Folklore: Implementing Data-Untraceable and Scalable Distributed Storage

Ramsés Morales and Indranil Gupta

Abstract

We have built and deployed a system called **Folklore** that implements a new property called *data untraceability* through a proactive *migratory replication protocol* in a large-scale distributed setting. A data untraceable object, i.e., an object whose replicas cannot be located, is probabilistically difficult to delete from the network. Folklore is a proof-of-concept system that allows data untraceability to be provided hand-in-hand with data availability, updatability, integrity, and scalability. Folklore consists of two subsystems – a pure "Proactive Folklore" system, and a "Reactive Folklore" system (R-Folklore). R-Folklore is a distributed file storage system that makes files untraceable *on-demand* at run-time, e.g., when a file is sensed to be under attack through intrusions. Intuitively, R-Folklore stealthily turns vulnerable files into an untraceable form. We have deployed Folklore on the CSIL cluster at UIUC (our undergraduate laboratory). We present experimental results from the deployment, as well as simulation experiments that match deployment results but extend to larger system sizes.

Index Terms

D.4.7.b Distributed Systems, Availability, Distributed Storage, DHT.

1 Introduction and Motivation

The increase in the popularity of networked information stores with hundreds of nodes, such as digital libraries [1], global-scale storage [2], "brick"-based storage [3], etc., has raised questions about the survivability properties of the data stored in these systems.

• ***Author Affiliations:*** *Department of Computer Science, University of Illinois at Urbana-Champaign, USA.*

• *This work was supported in part by NSF CAREER grant CNS-0448246*

While existing designs seek to provide "forever" [4] storage, where data is forever available, updatable, and has integrity, we argue in this paper that *data untraceability* is an additionally important property that needs to be addressed by these systems.

We call a piece of data (e.g., a file) as "untraceable"[1] if it is improbable for any client (including the owner) to find out where *all* replicas of that data are *currently* located. Thus, data-untraceability is a desirable property in systems wherever availabiliy is required. Systems where data is not untraceable suffer from several attacks that are highly-aggressive. For instance, if all replicas are static (and therefore traceable), consider a malicious node that uses the normal "lookup" mechanism exported by the data store to find out all such host locations' IP addresses, and then targets each such location through one of many different kinds of attacks. Once an attacker knows the IP addresses of all locations of a file, they could harm the file's availability (e.g., by launching a Denial of Service Attack). Worst of all, a highly-aggressive attacker could cause all replicas of a traceable file to be wiped out, thus effectively deleting the file from the network altogether. This means that replicas of a piece of data need to *migrate dynamically*.

All static location schemes are subject to the attacks just described. Whether file replicas are stored through a DHT at randomized (but queriable) set of nodes, or whether they are stored at a fixed set of servers, all of the above attacks could be initiated by a node that otherwise was innocently trying to locate all replicas of the given file. Thus, if data is traceable, it can never be stored forever!

In this paper, we describe a system called **Folklore**, which introduces *migratory and self-stabilizing replication* as a solution that converts data stealthily into an untraceable form, and avoids the above kinds of attacks. The paper first presents a pure version of Folklore, called *Proactive Folklore* where all replicas of the given file are migratory all the time. Proactive Folklore is presented as a system that can be used in a software stack, to allow higher layers to take advantage of its properties. Secondly, we describe a *Reactive Folklore*, a distributed file storage system that aims to *reactively* (i.e., on demand) convert data into a migratory and untraceable form *at run-time*, by layering itself on top

1. Note that our *data untraceability* is not related to the cryptography literature's *(sender/receiver) untraceability* [5], [6], which is an anonymity feature to hide the creator of a data object. Furthermore, we are not addressing user anonymity in any way, like Freenet [7] or Tarzan [8] do.

of Proactive Folklore. This enables data to be stored statically by default, and also to be autonomically made migratory under certain trigger conditions. The trigger conditions for this reaction can be defined by the owner of the file, e.g., based on the intensity of DDoS attacks or intrusions at replica-holding nodes. Our systems are not aimed at online p2p data sharing; instead they are aimed at persistently storing data, i.e., a distributed storage system with self-organizing data-availability. In addition, our Reactive Folklore system allows multiple users to read and update a file.

Somewhat non-intuitively, Folklore also allows *efficient lookups* in spite of the migration, consumes *low network bandwidth*, and is *scalable* in distributed systems with several hundreds of nodes. Our design describes how a file can be looked up within a few seconds through Folklore. The bandwidth utilization at each Folklore host is surprisingly low, as can be seen in Section 5. The bandwidth is independent of the number of hosts in the system, independent of the neighbors at a node, and depends on the average file size and on how fast the self-stabilzation of the number of replicas is achieved.

At the heart of Folklore lies a single core probabilistic protocol that provides both *probabilistic security* and *probabilistic scalability*. This core protocol, called an *Endemic Protocol*, is based on a mathematical model representing the survival of folklores and diseases in human populations – diseases such as measles are practically impossible to eradicate because it is difficult to find out all the people who currently are infected, yet the number of infected people usually stays small.

Endemic Protocol is *not* Gossiping. The main distinction between an endemic and an epidemic (i.e., gossiping) protocol lies in their goal: Gossiping [9], [10] transfers a message to *all* processes within a group, whereas the endemic protocol attempts to probabilistically ensure that data (e.g., a file) is permanently available on a *small* set of processes within a group. Gossiping provides probabilistic message propagation, while the endemic protocol provides probabilistic forever storage.

Unlike the proactive version, the Reactive version of Folklore migrates replicas of a file only when that file is sensed to be under attack, but behaves like a static replication scheme in the common no-attack case. Additional techniques such as erasure coding, encryption, etc., are used to guarantee well-known notions of security and availability. Reactive Folklore (R-Folklore) is generic enough that different attack sensors can be used to trigger the reactive migration, e.g., intrusion detectors, DoS detectors, etc.

We have implemented Folklore, and have deployed it on the Computer Science Instructional Labs (CSIL) cluster at UIUC. Experiments are run on 64 machines.

Other Related Work: The Eternity Service [4] raised issues that have to be solved to provide forever storage on a distributed system. Our endemic protocol can enhance the availability of the data on any system that is influenced by Eternity's ideas. Among the popular distributed storage systems not all rely on full replication, e.g., PAST [11]; there are several that also implement erasure coding [2], [3], [12], [13] for reliability and availability.

Total Recall [12] shares with our work the characteristic that it tries to provide dynamic replication, to adjust the number of replicas depending on the current state of the network, aiming for specific availability. Unlike our system, replica location can easily be tracked by an attacker, and the target number of replicas is not achieved in an self-organizing way. Carbonite [14] provides data durability on a distributed storage, by monitoring replicas and trying to repair replicas at a higher rate than the permanent disk failure rate. Unlike our work, Carbonite requires monitoring of host availability, and resilience to malicious nodes has not been tested.

Systems that address user anonymity, like Freenet [7] and Tarzan [8], by extension make it difficult to locate replicas of an object. Our work directly address the latter, but we do not consider user anonymity, rather providing a simpler data untraceability mechanism. Unlike Freenet, an object inserted into our system will, with high probability, be available.

We design our own membership protocol in this paper (Section 2.7), but there are other protocols providing uniform random partial membership lists. Any of these protocols could be used instead. SCAMP [15] is a gossip-based protocol that builds a partial membership list of a group (at each node) that is guaranteed to be both random, and size $log(N)$. It is intended primarily to support multicast messages by gossip. Our membership protocol provides a random membership list, but unlike SCAMP, it must periodically refresh itself for security issues. Another protocol that provides a partial random membership is CYCLON [16], and it can be used instead of Section 2.7.

Malicious attacks on small-scale distributed systems can be mitigated by using Byzantine fault tolerance [17], [18]. Douceur et. al. identified the limitations of decentralized authorities [19]. Intrusion detection schemes have been proposed by several researchers [20]. Finally, denial of service attacks have been characterized and mitigated for limited settings [21], [22], [23].

The rest of the paper is organized as follows. Section 2 describes endemic protocols and the design of the pure Folklore system, Section 3 describes the Reactive Folklore system. Section 4 presents large-scale simulation experiments, while Section 5 describes results from our deployment in the CSIL lab at UIUC. Section 6 concludes.

2 The Proactive Folklore System

The pure Folklore system is built by augmenting a core *endemic replication protocol* [24]. Endemic replication is a migratory replication protocol that ensures that at any time, each data object in the system has a small number of replicas located at a random set of nodes throughout the system. Further, the set of these locations changes continuously and in a distributed manner.

Most importantly, in order to guarantee the untraceability property, the endemic protocol is self-stabilizing: any deletion of some of these migratory replicas, or creation of extra replicas, autonomically converges back to a stable number of replicas. This autonomic convergence is an emergent behavior of a completely distributed mechanism, and it requires no global knowledge.

We first describe the canonical endemic protocol below (Section 2.1), then explain how it is augmented with essential functionalities such as object insertion, heartbeating, membership management, and object updating (Sections 2.2 through 2.7).

2.1 Core Protocol: Endemic Replication

This section describes the variant of the endemic replication protocol [24] that is used as basis for Proactive/Reactive Folklore. The protocol replicates data objects (or simply objects) that could be file blocks, full files, software services, etc., depending on the system design. Now consider a host (henceforth a "node") H that is currently storing replicas of some of the objects inserted into the system (perhaps by other nodes). We say that the node is currently *stash*ing each of these objects, that it is currently sharing,

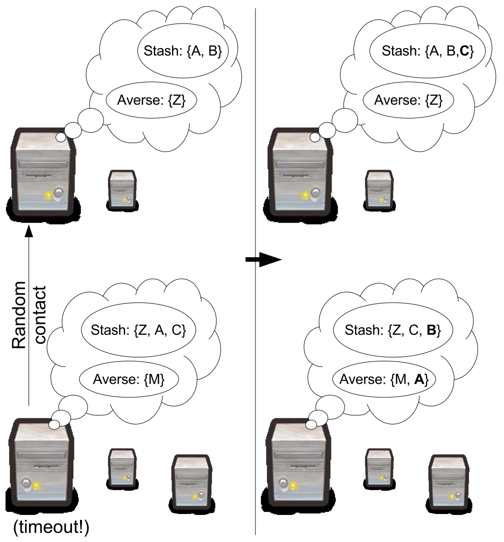


Fig. 1. Basic Operation in the Folklore System: Two nodes talk to each other, thus changing the set of objects that each is stashing (storing). In addition, each node is averse to storing certain objects (which it currently does not have). A node is receptive to any object that the node does not remember, i.e., the node will accept a transfer of that object sent from another node. The protocol is periodic – the bottom node contacts a random node, resulting in the transfer of C to the top node (which is receptive to C) and B to the bottom node (which is receptive to B). Note that the bottom node also modified the state of A.

or that it is in a *stash* state with respect to (w.r.t.) each of these objects. For all the objects in the system that are not locally stashed at H, H is either *receptive* or *averse*. H is said to be receptive to an object i when it is willing to accept a transfer of i's replica from another stasher node of i. If it refuses any such transfer, it is said to be averse w.r.t. that object. In our implementation H maintains a list of identifiers for objects that it is averse to, while it does not need to remember the identifiers of the remaining objects for which it is receptive.

