Implementing and Evaluating Network Positioning in the Resource Addressable Network

Gao Yuanyuan

School of Computer Science, McGill University
Montréal, Québec, Canada

January, 2006

A thesis submitted to
McGill University in partial fulfilment of
the requirements of the degree of
Master of Science

Your file Votre référence
ISBN: 978-0-494-24673-3
Our file Notre référence
ISBN: 978-0-494-24673-3

Abstract

This thesis focuses on the design, implementation and evaluation of a coordinates-based mechanism to compute network positions of Internet hosts in the resource addressable network. This mechanism is based on absolute coordinates calculated from modelling the Internet as a D-dimensional Cartesian space. A small distributed set of infrastructure hosts called landmarks is chosen to first compute their coordinates and serves as a frame of reference for other hosts. I study two algorithms for landmark positioning: distributed spring algorithm and spring equilibrium algorithm. The position of the non-landmark host is calculated using a centralized algorithm. I also evaluate a schema to adjust a host's position based on the positions of its nearby peers.

A location-based RAN prototype is implemented by integrating network positioning with naming and discovery modules. The prototype distribution is installed on 127 PlanetLab machines for evaluation. By performing real-time experiments on the distributed testbed, I show that the spring equilibrium algorithm outperforms the distributed spring algorithm in the aspects of efficiency and stability. Furthermore, the adjustment schema based on nearby peers is beneficial to stablize host's position under normal network variance.

Sommaire

Ce mémoire porte sur la conception, l'implémentation et l'évaluation d'un mécanisme faisant appel à un système de coordonnées pour déterminer la position des ordinateurs hôtes sur un réseau de type RAN (Resource Addressable Network). Ce mécanisme est basé sur des coordonnées absolues, calculées à partir d'une modélisation de l'Internet en tant qu'espace cartésien D-dimensionnel. Un ensemble restreint et distribué d'ordinateurs hôtes, appelés points de repères, est choisi et les coordonnées de ceux-ci sont calculées afin de servir de cadre de référence pour les autres ordinateurs hôtes. J'ai étudié deux algorithmes pour le positionnement des points de repères: l'algorithme ressort de distribution et l'algorithme ressort à l'équilibre. La position de tout ordinateur hôte, autre qu'un point de repères, est calculée en utilisant un algorithme centralisé. J'ai également évalué un schéma selon lequel la position d'un ordinateur hôte est ajustée en se basant sur la position de ces voisins immédiats.

Un prototype de réseau RAN a été implanté en intégrant le positionnement réseau avec des modules de nommage et de découverte. Le prototype de réseau a été installé sur 127 ordinateurs au PlanetLab pour en faire l'évaluation. En effectuant des expériences en temps réel sur un banc d'essai, je démontre que l'algorithme ressort à l'équilibre surpasse l'algorithme ressort de distribution en terme d'efficacité et de stabilité. De plus, le schéma d'ajustement basé sur les ordinateurs voisins est bénéfique pour stabiliser la position de l'ordinateur hôte sous des conditions observées de variations normales de réseau.

Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. First, I want to thank my thesis supervisor Professor Muthucumaru Maheswaran for his guidance, advice, encouragement and financial support throughout the research and the preparation of this thesis. I have benefited enormously from his valuable insights and tutoring on a great number of occasions.

Many thanks to my colleagues in ANRL who have provided consultation and comments on my research. My colleague of the RAN project Balasubramaneyam Maniymaran kindly answered many of my questions with extreme patience and made countless suggestion for improving the accuracy and quality of this thesis. Arindam Mitra, our system administrator, was always available to help with system problems, which made my life much easier. I would also like to thank the other members in our lab: Asad, Grant, Luke, and Samuel. You gave me the feeling of being at home while at work.

I wish to thank the School of Computer Science for the graduate courses and the research environment. Thanks to Diti Anastasopoulos, Vicki Keirl, Lise Minogue, and Lucy St-James, for easing the official procedures.

Last but not least, I owe special thanks to my loving fiance Yaokang Li, for his unconditional and unwavering support, understanding and sacrifice during my study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

This thesis describes the building of a network positioning service and its integration with the prototype design of a *public computing utility* called "Galaxy," now being developed at the Advanced Networking Research Lab at McGill University. To explain the motivation of my research topic, firstly, I introduce the Galaxy project in Section 1.1, focusing on the major research issues and the prototype architecture. Secondly, in Section 1.2, I analyze the design issues of a network positioning service, including general characteristics and specific properties studied by the Galaxy research. Lastly, in Section 1.3, I describe the contribution of this thesis to the Galaxy project.

1.1 Overview of the Galaxy Project

The Galaxy project [1] is concerned with studying issues related to *public computing utilities*. It proposes a novel way to organize and distribute computational services, ensuring both scalability and high *quality of service* (QoS).

Utility computing [2, 3, 4, 5, 6] is a service provisioning model, much like *Grid computing* [7, 8], to maximize the use of a large number of resources from multiple sites, while minimizing the associated costs. The difference between utility computing and grid computing lies on that the utility computing facilitates the *commoditization* of its services such as storage, processing power, applications and security. The word commoditization refers to the activity that provider makes computing resources available to customers as needed, and charges them for specific usage. This process makes computing a true utility, like electricity and water. To construct such computing utilities, it's necessary to virtualize computing resources according to unified standards to make provisioning easier [2]. In Galaxy, the process that changes traditional computing resources to metered services is done by classifying the computing resources into virtual resources types that provide predefined sets of services.

The idea of utility computing first emerged as a business solution for big companies to organize their computational power [9, 10]. In this context, the scalability of the computing utility is naturally limited. Inspired by the success of *peer-to-peer* (P2P) file sharing systems [11, 12, 3, 13], the Galaxy project extends the research scope to both public and dedicated resources. Public resources refer to the network resources hosted in a public network like the Internet.

This extended research topic is named *public computing utility* (PCU). PCU is an open system to encourage the contribution of public resources. One of the biggest challenges of such open systems is to ensure a predictable quality of service, which is a very crucial feature to achieve direct business goals. To obtain the QoS as needed, solely depending on public resources is not practical. When public resources cannot meet the expected performance, the approach of Galaxy is to supplement the resultant deficiency with dedicated resources to meet the expectation of the resource user.

The prototype of Galaxy PCU fully reflects the design objectives described above. It is a middleware that functions between physical resources and user applications. It adopts a layered architecture that consists of a resource virtualization system, a management infrastructure and a stack of services as shown in Figure 1.1.

The lowest layer of the architecture is a P2P overlay called *resource addressable network* (RAN) [14]. The RAN provides uniform resource naming and discovery services [15, 16] to the upper layer, the *galaxy resource management system* (GRMS). GRMS manages the behaviors of resources, such as trust [17] and QoS. The next upper layer is the Galaxy services. Example services include application-level QoS management and shell

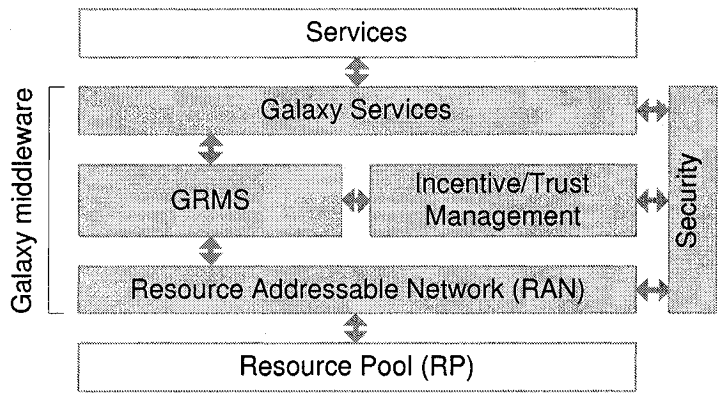


Figure 1.1: Galaxy architecture

interfaces. Other than these major functional layers, a security layer plays a monitoring role to protect these three layers from malicious activities through the Internet.

Galaxy project aims at designing a resilient, flexible, scalable, and QoS-aware provisioning mechanism of computer-based services, especially large-scale services. This paper focuses on implementing a network positioning service, which is part of the prototype design of the RAN. In the next section, I analyze the design issues related to the network positioning service.

1.2 System Design Issues for Network Positioning

Network positioning is concerned with representing the network distance relationships among network hosts. It can be very useful for wide-area network applications [18, 19, 20, 21, 22]. For instance, large scale interactive online games, such as *massively multi-player online games* (MMOG) can use network positions to direct the game client to connect to the closest game server to reduce real-time delay [23]. Some characteristics for network positioning are discussed below. These features shed light on the objectives of constructing a network positioning service.

- **Accuracy**: network positioning explores algorithms to compute network positions based on limited amount of real network measurements. Accuracy refers to the quantitative measure of the magnitude of error in representing the distance relationships using these positions. It is a crucial metric to judge the viability of a network positioning service.

- **Adaptivity:** the network environment is highly dynamic. The network positioning service needs to update hosts' positions to reflect the intrinsic network distance changes caused by network topology change.
- **Stability:** the positions in the adaptive system may fluctuate even when there is no change in the network topology change. This problem can cause unnecessary drifting of the positions, even affecting the positioning accuracy. The stability of a network positioning refers to properly reacting to topology changes and reducing unnecessary fluctuations.

In addition to these general issues, some properties are considered to be necessary for integrating the network positioning service with the Galaxy system:

- **Decentralized Design:** Galaxy nodes are organized in P2P manner. Public resources may join and leave frequently. It is not practical to have a central authority to assign positions to those resources every time they join. The ideal solution is to let resources compute and update their own positions according to a systematic mechanism.
- **Efficiency:** Galaxy is a middleware. It can be integrated with many applications, especially utility-like applications. The building and running of Galaxy services should not add much overhead to the application systems. It is also essential to deliver the services in a timely fashion.

1.3 Thesis Contribution

Galaxy uses two metrics to classify resources: resource types and location. The RAN generates *profile-based names* and *location-based names* for resources. Profile-based name is created by profiling resource according to different resource types. Location-based name is generated based on resource's geographic location. Resources are organized into *type rings* and *neighborhood rings*. The order of resources on the ring is decided by applying an indexing method to the profile-based names and location-based names respectively. RAN discovery mechanism makes use of these ring structures to search for resources that meet specific conditions of resource types and/or location. A detailed explanation is given in Chapter 3.

This thesis contributes to the research of the RAN in two aspects. First, this thesis is concerned with the design of a network positioning service, which is an indispensable component for generating location-based names in the RAN. It provides viable solution to represent distance relationship among hosts such that RAN discovery mechanism can build on top of it to realize location-based resource discovery. Second, a prototype location-based RAN is implemented and its positioning functionality is evaluated on 127 *PlanetLab* [24] nodes. It consists of machines spanning over 25 countries. Experiments are conducted to evaluate different metrics concerning the network positioning service and the performance of different positioning algorithms.

Chapter 2

Background on Network Positioning

Large distributed applications, such as a PCU like Galaxy and P2P file sharing systems like Napster and Gnutella [25], have much flexibility in choosing the peers for service. For example, in a PCU, a job allocation program wants to know the available latency between itself and all the peers that have the wanted resources. Although these network performance characteristics can be accurately measured on-demand, for a system with large number of nodes spanning a wide area, this process becomes very time-consuming, generating huge number of probing messages . The ideal solution is to predict network distance in an accurate, scalable and timely fashion and use the prediction as a metric to minimize the need for network measurements.

Basically there are two kinds of approaches in predicting network distance. The first kind proposed in IDMaps [26], models the Internet as a simple topology. The network distance information is represented using individual path distances. The second kind, called *coordinates-based approach,* was first proposed in GNP [27]. Figure 2.1 illustrates the basic idea of coordinates-based approach. It models the Internet as a Cartesian space. The network distance information is represented using coordinates. Compared with the first approach, coordinates-based approach have several advantages. First of all, a distance prediction in the Cartesian space is simply an evaluation of the distance function which is both straight-forward to implement and fast to compute compared to a shortest path search in the topology model. Secondly, the distances of all paths between K hosts can be represented by K sets of coordinates of size D each (i.e. $O(K \cdot D)$ of data), where D is the dimension of the Cartesian space, as opposed to $K(K-1)/2$ inter-host distances (i.e. $O(K^2)$ of data). Thirdly, host coordinates have relatively fixed local properties that can be exchanged easily among hosts when they discover each other, allowing network distance predictions to be computed in a timely fashion. For these advantages, I consider the coordinates-based approach to be a better choice in devising my positioning methods.

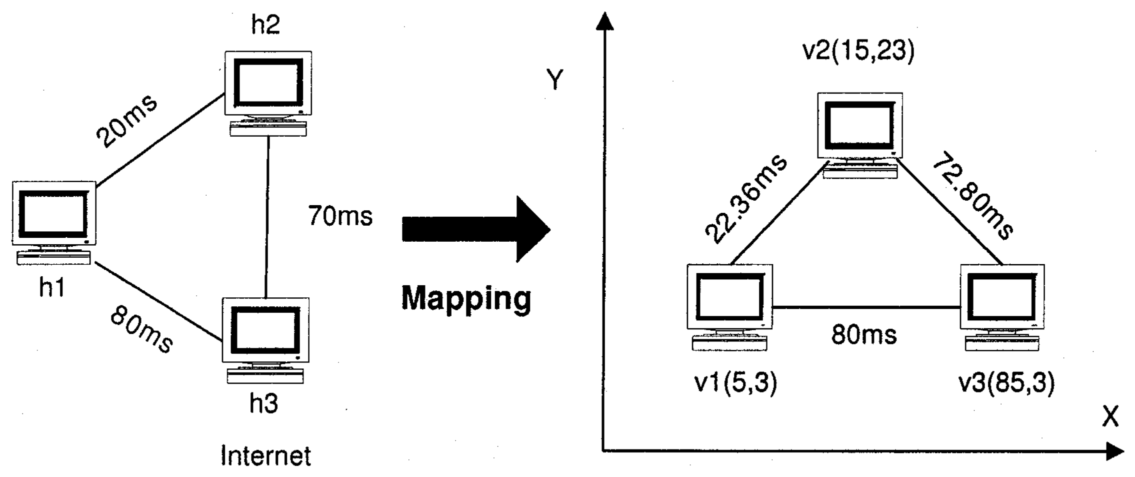


Figure 2.1: Coordinate-based Network Positioning

In the following sections, I introduce a few research projects studying the coordinates-based approach. These projects propose different mechanisms to generate node coordinates. Analyzing their pros and cons will lead to a clear and proper solution to my design.

