dFault: Fault Localization in Large-Scale Peer-to-Peer Systems

Pawan Prakash[1], Ramana Rao Kompella[1],
Venugopalan Ramasubramanian[2], and Ranveer Chandra[2]

[1] Purdue University, {pprakash,kompella}@cs.purdue.edu
[2] †Microsoft Research, {rama,ranveer}@microsoft.com

Abstract. Distributed hash tables (DHTs) have been adopted as a building block for large-scale distributed systems. The upshot of this success is that their robust operation is even more important as mission-critical applications begin to be layered on them. Even though DHTs can detect and heal around unresponsive hosts and disconnected links, several hidden faults and performance bottlenecks go undetected, resulting in unanswered queries and delayed responses. In this paper, we propose dFault, a system that helps large-scale DHTs to localize such faults. Informed with a log of failed queries called *symptoms* and some available information about the hosts in the DHT, dFault identifies the potential *root causes* (hosts and overlay links) that with high likelihood contributed towards those symptoms. Its design is based on the recently proposed dependency graph modeling and inference approach for fault localization. We describe the design of dFault, and show that it can accurately localize the root causes of faults with modest amount of information collected from individual nodes using a real prototype deployed over PlanetLab.

Keywords: distributed systems, DHTs, fault localization, PlanetLab

1 Introduction

Distributed Hash Tables (DHTs) have gained widespread acceptance as building blocks of large-scale distributed systems. From the initial success of publicly deployed academic research projects such as CoDoNS [19], CoralCDN [5] and OpenDHT [22] to the more recent enterprise, cloud storage systems such as Amazon's Dynamo [7] and its open-source equivalent Voldemort [26], DHT-based systems are becoming increasingly prevalent. As even more critical services and commercial applications begin to rely on DHTs, their robustness—ability to run free of errors and inefficiencies—would be crucial.

Fortunately, DHTs have high resilience and can "heal" themselves quickly and effectively upon detecting a failure. In fact, they do so well for failures that are visible to their protocols—namely, broken network connections and unresponsive or crashed hosts. However, they are not currently equipped to detect other types of critical but "latent" failures. For example, a fault (*e.g.*, bug or race condition) in a critical application component might keep a host up and

responsive to protocol messages while silently dropping application messages. Or, a bad network connection might unpredictably delay the delivery of some messages, without triggering the DHT host to break the connection. These and other subtle faults identified by recent studies on widely deployed DHTs [6, 16, 21], which go undetected and therefore not recovered from, ultimately affect the performance and availability of the system.

Our goal is to automate the *localization* of such faults. That is, identify components that potentially contain faults, so that the necessary repair actions can then be taken. One approach for fault localization is to instrument the DHT code heavily to individually track messages as they pass between different hosts and components and log them. However, this approach is likely to result in a massive collection of logs to be pored over—an overwhelmingly expensive task for large-scale deployments.

We propose a light-weight probabilistic inference approach for fault localization. We start from the *symptoms* of the faults, namely unanswered queries and unusually delayed responses, visible to the consumers or clients of the DHT. Their *root causes* lie in the failures and performance bottlenecks at some system component. Now, if many of the failed queries are meant for keys that have a common home node (the primary host in a DHT responsible for storing items associated with that key), then we can infer that a fault at that home node likely caused the failure of those queries.

The above simple example highlights the *key intuition* behind our approach. However, probabilistic inference in practice is more complicated. For instance, it is possible in the above example that the queries share another common host through which they all traverse before reaching the home node. In this case, the root cause could be either of the two hosts, or both, or in the network route between them. Additional information from the routing tables at the DHT hosts and failure statistics of other queries is necessary to refine the inference further. Similarly, if there is a cache at an intermediate host in the path taken by a query that happens to contain the response for the query, the cache would mask faults in subsequent hosts in the path. A suitable cache model or some knowledge of cache contents may be necessary to improve the effectiveness of fault localization even further.

In this paper, we present a new system called dFault for localizing faults in DHT-based applications and services. dFault is designed to work on any DHT. It runs as a centralized service that collects a list of query failures from participating DHT clients and a small amount of routing and cache information from individual DHT hosts. (It uses several optimizations to ensure that the system incurs low communication bandwidth and scales well.) It then uses the collected information to diagnose the root causes for a set of query failures, which includes both hard failures (unanswered queries) as well as performance degradations (delayed responses). Internally, dFault builds a dependency graph consisting of the *symptom set* (failed queries) and *root causes* (hosts, overlay links) and uses a new probabilistic inference algorithm that meets the scalability requirements of these systems.

We have evaluated dFault through a Pastry [23] deployment on 100 PlanetLab hosts, as well as scaled simulations. We found that dFault could localize 5 simultaneous failures (nodes injected with faults) with 100% accuracy and no false positives with as low as 110 symptoms. Furthermore, our estimates show that the total communication bandwidth required for accurate fault localization is of the order of about 1.2 Kbps per node (for a system of 100,000 hosts this translates to about 100 Mbps total bandwidth at the diagnosis entity).

Overall, this paper makes the following key contributions: (1) First, it presents a new fault localization framework for DHTs based on the light-weight approach of dependency modeling and inference. It shows how to develop probabilistic dependency graphs connecting queries with the DHT components they depend on, incorporating practical aspects of caching and routing. (2) Second, it presents a novel inference algorithm suitable for the large probabilistic dependency graphs we derive. (3) Finally, it characterizes the tradeoff between the fidelity of information included in the model and the effectiveness of fault localization, showing that accurate and precise localization can be achieved for large scale DHT-based systems with modest overhead.

2 Background

A DHT is a networked system of hosts that collectively provides a *key-value* storage service. The DHT provides a core interface of *put(key,value)* and *get(key)* operations for accessing the storage service. Several DHTs [17, 20, 23, 25] have been designed to provide this abstraction, each with a different topology and routing algorithm; we base our work on principles common to most DHTs.

DHTs assign a *home node* to each key as the host responsible for storing the items associated with the key. Other hosts in a DHT may further replicate the items for failure resilience and performance improvement. The most common approach is *consistent hashing* [12], where both host identifiers and item keys are hashed to the same space, and the host whose identifier falls "closest" to the key in the hashed space is set as its home node.

