Folklore System: Providing Both Scale and Security through a Single Core Probabilistic Protocol*

Ramsés Morales and Indranil Gupta
Department of Computer Science
University of Illinois at Urbana-Champaign
{*rvmorale,indy*}*@cs.uiuc.edu*

Abstract

Distributed systems are typically designed for scale and performance first, which often makes it difficult to add security later without affecting the original properties. This paper proposes the design of the Folklore persistent distributed file system, which adopts an alternative design methodology. Folklore's design relies on a single core protocol for providing both probabilistic scalability and untraceability, the latter being a special notion of probabilistic security. The core protocol is a biologically inspired model of endemic replication *that migrates replicas of files among all hosts in a continuous and proactive manner. The emergent behavior is chaotic, meaning that the exact number and location of all replicas of any file is changing all the time. This makes it difficult for an attacker to target any file. Yet, the protocol is scalable – it consumes constant per-host bandwidth, and the number of replicas per file stays close to a small self-stabilizing value. The simplicity of the core protocol allows augmentation with mechanisms that allow data integrity, availability, and updatability. These mechanisms ensure that Folklore is also resilient to massive-scale attacks. We describe the internals of the Folklore system, present attack analysis, and give experimental results from a prototype that shows high resilience to severe attacks.*

1 Introduction

Several networked information stores such as digital libraries [9], global-scale storage [8], "brick"-based storage [11], etc., have emerged over the past few years. All require an ability to scale with thousands of hosts, as well as to provide such notions of security as resistance to attackers, failures, and churn.

*This research was partly supported by National Science Foundation Grant ITR-0427089.

However, many existing mechanisms that ensure security tend to be unscalable. For instance, Byzantine agreement protocols, e.g., [4] require several rounds of all-to-all communication, preventing scalability with even hundreds of hosts. Since the prevalent methodology to designing distributed systems addresses scale and performance first, this makes it difficult to add security later without affecting the original scalability.

We are exploring the design of a persistent and distributed file system called *Folklore* that provides both scalability and certain notions of security. This is achieved by relying on a *single* core protocol that is probabilistic in nature. This core, called the *endemic replication protocol* [7], continuously migrates replicas of files among all the hosts in the system. The protocol is inspired by phenomena resulting in survival of diseases, e.g., influenza.

The basic notion of security provided by Folklore is called **untraceability**. The continuous migration of replicas due to the endemic replication occurs *chaotically*, i.e., the exact number and location of the replicas of a given file is changing all the time, and in an unpredictable manner. This makes it very difficult for any attacker to discover and destroy all replicas of a targeted file. In fact, the average attack window times are very short (typically tens of seconds), beyond which the endemic protocol automatically creates additional replicas that the attacker will not know about.

Folklore is scalable because in spite of the chaotic nature of the replication, the number of replicas of each file stays centered around a stable value, resulting in a small storage utilization. The bandwidth

utilization at each Folklore host is surprisingly low – it is independent of the number of hosts in the system, and depends only on the average file size. For an average file size of 128KB, our experiments indicate that the per-host bandwidth is 3.38 KBps.

The design of Folklore augments the core protocol with simple mechanisms that provide (1) **data availability and updatability**, i.e., the owner of a file can quickly access and modify the file, (2) **data integrity**, i.e., a file cannot be corrupted by an attacker. These sub-protocols do not affect the scale or security notions of the system.

LOCKSS [9] is the only other system that provides probabilistic security and scale. Several persistent distributed stores have been built using distributed hash tables (e.g., [8, 10]), but these do not provide untraceability. In those systems, a surreptitious attacker can first find out the IP addresses of the replica nodes for a given file (through normal lookups), and then bring down these computer hosts (either manually or through a network-based attack), thus removing the file from the system. This attack also works when erasure-coding based replication strategies [3] are used.

2 Design of Folklore

2.1 Core Protocol: Endemic Replication

We describe the variant of the endemic replication protocol [7] that is used in the Folklore system. Consider a host (henceforth a "node") H that is currently storing replicas of some of the files inserted into the system (perhaps by other hosts). We say that the node is *stash*ing each of these files, or that it is in a *stash* state w.r.t. each of these files. For all the files in the system that are not locally stashed at H, H is either *receptive* or *averse*. H is said to be receptive to a file f when it is willing to accept a transfer of f's replica from another stasher node of f. If it refuses any such transfer, it is said to be averse w.r.t. that file. H maintains a list of names for files that it is averse to, while names of the remaining files (for which it is receptive) are not remembered.

The actions of the endemic replication protocol are determined by four important protocol parameters - protocol period T, 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. Parameter values are fixed at all nodes; Section 4 discusses the effect of attacks that modify these values.

