A Secure DHT via the Pigeonhole Principle

Randy Baden Adam Bender Dave Levin Rob Sherwood
Neil Spring Bobby Bhattacharjee
{randofu, bender, dml, capveg, nspring, bobby}@cs.umd.edu

Department of Computer Science
A. V. Williams Building
University of Maryland
College Park, MD 20742

September 24, 2007

TR-CS-4884

Abstract

The standard Byzantine attack model assumes no more than some fixed fraction of the participants are faulty. This assumption does not accurately apply to peer-to-peer settings, where Sybil attacks and botnets are realistic threats. We propose an attack model that permits an *arbitrary* number of malicious nodes under the assumption that each node can be classified based on some of its attributes, such as autonomous system number or operating system, and that the number of classes with malicious nodes is bounded (e.g., an attacker may exploit at most a few operating systems at a time). In this model, we present a secure DHT, *evilTwin*, which replaces a single, large DHT with sufficiently many smaller instances such that it is impossible for an adversary to corrupt every instance. Our system ensures high availability and low-latency lookups, is easy to implement, does not require a complex Byzantine agreement protocol, and its proof of security is a straightforward application of the pigeonhole principle. The cost of security comes in the form of increased storage and bandwidth overhead; we show how to reduce these costs by replicating data and adaptively querying participants who historically perform well. We use implementation and simulation to show that evilTwin imposes a relatively small additional cost compared to conventional DHTs.

1 Introduction

Distributed systems can be made resilient to random failure through replication and careful protocol design. As the incidence of random failures scales with size, large systems must be resilient to such failures in order to maintain availability and encourage usage. However, resilience to correlated or Byzantine failure is not a conventional concern. The cost in messages, computation, and complexity of such resilience can be high, implying such robustness is rarely a goal of practical distributed systems.

The correlations between hosts in recent high-profile failures (not the failures themselves) could have been predicted: dependence on the same power grid, on the same software implementation, or on the same Internet provider, for example. In this paper, we explore partitioning distributed systems into replica groups using these predictable correlations, or "attributes," so that at least one group will be unaffected by the failure or attack: this group will then preserve the correctness and availability of the system as a whole. We do not distinguish fail-stop failures from compromised, "Byzantine" participants because the same approach applies to both: as long as the attribute can be remotely-inferred, for example by geographic certificates [2], subverted nodes can be separated from uncompromised.

If such a system is practical, the pigeonhole principle leads to a trivial proof of correctness. We can partition nodes into classes by the attribute deemed most likely to fail, such as operating system or power grid, and assemble replica groups so that nodes that share the same attribute are in the same group. Replica groups may include nodes from more than one class: if we want only to tolerate a single power grid failure, nodes in North America, Australia, and Africa can be part of the same replica group. If the system contains at least $f+1$ replica groups, and a failure can stop (or an adversary can corrupt) at most f classes, then by the pigeonhole principle, at least one replica group is "clean," and hence, the entire system is still "correct."[1]

Constructing a system from smaller replica groups raises technical challenges. Foremost, a recipient should be able to recognize corrupt responses. Self-verifying data, as is often used in distributed hash tables (DHTs), provides such a primitive: a corrupt DHT can discard "put" requests and may delay "get" requests, but cannot produce "fake" items. This avoids the voting required in more general Byzantine-fault-tolerant (BFT) systems. Second, system capacity is reduced by replication, and unevenly-sized classes may further aggravate this limitation. For example, a system in which the group of Windows nodes can only store the same amount of data as the group of Solaris nodes will leave many resources under-used. Third, although the high-level segregation of classes may be straightforward along a single attribute, applying the technique to more than one attribute at a time is not.

The goal of this paper is to develop a realistic system that addresses these and other concerns. Although the techniques we describe apply to any protocol for which the simplifying assumption (replica groups can be prevented from issuing "false" statements) holds, we restrict our focus to DHTs as a concrete example. Our contributions are as follows:

- *A prototype system, evilTwin,* that implements replica-group DHTs and shows the efficacy of our techniques.

[1]The pigeonhole principle says that if no more than f pigeons (corrupt classes) are mapped into $f+1$ or more pigeonholes (replica groups), then at least one one pigeonhole (group) must be empty (uncorrupted).

- *Multiple-attribute partitioning* that allows nodes to join replica groups based on more than one attribute, without an explosion in the number of replica groups.
- *Storage overhead reduction* based on erasure codes and, counter-intuitively, constructing *more* replica groups than are necessary for resilience. Such coding implies a cost when fetching items, since lookups span multiple groups. Reducing storage overhead also reduces the amount of data exchanged as nodes join and leave the system.
- *Adaptive-get* strategies for querying responsive and underutilized rings when the system is uncompromised, and for avoiding corrupt replica groups when it is.
- *Analysis* of the performance and costs of replication across classes, using theory, simulation, and implementation.

We next present related work in Section 2, our assumptions and attack model in Section 3, and top-level strategy in Section 4. Section 5 provides the protocol design. We analyze the scalability and expected costs in Section 6, and measure through simulation and an OpenChord-based implementation in Section 7. We then conclude.

2 Related Work

BFT Systems Byzantine fault tolerant (BFT) systems are resilient to arbitrary behavior as long as a $\frac{2}{3}$ majority of participants are correct. Many of these systems [6, 31, 1] require at least $3f+1$ replicas and a Byzantine agreement protocol which does not scale well as f increases [12].

evilTwin uses the same basic principle of replicating a service across many instances, but can handle f faults with only $f+1$ replicas. This is possible since we rely only on *fork* consistency* rather than *ideal consistency* [20]. Specifically, we assume that lookup operations take a bounded amount of time (so liveness is not a concern), and we use self-certifying data [3] to guarantee that lookups succeed when a single node responds correctly.

The BAR Gossip protocol [19] protects against Byzantine failures in data streaming applications. BAR Gossip and evilTwin both protect against Byzantine failures by constraining the set of peers with which a node is allowed to interact. BAR Gossip does this by restricting the number of nodes that can be contacted per round, while evilTwin does this by segregating the nodes into disjoint classes that operate independently.

Haeberlen et al. [12] use Byzantine fault detection to moderate behavior in a system where faults are recoverable. They use an accountability system to identify misbehaving nodes and remove them from the system. This is not realistic in our attack model, since an attacker can generate arbitrarily many nodes to replace the nodes that are ejected.

DHTs Early DHTs were designed to be resilient to random failures only [33, 29, 25, 35]. In these systems, malicious nodes can silently and cooperatively corrupt overlay routing tables, refuse to forward messages, and refuse to store items. These DHTs are vulnerable to attacks in malicious environments, including complete denial of service [3].

Secure DHTs Castro et al. [3] and Fiat et al. [9] both consider DHTs where at most $\frac{1}{4}$ of the nodes are malicious. Castro et al. use a centralized certificate authority to prevent Sybil attacks, which makes the bound on the number of malicious nodes reasonable, but requires centralized administration which is undesirable in a distributed setting. Fiat et al.'s S-Chord groups sets of contiguous nodes into *swarms*. Nodes flood requests to every node in a swarm, which requires $O(\log^2 n)$ messages. S-Chord makes no assurances against a Sybil attack.

