ADE: Utility Driven Self-management in a Networked Environment

Debzani Deb
Montana State University, Bozeman, USA
E-mail: debzani@cs.montana.edu

M. Muztaba Fuad
Winston-Salem State University, Winston-Salem, USA
E-mail: fuadmo@wssu.edu

Michael J. Oudshoorn
The University of Texas at Brownsville, Brownsville, USA
E-mail: michael.oudshoorn@utb.edu

Abstract – **ADE, autonomic distributed environment, is a system which engages autonomic elements to automatically take an existing centralized application and distribute it across available resources. The autonomic elements provide self-management to handle the complexities associated with distribution, configuration, coordination and efficient execution of program components. The proposed approach models a centralized application in terms of an application graph consisting application components and then deploys the application components across the underlying utility-aware hierarchically organized distributed resources so that all constraints and requirements are satisfied and the system's overall utility is maximized. Then, based on the observations obtained by the monitoring of the system resources, ADE redeploys the application graph to maintain maximized system utilization in spite of the dynamism and uncertainty involved in the system. One important aspect of ADE is that, the deployment decisions can be made based solely on locally available information and without costly global communication or synchronization. The proposed model is therefore decentralized and adaptive.**

Index Terms – **Automatic partitioning, autonomic computing, self-management, self-optimization, utility function.**

I. INTRODUCTION

The growth of the Internet, along with the proliferation of powerful workstations and high speed networks as low-cost commodity components, is revolutionizing the way scientists and engineers approach their computational problems. With these new technologies, it is possible to aggregate large numbers of independent computing and communication resources with diverse capacities into a large-scale integrated system. Many scientific fields, such as genomics, phylogenetics, astrophysics, geophysics, computational neuroscience or bioinformatics require massive computational power and resources and can benefit from such an integrated infrastructure.

In most corporations, research institutes or universities, there are significant numbers of computing resources underutilized at various times. By harnessing the computing power and storage of these idle or underutilized resources, a large-scale computing environment with substantial power and capacity can be formed. With such an infrastructure, it is possible to solve computationally intensive problems efficiently in a cost effective manner as opposed to replacing these systems with expensive computational resources such as supercomputers. Having such a large-scale system, one can effectively partition an existing centralized application in terms of communicating components and distribute those components among the available resources in a manner which results in efficient execution of user program and maximized resource utilization.

However, there are many challenging aspects associated with effectively partitioning large-scale applications into several components as well as the mapping and scheduling of those components over the heterogeneous resources across the system. Firstly, the programmer wishing to execute such an application may not have necessary skills to rewrite the application to achieve effective partitioning and distribution across the network. To transform a regular, centralized application into a distributed one, the programmer needs to perform a large number of changes and most of them require thorough knowledge of both the application structure and the underlying architecture where the applica-

Based on "On Utility Driven Deployment in a Distributed Environment" by D. Deb and M. J. Oudshoorn which appeared in the proceedings of the Fourth IEEE Workshop on Engineering of Autonomic Systems (EASe 2007), Tucson, Arizona, USA, March 2007. © 2007 IEEE.

tion is going to be deployed. Secondly, ensuring maximum utilization requires mechanisms to estimate the application components computation and communication needs and their interdependencies so that an efficient mapping of components to resources can be achieved and the mapping's communication cost is minimized. Thirdly, application behavior is highly dynamic and a different distribution configuration may be appropriate in different phases of the execution of an application. Therefore, application components should be easy to migrate at runtime to enhance locality and to minimize communication cost. Finally, the application components should be tailored to dynamically respond to their environment by expanding their functionality or enhancing their performance as the underlying infrastructure varies over time. Each of these factors introduces additional complexities. In order to deal with them the system must be adaptive and dynamic in nature.

