算法,见算法 2.

算法 2. 选择目标 Cell 和 RS 子算法 *SelectCellAndRS* 的伪代码.

```
SelectCellAndRS(RS_i,RS_j,Cell_m)
{
    //RS_j 和 Cell_m 为引用参数
    float weightDstCell=0.0;
    float weightCell=0.0;
    for(each Cell_n IN BufferCell(RS_i)){
        for(each RS_t 属于 NB(RS_i)){
            //计算 Cell 权值
            weightCell=weight(Cell_n,RS_i,RS_t);
            if(weightCell>weightDstCell||
                weightCell==weightDstDell && L(RS_t)<L(RS_j)){
          //有更合适的 Cell 和 RS
          weightDstCell=weightCell;
          Cell_m=Cell_n;
          RS_j=RS_t;
            }
        }
    }
}
```

2.4 迁移Cell

在用 *SelectCellAndRS*(RS_i,RS_j,$Cell_m$)子算法完成目标 $Cell_m$ 和 RS_j 的选择之后,用 *TransferOneCell*(RS_i,RS_j,$Cell_m$)子算法完成 RS_i 将 $Cell_m$ 迁移给 RS_j 相关的处理过程.

我们首先给出 *TransferOneCell* 子算法影响的一些数据定义.

令 *Type*($Cell_m$)为 $Cell_m$ 的类型,若 $Cell_m$ 属于 RS_i 的缓冲区 Cell 集合 *BufferCell*($Cell_m$),则其类型为缓冲 Cell,记为 *Type*($Cell_m$)=*BUFFERCELL*,若 $Cell_m$ 属于 RS_i 负责的其他 Cell,则其类型为内部 Cell,记为 *Type*($Cell_m$)=*INTERCELL*.

令 *RelativeRS*($Cell_m$)为 $Cell_m$ 的关联 RS 集合,获取一个 $Cell_m$ 直接相邻的 Cell 列表(包括对角相邻),相邻 Cell 列表中每个 Cell 所属的 RS 构成的集合即为 *RelativeRS*($Cell_m$);对于 $Cell_m$,若 *RelativeRS*($Cell_m$)元素个数大于 1,则其类型为 *BUFFERCELL*,*RelativeRS*($Cell_m$)元素个数等于 1,其类型为 *INTERCELL*.

TransferOneCell(RS_i,RS_j,$Cell_m$)子算法主要完成如下迁移相关操作:

(1) 将 $Cell_m$ 属于 RS_i 更新为 RS_j;

(2) 对 RS_i 和 RS_j 维护的缓冲区 Cell 集合 *BufferCell*(RS_i)和 *BufferCell*(RS_j)进行更新;

(3) 若 $Cell_n$ 与 $Cell_m$ 相邻,$Cell_n$ 属于 RS_t,需要对 $Cell_m$ 迁移影响到的 *RelativeRS*($Cell_n$),*Type*($Cell_n$),*BufferCell*(RS_t)进行更新;

(4) 属于 $Cell_m$ 的用户信息由 RS_i 迁移到 RS_j.

例如在负载调整过程中,区域服务器 RS_0 需要迁移 $Cell_4$ 给 RS_1,如图 4 所示,*TransferOneCell* 的处理过程如下:首先,将 $Cell_4$ 的所属 RS 进行修改,由 RS_0 改为 RS_1;再次,对 *BufferCell*(RS_0)和 *BufferCell*(RS_1)进行更新,将 $Cell_4$ 从 *BufferCell*(RS_0)中删除,加入到 *BufferCell*(RS_1)中;第三,$Cell_4$ 的迁移可能影响其相邻 Cell.以 $Cell_3$ 为例,迁移前 *RelativeRS*($Cell_3$)={RS_0}, 迁移后 *RelativeRS*($Cell_3$)={RS_0,RS_1},*Type*($Cell_3$) 需要更新为 *BUFFERCELL*, 同时,*BufferCell*(RS_0)需要增加元素 $Cell_3$;最后,需要将 $Cell_3$ 内的用户信息由 RS_0 迁移到 RS_1.算法 3 给出了 *TransferOneCell*(RS_i,RS_j,$Cell_m$)子算法伪代码实现.

