Atum: Scalable Group Communication Using Volatile Groups

Rachid Guerraoui[1], Anne-Marie Kermarrec[2], Matej Pavlovic[1], Dragos-Adrian Seredinschi[1]

[1]École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2]Inria Rennes, France

{firstname}.{lastname}@epfl.ch, anne-marie.kermarrec@inria.fr

ABSTRACT

This paper presents Atum, a group communication middleware for a large, dynamic, and hostile environment. At the heart of Atum lies the novel concept of *volatile groups*: small, dynamic groups of nodes, each executing a state machine replication protocol, organized in a flexible overlay. Using volatile groups, Atum scatters faulty nodes evenly among groups, and then masks each individual fault inside its group. To broadcast messages among volatile groups, Atum runs a gossip protocol across the overlay.

We report on our synchronous and asynchronous (eventually synchronous) implementations of Atum, as well as on three representative applications that we build on top of it: A publish/subscribe platform, a file sharing service, and a data streaming system. We show that (a) Atum can grow at an exponential rate beyond 1000 nodes and disseminate messages in polylogarithmic time (conveying good scalability); (b) it smoothly copes with 18% of nodes churning every minute; and (c) it is impervious to arbitrary faults, suffering no performance decay despite 5.8% Byzantine nodes in a system of 850 nodes.

CCS Concepts

•**Computer systems organization** → **Fault-tolerant network topologies;** *Reliability; Redundancy;*

Keywords

Distributed systems; Byzantine fault tolerance; Gossip; Group communication

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Middleware'16, December 12 - 16, 2016, Trento, Italy

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4300-8/16/12…$15.00

DOI: http://dx.doi.org/10.1145/2988336.2988356

1. INTRODUCTION

Group communication services (GCSs) are a central theme in systems research [14, 18, 23, 45, 53]. These services provide the abstraction of a node *group*, and typically export operations for *joining* or *leaving* the group, as well as *broadcasting* messages inside this group. For a node in the group, the GCS acts as a middleware between the application and the underlying communication stack. The application simply sends and receives messages, while the network topology, low-level communication protocols and the OS networking stack are abstracted away by the GCS.

A wide range of applications can be built using this abstraction, spanning from infrastructure services in datacenters [5, 39], to streaming and publish/subscribe engines in cooperative networks [18, 19, 47, 53], or intrusion-tolerant overlays [24, 44].

To cope with the needs of modern applications, GCSs need to be *scalable*, *robust*, and *flexible*. Scalability is a primary concern because many applications today involve thousands of nodes [59] and serve millions of users [5, 72]. The main indicator of scalability in a GCS is the cost of its operations, which should ideally be sublinear in system size.

Given the scale of these systems, faults inevitably occur on a daily basis. Crashed servers and buggy software with potentially arbitrary behavior are common in practice, both in cooperative systems (e.g., peer-to-peer) and datacenter services [3, 28, 57, 59]. For instance, according to Barroso et al. [11], in a 2000-node system, 0.5% nodes fail each day. Clearly, GCSs have to be robust by design.

GCSs also need to be flexible, tolerating a considerable fraction of nodes that join and leave the system, i.e., *churn*. Peer-to-peer services are naturally flexible [72]. For datacenter services, churn emerges as a consequence of power saving techniques, software updates, service migration, or failures [11]; cross-datacenter deployment of services tends to further exacerbate the churn issue.

There are well-known techniques to address individually each of robustness, scalability, and flexibility. To obtain a robust design, the classic approach is *state machine replication* [50, 66]: Several replicas of the service work in parallel, agreeing on operations they need to perform, using some consensus protocol [20]. State machine replication (SMR) is powerful enough to cope even with arbitrary, i.e., *Byzantine* faults [7, 20, 48]. To provide a scalable design, *clustering* is the common approach. The nodes of the system are partitioned in multiple groups, each group working independently; the system can grow by simply adding more groups [13, 41]. To achieve flexibility and handle churn, join and leave operations should be lightweight and entail small, localized changes to the system [53]. A standard approach to attain flexibility in a GCS is through *gossip protocols* [29]. With gossip, each participant periodically exchanges messages with a small, randomized subset of nodes. Gossip-based schemes disseminate messages efficiently, in logarithmic time and with logarithmic cost [29].

It is appealing to combine clustering with SMR and gossip to tackle all of the issues above. At first glance, it seems natural to organize a very large system as a set of reliable groups and have them communicate through gossip. Unfortunately, this combination poses a major challenge due to a conflict between robustness and flexibility. Churn induces changes to the system structure and calls for groups that are highly dynamic in nature, i.e., fluctuate in number and size. In contrast, (Byzantine-resilient) SMR has opposing requirements, imposing strict constraints on every group, to keep groups robust and efficient. In particular, to ensure robustness, SMR requires a bounded fraction of faults in every group [16, 32]; to achieve efficiency, it is important to keep groups small in size, as SMR scales poorly due to quadratic communication complexity [25].

In this paper, we report on our experiences from designing and building *Atum*, a novel group communication middleware that seeks to overcome this challenge. To mitigate the above conflict, we introduce the notion of *volatile groups* (vgroups). These are clusters of nodes that are small (logarithmic in system size), dynamic (changing their composition frequently due to churn), yet robust (providing the abstraction of a highly available entity). Every vgroup executes a SMR protocol, confining each faulty node to that vgroup. Among vgroups, we use two additional protocols: *random walk shuffling* and *logarithmic grouping.*

Random walk shuffling ensures that faulty nodes, if any, are dispersed evenly among vgroups. Whenever a vgroup changes, e.g., due to nodes joining or leaving, Atum uses random walks to refresh the composition of this vgroup to contain a fresh, uniform sample of nodes from the whole system. This technique is particularly important for cases when faults accumulate over time in the same vgroup.[1] We thus refresh a vgroup after any node joins or leaves it. Our *logarithmic grouping* protocol guarantees in addition that every vgroup has a size that is logarithmic in the system size. Whenever a vgroup becomes too large or too small, Atum splits or merges groups, to keep their size logarithmic.

To efficiently disseminate messages *among* vgroups, we use a gossip protocol. The complexity of gossip is known to be logarithmic in system size [29]. In Atum, we address each gossip to a vgroup of logarithmic size, resulting in an overall polylogarithmic complexity.

We implement two versions of Atum, one using a synchronous SMR algorithm, and one based on an asynchronous algorithm.[2] To illustrate the capabilities of Atum, we build three applications: *ASub*, a publish/subscribe service, *AShare*, a file sharing platform, and *AStream*, a data streaming system. Given these two implementations and the three applications, we report on their deployment over a variety of configurations in a single datacenter, as well as across multiple datacenters around the globe. We show that Atum: (a) supports an exponential growth rate, scaling well beyond 1000 nodes, and smoothly copes with 18% of nodes churning every minute; (b) is robust against arbitrary behavior, coping with 5.8% Byzantine nodes in a 850-node system; (c) disseminates messages in polylogarithmic time, incurring a small overhead compared to classical gossip.

It is important to note that the goal of our work is to explore the feasibility of a general-purpose GCS – in the same vein as Isis [14], Amoeba [45], Transis [31], or Horus [70], but for large, dynamic, and hostile networks. Our experiences with Atum highlight the numerous complications that arise in a GCS designed for such an environment (§7). The protocols underlying the vgroup abstraction – SMR; group resizing, splitting and merging; distributed random walks – are challenging by themselves. Combining them engenders further complexity and trade-offs. We believe, however, that the vgroup abstraction is an appealing way to go, and we hope our experiences pave the way for new classes of GCSs, each specialized for their own needs. Note also that we do not argue that the vgroup abstraction (and its underlying protocols) is a silver bullet, necessary and sufficient for every part of an application that requires multicasting in a challenging environment. For instance, in our streaming application, we use Atum to reliably deliver small authentication metadata; to disseminate the actual stream data at high throughput, we use a separate multicast protocol.

To summarize, our main contributions are as follows:

- We introduce the notion of vgroups – small, dynamic, and robust clusters of nodes. The companion random walk shuffling and logarithmic grouping techniques ensure that every vgroup executes SMR efficiently, despite arbitrary faults and churn.
- Using a gossip protocol among vgroups, we design Atum, a GCS for large, dynamic, and hostile environments.
- We report on the implementation of two versions of Atum and three applications on top of it.

The remainder of this paper is structured as follows. In §2, we describe the assumptions and guarantees of Atum. §3 and §4 present Atum's design and the three applications we built on top of it, respectively. We move on to discussing some practical aspects of our synchronous and asynchronous Atum implementations in §5. §6 reports on our extensive experimental evaluation, using the above-mentioned three applications. We discuss our experiences and lessons learned in §7. Finally, we position Atum with respect to related work in §8 and conclude in §9.

2. ASSUMPTIONS AND GUARANTEES

Atum addresses the problem of group communication in a large network. Despite arbitrary faults or churn, Atum guarantees the following properties for correct nodes. *Liveness*: If a node requests to join the system, then this node eventually starts to deliver the messages being broadcast in the system; this captures the liveness of both join and broadcast operations. *Safety*: If some node delivers a message m from node v, then v previously broadcast m.

