Persistence of Data in a Dynamic Network

Hakim Weatherspoon and Rachel Rubin
University of California, Berkeley
{hweather, rrubin}@cs.berkeley.edu

Abstract

We explore the possibility of achieving robust and efficient peer-to-peer storage by differentiating between reliable and unreliable nodes. Central to our technique is the use of a distributed directory (DD) to provide a level of indirection, thereby permitting the system to exploit heterogeneity. We compare and contrast the DD approach to a distributed hash table (DHT) when constructing reliable storage systems from components of varying reliability. By comparing these techniques, system architects can design systems that are more efficient and scalable. For instance, we sketch a configuration that is durable, available, and scalable with a Gnutella-like distribution of node reliability. In addition, we describe a data structure and interface that lends itself to systems that exhibit churn. Our results, show that a DD uses up to three orders of magnitude less bandwidth per node than a DHT to maintain data. We conclude with a discussion of an implemented DD based solution.

1 Introduction

Maintaining state in a structured P2P overlay network using minimal resources is a non-trivial problem (e.g., utilizing less than 15% of the bandwidth capacity of each node). While such overlays have been shown to handle dynamic node membership in both theory and practice [11, 12, 14, 17, 20, 27], the bandwidth required to do so can be quite expensive. Several proposed storage applications [3, 6] leverage the fact that structured P2P overlays handle churn efficiently and gracefully by using the overlay maintenance mechanisms directly (i.e., the overlay remains connected under constant or transient change [13]). However, Blake and Rodrigues [2] showed that tightly coupling the storage and networking in this manner is prohibitively expensive. The bandwidth required to maintain data availability and durability in a dynamic network exceeds the bandwidth capacity, given that the disks of the participants are well-utilized. In other words, participants can only contribute an insignificant amount of disk space to keep the maintenance bandwidth below link capacities.

The fundamental problem with combining networking and storage is that the two tasks require separate policies for efficient use and maintenance. In this paper, we show that by adding a level of indirection at the storage level, the separate layers can be maintained efficiently with different policies using similar mechanisms. After comparing coupled and decoupled storage and networking, we show an architecture and tradeoffs for implementing and maintaining a level of indirection. Finally, when designing distributed storage solutions, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is known to be impossible to achieve all three [7]. We show how to achieve availability and partitioning tolerance by relaxing the consistency on data updates. In particular, we motivate the infrastructure supporting disconnected operations by allowing applications that detect conflicts to branch.

Branching permits conflicting objects to be durable while allowing the application to resolve conflicts out-of-band. In particular, we discuss macrobranching. Macrobranching allows the application to branch when a conflict is detected. The advantages of macrobranching are that each branch is individually maintained, located, and allow user-specified policies to create and remerge branches. The disadvantage of macrobranching is that multiobject consistency between branches becomes difficult. We do not discuss different branching/merging policies; instead, motivate the need for infrastructure branch-
ing mechanism support that allow users to control branching policy.

The contribution of this paper is quantifying the cost and benefit of a layer of indirection and analyzing the tradeoffs, an analysis not yet seen in the body of P2P literature. We show performance comparisons and tradeoffs between coupled and decoupled storage designs. Given the tradeoffs, we show an architecture and implementation of a level of indirection. Finally, we show a data structure that prefers availability and network partitions over data consistency.

Section 2 begins by comparing the costs and benefits between distributed hash tables (DHTs), in which networking and storage are united, and distributed directories (DDs), in which they are separated. After describing our analytical model and data structure in Sections 3 and 4, we provide a quantitative evaluation and comparison of the two solutions in Section 5. Section 6 and 7 discuss the architecture and implementation of a DD in more detail. We present preliminary results in section 8. We discuss these results, future work and their implications in Sections 9 and 10.

2 Design Comparisons

Storage solutions, such as CFS [3] and PAST [6], have proposed using DHT mechanisms and policies directly, and as a result are both simple and elegant. However, such solutions assume servers have independent and identically distributed availability and failure distributions, use replication or other forms of redundancy to compensate for server faults and failures, and argue that random placement of replicas are sufficient to maintain desired levels of data availability. Unfortunately, the bandwidth required to maintain the redundancy is the bottleneck [2]. Furthermore, selective placement of data may be desirable for some systems [10, 23]. In this section we compare and contrast the design points for DHT-based and DD-based storage solutions.

2.1 Distributed Hash Tables

The distributed hash table (DHT) abstraction [4] maps a given key in the node/object identifier space to a particular node in the overlay. An application can insert data into the DHT, which places that data on the node responsible for the key of the data; the data is retrieved by using the DHT to find the responsible node and asking it for the data. The DHT maintains this mapping between key and node consistently, even as nodes join and leave the network.

**DHT Pros:** The DHT abstraction is an elegant solution for maintaining the storage layer because the mapping function of the DHT inherently decides who owns the mapping of a key to an object, when to remap that responsibility, and where the responsible node is located. This shields the storage layer from making similar decisions.

