The Design of TerraDir *

Bobby Bhattacharjee Pete Keleher Bujor Silaghi

University of Maryland, College Park, Maryland, USA
{*bobby, keleher, bujor*}*@cs.umd.edu*

Abstract

We present the design and initial evaluation of TerraDir: an approach for implementing customizable, distributed, peer-to-peer directories over which a broad range of wide-area resource discovery applications can be implemented. TerraDir's structure is two-tiered, consisting of a base protocol for providing arbitrary application-layer connectivity, and dynamic view materializations that efficiently realize different user-specified views of application-layer resources. TerraDir is specifically designed for data that can be arranged in a rooted hierarchy, and provides very efficient and robust lookups over such data while still being able to allow efficient searching.

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 [2, 3], 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[1] 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 the resources are advertised and accessed using cooperating nodes distributed throughout the network.

Each TerraDir directory is a private namespace; multiple TerraDirs can be active simultaneously and can be composed to form arbitrary virtual directories. TerraDir directories can be initiated by arbitrary application-level processes without expending vast amounts of resources. Directories are managed in a decentralized manner, as application-peers add directory nodes and the underlying TerraDir protocols maintain application-specified levels of performance, fault-tolerance, and consistency. TerraDir can locate specific directory nodes, and can also be customized to perform sophisticated node-attribute-based distributed searches over the directory namespace. Lastly, TerraDir is designed to be implemented over the existing Internet infrastructure; as such, all TerraDir protocols are implemented end-to-end, and they do not require any special network support beyond

*This work is supported by a grant from the National Science Foundation (NSF Grant #: ANI0123765).

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

unicast forwarding. As widely distributed peer-to-peer applications become common over the Internet, the need for scalable, application-customizable distributed directory protocols has become evident. The TerraDir protocols fill this basic need by providing powerful and flexible abstractions that can be used to scalably implement the infrastructure required by the emerging generation of Internet applications.

This paper describes TerraDir: a set of protocols for implementing customizable, distributed, peer-to-peer directories over which a broad range of wide-area resource discovery applications can be implemented. TerraDir has a two-tier structure consisting of a *base protocol* for providing arbitrary application-layer connectivity, and *dynamic view materialization* that efficiently realize different user-specified views of application-layer resources.

Overview of the base TerraDir protocol

The base TerraDir protocol is used to construct and maintain a distributed directory, and can be used to efficiently query the directory for name-based searches. The base directory structures scale across multiple dimensions, including the number of directory entries, the number of nodes the directory is distributed across, and available network, processing, and storage resources. The TerraDir protocols can be configured for application-specified levels of security, failure tolerance, and consistency. The base TerraDir query resolution algorithm can efficiently search for node name based queries and incorporates a sophisticated (and application-configurable) caching scheme that can exploit global directory access patterns to provide low latency access for large directories distributed across wide-area networks.

Obviously, distributed directory protocols have been investigated previously, and large distributed directories (e.g. the DNS directory and X.500 directories) have been in regular use for about two decades. However, the motivation, design, and underlying mechanisms of TerraDir differ considerably from DNS, X.500, and other prior work in distributed directories. Further, as described next, the dynamic view materialization extensions of TerraDir —essential for flexible implementation of different applications— are capabilities not available in any previous distributed directory protocol.

Dynamic views in TerraDir

The extended protocol builds on the base protocol by providing for dynamic view creation and dissemination across the entire structure. The base protocol provides for efficient, scalable handling of queries whose structure matches that of the directory structure. For example, a general auction application may segregate cars into one (possibly large) subtree of the directory structure. Queries searching for objects pertaining to cars are handled efficiently in the sense that only nodes in this subtree are touched during the query processing. However, a query looking for red objects would not be confined to any subtree. In fact, such queries could cause large portions of the directory to be searched.

More generally, query handling can become costly any time that query structure does not match directory structure. Possible reasons for such a mismatch are many: query types might not be available when the directory is first created, queries might evolve over time, and there might be multiple classes of queries, each class containing queries that are very different in structure from those of other classes.

We allow these cases to be handled efficiently by providing the means for dynamically building new directory views. The structure of these new views can be designed to match any particular query structure. Views are created according to a *tag structure*, which can be specified as either an ordering of node attributes, or based on view-specific functions defined at view time. Views are created in a decentralized fashion, diffusing through a directory through pairwise anti-entropic communication. Views can be seen as an overlay hierarchy on top of the base hierarchy. As such, they can be used to handle queries in the same manner as the base hierarchy, but with greater efficiency.

The rest of this paper is organized as follows: we describe major TerraDir abstractions in Section 2, and outline the base protocols in Section 3. Section 4 expands on dynamic views and how they could be implemented on top of the base protocol. Initial results of the base protocol are available in Section 5. Finally, Section 6 concludes the paper and discusses related work in the area.

2 TerraDir: Approach and Abstractions

We adopted an axiomatic approach to designing TerraDir's underlying protocols. Our goal was to derive the simplest, most scalable, and most general protocol allowed by the following axioms:

- **Decentralization:** The primary design goal of TerraDir was to enable a scalable wide-area directory that does not rely on a single authority or have any single point-of-failure. Thus, as far as possible, the design of TerraDir relies only on decentralized mechanisms.
- **Local techniques:** Large-scale structures will quickly become congested unless the effects of operations are confined to as local an area as possible. Hence, we try to avoid techniques that access or affect large portions of the system, such as distributed garbage collection or flooding.
- **Configurability:** To the extent possible, consistency, fault-tolerance, security, and performance enhancing mechanisms like caches are configurable by the application.
- **End-to-end solutions:** No network layer support beyond plain IP forwarding is expected. Of course, if there exists opportunity to use such services, such as IP multicast, we will avail of them.

We evaluate each aspect of the TerraDir design from the perspective of scalability, query and update performance, and global consistency, in that order. Underlying our approach is a strong desire for simplicity. For example, we do not rely on complicated load-balancing or introspection techniques [27]. Tuning such mechanisms on large-scale distributed structures introduces whole new layers and complexity. Further, it increases the odds that the system becomes unusable when the system is exposed to unanticipated situations.

2.1 The TerraDir Namespace

The primary abstraction exported by the 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 are constructed much like DNS or Unix file system names. Each node, except the root, has a non-null label. Each non-root node n_a in the namespace must have at least one directed edge incident upon itself. Each non-root node marks a single incoming edge as a *canonical edge*. Thus, the set of nodes and canonical edges form a directed tree rooted at the root node. The *canonical name* of node n_a is formed by the sequence of labels encountered in the unique path from the root to n_a, using only canonical edges. The *canonical parent* of a node n_a is the node n_p such that there is a directed canonical edge from n_p to n_a. Node n_a is a *child* of n_p.

We allow the creation of additional *symlinks* between nodes in the directory such that the overall structure is that of a general rooted graph, and not just a tree. Symlinks are necessary to support the common situation where an object might be filed under multiple names. Symlinks create additional, non-canonical, names for a node in the directory namespace. A node can be accessed using any of its names (including its canonical name). However, as explained in Section 3.2, only canonical names are used internally. Any non-canonical name used by a query is mapped onto a canonical name as soon as the target nodes are resolved.

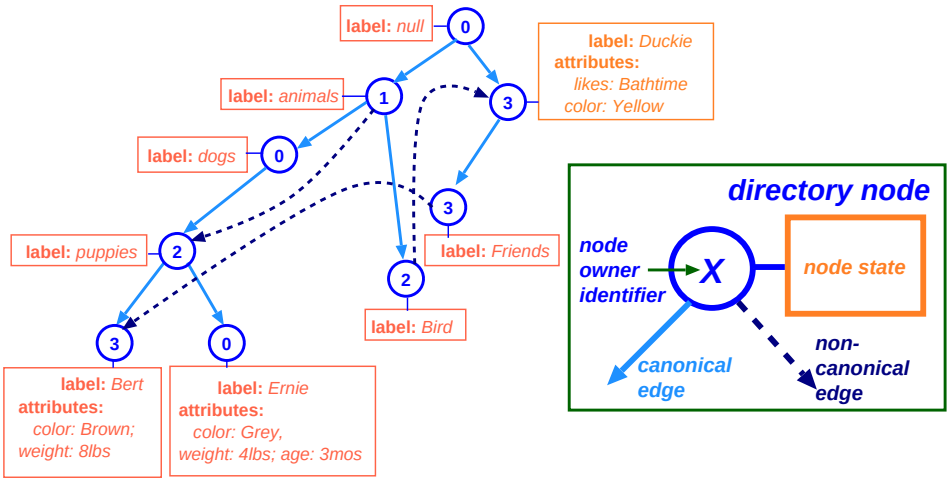

Figure 1: Example namespace

The state at each directory node consists 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.