It would be tremendously useful to have a system that can automatically transform an existing application to a distributed one without the programmer being concerned about distribution and management issues, and which can deploy the distributed application across a large-scale integrated infrastructure efficiently. The complexity and cost associated with the management of such an infrastructure and with the manual transformation of applications to be deployed on that infrastructure is enormous. Therefore, automation is paramount 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 research envisions such an automatic system as an autonomic computing challenge [1]. In this paper, a self-managing distributed system that incorporates autonomic entities to handle the complexities associated with distribution, configuration, coordination and efficient execution of program components is described. The system is referred as ADE (Autonomic Distributed Environment) [3-8]. The specific objectives that ADE tries to pursue are:

1. To automatically identify the communicating application components, and their dependencies, within a centralized application and to deploy them to best take advantage of the underlying distributed computing resources.
2. To enable distributed resources to self-organize and self-manage.
3. To develop techniques that self-optimize the program execution and the use of distributed computational resources in order to maximize the system's overall business utility rather than focusing on single metrics such as minimizing execution time or maximizing throughput or utilizing resources in best possible way.

ADE's approach is to construct an application model for a centralized application, represented as an application graph and then deploy the application components across the underlying hierarchically organized distributed resources with little or no human intervention. Then, based on the observations obtained by the monitoring of the system resources, ADE automatically migrates application components, reallocates resources and redeploys the application graph. With respect to the self-optimizing criteria, ADE maximizes a specific utility function [2] that returns a measure of the overall system utility based on executing application's requirements, system's operating conditions as well as some user policies, priorities and constraints. 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.

This paper provides an overview of ADE [3-8], but focuses primarily on two aspects of self-management: self-configuration and self-optimization. The other two aspects of autonomic computing defined in [1] are the focus of our earlier works [5-7]. Ref. [4] provides a detailed description of the static analysis and application graph construction approach taken by ADE. This paper specifically focuses 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 [9,10] of distributed resources has been adopted that offers self-organization of distributed nodes in a utility-aware way. To accomplish the self-optimization, 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. Moreover, self-management is decentralized to provide adaptability, scalability and robustness.

A preliminary version of this paper [8] describes early concepts. However, this paper details the design and architecture of ADE and elaborates how the main focuses of this paper such as self-configuration and self-optimization fits in the overall flow of operation. This paper also includes a detailed and more formal description of the task of configuring and optimizing deployed application graph on the underlying distributed resources while maximizing the utility from those resources. The remainder of the paper is organized as follows. Section II presents an overview of ADE along with the different steps associated with application deployment process. Section III and IV describes the graph and tree representation of the application and the underlying networked environment respectively. Section V illustrates the formulation of the utility function followed by the detailed discussion of initial deployment and optimization. Section VI presents experimental evaluation of the proposed deployment with the help of an illustrative example. Section VII discusses the related researches and section VIII concludes the paper.

II. Autonomic Distributed Environment (ADE)

ADE aims to achieve self-management of a distributed system via interconnections among autonomic elements across the system. ADE targets existing Java programs, consisting of independent or communicating objects as components, and automatically generates a self-managed distributed version of that program. Based on the cross-platform Java technology, ADE is expected to support all major contemporary platforms and handle heterogeneous issues successfully. The availability of the source code can not always be assumed, so the proposed system performs analysis and transformations at the byte-code level. However, applications for which source code is available can be transformed to byte-code and can exploit the benefits offered by this research.

A. Flow of Operation

Fig. 1 shows ADE's overall flow of operation. At first, a code analyzer statically inspects the user supplied byte code to derive an object interaction graph. Based on this graph, the partitioner then generates several partitions (consists of a single object or grasps of objects) along with the distribution policies and deploys those partitions to a set of available resources. Deployment decisions are based on several criteria such as the resource (CPU, memory, communication bandwidth etc.) requirement of the objects and their interactions, various system information collected via monitoring services such as resource availability, workload, usage pattern etc. or any user supplied policy.

During deployment, an autonomic transformer injects the distribution and self-management primitives to the partitions according to the system deduced distribution policies and any other user-supplied policies/constraints (e.g. the component *C* requires some input data that resides on Machine *M*, so the component *C* should execute on Machine *M*) so that the resultant self-managed partitions can execute on different nodes in a distributed fashion. The underlying system comprises a platform-agnostic language and the associate pre-processor for byte-code to byte-code translation. The transformed program is based on self-contained concurrent objects communicating through any standard communication protocol and incorporates salient features from existing middleware technologies.