An application can store or retrieve the items associated with a key by locating the key's home node. This process usually involves a multi-hop routing protocol. First, the application issues a *get* or a *put* operation for a specific key K to the local DHT host A it interacts with. A then forwards the operation to another host B, which it deems to be closer to the home node. Unless, A itself happens to be the home node, in which case, it provides the items to the application. The above step continues until the operation reaches the home node. Each DHT host maintains links to a subset of other hosts in the DHT in a *routing table*, using which given a key K, a host can easily find a next hop that is closer (in the hashed space) to the key than A to forward to. DHTs have protocols for failure-detection and routing-table repair in order to ensure that correct and efficient routing tables are maintained at the hosts and operations complete within asymptotically bounded worst-case latency (often $O(\log N)$ hops, where N is the number of hosts).

However, despite the asymptotic worst-case bounds, DHT operations can incur long delays as the multi-hop routing protocol crisscrosses over many links.

So, DHTs often use caching to decrease the average latency of *get* operations. While there are aggressive DHT caching solutions that guarantee high performance through sophisticated algorithms [18], the most commonly used approach is to cache the results of a *get* operation at intermediate hosts through which it was routed through. When a host finds a hit in its cache, it returns the items directly instead of forwarding it to the home node. To manage the cache, hosts may use any well-known cache eviction policies such as least-recently-used (LRU) or least-frequently-used (LFU).

3 dFault system overview

Our system, dFault, focuses on a class of *latent failures* that have been known to exist in DHTs which the in-built failure detection mechanisms cannot automatically detect and repair. These pathologies and inefficiencies typically lead to failed queries or excessively delayed responses [6, 16, 21]. Examples of such latent faults include: (1) a failed application or component that stops processing messages at a host (even though the host itself is responsive to protocol messages); (2) a subtle network problem such as the lack of transitivity (*i.e.*, a scenario where a host B can forward a query it received from host A to host C, but host C is unable to send a response back to host A due to a network problem); or, (3) a weak network connection or performance bottleneck at a host that delays a message or slows down its processing.

Our goal is to design a system that can localize the root causes of failures and inefficiencies in DHT-based applications. A system for fault localization in DHTs should ideally possess three primary properties.

• *P1) Accuracy.* Our system should be able to accurately detect a significant fraction if not all the failures. Further, the number of false positives reported should be quite low.

• *P2) Scalability.* We seek to apply dFault in large-scale DHT systems with thousands of nodes and queries per second. Therefore, dFault needs to be lightweight—prudent in the quantity of information it uses and quick in the execution of the inference algorithm. This rules out several Bayesian schemes that have been proposed in the literature [10, 2].

• *P3) Non-intrusiveness.* We seek to minimize heavy, intrusive modifications to the DHT system and its applications. Although more accurate localization might be possible by heavily instrumenting the DHT to log more data, such an approach is cumbersome and excludes legacy applications.

3.1 Architecture of dFault

dFault targets applications running on managed DHTs, where it is possible to monitor application performance and collect required information for diagnosis. A *centralized diagnosis server* (CDS) forms the core of dFault where appropriate information is collected from individual nodes in a DHT for diagnosis. For redundancy purposes, multiple CDS servers can be used by the management entity. The CDS obtains three types of information from DHT nodes: (1) failure symptoms in the form of failed or delayed queries, (2) routing tables, and (3)

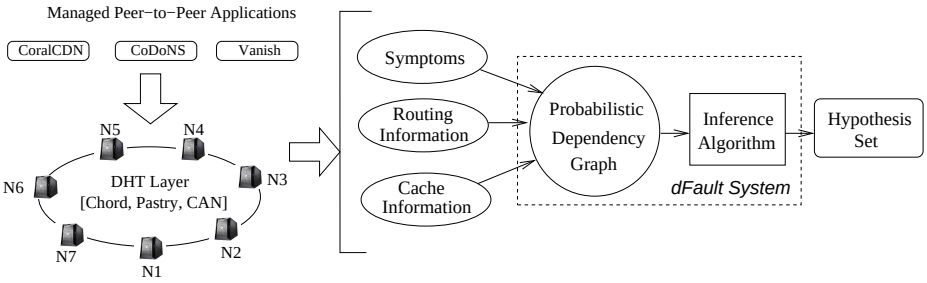


Fig. 1. dFault system overview

relevant caching information in the form of, say, cache contents. It then constructs a probabilistic dependency graph that codifies the dependencies between *root-causes* (nodes and overlay links in the DHT) and the failure symptoms. Finally, the CDS executes an inference algorithm to determine the most likely set of root causes that can help explain the particular set of failed queries. Figure 1 illustrates this architecture of dFault.

In order to scale to large deployments, CDS minimizes the amount of information for diagnosis from each node. For instance, it uses only a sample of failure symptoms for diagnosis (evaluation indicates 100 failed symptoms are sufficient for good accuracy). Routing tables are already quite compact (scale as $O(\log N)$) and the associated overhead can be further reduced by encoding only the changes in routing table entries. We perform similar optimizations for cache details that makes the system overall quite scalable.

Failure symptoms. A query that fails to evoke a response or obtains a response delayed beyond a certain time threshold is considered a *failure symptom*. A failure symptom is detected at the first DHT node encountered by the query, either by the application running on that node or the DHT component that originates the query on behalf of the actual user client. dFault can monitor application failures in several ways. Actively monitoring each node for failed queries is unscalable; therefore, we follow a passive 'reporting' approach, whereby each node alerts the CDS whenever a query fails. In a real operational system, it is impractical to assume all the failure symptoms will be accurately reported in a timely fashion. Further, there could be a large number of spurious and transient symptoms in a large scale distributed environment such as in our setting. Our system therefore does *not* require all the failure symptoms from all the nodes before it can perform diagnosis. Indeed, dFault can generate an inference hypothesis based on whatever subset of the failed queries that are input to the system.

Routing tables. Fault localization in dFault is enabled with the help of a dependency graph that dictates which set of nodes $N_{i1}, N_{i2}, ..., N_{ik}$ in the system a given query q_i that originates at node N_{i1} depends on. One way to achieve this is to allow every node in the system to log every query that passes through them, and subsequently collect all these logs at the CDS. This approach, however, is likely to be extremely intensive in communication going against our scalability goal ($P2$). As discussed in Section 2, nodes in a DHT use a routing table that

helps identify the host closest to the query key. Instead of enabling each node to track dependencies directly, therefore, it is prudent to collect the routing tables from each node, that are typically compact consisting of only $O(\log N)$ entries. Once the base routing tables are collected, subsequent runs require only the incremental changes which will be even smaller than the entire routing tables. Thus, in dFault, each node transmits its routing table entries periodically to the CDS which, as we describe later in Section 4.1, constructs the appropriate dependency graph required for fault localization.

