Peer-to-Peer Netw Appl
DOI 10.1007/s12083-009-0030-1

NodeWiz: Fault-tolerant grid information service

Sujoy Basu · Lauro Beltrão Costa · Francisco Brasileiro · Sujata Banerjee · Puneet Sharma · Sung-Ju Lee

Received: 11 April 2008 / Accepted: 26 January 2009

Abstract Large scale grid computing systems may provide multitudinous services, from different providers, whose quality of service will vary. Moreover, services are deployed and undeployed in the grid with no central coordination. Thus, to find out the most suitable service to fulfill their needs, or to find the most suitable set of resources on which to deploy their services, grid users must resort to a Grid Information Service (GIS). This service allows users to submit rich queries that are normally composed of multiple attributes and range operations. The ability to efficiently execute complex searches in a scalable and reliable way is a key challenge for current GIS designs. Scalability issues are normally dealt with by using peer-to-peer technologies. However, the more reliable peer-to-peer approaches do not cater for rich queries in a natural way. On the other hand, approaches that can easily support these rich queries are less robust in the presence of failures. In this paper we present the design of NodeWiz, a GIS that allows multi-attribute range queries to be performed efficiently in a distributed manner, while maintaining load balance and resilience to failures.

Keywords Grid information service · Peer-to-peer · K-d-tree · Failure detection · Availability

S. Basu · S. Banerjee · P. Sharma · S.-J. Lee
Hewlett-Packard Laboratories, Palo Alto, CA 94304, USA

S. Basu
e-mail: sujoy.basu@hp.com

S. Banerjee
e-mail: sujata.banerjee@hp.com

P. Sharma
e-mail: puneet.sharma@hp.com

S.-J. Lee
e-mail: sungju.lee@hp.com

L. B. Costa · F. Brasileiro (✉)
Universidade Federal de Campina Grande,
58.109-970, Campina Grande, Paraíba, Brazil
e-mail: fubica@dsc.ufcg.edu.br

L. B. Costa
e-mail: lauro@dsc.ufcg.edu.br

1 Introduction

Efficient discovery of resources and services is a crucial problem in the deployment of computational grids, especially as these evolve to support diverse applications including interactive applications with real-time QoS requirements (e.g., multi-player networked games). Within such an environment multitudinous services made available by different providers co-exist. Once services are deployed and properly advertised, users can search for the available services and select the most suitable ones to cater for their needs. It is anticipated that clients will search for raw computing and storage resources (e.g., machine with Pentium 1.8 GHz CPU and at least 512 MB memory) as well as services (e.g., lightly loaded Everquest game server). Furthermore, the attributes may be dynamically changing (e.g., available bandwidth between two nodes) rather than static (e.g., OS version). Finally, services may appear and disappear in the grid, and the quality of service delivered by the more stable services may vary widely over time. Thus, providers should be constantly renewing their advertisement, while users should be constantly querying for the availability of better services. These trends make the resource or service discovery problem

Published online: 17 March 2009

challenging. The information service must be architected to support multi-attribute range queries in an efficient manner in this environment.

Grid Information Service (GIS) has been proposed to help users in the task of choosing which service to use to better fulfil their needs [7]. The GIS can be seen as a directory in which providers publish the static and dynamic attributes of their resources and services, and to which the consumers of these services submit their queries. Obviously, to be useful for large grids, a GIS implementation must be scalable. Moreover, in a system with potentially many thousands of components, failures are the norm and not the exception. Therefore, fault-tolerance of the GIS is another requirement.

Early implementations of the GIS were either centralized or distributed over a static hierarchy of information severs. Centralized solutions do not scale well in large systems or with dynamic attributes that change rapidly. Many centralized solutions can be augmented by replication, but then managing consistent replicas can incur significant overhead. Hierarchical distributed systems alleviate some of the issues with the centralized systems. However, most of these are inefficient in retrieving the answers to a multi-attribute range query; the dynamic nature of the attributes queried implies that the query has to be forwarded inefficiently to the hierarchy of information servers. Further, there is limited recourse available if due to the query load patterns, some information servers get heavily loaded while others are essentially unloaded.

More recent approaches rely on some scalable structured peer-to-peer (P2P) substrate on top of which the directory service is built [1, 4, 13, 19, 21, 27, 28]. Most of these systems rely on Distributed Hash Tables (DHTs) to implement structured P2P directories. DHTs are scalable and very robust to failures. On the other hand, the only search operation that is efficiently supported by DHTs is exact match. These systems do not provide a natural way to perform complex multi-attribute range queries while maintaining load balance.

Our goal is to design a GIS that allows multi-attribute range queries to be performed efficiently in a distributed manner. We emphasize this class of queries because these are among the more useful and common types of queries that a client of the GIS would need to execute to identify services or resources that meet its requirements. In this paper, we present **NodeWiz**, a GIS that is organized as a P2P system. The multi-attribute search space is distributed among the NodeWiz peers according to a distributed tree structure. NodeWiz is self-organizing such that loaded peers can dynamically offload some of their load onto other peers. Further, as described later, the information storage and organization is driven by query workloads, thereby providing a very natural way, not only to balance the query workload but also to optimize the performance for more common multi-attribute range queries. However, systems based on distributed tree structures are in general less resilient to failures than DHT-based ones. In this paper we analyze the impact of failures in such systems and propose mechanisms for dealing with these failures. They have been implemented in NodeWiz and evaluated in this paper.

The next section provides the background and related work. Section 3 describes the NodeWiz architecture in detail and presents the associated algorithms. Next, in Section 4, we analyze the impact of failures. Then in Section 5, we describe the fault-tolerance mechanisms that have been implemented in NodeWiz. Implementation issues are discussed in Section 6. This is followed by an evaluation of our NodeWiz implementation in Section 7. Finally, our conclusions are presented in Section 8.

2 Related work

A GIS is a key component of any large grid installation. It addresses the important problem of resource and service discovery which enables such large-scale, geographically-distributed, general-purpose resource sharing environments. Deployed grids based on first version of the Globus Toolkit [10] employed the Metacomputing Directory Service (MDS) [16]. The initial architecture was centralized. Subsequently, MDS-2 [7] was implemented with a decentralized architecture. The X.500 data model used by LDAP [25] is employed in MDS-2 to organize objects in a hierarchical namespace. Each entry has one or more object classes, and must have values assigned to the mandatory attributes for these classes. Values for optional attributes may also be present. The query language, also borrowed from LDAP, allows search based on attribute values, as well as on the position of objects in the hierarchical namespace. The MDS-2 system architecture consists of directory servers and other *information providers* maintained by the different organizations participating in a grid. They use soft-state registration to join *aggregate directory* servers, which in turn can query them to get details of their content. More recently, MDS-4 [23] has been released as part of Globus Toolkit version 4. The interfaces of MDS-4 have been standardized using web services. The information providers can be infrastructure monitoring tools like Ganglia [9] or any service that is provided by the

grid and needs to be monitored, such as a job queueing service.

NodeWiz can be viewed as a self-organizing, distributed directory that specializes in multi-attribute range queries. It treats attribute values advertised by resources and services, and the queries on them in a symmetric fashion. We view the query process as distributed matchmaking in which the advertisements and queries are routed through the NodeWiz *structured* P2P network until they reach the node where a match is found. Thus the matchmaking is done by NodeWiz peers in a wide-area distributed system, as opposed to Condor [24], which has used a centralized matchmaker.

There are several works that support multi-attribute and range queries on DHT-based overlays (e.g. [5, 19]). Fault tolerance is provided by the underlying P2P substrate. However, to allow operations on multiple attributes, several DHTs—one per attribute—need to be maintained. Maintaining multiple overlays involves either updating each of them whenever a new advert is made or sending queries to all of them. As the number of attributes increase, this implies a proportional increase in the update or query traffic. PHT [21] uses a single DHT but requires linearization of the attribute space. Thus, queries return a superset of the matching data, demanding a filtering procedure to be applied. DST [28] also uses a single DHT, but requires maintaining one segment tree for each attribute.

Mercury [4] is another information service that supports range queries over multi-attributes. Like [19] and [5], Mercury maintains a separate logical overlay for each attribute, however, it does not use DHTs as overlay substrate. Mercury peers maintain a range for a given attribute and pointers to peers that keep different ranges of the same attribute (including the previous and next ranges). Adverts are sent to every overlay (one for each attribute), while queries are sent to the most selective attribute overlay. When a peer leaves, the pointers are broken and, periodically, a peer replaces pointers to failed peers by new ones. In the meantime, queries may be unsuccessful.

There have been other proposals for supporting multi-attribute range queries in distributed environments without utilizing DHT. In [8], two spatial-database approaches are compared for supporting multi-dimensional range queries in P2P systems. The first approach uses space-filling curves to map multi-dimensional data to a single dimension. The latter is then partitioned by ranges among the available nodes. The second approach uses k-d-trees to partition the multi-dimensional space into hypercuboids, each of which is assigned to a node. In both cases, skip graphs are used to increase routing efficiency. SkipNet [12] enables range-queries on a single attribute by using the skip list data structure and ordering nodes in the overlay using string names, instead of hashed identifiers. Hence, explicit load balancing is required. Distributed Index for Multi-dimensional data (DIM) [17] is a data structure designed for multi-attribute range queries in sensor networks. It uses a geographic hash function to map the multi-dimensional space into a two-dimensional geographic space, and then uses a geographic routing algorithm. If instantiated as a k-d-tree, BrushWood [27] is very similar to NodeWiz. However, BrushWood has no fault tolerance mechanism for routing.

