Load Balancing for Moving Object Management in a P2P Network

Mohammed Eunus Ali[1,2] Egemen Tanin[1,2] Rui Zhang[2] Lars Kulik[1,2]

[1] National ICT Australia
[2] Department of Computer Science and Software Engineering
University of Melbourne, Victoria, 3010, Australia
{eunus, egemen, rui, lars}@csse.unimelb.edu.au

Abstract. Online games and location-based services now form the potential application domains for the P2P paradigm. In P2P systems, balancing the workload is essential for overall performance. However, existing load balancing techniques for P2P systems were designed for stationary data. They can produce undesirable workload allocations for moving objects that is continuously updated. In this paper, we propose a novel load balancing technique for moving object management using a P2P network. Our technique considers the mobility of moving objects and uses an accurate cost model to optimize the performance in the management network, in particular for handling location updates in tandem with query processing. In a comprehensive set of experiments, we show that our load balancing technique gives constantly better update and query performance results than existing load balancing techniques.

Keywords: P2P Data Management, Spatial Data, Load Balancing

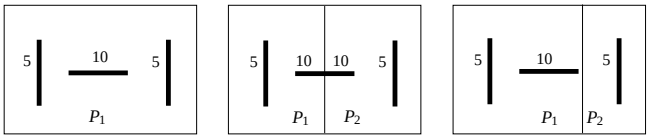
1 Introduction

Decentralized distributed systems, in particular peer-to-peer (P2P) systems, are an increasingly popular approach for managing large amounts of data. These systems do not have a single point of failure and can easily scale by adding further computing resources. They are seen as economical as well as practical solutions in distributed computing. For example, a police department could deploy a P2P system of hundreds of generic, cheap low-end processing units (nodes) for a traffic monitoring system. With similar constraints and needs, massively multi-player online games as well as new location-based services now form the potential application domains for the P2P paradigm. With recent research in P2P data management, managing complex data and queries is now a reality and such new P2P applications that go beyond file sharing are about to emerge [1].

In this paper, we consider a large set of moving objects maintained by a P2P network of processors. A moving object data management system typically assumes that objects either periodically report their locations or their location changes [2]. The workload of a moving object data management system mainly consists of two parts: handling location updates and processing queries. As it is the case for any distributed system, in a P2P

system for moving objects, balancing the workload is essential to optimize the overall performance. However, existing P2P load balancing techniques were designed for stationary data and can produce undesirable workload allocations for moving objects.

We show two examples for processing updates and queries that highlight different workload allocations. Fig. 1 shows two different approaches to partition the load from updates for two processing nodes, P_1 and P_2. Fig. 1 (a) shows the initial state with only one node, P_1, managing the entire load (20 updates). The thick lines represent roads and the numbers adjacent to them represent the number of location updates from objects, e.g., cars, moving on the roads during a period of 5 minutes.

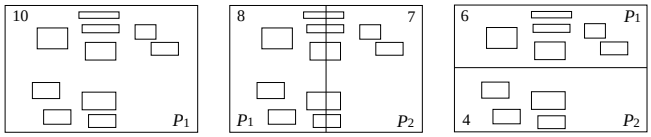


(a) (b) (c)

Fig. 1. Examples of workload partitioning for updates

A traditional load balancing scheme might partition the data space as in Fig. 1 (b) for the nodes P_1 and P_2, which results in a perfectly balanced load for P_1 and P_2. The cars on the vertical roads require 5 updates per node. The cars on the horizontal road move from one partition to the other partition and cause two updates: a new update message for the node entered and another update message to delete the object from the node left. As a result, we get a total of $5 + 10 + 5 + 10 = 30$ updates for both nodes, where each node handles 15 updates. However, if the data space is partitioned as in Fig. 1 (c), although the load is not balanced among the two nodes, we only get a total load of 20 updates instead of 30 updates. This interesting example shows that a traditional load balancing scheme can result in higher total workload than unbalanced load partitioning that optimizes the total load with respect to the movement of objects.

Unbalanced partitioning can also improve query processing. Assume the same data distribution as in Fig. 1 (a) and suppose the rectangles shown in Fig. 2 (a) represent two-dimensional range (window) queries. A traditional load balancing scheme might partition the space similar to the previous example leading to the partitioning as in Fig. 2 (b). In this case, node P_1 has to process 8 queries because the data space completely includes 3 query windows and overlaps with 5 query windows; correspondingly node P_2 has to process 7 queries, which leads to a load of 15 queries in total. If we partition the data space as Fig. 2 (c), P_1 has to process 6 queries and P_2 4 queries, which leads to a total load of 10 queries, significantly less than the more balanced partitioning.



(a) (b) (c)

Fig. 2. Examples of workload partitioning for range queries

These examples show that a traditional load balancing scheme can be inefficient for moving objects if their movement is not taken into account. This inefficiency applies to both updates and queries in dynamic spatial settings that use multiple processing nodes. Motivated by these observations, we propose a novel load balancing technique for moving object management in a P2P network. Our technique considers the mobility of moving objects to model and minimize the average cost of handling updates and processing queries. To optimize the overall system performance, we make a trade-off between balancing the workload and minimizing the extra workload overheads for crossing updates and overlapping queries. We propose to model nodes and their workloads using an undirected weighted graph, which allows us to use a graph partitioning algorithm for load partitioning. We then develop an accurate cost model that estimates the cost of handling updates and the processing of queries in order to optimize the performance of the management network. Through an extensive experimental study, we show that our spatial approach to load balancing gives constantly better update and query performance results than existing P2P load balancing techniques.

