Cooperative Autonomic Management in Dynamic Distributed Systems

Jing Xu[1], Ming Zhao[2], and José A.B. Fortes[1]

[1] ACIS Lab, Electrical and Computer Engineering, University of Florida
[2] Computing and Information Sciences, Florida International University
{jxu,fortes}@acis.ufl.edu
mzhao@fiu.edu

Abstract. The centralized management of large distributed systems is often impractical, particularly when the both the topology and status of the system change dynamically. This paper proposes an approach to application-centric self-management in large distributed systems consisting of a collection of autonomic components that join and leave the system dynamically. Cooperative autonomic components self-organize into a dynamically created overlay network. Through local information sharing with neighbors, each component gains access to global information as needed for optimizing performance of applications. The approach has been validated and evaluated by developing a decentralized autonomic system consisting of multiple autonomic application managers previously developed for the In-VIGO grid-computing system. Using analytical results from complex random network and measurements done in a prototype system, we demonstrate the robustness, self-organization and adaptability of our approach, both theoretically and experimentally.

1 Introduction

Scalability, cost and administrative overheads make it desirable for large dynamic distributed computing systems to be self-manageable. This is a particularly challenging goal in dynamic environments, such as grids, where large numbers of resources are discovered or aggregated on-demand and are subject to hard-to-predict loads, failures or off-times. With the increasing complexity of system management, the need for self-managing systems, as proposed in [24], has never been more important than today. Extensive research [11][12][22] has focused on providing autonomic capabilities to individual system components, such as databases, application servers and middleware components. In general, these autonomic components use an application-level manager that is capable of monitoring and/or predicting performance and allocating resources as needed to deliver reliable applications with the expected Quality of Service (QoS). One can envision the use of these or similar components and their autonomic capabilities as the basic building blocks of large distributed systems.

Three questions that arise in this context are addressed in this paper. First, what interactions should take place among individual components, in order to achieve system-level self-management needed to support application-level autonomics? Implicit in this question is the need for information sharing among different components. Second, what type of network should be used to support the interactions? Implicit in this question is the need for the network to be highly scalable and robust to failures. Third, how should autonomic managers be designed to interact with other components, and enhance their autonomic ability? Implicit in this question is the need for cooperation among managers to efficiently collect and share information.

This paper proposes an approach for distributed-system self-management arising from interactions among the autonomic components deployed in the system. The key features of the proposed design are the effective use of components' limited monitoring and communicating capability, and their adaptation to the surrounding environment on the basis of information provided through a management overlay. The proposed system has the following properties:

- Self-adaptation: The system can dynamically respond to a changing environment to provide individual application managers with information and resources needed for achieving the desired QoS.
- Self-organization: The decentralized coordination enables the system to adapt to changes without external control. The global optimization is achieved through local decisions and interactions among neighbors.
- Robustness: There are no central resources that could become single points of failure or performance bottlenecks. Reconfiguration mechanisms effectively deal with dynamic resource availability.

An application of proposed approach in the context of the In-VIGO grid-computing system [1], is presented in this paper. In-VIGO provides a distributed environment where multiple application instances can coexist in virtual or physical resources. A virtual application manager (VAM) is a middleware component used to process user requests and manage application execution. Previous work considered the integration of autonomic capabilities into VAM to achieve self-optimizing and self-healing computation [22]. In this paper, a decentralized autonomic virtual application management system (DAVAM) is designed and implemented to further improve the scalability, efficiency and robustness. The DAVAM system is deployed on a large testbed that consists of tens of dynamic VAMs managing continuous jobs on hundreds of virtual machines with time-varying loads. Compared with our previously proposed centralized approach, the DAVAM system produces much lower job execution time and higher throughput in highly dynamic environments.

The rest of the paper is organized as follows. Section 2 describes the architecture of the decentralized autonomic system. Section 3 presents an analytical analysis of the system. The case study on DAVAM is presented in Section 4 and its experimental evaluation is discussed in Section 5. Section 6 reviews related work and Section 7 concludes the paper.

2 Autonomic System Model

We consider a highly dynamic distributed computing system consisting of a large collection of autonomic components [10]. Multiple components share distributed resources, as exemplified by grid-computing systems.

2.1 Autonomic Manager (AM) Model

The distributed system contains multiple autonomic components, each consisting of one or more managed components (e.g. jobs and resources) and an autonomic manager (AM). The behaviors of the components are independently managed by their AMs. To make optimal decisions towards desired states, AMs require global knowledge of the changing environment. However, in large distributed systems it is not scalable to collect and provide global knowledge through a central location.