SDHT was the first system to address the attack model in this paper, and is the inspiration for this work [24]. SDHT virtualizes nodes similarly to S-Chord's swarms, but its scalability is limited by the Byzantine agreement protocol.

Correlated Failures Glacier [13] and Phoenix [17] provide resilience to correlated failures. Both systems require a secure overlay to operate correctly.Glacier and Phoenix use the system described by Castro et al. [3] as a secure overlay; using evilTwin would be a more efficient way to protect against attacks which fall under our attack model.

Glacier uses erasure codes to reduce storage requirements and provides a way to make encoded blocks self-certifying. evilTwin makes use of both of these techniques.

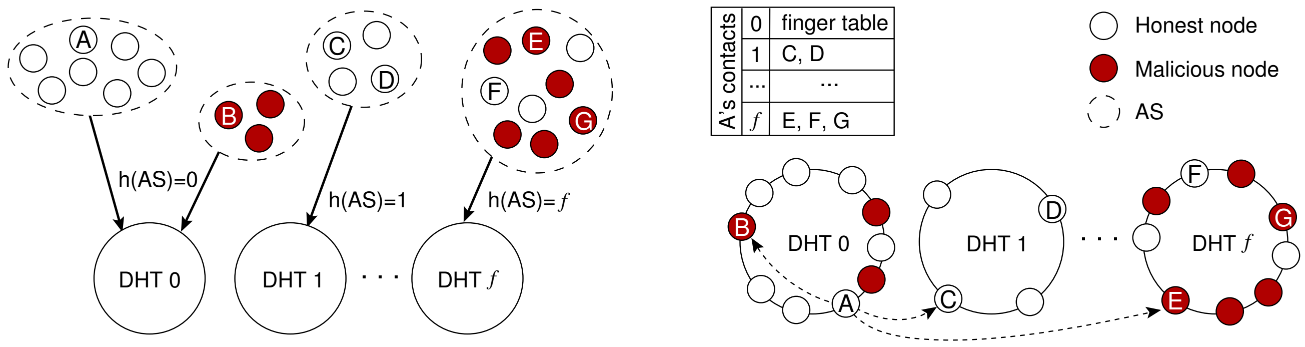
Phoenix partitions nodes into *cores* based on their attributes, guaranteeing that at least one node in every core will survive a correlated failure. evilTwin is similar conceptually, except that in evilTwin the members of a given "core" are likely not aware of each other since they reside in disjoint DHTs. This separates routing and forwarding functionality from core creation and maintenance. Phoenix and evilTwin both assume that the set of attributes which may fail are known before the system is instantiated.

Remote Attribute Verification evilTwin relies on remotely-verifiable attributes to partition the nodes in the system. Bazzi and Konjevod [2] provide a method for creating certificates that nodes may use to remotely identify which nodes are near them geographically. Lippmann et al. [21] describe a method for identifying OS based on TCP/IP headers. Mao et al. [22] provide a utility for mapping IP addresses to ASes, which allows evilTwin to partition nodes based on AS number.

Multiple DHTs Several previous works create a hierarchy of DHTs [16, 23]. The first system to employ multiple rings for security was SkipNet [14] wherein nodes use content and path locality while allowing administrative control over item placement.

3 Attack Model and Assumptions

evilTwin's attack model extends the standard Byzantine attack model by giving attackers the capability to launch Sybil attacks [8]: an attacker can control an arbitrary fraction of the nodes in the system, but can only exploit nodes that share some bounded number f of *attributes*. These assumptions accommodate the practical realities of building secure, cooperative systems today. Botnets and Sybil attacks preclude bounding the number of malicious nodes. However, both originate from a few attributes; botnets typically exploit a specific software bug, such as an



(a) Mapping ASes to DHTs.

(b) Issuing a query.

Figure 1: evilTwin design. (a) Nodes join the DHT that corresponds to the hash of their AS's number. Multiple ASes may be mapped to the same DHT. At least one DHT will have only honest nodes (DHT 1 here). (b) Clients replicate queries across every DHT and maintain a small list of contacts for each DHT.

unpatched Windows server (viz. CodeRed [7]), and Sybil attacks are easily launched from a few machines on the same subnet.

We further assume that the "exploitable" attributes can be *remotely inferred* accurately. This assumption is reasonable in practice, as well. A node's autonomous system (AS) number, for instance, can be determined based on its IP address with WHOIS queries, iPlane's [15] periodically published IP-to-AS lists, or techniques described by Mao et al. [22]. Remote OS identification has also been studied [21].

Although designed to address malicious behavior, this model also captures correlated failures. Power outages can cause massive, correlated stop-failures; the 2003 black-out of the Eastern Interconnection caused an estimated 50 million people in Canada and the northeast US and to lose power for more than two days [34]. Natural disasters, such as hurricanes or earthquakes, can lead to similar, geographically correlated failures. Both of these examples typically affect a large but contiguous geographic location, which even coarse-grained geolocation techniques [11] can remotely determine.

Since evilTwin handles both malicious and random faults identically, we use "attack" and "failure" interchangeably.

Lastly, we assume the users store self-certifying items [3] so that when an item is returned its validity can be checked. In Section 5, we show how this assumption allows evilTwin to achieve fork* consistency [20].

4 Partitioning by Attribute

evilTwin's approach is to *identify and collect nodes that are likely to fail together.* evilTwin does this by partitioning the participants into classes such that each class may experience massive correlated failures (or Sybils), but any two classes are expected to fail (or be compromised) independently.

Failure type	*Suggested partition*
AS compromise	AS number or routable prefix (/16)
Network outages or admin error	Routable prefix (/24)
Worms, Viruses	OS type/version and running services
Power loss (local)	Geographic location (within a small radius)
Power loss (grid)	Geographic location (grid boundary)
Disk failure	Vendor and age (5+ years old or not [30])

Table 1: Failure types and possible partitionings.

This partitioning requires two system-wide decisions: (1) the attribute(s) on which to classify nodes and (2) how many concurrent failures, f, to allow. The user that initiates an instance of evilTwin chooses the node-classifying attribute based on the types of failure the system is intended to be resilient to. In Table 1, we give examples of remotely verifiable attributes for different failures and attacks. Choosing which faults to protect against depends on the deployment's goals and context; if being deployed in a developing country, an unreliable power infrastructure would likely be a greater concern than a botnet.

Choosing f depends on the type and the expected scope of the failure. For instance, power failures at the grid level (of which there are only 3 in the US [34]) are likely to be much more rare than at the transformer level (of which there are many within a several-mile radius of one another). f gauges the trade-off between security and overhead; the larger f, the greater the number of faults handled, but as we will see in Section 6, the greater the overhead.

5 Protocol

We present our protocol, evilTwin, that guarantees publish and lookup success despite malicious or otherwise faulty nodes. We assume nodes can be partitioned into disjoint classes, as discussed in Section 4, up to f of which can fail at any one time. Our base protocol achieves security through item replication which reduces the total system-wide storage capacity. We propose two extensions to evilTwin that use coding techniques and adaptive querying to mitigate increased storage and communication overhead. Lastly, we show how evilTwin can be used to provide resilience across multiple attributes simultaneously.

