Self-Managed Deployment in a Distributed Environment via Utility Functions

Debzani Deb
Montana State University, Bozeman, MT, USA
debzani@cs.montana.edu

Michael J. Oudshoorn
The University of Texas at Brownsville, TX, USA
michael.oudshoorn@utb.edu

John Paxton
Montana State University, Bozeman, MT, USA
paxton@cs.montana.edu

Abstract

This paper proposes algorithms and mechanisms for achieving self-managed deployment of computationally intensive scientific and engineering applications in highly dynamic and large-scale distributed environment. The primary focus is on the modeling of the application and underlying architecture into a common abstraction and on the incorporations of autonomic features to those abstractions to achieve self-managed deployment. To represent the underlying heterogeneous infrastructure, a hierarchical (tree) model of distributed resources has been adopted that offers self-organization of distributed nodes in a utility-aware way. To accomplish the self-adaptive deployment, a utility-function has been formulated that governs both the initial deployment of an application and maintains the optimality during execution despite the dynamism and uncertainty associated with the application and the networked environment. In our approach, the deployment decisions are made solely based on locally available information and without costly global communication or synchronization. The self-management is therefore decentralized to provide better adaptability, scalability and robustness.

1. Introduction

Many scientific fields, such as genomics, astrophysics, geophysics, computational neuroscience or bioinformatics require massive computational power and resources and can benefit from a large-scale integrated infrastructure, formed by harnessing the spare compute cycles of distributed computation and communication resources. Typically these applications composed of a large number of distributed components and it is important to deploy them in the underlying network in a way that meets the computational power and network bandwidth requirements of those components and their interactions. However satisfying these requirements in a large-scale, heterogeneous, and highly dynamic distributed environment is a significant challenge. As systems and application grow in scale and complexity, attaining the desired level of performance in this uncertain environment using current approaches based on global knowledge, centralized scheduling and manual reallocation becomes infeasible. Therefore, self-managed deployment is paramount in order to lower operation costs, to allow developers to largely ignore complex distribution issues, to manage system complexities and to maximize overall utilization of the system.

This paper proposes algorithms and mechanisms for achieving self-managed deployment of computationally intensive scientific and engineering applications in highly dynamic and unpredictable distributed environment. The main focus of this paper is to model the application and the underlying architecture into a common abstraction and to incorporate autonomic features [1] to those abstractions to achieve self-managed deployment. To model the underlying heterogeneous infrastructure, we developed techniques that allow the distributed resources to self-organize in a utility-aware way while assuming minimal knowledge about the system. To accomplish self-managed deployment of the application components to the network nodes, we designed a scalable and adaptive deployment algorithm that is governed by a utility function [2]. The utility function, which returns the overall system's utility based on different application and system level attributes, governs the initial deployment of the application and maintains the optimality during execution despite the dynamism and uncertainty associated with the application and the networked environment. The self-management techniques described in this paper are decentralized and assume minimal knowledge about the environment to provide better adaptability, scalability and robustness.

Fully automating the organization and optimization of a large distributed system is a staggering challenge and there are numerous research groups working toward this goal. Approaches described in [3,4] targets the development of new autonomic applications to realize the desired benefits of self-management in a distributed environment. In their prototype implementation, Unity [5] achieves self-management via interconnections amongst a number of autonomous agents, however assumes global knowledge in order to optimally allocate the resources in

the system. Astrolabe [6] operates by creating a virtual system-wide hierarchical database of the state of a collection of distributed resources, which evolves as the underlying information changes. The AutoFlow [7] project aims to develop a self-adaptive middleware and utilizes a hierarchical organization of underlying resources clustered according to various system attributes for deployment.

The rest of the paper is organized as follows. Section 2 details the design and implementation of different aspects of application deployment process. Section 3 presents the experimental evaluation of the proposed deployment and section 4 concludes the paper.

2. Self-Managed Deployment

As the application components within an application execute with different constraints and requirements, they should be mapped to appropriate hardware resources in the distributed environment so that their constraints are satisfied and they provide the desired level of performance. Mapping between these resource requirements and the specific resources that are used to host the application is not straightforward.