The actions of the endemic replication protocol are determined by four important protocol parameters: protocol period T (typically 300s), fixed probabilities $\alpha, \gamma \in [0, 1]$, and an even integer β. The basic endemic protocol is periodic, with each node executing certain actions once every T time units. Protocol periods are asynchronous across nodes, and do not require synchronized clocks. We will assume that parameter values are fixed at all nodes; however, our experiments in Section 4 show the effect of malicious parameter setting.

At the start of each protocol period at node H, several actions are executed: **(1)** First, each locally stashed object is moved into the averse state with probability γ, independently applied to each stashed object. In the averse state, the object can be deleted, although its identifier needs to be remembered in order to refuse any attempted incoming transfer. Note that the probability of simultaneously moving all the replicas of an object to the averse state is γ^S, where S represents the expected number of replicas in stash state; e.g., our common experimental settings, this propability $= 0.4^{106} = 6.58 \times 10^{-43}$. **(2)** Then, each averse identifier (excluding any that were stashed in the current period) is subject to being forgotten (i.e., moved into receptive state) with probability α. **(3)** Finally, H creates a constant sized `stash advertisement` message, containing a small random permutation of identifiers of locally stashed objects. The stash advertisement is sent to a constant number $\frac{\beta}{2}$ of target nodes, each target selected uniformly at random from across the group (see Section 2.7). Target nodes receptive to the advertised identifier may request a transfer from H based on the *Stash-Receptive Contact* mechanism. In turn, these target nodes advertise their stashed objects to H, allowing it to request objects it is receptive to. This *Stash-Receptive Contact* mechanism is detailed in Section 2.3. Figure 1 shows a basic example illustrating our description so far.

Data-Untraceability: The analysis of [24] shows that for any given object, the *fractions* of nodes in the system that are $(receptive, stasher, averse)$ stabilize around a single equilibrium point – $(\frac{\gamma}{\beta}, \frac{1-\frac{\gamma}{\beta}}{1+\frac{\gamma}{\alpha}}, \frac{1-\frac{\gamma}{\beta}}{1+\frac{\alpha}{\gamma}})$. This equilibrium point is stable, which means the endemic protocol always self-stabilizes around it. However, the protocol exhibits *chaotic perturbation* in the number of replicas in the vicinity of this point, because of random and unpredictable choices by each node while applying the probabilistic protocol actions (1-3) above. Further, although the *number* of stashers is stable, the actual set of nodes stashing replicas of an object is *random and continuously changing*. However, there is a small window of vulnerability, i.e., a short span of time before the set of nodes changes. The time before a single node in the set changes $= T/[S \cdot (1 - S/N)] = \theta(1/N)$, where

$S = N \cdot \frac{1-\frac{\gamma}{\beta}}{1+\frac{\gamma}{\alpha}}$. Thus, the window of vulnerability is very small compared to the network size, e.g., using our experimental settings, the window of vulnerability is only $3.33s$ in a 1000 node network with $T = 300s$. The chaotic perturbation and continuous migration form the core of the data untraceability property of Folklore.

Self-Stabilization: The most important property of the endemic replication just described is its self-stabilization. If the system starts with a small number of replicas of the object (even 1), the *endemic protocol* autonomically and quickly converges to the stable number of replicas. Similarly, if the system is started with a very large number of replicas of the object, convergence occurs downwards to the theoretically-predicted number of stable replicas. We will refer to this self-stabilization property in later sections while describing the system.

Extensions to the core protocol are described in the following sections. From now on, we will use the name *Folklore* to refer to the extended proactive endemic replication protocol. Below we describe the systems considerations on how to insert objects into the Folklore, how to implement peer contacts, object updates, object heartbeating, and system optimizations.

2.2 Object Insertion

An object (file or data item) is inserted into Folklore by its creator, and it is assigned a globally unique id. To create the id, the creator's public key is concatenated with the object name, a SHA1 hash is computed from this string, and the resulting hash is encoded in base64. Reactive Folklore (R-Folklore) uses full filenames as the object name.

The node inserting the object initially warms up the replication by forcing a small random set of nodes to become stashers for the new object; the rest of the nodes in the system stay receptive to the object since they have no knowledge about the object or its id. To ensure that an inserted object ramps up to the stable number of stashers, initial replicas are associated with a *TTL* (time to live). The TTL mechanism is explained better in Section 2.4, after we discuss the stash-receptive contacts.

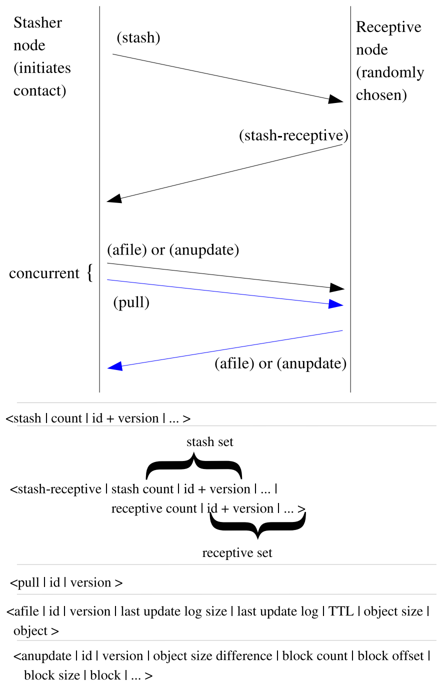

Stasher node (initiates contact)
Receptive node (randomly chosen)
(stash)
(stash-receptive)
concurrent {
(afile) or (anupdate)
(pull)
(afile) or (anupdate)
<stash | count | id + version | ... >
stash set
<stash-receptive | stash count | id + version | ... | receptive count | id + version | ... >
receptive set
<pull | id | version >
<afile | id | version | last update log size | last update log | TTL | object size | object >
<anupdate | id | version | object size difference | block count | block offset | block size | block | ... >

Fig. 2. Message interchange when a node contacts a peer with its stash advertisement.

2.3 Stash-Receptive Contacts

As described in Section 2.1, each node periodically sends out a stash advertisement message to $\frac{\beta}{2}$ target node(s), chosen uniformly at random. The message interchange is depicted in Figure 2, and detailed below.

The message advertises objects using $\langle id, version \rangle$, the latter field being the version number of the object. Suppose node i sends a stash advertisement message, and the recipient node is j. Node j replies to i with a `stash-receptive` message, which contains two sets of object identifiers: a *stash set* and a *receptive set*.

The *stash set* is a random permutation of identifiers of objects that **(1)** are stashed at node j, and **(2)** were not specified in the stash advertisement message from i (thus could potentially be transferred to i). The *receptive set* identifies only those objects chosen from i's stash advertisement that either **(a)** j is receptive to, or **(b)** j is stashing with a lower version number.

Bandwidth Saving Optimization: To save bandwidth, even when a node becomes averse or receptive to the object, we allow the node to retain the object replica. Accordingly, we change condition (a), for selecting the receptive set, to: (a′) include objects that j is receptive to, and those receptive and retained with a lower version number than that advertised by i. In addition, any locally retained objects at j, that are advertised with a lower version number in i's stash advertisement, can be moved into a stash state at node j. Moreover, any retained object at j that happens to be advertised by i with a higher version number gets deleted from j. Finally, if any retained object at j is included in i's advertisement with an equal version number, then it is upgraded at j to the stash state (in effect this is a *virtual transfer*).

Push-Pull mechanism: On receiving the `stash-receptive` reply message, denoted as M', from node j, node i selects one id from M''s receptive set that identifies an object still stashed at i. Node i then pushes the replica identified by id to j. If there is a version number difference of 1, a per-block update (`ANUPDATE`) will be performed. Otherwise, a full transfer (`AFILE`) is used.

Node i also selects one of the objects from M''s stash set that i is still receptive to. Node i then sends a `PULL` message, containing this object's id, to node j. Node j then replies with either a full transfer or update. For all remaining id's in M''s stash set that are retained locally at node i, those objects with local version numbers that are greater or equal are quietly moved into the stash state (virtual transfer); the ones with lower local version numbers are deleted.

Each node expects on average β contacts every protocol period, and each contact has the potential to result in two real transfers. Therefore, the theoretical bandwidth used at each node, from pushing and pulling replicas, is $\frac{2\beta \cdot \text{MeanObjectSize}}{T}$, where T is the protocol period duration, and MeanObjectSize is the *system-wide* average object size. Note that having $C \cdot N$ objects, for some integer $C > 1$, will increase the time to converge, though not the bandwidth. This can be addressed by scaling down T appropriately, to keep the time to convergence constant.

Finally, the following is done to provide a balance between update propagation and replica transfer: Each item in j's advertised receptive set refers, with probability 0.5 to an object that is not stored at node j, and with probability 0.5 to an object that is stored at j with a lower version number. This allows fairness in propagation of updates, and avoids dampening of migration.

2.4 Setting the TTL for Insertions

During insertion of an object, the inserting node warms up the system by inserting a small number ($\ll S_\infty$) of initial replicas, where S_∞ is an estimate of the stable number of stashers (e.g., based on expected system size). In order to ensure initial survivability of the object, these initial replicas are not immediately subject to prospective deletion (by having the nodes turn from stasher to averse), but instead are forced to be stored at those nodes for TTL time units. The temporary effect of the TTL mechanism will help an object achieve the target number of replicas S_∞. Once TTL time units expire, initial replicas affected by it will no longer be artificially manipulated – they will be left to the endemic.