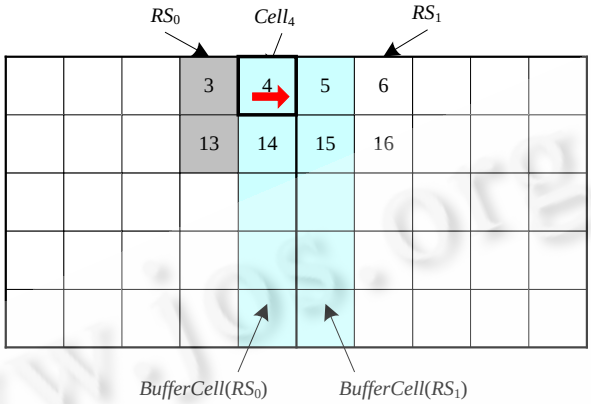


Fig.4 An example of Cell migration

图 4 迁移 Cell 处理过程示例

算法 3. 迁移 Cell 子算法 *TransferOneCell* 的伪代码.

```
TransferOneCell(RSi,RSj,Cellm)
{
    将 Cellm 所属 RS 更新为 RSj;
    BufferCell(RSi).remove(Cellm);
    BufferCell(RSj).add(Cellm);
    NBCellSet(Cellm)=getNBCellSet(Cellm);              //获取 Cellm 相邻的 Cell 集合
    for(each Celln in NBCellSet(Cellm)){
        获取 Celln 所属的 RSt;
        RelativeRSnew(Celln)=getRelativeRSnew(Celln);  //计算 Celln 新的关联 RS 集合
        //与其旧的关联 RS 集合进行比较
        if(RelativeRSold(Celln).size==1 &&
           RelativeRSnew(Celln).size>1){               //Celln 成为缓冲 Cell
            Type(Celln)=BUFFERCELL;
            BufferCell(RSt).add(Celln);
        }
        if(RelativeRSold(Celln).size>1 &&
           RelativeRSnew(Celln).size==1){              //Celln 成为内部 Cell
            Type(Celln)=INTERCELL;
            BufferCell(RSt).remove(Celln);
        }
    }
    将 Cellm 内的用户信息由 RSi 迁移到 RSj;
}
```

3 实验及结果分析

我们基于用户在虚拟环境中的两种典型分布,对分层迭代动态负载平衡算法进行验证,并与局部、全局和动态负载共享[17]3 种算法进行比较.首先介绍实验环境和测试指标,然后给出实验的结果与分析.

3.1 实验环境

用户在虚拟环境中存在均匀、倾斜和聚簇 3 种典型分布[15].因为均匀分布中各服务器的负载基本相当,不需要进行平衡,本文基于倾斜和聚簇两种分布进行算法测试,如图 5 所示.我们实现了一个负载平衡测试工具DLBTool,对基于多服务器结构的分布式虚拟环境系统进行模拟,整个虚拟环境分割成 N 个区域,由 N 个区域服务器负责,M 个用户以两种不同的分布加入到虚拟环境中,有一个主控服务器收集各个区域服务器的负载信息,必要时触发动态负载平衡机制.同时,我们实现了现有的局部、全局和动态负载共享 3 种算法.

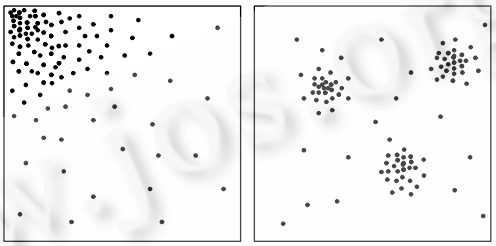


Fig.5 Skewed and clustered distribution of avatars in the virtual environment

图 5 用户在虚拟环境中的典型分布(倾斜和聚簇)

实验环境设置为主机配置:P4 2.8GHz CPU,512M 内存,Windows XP;整个虚拟环境分别划分为4,9,16,25,36 个 Region,由相应数目的 RS 来负责每个区域,每个 Region 划分为 15×15 个 Cell;$L(RS)$为 RS 的负载,可由 RS 管理的用户数进行衡量,每个 RS 的过载阈值 $OL(RS)$设置为 200, α取值 0.9,即每个 RS 的安全负载阈值 $SL(RS)=0.9\times200=180$;对于不同的 RS 数目 N,实际加入的总负载量:倾斜分布为 $N\times0.8\times SL(RS)$,聚簇分布为 $N\times0.7\times SL(RS)$;算法中最大层数 k 值取 $\sqrt{N}$.

