GUARD: Gossip Used for Autonomous Resource Detection

Sagnik Nandy   Larry Carter   Jeanne Ferrante

Department of Computer Science and Engineering

University of California at San Diego

{snandy, carter, ferrante}@cs.ucsd.edu *

Abstract

A growing trend in the development and deployment of grid computing systems is decentralization. Decentralizing these systems helps make them more scalable and robust, but poses several challenges. In this paper we address one such problem - that of locating computing resources meeting specified requirements in a large scale heterogenous system. The heterogeneous and dynamic nature, coupled with their multiple occurrences of these resources, makes the problem distinct from traditional data location problems found in the context of content-sharing systems. We propose GUARD (Gossip Used for Autonomous Resource Detection), a protocol that uses gossiping between neighbors to propagate the current knowledge of distances from available resources. GUARD is *autonomous* (all decisions are made locally, using knowledge based only on interaction with immediate neighbors) and does not make any assumptions about the underlying network topology. Our simulations show

*This work was supported in part by NSF grant ACI-0234233

GUARD is more efficient than other techniques such as random routing, history-based routing and frequency-based routing that have been used for similar purposes. We also show how GUARD can be modified to locate multiple categories of resources meeting multiple criteria.

Key Words : Dynamic resource location, autonomous protocol, scalability.

1 Introduction

The growth in size and popularity of the Internet has given birth to wide area computing efforts such as [24, 14, 3, 1]. The computing power of several thousands of machines are now being combined to match the strength of super-computers[1]. As these systems grow in size, it becomes increasingly difficult to control and monitor them using a client-server like architecture. This has led to the decentralization of these systems, shifting them from strictly centralized systems [24, 3, 23] to hierarchical [4, 8, 1] and even completely decentralized systems [26, 5, 9, 12, 16, 17, 15]. While decentralization helps in issues of scalability and deployment, it brings several new challenges. One is *resource location*. This is the problem we address.

Computing systems may be comprised of several resource categories (storage, processing speed, memory, operating system, supporting software etc.) that are available to users. To use these resources it is necessary to locate them. In a centralized system, a central server can keep track of resources [13, 6] and nodes requesting them can query the server to get access. In the absence of centralized control it is unclear which layer monitors the resources and how one should route to a desired resource efficiently. Note also that distributed computing environments are more challenging than distributed data-centric systems, since the former must deal with heterogeneity, multidimensional queries

[1]E.g., SETI@home operates at 50 TFlops, while IBM Blue Horizon is about 36 TFlops.

(requests for a resource that meets several properties simultaneously), and more types of resource properties.

We propose GUARD(Gossip Used for Autonomous Resource Detection), a protocol that is designed to track dynamic resources in a heterogenous distributed system. GUARD works in a decentralized fashion where each node only interacts with its immediate neighbors, making all its decisions locally. The protocol is generic in nature and does not make assumptions about the underlying system topology.

The paper is organized as follows - Section 2 describes the GUARD protocol. Section 3 gives experimental results that highlight different aspects of GUARD. Section 4 is an overview of related work, we conclude in Section 5 with a summary of our findings and future research directions.

2 The GUARD protocol

We first describe the basic GUARD protocol, whose goal is to locate an instance of a given resource in a large heterogeneous system without the use of centralized control or information. We also derive an analysis of GUARD's performance, and describe how the protocol can be extended to locate resources that satisfy a set of properties simultaneously.

2.1 The basic GUARD protocol

GUARD uses distance vectors [18] to maintain *likely* distance from resources. Unlike the use of distance vectors in routing IP addresses, where the target of a node is a unique node, there are likely to be *multiple* resources satisfying a request in a large distributed system. GUARD maintains the distance from the closest node having a particular resource. This information is updated via "gossiping" (nodes exchange information with their neighbors periodically) to reflect the consumption/addition/deletion of resources in

the system. Nodes in the system only interact with their immediate neighbors, and do not communicate directly with any other node.

We will first introduce the various terms used in describing the protocol. We assume that the underlying system is represented as a graph $G = (V, E)$ with $N = |V|$ nodes. We shall initially assume that there are K types of trackable resources $(R_1, R_2, ..., R_K)$ in the system, but later show how GUARD can be used for resources, such as memory, that can be requested in different sizes. Nodes make requests for one of the K types of resources (Section 2.3 shows how GUARD can handle multidimensional requests) and the protocol tries to locate the closest node that satisfies the request.