The TTL value is determined and inherited as follows. Let $S_i = max(S_\infty, S_{avg})$, where S_{avg} is the average size of the stash at the node introducing the object. The inserting node initializes the TTL to $w \cdot \lceil \log_{(\frac{\beta}{2}+1)} S_i \rceil$ periods, $w = \frac{S_i}{\frac{1}{2}\beta}$ (expected number of periods to replicate). Thereafter, every new replica created by a stash-receptive contact inherits the TTL from its stashed parent, less a quantity w.

The logic for the above strategy is as follows: the total number of forced replicas is expected to grow by a factor of $(\frac{\beta}{2} + 1)$ every w periods due to the endemic protocol, thus the logarithm's base. It should be evident that all TTLs reach 0 a while after object insertion. Any replica transferred after $\beta/2$ replicas have been transferred inherit $TTL = 0, w = 0$, even if the local replica still has a $TTL > 0$. Figure 3 illustrates this mechanism.

When the TTL for an object replica expires, the normal Folklore actions described previously apply, i.e., the replica can be changed to averse with probability γ per protocol period.

2.5 Parameterized Heartbeating

Parameterized replica heartbeating is a lightweight mechanism that is used to periodically hear about a fraction of the replicas of an object. Who listens, and for what

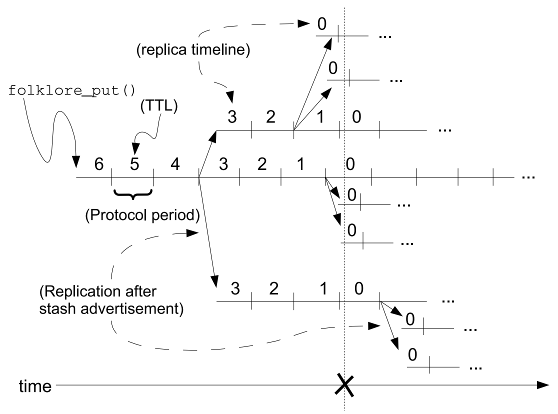


Fig. 3. Use of TTL at Insertion Time: The numbered horizontal lines represent the timeline of replicas created at different nodes. For this toy example, assume that $\beta = 4$, $S_\infty = 6$. This gives us the initial $TTL = 6$, which will be decremented by 3 when inherited by the first two $\beta/2$ replicas. The cross on the time-axis shows when S_∞ replicas are reached.

purpose, is left to the upper application layers taking advantage of Folklore[2] – Section 3, for instance, elaborates on concrete heartbeat targets.

Each object replica f is associated with a heartbeat probability parameter $\epsilon(f) \in [0, 1]$. Each Folklore protocol period at node i, for each locally stashed object f, node i sends a heartbeat message to a target node with probability $\epsilon(f)$. The target node is typically the node that introduced the object, but can also be alternately specified by the software layer above Folklore, when the object is introduced. In Reactive Folklore (§3), one possible target node is a special node that helps locate a replica of an object. The heartbeat message contains the object's $\langle id, version\rangle$, plus any other data the upper software layer wants to append.

$\epsilon(f)$ can be specified by the inserting node, to ensure that the heartbeat bandwidth, into the target node, is a constant. This incoming bandwidth is $hb = \epsilon(f) \cdot N \cdot \frac{1-\frac{\gamma}{\beta}}{1+\frac{\gamma}{\alpha}}/T$, where N is the number of nodes in the system. Size estimation algorithms, e.g., [25], can be used to estimate N.

Heartbeats can be used to locate a fraction of stashed replicas, and thus can enable

2. Folklore is implemented as a C library, allowing any distributed application to take advantage of Folklore's properties.

object retrieval (see §2.6) mechanisms without flooding a search or querying a DHT. Our previous system [26] used heartbeats for this purpose, where the target node was the author of the file. Section 3.4 explains a way to use them as stimulus to Reactive Folklore (§3).

Compromised Target Node: Consider an attack where a compromised target node has the power to immediately delete all heartbeating replicas it hears. Our protocol prevents destruction of the object. In the worst case, if there are very few replicas left ($S_t \ll N$), then we can write the expected change in replicas, after one protocol period, as $\Delta S_t = (S_t - \epsilon \cdot S_t) \cdot \frac{\beta}{2} - [1-(1-\epsilon)(1-\gamma)]S_t$; in other words, this is the new replicas created while under attack minus the replicas that were changed to averse or were deleted by the attacker. To sustain replicas we want $\Delta S_t > 0$, which means that $\epsilon < \frac{\beta/2-\gamma}{\beta/2+1-\gamma}$. As an example, with our experimental parameters $\gamma = 0.4, \beta = 10$, gives us $\epsilon(f) < 0.82$. Our experiments use $\epsilon(f) = 0.1$.

2.6 Propagating Object Updates

Object updates are propagated in Folklore in two ways: **(1)** the heartbeat target node looks at the heartbeat messages – if the version number therein is smaller than the local one, an update is sent to the heartbeating node; **(2)** in addition, updates also propagate through the *stash-receptive contacts*, as described in Section 2.3.

It is interesting to compare which of (1) or (2) transfers updates faster. Suppose the stable number of stashers for an object is S_∞, and the average number of heartbeats received per protocol period is hb. Then, the number of updated stashers S_i, after i rounds since the update started, satisfies $(S_\infty - S_i) = \quad (S_\infty - S_{i-1})[(1-\frac{hb}{S_\infty})(1-\frac{\beta.S_{i-1}}{2N})(1-\frac{1}{N})^{\beta.S_{i-1}/2}]$; in other words, the number of replicas left to update decrease with the number of old-replicas that either send a heartbeat, or contact an updated replica, or are contacted by an updated replica. A quick simulation of this equation shows that the last two terms (hence (2) above) dominate over heartbeating ((1) above). Thus, updates are distributed faster in a decentralized manner, yet the heartbeating to the target ensures *eventual* propagation of the update (i.e., 100% guarantee).

Update Optimization: To reduce undue bandwidth usage, if the difference in version numbers, between two replicas at two nodes, is 1, then Folklore does not transfer the entire object as an update. Instead, a log of the update is transferred – this log identifies modified blocks, new blocks, and deleted blocks, thus reflecting the update.

Data Consistency: Folklore is agnostic to data retrieval mechanisms. One way to retrieve data is to let the object's introducer learn the location of a replica using the heartbeat mechanism, thus allowing it to immediately download the replica (as done by the creator and owner of a file in [26]). In this case, the downloaded replica is not guaranteed to match the latest version of the object. However, due to the way updates are propagated, the introducer is likely to receive a heartbeat from a current replica within T units of time. Therefore, if the rate of updates is $< \frac{c}{T}$, then most reads will return the latest version of the object. R-Folklore provides a more sophisticated way to retrieve data.

2.7 Membership Management

Folklore maintains an unstructured overlay among its nodes. The overlay provides nodes with a random selection of targets, thus supporting both the stash-receptive contact described in Section 2.3, and data untraceability. Each node "knows" a small list of a few other nodes in the system – we call this the "partial membership list" at that node, since we do not require nodes to know about all other nodes in the system.

The membership maintenance is not for specific objects, but for the system as a whole. Each Folklore node runs a decentralized membership protocol that seeks to **(1)** have this partial membership list always represent a random sample from across the global set of nodes; **(2)** continuously update and manage the list in the face of node joins, leaves, and failures; and **(3)** in order to prevent an attacker from gaining a hold on certain membership lists by populating it with malicious entries (e.g., as described in LOCKSS [1] or through a Sybil attack [19]), the partial membership list at each node is continuously changed, even when there are no node failures or joins in the system.

The partial membership list (henceforth the membership list) contains a maximum number of entries (chosen as $log($upper bound on size of system), e.g., the upper bound is chosen as $100,000$ in our experiments). In addition, some of these entries might be marked as *suspected*, referring to a node that is suspected to have failed in the recent past. Notice that by failure here, we do not mean a Byzantine failure, but instead refer to a crash-stop failure or departure from the system.

Some of the operations on the membership list are performed periodically – we elaborate individual operations below. The membership management's protocol period is unrelated to Folklore's protocol period, and synchronization across nodes is not required.

Joining: A new node sends a `JOIN` message to any one node currently in Folklore, which in turn replies back with a `JOINREP`(ly) message that includes that node's current membership list, but with suspected entries left out. In addition, the second node adds the new node to either an empty slot in its list, or replaces a suspected entry. If there are no empty slots or suspected entries, then a random entry is dropped to make space for the new node.

Ping and Ping-Reply: A given node periodically pings a small, random subset of its membership list. If a ping-reply is not received from a ping-ed neighbor, before the start of the next membership's protocol period, that neighbor entry is marked as *suspected*. Suspected entries are not considered in ping target selection, but are kept around because the suspected node may not have failed and may yet send a message (e.g., through a stash-receptive contact).

The membership protocol *promiscuously* listens to other messages received by the node (e.g., ping, stash, stash-receptive, etc.), and updates the membership list opportunistically. If no empty slots are available to update the list, a suspected entry can be dropped. If nothing is heard from a suspected entry for a second protocol period, then that entry is deleted.

The above strategy might appear to be aggressive, but they follow from our requirements (1)-(3) above. While there is no need to maintain a membership list that is "correct" at all times, there is a need to continuously change this membership list – we call the latter property as *partial-view fluidity*. In addition, since suspected entries are kept around, they can be rejuvenated if the suspected node does initiate contact before a second membership protocol period expires.

Enforcing Partial-view Fluidity: We note that fluidity in the membership list improves replica untraceability, since finding a replica of some object at a node i will not necessarily mean the node's current neighbors hold more replicas of that object, i.e., that they had a stash-receptive contact with node i in the past. In addition, we also take the following actions to increase this fluidity. If a node spends more than c membership protocol periods (c being a parameter, set below) without adding a new peer to its list, then it requests the membership list from a random non-suspected neighbor. New entries from this list are used to fill empty or suspected slots in the local membership list, and then a small fraction of existing un-suspected nodes is randomly picked and replaced with entries from the received membership list. This is similar to the LOCKS churning of neighbor lists [1].