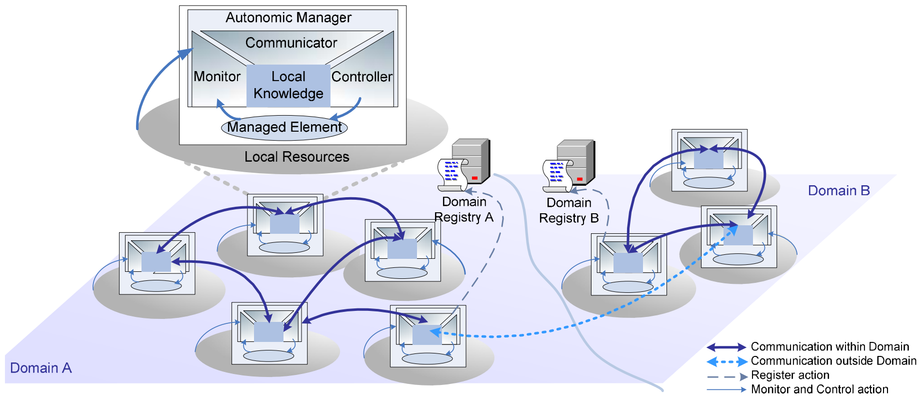


Fig. 1. A distributed autonomic system consisting of autonomic managers (AMs) across two domains, each with a registry indexing resources in the domain. Each AM contacts its domain registry to choose both the resources to be monitored (called local resources) and other AMs (called neighbors) to exchange local information.

To solve this problem, individual AMs are extended to monitor a small piece of their environment (hereon called *local* resources). Each AM has only a local view of the whole environment. However, interactions among the managers provide them with a global view of the system. The AM model (Fig. 1.) consists of the following components:

- **Monitor:** it collects, aggregates and filters the status information from its managed elements and its local resources.
- **Controller:** it manages the elements' behaviors based on analysis and prediction using the local knowledge.
- **Communicator:** it supports information exchanges with other autonomic managers.
- **Local Knowledge Base:** it stores the information obtained locally and through information exchanges between neighbors.

2.2 Decentralized Autonomic System

Because the computing resources are organized into domains which may correspond to administrative domains, a distributed domain registry infrastructure is designed to provide scalable and reliable resource location and AM discovery services. Each registry maintains an index of resources and the list of existing AMs in its domain.

When an autonomic component joins the domain, its AM registers its unique id in the registry, and chooses some existing AMs to cooperate with and selects some resources in the domain as its local resources. To improve reliability, nearby domain registries periodically exchange information so that each registry's local resource and AM lists are replicated in some other registries.

Local resource claiming: Each AM randomly selects a number of resources in the domain which have not yet selected by other AMs registry and claims them by marking the corresponding entries with its id. Once a resource is claimed by an AM, its status is monitored by the AM and stored in its local knowledge base during the claiming period. An AM disclaims its resources by unmarking them in the registry before its departure from the system.

Neighborhood building: When an AM joins a domain it selects *m* existing AMs in the same domain as its potential neighbors. AMs in the same neighborhood cooperate with each other by exchanging information. The neighbor selection can take place randomly, or preferentially which means that some AMs are more attractive and have a better chance to get neighbors. When departing from its domain, an AM unregisters itself by deleting its id from the domain registry and sends a message to its neighbors. In case an AM needs other domain's information, it can ask its domain registry for AMs in other domains to build a "cross-domain" neighborhood.

Information sharing and filtering: During its lifecycle, each AM becomes a dynamic information source by monitoring its local resources. This local information can be propagated through multi-AM cooperation. Every AM that receives a message from a neighbor must store it and later forward it to its other neighbors. Two approaches are used toghether to reduce the number of messages transmitted among the AMs. One is to define an obsolescence relation [14] between messages: a message m_1 is recognized as obsolete if m_2 contains more recent information that subsumes m_1. The other way is to evaluate how useful each message is, and drop the low-value messages.

2.3 Dynamic AM Network

The AM neighborhoods define a dynamic overlay network that changes as the AMs join and leave the system, in a manner similar to a peer-to-peer network [18][17]. The AMs must adapt their behaviors and interactions to the changing state. For example, an AM leaving or crashing may cause serious effects - claimed local resources may be no longer monitored by anyone, and some AMs may become isolated from others. To prevent and repair the damages, the following mechanisms are proposed.

Dynamic resource claiming: By periodically checking the domain registry, AMs can obtain the domain information such as the number of resources and AMs currently in the system, and then adjust the number of resources it should monitor to balance the monitoring load over the network. However, the information provided by domain registries might be incorrect because of AMs' unpredictable failures. To solve this problem, once an AM detects its neighbor's failure, it informs the domain registry and reclaims the resources that became unmonitored because of the failure.