2 Related Work

Moving object data management in centralized systems has been extensively studied in [3,4,5,6,7]. Recently, for static spatial data, decentralized systems have been developed [1,8,9,10]. In addition, recent research also focused on moving objects in distributed settings [11,12]. None of these existing systems address load balancing issues for moving object data management in P2P systems.

Distributed algorithms and data structures for P2P systems have become the main research topic for large scale distributed data management since early 2000s (e.g., [13,14]). These systems rely on distributed hash tables (DHTs). DHTs maintain logical neighbor relationships between the nodes of a P2P system and each node maintains only a small set of logical neighbors for routing messages closer to the destination nodes. Among these DHTs, the Content Addressable Network (CAN) [13] uses a space partitioning mechanism that can be easily adapted for spatial data. We use CAN as a base approach for large scale moving object data management.

In CAN, with a two-dimensional setting, the data space is mapped onto a $[0, 1] \times [0, 1]$ virtual coordinate space and is divided among the nodes in a P2P system. Thus, the virtual coordinate space is used as an intermediate space to map the data space onto node addresses. Data is stored as (key, value) pairs, in which the key is deterministically mapped onto a point p in the coordinate space and the corresponding pair is stored at the processing node that owns the subregion containing p. The same mapping is used for data retrieval. For two-dimensional spatial data the mapping onto the virtual coordinate space is straightforward. For routing, each node in CAN maintains a routing table containing the IP addresses of the nodes and the extent of the sub-regions adjacent to its own subregion. A node routes a message towards its destination by forwarding it to the neighbor with coordinates which is closest to the destination coordinates. A CAN-based system is built incrementally, where a new node can randomly select an existing node to join in the system by taking over the half of the region from the existing node. Since CAN does not perform any explicit load balancing, a dedicated load balancing strategy is necessary to improve the performance.

Godfrey et al. in [15] propose a load balancing strategy for P2P systems using the concept of virtual servers. The storage and routing occurs at virtual nodes (virtual servers) rather than real nodes (processing nodes) and each real node can host one or more virtual servers. Each virtual server maintains a sub-region. Every virtual server has its own logical address defined by its sub-region and data such as the neighbor table. A virtual server uses a similar greedy algorithm as CAN. An overloaded node transfers some of its virtual servers to an underloaded node. In [15] some nodes in the system act as a load balancer. All nodes send their load information to a randomly chosen load balancing server. Based on the workload of each node, the balancers distribute the virtual servers among the participating nodes to achieve a load balanced system. It is assumed that the overheads for transferring virtual servers among the participating nodes is commonly accepted as negligible in comparison with the benefits that the system can get from load balancing. We use the virtual server based approach for moving object data as a starting point.

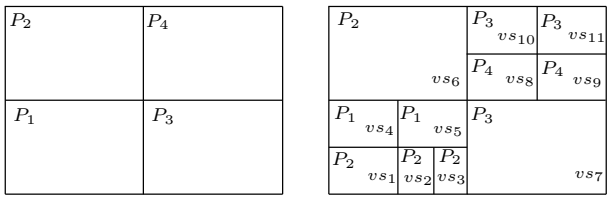
Recently, several proposals aim to address the load balancing issues for range partitioned data sets and to preserve the locality of the data. Aspens et al. [16] adopt a pairing strategy in which heavily-loaded machines are placed next to lightly-loaded machines in the data structure to simplify data migration. Similarly, Karger and Ruhl [17] achieve load balancing by periodically moving underloaded nodes next to overloaded nodes. Ganesan et al.'s [18] online load balancing algorithm achieves load balancing by adjusting partition boundaries of range partitioned data and moving data among participating nodes. These techniques are designed for static non-spatial data.

It is important to note that some earlier works [19,20] in spatial database focus on load balancing using parallel systems or a cluster of workstations. Based on the access patterns of the data these systems distribute an index to balance the workload for optimizing the query response time. However, none of these techniques consider the movement patterns of objects and thus are unable to handle extra workload overheads from moving objects.

3 Mobility-Aware Load Balancing

In this section, we propose our load balancing technique that considers the movement of objects. We named our method as Mobility-Aware Load Balancing (MALB). Our technique makes a tradeoff between balancing the workload and minimizing the workload overheads from moving objects to minimize the cost function of the system. In this section, we first give a brief overview of modeling the workload. Then we define a cost function for the system. Finally, we will describe our load balancing technique.