Although the membership protocol runs periodically, our implementation requires it to have a higher frequency than the basic endemic protocol's frequency. This is to ensure that malicious nodes have a low likelihood of populating a membership list with malicious entries soon enough to disrupt the next stash advertisement. Let T_M be the period of the membership protocol (e.g., 90 seconds), and T be the period of the endemic protocol (e.g., 360 seconds). We introduce a new parameter c, where $c > 1$, and T_M must be chosen in such a way that $c \cdot T_M < T$ (e.g., $c = 3$), to ensure that the membership will change between executions of the stash-receptive contact (e.g., by going to at least 3 T_M).

Finally, note that our membership mechanism can be replaced by any other mechanism that provides **(1)** a partial view of the system, **(2)** nodes that are chosen uniformly at random, **(3)** a continuously changing view. There are a number of systems that meet such requirements, e.g., CYCLON [16], AVMON's coarse view [27] and others.

3 Design of R-Folklore System

In this section we present the design of Reactive Folklore (R-Folklore). This effort builds on top of Proactive Folklore and implements a distributed storage system. R-Folklore is useful when data-untraceable, efficient, and self-organizing data-availability is required. It is implemented as a C library that in turns calls Proactive Folklore, and can be used by itself or as part of another system, e.g., distributed file backup.

R-Folklore's design goals are **(1)** lower bandwidth usage than Proactive Folklore, **(2)** on-demand endemic (i.e., the endemic can be running or not), **(3)** low probability of

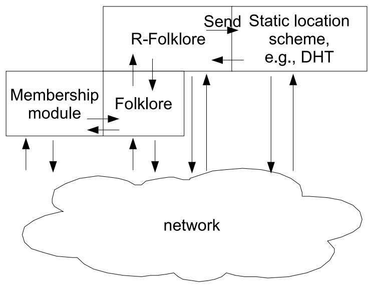


Fig. 4. R-Folklore sends messages through the DHT, Folklore, and directly to the network. Any message routed to a node, through the DHT, is exclusively handled by R-Folklore. As the figure suggest, the membership mechanism could be replaced as long as the new one provides the interface required by Folklore and a random and changing view of the overlay.

losing a file if the endemic is not running (i.e., when files are static), and **(4)** run-time reconfiguration of the endemic. R-Folklore autonomically migrates files when threat of attack is sensed. It provides data privacy and data confidentiality, efficient file read performance, efficient update performance, and multi-user file update.

Figure 4 shows the components of the system and their relationships. The figure shows that R-Folklore takes advantage of a Distributed Hash Table (DHT) to do file location. In the following sections it will be clear how this does not negatively affect data-untraceability. Any existing DHT such as Chord, Pastry, CAN, etc., can be used.

Below we describe how R-Folklore does file insertion (§3.1), file retrieval (§3.2), file updates (§3.3), and we discuss how existing attack-detection mechanisms can be used as a stimulus into the R-Folklore system in order to trigger migration (§3.4).

3.1 File Insertion

To insert a file, R-Folklore takes advantage of a static location scheme (the DHT), and Folklore. The former is used for fast location through a global id, and the latter for the availability it provides under proactive replication. The following paragraphs describe how a file is manipulated when it is inserted.

The first set of operations are taken to provide privacy and improve availability.

To insert a file, the inserting node **(1)** splits the file into blocks of size B (we use 128 KB); **(2)** encrypts each of these blocks using a symmetric key[3]; **(3)** performs erasure coding to the entire encrypted file, transforming it into a set of *erasure packets*; **(4)** inserts each of the erasure packets as a separate Folklore-object (R-Folklore calls `folklore_put(packet_i)`), giving each one a globally unique identifier (in the following paragraphs we describe how).

Step (2) (symmetric encryption) adds privacy and confidentiality. This form of encryption is also faster than using public/private key encryption. Step (3) (erasure coding) improves availability and helps reduce bandwidth – details are below.

Erasure Coding [12], [28]: This technique breaks each (encrypted) file block into x *packets*, and then re-encodes it into y *erasure packets* (we use $y = 8$). The value of y/x is called the *redundancy factor* (we use a redundancy factor of 2, giving us a packet size of 32KB). Erasure coding provides improved file block availability, thus improved file availability. To recover a block, it is sufficient to locate and then decode any x of the y erasure packets.

For instance, a redundancy factor of $y/x = 2$ means that a 10MB file has a total storage overhead of 20MB. This overhead is worth the increased availability – when retrieving a file, it suffices to locate any 4 of the 8 erasure packets of a block.

For a file with b (encrypted) blocks, erasure coding results in $b \cdot y$ packets. Each of these erasure packets is named separately and then passed separately into both the DHT and Folklore (§2.2).

Splitting a file into equal-sized blocks, which are then encoded into equal-sized packets, means that when the file becomes migratory (later in R-Folklore) *the per-node bandwidth due to migratory replication is a constant quantity, independent of the number of nodes in the system and the average file size in the system*. This is because this bandwidth value is directly proportional to the size of the objects handled by Folklore (which is fixed).

Global ids: To globally identify the file being introduced, say foo.txt, its global id is computed by the Folklore layer (§2.2), using the filename as part of the input. Each

3. In comparison to encrypting the entire file, this approach allows us to do partial update propagation rather efficiently.

packet's global id is also computed by the Folklore layer. To that end, R-Folklore hands the following tuple to Folklore as the packet's "packet-filename:" <foo.txt's global id, file-block number, erasure packet number>. All these id's are used for identification in the DHT.

Master Block: Finally, in order to facilitate efficient retrieval of a file, a separate *master block* is created for the file in question. This master block is passed to both the Folklore layer and the DHT. The master block's id is the file's global id. The block's contents are the number of blocks b of the file, the redundancy y/x of the encoding, and the global id for each of the previously built packets. Moreover, each packet record contains the number of the file-block encoded by the packet, and the version number for that block (§3.3).

Signatures: R-Folklore provides data integrity, during file insertion, in the following way: **(1)** Each erasure packet and master block (i.e., each *R-Folklore object*) is signed by its creator. The signature of an object is computed using the object's data and metadata. **(2)** The public key of the file-inserting-user is transferred with each packet and master block, thus allowing verification of the signature – this suffices since user anonymity is not a goal. **(3)** A mini-certificate is also transferred with the object – this is a trusted third party signature of the file-inserting-user's public key.

A node's Folklore layer receives object replicas, i.e., erasure packet or master block replica, either from initial replication (§2.2) or endemic propagation. After such an event, Folklore upcalls R-Folklore, prompting **(1)** the verification of the object's signature using the attached public key, and **(2)** the verification of the mini-certificate using the trusted third-party's public key (which we assume is preinstalled at each node). If the verification is OK, R-Folklore lets Folklore store the replica, otherwise it is not accepted.

DHT Assumptions: In our design we assume that the static location scheme used is a DHT. Furthermore, the DHT is capable of efficiently mapping a key, and messages for that key are routed to a node in the overlay. If the overlay changes, then messages to the key may be routed to a new node. Erasure packet ids and master block ids are used as DHT keys.

We call a node *responsible node* (of a packet or master block) if it is the DHT's routing destination for the key that identifies a packet or master block. The responsible node will cache a replica of the packet or master block. During file introduction, responsible nodes for the erasure packets and the master block are immediately contacted.

For instance, in the Chord DHT, the responsible node is the node in the Chord ring whose key is immediately larger than the packet/master-block key, i.e., the so called successor node. In the Pastry DHT, the responsible node is the node whose key is numerically closer to the packet/master-block key among all nodes.

3.2 File Retrieval: Building a File From the Network

Our file retrieval scheme needs to work for both the case when the file's components are currently statically located, and the case when they are currently migratory. To bridge this gap, we use the *responsible node* for the master block of the file. Having a copy of the master block – usually cached during the file introduction procedure – means that the resonsible node has the knowledge to rebuild the file. If the responsible node fails, or changes because the DHT reconfigured itself, replica heartbeating (§2.5) offers an efficient way to ensure that the new responsible node will have a cached copy.

Heartbeating: Folklore's periodic heartbeat described in Section 2.5 is used to ensure that the responsible node of a master block or erasure packet will have a cached copy. When any of those objects sends a heartbeat, the target node is the resposible node of the object. If a node receives a heartbeat from an object, and the node is indeed responsible for the object, it can retrieve a copy to keep a local cache of the object, unless the object was already cached.

Building a File: When a node wants to retrieve a file identified by id, R-Folklore locates the master block by sending a request, through the DHT, to that block's responsible node. Recall that the master block's DHT-key is the file's id. This responsible node *build*s the file from the network, if it was not previously built, then sends it back to the querying node.

To "build" a file, the master block's responsible node retrieves that file's erasure packets from the network. Because the erasure packet's ids are recorded in the master lock, each packet can be requested through the DHT. As soon as enough packets have arrived, the node can start decoding to recover individual file blocks.

Note that *the master block is not a single point of failure*, since it is also replicated by the Folklore system, and has a self-stabilized number of replicas. Furthermore, file building can be enabled even if the master block is lost from the system: erasure-packet-holding nodes also send heartbeats to the responsible node for the master block, but at a very low rate, from them the master block can be rebuilt. The heartbeat includes the Folklore id of the master block, the Folklore object id of the packet, the packet index, and the block index. These heartbeats are verified, including a signature of the message, precomputed when the file was created, and the previously mentioned mini-certification. This latter mechanism is not implemented currently in Folklore, because losing master block replicas is not the common case.

Like the master block, the responsible node is not a single point of failure. Attacking a responsible node cannot eliminate a packet or master block from the network, because the responsible node only keeps a cached copy, independent of Folklore's migratory replicas. Furthermore, DHTs reconfigure themselves after a node fails, thus there will always be a replacement for the responsible node.

Data Untraceability is Preserved: Although R-Folklore locates a packet or master block by querying a DHT, the query arrives at a node that keeps an R-Folklore-cached replica of the object. Eliminating that node, and the cached replica, has no effect on the replicas that are under endemic replication. Furthermore, compromising the responsible node for an object does not reveal information about the current location of all replicas of the object, only about a small fraction (due to the received heartbeats). Finally, destroying those replicas, if they have not already migrated, will have a negligible effect on the endemic protocol.

