

## Latency-Driven Replica Placement

MICHAL SZYMANIAK,<sup>†</sup> GUILLAUME PIERRE<sup>†</sup>  
and MAARTEN VAN STEEN<sup>†</sup>

This paper presents HotZone, an algorithm to place replicas in a wide-area network such that the client-to-replica latency is minimized. Similar to the previously proposed HotSpot algorithm, HotZone places replicas on nodes that along with their neighboring nodes generate the highest load. In contrast to HotSpot, however, HotZone provides nearly-optimal results by considering overlapping neighborhoods. HotZone relies on a geometric model of Internet latencies, which effectively reduces the cost of placing  $K$  replicas among  $N$  potential replica locations from  $O(N^2)$  to  $O(N \cdot \max(\log N, K))$ .

### 1. Introduction

Replication is commonly employed by modern distributed systems to improve the communication delay experienced by their clients. Such systems typically deploy several replicas of their service in different parts of the Internet and automatically redirect each client to its proximal replica<sup>1)</sup>. Doing that can significantly improve the communication delay between the client and the service, resulting in a better client experience.

An important issue that must be addressed before a service can be replicated is where to place replicas. Different replica placements are likely to result in different client-experienced communication delay, hence this decision is crucial for the performance of the entire system.

Several algorithms have been proposed to address the replica placement problem<sup>2)</sup>. The Greedy algorithm, for example, places replicas one-by-one, each time exhaustively evaluating all the possible replica locations. It has been shown to produce very good placements, yet its computational cost is quite large:  $O(K \cdot N^2)$ , where  $K$  is the number of replicas, and  $N$  is the number of potential replica locations. Another algorithm, called HotSpot, places replicas on nodes that along with their neighbors generate the greatest load. It has a slightly lower computational cost:  $O(N^2 + \min(N \cdot \log N + K \cdot N))$ . On the other hand, its produced placements are not as good as those of Greedy. The high computational cost of these two algorithms prevents them from scaling for systems with more than  $10^4$  potential replica locations<sup>2)</sup>. This bound is

unacceptable for current worldwide replicated systems, as they need to consider at least  $10^5$  of such locations<sup>1)</sup>.

We propose to reduce the time needed for placement computation by taking a two-step approach. The first step is to select *network regions* where replicas should be placed. A network region is a group of nodes whose latencies to each other are relatively low. Other properties of nodes, such as the available storage space, network connection bandwidth, and availability, are irrelevant at this stage. This simplification is likely to accelerate the corresponding region-selection algorithm, especially if the latencies are modeled efficiently.

Once the network regions have been identified, the second step is to choose individual replica-holding nodes in different regions. This time, however, it is possible to consider all the factors ignored during the first step, as the number of nodes in a region is much smaller. Note that this two-step approach prioritizes client-to-replica latency over the other metrics. Since these metrics tend to be node-specific, we believe that it is not necessary to consider them at the global level.

This paper proposes HotZone, an algorithm that addresses the first step, that is the identification of the regions where replicas should be placed. For evaluation purposes, we assume that the second-step algorithm simply selects the node with minimal average distance to all other nodes in the region.

HotZone relies on the fact that Internet latencies can be modeled in an  $M$ -dimensional geometric space<sup>3)</sup>. In this model, nodes are assigned  $M$ -dimensional coordinates. The latencies between any two nodes are modeled as the distance between their corresponding coordinates.

---

<sup>†</sup> Vrije Universiteit Amsterdam Department of Computer Science Amsterdam, The Netherlands

dinates. HotZone identifies network regions as groups of nodes with proximal coordinates and places replicas in the most active regions. In this way, it avoids the costly pairwise latency estimations between all potential replica locations. This reduces the computational cost of HotZone to  $O(N \cdot \max(\log N, K))$ , which is significantly lower than those of the previously proposed algorithms.

We compare HotZone with four other algorithms, including Greedy and HotSpot. We show that the placement quality offered by HotZone is on average off by 5% from that of Greedy. We also show that HotSpot does not produce satisfactory results when directly applied to node coordinate sets, and discuss how a simple modification can make this algorithm achieve the performance close to that of HotZone, although its computation cost remains unsatisfactory. Finally, we demonstrate that the lower complexity of HotZone leads to significant gains in placement computation times, up to 3 orders of magnitude when placing 20 replicas.

The rest of this paper is structured as follows. Section 2 discusses relevant research efforts. Section 3 describes the details of our replica placement algorithm, and analyzes its computational complexity. Section 4 evaluates the algorithm performance. Finally, Section 5 concludes.

## 2. Related Work

### 2.1 Replica Placement

Many replica placement algorithms have been proposed in the literature<sup>2)</sup>. In general, their goal is to either optimize the client performance given an existing infrastructure, or minimize the infrastructure cost while achieving given client performance at the same time. This performance goal can be expressed by means of many different metrics, but placement algorithms typically optimize only one of them: client-to-replica latency. Since it corresponds to the actual communication delay experienced by the client, it is also likely to determine the client-observed performance.

Several placement algorithms try to optimize the client-to-replica latency by minimizing the hop count between clients and replicas<sup>4),5)</sup>. The underlying assumption is that the latency of a network path mainly depends on the number of individual links that this path consists of. Optimizing hop counts is attractive, be-

cause they are relatively stable and easy to measure<sup>6)</sup>. On the other hand, the accuracy of hop-based latency estimation is relatively poor<sup>7)</sup>.

Since the replica placement problem is NP-complete, placement algorithms optimize their chosen metrics by means of heuristics<sup>8)</sup>. An example of such a heuristic is the Greedy algorithm, which determines replica locations one-by-one<sup>9)</sup>. It starts with exhaustively evaluating all possible locations for the first replica, and then choosing the location yielding the best performance. The subsequent replicas are placed in the same manner, except that starting from the second replica, all the previously placed replicas are considered to be fixed during metric calculation. Greedy can be used to optimize any metric. When optimizing the hop count metric, the Greedy algorithm produces placements that have been shown to be within a factor of 1.5 of those yielding optimal client performance.

