Query Routing in the TerraDir Distributed Directory *

Bujor Silaghi Bobby Bhattacharjee Pete Keleher

Department of Computer Science
University of Maryland, College Park, MD 20742, USA
{*bujor, bobby, keleher*}*@cs.umd.edu*

ABSTRACT

We present the design and evaluation of the query-routing protocol of the TerraDir distributed directory. TerraDir is a wide-area distributed directory designed for hierarchical namespaces, and provides a lookup service for mapping keys to objects. We introduce distributed lookup and caching algorithms that leverage the underlying data hierarchy. Our algorithms provide efficient lookups while avoiding the load imbalances often associated with hierarchical systems. The TerraDir load balancing scheme also incorporates a node replication algorithm that provides configurable failure resilience with provably low overheads.

Keywords: Peer-to-peer systems, distributed directories, query routing, hierarchical namespaces

1. INTRODUCTION

Consider a distributed application in which large numbers of resources are exported by principals located in different parts of the global Internet. The application is composed of coordinated views of these different resources. Examples of such applications include Internet-based encyclopedias,[1,2] distributed auctions conducted over the network without involving a central authority; ubiquitous file systems that allow distributed access to data regardless of the user's point of attachment to the network; distributed resource-sharing applications which allow processing and other resources† to be shared or coordinated over a wide-area network; and distributed publish-subscribe systems where resources are advertised and accessed over the network. These examples represent emergent, multi-party peer-to-peer applications that will constitute the next generation of Internet applications. We have developed a set of distributed protocols called TerraDir, which can be used to build a broad range of wide-area resource discovery applications, including all of the applications described above. The TerraDir protocols implement a distributed directory, in which resources are advertised and accessed using cooperating nodes distributed throughout the network.

TerraDir is specifically designed for data that can be arranged into a rooted hierarchy. The entire suite of TerraDir protocols consist of a number of sub-protocols for implementing customizable security, consistency, and availability. In this paper, we focus on the caching and replication algorithms in TerraDir. TerraDir is designed to handle millions of directory nodes distributed over thousands of peer servers located across the global Internet. We rely heavily on caching in order to provide high performance over such a distributed system. The TerraDir caching algorithms specifically leverage the hierarchical structure of the underlying data and can substantially lower access latencies even with very small caches. Since TerraDir is a protocol designed to run over the end systems in the Internet, it must be particularly resilient towards node failures. To this end, we describe a low overhead replication algorithm that provides configurable-levels of fault-tolerance. Further, the TerraDir fault-tolerance algorithm is an integral part of the caching scheme, and is also used to balance query processing load on different levels of the directory tree.

A number of interesting new algorithms[3–7] for searching in peer-to-peer systems have recently been developed. Unlike the schemes presented here, these algorithms do not require a hierarchical structure in the underlying data. Some of these schemes are based on flooding[8] and thus do not scale due to their high resource

*This work is supported by a grant from the National Science Foundation, ANI0123765.

†Napster, Gnutella, and Morpheus are examples of applications in this category; the resources currently being shared are compressed audio files.

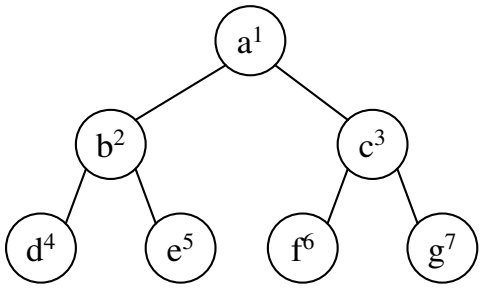

Figure 1. An example TerraDir. A node's superscript represents the server that owns the node.

usage. A particularly elegant algorithm is described in[4]: it is based upon the idea of consistent hashing,[9] and provides both excellent load balancing and bounded worst case access latencies. The CAN scheme[7] also provides bounded latencies, and by using different number of CAN dimensions, is able to trade-off latency versus the amount of state at a server.

This paper is organized as follows: in Section 2, we present an overview of the TerraDir directory protocols and discuss our caching and replication algorithms. Section 3 describes the process of routing a query in Terradir, and Section 4 evaluates the performance of our routing protocol. We compare the TerraDir protocols to other distributed search algorithms in Section 5, and conclude in Section 6.

2. TERRADIR OVERVIEW

The TerraDir protocol is used to construct and maintain a distributed directory, and can be used to efficiently query the directory for name-based searches. TerraDir consists of a number of component protocols that are run across a set of peer *servers*. Use of the term "server" does not imply the existence of clients. Instead, a server is just a machine that is running a copy of the TerraDir code and participating in the same shared directory. The TerraDir protocols can be configured for application-specified levels of security, failure tolerance, and consistency. In this section, we provide a brief overview of the various TerraDir protocols while a detailed account of all of the protocols is given in the extended version of the paper.[10]

2.1. The TerraDir Namespace

The primary abstraction exported by TerraDir is a hierarchical namespace. TerraDir applications use the namespace to store meta-information about a potentially very large number of objects. These objects can range from network resources, to objects being offered for an on-line auction, to files in a distributed filesystem.