5.1 Base Protocol

evilTwin consists of $f + 1$ node-disjoint DHTs. Individual DHTs can be any implementation that supports publish and lookup; within the same deployment, implementations may differ. We require no modifications to or security guarantees from the individual DHTs. For ease of explanation, we will assume that each individual replica DHT implements Chord [33], and refer to each instance as a *ring*. We illustrate evilTwin in Figure 1.

Joining Each class c is mapped using a globally-known hash function h, and nodes that belong to c are mapped to ring $h(c) \in \{0, \ldots, f\}$. We assume that the number of classes, C, is much larger than the number of concurrent faults, f: many nodes from different classes may be mapped to the same ring. That is, there may be many classes c and c' such that $h(c) = h(c')$; this has no adverse effect on our system. Node p joins the system by first contacting a bootstrap service, which may be distributed. The service stores a set of previously-joined nodes for each ring, and returns to p the IP addresses of nodes in those sets. p uses these addresses to contact nodes in its own ring and execute the join protocol of that DHT implementation, as well as create a list of nodes in other rings.

The join protocol, by construction, provides the security guarantees of our system. The hash h enforces that all nodes from a class will be mapped onto only a single ring, even if that class consists strictly of malicious nodes. Honest classes may be mapped to that ring as well, but they will still be able to publish and retrieve items on honest rings, as in the case of node A in Fig. 1(b). Because there are at most f malicious classes, they can be mapped to at most f rings; by the pigeonhole principle, at least one of the $f + 1$ rings contains no malicious nodes.[2]

Issuing System-wide Requests Nodes issue publish and lookup requests on every ring in parallel. Each of the $f + 1$ rings performs each publish and lookup independently, resulting in $f+1$ replicas of every item. Load imbalance and redundant use of disk space are the fundamental costs of security in our system. Publish or lookup requests succeed on each honest ring, of which there is at least one. Lookups can be performed by querying rings iteratively instead of in parallel to limit unnecessary messages, at a potential cost in latency. We describe this trade-off in Section 5.3.

To issue a system-wide query, participant p must be aware of at least one node from each of the other f rings. p issues requests to these nodes; each of these nodes then executes the request on its ring and returns the result to p. If a ring is empty, publishes and lookups are not guaranteed to succeed. Nodes that join early periodically query the bootstrap service to learn when all rings are available.

When a client issues a lookup for an unpublished item, it must wait for a response from or timeout on all $f + 1$ rings before concluding that the item is not available. Since the requests are done in parallel, this is equivalent to waiting for the slowest ring to respond. To prevent unresponsive, potentially malicious, nodes from causing excessive delay, we use timeouts and ignore rings that do not respond in time.

Verifying Responses Item and routing information in evilTwin must be verified to ensure correctness. By assumption (Section 4), all items are self-certifying, so when an item is returned nodes may check its validity.[3] Node IDs in routing responses (e.g., from an iterative lookup or when joining a ring) must also be verified to ensure malicious nodes remain in their designated ring. By construction, any node can remotely verify another node's attributes and verify that the hash of those attributes correspond to the given ring. A simple challenge-response suffices to verify the source of a response.

[2]When $f = 0$, evilTwin is identical to the base DHT, and does not provide any extra security properties.

[3]If items are not self-certifying, additional mechanisms, perhaps based on voting, will be required to validate items.

Fork* Consistency Typical BFT systems guarantee *ideal consistency* [20], which requires $3f + 1$ replicas. These additional replicas provide liveness (f replicas may fail to respond) and agreement (f replicas may disagree). We eliminate the liveness requirement by imposing timeouts on lookup and publish requests, and we eliminate the agreement requirement with self-certifying data. After removing those requirements, we provide *fork* consistency* using only $f+1$ replicas. The tradeoff of this approach is that lookups for non-existing items must wait for the entire timeout to conclude that the item has not been published. Timeout values can be chosen aggressively if necessary.

Recovering From Ring Failure We do not provide explicit mechanisms for detecting or recovering from ring failures. We expect that such failures will either be so infrequent that repairing a ring can be done using an out-of-band mechanism, or that such failures will be so pathological that repairing a ring will be pointless. In either case, a failed ring will not damage the correctness of the system unless more than f classes have failed since the instantiation of the system.

Periodic Republication In some circumstances, a failed ring may be recovered without losing the routing capabilities of the underlying DHT, for instance, if a faulty operating system is patched. Since only the data is lost in this case, requiring that owners periodically republish their data to keep it alive will allow the previously failed ring to discover all of the data and regain its status as a good ring.

5.2 A Storage-Latency Trade-Off

The primary limitation of the base protocol is that $\frac{f}{f+1}$ of the system-wide storage space is consumed to store redundant copies of items across all of the rings. We can improve the space efficiency of the system by *increasing* the number of rings and storing data with a more efficient data encoding technique, similar to the approach of Glacier [13]. It is counter-intuitive that increasing the number of rings would save space, so with a concrete example, Reed-Solomon codes [27], we demonstrate the potential space savings.

Reed-Solomon codes take as input an item of size S, separated into b blocks, each of size S/b, and output $b + k$ symbols, also of size S/b. Clients must retrieve any k symbols to reconstruct the published item. evilTwin can apply this coding by creating $f + k$ rings instead of $f + 1$ and mapping classes to rings via h to $\{0, \ldots, f + k - 1\}$. Clients store encoded symbols, each of size S/k, on all $f + k$ rings; since at least k of these rings are good, clients can retrieve at least k symbols and successfully decode any items. Such a system uses $\frac{f}{f+k}$ of the system-wide storage for redundant copies, so storage efficiency improves with greater k.

5.3 Reduced Message Overhead vs. Lookup Latency

A client may not always need to send queries in parallel to all $f + k$ DHTs. Instead, if the client believes that the item exists and that no rings have failed, it may query k historically well-performing DHTs using Adaptive-Get (Algorithm 1). The client sorts the rings based on how well they perform—how often or how quickly they successfully return items. To remain agile to

Algorithm 1 Adaptive-Get: For items expected to exist

```
1. Let $R$ be the list of rings, and for each $r \in R$, let $\rho(r)$ denote the *observed* probability that $r$ successfully returns queried items.
2. Sort $R$ in decreasing order of $\rho(r)$.
3. While $R \neq \emptyset$ and the item has not been returned:
   (a) $Q \leftarrow \mathsf{pop}(R)$ (on the first iteration, pop the first $k$ elements).
   (b) For any set $S$ of rings, let $P(S)$ denote the probability that at least one ring in $S$ will successfully return a query: $P(S) = 1 - \prod_{i \in S} (1 - \rho(i))$.
   (c) While $R \neq \emptyset$:
       i. $q \leftarrow \mathsf{head}(R)$
       ii. Remove $q$ from $R$ and add it to $Q$ if $\frac{P(Q \cup q)}{P(Q)} \geq \alpha$; otherwise, break.
   (d) Query all rings in $Q$ in parallel.
   (e) Wait for time $\tau$ for the queries to return.
```

changing network conditions, we maintain a weighted average over time in our simulations. The algorithm takes two parameters, $\alpha \geq 1$ and $\tau \geq 0$. α allows the client to speculate that fewer rings need be contacted in a lookup: at the extremes, $\alpha = 1$ represents our standard protocol, query all $f + k$ nodes in parallel, while $\alpha = \infty$ results in the client querying k rings initially, then one at a time until the item is returned. τ represents how long to wait before querying the next group of rings, allowing the client to try more rings before any request times out, but likely after the request has taken longer than expected. $\tau = 0$ corresponds to standard evilTwin, in that the client will query all $f + k$ rings at once.