2.2 Namespace Credentials

Namespaces in TerraDir are accessed using *namespace credentials*: a (namespace) credential consists of a namespace, a set of attributes, and a set of cryptographic secrets. At the very least, the attributes describe what level of access (e.g. read-only, read-write, etc.) the holder of the credential has over the nodes in the associated namespace. Other attributes may describe the caching characteristics, consistency semantics, and version information (temporal constraints) for the directory.

Credentials provide a convenient means of securely exporting namespaces and named views. Credential attributes allow users to exchange access rights using out of band means (e.g. email), to give others diminished rights in a given namespace, and to specify consistency/caching semantics. Figure 1 shows an example namespace. The owner of nodes in this directory can produce a credential corresponding to this namespace and perform various operations on the credentials. We have the following set of operations available for use with credentials.

1. *Copying:* Credentials are merely byte-streams, and can easily be copied and distributed. For example, server 0 can distribute a credential to the entire directory to some other principal in the system.

2. *Grafting:* Namespaces can be combined to form derived namespace (see Section 2.4).

3. *Splitting:* The owner of a directory might not always want to export all nodes with the same permissions. Such operations are possible by splitting the rights contained in a credential. For example, server 0 may export a credential which does not contain the subtree rooted at the node dogs.

4. *Constraining:* A namespace can be constrained in several ways. For example, the extent of the namespace can be constrained by changing the root to point a child node. A namespace

can also be constrained by using attributes to limit the ways it can be viewed or manipulated.

2.3 Query Semantics

TerraDir base queries consist either of *lookups*, or *searches*.

- A *lookup* is a request for metadata of a specific object. The object can be specified using either a canonical name or a non-canonical name. For example, the leftmost node in Figure 1 can be searched using canonical name /animals/dogs/puppies/Bert and also the non-canonical name /Duckie/Friends/Bert.

- A *search* is a request for a set of nodes that match specific search criteria . This search criteria has two parts. The first part defines a specific subset of the namespace to be searched. The second part of the search criteria consists of a named set of attributes.

  For example, the expression " /Duckie/Friends/$\star$; color:brown" on the directory in Figure 1 would search through all children of node Friends and match all nodes that have attribute type color and value brown. It is possible to use wildcards/regular expressions over both attribute types and values. As described in Section 4, TerraDir dynamic views allow efficient searches of the entire directory using arbitrary attributes (and functions over these attributes).

Note that the structure of the base search criteria is both expressive and limiting. It is expressive in that searches can be made across multiple levels of the namespace, and allows a great deal of freedom in using arbitrary attributes to narrow a search. It is limiting in that regular expressions do not extend over multiple levels, e.g. there is no way to directly write a general search that will return all nodes in the subtree rooted at that node. This limitation is crucial to the scalability of the system, as the number of objects whose metadata can be stored and accessed via a single namespace may reach arbitrarily large values.

2.4 Derived Namespaces: Composing Aggregate Views

We have so far only discussed directories in which all links are to nodes within the same namespace. We call such a self-contained namespace a *primitive namespace*. However, it is often useful to be able to support composite directories that contain nodes from multiple primitive namespaces, e.g. for a resource-discovery application that lists resources from different domains. TerraDir supports such abstractions using *derived namespaces.* A derived namespace is a namespace that has links to nodes of other primitive or derived namespaces. Note that all of the (non-root) nodes in a derived namespace are reachable using at least two names (one in its primitive namespace and one using the derived namespace). Derived namespaces allow the creation of "virtual distributed directories" by combining nodes from arbitrary other namespaces.

Derived namespaces may contain links to any node in some other derived or primitive namespaces. Let D be a derived namespace which contains nodes from some other namespace N. In order for a node from N to be included in D, there must be an edge from a node (say d) in D to some node (say n) in N. The node linked *to* (i.e. n) has no state for such incoming derived links, and such links may be added or deleted without interacting with the owner of n[2]. Once created, derived namespaces can be accessed and searched much like primitive namespaces, and credentials for derived namespaces can be distributed analogously. The creation and modification of derived namespaces is a particularly lightweight (local) operation, since nodes upon which derived edges are incident are not involved in any derived namespace-specific operation.

[2]All local polices are still followed. Specifically, if a read-key is required to read the contents of n, then such a key must be available to the owner of d as well. (Security issues are discussed in more detail in Section 3.6.)

3 TerraDir Internals: Algorithms, Protocols, and Mechanisms

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. In the following sections, we describe the protocols for updating the TerraDir namespace, the internal caching algorithms for improving access latencies, and the TerraDir failure detection and recovery protocols. We begin with the basic protocols for modifying the TerraDir namespace.

3.1 Creation and Maintenance of the Namespace

Edge creation
Nodes are easily created, as the data structures used to hold node information are allocated and populated in a straightforward manner. A directory's root node can be created by any application/server that wants to start a new TerraDir directory[3]. The root node's existence and address are both assumed to be either well-known, or disseminated in an application-specific manner. Non-root nodes are "grafted" onto the directory tree by adding edges from existing nodes to the new node.

Assume server x wants to add new node n_a to a directory. Unless otherwise specifically delegated, we assume that the *owner* of a new node is the server that hosts the node's metadata. Thus, the owner of n_a (denoted by *owner(n_a)*) is x. Server x has complete control over the data named by a node, its metadata, its degree of replication, and handles all operations regarding n_a [4]. The first edge whose destination is node n_a is the canonical edge, and the edge's source is the canonical parent.

Further, assume n_b is n_a's canonical parent (if n_a could be added successfully) and let y be *owner(n_b)*[5]. Node x communicates to y a request to "graft" (add) a child (namely n_a) under n_b in the directory. Node y either decides to add n_a to the directory or reject server x's request, subject to application-specific policies, and notifies node x of its decision. After y adds n_a to the directory, new nodes can be added under n_a without further notifying y. Thus, in the simplest case, nodes (and by closure arbitrary canonical trees consisting of nodes) can be added to a directory by just communicating with a single node that owns some part of the directory.

Node deletion and renaming
While creation of a node is relatively straightforward, renaming a node is more complicated. Assume that the owner of node n_a ("/animals/dogs/puppies") wishes to rename it to "/animals/dogs/small dogs". This may not be acceptable to the owners of nodes below n_a, e.g. "/animals/dogs/puppies/Bert" and "/animals/dogs/puppies/Ernie", each of which might be owned by different servers.

The system accommodates this by augmenting the node creation protocol to include the notion of *prefix immutability*.When n_a is created by parent n_b, n_a might ask for, and n_b might grant, a guarantee that the ancestors of n_a will not change until after n_a itself has been deleted. If either n_b or an ancestor of n_b did not get this guarantee, however, then n_a can not be granted prefix immutability either.

This notion of immutable paths does not preclude subtrees from being grafted to new positions in the graph in addition to the immutable position, nor does it actually preclude the immutable graft from being deleted. However, unless immutability is explicitly revoked (as described below), all

[3]Our discussion refers to operations on *the* directory, e.g. adding a node to *the* directory. In general, there can be an arbitrary number of directories in simultaneous use, and presumably, the notion of *the* directory is clear from application context.

[4]However, queries directed to n_a can also be served by one of n_a's replicas.

[5]We discuss how x finds *owner(n_b)* in Section 3.2

additional incoming edges added to this node must also agree to the prefix immutability condition. The owner of node n_a can change n_a's prefix immutability bit if and only if all immediate children of n_a have their immutable prefix bits clear as well. Thus, prefix immutabilities of entire subtrees can be changed using a bottom-up approach by recursively revoking immutability starting at the leaf nodes.

The owner of a node may delete it if and only if the node does not have prefix immutability. Of course, owners or some nodes may fail or simply choose to not participate in the TerraDir node deletion protocol — these are the cases handled by the failure detection and ownership revocation protocols (Section 3.4).

3.2 Query Processing

Lookups
A lookup is a request for the attributes (all metadata) of a given object. A query consists of the name of the desired node in the directory, and can be submitted by any participating TerraDir server of the current application. In its most basic form, a query is handled by being passed from a source node, up and down the directory, to the destination node. For this discussion, we will ignore the effects of caching and replication.

Assume server Q submits a query for node n_d to server S. The query argument n_d is a sequence of labels that name edges (canonical or otherwise) on a path from the root to n_d. In the degenerate case, S is the owner of node n_d and returns to Q the relevant information. More frequently, S is not the owner of node n_d, and must use the labels in n_d to find the server that owns n_d.

In the general case, therefore, a query must traverse a portion of the directory tree. Note that "traversing the tree" implies network communication each time the query is passed between two nodes that are not owned by the same server. A path from S to the query destination is constructed by repeatedly forwarding the query "upwards" in the hierarchy until reaching the node whose label is the longest common prefix of n_d, and then "downwards" to the destination. Since each level so traversed may require network communication, finding shorter paths can significantly affect latency and overall resource usage.