The way GUARD works is simple. Each node in the system maintains a table of size K to track down the K different resources (we show in Section 2.3 how the table size can be reduced). The entries in the table are of the form $< D_i, nbr_i >$ $(i = 1, 2, ..., K)$, where D_i is the node's current knowledge on the number of hops that need to be travelled to reach a resource of type R_i, and nbr_i is the neighbor with the least value of D_i. Initially all nodes have their tables initialized with all D_i's set to ∞. If a node itself has any of the resources R_i, it sets the value of D_i in its table to 0. Periodically nodes *gossip* by sending a copy of its table of D_i's to its neighbors. Whenever a received value D_i' from neighbor k is less than the node's own table entry, the node updates its table to be $< D_i' + 1, k >$.

On receiving a request for R_i, a node checks if its value of D_i is 0 (implying it has the resource), in which case it services the request and sets the value of D_i to ∞ (similarly a node resets the value of D_i to 0 once the resource is released). If the node doesn't have the resource it forwards the request to nbr_i. A request that is forwarded more than TTL (set to 100 in our experiments) is killed.

2.2 Analysis of the GUARD protocol

GUARD uses the distance vector approach that has been widely used in the networking community. While this approach is unscalable for its traditional use of locating *unique* nodes in an Internet-like scenario, it is extremely well suited for our *resource location* problem when the number of resources needed in a single application is fairly limited. Also, there are likely to be multiple nodes that might satisfy a particular request. This makes it easier to propagate resource information in the system (since updates in a node's status will only affect those nodes that were using it as a closest resource source) and the problem often reduces from locating a unique node (routing in the Internet) to locating one of several satisfying nodes. This makes the distance vector approach more conducive to the dynamic resource location problem.

We now give a theoretical estimate of GUARD's performance. The estimate is an upper bound that allows users to determine whether such an approach is suited for their system and/or to tweak the protocol's parameters to meet their requirements.

We begin by introducing the terms used in our analysis. Let F_u be the frequency at which nodes gossip about their information and F_r be the frequency at which requests are made in the system. Resources are borrowed for an average time length of L, i.e. a resource once allocated remains unusable on average for L time. P_r denotes the percentage of nodes in the system that initially have a resource of size r and are willing to share it.

Suppose a request for resource r ($r \epsilon \{R_1, R_2, ..., R_K\}$) originates in the requesting node, n_r, at time T. Let n_d be the node we expect to be the final destination of the query and l_{n_r} be the distance (in number of hops) between n_r and n_d, i.e. l_{n_r} is the value of D corresponding to resource r in n_r's routing table. For our derivation, we assume the latency of requests is negligible compared to the frequency of gossiping. We also make the uniformity assumption that resources and requests are distributed among all the nodes in a way that makes each resource be the "closest" resource to about an equal number

of requests. While this is not true for all topologies, it may hold for important scenarios including sensor-like graphs and existing peer-to-peer systems like CHORD [26] (and it makes our analysis possible).

Let $P(f_{r,T,n_r})$ = probability of failure of request for r at time T, starting at node n_r.
$\leq$ probability of r being taken in n_d in time $T - \frac{l_{n_r}}{F_u} \leq t \leq T$

- The $\leq$ inequality is used because the request might be satisfied by a different node.

$\leq$ probability that **any** request for r is serviced by n_d in time $T - \frac{l_{n_r}}{F_u} \leq t \leq T$
$\simeq$ 1 - $e^{\frac{-l_{n_r}.F_r}{F_u.N_{r_T}}}$, where N_{r_T} is the number of nodes having resource r at time T.

- By our uniformity assumption, the number of other requests that are issued in the time period $T - \frac{l_{n_r}}{F_u} \leq t \leq T$ and which are closest to node n_d is Poisson distributed with mean $\frac{l_{n_r}}{F_u}.\frac{F_r}{N_{r_T}}$. The above expression is 1 - (probability of **no** such request).