**DHT Cons:** The problem with using the DHT abstraction directly for the storage layer is that it provides no control over data placement, causing maintenance bandwidth to become prohibitively expensive under churn. Tightly coupling the storage layer with the networking layer wastes bandwidth by automatically transferring data each time the key for that data is remapped; this transfer may take place over the wide area to a node that may only live for a short period of time. It also may remap responsibility for data away from a node experiencing a short transient failure, potentially wasting wide area bandwidth.

2.2 Distributed Directories

A distributed directory (DD) decouples the storage layer from the underlying transient network. At its core, a DD is a level of indirection—utilizing pointers within the network to achieve flexibility in replica placement. Since DD functionality does not require locality, it can be provided by any DHT that can reliably store small pointers to objects. DD functionality is also provided by Decentralized Object Location and Routing (DOLR) layers [4]. The DD approach to storing data is a hybrid technique: it utilizes the peer-to-peer system to maintain pointers and more sophisticated methods to maintain data.

**DD Pros:** Decoupling the storage layer from the networking layer insulates the data from the transient
network, saving wide area bandwidth. Also, the separation allows the placement of data to be biased towards more available and reliable nodes, increasing data availability and durability. Finally, since a DHT still identifies a root node responsible for the pointer of each object, the repair of an object can be triggered after a specified threshold has been reached, further insulating the data layer.

**DD Cons:** The extra level of indirection must be maintained. The pointers need to be replicated to prevent memory leaks (i.e. a live node cannot locate data that exists). Another issue is outdated pointers; namely, dangling pointers that no longer point to the correct location for data. Additionally, system designers need to be careful that the aggregate storage due to pointers does not increase the maintenance bandwidth; that is, is some small percentage of the total storage. Furthermore, the storage layer becomes more complex because it requires an independent data placement and replication mechanism.

The remainder of this paper analyzes and compares the cost of DHT and DD storage systems. By understanding the tradeoffs, system architects can design systems that are more efficient and scalable.

### 3 Data Model

In this paper, we desire to provide data availability in the face of churn and network partitions. Given persistent data in the mist of churn, branching is a natural conflict-resolution mechanism to allow users to update data while being disconnected (i.e. a partition of one node). For example, a mobile user may want to access and modify data at their leisure. Providing branching allow users to modify data while disconnected from the system[19]. An application can hoard objects from the system, allowing the user to continue to modify data even when the user is offline. When the user reconnects, the data is reintegrated as a separate branch before merging allowing the user to later merge the conflicting changes.

We base our branching solution on the following data model. Data objects are linearly ordered sequence of versions, where each version is a read-only sequence of bytes. New versions may be added to the end of the version sequence of a given data object through update operations, each of which generates a new version. Data may be read from a specific version through read operations. Each version is a stand-alone entity and is abstractly unrelated to any previous versions. For concreteness, a data object might be a file, a directory, or a database record. Data objects may also contain the names of other data objects.

We assume the ability to generate globally-unique identifiers (GUIDs). Data objects are uniquely specified by active GUIDs (A-GUIDs). Within a data object, each version is specified by a locally-unique version-ID and a globally-unique version GUID (V-GUID). Versions are immutable and provide for time-travel [21], the ability to reconstruct the view of a data object as it appeared at any time during its lifetime.

When multiple updates are simultaneously submitted, some entity in the network must be a point of serialization to provide atomicity. This serializer may be the users workstation or another node in the network contracted to commit data for the user. This serializer takes each update, verifies that the client which submitted the update has update privileges\(^1\), atomically applies it to the data object, then generates a new V-GUID. Consequently, when a client seeks the most recent version of a data object, a request is sent to the serializer to obtain the V-GUID of this latest version. More generally, the system provides a mapping such that, given an A-GUID and some version information (for instance, a timestamp), the GUID of a particular version can be retrieved. The data object described above is maintained over time by the system (DHT or DD) as described in Section 2.

#### 3.1 Data Object Interface

To generate a new data object, a user must specify a human-readable name for the data object, the user’s identity as a public key, and public/private key pair to encrypt and sign the data:

\[
\text{create}(\text{name}, \text{identity}, \text{keys}) \Rightarrow \text{A-GUID}
\]

\(^1\)The details of access control are beyond the scope of this paper.
An empty first version is produced as well. We assume that users keep the A-GUID of one “root” data object with them at all times. This can be used to construct an arbitrary, hierarchical name space to store mappings between user-relative names and their associated A-GUIDs.

Next, we provide a query operation to acquire specific version information from a given archive:

\[
\text{query(A-GUID, Spec)} \Rightarrow \text{V-GUID}
\]

This takes a specifier for a version (which may be a timestamp, version-ID, or other means of identifying a version) and returns an appropriate V-GUID. If the specifier is \textit{nil}, the \text{query} returns the V-GUID current as of the time the request reaches the serializer.

To read data, we assume that a client provides a V-GUID and a specification about which data to read:

\[
\text{read(V-GUID, offset, length)} \Rightarrow \text{data}
\]

This operation returns data from the specified version.