Note that our system only depends on the ability to determine the path from a source node to a home node within a given DHT. Thus, it can work with several DHTs such as Pastry, Chord, and others where it is easy to compute such paths just with the knowledge of routing tables from individual nodes. In some environments, keeping track of all the DHT nodes' routing tables at the CDS precisely synchronized can be hard, especially when the number of DHT nodes is really huge. Updates from the DHT nodes may be lost, connections may be terminated or could be extremely slow in transferring the routing tables. Thus, dFault is designed to work with partial and stale routing tables in diagnosing failures. We evaluate the efficacy of dFault under such conditions in Section 5.

Cache information. In the context of dFault, caching in the DHT makes fault localization harder because, the path a query is supposed to take from a source node to a home node may only be *partially* traversed. Thus, while hypothetically, the query depends on all the nodes along the path, it is actually only dependent on a subset of nodes until it hits the first cache along the path that contains the query's response.

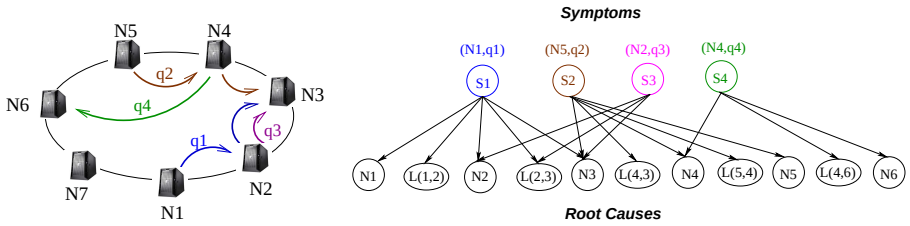
We consider three options that explore different tradeoffs in modeling caches.

• 1) *No cache.* The first option is to choose not to model the caching effects at all in our dependency graphs. This simplest of all approaches may yield reasonable results if the localization algorithm itself is resilient to small amounts of errors in modeling dependencies.

• 2) *Modeling query popularity.* The second approach we consider is based on the observation that the popularity of most objects stays stable for reasonable lengths of time. For example, in CoDoNS, which uses a DHT to store DNS records, many extremely popular records continue to remain popular for days and months (same for many unpopular records). Thus, we can estimate object popularity and obtain a probabilistic value that dictates how likely a given object is to be cached at any given node.

• 3) *Polling cache contents.* Finally, we consider collecting cache contents at periodic intervals of time from nodes in the system. Note that nodes need not push all of their cache contents; they can just report the differences from the previous transmission to keep the communication overheads low.

From the routing table and caching information collected from the network, dFault needs to construct a dependency graph that describes the relationship between possible root causes and the failure symptoms collected from the network. The construction of the dependency graph is dependent on the particular



(a) Example network. (b) Corresponding dependency graph.

Fig. 2. Constructing dependency graph from routing tables.

information available; we show this construction for each of these three options next.

4 Probabilistic inference

dFault uses the failure symptoms, routing table and cache information collected from individual DHT nodes to create a probabilistic dependency graph that models the relationship between the observed failure symptoms and the set of root causes (DHT nodes and overlay links between these nodes). Based on this dependency graph, it outputs a hypothesis set of root causes that explain the observed set of failure symptoms using a new scalable inference algorithm. We explain these individually next.

4.1 Dependency graph

In dFault, we formulate the dependencies in the form of a bi-partite dependency graph $G(V, E)$. An edge $e_i \in E$, from an observed failure symptom $S_i \in V$ to a root cause $C_i \in V$ implies that the failure S_i depends on the root cause C_i. Note that the root causes C_i that dFault models (as discussed before in Section 3) include application-level failure at a DHT node (henceforth, referred to as *node*) or network-level failure at an overlay link between two DHT nodes (referred to as *link*).

We define the symptoms collected from the network S_j as the tuple (N_k, q_k) where q_k is the query that has failed in the network, and N_k is the first node in the DHT the query is sent to, referred to as source node. Simplistically, given a DHT maps queries and nodes on to consistent hash space and forwards the queries to the home node N_l, the query q_k is dependent on all the nodes, which according to the routing tables, lie along the path between the source node N_k and home node N_l.

We illustrate this using the following example. In Figure 2(a), we show a DHT network with 7 nodes N_1 through N_7 logically laid out on top of a ring. Symptoms $S_1 = (N_1, q_1)$, $S_2 = (N_5, q_2)$, $S_3 = (N_2, q_3)$, and $S_4 = (N_4, q_4)$ are shown in the Figure, with q_1 being a query issued at source node N_1 that is forwarded to the query's home node N_3. The success or failure of the query is dependent on the set of nodes between and including the source and home nodes, which happens to be the set $\{N_1, L(1,2), N_2, L(2,3), N_3\}$ for S_1, where $L(i,j)$ indicates the link between nodes N_i and N_j. Similarly, symptom S_2 depends on the set of

nodes $\{N_5, L(5,4), N_4, L(4,3), N_3\}$. These nodes are determined directly using the routing table information obtained at the CDS. The corresponding bipartite dependency graph is shown in Figure 2(b), with edges between symptoms and the corresponding root causes.

Each of the edges in the dependency graph is assigned a weight w_i that reflects the strength of this dependency between the symptom and the root cause. The assignment of this weight is dependent on the amount of information we collect and use in modeling the dependencies (as dictated by the three options in Section 3.1). In the first case of *no cache modeling*, the system does not have any additional information to differentiate different root causes in terms of their edge weights. Hence, all edge weights are the same in this simplest of the cases. We discuss the other two cases next.

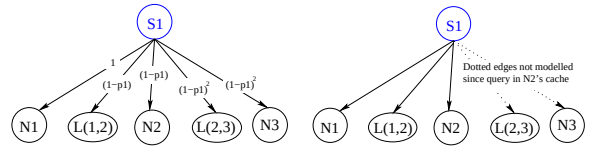
Modeling object popularity. dFault incorporates query popularity to model caching in the dependency graph by associating a probability with each dependency. The probability captures the likelihood that the response to the query will be found in the cache of the dependent node. In general, this cache hit probability needs to be specific to each query; a single global cache hit probability would only work if the queried items are uniformly popular—a rarity in practice [3, 9]. The probability depends on the relative popularity of the query and the number of items the cache can contain. It is higher for popular queries than unpopular queries, and for a query of given popularity, a larger cache is more likely contain its response than a smaller-sized cache.