A three step process is designed to perform this mapping as shown in Figure 1. In the first step of the mapping, an application model is extracted that represent an application in terms of its components and their internal dependencies along with the estimated resource requirements of the components and their links. The next step involves constructing a model of the underlying network by organizing them according to network proximity (considering latency, bandwidth, etc). The third and final step allocates a specific set of resources to each application with respect to the resources required by the application components and the resources available in the system. The goal of the mapping is to maximize the system's overall utility based on certain policies, priorities, user-defined constraints and environmental conditions. The important aspects of this deployment process are detailed in the following few sections.

2.1. The Application Model

In this paper, an application is modeled as a graph consisting of application components and the interactions among them. Analyzing and representing software in terms of its components and their internal dependencies is important in order to provide the self-managing capabilities because this is actually the system's view of the run-time structure of a program. Well structured graph-based modeling of an application also makes it easier to incorporate autonomic features into each of the application components.

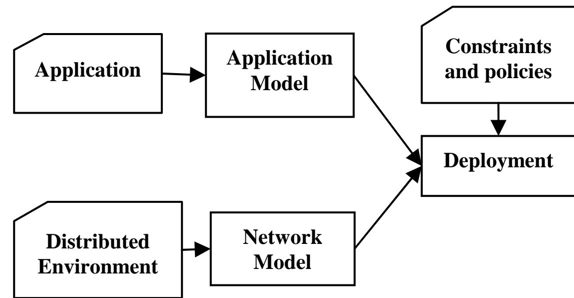

Figure 1. Application deployment process

An application is represented as a node-weighted, edge-weighted directed graph $G = (V, E, w_g, c_g)$, where each vertex $v \in V$ represents an application component and the edge $(u,v) \in E$ resembles the communication from component u to component v. The computational weight of a vertex v is $w_g(v)$ and represents the amount of computation that takes place at component v and the communication weight $c_g(u,v)$ captures the amount of communication (volume of data transferred) between vertices u and v. When deployed across a distributed heterogeneous environment, these weights along with various system characteristics, such as the processing speed of a resource and the communication latency between resources, determine the actual computation and communication cost. The detailed process of the extraction of the graph form of an application is out of the scope of this paper. However, Reference [8] provides a detailed description of our static analysis based application graph construction approach.

2.2. The Network Model

In this research, the target environment for the deployment of the application is a distributed environment consisting of a non-dedicated heterogeneous and distributed collection of nodes connected by a network. To organize the computation around this heterogeneous and distributed pool of resources, traditional approaches rely on the assumption that sufficiently detailed and up-to-date knowledge of the underlying resources is available to a central entity. While this approach results in the optimized utilization of the resources, it does not scale to large numbers of nodes. Maintaining a global view of a large-scale distributed environment becomes prohibitively expensive, even impossible at a certain stage, considering the unprecedented number of nodes and the unpredictability associated with a large-scale and dynamic computing system.

We propose a different approach that addresses the above problems and allows the heterogeneous pool of resources to self-organize in a structure that facilitates

their effective use. Our aim is to organize the distributed resources in a structure such that nodes that are *closer* to each other in the structure are also *closer* to each other considering network distance (latency, bandwidth, etc.). Once structured in this way, it is possible to detect higher utility paths locally that correspond to low latency and high bandwidth between network nodes. As a result of that, the deployment of the application graph can be performed in a utility-aware way, without having full knowledge about the underlying resources and without calculating the utility between all pairs of network nodes.

The proposed organization is obtained by modeling the target distributed environment as a tree in which the nodes correspond to compute resources, edges correspond to network connections and execution starts at the root. More specifically, a tree structured overlay network [9,10] is used to model the underlying resources, which is built on the fly on top of the existing network topology. Having such a structure, a simple but effective autonomic deployment algorithm is used to organize computation on the available nodes. Here is an intuitive description of the algorithm.

> *Each node in the tree either completely executes the tasks assigned to it or divides the computation (if it is too large to execute by itself within a reasonable amount of time) and propagates the parts down the hierarchy to its best child's subtree.*

The important aspect of our design is the emergence of the tree topology, which structures the distributed nodes, in a utility-aware way while assuming minimal knowledge about the environment. Each parent monitors only a limited number of nodes and the deployment decision is made based on this locally available monitored data, therefore the design is appropriate for dynamic and large-scale system. Also this model allows us to limit the utility evaluation within a subtree performed by the parent of that subtree, instead of performing the costly utility evaluation globally to determine the highest utility node. Figure 2(a) shows a small computing environment where resources are distributed in three domains and Figure 2(b) illustrates a tree overlay network that is built on top of this physical topology.