Now assuming we want the failure rate to be below a given fraction f_F we can write
1 - $e^{\frac{-l_{n_r}.F_r}{F_u.N_{r_T}}} \leq f_F$
$\Rightarrow e^{\frac{-l_{n_r}.F_r}{F_u.N_{r_T}}} \geq (1\text{-}f_F)$
$\Rightarrow e^{\frac{l_{n_r}.F_r}{F_u.N_{r_T}}} \leq \frac{1}{1-f_F}$
Now taking natural logarithm on both sides and simplifying we get

$$\boxed{F_u \geq \frac{l_{n_r}.F_r}{|ln(1-f_F)|.N_{r_T}}}$$

Now both l_{n_r} and N_{r_T} depend on several factors like the topology of the network, the values of P_r, L etc. but can be estimated, given a particular scenario[2]. This implies that given a particular setup one can simply tweak the gossiping frequency to achieve a desired rate of success. We verify these predictions in Figure 3 of Section 3.3.

[2]For most peer-to-peer topologies like CHORD [26], PASTRY [22] etc. l_{n_r} (bounded by the diameter of the network) grows logarithmically with the number of nodes and even if N_{r_T} is a small fraction of N, F_u can be reasonably small.

2.3 Modifying GUARD to locate group of resources

The GUARD protocol that we have described is capable of locating nodes having a specific resource. However, in a realistic scenario the application is likely to need resources meeting *multiple* criteria (e.g. a Pentium IV processor with 1 GB memory running Linux). While we can use the already described version of GUARD to locate individual nodes meeting these requirements, there is no guarantee that these resources will be in the same node.

One way to tackle this problem would be to track all possible request types. While this would solve the problem, it is impractical since even a small number of trackable properties can lead to a huge number of possible requests. We therefore consider a variant of the GUARD protocol, similar to a technique used for data-centric systems in [11], that populates the node tables with the most popular queries.

Assume that there are K different types of resources $< R_1, R_2, ..., R_K >$ (for simplicity we assume that a resource is Boolean, i.e. $R_i = 1$ if a node has the resource, otherwise it is 0). A user may want to locate a node having one or more of the K resource types. We use the following approach. Each node has a table of fixed size, but now the table entries are of the type $< request_type, distance, nbr, count >$ where *request_type* defines the set of types constituting the request, *distance* is the number of hops from the resource, *nbr* refers to the neighbor of the node to forward requests for this combination of resources, and *count* keeps track of how many times this particular resources type has been requested. Initially all the table entries are initialized with a single entry, with all fields corresponding to the resources the node owns set to 1 in the entry.

Whenever a node gets a request, it makes a table entry for the request (if it doesn't already have one), or increments the entry's *count* (if it does). Since the table is of finite size, an entry with the minimal value of *count* is evicted. If the node can satisfy the request itself, it does so. If its table has an entry that can satisfy the request or a superset, it forwards the request to the indicated node. Otherwise, it sets the distance in the table

to infinity, and the request fails. Via gossiping, table entries are propagated to neighbors and beyond. Eventually, a node is found with the resource and the gossiping results in table entries with finite distances.

The above modification to the protocol maintains the simplicity and autonomous behavior of GUARD while addressing the issue of multiple request parameters. The table entries get populated with the more commonly issued request types.

2.4 Non-Boolean resources

Some resources, such as memory, are not Boolean but come in different sizes. Any request for a particular size can be satisfied with a resource of greater size. In this case, GUARD has a finite set of Boolean requestable sizes (e.g. 64, 128, ..., 2048 MB). We used a "best fit" strategy, where if a node cannot satisfy a request itself, it forwards it to the node in its table that has the smallest acceptable value (even if that node is further away than one with a larger resource). Other strategies, such as "closest acceptable node", are possible and worth investigating.

3 Experimental Results

In order to test the effectiveness of GUARD we built a simulator that allowed us to experiment with several aspects of the protocol. We tested the protocol for two sets of topologies - (i) sensor-like[3] networks (SN) where we randomly scattered the nodes over a square grid and then connected all nodes within a given radius, and (ii) tree-shaped networks (TN) that reflect the hierarchical topology of networked clusters. Each experiment had a *running* time and a *startup* time. The startup time was provided to have the system reach steady state. For all experiments the startup time was fixed at

[3] We anticipate that one of the major uses of large scale decentralized computing will be sensor networks.