If node n_d does not exist, then the lookup must return null ($\perp$). There must be a largest integer i, such that d_i belongs to the directory and d_{i+1} does not. Thus, *owner(d_i)* can reject the query since it is the authoritative source for all its children.

Example: Assume server Q connected to server 3 in Figure 1, wishes to lookup entries " /animals", " /animals/dogs/puppies/Ernie", and /animals/dogs/little-dogs. The query for " /animals" is forwarded from server 3 to server 0 (since " /" is the prefix closest " /animals" from the set of prefixes owned by server 3). Server 0 forwards this message to server 1 where a complete match is found and the server 1 directly sends the data corresponding to " /animals" back to server Q. Server 3 owns the node " /animals/dogs/puppies/Bert" and hence knows the owner of the node " /animals/dogs/puppies". Since this is server 3's best known longest common prefix for the query " /animals/dogs/puppies/Ernie", this query is forwarded from server 3 to server 2 and subsequently to server 0, where it is resolved. Server 3 forwards this request to server 0.

Server 0 immediately rejects the last query since it owns the node " /animals/dogs" and knows authoritatively that " /animals/dogs/little-dogs" is not part of the directory.

It is easy to see even in the worst case, this lookup algorithm will incur costs on the order of the directory tree's height. Furthermore, the use of symlinks implies that there may be multiple paths to any node, and the path described above may not always be the shortest. However, Section 3.3 describes how the costs can be reduced significantly by using a modest amount of caching.

Searches
Searches are queries that contain at least one regular expression. In general, a search expression with one wildcard has the form $/d_0/\ldots/d_i/\star/d_i+1/\ldots/d_k$. A search on an expression of this form proceeds as follows: the prefix of the search expression (i.e. the portion that does not contain any wildcards) $/d_0/\ldots/d_i$ is routed exactly the same as a lookup. Assume the owner of node $/d_0/\ldots/d_i$ is O; a set of new sub-queries is initiated at from server O along each of node $/d_0/\ldots/d_i$'s *canonical* children. For each child with label c_j, this query has the form $/d_0/\ldots/d_i/c_j/d_i+1/\ldots/d_k$. The union results of each of these sub queries is the result of the search. For both lookups and searches, each resultant node returned to Q carries with it its canonical name (which Q can then use to locate the node even if the node owner fails).

Example: Assume server Q connected to server 3 searches for " /animals/$\star$/puppies". This query is routed to server 1 from server 3 precisely as in the lookup example. Server 1 initiates two sub-queries for " /animals/dogs/puppies" and /animals/Bird/puppies and sends these queries to servers 0 and 2 respectively. The result of the sub-queries are eventually sent back to server 1. When server 1 gets a reply from both its children, it sends the result union of the non-null results back to node Q.

The exact same procedure would be used recursively and a separate sub-query initiated if the suffix of the query also contained wildcards. It is important to note that wildcard queries are only sent along canonical paths, otherwise queries may encounter cycles and recurse infinitely within the directory.

It is possible to allow queries that do traverse non-canonical links. In this case, additional mechanisms are required to ensure queries are not forwarded in cycles, e.g. each sub-query could contain the list of directory nodes it has already traveled and the same sub-query would not be forwarded multiple times over a single directory node.

3.3 Caches and Digests

As described so far, each query starts at some point in the directory namespace, goes *up* until a common prefix with the destination is found, and then *down* till it can either be resolved or rejected. Each directory tree edge visited during this traversal may require network communication. However, there can clearly be a great deal of locality in queries, and this locality is exploited by 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 of query endpoints.

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 gives the routing algorithm shortcuts to other portions of the TerraDir.

A straightforward enumeration of all hosted nodes in a cache entry's state might prove expensive Instead, we add a *digest* of the list of hosted nodes to the state kept for a server. Thus, whenever a node is cached, the digest of the nodes held by the server is also available. We currently implement the digest using a Bloom filter [11].

Our digests (Bloom filters) 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.

Query routing
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.
- 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.

3.3.1 Cache Inconsistencies

However constructed and populated, caches becomes inconsistent when the canonical name or the owner of a node changes because *caches are not updated* when directory structure changes. First, the owner of a node might change due to either failure and recovery or due to explicit handoff. In the former case, access to failed node would fail, causing the cache to discard the tuple. In the latter case, the old owner would explicitly deny ownership, again causing the cache to discard the tuple. Additional mechanisms (e.g. the old owner forwarding the query to the new owner) are required to distinguish a ownership denial from a failed query.

Second, the canonical name of a node might change because an ancestor has been renamed, deleted, or in some other way modified. Any such modification immediately makes each descendant's notion of its canonical name inaccurate. Since modifications are not propagated to descendants, this discrepancy is only revealed when a query is delivered to such a node via a cached tuple. The name specified by the query will be different than the node's copy of its name. However, the node can verify that the query name is correct by using the *path check* mechanism described below.

More troubling are the cases illustrated by the following example. Assume node "/animals/puppies/Bert" is renamed to "/animals/little-dogs/Bert", and a new node with the same name ("/animals/puppies/Bert") is created at the same point in the directory. The interaction of cached names and such modifications lead to a number of potential problems:

- A query destined for the original "/animals/puppies" is routed to the new "/animals/puppies" if the query creator was not informed of the change.
- More problematic, however, would be a query for the new "/animals/puppies/Bert" delivered directly to "/animals/little-dogs/Bert" because of a cache record that referred to the first "Bert". In the absence of checks, the owner of "/animals/little-dogs/Bert" will serve the node "

/animals/puppies/Bert" without knowing that its canonical name has changed (since we do not require the owner of " /animals/puppies" to inform its subtree of any name change)[6].

- The last problem is the worst case extension of the previous case in which the owner of " /animals/little-dogs/Bert" obliviously responds to a query for " /animals/puppies/Bert" even when a *new* " /animals/puppies/Bert" node has been grafted on to the directory!

In the first case, the directory is in fact resolving queries correctly and the "problem", if there is any, has to be solved using application-specific mechanisms, either at the the application or session levels. In the last two cases, correctness is violated and explicit mechanisms are required to handle the problems. We do not impose a specific consistency model on TerraDir. Instead, we describe a new technique, called *path checks*, that allows the application developer to choose their own tradeoff between flexibility, overhead, and consistency guarantees. Path checks are a mechanism to ensure consistency, and can be configured to provide absolute guarantees (i.e. all queries are answered correctly) or *eventual consistency* (i.e. there is a configurable "window of vulnerability" after an update to the namespace during which queries may be answered incorrectly, but eventually this will be detected and corrected for all subsequent queries). As expected, the cost for implementing path checks that provide stronger guarantees incur correspondingly higher storage, processing, and communication overhead.

Path checks
Assume the directory is configured to have a window of vulnerability of t seconds. For some applications, e.g. a phone directory, this window could be hours, while it may be on the order of minutes for others, e.g. a distributed file system. Further, each cache entry is tagged with a timestamp[7]. Assume the server A is the owner of n_a and is about to resolve a query for n_a. It is possible that the query used some cached routes to find A and in fact the set of labels n_d used in the query is no longer valid because of updates to n_a's ancestors. As a part of routing the query, every server on the way updates the timestamp for any cache entry it may have used to route the query such that the query contains the *earliest* timestamp (say ts) that was used to route the query. If ts is more than t seconds in the past, then server A initiates a path check before resolving the query.

Once again, let the query n_d be the set of labels $d_0, \ldots, d_k$. A path check for the query is a set of messages from the owners of d_k to d_0 that checks to make sure the labels in the path are, in fact, valid. Note that the path check does not always have to traverse all the way back to the owner of d_0. It can stop and the path can be returned as valid as soon as the owner of some node containing label d_i on the path can validate the prefix $d_0, \ldots, d_i$. (A prefix is valid if and only if it has been checked or otherwise proven valid within the last t seconds).

3.4 Replication and Failure Recovery

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 [42].

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 or deleted. Thus, the system must have a process for forcibly reassigning ownership of ownerless nodes to a different server.

[6]The simple solution that just informs children of name changes is not sufficient since in the general case, a label for any ancestor in the tree may change (and not just the immediate parent as in this example). For a very large directory, it is not viable to reliably inform all descendants when the label of a node close to the root changes its label.

[7]We are implicitly assuming that clocks are synchronized within some tolerance τ, $\tau << t$.

It is possible for a node to fail for a transient period (e.g. 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.

Replication

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. (Note that this derivation can be applied to any balanced tree).

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[8], 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.

The network addresses of servers that replicate node n_a become part of node n_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[9]. The replication factor k is application-specified and can potentially be different for different nodes in the directory.

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 extend the TerraDir routing algorithm from Section 3.3 as follows to handle server failures (the pseudo-code can be found in Appendix 1):

- 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.

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

[9] 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.

Node failures are detected via timeouts either by querying servers, or parent servers that attempt to forward message 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".

