The Role of an Agent Organization in a Grid Computing Environment

Sherief Abdallah , Haizheng Zhang, and Victor Lesser
Computer Science Dept.
University of Massachusetts
{shario,hzhang,lesser}@cs.umass.edu

Abstract

In this paper we show how imposing an organization (topology and search mechanism) on agents in a peer-to-peer network can improve system performance. The organizational structure is the underlying topolgy connecting agents in the system. The search mechanism is how agents are traversed in such a topology. In particular, we discuss our solutions to a resource allocation problem, involving coalition formation, and a content retrieval problem, involving locating relevant documents.

Introduction

Mechanisms for handling issue of scale involving resource allocation and information retrieval will become increasingly important as the size of Grid computing environments inevitably grow larger. One of the important ways to achieve scalability is to impose an organization on the set of agents in the system. While the term "organization" has different meanings in different contexts, we focus here on two aspects of an organization: its structure and the associated search mechanisms. The organizational structure is the underlying topolgy connecting agents in the system. The search mechanism is how agents are traversed in such a topology. For example, we can view the Internet as a hierachical organizational structure and the routing protocols as search mechanisms for this structure.

In this paper we present work that studies the effect of organizations (topology and search mechanism) on the performance of a system. In particular, we discuss our solutions to a resource allocation problem, involving coalition formation, and a content retrieval problem, involving locating relevant documents. Though these ideas originated in the area of multiagent systems, we believe that they naturally lend themselves for use in a Grid computing environment.

The first part of this paper proposes a scalable, distributed solution to the coalition formation problem. This problem has received considerable attention in multiagent systems community (Shehory 1998; T. Sandholm 1999). The input to the coalition formation problem is a set of nodes, each controlling some amount of resources, and a set of tasks, each requiring some amount of resources and each worth some utility. The solution assigns a subset of nodes to a subset of tasks, such that each task's requirements are satisfied and total utility is maximized. This problem is related to the Grid Information Service, and in particular to *composed queries*. The Grid Information Service (GIS) is responsible for keeping track of the status of every resource in the system and enabling applications to query about it these resources (Czajkowski & et al 2001). For example, a composed query of the form "find me a subset of nodes that controls (collectively) at least 1GB of memory and 1 teraFLOPS" can be directly mapped to a coalition formation task. Our solution uses an underlying organization that interconnects nodes in the system. Reinforcement learning techniques and neural-nets are used to learn appropriate policies. We studied different combinations of organization structures (topologies) and search mechanisms and our results show how both affect system's performance.

In the second part of this paper, we also show that such an organizational approach can benefit a Grid-based information retrieval system.[1] Specifically, we exploit an incrementally accumulated agent-view approch that implicitly defines an organization among agents to efficiently locate relevant documents. During this accumulation process, each agent tailors its neighbors based on content similarity and other criteria. This process can be considered as an implicit distributed clustering procedure. Thereafter, two context-aware search approaches are employed to search on this reorganized topology.

Organization-Based Coalition Formation

Problem Definition

Let $T = \{T_1, T_2, ..., T_q\}$ be the set of tasks coming into the Grid during one time window. Each task T_i is defined by the tuple $\langle u_i, rr_{i,1}, rr_{i,2}, ..., rr_{i,m} \rangle$, where u_i is the utility gained if task T_i is accomplished; and $rr_{i,k}$ is the amount of resource k required by task T_i. Let $I = \{I_1, I_2, ..., I_n\}$ be the set of individual agents in the system. Each agent I_i is defined by the tuple $\langle cr_{i,1}, cr_{i,2}, ..., cr_{i,m} \rangle$, where $cr_{i,k}$ is the amount of resource k controlled by agent I_i.

[1] This work was jointly authored by H. Zhang, B. Croft, B. Levine, and V. Lesser (Zhang *et al.* 2004)

The coalition formation problem is finding a subset of tasks $S \subseteq T$ that maximizes utility while satisfying the coalition constraints, i.e.:

- $\sum_{i|T_i \in S} u_i$ is maximized
- there exists a set of coalitions $C = \{C_1, ..., C_{|S|}\}$, where $C_i \subseteq I$ is the coalition assigned to task T_i, such that $\forall T_i \in S, \forall k : \sum_{I_j \in C_i} cr_{j,k} \geq rr_{i,k}$, and $\forall i \neq j : C_i \cap C_j = \emptyset$

In other words, each task is assigned a coalition capable of accomplishing it and any agent can join at most one coalition. This means if the resources controlled (collectively) by a coalition exceed the amount of resources required by the assigned task, the excess resources are wasted.[2]

Problem Solution

Because the coalition formation problem is NP-hard, an optimal algorithm will need exponential time in the worst case (unless NP = P). We need an approximation algorithm that can exploit information about the problem. If the environment (in terms of incoming task classes and patterns) does not follow any statistical model, and agents continually and rapidly enter and exit the system, there is little information to be exploited. Luckily, in many real applications the environment does follow a model, and the system can be assumed closed. In such cases, it is intuitive to take advantage of this stability and *organize* the agents in order to guide the search for future coalitions. We chose to organize agents in a *hierarchy*, which is both distributed and scalable.