1000 seconds of simulation time. This value was chosen as sufficient in our experiments for the performance to stabilize to a steady or periodic behavior. The running time, after startup, was set to 10,000 seconds.

We begin by choosing plausible values of the parameters for our initial set of experiments. (In Section 3.2 we will show how some of these parameters affect the performance of GUARD.) The requests were distributed randomly across the lifetime of the experiments. Resources once claimed remained with the requesting node (the duration of a task was distributed randomly between 15-30 minutes). Not all nodes in the system shared their resources (to make our simulator more realistic) and the percentage of nodes that shared their resources p_s was set to 70% for the initial experiments. Nodes updated their information once every 50 seconds. For the initial experiments we considered the resource to come in six different sizes. Each of the nodes sharing their resource with others had a resource of one of these sizes with the probabilities {*.05,.1,.3,.3,.15,.1*} and the requests for these resources had a probability of {*.05,.1,.2,.35,.15,.15*} (chosen to make some of the less available resources more heavily requested).

We now discuss the changes to these setups for the individual experiments and the results we observed.

3.1 Performance of GUARD

The main test for a protocol is its performance. We tested the performance of GUARD by varying several parameters to test its efficiency and feasibility. To test the basic protocol we created systems of size N = 1000, 2000, 4000, 8000, 16000 and 32000 for both the topologies *SN* and *TN*. For each experiment, the number of requests was $2N$ and the requesting nodes were chosen randomly from all the nodes. We compared GUARD to three other protocols used for similar purposes described in [10], namely:

- **RAND**: In this protocol nodes randomly routes the request to one of its neighbors until it is successfully answered or it reaches a maximum distance TTL (for our experiments we set $TTL = 100$).
- **HIST**: Nodes keep a history corresponding to each resource type, tracking the neighbor that successfully routed it the last time. Requests for a resource are routed for a maximum of TTL times by this method, thereafer, RAND is used for up to an additional TTL steps. On successful completion of a request, all nodes on the path updated the entries to show the successful route.
- **FREQ**: Nodes keep track of the number of requests (irrespective of the resource type) that each neighbor answered. Requests are routed to the most successful neighbor. Like HIST, on failure the protocol resorts to RAND.

The results for Boolean resources are shown in Figure 1 (a) to (d). We evaluated performance based on two criteria - (i) the percentage of requests that were successfully answered (a failure means there existed a resource of a particular size but the protocol failed to locate it; if there is no such resource, it is not counted either way) and (ii) the average number of hops taken to reach the resource.

One can see that GUARD significantly outperforms the other protocols in the percentage of requests answered, and it always uses a smaller number of hops. GUARD is scalable and performs well for both topologies and all graph sizes used. Intuitively, this improvement comes largely because the distance vector approach helps to identify the *nearest* resource while the other protocols target "any resource". Though GUARD spends effort in periodically updating its information, this effort helps in achieving significant benefits. It must also be mentioned that the small value for the average number of hops needed by GUARD does not mean it wasn't suitable for locating distant resources. In our experiments GUARD worked successfully even when the nearest resource was 30-40 hops