3.2 测试指标

我们通过算法的执行效率来测试调整负载的动态性,参考文献[16,17],用算法执行前后过载状态的对比来衡量算法调整负载的有效性,用负载调整过程中引入的用户迁移来衡量算法开销,采用如下测试指标:

(1) 算法执行效率(efficiency)

用来测试算法执行速率,用算法迭代 1 次的平均时间(ms)来计算,表示为

$$Efficiency = T_{total}/C_{total}.$$

T_{total} 和 C_{total} 分别为从算法执行开始到系统到达负载平衡状态或不能调整负载状态,算法执行的总时间和总次数.本指标主要衡量迭代执行 1 次分层迭代负载平衡算法的时间,不包括主控服务器和区域服务器进行协调的网络通信时间,值越小,算法在进行负载调整时的动态性越强.

(2) 过载调整比率 OAR(overload adjustment ratio)

用来衡量负载平衡算法对一个过载虚拟环境系统进行动态负载调整后,解除负载失衡状态的有效程度.由算法执行后消除过载值的比率计算,表示为

$$OAR = \frac{\sum_i (L(RS_i) - OL(RS_i)) - \sum_j (L(RS_j) - OL(RS_j))}{\sum_i (L(RS_i) - OL(RS_i))}.$$

RS_i 为算法执行前过载的区域服务器,RS_j 为算法执行后过载的区域服务器.过载调整比率取值属于[0,1],其值越大,越能有效地消除负载失衡状态,负载调整的有效性越好.当过载调整比率等于 1 时,表明算法执行完后,所有区域服务器都没有过载.

(3) 用户迁移比率 UMR(user migration ratio)

用来衡量负载平衡算法执行过程中负载调整引入的开销.由算法执行引起迁移的用户比率计算,表示为

$$UMR = N_{migrate}/N_{total}.$$

$N_{migrate}$ 是算法引入的迁移用户的总数,N_{total} 是系统中的总用户数.UMR 越小,表示引入的迁移开销越小.

3.3 结果分析

在倾斜和聚簇两种分布情况下,对算法的执行效率进行测试,当服务器个数从 4 个增加到 36 个时,分层迭代负载均衡算法的平均执行时间略有增加,但不超过 0.2ms,如图 6 所示.算法迭代一次时间很短,可以经过多次迭代逐渐达到负载平衡.在到达负载平衡状态之前,算法每一次迭代都会获取系统最新的负载情况进行负载调整,适合于实时性强并且随时间不断变化的分布式虚拟环境系统.

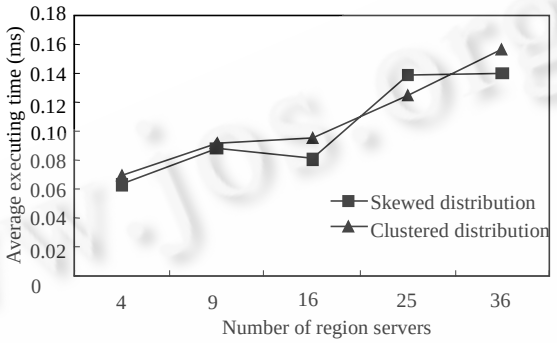


Fig.6 Average executing time of algorithm

图 6 算法执行平均时间

倾斜分布和聚簇分布下,本文算法、动态负载共享算法及全局负载平衡算法可以很好地减轻区域服务器负载过重的情况.图 7 给出了两种分布情况下过载调整比率的测试结果,本文算法的过载调整比率可以达到 92%以上,比动态负载共享算法略高,算法执行后,整个虚拟环境中基本没有过载的服务器;因为局部负载平衡算法只能将负载转移给其邻居区域服务器,所以不能很好地缓解服务器负载失衡状态,特别是当区域服务器数目增加时,局部负载平衡算法起到的作用很有限,在有 36 个服务器时,倾斜分布和聚簇分布情况下,其过载调整比率仅为 23%和 35%.