To confirm that an item does not exist, a client may send to any $f + 1$ rings, contacting additional rings only if responses are missing. If all $f+1$ claim that the item does not exist, then at least one of these must be from an uncompromised ring; contacting the other $(k - 1)$ rings is then not necessary.

5.4 Partitioning on Multiple Attributes

evilTwin also allows for partitioning on multiple attributes. Suppose there are A attributes, and each attribute a has C_a classes, at most f_a of which can concurrently fail. f_a failures in *each* attribute would result in at most $F = \prod_{a=1}^{A} f_a$ concurrent failures, so we must have at least $F + k$ rings. Naively making a new ring for each point in the cross product of the attributes would require $\prod_{a=1}^{A} C_i$ rings. This is potentially much greater than the lower bound, $F + k$; by definition, $f_a \leq C_a$. We show how to partition nodes so that there are as few uncompromised rings as necessary; reducing the total number of rings to the lower bound is open for future work.

We begin by assigning to each attribute, a, a hash function $h_a : C_a \rightarrow \{0, \ldots, f_a + k - 1\}$, and place a node with attributes $a_1, \ldots, a_A$ into the ring represented by the string "$h_1(a_1)|h_2(a_2)|\ldots$" where "|" denotes concatenation. This yields $\prod_{a=1}^{A}(f_a + k)$ rings, which is greater than the lower bound, $F + k$, but much less than $\prod_{a=1}^{A} C_a$. Correctness is maintained; for each attribute a, there will be k values of h_a to which no compromised nodes map, hence at least k^A rings will contain no compromised classes at all.

It is not necessary to have so many uncompromised rings. We reduce this number from k^A to the minimum necessary, k, by using a different k for each attribute, as follows. Let h_a map to $\{0, \ldots, f_a + k_a - 1\}$ and pick the k_a's that solve the following:

1. $\forall a : k_a \geq 1$ (so that no attribute can fail for all classes)
2. $\prod_{a=1}^{A} k_a \geq k$ (obtain at least the k rings to perform the encoding)
3. minimize $\prod_{a=1}^{A}(f_a + k_a)$ (obtain as few rings as possible)

If all f_a's are equal, the solution is to set each k_a to $k^{\frac{1}{A}}$ (taking floors and ceilings as necessary to obtain integer values that do not violate the three constraints). If one f_a is disproportionately large, the solution is to set that attribute's k_a to k and the others' to 1.

6 Theoretical Analysis

DHT performance, such as lookup times or the system-wide storage capacity, is a function of the number of participants, N. In this section, we analyze evilTwin's asymptotic performance and summarize our results in Table 2. evilTwin splits nodes uniformly at random into the $f+1$ different rings, so its properties are determined by the number of nodes in the smallest and largest instances. We assume for ease of exposition that participants are chosen uniformly at random from the set of all classes. Even under this assumption, it is highly unlikely that each instance will contain exactly $N/(f+1)$ nodes. With high probability, the smallest instance will be of size no smaller than $m = N/(f+1)^2$, while the largest will be no larger than $M = \frac{N \log(f+1)}{f+1}$ [18, 33].

6.1 Base System ($k = 1$)

Request Times Hop count is the standard approximation of query completion times for DHTs. Lookup and publish requests in any given (honest) ring in our system are faster than in a single N-node ring, since each of the $f+1$ rings has only a fraction of the N participants. Let us assume for ease of exposition that each of the rings execute requests over N participants in $O(\log N)$ time. If the source of a query successively issues recursive requests to each of the $f+1$ instances, the time at which the first and last instance return would be no greater than $(f+1+O(\log m))$ and $(f+1+O(\log M))$, respectively.

Request Overhead The number of messages required to perform a request in our system is the sum of the number of hops required in each of the $f+1$ rings. We can achieve an upper bound on this by assuming that each ring is of size M, achieving $O((f+1)\log M)$.

Finger Table Sizes Similar to request times, we assume the underlying DHTs require finger tables of size $O(\log N)$. Each participant would have to maintain at most $O(\log M)$ finger table entries for its own ring as well as $O(f)$ nodes to perform requests across all rings. Participants must therefore maintain finger tables of size at most $O(f + \log M)$, $o(f)$ more than standard DHTs.

Aggregate Disk Space Recall that every data item is published to each of the $f+1$ rings. Providing Byzantine fault tolerance comes predominately at the cost of the system's *aggregate storage capacity*, that is, the storage capacity of the system as a whole. Suppose that each participant i can contribute $d(i)$ units of storage to the system, and for ease of discussion, that $d(i){=}1$ for all i. In an ideal DHT, the aggregate storage capacity would be $\sum_i d(i) = N$. In evilTwin, storage capacity is proportional to the *minimum size* of the $f+1$ different DHTs: no less than $N/(f+1)^2$.

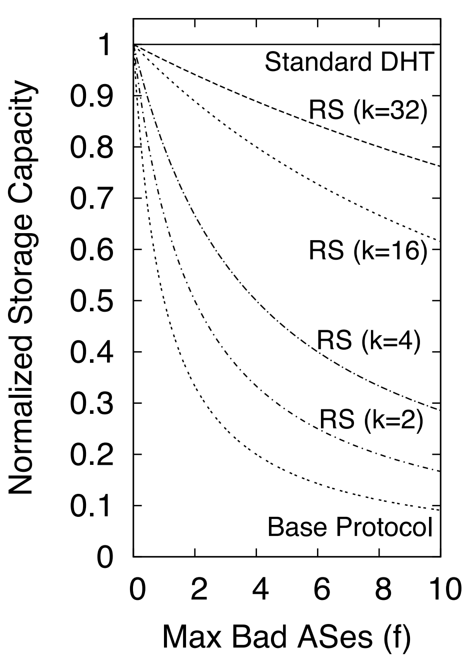
Bootstrapping Our system requires at least one node to be present in each of the $f+1$ DHT instances in order to operate. Since each class gets mapped uniformly at random to a different DHT instance, then, with high probability, there must be participants from $\Theta((f+1)\log(f+1))$ different classes to bootstrap the system.

6.2 Reed-Solomon Extension ($k > 1$)

Recall from Section 5.2 that, when publishing a file of size S, each of $(f+k)$ rings stores a Reed-Solomon-encoded block of size S/k. This approach requires $(f+k)S/k$ system-wide storage to store a file of size S, a factor $\frac{f+k}{kf+k}$ less than our base protocol. Although the smallest ring in the Reed-Solomon extension is a factor $\frac{f+1}{f+k}$ smaller, the storage capacity increases from $N/(f+1)$ to $kN/(f+k)$. In Figure 2, we demonstrate how standard DHTs, our base protocol, and the Reed-Solomon extension trade off between storage capacity and network overhead. Observe that, as k grows, the storage capacity from the Reed-Solomon extension asymptotically approaches that of a standard DHT.