Formally, the entire network is represented as a weighted tree $T = (N, L, w_t, c_t)$, where N represents the set of computational nodes and L represents network links among them. The weights attached to the nodes and edges represent the associated computation and communication costs. The computational weight $w_t(n)$ indicates the cost associated with each unit of computation at node n. The communication weight $c_t(m,n)$ models the cost associated with each unit of communication of the link between node m and n considering both bandwidth and latency.

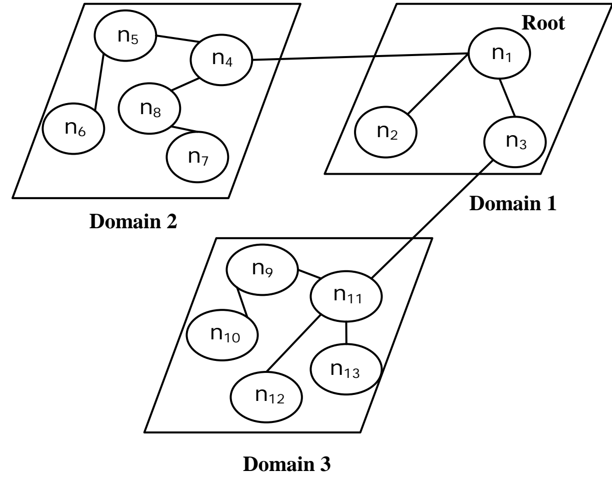

Figure 2(a). Sample distributed environment.

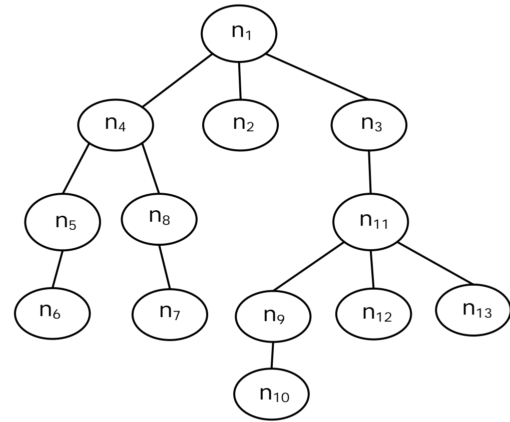

Figure 2(b). Overlay Tree.

When two nodes are not connected directly, their communication weight is the sum of the link weights on the path via their predecessors or successors. Therefore, larger values of node and edge weights translate to slower nodes and slower communication respectively.

To construct an overlay tree, each node is assumed to have a *children list* signifying the URLs of its neighbors that have direct connection with it. The problem of how to generate this list is out of scope of this research however can be addressed by using several tools [11,12]. Once a user starts an application in his/her machine, the graph representation is extracted from the application code. The initiator node then delegates some of the graph vertices (application components) to the best utility nodes considering all the nodes listed in its children list. The delegated nodes again spread the computation in this manner. The topology of the resulting overlay network thus becomes a tree with the originating machine at the root node.

2.3. The Utility Function

In this research, both the initial placement of application components and their reconfigurations are governed by utilizing utility functions. Several applications and environment specific attributes are combined in a single utility function. This multi-attribute function returns a scalar value signifying system's overall utility for each possible state of a system and the goal becomes to select a state that maximizes system's overall utility. During execution, resource allocation and other operating conditions may change; the corresponding change in the overall utility of the system can be calculated by this utility function and decisions can be taken toward maximizing this value. As computing environments are becoming increasingly large, distributed, complex and dynamic in nature, the optimal actions are likely to evolve over time and a utility function that continuously computes the most desired state is expected to be more suitable in such cases.

In general, our utility function considers the following application, environment and user specific high-level policies:

1. While mapping partitions containing a large number of application components in the tree network, node that leads to a wider subtree (higher degree of connectivity) should be preferred as higher degree allows more directions for partition growth.
2. Faster and less busy nodes should be favored over slower and overloaded nodes when assigning components to resources.
3. Nodes with faster communication links should be preferred over nodes with slower communication links when dealing with communication intensive components.
4. High priority applications should be preferred during deployment over low priority jobs.