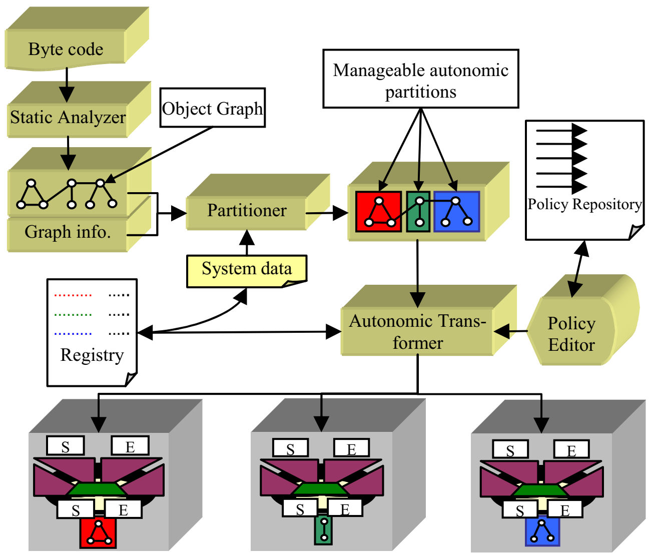

Figure 1. Overall flow of operation.

B. Autonomic Elements in ADE

In ADE, every distributed site (i.e. PCs, laptops, workstations, servers etc.) is managed by an Autonomic Element (AE) that control resources and interacts with other AEs in the system. More specifically, each AE encapsulates the program partition allocated to the site managed by the AE as its Managed Element (ME) and interacts with the environment by using standard autonomic metaphors. Each AE monitors the actual execution of the application and the behavior of the resource itself. AEs also set up a mutual service relationship to interact with each other so that information can be shared among them. Based on the information, the underlying autonomic system then adjusts the static parameters such as resource consumption, amount of communications to their run time values and if needed dynamically repartitions the graph. Besides this basic functionality, some of the AEs in the system are given some higher level management authority such as managing system registry or policy depository; acting as the user interface for program partitioning and transformation; being the source or destination of program input and output; managing workload etc.

To use the autonomic resources, a potential user must first register her computer with the autonomic system through some user portals. Once registered, an AE is initiated on that machine and configures itself properly with all the necessary system data and policy information and consequently makes its services available to other AEs. A user may deregister their machine at any time and consequently the AE running on that machine will delegate its current managed element to other available AE without the loss of useful computation. Since the distributed environment is shared by many users, the environment can change at run-time and so does the applications communication pattern, consequently AE need to adapt accordingly. To achieve that, AE provide monitoring services and based on the monitored data automatically takes decisions such as migrate the managed element (or portion) to a less busy AE, delays other non dedicated tasks to consume more resource, initiate backup to accommodate more tasks etc.

C. Application Deployment in ADE

ADE supports multiple, logically separated application environments each capable of supporting a distinct application. 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.

ADE uses a three step process to perform this mapping as shown in Fig. 2. In the first step of mapping, the application's code is statically analyzed to extract an application model expressed as lower-level resource requirements such as processing, bandwidth, storage etc. The next step involves constructing a model of the underlying network by obtaining knowledge about available resources such as their computational and storage capabilities, workloads etc. and then 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 systems overall utility based on certain policies, user-defined constraints and environmental conditions.

D. An Illustrative Example

Throughout this paper, a simulate-analyze-visualize application (Fig. 3) is used to illustrate and evaluate the important concepts described in this paper. A large number of scientific and engineering applications can be characterized by this notion, where information is simulated in a simulator (a high performance server may be required for execution), then passed to some analyzer that analyzes, operates and transforms the data, and eventually delivered to some clients (high end graphics may be necessary) for visualization. This specific application is chosen for

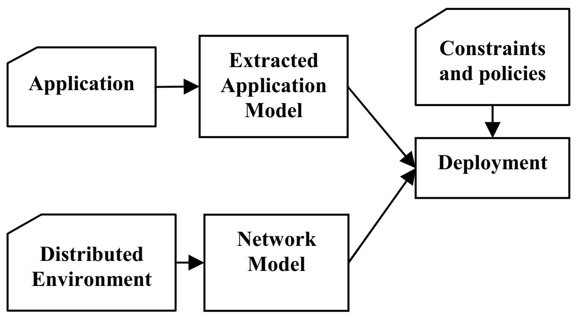

Figure 2. Application deployment process.