2.1 Pros and Cons of Existing Solutions

2.1.1 Global Network Positioning (GNP)

GNP [27] is the first network positioning method that uses the coordinates-based approach. It models the Internet as a D-dimensional Euclidean space and represents hosts as points in this space. Their positioning strategy uses a small distributed set of hosts called *landmarks* to serve as a frame of reference. As shown in Figure 2.2, at the system initialization, land-

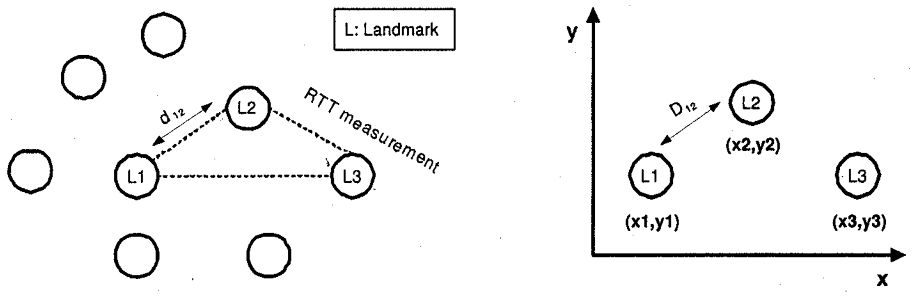


Figure 2.2: Landmark-aided Positioning

marks L_1, L_2 and L_3 first compute their coordinates in the Euclidean space. Any other host that wants to participate computes its own coordinates relative to these landmarks. This type of positioning is named as *landmark-aided positioning* (LAP). The computation of coordinates is done by minimizing the error between measured network latencies and predicted distances computed using the distance function in the Euclidean space. To formulate the *objective function*, let $\mathbf{C}_H$ denote the coordinates of host H, and $d_{HL_i}, i = 1, 2, \ldots, N$ denote the measured distance between host H and landmark L_i, where N is the total number of landmarks. The predicted distance between host H and landmark L_i is denoted as $\hat{d}_{HL_i}$. The coordinates of ordinary host H is computed by minimizing the following objective function:

$$f(\mathbf{C}_H) = \sum_{i=i}^{N} \varepsilon(d_{HL_i}, \hat{d}_{HL_i}) \tag{2.1}$$

$$\hat{d}_{HL_i} = |\mathbf{C}_H - \mathbf{C}_{L_i}| \tag{2.2}$$

where $\varepsilon()$ is an *error measurement function* defined as:

$$\varepsilon(d_{H_1H_2}, \hat{d}_{H_1H_2}) = (d_{H_1H_2} - \hat{d}_{H_1H_2})^2 \tag{2.3}$$

The computation of landmark coordinates is slightly different from that of an ordinary host. The goal of landmark positioning is to find a set of coordinates, $\mathbf{C}_{L_1}, \ldots, \mathbf{C}_{L_N}$, for the N landmarks such that the overall prediction error among landmarks is minimized. Formular 2.4 defines the objective function to be minimized to obtain the coordinates of the landmarks:

$$f(\mathbf{C}_{L_1}, \ldots, \mathbf{C}_{L_N}) = \sum_{i=1}^{N} \sum_{j=1}^{N} \varepsilon(d_{L_iL_j}, \hat{d}_{L_iL_j}) \tag{2.4}$$

With the above functions, the computation of coordinates can be formulated as a multi-dimensional global minimization problem that can be approximately solved by *simplex downhill method* [28].

GNP follows a peer-to-peer architecture such that its positioning service can be integrated with most of the existing P2P applications. The information of a host's coordinates can be piggy-backed to application messages to disseminate. Moreover, GNP does not add much overhead to set up and maintain (it requires network measurements only to a small set of nodes at the system initialization step, and during periodical updates).

There are some key issues that affects GNP's performance. First of all, how to choose the locations and the number of landmarks remains an open question. The authors propose and evaluate three heuristics to choose the locations of landmarks: *N-cluster-medians*, *N-medians* and *Maximum separation*. All of them are used to choose well-distributed infrastructure nodes. Their experiments also showed that the accuracy of GNP only improved from 6 to 9 landmarks. Furthermore, the computation of landmark coordinates is processed centralized. This schema increases the risk of single-point failure and security problems.

2.1.2 Network Positioning System (NPS)

NPS [29] is a version of GNP which addresses the system building issues involved in deploying a coordinate system. NPS includes a hierarchical reference system for reducing the load on the landmark nodes. The ordinary hosts that derive their coordinates from the initial set of landmarks can be selected also as reference points. NPS distributes the computation of landmark positions to all the landmarks, such that the single-point failure problem is avoided. Other design issues include a congestion control mechanism and a work-around for NATs. The NPS implementation is tested on 127 PlanetLab nodes using system parameters such as accuracy and convergence time for positioning.

The construction of NPS provides practical experience in building the network positioning system. The design issues discussed in this paper and their experimental results are useful guidelines for the design and testing of my network positioning service.

2.1.3 Vivaldi: Pracical Distributed Network Coordinates

Vivaldi[30, 31] is a fully distributed network positioning algorithm. It proposes a novel way to compute node coordinates: it models the network as a collection of springs, each of which pulls on the coordinates of a pair of nodes. The initial length of the spring is set to the measured network latency, and the current length of the spring is considered to be the distance between nodes in the coordinate space. The energy of the spring is proportional to the displacement from its rest length squared. So minimizing the sum of energy over all the springs is identical to minimizing the prediction error in the coordinate system. Node adjusts its coordinates by simulating the movement under the spring force. Coordinates become more accurate and stable with each successive adjustment. This process stops when a local minimization has reached. Chapter 3 explains this algorithm in detail.

Some important issues addressed by Vivaldi are: convergence time, accuracy, probing traffic, and responsiveness to network changes. The length of the step by which a node adjusts itself to the next optimal position affects the convergence time and accuracy. Big steps may result in oscillation or even failure to converge, while small steps cost unnecessary time. In Vivaldi, an gradually decreasing step size is used. It enforces big movement at the beginning of the algorithm to help nodes move to reasonable positions quickly and tiny steps at the end to avoid oscillation.

Vivaldi data can be piggy-backed on the packets of distributed applications to reduce probing traffic. Nodes in Vivaldi recompute their coordinates when new nodes come in. The advantage of Vivaldi is that it is fully decentralized. Without the construction of reference infrastructure as in GNP, Vivaldi is much simpler to implement.

2.1.4 Practical Internet Coordinates

In this paper [32], the multi-dimensional optimization method is adopted to compute host's coordinates. It is different from GNP in that the landmark framework is eliminated. Every node that has computed its coordinates can become the reference node for others. The authors mainly study different strategies to choose reference nodes for a given host. Three methods are discussed: *random selection*, *closest selection* and a *hybrid* of the two. Simulation results reveal that the hybrid selection outperforms the other two methods. It is advantageous by positioning a new node based on both distant and close references. To pick the closest elements, a node first computes its coordinates only based on random references. With the computed coordinates, it estimates the distances to a few references, selects the closest ones and recomputes its coordinates based on newly selected references.

PIC is a possible extension of GNP. The major difference is that it does not rely on infrastructure nodes and spreads load evenly over all nodes in the system. A heuristic approach for rating the references is included in PIC to protect its performance from malicious nodes.

2.1.5 PCoord: Network Position Estimation

PCoord [33] uses the simplex downhill method to compute the coordinates. It also eliminates the landmark framework to operate in a fully decentralized fashion. It differs from PIC in the strategies of selecting reference nodes. The three strategies it proposes are: *RandPCoord*, *ClusterPCoord* and *ActivePCoord*.

RandPCoord is random selection, similar to the one proposed by PIC. ClusterPCoord is based on *cluster* information (cluster refers to a set of nodes located in a bounded geographic area). Every node stores a database containing the positions of the nodes it has interacted with. A newly joining node contacts with some existing nodes to collect this position information, divide them into clusters, and then pick up nodes from each clusters to obtain well-distributed reference points. Well-distributed references can maximize the useful information provided in the reference frame. ClusterPCoord requires extra storage space for each node to store topology information. Moreover the position data needs periodically updating to adapt to network changes. ActivePCoord selects well-distributed references based on triangulated distances to other peers. It is assumed that each node initially knows of some other peers in the overlay. Nodes discover other peers by exchanging the list of peers they know. The communication cost of ActivePCoord is a major problem.

The simulation shows, within a network system of 3400 nodes, 93% of the peers can predict their nearest nodes by probing around 3.7% of the total population. But the authors did not give any hint about how this communication cost increases as the system size grows.

Above I have described five coordinates-based network positioning systems. Considering the computational methods, there are two dominant methods for computing network positions: (a) multi-dimensional optimization method, such as the simplex downhill method; and (b) heuristics such as the spring algorithm used in Vivaldi. Compared with the simplex downhill method, the spring algorithm has lower computation complexity and is simpler to implement without sacrificing accuracy. Considering the strategies of selecting reference nodes, these positioning systems also divide into two categories: (a) use a fixed set of reference nodes, as landmark-aided positioning proposed in GNP; and (b) dynamically choose reference nodes through some heuristics. Fixed set of references ensures a more stable system performance, but may cause system bottleneck when a large amount of nodes compute coordinates at the same time. Dynamic selection of references spreads load evenly over all nodes in the system. By applying the heuristic metrics such as cluster information or triangulated distance, dynamic reference selection can adapt to the real topological distribution of the nodes. A summary table is shown below.

Table 2.1: Properties of Network Positioning Systems

	GNP	Vivaldi	PIC	PCoord
infrastructure node	landmark	none	none	none
algorithm	simplex downhill	spring algorithm	simplex downhill	simplex downhill
selection of reference nodes	well-distributed	random	hybrid of random and close heuristics	cluster or triangle inequality heuristics
service bottle neck	yes	no	no	no

Chapter 3

Architectural Design

Galaxy is a public computing utility. Public resources are highly dynamic because they are likely to join and leave the Galaxy overlay frequently. To ensure positioning consistency and availability of reference points, I choose to use infrastructure nodes like in GNP. These nodes are still named as landmarks. Landmarks are supposed to be quite stable nodes, i.e. they have low variance in network conditions and are available to provide service for long period of time. My positioning solution follows the two-part architecture: a small set of landmarks first compute their coordinates to serve as a frame of reference. Other end hosts refer to the coordinates of the landmarks to compute their own coordinates. Also a novel mechanism is used to adjust host's position within its nearby region.

In the following sections, the solution [1] to the network positioning problem is explained in detail. The positioning functions are integrated with RAN indexing method, routing mechanisms and ring organizing procedures to set up the location-based RAN.

3.1 Landmark Positioning

I model the Galaxy overlay as a D-dimensional Cartesian coordinate space. Network delays are mapped to the distances in the space. Considering the dimensions, more dimensions provide better accuracy. But the improvement is tiny after two dimensions. This observation is pointed out by principle component analysis on the matrix of latencies [34] and by similar conclusion in GNP [27]. In my design, I assume a space of dimension 2. Land-

[1]B. Maniymaran designed the positioning algorithms used for sequential simulations. I implemented them with decentralized strategies.

marks are randomly selected. Regarding the number of landmarks, I conduct a series of tests to study its effect on the positioning performance in Chapter 5.

I study two decentralized methods to compute landmark positions: *distributed spring algorithm* (DSA) and *spring equilibrium algorithm* (SEA). Both methods model the network as a collection of springs, each of which pulls on the coordinates of a pair of nodes. Solution is achieved by minimizing the energy in the spring system, which is proportional to the spring's displacement from its rest length squared. This is analogous to minimize the prediction errors in the network system. DSA uses the same mechanism as Vivaldi to adjust node's coordinates. It moves the node by simulating the movement under spring force. SEA formulates the force minimization problem as a linear system of equations, and creates a decentralized algorithm to solve the equations.

3.1.1 Distributed Spring Algorithm

The DSA is outlined in Algorithm 1. Each landmark node in the DSA calculates its own movement in the spring system. The input to the algorithm is a list of landmark addresses. The starting coordinates are set randomly or using a previous value if there exists one. In each iteration of the inner loop, the current node communicates with one of the landmarks on the list to measure the network latency to it, and to obtain the landmark's current position. In response to the landmark's coordinates, this node moves for a short step along the corresponding spring to reduce the current node's prediction error with respect to the landmark node. The length of the step is a fraction σ of the prediction error. As the node continually communicate with other landmarks, it converges to coordinates that predict distance well. The algorithm stops when the heuristic threshold is reached for 5 consecutive iterations. The node's new coordinate is returned as result.

The rate of convergence is governed by σ. I use a gradually decreasing σ value, as proposed in Vivaldi [30, 31]. σ is initialized to 1.0 when the algorithm starts. It is deducted by 0.025 in each iteration, and will not go below 0.05.

3.1.2 Spring Equilibrium Algorithm

Spring equilibrium algorithm [35] also models the network as a physical spring system. But it explores a different principle for converging nodes to proper positions.

According to the Hooke's law [36],

$$\overrightarrow{F} = k\overrightarrow{\delta}$$

Algorithm 1 Distributed Spring Algorithm

```
Input: $L = \{l_1, l_2, \ldots, l_N\}$: $l_i$ is the address of landmark $i$, $N$ is the number of landmarks
Output: $coords$: new coordinates of this node
1: $coords = random\ coordinates$
2: $\sigma = 1.0$
3: $change = conv\_limit + 1$
4: $iter = 0$
   {Heuristic metrics based on experiments: $conv\_limit = 5(ms), conv\_iter = 5$}
5: while $change > conv\_limit$ or $iter < conv\_iter$ do
6:   $old\_coords = coords$
7:   for every $i$ in $L$ do
8:     $\mathbf{C}_i = getRemoteCoords(i)$ {$\mathbf{C}_i$ is the coordinates of landmark $i$}
9:     $d_i = getMinDistSample(i)$ {$d_i$ is the distance to landmark $i$}
10:    $u = \frac{(c_i - coords)}{|c_i - coords|}$ {$u$ is the directional unit distance}
11:    $dist\_diff = \mid \mathbf{C}_i - coords \mid - d_i$
12:    $displacement = u \cdot dist\_diff \cdot \sigma$
13:    $coords = coords - displacement$
14:  end for
15:  $\sigma - = 0.025$
16:  $\sigma = \max(\sigma, 0.05)$
17:  $change = \mid old\_coords - coords \mid$
18:  if $change \leq conv\_limit$ then
19:    $iter = iter + 1$
20:  else
21:    $iter = 0$
22:  end if
23: end while
```

the force $\overrightarrow{F}$ of the spring connected is proportional to the deformation $\overrightarrow{\delta}$ from its rest length. k is the spring constant. Using the displacement of the two ends of the spring to replace the deformation vector $\overrightarrow{\delta}$, I get the following formula:

$$\overrightarrow{F} = k_{ij}(\overrightarrow{\delta_j} - \overrightarrow{\delta_i})$$

where i, j denote the two ends of the spring. As described above, the prediction errors in the positioning system are mapped to the spring deformation in the spring network. The spring equilibrium algorithm aims at finding a set of positions of nodes so that the spring system is in *equilibrium*, i.e. the force acting upon each node is zero. In the spring network, the resultant force on the landmark node i is:

$$\sum_{j \in v_i} [k_{ij}(\overrightarrow{\delta_j} - \overrightarrow{\delta_i})]$$

$$\sum_{j \in v_i} k_{ij}\overrightarrow{\delta_j} - \overrightarrow{\delta_i} \sum_{j \in v_i} (k_{ij})$$

where v_i is the list of nodes that locate on the other end of the springs pulling on landmark i. $\overrightarrow{\delta_i}$ is the displacement of the end i of the spring. I want to find the solution of vector $\overrightarrow{\delta_i}$ to satisfy the linear system of equations::

$$A\overrightarrow{\delta} = 0$$

where,

$$A = (a_{ij}) = \begin{cases} k_{ij} & j \neq i \\ -\sum_{j \in v_i} k_{ij} & j = i \end{cases} \qquad \overrightarrow{\delta} = \{\overrightarrow{\delta_i}\}$$

To solve this linear system of equations, I use the *successive overrelaxation method* (SOR) [37]. This is an extrapolation method that takes the form of a weighted average between the previous iteration and the computed Gauss-Seidel iteration successively for each component,

$$pDx^{(i)} = -(1-p)Dx^{(i-1)} - Ux^{(i-1)} - Lx^{(i-1)} \tag{3.1}$$

where the matrices D, U, and L represent the diagonal, strictly lower-triangular, and strictly upper-triangular parts of A, respectively. If $p = 1$, the SOR method simplifies to the Gauss-Seidel method. A theorem due to Kahan (Kahan, 1958) shows that SOR fails to converge if p is outside the interval $[0, 2]$. I choose a p value of 0.85 based on simulation results.

The pseudo code for spring equilibrium algorithm is shown as Algorithm 2. δ denotes the node's displacement. It is also the solution that the SOR method produces.