The computational complexity of the Greedy algorithm is  $O(K \cdot N^2)$ , where  $K$  is the number of replicas to be placed and  $N$  is the number of potential replica locations. In a large-scale distributed system,  $N$  can be very large, leading to long-lasting computations. In such cases, the value of  $N$  can be reduced by clustering<sup>10)</sup>. However, even relatively coarse-grained clustering may be unable to reduce  $N$  to a value for which the Greedy algorithm can be run efficiently. For example, clustering based on Autonomous Systems typically groups nodes into a few thousands of Autonomous Systems, which means that the Greedy algorithm still has to perform millions of latency estimations to place a single replica.

Another heuristic, called *HotSpot*, places replicas on nodes that generate the greatest load<sup>9)</sup>. It first orders the nodes according to the amount of traffic generated jointly by each node and its neighboring nodes. Then, it places replicas on the first  $K$  nodes with the most active neighbors. Being a neighbor node is defined in terms of some network distance metric. For example, a node can consider all nodes whose latency is less than  $X$  milliseconds to be its neighbors. The placements produced by the HotSpot algorithm have been shown to be within a factor of 1.6 – 2.0 of the optimal one when optimizing on the hop count metric. Compared to the Greedy algorithm, the drop in efficiency is traded for only a slightly more attractive complexity, which is

$O(N^2 + \min(N \cdot \log N + N \cdot K))$ . This can still be too much for globally-distributed systems.

Since several heuristics are capable of producing a placement decision, it may be desirable to dynamically select the one that produces the best results for a given system topology and workload<sup>8)</sup>. A naive solution could be to choose the best heuristic after simulating all the possible ones. This is often infeasible, as such a simulation is costly in terms of time and computing resources. Additionally, heuristic selection may have to be performed every time the system workload changes.

A promising alternative for costly heuristic simulations is model-based heuristic analysis<sup>8)</sup>. The authors propose to model each placement heuristic in terms of an integer programming formulation of the replica placement problem. It is then possible to calculate an upper bound for the performance offered by each heuristic given a system topology, a workload, and a performance goal. Assuming that the actual performance of a heuristic is close to the upper bound, one may select the best heuristic as the one with the highest upper bound.

The model-based heuristic analysis may accelerate the process of heuristic selection by saving the time otherwise needed for heuristic simulations. However, it is still required that all the heuristics are modeled and exhaustively evaluated, which can still be too costly, especially in the case of large systems. Moreover, if the number of constraints that must be considered by an algorithm is low, we may be able to identify a heuristic that invariably produces near-optimal results. In that case, the two phases required by the model-based analysis may simply turn out to be redundant.

## 2.2 Latency Estimation

Latency-driven replica placement algorithms require that latencies between each pair of nodes can be estimated. A naive approach could be to let each node measure its latencies to all the other nodes. However, this may be infeasible if the number of nodes in the system is large, or if some of them remain out of the system's control. For example, in a Content Delivery Network, most of the nodes are Web clients, which typically cannot be controlled by the CDN operator.

A promising technique for latency estimation has been proposed in GNP, which models the Internet as an  $M$ -dimensional space<sup>11)</sup>. GNP estimates the latency between any pair of

nodes as the Euclidean distance between their corresponding  $M$ -dimensional coordinates. A typical value for  $M$  is 6. The coordinates of node  $X$  are calculated based on the measured latencies between  $X$  and  $Z$  designated "landmark" nodes, where  $Z$  is slightly larger than  $M$ . Consequently, estimating pair-wise latencies between  $N$  nodes requires much fewer measurements ( $N \cdot Z$ ) than in the naive approach ( $\frac{1}{2}N^2$ ). The GNP authors show that, in 90% of cases, the latency estimations derived from their system satisfy the following constraint:

$$\frac{2}{3}L_{observed} \leq L_{estimated} \leq \frac{3}{2}L_{observed}$$

where  $L_{observed}$  and  $L_{estimated}$  denote the observed and estimated latency, respectively. This relatively large error margin limits the applicability of GNP to systems that can tolerate such estimation inaccuracies. Since our algorithm operates on groups of nodes, it does not require high estimation precision. Consequently, we can rely on GNP to provide latency estimations at low cost.

Apart from facilitating latency estimation, the coordinates produced by GNP can also be used for node clustering<sup>12)</sup>. Node clustering reduces the number of nodes that an algorithm must process as it treats each group as a single node. Since nodes with similar GNP coordinates are supposed to have minimal latency between them, coordinate-based clustering is natural in latency-optimizing systems.

In our earlier work, we have demonstrated that a system of distributed cooperating Web servers can employ GNP to locate Web clients such that the positioning application is completely transparent to the clients<sup>3)</sup>. In the next sections, we show how such a system can efficiently calculate nearly-optimal replica placements.

## 3. Algorithm

The goal of a replica placement algorithm is to determine which nodes should host replicas. While choosing these nodes, the algorithm optimizes on a certain metric, such as client-to-replica latency. Deciding on replica placement, however, typically requires that other factors are considered as well, such as the amount of storage space available at candidate replica-holding nodes, or their network connection speed.

To solve this problem, we take a two-step ap-

proach. The first step is to select *network regions* where replicas should be placed. A network region is a group of nodes whose latencies to each other are relatively low. Since other node properties are irrelevant for region selection, only the relative latencies between nodes must be analyzed at this stage.

Once the network regions have been identified, the second step is to choose individual nodes that shall act as replica servers in different regions. During node selection, we consider various node-specific factors ignored during the region selection. However, because node selection takes place inside a region, the number of nodes that must be considered is much smaller, which makes the problem easier to solve. Note that this two-step approach prioritizes client-to-replica latency over any other metrics that depend on node-specific factors, which is in line with the overall goal of latency-driven replica placement.