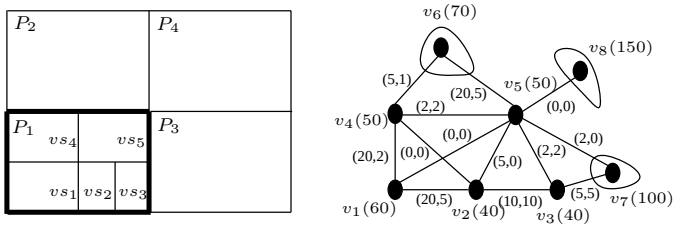
We use CAN as the underlying distributed P2P system. Fig. 3 (a) shows the assignment of four subregions to four processing nodes (or nodes) P_1, P_2, P_3, and P_4 in CAN. Furthermore, we adopt the concept of virtual servers (Section 2) for load balancing, i.e., each node maintains one or more non-contiguous subregions resulting from CAN subdivision. Each subregion is called a virtual server or virtual node. Fig. 3 (b) shows the assignment of a set of virtual nodes $\{vs_4, vs_5\}$, $\{vs_1, vs_2, vs_3, vs_6\}$, $\{vs_7, vs_{10}, vs_{11}\}$, and $\{vs_8, vs_9\}$ representing different subregions to the nodes P_1, P_2, P_3, and P_4, respectively. Note that in the virtual server approach, the load balancer does not consider the movement of objects and queries among subregions while distributing loads. For



(a) CAN (b) Virtual Servers

Fig. 3. Region assignment

example, a scenario where a large number of objects cross between vs_4 in P_1 and vs_6 in P_2, the above possibly highly balanced assignment shown in Fig. 3 (b) can result in poor performance due to communication overheads. MALB reduces these communication overheads by only allowing the assignment of virtual nodes from one node to the other if the desired transition optimizes the cost given in Section 3.1 (see equation (1)).



(a) Region assignment in the P2P network (b) Workload graph of P_1

Fig. 4. Workload representation with a graph

Fig. 4 (a) shows a set of virtual nodes hosted at the node P_1. Each virtual node keeps track of its own load, its neighbors' loads, the number of objects leaving or entering each neighboring virtual nodes, and the overlapping queries with its neighbors. We model the virtual nodes, their relations within a node, and the load relations with neighboring processing nodes as a weighted graph, $G = (V, E)$. Each vertex $v \in V$ represents a virtual node or a neighboring processing node. The weight of a vertex representing a virtual node is the total load of updates and queries of that virtual node and corresponds to the load of its subregion. If a vertex represents a neighboring processing node, its weight reflects the total load obtained from all maintained virtual nodes. An edge $e \in E$ between two vertices in V indicates that the regions represented by these two vertices share a border. Each edge e has a pair of weights (e_{oc}, e_{qc}): e_{oc} represents the number of crossing objects and e_{qc} the number of overlapping queries.

Fig. 4 (a) shows initial assignments of four subregions to four nodes P_1, P_2, P_3, and P_4. Five virtual nodes $\{vs_1, vs_2, vs_3, vs_4, vs_5\}$ hosted at the node P_1 are shown inside thick border lines. The vertices $\{v_1, v_2, v_3, v_4, v_5\}$ represent five virtual nodes of P_1 and the vertices v_6, v_7, and v_8 represent three neighboring nodes P_2, P_3, and P_4, respectively, as shown in Fig. 4 (b). The vertices v_1 and v_2 having weights 60 and 40 represent the total update and query load for the virtual nodes vs_1 and vs_2, respectively.

The edge between these two vertices is labeled as $(20, 5)$, where 20 is the number of crossing objects and 5 is the number of overlapping queries between vs_1 and vs_2. Since each crossing object results in an update operation to each of the two virtual nodes, and each overlapping query is also counted on both of the virtual nodes, the total load of these two virtual nodes is $60 + 40 - 20 - 5 = 75$. Similarly, the total load of a node is obtained by combining all loads from the virtual nodes hosted at that node.

3.1 Update and Query Costs

The performance of a moving object management system is determined by its two major tasks: handling updates and processing queries. The performance can be measured by the average response time for handling an update T_u and for processing a query T_q. The system performance can be measured by the following weighted cost function:

$$cost = w \times T_u \times N_u + (1 - w) \times T_q \times N_q, \quad (1)$$

where N_u and N_q are the total number of update requests and queries, respectively, for a period of T time units; w is a weight between 0 and 1 that adjusts the relative importance of the two operations in the system. In certain applications, immediate precise location information about the objects may be important requiring a higher weight for the update response time. A higher priority for an immediate prompt answer of a query, needs a higher weight for the query response time.

The goal of our load balancing scheme is to distribute the workload among the nodes in a P2P network such that the cost function (1) is minimized. To calculate the value of (1) for a given workload partitioning, we need to calculate T_u and T_q (Section 4).

3.2 Algorithm

In this section, we describe an algorithm, named $RegionAdjustment$ (Algorithm 1), that every node runs to trade its load by transferring some of its virtual nodes to a neighboring node. If there are multiple neighbors of a node, the node selects the neighbor that result in the highest performance improvement using the cost function. Note that we do not use any explicit load balancing servers. Instead, every node acts as its own load balancing server using only local information.

A node constructs and updates its load interaction graph $G = (V, E)$ periodically. Let V_1 and V_2 be the two initial partitions of V. V_1 is a set of vertices representing a set of virtual nodes of that node and V_2 is a set with a single vertex for the selected neighboring node. Algorithm 1 refines the initial partitions and returns two new partitions that minimizes the cost in equation (1). The cost function is minimal for V_1 and V_2 when their load is equal and they have no crossing objects and overlapping queries. The algorithm determines the vertices to be migrated from V_1 to V_2.