We capture the above trade-off using the following heuristic formula.

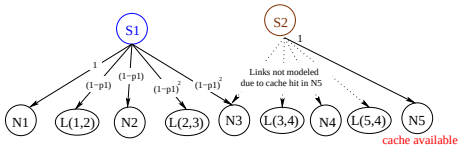
$$p_i = q_i^{\left(\frac{1}{2c-i}\right)} \text{if } i < 2c, \text{ and 0 otherwise.}$$

In this formula, i denotes the rank in the popularity order of the queried key, (*i.e.*, key i is the ith most popular key in the system), q_i denotes the relative popularity of the key i (*i.e.*, the number of queries to key i over total number of queries), and c denotes the number of responses the cache can hold on average.

This formula is designed for heavy-tailed popularity distributions typically found in DHTs [3, 9]. It gives a value very close to 1 for really popular keys, as they are likely to be cached everywhere. Unpopular keys have a negligible likelihood of being cached anywhere. Here, we consider $2c$ as a threshold to define the number of popular keys for which the cache will play a significant role. Finally, for those keys whose popularity falls in the middle, the formula gives a number less than 1 but proportional to the relative popularity. We compute this directly from the query logs at just the source node through random sampling. Each source node forwards periodically (say, daily) its local ordering of queries (along with the number of times a query has been issued) for the top $2c$ queries over a given day. The relatively low frequency of updating this information does not affect the accuracy since object popularities have been shown to persist for days in many systems (*e.g.*, [19]). The CDS computes a global ordering of the queries by aggregating this per-node information to identify the popularity of a given query, and hence, the probability according to the heuristic formula outlined before.



(a) Probabilistic query modeling. (b) Using cache contents.



(c) Hybrid model.

Fig. 3. Dependency graphs depending on query modeling and factoring cache contents.

In Figure 3(a), we show the corresponding dependency graph when we factor in the query popularities. The sequence in which nodes are accessed for the query corresponding to symptom S_1 are N_1, N_2 and N_3. The query is dependent on the first node N_1 with probability 1 since N_1 is the source node for the query. S_1 is dependent on N_2 only when there is no cache hit in the node N_1 which happens with probability $(1-p)$, where p is the probability calculated as above. Generalizing this further, the probability with which a query is dependent on the nth node along the path from the source node to the home node is $(1-p)^{n-1}$.

Using cache content information. This case addresses option (3) discussed in Section 3.1 where cache contents are fetched from individual nodes. Although the routing tables may indicate that a given query is dependent on all the nodes from the source node to the home node, the query may be satisfied much earlier due to a potential hit in the cache at an intermediate node. In that case, all the subsequent nodes should not be considered as part of the dependency graph. For the example in Figure 2(b), suppose that N_2's cache contents indicate the presence of the query corresponding to the symptom S_1. In this case, S_1 will not further depend on the node N_3. Thus, as shown in Figure 3(b), we remove the edge between S_1 and $L(2,3)$, and S_1 and N_3 from the dependency graph.

Hybrid dependency graph. We construct a hybrid dependency graph if we have access to both the popularity model as well as the cache contents at the end of a measurement interval. The hybrid dependency model uses the cache contents to prune out the links that are not required, and assigns probabilities for the other edges just as the dependency graph that only models query popularity would. In this paradigm, we can also model cases where we have cache contents from only a small subset of nodes. We show an example in Figure 3(c) where we have cache contents from N_5, and had a hit in the cache, which allowed pruning out S_2's dependencies with N_3, $L(3,4)$, N_4, and $L(5,4)$. For other nodes, as we can see in the figure, the probabilities are assigned just as we have in Figure 3(a).

4.2 Inference algorithm

Once the dependency graph has been created, dFault runs an inference algorithm on the graph to output a *hypothesis set* that contains the most likely set of failed nodes that can completely explain the set of observed symptoms. There are two standard approaches for inference that we can use in our setting. First, similar to Sherlock [2] or Shrink [10]), one could apply Bayesian inference on the dependency graph. The problem, however, is that it can take very long (more than 10 minutes) as discussed in Sherlock for large graphs such as what we have in our case. Second, we can model the problem as finding a minimum set cover in a bipartite graph, as systems such as SCORE [14] proposed in the past. Although SCORE is quite scalable, SCORE is designed to operate on a deterministic bipartite graph; our dependency graph, in contrast, is probabilistic in nature. Given the importance of scalability in our setting, we follow the minimum set cover framework of SCORE, while adapting it to our probabilistic setting.

We formulate the problem of identifying the most probable hypotheses that can explain the set of symptoms as finding a minimum *set cover*. Let us denote the set of symptoms dependent on a given node N_i, *i.e.*, the nodes that have an edge incident on N_i, as a set $\Psi_i = \{S_{i1}, S_{i2}, ...\}$, where S_i is an observed symptom. Then, the problem is to find the set $\{\Psi_{j1}, \Psi_{j2}, ...\}$ such that $\bigcup \Psi_{jk} = S$, where S is the set of all observed symptoms. Out of multiple hypotheses that can explain (cover) the set of all observed symptoms (set covers), the inference algorithm typically should favor smallest hypothesis. This is in accordance with the principle of Occam's razor that suggests that out of all explanations for a given observation, the simplest is most likely. Thus, the inference problem is therefore reduced to finding the minimum set cover of the dependency graph, that is known to be NP-hard [13] in general.

dFault uses greedy approximation to the minimum set-cover problem that finds a solution guaranteed to be an $O(\log n)$-approximation to the optimal. Of course, the size of the hypothesis set is not necessarily of great interest, as much as the overlap with the actual failures. Below, we show the pseudo-code for the inference algorithm.

procedure $infer\_hypothesis(R, S, W)$

1: $R \leftarrow$ root_causes, $S \leftarrow$ symptoms
2: Matrix $W[S][R]$ // stores the edge weights
3: unexplained $U = S$, hypothesis $H = \{\}$
4: **while** (U != empty) **do**
5: $\quad find\_score(U, R)$;
6: $\quad$ find R_i $s.t$ $\forall j$ $score(R_i) > score(R_j)$;
7: $\quad H = H \cup R_i$;
8: $\quad R = R - R_i$;
9: $\quad U = S - S_i$;
10: **end while**

The key idea in the inference algorithm is to iteratively find a candidate root cause that gets the highest score within each iteration, remove the set of symp-

toms that can be explained by this root cause, and repeat the process until all the symptoms are explained by some root cause or the other.