At the start of each protocol period at node H, several actions are executed: (1) First, each locally stashed file is moved into the averse state with probability γ. In the averse state, the file can be deleted, although its filename needs to be remembered in order to refuse any transfers. (2) Then, each averse filename is forgotten (i.e., moved into receptive state) with probability α. (3) Finally, H creates a constant sized `stash advertisement` message, containing names of some locally stashed files. The stash advertisement is sent to a constant number $\frac{\beta}{2}$ of target numbers, each target selected uniformly at random from across the group [1]. Target nodes receptive to the advertised filenames may request a transfer from H. In turn, H requests these target nodes to advertise their stashed files, and transfers to itself some files that it is receptive to. This *Stash-Receptive Contact* mechanism is detailed in Section 2.3.

The analysis of [7] showed that for any given file, the fractions of nodes 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 vicinity of this point because of random and unpredictable choices by each node while applying the probabilistic protocol actions (1-3) above.

2.2 Basic File Operations

A file inserted into the Folklore file system is assigned a globally unique id using the base64-encoded SHA1 [2] hash of the owning node's IP address and the filepath at the owning node.

File replicas are encrypted for privacy and plausible deniability to stashers. We use symmetric encryption that is calculated per 4 KB file-blocks. This partial encryption will help in propagating par-

[1] We use a variant of the SWIM membership protocol [5] at each node to maintain partial, local, membership lists.

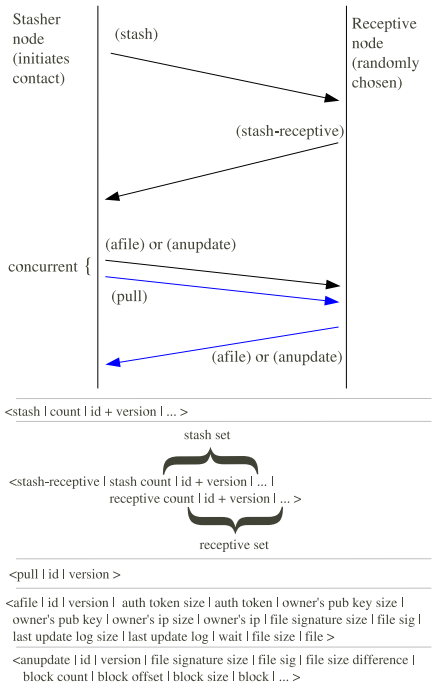

Figure 1: *Message interchange when a node contacts a peer with its stash advertisement.*

tial file updates (Section 2.4). We use the efficient Blowfish [1] encryption algorithm in Cypher Feedback (CB) mode, where each cleartext 4KB block will result in a 4KB cipher.

File replicas are provided with a signature, computed as privateKey.sign(SHA1(file, version, id)). A node will refuse to store a file replica whose signature cannot be validated. The signature is verified with an *authentication token*, a third party signature of the file owner's public key concatenated with its IP address. The public key of the third party is preinstalled on all nodes. Such a third party is unavoidable due to the Sybil attack impossibility result [6].

Finally, for a newly inserted file, the owning node warms up the system by forcing a small random set of nodes to become stashers for the new file; other nodes are of course receptive.

2.3 Stash-Receptive Contacts

As explained before, each node i periodically sends out a stash advertisement message to target node(s) j, chosen uniformly at random. The message interchange is depicted in Figure 1, and detailed below.

Each filename in all advertisement messages is specified as $\langle id, version \rangle$, the latter field being the owner-set version, incremented whenever the owner makes an update. Suppose node i sends a stash advertisement message, and the recipient node is j. j replies to i with a `stash-receptive` message, which contains two sets of filenames: a *stash set* and a *receptive set*.

The *stash set* contains names of all files stashed at node j that were not specified in the stash advertisement message (and could thus be transferred to i). The *receptive set* consists of only those filenames chosen from the stash advertisement message from i that either (1) j is receptive to, or (2) that are also stashed at j but have a lower version number.

To save bandwidth, we allow a node to retain a file replica even when it becomes averse or receptive to the file. Accordingly, condition (1) for selecting the receptive set changes to include those files that: (1)* j is receptive to, and if it is retaining a copy of, that has a lower version number than advertised by i. In addition, any locally retained files 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.

Push-Pull mechanism: On receiving the `stash receptive` reply M' from node j, node i selects *one* id from M''s *receptive set* that is still stashed at i. Node i then *pushes* this file id to j. If the difference between version numbers of this file locally and in M' is 1, a per-block update (`anupdate`) is done, otherwise a full file transfer (`afile`) is used.