Dynamic neighborhood building: If an AM decides to leave, it informs its neighbors by sending them a farewell message. In the case of AM or network failure, each AM measures the interval between two successive messages sent from the same neighbor and sets a timeout to detect the failure. When an AM is informed of a neighbor's departure or detects a neighbor's failure, it chooses its new neighbor with probability p (set to 0.5 as explained in Section 3.3). This mechanism allows AMs to maintain network connectivity.

3 Analytical Evaluation

3.1 Network Model

We use the conceptual framework and notations from complex network theory [2][6] to model the AM network and analyze its topology features. The decentralized autonomic system is modeled as a network in which each AM is represented by a node, and two nodes are linked if they are neighbors. The following notations are used to describe the network.

$n(t)$: the total number of nodes at time t.

$r(t)$: the total number of resources at time t.

m: the number of neighbors a node connects to when joining the network.

$k_i(t)$: the degree (the number of neighbors) of the ith node at time t.

$o_i(t)$: the local load (the number of claimed resources) of the ith node at time t.

The first two parameters describe the entire network and can be obtained directly from the domain registry, while the rest of the parameters describe the behavior of individual nodes.

3.2 Node Joining and Neighbor Selection

Consider the case where the network starts with one node, and at each step, a new node joins and connects to m existing nodes. At time t the network has a total of $n(t)$ nodes ($n(t) >> m$, for a large system). It is well known that the resulting network has the following properties [6].

$$\text{Total number of links: } e(t) = mn(t) - (m^2 + m)/2 \approx mn(t) \tag{1}$$

$$\text{Average degree: } \bar{k}(t) = 2e(t)/n(t) \approx 2m \tag{2}$$

$$\text{Diameter: } d(t) = \ln n(t) / \ln \bar{k}(t) \approx \ln n(t) / \ln 2m \tag{3}$$

Eq. (3) shows that the network diameter (shortest-path length between any two nodes) is small even for a large network. This "small world effect" [20] ensures that local information of one node can be propagated to any other node very quickly even in large networks. Different neighbor selection policies result in different network degree distributions. The random selection results in exponential distribution. In contrast, the preferential linking (the likelihood of connecting to a node is proportional to the node's degree) leads to a power-law distribution [2]. The major differences between these networks are their robustness against random network errors as discussed next.

3.3 Neighborhood Rebuilding

The effect of random damage on networks was simulated in [2] and the results show that scale-free networks display a high degree of tolerance against random failures. For exponential networks, Eq. (4) indicates that average degree decreases linearly with growing f (the fraction of removed nodes), which in turn increases network diameter (see Eq. (3)).

$$\bar{k}' = \bar{k}(1-f) \tag{4}$$

A dynamic neighborhood rebuilding mechanism is proposed to avoid this impact. When a node leaves the network, a fraction p of its neighbors establish new relationships with other nodes. Eq. (5) indicates that by choosing p equal to 0.5 the average degree can remain approximately constant, so does the network diameter.

$$\bar{k}' = 2e'/n' \approx 2\left(\bar{k}n/2 - (1-p)\bar{k}fn\right) / n(1-f) \overset{p=0.5}{\Rightarrow} \bar{k} \tag{5}$$

3.4 Local Load Adjustment

We use $\tilde{o}(t) = r(t)/n(t)$ to express the average ratio of the number of resources $r(t)$ to the network size $n(t)$ at time t. To balance the load on all the nodes, when a node joins the network, $o_i(t)$ is initialized as follows:

$$o_i(t) = \begin{cases} \lceil \tilde{o}(t) \rceil & if \quad \tilde{o}(t) < o^{\max} \\ o^{\max} & otherwise \end{cases} \tag{6}$$

The maximum number of resources each node can monitor is bounded to avoid overloading. Because the value of $\tilde{o}(t)$ may change as the network size and resource availability vary, each node periodically compares its current load with $\tilde{o}(t)$ and adjusts it accordingly.

3.5 Communication Cost

Each node in the network sends messages to its neighbors at constant time interval ΔT. With information filtering, the message size s_i can be bound to a fixed value S. The global communication cost of the network is

$$C = \sum k_i(t) \cdot s_i \le 2e(t) \cdot S = 2m \cdot n(t) \cdot S \tag{7}$$

which grows linearly with the network size. But from the perspective of a single node, the average communication cost stays almost constant.

$$\bar{c} = C / n(t) \approx 2m \cdot S \tag{8}$$

4 Case Study: DAVAM System

In order to validate the proposed model, we used In-VIGO [1] grid middleware to implement a decentralized Autonomic Virtual Application Management (DAVAM) system.