Algorithm 1 pair-wise adjusts the load of virtual nodes between two neighboring nodes by estimating a local approximation of the global cost function equation (1). We show an example run of the algorithm in the workload scenario given in Fig. 4. Suppose node P_1 runs the algorithm to adjust its regions with a neighbor P_2. Initially, the algorithm creates two partitions V_1 and V_2, where $V_1 = \{v_1, v_2, v_3, v_4, v_5\}$ represents the

Algorithm 1: RegionAdjustment

```
1.1  Let X be the set of border vertices between V1 and V2;
1.2  Let Y be an empty set of vertices;
1.3  improving = true;
1.4  Let initcost be the value of cost using equation (1) for the initial partitions;
1.5  mincost = initcost;
1.6  while improving do
1.7      improving = false;
1.8      while X is not empty do
1.9          for each border vertex x ∈ X do
1.10             Let cost[x] be the value from equation (1) for the partitions assuming x is
                 transferred to the other partition;
1.11         Let v be a vertex in X, such that cost[v] = min_{x∈X} cost[x];
1.12         Transfer vertex v from the current partition to the other partition;
1.13         if cost[v] < mincost then
1.14             improving = true;
1.15             mincost = cost[v];
1.16             Confirm the vertex migration from the current partition to the other;
1.17             Clear set Y;
1.18         else
1.19             Put vertex v in a temporary set Y;
1.20         Remove v from X and set visited flag for v;
1.21         Update X by adding non-visited new border vertices to X and by removing
             non-border vertices from X;
1.22     for each vertex y ∈ Y do
1.23         Transfer the vertex y from the current partition to the other;
1.24     Clear visited flag for all the vertices;
1.25     Populate X with border vertices excluding the vertices in Y for the next iteration;
```

set of virtual nodes of P_1 and the set $V_2 = \{v_6\}$ corresponds to the neighboring node P_2. In this scenario, X contains the border vertices $\{v_4, v_5\}$, i.e., the set of vertices that shares a border with a vertex in the other partition. Initially, the workload of nodes P_1 and P_2 are 160 and 70, respectively, and the crossing objects and the overlapping queries between these two nodes are $(20 + 5)$ and $(5 + 1)$. The workload of these two nodes can be calculated from the two sets of vertices V_1 and V_2. The response time for each of the 25 crossing updates is determined by the sum of the required update time in the two participating nodes. The response time for each of the 6 overlapping queries is determined by the maximum response time of both nodes. Again, the response time of all other non-crossing updates and non-overlapping queries only depends upon the service time of the corresponding node. Using these workload conditions, the algorithm determines the cost using equation (1) as initcost and determines the initial mincost.

The algorithm calculates the cost for each of the vertices in X in Line 1.9 and 1.10. The cost of a border vertex is the cost in equation (1) for the modified partitions if the vertex migrates to the other partition. If v_5 migrates from V_1 to V_2, the load of V_1

and V_2 would be 123 and 95, respectively, with 9 crossing updates and 4 overlapping queries. Assume the cost for v_5, $cost[v_5]$, is the minimal cost for all candidate vertices. If $cost[v_5]$ is less than the previous value mincost, then we expect that the migration of v_5 from V_1 to V_2 reduces the overall weighted cost function. Therefore, v_5 migrates from V_1 to V_2 and we update the cost variables in lines 1.14–1.17. This process continues as long as it minimizes the cost function. This algorithm aims to balance the load while minimizing the overhead for crossing updates and overlapping queries.

In summary, the outer loop refines the two initial partitions (lines 1.6–1.25). The inner loop (lines 1.8–1.21) checks for each border vertex if its migration minimizes the cost function. To avoid local minima in Algorithm 1, vertices can temporarily move from one partition to the other (lines 1.18–1.19) even if this move does not immediately optimize the cost. If the algorithm does not find any minima, these vertices are put back (lines 1.22–1.23). Then, the variables are re-set for the next iteration of the outer loop (lines 1.24–1.25). The algorithm stops if no vertex is found whose migration further minimizes the cost.

Our balancing scheme considers both periodic and emergency load balancing measures. Each node periodically wakes up after a time period t_p and runs Algorithm 1 to balance its load with its neighbors. A carefully chosen value of t_p can balance two extreme conditions: a small value can lead to oscillations among participating nodes, and a high value may result in an imbalanced system. To avoid emergencies such as a sudden burst of traffic load, each node ensures that its load does not become higher than some threshold k_n in comparison with the load of its neighbors. A node locks all of its neighbors before running Algorithm 1 to avoid inconsistencies that may arise from concurrent load adjustment procedures among neighbors.