3.3 File Updates

Recall that a file is represented by the erasure packets of each file block, and by the master block that identifies those packets – this happens via their id, the number of the file-block they encode, and the version number of the block. Therefore, updating a file entails the creation of the erasure packets that encode modified blocks, and the modification of the master block to reflect the update.

The following steps are taken to modify the master block. For each modified block, the version number of the packets that encode the block is increased in the master block. For each new block, new records are created inside the master block for each erasure packet encoding the block.

Finally, the new master block and the new packets are inserted to the network. This will result in the respective responsible nodes caching the newer version, and the endemic replicas being updated by by Folklore's object update propagation (§2.6).

To avoid malicious updates, R-Folklore implements security policies via an upcall from Folklore triggered by the receipt of an updated replica. R-Folklore checks that the object and its version are signed by the same public key that signed the older object. If it is not, the update is ignored, otherwise it is applied. If the responsible node maliciously tries to reject a valid update, it can be detected and then either blacklisted or potentially forced out of the DHT. The detection is possible because a mangled master block or the continuous use of an old master block will be noticed – it would be easier for a malicious responsible node to merely erase all the objects it receives. We do not evaluate these detection schemes in this paper.

Data Consistency: Due to the serialization at the responsible node and the master block, R-Folklore guarantees that retrieving a file will yield the latest version. Since all update operations contact the responsible node of the master block, R-Folklore naturally serializes concurrent updates. Aditionally, application-level mechanisms can easily be defined atop R-Folklore to provide other serialization policies.

Versioned Multi-user updates: To implement multi-user updates, we choose not to replace old packets and old master blocks, instead allowing for retrieval of historic data, as in systems like [2], [29], [30]. Master blocks under this scenario include the public keys of users that are allowed to do updates. When a user creates a new version of a file, a new master block will be created, with a new version number, and new id. To do this, the user needs the symmetric encryption key from the creator of the file. The new master block will be signed by the updater, and its updated content will include: **(1)** incremented version number for packets that encode modified blocks, **(2)** entries for packets that encode new file blocks (if any), **(3)** ids of packets that encode deleted blocks if the file decreased in size. Under a multi-user scenario, the packets encoding an updated block will not use the ids of the packets that encode the previous version of that block. Instead, their id will be computed, using SHA1, from the original packet id and their version number. The id for the new master block follows the same rule. The new master block will be signed by the user that created it and will include the user's mini-certificate. This provides a versioning system somewhat like Parallax [29].

3.4 Using Existing Attack-Detection Schemes as a Stimulus for R-Folklore

Existing attack-detection schemes such as intrusion detectors [20], DDoS attack detectors [21], [22], etc. can be used to trigger the migration of a previously-static file. We briefly detail below the design decisions involved in building various alarm-detection schemes into R-Folklore. However, to maintain focus in the paper, we leave out the experimental evaluation of these myriad interactions.

1. *Detection by Piggybacking on Heartbeat Messages:* In this scheme, we assume that the attack detection scheme reports a "threat value". Concrete measurement of the threat level could be done through an intrusion detector such as BackTracker [20], or by gateway techniques for measuring the likelihood of being under a DDoS attack or measuring backscatter [21], [22][4].

A node storing a replica continuously piggybacks the perceived threat value, at that node, on Folklore heartbeat messages. The heartbeat target(s) (e.g., the replica's responsible node) continuously monitors the received threat values, and could raise an alarm when either **(a)** k of the threat values crosses a threshold, or **(b)** the average of the threat values crosses a threshold, where the average is computed from all heartbeats received in the last Folklore protocol period. k is a fraction of the heartbeats that are expected on a Folklore protocol period, e.g., k can equal four and the number of expected heartbeats can be eight. These parameters are tuned depending on the application and the environment.

Once an alarm has been raised, it is sent explicitly and immediately to the responsible node for the file, which in turn passes it along to all the responsible nodes for each erasure packet for this file (obtained from the master block it stores). All nodes receiving

4. We are only aware of one work [23] so far that promotes replication as a way to combat DDoS.

the alarm will transition to executing the Folklore migratory replication protocol. This completes the response to the attack stimulus.

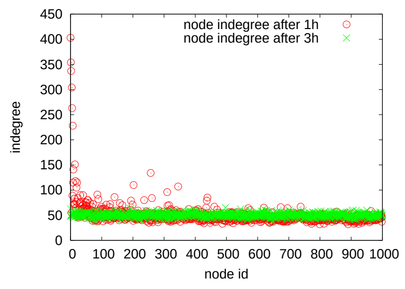
The threshold parameters specified above can also be changed online, by the user that created the file, depending on how sensitive the data is at some time, and depending on the perceived expected kind of attacks. To set the parameters, `parameter-set` message is routed through the DHT using the id of the master block as the key. This message is digitally signed by the user, and includes the mini-certificate.

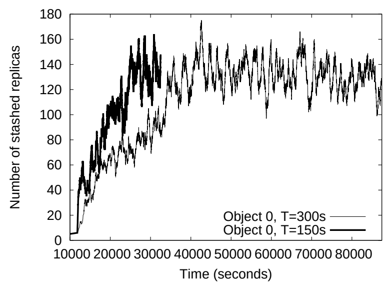
Finally, note that the responsible node is a good heartbeat target – if the reponsible node fails, the DHT will transparently reconfigure itself enabling a new target node. Note that resilience can also be improved by having multiple target nodes, e.g., the responsible nodes for a number of packets could also receive heartbeats.

2. *Detection by Measuring the Heartbeat Rate:* Another major stimulus is the heartbeat itself. Recall that the average number of heartbeats received during a protocol period is known. If the actual number of heartbeats received during x consecutive periods falls below a user-configured threshold, then an alarm is issued, which will start the endemic, or speed it up. The endemic is sped up by decreasing Folklore's protocol period across all nodes in the endemic.

3. *Other Detection Mechanisms (Byzantine Fault-tolerance and LOCKSS):* Byzantine agreement protocols and voting protocols could also be used as detection mechanisms. If replica-holding nodes know partially about each other, a *subset* of these nodes periodically run a Byzantine agreement protocol [17], [18] to agree on the integrity of that replica. The latest results of the agreement can be sent as the perceived threat value. Along the same lines, a voting protocol such as LOCKSS [1] could be used to measure the perceived threat. One downside of both these schemes is that membership entries are correlated with replica locatios, and this may allow an attacker to trace files easier than previously. A study of the tradeoff between untraceability and detection is beyond the scope of this paper.

Finally, note that R-Folklore uses the intensity of the alarm to adjust Folklore protocol parameters α, β, γ as well as the duration of the protocol period (Section 2.1). This is done to adjust the speed of the migration to the perceived threat. These parameters can be changed either *pessimistically*, i.e., for all files in the system, or more *conservatively*,





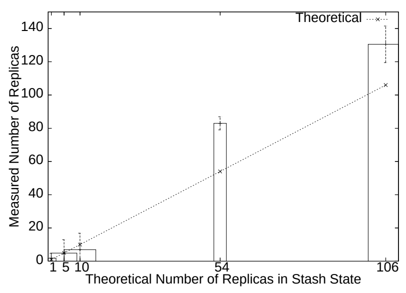


Fig. 5. **Simulation I: Indegree.** As time advances, the indegree of the membership lists gets uniformly distributed.

Fig. 6. **Simulation II: Live replica propagation** with 1000 objects in the network. The theoretical number of stashed replicas is 120. This shows what happened to object 0. The rest of the objects behaved similarly. A halved protocol period means twice as fast convergence (this plot is cut at $t = 32500s$ for clarity).

Fig. 7. **Simulation III: Measured vs Theoretical Stash.** There is a close relation between the values predicted by the equation, and the measured number of replicas in stash state from the system. Even a number as small as 1 expected replica in stash state works (if there is no churn).

i.e., for only the file group sensed under attack. Our implementation uses the pessimistic approach. When the threat disappears, the endemic replication could either go back to conservative parameters, or it could stop altogether.

4 Simulation Results

Before presenting our deployment results, this section presents larger-scale experiments from a discrete event simulation of Proactive Folklore (Figures 5, 6, 7, 8, 9, 10, 12, 13) and R-Folklore (Figures 5, 15, 16). The simulations were run on a 3GHz P4, 1GB RAM, Linux PC. By default, the system has 1000 nodes, with a protocol period $T = 5$ minutes, averse to receptive conversion rate $\alpha = 0.05$, stasher to averse conversion rate $\gamma = 0.4$, and each node sends 5 stash advertisements every protocol period ($\beta = 10$). R-Folklore simulations assume eight encoding packets, a redundancy factor of two, and a 512KB file with 128KB blocks, giving a total of 32 packets, each one 32KB long.

Note that Proactive Folklore's results apply to R-Folklore in general. When R-Folklore calls Proactive Folklore to introduce an encoding packet or a master block in the endemic, this becomes an *object* in Proactive Folklore's terminology.

Membership Indegree: Figure 5 shows the membership protocol (Section 2.7) indegree at each node. The membership protocol was parameterized to maintain a view of 50 peers at each node (outdegree), and with a 2 minute period. The experiment is set so that nodes join the network sequentially, in order of id, with a 5 second separation. This biases the indegree towards low id nodes, and that can be seen in the graph for the 1h values. After 3h, however, even with the conservative parameters used, the indegree is distributed uniformly. Therefore, the membership protocol can be used effectively by Folklore.

Proactive Folklore's Object Propagation: At time $t = 10000$s, 1000 objects are simultaneously inserted, each from a different node. The simultaneous insertion maximizes competition among replicas. The number of replicas in stash state for each object are plotted during 24 simulated hours. Figure 6 shows how the number of replicas (for object 0) varies over time; similar trends were observed for the other objects. The stable theoretical number of replicas in stash state is 120. The chaotic behavior of the protocol around this stable value is evident from Figure 6. The same experiment, run with half the standard protocol period ($T = 2:30$ min), shows twice as fast convergence proportional to T's reduction. Finally, note that file operations can start after a few replicas are created, well before convergence (e.g., within 5 minutes in this experiment).