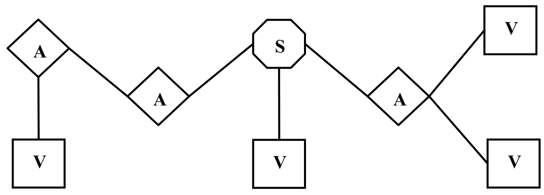

Figure 3. Simulate-Analyze-Visualize Application

investigation to realize the concepts of self-management in ADE because it is inherently distributed in nature and closely resembles vast majority of large-scale scientific and engineering applications this research is concerned with.

III. The application model

To be truly autonomic, a computing system needs to know and understand each of its elements. One such element is the application executing on the autonomic infrastructure; the system needs detailed knowledge of it. In ADE, 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. Moreover, graph theory algorithms can be exploited during deployment of such an application.

To construct such an application graph, two pieces of information must be determined, namely: 1) the resources (i.e. computation time, storage, network etc.) required by each application component and 2) the dependencies (directionality and weight) among the components which is caused by the interactions among them. More formally, it is necessary to construct 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.

IV. The network model

In this research, the target environment for the deployment of the application is a distributed environment consisting non-dedicated heterogeneous and distributed collection of nodes connected by a network. A resource (node) in this environment could be a single PC, laptop, server or a cluster of workstations. Therefore, each node has different resource characteristics. The communication layer that connects these diverse resources is also heterogeneous considering the network topology, communication latency and bandwidth. This research aims to organize this heterogeneous pool of 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 that correspond to low latency and high bandwidth between network nodes. The deployment of the application graph then 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.

To achieve this, a hierarchical model of the computing environment is utilized where the execution begins at the root and each node either executes the tasks assigned to it or propagates them to the next level. More specifically, ADE adopts a tree to model the underlying heterogeneous infrastructure. Each node in the tree is solely responsible for deciding whether to execute the application components allocated to it or propagate them down the hierarchy. Each parent node is capable of calculating the utility of its child based on processor speed, workload, communication delay, bandwidth etc. and selects its best child's subtree to delegate the components to, so that the overall communication is minimized and the delegated component's resource requirement is satisfied.

There are three main advantages of representing the underlying layer as a tree network. Each deployment decision can be made locally, which may not be optimal compared to centralized deployment, but certainly is efficient and adaptive. Maintaining a global view of a large-scale distributed environment becomes prohibitively expensive, even impossible at a certain stage, considering the potentially large number of nodes and the unpredictability associated with a large-scale computing system. The tree model adopted in this paper can grow and reconfigure itself to adapt to the dynamically evolving computing environment.

Second, this model relieves ADE from the costly evaluation of the utility function globally by limiting the utility evaluation within a subtree performed by the parent of that subtree. Each parent is capable of monitoring its child and calculating the corresponding utilities, as a result, able to redeploy the assigned subtrees corresponding to the changes in utility of its child. Optimizing certain utility function globally is certainly more attractive, however this does not scale very well when the number of deployed application and/or number of resources grow within the system. On the other hand, ADE may not be able to optimize utility or resource allocation, but by reducing the problem of evaluating and maintaining the utility across the whole system to the problem of managing the utility within a subtree, ADE promises to provide better adaptability and scalability in such a dynamic environment.

Third, the model fits very well with the classes of applications this research is concerned with. For instance, applications with large number of independent tasks basically exhibit master-slave behavior and coincide well with the tree model. Inherently distributed applications, on the other hand, can be modeled as divide and conquer type applications where components are divided among partitions and allocated to the children for execution. Even in the case of applications consisting of a large number of communicating components, there is still a single entity that initiates the set of communicating components and allocates them to processors. It is natural to think of the initiator as the root of the tree.