One key function within the inference algorithm here is the scoring function $find\_score(U, R)$. The scoring function varies depending on the particular type of the dependency graph input to the system in the form of the *edge weight* matrix $W[S][R]$. If all edge-weights are the same, such as when there is no cache model, then the scoring function is just the raw count of number of symptoms explained by each node (each element of $W[S][R]$ is 1).

In case of a global query popularity model, each of the elements in edge weight matrix $W[S][R]$ are the dependency probabilities computed between root causes and symptoms. In this case too, we use the same algorithm that we used for the case when all edge weights are the same. In our inference algorithm, we just add the raw probabilities to assign a score to each node. The intuition is that multiple low probability edges cannot increase the score of a node by so much that the score exceeds that of a few high probability edges. We show later in Section 5 that our simple heuristic is remarkably effective and accurate.

Post-processing the hypothesis. Given a DHT, there could be failures due to inherent churn in network due to nodes getting added, nodes dying and so on. But these are not of our interest as DHT reconverges. We refer to these as spurious failures. Because there is no correlated reason for the spurious failures, the number of candidates in the hypothesis set could be quite large. To address this problem, we perform post-processing of the hypothesis set generated by the inference algorithm to rank the elements and choose the ones that are most important. Our *ranking algorithm* sorts the scores of individual nodes in ascending order and computes a normalized score based on the highest score assigned to any node (*i.e.*, $s_i/Max\{s_i\}$), where s_i is the score of the ith root cause in hypothesis set). We consider only those candidates that exceed a given threshold τ. Choosing high value of τ will only output failures that contribute the most towards failed symptoms but may miss a few genuine failures (higher false negatives), while choosing a low value of τ will include several false positives in the hypothesis. We conduct experiments in Section 5 to identify the right balance between false positives and false negatives.

5 Evaluation

We use a combination of simulations and a real PlanetLab DHT deployment to evaluate the efficacy of dFault in localizing failures. The goals of our experiments are three-fold: First, we wish to compare and contrast the different variants of our system with different kinds of dependency graphs outlined in Section 4.1. Second, we intend to study the effect of *incomplete information* both in terms of the number of symptoms, routing information, and cache information available for diagnosis. For these two experiments, we use a prototype Pastry DHT implementation over PlanetLab. Given we cannot scale beyond few hundreds of nodes in PlanetLab, the third part focuses on simulation results with hundreds of thousands of nodes.

5.1 Prototype implementation

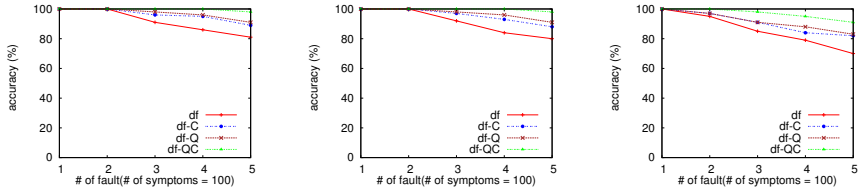
We use the FreePastry implementation from Rice University [23], further revamped by the Cornell Beehive project as the canonical DHT for our evaluations. There are two configurable parameters in Pastry, the base parameter and the leaf-set size, which we set to 16 and 32 respectively. On top of the Pastry DHT layer, we deploy a simple application wrapper that emulates the basic functionality such as distribution of objects across different nodes in the system, issuing and accepting queries for those objects from the hosts in the network, caching of responses received (we use LRU scheme for cache eviction), and so on. The objects in our system have unique 128-bit identifiers obtained through the SHA-1 hashing algorithm. The home node, the closest node in the identifier space, stores permanent copies of the objects for which it is responsible.

Our implementation also includes a query injection mechanism that allows us to query for objects following a Zipf distribution to simulate realistic caching effects (described before in more detail in Section 3.1). Each node issues a query, waits for the response, caches the response. In case the node does not get the response within a stipulated time interval (5 minutes), it flags the query to have failed and reports it as a symptom. Apart from failure reporting, each node is also responsible for reporting any churn in the DHT network in form of routing updates. We also implement a *suspend/resume* API, that allows us to control the behavior of each node directly and inject realistic faults into the system. Thus, if we wish to simulate a fault, we trigger the suspend routine which will essentially drop (or delay) the forwarding of all application queries, but continues to participate in the DHT itself, in effect simulating the realistic failure modes we have discussed in Section 2. We also include a mechanism to inject network delays between two DHT nodes by delaying selected queries within the sender side of the overlay link.

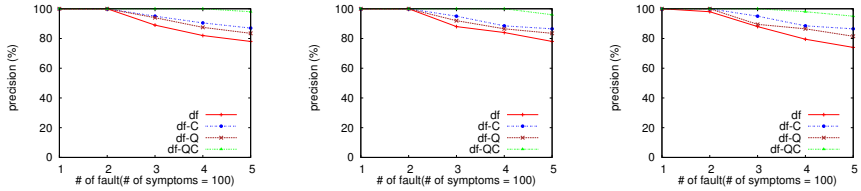
In addition to the Pastry DHT implementation, we implement a prototype version of our dFault system. Our current implementation polls individual DHT nodes for any failed query messages every 'diagnosis interval', collects routing and cache contents information from (some or all nodes), builds the appropriate dependency graph and generates the most likely set of failed nodes.

In section 3.1, we discussed about different alternatives which can be used to model the effect of caching in the DHT network. Depending on which model we choose to use, we can have possibly 4 different adaptations of dFault: (1) dFault indicates the version that employs no cache model. (2) dFault-**C** includes a cache snapshot. (3) dFault-**Q** uses query popularity model. (4) dFault-**QC** uses both query popularity model and the cache snapshot. We present a comparison of the performance of all these alternatives in this section.