Proactive Folklore's Theoretical vs Measured Stash: In these experiments, we compare the theoretical number of replicas in stash state versus the measured number of replicas in stash state. Just like the previous experiment, the simulation is done with 1000 nodes, and 1000 objects are inserted simultaneously. At the end of the simulation we take the number of replicas in stash state of each object, and use that snapshot value to compute an average. The protocol parameters were varied to experiment with the following theoretical number of replicas in stash state: 1, 5, 10, 54, 106. The experiments with the three smallest expected values were done with a system modification: all replicas are deleted as soon as they leave the averse state, i.e., there are no virtual transfers. This was done to test these small values under harsher conditions. As Figure 7 shows, the protocol is capable of maintaining an expected number of stashers as small as 1. In practice such a small value would not be advisable if node churn is expected. The experiments where 54 and 106 replicas in stash state are expected have a higher measured value, about 30 more replicas than expected on both. For comparison, PAST [11] and CFS [31] use 32 - 64 replicas per object. This is because virtual transfers were not disabled for them and the simulated disk sizes are big enough not to cause deletion of replicas that reach the receptive state. Therefore, the stash advertisement is capable of moving many receptive replicas to stash on node contact (many virtual transfers are performed on each contact).

Proactive Folklore's Endemic Propagation Under Churn: We simulated a Folklore network with 526 nodes, using churn traces from the Overnet network. Traces are injected continuously (i.e., every 20 minutes). The traces provide information on the status of each node. Each node introduces one object to the network. Parameters T, α, β, and γ are unchanged. On average, almost half of the nodes were offline at any point in time. Figure 8 shows that the migratory replication does not lose the file under churn. The two plotted lines diverge when the simulation starts injecting the churn traces (at time $t = 18557$). Although only one object is shown, we found similar behavior for all other objects. The expected number of replicas in stash state with no churn is around 80 for this scenario; this number halved under churn, but no object was lost and a new equilibrium point for stash replicas was found by the migratory replication.

Stressing Proactive Folklore's Endemic Under Churn: We test how many objects survive after one day of churn, if protocol parameters are set so that the expected number of nodes holding a replica in stash state are 6, 13, 20 and 28 respectively. The setting is the same as in the previous experiment. As Figure 9 shows, all objects survive the Overnet traces, for up to 24 hours, if 28 of the nodes are expected to have a replica in stash state. If 6 of the nodes are expected to have a replica in stash state, a little over 50% of the objects still survive after one day of churn, due to the chaotic nature of the endemic.

Proactive Folklore's Object Update Propagation: Figure 10 shows the time it took for a typical update to propagate to all its replicas in stash state. It took almost 70 minutes (or ≈ 14 protocol periods) for the update to replace all old replicas. On average, 99% of the updates were propagated by stash-receptive contacts. This indicates the update propagation through heartbeats would not overload the node that receives the heartbeats from the replicas.

To see the effect that β and γ have on update propagation time, we numerically

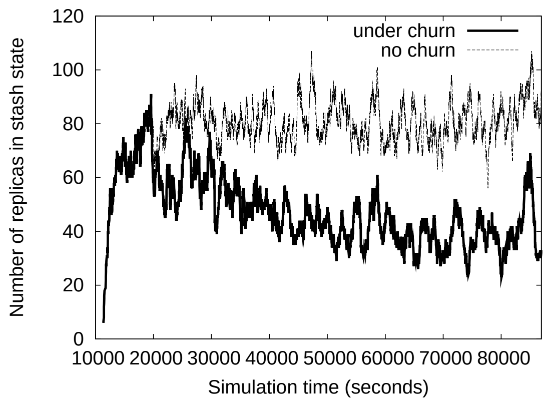


Fig. 8. **Simulation IV: Churn-resistance.** The protocol is resistant to churn. The injected Overnet traces did not destroy the file – instead, the protocol automatically converges to a value that reflects the lowered host availability.

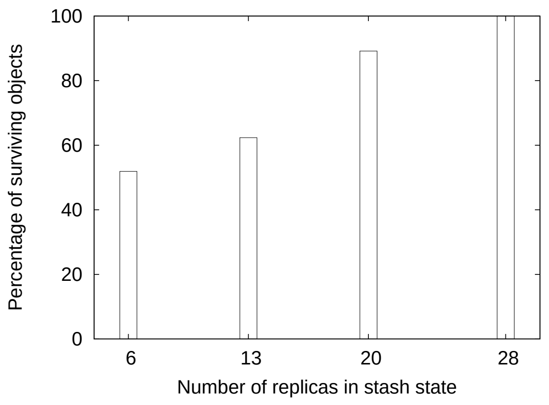


Fig. 9. **Simulation V: Surviving Objects vs. Expected Stashers.** Depending on the level of churn and the expected number of replicas in stash state, there is a point where objects will be lost from the endemic environment.

simulate the recurrence shown in Section 2.6, using $N = 100000$. Under "Normal", we use the same parameters as in the previous experiment, resulting in $S_\infty \approx 10666$ – notice that it takes 13 protocol periods to propagte the update. Doubling the number of contacts per period ("Beta x2") shows faster convergence as expected, which is also afected by the resulting larger $S_\infty \approx 10888$. Finally, as doubling γ ("Gamma x2") halves $S_\infty \approx 5411$, the propagation becomes slightly slower.

Below we experimentally study the effect on Folklore of three kinds of attacks: deletion, bad protocol parameters, over-replication. For simplicity, three of these experiments consider only one object, but the last one evaluates up to 12 objects per node.

Attack: Replica Destruction in Proactive Folklore: Malicious nodes in the network could each delete all the replicas it receives, or may tamper with some of them, i.e., modify its contents arbitrarily. If digital signatures are used, the tampering problem reduces to replica deletion[5]. In this experiment, malicious nodes participate in stash-receptive contacts, but they delete all replicas they receive. Figure 12 shows how the expected number of stashed replicas for an object behaves under different fractions of

5. Note that a Byzantine protocol can be used to avoid replica corruption, if no digital signatures are used, as explained in the Stimulus section 3.4.

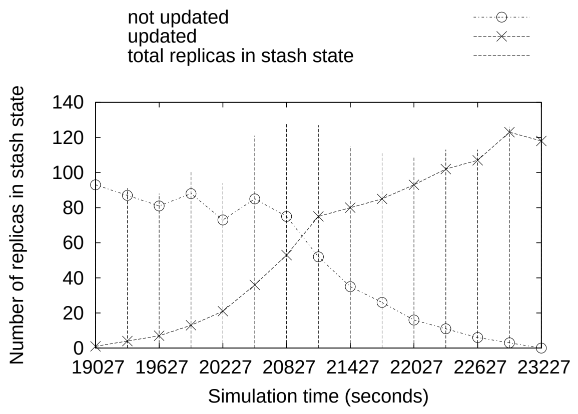


Fig. 10. **Simulation VI: Update propagation.** The graph shows the relationship of old replicas and new replicas as the update propagates. The majority of the updates were through endemic replication.

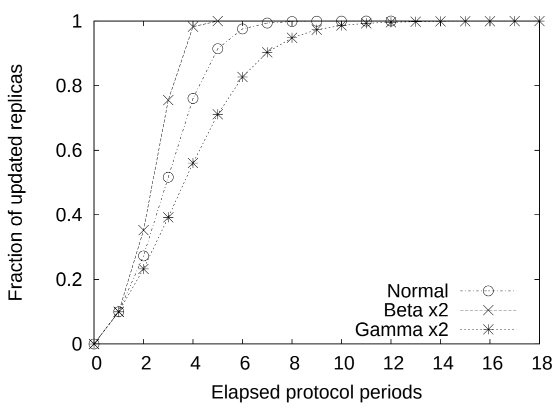


Fig. 11. **Numerical Result I: Update propagation with changed parameters.** With $N = 100000$, $S_\infty \approx 10666$ ("Normal" plot), all replicas are updated after 13 periods. Halving S_∞ by doubling γ shows a small increase. Doubling β reduces the propagation time as expected.

malicious nodes. Even with 50% bad nodes, from the 1000 node network, the number of replicas is still 50. This is due to the endemic protocol converging to a new number of expected replicas, under the interaction of good nodes. Note that good nodes do not know which nodes are bad – any node could be selected as stash-advertisement target.

Attack: Bad Protocol Parameters in Proactive Folklore: Malicious nodes could also use arbitrary values for protocol parameters α and γ. We see on Figure 12 that as the number of nodes with arbitrary parameter values increases, the number of replicas of a given object increases. With 50% bad nodes on the network the number of stashers doubles. As our concern is availability of the object, this is a reasonable result.

Attack: Over-replication in Proactive Folklore: A misbehaving or selfish node could also attempt to over-replicate an object. In this experiment, a number of nodes each locks their replica for object i in the stash state and changes the Folklore protocol period down to 30s (instead of 5 minutes). Figure 13 shows that the increase in stashed replicas is close to the number of nodes trying to over-replicate the object. This means that increased frequencies for the endemic do not cause over-replication at good (i.e., non-over-replicating) nodes. This resilience arises from the presence of the averse state at

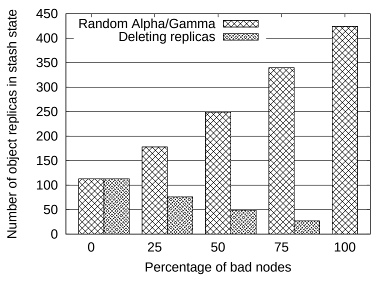


Fig. 12. **Simulation VII & VIII: Effect of Malicious Nodes & Effect of bad protocol parameters.** "Deleting replicas" shows that Folklore attains a stable number of replicas when a large number of nodes delete replicas. "Random Alpha/Gamma" shows the behavior when a fraction of nodes have random values for α and γ.

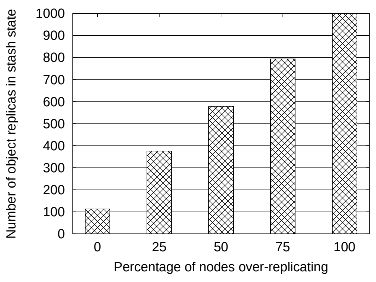


Fig. 13. **Simulation IX: Effect of over-replicating nodes.** We see in this experiment that nodes trying to maliciously overreplicate have little success, due to the behavior of the averse state on well-behaved nodes.