The proposed hierarchical 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 is used to model the underlying resources, which is built on top of the existing network topology. Such an architecture was utilized recently in [9,10] to investigate applications with master/worker paradigm. Fig. 4(a) shows an example computing environment where resources are distributed in three domains and Fig. 4(b) illustrates how this environment translates into a tree.

Formally, the entire network is represented as a weighted tree $T = (N, L, w_t, c_l)$, where N represents the set

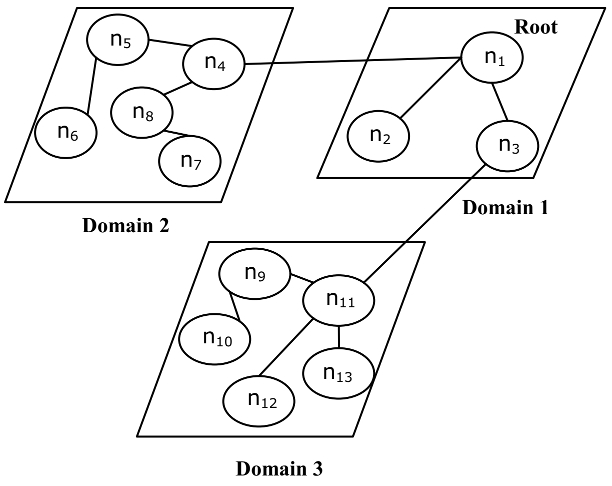

Figure 4(a). Sample distributed environment spans at 3 domains.

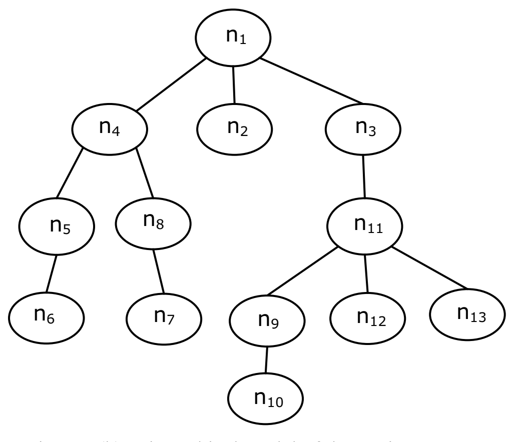

Figure 4(b). Hierarchical model of the environment.

of computational nodes and L represents network links among them. The weights associated with the nodes and edges represent the corresponding expected 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. When two nodes are not connected directly, their communication weight is the sum of the link weights on the shortest path between them. Therefore, larger values of node and edge weights translate to slower nodes and slower communication respectively.

V. Self-configuration and self-optimization

This section discusses the application graph deployment process and formulates the utility function that basically drives the initial deployment and further optimization.

A. Utility Function

In a distributed environment, the utility function can be calculated based on many criteria such as application performance, resource utilizations, user defined policies, or economic concerns. These issues can be associated with different objective functions of the optimization problem. In this research, the utility function governs both the initial placement of application components and their reconfigurations. The goal is to minimize the average application execution time and to provide high utilization of resources ensuring that both application-level (each application may have different importance to the system) and system-level requirements and constraints are satisfied. Toward our goal, the following criteria need to be considered by the utility function:

1. While mapping partitions containing a large number of application components in the tree network, nodes with a higher degree of connectivity should result in a higher utility as a 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.

B. 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. In ADE, 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. The nodes in the system are structured into a tree-shaped environment rooted at the source of execution according to the model described in section IV.

Once 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 [11] is exploited to collapse several application components into a single partition, so that |P| or less than |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 it is a mathematical function that returns the utility based on different application and system level attributes and considers the factors discussed in Section V.A. Considering all these issues, 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 and n_q represents a potential child for delegation of application node v_k, is calculated by using the following function:

$$U(v_j,v_k)=\frac{nc(n_q)}{np(v_k)}\times f_1\left(w_g(v_k)\times w_t(n_q)\right)\times f_2\left(w_g(v_j,v_k)\times w_t(n_p,n_q)\right),$$

where $nc(n_q)$ represents the number of children of network tree node n_q and $np(v_k)$ signifies the number of application components resident on application graph node v_k, if v_k is a collapsed node and represents a set of components. The function f_1 should produce lower utility when the computation cost associated with executing component v_k at node n_q

increases. In the same way, function f_2 should return lower values for increasing communication cost resulting from mapping edge (v_j,v_k) to link (n_p,n_q).