Figure 1 shows a sample hierarchical organization.[3] An *individual* (the leaves in Figure 1) represents the resources controlled by a single agent. A *manager* (shown as a circle in Figure 1) is a computational role, which can be executed on any individual agent, or on dedicated computing systems.

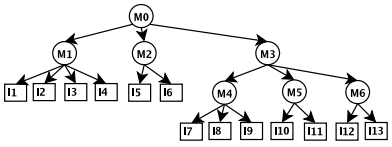

Figure 1: An Organization Hierarchy

Each manager M has a set of children, $children(M)$, which is the set of nodes directly linked below it. So for instance, in the organization shown in Figure 1, $children(M6) = \{I12, I13\}$, while $children(M3) = \{M4, M5, M6\}$. Conversely, each child C has a set of managers $managers(C)$. For example, $managers(M4) = \{M3\}$. For completeness, children of an individual are the empty set, and so are the managers of a root node. Also for each agent A, we define $organization(A)$ to be the set of all agents reachable from A. In the above example, $organization(M3) = \{M3, M4, M5, M6, I7, I8, I9, I10, I11, I12, I13\}$.

Local Decision Algorithm 1 describes the decision process made by each manager in the organization. The algorithm is executed every time a task T is received by a manager M (either from the environment or from another agent). LOC is the list of coalitions allocated for previous tasks. The algorithm works as follows. M evaluates its current state s (see Section State Abstraction). M then selects an action a based on its policy (see Section Learning). This action a can either be to stop forming the coalition or to select a child $M_i \in children(M)$. If a child is selected, a subtask T_i of T is dynamically created based on M_i's state (Section Task Decomposition). M then asks M_i to form a subcoalition capable of accomplishing T_i. (The notion $M_i.allocateCoalition(T_i)$ means that the function $allocateCoalition$ is called remotely on agent M_i). M_i forms a subcoalition C_{T_i} and sends a commitment back to M. M updates C_T and learns about this action. M updates its state, including the amount of resources to be allocated (UR) and the corresponding utility to be gained (uu).

M selects the next best action and the process continues as long as all the following conditions hold (step 3): T requires more resources than currently allocated, M still controls some unallocated resources that are required by T, and the stop action has not been selected. At the end M adds the formed coalition C_T to its list of commitments LOC and returns C_T. Note that manager M executes Algorithm 1 if and only if $organization(M)$ has enough resources to accomplish T. Otherwise, M passes T up the organization hierarchy until it reaches a capable manager (T is rejected if even the root manager does not have enough resources). Also to simplify handling of multiple tasks, we do not allow coalition formation of a task to be interrupted.

Example Figure 2 shows how a group of agents, organized in a hierarchy, can cooperate to form a coalition. A task $T = \langle u = 100, rr_1 = 50, rr_2 = 150 \rangle$ is discovered by agent $M6$. Knowing that $organization(M6)$ does not have enough resources to accomplish T, $M6$ sends task T to its manager $M3$. Since $organization(M3)$ has enough resources to achieve T, $M3$ uses its local policy to choose the best child to contribute in achieving T, which is $M5$. $M3$ decomposes T into subtask $T_5 = \langle u_5 = 50, rr_{5,1} = 0, rr_{5,2} = 100 \rangle$, and asks $M5$ to allocate a coalition for it. $M5$ returns a committed coalition $C_{T_5} = \{I10, I11\}$. The process continues until the whole task T is allocated. Finally, $M3$ integrates all subcoalitions into C_T and sends it back to $M6$.

State Abstraction Since managers control exponentially more individuals as we ascend in the organization, abstraction of state information is necessary to achieve scalability (otherwise we are effectively centralizing the problem). In our solution, each manager M abstracts the state of its organization, $organization(M)$. The price of this abstraction is loss of information (a manager higher in the hierarchy

[2]Having more than one type of resource means that there will be trade-offs, where decreasing the excess of one resource type may increase the excess of another resource type.

[3]Note that the example in Figure 1 shows a strict tree organization. In general, an organization may be represented by a directed acyclic graph, where the same agent may have more than one manager.