Algorithm 2 Spring Equilibrium Algorithm

Input: $L = \{l_1, l_2, \ldots, l_N\}$: l_i is the address of landmark i, N is the number of landmarks

Output: $coords$: new coordinates of this node

```
1: $coords = random\ coordinates$
2: $\delta$ is initialized as a vector (the same length as $coords$) of 0's
3: $relaxFactor = 0.85$
4: $change = conv\_limit + 1$
5: $iter = 0$
   {Default system parameters: $conv\_limit = 5(ms), conv\_iter = 5$}
6: while $change > conv\_limit$ or $iter < conv\_iter$ do
7:    $old\_coords = coords$
8:    $sum\_j$ is initialized as a vector (the same length as $coords$) of 0's
9:    for every $i$ in $L$ do
10:      $\mathbf{C}_i = getRemoteCoords(i)$ {$\mathbf{C}_i$ is the coordinates of landmark $i$}
11:      $d_i = getMinDistSample(i)$ {$d_i$ is the distance to landmark $i$}
12:      $u = \frac{(c_i - coords)}{|c_i - coords|}$ {$u$ is the directional distance unit}
13:      $dist\_diff = \mid \mathbf{C}_i - coords \mid - d_i$
14:      $sum\_j = sum\_j - dist\_diff \cdot u$ $\{-Ux_j^{(k-1)} - Lx_j^{(k-1)}\}$
15:   end for
16:   $\delta = \delta - (1 - relaxFactor) \cdot N \cdot \delta$ $\{-(1-p)Dx_i^{(k-1)}\}$
17:   $\delta = (\delta + sum\_j)/(relaxFactor * N))$
18:   $coords = coords - \delta$
19:   $change = \mid old\_coords - coords \mid$
20:   if $change \leq conv\_limit$ then
21:      $iter = iter + 1$
22:   else
23:      $iter = 0$
24:   end if
25: end while
```

3.2 Ordinary Node Positioning

Ordinary node uses the coordinates of landmarks to derive its own coordinates. The detailed method is outlined in Algorithm 3. Since landmarks are passive during this process and do not change their positions, ordinary host communicates once with each landmark to measure the network latency to it and obtain its coordinates. Based on the coordinates of the landmarks, the node adjusts its position to reduce prediction error to the landmarks. The algorithm is pretty much similar to the distributed spring algorithm except two points: first, the coordinate samples and distance samples are collected outside the outer loop (the 2nd and 3rd steps in the pseudo code); second, the criterion of convergence also changed. Once the node has not moved by more than 5 milliseconds, the algorithm is declared to be converged. This is because the positions of reference nodes are fixed. After the node has reaches the heuristic metric, the position of the node will not change by bigger steps.

Algorithm 3 Ordinary Node Positioning Algorithm

```
Input: $L = \{l_1, l_2, \ldots, l_N\}$: $l_i$ is the address of landmark $i$, $N$ is the number of landmarks
Output: $coords$: new coordinates of this node
 1: $coords = random\ coordinates$
 2: $\mathbf{C}_i = getRemoteCoords(i)$ {$\mathbf{C}_i$ is the coordinates of landmark $i$}
 3: $d_i = getMinDistSample(i)$ {$d_i$ is the distance to landmark $i$}
 4: $\sigma = 1.0$
 5: $change = conv\_limit + 1$
    {Default system parameters: $conv\_limit = 5(millisecond)$}
 6: while $change > conv\_limit$ do
 7:   $old\_coords = coords$
 8:   for every $i$ in $L$ do
 9:     $u = \frac{(c_i - coords)}{|c_i - coords|}$ {$u$ is the directional distance unit}
10:     $dist\_diff = \mid \mathbf{C}_i - coords \mid - d_i$
11:     $displacement = u \cdot dist\_diff \cdot \sigma$
12:     $coords = coords - displacement$
13:   end for
14:   $\sigma - = 0.025$
15:   $\sigma = \max(\sigma, 0.05)$
16:   $change = \mid old\_coords - coords \mid$
17: end while
```

3.3 Cluster-based adjustment of coordinates

In PIC paper[32], it is proposed a "closest" strategy for choosing reference nodes to enhance the distance estimation in the nearby region. Compared with long distance estimation, predicting distances among close nodes are more sensitive to the accuracy of the node positions. In order to provide reasonable distance estimation, for both large and small range, I compensate the random landmark selection with the following cluster-based position adjusting method [35]. I define a *cluster* as a set of nodes located in a bounded region in the coordinate space.

Before adjusting its position, each node in the cluster probes all other hosts in the cluster, divides the largest measurements by half, and uses the value as the *radius* of the cluster. It then disseminates the radius and its position to all other cluster hosts. Upon receiving another node's position and radius measurement, this node saves the position for the other node, compares the radius with its own, and the larger value is accepted as the radius. As the adjusting process starts, the node first uses the coordinates of all other hosts in the cluster to compute a center point in the Cartesian space. The second step, it computes its distance to the center point using the distance function. This value should not be bigger than the cluster radius. If its coordinates fall out of the radius circle around the center, it needs to adjust its position.

The adjustment is to move its coordinates along the line between itself and the center point until it goes within the circle. After the adjustment, the node sends its new coordinates to other cluster peers. Since this adjustment also affects the center of the cluster, the node re-computes the center point and repeats the adjustment until no further movement is needed. Figure 3.1 shows the flow of the adjustment process.

3.4 Creation of Location-based Names

Based on the positioning functions, the locations of RAN nodes are calculated and represented as coordinates. These values are utilized by the RAN to generate location-based names [35].

Coordinates are multidimensional data. Efficient query processing in coordinate space such as discovery and route selection is often based on range search or nearest neighbor search. Multidimensional index structures can be applied in order to achieve a satisfactory performance. It's proposed in [38] a solution to solve the multidimensional indexing problem. It's called *space filling curve* (SFC) method [39, 40]. SFC is a one dimensional curve

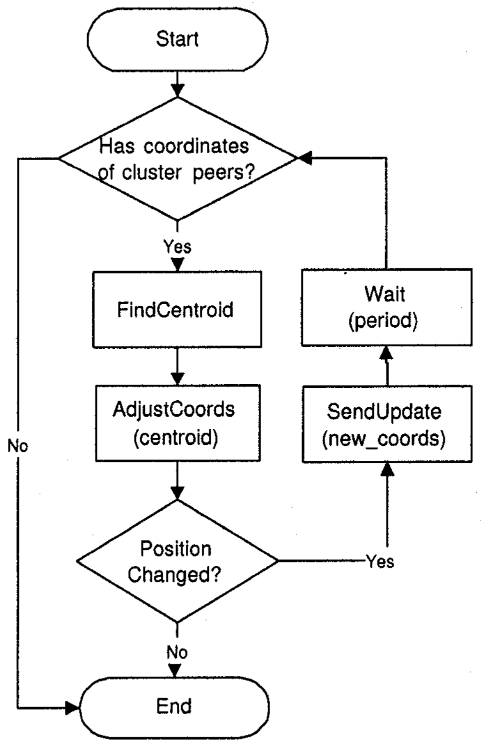


Figure 3.1: Cluster-based Adjustment

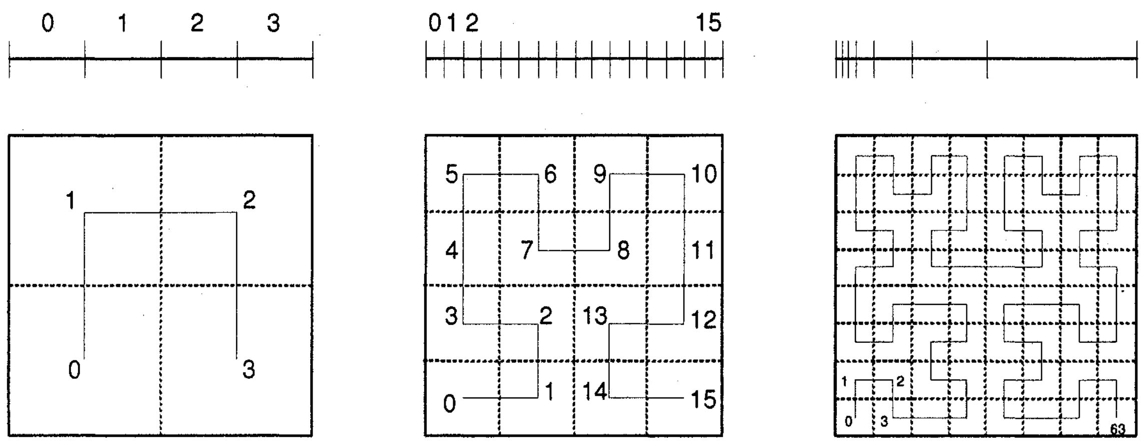
which visits every point within a bounded multi-dimensional space.

There are many derivatives of SFC. The design of the RAN uses one of them called *Hilbert curve* (HC). *Hilbert index* is derived from Hilbert curve. The index number can be single numbered as shown in Figure 3.2. Hilbert curve can be recursively constructed. Figure 3.2 shows the different levels of *approximation* of the Hilbert curve. The index number can also be represented in a hierarchical way. For instance, the point indexed "5" in Figure(b) can also be indexed as "2.2". The single numbered index is easy for comparing and ordering the points, so it performs better in nearest neighbor search. The hierarchical index indicates the approximation level of the node, which is good for range search. In the RAN implementation, I define the data structure for node index to include both features of the Hilbert index. I summarize the advantages of using Hilbert curve in RAN as follows:

1. It introduces a natural order as the curve goes through every node in the space.

2. It compresses the multi-dimensional space to a single dimensional curve.

3. The points that are close to each other in the multi-dimensional space are also close on HC. This is very important to conserve the proximity information.



(a) First level (b) Second level (c) Third level

Figure 3.2: Hilbert Curve

4. HC enables a hierarchical naming of the resources. This feature is useful to define *range* in routing and discovery mechanisms.

3.5 RAN Routing Mechanism

RAN node has three sections in its routing table [35]. The first section is called "self", keeping the pointer to itself. Every node has a left and right *ring pointers* in the "ring" section pointing to its two neighbor nodes on the ring. Ring is the basic topology that the RAN uses to manage profile-based names and location-based names. The two neighbors of the node on the ring are also the two neighbor nodes on the Hilbert curve. A node may have some other *jump pointers* that point to other non-neighbor nodes in its "jump" section.

Based on the data in the routing table, RAN performs two kinds of application-level routing:

1. Jump Routing: the message is sent directly to the destination, or to a *nearest* node of the destination from the list of jump pointers. In practice, I apply *longest prefix match* to the destination node and one of the nodes in the jump pointer section. If the length of the longest common prefix is beyond some threshold (I set the threshold to the number of digits of the destination index minus 1), the jump pointer is a nearest node of the destination. For example, index number "2.3.0.1.2" and "2.3.0.1.3" are the nearest node of each other, while "2.3.0.1.2" and "2.3.2.2.0" are not. If neither

exact match nor nearest match is found, routing is delegated to the ring routing.

2. Ring Routing: the message is sent along the ring structure to the destination. This routing mechanism is chosen when the node cannot route the message through jump routing. The operation is to compare the destination's Hilbert index with that of the right and left neighbors. The side closer to the destination is selected as the next hop.

3.6 Self-organizing Activities

RAN is a self-organizing P2P network [35]. RAN nodes can discover their neighbors and distribute information to others automatically. These nodes forms a ring configuration. Some procedures are designed for individual node to manage this configuration. The basic

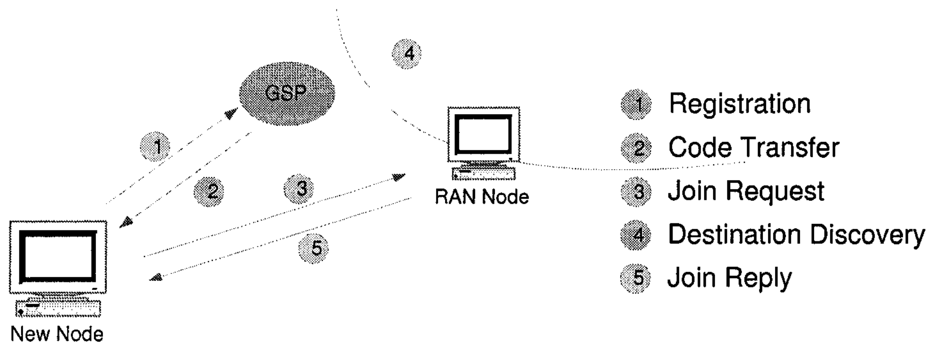


Figure 3.3: RAN Joining Process

scenarios are shown as follows:

- **Joining Process:** when a node newly joins the P2P network, it first contacts a list of landmarks to compute its coordinates and calculates its Hilbert index. After positioning itself, it contacts a node from a list of existing nodes, called its *entry node*, to request for inserting itself into the ring configuration. The entry node will forward the request to a destination with the most similar Hilbert index to the new node from the list of nodes it knows. The destination node of the joining request decides on which side the new node should be inserted according to index order, changes the ring pointer at that side, and sends the new node with the information of itself and its previous neighbor so that the new node can link itself between them. The list of landmark addresses and existing nodes can be obtained from the *Galaxy service* *provider* (GSP) through a secure web registration interface. After registration, GSP provides the new comer with the information of landmarks and existing nodes from its database. This service only functions at the node initialization stage. Figure 3.3 shows the steps of joining to the RAN.

- **Updating Process:** the positions of RAN nodes are updated periodically. For the landmarks, the update process is not often because changing landmark framework will result in changes to the location-based names of most nodes in the system. Nodes may need to rejoin the system as a result, which causes much overhead. So landmarks are configured to have an long update period. The updating process can be triggered by any of the landmarks. All the landmarks recompute their coordinates simultaneously. The update completes until all the landmarks have agreed distributedly on their new positions. The updating frequency of ordinary hosts is configured according to user's specification. The updating process includes rerunning the positioning algorithm and adjusting the new coordinates according to cluster information. After the update, it sends its new position to all the cluster neighbors.

- **Leaving Process:** the major task in the leaving process is to maintain the consistency of the ring configuration. This is done by informing left and right neighbors to connect themselves together.

- **Rejoining Process:** as a result of position updates, a node may change its coordinates and index number. If its new index requires a new place on the ring, it needs to rejoin the ring configuration. The rejoining process is first performing the leaving process, and then going through the same steps as joining.

Chapter 4

Implementation Description

This chapter describes the detailed implementation of the location-based RAN. A previous version of RAN [41] was implemented by Wadih Maalouf and Hasan Mirza. Their work was used to set up an emulation of the RAN overlay on one machine. I extend their software to be able to execute under a real distributed environment. The RAN prototype is implemented using Java and XML Schema. Section 4.1 provides an overview of the package hierarchy of the application. Section 4.2 defines the external service interface for the GRMS layer. Section 4.3 introduces the necessary libraries. The last three sections describe the classes of each package in detail.