Metrics for evaluation. We measure the goodness of our system using two basic metrics, *accuracy* and *precision*, as used before in the literature [14, 15]. Accuracy is defined as the percentage of real faults (ground truth) captured in the hypothesis. Define H to be the hypothesis set, and G as the real ground truth, and $|.|$ denoting the usual cardinality of a set, then, accuracy is computed as $|H \cap G|/|G|$ and precision as $|H \cap G|/|H|$. In our evaluation, we consider



(a) Node failure, Accuracy (b) Node delay, Accuracy (c) Link delay, Accuracy



(d) Node failure, Precision (e) Node delay, Precision (f) Link delay, Precision

Fig. 4. Comparison of accuracy and precision for inference algorithms df (short for dFault), df-Q, df-C, df-QC for different number of simultaneous failures for node failure, node delay and link delay types of failures.

accuracy and precision above 80% as a good diagnosis. This effectively means that 4 out of 5 faults have been accurately diagnosed and that, 4 out of 5 faults in the hypothesis set are real failures.

5.2 PlanetLab experimental methodology

For our experiments, we have deployed our DHT prototype system on 110 nodes in PlanetLab. We use a total of 100, 000 unique objects for the DHT. We choose a relatively small cache size of 50 objects at each server, to ensure that caching alleviates, but does not completely eliminate routing in the DHT network. Queries are injected using a Zipfian distribution with parameter value of 0.9.

We inject faults using the suspend/resume API mentioned before in Section 5.1 for our Pastry implementation. To simulate a scenario, we suspend a given node (or a set of nodes to inject multiple simultaneous failures) until a total of 5,000 queries are injected into the DHT at all the nodes put together. For all our experiments, we choose the averages of 20 different random runs in order to smooth out the results. This ensures, that about 50 queries on an average are issued per node in any given diagnosis interval. Note that while diagnosis interval is typically in units of time, there is one-to-one relation between number of queries issued in the system and diagnosis interval time for a given rate at query injection. Since our query injection rate is somewhat arbitrary, we focus on number of queries instead of absolute time. On average, injecting 5,000 queries into our system takes about 4-5 minutes of time (with mean inter-query interval of about 50ms) . We use a threshold value of 0.5 (τ discussed in Section 4.2) that we found to represent a good trade-off between accuracy and precision.

We simulate three types of faults—node failure, node delay and link delay. For node failure, we use the suspend/resume API to silently drop DHT queries at the node (without affecting DHT layer connectivity itself). Failure symptoms at individual nodes are collected at individual nodes by using a timeout of 1 second (any query not responded to within 1 second is deemed a failure symptom). Node delays simulate adding 1 second latency for each and every query that passes through a given DHT node (at which we inject the failure). For these failures, we use 1 second as a timeout for failure symptoms. For link delays, we add an addition 300 millisecond latency at the sender-side of the failed link. We use 500 millisecond latency as a timeout for nodes to report as a failure. These numbers are somewhat arbitrary and chosen based on our setting; an operator can easily choose which ever timeout values to report the failure symptoms based on their setting.

5.3 Comparing different dFault variants

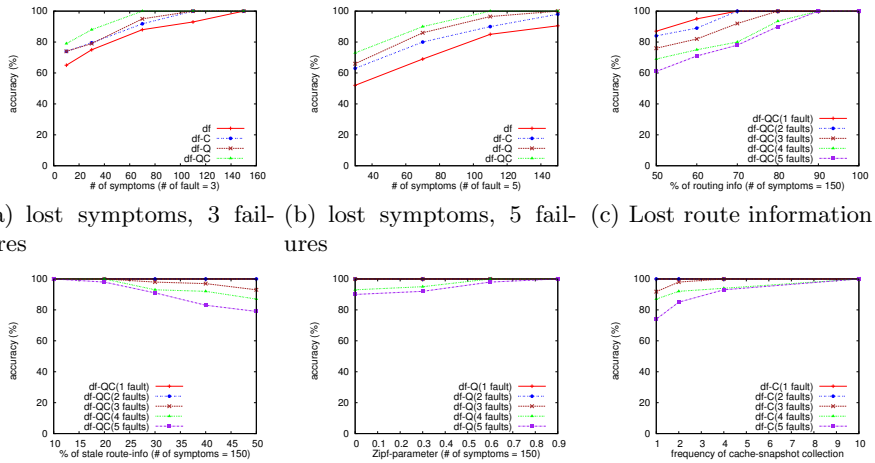
We compare the different variants of dFault across different failure scenarios (by varying the number of simultaneous injected failures). For this experiment, we only select (randomly) a total of 100 symptoms out of potentially many more. Figure 4 compares the accuracy and precision across these different variants for different types of failures injected (node fault, node delay and link delay). The base variant of dFault assumes no cache information or query popularity, and yet, as we can see from the figure, in all types of failures, it achieves almost 100% accuracy and precision for up to 2 faults. However, as the number of faults increases beyond 2, both accuracy and precision reduces to almost 75% at 5 faults (which is still reasonable since 3 out of 4 faults are localized accurately).

The effects of bringing in caching information and query popularity can be observed in df-C and df-Q variants. Somewhat curiously, both these techniques have similar accuracy and precision numbers, although one of them employs a probabilistic dependency model, while the other uses cache information to refine dependencies obtained from the routing table information alone. The query popularity modeling appears to slightly out-perform even obtaining cache contents, although, we believe this is in part because of the reduction in precision (which naturally lifts up the accuracy as more nodes are added), and hence, not significant. Note that, although the significance of these two types of information is comparable, it does not mean both are exactly doing the same thing. This is exemplified by the fact that, when we combine both caching and query popularities, we obtain the "best of both worlds" marked by a dramatic increase in accuracy and precision—almost 100% up to 5 failures in all types of faults.

5.4 Reducing the volume of symptoms

We study the effects of lost or delayed symptoms by *randomly sampling* a candidate set of symptoms from the actual set of observed symptoms and feed them into our system for diagnosis.

Figures 5(a) and 5(b) depicts the accuracy of dFault (for the different variants) as a function of the number of symptoms available for fault localization



(a) lost symptoms, 3 failures (b) lost symptoms, 5 failures (c) Lost route information

(d) Stale route information (e) Object popularity (f) Cache update.

