A large scale fault-tolerant grid information service

Francisco Brasileiro (Contact author), Lauro Beltrao Costa, Alisson Andrade, Walfredo Cirne
{*fubica,lauro,aandrade,walfredo*}*@dsc.ufcg.edu.br*
Universidade Federal de Campina Grande
Departamento de Sistemas e Computaçao
Av. Aprígio Veloso, s/n
58.109-970, Campina Grande, PB, Brazil

Sujoy Basu, Sujata Banerjee
{*Sujoy.Basu,Sujata.Banerjee*}*@hp.com*
Hewlett-Packard Laboratories
Palo Alto, CA 94304, USA

ABSTRACT

Large scale grid systems may provide multitudinous services, from different providers, whose quality of service will vary. Moreover, services appear (and disappear) in the grid with no central coordination. Thus, to find out the most suitable service to fulfill their needs, grid users must resort to Grid Information Services (GISs). These services allow 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 GISs. 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 faults. In this paper we focus on peer-to-peer GISs that efficiently support rich queries. In particular, we thoroughly analyze the impact of faults in one representant of such GISs, named NodeWiz. We propose extensions that increase NodeWiz's resilience to faults.

Keywords: Grid Information Service; peer-to-peer; kd-tree; Failure Detection; Availability.

1. INTRODUCTION

Computational grids have recently emerged as a promising environment for the provision of computation as services [8]. 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.

Grid Information Services (GISs) have been proposed to help users in the task of choosing which service to use to better fulfil their needs [6]. A GIS can be seen as a directory in which providers publish their adverts and to which users submit their queries. Obviously, to be useful for large grids, GISs must be scalable. Moreover, in a system with potentially many thousands of components, faults are the norm and not the exception. Therefore, GISs should also be fault-tolerant.

The early GISs were either centralized or based on a static hierarchical directory to which all advertisements and queries were sent [6, 7]. These architectures, however, are not scalable or too inflexible to support the rich queries a GIS needs to support. More recent approaches rely on some scalable structured peer-to-peer substrate on top of which the directory service is built [1–3, 9–12]. Most of these systems rely on Distributed Hash Tables (DHTs) to implement structured peer-to-peer directories. DHTs are scalable and very robust to faults. On the other hand, the only search operation that is efficiently supported by DHTs is exact matches. Unfortunately, this is too restrictive for a GIS. In order to describe appropriately the required service, users usually need to send out queries that contain multiple attributes and that specify ranges of admissible values. Such queries are not supported in a natural way by DHT-based substrates [10].

Tree-based structured peer-to-peer substrates have been proposed as an alternative to the implementation of scalable GIS, supporting rich queries in an efficient way [2, 12]. However, tree-based substrates are much less resilient to faults than DHT-based ones. In this paper we analyze the impact of faults in such systems and propose a mechanism that can be added to them such that faults are appropriately dealt with. For the sake of clarity, we present the proposed mechanism in the context of a particular tree-based, peer-to-peer GIS, named NodeWiz [2].

The rest of the paper is structured as follows. We start by discussing related work in Section 2. To render the paper self-contained, in Section 3 we give a brief description of NodeWiz's design. Then, in Section 4 we evaluate the impact of faults in NodeWiz. In Section 5 we discuss how NodeWiz's design can be extended so that faults are successfully treated. Section 6 concludes the paper with our final remarks and a brief discussion on future work.

2. RELATED WORK

There are several works that support multi-attribute and range queries by building overlay DHTs (e.g. SWORD [10], PHT [11], and MAAN [4]). Fault tolerance is provided by the underlying peer-to-peer substrate. However, to allow operations on multiple attributes, several DHTs – one per attribute – need to be built and 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, the number of updates associated to either the update or the query operations increases proportionally.

Mercury [3] is an example of a GIS that supports range queries over multi-attributes. Like SWORD and MAAN, 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.

NodeWiz [2] is a tree-based peer-to-peer substrate. It efficiently supports multiple attributes and range queries maintaining a single distributed index. However, it was designed for a setting in which faults are rare and faulty peers would eventually recover from failures, restoring their state from stable storage. BrushWood [12] follows a similar approach. If instantiated as a kd-tree, it is very similar to NodeWiz. Like NodeWiz, BrushWood has no fault tolerance mechanism for routing. Therefore, as we will show in the next section, many operations may be unsuccessful.