The replication and fault-recovery algorithm described above can recover every node in the directory *except the root node*. Instead of adding a complicated recovery procedure, we merely replicate the root node at an application-specified number of replicas and make this state part of every server. This solution has the desirable properties of application-configurable overhead and provides ready access to the root information in case the root node encounters a transient failure. Further, this procedure simple to implement, is required to boot the application, and has proven robust in DNS for the last two decades.

3.5 Network Layer Algorithms

A scalable and efficient implementation of TerraDir presents a number interesting and challenging problems for the underlying communication subsystem. We specifically consider mechanisms for application peers to find *nearest* servers that participate in a given TerraDir. This peer-finding problem has to be solved for implementing TerraDir but is generally applicable to any peer-to-peer application with unrestricted participation.

The peer-finding problem can easily be solved if we assume network-layer assistance or using TTL-scoped searches on a global IP multicast group. However, since we desire a solution deployable on the global Internet, special network-layer assistance is not tenable (even global IP multicast is still not available). Obviously, if our solution is to scale, we cannot resort to network level broadcast or uncontrolled flooding. A number of scalable solutions can be designed using network-layer assistance or overlays of global application-layer services. These include global anycasting [26, 25], stateful forwarding [49], the application-layer anycast architecture we developed in [9, 51], application-layer server selection [17], and Internet Distance Maps [24, 18]. However, by design, we only consider using services that are already deployed over the Internet; in the absence of network-layer assistance or global multicasting capabilities, the nearest server location is indeed a difficult problem.

We have adopted the following solution for TerraDir: We designate a set of k servers ($B_0, \ldots,$ B_{k-1}) to be *beacons*. The set of beacons could be the same set of servers that are compiled into the application as part of the replicated root state (See Section 3.4). We assume the address of at least one beacon is known *a-priori* to each application peer. Note that beacon can participate in the application as peers and can even be node owners.

Assume server n is new and wants to locate a nearby node owner. Under TerraDir, it would use the beacons and the following algorithm [10]:

- Each node owner periodically reports its distance (in hops) to the k beacons.
- Server n sends its own distance d in hops to a single beacon (say b_0).
- Beacon b_0 sends to n a list containing all the identities of all servers within distance $d \pm \delta$ of itself.
- Server n repeats this procedure by sending its own distance (in hops) to some other beacons b_i and accumulates a number of prospective *near* servers.

[10] The algorithm described is a simplified version and is correct under the the following assumptions: (1) distances in the topology obey the triangle inequality and (2) the nearest directory is closer to the searching node that to a beacon.

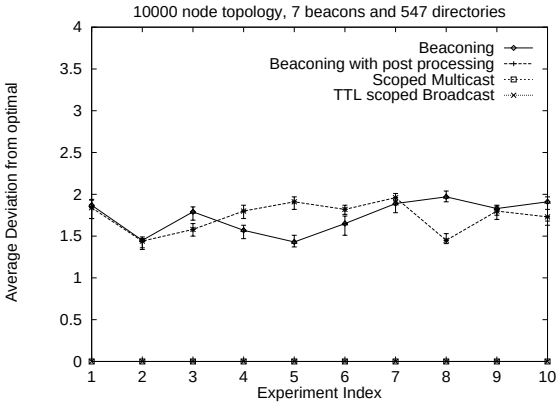

Figure 2: Accracy of Beaconing versus network-based schemes

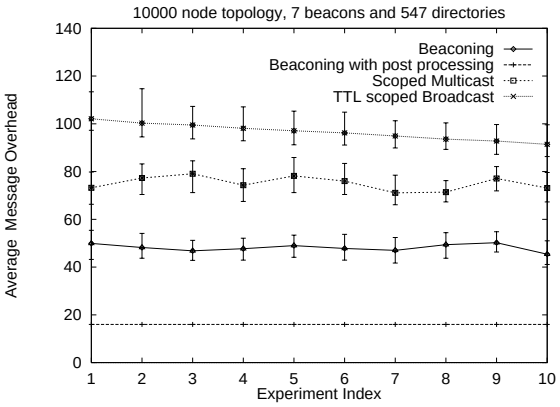

Figure 3: Message overhead of Beaconing

- After server n communicates with a suitable number of beacon nodes, the intersection of suitable nodes reduces to a *reasonable* number and n can find nearby servers by measuring its own distance against this much reduced set. As we show in our preliminary results, it is also possible to apply heuristics to choose from this set.

We have begun experimenting with this algorithm and initial results are encouraging. Figures 2 and 3 show the accuracy and overhead of our scheme when it is simulated on a 10, 000 node graph created using the GT-ITM topology modeler [12]. The figures show the average deviation from the true closest server (in hops) and the message overhead for ten different beacon and server placements. For a given experiment, each point is an average of ten runs; in each run we varied the location of the new node uniformly at random while keeping location of the servers and the beacons fixed. Even using only seven randomly placed beacons, the message overhead for beaconing is in fact smaller than even native multicast-based solutions, (but does require the centralized state at the beacons). The figures also show that our post-processing techniques for further reducing overhead are effective and that the beaconing scheme is extremely accurate. (In these experiments, beaconing does not find the "nearest" server all the time because we did not enforce the two correctness assumptions). Lastly, our preliminary studies also show that beaconing more accurate and incurs orders of magnitude less overhead than techniques previously reported [23, 13].

Optimizations and refinements

The protocol, as described above, requires all k beacon nodes to be available at all times and for each beacon node to maintain state about every server. Further, the set of beacon nodes is fixed *a-priori* — this can lead to a bottleneck if the number of servers accessing the directory is very large (and k is not proportionately large enough). All three of these stipulations can be relaxed. Briefly, new beacons can be added and only the original set of beacons need to know about the newly added beacons. Then as long as one of the k beacons are available, new servers can find enough beacons (the one beacon they communicate with informs them about the new list of beacons) to find nearby servers. Lastly, we note that as long as fine-grained distance information is maintained between beacons, recent techniques described in [24, 45] can be used to *infer* distances between beacons and existing servers and between existing and new servers.

3.6 Security

While security is not the primary focus for TerraDir, node and server security are an integral part of the TerraDir design. We have designed a general security model for hierarchical data arranged in

a peer-to-peer architecture. The techniques developed for this architecture are directly applicable to TerraDir and we briefly summarize our basic approach.

For each type of access (read, write, etc.), we provide a means of deriving per-node keys from a single *secret* per namespace. Consider namespace NS_α, and and let S_α, be the secret for some permission for NS_α. Let n be some node in the namespace with label $label(n)$ and let $parent(n)$ be the parent node of n in the namespace. The key for node n, K_n in NS_α is derived as follows:

$$K_n = \begin{cases} H(S_\alpha) & \text{if } n \text{ is the root} \\ H(label(n)|K_{parent(n)}) & \text{otherwise} \end{cases}$$

Thus, the key for node n is the cryptographic hash of n's label concatenated with the key for n's parent. Since the root has no parent, it's key is merely the hash of the secret. Note that in order for this technique to be useful, it should be computationally infeasible to invert the hash function $H(\cdot)$.

The scheme described above is a simple and efficient technique for key derivations within a single primitive namespace. It allows the implementation of different types of key relations between TerraDirs (e.g. if needed, keys for a directory can be derived from the nodes of a different directory). Further, our approach allows clients to derive less powerful credentials through a completely local process. We have extended the basic protocol to handle derived namespaces.

Note that the use of key derivations allow TerraDir servers to hold only constant amount of cryptographic state per namespace. Specifically, there is no per-node or per-client security state. Any request for a directory operation is verified using the appropriate key. We use a simple protocol to prevent eavesdropping and replay attacks on keys during remote access. All communication that require security is over secure channels (e.g. TLS tunnels) and consist of a request and a credential. The credential C is of the following form (much like credentials in Kerberos [33]) $C = \{H(K|i|ts), i, ts\}$. We have also developed techniques to efficiently handle rekeying, node renaming, changing the local key derivation policy, and key revocation.

4 Dynamic View Materialization

TerraDir's base protocol uses a decentralized structure to build a relatively static hierarchy. The directory structure is dynamic, but each node and edge must be individually created and deleted. The central purpose of a directory structure is to structure information in a way that makes it easy to answer queries. A static, hierarchical structure makes it easy to answer queries, *as long as those queries are structured like the directory.*

As a trivial example, Figure 4a shows workstations organized into a flat namespace, with attributes `arch` (processor architecture), `MHz` (processor clock speed), and `mem` (megabytes of RAM) in curly braces. This organization would be inefficient for attribute-based queries. For example, searching for resource-poor workstations to run Windows-based terminal software would require visiting all nodes to read and compare their attributes.