Finally, new versions can be either unique or derived from previous versions. Each write() operation generates a new version of a data object, returning a V-GUID in the process.

\[
\text{write(V-GUID, offset, data)} \Rightarrow \text{new V-GUID or nil}
\]

write() provides the ability to derive a new version from a previous version. The user has the option to branch if the V-GUID is not the latest for the data object at the time this operation is applied. If a conflict occurs, the user application has the ability to create a branch from the main trunk. It is up to the user to place additional semantics on top of branches if they occur.

### 3.2 Branching

**Creating** In macrobranching, branches are maintained and accessed as separate objects from the main trunk. A new branch is created when the client or application requests a new branch. A createBranch event is issued. The createBranch takes the AGUID/VGUID pair of the data object that the new branch is based. The new data object records, in the metadata, that it is a branch based on the AGUID/VGUID pair. Figure 1 shows an example of a series updates over time, some of which create a branch.

![Figure 1: Macro Data Branching](image)
Updates and Reading  All updates are made as specified by the above interface, since the branch, once it is created, is a unique object, operations such as updates and reads require no modification to the interface.

Close  When the branch is closed, there is a copy-on-write to the original data object. There is a reference to the end of the branch placed in the metadata. The close indicates that the branch is no longer writable.

4 Storage Maintenance Model

Another goal of this paper is to explore different configurations that allow the storage layer to be robust and bandwidth-efficient. That is, we desire to maintain the data, as represented by the data model in Section 3, efficiently. By robust, we mean that data is durable and available with high probability. The storage layer also needs to be bandwidth-efficient in maintaining storage. Finally, the system needs to recognize and use new resources efficiently, as well as compensate for absent resources. This section describes the model we use to quantify these properties.

4.1 Background

The data maintenance bandwidth per node is approximated by the following equation derived by Blake and Rodrigues [2]:

\[
\frac{BW}{N} = \frac{2kD}{Na} \cdot \frac{1}{T}
\]  

(1)

Where \( D \) is the amount of unique data in the system and \( k \) is the redundancy factor required to achieve a desired level of data availability. \( T \) is the average lifetime of a node in the system, and \( N \) is the total number of nodes. The number of nodes available to store data is reduced by the average node availability, \( a \). Given a target data availability (i.e. \( 1 - \epsilon \), for some small \( \epsilon \)), Blake and Rodrigues derive equations for the redundancy factor needed for replication and coding [1, 22] schemes (\( k_r \) and \( k_p \) respectively).

All values and comparisons throughout the rest of the paper will be based on extending Equation 1 to account for different data allocation and heterogeneous availability and lifetime.

<table>
<thead>
<tr>
<th>Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>model(1)</td>
<td>DHT spreads data evenly over all nodes</td>
</tr>
<tr>
<td>model(2)</td>
<td>DD spreads data over reliable nodes only and pointers over all nodes</td>
</tr>
<tr>
<td>model(2.a)</td>
<td>Reliable model(2) nodes. Stores all data and stores ( \frac{NR}{N} ) fraction of pointers.</td>
</tr>
<tr>
<td>model(2.b)</td>
<td>Unreliable model(2) nodes. Stores ( \frac{NU}{N} ) fraction of pointers (and no data).</td>
</tr>
<tr>
<td>model(3)</td>
<td>DD spreads data and pointers over reliable nodes only. (unreliable nodes are not used)</td>
</tr>
<tr>
<td>model(3.a)</td>
<td>Reliable model(3) nodes. Stores all data and pointers. (Similar to a DHT of reliable nodes only).</td>
</tr>
<tr>
<td>model(3.b)</td>
<td>Unreliable model(3) nodes. Not used (i.e. stores no data or pointers), ( \frac{NU}{NR} = 0 )</td>
</tr>
</tbody>
</table>

Table 1: Storage Models. Defines models used to compare different storage solutions, using various combinations of storing pointers and data on (un)reliable nodes.

4.2 Parameter Setting

We use three basic models to compare DHT and DD based storage solutions. The first model uses a DHT to distribute data evenly over all nodes. The second model uses a DD to distribute indirection pointers evenly over all nodes and distributes the data only over the reliable nodes. The third model uses a DD to distribute the pointers and data only over the reliable nodes (similar to a DHT of only reliable nodes); unreliable nodes are not used for storage. We refer to the models throughout the rest of the paper as model(1), (2), and (3), respectively. Table 1 summarizes these models. The equations given for the data maintenance bandwidth per node, given \( N_R \) reliable nodes and \( N_U \) unreliable nodes (where \( N_R + N_U = N \)) are as follows:

Model (1)

\[
\frac{BW}{N} = \frac{2kD}{Na} \left( \frac{1}{aT} \right)_N
\]  

(2)

Model (2)

\[
\frac{BW}{N_R} = \frac{2k_r (D + k_p \cdot P \frac{NR}{N})}{N_R} \cdot \left( \frac{1}{aRTr} \right)_{NR}
\]  