Increasing the number of rings affects lookup times and network overhead as well. With more rings, the expected size of the largest ring decreases to $M = N\log(f+k)/(f+k)$, and the smallest ring decreases to expected size $m = N/(f+k)^2$. The search-time analysis follows as for the base protocol with these values of M and m; the smaller rings improve lookup times and decrease the expected finger table size. Note, however, that as k grows, more classes must be present to bootstrap the system. The network overhead to publish and retrieve items in terms of number of messages sent increases sub-linearly in k. For example, for $k = 32,\ f = 0$ in Figure 2 (right), there is roughly a 25-fold increase in network overhead over standard DHTs even though there are 32 rings. We believe the appropriate balance between network overhead and storage capacity is application-dependent.

We have assumed the worst case in this analysis, that the number of participants, N, is sufficient to run evilTwin, but small, $\Theta((f+k)\log(f+k))$, resulting in disparate ring sizes. If N increases to at least $\Omega((f+k)\log^2(f+k))$, ring sizes more closely approximate the expected ring size, $N/(f+k)$. In fact, ring will be of size $\Theta(N/(f+k))$, with a small constant factor that decreases as N increases. We summarize our analytical results in Table 2 under this likely scenario; observe that in nearly all cases, evilTwin incurs only a small additive cost. In Section 5.3, we demonstrate experimentally that Adaptive-Get mitigates the communication overhead (row 3).



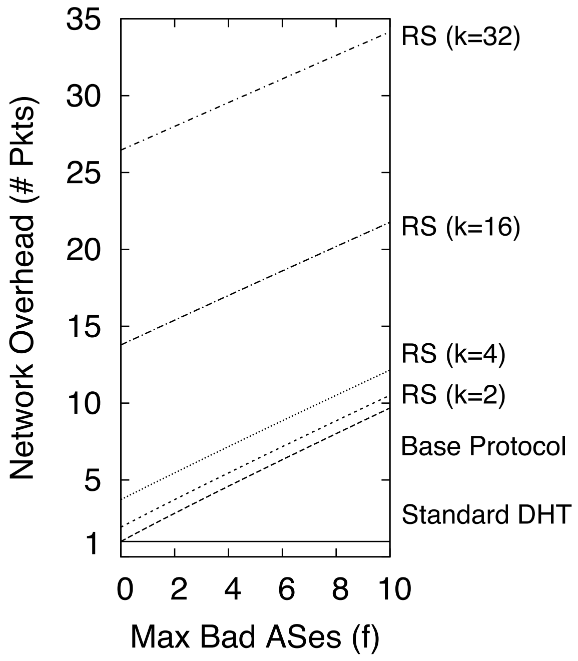


Figure 2: Comparison of storage capacity (left) and network overhead (right) for standard DHTs, our base protocol, and our Reed-Solomon (RS) extension, normalized to that of standard DHTs.

6.3 Encoding Across Multiple Attributes

Recall from Section 5.4 that A attributes, where attribute a can have f_a concurrent failures, can be handled in evilTwin with at most $\prod_{a=1}^{A}(f_a + \lceil k^{\frac{1}{A}} \rceil)$ rings when each f_a is equal. Encoding with k blocks increases the aggregate storage capacity by a factor of

$$k \cdot \frac{\prod_{a=1}^{A}(f_a+1)}{\prod_{a=1}^{A}\left(f_a + \lceil k^{\frac{1}{A}} \rceil\right)} \quad = \quad \prod_{a=1}^{A} \frac{kf_a + k}{f_a + \lceil k^{\frac{1}{A}} \rceil} \tag{1}$$

This value is strictly increasing in k (since $kf_a + k > f_a + k^{\frac{1}{A}}$), A, and f_a's; *the more sources of failure (i.e., the more attributes and classes), the greater the impact of encoding.* This is a generalization of our single-attribute case, which increases storage capacity by a factor of k.

7 Implementation Results

We implemented evilTwin in Java by making only one functional modification to OpenChord 1.0.1; each peer caches a list of representatives from other rings through which it will issue publish and lookup requests. (We also added instrumentation code to track request hop-count and latency.) When $f = 0$ and $k = 1$, evilTwin is identical to the base OpenChord protocol. We use this to provide a point of reference to compare evilTwin to existing DHTs.

Our testbed consisted of 27 hosts connected on a LAN. Since lookup times on a LAN are not representative of wide-area lookup times, we provide both lookup times and the number of hops required to complete the lookup. Hop count indicates the number of messages a request takes.

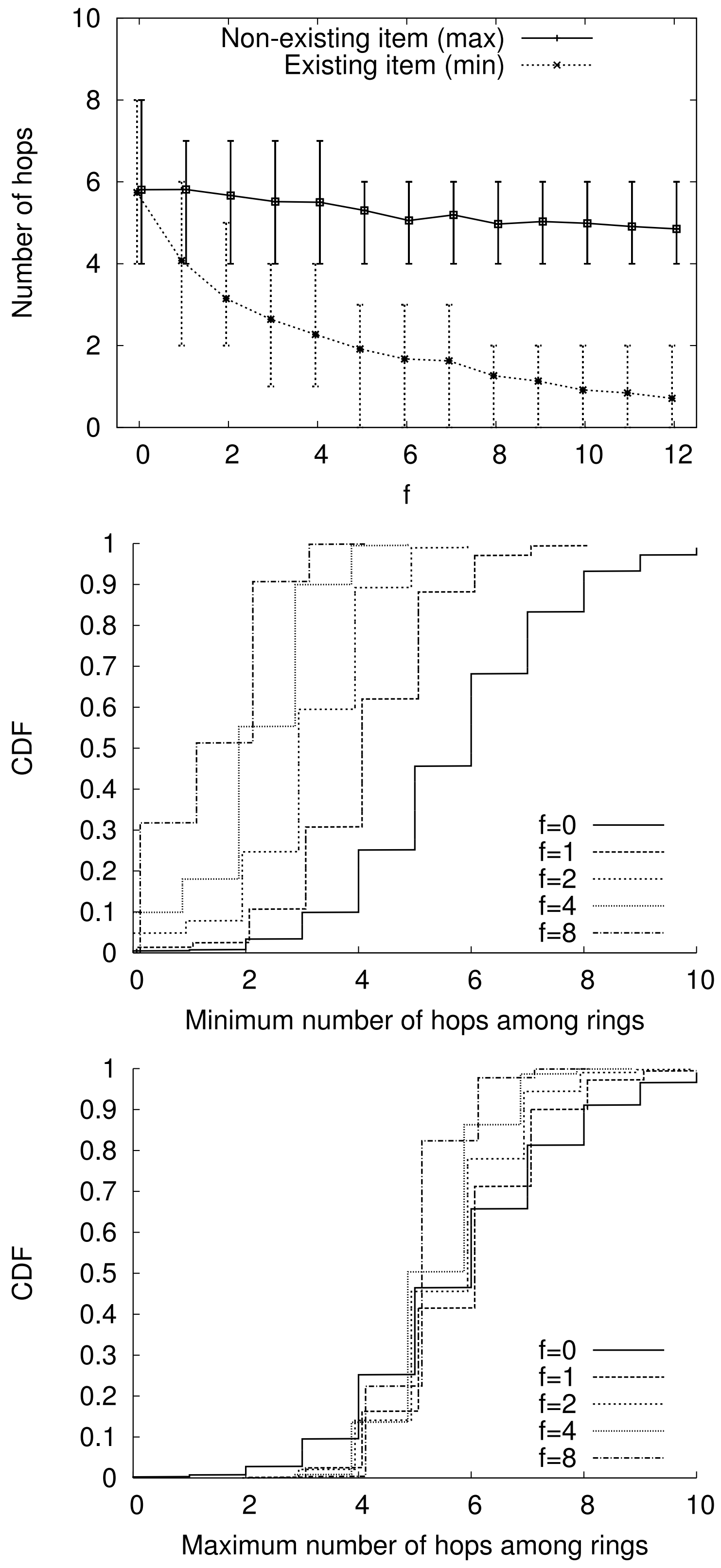


Figure 3: Hop counts (without Adaptive-Get and Reed-Solomon codes). (left) 10% and 90% intervals of hop counts for existing and non-existing items. CDFs of the minimum (middle) and maximum (right) number of hops within each ring.