[1]This may happen due to bugs [57, 59, 73] or join-leave attacks [8]; both of these situations reflect the concentration of faults in the same group.

[2]Strictly speaking, an asynchronous SMR implementation is impossible [37]. In this context, we call a SMR implementation asynchronous if it requires synchrony only for liveness (eventual synchrony), like PBFT [20].

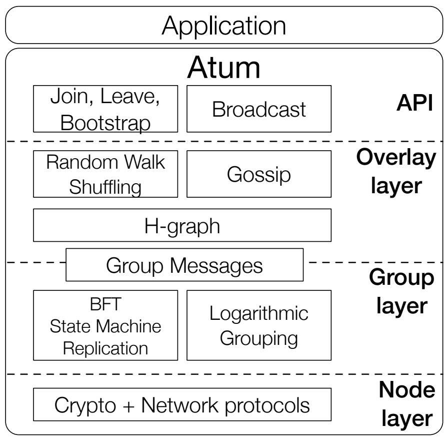

Figure 1: Atum's layered architecture.

Atum uses SMR as a building block (inside every vgroup) and it inherits all assumptions made by the underlying SMR protocol. We assume that a bounded number of nodes are subject to arbitrary failures such as bugs or crashes, so we consider SMR protocols with Byzantine fault tolerance guarantees. Depending on the target environment, we can use either an asynchronous protocol [20], or a synchronous one [32]. Atum itself, however, has a general design and is oblivious to the specifics of this protocol. As we will discuss later, we experiment with both versions of this protocol.

We model the system as a large, decentralized network, where a significant fraction of nodes can join and leave (i.e., churn). For liveness, we only expect the network to eventually deliver messages; this is a valid assumption even in highly unstable networks such as the Internet. Safety relies on the correctness of the underlying SMR protocol.

We use cryptography (public-key signatures and MACs) to authenticate messages, and assume that the adversary is computationally bounded and cannot subvert these techniques. We do not consider Sybil attacks in our model; these can be handled using well-known techniques, such as admission control [33] or social connections [52]. An alternative, decentralized solution to deter Sybil attacks is to rely on cryptographic proofs of work, as in Bitcoin [56].

Atum tolerates a limited number of nodes isolated by a network partition. In practice, we treat isolated nodes as faulty, so the bound on the number of faults also includes partitioned nodes. Aside from the SMR algorithm, a severe network outage might break liveness, but not safety.

3. DESIGN

Atum is a group communication middleware positioned between a distributed application and the underlying network stack. It has a layered design comprising four layers, as depicted in Figure 1. At the bottom, the *node* layer handles inter-node communication. We use standard techniques here – cryptographic algorithms to secure communication, and a network transport protocol for reliable inter-node message transmission. Since these are orthogonal to our design, we do not dwell on their details.

At the *group* layer (§3.1), Atum partitions nodes into vgroups of logarithmic size. We ensure the robustness of each vgroup using a state machine replication protocol with BFT guarantees. For inter-vgroup communication we use special messages, called *group messages*, that ensure reliable communication for pairs of vgroups.

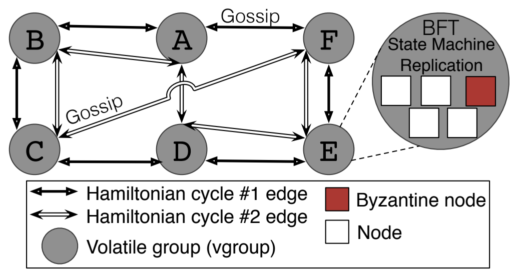
Gossip
Gossip
B
A
F
C
D
E
BFT State Machine Replication
Hamiltonian cycle #1 edge
Hamiltonian cycle #2 edge
Volatile group (vgroup)
Byzantine node
Node

Figure 2: An instance of Atum: Vgroups interconnected by an H-graph overlay with two cycles.

The *overlay* layer (§3.2) connects vgroups and enables them to communicate. The network formed by vgroups has the structure of an H-graph [51], as Figure 2 depicts. At this layer, the protocols are typically randomized (based on gossip and random walks), and rely on group messages.

At the topmost layer sits the Atum *API*. For membership management, we provide *bootstrap*, *join*, and *leave* operations; for data dissemination, we expose a *broadcast* operation. In the remaining parts of this section we describe the interplay between these layers and their corresponding techniques, including the API operations.

3.1 Group layer

The purpose of the group layer is to mask failures of individual nodes and provide the abstraction of robust vgroups. We partition nodes in vgroups of size g, and apply a BFT SMR protocol in every vgroup. We design Atum to be agnostic to this underlying protocol, so our system can support either a synchronous version, which tolerates at most $f = \lfloor (g-1)/2 \rfloor$ faults in every vgroup [32], or an asynchronous version, which has a lower fault-tolerance, at $f = \lfloor (g-1)/3 \rfloor$ faults per vgroup [20]. We say that a vgroup is robust if the number of faults in that vgroup does not exceed f.

Assuming each node in a vgroup has the same constant probability of being faulty (we will discuss this assumption closely in §3.2), the size g of a vgroup is critical for ensuring its robustness. The more nodes a vgroup contains, the higher the probability of being robust. To get the intuition, consider a synchronous system with 1 failure out of every 20 nodes, i.e., with failure probability of 0.05. A vgroup with $g = 4$ nodes tolerates $f = \lfloor (4-1)/2 \rfloor = 1$ faults and fails with probability $Pr[X >= 2] = \mathbf{0.014}$; the random variable X denotes the number of failures and follows the binomial distribution $X \sim B(4, 0.05)$. But a 20-node vgroup, with $f = 9$, will fail with $Pr[X >= 10] = \mathbf{1.134 \cdot 10^{-8}}$. Thus, larger vgroups are more desirable from a robustness perspective. On the other hand, large vgroups entail a bigger overhead of the BFT protocol, penalizing performance [25]. Efficiency thus requires a smaller g.

At the group layer, there is thus a clear trade-off between robustness (larger g) and performance (smaller g). Whatever vgroup size we pick, the probability of *all* vgroups being robust decreases as the number of vgroups in the system grows. To understand the trade-off, let us denote the expected system size by n, and the number of vgroups by

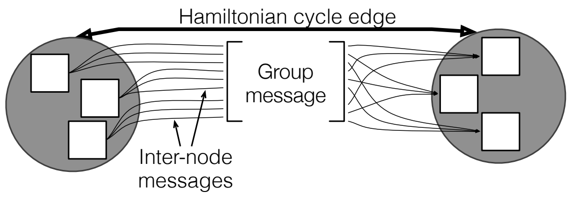

Figure 3: Two vgroups communicate (e.g., gossiping) through a group message, which consists of multiple inter-node messages.

n/g.[3] Consider a growing system. At one extreme, if vgroup size g is constant, we promote efficiency at the cost of robustness; as the system grows and accumulates vgroups, the probability of *all* vgroups being robust diminishes. At the other extreme, if the number of vgroups n/g is constant, we favor robustness to the detriment of efficiency: Robustness improves as n grows, but the BFT overhead may become impractical. We argue that a middle-ground between these two extremes is the best option.

Logarithmic grouping. We favor both efficiency and robustness in a controlled manner by making g and n/g grow slowly, sublinearly in system size. We do so by setting $g = \texttt{k} \cdot \log(N)$, i.e., vgroups have their size logarithmic in the system size; it has previously been proven that this is the optimal choice [9, 42, 65]. The system parameter `k` controls the above-mentioned trade-off. With bigger `k`, robustness increases at the cost of performance, independently of system size. In practice, we believe $\texttt{k} = 4$ is a good trade-off: Even in a system with 6% *simultaneous* arbitrary faults, there is a probability of 0.999 of *all* vgroups being robust.

In a dynamic environment, vgroups do not have a fixed size g, but their size fluctuates due to churn. We introduce two system parameters, $\texttt{g}_{\texttt{min}}$ and $\texttt{g}_{\texttt{max}}$, defined by a system administrator at startup; these define the minimum and maximum vgroup size, respectively. If a vgroup grows beyond $\texttt{g}_{\texttt{max}}$, then we split that vgroup in two smaller ones. When a vgroup shrinks below $\texttt{g}_{\texttt{min}}$ nodes, we merge it with another vgroup. Parameters $\texttt{g}_{\texttt{min}}$ and $\texttt{g}_{\texttt{max}}$ depend on $g = \texttt{k} \cdot \log(N)$.[4]

Group messages. At the group layer, we can view Atum as consisting of a host of robust vgroups. To achieve coordination among vgroups and implement data dissemination, we introduce group messages as a simple communication technique for pairs of vgroups. A group message from vgroup A to vgroup B is a message that all correct nodes in A send to all nodes in B. A node d in B accepts such a message iff d receives this message from the majority of nodes in A, which guarantees correctness of the group message.

Group messages are a central building block of Atum. Two vgroups can exchange group messages only if they know each other's identities, i.e., nodes in vgroup A know the composition of vgroup B and vice versa. We illustrate a group message in Figure 3.