4.1 Application Architecture

This section describes the package hierarchy of the RAN application. I follow the same package structure as that of the previous version RAN implementation. It uses a *layered architecture*, in which the system is divided into three layers. Each layer uses only the services of the layer immediately next to it. This layered structure provides clear modularity, shown in Figure 4.1

The *network layer*, defined in package `ran.net`, performs all the network access, like opening the socket, writing out the messages, and reading in the replies. It provides services of sending, exchanging messages, and notification of incoming messages. The *message layer* is defined in package `ran.message`. Message layer translates the message strings to message objects. The message objects are defined as part of the application logic using XML. Another functionality of this layer is to notify the *application layer* of receiv-

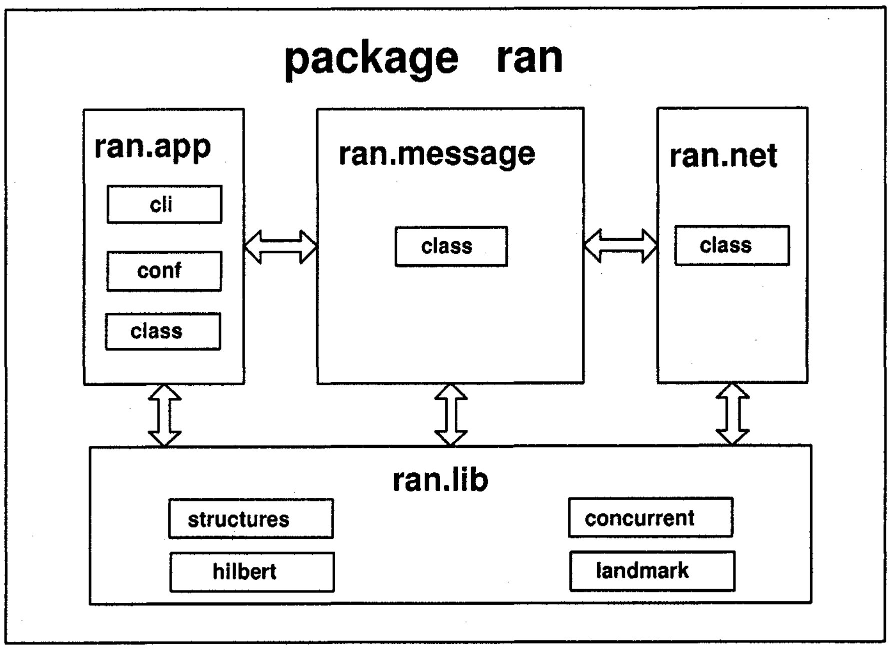


Figure 4.1: Package Structure

ing particular messages. The message notification is delivered by having the listeners registered to hear for different incoming messages. The *application layer*, defined in package `ran.app`, implements the application logic, including the positioning algorithms, indexing methods, routing mechanisms and the ring organizing procedures. Package `ran.lib` consists of supporting libraries for the other three layers in defining complex data types and operations.

4.2 External Interface

RAN provides a service interface to the GRMS. It is defined in Program 1.

Program 1 RAN External Interface

```
interface ran_discovery {
   boolean discover_execute(HilbertNumber id, String command)
}
```

This interface is devised for discovering particular resources through the RAN. It contains one method `discover_execute`. The method takes two input parameters, an `id` in *HilbertNumber* type and a `command` in *String* type. The return value is typed *boolean*.

This method can be implemented to search for a resource matching the desired Hilbert index number over RAN and execute the command string on it. The result of command execution, "success" or "failure", is returned to the object that calls the method. This interface can be implemented by any application that wishes to realize location-based resource discovery through the RAN.

4.3 System Requirements

The RAN application is implemented in Java. To run the application programs, a standard Java 2 Runtime Environment is needed. This would typically be using a Sun Microsystems J2SE. Except for this main environment, some external libraries are used:

- **Java Architecture for XML Binding (JAXB)**: JAXB is part of the Java Web Services Developer pack (Java WSDP). It contains all of J2EE's XML technologies for building, testing and deploying XML applications. I use the following JAXB compiler and runtime libraries included in this package: `jax-qname.jar`, `jaxb-qname.jar`, `jaxb-api.jar`, `jaxb-impl.jar`, `jaxb-lib.jar`, `namespace.jar`, `relaxngDatatype.jar` and `xsdlib.jar`.
- **EDU.oswego.cs.dl.util.concurrent**: written by Doug Lea, this package provides standardized, efficient versions of utility classes commonly encountered in concurrent Java programming.
- **java-getopt-1.0.9**: a Java port of GNU `getopt`, a class for parsing command line arguments passed to programs. It is based on the C `getopt()` functions in glibc 2.0.6.

4.4 Network Layer

The functions of network layer are encapsulated in package `ran.net`. They are utility classes necessary for physical network access. Messages are transferred over TCP connections. Each message is a UTF-8 character string terminated by a single blank line. The Doug Lea concurrency utilities are used here to construct an extensive thread pool for incoming socket connections. There are two types of connections, *one-way* and *two-way communication*. The one-way connection carries only one request from source host to destination, and closes after sending. This type of operation is implemented as `sendMessage` method. The two-way connection is opened when the host wants a reply. In this case, the connection will not be closed until a reply is returned or some timeout exception is catched. The `exchangeMessage` method is implemented according to this type.

Another important service in this package is the `MessageListerner` interface. It can be implemented by any class that wishes to be notified of some incoming messages. The instances of the implemented class are registered with this layer so that the method of `messageReceived` will be called each time an incoming message is received.

4.5 Message Layer

The class `MessageLayer` implements the service methods provided to the application layer. This class is going to be instantiated as a *singleton* (only one instance of a class). It provides the listener registration functions for the application layer to register their implemented procedures for dealing with different incoming messages.

The classes related to message types in package `ran.message` are generated by the JAXB schema compiler. Their definitions extend from the general definition of "message" object. The message structure is defined as Figure 4.2:

A "message" object is composed of two parts: header and body. The header has four elements: "to", "from", "type" and "subtype". "to" and "from" are the addresses of the message sender and receiver, respectively. "type" is the type of function of this message, such as "ping", "landmark", etc.. "subtype" is to mark the message whether it is a request or reply. Body is extended into two types: "Request" and "Reply". They are further extended by different message definitions.

The following list explains the functions of each message:

- **PingRequest and PingReply**: these types of messages are used to measure the round trip time between nodes. Both request and reply contain a single timestamp that is the time when this message has been sent by the sender. The sender may use this value to measure the round-trip latency.

- **JoinRequest and JoinAck**: this request message is sent when a node wishes to connect itself to the ring configuration. It is sent to an entry node for processing and does not require a reply. The join request message contains a route entry (the

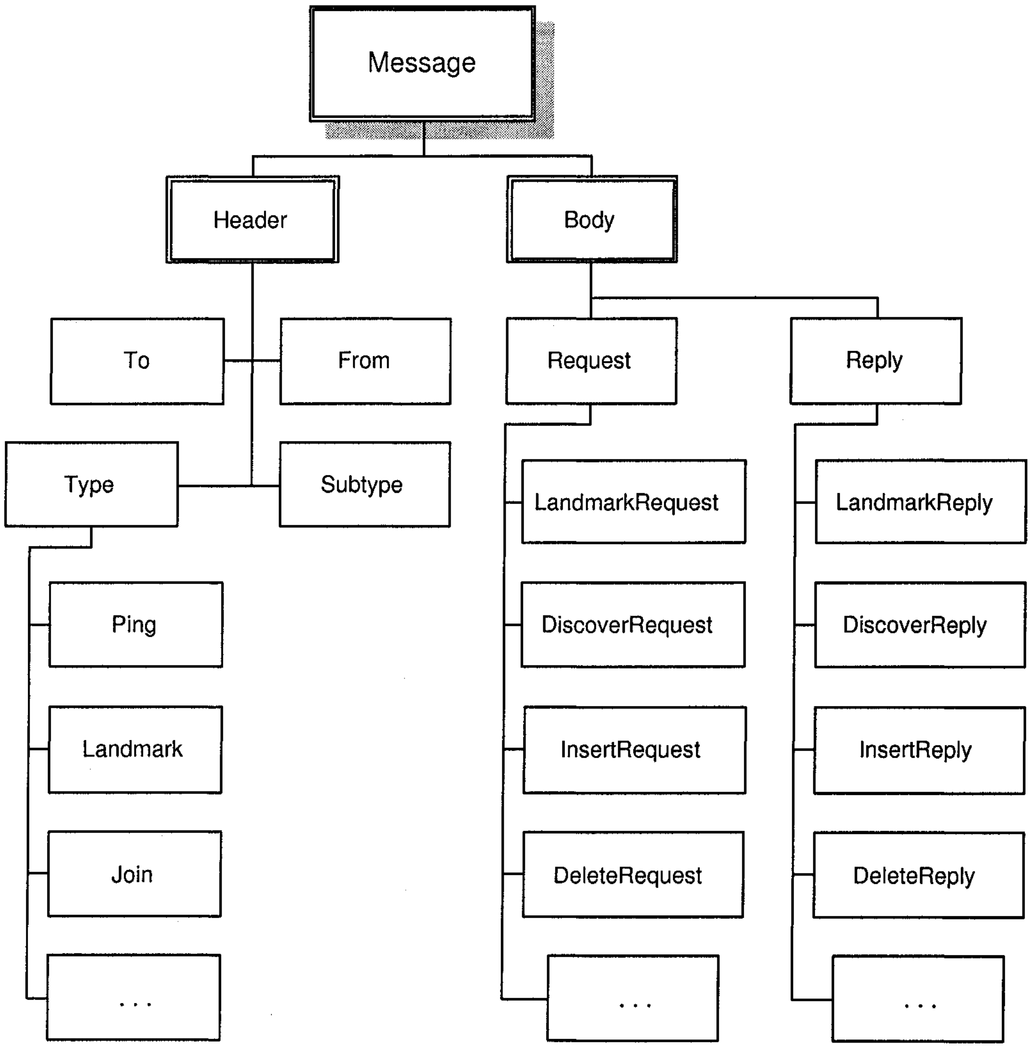


Figure 4.2: Message Object Definition

structure of a route entry is described in Section 4.6.5) of itself, plus a list of route entries of all nodes this message has passed through since the entry node, called "transit nodes". The join acknowledgement is not returned over the same route the request took, so the communication channel is closed after the request message has been received. The `JoinAck` message is sent after the joining request has been processed. It contains the route entries of the two RAN nodes the new node will insert between.

- **InsertRequest and InsertReply**: this request is used to inform the insertion of a new node between the destination node and its neighbor. An `InsertRequest` contains the route entry of the new node and the side on which the new node should be inserted (left or right) An `InsertReply` contains an acknowledgement status specifying whether or not the insertion was successful.

- **LandmarkRequest and LandmarkReply**: `LandmarkRequest` is sent to request for coordinates from a landmark. It contains a field called "operation" to specify the purpose of this request, whether it is sent by another landmark who is calculating its position or by an ordinary host asking for reference. The reply message contains the coordinates of the landmark and the network latency between sender and receiver measured by the reply sender. This latency value is used to unify the delays measured by the two ends of a network path when running the landmark positioning algorithm. The request and reply messages also contain a timestamp element for the request sender to measure the round-trip latency.

- **DeleteRequest and DeleteReply**: when a node wishes to leave the network, it sends `DeleteRequest` to its two ring neighbors. The message contains the route entry of the new neighbor to replace the sender, and the direction to which side the new neighbor should be connected. The Delete reply is an empty message just to force the leaving node waiting for the process finishing.

- **LocUpdateRequest**: this type of message is sent whenever a node wants to send its updated position to other nodes in the cluster. The message contains the coordinates of the sender and its measurement of the cluster radius. No reply is needed.

- **DiscoverRequest and DiscoverReply**: these type of messages are used in the resource discovery process in support of the `ran_discovery` interface. The `discoverRequest` message contains the Hilbert index of the desired resource, the route entry of the sender, a hop counter that counts how many nodes the message passes through, a *time-to-live* (TTL) value that specifies the maximum hops the message can take and a command string to be executed on the destination. The `discoverReply` message is returned directly to the request sender after the command execution has finished. It contains the route entry of the reply sender, the hop number that equals to the hop counter carried by corresponding request, and the result of command execution in boolean type.

4.6 Application Layer

The classes in this layer are contained in the package `ran.app`. They realize the positioning algorithms, indexing methods, as well as the the functions that maintain the ring configuration. I describe the major classes using the flow of the *main* method as a clue, which is showed in the following section.

4.6.1 Main Class

This class is the entry point for the RAN application. The *main* method goes through the following steps:

1. Parse the command-line arguments.
2. Parse the configuration file.
3. Test the network availability.
4. Register appropriate message handlers.
5. Initialize routing table.
6. Start the network layer.
7. Start the landmarking service, if the current node is landmark.
8. Run ordinary node positioning process if the current node is not landmark.
9. Run the command-line interface
10. Perform the RAN exit process if an "exit" command is encountered on command-line.

4.6.2 CommandLineParser Class

CommandLineParser uses the `getopt` library for parsing command-line arguments. The three options available now are:

- `-s`: specifies the directory where to search for the configuration file. By default is the current user's home directory.
- `-c`: specifies the name of the configuration file. By default is "config.xml".
- `-h`: prints out the usage of command line options.

4.6.3 SystemParameters Class

The system parameters are initialized by reading in a configuration file in XML format. Configuration file syntax is defined in the XML schema file "conf.xsd". The root element is bound to the namespace `http://www.ece.mcgill.ca/wmaalo/conf`, named as `conf.configuration`. The configuration elements includes the parameters needed to set up the network service, to initialize the routing table and to initialize the positioning algorithm. A brief description is shown as follows:

- **serverPort** (optional): contains an integer specifying the port on which the node listens for incoming TCP connections. By defaults 4050 is used.
- **hilbertDimension** (optional): contains an integer defining the dimension of the coordinate space to use. The default is 2.
- **entryNodes** (optional): contains a series of zero or more `node` subelements, each one containing the address (IP and port number) of an entry node. IP addresses must be in numerical form. If no entry nodes are specified, this node will be set up as a "lone" node, with left and right ring pointers pointing to itself.
- **clusterNodes** (optional): contains a series of zero or more `node` subelements, with the same data type as defined in `entryNodes`.
- **functionType** (required): contains a value in `NodeFunction` type defining the type of the node. `NodeFunction` is defined in enumerate values of "landmark" and "standard", standing for landmark node and ordinary node respectively.

- **algType** (optional): if a node is landmark, the value in this tag is processed to choose which computational algorithm to use for calculating landmark position. It contains a value in `AlgTypes` type, which is defined as enumerate values: "spring" stands for distributed spring algorithm, and "springEq" stands for spring equilibrium algorithm. By default, "springEq" is used. If this node is not a landmark, any specification in this tag will not have any effect.

- One of **landmarkCoords, landmarks**, or **forcedHilbert**. (required). The `landm-arkCoords` tag is used to assign a particular position to a landmark node. It contains two subelements, `position` and `range`, each of which contains a space-separated list of integers indicating, respectively, the node's position and the size of the coordinate space; The `landmarks` tag is used for specifying landmark nodes to be referred , either for landmark positioning or ordinary node positioning. It contains one or more `node` subelements that are the addresses (IP and port number) of landmarks; The `forcedHilbert` tag is used to force a particular Hilbert index for this node. It contains two subelements, `hierarchical` and `flat`; `hierarchical` contains a space-separated list of numbers forming the hierarchical index, and `flat` contains a single integer specifying the flat index.

4.6.4 Message Handlers

The message handler classes implement different types of listener interfaces defined in the message layer. One handler interface is provided for each message type. After an instance of these handler classes is registered with the message layer, the contained `_Received` method will be called whenever a message of the corresponding type is received. The handler classes are introduced below:

- **PingHandler**: handles incoming `PingRequest` messages. It is to reply to the sender with the same timestamp as of the request.

- **LandmarkHandler**: handles incoming `LandmarkRequest` messages. Only landmark node replies to this type of messages. If the value of the "operation" element in the request is "query" (a request from an ordinary host asking for reference), the handler checks if the landmark has finished running positioning algorithm. If so, it retrieves the coordinates and returns that with the reply message. Otherwise it sends back a null value in the reply. If the "operation" is assigned as "alg" (a request from another landmark when running the positioning algorithm), the handler first checks if a landmark positioning algorithm is running. Since landmarks may not reach the convergence at the same time, I define a *wait-for-end* time (in my experience, 10 minutes is adequate for 30 landmarks to finish their computation). A landmark just finishing its computation will not be triggered by other landmarks' request messages for the wait-for-end time. If the wait-for-end time has passed since last update, the handler triggers a new round of algorithm running. Then the handler returns in the reply with the current coordinates and the delay it measured between itself and the request sender.

- **LocUpdateHandler**: handles incoming `LocUpdate` messages. The handler uses the coordinate value in the request to update sender's position. If the radius value in the message is bigger than the local measurement, it is accepted as the new cluster radius. No reply message is returned for this message.