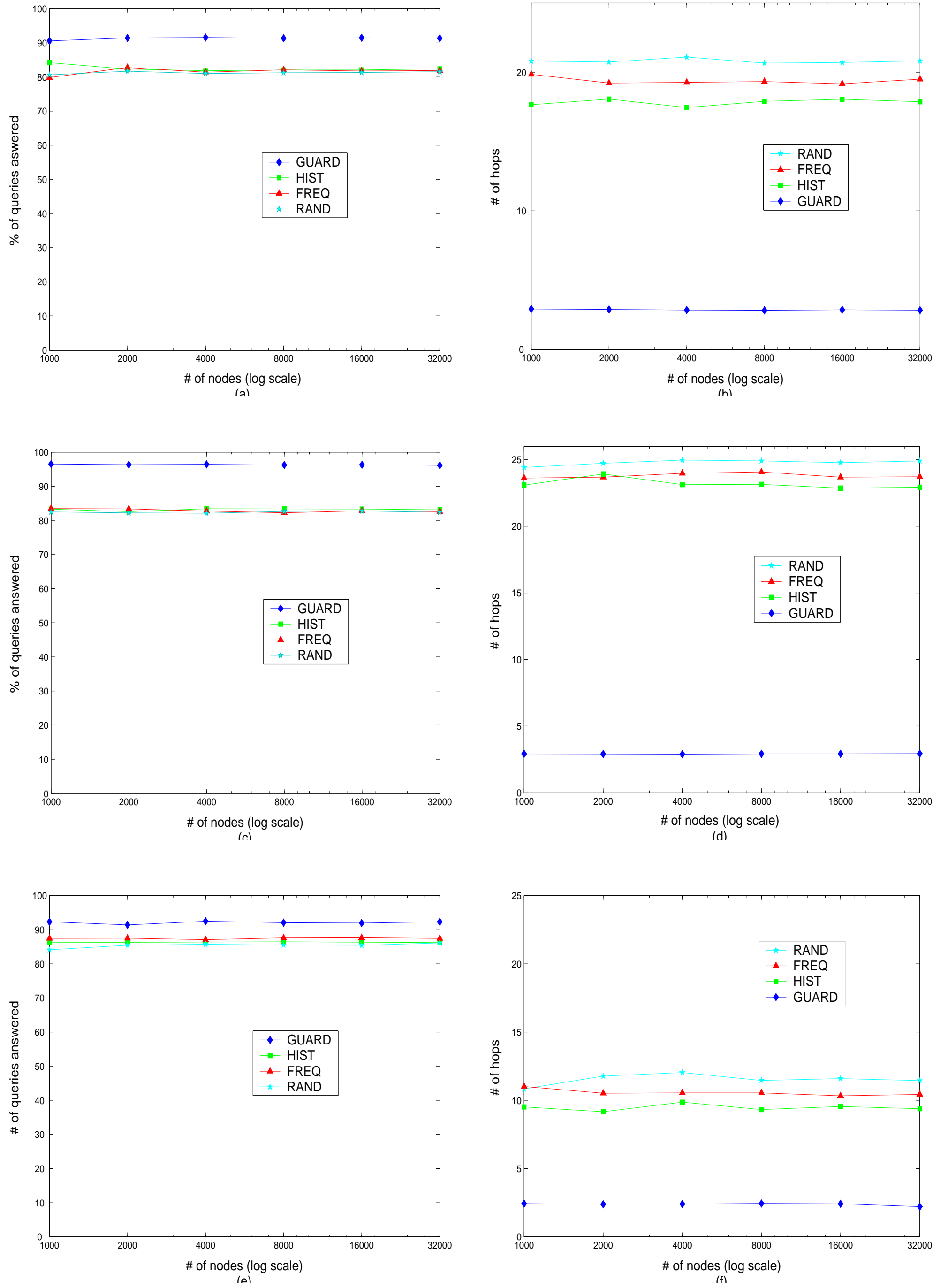

Figure 1: Performance of GUARD in comparison to other protocols. (a) % of successful queries for SN (c) Avg number of hops for SN (c) % of successful queries for TN (d) Avg number of hops for TN (e) % of successful queries for SN with resources of different size and (f) Avg number of hops for SN with resources of different size

away.

The results in Figure 1 (e) and (f) use variable-sized resources. One can see that even though the performance of RAND, HIST and FREQ improves both in terms of percentages of requests answered and number of hops taken to locate the resources, GUARD still consistently outperforms these heuristics. This suggests that GUARD can be used to locate both resources having a Boolean value (like OS, software, architecture etc.) or resources that can satisfy a range of values (memory, storage etc.)[4].

3.2 Effect of various parameters on GUARD

To test the effect of various parameters on GUARD's performance, we used the setup of the previous experiment for N=4000 and varied one of the parameters, keeping the others constant. The parameters we studied were (i) the percentage of nodes sharing their resources, (ii) the inter-update time at which nodes update their information, (iii) the number of requests made and (iv) the average duration of a task. We conducted the experiments for both topologies and the results are shown in Figure 2.