We discuss in detail the region-selection algorithm, called HotZone, in the following sections. The complementary algorithm for node selection within a region is currently under development. For evaluation purposes, we assume that it simply chooses the node with minimal average distance to all other nodes in the region.

### 3.1 Region Selection

HotZone places replicas in network regions based on the relative latencies between nodes. Essentially, it works similar to the HotSpot algorithm: it first identifies network regions, then orders the regions according to the load they generate, and finally places replicas one-by-one in subsequent regions starting from the most active region. Placing a replica inside a heavily-loaded network region seems to be attractive, as it should improve the access latency for a large number of requests.

Identifying network regions means determining groups of nodes whose latencies to each other are relatively low. In general, it would require analyzing all pair-wise latencies between nodes, which is likely to be computationally expensive in large-scale systems.

To reduce the computational overhead, we identify network regions based on node coordinates produced by GNP. Recall that GNP approximates the latency between two nodes with the distance between their corresponding coordinates in an  $M$ -dimensional Euclidean space. The main observation we make at this

point is that node coordinates are not uniformly distributed over the Euclidean space. Moreover, each cluster of node coordinates denotes a highly concentrated group of nodes whose latencies between each other are very low. It is therefore natural to identify network regions by determining clusters of node coordinates in the Euclidean space.

A question remains how to identify and measure coordinate clusters. To this end, we split the entire  $M$ -dimensional space into *cells* of identical size. A cell is an  $M$ -dimensional hypercube whose edge length is equal to some fixed value  $C$ . Each cell is uniquely defined by its *center point*. Because of the geometric properties of our space partition, the coordinates of each center point are

$$\left( C.k_1 + \frac{1}{2}C, \dots, C.k_M + \frac{1}{2}C \right)$$

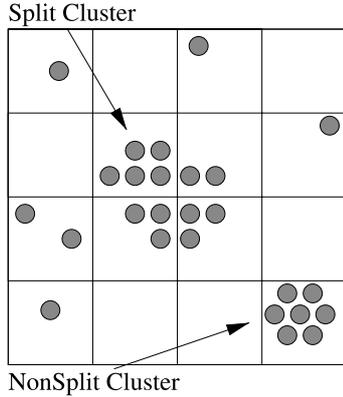
for some values of  $k_i$ , where  $i$  ranges from 1 to  $M$ . We identify each cell by means of the integer-valued vector  $(k_1, \dots, k_M)$  yielding the center point of that cell. The *density* of a cell is defined as the number of nodes whose coordinates fall within that cell. Note that this definition can easily be extended to support different node weights depending on the load generated by each node. To calculate the densities of all cells, the coordinates of each node  $(x_1, \dots, x_M)$  are mapped to their corresponding cells  $(k_1, \dots, k_M)$  according to the formula

$$k_i = \lfloor x_i/C \rfloor \quad i = 1, \dots, M$$

In this way,  $N$  nodes can be mapped into their corresponding cells in  $O(N)$  time.

In a straightforward approach to the cluster identification problem, we could treat each cell as a potential cluster, and place replicas in the most dense cells only. However, a problem that arises at this point is that clusters may span multiple cells. This happens when cell boundaries divide a coordinate cluster into several parts with each part falling into a different cell (see **Fig. 1**). A split cluster is less likely to be given a replica, since each of its parts can be too little to outweigh smaller, but undivided clusters. This could lead to suboptimal performance of our algorithm.

To alleviate the problem of split clusters, we introduce the notion of a *zone*. Each zone is uniquely defined by a cell, and consists of that cell plus all the immediate neighboring cells. Each zone contains therefore  $3^M$  cells in total. In other words, a zone is a group of adjacent cells, which together form a hypercube with



**Fig. 1** Split and non-split coordinate clusters in a 2-dimensional space.

edge length  $3C$ . Each zone shares its center point with the cell that defines that zone. We define the density of a given zone as the sum of the densities of all its member cells. Since zones do overlap, the density of a single cell contributes to the densities of several zones.

Operating on zones instead of cells does not completely solve the problem of split coordinate clusters, because coordinate clusters can still be large enough to be scattered over several disjoint zones. We return to this issue below, when discussing how to choose the cell edge length  $C$ .

When expressed in terms of zones, HotZone works similar to the Greedy algorithm. In every iteration, it identifies the most dense zone, marks this zone as a “replica holder,” and removes all the node coordinates in this zone so that they are not considered in the remaining iterations. In this way, we implicitly assign the removed nodes to the replica that is currently being placed. More importantly, however, we reduce the possibility that replica-holding zones overlap, as empty cells are unlikely to fall within a zone considered to be dense. This ensures that all the replica-holding zones together cover as many node coordinates as possible, and that the replicas are ultimately placed in different parts of the system.

### 3.2 Cell Size Choice

The notion of zone allows us to reduce the problem of identifying coordinate clusters to that of identifying cells that yield dense zones. Note that there is no one-to-one correspondence between zones and coordinate clusters. Depending on the cell edge length  $C$ , a zone can contain several coordinate clusters, or a coordinate cluster can be scattered over multiple disjoint zones. Therefore, selecting the value of  $C$

plays a key role for the efficiency of HotZone.