Algorithm 1 allocateCoalition(T)

INPUT: task $T = \langle u, rr_1, ..., rr_m \rangle$
OUTPUT: coalition $C_T = \{I_1, ..., I_{|C_T|}\}$

let $C_T = \{\}$, $uu \leftarrow u$, $UR \leftarrow \langle rr_1, ..., rr_m \rangle$, $stop \leftarrow$ false, $AR \leftarrow$ the amount of available resources controlled by $M = availableResources() = totalResources(M) - \sum_{C \in LOC} totalResources(C)$
$s \leftarrow$ encodeState(uu, UR)
while $UR > \overline{0}$ AND $UR.AR > 0$ AND $stop =$ false **do**
  $a \leftarrow$ selectAction(s)
  if a is the **stop** action **then**
    $stop \leftarrow$ true
  else
    let M_i be the child corresponding to a.
    $T_i \leftarrow$ decomposeTask(T , M_i)
    $C_{T_i} \leftarrow M_i$.allocateCoalition(T_i)
    $C_T \leftarrow C_T \cup C_{T_i}$
    $UR \leftarrow UR - totalResources(C_{T_i})$, $uu \leftarrow u_{T_i}$, and $AR \leftarrow AR - totalResources(C_{T_i})$
    $r \leftarrow$ time and communication costs of forming C_{T_i}
    if $UR = \overline{0}$ /* T does not need more resources */ **then**
      $r \leftarrow r + u$
    end if
    $s' \leftarrow$ encodeState(uu, UR) /* the next state */
    learn(s, a, r, s')
    $s \leftarrow s'$
  end if
end while
$LOC \leftarrow LOC \cup C_T$ /* to exclude agents in C_T from next allocations */
return C_T

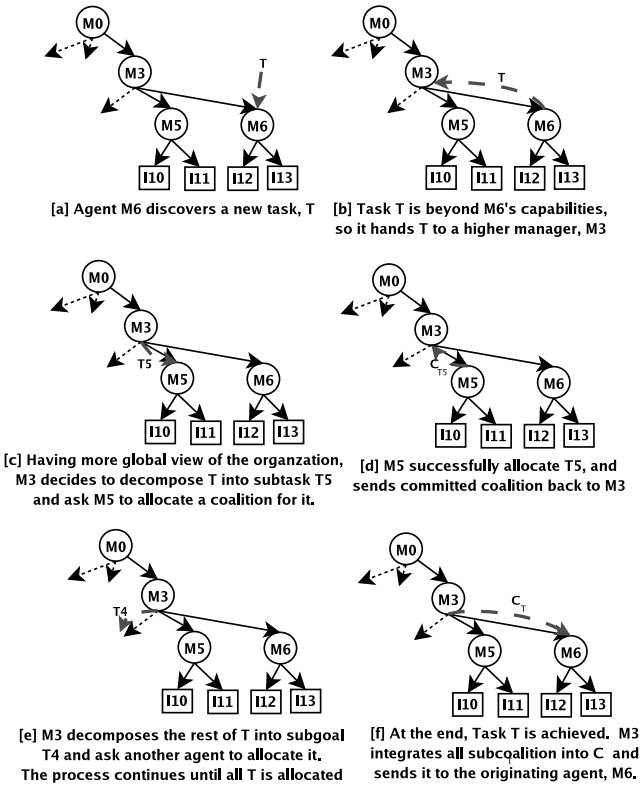

Figure 2: An example of organization-based coalition formation.

"sees" fewer details about its organization). For a manager M, the function $encodeState$ collects the abstract states of each child $M_i \in children(M)$ and encodes this information along with the vector of resources to be allocated, UR, and the utility to be gained, uu, to produce the current state of $organization(M)$. (This encoding is then fed to neural nets to get action values, as we discuss in Section Learning.)

Due to the large state space, we use a factored state. That is, a state is defined by a set of features. To abstract a feature at a manager M, we need to define it recursively in terms of features abstracted at M's children. For example, we defined the feature vector $totalResources(M) = \langle tr_1, ..., tr_m \rangle$ as the total amount of resources controlled by manager M (where m is the number of different resource types). It can be defined recursively as follows: $totalResources(M) = \sum_{c \in children(M)} totalResources(c)$. That is, the total resources controlled by a manager is the sum of the total resources controlled by its children. For an individual I_i, $totalResources(I_i) = I_i$.

Some features cannot be abstracted directly, but can still be inferred from other abstracted features. For example, $averageResources(M)$ is a feature vector of the average amount of resources controlled by any individual in $organization(M)$. To compute this feature, we need another abstract feature: the total number of individuals in an organization, $size(M) = \sum_{c \in children(M)} size(c)$. Then we have $averageReources(M) = totalResources(M)/size(M)$.

The above features are assumed constant throughout the system lifetime. For each constant feature we define a corresponding dynamic feature, preceded by $avail$, to indicate the current value of the feature. For example, the number of individuals not allocated to tasks $= availSize(M) = size(M) - \sum_{C \in LOC(M)} size(C)$, and their aggregated resources $= availTotalResources(M) = totalResources(M) - \sum_{C \in LOC(M)} totalResources(C)$.