[3]The expected system size n need not be exact, an estimation suffices. If n is conservative (too large), then the system trades efficiency for better robustness; and vice versa if n is too small.

[4]The sole purpose of g and k is to better understand Atum's robustness. It is only $\texttt{g}_{\texttt{min}}$ and $\texttt{g}_{\texttt{max}}$ that are used as configuration parameters in practice.

3.2 Overlay layer

At this layer, Atum maintains an overlay network on top of vgroups that enables the use of group messages – such that vgroups can communicate through gossip. The overlay layer also manages the composition of every vgroup using random walk shuffling. At this layer, each pair of connected vgroups informs each other of any composition change.

The overlay has the form of an H-graph [51], in which vgroups correspond to vertices and vgroup connections to edges (see Figure 2). An H-graphis a multigraph composed of a constant number of random Hamiltonian cycles. Each vertex thus has two random neighbors for each cycle. This structure is sparse (constant degree), well connected, and has a logarithmic diameter with high probability. Thus, we can apply gossip efficiently on top of this overlay, because messages can permeate rapidly through the whole network. The sparsity of the overlay allows Atum to scale, since every vgroup only has to keep track of a limited (constant) number of neighboring vgroups. A further reason for using this overlay is its decentralized random structure, which is well suited for efficient vgroup sampling using random walks [51].

Gossip. We use gossip along the edges of the H-graph, so any two neighboring vgroups can gossip using group messages. We use this technique to disseminate application messages whenever a node invokes a `broadcast` operation. To transform gossip's probabilistic delivery guarantees into deterministic ones, we have each vgroup gossip at least with neighboring vgroups on a specific cycle of the H-graph.

Random walk shuffling. Like gossip, we also run this protocol along the edges of the overlay. We use random walk shuffling to handle churn, i.e., join and leave operations. Recall that at the group layer we assume that each node has the same constant probability of being faulty (§3.1). Random walk shuffling guarantees this assumption by assigning joining nodes to vgroups selected uniformly at random from the whole system. As the name of this technique suggests, we use random walks to sample vgroups.

A random walk is an iterative process, where a message is repeatedly relayed across the overlay network. The length of the walks is a system parameter that we denote by `rwl`. At each step of the walk, a vgroup sends a group message to another vgroup using a random incident link of the overlay. After `rwl` steps, the walk stops at some random vgroup from the network – this is the vgroup which the random walk *selected.* Multiple parameters impact the uniformity of vgroup selection: `rwl`, n/g (i.e, the number of vgroups in the system), and the density of the network (given by `hc`, the number of H-graph cycles). Intuitively, for uniform selection, a small and dense system needs shorter random walks than a larger, sparser system. In Table 1, we summarize the important parameters of Atum.

In order to find proper combinations of Atum parameters (to obtain uniform random vgroup selection), we carry out a simulation. The aim is to derive a guideline that shows the relations between these parameters, so we can properly configure Atum to provide uniform sampling.

Figure 4 shows the simulation results. We consider the length of the random walk optimal if Pearson's χ^2 test with a confidence level of 0.99 cannot distinguish the distribution of the simulated random walks from a truly uniform distribution. The interpretation of this guideline is straightforward; e.g., in a system of roughly 128 vgroups, we set `rwl` to 9 and `hc` to 6. In §6.1.2, we evaluate experimentally the

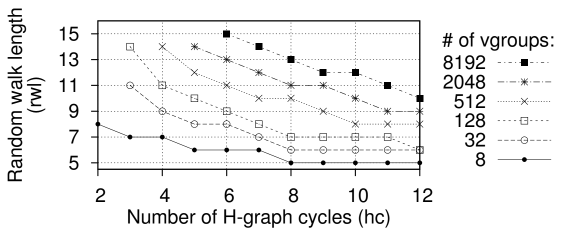

Figure 4: Guideline with optimal `rwl` and `hc` system parameters.

trade-off between `rwl` and `hc`.

Even if new nodes join random vgroups, bugs can lead to faulty nodes accumulating in the same vgroup over time (or an adversary can mount a join-leave attack [8]). To counter such a situation, after a node joins or leaves a vgroup, we refresh the composition of that vgroup through a shuffling technique: We exchange all nodes of this vgroup with nodes selected uniformly at random from the whole system. To select a random node from the system, we first use a random walk to select a vgroup, which in turn picks a random node from its composition.

Random walk shuffling ensures that the composition of every vgroup is sampled randomly from the whole system. By keeping vgroups random, we provide a sufficient condition to ensure their robustness. To also ensure efficiency, logarithmic grouping maintains the size of each vgroup logarithmic in system size.

3.3 API operations

Atum exports four basic operations:

- `bootstrap`(*ownIdentity, params*),
- `join`(*contactNode*),
- `leave`(), and
- `broadcast`(*message*).

In addition, our system requires the application to provide two callback functions:

- `deliver`(*message*), and
- `forward`(*message, neighbor*).

In the following, we describe these operations in detail.

3.3.1 `bootstrap`*(*ownIdentity, params*)*

The `bootstrap` operation creates a new instance of Atum that consists of a single vgroup containing only one node – the calling node. Trivially, this vgroup is a neighbor to itself on every cycle of the H-graph. The parameter *ownIdentity* identifies the calling node. It contains the network address (IP address and port) that other nodes can use to join this instance of Atum. The *params* argument specifies system parameters as we present them in Table 1 (except for `k`).

Param.	Description	Typical values
`hc`	Number of H-graph cycles.	$2, \ldots, 12$
`rwl`	Length of random walks.	$4, \ldots, 15$
$\mathtt{g_{max}}$	Maximum vgroup size.	$8, 14, 20, \ldots$
$\mathtt{g_{min}}$	Minimum vgroup size.	$0.5 \cdot \mathtt{g_{max}}$
`k`	Robustness parameter	$3, \ldots, 7$

Table 1: System parameters.

3.3.2 `join`*(*contactNode*)*

As in other BFT systems [49], [62], Atum uses a trusted entity to orchestrate the first contact between a joining node and the system. In Atum, any correct participating node can take this role; we call such a node a *contact node*. In practice, the contact node can be a social connection of the joining node, and it is well-known that, without a centralized admission control scheme, a trusted entity is a necessary prerequisite for joining an intrusion-proof system [27].

Let c be a contact node belonging to vgroup C. A `join` operations proceeds as follows. A joining node j contacts c, which replies with the identities and public keys of nodes in C; this is the only step where j needs to trust c. The joining node then sends a request to be added to the system to all nodes of C. After receiving a join request, the nodes in C execute an SMR agreement operation [20, 32] to make sure that either all correct nodes of C handle the request or none of them does. This agreement handles the case when j is faulty and sends the join request only to a subset of C.

After agreeing on the join request, C starts the *random walk shuffling* protocol by initiating a random walk. A vgroup D selected by this walk will accommodate j. After the walk finishes, vgroup C sends a group message with the composition of D to j. In the next step, j contacts all nodes in D, these nodes agree on j's request, update their state, and notify their neighboring vgroups about the new member j. Since we use SMR inside vgroup D, j synchronizes its state with D. The state replicated at each node includes information needed to participate in all protocols, e.g., neighboring vgroup compositions, state of ongoing random walks, or pending join or leave operations.

After vgroup D receives the new node j, random walk shuffling continues by exchanging all nodes of D (including j) with random nodes from the whole system. First, D starts a random walk for each of its nodes to select exchange partners. Let $\mathcal{S}$ denote this set of partners. The next step is exchanging the nodes: (1) each node in D joins the vgroup of its exchange partner, and (2) the partners in $\mathcal{S}$ become members of D.

The last part of the join operation is to check if the size of D exceeds $\mathtt{g_{max}}$. If it does, we trigger the *logarithmic grouping* protocol. This protocol splits the nodes of D into two equally-sized random subsets – one remains in D, the other forms a new vgroup E. After the split, D starts one random walk for each cycle of the H-graph. Each vgroup selected by such a random walk inserts E between itself and its successor on the corresponding cycle of the H-graph.

3.3.3 `leave`*()*

With this operation, a node l sends a request to leave the system to all nodes of its vgroup L. Nodes in L agree on this request, reconfigure to remove l, and inform their neighbors about the reconfiguration. After l leaves, random walk shuffling refreshes the composition of L as described above. If this group performs a *merge* (described below), we defer the shuffling until after merging.

If L shrinks below $\mathtt{g_{min}}$ nodes, we trigger logarithmic grouping to *merge* L with a random neighboring vgroup M: All nodes of L join M, and we remove L from the overlay. This removal leaves a "gap" in each cycle of the H-graph. To close these gaps, the predecessor and successor of L on each cycle become neighbors; they receive the information about each other from L. M informs its neighbors about the reconfiguration, shuffles, and splits if necessary.

3.3.4 `broadcast`(message)

This operation allows a node to broadcast a message to all nodes. A `broadcast` operation comprises two phases. In the first phase, the calling node initiates an SMR operation to do a Byzantine broadcast inside its own vgroup [12, 32]. In the second phase, Atum uses gossip to disseminate the message throughout the overlay.