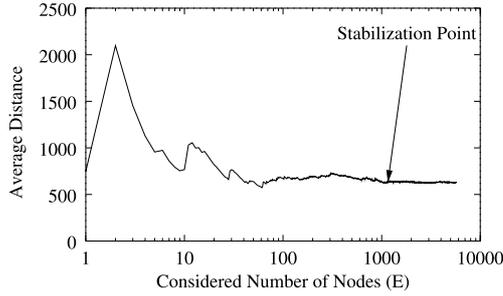
There are two factors on which the cell edge length  $C$  should intuitively depend. The first factor that influences it is the number of replicas  $K$ . Recall that HotZone effectively splits the entire space into  $K$  parts, and assigns each part to a different replica. The larger the value of  $K$ , the smaller the parts should be that are produced by the algorithm. Given how HotZone works, each such part should ideally be identified as a separate zone. The cell edge length  $C$  should therefore be inversely proportional to the number of replicas  $K$ , which is given to HotZone as a parameter.

The second factor that should affect the cell edge length  $C$  is the distribution of node coordinates in the space. Since we are using zones to identify coordinate clusters, it is natural that the cell size depends on the typical cluster size. For example, if most node coordinates fall in a small number of dense clusters, then these clusters can be identified with a small cell edge length  $C$ . On the other hand, if node coordinates are more evenly dispersed over the entire space, then the cell edge length  $C$  must be longer. This is because the number of nodes falling in a single zone must be large enough to ensure that all the zones can be unambiguously ranked according to their density. Only then can HotZone correctly determine the most dense zones where replicas should be placed.

Coordinate distribution can be represented in different manners. We decided to use the *average* distance between node coordinates, as it is easy to calculate. Note that calculating the average distance typically requires calculating the distances between the coordinates of each node pair. This would require  $O(N^2)$  operations, which we strive to avoid. In our case, however, it is enough to compute a good estimate of the average distance, as the positioning procedure itself is imperfect.

To obtain an estimate of the average distance, we calculate the average distance between an incrementally-growing number of node coordinates until the resulting value stabilizes. Determining the stabilization point is not trivial, as the computed value changes up until all the node coordinates have been considered. However, as it turns out, if we take node coordinates in a random order, then the incrementally-computed values quickly converge to the actual average distance.

More formally, we iteratively compute the av-



**Fig. 2** Incremental calculation of the average inter-node distance

average distance  $D_E$  between an incrementally-growing number of nodes  $E$ :

$$D_E = \frac{2}{E^2} \sum_{1 \leq i \leq E} \sum_{1 \leq j < i} \text{distance}(P(i), P(j))$$

where  $P(k)$  denotes the coordinates of  $k$ -th node in a randomly-ordered table. We determine the stabilization point by checking whether the value of  $D_E$  has stabilized over the last 100 iterations. To do so, every 10 iterations, we calculate the difference between the maximum and minimum value of  $D_E$  that has occurred within the last 100 iterations, and verify whether it is less than some threshold value  $\epsilon$ :

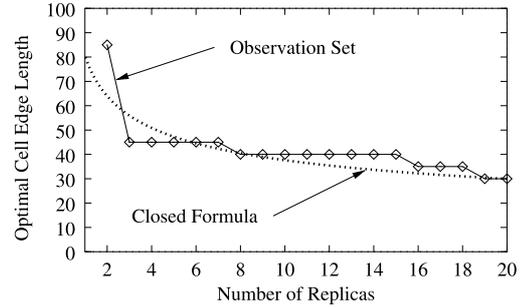
$$\max(D_{E-99}, \dots, D_E) - \min(D_{E-99}, \dots, D_E) < \epsilon$$

If the threshold is exceeded, then we increase  $E$  and proceed with the next 10 iterations. Otherwise, we treat  $D_E$  as our estimate of the average distance.

We verified this method on a sample data set that contained 5,728 node coordinates calculated in a 6-dimensional space. The positioned nodes were the Web clients that accessed our departmental Web server between the 1st and 30th of November 2003. The node coordinates were produced by SCoLE, which essentially runs a GNP instance in cooperation with a number of other hosts, acting as landmarks<sup>3</sup>). We configured SCoLE to cooperate with landmarks deployed on 12 different PlanetLab nodes<sup>13</sup>).

According to our experiments, the value of  $D_E$  converges after  $E = 1110$  iterations when estimating the average distance between 5,728 node coordinates with the  $\epsilon$  value set to 10 (see **Fig. 2**). As we show in Section 4, similar numbers of iterations are sufficient to calculate the average inter-node distance also for significantly larger sets of node coordinates.

In order to discover how the number of repli-



**Fig. 3** The  $(K, C)$  observation set and its approximation.

cas and the average inter-node distance contribute to the cell edge length  $C$ , we empirically determined the optimal values of  $C$  in a wide range of scenarios. We then used non-linear regression to determine a function which outputs a good value of  $C$  for any combination of  $K$  and  $D$ . Considering that  $C$  is expected to be inversely proportional to the number of replicas  $K$ , and proportional to the average distance  $D$ , we decided to investigate the following family of functions:

$$C = \alpha \cdot \frac{D}{K^\beta}$$

where  $\alpha$  and  $\beta$  are the coefficients that we need to determine.

To obtain the optimal values of  $C$ , we repetitively applied HotZone to the sample data set for each number of replicas  $2 \leq K \leq 20$ . We ignored the case when  $K = 1$ , as the best location is obviously the node whose average distance to all the other nodes is minimal. For each value of  $K$ , we repetitively ran HotZone with values of  $C$  ranging from 5 to  $D$ , taken in steps of 5. For each value of  $C$ , we evaluated the resulting placement by calculating the median distance between all nodes and their closest replicas. The  $C$  value yielding the shortest median distance was considered to be the best for the given number of replicas  $K$ . The outcome of this experiment was a set of observations, each being a pair of  $(K, \text{best value of } C)$ . This set is depicted in **Fig. 3**.