Task Decomposition When a manager M selects a child M_i to ask for contribution regarding task T, M decomposes T heuristically to T_i . As we described in Section State Abstraction, a manager M only sees abstract features of its child M_i. Using this information, M needs to find T_i that is more suitable to $Organization(M_i)$. The heuristic we use is to try to ask each child a multiple, α, of the average available resources it controls; i.e., $\alpha \times \frac{availTotalResources(M_i)}{availSize(M_i)}$. We want to chose α such that the *expected* excess of resources is minimized.[4]

[4]When M decomposes T into T_i, it does not know what coalition M_i would return, which makes it difficult to minimize the

The intuition behind the heuristic is as follows. If all individuals controlled by M_i are identical, the heuristic is the only choice to avoid wasting resources. As individuals become more diverse, the multiple of average available resources remain the most likely to succeed without wasting any resources. Because agents can not participate in more than one coalition, the minimum of the ratio l_j over all resource types is selected and used for all other resource types. Also to ensure progress, α is at least 1. Finally, the utility of the decomposed task is proportional to the total of the decomposed resources.

Learning A key factor in the performance of our system is how a manager selects its actions (function $selectAction$ in Algorithm 1). In particular, in what order should a manager ask each child for its contribution?[5] We considered three possible policies: random, greedy, and learning. The random policy just picks a child at random. The greedy policy selects the child M_i with the highest preference value $p_i = \sum_{k=1}^{m} min(cr_{i,k}, rr_k)$, which measures how much resources M_i can contribute to the incoming task. For example, let the incoming task $T = \langle u = 100, rr_1 = 50, rr_2 = 150 \rangle$ and let manager M has two possible children M_1 and M_2 where $availTotalResources(M_1) = \langle cr_{1,1} = 200, cr_{1,2} = 0 \rangle$ and $availTotalResources(M_2) = \langle cr_{2,1} = 0, cr_{2,2} = 200 \rangle$. Then $p_1 = 50$ and $p_2 = 150$, hence M will select M_2.

In the learning approach, we used the Q-learning algorithm (Sutton & Barto 1999) with neural nets to approximate action values. Unlike value or policy iteration, Q-learning is a model-free algorithm that does not require an environment model. Q-learning also learns in an incremental manner; as an agent gains more experience, its performance improves. This is important in domains containing huge number of states, many of which will not be visited.

We used a decaying exploration rate to select actions so that agents explore less as they gain more experience. We also used a separate neural net for each action. This uses more memory space, but provides better approximation.

In reinforcement learning, rewards determine what an agent learns. From Algorithm 1, intermediate rewards are small negative rewards to reflect the communication and the processing costs of each additional step spent forming the coalition. Once a manager M successfully allocates a coalition to task T, it gains a reward equal to T's utility.[6] Note that even if T is a subtask of another task T', the rewards received by M are independent of whether the coalition formation for T' will succeed or not. This *recursive optimality* speeds up learning, while not affecting the quality of the formed coalitions.

wasted excess of resources.

[5] A more sophisticated decision process would consider parallelism. Here we focus on strictly serialized orderings only.

[6] We can implicitly indicate our preferences by modifying the reward function. For example, in (Shehory 1998) the author prefers coalitions of smaller size. This can be achieved by adjusting the reward function accordingly (e.g., dividing the utility gained by the size of the coalition formed).

We explored several techniques to speed up learning further. One technique involved minimizing the input fed to each neural net. The key observation is that the value of choosing a child M_i depends mainly on M_i's state, and to a lesser extent on the other children's states. We also tried using eligibility tracing, but the learning algorithm often diverged so this approach was dropped.

Organization Structure If we view the underlying organization as a search tree, our distributed algorithm searches the same search tree several times for each task and for each episode. Each time, the search has a different start state (where and when the task is discovered) and different goal state (the set of individuals — leaves — that form the coalition.)

To optimize performance, not only do we need to find a good *search mechanism*, but we also need to find an *organization* that for a specific environment model and agent population yields the best performance. In other words, we are modifying the search tree so that the search mechanism can perform better. The closest analogue in classical AI is the use of macro operators, which adds edges to the search tree to speedup the search. In our case we have more flexibility, as we can modify the search tree in whatever way we see appropriate. In our experiments we verify this by testing different organization structures of the same agent population and same tasks distribution, as we describe in Section Results.

Experiments and Results

Setup In our experiments, we wanted to know if using an underlying organization improved the system's performance. To do so, we compared our approach to centralized (a single manager controlling all individuals) random policy (CRP) and to centralized greedy policy (CGP). We also investigated the effect of learning in an organization by comparing three local policies: distributed learned policy (DLP), distributed random policy (DRP), and distributed greedy policy (DGP). Finally to measure the effect of the organization structure on system performance, we collected results using different organizations, all constructed from the same population of individual agents as shown in Figure . More details can be found in (Abdallah & Lesser 2004).

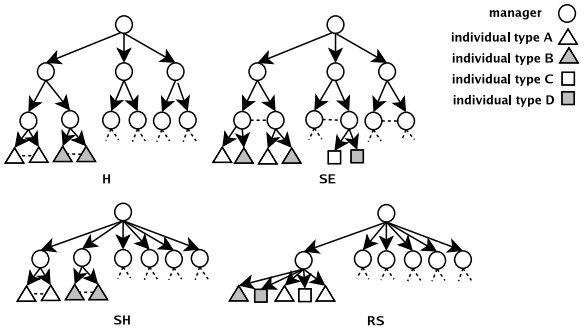

Figure 3: Different Organizations.