By contrast, Figure 4b shows a reorganized hierarchy where machine attributes are reflected in the node structure. This hierarchy would be able to support the above query much more efficiently than 4a. Note that the second hierarchy is efficient for two reasons. First, the attributes are reflected in the hierarchy, making exhaustive searches among all nodes for specific attributes unnecessary. Second, the attributes are arranged in an order consistent with query structure. The structure of 4b would not as useful if the query were intended to find the machines with the most memory. The problem of ensuring that the hierarchy's structure matches that of the queries is difficult because the form of the eventual queries might not be known when the original structure is created. Additionally, queries might come in many forms, and there may be no one organization that performs well for a majority of the queries over the lifetime of the namespace.

We address these problems by dynamically building new *views* in response to *view queries*. A view is a new overlay hierarchy based on a total ordering of a set of per-node tags. At it's simplest, a tag may just be an attribute of the node. Views therefore can be thought of as a way to

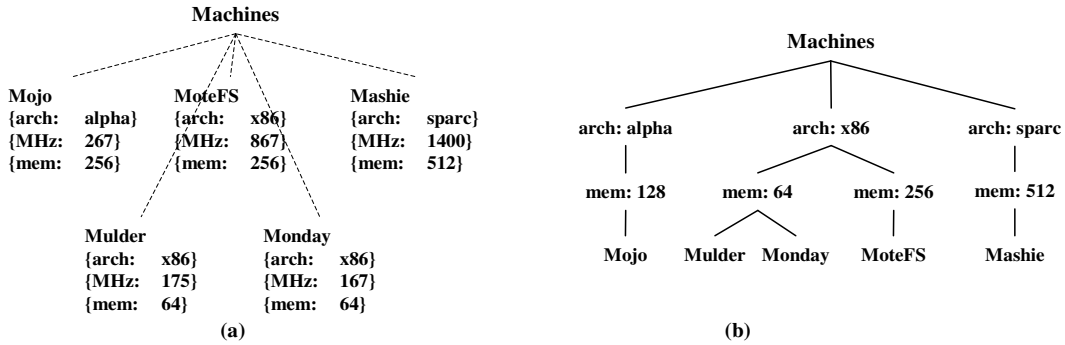

Figure 4: Workstations organized into **(a)** a flat hierarchy, with architectural attributes (amount of memory, speed (MHz), processor architecture) shown, and **(b)** an attribute-based naming tree that could result from view-query "/arch/mem". Note that not all attributes need be used.

create attribute-based naming trees. For example, Figure 4b could be the result of the view query "/arch/mem". Views are created by reorganizing the leaves of the canonical directory structure (referred to as "the namespace" in the remainder of this section) into a new hierarchy, and may be viewed as a completely independent overlay namespace that uses the same set of leaf nodes. Views may be used to satisfy queries in exactly the same fashion as namespaces, although they may not be combined and aggregated. Interior nodes of a view are synthesized during the view creation by the system. Among other advantages, this also gives the system flexibility to replicate interior nodes at will.

The state that composes a view is scattered among the leaf nodes. This means that an individual node potentially has state for both the namespace and an arbitrary number of views. However, a view is essentially treated as a cache by the system. Views are dynamically created in response to view queries, but view state can be unilaterally discarded by nodes in response to resource limitations.

A view can be named by a credential in much the same way as a namespace. Such a credential contains a *view identifier*, an identifier for the underlying namespace, and a pointer to both the root node for the view and for the namespace. As an optimization, a view might also be partially materialized in the handle. For example, the handle might contain edges and node-ids/network-addresses for the top few levels of the hierarchy.

Dynamic view queries

Using attributes as tags allows resources to be organized as attribute-based naming trees. Such a naming tree has many uses in general resource-discovery problems. However, such situations can usually benefit from the ability to define queries that return the "best" answer.

We enable such functionality by extending queries to include limited comparison operators, such as "highest" (denoted by >), and "lowest" (denoted by <). With Figure 4b, for example, a query for the Intel-based machine with the most memory could be constructed as "/x86/>/*. These static attributes might be cached at the parent node to allow for faster response.

We also enable a limited form of dynamicism by allowing such values to be returned by callback functions. Such a function could return time-varying (or node-specific) load properties.

View creation

The above description of views assumes an efficient implementation of view creation. A naive approach for view creation incurs time overhead linear with the number of nodes. The view is created by propagating the query and attribute information up and down the canonical namespace

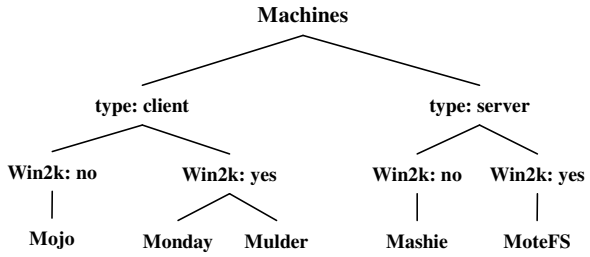

Figure 5: New view created by a tag function that uses existing attributes to categorize machines as either servers or workstations, and by whether or not they will run Windows 2000.

a maximum of four times (query up, query down, attribution information up, final view construction down).

However, large systems would be better served by a more decentralized solution. Our approach will be similar in spirit to that of Plaxton et al. [36]. Information about a new view query is disseminated from the query node to neighbors in the namespace in pairwise fashion. The query node creates nodes above it in the view. When information about the query propagates to other nodes, it carries with it a partially materialized view consisting of some of the nodes at the top of the view. Nodes receiving this information attach themselves to interior nodes, and create other interior nodes, by disseminating information elsewhere in the namespace. In this way, the new view slowly condenses and diffuses through the namespace.

There remain a wealth of issues that we have not yet explored. We have not yet defined view semantics in all circumstances. Views are clearly "best effort" because state may be unilaterally discarded by individual nodes. Incomplete views might still be able to fully satisfy queries that only need undiscarded state. However, other queries might depend on state that has been discarded. Finally, we have not at all addressed how, or even whether, to integrate newly created nodes into existing views.

Tag functions

We have so far assumed that views are constructed through a straightforward ordering of pre-existing attributes. However, this assumes that pre-existing queries precisely capture the grouping and locality that we wish the hierarchy to reflect. This might not always be true. Queries might be more efficiently served via a view hierarchy whose structure is synthesized from the attributes, rather than being a simple ordering on the tag values themselves. We enable this functionality by supporting the use of "tag functions". Tag functions are arbitrary functions that map attribute and data values to an ordered set of tags. The resulting tags are then used to construct a view as above. Moreover, values produced by tag functions can also be used for dynamic view queries.

As an example, we might wish to create a view that categorizes machines as either potential workstations or potential clients, and then further distinguishes them by whether they can run Windows 2000. In this case, a tag function might classify a machine as a potential server if it meets or exceeds speed and memory thresholds. A machine can run Windows 2000 if it is based on an x86 processor, and has at least 64 MB of RAM. Figure 5 shows one possible resulting hierarchy.

As namespaces are potentially long-lived, persistent structures, arbitrary tag functions can not be presumed to be pre-linked into TerraDir libraries. Instead, we will use some form of mobile code [43, 32, 47] to transfer the tag function from the view query initiator to system nodes. Our current plan is to use a verified form of TCL [35]. Recall from above that view creation is a decentralized operation. The tag function can be disseminated through the directory merely by replacing the simple view query with the tag function.

Note that this works because our current plan is to only allow tag functions to use data and

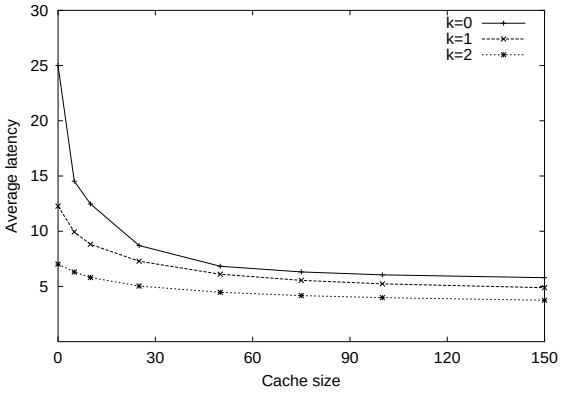

Figure 6: Query latency as a function of cache size and replication factor (k).

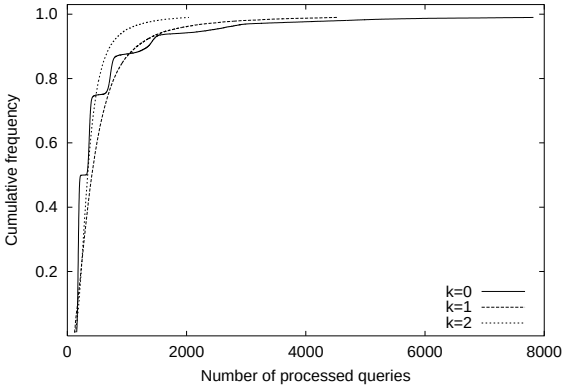

Figure 7: CDF of served load, cache size 25.

attributes local to a single node when generating the tags for that node. We do not allow tag functions to access data from multiple nodes, even though such access might be useful in creating views sorted on attribute or data values. The problem with allowing tags to be dependent on data or attributes for multiple nodes is that it introduces coherence and ordering problems among different nodes during view creation. Currently, view queries (and tag functions) may be disseminated to system nodes in any order without affecting the final view structure.