(3)

\[
\frac{BW}{N_U} = \frac{2k_p \cdot k_p \cdot P \frac{NU}{N}}{NU} \cdot \left( \frac{1}{aUTU} \right)_{NU}
\]  

(4)

Model (3)

\[
\frac{BW}{N_R} = \frac{2k_r (D + k_p \cdot P)}{N_R} \cdot \left( \frac{1}{aRTr} \right)_{NR}
\]  

(5)

The term \( \left( \frac{1}{aT} \right)_N \) denotes the average of the inverse combination of availability and lifetime, for the set of \( N \) nodes. \( P \) is the amount of unique pointers in the system, corresponding to the amount of unique data \( D \). The total storage for the DHT (model(1)) is \( k_r D \), but for the DD it is \( k_r (D + k_p P) \) (the pointers
are replicated $k_p$ times to prevent memory leaks). We defined the redundancy factor in terms of the replication factor $k_r$: for coding, we replace the replication factor $k_r$ with the expansion factor $k_e$. Note that in the case of coding $k_e k_p P \Rightarrow k_p m P$; that is for each unique object, $k_e m$ unique fragments are produced, and a pointer is needed for each fragment. The result is that pointers could cost more to store than the data itself, unless the ratio of data to pointers is greater than $k_p$ (i.e. $k_e \frac{D}{k_p P} = \frac{D}{\frac{P}{k_p}} > k_p$ for replication and $k_e \frac{D}{k_p P} = \frac{D}{\frac{P}{k_p}} > k_p m$ for coding). The changes are summarized below:

$$
\left( \frac{1}{a_i T_i} \right)_N = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{a_i T_i}
$$

$kD \Rightarrow k_r (D + k_p P)$ (for DD replication)

$kD \Rightarrow k_e (D + k_p m P)$ (for DD coding)

We used the Overnet trace [1] to form a basis for comparing for models (1), (2), and (3). Table 2 summarizes the values we use for the comparisons.

### Table 2: Baseline Trace and Comparison Parameters.

The parameters $N$ ($N_R$ and $N_U$) and $a$ ($a_R$ and $a_U$) were extracted from the trace conducted by Bhagwan et al [1]. The rest of the parameters were derived from $N$ and $a$, or chosen to be a “reasonable” setting.

<table>
<thead>
<tr>
<th>Param</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$</td>
<td>1TB</td>
<td>unique data</td>
</tr>
<tr>
<td>$N$</td>
<td>1409</td>
<td>total nodes</td>
</tr>
<tr>
<td>$N_R$</td>
<td>136</td>
<td>nodes with $\geq 0.7$ availability (9.26% of $N$)</td>
</tr>
<tr>
<td>$N_U$</td>
<td>1333</td>
<td>nodes with $&lt; 0.7$ availability (90.74% of $N$)</td>
</tr>
<tr>
<td>$a$</td>
<td>0.3</td>
<td>average availability of $N$</td>
</tr>
<tr>
<td>$a_R$</td>
<td>0.8</td>
<td>average availability of $N_R$</td>
</tr>
<tr>
<td>$a_U$</td>
<td>0.2</td>
<td>average availability of $N_U$</td>
</tr>
<tr>
<td>$\max T_R$</td>
<td>256</td>
<td>max lifetime (in days) for reliable nodes.</td>
</tr>
<tr>
<td>$\max T_U$</td>
<td>1</td>
<td>max lifetime (in days) for unreliable nodes.</td>
</tr>
<tr>
<td>$1 - \epsilon$</td>
<td>6 9s</td>
<td>data availability</td>
</tr>
<tr>
<td>$k_r$</td>
<td>10</td>
<td>replication factor to achieve data availability</td>
</tr>
<tr>
<td>$k_c$</td>
<td>2</td>
<td>coding factor to achieve data availability</td>
</tr>
<tr>
<td>$k_p$</td>
<td>10</td>
<td>pointer replication factor to avoid memory leaks</td>
</tr>
<tr>
<td>$m (= b)$</td>
<td>16</td>
<td>fragments required for reconstruct under coding</td>
</tr>
</tbody>
</table>

5.1 Varying Lifetime, $T$

In Figure 2.a we hold $T_U$ constant at 1 day, and vary $T_R$ from 1 to 256 days. For model(1) (the standard DHT), the bandwidth used to maintain data availability in the face of unreliable nodes dominates the bandwidth cost and holds it steady at 32Mbps. For model(2), the reliable nodes bandwidth decreases as $T_R$ is increased, but the unreliable nodes bandwidth stays constant at 4Mbps because pointers are being maintained across all nodes. For model(3), the reliable nodes maintenance bandwidth decreases as the lifetime increases. Since the unreliable nodes are not being used for data storage, model(3) is able to achieve a low per-node maintenance bandwidth of 64Kbps when the lifetime is 8 months.

For Figure 2.b we vary reliable node lifetimes along the $x$-axis and vary the unreliable node lifetimes at a ratio of eight times less than the reliable nodes lifetime. The per-node maintenance bandwidth for all models decreases as lifetime increases; the models vary only by a constant difference in bandwidth. This constant difference is due to the constant difference in the combination of node availability, lifetime and the ratio of data to pointers.