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

To ensure that an inserted file ramps up to the stable number of stashers, initial replicas are associated with a *TTL=time to live*. Any replica created with non-zero TTL is forced to stay stashed for the next TTL periods, after which normal Folklore actions apply. TTL is initialized by the owner, as described below. Every new replica created by a stash-receptive contact inherits the TTL from its stashed parent, less a quantity $w = \frac{S_i}{\frac{1}{2}\beta}$. Here, S_i is parent node's stash size, thus w is the expected time (in pe-

riods) to advertise the replica once. The owner initializes the TTL to $w.\lceil \log_{(\frac{\beta}{2}+1)} S_\infty \rceil$ periods. S_∞ is an estimate of the stable number of stashers, e.g., based on expected system size. The total number of forced replicas grows by a factor of $(\frac{\beta}{2}+1)$ every w periods, thus the logarithm's base. It should be evident that all TTLs reach 0 a while after file insertion.

2.4 File Location, Update, and Recovery

The untrusted nature of hosts allows us to impose the restriction that all such accesses are done by the owner node of the file. This allows us to build in relatively simple solutions that leverage off a *heartbeating strategy*.

The heartbeating strategy requires each file f to be associated with a heartbeat probability parameter $\epsilon(f) \in [0,1]$. At the beginning of a protocol period at node H, for each locally stashed file f, and with probability $\epsilon(f)$, H sends its owner a heartbeat message specifying the file's $\langle id, version \rangle$. The expected number of heartbeat messages received at a file's owner, per protocol period, is thus $hb = \epsilon(f) \times N \times \frac{1-\frac{\gamma}{\beta}}{1+\frac{\gamma}{\beta}}$, where N is the number of non-faulty nodes currently in the system. If the expected range for N is known, ϵ can be set so that the owner receives a constant number of heartbeats per file per period. ϵ also trades between response time for a lookup and the bandwidth used by heartbeats.

Due to heartbeats, the owner of a file f is not required to store the file anymore. Only knowledge of the latest version number and symmetric key suffices. When the owner wishes to locate a copy, it simply waits to receive the next heartbeat for file f with the latest version number. For recovery, if the owner does not know the latest version number, it (1) first waits a while to estimate the highest version number in received heartbeats, and (2) then transfers a replica from the next node sending a heartbeat with that maximum version number.

The heartbeating mechanism can be used to spread updates. An update can be spread immediately to a node that sent a heartbeat – if the version number difference is 1, then only modified blocks are transferred, otherwise the entire new file is transferred. The previous section discussed updates through stash-receptive and stash-stash contacts.

Suppose the stable number of stashers for a file 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 start satisfies $(S_\infty - S_i) =$
$(S_\infty - S_{i-1})[(1 - \frac{hb}{S_\infty})(1 - \frac{\beta.S_{i-1}}{2N})(1 - \frac{1}{N})^{\beta.S_{i-1}/2}]$

Analysis and experiments show that within the square brackets, the last two terms (stash-stash and stash-receptive contact) dominate the first term (heartbeating).

3 Attack Analysis of Folklore

We discuss vulnerabilities of the Folklore system and its behavior under various types of attacks.

Replica or Version Corruption: A malicious stasher or a man in the middle that causes changes to a replica, or its version, or its id will be detected by a good node during signature verification. This can also be used to raise an alarm about malicious nodes.

Owner Node Takeover: If the owner node is taken over, but the private key of the owner is not compromised, the attacker can view heartbeats and files, but will be unable to propagate updates. Thus, a Folklore file cannot be corrupted by an owner takeover attack.

Intentional File id Collisions: If a node detects an id conflict, it will ask all conflicting file owners to rename their files. If any of the owners refuse, they are blacklisted at the detecting node.

Sybil Attack: This attack occurs when a large number of nodes are (or appear to be) malicious in the system. For instance, an extreme variety works by breaching a gateway. Although such an attack can cripple any system, Folklore allows a good node to be able to detect such an attack if the number of heartbeats received per period is too low.

4 Experimental Results

Folklore is implemented in C. We present experimental results from a simulated Folklore system running on a 3GHz PIV, 1GB RAM, Linux PC. The system has 1000 nodes, with protocol period $T = 5$ minutes, and $\alpha = 0.05$, $\gamma = 0.4$. A node contacts 5 peers each protocol period (thus $\beta = 10$).