5 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 (see Section 5.5 for a discussion of this choice).

5.1 Routing Performance

Figure 6 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 7 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 3.4). 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 queries represented by the point's x value. (Each satisfied query must be routed either to the owner of the query or to a replica). 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

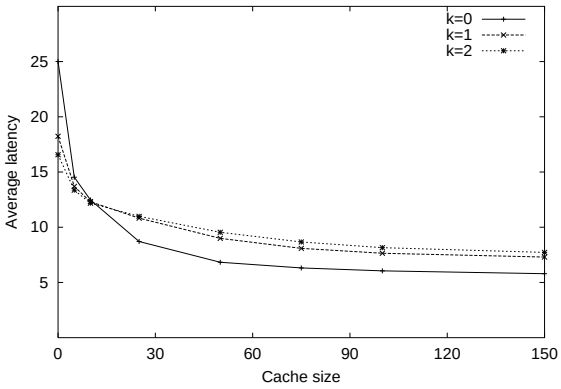

Figure 8: Query latency w/o digests.

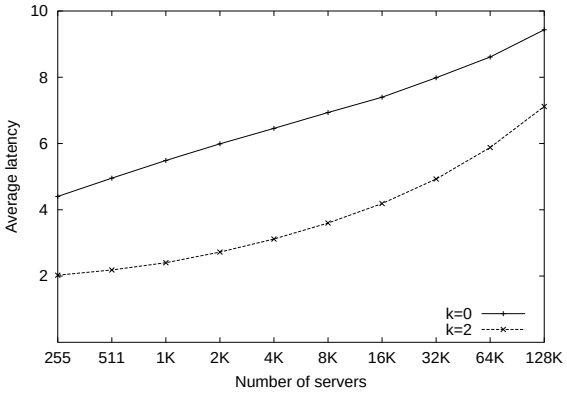

Figure 9: Query latency vs. size of network.

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, the replication decreases latencies and helps balance load in the system.

Figure 8 plots query latency without using the digests described in Section 3.3.

Comparison of Figure 8 and Figure 6 shows that the use of the digests vastly improve query latency for non-zero k values. For $k = 2$, for example, round-trip latency is less than seven hops *even without caches*, and falls below five when even very small caches are used.

In Figure 8, 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.

Figure 9 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, 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 replication, 34 cache slots is 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 (see Section 5.3).

The basic TerraDir caching schemes can be made even more effective using self-organizing caching techniques we introduced in [10] and the techniques described in [16]. Since caching results will invariably improve due to query locality and application-specific replacement policies, we expect the TerraDir caching scheme to scale extremely well with respect to both query path length and load on servers.

5.2 Availability Results

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 [42]; 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 10 plots the overall number of queries served versus the faction of failed servers. With $k = 2$ (our default for many of

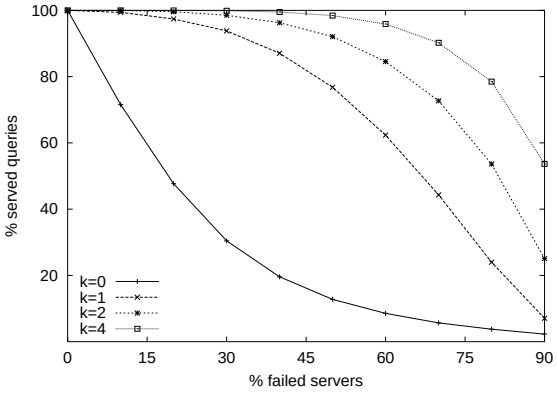

Figure 10: No. of served queries vs. fraction of failed servers.

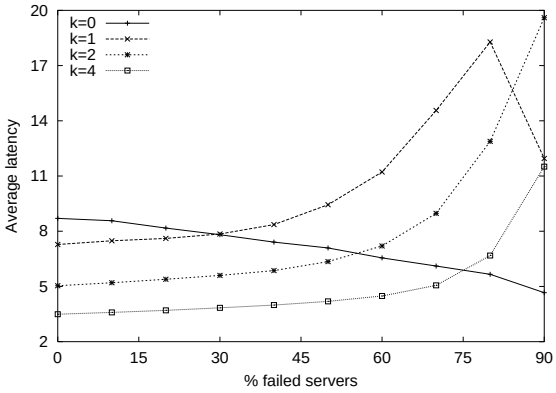

Figure 11: Latency of served queries vs. fraction of failed servers.

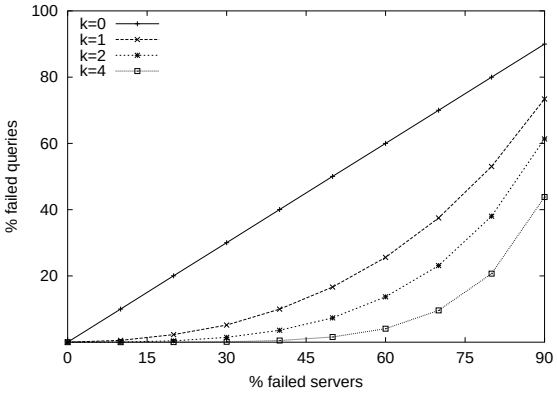

Figure 12: Failed query due to dest. failure vs. repl. factor (k).

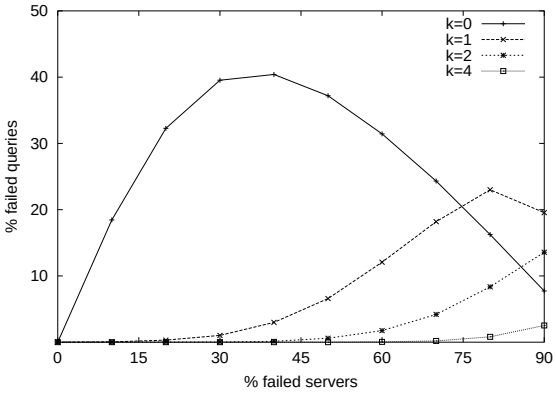

Figure 13: Fraction of queries failed en-route vs. replication factor (k).

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 11) 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 12 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

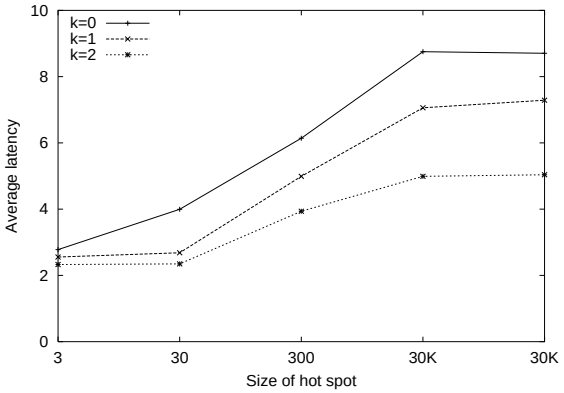

Figure 14: Latency with global locality vs. repl. factor (k); cache size: 25.

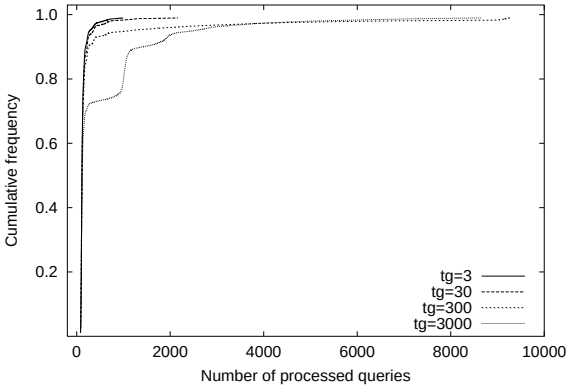

Figure 15: Load balance versus size of "hot spot"; replication factor $k = 0$.

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 13 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.

5.3 Query Locality

By not virtualizing the key space, as is done in CAN [38], Chord [44], and PAST [40], TerraDir is able to exploit locality in query stream. In all cases, latency vastly improves with with increasing latency. Figure 14 shows latency versus the number of size of the "hot spot" in the network. We assume that 90% of all queries go to this set, which is chosen uniformly at random from the set of all nodes.

Unsurprisingly, caches are able to profitably exploit increasing locality.

More worrisome is the effect of locality on load balance. Servers hosting nodes in or near the "hot spot" could be overstressed. Figures 15 and 16 show CDF's for the load on hosts in the system for $k = 0$ and $k = 2$, respectively.