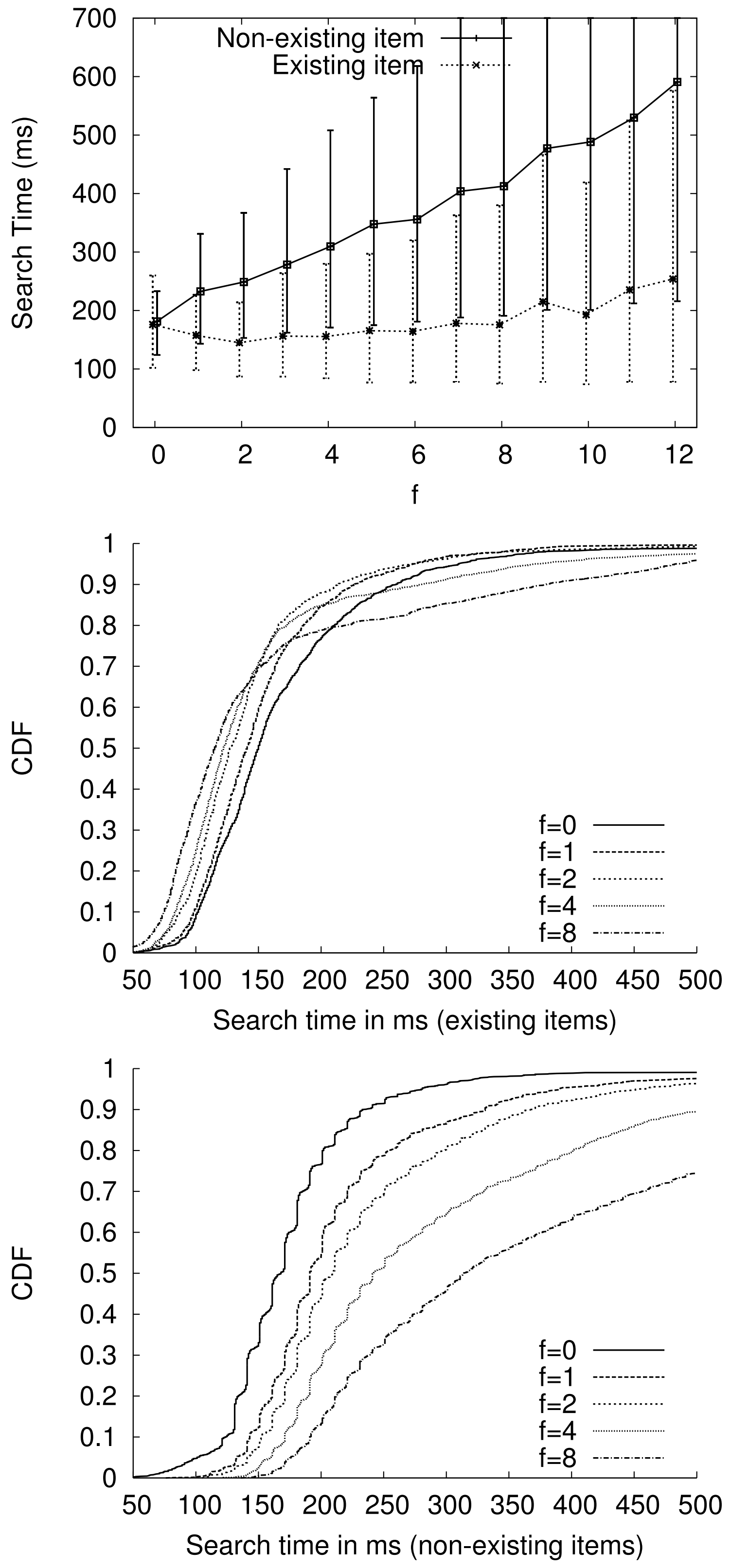


Figure 4: Lookup times on a LAN (without Adaptive-Get and Reed-Solomon codes). (left) 10% and 90% intervals of lookup times for existing and non-existing items. CDFs of lookup times for (middle) existing and (right) non-existing items

Property	*evilTwin*	*Std. DHT*
Request time (hops)	$O\left(f+k+\log\frac{N}{f+k}\right)$	$O(\log N)$
Finger table sizes	$O\left(f+k+\log\frac{N}{f+k}\right)$	$O(\log N)$
Request overhead	$O\left((f+k)\cdot\log\frac{N}{f+k}\right)$	$O(\log N)$
Aggregate disk space	$\Theta\left(\frac{kN}{f+k}\right)$	N
Nodes to bootstrap	$\Theta((f+k)\log(f+k))$	1

Table 2: Expected asymptotic performance of evilTwin.

7.1 Lookup Performance Results

We first measure hop counts (Figure 3) and lookup times (Figure 4) in the system using 300 nodes spread evenly among the 27 testbed machines. Each host was assigned to its own class, and each class mapped to one of the $f+1$ rings by a hash function. Varying f from 0 to 12, we performed 1000 lookups on both existing and non-existing items. There were no failures in this experiment, though our analysis indicates the effect that failures would have on the results. We measure lookup times from before the first request is sent until the requesting peer receives the first successful response.

Existing Items Figures 3 and 4 show that hop counts and lookup times for existing items decrease as f increases. As indicated in our theoretical analysis, the average ring size decreases with increasing f, hence the average number of hops decreases for each ring. This holds even under an attack, when good responses may come from only one ring. Also, peers that a client caches for each of the $f+1$ rings are randomly dispersed throughout their respective rings; the probability increases that at least one of them will be within a small number of hops from the desired object.

On the other hand, Figure 4 shows that lookup times for existing items increases slightly as f increases. Though there are fewer hops within each ring (Fig. 3 left decreases sub-linearly), each request contacts more nodes (f increases linearly). This increased lookup time has been attributed to the phenomenon that, the more nodes contacted, the greater the chance one of them is slow [28].

Non-Existing Items Hop counts for non-existing items change little as f increases, as shown in Figure 3. This is due to two opposing factors: (1) having more rings means it is more likely that there will be a ring that has a higher hop count than the rest of the rings, and (2) the rings are smaller, so we expect that the hop count will be smaller on every ring, including the worst one.

Lookup times for non-existing items, however, increase with f (Figure 4). As before, the greater the number of rings, the more likely that at least one of the rings will return slowly. Failed rings may not respond to a lookup request, so clients would be forced to wait for a timeout before concluding that the item does not exist.