The sole use of local load balancing measures cannot guarantee an optimal load balance among all nodes, specially at dynamic load conditions (e.g., large traffic load for a city center in the morning hours). However, in a P2P system, an all-to-all communication based global load balancing scheme may be problematic in itself. Thus, we adopt a scheme from [18] where the system maintains a skip-graph data structure built on load conditions of the nodes, and can find the nodes with a maximum or minimum load in $O(\log n)$ time. Then, an overloaded node can share the load with the minimum loaded node if the load of the overloaded node is k_g (a threshold value) times higher than the minimum load. Similarly, an underloaded node can share the load with the maximum loaded node. If a new node joins the system or an existing node changes its position to split an overloaded node, the serving node for this request splits its virtual nodes into two sets of virtual nodes. It then retains a set of virtual nodes and gives the other set to the requester. The *Split* procedure is very similar to Algorithm 1. In this paper we do not consider the costs for transferring virtual nodes among the participating processing nodes because this cost is negligible in comparison to total workload for handling large updates and processing queries for a period of time.

4 Cost Model

In algorithm RegionAdjustment, we need to calculate the cost function (1) given a workload partitioning. To calculate the value of (1), in this section, we derive a model to

estimate T_u and T_q based on the system knowledge of updates, queries, and the service rate of the processing nodes for a period of T time units.

When update or query requests arrive at a high rate, these requests are queued up in different nodes in the form of messages. In our system, a message is an operation that can be served in one node, while a request may consist of (or create) several messages. A request (update or query) may need to travel several nodes for the desired objective in a P2P system. For example, a range query request that spans over four nodes generates four messages (one message per node). Similarly, an update request may generate two messages (a delete message to the leaving node and an insert message to the entering node) when an object crosses the border of two nodes. A new update or a query message in a node has to wait until all the pending messages are served by the node. We assume that the objects in a node are maintained in a in-memory structure (disk-based structures are not suitable for a very high update rate) so that an update or query message can be processed in a very short amount of time, which is much faster than the queuing time of the message. Therefore, the response time of a message is actually the average queuing time; and the workload is proportional to the number of messages. Given this assumption, we can derive the average response time of a message in a node i as follows.

Let λ_{u_i} and λ_{q_i} be the update and query message arrival rates, respectively, at node i. The total arrival rate at node i is $\lambda_i = \lambda_{u_i} + \lambda_{q_i}$. Let the service rate of node i be μ_i, that is, node i can serve messages at the rate μ_i and assume $\mu_i > \lambda_i$. A node can be seen as a $M/M/1$ queue [21] (where M stands for "Markovian", implying exponential distribution for service times or inter-arrival times). According to Little's law [22], the average queue length (Q) and Average Queueing Time (AQT) of node i are given by the following equations:

$$Q_i = \frac{\frac{\lambda_i}{\mu_i}}{1 - \frac{\lambda_i}{\mu_i}} \tag{2}$$

$$AQT_i = \frac{Q_i}{\lambda_i} \tag{3}$$

According to the analysis given in the previous paragraph, the average response time of a message in node i is also given by (3).

During a period of T time units, $\lambda_{u_i} \times T$ update messages (including update messages created due to objects crossing nodes) arrive and the average response time of a message is AQT_i, so the total time to process the update messages is $\lambda_{u_i} \times T \times AQT_i$. If there are n nodes in the P2P system and the update request rate of the whole system is γ (the actual update requests from data sources), then there are $\gamma \times T$ update requests. So the average service time for an update request T_{us} is given by:

$$T_{us} = \frac{\sum_{k=1}^{n}(\lambda_{u_i} \times T \times AQT_i)}{\gamma \times T} = \frac{\sum_{k=1}^{n}(\lambda_{u_i} \times AQT_i)}{\gamma} \tag{4}$$

However, objects that cross the borders of nodes cause communication overheads from extra update messages. Assume an object crosses the border of a node with a probability p (i.e., the number of objects that crosses the border / the total number of objects) and the average communication time between two nodes is T_c. Then average communication overhead of an update request T_{uc} is $T_{uc} = p \times T_c$.

By adding the service time and the communication time of an update request, we get the average response time of an update request as follows:

$$T_u = T_{us} + T_{uc} \tag{5}$$

Similarly, a query may create several messages for all the intersected nodes by the query. Every intersected node processes its query message and return the answer to the query originating node. The query originating node combines the answers from all the intersected nodes to obtain the answer for the query. Therefore, the response time of the query is the maximum of the response times of all the messages created by the request.

In our system, the total geographic space is normalized into a $[0, 1] \times [0, 1]$ coordinate space and is partitioned among n participating nodes. A window query q of size $w \times h$ whose top-left corner is at (x, y), may intersect 1 to n nodes depending on the location and the size of the query. Let function $f(q_{x,y})$ return the number of nodes k that q intersects. Thereby q creates k query messages and the response time of these query messages are $RT_1, RT_2, ..., RT_k$, respectively, from k participating nodes. Then the response time of q is $\max\{RT_1, RT_2, ..., RT_k\}$. The other cost involved for q is the communication overhead $T_{c_{j\rightarrow i}}$ from a source node j (query originating node) to a destination node i. Thus, the response time to the query q, that intersects $f(q_{x,y})$ many nodes can be defined as:

$$g(x, y, q) = \max_{1 \le i \le f(q_{x,y})} [T_{c_{j\rightarrow i}} + RT_i(Q_i, q_{x,y})] \tag{6}$$

Given a query size $w \times h$, if we sum up the response time of queries of all possible query locations in the data space and divide the sum by the total number of queries, we find the average response time of the queries for query size $w \times h$ as follows:

$$T_q = \frac{1}{(1-w) \times (1-h)} \int_0^{1-w} \int_0^{1-h} g(x, y, q) dx dy \tag{7}$$