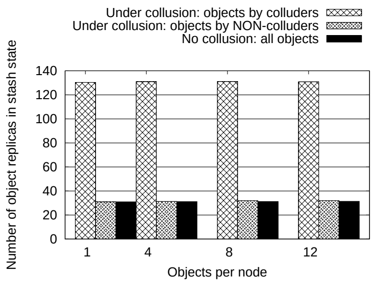


Fig. 14. **Simulation IX (b): Effect of over-replicating nodes and many objects.** 50% out of $N = 250$ nodes over-replicate. The plot shows results for colluding and non-colluding behavior.

good nodes.

To determine if objects introduced by over-relicating nodes can starve the objects of well-behaved nodes, we experimented allowing each node to introduce $1, 4, 8,$ or 12 objects, and having 50% of the nodes over-replicating. The over-replicating nodes can collude or work alone. Here $N = 250$, and $S_\infty \approx 27$. Figure 14 shows that no starvation occurs. If over-replicating nodes do not collude, no one benefits, although convergence to S_∞ is faster. When they do collude, we see that the number of replicas, from objects inserted by over-replicating nodes, goes up to 125 as predicted by our previous result. However, replicas from normal nodes do not starve. Also notice that results are almost equal as the number of objects grows. Further experiments performed under severe storage limitations per node showed degradation in the number of stashed replicas overall (as would invariably happen), but we found that no object from well-behaved nodes were lost. Finally, Figure 13 shows that additional resource usage due to attacks is small, e.g., at 50% overreplicating nodes, only an additional 90 (9%), beyond the number of attackers, is stashing the object.

From Stimulus to Response in R-Folklore: Next we experiment with R-Folklore to

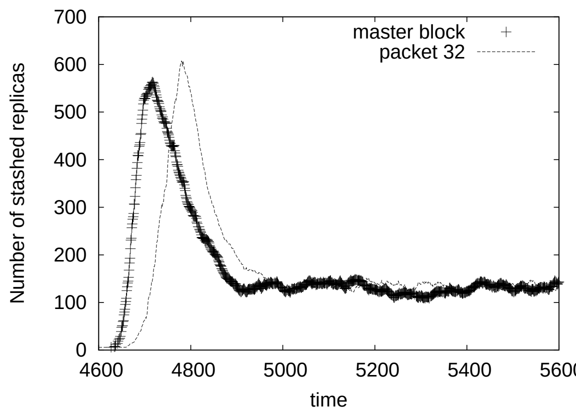


Fig. 15. **Simulation X: Insertion Speed.** After a heartbeat with a stimulus above the threshold arrives, the endemic is initiated for the master block, and it informs the 32 packets that conform the file. With $T = 30$, it takes 5 minutes for the last packet to reach safe endemic replication. **This plot closely matches the deployment results in Figure 23.**

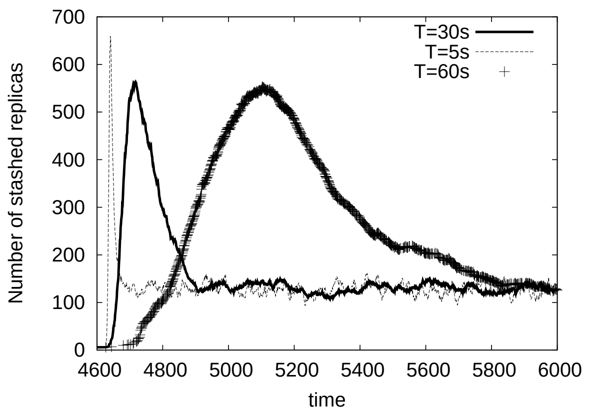


Fig. 16. **Simulation XI: Effect of varying Protocol Periods.** Here we compare how fast a master packet converges to the expected number of replicas, depending on T, after being triggered by a heartbeat with a high threat value.

see how fast after an alarm is raised the components of a file reach safety, i.e., how fast the endemic replication converges to the expected number of replicas after the alarm. Parameters $\alpha = 0.05, \beta = 10, \gamma = 0.4$ are the same as for the previous experiments. For our first experiment, we have a file that is composed by 32 erasure packets and the master block. The simulated alarm is initiated at the node storing the master block, and it starts a moderately fast endemic, with $T = 30$s. The master block sequentially informs the responsible node for each packet to start the endemic replication. Figure 15 shows the detail for the master block and the 32nd packet (remaining in-between packets had similar plots). We found that all the components of the file converged to the expected number of replicas around 5 minutes after the heartbeat triggered the reaction. Figure 16 compares how fast a master block would reach the expected number of replicas for $T = 5$s, $T = 30$s, and $T = 60$s.

Finally, we note that Figure 15 matches very closely with the behavior from the deployment results shown in Figure 23.

5 Deployment Results

Our full-scale implementation of Proactive-Folklore and R-Folklore is 12,000+ lines of C code. For R-Folklore, we chose a lightweight Chord [32] implementation developed for the Internet Indirection Infrastructure [33] project. The erasure coding code used comes from ICSI, and implements the commonly used XOR-Based Erasure-Resilient Coding Scheme [34]. We use eight encoding packets and a redundancy factor of two, meaning that only four encoding packets are needed to rebuild a block. Files have a 128KB block size. Full file blocks result in eight encoding packets, 32KB each, due to a redundancy factor of two for the encoding.

The results below are from our deployment on the CSIL cluster at UIUC. CSIL is the undergraduate research lab at UIUC, and consists of Pentium IV machines over a 100 MBps LAN. Each machine runs Fedora 4 Linux, sharing a home directory through NFS. At the time of the experiments, only 64 machines were available.

File Insertion Behavior: For this evaluation we used a 459KB file. After encoding, the three first blocks result in 24 packets, 32KB each, and the last block in 8 packets, 18.8KB. The generated corresponding master block is only 930 bytes, and Proactive Folklore creates 8 initial replicas for it. The creation of each initial replica is synchronous, i.e., the inserting node waits for an acknowledgment from the peer after it stores the replica on disk. The whole encryption+encoding+initial replication process took (24.081 ± 1.312)s (avg. $\pm$ std), averaged over 10 runs. Figure 17 shows the *worst case* run. At the end of the run, notice the 8 ticks that are vertically stacked. These are the 8 extra initial forced replicas that Proactive Folklore creates for the master block. They appear vertical because Proactive Folklore's implementation pushes these replicas in a multithreaded fashion, after R-Folklore calls `folklore_put(..., masterBlock, 8)`. R-Folklore could be optimized to encode the file using one thread per block, and creating one new thread to insert each packet. This would change the sequential behavior of the packets in the graph, and greatly reduce the insertion time, which is a drawback of the current sequential implementation of the file block encoding. The insertion of a 2MB file (Figure 18) further shows that parallel packet insertion would improve performance. We estimate that insertion time would be halved because the network interface is currently

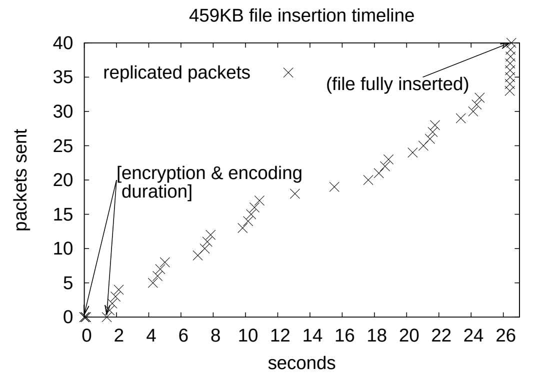


Fig. 17. **Deployment I: File Insertion.** The inserted file is encrypted, encoded, and the resulting 32 packets, plus the master block, are statically replicated into a random location.

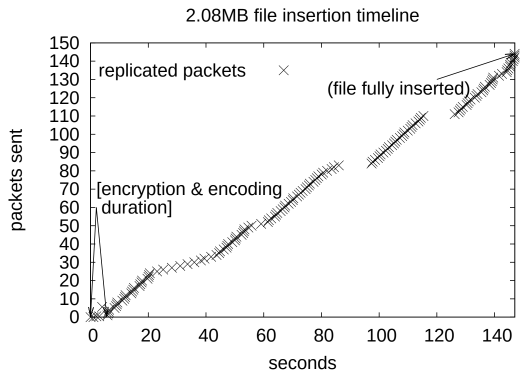


Fig. 18. **Deployment II: Larger File Insertion.** Here we show the insertion of a 2MB file. The sequential insertion in this figure and in Figure 17 is an artifact of R-Folklore's current sequential implementation of block encoding and packet insertion. Using multi-threading would halve the time.

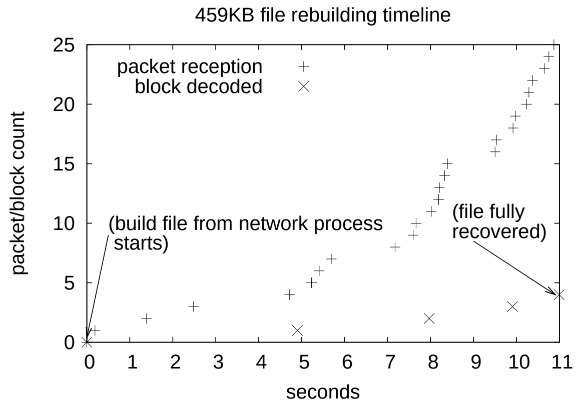


Fig. 19. **Deployment III: File Retrieval.** The first step to recover a file is to request packets that are not cached. In this test, only one of the 32 packets was cached. As soon as enough packets to decode a block are present, the block is decoded and written to disk. As the graph shows, the last block is ready before all the requested packets arrive.