For Figure 2.c we use a distribution for lifetimes among individual nodes. We vary lifetime in a similar manner conjectured by Gummadi et al [9]; that is, a node’s lifetime is proportional to its availability (i.e. $T_i = a_i T_{max}$). As before there is a constant difference between the different per-node bandwidth maintenance costs, but it is dramatically increased by the difference in lifetime.
For this comparison we consider only replication as we vary the data to pointer ratio in Figure 3.a. In model(1), the bandwidth is constant at 32Mbps, due to the maintenance of data placed on unreliable nodes. For model(2.a), the bandwidth of the reliable nodes is fairly constant at 64Kbps; however, it is higher than 64Kbps when $\frac{D}{P} \leq k_0 = 10$. For model(2.b), as we decrease the amount of pointers in the system (i.e. increase $\frac{D}{P}$), we decrease the amount of work each unreliable node is required to do. For model(3.a) reliable node maintenance is constantly low at 64Kbps (bandwidth per node is higher than 64Kbps when $\frac{D}{P} \leq 10$).

5.3 Replication vs. Coding, Varying $\frac{D}{P}$

In this section, we compare the maintenance bandwidth of replication versus coding. This analysis allows to evaluate if more efficient coding techniques can actually be leveraged, or if the increased complexity of the infrastructure cancels out the redundancy gains made by coding techniques.

We compare both replication and coding schemes by varying the data to pointer ratio Figure 3.b. For model(1), the bandwidth per node is decreased by using coding techniques. The bandwidth per node is still dominated by the unreliable nodes, but coding reduces bandwidth from 32Mbps to 8Mbps. For model(2.a) the reliable nodes bandwidth is fairly constant and reduced from 64Kbps to 16Kbps by using coding techniques. The interesting point is that maintaining fragments is actually more expen-
sive than maintaining replicas (in terms of bandwidth per node) when \( \frac{D}{P} \leq k_p mn = 10 \cdot 16 = 160 \). For model(2.b), the DHT is maintaining the pointers. The DHT does not discriminate between reliable and unreliable nodes, so the unreliable nodes dominate the maintenance cost. But as we decrease the amount of pointers in the system (i.e. increase \( \frac{D}{P} \)), we decrease the amount of work the DHT is required to do. Another interesting point is that maintaining coding pointers is always more expensive than maintaining replication pointers by a constant factor \( mn \) (i.e. \( k_p < k_p mn \)). For model(3.a) reliable node maintenance is constantly low at 16Kbps when \( \frac{D}{P} > 160 \).

5.4 Varying Number of Node, \( N \)

In Figure 4, we show that the system aggregate storage scales linearly as the number of nodes increase. That is, bandwidth per node characteristics is constant because the storage per node is constant.

5.5 Summary

DHTs do not take into account the suitability of a given peer for a specific task before explicitly or implicitly delegating that task to the peer[18]. In our analysis, differentiating among high- and low-availability nodes saves bandwidth. The savings increase as the gap widens, and if lifetimes are longer for highly available nodes (evident in Figures 2). Finally, storing pointers only works if the amount of data to pointer ratio is high (Figures 3).

By understanding the system dynamics and parameters, we can create a self-organizing option that is durable and available. One self-organizing option, is to use a DD. Figure 4 used the self-organizing mode to achieve six 9’s of data availability. The DD configuration used a DHT of all nodes for pointers and reliable nodes for data, used coding, had an average of large objects, the average lifetime of the reliable nodes was eight months, and the reliable nodes were designated as having availability greater than or equal to 70%. Another non-self-organizing option for robust and bandwidth-efficient storage is to explicitly designate two rings [10], one reliable ring used for storage another unreliable ring not used at all.

6 Distributed Directory

In this section, we describe a self-organizing Distributed Directory (DD) architecture and implementation. Recall that the difference between the DHT-based storage and DD-based storage is that the extra level of indirection must be maintained. In particular, We briefly revisit how the DD prevents memory leaks by using the DHT to maintain a redundant set of pointers. Next, we describe preventing dangling pointers using heartbeat mechanisms. We also introduce the term data static resilience. Data static resilience is the number of replica or fragment failures the object can withstand before the object is lost. Finally, we describe how to use the pointers for routing to objects. Additionally, an overall theme of DD-based storage solution is the increased complexity of the storage layer because they require independent data placement and replication mechanism. We show examples of data placement and replication schemes that work in a DD-based storage environment.