3.3 Performance of GUARD vs predicted performance

To test the validity of the analysis presented in Section 2.2 we calculated the failure rate of the experiments mentioned above and compared it to the upper bound provided by our analysis. The value of N_{r_T} was calculated by the simulator using the actual number of resources of a size that was left in the system[5]. We have provided the results for both the topologies and have also provided the ideal curve (where the predicted rate equals the observed rate) for comparison in Figure 3.

[4]The dividing of resources like memory, storage etc. into smaller categories can lead to defragmentation of the resource and has not been dealt with in this paper.

[5]This might be difficult to do in an off-line context but can be estimated using an expected steady state value based on the rate of requests and the rate of release of resources.

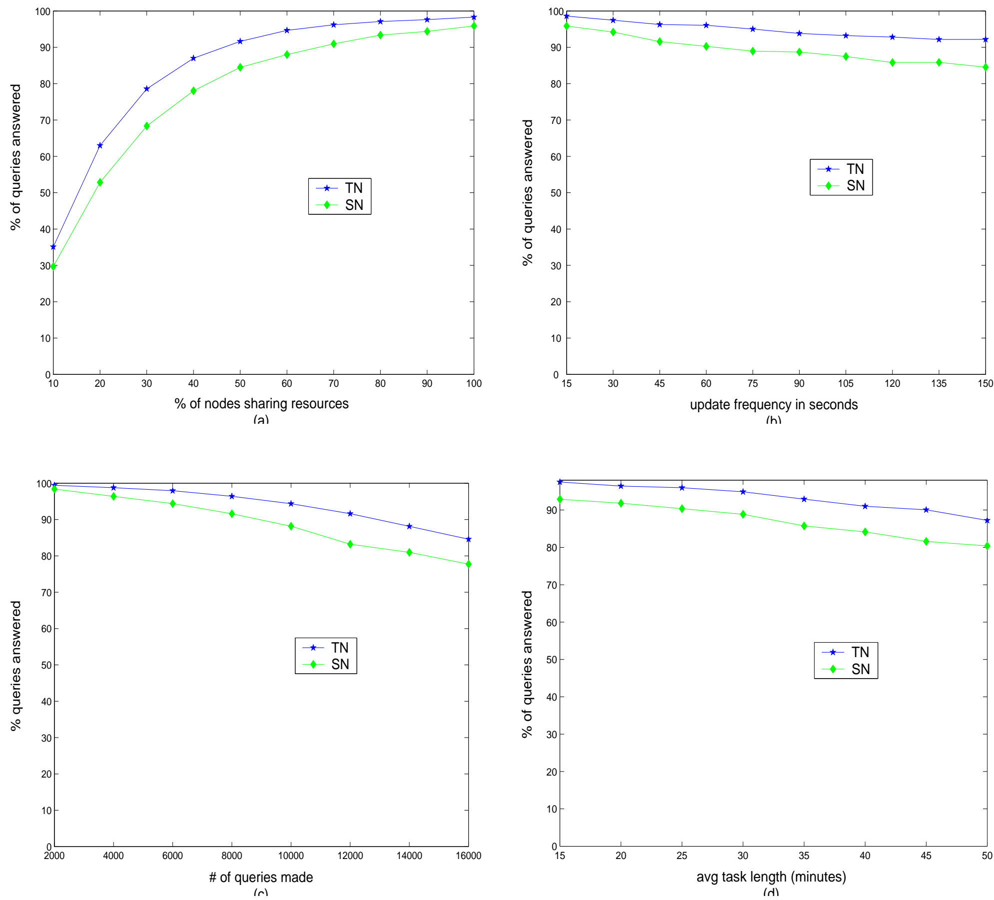

Figure 2: Effect of various parameters on GUARD. (a) percentage of nodes sharing their resources (b) inter-update time (c) number of requests and (d) average task length

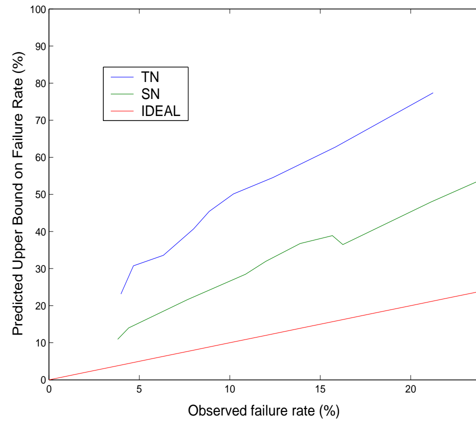

Figure 3: Comparison of upper bound on failure provided by our analysis vs. actual failure rate