There are many distributed information services, built on a P2P substrate, that have not been designed for multi-attribute range queries. PIER [13] is a distributed query engine performing database queries over a DHT. INS/Twine [2] also describes a P2P information service. However, the focus is on semi-structured data (e.g., in XML syntax) containing only attribute and values that may be matched. Range queries are not supported. A DHT-based grid information service is presented in [1], supporting range queries on a single attribute; this work presents the study of various query request routing and update strategies.

Another P2P solution for discovering resources in grid environments was described in [14, 15]. Their approach differs from ours and others surveyed above in that they use an unstructured P2P system. They do not maintain a distributed index that can efficiently lead to the nodes that can answer the query. Instead, they use heuristics such as random walks, learning-based strategy (best neighbors that answered similar query) and best-neighbor rule (one that answered most queries, irrespective of type) to contact neighbors and propagate the search through the P2P network.

The way we divide the attribute space among the NodeWiz nodes has some resemblance to various data structures studied in computational geometry, if we consider the attribute space as a multi-dimensional space. k-d-trees recursively divide a multi-dimensional space, but at each level of the tree, one of the dimensions is used. Interval trees organize line intervals in tree data structures, so that the intervals intersecting a query range can be efficiently found. Multi-dimensional range trees are recursive binary search trees. First, a balanced binary search tree is built on the first attribute, and for each subtree, all the points contained in it are used to build a balanced binary search tree on the next attribute. Since we are building a P2P distributed system, a data structure that allows efficient search in a centralized environment is not enough. We need to have efficient ways of mapping the structure among

the nodes. k-d-trees provide the most obvious mapping. However, using a k-d-tree strictly would imply that all nodes at the same level would split on the same attribute, using the median value of the local data. In NodeWiz, nodes at the same level of the k-d-tree decide independently which attribute to split on, and the splitting value is not necessarily the median value. Hence the term k-d-tree is used in this paper without adhering strictly to its definition.

3 NodeWiz architecture

NodeWiz is a scalable P2P GIS whose main goal is to allow multi-attribute range queries to be performed efficiently and reliably in a distributed environment. In this section, we present the most fundamental aspects of the NodeWiz P2P architecture, namely the mechanisms for routing the queries and advertisements. This is closely related to the distribution of the attribute space among the peers, which changes as peers join or leave. NodeWiz adopts a soft-state approach for storing resource information for dynamic attributes. Resources update their information by periodically advertising their current attribute values. Therefore the workload of a peer consists of routing advertisements and queries, storing advertisements and responding to queries that overlap the attribute subspace assigned to it.

When NodeWiz is bootstrapped with one node, the situation is similar to the centralized matchmaker in Condor [22]. As the number of nodes grow, each new node that joins NodeWiz does that in a way that seeks to distribute the workload between itself and an existing node that was identified previously as having maximum workload. The algorithm used to identify the existing node with the highest workload is described in Section 6.1. The next step is to identify the attribute based on which the identified node will split its attribute subspace with the new node, and the splitting value of that attribute. This algorithm is described in Section 6.2. The motivation for choosing an attribute and dividing its range of values among the existing and new node comes from the observation that temporal and spatial distributions in attribute values can be exploited to maintain the load balanced among the peers and localize traffic.

The attribute space is divided among the peers according to a tree structure. This structure will be referred to as a k-d-tree, as explained in Section 2. Figure 1 shows an example where *A* bootstraps the system, then, *B* joins the system and splits the attribute subspace of peer *A* based on the attribute *Load* at the value 0.6. All advertisements and queries associated with load less than 0.6 are now routed to *A*, while those associated with load greater than or equal to 0.6 are routed to *B*. After that, *A* splits its attribute subspace first with *C*, using *Mem* as the splitting attribute, and later with *E*, again using *Load* as the splitting attribute. The other splits can be easily identified from Fig. 1. Since the node selected for splitting is chosen with the goal of distributing the workload evenly among all nodes, the tree will grow in a balanced fashion, provided the workload does not show a sudden change in characteristics. In practice, the workload pattern can change, and so subtrees can receive unbalanced workloads. When the traffic received by a node falls below a predetermined threshold, it can leave and rejoin by splitting with the currently overloaded nodes. We will explain such voluntary leaves in Section 5.

In order to efficiently route operations, each NodeWiz peer maintains a routing table that keeps track of some of the peers responsible for other parts of the attribute space. An advertisement or query may specify values for some or all attributes, and the values in a query can be specified as a range. If an attribute is not specified, it can match any value. Each entry of the routing table contains one attribute, a range for this attribute, and the identification of the peer that should be forwarded any advertisement or query that matches or overlaps the range for this attribute. The routing operation at a peer involves lookup of entries in this table starting at level 1, and progressing to the maximum level as follows. If the range or value specified

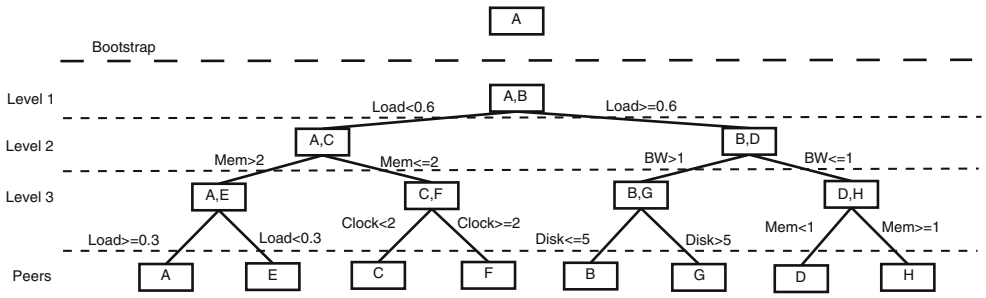

Fig. 1 NodeWiz's tree structure

Table 1 Peer A's routing table

Level	Attribute	Min	Max	Peer in charge
1	*Load*	0.6	$+\infty$	B
2	*Mem*	0	2	C
3	*Load*	0	0.3	E

in the query or advertisement has no intersection with the range in the routing table entry, the comparison is repeated with the entry at the next level. When there is an intersection, two situations are possible. If the range or value specified lies within the range in the routing table entry, the query or advertisement is forwarded to the peer identified in the entry. The routing operation is over in this case. When there is an overlap that spans beyond the range in the routing table entry, the query[1] is split into 2. One is forwarded to the peer identified in the entry, after its range is set to the overlapping range. The other query keeps the remaining range, and the routing operation progresses to the next level in the table with it. By excluding all attributes and corresponding ranges present in its routing table, a peer obtains the attribute subspace for which it is responsible. Thus, if the peer is still left with a query or advertisement at the end of the routing operation, there is a match in the attribute subspace. Therefore, the peer must either respond to the originator of the query with matching advertisements from its repository or add the advertisement received to the repository, superseding any older advertisement for the same resource.

As discussed before, when a new peer J joins the system, it must contact an existent peer E whose attribute subspace is going to be divided with J. At this point, J gets a copy of E's routing table. Then, both peers add an entry in their routing tables that points to each other and record the corresponding attribute and ranges defined by the splitting algorithm. Table 1 is the routing table for peer A in Fig. 1 (routing tables for the other peers can be easily derived from Fig. 1).

Now, consider a query "$Load < 0.2 \wedge Mem > 3$" that is sent to peer G (see Fig. 1). In level 1, peer G's routing table indicates that peer G should forward the operation to peer A. Then, after receiving the operation, peer A scans its routing table and finds out that it should forward the operation to peer E (according to the entry at level 3). According to peer E's routing table, peer E should not forward the operation to any other peer, since it is the only peer whose attribute subspace matches the operation values. Note, however, that the less selective the operation is, the more peers may receive the same operation. For instance, a query "$Load < 0.5$" will be routed to peers A, C, E, and F.

[1]This situation cannot occur for advertisements, since these specify a value instead of a range.

4 On the consequences of failures

In some environments, NodeWiz peers can be expected to fail. These peers may take a long time to recover if human intervention is required. Moreover, peers may voluntarily or involuntarily leave the system forever. In all these cases the routing tables of the remaining peers must be appropriately updated so that operations are successfully executed. In this section, we quantify the impact of failures in NodeWiz

4.1 System model and definitions

For the sake of simplicity, we will analyze the impact of failures by calculating the probability of having unsuccessful operations on a *well-formed* system subjected to a *well-balanced* workload. A well-formed system is characterized by a perfectly balanced k-d-tree. On the other hand, a well-balanced workload has the following characteristics: i) all operations match the attribute subspace of a single peer; ii) peers have the same probability of being the target of an operation; and, iii) peers have the same probability of being the recipient of the operation. Although a NodeWiz system does not need to be either well-formed or well-balanced, the way nodes are joined to the system and attribute subspaces are split naturally drive the system to these states. Thus, they constitute an interesting scenario for analysis.

More formally, consider that a GIS is a set of peers $\mathcal{G}$, $|\mathcal{G}| = \mathcal{N}$, that together store a global attribute space $\mathcal{S}$. A system is well-formed iff: i) the routing table of every peer has $\mathcal{L}$ entries; and, ii) $\mathcal{N} = 2^{\mathcal{L}}$.