- **JoinRequestHandler**: handles incoming `JoinRequest` messages. The routing method is called to determine the next hop of this request. If the current node is not the destination, it is added to the list of "transit nodes" in the request, and the request is forwarded to the next hop. Otherwise, the joining request is processed at the current node. If current node's status is "leaving" (see class description of `StateTracker` in Section 4.6.10), the handler waits for the exit process to complete and forwards the request to its left neighbor (without adding itself to the "transit nodes" list.) Otherwise, the handler chooses the side for the new node to insert according to the Hilbert index order, then attempts to acquire the lock of the ring pointer for that side (see class description of `RouteTable` in Section 4.6.5). If unsuccessful, the message is re-routed after a short, arbitrarily chosen delay (currently 750 ms) for the network topology may be changing. If the lock is successfully acquired, an `InsertRequest` is sent to the neighbor to the side on which the new node is to be inserted. This request may produce a "positive" or "negative" reply. If "positive", the current node's ring pointers are updated, and a join acknowledgment is sent to the new node, instructing it to set its ring pointers. If "negative", the message is re-routed.

- **InsertHandler**: handles incoming `InsertRequest` messages. If the node's status is "leaving", a "negative" reply is immediately returned. Otherwise, the handler tries to acquire the lock on the side where the new node is to be inserted. If acquired, the appropriate ring pointer is updated and a "positive" reply is returned. It also adds the new node to its jump pointer table for use. If not, the possibility is that two adjacent nodes send insertion requests to each other. In this case, to avoid deadlock, only insertion on the right side waits until the lock is free. The insertion to the left side will result in a "negative" reply.

- **JoinAckHandler**: handles incoming `JoinAck` messages. The node replies to the first such message if there are multiple ones. The handler sets its two ring pointers to the values specified in the message and adds the jump pointers in the request into local jump table. Another operation is to set the node's state to "active" so that `main` method may proceed to run the command-line interface.

- **DeleteHandler**: handles incoming `DeleteRequest`. The handler acquires the ring lock on the appropriate side, then updates the corresponding ring pointer and returns an empty reply

- **DiscoverRequestHandler**: handles incoming `DiscoverRequest`. The handler employs the routing algorithm to determine the next hop for the Hilbert index specified in the request. If the destination is itself, the node executes the command string and generates a `DiscoverReply` to the request source; otherwise it increases the hop number by 1 and routes the request to next hop.

- **DiscoverReplyHandler**: handles incoming `DiscoverReply`. It prints out sender's address, Hilbert index and the total number of hops as in the message.

4.6.5 RouteTable and Router Class

The routing table has three sections: "self", "ring", and "jump". The "self" section contains only one route entry of itself. It is accessed with the `setSelf` and `getSelf` methods. The data type `RouteEntry` is defined in XML schema file "structures.xsd". It contains two elements: node's Hilbert index and IP address.

The "ring" section contains two entries, one pointing to each immediate ring neighbor. It is implemented using a *hashtable*, with the "direction" as the key. Ring pointers are accessed with the `getRingPointer` method. Since multiple insertions and/or deletions may be attempted on one side of a node, each ring pointer is protected by a lock. A thread may only update a ring pointer if it holds the corresponding lock, and each lock may be held by only one thread at a time. The `acquireRingLock` and `tryRingLock` methods allow threads to attempt to acquire locks in both blocking and non-blocking fashions. Once a lock is acquired, the corresponding ring pointer may be set using `setRingPointer`, and then the lock released using `releaseRingLock`.

The "jump" section contains route entries pointing to nodes at arbitrary locations in the network. It is implemented using a *two-dimensional array*. Row number corresponds to the approximation level of a Hilbert number, and column number equals to the digits on that level. The `getJumpPointer` method returns the jump pointer at a specific column and row, or null if none exists there. The `addJumpPointer` method adds a jump pointer to the table, automatically calculating the correct row and column from the pointer's Hilbert index. The `addAllJumpPointers` method adds a collection of jump pointers to the table; the `getAllJumpPointers` method retrieves all jump pointers currently in the table.

Router class implements the RAN routing algorithm. The routing algorithm is implemented in methods `route`, `jumpRoute` and `ringRoute`. The `route` method is the entry point that triggers the algorithm to be applied. The `getDest` and `destIsSelf` methods return the results of the algorithm. The routing process can be controlled using the "exactJumpsAllowed" flag. If it is set, the algorithm will not select a jump pointer that exactly matches the target Hilbert index. This avoids the problem of the nodes that has left the network already, which causes the pointers to be obsolete.

4.6.6 Landmark Class

Implements the landmarking service at this node. This class is implemented as the singleton pattern. Only landmark node initializes the unique instance of this class. The unique instance can be accessed using the static method `getLandmark`. There are two `initialize` methods. The `initialize` method with one parameter in "Coords" type is used to assign the parameter as landmark coordinates. The `initialize` method without parameters is used to assign a random coordinates to the current landmark node.

The method `compLanmdark` is the entry point that triggers the positioning algorithm. The `DSAlg` and `SEAlg` methods compute the landmark position according to distributed spring algorithm and spring equilibrium algorithm respectively. Inter-landmark latencies are stored in a hashtable called "distsTable" with landmark addresses as the keys, and can be accessed using `getDelays` method. The coordinates of the landmark can be retrieved by `getCoords` method.

A nested private class `UpdateLandmark` extends the Java class `TimerTask`. This

Java class is used to create a timer task. It has a `run` method that defines the action to be performed by the timer task. I extends the `run` method to realize the updating process of landmark position. The instance of `UpdateLandmark` class is registered with a private timer in the `Landmark` class using `registerLMUpdate` method, and is triggered to execute periodically by the `main` method. The `run` method calls the positioning method to compute landmark coordinates, generates the Hilbert index, and triggers the joining or rejoining process if applicable. It may also sends the new position to all cluster neighbors if cluster nodes are known. Figure 4.3 shows the methods in the `Landmark` class.

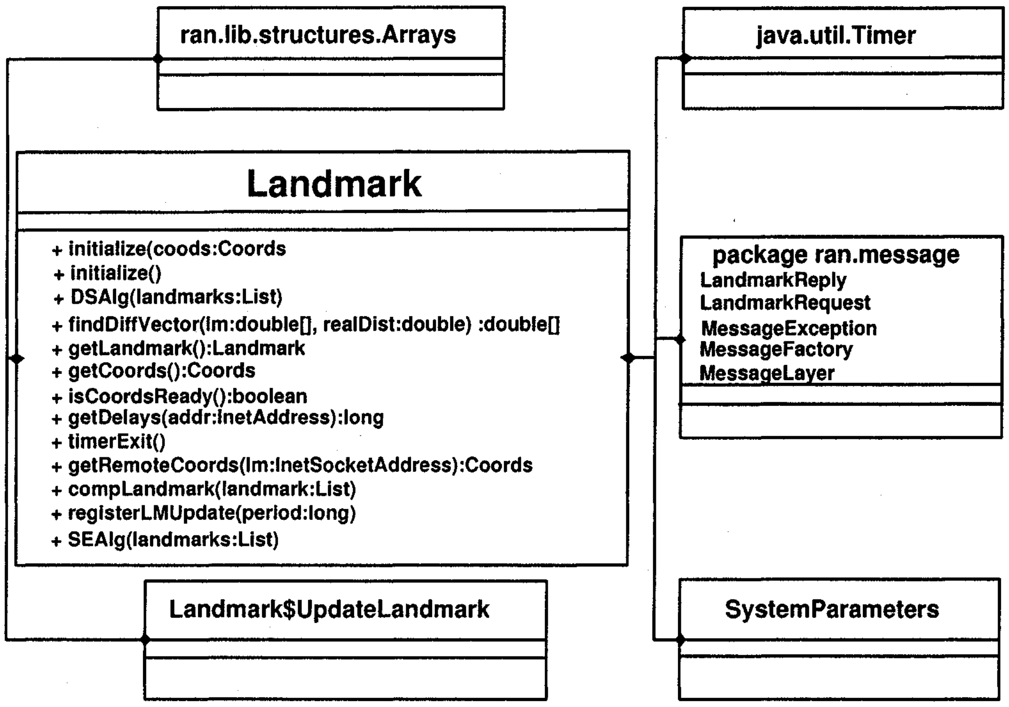

ran.lib.structures.Arrays
java.util.Timer
Landmark
+ initialize(coods:Coords
+ initialize()
+ DSAlg(landmarks:List)
+ findDiffVector(lm:double[], realDist:double) :double[]
+ getLandmark():Landmark
+ getCoords():Coords
+ isCoordsReady():boolean
+ getDelays(addr:InetAddress):long
+ timerExit()
+ getRemoteCoords(lm:InetSocketAddress):Coords
+ compLandmark(landmark:List)
+ registerLMUpdate(period:long)
+ SEAlg(landmarks:List)
package ran.message
LandmarkReply
LandmarkRequest
MessageException
MessageFactory
MessageLayer
Landmark$UpdateLandmark
SystemParameters

Figure 4.3: Class Landmark

4.6.7 LocationUpdate Class

Implements ordinary node positioning. It is designed as a singleton. Ordinary node initializes the unique instance of this class. This unique instance can be accessed using the static method `getLocationUpdate`.

A nested private class `UpdateLocation` extends the Java class `TimerTask`. It realizes the position updating process of the ordinary node. An instance of this class is registered with the private timer of the `LocationUpdate` class using `registerLocati-`

`onUpdate` method and is triggered to execute periodically by the `main` method. The `run` method first calls the `doLandmarking` method to compute node's coordinates, generates the node's Hilbert index and performs a joining or rejoining process if applicable. If cluster nodes are known, the `updateRadius` method is called to find the radius of the cluster. The node's new position and cluster radius are sent to all cluster neighbors. The final step, it triggers the cluster-based adjustment described below to start. The data of cluster radius can be accessed through `setRadius` and `getRadius` methods.

The cluster-based adjustment is implemented in another nested private class `Cluster-Adjust`. It is also a timer task, and registers its instance with the private timer using `registerClusterAdjust` method. This task is triggered to execute by the `run` method in class `UpdateLocation`. This timer task includes calculating the cluster center using the `findCentroid` method, and adjusting the node's position to be within the radius circle around the center using the `adjustSelfCoords` method. If the node's position does not change after the adjustment, the adjusting process is finished by cancelling itself with the timer. This adjusting task is configured to run in much shorter period than the position updating task so that the adjustment can finish before next position update starts.

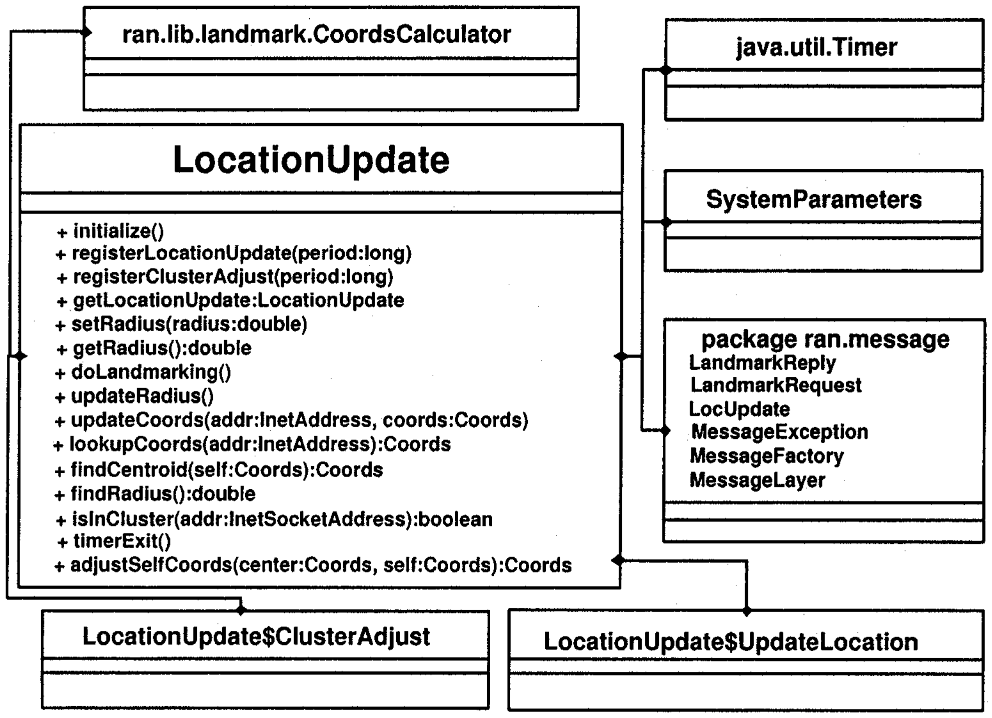

ran.lib.landmark.CoordsCalculator
java.util.Timer
LocationUpdate
+ initialize()
+ registerLocationUpdate(period:long)
+ registerClusterAdjust(period:long)
+ getLocationUpdate:LocationUpdate
+ setRadius(radius:double)
+ getRadius():double
+ doLandmarking()
+ updateRadius()
+ updateCoords(addr:InetAddress, coords:Coords)
+ lookupCoords(addr:InetAddress):Coords
+ findCentroid(self:Coords):Coords
+ findRadius():double
+ isInCluster(addr:InetSocketAddress):boolean
+ timerExit()
+ adjustSelfCoords(center:Coords, self:Coords):Coords
SystemParameters
package ran.message
LandmarkReply
LandmarkRequest
LocUpdate
MessageException
MessageFactory
MessageLayer
LocationUpdate$ClusterAdjust
LocationUpdate$UpdateLocation

Figure 4.4: Class LocationUpdate

4.6.8 JoinProcess, LeaveProcess and ExitProcess

The `doJoining` method in the `JoinProcess` class implements the RAN joining process. The constructor takes the node's Hilbert index as parameter. The `doJoining` method first sets the self pointer in the routing table using `setMyRouteEntry` method. If no entry nodes were specified, the node set the two ring pointers pointing to itself using `setupLoneNode` method; otherwise it calls the `sendJoinRequest` method to send joining requests until one is successfully reached or until all nodes are exhausted.

The `doleaveProcess` method in `LeaveProcess` class implements the node's leaving process. It first checks if the node's state has changed to "active", which means the node has successfully joined the RAN. Only as a joined node, it sends delete messages to its two ring neighbors and waits for the replies. The `doExitProcess` method in the `ExitProcess` class calls the leaving process to function, and cancels the timer for updating position. The RAN process ends.

4.6.9 CLI Class

CLI implements a simple command-line interface. Commands include:

- debug: shows the items now in the routing table.
- help: shows the available commands and its abbreviated forms.
- time: shows the current system time in milliseconds.
- ping: pings other hosts using IP addresses.
- exit: terminates the RAN program.

4.6.10 NodeStartupException and StateTracker Class

This exception class represents a wide range of errors when setting up the RAN node, from the failure to parse command line arguments to the failure in initializing node's position. These errors are handled in the same way: print out an error message and terminate the program running.

The StateTracker class defines the phases RAN node may go through as: "startup", "waiting for join ack", "active", "leaving" and "finished". The reason to have those phases is that some actions may happen depending on the satisfaction of node's current state. Such as in the leaving process, the deletion of ring pointers is based on whether this node has joined the ring configuration. Methods are provided for setting, retrieving and waiting for states. The state is represented by a nested public type-safe enumeration class `State`.

Chapter 5

System Tests and Results

5.1 Testbed Description

I performed an experimental evaluation of the positioning schemas based on the *PlanetLab* [24], which is an ideal testbed for overlay networks. It provides distributed virtualization, the ability to allocate PlanetLab's Internet-wide hardware resources to different applications in the form of "slice." Organizations or institutes may register for creating a *site* on PlanetLab. A site usually consists of 3 to 10 machines. PlanetLab presently consists of 583 machines, spanning over 25 countries. All those machines run a common software distribution that includes a Linux-based operating system. The advantage of using PlanetLab is that I am able to do experiments with new services under real-world conditions, and at large scale. Moreover, as a realistic network testbed, experiments are going through complex conditions like congestion, failures, diverse link behaviors, even the potential for a realistic client workload.