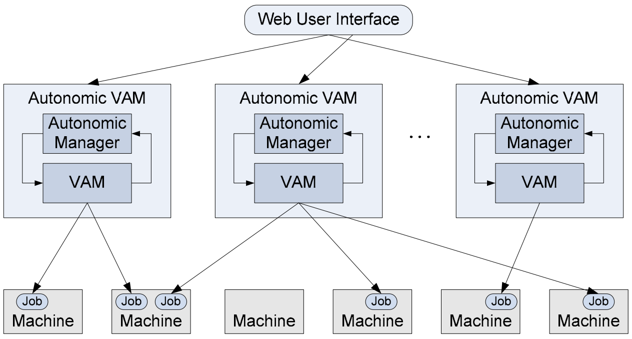


Fig. 2. The high-level view of autonomic VAM in In-VIGO. This figure shows multiple VAMs that submit jobs on multiple machines.

4.1 Background

In-VIGO is a grid-computing infrastructure that uses virtualization technologies to provide secure application execution environments. Fig. 2 provides a high-level view of the role of the autonomic Virtual Application Manager (AVAM) in In-VIGO (detailed in [22]). Typically, a user initiates an application session to run instances of a computational tool on grid resources[1].

Each session is managed by a middleware component, called the Virtual Application Manager. Autonomic features including self-optimization and self-healing are integrated into the AVAM. It relies on monitoring of job and resource conditions, predicting violations of user- and/or system-expected execution times, and restarting jobs in resources capable of delivering acceptable times. To achieve desired performance, each AVAM requires global knowledge of the time-varying resource information. However, the centralized approach in [22] using a global controller to collect and maintain the whole system status does not scale well in large-scale distributed systems.

4.2 Cooperative AVAM

Fig. 3. shows the major functions implemented in an AVAM. The local knowledge base stores information such as dynamic local resources' status, application run-time performance, the list of the neighbors and local resources claimed by the AVAM.

[1] A "tool" or "application" can consist of more than a single application, e.g., it could entail the execution of a workflow of application.

4.2.1 Controller

The controller is responsible for controlling the application execution to achieve reliable and optimized performance. The functions in the controller are listed below:

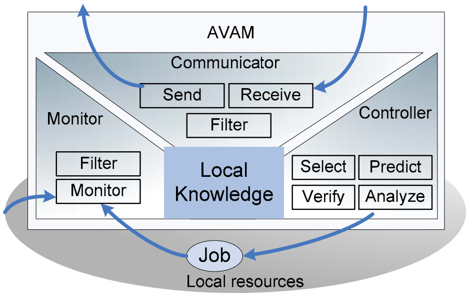


Fig. 3. The functions and information flow of a cooperative AVAM

Predict function: A memory-based learning algorithm [22][9] is used to predict resource usage for a given job, such as CPU cycles and memory usage. The basic idea is that the resources consumed by a job often depend on the input parameters supplied to the tool. Therefore, the “similarity” of two jobs is defined by the distance metric of two sets of inputs and resource usage is predicted based on the tool execution history.

Select function: The controller scans the list of resources in the local knowledge base and ranks them based on the job’s resource requirements and the resources’ capacity. To optimize the job’s performance, the controller selects the resource with the highest score. However, resource contention may happen if multiple AVAMs try to submit jobs to the same “best” resource simultaneously. A ε-random rule is used to deal with this problem. A randomly generated small number ε in the range [-0.1, 0.1] is added to each resource’s score, and then *Select* function ranks the resource list with these “modified” scores. By setting a small number ε, the ε-random rule is able to mitigate resource contention to a certain extent.

Verify function: After a resource is selected, this function checks the current status of the resource and verifies whether its score is still valid. If not, the controller selects the next candidate resource in the ranked list and repeats this verification process.

Analyze function*:* After a job is submitted to the chosen resource, the monitor keeps collecting the job's running status (e.g., current CPU time, elapsed time, and CPU utilization consumed by the job), which is used to estimate the job’s progress (see [22]). If it is predicted that the job cannot finish before the deadline, the controller will try to find a better resource that can satisfy the job requirements and reschedules the job to that resource. In the case when all the resources in one domain are heavily loaded, the controller selects its “cross-domain” neighbors and communicates with them to quickly get the resource information in other domains and determine on which resource it can submit the job.