Let $\mathcal{S}(op) \in \mathcal{S}$ be the subspace that matches operation op (a query or an advertisement); $\mathcal{T}(op) \in \mathcal{G}$ be the set of peers whose attribute subspaces intersect with $\mathcal{S}(op)$; and, $P(op, recipient)$ represent the probability that the operation op sent to peer *recipient* be issued. In a well-balanced workload: i) all operations op issued are such that $|\mathcal{T}(op)| = 1$; and, ii) $P(op, recipient) = cte$, for all operations op issued and all $recipient \in \mathcal{G}$, where cte is a constant.

As discussed in the previous section, peers use their routing tables to route operations on the k-d-tree formed by them. The routing tables allow any peer to route an operation to any other peer in the system. We name the *routing tree* of a peer P the tree that indicates the path that any operation received by P follows in the way to any other peer in the system. This tree is built in the following way. The root of P's routing tree is P

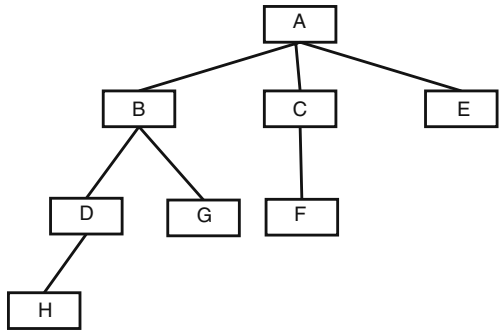

Fig. 2 Peer A's routing tree

itself. All peers that appear in P's routing table are then attached to P's routing tree at the next level. Then, for every new peer Q attached to P's routing tree, every peer that appears in Q's routing table and that is not yet present in P's routing tree is attached below Q in P's routing tree. This procedure continues until all peers in the system appear in the routing tree. Figure 2 shows the routing tree for peer A in the k-d-tree presented in Fig. 1.

In a well-formed system, the maximum number of hops in a route is $\mathcal{L} = log_2\mathcal{N}$, while the minimum number of hops is zero, corresponding to the route the peer has to itself. Let $R_i(P)$ be the number of i-hop routes peer P has, then $R_i(P)$ is given by Eq. 1:

$$R_i(P) = C_{\mathcal{L},i} = \frac{\mathcal{L}!}{(\mathcal{L}-i)!i!} \tag{1}$$

4.2 Evaluating the impact of failures

Let us assume that at any given time t, the system has a fraction f_t of peers that are faulty. Thus, in a well-formed system $\mathcal{N} \cdot f_t$ peers are faulty at time t. Since we are assuming that all operations have the same probability of being issued, this implies that all routes have the same probability of being used to execute an operation. For an operation op issued to a peer P to be successful, the route from P to $\mathcal{T}(op)$ must be composed exclusively of non-faulty peers. The probability of an i-hop route to contain at least one faulty peer is given by:[2]

$$PFR_i = 1 - (1 - f_t)^{i+1} \tag{2}$$

The probability of having an unsuccessful operation issued to peer P, named $PUO(P)$, is given by the sum on all possible sizes of routes of the probability of an i-hop route to have at least one faulty peer (PFR_i) times the probability of an operation to be routed though an i-hop route. This is expressed by Eq. 3:

$$PUO(P) = \sum_{i=0}^{\mathcal{L}} \frac{R_i(P)}{\mathcal{N}} \cdot \left[1 - (1 - f_t)^{i+1}\right] \tag{3}$$

Figure 3 plots Eq. 3 for $f_t = 10\%$, $f_t = 2.5\%$ and $f_t = 1\%$. As expected, as the fraction of faulty peers increases, the ratio of unsuccessful operations gets worse. Furthermore, as the size of the system increases, the amount of unsuccessful operations also increases. This is because as $\mathcal{N}$ increases, so does the size of the routes, which reduces the probability of having only correct peers in a route. Moreover, even low fractions of failures cause many unsuccessful operations. For example, when $f_t = 1\%$ in a system with 128 peers, a mean of 4% of the operations are not successful due to failures in routing or in peers that keep the target attribute subspace. For $f_t = 10\%$ the ratio of unsuccessful operations increases quickly, achieving values greater than 35% for systems with as little as 128 peers.

If peers are not able to autonomously recover from failures, then it is mandatory to take recovery actions so that the system will not collapse. In the next section we show how failures and voluntary leaves in NodeWiz are dealt with.

5 Dealing with voluntary and involuntary leaves

For the sake of clarity, before discussing the approach that we propose to deal with involuntary leaves, i.e. failures, we will first explain in details how voluntary leaves can be treated, assuming a failure-free scenario.

5.1 Voluntary leaves in a failure-free system

When a peer leaves the system, the attribute subspace that it stores must be reclaimed by another peer in the system. The simplest choice is to make this peer the last peer with whom the leaving peer has split its attribute subspace (or its replacement, if that peer has already left the system). Moreover, the system must ensure that every routing table that contains a reference to the leaving peer will be updated, by either removing this entry from the routing table or replacing the reference to the leaving peer by a reference to the peer that is reclaiming its subspace.

When the leave is voluntary, the leaving peer (say, L) contacts the peer which will be its replacement (say, R), informing that it wants to leave the system. R is the peer that appears in the last entry of L's routing

[2] The exponent $(i + 1)$ means the number of hops in the route added to 1, which represents the recipient peer itself.

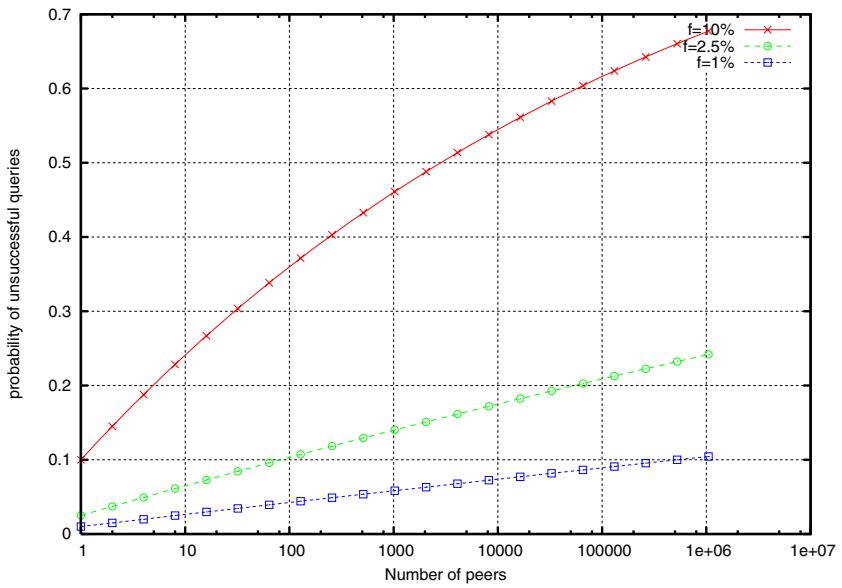

Fig. 3 Probability of unsuccessful operations in a system with $f_t \cdot \mathcal{N}$ faulty peers

table. At this point, L also sends to R its local state, i.e. all adverts it has locally stored. Then, R takes the necessary actions so that the appropriate routing tables in the system are updated, reflecting L's departure. During the execution of the leaving procedure, routing tables will become temporarily inconsistent, and if no precautions are taken, some operations may fail. To account for that, L only leaves the system after receiving an authorization from R. In the meantime, any operation that is routed through L is forwarded by L to R, so that R can process it appropriately. This approach guarantees that no operation is ever routed to a non-existent peer.

There are two different types of updates that may be necessary to maintain routing tables consistent. Regarding these updates, the peers in the system are divided in two classes. The first class is formed by R itself and the peers that have inherited from R an entry for L in their routing tables (if any), i.e. those peers that are in the branches of R's routing tree whose roots are any of the peers that appear in R's routing table below L. The second class is formed by all other peers. To update their routing tables, peers in the first class must remove the entry they have associated to L. On the other hand, all peers in the second class that have L in their routing tables should replace the reference to L by a reference to R. Identifying which routing tables should be changed when a peer leaves the system is not difficult. A simple recursion approach can be taken to implement the required updates.

After receiving the leave notification from L, R performs the following steps. It removes the entry it has associated with L in its routing table and asks all peers with which it has split the attribute subspace after its split with L (if any) to do the same. If there are such peers, they will appear in the entries of R's routing table below L's entry. Recursively, the request is sent down in the branches of R's routing tree whose roots are these peers. Acknowledgments are recursively sent in the reverse direction to inform the requester that the update has been performed in all peers in that branch of the routing tree. Thus, when R receives these acknowledgments it knows that the replacement has been completed by all peers that had inherited from R an entry for L in their routing tables.

R also begins the propagation of the departure of L to the other peers that may have L in their routing tables. To do so, it asks the peer that is one level above L in R's routing table (if any) to replace the reference to L by a reference to R in its routing table. This peer (say, U) starts a recursive update until all references to L are replaced by references to R. When this happens, U sends an acknowledgment to R.

The update initiated by U works as follows. U asks all peers that appear in its routing table at levels below the level U is in the system[3] (if any) to replace references to L by references to R. Also, if the peer that has asked U to do the update is not at a level above U in the system, then U asks the peer that appears in its routing table that is one level above the level it is in the system (if any) to replace references to L by references

[3]This will usually be the level at which U joined the system, but can be another if U has replaced a peer that has joined the system earlier than it did.

to R. U then waits for acknowledgments from the peers that it has asked to perform updates. This procedure is executed recursively until all references are updated. To avoid unnecessary message exchanges, peers should only keep propagating the information about L's departure if they have L in their routing tables.