3. NODEWIZ FUNCTIONING

NodeWiz [2] is a scalable peer-to-peer GIS whose main goal is to allow multi-attribute range queries to be performed efficiently in a distributed environment. As originally presented, the GIS is implemented by a set of stable infrastructure nodes (or peers) that collectively store adverts from service providers and answer to client queries. Note that NodeWiz peers are GIS entities and should not be confused with server nodes maintained by resource providers, on which application services are hosted. Service providers running at the server nodes store adverts in the GIS, while clients send queries to the GIS so to find the required service. Adverts and queries may specify values for any number of attributes. Moreover, values can represent ranges on the attribute space. For instance, in a grid that provides raw CPU as one of its services, a client that requires a Linux-box with at least $1024 Mbytes$ of memory and low load, could issue the following query: $OS = linux \wedge Mem \geq 1024 \wedge Load < 0.2$.

Each peer is in charge of a subspace of the entire attribute space available in the system. Therefore, depending on the queries and adverts submitted to the GIS, some of the peers may get overloaded. In order to solve this problem, NodeWiz allows any overloaded peer to offload some of its load onto other peers, thus keeping the workload balanced. When a new peer J wants to join the system, it asks to any of the peers in the system for the identity of the most overloaded peer in the system. Then, J contacts this peer – say its identity is E – informing that it wants to join the system. E will then divide its subspace with J. E executes an splitting algorithm to identify the best attribute on which to perform the split, as well as the range of values for this attribute that will make both peers have high probability of receiving equal loads of queries and adverts (see [2] for the details on these algorithms).

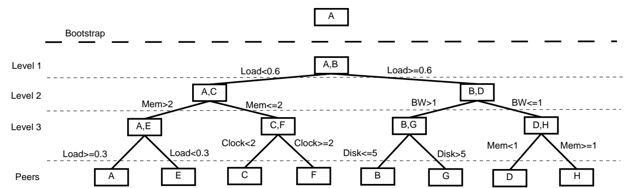

Figure 1: NodeWiz kd-tree (adapted from [2])

The way the attribute space is divided renders a structure that is represented by a kd-tree. 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*. 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 Figure 1.

From now on we will use the term *NodeWiz user*, or simply *user* to refer to both clients and service providers that interact with the system. Also, we will use the term *operation* to refer to queries and adverts issued by users.

When a user wants to issue an operation, it sends the operation request to any peer that it knows in the system; we will refer to this peer as the *recipient* of an operation. The recipient is then in charge of routing the operation request to the appropriate NodeWiz peers, where the wanted subspace is found;we will refer to these peers as the *targets* of an operation.

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. Each entry of the routing table contains an attribute, a range for this attribute, and the identification of the peer that is in charge for serving the attribute and range specified in the entry. Thus, each entry in the routing table indicates the range for a single attribute and a corresponding peer to which operations matching that range for that attribute should be sent. By excluding all attributes and corresponding ranges present in its routing table, a peer obtains the range of attributes for which it is responsible.

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 Figure 1.

Level	Attribute	Min	Max	Peer in charge
1	Load	0.6	+ INF	B
2	Mem	0	2	C
3	Load	0	0.3	E

Table 1: Peer A routing table

NodeWiz was proposed assuming that the peers joining the

system were stable infrastructure nodes. In this setting, it is feasible to assume that nodes are able to store their state in stable storage and can quickly recover from failures. Therefore, the system could be designed assuming a fault-free execution environment and requiring peers to never give up when trying to communicate with another peer.

However, in a more general setting, faulty peers may take a long time to recover. 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. Given the assumptions made in the paper that proposed NodeWiz, these issues were not discussed in depth. In this paper we remove the assumption that the GIS is supported by a stable set of reliable peers. We first quantify the impact of faults in NodeWiz, and then propose extensions to the NodeWiz design to deal with voluntary and involuntary departures in an appropriate way.

4. ON THE CONSEQUENCES OF FAULTS

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 kd-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.