The terms involved in the above utility function are derived to fulfill the requirements (1-3) specified in the previous section. The utility model in this scenario is the "highest-degree child with the fastest computation capability and fastest communication link is more suitable for utility". To ensure that the partitions with the largest number of components are delegated to the highest degree child, candidate partitions need to be sorted according to their sizes and then deployed according to that order. In the case of simultaneous scheduling of multiple applications with different priorities, ADE needs to guarantee that higher priority applications execute before applications with lower priority. To achieve this, applications need to be 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:

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

C. *Self-optimization*

After initial placement, the environment may change and as a result utility may drop. Therefore it is necessary to monitor the utility and trigger reconfiguration as required. Reconfiguration within a subtree is expected to be a light weight process because of the way the underlying network is modeled. Through the resource monitoring module, each parent node periodically measures the workload at each child node and its bandwidth to that child node and changes computational and communication weights w_t and c_t accordingly. By employing these new values in the utility function, the parent observes the change in utility due to the changes in the network and computes nodes, and therefore initiates reconfiguration autonomously. Reconfiguration is costly and disruptive, therefore, it is not feasible to initiate reconfiguration unless it is productive. ADE triggers reconfiguration whenever the utility drops below a certain threshold (user specified or system generated by comparing the utility during initial deployment).

VI. Experimental evaluation

In order to evaluate the approaches described in the previous sections, we set up an experimental environment that simulates the deployment of the example application discussed in section II.D on a networked environment where nodes are organized as a tree. The network is represented as a collection of domains and each of them contains a random number of nodes. Each domain acts like a LAN and contains a single gateway through which communication with other domains is performed. Once created, the domains are populated with a random number of nodes and links are created to connect two randomly-chosen nodes, provided that adding the link does not create a cycle. Essentially, each domain is modeled as a tree whose root is its gateway and each node has a randomly chosen number of children. Each node is assigned a computational weight of the range [3.0, 15.0] and each link is assigned a communication weight in the range [1, 10]. Similarly the vertices and edges of the application graph (Fig. 3) are also populated with random computation and communication weights.

At this stage, we justify the applicability of our decentralized utility-driven deployment approach by finding the highest utility mapping of an individual application graph to the underlying network. Fig. 5 shows such a deployment as a result of the highest-utility mapping of the application graph (Fig. 3) to the underlying network (Fig. 4). Fig. 5 also illustrates other implementation aspects such as coarsening, partitioning etc. The resultant placement of application components to the network nodes (Fig. 5(c)) validates that it is possible to find the most efficient deployment by applying the utility function derived in this paper locally, between the parent-child pair, and without having the full knowledge of the network and the utility between all pairs of network nodes. As the nodes in the underlying layer are organized in a hierarchy according to network proximity, partitions are allocated to nearer nodes and communication overhead is minimized. By decentralizing the problem of finding the highest utility association among the set of components and the set of nodes, this solution also makes self-management feasible in the case of an infrastructure containing large number of nodes.

VII. Related Works

Fully automating the organization and optimization of a large distributed system is a staggering challenge and the autonomic computing research community is working towards that. Accord [12] is a component-based programming framework to support the development of autonomic application in grid environments. In their work, they have proposed a new programming paradigm where the composition (configuration, interaction and coordination) aspect is separated from computations in a component/service based models and both computations and compositions can be dynamically managed by the rules that are introduced during run time. AutoMate [13] enables the development of Grid-based autonomic applications that are context aware and capable of self-managing. AutoMate develops an autonomic composition engine to calculate a composition plan of components based on dynamically defined objective constraints that describe how a given high-level task can be achieved by using available basic Grid services. Autonomia [14] is a software development environment that provides application developers with tools for specifying and implementing autonomic requirements in a distributed application. The aforementioned approaches are developing

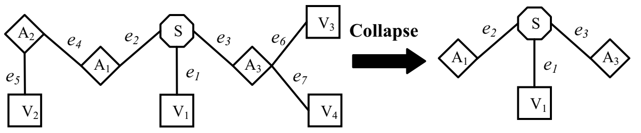

Figure 5(a). Application graph collapsed to a coarser level.