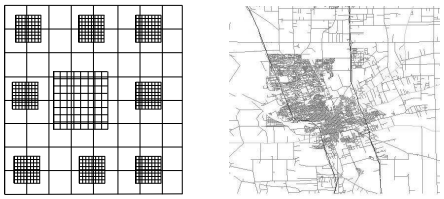
We have derived both T_u and T_q. Substitute T_u and T_q in function 1 by 5 and 7, respectively. We can calculate the cost for a given workload partitioning given the information on crossing objects, update rate, query rate, service rate, etc. However, as in a P2P network each node only has information about its own load and the load of its neighbors, we can only locally calculate the cost and optimize the performance.

5 Experimental Study

We compare the performance of our load balancing algorithm MALB with the virtual server (VS) load balancing technique from [15] on an experimental setup for location-based services.

5.1 Experimental Setup

We use both synthetic and real road networks as shown in Fig. 5. The synthetic road networks are constructed by connecting a large number of small, grid-like, road networks, modeling a set of suburbs connected to each other with freeways. A larger embedded grid is also used as a metropolitan city center. The real road network is from



(a) Synthetic (b) Real

Fig. 5. Road networks

the city of Stockton in San Joaquin County, CA. Again, the density of the network is more at the city center in comparison to the other areas. The movements of the objects within the road networks are generated by the Network-based Generator of Moving Objects [23]. We have used J-Sim [24] to develop our simulation environment, and also used BRITE [25] topology generation tool to create a communication network topology. Each of our nodes are connected with its neighbors through high bandwidth lines (2Gbps). We build the network in an incremental fashion, where each node contacts an existing node to join the system. A summary of experiment parameters (default parameters are shown in bold) are given in Tab. 1. We run each simulation multiple times and give the averages in our results. As we shall see, simple CAN cannot scale well for a skewed load distribution, we present the performance of our technique in comparison to the VS load balancing technique.

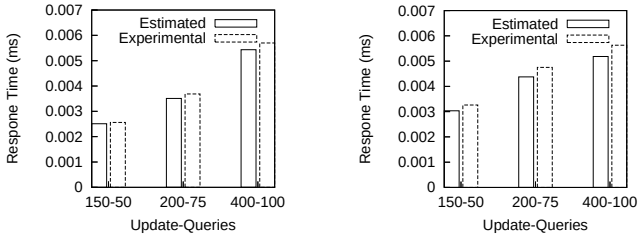
Parameter	**Value**
Road network data	Synthetic, Real
No. of nodes	100
Update arrival rate (per sec.) (λ_u)	4K, 6K, **8K**, 10K, 15K
Query arrival rate (per sec.) (λ_q)	200, **400**, 800, 1600
Service rate in a node (per sec.) (μ)	400
No. of crossing updates in % of all updates	5, 15, **25**, 35, 45
Query size in % of whole data space	1, **5**, 10, 20
Average no. of virtual nodes per node	8
Periodic load balance timeout (sec.)	10
Emergency load balance parameters	$k_n = 2$, $k_g = 4$

Table 1. Summary of parameters for the experiments

5.2 Evaluation of the Cost Model

We have measured the accuracy of our cost model given in equation (1) using a two node setting. We have chosen a two node setting as our algorithm works on individual nodes and only uses the local neighborhood information available to that node to calculate the cost. For various arrival and service rates, we first calculate the value of the cost

function (1), F_{est}. Then we run the experiments to find the experimental values, the actual cost F_{exp}. Finally, we obtain the accuracy of the cost model by using $\frac{|F_{est}-F_{exp}|}{F_{est}}$, that is, the relative error, as the metric. The accuracy of the cost function depends on how well we can estimate the average update response time (AURT) in equation (5) and the average query response time (AQRT) in equation (7). We plot these two values as functions of different arrival rates in Fig. 6 (a) and 6 (b), respectively. We found that the deviation of the cost function from experimental values is always less than 10%.



(a) Updates (b) Queries

Fig. 6. Accuracy of the cost model

5.3 Scalability

Fig. 7 shows that the AURT increases with an increase in update arrival rate and CAN cannot sustain its performance with the increasing load. The graph also shows that the improvements from our approach over the VS remains constant up to an arrival rate of 6000. The improvement increases with the increase in the update arrival rate (up to 28% from the VS approach). The reason for this is, more updates would mean that there are more crossings of objects in the system. In this case, the VS load balancing needs to handle more load due to overhead crossings. Our technique continues to scale better by reducing the overhead crossings from updates.

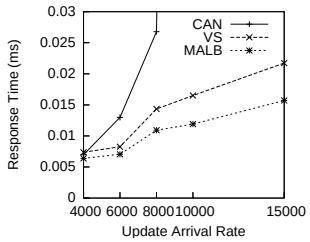


Fig. 7. Effect of update rate

5.4 Effect of w

We measure the value of the cost function defined in (1) by varying the weight, w. As w increases, the value of the cost function increases as we put more weight on updates and the number of updates is much more than the number of queries in the system (Fig. 8). We get a reduced average response time for location updates if $w = 1.0$ and, interestingly, also a good query performance. As we reduce the updates between regions this also reduces the total load in the system leading to good query performances.