For example, Figure 15 shows that load balance disintegrates without replication. Figure 16, on the other hand, shows that replication enormously improving the worst load on any level. For example, with a tight hot spot of three nodes, the worst load for $k = 0$ is approximately 60,000, as opposed to 800 for $k = 2$. Note that load imbalance *decreases* with decreasing size of the hot spot, approaching the 10–1 load imbalance of random queries as the hot spot size increases. This is because caching alleviates the stress on nodes near the hot spot, and replicas alleviate the strain on the destination nodes themselves.

5.4 Irregular Trees

The synthetic data structures used in the rest of this study consist of balanced binary trees. We have experimented with trees in which the node degree is distributed exponentially between 0–31, with the values of mean degree ranging from $2 - 8$.. This is shown in Figure 17; each value of a is a different average degree for the namespace tree. We only consider caches sizes up to 75 in order to show more detail at smaller cache sizes.

Our results show the tree irregularity does not hurt latencies. In fact, as the average degree increases, the path lengths in the tree decrease, and the average query latencies also decrease (e.g. compare the "a=2" lines with Figure 6).

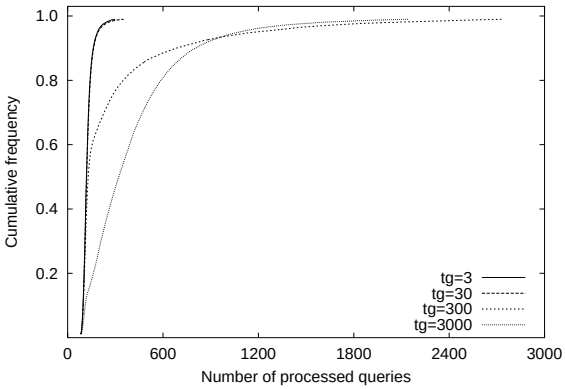

Figure 16: Load balance versus size of "hot spot"; replication factor $k = 2$.

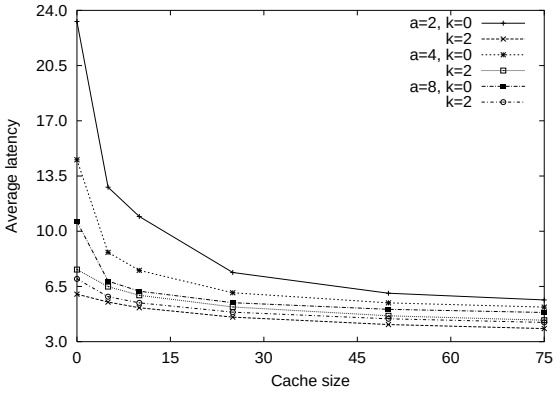

Figure 17: Query latency for random trees; a is the average degree.

For a random query stream, the balanced binary trees considered before represent rather demanding load imbalance scenario. This is because in such a tree, exponentially many paths may require the traversal of an exponentially small number of nodes. Our results show that, as expected, the load balance improves using random trees.

5.5 Multiple Nodes Per Server

We have so far assumed a single node per server. This is probably unrealistic, as individual servers will likely have multiple items to export for the majority of applications we envision (digital libraries, auctions, file systems). However, the number of exported nodes does not have a first-order effect on the amount of required network communication. Our routing algorithm routes queries to *servers* that host nodes closer to the destination, not to nodes. So, even in a worst case where the query has to be routed along the tree structure, network communication only occurs when a query is passed between nodes on distinct servers.

5.5.1 Replica Selection

We have so far assumed that replicas are randomly assigned among all other servers in the system. This does not reflect the temporal aspect of tree-building, i.e. replica assignment for the nodes at the top of the tree will be made when there are few servers in the system. Hence, initial replica assignments made in this manner will tend to be biased towards the top of the tree. During our experiments, we observed load imbalance up to eight times higher with initial replica placements than for corresponding runs with uniform replica assignment.

We rely on reassignment of replicas after startup in order to approach a random replica distribution. This can be accomplished in a number of ways, most easily by local exchanges between two servers interacting for other purposes (i.e. a query). Note that the set of replicas for a given node eventually need to be made consistent (i.e. the owner should be informed), but there is no requirement that this be immediate.

6 Related Work

6.1 Distributed Directories

DNS DNS [29, 30] can be considered to be one of the cornerstones of the Internet and is possibly the most widely deployed directory service, ever. The design of DNS was a careful compromise

between functionality, scalability, and implementability [31] — years of experience with DNS has shown that the designers almost all of the original design choices were the correct choices. If it were possible to the customize DNS infrastructure for arbitrary applications, the base TerraDir protocols would not be required. However, current DNS servers are required for their name resolution functions and obviously, the deployed infrastructure is not provisioned to handle queries for arbitrary RRs from arbitrary applications. In fact, it has proven somewhat difficult to upgrade the existing DNS infrastructure to support even Secure DNS [15].

A much more feasible alternate approach would be to use the DNS *protocols* and overlay application-specific directories on the Internet (much like a TerraDir). However, since DNS was designed for a specific application (i.e. serve domain names for network hosts), unlike TerraDir, DNS is not flexible enough to be used by arbitrary applications since it does not provide for configurable consistency, fault tolerance, or sophisticated searches.

X.500, DAP, and LDAP X.500 [6, 5] is an international distributed directory standard jointly developed by the ISO and CCITT (now ITU/T). The X.500 *Standard* was developed over a period of ten years and includes hierarchical name space, a class-based hierarchical object model, a hierarchical attribute model, and an administrative information model. X.500 allows symlinks (called *aliases*); node attributes in X.500 are expressed in ASN.1 [7]. X.500 also defines a family of internal protocols including DAP (Directory Access Protocol [4]) used to query a directory, DSP (Directory System Protocol) used to cooperative answer a query, DOP (Directory Operational Binding Management Protocol) used to set up long-term peering relationships between X.500 peers, and DISP (Directory Information Shadowing Protocol) used to replicate directory nodes. X.500 also defines an internal user authentication protocol and a master-slave consistency model for replicated data.

We should distinguish between the global X.500 directory which requires institutions to register their directory node name with ITU/T and the X.500 directory protocol. Obviously, the former is not useful in the context of arbitrary applications that want to start directories on a per-application basis. In the rest of this section, we explain why X.500 protocols are not appropriate for our purposes.

X.500 is a protocol well-suited for federated international directories for large organizations, but imposes far too much overhead to be useful in our context. TerraDir was designed to work across a much wider range of connectivity, thus inter-server links in TerraDir are implicit and failures are detected only when a server that owns node cannot be accessed during a query. This model of extremely relaxed connectivity requirements in TerraDir is directly counter to the requirements of X.500 which is designed for long-term (on the timescale of years) inter-server relationships that are negotiated using the Directory Operational Binding Management Protocol. Even DAP, the basic access protocol was so resource intensive that the *defacto* standard for accessing X.500 is LDAP (the Lightweight Directory Access Protocol [50]). LDAP does not incur the presentation and session layer overheads of DAP.

In designing TerraDir, we made a conscious choice to *decouple the directory protocol from application-level issues*, *viz.* the object schema, the level of reliability, security, and the consistency model. In retrospect, all general directory protocols should allow applications to specify these properties — this is a direct corollary of the end-to-end arguments [41] that have proven successful in the design of the Internet itself. Unfortunately, X.500 does not heed the basic end-to-end design principle, and as a result, it is not flexible enough to be useful for arbitrary applications.

6.2 Lookup Strategies and Query Routing

A number of interesting new algorithms [1, 44, 40, 21, 38] 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 [1] and thus do not scale due to their high resource usage. A particularly elegant algorithm is described in [44]: it is based upon the idea of consistent hashing [28], and provides both excellent load balancing and bounded worst case access latencies. The CAN scheme [38] 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.

Both Chord [44] and CAN [38] 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 [52], the lookup service used in the Oceanstore [27] global file systems, and PASTRY [39], the lookup service used in the PAST [40] global file system, both use virtualized hierarchies similar to Plaxton trees [37]. 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 (exporting) server. Such a design will lead to lower access latencies for correlated accesses (e.g. 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.

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. 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. Section 5.3 presents results when queries have *global* locality.

6.3 View Materialization and Naming

Our extended protocol can be seen as an extension and application of view materialization techniques from databases [22], but actually has more in common with file system work that materializes query results as addressable directories [19, 20]. Our work differs in both its context and functionality. For the former, TerraDir is a wide-area, extremely distributed, and decentralized directory service rather than a local-area file system.. Views are materialized by caching state throughout the system rather than at a single location. For the latter, TerraDir supports queries that are either orderings of per-node attributes, or of tags generated from node data by arbitrary functions.

View materialization gives TerraDir the ability to name objects and resources by attribute, a functionality provided in some form or another in Intentional Naming [8], the Information Bus [34], and T-Spaces [48], as well as several popular peer-to-peer applications [1, 14]. TerraDir differs from these systems in its flexibility and decentralization. INS also allows *anycasting*, i.e. indirect addressing by attributes. TerraDir is not a transport-layer service, but could easily be extended in this direction.