The second phase is customizable, and the application-provided callback `forward` drives the gossip protocol. When a vgroup receives a broadcast message for the first time, Atum delivers this message by calling `deliver`. It then calls `forward` once for each neighbor of that vgroup; this function decides, by returning *true* or *false*, whether to forward a message to a neighbor on the H-graph or not. The default behavior in Atum is to forward broadcast messages to random neighbors, akin to gossip protocols [29].

By modifying the `forward` callback, an application designer can trade-off between message latency, throughput, and fairness. For instance, in latency-sensitive applications, Atum can gossip along *all* H-graph cycles, flooding the system, to disseminate messages fast. For throughput, an application can gossip along a *single* cycle, allowing higher data rates, but increased latency. We experiment with this callback in the evaluation of our data streaming application (§6.3). We note that an unwise choice of `forward` can break the guarantees of broadcast, for instance, if this callback specifies to *not* forward messages to *any* neighbor.

4. APPLICATIONS

In this section, we illustrate the usage of Atum by designing three applications, which we layer on top of our GCS. We first describe a simple publish/subscribe service, then a file sharing system, and finally a data streaming application.

4.1 ASub

Publish/subscribe services are an essential component in cooperative networks and datacenter systems alike [19, 34, 39]. ASub is a topic-based publish/subscribe system which relies entirely on the capabilities and API of Atum. We remark that topic-based pub/sub is essentially equivalent to group communication, since the programming interfaces of these two paradigms coincide. The abstraction of a *topic* matches with the abstraction of a *group*, because subscribing to a certain topic involves joining the group dedicated for the said topic; similarly, publishing an event is analogous to broadcasting a message to a group of nodes [15].

Given this equivalence between group communication and pub/sub systems, to build ASub we only need to add a thin layer on top of Atum. Due to space considerations, we do not dwell on the details of this system. Suffice to say that the operations of ASub map directly to the Atum API (§3.3). Thus, we obtain the following pub/sub operations: `create_topic`, `subscribe`, `unsubscribe`, and `publish` from Atum's `bootstrap`, `join`, `leave`, and `broadcast`, respectively.

4.2 AShare

In this file sharing application, Atum plays a central role by providing the messaging and membership management layer. In AShare, we distinguish between *data*, i.e., file content, and *metadata*, i.e, mapping between files and nodes, file sizes, owners, and file checksums. AShare relies on two protection mechanisms to ensure data availability and authenticity: a novel *randomized replication* scheme to account for high churn, and *integrity checks* to fight file corruption.

AShare stores metadata as soft state, keeping a complete copy at each node, inside a structure called the metadata index.[5] Whenever a node wants to update the index, it initiates a broadcast, informing every node about the update. We use Atum's `broadcast` to ensure reliable delivery, so every node correctly receives the broadcast (§3.3).

4.2.1 Interface and namespace

To add a file with name f in AShare, the owner u calls $\langle \mathsf{PUT}, u, f, c, d\rangle$, where c is the file content and d is the digest of the content. Conversely, $\langle \mathsf{DELETE}, u, f\rangle$ triggers the system to remove all the replicas of f. When nodes want a specific file, they do a $\langle \mathsf{SEARCH}, e\rangle$, where e is the search term, e.g., file or owner name. As a result, SEARCH might yield a file f' previously added by a node u'. To obtain f', a node calls $\langle \mathsf{GET}, u', f'\rangle$.

The namespace is similar to that of file sharing networks. Every user has its own flat namespace, so we identify files by both their owner and their name (u, f); for simplicity, we often omit the owner u when referring to a file. Users have exclusive write access (PUT and DELETE) to their own namespace, and read-only access to foreign namespaces (GET or SEARCH). Being a file *sharing* network, we do not aim at ensuring privacy, but the partitioned namespace restricts malicious activities, given the read-only access. Another advantage of the partitioned namespace is that no updates on the index can ever conflict.

4.2.2 Operations and protection mechanisms

For the sake of availability, AShare replicates every file when the owner calls PUT for that file. In the first step of this operation, the owner u broadcasts a message with the tuple (u, f, d), making the file available for everyone to read. Upon delivery of this message, every node updates their index to include this new tuple, and then they run a randomized replication algorithm. This algorithm creates multiple replicas of f at random nodes; AShare aims to maintain at least ρ replicas per file, where ρ is a system parameter. In practice, ρ should correspond to a fraction (e.g., 0.1 to 0.3) of the system size. We ensure the availability of every file as long as at least one correct replica exists for each file. ρ replicas thus protect against $\rho - 1$ failures.[6]

Randomized replication. The basic replication algorithm that every node executes is as follows. Given a file f, each node consults its index to compute c, the replica count for f; if c is smaller than ρ, then every node replicates f with probability $\frac{\rho-c}{n}$, n being the current system size. The outcome of the algorithm is that a random sample of nodes nominate themselves to replicate f, yielding ρ replicas on expectation. To attain ρ copies with certainty, we introduce a feedback loop that triggers the randomization algorithm repeatedly.

Figure 5 depicts the feedback loop. To replicate a file (u, f), a node x simply reads the file by calling $\langle \mathsf{GET}, u, f\rangle$.

[5]An alternative is to use a DHT [69]. This method, however, is fraught with challenges if we want to tolerate arbitrary faults and churn [74]. We leave this for future work.

[6]A failure in this context means that a node holding a file replica misbehaves (e.g. by corrupting replicas), or leaves the system.

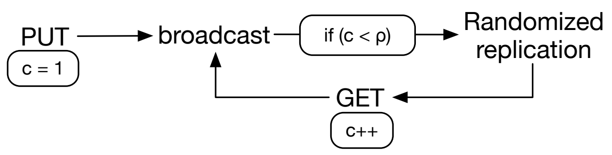

Figure 5: AShare: A feedback loop triggers the randomized replication algorithm repeatedly. c is the number of replicas for a file.

When GET finishes, x broadcasts the tuple $((u, f), x)$; this broadcast informs every node that x now stores a replica of f. Upon delivering this broadcast, all nodes update their index, and then the feedback loop kicks in: Nodes which do not already store f execute the randomized replication algorithm once more, using the same basic steps we described earlier. The feedback loop deactivates when c (the number of replicas for f) becomes greater or equal to ρ.

During a GET operation, the calling node consults its index to obtain the addresses of all the nodes which store the target file f. The node needs all these addresses because it performs a chunked transfer from multiple nodes at a time (not just from the owner). A problematic situation can appear, however, if some node that stores a replica of this file is faulty and the replica is not consistent with the original file. To solve this issue, we introduce integrity checks.

Integrity checks. This protection mechanism preserves the safety of the service. It allows a node to verify if a replica of a file is authentic, and fights against file corruption that can arise as a result of disk errors [67] or Byzantine faults.

As described earlier, as part of the PUT operation, the owner broadcast a tuple (u, f, d), containing file identification and digest. We compute the digest using a SHA-2 collision-resistant hash function. The nodes store this digest in the index, and then use it to verify data authenticity.

Nodes pull files from each other in chunks, i.e., every file has a predetermined number of chunks, established by the owner. Chunks are the units of transfer during GET. This scheme has two benefits: (1) A node can pull file chunks in parallel from all the nodes which replicate that file; (2) digest computation is faster because it can take advantage of multithreading, by computing digests for multiple chunks in parallel. If the integrity check for any chunk fails, then the chunk is pulled from another node. Given this chunked transfer scheme, parameter d in a PUT operation is actually a set of digests, each corresponding to one of the chunks.

We implement the index as a general key-value store using SQLite [1]. It is useful both to resolve file lookups (by checking the files-to-nodes mapping) and to verify the authenticity of chunks (using digests). If a node detects that its index is corrupted (e.g. due to disk errors or bugs in auxiliary software such as SQLite), it can leave and rejoin the system to obtain a fresh, correct copy of the index.

We implement DELETE using a broadcast, which informs every node to update their index accordingly. If a node stores a replica of the file being deleted, then it also discards the chunks of this replica. SEARCH is straightforward, since every node has the metadata index; we implement this operation on top SQLite's query engine.

4.3 AStream

AStream is a streaming application with a two-tier design. Atum represents the first tier, which reliably disseminates stream authentication (digests) from the source node to other nodes. The second tier is a lightweight multicast algorithm which disseminates the actual stream data. Every node uses the digests from the first tier to verify data from the second tier. This second tier has two modules:
A decentralized algorithm to construct a set of spanning trees, and a push-pull multicast scheme to propagate data. We consider these modules interesting in their own right, but due to lack of space we only give a high-level sketch.

Our second tier is inspired from previous solutions on forest-based reliable multicast [18]. We construct a graph (union of trees) with two important properties: (1) It is rooted in the source node (i.e., the broadcasting node), and (2) every node – except the root – has at least one parent which is correct. Intuitively, these two properties ensure that all nodes receive the data stream from the root correctly.