We can see that the observed failure was always bound by the upper bound we provided. This shows that the analysis does indeed provide a correct upper bound that can be used to determine a desired value of F_u. It can also be seen that the predicted rates are almost a constant factor higher than the observed rates (this factor varies for the two topologies). We conjecture this is due to the possibilities of multiple paths to destination nodes and the possibility of multiple destination nodes. Moreover, the l_{n_r} term appearing in the exponent is representative of the worst case scenario. In case one desires a tighter upper bound they can use the average value of $\frac{l_{n_r}+1}{2}$ for the analysis instead.

3.4 The revised GUARD protocol

We now evaluate the performance of the revised version of the GUARD protocol where nodes can request a combination of the K resources and the lookup table has a limited size. Like the previous experiments each node had a 70% probability of having a resource. All combination of requests for a given number of resources occurred with equal probability. We tried experiments with K=5 and 6. For K=5 the requests comprising of 1-5 resources appeared with probabilities of {*.35,.28,.21,.11,.05*}. For K=6 these probabilities were {*.30,.27,.22,.08,.08,.05*}. We ran experiments for both the topologies by varying the size of lookup table (note: for size=2^K the protocol reduces to the standard version of GUARD, except for the startup time taken to propagate the requests). The results are given in Figure 4.

We observe that the modified version of GUARD performs well: Even for small sizes of the table, GUARD manages to answer more than 80% of the queries successfully. This is comparable to the performance of the other protocols observed earlier in terms of the success rate, while GUARD uses many fewer hops to answer these requests[6].

[6] HIST, which was the second most efficient in terms of number of hops is likely to run into the same problem of a limited size lookup table.

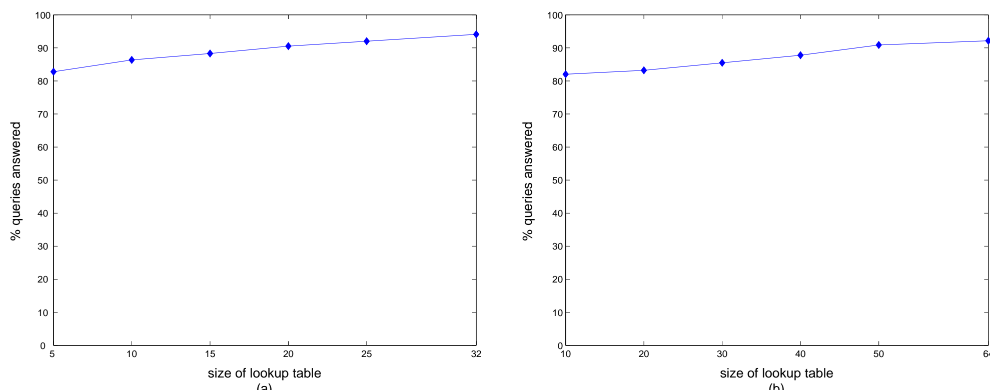

Figure 4: Performance of the revised GUARD protocol for (a) 5 resource types and (b) 6 resource types.

4 Related Work

There has been a large volume of research addressing the issue of locating resources in a distributed system in a decentralized fashion (see for example [26, 22, 9, 5]). However, all these efforts try to locate unique data or a unique node (address) in a distributed system. The unique constant value make it difficult to apply these techniques to map resources. For example, with computing resources there is a limited number of resources, and they often have a small set of values they assume. Using a DHT (Distributed Hash Table) based approach would save all the information in a few select nodes and make them the potential bottleneck of performance.

There have been some efforts that directly address the issue of locating computing resources in distributed systems. Condor [13, 20] uses a *Matchmaking* algorithm where nodes periodically update their state information with server-like *Matchmaker* nodes in a hierarchical fashion. While this solution is more scalable than a completely centralized approach it might not be scalable enough for global-scale distributed systems. The SHARP system [7] deals with the issue of trusted resource sharing in a decentralized fashion in context of the PlanetLab [19] system. This work deals with providing resource reservation in a trusted fashion rather than the mechanism of locating these resources.