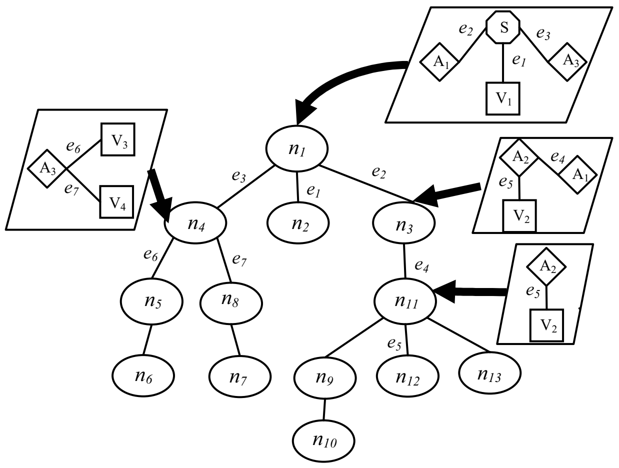

Figure 5(b). Graph partitions are allocated to the network nodes.

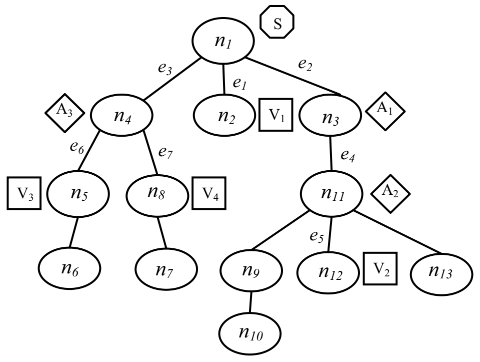

Figure 5(c). Final placement.

environments and suggest new programming metaphors in order to realize the desired benefits of self-management in a distributed environment. On the contrary, ADE is an execution environment intended for transforming applications written in a centralized fashion to the corresponding distributed version and executing it in a networked environment while offering self-management at both application and network level.

Unity [15] shares the same goal as ADE and provides a platform designed to help autonomic elements interact with each others and their environment. Unity realizes a number of autonomic system behaviors such as self-organization, self-healing and self-optmization in a distributed environment. However, in Unity resource-level utility functions for multiple application environments are sent to an element called Resource Arbiter, which computes a globally optimal allocation of servers across the application. In contrast, by utilizing tree model, ADE is capable of making local optimization decisions that are lightweight and decentralized, and as a result, provides better adaptability and scalability in a dynamic environment.

Kinesthetics eXtreme [16], or KX retrofits autonomic capabilities onto legacy systems designed and developed without monitoring and dynamic adaptation in mind. KX is being used to add self-configuration and self-healing functionality to several legacy systems, whereas ADE is focused to provide self-configuration and self-optimization. In addition to retrofit autonomic computing, externally, without any need to understand or modify the target system's code, ADE also tries to relive the programmer from the distributional concerns by automatically transforming a centralized application to a distributed one and deploying it to the underlying network.

The system proposed in [17] shares similar goal and presents a self-adaptive middleware for distributed stream management that deploys a large number of data flow operators across the underlying network to optimize the business utility. To achieve their goal, they adopt a hierarchical organization of underlying resources clustered according to various system attributes. On the other hand, ADE uses a tree to model the distributed environment.

Several recent works are classified as automatic partitioning system [18-21] and have similar goal of distributing an existing centralized application without rewriting the application's source code or modifying the existing runtime environment. The main difference between these systems and ADE is that ADE's partitioning and distribution decisions are utility driven. Automatic partitioning system only contains knowledge about the internal structure of the application, not about the environment the application is going to be deployed. ADE, on the other hand, is knowledgeable about both the application and the underlying network and automatically reallocates resources and reconfigures the deployed application graph to maximize the overall utility of the system.