To build a graph with these properties, we leverage the underlying structure of the Atum overlay (see Figure 2 for an illustration) as follows. First, we use a deterministic function that every node knows, to pick one of the cycles of the H-graph, denoted w, and a direction d on that cycle (either *left* or *right*). Each node then builds a set of parents of size $f + 1$, chosen randomly from vgroup V, where V is the neighboring vgroup on cycle w and direction d. The nodes which are neighbors with the source choose the source as their single parent – forming the connection to the root. Given the properties of Hamiltonian cycles and the fact that every vgroup has a majority of correct nodes, this ensures that every node has at least one correct parent, the source node being the root. In addition to this, nodes also select a parent from all other neighboring vgroups, which they may use as shortcuts, in case they are very far from the source node on the selected cycle w.

For disseminating the data, we use a simple, redundant scheme. The root first splits the data in successive chunks, and pushes the first chunk to each of its children. These children, in turn, push this chunk to their children. The algorithm then switches from a push phase to a pull phase, as follows: Each child selects the first parent that pushed a valid chunk, and periodically pulls the subsequent chunks. A node that fails to obtain stream chunks (after receiving the corresponding digests through Atum) tries pulling from another parent. While it is simple, this technique ensures delivery of all data, as at least one parent is always correct.

5. DEPLOYING Atum

In this section, we discuss some practical aspects of Atum. We then present our two different implementations of it.

5.1 Practical considerations

Message digests. Similar to [20], we reduce network bandwidth usage by substituting the content of some messages with their digest. In Atum, a majority of the nodes in any vgroup send the entire group message (§3.2); the remaining nodes only send a digest of the corresponding message. Since every vgroup has a correct majority of nodes, this strategy ensures that at least one correct node sends the entire message, so we never need to retransmit a message.

Random walk communication. When a vgroup G starts a random walk (§3.2), the vgroup S selected by this random walk cannot communicate directly with G, because S is a random vgroup from the system, not necessarily a neighbor of G, and thus might not know G's composition. To deal with this issue, random walks comprise a *backward phase*.

The backward phase of a random walk carries a message from S back to G, relayed by the same vgroups that initially forwarded the random walk. After the backward phase finishes, G and S can start communicating directly, as we piggyback their compositions on the relayed messages.

An alternative solution is what we call *random walk certificates*. They work as follows. At each iteration of a random walk, the forwarding vgroup appends a certificate to the message used to carry out the random walk. This certificate consists of the identity of the chosen neighbor, signed by the forwarding vgroup. When the walk reaches the selected vgroup, it contains a chain of certificates, where each vgroup certifies the identity of the next one. The selected vgroup S can then send a reply directly to the originating vgroup G, with the whole certificate chain appended. This way, G can verify the identity of S by verifying the certificates in the chain. The advantage of this approach is that a backward phase is not necessary and that vgroups need not keep state of ongoing random walks. Depending on the length of the random walk, however, the certificate chain can become bulky in size (which is linear in `rwl`).

We experiment with both approaches to random walk communication (backward phase and certificates) in our two implementations. The asynchronous implementation uses random walk certificates, since they are conceptually simpler, easier to implement, and incur less total overhead. However, verifying all signatures in a long certificate chain is computationally expensive. Since certificate verification would make it hard for the synchronous implementation to meet its timing deadlines, we opt for the backward phase in the synchronous case.

Bulk RNG for random walks. A random walk of length `rwl` requires that `rwl` vgroups generate a random number – each vgroup that forwards the walk to a random neighbor. Distributed random number generation algorithms, however, are expensive [46]. Thus, we generate *all* of the `rwl` random numbers in bulk at the first iteration of the walk, and we piggyback these numbers on the random walk messages. At each subsequent iteration of the walk, the forwarding vgroup uses one of the `rwl` random numbers.

Intuitively, one could consider a simpler approach of precomputing random numbers at each vgroup, and using such a random number pool whenever a random walk needs to be relayed. Interestingly, this approach turns out to be incorrect. A single Byzantine node could bias the random decision by repeatedly triggering operations that consume random numbers from the pool. Thus, to keep our system robust against such an attack, we require that random numbers are not generated before knowing exactly what to use them for.

Randomized message sending. During early experiments with Atum we noticed that the problem of throughput collapse can arise [22]. This can happen when a vgroup has to send one or multiple large group messages. Typically, the size and number of inter-node messages in a group message depend on the size of the communicating vgroups (see Figure 3). After the nodes of the sending vgroup generate outgoing messages, they send them in a short burst to the first node of the destination vgroup, then the second node, and so on. In the worst case, there is an upsurge of download bandwidth at each destination node, leading to packet loss.

To address this issue, each sending node randomizes the order of the outgoing messages.

Removing unresponsive nodes. Nodes become unresponsive due to crashes, bugs, network partitions, etc. We use a mechanism similar to `leave` to evict unresponsive nodes. To this end, every node in Atum sends periodic heartbeats to its vgroup peers. If a node fails to send heartbeats, the other nodes in the vgroup eventually agree to evict this node. Eviction proceeds in the same way as a `leave`.

In an asynchronous system, it is impossible to distinguish a failed node from a slow node, so our heartbeats are coarse-grained, e.g., one every minute. If a node is silent and omits to send many successive heartbeats, amounting to a predefined period of time, then its peers agree to evict the silent node. If an evicted node recovers, it can rejoin the system using `join`. This eviction scheme does not endanger safety, because we evict nodes at a very slow rate. If an attacker wants to break the safety of our system by attacking correct nodes, the attacker would have to mount a persistent barrage of DDoS attacks on many nodes; we believe the resources needed for such an attack outweigh the benefits.

5.2 Atum implementations

As explained in §3, at the design level we make no explicit choice of which SMR algorithm to use inside vgroups. In the first implementation, we choose to use a synchronous algorithm, in particular the Dolev-Strong agreement protocol [32] for SMR; synchronous algorithms are significantly simpler to implement, reason about, and debug, compared to their asynchronous counterparts [12, 20]. To see how this choice impacts performance and to obtain a comprehensive evaluation, we also implement a version of Atum based on the PBFT asynchronous SMR [20], combined with an adaptation of the SMART protocol [55] for reconfiguration. We examine the differences between these two implementations in our evaluation (§6) and further discuss them in the experiences section (§7). In addition, we also implement the three applications described in §4.

6. EVALUATION

We report on our experiments with Atum on Amazon's EC2 cloud. For the synchronous version (Sync), we use a single datacenter in Ireland. Due to a high level of redundancy, intra-datacenter networks are synchronous [68]; indeed, infrastructure services in datacenters often rely on synchrony [21, 26, 40]. For WAN experiments, we use the asynchronous version (Async) because the network is less predictable. We deploy Async across 8 different regions of the world, located in Europe, Asia, Australia, and America. For both Sync and Async, each node runs on a separate virtual machine instance of type micro, which provides the lowest available CPU and networking performance [2].

6.1 Base evaluation of Atum

We study here the behavior of the main operations in Atum, so these results pertain to any application layered on top of Atum. Since Atum implementations are user-space libraries, we use the ASub application to carry out these base experiments. We deploy both Sync and Async and address four important questions: (1) How fast can the Atum system

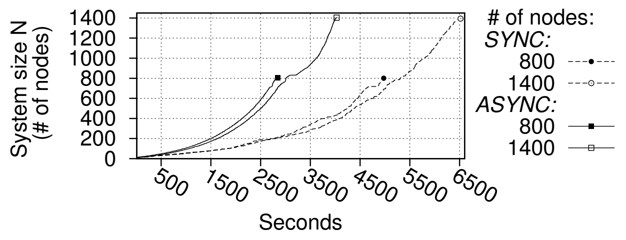

Figure 6: Growth speed for systems with up to 1400 **nodes.**

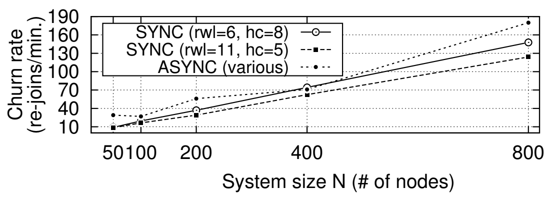

Figure 7: Maximal tolerated churn rates in systems of size 50, 100, 200, 400 **and** 800 **nodes.**

grow? (2) What continuous churn rate can it sustain? (3) How fast does Atum disseminate messages (a) in a failure-free scenario (b) and in presence of Byzantine nodes?

6.1.1 System growth speed

We first consider the join throughput; this may reflect, for example, the arrival rate of new subscribers in ASub. We use different configurations of (`hc`, `rwl`), depending on the target system size, according to our guideline in Figure 4. E.g., for a system with 800 nodes in roughly 120 vgroups, (`hc`, `rwl`) = (5, 10). For the Sync system we use rounds of 1 second, and we evaluate both versions using systems of 800 and 1400 nodes.