To control large collections of nodes in the wide-area, I use the user service package called *pssh*[42], developed by Intel Research Berkeley. It contains parallel openssh tools *pssh* (parallel ssh), *pscp* (parallel scp), *prsync* (parallel rsync), *pnuke* (parallel nuke), and *pslurp* (parallel slurp). They are Python scripts implemented in a multi-threaded fashion. I use these tools to configure and execute programs on more than one hundred PlanetLab machines.

The installation of RAN distribution on PlanetLab includes the following two steps:

1. Install Java software and change the profile to include Java executables. In the original PlanetLab slice assigned for RAN experiments, there was no Java environment.

So I wrote a Shell Script to install Java to "/usr/share/" directory and change the profile to include the Java path.

2. Transfer the RAN distribution to PlanetLab machines. Untar the package to the home directory of the RAN executable. The package contains the JAR file to run RAN service and a subdirectory called "lib" containing the required libraries. The libraries include the following 10 JAR files:

    - concurrent.jar
    - java-getopt-1.0.9.jar
    - jax-qname.jar
    - jaxb-qname.jar
    - jaxb-api.jar
    - jaxb-impl.jar
    - jaxb-lib.jar
    - namespace.jar
    - relaxngDatatype.jar
    - xsdlib.jar

5.2 Node Selection

I use a script to probe all the machines hosted in PlanetLab sites and obtain responses from machines in 127 different sites. Most of these machines are hosted by research institutes, mainly located in North America, Europe and Asia. I chose one machine from each site and installed the RAN prototype on them. Then I gather the inter-host RTTs by using a script to send out ICMP ping packets (10 RTT samples per path). The data collection lasted for one week. I run the script for ten times at different time of the day. The data is processed by always choosing the minimum sample as the real network delay [43]. I note that not every node pair has a packet route. The absence of network delay is denoted as "-1".

The landmark nodes are randomly selected. Note that PlanetLab is a shared testbed. Furthermore, PlanetLab machines reboot periodically (unknown period) for maintenance and upgrade purposes. In the experiments, I filter out those landmarks that are frequently *out-of-service* (i.e. machine is down or refuses connection to port 22) by practical observation.

In order to test the cluster-based adjustment schema, I define the diameter of a cluster area as 30 milliseconds. By filtering the 127 nodes, I obtain 3 groups with 24, 14 and 11 nodes respectively. These groups are located in different regions in North America.

5.3 Experiments

I study four issues in the experiments:

1. The accuracy of the distance estimation, convergence time, and position stability of the RAN positioning.
2. The performance of distributed spring algorithm and spring equilibrium algorithm for landmark positioning. Performance in the following aspects: accuracy, convergence time and message overhead.
3. The influence of different number of landmarks on ordinary node positioning.
4. The improvement of accuracy on distance estimation and position stability using the cluster-based adjustment schema.

5.3.1 RAN Positioning Performance

Two metrics are used to measure the accuracy of distance prediction. The first one is called "correlation." Correlation is the degree to which two or more quantities are linearly associated. The correlation ρ_{xy} between two sets X and Y is defined as:

$$\rho_{xy} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E((X-\mu_X)(Y-\mu_Y))}{\sigma_X \sigma_Y} \tag{5.1}$$

where σ_X, σ_Y are standard deviations, and μ_X, μ_Y are mean values. If the variables are independent then the correlation is 0, while the correlation approaches 1 with increasing linear relationship. In my experiment, X and Y are the predicted and measured distances, respectively.

Another concept is called the *relative error*. It is defined as:

$$\frac{|predicted\ distance - measured\ distance|}{\max(measured\ distance, predicted\ distance)} \tag{5.2}$$

The relative error value ranges from 0 to 1. A value close to zero indicates a perfect prediction; on the other hand, a value close to 1 is considered to be a weak estimation.

I randomly select 30 nodes as landmarks. The remaining 97 nodes act as ordinary hosts. The Cartesian space is configured to have 2 dimensions. The landmarks are started 10 minutes prior to the starts of ordinary hosts. Landmarks update their positions once every 3 hours. The experiment lasted for 5 landmark updating periods. Ordinary hosts update their positions once every hour.

The convergence time is measured starting from the first probe to any landmark is sent until the node's position has not changed by more than 5 milliseconds over 5 consecutive computation iterations. The convergence time of ordinary node positioning is determinable because the algorithm is centralized. In the experiments, I mainly study the convergence time of landmark positioning which is undeterminable since landmarks simultaneously update their positions and need to agree on their positions distributedly. Figure 5.1 shows the average algorithm convergence time for 30 landmark during 5 landmark updating periods. Spring equilibrium algorithm is used to compute landmark positions. Error bars indicate

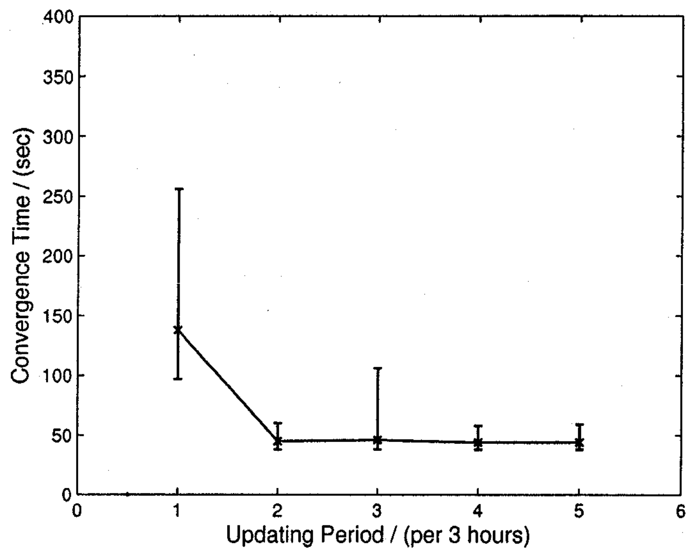


Figure 5.1: Positioning Evaluation: Landmark Convergence Time

the time difference among landmarks in finishing the positioning algorithm at each update. I observe that in the first update, all landmarks finish the algorithm in around 260 seconds. There are two factors that affect the convergence time of landmarks: the probing latencies to other landmarks and the number of iterations it repeats the probing until reaching the heuristic criterion. The bigger time difference at the initial update can be the result of more iterations each landmarks made to converge than the latter updates. A significant drop of average convergence time occurs from the second update. The average algorithm time for the latter 4 updates maintains at less than 1 minute. This performance is much better than the experimental results reported by the NPS which used 15 landmarks (also on PlanetLab) and obtained average convergence time as 160 seconds [29]. I notice that a jitter happens at the third update in the value of maximum convergence time. Since the average convergence time does not change much, the most possible reason for this jitter is network congestion or heavy workload on one landmark.

The Figure 5.2 shows the correlation results of the landmarks and the whole system of 127 nodes during 5 landmark updating periods. The correlation of landmarks stays at

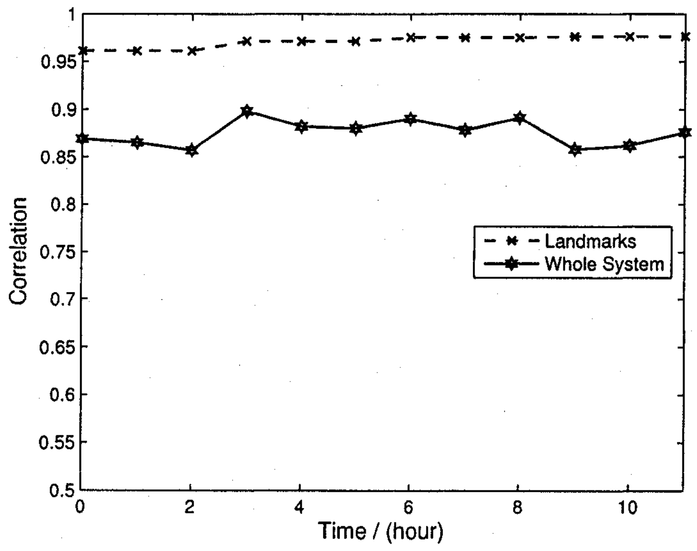


Figure 5.2: Positioning Evaluation: Correlation

a very stable level, all achieving above 0.96, which indicates high association between the predicted distances and the measured ones among them. The correlation of the whole system fluctuates in a small range around 0.85. There is no degradation to this metric from the beginning to the end. This is an important validation of the system's ability to maintain position accuracy over time in a dynamic environment. The reason why landmarks achieve higher correlation is that they adjust their positions directly according to their inter-landmark latencies, while the network latencies among ordinary hosts are not used in computing their positions. The result of another accuracy metric, relative error, is shown in Figure 5.3. In the figure I compare the distance prediction accuracy at the test

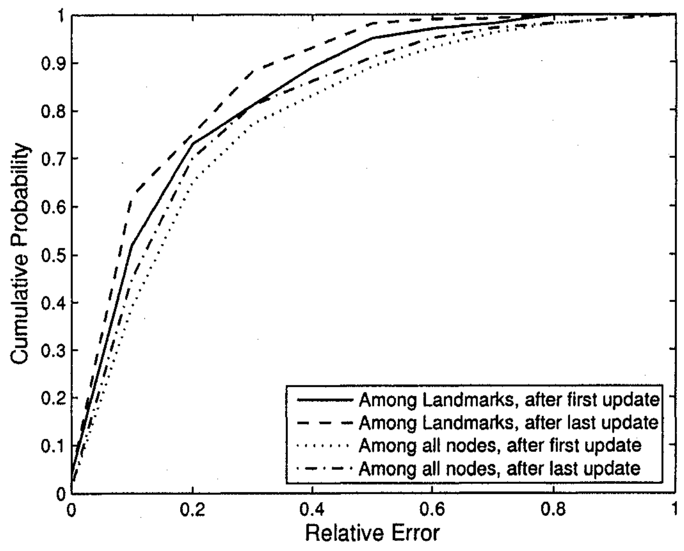


Figure 5.3: Positioning Evaluation: Relative Error

beginning and end phases, among landmarks and the whole system of 127 machines. The 50 and 90 percentile relative error for landmarks are 0.08 and 0.41 respectively after the first update. These figures decrease to 0.05 and 0.34 at the end of the test, showing that the level of accuracy for landmark positions has improved. For the whole system at the test beginning time, the 50 and 90 percentile relative error are 0.13 and 0.55, respectively. The whole system also has a better level of accuracy at the end of the test, showing relative error 0.11 and 0.48 for the two percentiles.

I also study the drifting of node's coordinates after each update. Big movement of node's position cause the node's Hilbert index to change, and a rejoining process may be needed for this node to find a new ring location. To avoid unnecessary rejoining workload, it's very useful to learn the range of position drifting under normal fluctuation of network latencies. I assume the changes of inter-host latencies among PlanetLab nodes to be normal fluctuations. To analyze this issue, I introduce a concept called *average occupancy diameter*. Occupancy diameter of a node at time t is the minimum diameter of the area in which all of its T past coordinates are found. In the experiments, I use T value of 4. As shown in Figure 5.4, the average diameter values for landmarks saturated as the test continues, while the line which shows the average diameter values for the whole system can be seen to increase like stairs. Each stage indicates a relative big movement of the nodes' positions, and the time for each stage is coincident with each landmark update. Figure 5.5 shows the average occupancy diameter of the whole system with minimum and maximum data samples. While the maximum diameters are abnormally high, the majority of nodes should have diameters equaling to or below the averages. The variance in network latencies could be a reason of this phenomenon. I use previous data of inter-landmark distances to smooth latencies used to compute landmark positions. But this strategy will make landmarks insensitive to the topology change by always using the minimal latencies [27, 34]. For ordinary nodes, I assume them to be less stable than landmarks. So I do not apply the same method to their distance measurements. As a result, the positioning of ordinary nodes are much more sensitive to network latency changes. Since only few nodes are affected by big network variance, the dominant reason for node drifting is the change of the landmark framework. To minimize unnecessary changes of landmark positions, one possible solution is to define a changing threshold. If the distance from landmark's new position to the old one is smaller than the threshold, the landmark keeps its old position. This schema can help landmarks to stick to their positions under normal network variance, and therefore keep the whole system from big shifting.

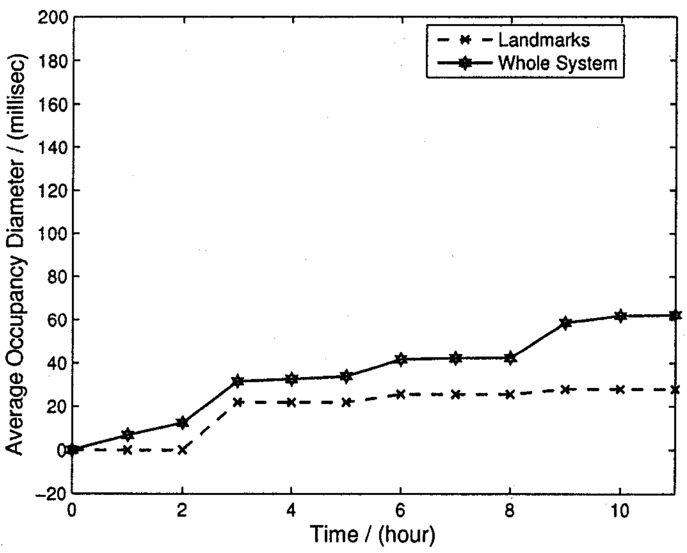


Figure 5.4: Positioning Evaluation: Average Occupancy Diameter

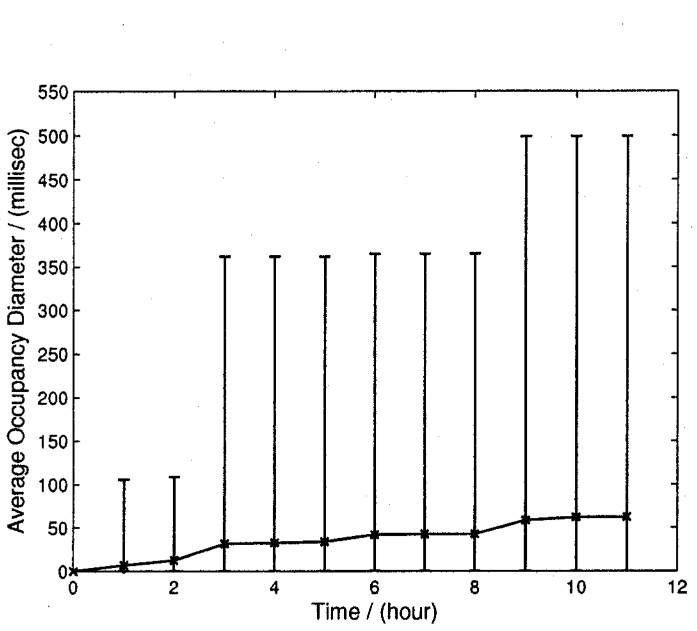


Figure 5.5: Positioning Evaluation: Average Occupancy Diameter (Error Bar)

5.3.2 Distributed Spring Algorithm vs. Spring Equilibrium Algorithm

I present in this section the experimental results of two different landmark positioning algorithms. The experiments include tests of accuracy, convergence time, and number of messages sent out from each landmark. I perform two sets of experiments:

1. Evaluation of the two algorithms with different number of landmark participants.

2. Evaluation of the two algorithms during 5 landmark updating periods

The convergence time uses the same definition as above. The number of messages sent out from each landmark is measured during the convergence time. Correlation and relative error are used to measure the accuracy.

I select landmark sets of 10, 15, 20, 25, and 30 nodes respectively. In order to reduce the impact of node selection on the results, for each number set I run the experiments for 5 times, and use different combination of nodes at each time.