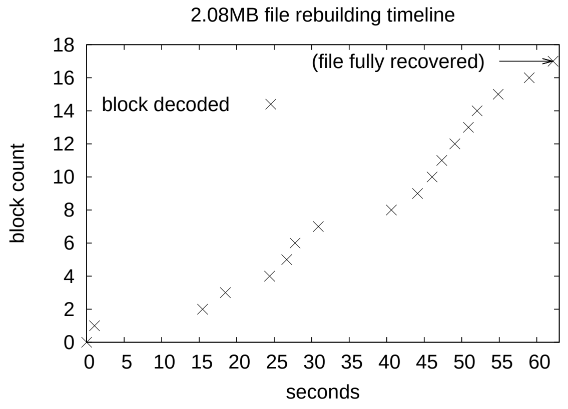


Fig. 20. **Deployment IV: Large File Retrieval.** This graph shows how the rebuilding of a 2MB file progresses. The file is divided into 17 blocks. The time to recover the file is an artifact of R-Folklore's sequential implementation of incoming packet handling. A multithreaded version would drastically reduce time.

underutilized on insertion.

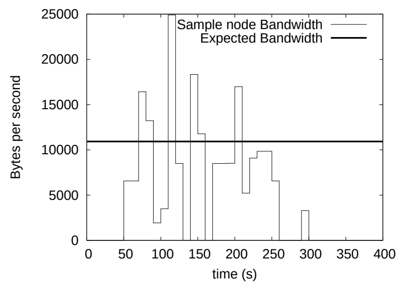
File Building Behavior: It took on average (10.968 ± 0.433)s to rebuild a file of size

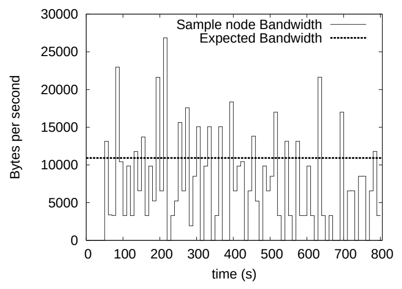
459 KB from the network. Figure 19 shows the build corresponding to the previous section example. Note that, due to erasure coding, the file is completely recovered before all 32 corresponding packets have been downloaded. In the evaluation scenario, the node rebuilding the file happened to be responsible for only one of the packets, so Figure 19 shows a worst case scenario. Figure 20 shows one instance of retrieving a 2.08MB file, which took 62.1756 seconds. As with the insertion, the build time can be improved by changing the implementation to allow blocks to be decoded in parallel. As in file insertion, our implementation is not optimized. Multi-threading on block decoding would drastically reduce file rebuild time.

Bandwidth Under Endemics: Here we evaluate the bandwidth used during the endemic replication of the 459KB file. The protocol period is $T = 30$s, and other protocol parameters are $\beta = 10$, $\gamma = 0.3$, $\alpha = 0.2$. There are 64 nodes in the network. Note that this period is shorter than what would be used in a real deployment, e.g. 5 min. It was chosen to stress that the endemic replication is not expensive. The theoretical bandwidth at each node, is *10.67KBps* (recall that the packet size is 32KB). During the first 5 minutes of the experiments, nodes transferred on average 1.71MB each, thus the bandwidth, *5.84kBps*, was just 55% of the expected theoretical. Note that during these early minutes, the Folklore protocol is working to converge, and all nodes are initially receptive to all replicas, there are initially no stashers and no averse, thus creating more replica transfers than what would be observed after the protocol converges. Figure 21 illustrates the usefulness of retaining a replica after it leaves the stash state. After the first 5 minutes the protocol continues to operate as expected, but with virtual transfers.

Figure 22 shows how one host behaves when replicas are deleted as soon as they reach the averse state. This experiment lasted for 12.5 minutes, and the average bandwidth used by each of the 64 nodes was *7.27KBps*.

Stash Behavior: For the previously described experiment, we counted the number of replicas on stash state for each of the 32 packets and the master block. The predicted stable number of stashed replicas is 23.67, and the curve in Figure 23 follows this number after the Folklore protocol stabilizes, which happens near the fifth minute.





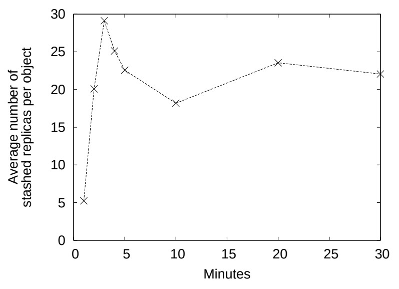


Fig. 21. **Deployment V: Bandwidth usage** at one of the nodes. This node made its last real replica transfer 5 minutes after it joined the network. From then on, transfers of stashed replicas are virtual (due to retained replicas). The horizontal line shows the theoretically-predicted bandwidth.

Fig. 22. **Deployment VI: Bandwidth use without file retention.** This graph shows the bandwidth used, by one node, to transfer replicas if replicas are not retained, i.e., if they are deleted as soon as they reach the averse state.

Fig. 23. **Deployment VII: Insertion Speed.** Average size of the stash for 32 packets and a master block generated from a 459KB file. After stabilization, near the fifth minute, the number of stashed replicas follow the expected theoretical value of 23.67. **This plot closely matches the simulation results in Figure 15.**

6 Conclusion

In this paper, we have argued that networked data stores need to provide an important new property called data untraceability. Without data untraceability, an attacker can find out all current replica locations for a given file, and target each, thus effectively deleting the file from the network. We present the design the Folklore system, which provides data untraceability as a consequence of endemic migratory replication, and thus prevents a file from being deleted by a malicious attacker. At the same time, Folklore provides efficient performance for file insertion, file reading, and file updating. For improved data location services, availability under no migratory replication, on-demand migration, and for incremental deployment of Folklore atop DHT's, we proposed a variant called R-Folklore. Our design and deployment have shown that data untraceability can be cleanly added to existing properties such as availability, integrity, and updatability.

References

[1] P. Maniatis *et al.*, "Preserving peer replicas by rate-limited sampled voting," in *Proc. ACM SOSP '03*, 2003, pp. 44–59.

[2] J. Kubiatowicz *et al.*, "OceanStore: an architecture for global-scale persistent storage," *SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 190–201, 2000.

[3] Y. Saito *et al.*, "FAB: building distributed enterprise disk arrays from commodity components," *ACM SIGPLAN*, vol. 39, no. 11, pp. 48–58, 2004.

[4] R. J. Anderson, "The Eternity Service," in *Proc. Pragocrypt '96*, 1996, pp. 242–252.

[5] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–88, February 1981.

[6] ——, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journ. Cry.*, vol. 1, no. 1, pp. 65–75, January 1988.

[7] I. Clarke *et al.*, "Freenet: A Distributed Anonymous Information Storage and Retrieval System," in *WDIAU*, no. 2009, July 2000, pp. 46–66.

[8] M. J. Freedman *et al.*, "Introducing Tarzan, a Peer-to-Peer Anonymizing Network Layer," in *Proc. IPTPS '01*, 2002, pp. 121–129.

[9] A. Demers *et al.*, "Epidemic algorithms for replicated database maintenance," in *Proc. ACM PODC '87*, 1987, pp. 1–12.

[10] R. van Renesse, Y. Minsky, and M. Hayden, "A Gossip-Based Failure Detection Service," in *Proc. Middleware '98*, September 1998, pp. 55–70.

[11] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Proc. ACM SOSP'01*, 2001, pp. 188–201.

[12] R. Bhagwan *et al.*, "Total Recall: System Support for Automated Availability Management." in *Proc. USENIX NSDI*, 2004, pp. 337–350.

[13] M. Aguilera, R. Janakiraman, and L. Xu, "Using erasure codes efficiently for storage in a distributed system," in *Proc. IEEE DSN '05*, 2005.

[14] B.-G. Chun *et al.*, "Efficient Replica Maintenance for Distributed Storage Systems," in *Proc. USENIX NSDI*, 2006, pp. 45 – 58.

[15] A. J. Ganesh *et al.*, "Peer-to-Peer Membership Management for Gossip-Based Protocols," *IEEE Trans. Comput.*, vol. 52, no. 2, pp. 139–149, 2003.

[16] S. Voulgaris, D. Gavidia, and M. van Steen, "CYCLON: Inexpensive membership management for unstructured P2P overlays," *Journ. Network and Systems Management*, vol. 13, no. 2, June 2005.

[17] A. S. Aiyer *et al.*, "BAR fault tolerance for cooperative services," in *Proc. ACM SOSP*, 2005, pp. 45–58.

[18] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. OSDI '99*, 1999, pp. 173–186.

[19] J. R. Douceur, "The Sybil Attack," in *Proc. IPTPS '01*, 2002, pp. 251–260.

[20] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proc. ACM SOSP'03*, 2003, pp. 223–236.

[21] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *Proc. ACM SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, 2004.

[22] D. Moore, G. M. Voelker, and S. Savage, "Inferring internet Denial-of-Service activity," in *10th Usenix Security Symp.*, August 2001, pp. 9–22.

[23] J. Yan, S. Early, and R. Anderson, "The XenoService – A distributed Defeat for Distributed Denial of Service," in *Proc. ISW 2000*, October 2000.

[24] I. Gupta, "On the design of distributed protocols from differential equations," in *Proc. ACM PODC*, 2004, pp. 216–225.

[25] D. Kostoulas *et al.*, "Decentralized Schemes for Size Estimation in Large and Dynamic Groups," in *Proc. IEEE NCA*, July 2005, pp. 41–48.

[26] R. Morales and I. Gupta, "Providing Both Scale and Security through a Single Core Probabilistic Protocol," in *Proc. StoDiS 2005*, December 2005, pp. 1–5.

[27] ——, "AVMON: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems," in *Proc. ICDCS*, 2007, p. 55.

[28] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. IPTPS '01*, 2002, pp. 328–338.

[29] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, "Parallax: Managing storage for a million machines," in *Proc. USENIX HotOS X*, 2005.

[30] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding when to forget in the elephant file system," in *Proc. ACM SOSP'99*, 1999, pp. 110–123.

[31] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. ACM SOSP'01*, 2001, pp. 202–215.

[32] R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *Proc. ACM SIGCOMM 2001*, San Diego, CA, September 2001.

[33] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet Indirection Infrastructure," in *SIGCOMM*, 2002, pp. 73–86.

[34] J. Blomer *et al.*, "An XOR-Based Erasure-Resilient Coding Scheme," International Computer Science Institute, Berkeley, California, Tech. Rep., 1995.