When R receives acknowledgments from the peers below L (if any) and from the peer immediately above L (if any) in its routing table, it knows that there are no more references to L in the system and it can authorize L to leave the system.

Based on the example in Fig. 1, the departure of B is performed as follows:

1. B asks G to leave the system and sends G its local state; from this point on, B forwards to G any operation that is routed through B;
2. G removes the entry in its routing table that refers to B; since G's routing table has no entries below B, there are no peers that have inherited entries to B from G and that should be informed about B's departure; thus, G simply asks D to replace references to B by references to G and keeps waiting for an acknowledgment from D;
3. D replaces the reference to B by a reference to G in its routing table and asks H and A to do the same; D then keeps waiting from acknowledgments from both A and H;
4. H replaces the reference to B by a reference to G and sends an acknowledgment to D;
5. A performs the replacement and asks C and E to do the update;
6. E performs the update and sends an acknowledgment to A;
7. C performs the replacement, asks F to do the update, and waits for an acknowledgment from F;
8. F performs the replacement and sends an acknowledgment to C;
9. C receives the acknowledgment from F and sends an acknowledgment to A;
10. A receives acknowledgments from both C and E and sends an acknowledgment to D;
11. D receives acknowledgments from both H and A and sends an acknowledgment to G;
12. G receives an acknowledgment from D and informs B that it can leave, completing the leaving procedure. Figure 4 shows the resulting k-d-tree.

Note that when the leaving peer is a "hub" (for example, B was the second peer to join the system), the leave operation is very costly, since every peer in the system, except B itself, has an entry that references B. However, for peers that are fresher in the system, the overhead of leaving is much lower. In particular, in a well-formed system, half of the peers are in the $\mathcal{L}^{th}$ level and will require a single update operation to leave the system. In general, the number of updates in routing tables when a peer that is at level i in the system leaves a well-formed system with $\mathcal{N}$ peers is $\frac{\mathcal{N}}{2^{i-1}} - 1$, while the number of messages exchanges is twice as much. In average, the number of updates required is less than $log_2\mathcal{N} + 1$ and the number of message exchanges is less than $2 \cdot (log_2\mathcal{N} + 1)$.

5.2 Dealing with involuntary leaves

It is important to point out that we are not concerned with preserving the state of faulty peers. That is to say, adverts that were stored in faulty peers will be lost. The fault tolerance mechanism proposed aims only to recover the consistency of the routing tables, therefore, preventing operations from failing. Given that adverts are constantly renewed, if the routing tables are kept consistent, then the loss of state is not permanent. On the other hand, if an effort is not made to keep the routing tables consistent, then a substantial number of operations will fail, as we have shown in Section 4.

For the sake of clarity, we first describe the rationale of a fault tolerance mechanism for the simpler case in which only one peer may fail. Then, we present how this mechanism can be enhanced to cope with the general case in which multiple peers may fail.

5.2.1 The single failure case

Failure detection is at the heart of any fault tolerance mechanism, thus, we first discuss how failure detection can be added to the NodeWiz design presented so far.

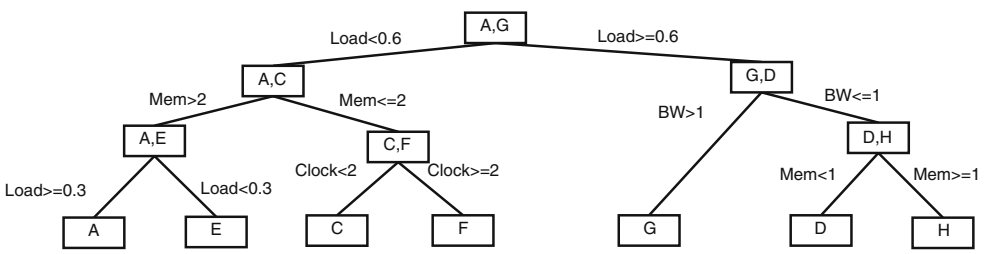

Fig. 4 NodeWiz k-d-tree after B's departure

Failure detection Failure detection is performed via a simple pull mechanism. If peer P is responsible for monitoring peer Q, then P periodically issues "Are you alive?" requests to Q. Q must reply to them with "I am alive!" responses. If a timeout expires, then P suspects that Q has failed and signals a failure detection. Initially, we consider that failure detection is perfect, i.e. if P signals that Q has failed, then Q has indeed failed. In Section 5.3 we discuss the effects of imperfect failure detection.

For now, let the monitor of any peer in the system be the peer that appears in the last entry of its routing table. We name $M(P)$ the peer that monitors a peer P. Note that if P wanted to voluntarily leave the system, it would ask $M(P)$ to execute the appropriate procedure.

Although it is easy for a peer to identify who is its monitor, it is not straightforward for a peer to identify which peers (if any) it should monitor. For that, each peer must maintain extra information. When a new peer P joins the system, then it selects a peer Q whose attribute subspace will be divided with P. At this point, Q should start monitoring P, and P should start monitoring Q. Moreover, the peer that was previously monitoring Q—if there is such a peer it will be the one with whom Q had split its attribute subspace before splitting with P—should be informed that it is P that is performing this task from that point on. Similarly, when a peer P leaves the system, $M(P)$ should stop monitoring P, and another monitor must be assigned to all peers S, such that $M(S) = P$ (if any). The first requirement is trivially met, since $M(P)$ is the peer that P contacts to leave the system. To perform the second update, P must inform all peers S that it is monitoring that they should be monitored by another peer. Each peer S then contacts the peer above P in S's routing table and ask it to start monitoring S. (Note that these actions can be easily embedded in the leave procedure described earlier, obviating the need for any extra exchange of messages.)

Involuntary leaves for the single failure case When there is only one faulty peer and no other peer fails while recovery is taking place, fault tolerance is achieved in a very simple way. When a peer P fails, $M(P)$ detects P's failure and creates a virtual peer that assumes P's identity. We name the virtual peer that impersonates P its *shadow*, and refer to it as $S(P)$. At this point the node that executes $M(P)$ is also executing the virtual peer $S(P)$. When $S(P)$ starts to execute it asks $M(P)$ to leave the system and waits for the corresponding authorization. This mimics a voluntary leave operation issued by P. Upon receiving such request, $M(P)$ proceeds as if P had asked to leave the system, with the only difference that P's local state is empty—P's state was lost when P failed. When $S(P)$ receives the authorization to leave the system, it is guaranteed that all references to P in the routing tables of the other peers have been either removed or replaced by references to $M(P)$. $S(P)$ then terminates its execution and the system is consistent again.

Recall that the entries in the routing table store not only the identity of a peer, but also the information required to access this peer (e.g. IP address and port number). Thus, unlike the case for voluntary leaves, when the leave is involuntary some operations issued while the system is not consistent will fail. Peers that still have a reference to P do not know how to contact $S(P)$, thus, operations that are routed through the faulty peer will fail. However, this can be circumvented with a simple retry mechanism and the support of a name service.

In the more dynamic setting we target, one could have the access information for a peer stored on a name service, and when necessary, the information could be looked up, using the peer's identity as the key. The information would be cached by the peers, and when contacting a peer resulted in a failure, the information could be refreshed from the name service. Assuming the existence of this name service, when $S(P)$ starts executing, it replaces the access information for P in the name service by the information necessary to access $S(P)$. When a peer Q tries to access the faulty P using stale information it may have in cache, the operation will fail. Q refreshes the information from the name service and will then contact $S(P)$ as if it were P. Like in the voluntary leave case, $S(P)$ forwards to $M(P)$ any operation that is routed through $S(P)$.

Impact of a failure during the execution of a voluntary leave Another concern is the impact that a failure may have on the execution of a voluntary leave. The state of a peer comprises not only the adverts it stores, but also information related to the execution of on-going leave operations. As described in the previous subsection, a peer that is engaged in the execution of the leaving procedure of another peer may block waiting for an acknowledgment sent by another peer. If the latter fails before sending this acknowledgment the former will block forever and the system will enter a deadlock.

Fortunately, this problem can be easily fixed. Whenever the routing table of a peer engaged in the execution of a leaving procedure needs to be updated, the peer checks if there are new peers that need to be asked to update their routing tables; these peers are asked to do so, and acknowledgments from them are awaited. Also, any peer engaged in the execution of

the leaving procedure updates its routing table after it receives acknowledgments for the updates requested by it. Moreover, any pending acknowledgment from a peer that is no longer in the routing table is marked as received.

Dealing with multiple failures The simple approach presented so far was possible because we did not consider that other peers could fail while the failure of a peer was being dealt with. The possibility of simultaneous failures complicates matters. This is because multiple failures may lead to undetected failures, in the case that a set of peers monitor each other and all of them fail before being able to act on the failure of the peer they monitor. For instance, in a well-formed system, pairs of peers monitor each other (e.g. A and E, C and F, B and G, and D and H in Fig. 1); if both peers in any of these pairs fail simultaneously, their failures will not be detected, and the routing tables will remain inconsistent.