To obtain a closed function formula, we applied the non-linear regression algorithm implemented in Gnuplot to our set of observations. The resulting values of  $\alpha$  and  $\beta$  were 0.126 ( $\approx \frac{1}{8}$ ) and 0.329 ( $\approx \frac{1}{3}$ ), respectively, which gave us the following closed formula for the cell edge length  $C$ :

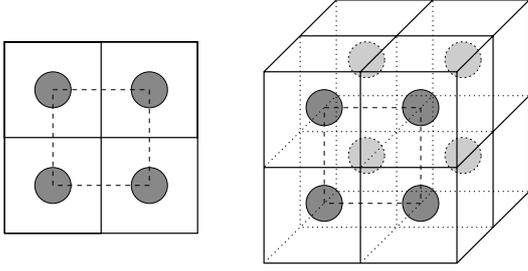


Fig. 4 Theoretical even replica distribution.

$$C \approx \frac{1}{8} \cdot \frac{D}{\sqrt[3]{K}}$$

where  $D$  is the average distance between nodes, and  $K$  is the number of replicas. The corresponding function is plotted in Fig. 3. Note that since we computed the set of observations based on a single data set, we used the set-specific value of the average inter-node distance  $D$  equal to 670. As we show below, however, the resulting closed function formula works also for other data sets, for which the value of  $D$  is completely different.

Interpreting the values of  $\alpha$  and  $\beta$  is difficult. As for the value of  $\alpha$ , we believe that it is determined by the uneven distribution of nodes over the space. The value of  $\beta$ , however, required a more detailed investigation. Before running the experiments, we expected  $\beta$  to be  $\frac{1}{6}$  (rather than  $\frac{1}{3}$ ), which would correspond to the space dimensionality. Our intuition was that, if the replicas are distributed more-or-less evenly over the space, then they will probably themselves form a regular structure similar to a hypercube. For example, if we evenly divided a square (2D) space among 4 replicas, then the replicas would form a square. Similarly, if we were placing 8 replicas in a cubic (3D) space, then the replicas would form a cube (see Fig. 4). In both cases, the zone edge length would be the space edge length divided by  $\sqrt[M]{K}$ , where  $K$  is the number of replicas, and  $M$  is the space dimension.

The reason why this intuition is wrong is that spaces produced by our GNP implementations have dimensions of very different “width.” By the width of the  $i$ -th dimension we mean the span of node coordinates along the  $i$ -th coordinate. Wide dimensions contribute to the actual distance between nodes, as they allow for large differences between node positions. Narrow dimensions, in turn, only slightly change node positions, but are important for the accuracy of latency estimation. However, because

HotZone groups nodes that have similar node coordinates, it ignores narrow dimensions and benefits only from wide ones. As it turns out, the space from which we derived the observation set contained only 3 broad dimensions with widths of 4,920, 3,840, and 1,855 milliseconds (and 3 narrow ones with widths of 320, 310, and 10 milliseconds). This may indicate why the value of  $\beta$  is approximately  $\frac{1}{3}$  instead of the expected  $\frac{1}{6}$ .

### 3.3 Complexity Analysis

In this section, we analyze the computational complexity of HotZone. Similar to the notation used in the previous sections, we use  $N$  to denote the number of nodes,  $K$  to denote the number of replicas, and  $M$  to denote the GNP space dimension.

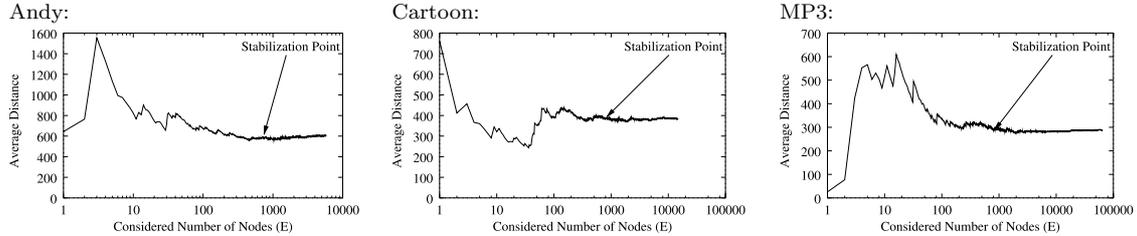
The computational cost of HotZone consists of three parts, each corresponding to a single step. The first step is to determine the average distance between nodes. As we show in Section 4, HotZone obtains a good estimate of this distance by calculating the distance between a fixed number of randomly-selected nodes. The cost of this operation is constant.

The second step is to construct the set of zones. HotZone first assigns nodes to their corresponding cells, which costs  $O(N)$  operations. Then, the set of non-empty cells is translated to the set of zones. To do that, HotZone identifies the neighboring cells of each cell, and sums their densities. Given that each zone contains a constant ( $3^M$ ) number of cells, and that the number of cells cannot exceed  $N$ , this operation costs  $O(N)$  cell-accesses. In our implementation, we sort all the cells according to their center points right after all the nodes have been assigned to their cells. We do so using Radix sort, which costs  $O(N \cdot M) = O(N)$  operations. Then, we access individual cells using binary search, which yields the cost of  $O(\log N)$  per a cell access. This results in the total cost of the second step being  $O(N \cdot \log N)$ .

The third step is to iteratively place replicas. For each replica, we identify the most dense zone, which requires inspecting all the zones. Since the number of zones cannot exceed  $N$ , the identification of the most dense zone costs  $O(N)$  operations in the worst case, as we process all the cells in our cell table directly, and without using binary search this time. Given that the same operations are performed for each replica, the total cost of the

**Table 1** Dataset statistics.

Dataset	Web Site Description	Unique Clients	Client Profile
Andy	Andrew Tanenbaum's home page	5,758	Universities all around the world
Cartoon	Looney Tunes™ fan site	14,682	US schools and broadband users
MP3	Dutch MP3 fan site	64,041	European broadband users

**Fig. 5** Incremental calculation of the average inter-node distance.**Table 2** Incremental calculation of the average inter-node distance – Statistics.