Results for every organization/technique combination

were computed over 10 simulation runs. Each simulation run consisted of 30,000 episodes. Seven tasks arrive at every episode and are randomly picked from a bag of tasks (to simulate a stable environment). Tasks in the bag are generated randomly such that each task requires between 4 and 10 agents to be accomplished. At any episode, the resources required by arriving tasks exceed the resources available to the system.

Our experiments focused on a population of 40 individuals and 10 managers so we can easily hand code different organization structures and study their effect. However, to verify the scalability of our approach, we also tested a population of 90 individuals and 13 managers.

Results Figure 4 illustrates how the performance of our system improves as agents gain more experience. CRP achieved least average utility. DRP performed better than CRP.[7] CGP is better than both. Our approach, DLP, outperformed all other policies for all organization structures, except when using a random organization structure. Interestingly, DGP performed worse than DRP and DLP in all organizations except RS, where it performed better than both.

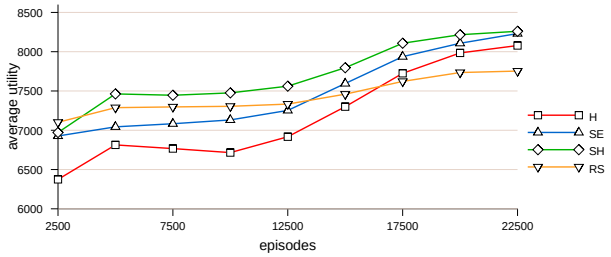

Figure 4: Learning curve.

This reinforces our belief that organization structure does affect performance. Learning the local policy lessens this effect, but state abstraction and task decomposition remain sensitive to the structure of the organization. For the abstraction and task decomposition algorithms that we used, if agents are randomly organized, little can be gained by learning. In our experiments with larger agent population (90 agents), DLP was better than other policies, achieving 35% more utility than CRP and at least 20% better than DRP and DGP.

More importantly, DLP is more stable than other approaches. The standard deviation (of achieved utility) using CGP was 70% worse than DLP with SE organization. CRP was 30% worse than DLP. Also DGP was the worst for all organizations except RS. We had similar results with the larger agent population. DLP had the least standard deviation, which was around one third that of DGP.

Centralized approaches exchange fewer messages. Still, learning the local decision reduces the number of exchanged messages. CGP wasted 20% more resources than DLP, while CRP wasted 40% more. We got similar results for the larger agent population.

[7] We believe this is due to the goal decomposition component of the organization, which encodes part of the domain knowledge.

A Mediator-Free Document Retrieval System

System Architecture

This section presents a document retrieval architecture for the grid computing environment. In such a system, we assume that agents are connected to each other in a peer-to-peer fashion. An agent at any time can generate queries into the Grid in order to find relavant documents. In the absence of a mediator, agents must cooperate to forward the queries among themselves so as to locate appropriate agents, rank the collections, and finally return and merge the results in order to fulfill the information retrieval task in a distributed environment. Figure 5 illustrates part of a document retrieval system in Grid computing that could comprise thousands of agents. Each agent is composed of five components: a collection, a collection descriptor, a search engine, an agent-view structure and a control center. The collection is a set of documents to share with other peers. Collection descriptor can be considered as the "signature" of the collection. By distributing collection descriptors around, agents can have better knowledge about how content is distributed in the agent society. Specifically, in this system we use collection model as collection descriptors. A collection model is the language model built for a particular collection. It characterizes the distribution of the vocabulary in the collection. The language model concept was originally introduced in information retrieval research (Ponte & Croft 1998) and has proven effective in the distributed IR field (Callan 2000)(French *et al.* 1999). It has many interesting properties which are easily exploitable in the Grid computing environment: first, a collection model is lightweight since it significantly condenses the description of the content of the collection and thus is much smaller in size compared to the collection. Additionally, the size of the collection model grows minimally with the size of the document collection. Secondly,the collection model is a relatively accurate indicator of the content of the collection. The agent control center is the unit that accepts user queries and is also responsible for performing the distributed search algorithm. The local search engine allows each agent to conduct a local search on its document collection so as to determine whether there are any documents that meet the criteria of a specific user query and then return relevant documents. The agent-view structure, also called the local view of each agent, contains information about the existence and structure of other agents in the grid and thus defines the underlying topology of the agent society.