Node names in TerraDir are constructed much like DNS names or Unix file system names. The directory structure is that of a rooted graph, i.e. it is possible to reach a single node with two different names. [‡] We show a simple example TerraDir namespace in Figure 1.

Each node has a single owner, and may be replicated at servers other than the owner. The information about a node may also be cached by other servers. Node owners permanently maintain some state on behalf of each owned node: this state consists of of a label, a set of incoming edges, a set of outgoing edges, a set of attributes, a record, and some bookkeeping information. The attributes are a set of *(type, value)* pairs that contain meta-data about the node and can be used in user-initiated searches of the directory. The record consists of the actual data that this node represents, and the bookkeeping information is used in failure recovery. Each server in a TerraDir may own an arbitrary set of nodes, subject to the restriction that any node has a single owner. The permanent state at a server consists of the state for the owned nodes, and also the state for nodes "replicated" at this server. If a server replicates a node, it permanently maintains the meta-data for the node; it could also permanently hold the data for the node, but this is an application-specific choice.

[‡]However, each node does have a single canonical name that is used within the TerraDir protocols. If we only consider canonical names, then the namespace is a rooted tree.

2.2. Caching

Each TerraDir server contains a small LRU cache for recently accessed nodes. A cache entry contains information about the node, its parent and children, and its owning server.

TerraDir differs from a straightforward cache in that the entire path "so far" is cached at each step along the query path; culminating in the entire query path being cached at the source when the query completes. Caching entire paths in this way not only brings "far" nodes into the cache (the source caches the destination, and vice versa), but it also brings in nodes from different levels of the tree (a typical query goes "up", and then "down"), and nearby nodes. This mixture of close and far nodes performs significantly better than simply caching the query endpoints.

2.3. Replication

TerraDir replicates nodes in order to obtain high availability and to balance load, but replication is also effective in reducing query latencies. In TerraDir, nodes can fail deliberately (e.g. the human owner of a server decides to cease participation in a directory), or through machine failures. We assume a fail-stop model[11] in this paper.

A server failure may cause a set of directory nodes to become ownerless, and to cease responding to queries. In the absence of failure recovery mechanisms, nodes whose owners fail are lost permanently, causing irreparable damage to the integrity of the directory. Even if the data for an ownerless node can be recovered, possibly by locating a cached copy, the node can neither be modified nor deleted. Thus, the system must have a process for forcibly reassigning ownership of ownerless nodes to a different server.

It is possible for a node to fail for a transient period, for instance while it is rebooted. If this failure is not detected by other nodes and the node can reconstruct all of this state, then this is not considered a failure. However, if the failure is detected (possibly during a query or an update that required participation from the failed node), then the node may not assume ownership of any of its nodes even if it can reconstruct all of its state.

TerraDir replicates each node at randomly selected servers. We wish to make total replication overhead constant per server, but distribute the replicas such that nodes with higher load (those higher in the tree) receive more replicas.

Let N be the number of nodes and i be a node's level, s.t. $i = 1$ are the leaf nodes, $i = 2$ is the next level up, etc. We give each node at level i a total of i replicas. It is possible to show that for a balanced tree, the total number of replicas created in this case is bounded by $2N$. Thus, if these replicas are distributed uniformly amongst all servers, then each server maintains a constant number of replicas. We show how this property is derived for a balanced full binary tree next.[§]

For a full binary tree, there are $\frac{N}{2^i}$ nodes at each level i. Let n_i be the total number of replicas for the nodes at level i, and let R be the total number of replicas. Then

$$n_i = i\frac{N}{2^i}, \quad \text{and} \quad R = \sum_{i=1}^{\log N} n_i$$

As N increases, R converges to $2N$. Thus, each node has a constant number of replicas, but the actual numbers vary from 1 at the leaves to $\log N$ at the root.

In TerraDir, we implement a slight generalization of this scheme: the original scheme creates i replicas for a node at height i. We introduce a replication factor k, and create ki replicas for each node at height i. Thus, $k = 0$ implies no replication, $k = 1$ means the root would have $\log N$ replicas[¶], and the lowest leaves would have one replica. For an arbitrary k, there are $k \log N$ replicas of the root, and k replicas of the leaves, and on average, each node replicates $2k$ other nodes.