As Figure 6 shows, if we configure a system for a smaller maximum size, then the system can grow slightly faster. This is because larger systems require larger values for `rwl`, which increases the cost of adding nodes. As the system grows, however, it is able to handle faster rates of node arrival, resulting in an exponential growth; our flexible overlay allows the system to run multiple `join` operations concurrently, all of which execute within a confined (randomly selected) part of the network. During these experiments, we do not observe any scalability bottlenecks. We expect Atum to continue to exhibit this behavior (good scalability and exponential growth speed) in systems well beyond 1400 nodes, so, in the interest of time and budget, we choose to not experiment further.

Note the glitch around second 3000. The growth rate drops slightly due to temporary asynchrony, after which many nodes join in a burst and the system continues to grow normally. The short plateau after the burst is caused by the delay in creating and booting Amazon instances.

6.1.2 Churn tolerance

Next, we provoke continuous churn by constantly removing and re-joining nodes, for systems of up to 800 nodes. As we show in Figure 7, Sync can churn up to 18% of all nodes every minute, and Async reaches 22.5%. The nodes have an average session time between 5 and 6 minutes.

We use Sync to also evaluate how the choice of overlay parameters affect Atum's behavior under churn. The relevant parameters are random walk length (`rwl`) and number of H-graph cycles (`hc`). We use two combinations of (`rwl`, `hc`): $(6, 8)$ and $(11, 5)$. The intuition is that random walks are heavily used during churn, so we expect that a smaller `rwl` allows higher churn rates. Figure 7 confirms this intuition. The decrease in `rwl` does not translate, however, to a proportional increase of churn rate, because other sub-protocols also affect this rate, e.g., random number generation or SMR inside vgroups. Since the behavior of Async is less predictable, this effect, although present, is less prominent for Async, and we use a different configuration for each system size according to our configuration guideline.

As our configuration guideline (Figure 4) shows, if we decrease `rwl` (y-axis), we have to increase `hc` (x-axis) to preserve the random walks' uniform sampling property. A bigger `hc` means that groups have more neighbors, so nodes keep more state, which leads to bulkier state transfers. Going from `hc`= 5 to 8, however, turns out to have a smaller impact on the churn rate than the change of `rwl`.

6.1.3 Group communication latency

In this experiment, we instantiate a system and then disseminate 800 messages of length 10 to 100 bytes (comparable to Twitter messages). We experiment with system sizes of 200, 400, and 800 nodes in a failure-free case. To also evaluate Atum in the presence of faults, we use 800 correct nodes and subsequently add 50 (5.8%) nodes with injected faults.

To simulate arbitrary behavior of Byzantine nodes in Sync, we modify their algorithm such that they do not participate in any protocol except: (1) they send heartbeats, to prevent being evicted from the system (see §5.1); and (2) they pretend not to receive heartbeats from correct nodes, and periodically propose to evict all correct nodes from their vgroup. A Byzantine node has no incentive to send spoofed or corrupted messages while the majority of the nodes in its vgroup is correct; the recipient of such messages would discard them. To set the system parameters, we use the configuration guideline (§3.2), and we use rounds of 1.5 seconds. For Async, faulty nodes have no incentive to send corrupted messages, and therefore stay quiet.

We depict a CDF of the obtained latencies in Figure 8. For all scenarios with Sync, the latency has an upper bound of 8 rounds (12 seconds). The two phases of `broadcast` contribute independently to this latency. Nodes in the same vgroup with the publisher deliver a message immediately after the first phase, and nodes in other vgroups deliver it in the second phase (§3.3.4). We normalize the results to correspond to the expected latency of the first phase, which is 4 rounds in these experiments. The actual latency might differ by up to 2 rounds, depending on the size of the publisher's vgroup. Since faulty nodes do not reach majority in any vgroup, they do not affect correct nodes. Thus, Atum suffers no performance decay despite 5.8% faults, and the latency remains unchanged.

Figure 8 also shows how Sync compares with the two approaches it combines: synchronous SMR and gossip. The first baseline is a simulation of a classic round-based crash tolerant gossip protocol [29], with no failures. Every node has a global membership view and in every round exchanges messages with random nodes. To ensure fair comparison,

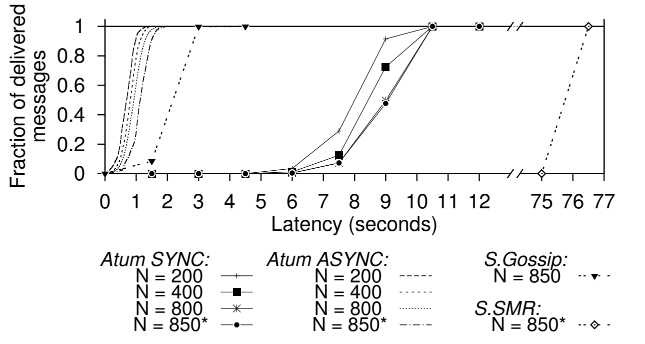

Figure 8: Group communication latency: Comparison between gossip, Atum, and SMR. We tag with * the systems with 50 faults.

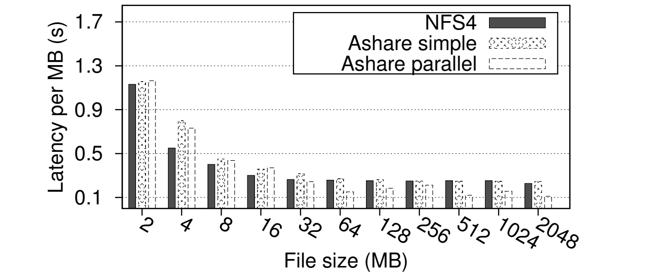

Figure 9: AShare: Read performance (latency per MB). We normalize the result to file size.

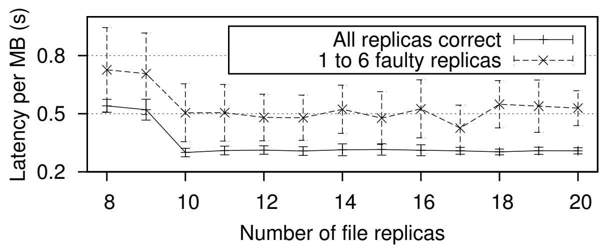

Figure 10: AShare: Impact of Byzantine nodes on read latency. Experiment with 50 nodes (7 Byzantine) and 500 files.

we set the fanout of this gossip protocol (i.e., the number of message exchanges per round) to the size of the view of a Atum node (this is a loose upper bound on the Atum fanout), and rounds to 1.5 second. As Figure 8 reveals, the latency penalty in Atum corresponds roughly to the latency of the SMR protocol in the first phase of `broadcast` (which is 4 rounds), with minimal additional overhead. This is the price we pay for Byzantine fault tolerance. The second baseline is the Byzantine agreement protocol [32] that we use to implement SMR in SYNC, when we scale it out to the whole system. The latency for this protocol is $f+1$ rounds (1.5 seconds each), where f is the number of tolerated faults.

Latencies for ASYNC are much lower, since there are no synchronous rounds, and so nodes do not need to coordinate their steps at a conservative rate. In contrast to SYNC, however, the tail latency reaches 105.5s, with less than 0.01% notifications delayed by more than 5s. With ASYNC, a small number of temporarily slow nodes might deliver notifications late, without affecting other nodes. To compensate for the lower fault tolerance of ASYNC ($\lfloor (g-1)/3 \rfloor$ instead of $\lfloor (g-1)/2 \rfloor$), we increase the robustness parameter `k` to 7, which results in a latency increase due to larger vgroups.

Failure rates observed in practice are about 0.5% nodes per day in a datacenter, according to Barroso et al. [11]. In this experiment, we tolerate 5.8% faults thanks to our logarithmic grouping and random walk shuffling schemes. In fact, the number of faults that Atum tolerates increases with system size. This is intuitive, given that larger systems imply larger vgroups (vgroups are roughly logarithmic in system size), so each vgroup can handle more faults.

We obtain the results hitherto using ASub, since this application maps exactly to the group communication API of Atum. Nevertheless, these results evaluate the basic operations of Atum, so they apply to all applications built on Atum, including AShare and AStream. In the following sections, we evaluate particular metrics for these two applications; this evaluation is orthogonal to the underlying GCS and independent of the group communication performance.

6.2 Evaluating AShare

We first evaluate the performance of GET in failure-free runs. This operation is equivalent with reading an entire file in a typical distributed filesystem, so we use NFS4 as baseline. By default, NFS4 has no fault-tolerance guarantees and is the standard solution for accessing files across the network. Results show that we provide comparable performance with NFS4, while offering stronger guarantees.

Figure 9 shows our results. We normalize the read latency to file size, so the y-axis plots latency/MB, using files from 2MB to 2GB. We consider three cases: (1) NFS4, where a client reads from a server; (2) AShare *simple*, where a node GETs files replicated by another node and the files have a single chunk – this is for fair a comparison with NFS4; and (3) AShare *parallel*, where a node reads files replicated by two other nodes and each file has 10 chunks. As we can see, the normalized read latency decreases as file size increases, because the constant overhead for transfer initiation (e.g., handshakes, TCP slow-start [6]) amortizes as transfer time grows. While the AShare simple execution can match the performance of NFS4 for larger files, we observe that the parallel execution outperforms NFS4 by up to 100% for files over 512MB. We attribute this gain to the use of parallel pulling and multithreaded digest computation.