Dataset	Nodes Total	Nodes Used	$D_{Real}$	$D_{Calc}$	Calc. Time
Andy	5,758	740 (12.85%)	602	580	92 msec
Cartoon	14,682	650 (4.42%)	385	393	71 msec
MP3	64,041	820 (1.28%)	290	290	113 msec

third step is  $O(K \cdot N)$ .

A potential source of overhead can be the update of zones. Recall that, in every iteration, once a replica has been placed in the most dense zone, we have to prevent the nodes in that zone from being considered in the subsequent algorithm iterations. This means updating not only all the  $3^M$  cells that belong to the most dense zone, but also all the neighboring cells of these cells, as the density of zones yielded by the neighbors must be updated as well. This leads to the update of up to  $3^{2M}$  cells. Although this constant is large, it makes HotZone independent of the number of nodes. This makes HotZone particularly efficient in large systems. In our implementation, each update costs  $O(\log N)$  operations, as we access the cell in question using binary search. Still, however, the cost of placing a single replica is  $O(N)$ .

The total computational cost of HotZone is the sum of the costs of the above three steps:

$$\begin{aligned} &O(1) + O(N \cdot \log N) + O(K \cdot N) \\ &= O(N \cdot \max(\log N, K)) \end{aligned}$$

This cost is significantly lower than that of the previously proposed algorithms, such as greedy ( $O(K \cdot N^2)$ ) and HotSpot ( $O(N^2 + \min(N \cdot \log N + N \cdot K))$ ).

#### 4. Evaluation

We evaluate HotZone based on three data

sets produced by SCoLE, a GNP-like system that we operate using 12 landmarks deployed worldwide<sup>3</sup>). SCoLE positions Web clients in a 6-dimensional space based on latencies between these clients and a number of Web servers acting as landmarks. Latencies are measured while Web clients open HTTP connections to the Web servers in order to retrieve small images. We embedded references to these images in the Web documents constituting the three Web sites participating in our node-positioning experiment. In this way, SCoLE is able to position Web clients visiting three independent Web sites, which we treat as three different data sets (see **Table 1**). All the measurements used to produce these data sets were performed between the 1st of February and the 31st of March, 2004.

##### 4.1 Average Distance Calculation

Determining the right cell size is crucial to good behavior of HotZone. Since the cell size depends on the average distance between nodes, we first investigated how the incremental calculation method performs on our data sets. The results are presented in **Fig. 5** and **Table 2**.

As can be observed, irrespective of the size of the data set, a similar number of nodes is necessary to calculate a good estimate of the average distance. This observation confirms that this cost should be treated as a constant in the evaluation of the computational complexity of Hot-

**Table 3** The values of the  $\alpha$  and  $\beta$  coefficients.

Dataset	$\alpha$ value	$\beta$ value
Sample	0.126	0.329
Andy	0.154	0.310
Cartoon	0.363	0.651
MP3	0.130	0.373

Zone. Furthermore, this constant time is very small compared to the overall time of replica placement computation (about 100 milliseconds vs. several seconds).

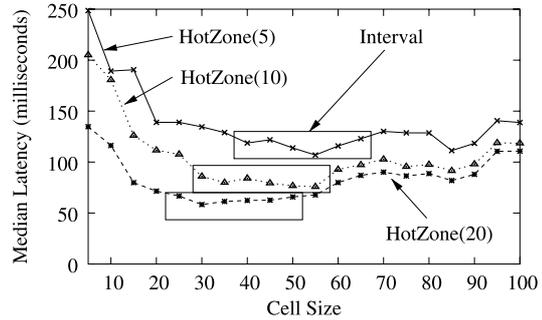
#### 4.2 Closed Formula Verification

HotZone calculates the cell size using a formula that depends on two parameters,  $\alpha$  and  $\beta$ . Recall that in Section 3, we determined  $\alpha$  and  $\beta$  by applying non-linear regression to a set of optimal cell sizes, which we computed based on a single sample data set. In this experiment, we verify how general these  $\alpha$  and  $\beta$  values are by computing them for each of the other three data sets. The results are presented in **Table 3**.

As can be observed, the values of  $\alpha$  and  $\beta$  computed for the Andy and MP3 data sets are very similar to their counterparts derived from the sample data set. However, it is clearly not so for the Cartoon data set, where the values of  $\alpha$  and  $\beta$  are significantly different. We believe that this is because of the fact that nodes in the Cartoon data set are more evenly distributed over the space. In that case, clients barely form any clusters, which leads to an optimal cell size significantly larger than expected. However, as we show in Section 4.3, even sub-optimal  $\alpha$  and  $\beta$  parameters lead to placements almost as good as those produced by Greedy.

According to our calculations, using the data set-specific parameters to compute placements for the Cartoon data set would improve the placement quality by at most 10%. This could indicate that the quality of placements is quite resilient to changes in the values of  $\alpha$  and  $\beta$ , or even in the value of  $C$  itself.

To verify this claim, we checked by how much imperfect values of  $C$  influence the quality of placements. We again analysed the results of the experiment where we repetitively placed replicas using all possible  $C$  values between 5 and 100. Previously, we only used the  $C$  value yielding the minimal median latency. This time, however, we also checked how fast this median latency increases as the  $C$  value drifts away from the optimal one. **Figure 6** shows median latencies computed for different  $C$  values. For sake of clearance, we only show the

**Fig. 6** Changes in placement quality vs.  $C$  values.

results for placing 5, 10, and 20 replicas based on the Andy data set. Other data sets exhibit similar behavior.

As can be observed, the optimal  $C$  value belongs to a relatively-long interval, where each value yields a median latency within 10% of the minimal one. According to our experiments, the  $C$  values calculated by HotZone belong to that interval. This explains why HotZone produces nearly-optimal placements even though the  $C$  values themselves may be imperfect.