4.2.2 Monitor and Communicator

The monitor periodically collects local resources' status information and checks every submitted job periodically. If the job finishes successfully, the monitor collects some statistic data about this execution and reports it to the local knowledge base for historical records. The communicator is responsible for sending and receiving messages to and from neighbors. There are four types of messages exchanged between neighbors.

Joining/leaving: An AVAM sends messages to its neighbors to notify its arrival or departure.

Local resource table: Each AVAM has its own current view of the resources' status and stores it in a local resource table. To disseminate this information, every AVAM periodically (every 10 seconds in our implementation) sends its local resource table to the neighbors.

Rewiring: Before leaving, an AVAM selects a fraction *p* (set to 0.5 in our case) of its neighbors and sends them rewiring messages. The receivers then choose some other AVAMs as their new neighbors.

4.2.3 Information filtering

The resources information collected by an AVAM must be filtered before being added to the local resource table to reduce message size. Each record has an *age* attribute to indicate the time elapsed since the last update. If two records contain the same resource's status, the older one gets filtered out.

Information filtering also happens by purging the lower-values records from the table. Concentrating on CPU-intensive applications, AVAMs are interested in resources with high CPU processing power. Thus, the value of the *i*th resource is defined as follows. If CPU utilization stays below 100%, the CPU capacity is calculated by the CPU speed and utilization; otherwise, it is computed using the CPU load (the queue length of the runnable processes). A weight of 0.01 is used to make these two measurements comparable.

$$Value_i = \begin{cases} CPU\_Speed_i \times (1 - CPU\_Utilization_i) & \text{if CPU\_Utilization}_\text{i} < 100\% \\ CPU\_Speed_i / CPU\_Load_i \times 0.01 & \text{otherwise} \end{cases} \quad (9)$$

Due to the dynamic nature of grid resources, the older a resource record becomes, the less accurate it is. Therefore, the record's value is reduced by a factor corresponding to its age, represented as α ($\alpha = 1 - age / max$), where the max is set to 60 seconds in our implementation. With this information filtering, a local resource table's size is reduced by only retaining the resources with high CPU processing capability.

5 Experimental Evaluation

This section evaluates the proposed DAVAM system with respect to scalability, efficiency and robustness.

5.1 Setup

The experiments were conducted on a subset of the In-VIGO system. The computer resources consist of 200 VMware-server virtual machines (each has 128 MB memory and runs Red Hat 7.3) hosted on a cluster of ten dual 2.4GHz hyper-threaded Xeon nodes. In the experiments, a considerable amount of background load was also introduced into the resources by launching CPU-intensive jobs. Dynamic loading environments were created by randomly choosing and loading different subsets of the resources (100 randomly chosen resources, unless otherwise noted) every 50 seconds. The domain registries are implemented with MySQL. TunProb (Numerical Calculation of the Transmission Probability for One-Dimensional Electron Tunneling), a tool available on the In-VIGO portal, is used as a benchmark representative of CPU-intensive workloads. In the experiments each AVAM was used to manage the execution of one or more instances of TunProb.

The DAVAM system initialization process starts with one AVAM. Then at each increment of time (one second) one new AVAM is started until the expected system size is reached. Each AVAM establishes connections with *m* (0~6) existing AVAMs in its domain. Each AVAM monitors up to five virtual machines as its local resources, and updates their status in its local resource table every ten seconds. AVAM neighbors exchange their local resource tables every ten seconds and the table can only keep up to ten records.

5.2 Experimental Evaluation of Efficiency

The efficiency of the DAVAM system is reflected by each AVAM being able to quickly obtain the current status of the entire system and find good resources for its jobs. The first experiment investigates how the performance changes with different numbers of neighbors each AVAM contacts when joining the system. Fifty AVAMs were initially started in the domain, and ten seconds later another five AVAMs joined and each selected *m* (0~6) neighbors. After ten seconds of their arrivals, the five AVAMs began to submit jobs continuously until they left the domain 140 seconds later.