2.4. Initial Deployment

Once both the application and underlying resources have been modeled, the deployment problem reduces to the mapping of different application components and their interconnections to different nodes in the target environment and network links among them so that all requirements and constraints are satisfied and system's overall utility is maximized. The assumption is that the application can be submitted to any node, which acts as the root or starting point of the application. Also the application may end its execution either at the root node or at one or more clients at different destination nodes.

When the application graph G is submitted to the root node of the tree network, the root then decides which application components to execute itself and which components to forward to its child's sub-tree so that the overall mapping results the highest utility. The child, who has been delegated a set of components again deploys them in the same way to its subtrees. For effective delegation of components at a particular node having |P| children, graph coarsening techniques [13] is exploited to collapse several application components into a single partition, so that ≤ |P| partitions are generated at that stage. The coarsened graph is projected back to the original or to a more refined graph once it is delegated to a child.

In the above approach, each parent selects the highest utility child to delegate a particular partition (set of components). Finding the highest utility child to delegate a partition to means finding the highest utility mapping M of the edges (v_j,v_k) where $v_j \in V_r$ (represents the set of components that the parent decided to execute itself) and $v_k \in V_s$ (represents the set of components that belong to a partition that a parent decided to delegate). More formally, a mapping needs to be produced, which assigns each $v_k \in V_s$ to a $n_q \in N$ in a way such that the network node n_q is capable of fulfilling the requirements and constraints of application node v_k and the edge (v_j,v_k) is mapped to the highest utility link considering all children available at that stage for delegation. The utility of an edge (v_j,v_k) is represented as $U(v_j,v_k)$, and returns the utility achieved due to the mapping of the edge (v_j,v_k) on certain network link. More specifically, the utility of an edge (v_j,v_k), while mapped to the network link (n_p,n_q), where n_p represents the parent in the tree-shaped network where v_j is already mapped and n_q represents a potential child for delegating application component v_k, is calculated by using the following function:

$$U(v_j,v_k) = \frac{d(n_q)}{f_1(w_g(v_k) \times w_t(n_q)) + f_2(w_g(v_j,v_k) \times w_t(n_p,n_q))}$$

where $d(n_q)$ represents the number of children of network tree node n_q, function f_1 models the cost of processing vertex v_k in node n_q and f_2 models the cost resulting from mapping edge (v_j,v_k) to link (n_p,n_q).

The utility model in the above scenario is the "highest-degree child with the fastest computation capability and fastest communication link is more suitable for utility". To ensure that the application graph partitions with the largest number of components are delegated to the highest degree child, candidate partitions are sorted according to their sizes and then deployed according to that order. In the case of simultaneous scheduling of multiple applications with different priorities, the system needs to guarantee that higher priority applications execute before applications with lower priority. To achieve this, applications are ordered according to their priorities and then mapped following that order. The

overall utility of an application graph G with priority p due to deployment M is then calculated as:

$$U(G,M) = p \times \sum_{(v_j,v_k)\in E} U(v_j, v_k)$$

Therefore, at the level of an individual application the problem of self-configuration becomes the problem of finding highest utility mapping M between edges E in the application graph and the Links L of the network graph.

2.5. Self-Optimization

After initial placement, the environment may change and as a result the utility may drop. Therefore it is necessary to monitor the utility and trigger reconfiguration as required. Reconfiguration is triggered in response to a variety of events such as changes in network delays, changes in available bandwidth, changes in available processing capability, etc. Some business specific events may also trigger reconfiguration such as the arrival of a higher priority job, etc. Reconfiguration within a subtree is expected to be a less expensive process because of the way the underlying network is modeled. Each parent node periodically measures the workload at each child and its bandwidth to the child and consequently changes computational and communication weights attached to that child. By incorporating this monitored information into the utility function, the parent then observe the change in utility due to the changes in network and compute nodes, and therefore reconfiguration is initiated autonomously. Reconfiguration is costly and disruptive, therefore, it is not feasible to initiate reconfiguration unless it is productive. This research plans to trigger reconfiguration whenever the utility drops more than a certain threshold (user specified or system generated by comparing the utility during initial deployment).

3. Experimental Evaluation

We evaluate the performance of the self-managed deployment using a simulation study. Our experiments were performed in a dual, quad-core Xeon processor with 16GB of RAM. We used system's overall business utility as the performance metric in all our experiments.