Memory Leak Prevention Memory Leaks are created when all pointers are lost. That is, a node can longer locate data that exists. Memory leaks can simply be prevented by allowing the DHT to store and maintain a redundant set of pointers. As Section 5.2 shows, the amount of pointer storage that the DHT maintains needs to be minimized. Additionally, the amount of redundancy may be reduced by considering that the pointers are soft-state and periodically republishing the location of data. The problem is that the republish rate may be very slow (e.g. order weeks or months) to keep the republish bandwidth at an acceptable level.
Dangling Pointer Prevention  Dangling Pointers are created when a set pointers are outdated and no longer point to the correct location for data. Dangling pointers can be prevented by the use of node (or host) heartbeats. The problem with host heartbeats is that each host maintains a local host database bounded by a maximum size of $O(n)$ host records. In particular, if the average number of objects per node is more than the total number of nodes, each node must maintain a local notion as to whether a destination is available; otherwise, a lot of dangling pointers would be created during node churn.

Host Heartbeats  Our proposed solution to the dangling pointer problem is host heartbeats. In addition, heartbeats are signed and contain capabilities (services that the node provides). The key issues are the rate of the heartbeats to keep the system efficient to maintain and still provide good liveness information. Our proposed solution to the storage layer deciding who, what, when, where, why, and how, is to use the host heartbeat database to bias data to be stored on nodes with higher availability and capabilities and to use triggers to decide when to repair data.

That is, a host database stores a local notion of the liveness/availability, lifetime, location, domain, capabilities, etc. of another node.

Similar to soft-state republishing, heartbeats have to be propagated such that the bandwidth does not exceed acceptable levels. The only way to keep the bandwidth at acceptable levels is to allow the local host information to be out-of-date. That is, allow the underlying P2P overlay to aggressively monitor routing table neighbors, while the host heartbeats are epidemically spread at a much slower rate. There are three well known epidemic algorithms, anti-entropy, gossiping (e.g. rumor-mongering), and direct mail [5]. Direct mail can be implemented as a form of multicast when the underlying P2P overlay has good convergence properties; routes initiated by different nodes destined for the same end node converge if the initial nodes are close (i.e. close in terms of proximity/locality/latency). For example, P2P overlays that select neighbors and routes based on proximity often exhibit convergence; proximity neighbor selection (PNS) and proximity routing selection (PRS), respectively [8]. For example, Tapestry[11], Pastry [17] and Chord that uses PNS+PRS exhibit convergence [8]. Heartbeats use the convergence property of the underlying P2P overlay to DirectMail (i.e. multicast) other nodes with an expanding radius. Figure 5 shows an example of an expanding host heartbeat that uses DirectMail. Notice that a heartbeat reaches all nodes in the system when the radius is $\log N$. When host heartbeats are stored in a local database, the resulting host database becomes a powerful tool for selecting servers for replica placement.

Now that the heartbeat mechanism has been discussed. The local host database that results is a great selection tool to select data placement nodes. For example, the following query can be run against the local node database

```
SELECT 64 distinct(node)
FROM hostDB
WHERE Probability(up) > 0.95 AND
      distinct(label/classification) AND
      distinct(OS) AND
      distinct(domain/AS) AND
      distinct(location) AND
      spread(time zone)
```

In this query, the client is requesting 64 different
nodes that have a high probability of being up, the intersection between their classification, OS, and location is empty (i.e. they are not correlated, have different operating systems, and are geographically spread), and their is an affinity to pick nodes in different time zones. The query result is simply a list of nodes that satisfies the query or an empty list if the query cannot be satisfied.

**Triggers** Another problem with DD-based storage is knowing when to repair data. By using the host database each root node has a local estimation of the number of replicas for each object that are available. Figure 6 shows an example of a root with two replica pointers. Recall that the DHT maintains pointers and the root is the node responsible for maintaining the mapping of pointer to location of data. When the number of replicas reaches an agreed threshold, the root node triggers a repair process. For example, the root node can send a message to the replica nodes indicating that replica redundancy is critical. An example query that uses the host and pointer database and triggers repair is below:

```
SELECT pointers
FROM p1 AS pointerDB,
    h AS hostDB
WHERE p1.objguid=objguid AND
    pl.threshold >
    (SELECT COUNT
    FROM p2 AS pointerDB,
        h AS hostDB
    WHERE p1.objguid=p2.objguid AND
        p2.src=h.hostguid AND
        h.state=UP)
```

The above query states that for a given object guid, select the pointers where the remaining number of replicas is less than threshold number of pointers to consider the object critical. That is, trigger repair.

An important property of replicas when using triggers is data static resilience: that is, the number of failures the redundant object can withstand before the object has to be repaired. A good generalization of data static resilience is $m$ of $n$ redundancy. $m$ of $n$ redundancy says that any $m$ out of $n$ replicas or fragments is sufficient to reconstruct the original object. Data static resilience in terms of $m$ of $n$ is $n-m$; that is, an object can withstand the quantity $(n-m)$ failures before data is lost. Data static resilience is important with a DD-based storage solution that uses triggers because the amount of time between the trigger and the actual repair, the repair time, needs to be greater than the time to lose $(n-m)$ replicas or fragments. Considering the churn rate, the average repair time needs to be less than the churn rate. For example, if $n = 64$, $m = 16$. threshold = 32, than the data static resilience is $(threshold - m)$ or 16. That is, the system needs to withstand 16 failures during the repair time. Notice that the data static resilience is reduced from $(n-m)$ to $(threshold - m) = 16$ because the repair time is the time between the trigger and the actual repair.