[§]Note that this derivation can be applied to any balanced tree.

[¶]Clearly, the root would have $\log N$ replicas only if the tree is balanced.

The network addresses of servers that replicate node a become part of node a's parent's state. Thus, instead of containing just a label and a network address, the per-child state in the parent node contains a label, a network address for the child, and network addresses for each of the n replica servers. A node replicates copies of its state to each replica at application-specified intervals [‖]. The replication factor k is application-specified and can potentially be different for different nodes in the directory.

Node failures are detected via timeouts either by querying servers, or parent servers that attempt to forward messages to the failed node. The server detecting the failure immediately assumes responsibility for reconstructing the failed node either locally or on a randomly selected server.

When a server failure is detected, the canonical parent is queried for the address of a replica. In the event that the parent has failed as well, its parent is first queried. For example, assume we have a graph that contains the path "/animals/puppies/bert", and that " puppies" and "bert" fail. The directory is reconstructed by querying "/animals" for the address of one of the replicas for the "/animals/puppies" node. This information in the replica is used to reconstruct "/animals/puppies", which contains the address for the replicas for "/animals/puppies/bert" node. One of these replicas is then used to restore "/animals/puppies/bert".

2.3.1. Fault Tolerance

Node replication also provides fault tolerance in a TerraDir. No other special purpose protocols are needed. A node is unavailable if all of its replicas have failed. Of course, the routing algorithm still has to find a path to a surviving replica. We describe how this is done in Section 3.

2.4. Other Sub-protocols

TerraDir includes a number of protocols that are not covered in detail here because of space constraints. We briefly describe the following protocols.

Namespace aggregation: Applications can create and manipulate multiple namespaces. These namespaces can be arbitrarily composed and overlayed to make higher level structures comprising data from a variety of sources and organized in different ways. As an example of when these *aggregate namespaces* are needed, assume user A wishes to annotate data exported by another user, B, by adding new annotation nodes as children of nodes create by B. If B has not, or will not, assigned permissions that allow A to alter B's nodes (adding children alters a node), then A could create a new namespace, derived from B's, that includes the new relationships. The new namespace would consist only of the new relationships and a pointer to B's namespace.

Searching: This paper evaluates the performance of TerraDir as a lookup service, similar in motivation to a variety of other recent projects.[4, 7, 12, 13] However, TerraDir was also designed to support searches efficiently. Lookups are essentially implemented as degenerate searches. They differ in that the target of a lookup is a fully-qualified pathname (from the root down to an interior node or a leaf), and the target of a search may include wild-cards. A wild-card potentially causes a query to be replicated over a set of siblings at some point in the graph; results returned by sub-queries are re-integrated and returned to the source as a single result.

View materialization: Searches are efficient if namespace and search structures are similar. For example, a search injected at the root for "/vehicles/cars/red/*" only affects the nodes in the subtree under "/vehicles/cars/red"; nodes under "/vehicles/planes" are not searched. However, a search for "/vehicles/*/red/*" would cause more of the tree to be searched. Such mismatches can be addressed by injected *view queries* into the tree. View queries cause new "views" of the tree to be lazily materialized. New searches that match a view query in form are handled efficiently when injected at the base of the new materialization.

Security: TerraDir provides secure access to all data. The centerpiece of the security sub-protocol is the ability to give out credentials from which less powerful credentials can be derived locally (i.e. without requiring access to a trusted mediator).

[‖] This period of vulnerability can be eliminated by synchronously updating replicas whenever the node state is modified, at the cost of additional overhead and network communication.

3. ROUTING IN A TERRADIR

The rest of this paper describes the design and evaluation of protocols that efficiently locate nodes in a TerraDir namespace. Since the canonical node names form a tree, there is a well-defined path, along this tree, from any server to the owner of any node. In the absence of failures, this path upper bounds the latency for servicing TerraDir queries, and is always bounded by twice the height of the tree. However, a query resolution algorithm that always routes queries along the canonical name tree suffers from two major problems:

- Any one failure partitions the directory such that nodes can no longer be located if they are in different components of the partition
- The load on the nodes "higher" in the namespace is disproportionately large

The main contribution of this paper is a set of protocols that greatly reduces the average latency for query resolution, while adding robustness and balancing the load across the entire directory. Our scheme is based upon a tree-walk, but incorporates both caching and replication.

Consider the TerraDir Figure 1. Routing in a tree is performed by going "up" the tree until the longest common prefix of the source and destination is reached, and then going "down" until the destination. For example, to route from "/a/b/d" to "/a/b/e", we go up until "/a/b", and then down to "/a/b/e". To go from "/a/b/d" to "/a/c/f", we go up to "/a" and then down.