We also study how Byzantine nodes impact GET latency. A Byzantine node in this scenario corrupts all the replicas it stores. We analyze the read latency in two scenarios – a 50-node system with 500 files, and a 100-node system with 1000 files. In both scenarios we set $\rho = 8$, so each file has a minimum of 8 replicas, and 7 random nodes are Byzantine. Every file consists of 10 chunks, with a fixed size of 1MB.

Figure 10 conveys AShare's resilience to corrupted replicas in the 50-node system. For moderately-replicated files, with 8 or 9 replicas, the read latency increases by up to 3x. This is expected, given that the majority of the pulled chunks are corrupted and have to be re-pulled from a correct node. We also observe that the positive effect of having multiple replicas per file diminishes. In the ideal configuration, the number of chunks equals the number of replicas of a file, such that each chunk can be pulled from a separate node and verified in parallel. In this configuration we can strike a balance between storage overhead (replicas count) and read latency. This can be seen in our results for files with 10 replicas. We draw similar conclusions from the results of the 100-node experiment in Figure 11.

We also used the Grid'5000 experimental platform for AShare evaluation, both for the failure-free and Byzantine scenario. Compared to EC2, we experimented on better machines (Xeon or Opteron CPUs, more memory) on a network with similar properties. We omit the results for brevity, as they are consistent with the results on EC2.

6.3 Evaluating AStream

To evaluate AStream, we consider a 1MB/s stream, which is an adequate rate for live video. As discussed in §3.3.4, the `forward` callback allows applications to customize the way Atum disseminates data in the second phase of `broadcast`. Specifically, applications like ASub would favor latency and flood the system by gossiping on all the H-graph cycles. In AStream, latency is not critical, so we customize the `forward` callback to use either one (*Single*) or two (*Double*) H-graph cycles. For SYNC, we set the round duration to 1 second. We run experiments with 20 and 50 nodes. In parallel with the first tier, the second tier transfers the data as soon as Atum delivers the digests.

In Figure 12 we plot the latency of the second tier of AStream. As we show, changing the `forward` function has an impact on dissemination. As expected, if we use more cycles to disseminate metadata, latency decreases. Since we use a lightweight multicast protocol in the second tier, this has a small impact on latency. For instance, in the 20-node system, Single scenario, the second tier takes .7 seconds; the total latency is 5.7 seconds with SYNC (which incurs 5s latency) or 1.7 seconds with ASYNC (which incurs 1s).

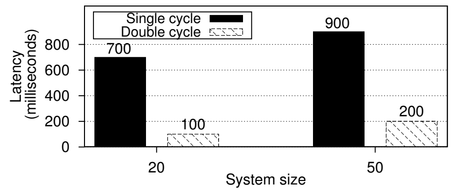

Figure 12: AStream: Latency for 1MB/s data stream.

7. EXPERIENCES AND LESSONS LEARNED

A big challenge we faced concerns the fundamental tradeoff between flexibility and robustness. On the one hand, Atum is designed to be flexible and adapt to high churn with ease: It resizes, merges and splits vgroups as befitting. On the other hand, to uphold robustness, Atum is also designed to restrict how vgroups evolve, placing bounds on their size, and composing them via random sampling. To convey how this trade-off manifests in practice, we strain Atum by joining nodes at an overwhelming rate; this generates many concurrent shuffle operations, and suppresses some node exchanges, because the chosen exchange partner already participates in another exchange. In our experiments (§6.1.1) we join nodes at a rate of 8% of the system size each minute. Figure 13 shows what happens when we intensify this rate to 20% and 24%: Higher growth rates suppress more node exchanges (penalizing robustness), but the system grows faster (is more flexible).

Another hard challenge was the interplay between the SMR algorithm inside vgroups and the distributed protocols running among vgroups. The two most salient issues here were the following. First, SMR reconfiguration by itself is a tricky business [12, 58]. The shuffle operation in Atum, however, involves multiple vgroups concurrently reconfiguring by exchanging nodes among themselves. Complications with this include deadlocks, missed operations, duplicate membership (nodes belonging to two different vgroups), dangling membership (nodes being left out of a vgroup), or other inconsistencies. Second, the H-graph overlay we use is decentralized and random, where every node has only a local view; this makes the split operation particularly intricate because it involves orchestration among multiple vgroups, on all cycles of the graph.

These challenges compelled us to simplify the implementation at all levels. So we initially considered a synchronous SMR [32], and implemented SYNC in $20K$ LOC in C. This decision is reasonable for highly-redundant networks (such as inside a datacenter availability zone [68]), but is not realistic for large, dynamic networks, where the round size has to be very conservative. Since we wanted a comprehensive evaluation, including on WAN, we decided to also implement ASYNC, based on asynchronous SMR, i.e., assuming eventual synchrony [20]. This version is more complex, but we used a high-level language ($8K$ LOC in Python), which helped us deal with the complexity.

As our results show, ASYNC outperforms SYNC. On the other hand, SYNC brings multiple benefits: It has predictable

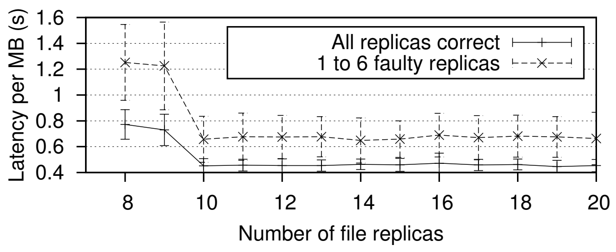

Figure 11: AShare: Impact of Byzantine nodes on read latency. Experiment with 100 nodes (7 Byzantine) and 1000 files.

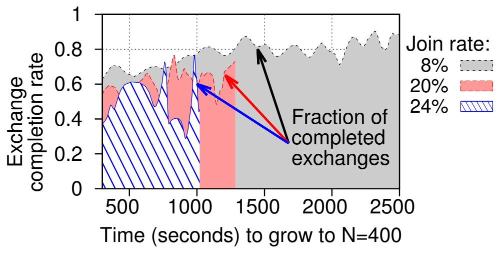

Figure 13: As a system grows faster, the quality of random vgroup composition suffers due to suppressed exchanges.

performance, is simpler to implement and reason about (due to the round-based model), and is thus less prone to bugs; moreover, SYNC was an important stepping stone to help us understand the complex interactions in Atum.

8. RELATED WORK

The work presented in this paper spans multiple research areas, including BFT, P2P applications, pub/sub, and file sharing. We review some of the systems which intersect with our work and highlight the differences.

S-Fireflies [30] is an effective self-stabilizing system for data dissemination, robust to Byzantine faults, and churn-tolerant. It creates random permutations of the system nodes and uses them to pick the neighbors for each node. A difference between S-Fireflies and Atum is that the former relies on global knowledge of all nodes at all times, meaning that every node of the system is aware of every other node.[7]

Scatter [41] is a distributed key-value store with linearizable operations. As Atum, Scatter also partitions the system into self-organizing, dynamic groups of nodes, and is churn-tolerant. Scatter focuses on performance and strong consistency at large scale, while in Atum, our main objective is large-scale BFT. Inside groups, Scatter relies on Paxos; across groups, it achieves coordination using a 2PC-based protocol. Atum ensures stronger fault-tolerance guarantees inside groups (by using a BFT agreement [32]) and coordinates groups using a scalable gossip scheme.

FlightPath [53] is a powerful P2P streaming system. It is robust towards rational and Byzantine nodes, and it tolerates high churn rates. In FlightPath, the focus is on streaming data efficiently from a designated source node, using gossip and a centralized tracker. In our work we also leverage gossip; in addition, Atum is a completely decentralized, general-purpose GCS, it does not rely on a central tracker, and it allows *any* node to broadcast.

Membership services are an important middleware for data dissemination [35, 43]. These services often rely on the non-determinism of a sampling protocol to obtain random connections with other peers and ensure low-latency and robust dissemination. In Atum, our sampling scheme is based on random walks over a well-connected H-graph. Other systems, such as RaWMS [10] or the wormhole-based peer sampling service (WPSS) [64], also leverage random walks to implement efficient sampling. In contrast with these systems, Atum provides a complete solution for data dissemination, which also handles high churn rates and Byzantine failures.

Gossip-based systems typically exploit randomness to bypass failures and ensure robust dissemination [29]. Vicinity [71] is a protocol for constructing and maintaining an overlay network, which investigates the importance of randomness in large-scale P2P networks. The analysis around this protocol shows that randomness must be complemented with structure (determinism) for effective large-scale P2P networks. In Atum, we also leverage non-determinism (shuffling) alongside determinism (SMR) to scatter faulty nodes and handle churn in volatile groups.