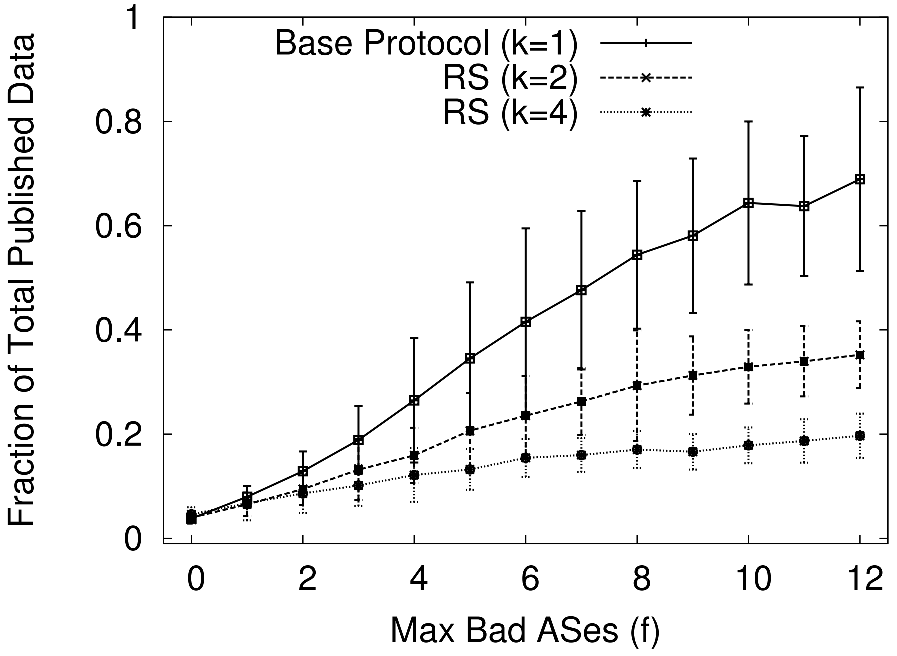


Figure 5: Maximum fraction of items stored on a single node, averaged over 100 runs

We expect that using Reed-Solomon codes will result in lookup times that lie between the lookups for existing and non-existing items, but we have not verified this experimentally.

7.2 Item Storage Results

We simulated running the system 100 times with different hash functions and varying class sizes to observe the expected maximum amount of data stored on any given node. We perform the simulation both with and without the Reed-Solomon extension to capture its effect on storage capacity. We varied f from 0 to 12, k among 1, 2, and 4, and used 27 classes and 300 nodes. In each experiment, nodes publish 600 items of size S, and the system breaks these items into $f + k$ parts, each of size S/k. As in OpenChord, we replicate data items on two successor nodes for resiliency.

Figure 5 shows that without Reed-Solomon codes, nodes in the system can become overloaded with data as we increase the number of rings. Using Reed-Solomon codes with $k = 4$ incurs less storage costs for any given node, even with many small rings. As we increase k, we expect to see more messages per request, or, if we use Adaptive-Get, increased request latency.

The dominant cost of node churn is the migration of existing data from the joining (or departing) node's immediate neighbors. The overhead of churn is directly related to the amount stored at each node (the amount that must be migrated). Our results hence indicate that *evilTwin's Reed-Solomon extension mitigates not only the amount of storage at node, but also the overhead of node churn.*

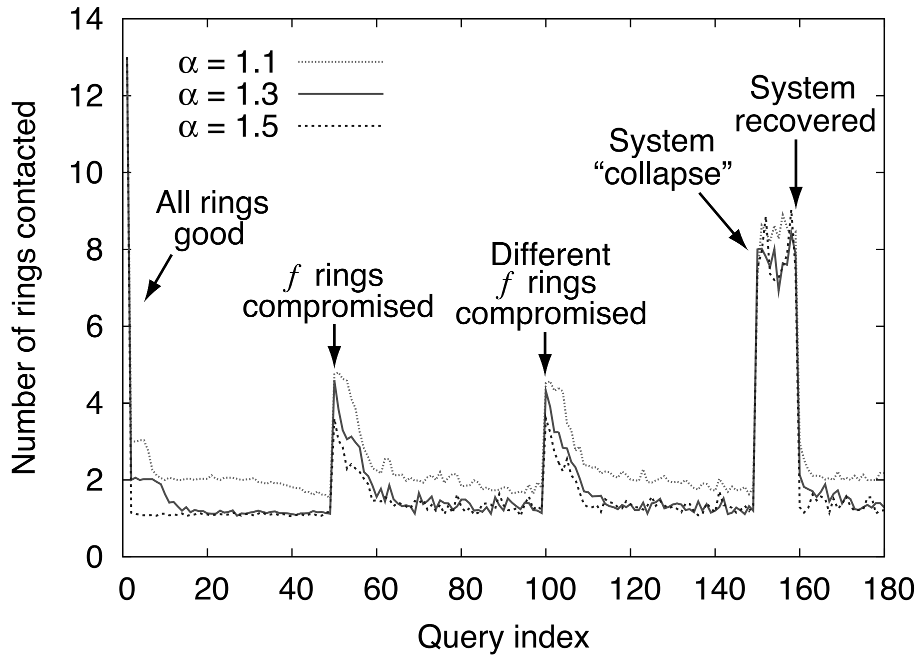


Figure 6: [Simulation] Adaptive-Get quickly adjusts to various major, system-wide events in terms of number of rings contacted when $f = 12, k = 1$.

7.3 *Adaptive-Get* Results

Adaptive-Get (Section 5.3) allows clients to speculatively decrease the number of rings they contact to reduce the message cost of lookups at a potential cost of latency.

Figure 6 demonstrates the agility of Adaptive-Get in the presence of various major, system-wide events: rings becoming (and ceasing to be) compromised, and system-wide collapse and recovery. The first time a client requests a file (query index 1), it contacts all $f+k$ rings; for each subsequent lookup, it uses Alg. 1. In this simulation, compromised rings return successfully only 10% of the time, uncompromised rings 90%. In the event of *system collapse*, each ring successfully returns 10% of the queries; this is in fact a more difficult setting than simply returning 0% of the time, as Adaptive-Get would quickly cull any non-responsive rings. In all cases, the client quickly adjusts, and finds the one good ring to query (if it is available).

The time to adapt depends on at least two factors: the parameter α and, more so, how quickly the environment changes. Adaptive-Get adjusts to new environments less quickly with decreasing α, favoring additional overhead over the latency of waiting for queries to time out. For any $\alpha > 1$, the more quickly the environment changes, the longer it takes to adjust. Clients adjust more quickly, for instance, when f rings become compromised starting when none were (time 50 in Fig. 6) than when a different f were. This is because, in the latter case, the client had to regain its "trust" in the formerly-compromised nodes.