Recall that node in TerraDir can also be "replicated", i.e. a copy of the node can be stored at more than one server. Note that each node still has only one "owner", and a configurable number of copies of the node is permanently stored at other servers. We assume that the owner of the node maintains an up-to-date list of the replicas, and handles consistency issues with updates to a node.

Replication decreases latencies by allowing a shortcut between all the nodes hosted by a server. The superscripts in Figure 1 represent the ID's of the servers that own the nodes. Let us assume that server 2 replicates node "/a/c". When routing from "/a/b/d" to "/a/c/f", the query ascends one level to server 2, owner of node "/a/b". Since server 2 has all of the internal state of node "/a/c", it knows the address of "/a/c"s child, "/a/c/f", and can route the query directly there. Hence, replication improves the hop count from four to two in this instance.

Caching, of course, further improves efficiency. The utility of cached nodes can be improved even more by including the identities of a server's *hosted nodes* in the cache entry of a node owned by that server. (Hosted nodes at a server are the set of nodes that a server owns or replicates). Knowledge of nodes replicated by a server potentially give the routing algorithm shortcuts to other portions of the TerraDir.

A straightforward enumeration of all permanent nodes in a cache entry's state might prove expensive (even though this set averages only 5 nodes for $k = 2$). Instead, we add a *digest* of the list of permanent nodes to the state kept for a server. Thus, whenever a node is cached, the digest of the permanent nodes held by the server is also available. We currently implement the digest using a Bloom filter.[14]

Our digests do not allow the set they represent to be enumerated; instead we must first generate a set of candidate nodes and check if they are in the digest. We generate the candidate nodes by enumerating the prefixes of all known nodes at a server. This includes the nodes it owns, the nodes it replicates, and the nodes in its cache. Given these replicas, prefixes and digests, we are ready to describe the TerraDir forwarding algorithm. Assume that a server s is forwarding a query towards target t. The server s proceeds as follows:

- Server s generates a list of prefixes P using the node names it knows. This set P includes the target t, and prefixes from nodes owned for s, replicated by s, and cached at s. Note that the set of prefixes for a node includes the node itself as well.
- For each prefix $p \in P$, s computes the distance (on the namespace tree) between p and the target t, and creates a list of prefixes sorted by distance from t. Note that the target t is part of this set of "candidate" prefixes, and is at distance zero from the destination.

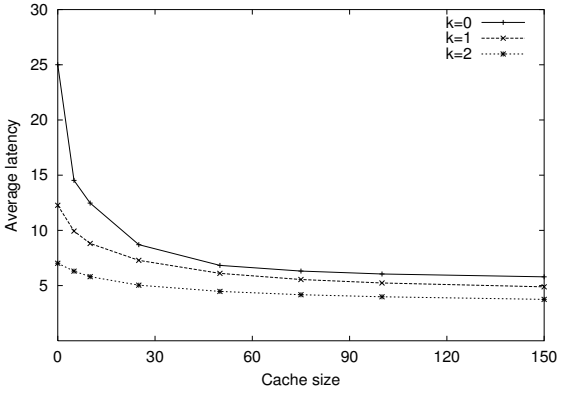

Figure 2. Query latency as a function of cache size and replication factor (k).

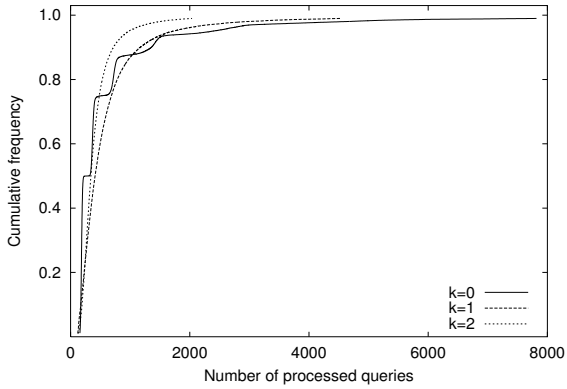

Figure 3. CDF of served load with cache sizes of 25.

- The server iterates through this candidate list in order of increasing distance from the target. For each prefix p, s checks to see if it knows which server owns (or replicates) p. Note that s uses the digests to determine if a server replicates a prefix p. This process starts with checking the target t, then all prefixes at a distance 1 from t, followed by prefixes at distance 2, and so on. The check is terminated when a server–prefix match is such that the new server (say s' has a smaller distance to t than s).

  We assert that the procedure we have described above always terminates, and finds the "best" server, in the sense that the procedure always finds the closest server to t that s knows about. Since the servers are checked in order of increasing distance from t, the check (if it terminates) always finds the "best" server. The check terminates since server s's parent and children are always known to s and their node labels are in the prefix set. If the target is not at server s, then one of these nodes (parent or a children of s) must be closer to t, and this node is found.