The rest of the section will present the framework in more details from the formulation and evolvement of the agent-view structure and the distributed search algorithm

Agent-View Algorithm

A common approach to forming an initial agent-view, as used in the Gnutella system, is for agents (when first joining the system) to initiate a discovery protocol by sending out

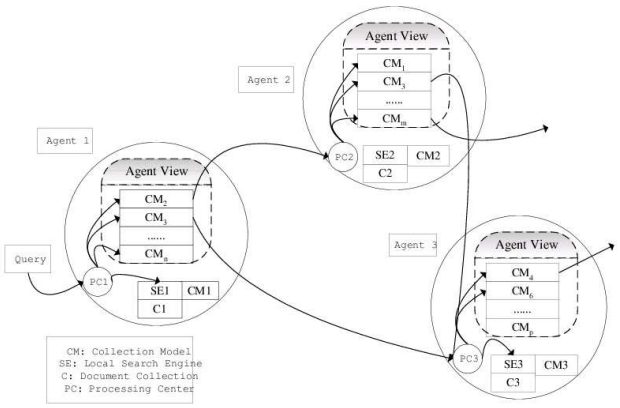

Figure 5: A Mediator-free Document Retrieval System.

ping and pong packets with their IP addresses. In our system, we slightly modify this approach by also transmitting the collection model of the agent. This agent discovery procedure results in a random-graph like topology being constructed, where each agent in the grid establishes an agent-view structure with the collection models and IP addresses of its neighboring agents. However, one obvious drawback of such a topology is the lack of search efficiency since there is no connection between the agent-view and how to effectively search for relevant documents. Therefore, we propose an agent-view reorganization algorithm (AVRA) based on the initial agent-view. The goal of AVRA is to create an agent-view that contains agents whose content is similar, implicitly creating semantically close agent clusters. For example, we can have a "sports" cluster, an "economics" cluster, and so on. These clusters are not disjoint which means that an agent can belong to multiple clusters. For example, an agent can belong to both a "basketball" cluster and a "college" cluster based on its content. Clusters are connected to each other, so if a query is issued to an agent without many relevant documents it can be routed swiftly to the appropriate clusters. To this end, agents exchange their local agent-views to expand the scope of their local agent-view so that each agent is more informed about the content distribution over the entire network. Specifically, each agent decides locally which agent in its agent-view to interact with so as to construct in a directed way an expanded view. The decision to expand along a particular direction results in sending an *Expand?* message to appropriate agents. The *Expand?* message includes the address of the sending agent so that the target agents can send back the answer. Upon receiving the *Expand?* message, each agent sends its own agent-view to the requesting agent. Such communication augments each agent's local view with further information about the content distribution in the agent society, and allows the agent to make a more informed decision about whom to forward queries. The agent-view reorganization algorithm then prunes the topology implied by the agent-view so that the agent-view does not become unreasonably large leading to the same scalability issues found in a centralized mediator architecture. One major concern of the AVRA algorithm is connectivity. Pruning the agent-view without caution can result in very poor connectivity, which in turn reduces the number of the agents that can be reached and thus decreases overall system performance. Additionally, agents in P2P networks often vary in their capacity in terms of the maximum number of outgoing connections they can maintain. Therefore, throughout the reorganization process, we keep the agents out-degree unchanged. Meanwhile, the in-degree of each agent is checked from time to time to ensure that it can be reached by other agents. The algorithm works as follows:

For each agent $A_i$in the system, we calculate its similarity $W_{cc}(A_i, A_j)$ with the neighbors A_j. After ranking, agent A_i probes its most similar K neighbor agents with *Expand?* messages. In our experiment, we use $K = 1$ to reduce communication. Upon receiving *Expand?* messages, each agent responds with its current agent-view. To prevent the same agents from being chosen repeatedly and thereby slowing convergence, we specify that no agent can be picked more than twice in three consecutive rounds. This heuristic helps an agent construct a more encompassing view so that the reorganization process proceeds smoothly. After expanding its view as a result of interacting with its neighboring agents, agent A_i prunes its agent-view according to the following rules:

(1) $M\%$ of its degree are designated as its most similar neighbors while the rest $(1 - M\%)$ neighbors are randomly chosen from the agent-views it has collected. This randomization has the effect of maintaining connectivity in the agent society. As experiments show, if all the neighbors are chosen from the most similar agents (M is 100%), the resultant network suffers from poor connectivity. Specifically, it contains many separate "clusters" though these clusters are quite semantically close. After testing different values, we set M to 80%.

(2) If the number of incoming connections (in-degree) of Agent A_i falls below 2, an empirical threshold which indicates whether this agent is easily reached by the outside world, then the agent contacts its neighbors to request that they add it as one of their neighbors in their local agent-view.

Distributed Search Algorithms

The distributed search process is initiated when an agent receives a query. The agent then needs to make a number of local decisions such as whether it should perform a local search to see if it can satisfy the query locally, whether it should forward this query to other agents and to whom, or whether it should drop the query. During this process, if an agent receives a query that it previously processed, it simply skips this message as Gnutella does. Otherwise, the search continues in the network until all the agents receiving the query drop the message. There is no explicit recognition by individual agents that query is no longer being processed by any agent in the network.