**Routing** It is possible to also use the indirection layer for routing. It still remains to be seen that the local host database can be kept up-to-date enough without exceeding bandwidth limits, such that most of the route to objects succeed. That is, the amount of dangling pointers is minimized. There are many options to develop a routing layer that interacts directly with pointers. We touch on some of them, but further research needs to be conducted. For example, the following query can be used to select live routes to objects:

```
SELECT pointer
FROM p AS pointerDB, h AS hostDB
WHERE p.objguid=h.hostguid
```
WHERE p.objguid=objguid AND p.src=h.hostguid AND h.state=UP

The above query states that given an object guid, select a pointer to a host that is storing the object and the host is up. The router would then use the returned pointers to forward the query to the storage node.

Do Pros Outweigh Cons? Very good question. The DHT solution is very attractive because of its simplicity, but it is also its simplicity that makes the solution unscalable (i.e. maintenance bandwidth becomes prohibitively expensive in dynamic networks[2] and cannot selectively place objects[10]). As an alternative, the DD solution can be made efficient but becomes more complex and harder to verify to do so. We have implemented all of the above mentioned DD-based mechanisms (redundancy, heartbeats, host database, direct mail, etc). We are currently evaluating the implementation.

7 Implementation

We implemented a prototype Distributed Directory (DD) in Java, using Tapestry [26] as the underlying DHT. The DD solution uses the DirectMail (i.e. multicast) to propagate heartbeats, uses the host database to perform queries for routing, triggers, and data placement. The code base is built upon Matt Welsh’s Staged Event-Driven Architecture (SEDA) [24], in which each distinct part of a system is implemented as a self-contained stage. Each stage has its own state and thread pool, and can communicate with other stages by sending events to a central dispatcher. This allowed us to implement our own stages dedicated to DD maintenance and run them along with existing Tapestry code. The inclusion of these stages is optional and unintrusive. Collectively, the stages and events in our implementation contain about 3000 semicolons.

As mentioned in Section 6, each node sends a host heartbeat to its neighbors with expanding radius; that is, nodes at a further radius receive heartbeats less often than nodes at a closer radius. In this experiment, heartbeats are received by a neighbor at radius-$i$ with a period $p_i = p_{i-1} * f$, where $p_1$ (the base period) and $f$ (the factor by which the periods grow exponentially) are configurable parameters. Note that when $f$ is 1, all neighbors in the network receive heartbeats at the same rate, regardless of their radius. A node is considered down if a heartbeat is not received.

We also implemented a branching mechanism in Java using Oceanstore[15] as the underlying architecture. We wrote a test application which opens an object, writes to the object, branches from the original, writes to both the object and the branch and reads both. We observed the following outputs

OSTestBranching-chimp:3642: Output String is foo
OSTestBranching-chimp:3642: Output String is foo branch
OSTestBranching-chimp:3642: Output String is foo original

8 Preliminary Results

This section describes the performance of our implementation in simulation. The results are preliminary. First we briefly explain our simulation environment, and then we go on to report and analyze results obtained from experiments run in this environment.

8.1 Simulation Environment

In order to evaluate the effectiveness of our implementation, we chose to run our experiments in a simulated environment. This allowed us to obtain results from experiments consisting of about a couple hundred nodes and spanning a period of fifteen minutes. The simulator we used was first described in [16]; we include a brief description here as well.

The simulator uses the transit-stub model of GT-ITM [25] to model a physical network as a graph of nodes. In this model, nodes can belong to one of two groups: stub nodes and transit nodes. Stub nodes are arranged in lightly-connected stub domains. Transit nodes exist in highly-connected transit domains, each of which connect to several stub domains. There are also several inter-stub domain edges. See Figure 7 for an example transit-stub graph.

Values $\alpha_{net}$ and $\beta_{net}$ are associated with each edge in the graph. $\alpha_{net}$ corresponds to the latency in sec-
Figure 8: Distributed Hash Table and Distributed Director under Churn. Routing to objects under churn, starting with 100 nodes. Churn uses a Poisson process with average inter-arrival time of 20 seconds and randomly kills nodes such that the average lifetime is two minutes.

Figure 7: A Transit-Stub Graph. This topology mimics the structure of large networks observed in nature. Shown also is an overlay network which minimizes the number of interdomain edge crossings. Such overlays allow the topology discovery properties of the source filtering algorithm to minimize interdomain bandwidth consumption. See Section 8.1 for details.

We generated graphs consisting of 5100 nodes each using GT-ITM, modifying only the bandwidth values as discussed above. For each simulation, we mapped every active Tapestry node onto one physical node in the graph, uniformly at random without replacement.

8.2 Simulation Experiment

We used a benchmark described in [27] to measure and compare the cost of a DHT-based and DD-based storage solution. The benchmark consists of every node publishing some number of objects at the beginning, and then having all nodes try routing to random objects that have been published. For this benchmark each object had four replicas maintained by the DHT or DD. Only the percentage of success route to objects are measured; that is, the queries that were routed to the storage node that actually did publish the object. Latency and bandwidth are measured as well.