- If a node $s' \neq s$ is found, the query is forwarded to s', and the procedure is repeated.

In our description above, we have implicitly assumed that no servers have failed. However, in reality, server failures have to be handled as well. We use the following scheme to handle failures:

- If the server containing the best prefix has failed, we choose a server with the next best prefix. This process continues till no server can be found with a better prefix match than the current node s.
- If no better prefix–server match can be found, then the query is routed to a replica of the current node that is still available and has not been visited yet.
- If such a replica cannot be found, the query is routed to a replica of the root server that has not been visited yet. In case, such a replica cannot be found, the query fails.

The pseudo-code for the algorithm, including the server failure cases, is given in the extended version of this paper.[10]

4. PERFORMANCE RESULTS

This section presents simulation results of the TerraDir routing protocol. Our simulator is written in C, and contains 3000 lines of code. Unless otherwise specified, all experiments assume a cache size of 25, three million queries and 32,767 servers, each exporting a single node.

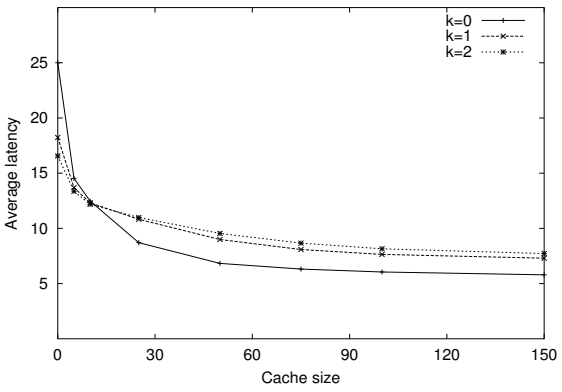

Figure 4. Query latency w/o digests.

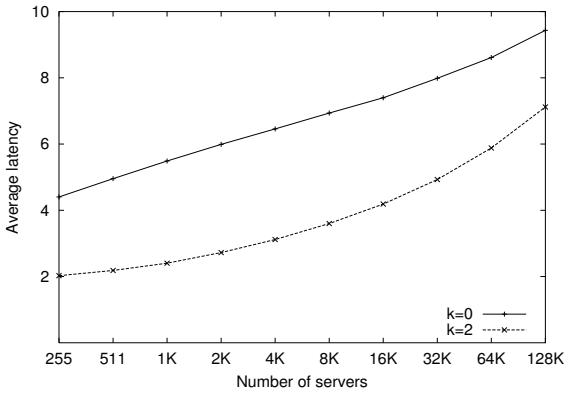

Figure 5. Query latency versus size of network.

Figure 2 shows latency versus cache size for our base system, for the non-replicated system and two different levels of replication. The queries were distributed uniformly at random among all nodes. There are two interesting points in these plots:

- Caching is extremely effective in TerraDir, even without any query locality. Further, very small caches are enough to provide most of the benefit of caching.
- Even a small replication factor (e.g. $k = 2$) provides enough extra "edges" in the namespace to reduce latencies by a factor of 2–4.

Figure 3 shows 0–99% cumulative distribution (CDF) of the load at each cache size and for different k values (k is the replication constant discussed in Section 2.3). At each point on the x axis, the height of a line represents the fraction of servers that served less than or equal to the number of of queries represent by the points x value. The most important point to bring out is that with $k = 0$, there are servers that process almost 8,000 queries, whereas the most heavily loaded server with $k = 2$ processed only about 2,000 queries. Further, the vast majority of the servers processed less than half this amount. It is also interesting to note that for the $k = 0$ curve the "levels" of the namespace tree are clearly evident in the load plot. Fifty percent of the servers (the leaves) are covered by the first level, 75% of the servers (leaves and their parents) are covered by the first two levels, etc. Such gradations in load do not occur when replication is used. Thus, replication decreases latencies and helps balance load in the system.

Figure 4 plots query latency without using the digests described in Section 3. Comparison of Figure 4 and Figure 2 shows that the use of digests vastly improve query latency for non-zero k values. For $k = 2$, for example, the round-trip latency is less than seven hops *even without caches*, and falls below five even when very small caches are used.

In Figure 4, the $k = 0$ line crosses over the $k = 1, 2$ lines because node replication tends to prevent the buildup of "good" information in the cache of the server that owns a node. Instead, this good cache state is distributed among all of the servers that host replicas, where it is less useful to any individual query. This effect is still present in the system that use digests, but is overwhelmed by the positive impact of the digests.