VIII. Conclusions and Future Works

This paper presents ADE, a self-managed distributed environment where a centralized application is automatically decomposed into self-managed components, and distributed across the underlying network to maximize system's utility. The approach is to construct an application model for a centralized application, 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 proposed model is decentralized and adaptive. Preliminary result shows that it is possible to achieve efficient deployment by applying the utility function derived in this paper based solely on locally available information and without costly global communication or synchronization.

Currently, ADE is able to configure an individual application graph to the underlying network at a time while maximizing the utility. In future, we plan to study the behavior of ADE in the presence of more than one application graph. By deploying applications with different priorities at the same time on the underlying network, we hope to see how the higher utility paths previously available to the low priority application may get assigned to the high-priority application to sustain the overall utility of the system. In future, we also 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]. J.O. Kephart and D.M. Chess, "The vision of autonomic computing", *Computer*, Vol. 36, No. 1, 2003, pp.41–52.

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

[3]. M.M. Fuad and M.J. Oudshoorn, "An Autonomic Architecture for Legacy Systems", *Third IEEE Workshop on Engineering of Autonomic Systems (EASe06)*, Maryland, USA, April, 2006.

[4]. D. Deb, M.M. Fuad, M.J. Oudshoorn, "Towards Autonomic Distribution of Existing Object Oriented Programs", *International Conference on Autonomic and Autonomous Systems (ICAS'06)*, 2006.

[5]. M.M. Fuad and M.J. Oudshoorn, "Transformation of Existing Programs into Autonomic and Self-healing Entities", *The 14th IEEE International Conference on the Engineering of Computer Based Systems (IEEE/ECBS)*, Arizona, USA, 2007, pp. 133-144.

[6]. M. M. Fuad, D. Deb and M. J. Oudshoorn, "An Autonomic Element Design for a Distributed Object System", *ISCA 20th International Conference on Parallel and Distributed Computing Systems (PDCS 2007)*, Las Vegus, Nevada, USA, September, 2007, Accepted for publication.

[7]. M. M. Fuad, "An Autonomic Software Architecture for Distributed Applications", *Ph. D. thesis*, Department of Computer Science, Montana State University, USA, 2007.

[8]. D. Deb and M. J. Oudshoorn, "On Utility Driven Deployment in a Distributed Environment", *Fourth IEEE Workshop on Engineering of Autonomic Systems (EASe 2007),* Arizona, USA, 2007.

[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]. 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.

[12]. H. Liu and M. Parashar, "Accord: A Programming Framework for Autonomic Applications,", *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems*, Editors: R. Sterritt and T. Bapty, IEEE Press, Vol. 36, No 3, pp. 341 – 352, 2006.

[13]. 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.

[14]. 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, pp. 61-68.

[15]. 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, pp. 140-147.

[16]. J. Parekh, G. Kaiser, P. Gross and G. Valetto, "Retrofitting Autonomic Capabilities onto Legacy Systems." *Journal of Cluster Computing*, Kluwer Academic Publishers , Vol. 9, No. 2 pp. 141-159, April 2006.

[17]. V. Kumar, B. F. Cooper, K. Schwan, "Distributed Stream Management using Utility-Driven Self-Adaptive Middleware", *Second International Conference on Autonomic Computing (ICAC'05)*, 2005.

[18]. M. Tatsubori, T. Sasaki, S. Chiba and K. Itano, "A Bytecode Translator for Distributed Execution of Legacy Java Software", *ECOOP*, Hungary, June 2001, pp. 236-255.

[19]. E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning", *ECOOP*, Malaga, June 2002.

[20]. R. E. Diaconescu, L. Wang, Z. Mouri and M. Chu, "A Compiler and Runtime Infrastructure for Automatic Program Distribution"*, IPDPS*, 2005.

[21]. A. Spiegel, "Automatic Distribution of Object-Oriented Programs", *PhD thesis*, Fachbereich Mathematik u. Informatik, Freie Universitat, Berlin, 2002.