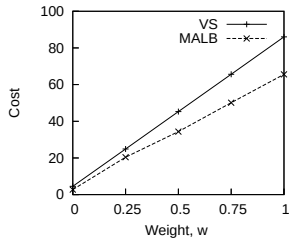


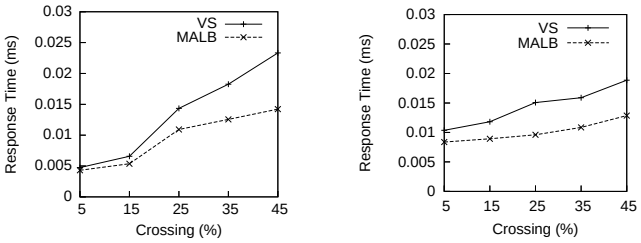
Fig. 8. Effect of weight, w

5.5 Effect of the Number of Region Crossing Object Updates

In this experiment we study the relative performance of the various techniques as the number of crossings among neighboring regions is varied from 5% to 45%. The AURT and AQRT with varying crossing updates are shown in Fig. 9. In Fig. 9 (a), we set $w = 1.0$ because we want to optimize the update performance in the system. Here the x-axis represents the percentage of crossing updates with respect to the total number of updates and the y-axis shows the average response time for an update. The results show that as the number of crossings is increased from 5 to 45 percent, our approach outperforms VS load balancing technique by a large margin, up to 40%. Since the extra workload overheads increase with the increase of crossing updates in the system, and traditional load balancing techniques (e.g., VS) are not aware of these overheads, the performance of the system degrades sharply with larger crossings. On the contrary, MALB reduces crossing overheads while balancing the workload, and thus perform much better than VS. Similarly, Fig. 9 (b) shows the comparison of two load balancing techniques over queries. This figure shows that even for a small number of crossings (i.e., 5%), the improvement is 19%, and as we reach to 45% the improvement becomes equal to 36%.

5.6 Effect of Query Size

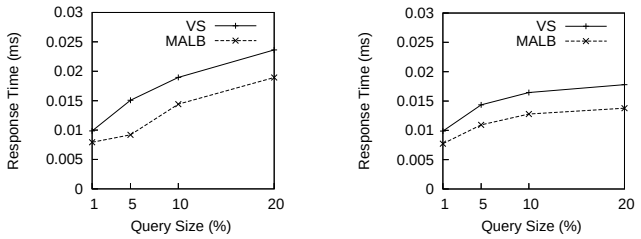
We have also run experiments by varying the query sizes. Fig. 10 (a) shows that the AQRT increases with an increase in the query size because larger queries overlap with more nodes and thereby generate more load for the system. MALB performs 35% better



(a) Updates ($w = 1.0$) (b) Queries ($w = 0.0$)

Fig. 9. Effect of crossing objects

than the VS approach for 5%, whereas the improvement is only approximately 20% when the query size is 20%. By using a pair-wise only local decision making method we can find better load balancing solutions for smaller query sizes (i.e., 5% -10%). Also, we see that in the case of update performance as shown in Fig. 10 (b), the improvement remains almost constant over the VS approach while the query size varies.



(a) Query Response Time (b) Update Response Time

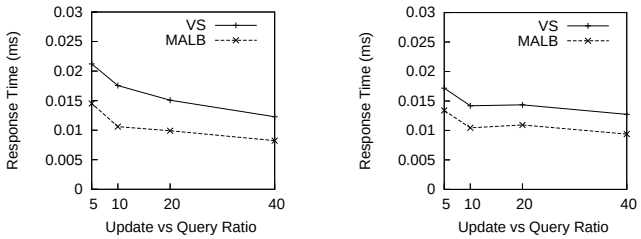
Fig. 10. Effect of varying the query size

5.7 Effect of Update to Query Ratio

We vary the update to query ratio from 5 to 40. We keep the update rate constant (8000) while varying the query rate. Fig. 11 (a) shows that our system achieves high performance gains for queries when the update to query ratio is small. As the number of queries increases, MALB can optimize more on the query performance. Fig. 11 (b) shows that the performance gain for updates does not vary as much with the increasing number of queries in the system. Therefore, our technique can achieve high query performance while still being very efficient for updates.

5.8 Experiments on a Real Road Network

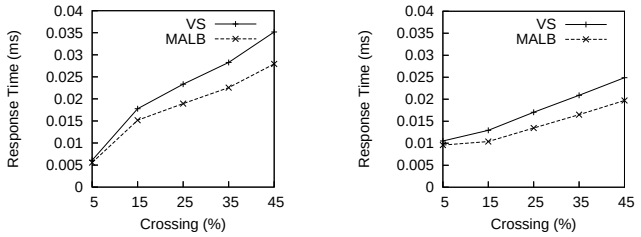
We have run our experiments on a real city road network Stockton in San Joaquin County, CA. We use 32 nodes to share the load for this small setting where the update and query arrival rates are 2000 and 200, respectively. The rest of the parameters are



(a) Queries ($w = 0.0$) (b) Updates ($w = 1.0$)

Fig. 11. Effect of update to query ratio

same as in the previous experiments. This setting presents a challenge for our local load balancing algorithm as the city has a dense center with many small roads, only a few highways, and no suburbs (Fig. 5 (b)). Thus, local decisions on the center have a smaller impact on the global load balance. However, we still see in Fig. 12 that MALB outperforms the VS approach and the gains can be up to 20%.