Figure 5.6 shows the average correlation results for the different number of landmarks with maximum and minimum data. (I increases the x values for SEA data by 1 to avoid overlapping of the two curves.) Both algorithms achieve very high correlation in the tests, all of which are above 0.96. A slight drop (less than 3 percents) shows for both algorithms when the number of participants increases. Since Landmark adjusts its position according to all other landmarks, more participants means less percentage of the effect for

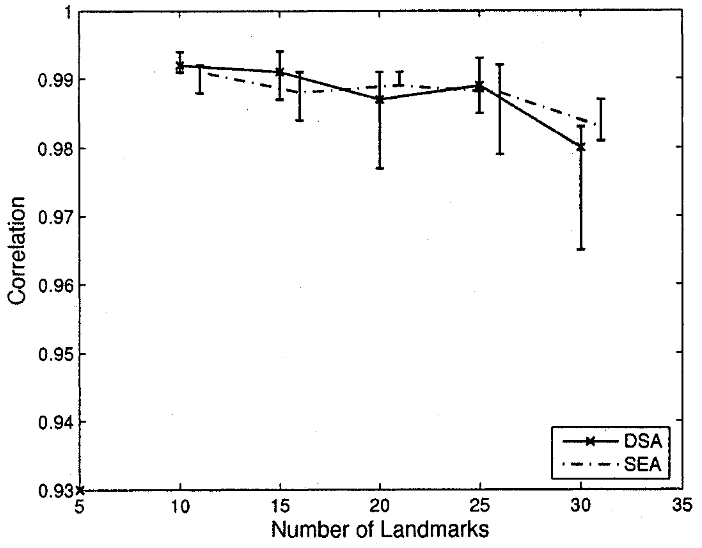


Figure 5.6: DSA vs. SEA (Number of Landmarks): Correlation Comparison

each landmark enforcing on the final adjustment. Comparing the two algorithms, spring equilibrium algorithm has more stable accuracy performance than distributed spring algorithm. I choose the groups with correlation results closest to the average values, and display their results of relative error as Figure 5.7. There is no significant difference between the two algorithms in this performance. However, I observed that distributed spring algorithm occasionally failed to converge after 25 landmarks. It has bigger dependence on node selection than spring equilibrium algorithm.

Another aspect of interest is the convergence time. Figure 5.8 shows the average convergence time of two algorithms with different number of landmarks. I find that, the convergence time for both algorithms increases quickly when more nodes participate in. Since landmark probes all other landmarks before tuning to the next position in the algorithm, more landmarks to contact cause more network delays. Moreover, large number of landmarks increase the difficulty for all the landmarks in the system to agree distributedly on their positions. Comparing the two landmark positioning algorithms, the convergence time of DSA increases more quickly than SEA. Besides, the time performance of DSA is less stable compared with SEA.

Figure 5.9 shows the result of another metric, the average number of messages sent out from each landmark. The message traffic for both algorithms increase as the number of landmark adds. This result confirms my analysis in the part of convergence time that the

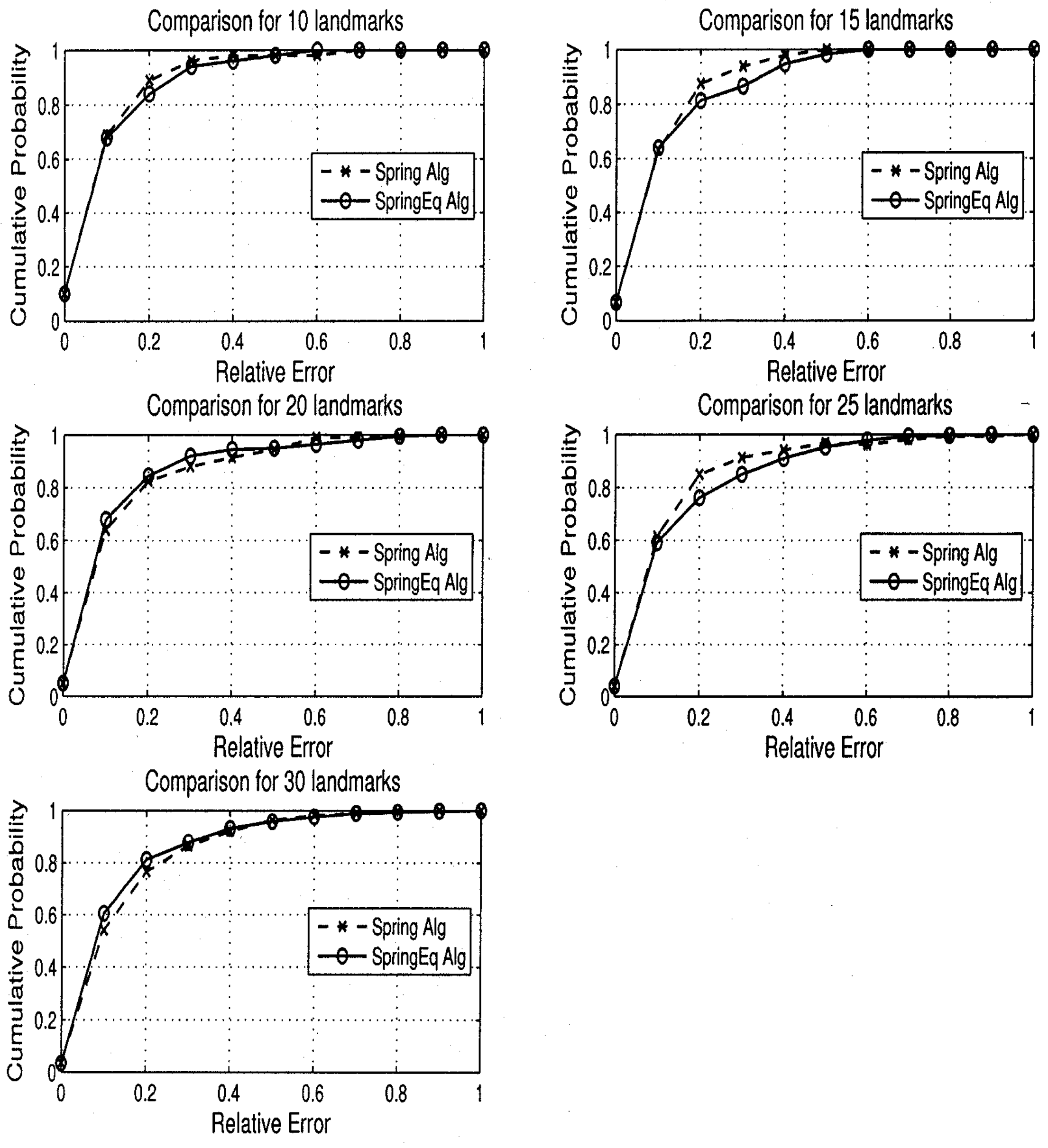


Figure 5.7: DSA vs. SEA (Number of Landmarks): Relative Error

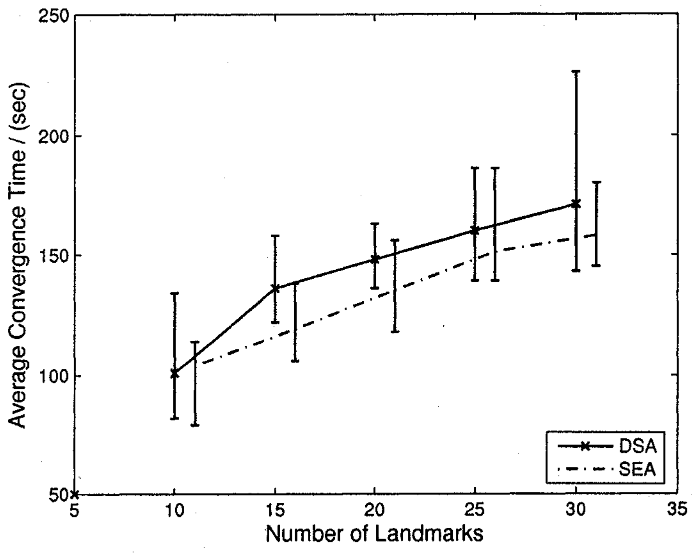


Figure 5.8: DSA vs. SEA (Number of Landmarks): Convergence Time

increase of messages is one of the major reasons for longer convergence time. The message overhead of DSA increases a little bit quicker than SEA. Considering the results of both the convergence time and message overhead, spring equilibrium algorithm is a better choice for landmark positioning for landmark number above 20.

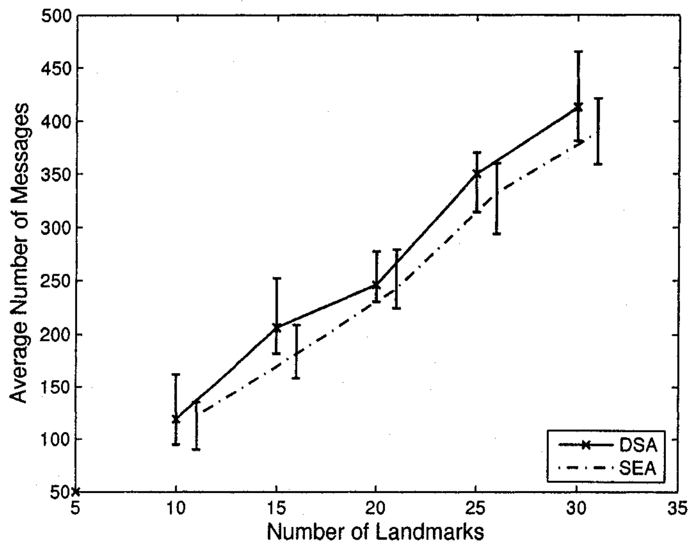


Figure 5.9: DSA vs. SEA (Number of Landmarks): Message Overhead

Next set of experiments I conducted is the evaluation the two algorithms for long-time execution. I ran the two algorithms on 30 landmarks for 5 updating periods. The updating period is 3 hours for both algorithms. Note that both algorithms use the previous measured distances to smooth the current probing latencies to other nodes. Firstly considering the level of accuracy, Figure 5.10 shows the correlation results of the two algorithms for 5 updates. The correlation for both algorithms maintains at high values, with slight improvement (less than 1 percent). There is no significant difference between the two algorithms concerning these results. Figure 5.11 shows the results of relative error after the first and last updates. The level of accuracy for both algorithms improves at the end, and SEA outperforms DSA to limited extend.

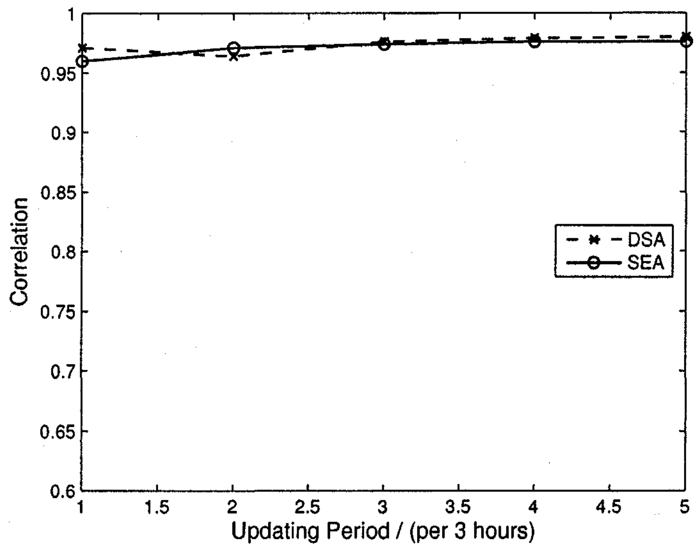


Figure 5.10: DSA vs. SEA (Long-time Running): Correlation

The performance of convergence time and message overhead show big difference between the two algorithms. Figure 5.13 and Figure 5.12 display the results of these metrics. For SEA, there is a big drop in the values of convergence time and number of messages sent out from each node starting from the second update. While for DSA, these two values maintain around the same level for all the updates. The result reveals significant advantage of SEA over DSA: less message overhead and quicker convergence during updates. Summarizing the performance of two landmark positioning algorithms, I consider SEA to be a better choice for landmark positioning.

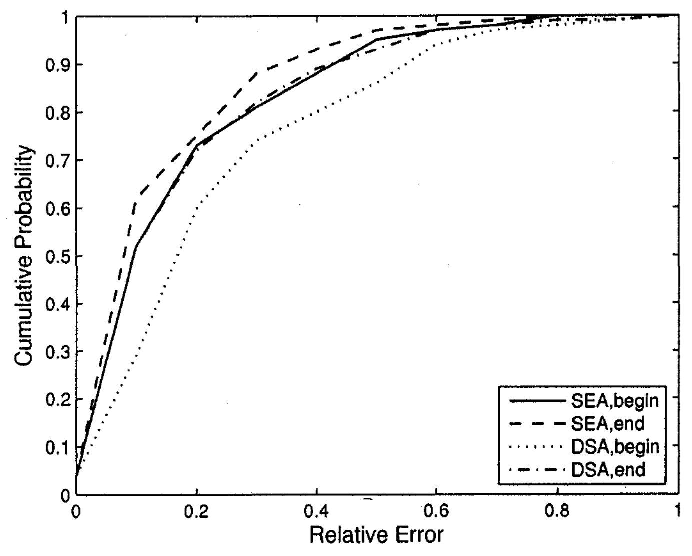


Figure 5.11: DSA vs. SEA (Long-time Running): Relative Error

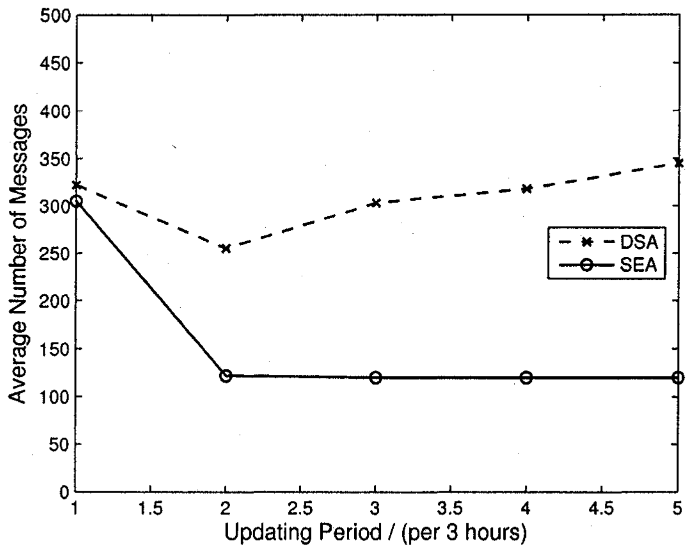


Figure 5.12: DSA vs. SEA (Long-time Running): Message Overhead

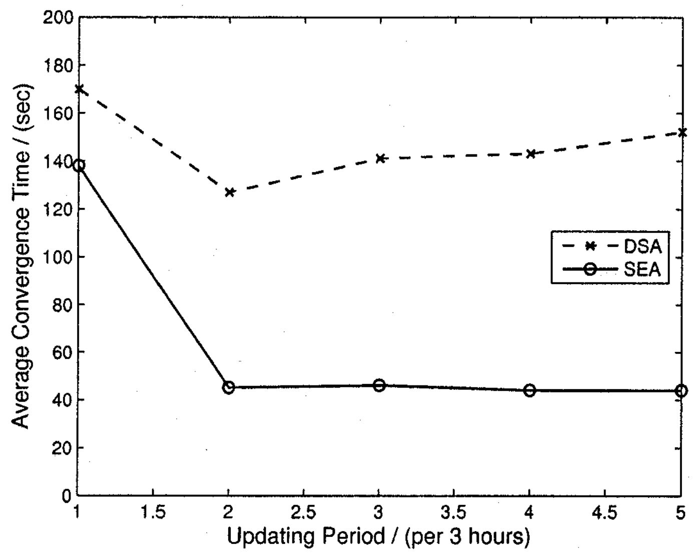


Figure 5.13: DSA vs. SEA (Long-time Running): Convergence Time

5.3.3 Ordinary Node Positioning

In this section, I mainly test the performance of ordinary node positioning under different numbers of landmark references in the system. I select 5 groups of ordinary nodes, each of which consists of around 20 nodes. I change the number of landmarks in the system to be 10, 15, 20, 25, 30, and let ordinary nodes compute their coordinates based on these different landmark frameworks. Figure 5.14 shows the average correlation results for the 5 groups of ordinary node. The level of accuracy improves as the number of landmarks in the system increases. But the improvement is not much after 20 landmarks. Figure 5.15 shows the relative error results for the group of nodes with the lowest correlation. As the group with lowest accuracy, it shows an average 50 percentile relative error 0.15 and 90 percentile relative error 0.52.