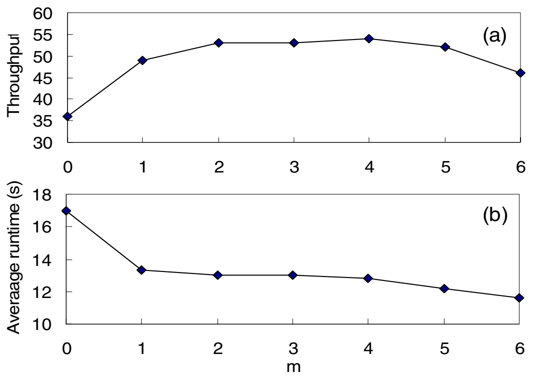


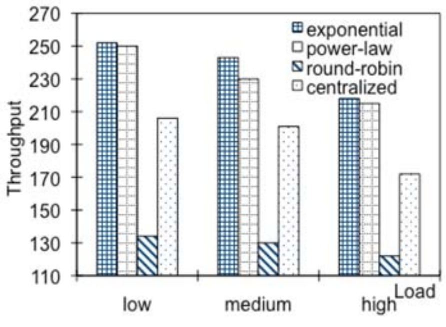
Fig. 4. The comparison of the total number of jobs finished by 5 AVAMs (a) and the TunProb jobs' average execution time (b) with different values of m during 150 seconds

Fig. 4. compares the average job runtime and the throughput (the total number of jobs completed by the five AVAMs) with different values of *m*. As expected, the worst performance occurs when each AVAM does not have any neighbors. As the value of *m* increases, the performance improves because AVAMs can learn more resources' information through interaction with their neighbors and select resources more wisely. Figure 4 also indicates that, when *m* exceeds five, the throughput drops because the benefit from contacting more neighbors is outweighed by communication overhead.

5.3 Experimental Evaluation of Scalability

In the second experiment, we studied the system scalability by comparing the performance of DAVAM with *centralized* and *round-robin* approaches. Forty AVAMs join the domain and each one submits jobs continuously for 150 seconds. In the DAVAM approach, each AVAM selects two neighbors. The neighbor selections, with and without preference, lead to two types of networks, *power-law* and *exponential networks* [2], respectively. The *centralized* approach uses a central monitor to collect and store resources' status in a central database. Each AVAM chooses the best resource currently available in the database to submit its jobs. The *round-robin* approach does not need any resource status information and chooses resources in a round-robin manner. The experiments were conducted in three loading environments – low, medium and high, in which 30%, 50% and 70% of randomly chosen resources were loaded with CPU-intensive processes, respectively.

Fig. 5. shows the average job runtime and the overall throughput of the different approaches. Both *exponential* and *power-law* AVAM networks deliver similar best performance because the small world property makes sure that each AVAM in the network can obtain the latest system-wide resource status very quickly. Furthermore, the ε-random resource selection rule avoids resource contention among multiple AVAMs. In contrast, the *centralized* approach suffers from database-access contention between the AVAMs and the central monitor. The *round-robin* approach gives the worst performance because it does not consider any dynamic information for resource selection.



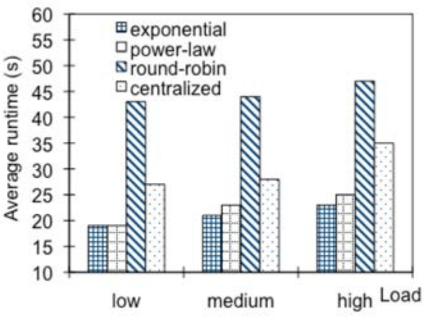


Fig. 5. The comparison of the jobs' average execution time and the total number of jobs finished by 40 AVAMs for the DAVAM and the *centralized* and *round-robin* approaches in three different loading environments

5.4 Experimental Evaluation of Robustness

The third experiment studies the robustness of the DAVAM approach, where the system-level information is constructed by the distributed cooperative AVAMs, in contrast with the centralized approach, where a central database is used to store the global knowledge. In the experiment, 50 AVAMs were started at the same time. After 200 seconds, half of them left and the others continued to work and submit jobs for another 200 seconds. In DAVAM the remaining AVAMs react to system changes by contacting new neighbors and reclaiming resources from the domain registry. The neighborhood rebuilding mechanism maintains the DAVAM network connectivity, and the resource reclaiming ensures that most of the resources are monitored by at least one AVAM. Fig. 6. compares the average job runtime and the throughput by the 25 AVAMs before and after the other AVAMs' leaving. For both exponential and power-law networks, the performance of the remaining AVAMs is almost unaffected even if a high number of AVAMs left the system.

For the *centralized* approach, on the contrary, if the central database fails, none of the AVAMs can retrieve any new information from the database, so they have to continue using the resources chosen before the database failure. Figure 6 shows that, without the dynamic resource information provided by the database, the performance drops dramatically. Similar effects can be observed if the central monitor fails.