4.1. Scaling

Figure 5 shows the scaling of query latency as the network size goes from 255 nodes to 128K nodes. As a rather severe test of TerraDir scalability, we use $2 * \log N$ cache entries for each point: this means that as the number of servers in the network doubles, the cache sizes increase by only two extra slots. Thus, the 255 node TerraDir uses cache size 16, and the largest (128K node) TerraDir uses only 34-element caches. The plot clearly shows the robustness and scalability the TerraDir caching and replication scheme: even with 128K servers and no

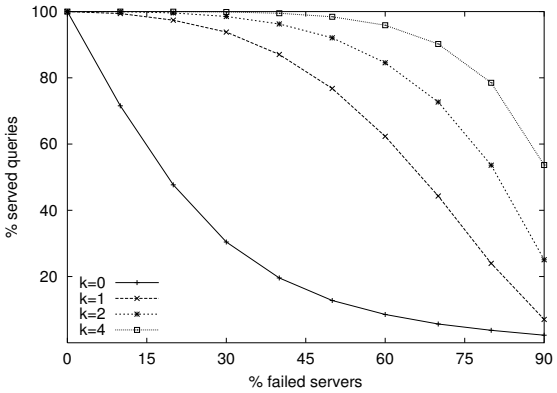

Figure 6. No. of served queries vs. fraction of failed servers.

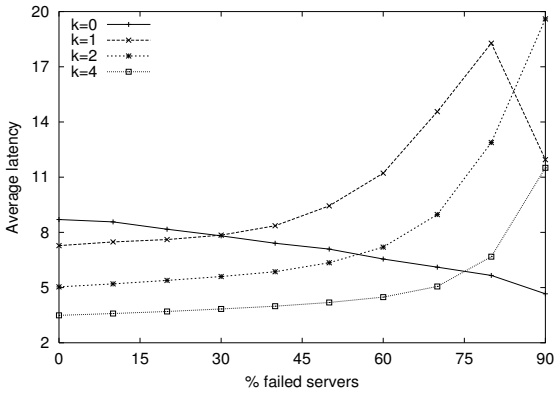

Figure 7. Latency of served queries vs. fraction of failed servers.

replication, 34 cache slots are enough to reduce the expected latency by 2/3. Lastly, note that these results are for uniformly distributed queries and we expect the results to improve with locality in the query stream.

4.2. Availability

This section shows the extent to which replication allows queries to be served in the presence of failed servers. We assume that all nodes are fail-stop[11]; we do not deal with Byzantine failures or malicious servers. Failed servers are assumed to be detected through timeouts. Recall that in TerraDir, replication is the only mechanism for providing high availability, and no extra failure-mode specific protocols are needed.

We repeated the previous experiment with 32K servers (and nodes), and 3M queries, but allowed a configurable fraction of servers to fail at the beginning of the simulation. Figure 6 plots the overall number of queries served versus the faction of failed servers. With $k = 2$ (our default for many of these experiments), we can route more than 90% of all queries even when a majority of servers have failed. With $k = 4$, we can route more than 90% of queries when more than 70% of the servers have failed.

The latency of those queries served in the presence of failures (Figure 7) shows some interesting characteristics. The latency of served queries strictly decreases with increasing failure rate for $k = 0$ (no replication), because queries requiring longer paths are failing in much higher numbers than shorter queries. With replication, query latency actually goes up because queries continue to be served in the face of massive failures, but the path between any two nodes becomes more and more roundabout in order to circumvent failed servers. For the $k = 1$ case, this latency eventually peaks and drops precipitously, reflecting the point at which queries can no longer be served and only shorter queries are successful, as with $k = 0$.

In general, failed servers impact query delivery in two ways:

- Failed destinations can prevent a query from being delivered. However, this only occurs when all replicas of the destination have failed. Figure 8 shows destination failures as we vary the percentage of failed servers. The query failure rate linearly increases with $k = 0$ (no replication), but rises much more slowly as the degree of replication increases.

- Failed servers can also cause queries to be dropped during the routing process. TerraDir notices such failures through timeouts, and then restarts the routing process at the last hop before the failure. Queries have a maximum time-to-live (TTL) of 100 hops. In practice, this limit is rarely reached, even when the majority of servers have failed. Figure 9 shows the percentage of failed queries because the routing protocol was unable to find a path from the source to the destination. Two of the lines have peaks, which reflect a hard limit on the possible number of route-failures set by the number of destination failures.