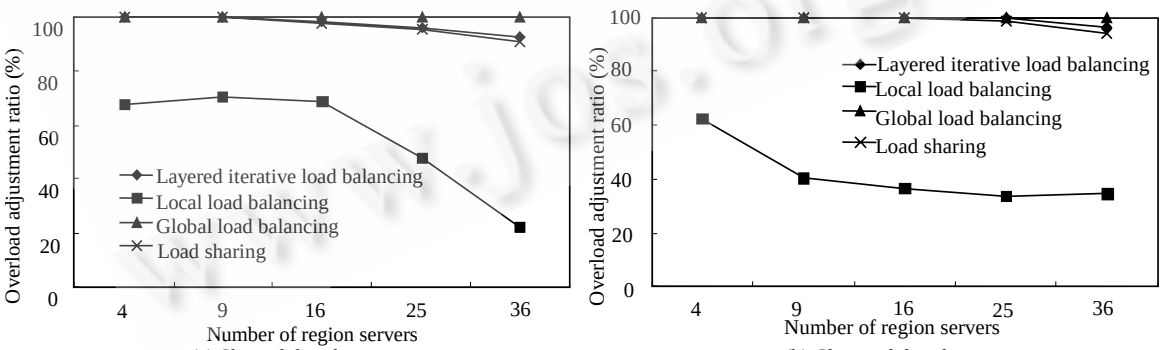
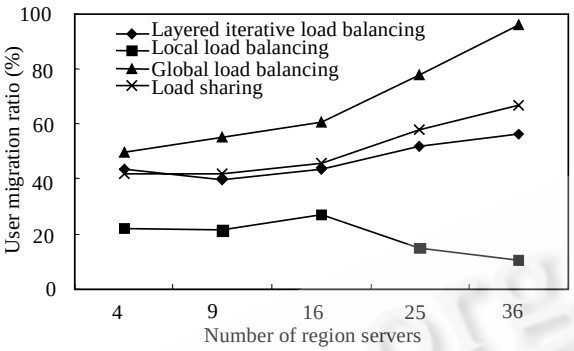


Fig.7 Overload adjustment ratio under skewed and clustered distribution

图 7 倾斜分布、聚簇分布下的过载调整比率对比

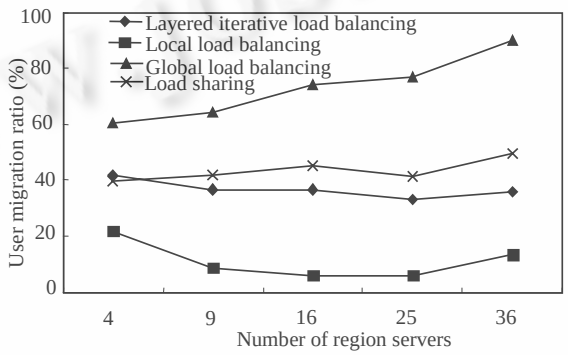
局部负载平衡算法的负载迁移比率不高于 30%,引入的迁移开销小,如图 8 所示,其原因是局部负载平衡算法仅在过载服务器和其相邻区域服务器间调整负载,虽然有小的迁移开销,但没有达到负载平衡状态,导致过载区域服务器的实时交互性下降甚至不能响应.在能够完成负载平衡的前提下,对于两种不同分布情况,本文算法

与全局负载平衡算法相比都有明显优势,并且比动态负载共享算法具有更低的负载迁移比率,最低可达 7%和13%,引入了更少的负载迁移开销,在倾斜分布情况下,迁移比率不高于 56%,聚簇分布情况下,迁移比率不高于45%.



(a) Skewed distribution

(a) 倾斜分布



(b) Clustered distribution

(b) 聚簇分布

Fig.8 User migration ratio under skewed and clustered distribution

图 8 倾斜分布、聚簇分布下的负载迁移比率对比

从上面的测试结果可以看出:在两种典型分布情况下,全局负载平衡算法过载调整比率接近 100%,能够有效地调整负载,但是与其他算法相比引入了过多的开销,特别是在服务器数目增加时,用户迁移比率增加到 90%以上;局部负载平衡算法开销小,用户迁移比率不超过 30%,但当服务器数目大于 25 个时,过载调整比率小于40%,不适合大规模分布式虚拟环境的负载平衡问题.本文算法的过载调整比率可达 92%以上,接近全局负载平衡算法,比动态负载共享算法略高,并且具有更低的负载迁移比率,在两种分布下不超过56%,在保持负载调整有效性的同时引入相对低的开销,算法迭代速度快,每次迭代的时间不超过 0.2ms.