Adaptive-Get performs similarly in implementation, as shown in Figure 7. We again use 300 nodes in 27 classes, and $f = 4$. The vertical lines at every fifteenth query in Figure 7 represent when a ring in its entirety fails. These massive (ring-wide) failures cause spikes in overhead for a successful lookup, but, as expected, Adaptive-Get recovers quickly. Additionally, we see that using

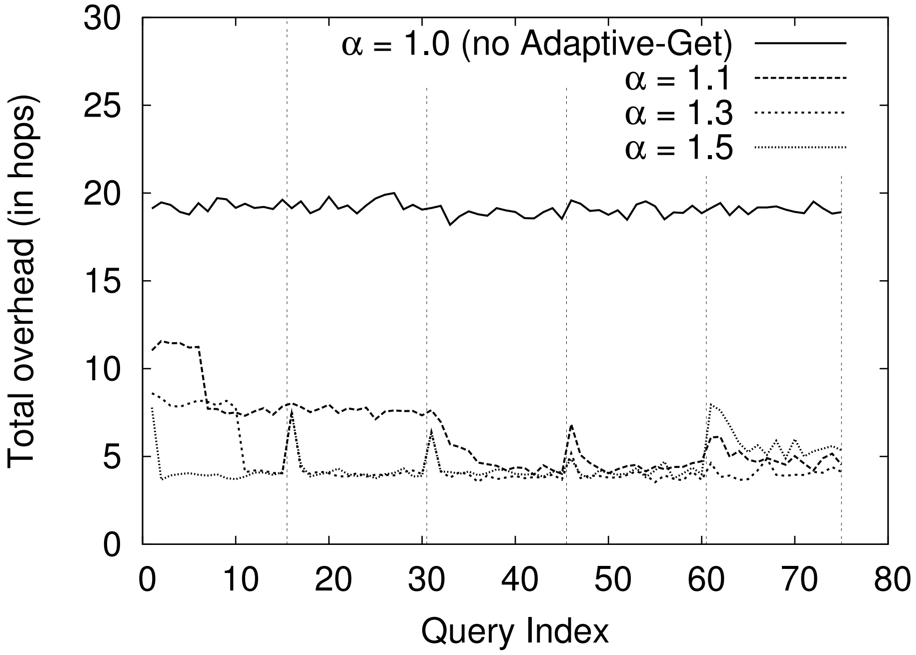


Figure 7: Amount of overhead with and without Adaptive-Get when $f = 4, k = 1$. Each vertical line represents when an additional ring fails.

Adaptive-Get is not likely to incur a significant latency penalty for existing items unless failure is frequent, since it typically only requires successive queries in the event of a failure. Even for a small number of rings (4) and massive failures, the difference in overhead is significant.

8 Discussion and Future Work

We have presented an approach, evilTwin, that provides reliable operation when faulty nodes can be grouped by the vulnerable attribute they share. We contend that *any DHT-like system can be made tolerant to massive correlated failures and Byzantine faults by deploying it in evilTwin.* Application-layer multicast [5, 26, 4], publish-subscribe [32] and anycast systems [10] have each been built with DHTs at the core of their design—such fundamental systems may benefit from the resistance to correlated failure that evilTwin provides. Each of the DHTs that comprise evilTwin are used without modification. These systems can thus be applied straightforwardly in evilTwin by simply running them on each of the $(f + k)$ rings in parallel.

Our basic approach permits optimizations and extensions. We presented an algorithm to allow a client to adapt to find the best rings to limit message overhead and request latency. Our approach uses strictly the observations of the client, but clients might also use an approach like PeerReview [12] to cooperatively choose DHTs least likely to be subverted.

We show evilTwin is feasible by demonstrating that the storage capacity, and hence the cost of node churn, can be made comparable to standard DHTs by employing erasure codes. We believe the costs of evilTwin to be reasonable in practice, but they may be mitigated further, perhaps on a per-application basis.

Although our evaluation combines analysis with local testbed implementation, we intend to further explore the behavior of evilTwin by deploying it on PlanetLab. PlanetLab should help us understand how our design fares in an environment characterized by partial connectivity, churn, and resource heterogeneity [28].

The techniques of evilTwin apply only when malicious nodes can be constrained to independent failure classes, leaving one "pigeonhole" free. Of course, botnets are not so limited, comprising machines in all autonomous systems. Techniques for addressing such unconstrained attackers may complement those we describe in this paper: systems may be made resilient to both types of attack. Specifically, one might apply the work of Castro et al. [3] or S-Chord [9] to some of the evilTwin rings to make them (and perhaps the entire system) resilient to a fraction of malicious nodes in otherwise good rings.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *SOSP*, 2005.

[2] R. A. Bazzi and G. Konjevod. On the establishment of distinct identities in overlay networks. In *PODC*, 2005.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowston, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI*, 2002.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *SOSP*, 2003.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *JSAC*, 20(8), 2002.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[7] CERT. Ca-2001-19.

[8] J. Douceur. The Sybil attack. In *IPTPS*, 2002.

[9] A. Fiat, J. Saia, and M. Young. Making Chord robust to Byzantine attacks. In *ESA*, 2005.

[10] M. Freedman, K. Lakshminarayanan, and D. Maziéres. OASIS: Anycast for any service. In *NSDI*, 2006.

[11] M. Freedman, M. Vutukuru, N. Feamster, and H. Balakrishnan. Geographic locality of IP prefixes. In *IMC*, 2005.

[12] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[13] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[14] N. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. Skipnet: A scalable overlay network with practical locality properties. In *USITS*, 2003.

[15] iPlane. http://iplane.cs.washington.edu/.

[16] H. Jin and C. Wang. A domain-aware overlay network to build a shared DHT infrastructure. http://grid.hust.edu.cn/chengweiwang/2.pdf.

[17] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. Voelker. Surviving Internet catastrophes. In *USENIX*, 2005.

[18] V. King and J. Saia. Choosing a random peer. In *PODC*, 2004.

[19] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. BAR gossip. In *USENIX*, pages 14–14, Berkeley, CA, USA, 2006. USENIX Association.

[20] J. Li and D. Maziéres. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, 2007.

[21] R. Lippmann, D. Fried, K. Piwowarski, and W. Streilein. Passive operating system identification from TCP/IP packet headers. In *DMSEC*, 2003.

[22] Z. M. Mao, J. Rexford, J. Wang, and R. H. Katz. Towards an accurate AS-level traceroute tool. In *SIGCOMM*, 2004.

[23] A. Mislove and P. Druschel. Providing administrative control and autonomy in structured peer-to-peer overlays. In *IPTPS*, 2004.

[24] R. Morselli. *Lookup Protocols and Techniques for Anonymity.* PhD thesis, University of Maryland, College Park, 2006.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *SIGCOMM*, 2001.

[26] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. *Lecture Notes in Computer Science*, 2233, 2001.

[27] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM*, 8(2):300–304, June 1960.

[28] S. Rhea, B.-G. Chun, J. Kubiatowicz, and S. Shenker. Fixing the embarrassing slowness of OpenDHT on PlanetLab. In *WORLDS*, 2005.

[29] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[30] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST*, 2007.

[31] P. Sousa, N. F. Neves, and P. Veríssimo. Resilient state machine replication. In *PRDC*, 2005.

[32] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, 2002.

[33] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM*, 2001.

[34] U.S.-Canada Power System Outage Task Force. Interim report: Causes of the August 14th blackout in the United States and Canada. http://www.nrcan-rncan.gc.ca/media/docs/814BlackoutReport.pdf.

[35] B. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *JSAC*, 22(1), 2004.