#### 4.3 Placement Quality Comparison

A popular method of evaluating replica placements is calculation of the *average* latency between nodes and their closest replicas. However, the results produced by this method very much depend on outliers, which are nodes whose latencies to any other nodes are very high. Outliers typically use slow network connections, such as modems, and do not tend to form clusters. Because of these two factors, it is hard to improve the replica-access latencies of outliers without placing replicas specifically on them. We therefore decided to concentrate on the remaining nodes by evaluating placements with the *median* latency between nodes and their closest replicas.

We evaluated HotZone against the Greedy and HotSpot algorithms described in Section 2. Recall that HotSpot needs to know how to determine whether two given nodes are neighbors or not. We configured HotSpot to consider nodes to be neighbors only if the distance between them is less than some threshold value. To ensure the fairness of comparison, we tried various threshold values and report only the best result.

We also considered a variant of HotSpot, which we call “HotClear.” After placing a replica on a given node, HotClear removes all the nodes from the neighborhood of this node so

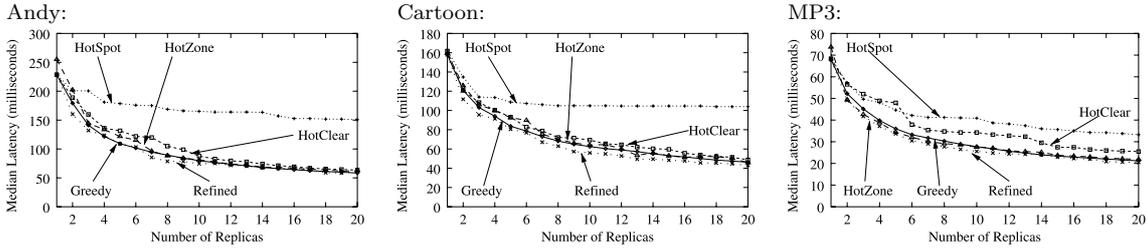


Fig. 7 Placement quality comparison.

that they are not considered in the subsequent iterations. Also in this case, we tried several threshold values and report only the best result.

Ideally, we should also compare HotSpot against placements yielding optimal median node-to-replica distance. However, since computing optimal placements is unfeasible, we decided to approximate them using Simplex-downhill, which is a multi-dimensional optimization algorithm<sup>14</sup>). When applied to the placements produced by HotZone, Simplex-downhill investigates similar replica locations, and tries to move replicas so that the median node-to-replica latency is reduced. We refer to this method as “Refined.” Simplex-downhill is computationally expensive, so it is unlikely that Refined can be used in practical applications. Nevertheless, the comparison between HotZone and Refined enables us to estimate by how much the original placement can be improved.

We apply the five evaluated algorithms to each of the three data sets. We iteratively place  $K$  replicas, for  $K$  between 1 and 20, and calculate the median node-to-replica latency for each value of  $K$ . The results are presented in Fig. 7.

As can be observed, the original HotSpot algorithm performs significantly worse than all the others. This is not surprising: as it turns out, HotSpot places replicas on nodes whose coordinates are close to the centers of a few very active neighborhoods. Although these neighborhoods may change depending on the threshold value, the replicas are ultimately placed close to each other, which results in poor performance. This is exactly the effect that HotZone tries to avoid by using overlapping network regions.

The remaining four algorithms produce comparable results. Compared with Greedy, HotZone offers median latency that remains within 13%, 14% and 9% of that offered by Greedy for

the Andy, Cartoon, and MP3 data set, respectively. The average difference between latencies offered by these two algorithms, however, is between 3% and 5% in all the data sets. Note that HotZone sometimes slightly outperforms Greedy, which can be seen in the graph depicting the replica placement based on the MP3 data set.

HotZone usually performs better than HotClear. In this case, however, the average difference in latencies is very small (between 5% and 7%) for the two smaller data sets, but it increases to 16% for the biggest data set. The case of HotClear shows that the original HotSpot algorithm can easily be improved to work effectively on node coordinate sets, but even then it still cannot outperform HotZone.

In comparison with Refined, HotZone obviously performs worse. However, the difference in the offered median latency is not large and on average equals 8%, 12% and 5% (in the respective data sets). This may indicate that the placements produced by HotZone cannot be improved much, especially that they are already comparable in terms of quality to those produced by Greedy.

#### 4.4 Placement Computation Times

The main advantage of HotZone over other algorithms is its low computational cost. In this experiment, we show how the differences in computational complexities of different algorithms translate into placement computation times. We measured the execution times of the five evaluated algorithms for each number of replicas  $K$  between 1 and 20, based on all the three data sets. The tests were performed on an idle PC equipped with a Pentium III 1 GHz. The results are depicted in Fig. 8 (note the logarithmic time scale).

As can be observed, the time needed by HotZone to compute its placements up to 3 orders of magnitude lower than the time needed by Greedy. As for Refined, the unpredictable na-

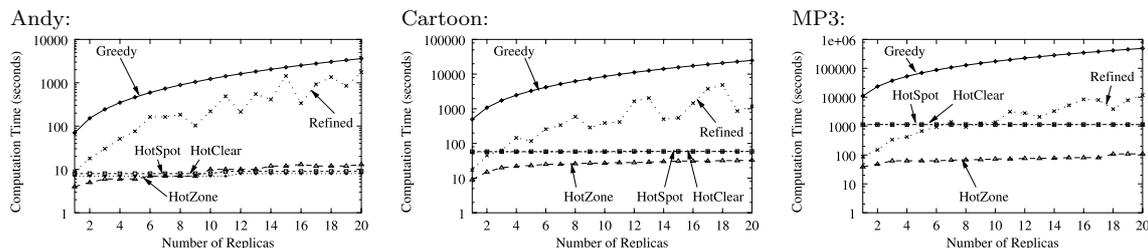


Fig. 8 Placement computation times.

ture of Simplex-downhill resulted in irregular execution times. Still, it is interesting to see that Refined is consistently faster than Greedy, even though Refined nearly always produces better results than Greedy.