5.3.4 Cluster-based Adjustment

The experiments in this section focus on the performance of the cluster-based adjustment schema. I study the experimental results in two aspects: first, will this adjustment improve the accuracy of the whole system? This adjustment move node's location in reference to a small number of close nodes. I would like to find out if this adjustment is correct when the whole system is concerned. Second, will this adjustment move the node dramatically?

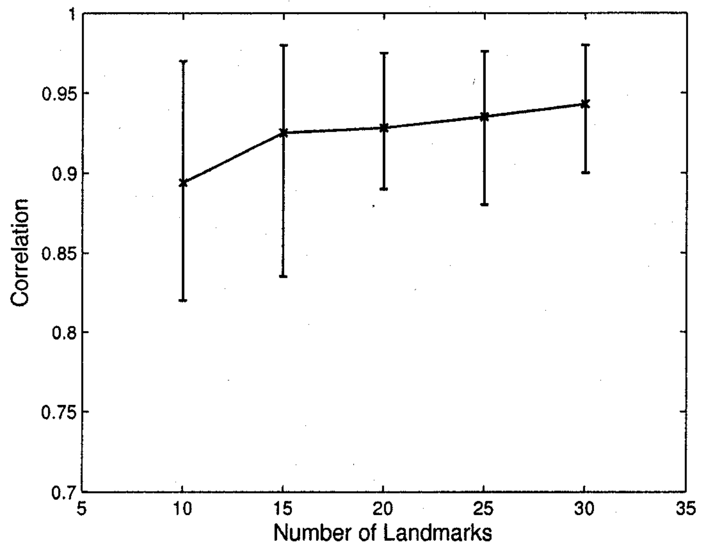


Figure 5.14: Ordinary Node Positioning: Correlation

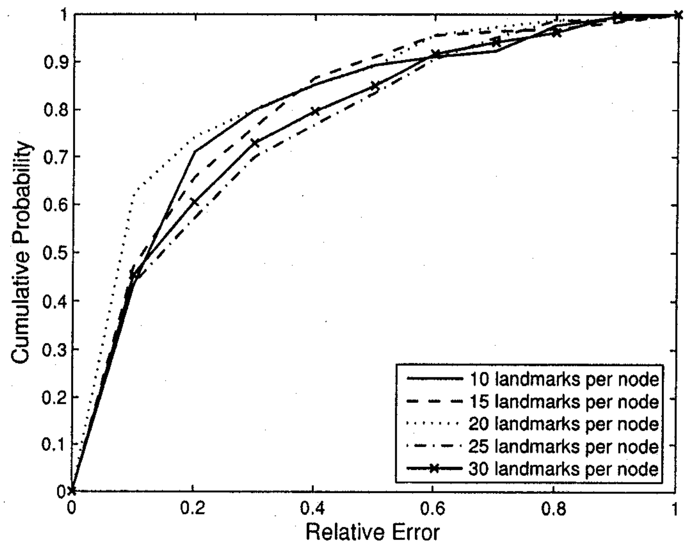


Figure 5.15: Ordinary Node Positioning: Relative Error

I run the RAN application twice on 127 nodes during 5 landmark updating periods, with and without position adjustment, to compare the results.

Figure 5.16 shows the correlation results. I can see that cluster-based adjustment does not decrease the level of accuracy for the whole system, but does not increase much either. The possible explanations can be that, only 40 percent of the 127 nodes have performed the cluster-based adjustment (as described in Section 5.2, I define 3 clusters of 24, 14, and 11 nodes respectively), so the influence may not be so significant because the other 60-percent nodes did not participate.

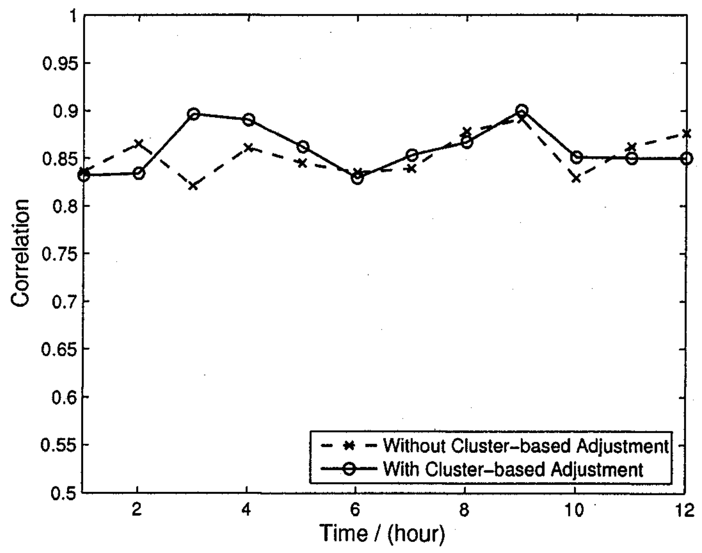


Figure 5.16: Cluster-based Adjustment: Correlation

Figure 5.17 displays the average occupancy diameter results. The system running cluster-based adjustment shows much lower movement of nodes' positions (average decrease of 8 milliseconds). And also, the adjustment does not increase nodes' movement between every two landmark updates. In a word, cluster-based adjustment can help fix nodes to their positions without decreasing the level of accuracy for the whole system. It is useful for stablizing the positioning performance under normal network variance.

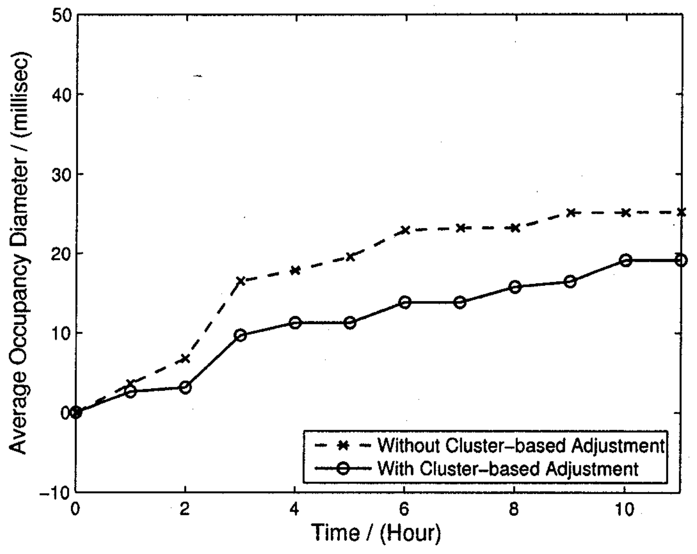


Figure 5.17: Cluster-based Adjustment: Average Occupancy Diameter

Chapter 6

Conclusion

In this thesis, I studied the problem of designing and building a network positioning mechanism in the RAN, which is the naming and discovery module of the Galaxy PCU. The network positioning scheme provides the functionality of defining network positions for the Internet machines that can be used to find proper resources through geographic locality.

To position the Internet hosts, I model the Internet delay space as a D-dimensional Cartesian coordinate space, and a host's position is denoted using coordinates in the space. I use a landmark-aided positioning method to compute host's coordinates. In this method, a basic idea is to have a small set of landmarks and first calculate their coordinates that serve as a frame of reference. Other machines can derive their coordinates in reference to this frame. I studied two algorithms to calculate landmark positions, distributed spring algorithm and spring equilibrium algorithm. Both algorithms are decentralized and simultaneously run on all the landmarks. The two algorithms apply different methodology to generate coordinates to attain the goal of minimizing inter-landmark distance prediction error. The coordinates of ordinary hosts are computed in a centralized way, using a principle similar to the distributed spring algorithm. In addition to these positioning mechanisms, I use the positions of a host's nearby peers also as references to adjust the host's position. After positioning the hosts, I utilize a multi-dimensional indexing method called space filling curve to generate location-based resource names.

Hosts running the location-based RAN distribution form a peer-to-peer overlay network. They can discover their neighbors and distribute information to others automatically. To facilitate resource discovery in the overlay, these hosts are connected in a ring configuration. The two ring neighbors of a host are the two peers with closest index numbers to it. Except for the pointers to the ring neighbors, a host may have a few random pointers pointing to resources with arbitrary indexes. The discovery message for a specific resource can be routed along the ring or directly to a matching machine.

The location-based RAN prototype is implemented using Java and XML. I set up a testbed with 127 machines spanning over the world on PlanetLab. The experimental results reveal that RAN positioning methods can maintain the level of accuracy in predicting network latencies during long-time system running. A significant improvement on the convergence time is observed for landmark positioning using spring equilibrium algorithm, in contrast to the results of distributed spring algorithm under similar experimental conditions. And also, the mechanism of adjusting host's position referring to close peers helps to stablize host's position under normal variance of network conditions.

The testing results prove the efficiency, consistency and stability of my positioning solution. Some practical issues such as adaptivity and fault tolerance are not fully studied in this thesis. Insight to these issues depends on conditioned system running or simulations.

Bibliography

[1] M. Maheswaran, B. Maniymaran, S. Asaduzzaman, and A. Mitra, "Towards a quality of service aware public computing utility," *1st IEEE NCA Workshop on Adaptive Grid Computing*, August 2004. Cambridge, Massachusetts, USA.

[2] J. Wikes, J. Mogul, and J. Suermondt, "Utilification," *11th ACM SIGOPS European Workshop*, pp. 19–22, September 2004. HP Laboratories, CA, USA.

[3] J. S. Chase, D. E. Irwin, L. E. Grit, J. D. Moore, and S. E. Sprenkle, "Dynamic virtual cluster in a grid site manager," *12th IEEE International Symposium on High Performance Distributed Computing*, p. 90, 2003.

[4] C. Sapuntzakis and M. S. Lam, "Virtual appliances in the collective: A road to hassle-free computing," *9th Workshop on Hot Topics in Operating Systems*, 2003. Stanford University, CA, USA.

[5] Hewlett-Packard, "Transforming data center economics," Tech. Rep. part 5982-3291EN, HP Utility Data Center, March 2004.

[6] IBM, "Utility computing," *IBM Systems Journal special issue*, no. 43(1), 2004.

[7] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, July 1998.

[8] I. Foster, C. Kesselman, and S. Tuecke, "The anatomy of the grid: Enabling scalable virtual organizations," *International J. Supercomputer Applications*, no. 15(3), 2001.

[9] VFrame Server Virtualization Software. http://www.topspin.com/solutions/vframe.html.

[10] FlexFrame for mySAP Business Suite. http://www.netapp.com/ftp/FlexFrame-Brochure.pdf.

[11] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," *Middleware*, 2001.

[12] S. Saroiu and S. G. P. K. Gummadi, "A measurement study of peer-to-peer file sharing systems," *MMCN02*, January 2002.

[13] C. Shirky, K. Truelove, R. Dornfest, and L. Gonze, *2001 P2P Networking Overview: The Emergent P2P Platform of Presence, Identity, and Edge Resources.* O'Reilly, 1 ed., October 2001.

[14] B. Maniymaran and M. Maheswaran, "On the benefits of profile-based naming for large network computing systems," *16th International Conference on Parallel and Distributed Computing and Systems (PDCS 2004)*, November 2004.

[15] S.Ratnasamy, P.Francis, M.Handley, R.Karp, and S.Shenker, "A scalable content-addressable network," *ACM SiGCOMM'01*, Aug. 2001.

[16] I.Stoica, R.Morris, D.Karger, F.Kaashoek, and H.Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM SiGCOMM'01*, Aug. 2001.

[17] F. Azzedin and M. Maheswaran, "A trust brokering system and its application to resource management in public-resource grids," *IPDPS*, 2004.

[18] S. Banerjee, Z. Xu, S.-J. Lee, and C. Tang, "Service multicast for media distribution networks," *IEEE Workshop on Internet Applications (WIAPP)*, June 2003.

[19] A.-C. Huang and P. Steenkiste, "Network-sensitive service discovery," *Proceedings of USENIX - USITS*, 2003.

[20] Z.Xu, C. Tang, S. Banerjee, and S.-J. Lee, "Receiver initiated justin-time tree adaptation for rich media distribution," *Proceedings of NOSSDAV*, June 2003.

[21] Y.Chu, S.Rao, and H.Zhang, "A case for end system multicast," *ACM Sigmetrics*, June 2000.

[22] J.Liebeherr, M.Nahas, and W.Si, "Application-layer multicast with delaunay triangulations," Tech. Rep. Tech. Rep, University of Virginia, Nov. 2001.

[23] K.-W. Lee, B.-J. Ko, and S. B. Calo, "Adaptive server selection for large scale interactive online games.," *Computer Networks*, vol. 49, no. 1, pp. 84–102, 2005.

[24] http://www.planet-lab.org.

[25] S. Saroiu, K. P. Gummadi, and S. D. Gribble, *Multimedia Systems: Measuring and analyzing the characteristics of Napster and Gnutella hosts.* Springer-Verlag, 2003.

[26] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. Gryniewiez, and Y. Jin, "An architecture for a global internet host distance estimation service," *IEEE INFOCOM '99*, March 1999.

[27] T. Ng and H.Zhang, "Predicting internet network distance with coordinates-based approaches," *IEEE Info-com02*, June 2002.

[28] J. A. Nelder and R. Mead, "A simplex method for function minimization," *Computer Journal*, vol. 7, pp. 308–313, 1965.

[29] T. Ng and H.Zhang, "A network positioning system for the internet," *USENIX Conference*, June 2004.

[30] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris, "Practical, distributed network coordinates," *SIGCOMM Comput. Commun. Rev.*, June 2004.

[31] F.Dabek, R.Cox, F.Kaashoek, and R.Morris, "Vivaldi: A decentralized network coordinate system," *Sigcomm'04*, August 2004.

[32] M. Costa, M. Castro, A. Rowstron, and P. Key, "Pic: Practical internet coordinates for distance estimation," Tech. Rep. Technical Report MSR-TR-2003-53, Microsoft Research, September 2003.

[33] L.Lehman and S.Lerman, "Pcoord: Network position estimation using peer-to-peer measurements," *Intl. Symposium on Network Computing and Applications*, August 2004.

[34] R.Cox and F.Dabek, "Learning euclidean coordinates for internet hosts," *http://pdos.lcs.mit.edu/ rsc/6867.pdf*, December 2002.

[35] B. Maniymaran, *Resource Addressable Network: An Adaptive Peer-to-Peer Discovery Substrate for In ternet-Scale Service Platforms*. Ph.d. proposal report, McGill Unviersity, July 2005.

[36] G. Mavko, T. Mukerji, and J. Dvorkin, *The rock physics handbook : tools for seismic analysis in porous*. media Cambridge ; New York : Cambridge University Press, July 1998.

[37] J. L. Buchanan and P. R. Turner, *Numerical methods and analysis*. New York : McGraw-Hill, 1992.

[38] H.Sagan, *Space-filling curves*. New York: Springer-Verlag, 1994.

[39] A.R.Butz, "Alternative algorithm for hilbert's space filling curve," *IEEE Transactions on Computers*, April 1971.

[40] S.Shekhar and S.Chawla, *Spatial Databases: A Tour, 1 Edition*. 2002.

[41] W. Maalouf and H. Mirza, "Ran report," 2004.

[42] Intel Research Berkeley. http://www.theether.org/pssh/.

[43] A. Acharya and J.Saltz, "A study of internet round-trip delay," Tech. Rep. Technical Report CS-TR-3736, University of Maryland,College Park, 1996.