DHTs such as Chord [69] or Tapestry [75] provide $O(\log N)$ lookup time and are commonly used for routing in storage system. DHTs can be adapted to handle Byzantine faults [17, 36, 38, 63], and the problem of churn has also been addressed in [54, 60]. Tentative lookup schemes that are both secure and churn-friendly are given in [74], where session times as low as 10 minutes are theoretically explored. In Atum, we focus on general-purpose group communication – instead of lookup – and we use the novel concept of volatile groups to allow even shorter session times.

BFS [20], Farsite [4], Pond [61], and Rosebud [62] are file storage systems that use PBFT [20]. In BFS, the replicas running PBFT also store the data objects, so this system fully replicates data across all the nodes and cannot scale well. Pond, Rosebud, and Farsite achieve scalability by separating the BFT mechanism from the storage subsystem. They use BFT quorums to agree on the operations that are performed, and the storage nodes perform these operations. The BFT and storage subsystems can scale independently in this case. Although these four systems assume a dynamic environment, only Rosebud provides details on this concern.

In Rosebud, a group of BFT replicas agree on the system configuration and monitor all the nodes in the system; this group periodically – once per *epoch* – propagates new configurations. The churn rate in Rosebud thus depends on the length of epochs. This system has an evaluation with epoch duration of a few hours; shorter epochs are possible but are not considered. Atum can cope with session times in the order of minutes (§6.1.2). Unfortunately, we were not able to find a Rosebud implementation to use for comparison.

9. CONCLUSIONS

This paper reports on our experiences with designing and building Atum, a general-purpose group communication middleware for a large, dynamic, and hostile network. At the heart of Atum lies the novel concept of volatile groups, i.e., small, dynamic, yet robust clusters of nodes. Specifically, Atum applies state machine replication at small-scale, inside each vgroup, and uses gossip to disseminate data among vgroups. We ensure that vgroups are robust and efficient by employing two techniques, namely random walk shuffling and logarithmic grouping.

We experimented with two Atum implementations – one synchronous and one asynchronous (eventually synchronous). We used Atum as a reliable core to build three applications: ASub, a publish/subscribe service, AShare, a file sharing system, and AStream, a data streaming platform. Using these applications, we verified experimentally the properties of our system: Our findings indicate that Atum tolerates arbitrary faults even in a large-scale, high-churn network.

The Atum library, together with our applications, is available online at `http://lpd.epfl.ch/site/atum`.

10. ACKNOWLEDGMENTS

We thank our colleagues from the LPD laboratory at EPFL, as well as the anonymous reviewers, for their thoughtful comments. They helped to substantially improve the quality of this paper. We also thank Mahammad Valiyev for his help with the implementation of SYNC. This work has been supported in part by the European ERC Grant 339539 - AOC and the Swiss National Science Foundation (FNS) grant 20021_147067.

[7]We also assume global knowledge in one of our applications (AShare, §4.2), but this assumption is not inherent in Atum.

11. REFERENCES

[1] https://www.sqlite.org/.

[2] https://aws.amazon.com/ec2/instance-types/.

[3] Amazon S3 Availability Event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI), 2002.

[5] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: a Client Notification Service for Internet-Scale Applications. In *SOSP*, 2011.

[6] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[7] P.-L. Aublin, S. B. Mokhtar, and V. Quéma. Rbft: Redundant byzantine fault tolerance. In *ICDCS*, 2013.

[8] B. Awerbuch and C. Scheideler. Towards Scalable and Robust Overlay Networks. *IPTPS*, 2007.

[9] B. Awerbuch and C. Scheideler. Towards a Scalable and Robust DHT. *Theory of Computing Systems*, 45(2), 2009.

[10] Z. Bar-Yossef, R. Friedman, and G. Kliot. RaWMS - Random Walk Based Lightweight Membership Service for Wireless Ad Hoc Networks. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[11] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: an introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 8(3):1–154, 2013.

[12] A. Bessani, J. Sousa, and E. Alchieri. State Machine Replication for the Masses with BFT-SMART. In *DSN*, 2014.

[13] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable state-machine replication. In *DSN*, 2014.

[14] K. P. Birman. Replication and Fault-tolerance in the ISIS System. In *SOSP*, 1985.

[15] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36, 1993.

[16] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1985.

[17] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. *ACM SIGOPS Operating Systems Review*, 36(SI), 2002.

[18] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: high-bandwidth multicast in cooperative environments. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 298–313, 2003.

[19] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8), 2002.

[20] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[21] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM TOCS*, 26(2), 2008.

[22] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *WREN*, 2009.

[23] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4), 2001.

[24] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end Containment of Internet Worms. In *SOSP*, 2005.

[25] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, 2006.

[26] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[27] G. Danezis, C. Lesniewski-laas, M. F. Kaashoek, and R. Anderson. Sybil-resistant dht routing. In *In ESORICS*. Springer, 2005.

[28] J. Dean. Designs, lessons and advice from building large distributed systems. *Keynote from LADIS*, 2009.

[29] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. E. Sturgis, D. C. Swinehart, and D. B. Terry. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1), 1988.

[30] D. Dolev, E. Hoch, and R. Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *OPODIS*, 2007.

[31] D. Dolev and D. Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), 1996.

[32] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4), 1983.

[33] J. R. Douceur. The sybil attack. In *IPTPS*. Springer, 2002.

[34] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2), 2003.

[35] C. Fetzer and F. Cristian. A fail-aware membership service. In *Reliable Distributed Systems*, 1997.

[36] A. Fiat, J. Saia, and M. Young. Making Chord Robust to Byzantine Attacks. *Algorithms–ESA*, 2005.

[37] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[38] R. Geambasu, J. Falkner, P. Gardner, T. Kohno, A. Krishnamurthy, and H. M. Levy. Experiences building security applications on DHTs. Technical report, Technical report, UW-CSE-09-09-01, 2009.

[39] H. Geng and R. van Renesse. Sprinkler - Reliable Broadcast for Geographically Dispersed Datacenters. In *Middleware 2013*. 2013.

[40] S. Ghemawat, H. Gobioff, and S.-T. Leung. The

Google File System. In *SOSP*, 2003.

[41] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

[42] R. Guerraoui, F. Huc, and A.-M. Kermarrec. Highly dynamic distributed computing with byzantine failures. In *PODC*, 2013.

[43] M. A. Hiltunen and R. D. Schlichting. The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, 2000.

[44] H. Johansen, A. Allavena, and R. Van Renesse. Fireflies: scalable support for intrusion-tolerant network overlays. *ACM SIGOPS Operating Systems Review*, 40(4), 2006.

[45] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *ICDCS*, 1991.

[46] A. Kate, Y. Huang, and I. Goldberg. Distributed key generation in the wild. *IACR Cryptology ePrint Archive*, 2012.

[47] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[48] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*, Oct. 2007.

[49] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. *SIGPLAN Not.*, 2000.

[50] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[51] C. Law and K.-Y. Siu. Distributed construction of random expander networks. In *INFOCOM*, 2003.

[52] C. Lesniewski-Lass and M. F. Kaashoek. Whanau: A sybil-proof distributed hash table. In *NSDI*, 2010.

[53] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin. FlightPath: Obedience vs. Choice in Cooperative Services. In *OSDI*, 2008.

[54] Z. Liu, R. Yuan, Z. Li, H. Li, and G. Chen. Survive under high churn in structured P2P systems: evaluation and strategy. In *ICCS*, 2006.

[55] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The smart way to migrate replicated stateful services. In *EuroSys*, 2006.

[56] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system.

[57] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *EuroSys*, 2011.

[58] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. *USENIX ATC*, 2014.

[59] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *USITS*, 2003.

[60] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *USENIX*, 2004.

[61] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *FAST*, volume 3, 2003.

[62] R. Rodrigues and B. Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical Report MIT-LCS-TR-932 and MIT-CSAIL-TR-2003-035, 2003.

[63] R. Rodrigues, B. Liskov, and L. Shrira. The design of a robust peer-to-peer system. In *ACM SIGOPS European workshop: beyond the PC - EW10*, 2002.

[64] R. Roverso, J. Dowling, and M. Jelasity. Through the wormhole: Low cost, fresh peer sampling for the internet. In *Peer-to-Peer Computing (P2P)*, 2013.

[65] C. Scheideler. How to spread adversarial nodes?: Rotate! In *STOC*, 2005.

[66] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[67] B. Schroeder, S. Damouras, and P. Gill. Understanding latent sector errors and how to protect against them. In *FAST*, 2010.

[68] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking data centers randomly. In *NSDI*, 2012.

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

[70] R. Van Renesse, K. P. Birman, and S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4), 1996.

[71] S. Voulgaris and M. van Steen. Middleware, 2013.

[72] L. Vu, I. Gupta, J. Liang, and K. Nahrstedt. Measurement and modeling of a large-scale overlay for multimedia streaming. In *QSHINE*, 2007.

[73] H. Weatherspoon, P. Eaton, B.-G. Chun, and J. Kubiatowicz. Antiquity: exploiting a secure log for wide-area distributed storage. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

[74] M. Young, A. Kate, I. Goldberg, and M. Karsten. Practical Robust Communication in DHTs Tolerating a Byzantine Adversary. In *ICDCS*, 2010.

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