3.1 Simulation Setup

We used GT-ITM internetwork topology generator [14] to generate a sample large-scale, heterogeneous computing environment for evaluating our self-deployment algorithm. We choose the Transit-Stub model that correlates well with the structure of the Internet, including hierarchy and locality. Table 1 lists the relevant

Table 1: Network model parameters used in this study

The number of Transit Nodes	4
The number of stub nodes/transit node	32
Number of total network nodes	132
Number of total network links	1986
Stub-stub bandwidth	100Mbps
Transit-transit and transit-stub bandwidth	500Mbps
Node's processing weight	[20-80]

parameters of the network topology used in this study. To generate traffic that simulates real world workload and bandwidth consumption in a shared environment, we used the ns-2 simulation package [15]. The traffic generator script cbrgen.tcl is available under ~ns/indep-utils/cmu-scen-gen and was used to create 1000 CBR traffic connections between network nodes. The simulation was then carried out for 2500 seconds and link delays (amount of time required for a packet to traverse a link considering both bandwidth and propagation delay) are measured between the directly connected nodes in the presence of the random traffic in 10 second interval period. Based on these snapshots, we then determined the communication weights of the network links in the presence of dynamic traffic.

We ran our tree construction algorithm that creates a tree overlay on top of the abovementioned network topology with the application originating machine at the root node. To create the children list, at first we went through all network links and make a list for each node $n \in N$, that n has direct connections with. Our tree construction algorithm then finalizes the children list for each network node n, starting from the root node, ensuring that adding a node to n's children list does not create a cycle.

3.2 Experiments and Results

We designed experiments that compare the utility and cost of a deployed application graph using optimal schemes based on the original network topology and global knowledge about the system as opposed to our approach that uses the self-organized tree and decentralized deployment decisions based on minimal amount of locally available knowledge. In the optimal scheme, the assumption is that a central node monitors every computational and communication resources in the system and based on this global knowledge makes optimal deployment decision. However, in this approach the central node becomes a bottleneck with a large number of communications arising from constant monitoring of all the resources in the system. Even if it is possible to gather up-to-date information about all the resources in a central node, finding optimal deployment

means trying every possible mapping of the application components to the network resources and selecting the one that produces optimal results, which grows exponentially with the number of nodes in the network and the number of vertices in the application graph.

Because of its exponential growth, the above mentioned optimal scheme becomes very costly even after considering a small number of application graph vertices. Therefore we developed another semi-optimal scheme that assumes global knowledge but instead of trying every possible mapping it uses heuristics to limit the number of cases to evaluate. For both schemes, we applied Dijkstra's All Pair Shortest Path algorithm at the central node to calculate the communication weights between every pair of network nodes. We also assumed one-to-one mapping of the graph vertices to the network nodes in all three cases. The results are presented in Figures 3 and Table 2. Figure 3 illustrates that the utility results from our approach is close to what is achieved by

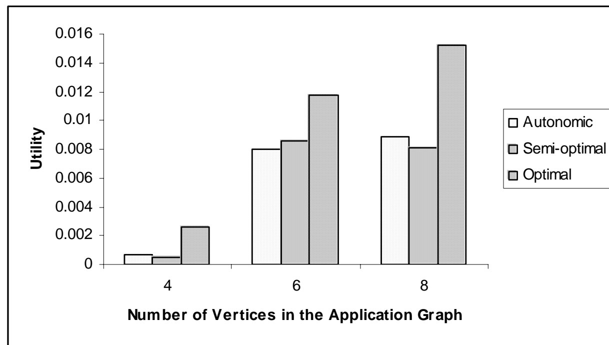

Figure 3. Utility Comparison

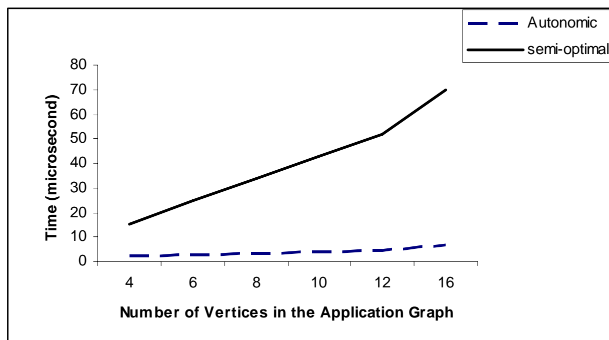

Figure 4. Scalability

Table 2. Execution Time Comparison

# of vertices	Optimal	Semi-optimal	Autonomic
4	8712 µs	37 µs	8 µs
6	15.68 sec	72 µs	11 µs
8	1 hour and 39 minute	134 µs	72 µs

using optimal approach. It is also evident that, in some cases the semi-optimal approach produces less utility than our approach.