To make the benchmark interesting, nodes continuously “churned”, joined and left the network at some specified rate. These network conditions demonstrate how the DHT or DD will maintain its performance under realistically (albeit controlled) dynamic membership. When nodes fail, they fail completely (i.e. failstop).
In the DHT case when a node fails, data is immediately copied to another node as described in Section 2.1. In the DD case, redundancy is not immediately refreshed when nodes die. Instead, only when enough nodes fail that the redundancy level becomes critical is the repair process triggered. That is, the remaining set of storage nodes are notified that redundancy is low. The remaining storage nodes than cooperate to refresh redundancy. Also, the DD is used to route to objects. Not that, all route to objects go to the root node and the object location requests do not benefit from locality in Tapestry. As a result, the mean latency is similar to the diameter of the network, or $\approx 300ms$. For the benchmark, churn started at five minutes into the test.

Figure 8.a and 8.b show the benchmark run with the DHT-based and DD-based storage, respectively. Figure 8.a shows that the DHT-based storage solution is able to route to all of the object most of the time. Figure 8.a provides a baseline for the DD-based storage solution in Figure 8.b. The DD-based storage solution had the base host heartbeat set to one minute. Since the network was size 100 and the base was 16 all nodes could be reached in two periods of the heartbeat (i.e. $\log_{16} 100 \approx 2$). Figure 8.b shows two periods, heartbeating neighbors radius-1 away, than radius-2. The problem with the experiment is that all nodes started at exactly the same time. Also, it is clear that a based period of a minute is too low and causes too much bandwidth to be used per node. We need to further investigate suitable base periods.

9 Future Directions

The self-organizing Distributed Directory (DD) storage solution is a powerful substrate to maintain storage. In particular the host heartbeat database can be used as a local reputation database. That is, it is possible to create a completely decentralized storage solution with no third responsible party entity because each node knows locally the reliability and capability of other nodes. That is, a clients credentials as a public/private key pair are the same as the their nodes credentials (same public/private key pair) and can be used to form a statement of responsibility. Specifically, each heartbeat is signed by the initiating host and contains a public key. Every receiving node can use the initiating hosts public key as tool to verify direct or indirect communications from the initiating host. To be clearer, we are describing self-organizing fair storage. That is, each node can individually decide how much local storage other nodes can consume. For example, a local storage policy may be to allow a node $i$ to consume a proportion of $\frac{\sum_{j=1}^{N} a_j T_j}{\sum_{j=1}^{N} a_j T_j}$ of the total local storage. Notice that no agreement is necessary for nodes to make local storage decisions. The resulting storage situation is that an individual node/client receives aggregate storage proportional to their reliability and/or capabilities.

10 Conclusion

We have analyzed the benefits of this system, and have chosen to favor availability and persistency over consistence. By allowing the data to branch, the data consistency regulation is relaxed and the infrastructure becomes more important to the system. We can then focus on the availability of nodes in the system to increase maximise our choice of availability and persistency.

DHTs do not take into account the suitability of a given peer for a specific task before explicitly or implicitly delegating that task to the peer[18]. In our analysis, differentiating among high- and low-availability nodes saves bandwidth. The savings increase as the gap widens, and if lifetimes are longer for highly available nodes (evident in Figures 2). Finally, storing pointers only works if the amount of data to pointer ratio is high (Figures 3).

By understanding the system dynamics and parameters, we can create a self-organizing option that is durable and available. One self-organizing option, is to use a DD. Figure 4 used the self-organizing mode to achieve six 9’s of data availability. The DD configuration used a DHT of all nodes for pointers and reliable nodes for data, used coding, had an average of large objects, the average lifetime of the reliable nodes was eight months, and the reliable nodes were designated as having availability greater than or equal to 70%. Another non-self-organizing option for robust and bandwidth-efficient storage is to explicitly designate two rings [10], one reliable ring used for storage another unreliable ring not used at
all.

We have shown that secure and immutable data assist in continuous adaption and repair of the storage system. Analytically, we showed that differentiating between nodes based on reliability saves bandwidth. A level of indirection allows the system to capitalize on more reliable nodes, while DHTs, to their detriment, do not take into account the suitability of a given peer for a specific task [18]. After showing how efficient DD-based storage solutions can be, we described an DD architecture and implementation. We used host heartbeats and a local host databases to route to live objects, select storage nodes for replica placement, and trigger repair. Simulation results the tradeoff between heartbeat and liveness need to be further investigated.

11 Acknowledgments

We would like to John Kubiatowicz and Anthony Joseph for the many valuable advice and suggestions. We would like to also thank Byung-Gon Chun for his assistance in analyzing the storage models. Next, we would like to also thank Dennis Geels, Sean Rhea, and Jeremy Stribling for building the simulation and benchmark infrastructure. Finally, we would like to thank all the OceanStore authors for the many years of hard work.

References