Fig. 5. Robustness to lost symptoms, lost or stale routing tables, and cache effects.

for 3 and 5 faults; we assume full routing information and cache snapshots at the end of every diagnosis interval. We do not show the precision curves since they follow very similar trends to accuracy. Two observations can be made from the figure. First, we observe that the number of symptoms required for achieving 100% accuracy in fault diagnosis is dependent on the number of faults in the system. For a single failure scenario (not shown in Figure), even as low as 20 symptoms is sufficient to achieve 100% accuracy and precision for all algorithms. As the number of faults increases, it requires increasingly more number of symptoms for the same accuracy and precision values. Second, the difference between the various versions of dFault becomes more apparent with more number of failures, with dFault-QC consistently out-performing both the C and Q variants. Note however that, dFault-QC suffers from low accuracy and precision just like others when the number of symptoms is less than 20 since there is not enough evidence collected to clearly identify the root causes. This is fundamental as collecting the cache contents and modeling the query popularities accurately cannot compensate for the lack of sufficient evidence.

5.5 Partial and stale routing information

One of the key ingredients in building the dependency graph is the routing information which CDS collects from the nodes in the DHT network. Once the base routing tables are collected, dFault needs to know only the routing updates representing any churn in the network. As with symptoms, routing updates could be lost, delayed and sometimes be incorrect or stale. To evaluate the efficacy of our system under such conditions, we perform two experiments, one with partial routing information, and the other with stale information.

Partial routing information. Figure 5(c) depicts the performance in cases when it has access to routing tables from only a fraction of nodes in the DHT. Here, we fix the number of available symptoms to 150, which is larger than strictly required. From the figure, we can observe that the dFault does not lose any accuracy even for the five-failure scenario without the availability of routing information from up to 10% of the nodes. The system is more robust to non-availability of routing information when the number of faults is less— for 1 and 2 fault scenarios, accuracy is 100% even without the routing information for about 30% of nodes. This, of course, is expected since the chance that the routing information necessary for diagnosing the fault is missing is smaller for less number of failures. One thing we observe from our experiments was that the loss of routing information can be somewhat compensated by the number of extra symptoms dFault gets to use in diagnosing compared to previous results in Figure 4. Precision graphs show a similar trend and hence omitted for brevity.

Figure 5(d) depicts the performance when a set of nodes have not been able to report the routing updates to the CDS. We created this scenario by starting the system with 110 PlanetLab nodes, then killing 10 nodes so that the eventual system has only 100 nodes. We collected our failure symptoms from the 100 nodes, but used the routing information from when there were 110 nodes, which means, some of the routing table entries will be stale and inaccurate. We varied the fraction of nodes that used old routing tables from the 110 node scenario on the x-axis in Figure 5(d). We can observe that, while the accuracy of the system goes down as the amount of stale information increases, the accuracy is higher than that of lost routing information. In essence, we observe that loss of routing information affects the performance to a much greater extent compared to the staleness of routing information, *i.e.*, out-dated information, that may be false, is still better than no information. This is because, even though there may be erroneous entries in the routing tables that will confuse the inference algorithm, they are not going to be a whole lot. If there is no information, however, there is no way to create a dependency graph that makes inference harder.

5.6 Impact of caching

We study the impact of cache model on our system by varying the Zipfian parameter and the frequency of cache snapshot collection.

Variation with Zipf parameter. A lower value of Zipfian parameter results in a more uniform query distribution; indeed a parameter of 0 is exactly the uniform distribution model. It results in more frequent changes to overall cache content. For this experiment, we used dFault-Q since we are mainly interested in modeling the effects of the popularity distribution while isolating the effects of cache, which we consider next. As expected, we can see from Figure 5(e) that the accuracy of dFault-Q in case of uniform query distribution (Zipf parameter $= 0$) is somewhat lesser than for higher values of Zipf parameter. For a smaller number of failures (< 4), the Zipf-parameter has no direct effect on the accuracy.

Effect of more cache information. As discussed in Section 3.1, given a finite size of the cache at any given node, the cache contents are likely to undergo

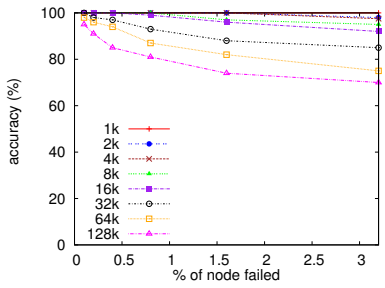


Fig. 6. Accuracy of dFault with size of network.

frequent changes. Since we collect cache contents only at the end of a diagnosis interval, the snapshot of cache collected may not be the true representation of the state of cache viewed by a given failed query. A way to reduce this effect to some extent is to access more number of cache snapshots per diagnosis interval. While this also increases communication bandwidth, we wish to determine the tradeoff in this experiment Indeed, in Figure 5(f), we can clearly observe the increased accuracy as a result of multiple cache updates per diagnosis interval. We experimented with different time interval of collecting cache snapshots. In our setup, dFault-C delivered 100% accuracy when the snapshots were collected every 10 times per diagnosis interval. Although, it may appear that this truly involves sending 10 copies of the cache, we can encode only the differences when transmitting to the CDS which will drastically reduce the communication bandwidth required.

5.7 Scalability experiments

Our PlanetLab implementation results are all based on around a 100 node deployment; in order to evaluate how dFault performs when deployed on large-scale DHTs consisting of thousands of nodes, we build a custom simulator that abstracts several key features of the DHTs and conduct our experiments in this simulator. In our framework, we simulate a simple $O(\log N)$ routing protocol, with N being the the total number of nodes. We start with $600,000$ objects which are distributed among all the nodes, *i.e.*, each object is assigned to a unique *home node*. We also simulate caching with LRU as the cache eviction policy. Every node has a cache of size 5% of the number of objects each node in the DHT is responsible for, i.e. it is a home node for. Similar to the PlanetLab experiments, we set the Zipfian parameter to 0.9. Just as before, we simulate faults by determining how many and which nodes to fail randomly. After a fixed interval of time, we collect these failed queries (symptoms) and perform failure analysis using the same technique as described above. So, our simulator simulates the typically behavior of DHTs, with hooks for fault injection.

Figure 6 shows the performance of dFault (the QC variant with full routing and cache snapshot information) as we inject a variable number of failures into the DHT. The number of failures in the Figure is not absolute, but a percentage

of total number of nodes. We varied the percentage from 0.1% to all the way 3.2% in multiples of 2. Thus, in the worst case scenario, in a network of 128,000 nodes, about 4,096 would have failed, which is quite a large number of failures. As can be observed from Figure 6, dFault can identify with almost $70-90\%$ accuracy. For a network experiencing less than 1% node failures, dFault performs with more than 80% accuracy even for a system with $128K$ nodes.