The reason for that is that since semi-optimal deployment takes a greedy approach based on utilities between each pair of nodes in the topology, the highest utility node at a certain stage may already have been delegated in some former stage. Table 2 reveals the cost associated with each approach and as expected the optimal approach incurs huge cost with the increasing number of vertices in the application graph and soon become inapplicable. To evaluate the scalability of our approach, we experimented the time taken by our approach to calculate the initial deployment for increasing number of application vertices and compared them with the times needed by the semi-optimal approach. The results are presented in Figure 4 and show that the cost incurred by our approach is minimal, therefore is well suited for larger applications.

4. Conclusion

In this paper, we have developed techniques that enable scalable and efficient deployment of user applications in a highly dynamic and large-scale distributed environment. The approach is to construct an application model, represented as a graph of application components and their interactions and then deploy that graph across the underlying distributed resources self-organized as a utility-aware tree. A suitable utility function is derived that controls both initial deployment and reconfiguration ensuring that system's overall utility is maximized while certain policies and constraints are satisfied. The main goal of our experimental study was to analyze the tradeoff between optimality and the execution time of our autonomic deployment. The results of our experiments show that the utility achieved by our approach is comparable with optimal utility while the cost is far less than the optimal approach. Our approach for self-configuration is therefore scalable, robust and more suitable for larger networks and applications. In future, we like to conduct experiments to evaluate our self-optimization approach that dynamically reconfigure the application graph based on the changes in the network.

References

[1] IBM Research. *Autonomic Computing.* http://www.research.ibm.com/autonomic.

[2] W.E. Walsh, G. Tesauro, J.O. Kephart, R. Das, "Utility functions in autonomic systems", *1st International Conference on Autonomic Computing (ICAC),* 2004.

[3] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang and S. Hariri, "AutoMate: Enabling Autonomic Grid Applications", *Cluster Computing: The Journal of*

Networks, Software Tools, and Applications, Special Issue on Autonomic Computing, Kluwer Academic Publishers, Vol. 9, No. 1, 2006.

[4] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao, "AUTONOMIA: An Autonomic Computing Environment", *Proc. of the 2003 IEEE International Performance, Computing, and Communication Conference*, 2003.

[5] D. M. Chess, A. Segal, I. Whalley and S. R. White, "Unity: Experiences with a Prototype Autonomic Computing System", *1st International. Conference. on Autonomic Computing (ICAC)*, 2004.

[6] R. V. Renesse , K. P. Birman , W. Vogels. "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining", *ACM Transactions on Computer Systems (TOCS)*, Vol.21 No.2, 2003.

[7] K. Schwan et al. "Autoflow: Autonomic information flows for critical information systems", Autonomic Computing: Concepts, Infrastructure, and Applications, CRC Press, 2006.

[8] D. Deb, M.M. Fuad, M.J. Oudshoorn, "Towards Autonomic Distribution of Existing Object Oriented Programs", *International Conference on Autonomic and Autonomous Systems (ICAS)*, 2006.

[9] O. Beaumont, A. Legrand, Y. Robert, L. Carter, J. Ferrante, "Bandwidth-Centric Allocation of Independent Tasks on Heterogeneous Platforms", *International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[10] B. Kreaseck, L. Carter, H. Casanova, and J. Ferrante, "Autonomous protocols for bandwidth-centric scheduling of independent-task applications", *International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[11] M. Ripeanu; I. Foster and A. Iamnitchi, "Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design", *IEEE Internet Computing*, Vol. 6, No. 1, 2002.

[12] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A Scalable Content-Addressable Network", *In Proceedings of ACM SIGCOMM* 2001.

[13] G. Karypis and V. Kumar, "Multilevel k-way Partitioning Scheme for Irregular Graphs", *Journal of Parallel and Distributed Computing,* vol. 48, 1998, pp. 86-129.

[14] GT-ITM: Georgia Tech Internetwork Topology Models. http://www.cc.gatech.edu/projects/gtitm.

[15] The Network Simulator ns-2. http://www.isi.edu/nsnam/ns.