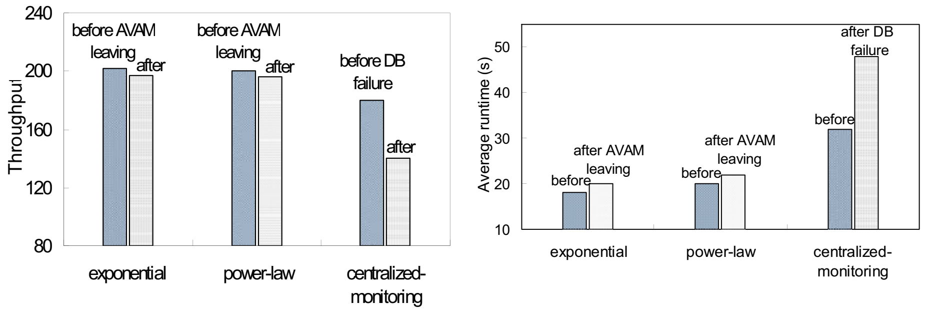


Fig. 6. TunProb jobs' average execution time and the total number of jobs finished by 25 AVAMs before and after AVAM leaving, for DAVAM exponential and power-law networks, and before and after failure of a central database when it is used for centralized monitoring

5.5 Discussion

It may possible to design a hierarchical system to circumvent scalability issues caused by a purely centralized approach and also achieve the similar performance with the p2p approach. However, in a dynamic environment where nodes can join and leave at any time, it is very difficult to construct and maintain a balanced, optimal hierarchical structure. Moreover, the supernodes (root notes) at the top level in the hierarchical system can potentially cause single-point system failures and/or lead to isolated nodes in the system. Although replication can compensate for potential unstable behavior of a supernode, it will add resource costs and communication overhead to keep replicas consistent.

6 Related Work

Agent-based [8][16] modeling is a very natural and flexible way to model distributed interconnected systems. In [2] several distributed and self-organizing algorithms are proposed for placement of services on servers. For each service a service manager is instantiated to create multiple “ants” (agents) and send them out to the server network. The ant travels from one server to another, choosing the servers along the path based on locally available information. The ant then finally makes a decision, based on the knowledge it has accumulated on its travel. Service manager and the spawned ants work with local information, which ensures scalability. Similarly, Messor proposed to use “ants” wandering over the network to explore load conditions. The goal is to achieve load balancing by ants moving jobs from the most overloaded node to underloaded ones.

In our system, each autonomic component can be identified as an agent, and the autonomic system as a multi-agent system. Each autonomic component is both cooperative (sharing its local knowledge with neighbors) and selfish (trying to find and allocate the best resources for its own jobs). The authors in [8] claim that no obvious gain can be achieved from communication between agents. The reason is that if all the agents have a "better" picture of the whole system, they all tend to use the best resources and thus cause competition. In contrast, the resource verification and ε-random selection mechanisms applied to our system can prevent this problem and their effectiveness is proved by the experiments.

The peer-to-peer model offers an alternative to the traditional client-server model for many large-scale applications in distributed setting. Epidemic (or gossip) algorithms [7][4] have proved to be effective solutions for disseminating information in large-scale systems. The basic idea is that each process periodically chooses a random subset of processes in the system and sends them the new information it has received.

Traditional epidemic algorithms rely on each process having knowledge of the global membership which is not realistic for large groups of processes. Our system uses a very simple membership protocol to establish and rebuild neighbor connectivity with support from the decentralized domain registry service.

7 Conclusions

This paper presents an autonomic computing system in which multiple autonomic components collaborate to optimize the behavior of the system. A general autonomic manager model is designed to control the managed elements’ internal state and manage its interactions with the surrounding environment. The autonomic manager is lightweight, making it suitable for many distributed systems. Each has a local view of the system state and communicates periodically its partial knowledge to its neighbors, thus contributing to building a common, shared global view of the system state. A decentralized registry provides scalable and reliable neighbor and resource discovery service for the system. The overlay network structured by the neighbor relationships is demonstrated to be highly reliable and efficient. The results show that the decentralized and cooperative nature of the system yields a number of desirable properties, including efficiency, robustness, and scalability under a highly dynamic environment.

In Introduction we raised the question of what component interactions are needed for system-level self-management to support autonomic applications. Our results show that simple exchanges of local information suffice to enable application managers to find resources that best suit performance requirements of an application. Another question asked which network should be used to support the communication needed to establish connectivity and share information. We found that both exponential and power-law networks yield small diameters to support low-latency communication needed for timely sharing of information among system components. The question of how to design autonomic managers capable of cooperatively interacting with each other has been answered by describing a set of functions implemented by the typical components of an autonomic manager: monitor, communicator, controller and a local knowledge base. The interaction between managers consists of simple information exchanges and each manager has a small cache to store partial "global" information to enhance its autonomic ability. The resulting design is rather lightweight and applicable beyond the concrete In-VIGO scenario used in this paper to validate the proposed approach.