To solve this problem, one must appropriately increase the number of monitors that a peer has. We do that by requiring that every peer in the system monitors particular sets of peers. Let n be the number of levels in which a peer appears in the k-d-tree (for instance, in the example of Fig. 1, A and B appear in three levels, while E, F, G and H appear only in one). A peer P will monitor n sets of peers, each of them associated to a corresponding *set leader* that the peer may replace. The set leaders are the peers that appear in P's routing table at or below the entry associated to the level at which P is in the system (in the example of Fig. 1, A and B are at level 1, while E, F, G and H are at level 3). Each of these sets contain all peers that belong to the branch of P's routing tree whose root is the corresponding set leader. P will replace a set leader only in the event that all peers in the corresponding set (including the set leader) have failed. Notice that if failures were sequential, i.e. if during the recovery of a faulty peer the system remained fault-free, then, P would only replace a set leader L if L had failed and all other peers in L's set (if any) had also failed.

5.3 On the impact of wrong suspicions

Wrong suspicions may lead to correct peers being removed from the system. When this occurs, two problems arise. Firstly, the adverts stored by the removed peer, say R, will not be considered when operations are issued to recipients other than R. Secondly, until R realizes that it has been removed, it will perform business as usual. This, in turn, may lead to either partial or total failures in operations. If R does not belong to the target of the operations it receives, then a failure will only happen if the information in R's routing table gets out-of-date (for instance, because of failures and departures of peers that appear in its routing table). On the other hand, if R belongs to the target of the queries it receives, then adverts that have been published after R's departure may not be returned to the user; also, if R belongs to the target of the adverts it receives, then these adverts will not be seen by the other peers in the system.

However, peers that are wrongly suspected may eventually detect that have been expelled from the system—for instance, by inquiring its monitor when it has not received "Are you alive?" messages for a long enough period of time—and may re-join the system. Therefore, it is important that the failure detector delivers good quality of service to avoid frequent wrong suspicions.

6 Implementation issues

In this section we discuss some issues that should be carefully addressed to render the implementation of NodeWiz scalable and efficient in dynamically balancing the load of the operations among all peers.

6.1 Load balancing

When a new node joins NodeWiz, it needs to identify one of the most overloaded nodes, which will split its attribute space with the joining node. We employ a distributed algorithm, henceforth referred to as the **Top-K Algorithm**, which orders the nodes in NodeWiz according to their workloads and identifies the most overloaded one. Depending on how frequently nodes join, this algorithm could run periodically or on demand. If another join request had already reached the identified node, resulting in it undergoing a split since the top-K workload information was disseminated, the request is forwarded to the node with the next highest workload identified during the run of the Top-K Algorithm.

The Top-K Algorithm is distributed and runs in two phases. Each node maintains a counter which represents its workload. The counter is incremented for each advertisement or query received by the node. Periodically, it is divided by 2 to give more weight to recent workload. In the first phase, each node sends a message to another node selected from its routing table according to a criterion to be explained soon. The recipient is selected such that these messages travel along the links of a tree composed of all the nodes in NodeWiz. Each

non-leaf node waits during a timeout period for its children to send their messages to it. After receiving their messages, the node includes its own workload, sorts and retains the Top-K workloads along with identities of the corresponding nodes. It sends out the retained Top-K workloads to its selected recipient. After the root of the tree builds the list of Top-K workloads among all nodes, the list is disseminated in the second phase to all nodes in NodeWiz. This is simply achieved by relaying the list to all nodes from which a node receives a message in the first phase. Thus the list travels back along the links of the tree to all the children. This algorithm may run once for the on-demand case, or it may run periodically, as mentioned earlier. In the second case, there is an epoch counter tagged to each message, so that messages delayed from one epoch, do not get processed by a node in the next epoch.

We have mentioned that each node sends a message in the first phase to one of the nodes selected from its routing table. The selection of the recipient node is based on the routing table. The recipient selection process retraces the order by which nodes join NodeWiz. Recall that each join results in the splitting of the range of one attribute remaining in the possession of the splitting node. To retrace the order of these joins, each node looks at the most recent join event it participated in, either as the splitting node or as the joining node. This will be the most recent (highest level) entry in its routing table. Recall that each entry in the routing table indicates a range of values for a single attribute, and a corresponding node to which advertisements or queries overlapping that range should be sent. By excluding all ranges present in the routing table for this attribute, the node obtains the range of values of this attribute for which it is responsible. If the values in its own range are greater than the values in the range of the routing table entry, the node will wait for the recipient node in that routing table entry to send a message to it. Otherwise, the node will send its own message to the recipient node. In case the node waits for the recipient's message, it checks the next most recent entry in its routing table. This might be for the same or different attribute. In any case, a comparison is again done for the values in the range of the corresponding attribute owned by this node and the recipient of this entry. If the node has to wait for the recipient's message, it adds this recipient to the list of nodes for whose message it is waiting. This list grows until the node reaches a routing table entry, while scanning back from the most recent entry, for which the comparison indicates that it should send the message. The node does not scan the routing table beyond this point. After it waits for the messages from all the nodes in its list of nodes to wait on, it includes its own workload, retains the Top-K values, and sends the resulting message to the recipient of the entry where it stopped scanning the routing table. Thus each node in NodeWiz will wait for zero or more nodes to send their message to it, and will send out exactly one message. The exception is the one node that will scan its entire routing table and add all nodes to its list of nodes to wait on. This is the node whose attribute subspace includes the maximum value of each attribute. This node is the root of the dissemination tree, and will start the dissemination of the list of Top-K values in the second phase.

6.2 Splitting the attribute space

The Splitting Algorithm has to identify an attribute, for which the range of values owned by the splitting node can be divided into two ranges of values. Two conditions have to be satisfied. Firstly, the values of the selected attribute in the advertisements and queries seen by the splitting node should show high probability of falling in clusters that are within the two ranges selected. This is based on the underlying assumption that an attribute which shows strong clusters will continue to do so, and has the better chance of maintaining even distribution of load between the splitting and joining nodes. For example, there might be a cluster of workstations which are kept busy by jobs submitted through a batch queuing system. There might be another cluster of desktop machines that are idle most of the time. If the splitting node finds the load averages of both sets of machines in the advertisements received by it, a clustering algorithm could easily select the load average attribute and a splitting value so that the advertisements from the two sets of machines are assigned to the two nodes. This brings us to the second condition that needs to be satisfied. Consider the case where the clustering algorithm finds two clusters for an attribute. However one cluster is very small is size compared to the other. This can clearly lead to load imbalance between the splitting and joining node. Hence we select among all the attributes the one for which our clustering algorithm leads to most even-sized clusters.

The input to the splitting algorithm in each node is the histogram of values, one for each attribute, accumulated from advertisements and queries received by that node since the last time the algorithm was run. First, the splitting algorithm invokes the k-means clustering algorithm [11], with $k = 2$, individually on each of the attribute histograms, with a limit on the number of iterations. The boundary between the last histogram bucket assigned to the first centroid and the first bucket assigned to the second centroid is the splitting value for

that attribute. Then we select the attribute for which this division is the most balanced in size, so that the workload gets distributed evenly between the existing and the new peer. Although we depict only binary splits in this paper, the scheme can be generalized to splitting the attribute space into more than two partitions at a given time.

6.3 Routing diversity optimization

Although NodeWiz allows for the attribute subspace to be distributed equally among the peers, the routing load is not balanced. The hubs, i.e. the peers that appear at the first levels of the tree, such as A and B in Fig. 1, are found in the routing table entries of several peers. As a result, they forward more messages than the peers that only appear at the levels closer to the bottom of the tree.

We have tried a simple solution for this problem. When a query or advertisement reaches its destination, the query results or an acknowledgment for the advertisement is sent back to the NodeWiz node that initiated the query or advertisement. When the routing diversity optimization is turned on, the destination peer piggybacks its routing table in the message that it sends to the initiator. The latter caches the routing table received in correlation with the routing table entry that was used to send the query or advertisement out. This ensures that another query or advertisement destined for the same sub-tree of the decision tree can be sent there with fewer overlay hops.

This routing diversity optimization has been evaluated, and the results are presented in Section 7. We also observe that the caching of multiple entries has the added benefit of increasing the fault resilience of the system.

6.4 Scalable and fault-tolerant name server

To avoid any operation from failing while an update is on-going, the fault tolerance mechanism requires a name service. Obviously, the name service also needs to be scalable and fault-tolerant, so that it will not constitute a bottleneck or a single point of failure in the system.

A simple way to implement a scalable and reliable distributed name service is to use a DHT, with the identities of peers as the search key. This gives scalability. Fault-tolerance is attained by replicating the information about each peer in the DHT. Note that although DHTs are not suitable for range queries over multiple attributes, they are very efficient for implementing the operations issued to this name service. Moreover, the DHT can be easily implemented by the same peers that implement NodeWiz.

6.5 Scalable monitoring

Although voluntary and involuntary leave procedures require a number of updates, with message exchanges and local storage capacity $O(log_2\mathcal{N})$ on average, these are not evenly distributed among the peers. As for the routing load previously discussed, the hubs have normally a higher overhead than the peers that are in lower levels of the tree.

This is not a big issue for algorithms that are executed sporadically, such as voluntary and involuntary leaves. On the other hand, failure detection is continuously executed. In particular, hubs should monitor $\mathcal{N} - 1$ peers. Nevertheless, there are several practical solutions to this problem. The simplest one would be for a peer to monitor only the set leader instead of monitoring all peers in a given set. When the failure of the set leader is detected, then the peer starts monitoring one of the peers in the next level of the branch of the peer's routing tree whose root is the set leader. This procedure would be carried on recursively, until all peers in the set failed, when the peer would take the necessary actions to replace the set leader. In this way, any peer would be monitoring only one other peer per set at a time. We note that even without this optimization, half of the peers in the system already monitor just one other peer.