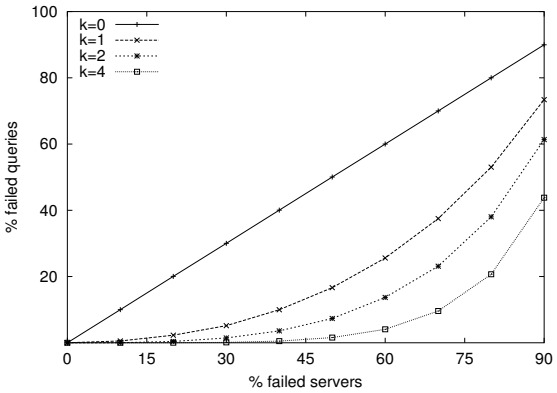

Figure 8. Failed query due to dest. failure vs. repl. factor (k).

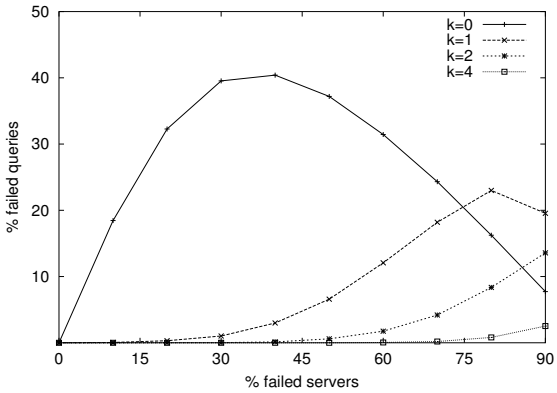

Figure 9. Fraction of queries failed en-route vs. replication factor (k).

Other results including locality in the query stream and irregular namespace trees, as well as discussions regarding multiple nodes per server and replica selection policies are presented in the extended paper.[10]

5. DISCUSSION AND RELATED WORK

Both Chord[4, 15] and CAN[7] implement distributed *virtual* lookup structures, and map elements of the namespace to these structures using global hash functions. The virtual structure in Chord is a circle, while CAN uses a $d-$dimensional tuple space. Chord further virtualizes the server addresses and maps these to the virtual structure as well, while CAN only virtualizes the namespace elements.

Tapestry,[13] the lookup service used in the Oceanstore[16] global file system, and PASTRY,[12] the lookup service used in the PAST[5, 17] global file system, both use virtualized hierarchies similar to Plaxton trees.[18] In the case of Tapestry, the virtual hierarchies are augmented with attenuated bloom filters and a limited form of caching.

TerraDir differs from all of these approaches by mapping the application-visible namespace directly onto the topology, and by not employing any virtualization. There are some interesting ramifications of such a design, e.g. in TerraDir, if a server "exports" a set of contiguous namespace elements, they are all mapped to the same server. Such a design will lead to lower access latencies for correlated accesses — typical accesses in a filesystem, the web, or a digital library. However, such a design can also lead to load concentration unless the underlying load balancing scheme is robust. In general, a complicated virtual structure may be more difficult to maintain when servers join and leave the system; in comparison, servers can be added and removed with very little cost in TerraDir.

We have limited our discussion to query routing, which is the sole service provide by Chord, CAN, and Tapestry. Direct performance comparison is difficult, but note that Chord and CAN use $O(\log N)$ information and route in $O(\log N)$ steps. While the size of the caches used in most of our experiments is also roughly $O(\log N)$, note that TerraDir's requires $O(\log N)$ steps to route a query only when neither replication nor caches are used (in this case our state is $O(1)$). When replication and caches are used, TerraDir's query latencies are significantly smaller, although no tight analytic bound can be derived.

However, while these other services only provide lookup services, TerraDir also provides for efficient searching. By retaining application information in the structure of the distributed tree, TerraDir is able to exploit the locality of most searches.[19] For example, assume that we wish to search for all red cars. If the TerraDir was set up with this sort of search in mind, all red cars will be combined to one subtree of the entire structure. By contrast, the other systems will need to either use some auxiliary mechanism, e.g. centralized indexes, or flood the entire distributed structure. This application information is also useful when directory accesses have locality.

Freenet[20] is a distributed, decentralized, peer-to-peer file system that emphasizes publishing anonymity. Search latencies are not bounded, searches are not guaranteed to find existing material, and the material itself has no guarantee of persistence.

The Globe[21] system is similar to TerraDir in that it implements a hierarchical structure,[22] complete with caching and limited replication. However, the system does not integrate these services to the extent that TerraDir does, and is more heavyweight.

TerraDir, as well as the lookup services discussed above, are quite similar in scope to directory services such as DNS.[23, 24] DNS is one of the cornerstones of the Internet and is possibly the most widely deployed directory service ever. Its decentralized design has withstood order of magnitude increases in both directory size and participating servers. While DNS provides somewhat similar lookup functionality to TerraDir, the deployed infrastructure is not provisioned to handle queries for arbitrary RRs from arbitrary applications. Further, DNS does not allow wildcard searches or searches using attributes, as TerraDir does.