There are additional questions that require further research. Among them, to what degree does our design mitigate the occurrence of races or oscillations among requests or job allocations? Our approach reduces their likelihood because communication latencies are small, age attributes are used to avoid using very dated information and an ε-random resource selection rule is used to mitigate the probability of resource contention. A complete answer would require a characterization of the conditions that lead to oscillations and races in distributed systems without (and with) our techniques. This is outside the scope of this paper and left as a challenge for future work.

Acknowledgments. The US National Science Foundation partially supports this work under grant numbers IIP 0925103, CNS 0855123, CNS 0821622, IIP 0823896, IIP 0758596. We also acknowledge the support of the Bell-South Foundation and Shared University Research grants from IBM.

References

1. Adabala, S., et al.: From Virtualized Resources to Virtual Computing Grids: The In-VIGO System. In: Future Generation Computing Systems (2005)
2. Albert, R., Barabási, A.: Statistical mechanics of complex networks. Rev. of Mod. Phys. 74 (2002)
3. Andrzejak, A., et al.: Algorithms for Self-Organization and Adaptive Service Placement in Dynamic Distributed Systems. HPL Tech. Rep. 9/02
4. Barabasi, A., Albert, R., Jeong, H.: Mean-field theory for scale-free random networks. Physica A (1999)
5. Birman, K.P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., Minsky, Y.: Bimodal Multicast. ACM TOCS 17 (1999)
6. Cohen, R., Erez, K., ben-Avraham, D., Havlin, S.: Resilience of the Internet to random breakdowns. Phys. Rev. Let. (2000)
7. Dorogovtsev, S.N., Mendes, J.F.F.: Evolution of networks. Adv. Phys. 51 (2002)
8. Eugster, P.T., Guerraoui, R., Kermarrec, A.M., Massoulie, L.: Epidemic Information Dissemination in Distributed Systems. IEEE Computer 37 (2004)

9. Jennings, N.R.: Building complex, distributed system: the case for an agent-based approach. Communications of the ACM 44(4), 35–41 (2001)
10. Kapadia, N., Fortes, J.A.B., Brodley, C.E.: Predictive Application-Performance Modeling in a Computational Grid Environment. In: Proceedings of the 8th IEEE international Symposium on High Performance Distributed Computing (August 1999)
11. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer (2003)
12. Liu, H., Parashar, M., Hariri, S.: A Component-based Programming Framework for Autonomic Applications. In: Proceedings of the First international Conference on Autonomic Computing (June 2004)
13. Melcher, B., Mitchell, B.: Towards an autonomic framework: Self-configuring network services and developing autonomic applications. Intel Technology Journal 8(4) (2004)
14. Pereira, J., Rodrigues, L., Oliveira, R.: Semantically Reliable Multicast: Definition Implementation and Performance Evaluation. IEEE Trans. Computers 52 (2003)
15. Qiao, Y., Bustamante, F.: 'Elders know best -handling churn in less structured p2p systems. In: 5th IEEE Intl. Conf. on Peer-to-Peer Computing (2005)
16. Schaerf, A., Shoham, Y., Tennenholtz, M.: Adaptive load balancing: A study in multi-agent learning. J. A.I. Res. (1995)
17. Schoder, D., Fischbach, K.: Core Concepts in Peer-to-Peer (P2P) Networking. In: P2P Computing: The Evolution of a Disruptive Technology. Idea Group Inc., Hershey
18. Steinmetz, R., Wehrle, K. (eds.): Peer-to-Peer Systems and Applications. LNCS, vol. 3485. Springer, Heidelberg (2005)
19. Thrun, S.B.: The Role of Exploration in Learning and Control. In: Handbook of Intelligent Control: Neural Fuzzy and Adaptive Approaches. Van Nostrand Reinhold (1992)
20. Watts, D., Strogatz, S.: Collective dynamics of 'small-world' networks. Nature 393 (1998)
21. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Kephart, J.O.: An architectural approach to autonomic computing. In: Proceedings of the First international Conference on Autonomic Computing (2004)
22. Xu, J., Adabala, S., Fortes, J.: Towards Autonomic Virtual Application Manager in In-VIGO system. In: Proceedings of the Second international Conference on Autonomic Computing (June 2005)
23. Xu, J., Zhao, M., Fortes, J.: Cooperative Autonomic Management in Dynamic Distributed Systems. Technical Report (April 2006)
24. IBM's Perspective on Autonomic Computing, `http://www.research.ibm.com/autonomic/`