In [10] the authors address the same issue as ours and evaluated the other heuristics we discuss in this paper (random forwarding, history-based forwarding, frequency-based forwarding etc.). [25] suggested giving virtual coordinates to the resources in a multi-dimensional co-ordinate space based on their attribute values, but did not provide a way of traversing this space.

5 Conclusion and Future Work

In this paper we presented a protocol, GUARD, that can be used to locate dynamic resources in a distributed system. GUARD is easy to implement (a single lookup table that is periodically updated by interacting with neighbors) and decentralized in nature. Our limited set of simulations show GUARD to be efficient, outperforming several other protocols significantly, and suitable for two different topologies.

The GUARD protocol is still in its initial stages. We are currently working on several aspects of the protocol. The principal focus of our current research is towards using GUARD for the problem of co-allocation [2, 21], where the resources can be scattered on multiple nodes. We are also working to refine the analytical model for GUARD to better understand its strength and limitations and where it will be most effective.

References

[1] BOINC: Berkeley Open Infrastructure for Network Computing. http://boinc.berkeley.edu.

[2] A.I.D. Bucur and D.H.J. Epema. Trace-Based Simulations of Processor Co-Allocation Policies in Multiclusters . In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), Seattle, Washington*, 2003.

[3] distributed.net. http://www.distributed.net.

[4] G. Fedak, C. Germain, V. N'eri, and F. Cappello. XtremWeb: A Generic Global Computing System. *In IEEE Int. Symp. on Cluster Computing and the Grid.*, 2001.

[5] Freenet. http://sourceforge.freenet.com.

[6] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Proceedings of the Tenth International Symposium on High Performance Distributed Computing (HPDC-10), IEEE Press*, August 2001.

[7] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. Sharp: an architecture for secure resource peering. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 133–148. ACM Press, 2003.

[8] Cecile Germain, Vincent Neri, Gille Fedak, and Franck Cappello. XtremWeb: Building an Experimental Platform for Global Computing. In *GRID*, pages 91–101, 2000.

[9] Gnutella. http://www.gnutella.com.

[10] Adriana Iamnitchi and Ian Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *International Workshop on Grid Computing*, Denver, Colorado, 2001. IEEE.

[11] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Mobile Computing and Networking*, pages 56–67, 2000.

[12] Project JXTA. http://www.jxta.org.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. *In Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS'88)*, 10(11–13):1063–1078, 1998.

[14] GIMPS: The Great Internet Mersenne Prime Search. http://www.mersenne.org/prime.htm.

[15] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, and Zhichen Xu. Peer-to-Peer Computing. *Technical Report HPL-2002-57, HP Lab*, 2002.

[16] Michael O. Neary, Sean P. Brydon, Paul Kmiec, Sami Rollins, and Peter Cappello. Javelin++: scalability issues in global computing. *Concurrency: Practice and Experience*, 12(8):727–753, 2000.

[17] Licnio Oliveira, Lus Lopes, and Fernando Silva. P^3: Parallel Peer to Peer - An Internet Parallel Programming Environment. *Lecture Notes in Computer Science*, 2002.

[18] Charles Perkins and Pravin Bhagwat. Highly Dynamic DestinationSequenced DistanceVector Routing (DSDV) for Mobile Computers. *Proceedings of ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.

[19] PlanetLab: An Open Platform for developing, deploying and accessing planetary-scale services. http://www.planet-lab.org/, 2001.

[20] Rajesh Raman, Miron Livny, and Marv Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing*, 2(2):129–138, 1999.

[21] Rajesh Raman, Miron Livny, and Marvin Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03), Seattle, Washington*, 2003.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany*, November 2001.

[23] Luis F. G. Sarmenta and Satoshi Hirano. Bayanihan: building and studying Web-based volunteer computing systems using Java. *Future Generation Computer Systems*, 15(5–6):675–686, 1999.

[24] SETI@home. http://setiathome.ssl.berkeley.edu, 2001.

[25] David Spence and Tim Harris. Xenosearch: Distributed resource discovery in the xenoserver open platform. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 216. IEEE Computer Society, 2003.

[26] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of SIGCOMM'01, San Diego, California*, August 2001.