The failure detection poses a second scalability challenge. As the algorithm was described, hubs must store the whole routing tree, which may not be feasible in some settings. One possibility to circumvent this problem, if it exists, is to construct the required parts of the routing tree on demand. The problem is that the information stored by faulty peers would be required to allow that. An alternative would be for peers to use the DHT that implements the name service to reliably store their routing tables. Then, parts of the routing tree could be reliably built on demand. Again, the operations that are required to build the routing tree are exactly the type of operation (exact match on the identity of a peer) for which a DHT is most suitable.

7 Evaluation

Our evaluation is based on both simulations and experiments with a NodeWiz implementation. In Subsection 7.1 we use simulations to evaluate the efficiency of the top-K and splitting algorithms in

maintaining balanced both the tree data structure, as well as the load among the peers. In Subsection 7.2 we use experiments with an implementation of NodeWiz to evaluate the response time of the query and advertisement operations, as well as how fast the system recovers from failures.

7.1 Evaluation of the top-K and splitting algorithms

We have built an event-driven simulation framework for NodeWiz. Our experiments use both synthetic and real datasets. For both of them, we have six attributes. In the synthetic dataset, each attribute is generated from a Pareto distribution which has been observed by other researchers to have good correlation to the attributes in a data-center trace [1]. For the real dataset, we used the measurements reported by the ganglia distributed monitoring system for PlanetLab nodes [9]. We selected six attributes from the dataset, namely the system load averages measured at 1, 5 and 15 minute intervals, and the amount of available disk, memory and swap space. Our discrete-event simulator reads one query and one advertisement at each clock cycle until all events in the input files have been consumed. The NodeWiz node which a client would contact with this query or advertisement is chosen randomly. Each simulation must specify the number of NodeWiz nodes. When all of them have joined NodeWiz, we reset the statistics and report only the values obtained at the at the end of the simulation. The number of events simulated in the synthetic dataset is 100 times the number of nodes, and usually a third of the events are simulated by the time all nodes have joined. However, due to the small size of the PlanetLab archive available, this is not always true in the PlanetLab dataset.

Figure 5 shows the variation in average number of hops for a query or advertisement as the network size increases exponentially from 10 to 10,000 nodes. We observe that the average number of hops increases very slowly. The queries in this experiment are for specific values of each attribute. The plots for queries and advertisements look similar. This is to be expected, since NodeWiz will treat a query and an advertisement with the same attribute values identically as long as we are not querying for a range. Both will be routed to the node with ownership of the attribute subspace in which these attribute values fall. If we were querying for a range, each query would visit all nodes overlapping the query range, and so the average number of hops will increase. This is explored in our technical report [3].

For this particular experiment, we have also run simulations with a DHT-based GIS for a system with

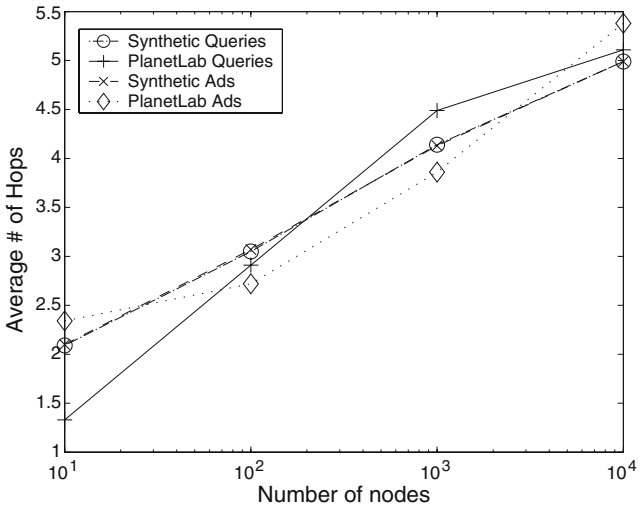

Fig. 5 Average number of hops for a query or advertisement as the network size increases

10,000 nodes. We have used PeerSim,[4] an event-based P2P simulator, and measured the number of hops required to store adverts in the GIS and to query them. As previously discussed, for a DHT-based GIS one needs to maintain as many DHTs as attributes. The accumulated average number of hops for updating all 6 DHTs was 28.83, which means that the number of hops increases linearly with the number of attributes, when compared to the number of hops required for NodeWiz. For the queries, the average number of hops will depend on the DHT that is used. The strategy to select the most appropriate attribute is not trivial. In our simulations, we have used all possibilities for all queries processed, so that we can bind the best and worst values that any strategy could achieve for the data we have simulated. Our results show that in the best case, the average number of hops is 3.24. On the other hand, it can be as high as 4.75. These values are not significantly different from those attained by NodeWiz. In summary, to avoid a much higher latency cost when storing adverts, updates need to be executed in parallel in all DHTs. The extra traffic generated for updating the DHTs, as well as the overhead required to manage these parallel updates, may turn out to be too high as the number of attributes (and DHTs) increases.

Figure 6 shows the increase in number of entries in the routing table, both maximum and averaged over all nodes, as the network size increases. From this figure, we conclude that the routing table size increases very slowly compared to the rate at which the network size increases. Moreover, the maximum number of entries

[4] http://peersim.sourceforge.net/.

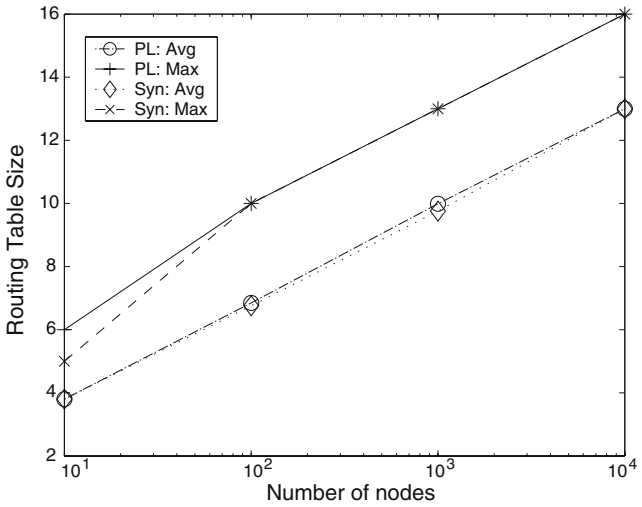

Fig. 6 Average and maximum number of entries in the routing table as the network size increases

is always very close to the average number of entries in the tables. Recall that the number of entries in the routing table of a node equals the number of times the attribute space has been split to obtain the node's attribute subspace. Since our joining algorithm limits the imbalance in the number of entries of different nodes, it is expected that this growth will be logarithmic in the number of nodes.

Figure 7 shows the standard deviation of the workload, as a measure of load imbalance, versus number of nodes. The workload is the number of advertisements and queries received by a node until the end of the simulation, from the steady state when all nodes have joined. Recall that we reset statistics at that point. For each dataset, we show 2 plots, one marked 'Clustering' which uses the clustering algorithm described in Section 6.2 to identify the attribute and the splitting value. To measure how well this is doing, we compare against the plot marked 'K-d tree'. Here the idea is to divide the attribute space as a k-d tree. So, at level i in the tree, attribute i is used, with a wraparound when maximum number of attributes is reached. Also, the splitting value is the median of all data points for that attribute that the node received in advertisements. Notice that our clustering technique is doing better than a k-d tree. Also, we do better usually on PlanetLab dataset compared to the synthetic dataset. This could be attributed to the fact that the synthetic data will not have clusters as much as the real PlanetLab data. We must also note that we are not comparing to other P2P schemes that use k-d trees. In particular, the NodeWiz techniques of maintaining routing tables, and the top-K workload vector remain invariant.

Figure 8 shows the variation in the number of attributes used by NodeWiz for the PlanetLab dataset. When we reduce the number of attributes from 6, the baseline in our experiments, to 3 and then to 1, the average number of hops taken by a query increases. This is to be expected, since each query specifies a range for each attribute. As the number of available attributes decreases, the range owned by a single node decreases for fixed number of nodes. As a result, the query gets flooded to more nodes. The advertisements specify a single value, rather than a range. Hence they are insensitive to the number of attributes. Furthermore, we did not find any sensitivity of the routing table size to the number of attributes. Hence, that data has not

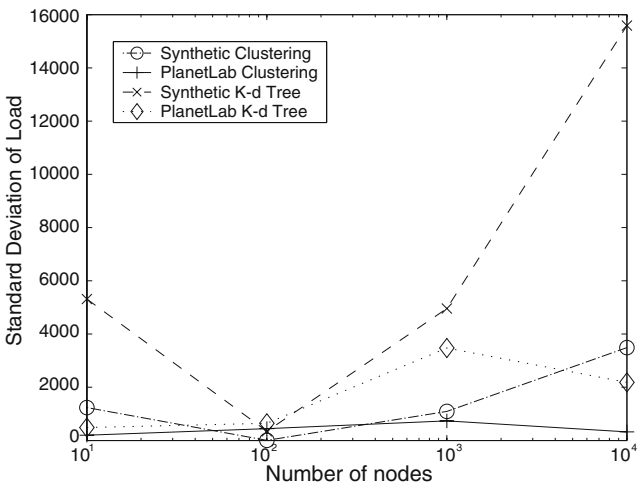

Fig. 7 Standard deviation of workload as a measure of load imbalance versus number of nodes