In comparison with HotSpot and HotClear, the computation time of HotZone is better only for the two larger data sets, and the advantage of HotZone over these two algorithms increases with the number of nodes in a data set. For the smallest data set, however, the computation time of HotZone is comparable to these of HotSpot and HotClear. This indicates that HotZone is particularly suitable for large-scale systems, where the number of nodes is very high.

## 5. Conclusion

We have presented HotZone, a novel replica placement algorithm that optimizes node-to-replica latency based on node coordinates produced by SCoLE. Similar to the previously proposed HotSpot algorithm, HotZone places replicas on nodes that along with their neighboring nodes generate the highest load. In contrast to HotSpot, however, HotZone does not require that  $O(N^2)$  operations are performed to determine the neighborhood composition for all the  $N$  nodes in a given system. Instead, it exploits the properties of node coordinates to determine the composition of all neighborhoods faster. We have demonstrated that the computational cost of placing  $K$  replicas using HotZone is  $O(N \cdot \max(\log N, K))$ , which is significantly lower than that of any previously proposed algorithm. This makes HotZone attractive for use in large-scale distributed systems.

HotZone produces results of comparable quality to those of Greedy. Furthermore, for large data sets, the computation time of HotZone is significantly lower than those of the other evaluated algorithms. In particular, it is up to 3 orders of magnitude lower than the com-

putation time of Greedy.

We plan to use HotZone in Globule, a peer-to-peer Content Delivery Network that our group is developing<sup>15</sup>). We believe that it will help us efficiently place replicas among a large number of Globule nodes.

**Acknowledgments** We would like to thank Andrew Tanenbaum, Jennie Zhang, and Johan Pouwelse for enabling us to deploy our client positioning software on their Web sites.

## References

- 1) Dille, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R. and Wehl, B.: Globally Distributed Content Delivery, *IEEE Internet Computing*, Vol.6, No.5 (Sep. 2002).
- 2) Karlsson, M., Karamanolis, C. and Mahalingam, M.: A Framework for Evaluating Replica Placement Algorithms, Technical report, HP Laboratories, Palo Alto, CA (2002).
- 3) Szymaniak, M., Pierre, G. and van Steen, M.: Scalable Cooperative Latency Estimation, *10th International Conference on Parallel and Distributed Systems* (July 2004).
- 4) Radoslavov, P., Govindan, R. and Estrin, D.: Topology-Informed Internet Replica Placement, *6th Web Caching Workshop* (June 2001).
- 5) Jamin, S., Jin, C., Kurc, A.R., Raz, D. and Shavitt, Y.: Constrained Mirror Placement on the Internet, *20th IEEE INFOCOM Conference* (Apr. 2001).
- 6) Paxson, V.: Measurements and Analysis of End-to-End Internet Dynamics. Technical Report UCB/CSD-97-945, University of California at Berkeley (Apr. 1997).
- 7) Huffaker, B., Fomenkov, M., Plummer, D.J., Moore, D. and Claffy, K.: Distance Metrics in the Internet, *International Telecommunications Symposium* (Sep. 2002).
- 8) Karlsson, M. and Karamanolis, C.: Choosing Replica Placement Heuristics for Wide-Area Systems, *24th International Conference on Distributed Computing Systems* (Mar. 2004).
- 9) Qiu, L., Padmanabhan, V.N. and Voelker, G.M.: On the Placement of Web Server Repli-

- cas, *20th IEEE INFOCOM Conference* (Apr. 2001).
- 10) Krishnamurthy, B. and Wang, J.: On Network-Aware Clustering of Web Clients. *ACM SIGCOMM* (Aug. 2000).
  - 11) Eugene Ng, T.S. and Zhang, H.: Predicting Internet Network Distance with Coordinates-Based Approaches, *21st IEEE INFOCOM Conference* (June 2002).
  - 12) Amini, L. and Schulzrinne, H.: Client Clustering for Traffic and Location Estimation, *24th International Conference on Distributed Computing Systems* (Mar. 2004).
  - 13) The PlanetLab Project.  
<http://www.planet-lab.org/>
  - 14) Nelder, J.A. and Mead, R.: A Simplex Method for Function Minimization. *The Computer Journal*, Vol.4, No.7 (1965).
  - 15) Pierre, G. and van Steen, M.: Design and Implementation of a User-Centered Content Delivery Network, *The 3rd IEEE Workshop on Internet Applications* (June 2003).

(Received February 22, 2006)

(Accepted July 4, 2006)

(Online version of this article can be found in the IPSJ Digital Courier, Vol.2, pp.000–000.)



**Michal Szymaniak** is a Ph.D. candidate in the Computer Systems group at the Vrije Universiteit Amsterdam. His research focuses on large-scale distributed systems for content delivery in the Internet. He is a student member of the IEEE and the ACM. Szymaniak holds a double MSc in Computer Science from Warsaw University (Poland) and from the Vrije Universiteit Amsterdam.



**Guillaume Pierre** is an assistant professor in the Computer Systems group at the Vrije Universiteit Amsterdam. He has been working in the field of Web-based systems for many years. He is a member of the IEEE and an editorial board member of IEEE DSONline. Pierre holds an M.Sc. and a Ph.D. in Computer Science from the University of d'Evry-val d'Essonne (France).



**Maarten van Steen** is a full professor in the Computer Systems group at the Vrije Universiteit Amsterdam. He is head of a research team developing large-scale distributed systems. Besides Web-based systems, his research interests include peer-to-peer and gossip-based distributed systems. He is senior member of the IEEE and member of the ACM. Van Steen holds an M.Sc. in Applied Mathematics from Twente University and a Ph.D. in Computer Science from Leiden University.