Bandwidth requirements. We now discuss briefly the bandwidth requirements for transfer of information from nodes to the CDS. Assuming nodes and queries are represented by 128-bit hash values, we can calculate the amount of bandwidth required for data communication for smooth functioning of dFault. The maximum number of entries each Pastry node has to maintain is given by the formula $(2^b - 1) \times \log_{2^b} N + l$. In our evaluation, we have used $b = 4$ and $l = 32$, which translates to roughly 100 entries per node at the most; assuming $20bytes$ per entry, routing tables are not more than $2KBytes$ in size. A cache size of about $2,000 \times 20Bytes = 40KBytes$ (assuming a relatively large cache of 2,000 elements) also needs to be transferred for modeling the cache effects. Including another $100 \times 20 = 2Kbytes$ for the network symptoms (assuming 100 symptoms per node), a total of about $44KBytes$ needs to be transfered from every node in the DHT. If we assume that the diagnosis time interval is 5 minutes, then a bandwidth of $44KBytes/300s = 1.2Kbps$ is sufficient per node. Even for a system with 100,000 nodes, this translates to a bandwidth of less than 100 Mbps aggregate at the CDS.

6 Related work

Fault isolation and diagnosis is a well-researched topic with several systems proposed for fault localization over the past few years. However, there have been very few systems for localizing application-layer faults in DHTs. To the best of our knowledge, dFault is the first attempt to systematically develop a fault localization framework for DHT applications.

Most practically deployed fault localization systems are in the context of enterprise services, where these tools provide invaluable service to IT administrators to pin-point the location of faults. For example, EMC's SMARTS [24], IBM Tivoli [1], HP OpenView [8], to name a few, provide general management support by pulling together alarms from several components and correlating them to understand where the failures occur. Unfortunately, while they provide general correlation support, these systems operate with SNMP-based [4] measurements, and hence do not localize application layer faults. SMARTS uses a codebook based approach to diagnose faults for some well known applications. The dependency for these applications is known a priori, and the codebook is built by application experts.

Our work is closest to systems that exploit dependency graphs for fault isolation [14, 10, 2, 11] involve the creation of a dependency graph that represents the complex chain of dependencies that exists in services, and a localization algorithm that amasses the set of symptoms from the network and outputs a hypothesis that contains the candidate set of root causes. While our system dFault also follows a similar general approach, there are significant differences as well.

Sherlock [2] provides new abstractions such as multi-tier dependency models and uses Bayesian inference for fault diagnosis in enterprise networks. Sherlock is not directly applicable in our framework, because dependencies in our system are not as complex; simple bi-partite modeling suffices. In addition, the inherently large scale of p2p networks makes scalability an important requirement, while the use of Bayesian inference in Sherlock increases the computation time for large p2p networks.

Bi-partite graphs have also been used before for modeling dependencies between root causes and observed symptoms. SCORE [14], for instance, uses bi-partite graphs for modeling 'shared-risks' and diagnosing IP link layer faults and optical components. Similarly, [15] uses bi-partite graph modeling to diagnose MPLS black holes in backbone networks. While dFault shares a similar setup, the presence of caching effects and dynamic routing information from the underlying p2p layer makes our modeling *probabilistic* in nature, and thus, different compared to their deterministic model.

7 Conclusions

Although DHT-based systems began mainly as research prototypes, they have quickly found mainstream adoption. As they make headway into more commercial applications, their robustness requirements increase tremendously. Unfortunately, there exist several latent failure modes in DHTs that can affect application performance significantly. In this paper, we present dFault for accurate localization of these latent failures. dFault collects a sample of failure symptoms from the individual DHT nodes and formulates a bi-partite dependency graph with the help of routing tables and caching information collected from nodes. It runs an inference algorithm over this graph to output a candidate set of faults. In our evaluation of dFault with a real deployment over PlanetLab as well as using simulations, we found dFault to produce accurate and precise hypothesis for multiple failures, even in the presence of erroneous and incomplete fault and dependency data.

References

1. `http://www.tivoli.com`.
2. P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM*, Aug. 2007.
3. L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of IEEE INFOCOM Conference*, New York, NY, Mar. 1999.
4. J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). RFC 1157, IETF, May 1990.
5. M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.
6. M. J. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-transitive connectivity and dhts. In *WORLDS*, San Francisco, CA, Dec. 2005.

7. D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, Stevenson, WA, Oct. 2007.
8. HP Technologies, Open View. `http://www.openview.hp.com`.
9. J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and Effectiveness of Caching. In *Proc. of SIGCOMM Internet Measurement Workshop (IMW)*, San Francisco, CA, Nov. 2001.
10. S. Kandula, D. Katabi, and J. P. Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *Proc. ACM SIGCOMM MineNet Workshop*, Aug. 2005.
11. S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and V. Bahl. Detailed diagnosis in computer networks. In *ACM SIGCOMM*, Aug. 2009.
12. D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigraphy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proc. of the ACM Symposium on Theory of Computing (STOC)*, El Paso, TX, Apr. 1997.
13. R. M. Karp. Reducibility among combinatorial problems. *R. E. Miller and J. W. Thatcher (editors): Complexity of Computer Computations*, pages 85–103.
14. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *NSDI*, May 2005.
15. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and localization of network black holes. In *IEEE Infocom*, May 2007.
16. Z. Li, A. Goyal, Y. Chen, and A. Kuzmanovic. P2PDoctor: Measurement and diagnosis of misconfigured peer-to-peer traffic. Technical Report NWU-EECS-07-06, North Western University, 2007.
17. P. Maymounkov and D. Mazires. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of the International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, Mar. 2002.
18. V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *NSDI*, San Francisco, CA, Mar. 2004.
19. V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *SIGCOMM*, Portland, OR, Aug. 2004.
20. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *ACM SIGCOMM*, San Diego, CA, Aug. 2001.
21. S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of opendht on planetlab. In *WORLDS*, San Francisco, CA, Dec. 2005.
22. S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: a public DHT service and its uses. In *Proc. of the ACM SIGCOMM Conference*, Philadelphia, PA, Aug. 2005.
23. A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *MIDDLEWARE*, Heidelberg, Germany, Nov. 2001.
24. SMARTS Inc. `http://www.smarts.com`.
25. I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM*, San Diego, CA, Aug. 2001.
26. Project Voldemort: A distributed database. `http://project-voldemort.com`.