(a) Updates ($w = 1.0$) (b) Queries ($w = 0.0$)

Fig. 12. Effect of crossing objects in a real road network

6 Conclusions and Future Work

In this paper, we proposed a novel mobility-aware load balancing (MALB) technique for moving object management in a P2P system. MALB considers the movement patterns of objects and achieves better performance than traditional load balancing schemes, which were designed for stationary data. In addition, we optimized the performance of handling updates in tandem with the processing of queries. Through experiments, we show that our load balancing scheme results in constantly better update and query performance results than existing load balancing techniques and the improvement is up to 40%. We show that we can find better load partitions that reduce communication and processing overheads by reducing object updates and queries that span multiple processors. However, accessing information available only at the neighbors can lead to suboptimal results in comparison to a global optimization strategy. Yet, it is not practical to devise a trivially centralized load balancer for a large-scale P2P system. As a direction for future work, we plan to build on our existing pair-wise load balancing scheme to include clustering-based techniques.

References

1. Tanin, E., Harwood, A., Samet, H.: Using a distributed quadtree index in peer-to-peer networks. VLDB Journal **16**(2) (2007) 165–178
2. Wolfson, O., Xu, B., Chamberlain, S., Jiang, L.: Moving objects databases: Issues and solutions. In: SSDBM, Capri, Italy (1998) 111–122
3. Cheng, R., Xia, Y., Prabhakar, S., Shah, R.: Change tolerant indexing for constantly evolving data. In: ICDE, Tokyo, Japan (2005) 391–402
4. Guting, R.H., Schneider, M.: Moving Objects Databases. Morgan Kaufmann (2005)
5. Lee, M., Hsu, W., Jensen, C., Cui, B., Teo, K.: Supporting frequent updates in R-trees: A bottom-up approach. In: VLDB, Berlin, Germany (2003) 608–619
6. Tao, Y., Papadias, D., Sun, J.: The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In: VLDB, Berlin, Germany (2003) 790–801
7. Song, Z., Roussopoulos, N.: Hashing moving objects. In: MDM, Hong Kong (2001) 161–172
8. Guan, J., Wang, L., Zhou, S.: Enabling GIS services in a P2P environment. In: CIT, Wuhan, China (2004) 776–781
9. Shu, Y., Ooi, B.C., Tan, K.L., Zhou, A.: Supporting multi-dimensional range queries in peer-to-peer systems. In: P2P, Konstanz, Germany (2005) 173–180
10. Wang, S., Ooi, B.C., Tung, A., Xu, L.: Efficient skyline query processing on peer-to-peer networks. In: ICDE, Istanbul, Turkey (2007) 1126–1135
11. Denny, M., Franklin, M., Castro, P., Purakayasatha, A.: Mobiscope: A scalable spatial discovery service for mobile network resources. In: MDM, Melbourne, Australia (2003) 307–324
12. Gedik, M.B., Liu, S.M.L.: Mobieyes: A distributed location monitoring service using moving location queries. IEEE Transactions on Mobile Computing **5**(10) (2006) 1384–1402
13. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content-addressable network. In: SIGCOMM, San Diego, CA (2001) 161–172
14. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for Internet applications. In: SIGCOMM, San Diego, CA (2001) 161–172
15. Godfrey, B., Lakshminarayanan, K., Surana, S., Karp, R., Stoica, I.: Load balancing in dynamic structured P2P systems. In: INFOCOM, Hong Kong (2004) 2253–2262
16. Aspnes, J., Kirsch, J., Krishnamurthy, A.: Load balancing and locality in range queriable data structures. In: PODC, Newfoundland, Canada (2004) 25–28
17. Karger, D., Ruhl, M.: Simple efficient load-balancing algorithms for peer-to-peer systems. In: IPTPS, San Diego, CA (2004) 131–140
18. Ganesan, P., Bawa, M., Garcia-Molina, H.: Online balancing of range-partitioned data with applications to peer-to-peer systems. In: VLDB, Toronto, Canada (2004) 444–455
19. Kriakov, V., Delis, A., Kollios, G.: Management of highly dynamic multidimensional data in a cluster of workstations. In: EDBT, Heraklion, Greece (2004) 748–764
20. Lee, M.L., Kitsuregawa, M., Ooi, B.C., Tan, K., Mondal, A.: Towards selftuning data placement in parallel database systems. In: SIGMOD, Dallas, TX (2000) 225–236
21. Penttinen, A.: Kendall's notation for queuing models. Introduction to Teletraffic Theory, Lecture Notes: S-38.145, Helsinki University of Technology (1999)
22. Little, J.D.C.: A proof of the queueing formula $l = \lambda w$. Operations Research **9** (1961) 383–387
23. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2) (2002) 153–180
24. J-SIM: http://www.j-sim.org/
25. BRITE: http://www.cs.bu.edu/brite/