The next two subsections propose two search algorithms, namely, k Nearest Neighbors (kNN) collection model-based approach and Gradient Search Scheme (GSS), to take advantage of collection models and the reorganized topology. We define agents with local documents that are relevant to

the query as "relevant agents" and "irrelevant agents" otherwise. As the agent reorganization algorithm aims to cluster the agents by content, the key to the search algorithms performance is to direct the queries to the relevant agents cluster swiftly.

We introduce two concepts to facilitate the description of the search algorithms.

Definition 1: Covered Agents Level (CAL) is defined as the number of agents visited during a query search divided by the size of the agent society.

$$CAL_n = n/N$$

Definition 2: Cumulative Recall Ratio (CRR) for a query after n agents are searched is defined as

$$CRR_{q_i,n} = \frac{\sum_{j=1}^{n} r_j}{R_{q_i}}$$

Here R_{q_i} is defined as the total number of relevant documents located in the entire network for the query q_i, and r_j is the number of relevant documents located at agent j. CRR is used as a metric to measure the performance of a distributed search algorithm in relationship to its CAL.

kNN collection model based approach. The collection model is a stable representation of a collection. This insight leads to an intuitive distributed search scheme. An agent first determines if the cluster it belongs to is a "relevant agent zone" or "bad agent zone" by comparing the similarity of the collection it hosts with the query qi, i.e $W_{cq}(A_i, q_i)$ and a threshold $Tsim$.[8] Specifically, the algorithm works as follows:

(0) If an agent Ai receives a query q_i, A_i would drop the query qi if it has been processed previously, otherwise calculate the similarity $W_{cq}(c_i, q_i)$

(1) If $W_{cq}(A_j, q_i)$ is above threshold T_{sim}, A_j is likely to be located in a "good agent zone" . In this situation, the agent computes the similarity of its neighbors A_j and the query q_i, i.e $W_{cq}(A_j, q_i)$ and select the k agents with highest $W_{cq}(A_j, q_i)$ value to forward the query. However, in practice, we face the same situation as in the agent-view reorganization process. If we forward all the queries to agents highly-similar with the initiator, then the most similar agents tend to receive the query repeatedly since they form a clique; thereby lowering the chance that other agents are examined as part of the search process. This leads to a low CAL value when the search is completed which motivates an alternative strategy. Thus, Instead of forwarding queries solely to highly-similar agents, we also forward queries to some high-degree agents. Researchers (Walsh 2001) have found that the high-degree agents are of special importance in the distributed search algorithm. After testing different parameters, we use the top 20% highest-degree neighbors and the top 40% most-similar agents for forwarding the queries.

(2) If $W_{cq}(A_j, q_i)$ is below T_{sim}, then the agent considers itself as part of an "irrelevant agent zone". It then tries to expand the search so as to get out of the "irrelevant agent zone" by forwarding the query to high-degree agents rather than highly-similar agents.

[8] After testing different values, we set T_{sim} as 0.15.

Gradient Search Scheme. The Gradient Search Scheme(GSS) differs from the kNN approach in how it deals with the situation when the initiator is in a "bad agent zone". As in reality, most of the agents in the grid computing environment are irrelevant to the queries. Therefore, the strategy of how to deal with queries starting from irrelevant agent zones is crucial to the search performance. Though the kNN collection model algorithm is designed to direct queries out of "bad agents zone" as soon as possible, experimental results show that the system performance is still very sensitive to where the initial search is originated; the kNN approach suffers from a drastic decrease in performance when it is initiated from irrelevant agents since it still takes a long time for the query to reach relevant agents. The GSS addresses this issue by first trying to locate an appropriate agent for initiating the search for the given query by distinguishing between good and bad starting agents based on the similarity value between the query and agent as the kNN approach does. If the initial agent is good, the Gradient algorithm simply follows the kNN collection model algorithm. Otherwise, the algorithm starts a gradient search process to find a new originator for this query. The detailed protocol works as follows:

(0) If an agent A_i receives a query q_i, A_i would drop the query q_i if it has been processed previously, otherwise calculate the similarity $W_{cq}(c_i, q_i)$ of the collection on Ai and the query q_i.

(1) If $W_{cq}(c_i, q_i)$ is above a certain threshold T_{sim}, it indicates that the agent is a good candidate for the query. The algorithm then follows the kNN collection model algorithm.

(2) Otherwise, for each neighboring agent A_i, pick the neighbor B which satisfies $argmax W_{cq}(c_j, q_i)$. A message is sent from A to B with the value $max W_{cq}(c_j, q_i)$.

(3) Step (2) is repeated N times. At each round, the old values and the new value are accumulated in the message that is then forwarded to the next node. For example let us assume that agent P is selected after N rounds. The message P receives will contain N maximum similarity values generated as the result of previous rounds. P will then pick the agent with the highest similarity value as the new originator to restart the search using the kNN algorithm. There is a trade-off involved in determining the value for N; the bigger the value of N the more likely that a good originator will be found while the smaller the value of N the quicker the search for relevant documents will begin. Considering these two factors, we used a value for N of three in our experiments.