4 结 语

针对当前动态负载平衡机制中存在的计算开销大、负载调整动态性差的问题,本文提出一种分层迭代的动态负载平衡算法,应用于多服务器结构的分布式虚拟环境系统中,以过载区域服务器为中心,分层地选择周围有限数量的区域服务器作为调整目标,将过载部分由内向外迭代地扩散到各层,通过多次迭代达到负载平衡状态.实验结果表明,本文算法迭代时间短,可以根据系统最新的负载状态进行调整负载;在倾斜和聚簇两种典型用户分布情况下,与现有的局部、全局及动态负载共享算法比较,本文算法可以有效地调整负载并且引入较少的开销.

References:

[1] Zyda MJ, Pratt DR, Monahan JG, Wilson KP. NPSNET: Constructing a 3D virtual world. In: Levoy M, Catmull EE, Zeltzer D, eds. Proc. of the 1992 Symp. on Interactive 3D Graphics. New Yowk: ACM, 1992. 147−156.

[2] Frecon E, Stenius M. DIVE: A scaleable network architecture for distributed virtual environments. Distributed System Engineering, 1998,5(3):91−100.

[3] Greenhalgh C, Purbrick J, Snowdon D. Inside MASSIVE-3: Flexible support for data consistency and world structuring. In: Churchill E, Reddy M, eds. Proc. of the 3rd Int'l Conf. on Collaborative Virtual Environments. New York: ACM, 2000. 119−127.

[4] Counter-Strike. http://www.counter-strike.net/

[5] Quake. http://www.idsoftware.com/

[6] Singh G, Serra L, Png W, Wong A, Ng H. BrickNet: sharing object behaviors on the net. In: Proc. of the Virtual Reality Annual Int'l Symp. Washington: IEEE, 1995. 19−25.

[7] Funkhouser TA. RING: A client-server system for multi-user virtual environments. In: Zyda M, ed. Proc. of the 1995 Symposium on Interactive 3D Graphics. New York: ACM, 1995. 85−92.

[8] Das TK, Singh G, Mitchell A, Kumar PS, McGee K. NetEffect: A network architecture for large-scale multi-user virtual worlds. In: Thalmann D, Feiner S, Singh G, eds. Proc. of the ACM Symp. on Virtual Reality Software and Technology. New York: ACM, 1997. 157−163.

[9] Hori M, Iseri T, Fujikawa K, Shimojo S, Miyahara H. Scalability issues of dynamic space management for multiple-server networked virtual environments. In: Proc. of the 2001 IEEE Pacific Rim Conf. on Communications, Computers and signal Processing. Washington: IEEE, 2001. 200−203.

[10] Ng B, Si A, Lau RWH, Li FWB. A multi-server architecture for distributed virtual walkthrough. In: Shi JY, Hodges L, eds. Proc. of the ACM Symp. on Virtual Reality Software and Technology. New York: ACM, 2002. 163−170.

[11] Lee D, Lim M, Han S, Lee K. ATLAS: A scalable network framework for distributed virtual environments. Teleoperators and Virtual Environments, 2007,16(2):125−156.

[12] Ultima Online. http://www.uo.com/

[13] Asheron's Call. http://ac.turbine.com/

[14] Morillo P, Orduna JM, Fernandez M, Duato J. An adaptive load balancing technique for distributed virtual environment systems. In: Gonzalez T, ed. Proc. of the 15th IASTED Int'l Conf. on Parallel and Distributed Computing and Systems. Baltimore: ACTA, 2003. 256−261.

[15] Lui JCS, Chan MF. An efficient partitioning algorithm for distributed virtual environment systems. IEEE Trans. on Parallel and Distributed Systems, 2002,13(3):193−211.

[16] Lee K, Lee D. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. In: Noma H, ed. Proc. of the ACM Symp. on Virtual Reality Software and Technology. New York: ACM, 2003. 160−168.

[17] Duong TNB, Zhou S. A dynamic load sharing algorithm for massively multiplayer online games. In: Landfeldt B, Moors T, eds. Proc. of the 11th IEEE Int'l Conf. on Networks. Washington: IEEE, 2003. 131−136.

[18] Chen J, Wu B, Delap M, Knutsson B, Lu H, Amza C. Locality aware dynamic load management for massively multiplayer games. In: Pingali K, ed. Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. New York: ACM, 2005. 289−300.

王少峰(1981－),男,山东昌邑人,硕士生,主要研究领域为分布式虚拟现实应用技术.

周忠(1978－),男,博士,副教授,CCF 会员,主要研究领域为分布式虚拟现实,计算机网络.

吴威(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布式虚拟现实,计算机网络技术与信息安全,分布式系统.