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 $S(op) \in \mathcal{S}$ be the subspace that matches operation op; $target(op) \in \mathcal{G}$ be the set of peers whose attribute subspaces intersect with $S(op)$; 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 $|target(op)| = 1$; 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 kd-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 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 kd-tree presented in Figure 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 Equation 1:

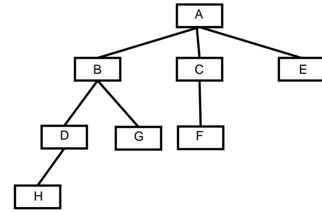

Figure 2: Routing tree for peer A

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

4.2 Evaluating the impact of faults

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 $target(op)$ must be composed exclusively of non-faulty peers. The probability of an i-hop route to contain only non-faulty peers is given by[1]:

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

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 through an i-hop route. This is expressed by Equation 3:

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

Figure 3 plots Equation 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 faults, then it is mandatory to take recovery actions so that the system will not collapse. In the next section we show how NodeWiz can be extended to cope well with faults.

9[1]The exponent $(i + 1)$ means the number of hops in the route added to 1, which represents the recipient peer itself.

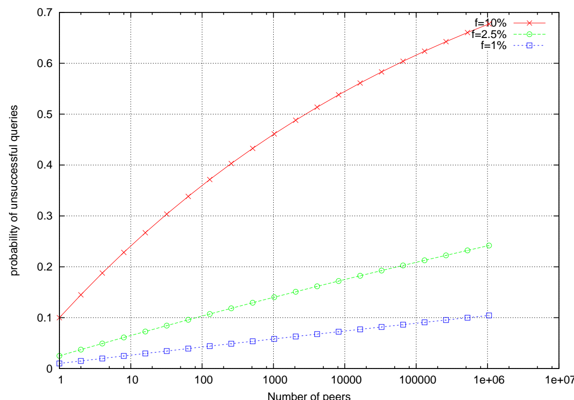

Figure 3: Probability of unsuccessful operations in a system with $f_t \cdot \mathcal{N}$ faulty peers

5. FAULT-TOLERANT NODEWIZ

As discussed in the previous section, faults have an important impact on NodeWiz functioning. In fact, even voluntary leaves must be dealt with care to maintain the routing tables consistent. For the sake of clarity, before discussing the approach that we propose to deal with involuntary leaves, i.e. faults, we will first explain in details how voluntary leaves is treated, assuming a fault-free scenario.

5.1 Voluntary leaves in a fault-free system

When a peer leaves the system, the attribute subspace that it stores must be reclaimed by another peer in the system. Let $R(L)$ be the replacement of a peer L that wants to leave the system. The simplest choice is to make $R(L)$ the last peer with whom the leaving peer L has split its attribute subspace (or its replacement, if that peer has already left the system) [2]. Moreover, the system must ensure that every routing table that contains a reference to the leaving peer L will be updated, by either removing this entry from the routing table or replacing the reference to L by a reference to $R(L)$. Note that $R(L)$ is the peer that appears in the last entry of L's routing table

When the leave is voluntary, the leaving peer L contacts $R(L)$, informing $R(L)$ that L wants to leave the system. At this point, L also sends to $R(L)$ its local state, i.e. all adverts it has locally stored. Then, $R(L)$ 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 being informed by $R(L)$ that the execution of the leaving procedure has completed. In the meantime, any operation that is routed through L is forwarded by L to $R(L)$, so that $R(L)$ 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 $R(L)$ itself and the peers that have inherited from $R(L)$ an entry for L in their routing tables (if any), i.e. those peers that are in the branches of $R(L)$'s routing tree whose roots are any of the peers that appear in $R(L)$'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 $R(L)$. 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 L's leave notification, $R(L)$ 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 $R(L)$'s routing table below L's entry. Recursively, the request is sent down in the branches of $R(L)$'s routing tree whose roots are these peers. Acknowledgements 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 $R(L)$ receives this acknowledgement it knows that the replacement has been completed by all peers that had inherited from $R(L)$ an entry to L in their routing tables.

$R(L)$ 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 $R(L)$'s routing table (if any) to replace the reference to L by a reference to $R(L)$ in its routing table. This peer (say, U) starts a recursive update until all references to L are replaced by references to $R(L)$. When this happens, U sends an acknowledgement to $R(L)$.