X.500[25, 26] is an international distributed directory standard jointly developed by the ISO and CCITT (now ITU/T). X.500 is well-suited for federated international directories for large organizations, but imposes far too much overhead to be useful in our context. As an example, in order to use X.500, applications must use the X.500 object schema, use ASN.1 to describe their object attributes, use master-slave replication, obey the X.500 consistency rules, and implement the entire suite of X.500 auxiliary protocols.

6. CONCLUSIONS

TerraDir is a distributed directory service that provides both lookup and searching primitives for large distributed applications, such as peer-to-peer object sharing systems, global file systems, and resource discovery. TerraDir provides these primitives in a secure, robust, and decentralized manner.

TerraDir is able to exploit query locality and to provide searching capabilities because, unlike several other current projects with similar motivations, TerraDir does not virtualize the name space of the data. Instead, TerraDir maps hierarchical structure in an application's namespace directly onto the structure of the TerraDir network. Load balancing and availability are addressed through an adaptive method of creating replicas that provides higher replication for nodes higher in the hierarchy, while constraining the overhead of replication to a constant factor when averaging across the entire structure.

More information about TerraDir can be found at *http://terradir.cs.umd.edu.*

REFERENCES

1. "Nupedia home page." http://www.nupedia.org.
2. "Wikipedia home page." http://www.wikipedia.com.
3. *The Gnutella protocol specification.* http://dss.clip2.com/GnutellaProtocol04.pdf, 2000.
4. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proc. of the ACM SIGCOMM Conference*, (San Diego, CA), August 2001.
5. P. Druschel and A. Rowstron, "PAST: a large-scale persistent peer-to-peer storage utility," in *Proc. of the 8th IEEE Workshop on Hot Topics In Operating Systems VIII*, (Schloss Elmau, Germany), May 2001.
6. S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler, "Scalable, distributed data structures for Internet service construction.," in *In Symposium on Operating Systems Design and Implementation*, (San Diego, CA), October 2000.
7. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in *Proc. of the ACM SIGCOMM Technical Conference*, (San Diego, CA), August 2001.
8. "Gnutella home page." http://www.gnutella.com.
9. D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, pp. 654–663, (El Paso, TX), May 1997.

10. B. Bhattacharjee, P. Keleher, and B. Silaghi, "The design of TerraDir," Tech. Rep. CS-TR-4299, University of Maryland, College Park, MD, October 2001.
11. R. Schlichting and F. Schneider, "Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems* **1**(3), pp. 222–238, 1983.
12. A. Rowstran and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, (Heidelberg, Germany), November 2001.
13. B. Zhao, K. Kubiatowicz, and A. Joseph, "Tapestry: An infrastructure for fault-resilient wide-area location and routing," Tech. Rep. UCB//CSD-01-1141, University of California at Berkeley, April 2001.
14. B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the Association for Computing Machinery* **13**(7), pp. 422–426, 1970.
15. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. of the 18th ACM Symposium on Operating Systems Principles*, (Chateau Lake Louise, Banff, Canada), October 2001.
16. J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao., "Oceanstore: An Architecture for Global-Scale Persistent Storage," in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, (Cambridge, MA), November 2000.
17. A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Proc. of the 18th ACM Symposium on Operating Systems Principles*, (Chateau Lake Louise, Banff, Canada), October 2001.
18. C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, pp. 311–320, (Newport, RI), June 1997.
19. P. Keleher, B. Bhattacharjee, and B. Silaghi, "Are virtualized overlay networks too much of a good thing?," in *Proc. of the 1st International Workshop on Peer-to-Peer Systems*, Springer-Verlag, (Cambridge, MA), March 2002.
20. I. Clarke, O. Sandberg, B. Wiley, and T. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, pp. 311–320, (Berkeley, CA), July 2000.
21. M. van Steen, P. Homburg, and A. S. Tanenbaum, "Globe: A wide-area distributed system," *IEEE Concurrency* **7**, pp. 70–78, January–March 1999.
22. M. van Steen, F. J. Hauck, G. Ballintijn, and A. S. Tanenbaum, "Algorithmic design of the globe wide-area location service," *The Computer Journal* **41**(5), pp. 297–310, 1998.
23. P. V. Mockapetris, "Domain names - concepts and facilities," Request for Comments 1034, Internet Engineering Task Force, November 1987.
24. P. V. Mockapetris, "Domain names - implementation and specification," Request for Comments 1035, Internet Engineering Task Force, November 1987.
25. "The Directory: Overview of Concepts, Models and Service.." CCITT Recommendation X.500., 1988.
26. "The Directory: Models.." CCITT Recommendation X.501 ISO/IEC JTC 1/SC21; International Standard 9594-2, 1988.