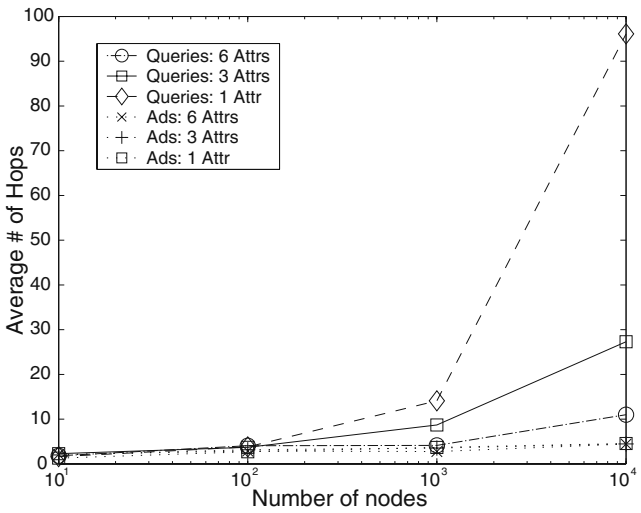

Fig. 8 Query and ad hops versus number of nodes as the number of attributes is varied

been plotted. From Fig. 8, we may also conclude that NodeWiz has an advantage over systems that support range queries using a single attribute. Unless the query has an extremely selective attribute, and a distributed index, such as a DHT, is available for that attribute, these systems will result in the query being flooded to a large number of nodes. On the other hand, our system can result in the query being flooded to a large number of nodes, only if a very large range or wildcard (any value acceptable) is specified for an attribute. This problem exists in DHT-based systems also. We can address this problem by limiting the query to some reasonable range on any attribute where a wildcard or very large range is specified.

7.2 Evaluation of response time and recovery time

We have developed a Java implementation of NodeWiz. In our implementation peers use a communication layer provided by the JIC (Java Internet Communication) [18] library. JIC is an asynchronous, Internet and NAT friendly communication infrastructure built on top of an XMPP (eXtensible Messaging and Presence Protocol) stack [26]. The NodeWiz prototype is currently being used as the GIS of the OurGrid middleware [6, 20]. It incorporates all the optimizations discussed in Section 6, except the implementation of the DHT-based name server.

We have run three different types of experiments. In the first one we evaluate the performance of the routing diversity optimization described in Section 6.3. The second type of experiment evaluates the response time of queries submitted to the system. Finally, we conducted experiments to measure the recovery time of the system after a failure.

In all experiments the system was set up in the following way. The system was started with a single peer and before a new peer could join the system, some adverts were added. After the system reached the desired size, the actual experiment was started and its associated metrics begun to be collected. Each system mounted received 10 times more adverts[5] than the number of peers. Peers joined the system every 400 ms and a new advert was issued every 40 ms. We set the Top-K algorithm to run periodically every 1, 500 ms. To speed up the execution of the experiments we checkpointed the initial states of the systems with different sizes, i.e. the state when the system reached the desired size. This way, the collection of the metrics for each run and each particular system size started with the system in the same initial state. We set the 'time-to-live' associated to the adverts to infinity, so that it was guaranteed that they would remain in the system during the whole experiment. Furthermore, after the system reached its final size, no other advert operations were issued.

[5]These adverts were those describing PlanetLab nodes, cited in Section 7.1.

Table 2 Average number of hops and effect of routing diversity optimization in the experiments with the prototype

Number of peers	Average number of hops	
	Without optimization	With optimization
10	2.23	0.97
100	2.76	1.21

We first conducted experiments for systems with size up to 100 peers. In all scenarios, the peers were evenly distributed in a 14-processor cluster. The processors in the cluster were Pentium 4 Xeon 2.4 GHz, running Debian Linux and connected through a 100 Mbps network. All peers that ran in a given machine executed in a separate process. Moreover, each machine ran an XMPP server. When a peer sent a message to another peer, the message passed though at least one XMPP server, even when both peers ran at the same machine.

The following results were obtained from several runs of each scenario, which were enough to yield a confidence interval of 95% with a maximum error of 5%.

In the first set of experiments we submitted a number of query operations to the system. These operations are crafted in such a way that there is always a single peer that holds the target attribute subset. Table 2 shows the effect of the routing diversity optimization mechanism.

As expected, the average number of hops decreases significantly when the routing diversity mechanism is in use. The reduction is by a factor of approximately 56% in both cases. By using the routing table of the last recipient recorded in the routing table entry, we increase the probability that the message will be sent closer to its destination on the first overlay hop, as long as there is some locality in the traffic.

Next we evaluated the mean response time of queries. The mean response time for systems with 10 and 100 peers are presented in Table 3, while Fig. 9 shows the response time of the query operations for

Table 3 Effect of routing diversity optimization on the response time for systems with different sizes

Number of peers	Mean response time (ms)	Mean response time with optimization (ms)
10	249.95	287.07
100	535.67	326.67

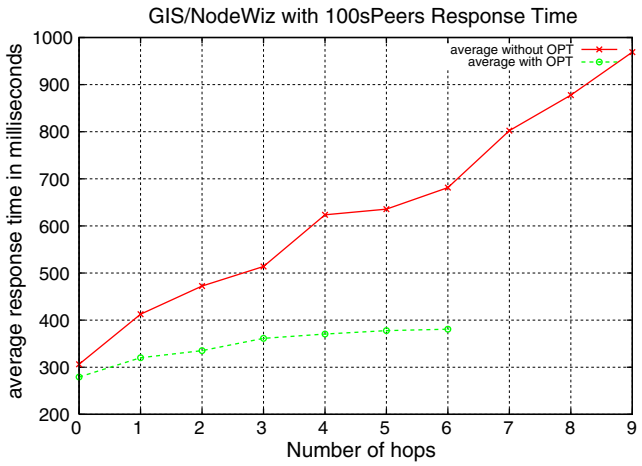

Fig. 9 Effect of routing diversity optimization on the response time for 100-peer systems

the same runs of the experiment previously discussed, considering the systems with 100 peers.

As expected, results for the optimization mechanism present a smaller mean response time and a smaller maximum number of hops. Moreover, the mean response time for queries with the same number of hops also varies. Results for systems with 100 peers show how routing load impacts the system. Routing diversity prevents operations from passing via peers in higher levels of the routing tree (see Section 6). It alleviates the number of operations needed to be routed in the whole system and, thus, the reduced load of the machines improves responsiveness. As a consequence, queries that had to pass though a path with the same length show better results in the system with optimization.

The next experiment focused on the performance of NodeWiz's fault tolerance mechanism. We measured the time required to fully recover the system after a crash. After setting up the system we randomly selected a peer and forced its crash. We then measured the time elapsed between the failure detection and the instant when all impacted tables had been updated. Recalling NodeWiz's failure detection procedure, a peer periodically sends heartbeats to its monitored peers asking if they are alive. If the peer does not receive a response after the amount of time set as the detection timeout, it considers that the monitored peer is faulty. Our prototype relies on the JIC built-in failure detection mechanism, and we have set the heartbeat interval to 10 s and detection timeout to 50 s. For a system with $\mathcal{N}$ peers, we repeated such procedure enough times to get results with the required confidence level. Table 4 shows the mean and median recovery time for systems with 10 and 100 peers.

Table 4 Recovery time for systems of different sizes subjected to a single failure

Number of peers	Mean recovery time (ms)	Median of the recovery time (ms)
10	87.10	18.70
100	4, 150.46	22.50

We measured both the mean and the median because the distribution of recovery times exhibits a long tail. Such behavior occurs because the departures of peers at higher levels yields more routing table updates and, therefore, the system takes longer to recover in these cases. Nevertheless, in most cases the time required to fully recover the system from a single failure is very small.

We have also conducted the experiments running in an emulated system with 1, 000 peers. We used the Emulab testbed to execute these experiments. In this case, the peers were evenly distributed over 50-nodes. Each node used had a 3.0 GHz 64-bit Xeon processor, with 2 GB RAM, running Red Hat Linux (Emulab pc3000 type). The experiments were configured so that the nodes were connected by a 1 Gbps network. Again, each node ran an XMPP server and each peer ran in a separate process. Table 5 shows the results for these experiments.

Table 5 Results for the experiments for 1, 000-peer emulated systems

Metric	Value measured
Average number of hops without optimization	5.93
Average number of hops with optimization	4.15
Average response time without optimization	170.00 ms
Average response time with optimization	149.57 ms

For a larger system, the routing diversity optimization still yields improvements on both the average number of hops and in the average response time. However, the improvement is less important in this case, when compared to that achieved for systems with smaller sizes. This is because for a larger tree, there is a higher probability of forwarding an operation to a node that will have to route the operation through a larger subbranch of the tree, increasing the number of hops. Still, the routing diversity optimization is very important to reduce the number of operations that are routed through the hubs, no matter the size of the system.

8 Conclusion

In this paper, we presented NodeWiz, a distributed and self-organizing information system for grid infrastructures. Our focus is to enable efficient execution of

multi-attribute range queries, which are expected to be an important and common class of queries. NodeWiz allows for information service nodes to be dynamically added and removed from the information system, addressing scalability and performance concerns. More specifically, the algorithms described as part of the NodeWiz system have the capability to balance the load across multiple information service nodes while optimizing the performance for popular multi-attribute range queries in a distributed manner. The prior work on this problem does not provide a natural way to deal with these kind of queries.