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[2] (if any) to replace references to L by references to $R(L)$. 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 to $R(L)$. 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 $R(L)$ receives acknowledgements 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. 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, avoiding failed operations. Given that adverts are constantly renewed, if the

9[2]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.

routing tables are kept consistent, then the loss of state is not permanent, and only results in fewer hits for queries that overlap the attribute subspace of the faulty peer during the time period when the adverts have not all been renewed. On the other hand, if an effort is not made to keep the routing tables consistent, then a substantial number of operations will fail.

The rationale of the fault tolerance mechanism proposed is very simple. It requires $R(P)$ to replace P upon detecting that P has failed. In a system where peers may fail, this peer will be either the peer with whom P has last split its attribute subspace, or the replacement of such peer, in case it has left the system or failed. Again, $R(P)$ is the peer that appears in the last entry of P's routing table. When P fails, $R(P)$ 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 $R(P)$ is also executing the virtual peer $S(P)$. When $S(P)$ starts to execute it asks $R(P)$ to authorize $S(P)$ to leave the system and waits for the corresponding authorization. This mimics a voluntary leave operation issued by the faulty P. Upon receiving such request, $R(P)$ 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 $R(P)$. $S(P)$ then terminates its execution and the system is consistent again.

To deal with simultaneous failures, each peer in the system monitors one or more sets of peers. Let n be the number of levels in which a peer P appears in the kd-tree (for instance, in the example of Figure 1, A and B appear in three levels, while E, F, G and H appear only in one). 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 Figure 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. In the the kd-tree of Figure 1, A monitors three sets: $\{B, D, G, H\}$, with B as set leader; $\{C, F\}$, with C as set leader; and, $\{E\}$, with E as set leader. On the other hand, E monitors only the set $\{A\}$, with A as 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.

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 (resp. P) starts monitoring a new set of peers that has as leader (and single member) P (resp. Q). Moreover, Q must inform all peers that have a monitoring set $\mathcal{M}$, such that $Q \in \mathcal{M}$, that they should also add P to $\mathcal{M}$. On the other hand, when a peer P leaves the system, all peers that monitor P should be notified and should stop monitoring P. Considering the kd-tree of Figure 1, when H joins the system, then D (resp. H) starts monitoring a new set with only H (resp. D) as member, and D informs B and A about the new peer that they need to add to their level 2 and 1 monitoring sets, respectively. (Note that these actions can be easily embedded in the leave procedure described earlier, obviating the need for any extra exchange of messages.)

In NodeWiz's original design, 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 it were P. Like in the voluntary leave case, $S(P)$ forwards to $R(P)$ any operation that is routed through $S(P)$.

Another concern is the impact that a fault may have in 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 is easy to be fixed. Whenever a peer's local table is updated, it checks if there are new peers that need to be asked to update their routing tables, and any required update operation for which a corresponding acknowledgement has not been received is issued. Also, peers should only update their tables after they have received acknowledgments for the updates they have requested.

5.3 Implementation issues

In this section we discuss some issues that should be carefully addressed to render the implementation of the fault-tolerant version of NodeWiz scalable.

To avoid that some operation fail while an update is on-going, the fault tolerance mechanism requires a name service. Obviously, the name service needs also 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 over a name service. Moreover, the DHT can be easily implemented by the same peers that implement NodeWiz.

Another concern is the fact that, although the attribute subspace is 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 kd-tree, have more routing load than the peers that only appear at the levels closer to the bottom of the tree. In [2] a caching strategy, named routing diversity, is used to effectively cope with this problem.

Finally, although voluntary and involuntary leave procedures require a number of updates, message exchanges and local storage capacity is $O(log_2\mathcal{N})$ in 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 fresher in the system. 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 instead of monitoring all peers in a given set, monitor only the set leader. When the failure of the set leader is detected, then the peer starts monitoring all 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, in a well-formed system, the maximum number of peers that any peer would be simultaneously monitoring is $O(log_2\mathcal{N})$, with half of the peers in the system monitoring 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.