Chaotic Replication: At time $t = 10000$ s, 1000 files are simultaneously inserted into the system, each from a unique owning node. Figure 2 shows

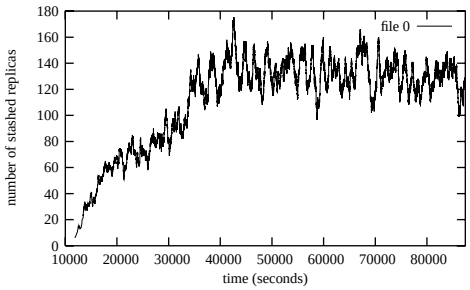

Figure 2: *Replica propagation. 1000 files are inserted into the network.*

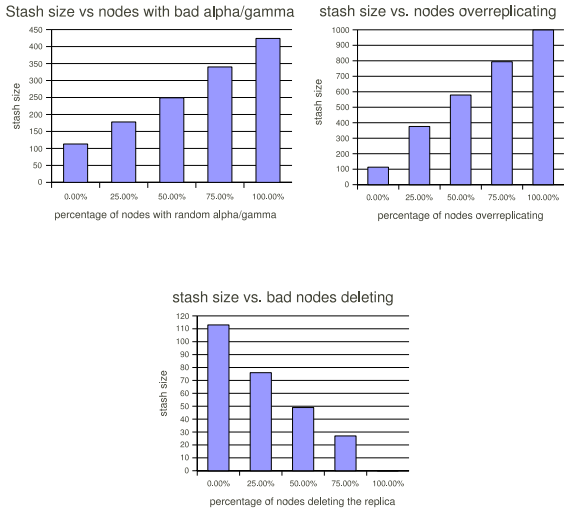

Figure 3: *Effect of Parameter Attack, Over-replication and Replica Destruction.*

how the number of replicas (for file 0) varies over time; similar trends were observed for the other files. The stable expected number of replicas is 120. The chaotic behavior of the protocol around this stable value is evident from Figure 2.

Parameter Attack: Figures 3(a)-3(c) show the effect of a large collection of malicious nodes using bad values for protocol parameters. Figure 3(a) shows the effect of a **random parameter attack** - a fraction of nodes choose random values for $\alpha, \gamma \in [0, 1]$. With 50% of the nodes malicious, the number of replicas of each file increases only two-fold. Figure 3(b) shows the effect of an **over-replicating attack**. Over-replicating nodes reduce the protocol period to 30 s and lock the replica in stash state, i.e., never changes to averse. With 25% of malicious nodes, the number of replicas only increases by a factor of 3. Figure 3(c) shows a **deletion attack**. Even when 75% nodes delete all files they purport to stash, the number of replicas stays non-zero.

Churn Attack: In a network of 526 nodes and

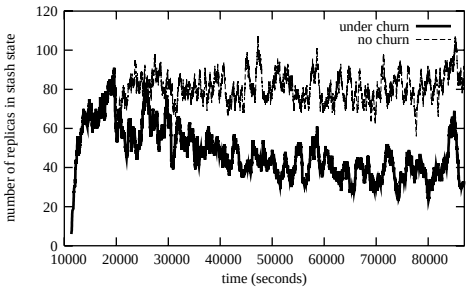

Figure 4: *Effect of Churn Attack.*

526 files, churn was introduced at time $t = 18557$. Figure 4 shows the typical effect for a file – churn causes the number of replicas to halve, but not disappear.

5 Conclusion

It is possible to build distributed file systems that use a single core protocol to provide both scale and notions of security. The Folklore distributed file system is built around scalable endemic replication, whose chaotic yet self-stabilizing behavior provides replica untraceability. Experiments show resilience to massive attacks. We leave details of the Folklore membership protocol, and PC cluster-based experiments to a later article. We are also augmenting Folklore to withstand other types of attacks.

References

[1] *Blowfish, http://www.schneier.com/blowfish.html*.

[2] *SHA-1, ftp://ftp.rfceditor. org/in-notes/rfc3174.txt*.

[3] R. Bhagwan et al. Total Recall: System Support for Automated Availability Management. In *Proc. NSDI*, pages 337–350, 2004.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI '99*, pages 173–186, 1999.

[5] A. Das, I. Gupta, and A. Motivala. SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol. In *Proc. DSN*, pages 303–312, 2002.

[6] J. R. Douceur. The Sybil Attack. In *Proc. IPTPS '01*, pages 251–260, 2002.

[7] I. Gupta. On the design of distributed protocols from differential equations. In *Proc. ACM PODC*, pages 216–225. ACM Press, 2004.

[8] J. Kubiatowicz et al. OceanStore: an architecture for global-scale persistent storage. *SIGARCH Comput. Archit. News*, 28(5):190–201, 2000.

[9] P. Maniatis et al. Preserving peer replicas by rate-limited sampled voting. In *Proc. SOSP '03*, pages 44–59, 2003.

[10] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. SOSP*, pages 188–201, 2001.

[11] Y. Saito et al. FAB: building distributed enterprise disk arrays from commodity components. *SIGPLAN Not.*, 39(11):48–58, 2004.