In NodeWiz, advertisements from service providers are placed strategically into the information system such that queries from the service consumers are routed efficiently (with minimum number of hops) to the nodes where the matching advertisements reside. We evaluated our algorithms using simulations on synthetic and real data extracted from PlanetLab. We presented results on the average number of hops for a query or advertisement as the network size (number of nodes), number of attributes and query selectivity are varied. We also evaluated load imbalance and a routing optimization. The results obtained indicate that NodeWiz has an advantage over systems that consider single attributes in isolation.

We have evaluated response time and the impact of failures in NodeWiz using our implementation. Our analysis shows that, if not treated, failures can substantially increase the probability of an operation not being appropriately executed. We proposed a simple fault tolerance mechanism that significantly decreases NodeWiz's unavailability due to failures.

We have a working implementation of NodeWiz which is being incorporated into the OurGrid middleware to enhance its matchmaking mechanism. OurGrid is a P2P, free-to-join grid for bag-of-tasks applications. It has been in production since December 2004 [6].

Acknowledgements This work has been developed in collaboration with HP Brazil R&D. Authors would like to thank Jonhnny Wesley Sousa Silva and Giovanni Farias da Silva for the help in running part of the experiments. Francisco Brasileiro thanks the support from CNPq - Brazil (grant 309033/2007-1).

References

1. Andrzejak A, Xu Z (2002) Scalable, efficient range queries for grid information services. In: Proceedings of the second IEEE international conference on peer-to-peer computing (P2P'02), Linköping University, Sweden. IEEE Computer Society Press, Silver Spring, pp 33–40. citeseer.ist.psu.edu/andrzejak02scalable.html
2. Balazinska M, Balakrishnan H, Karger D (2002) INS/Twine: a scalable peer-to-peer architecture for intentional resource discovery. In: Proceedings of the pervasive 2002
3. Basu S, Banerjee S, Sharma P, Lee SJ (2005) Nodewiz: peer-to-peer resource discovery for grids. Tech. Rep. HPL-2005-36, HP Labs
4. Bharambe AR, Agrawal M, Seshan S (2004) Mercury: supporting scalable multi-attribute range queries. In: Proceedings of the 2004 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM'04). Portland, pp 353–366
5. Cai M, Frank MR, Chen J, Szekely PA (2003) MAAN: a multi-attribute addressable network for grid information services. In: Fourth international workshop on grid computing (GRID'03), pp 184–191
6. Cirne W, Brasileiro F, Andrade N, Costa L, Andrade A, Novaes R, Mowbray M (2006) Labs of the world, unite!!! J Grid Comput 4(3):225–246. doi:10.1007/s10723-006-9040-x
7. Czajkowski K, Kesselman C, Fitzgerald S, Foster IT (2001) Grid information services for distributed resource sharing. In: 10th IEEE international symposium on high performance distributed computing (HPDC'01). IEEE Computer Society, Silver Spring, pp 181–194
8. Ganesan P, Yang B, Garcia-Molina H (2004) One torus to rule them all: multi-dimensional queries in p2p systems. In: Proceedings of the WebDB 2004
9. PlanetLab (2009) Ganglia archives for PlanetLab. http://planetlab.millennium.berkeley.edu/
10. Globus (2009) Globus Toolkit. http://www.globus.org/toolkit/
11. Han J, Kamber M (2001) Data mining: concepts and techniques, chap. 8: cluster analysis. Morgan Kaufmann, San Mateo, pp 349–351
12. Harvey NJA, Jones MB, Saroiu S, Theimer M, Wolman A (2003) SkipNet: a scalable overlay network with practical locality properties. In: Proceedings of the USITS 2003
13. Huebsch R, Hellerstein JM, Lanham N, Loo BT, Shenker S, Stoica I (2003) Querying the internet with PIER. In: 19th international conference on very large databases (VLDB'03), pp 321–332
14. Iamnitchi A, Foster I (2001) On fully decentralized resource discovery in grid environments. In: Proceedings of the international workshop on grid computing
15. Iamnitchi A, Foster I (2003) A peer-to-peer approach to resource location in grid environments. In: Weglarz J, Nabrzyski J, Schopf J, Stroinski M (eds) Grid resource management. Kluwer, London
16. von Laszewski G, Fitzgerald S, Foster I, Kesselman C, Smith W, Tuecke S (1997) A directory service for configuring high-performance distributed computations. In Proceedings of the IEEE HPDC-6, pp 365–375
17. Li X, Kim YJ, Govindan R, Hong W (2003) Multidimensional range queries in sensor networks. In: Proceedings of the ACM SenSys 2003
18. Lima A, Cirne W, Brasileiro F, Fireman D (2006) A case for event-driven distributed objects. In: 8th international symposium on distributed objects and applications (DOA). Springer, Berlin Heidelberg New York, pp. 1705–1721. doi:10.1007/11914952_46
19. Oppenheimer D, Albrecht J, Patterson D, Vahdat A (2004) Distributed resource discovery on planetlab with SWORD. In: Proceedings of the first workshop on real, large distributed systems (WORLDS 2004)
20. OurGrid (2009) OurGrid project. http://www.ourgrid.org/
21. Ramabhadran S, Ratnasamy S, Hellerstein JM, Shenker S (2004) Brief announcement: prefix hash tree. In: Proceedings of the 23rd annual ACM symposium on principles of

distributed computing (PODC'04). ACM, New York, pp 368–368. doi:10.1145/1011767.1011823
22. Raman R, Livny M, Solomon M (1998) Matchmaking: distributed resource management for high throughput computing. In: Proceedings of the IEEE HPDC-7. Chicago, IL
23. Schopf JM, Raicu I, Pearlman L, Miller N, Kesselman C, Foster I, D'Arcy M (2006) Monitoring and discovery in a web services framework: functionality and perform ance of globus toolkit mds4. Tech. rep., Argonne National Labs
24. Thain D, Tannenbaum T, Livny M (2005) Distributed computing in practice: the condor experience: research articles. Concurr Comput Pract Exper 17(2–4):323–356. doi:10.1002/cpe.v17:2/4
25. Wahl M, Howes T, Kille S (1997) Rfc 2251: lightweight directory access protocol (v3)
26. XMPP Standards Foundation (2007) Extensible messaging and presence protocol (XMPP). http://www.xmpp.org
27. Zhang C, Krishnamurthy A, Wang RY (2005) Brushwood: distributed trees in peer-to-peer systems. In: 4th international workshop on peer-to-peer systems. Lecture notes in computer science, vol 3640. Springer, Ithaca
28. Zheng C, Shen G, Li S, Shenker S (2006) Distributed segment tree: support of range query and cover query over dht. In: 5th international workshop on peer-to-peer systems (IPTPS'06), Santa Barbara

Lauro Beltrão Costa received a BS degree in Computer Science from Universidade Federal de Campina Grande (UFCG), Brazil and an MSc from the same University in 2004 and 2006, respectively. He has worked as research assistant and developer in the OurGrid project at the Distributed Systems Lab of UFCG. He joined LSD in 2001 as an undergraduate student and he has been contributing as research assistant since the beginning of 2006. He is interested in folksonomy and resource/service discovery for grids. Currently, he is a visiting fellow at the Fraunhofer Institute ITWM, Germany, contributing to the Jawari Grid benchmarking project.

Sujoy Basu is Senior Research Scientist in HP Labs. He joined HP in 2000 after receiving his PhD in Computer Science from the University of Illinois at Urbana-Champaign. He also holds Master and Bachelor degrees in Computer Science from Georgia Tech and Indian Institute of Technology (IIT) Kanpur respectively. At HP Labs, his recent research has been on grid and utility computing and overlay network services. The current focus of his work is on services automation and integration. He is a Senior Member of IEEE and a member of ACM. He has been awarded 8 patents, and has several more pending.

Francisco Brasileiro is a Professor at the Universidade Federal de Campina Grande, Brazil. He received a BS degree in Computer Science from the Universidade Federal da Paraíba, Brazil, in 1988, an MSc degree from the same University in 1989, and a PhD degree in Computer Science from the University of Newcastle upon Tyne, UK, in 1995. His research interests include dependability in distributed systems, grid computing and distributed algorithms and protocols. Currently, Professor Brasileiro is a research leader in the OurGrid project. He is a member of the Brazilian Computer Society, the ACM, and the IEEE Computer Society.

Sujata Banerjee joined HP Labs in August 2000 where she is currently Principal Research Scientist and leader of the Networking Research Group. Prior to joining HP, she was an Associate Professor of Telecommunications at the University of Pittsburgh. She earned a Ph.D. degree in Electrical Engineering from the University of Southern California in Los Angeles, and Bachelors and Masters degrees in Electrical Engineering from the Indian Institute of Technology, Bombay in India. Her research interests are in quality of service issues in networked systems. She is a recipient of the National Science Foundation CAREER award in Networking Research. She has served as an Associate Editor for the IEEE Transactions on Reliability, as well as on the technical program committee of several conferences and workshops. She is a senior member of the IEEE.

Puneet Sharma received a Ph.D. in Computer Science from the University of Southern California, Los Angeles in 1998. Prior to that he earned a B.Tech. in Computer Science and Engineering from the Indian Institute of Technology, Delhi. Currently, he is a Senior Research Scientist at Hewlett-Packard Laboratories, Palo Alto, California. At HP labs he conducts research in Wireless and Mobile Networking, Overlay Network Services, Network Measurement and Monitoring.

Sung-Ju Lee is a senior research scientist at HP Labs. He received his PhD in Computer Science from the University of California, Los Angeles in 2000. His research interests include wireless mesh networks, network management, and large scale network systems. He is a senior member of IEEE and ACM.