6. CONCLUSION

In this paper we have evaluated the impact of faults in NodeWiz, a tree-based, peer-to-peer GIS that allows for efficient query/advertisement operations involving ranges and multiple attributes. Our analysis shows that, if not treated, faults can substantially increase the probability of an operation not being appropriately executed. We proposed a simple fault tolerance mechanism that is easily adapted to NodeWiz and that significantly decreases its unavailability due to faults.

We are currently in the process of introducing the fault tolerance mechanism proposed in a NodeWiz implementation. Our goal is to use the fault-tolerant GIS to enhance matchmaking in the OurGrid community (see www.ourgrid.org), a peer-to-peer, free-to-join grid for bag-of-tasks applications that has been in production since December 2004 [5].

Acknowledgments

This work was partially developed in collaboration with HP Brazil R&D. Francisco Brasileiro (grant 307.954/2004) and Walfredo Cirne (grant 141.655/2002) thank the financial support from CNPq/Brazil.

7. REFERENCES

[1] Andrzejak, A., and Xu, Z. Scalable, efficient range queries for grid information services. In *Proceedings of the Second IEEE International Conference on Peer-to-Peer Computing (P2P'02)* (September 2002), Linköping University, Sweden, IEEE Computer Society Press, pp. 33–40.

[2] Basu, S., Banerjee, S., Sharma, P., and Lee, S.-J. Nodewiz: Peer-to-peer resource discovery for grids. In *Proceedings of* 5th *International Workshop on Global and Peer-to-Peer Computing* (May 2005), IEEE Computer Society Press, pp. 213–220.

[3] Bharambe, A. R., Agrawal, M., and Seshan, S. 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, OR, USA, September 2004), pp. 353–366.

[4] Cai, M., Frank, M. R., Chen, J., and Szekely, P. A. MAAN: A multi-attribute addressable network for grid information services. In *Fourth International Workshop on Grid Computing (GRID'03)* (2003), pp. 184–191.

[5] Cirne, W., Brasileiro, F., Andrade, N., Costa, L. B., Andrade, A., Novaes, R., and Mowbray, M. Labs of the world, unite!!! *Journal of Grid Computing* (2006). Accepted for publication.

[6] Czajkowski, K., Kesselman, C., Fitzgerald, S., and Foster, I. T. Grid information services for distributed resource sharing. In 10th *IEEE International Symposium on High Performance Distributed Computing (HPDC'01)* (August 2001), IEEE Computer Society Press, pp. 181–194.

[7] Fitzgerald, S., Foster, I., Kesselman, C., von Laszewski, G., Smith, W., and Tuecke, S. A directory service for configuring high-performance distributed computations. In 6th *IEEE International Symposium on High Performance Distributed Computing (HPDC'97)* (August 1997), IEEE Computer Society Press, pp. 365–375.

[8] Foster, I., Kesselman, C., and Tuecke, S. The anatomy of the Grid: Enabling scalable virtual organizations. *International Journal of High Performance Computing Applications 15*, 3 (August 2001), 200–222.

[9] Huebsch, R., Hellerstein, J. M., Boon, N. L., Loo, T., Shenker, S., and Stoica, I. Querying the internet with PIER. In 19th *International Conference on Very Large Databases (VLDB'03)* (September 2003), pp. 321–332.

[10] Oppenheimer, D., Albrecht, J., Patterson, D., and Vahdat, A. Distributed Resource Discovery on PlanetLab with SWORD. In *Proceedings of the* 1st *ACM/USENIX Workshop on Real, Large Distributed Systems (WORLDS'04)* (December 2004).

[11] Ramabhadran, S., Ratnasamy, S., Hellerstein, J. M., and Shenker, S. Brief announcement: prefix hash tree. In *Proceedings of the* 23rd *Annual ACM Symposium on Principles of Distributed Computing (PODC'04)* (New York, NY, USA, 2004), ACM Press, pp. 368–368.

[12] Zhang, C., Krishnamurthy, A., and Wang, R. Y. Brushwood: Distributed trees in peer-to-peer systems. In 4th *International Workshop on Peer-To-Peer Systems* (Ithaca, New York, USA, February 2005), vol. 3640 of *Lecture Notes in Computer Science*, Springer.