Experiments and Results

Setup Similarity measures are heavily used in both the AVRA and the distributed search algorithms. In our framework, both collection models and query models are treated as language models, and therefore, distributions. We use Kullback-Leibler (KL) divergence to measure the distance between collection models or between collection models and query models. The detailed computation formula can be found in (Zhang *et al.* 2004).

We created an experimental agent network as described in (Zhang, Croft, & Levine 2003) and distribute TREC VLC1

collections to the agents. TREC VLC1 is split to 921 sub-collections largely by source and therefore is denoted by TREC-VLC1-921. The statistics about TREC-VLC-921 can be found in (Callan 2000). We run query set $301 - 350$ on TREC-VLC1-921.

After the underlying topology and collections are distributed to the agents, we reorganize the topology with parameter $K = 0, 3, 10$ respectively. To perform the search algorithm, we randomly pick agents as originators. In this experiment, we examined the performance of 50 queries for each combination. Each query was repeated 50 times. By averaging 50 results for each query, there was more than 95 percent confidence interval.

Results The detailed results are available at (Zhang *et al.* 2004). The results show that, as expected, the central KL approach with its global view consistently outperforms the other three approaches. They are consistent with the conclusion that the collection model is a stable indicator of the collection. Correspondingly, both the GSS and kNN algorithms, which take advantage of collection models, significantly outperform the random search scheme when using the same underlying topology. When a topology with $K = 0$ is used, the cumulative recall ratio of kNN and GSS are significantly better than the random approach when CAL is below 50%. Another observation is that when CAL is low, which would be the expected case in real networks, GSS further improves the performance of kNN. However, with the increase of CAL this gain diminishes. For example, when CAL is above 30%, the difference between kNN and GSS is indistinguishable. Therefore, we conclude that although the GSS benefits from a good starting agent, this impact fades as more and more agents are reached. We can also gain the insight from the results that the importance of topology reorganization is dependent on the specific search algorithms used. There are no obvious gains from the AVRA algorithm for the random search strategy as the latter does not take advantage explicitly of collection models. On the other hand, there is considerable performance improvement using the GSS when K increases, ranging from $10\% - 30\%$ when $K = 3$. When K further increases, the performance benefits are more obvious. The fact that AVRA brings more benefits to the GSS than kNN search is based on GSS's ability to relocating to good originators sooner and the new originators are often surrounded by many other good agents. Performance results stabilize when $K > 3$. Therefore, we conclude that the local agent-view reorganization process tends to converge after three rounds. Of course, this number may differ with various applications and network sizes.

Conclusion

In this paper we discussed our solutions to a resource allocation problem, involving coalition formation, and a content retrieval problem, involving locating relevant documents. Our results verified the important role of organization on the performance of a system. In coalition formation, learning to work in an organization outperformed the unorganized solution. It achieved higher average utility (20-35% more than non-organized approaches) and was more stable (30-70% less standard deviation). Similarly, reorganizing clusters of relevent document proved crucial, with 10-30% increase in system performance. In future, we plan on applying our ideas in the Grid computing domain.

References

Abdallah, S., and Lesser, V. 2004. Organization-Based Coalition Formation. *UMass Computer Science Technical Report 2004-04*.

Callan, J. 2000. *Distributed information retrieval*. Reading, Massachusetts: Kluwer Academic Publishers.

Czajkowski, K., and et al. 2001. Grid information services for distributed resource sharing. *Proceedings of the 10th IEEE Symp On High Performance Distributed Computing*.

French, J. C.; Powell, A. L.; Callan, J. P.; Viles, C. L.; Emmitt, T.; Prey, K. J.; and Mou, Y. 1999. Comparing the performance of database selection algorithms. In *Research and Development in Information Retrieval*, 238–245.

Ponte, J., and Croft, B. 1998. A language modeling approach to information retrieval. In *In Proceedings of SIGIR, pages 275–281, 1998*.

Shehory, O. 1998. Methods for task allocation via agent coalition formation. *Artificial Intelligence Journal*.

Sutton, R., and Barto, A. 1999. *Reinforcment Learning: An Introduction*. MIT Press.

T. Sandholm, e. a. 1999. Coalition structure generation with worst case guarantee. *Proceedings of the Third Internation Conference on Autonomous Agents*.

Walsh, T. 2001. Search on high degree graphs. In *In Proceedings of International Joint Conference on Artificial Intelligence*, 266–274.

Zhang, H.; Croft, B.; Levine, B.; and Lesser, V. 2004. A multi-agent approach for peer-to-peer based information retrieval systems. In *University of Massachusetts, Amherst, MAS Technical Report, Submitted to AAMAS 2004*.

Zhang, H.; Croft, B.; and Levine, B. 2003. Efficient topologies and search algorithms for peer-to-peer content sharing. In *University of Massachusetts, Amherst, CIIR Technical Report IR-314*.