TerraDir shares with Active Names [46] the ability to name resources using arbitrary functions. However, Active Names does not provide a directory service.

References

[1] Gnutella home page. http://gnutella.wego.com/.

[2] Nupedia home page. http://http://www.nupedia.org.

[3] Wikipedia home page. http://www.wikipedia.com.

[4] The Directory: Abstract Service Definition. CCITT Recommendation X.511, ISO/IEC JTC 1/SC21; International Standard 9594-3, 1988.

[5] The Directory: Models. CCITT Recommendation X.501 ISO/IEC JTC 1/SC21; International Standard 9594-2, 1988.

[6] The Directory: Overview of Concepts, Models and Service. CCITT Recommendation X.500., 1988.

[7] Specification of Abstract Syntax Notation One (ASN.1). CCITT Recommendation X.208, 1988.

[8] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Symposium on Operating Systems Principles*, pages 186–201, 1999.

[9] Samrat Bhattacharjee, Mostafa Ammar, Ellen Zegura, Viren Shah, and Zongming Fei. Application-Layer Anycasting. In *Proceedings of INFOCOM 97*, 1997.

[10] Samrat Bhattacharjee, Ken Calvert, and Ellen Zegura. Self-organizing wide-area network caches. In *IEEE Infocom'98*, 1998.

[11] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[12] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.

[13] Robert L. Carter and Mark E. Crovella. Server selection using dynamic path characterization in wide-area networks. In *Proceedings of INFOCOM*, Kobe, Japan, April 1997.

[14] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system, 2000.

[15] D Eastlake and C. Kaufman. Domain Name System Security Extensions. RFC 2065, January 1997.

[16] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *Computer Communications Review (Proceedings of SIGOCOMM'98)*, 28(4):254–265, September 1998.

[17] Zongming Fei, Samrat Bhattacharjee, Ellen W. Zegura, and Mostafa Ammar. Finding the Best Server within the Application-Layer Anycasting Architecture. In *In Proceedings of INFOCOM' 98*, 1998.

[18] Paul Francis, Sugih Jamin, Vern Paxson, Lixia Zhang, Daniel Gryniewicz, and Yixin Jin. An architecture for a global internet host distance estimation service. In *Proceedings of INFOCOM*, New York, March 1999.

[19] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole. Semantic file systems, 1991.

[20] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Operating Systems Design and Implementation (OSDI)*, 1999.

[21] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for internet service construction. In *In Symposium on Operating Systems Design and Implementation*, 2000.

[22] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *Proc. of the International Conf. on Management of Data*, pages 157–166, 1993.

[23] S. Hotz. Routing information organization to support scalable interdomain routing with heterogeneouspath requirements, 1996.

[24] Sugih Jamin, Cheng Jin, Yixin Jin, Danny Raz, Yuval Shavitt, and Lixia Zhang. On the placement of internet instrumentation. In *Proceedings of INFOCOM*, Tel Aviv, Israel, March 2000.

[25] D. Katabi and J. Wroclawski. A framework for global ip-anycast (GIA). Internet Draft, Internet Engineering Task Force, June 1999. Work in progress.

[26] D. Katabi and J. Wroclawski. A framework for global ip-anycast (GIA). In *Proceedings of SIGCOMM*, 2000.

[27] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, 2000.

[28] D. Lewin. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, MIT, 1998.

[29] P. V. Mockapetris. Domain names - concepts and facilities. Request for Comments 1034, Internet Engineering Task Force, November 1987.

[30] P. V. Mockapetris. Domain names - implementation and specification. Request for Comments 1035, Internet Engineering Task Force, November 1987.

[31] Paul V. Mockapetris and Kevin J. Dunlap. Development of the Domain Name System. In *Proceedings of SIGCOMM*, pages 123–133, Stanford, California, August 1988. ACM. also in *Computer Communication Review* 18 (4), Aug. 1988.

[32] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 15–17 1997.

[33] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9), sep 1994.

[34] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus — An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, 1993.

[35] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[36] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[37] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[38] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *In Proceedings of the ACM SIGCOMM 2001 Technical Conference*, 2001.

[39] Antony Rowstran and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.

[40] Antony Rowstran and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[41] H. Saltzer, D. P. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computing Systems*, 2(4), 1984.

[42] Richard Schlichting and Fred Schneider. Fail-Stop Processors: an Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[43] Christopher Small and Margo I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.

[44] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[45] Wolfgang Theilmann and Kurt Rothermel. Dynamic distance maps of the internet. In *Proceedings of INFOCOM*, Tel Aviv, Israel, March 2000.

[46] Amin Vahdat, Michael Dahlin, Thomas Anderson, and Amit Aggarwal. Active names: Flexible location and transport of wide-area resources. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[47] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[48] P. Wyckoff, S. McLaughry, T. Lehman, D. Ford, and T. Spaces. Ibm systems journal, 1998.

[49] Koichi Yano and Steven McCanne. The Breadcrumb Forwarding Service: A Synthesis of PGM and EXPRESS to Improve and Simplify Global IP Multicast. *ACM Computer Communication Review*, 30(2), 2000.

[50] W. Yeong, T. Howes, and S. Kille. X.500 Lightweight Directory Access Protocol. Network Working Group RFC 1487, ISODE Consortium, 1993.

[51] Ellen W. Zegura, Mostafa Ammar, Zongming Fei, and Samrat Bhattacharjee. Application-Layer Anycasting: A Server Selection Architecture and Use in Replicated Web Service. IEEE/ACM Transactions on Networking, 2000.

[52] B. Zhao, K. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California at Berkeley Technical Report, 2001.

Appendix 1: TerraDir Routing Algorithm

$/$	the root node
q	queried node
$O(n)$	owner of node n and servers that replicate n
$R(s)$	servers that replicate nodes owned by s
$H(s)$	nodes owned and/or replicated at s
$Par(n)$	parent of node n
$Chil(n)$	children of node n
$C(s)$	cached list for server s

Table 1: Notation for algorithm description

```
function P( server s ) { permanent nodes of a server }
  Perm ← {/}
  ∀ n ∈ H(s)
    Perm ← Perm ∪ Par(n) ∪ Chil(n)
  return Perm
end P

function H( servers S ) { hosted nodes for a list of servers }
  Host ← ⊥
  ∀ s ∈ S
    Host ← Host ∪ H(s)
  return Host
end H

function O( nodes N ) { owners for a list of nodes }
  Owners ← ⊥
  ∀ n ∈ N
    Owners ← Owners ∪ O(n)
  return Owners
end O

function D( server s ) { list of digests for a server }
  return H(O(H(s) ∪ P(s) ∪ C(s)))
end D

function A( node n ) { ancestors of a node }
  if s = /
    return ⊥
  return Par(n) ∪ A(Par(n))
end A

function A( nodes N ) { ancestors for a list of nodes }
  Anc ← ⊥
  ∀ n ∈ N
    Anc ← Anc ∪ A(n)
  return Anc
end A

function knows_about( server s, node n ) {  server knows about node? }
  if n ∈ H(s) ∪ P(s) ∪ C(s)
    return yes
  else
    if n ∈ D(s)
      return yes
  return no
end knows_about

function Up( servers S ) { servers still running from list }
  Sup ← ⊥
  ∀ s ∈ S
    if s not failed
      Sup ← Sup ∪ {s}
  return Sup
end Up

function Not_visited( servers S ) { servers not visited yet from list }
  Snv ← ⊥
  ∀ s ∈ S
    if s not visited yet
      Snv ← Snv ∪ {s}
  return Snv
end Not_visited
```

```
function routing( server s, node q ) { the routing routine, called on behalf of the current server }
  if ¬ knows_about (s, q) {no direct hit }
    let n ∈ P(s) ∪ C(s) ∪ (A(P(s) ∪ C(s) ∪ {q}) ⋂ D(s))
      be the closest node to q
        such that Up(Not_visited(O(n))) ≠ ⊥

    if n ∈ P(s) ∪ C(s) { next hop is cached or known }
      forward query to random server from Up(Not_visited(O(n)))
    else
      if n ∈ A(P(s) ∪ C(s) ∪ {q}) ⋂ D(s) { hit in a digest }
        forward query to owner of corresponding digest
      else { n is ⊥ }
        if Up(Not_visited(R(s))) ≠ ⊥ { a replica of s is available }
          forward query to random server from Up(Not_visited(R(s)))
        else
          if not_visited(O(/)) ≠ ⊥ { try to send to root }
            forward query to random server from Not_visited(O(/))
          else
            return Fail
  if s ∉ O(q)
    if Up(O(q)) ≠ ⊥
      forward query to random server from Up(O(q));
    else
      return Fail
  else
    return data to query source
end routing

{ entry point: routing( query_source, q ) }
```