Maintaining Replicas in Unstructured P2P Systems

Christof Leng
TU Darmstadt
cleng@dvs.tu-darmstadt.de

Wesley W. Terpstra
TU Darmstadt
terpstra@dvs.tu-darmstadt.de

Bettina Kemme
McGill University
kemme@cs.mcgill.ca

Wilhelm Stannat
TU Darmstadt

Alejandro P. Buchmann
TU Darmstadt

ABSTRACT

Replication is widely used in unstructured peer-to-peer systems to improve search or achieve availability. We identify and solve a subclass of replication problems where each object is associated with a maintainer node, and its replicas should only be available as long as its maintainer is part of the network. Such requirement can be found in various applications, e.g., when objects are directory lists, service lists, or subscriptions of a publish/subscribe system.

We provide maintainers with proven guarantees on the number of replicas, in spite of network churn and crash failures. We also tackle the related problems of changing the number of replicas, updating replicas, balancing storage load in a heterogeneous network, and eliminating replicas left by crashing maintainers. Our algorithm is based on probabilistic methods and is simple to implement. We show by simulation and formal proof that our algorithm is correct.

1. INTRODUCTION

A common use of peer-to-peer systems is as an object store. In its role as a *client*, a peer can create objects that are stored in the system and inject queries that find objects with certain properties. In its role as a *server*, a peer provides storage capacity for objects and it answers queries for objects stored locally. If the peer-to-peer network is *unstructured*, objects are not placed on any particular nodes. Queries are forwarded to a subset of all nodes, and each node receiving such a query request checks whether its local objects fulfill the query. In comparison, in a *structured* network each object is placed on a specific node (or subset of nodes), typically the nodes whose hashed node ids are closest to the hashed value of one or more attributes of the object. Queries with search keys on these attributes can then be easily routed to the nodes that contain matching objects. However, other types of queries are usually not well supported. In contrast, unstructured networks easily support complex queries on arbitrary search attributes and any kind of query language. This allows them to reuse traditional implementations for query execution and object storage.

In early unstructured systems, search used simple forwarding mechanisms such as flooding or random walks, and thus, were often inefficient or unreliable. Modern unstructured overlays like BubbleStorm [23] or the similar approach in [9] provide reliable and exhaustive search even in very large networks. These systems use a large number of replicas for each object placed randomly in the overlay to enable their search algorithms (typically $O(\sqrt{n})$). Maintaining the desired number of replicas is challenging as peer-to-peer networks are very dynamic with nodes constantly joining and leaving the system, often by failing silently. If no corrective actions are taken, an object will loose replicas quickly, spoiling search reliability. Replica maintenance must keep the number of replicas for each object at the desired level.

In order to find appropriate replica maintenance mechanisms one has to first understand the requirements of the different applications. The most well-known application for peer-to-peer systems is file-sharing. In file-sharing scenarios it might be prohibitively expensive to create many copies of the files themselves as they are typically large binary objects such as videos. Thus, they usually have only replicas at few nodes. Instead, what is widely replicated are file descriptors and directory lists. A file descriptor contains the file id and a set of attributes describing properties of the file. A directory list contains for a peer the file ids of the files it stores. A user who wants to find files with certain properties first poses a query over the file descriptors retrieving the file ids of matching files. Then it poses a query over the directory lists to find the peers that store the matching files. Finally, it connects to these peers to download the files.

Although looking similar in concept, file descriptors and directory lists have different requirements regarding their lifetime. The directory list of a node should only exist in the system as long as this node is up and running. When the node leaves, all the copies of its directory list should disappear as the node cannot serve any file requests anymore. When

ACM CoNEXT 2008, December 10-12, 2008, Madrid, SPAIN

a peer joins the system, its directory list should be posted again. In contrast, the file descriptors should always remain in the system independently of which peers are currently up and running. With this, we identify two types of replica maintenance: *collective* and *maintainer-based*. In the collective mode, sustaining the desired number of replicas is the shared responsibility of all nodes. In the maintainer-based mode each object has a single node, called the maintainer, that is responsible for sustaining its replication degree. If a maintainer leaves the network, its objects should cease to exist. In the above example, the file descriptors might use collective maintenance while the directory lists should use maintainer-based maintenance.

Collective replication has been a subject of research already (see Section 8) but the maintainer-based approach is a new category. There are many more applications for the maintainer-based approach beyond file-sharing. For example, nodes might want to publish service lists, indicating the services they provide, and users query the system for nodes that offer certain services. In this case, the lifetime of the replicated object, namely the service list, is tied to the lifetime of the node offering these services. A dual to the query/store paradigm is the publish/subscribe paradigm. Here, a node puts out a persistent query, called subscription, which represents a request to be notified about certain events. For instance, a subscription could be "report Air Canada flights arriving in New York". When an event is published, all the subscriptions that match this event must be found. Thus, in some sense, the roles of data and query have been flipped; the query is persistent and replicated instead of the information. However, subscriptions are only relevant as long as the subscribing node is connected. They should be removed when the node leaves the system.

The question arises whether maintainer-based replication needs a special solution or can be modeled as a special case of collective replication. We argue for an independent solution as a maintainer-based approach has very different needs and implementation options. Subscriptions and service lists should disappear from the overlay quickly after the departure of their owner while collective replication should ensure that objects remain in the system no matter which set of nodes leaves. Manually deleting replicas on departure seems an obvious solution, but crashing nodes would not execute this leave algorithm. The resulting outdated *junk* replicas must be eventually removed. It is here that our solution departs significantly from collective replication.

Furthermore, maintainer-based replication solutions can take advantage of the existence of the maintainer. For instance, concurrent updates are not a problem because the maintainer can be used to serialize them. Also, the maintainer can easily keep track of nodes holding replicas. This makes it easy to update objects in place and to delete objects before a planned departure. Finally, a maintainer ensures that not all replicas of the maintained object disappear (it stores one itself). This provides durability.

Motivated by above discussion, this paper proposes a complete solution for maintainer-based replication in unstructured peer-to-peer networks. A particular challenge is that churn (i.e. continuous changes in the network configuration) is a major issue in peer-to-peer systems. Any solution must cope with frequent joins and leaves (voluntary or by crash), intermittent connectivity, and lost messages. Our general approach to this challenge is to adopt probabilistic methods that offer guarantees with high probability but do not block or leave inconsistencies in case of failures. The paper splits the discussion in several sub-topics.

Section 4 will discuss how to preserve the replication degree for *valid* objects, i.e., objects whose maintainer is online. We express the desired replication degree for an object in terms of density, i.e., the percentage of online peers in the network that should have a replica. Maintaining the replication density requires action when the network configuration changes. Additionally, maintainers have the ability to dynamically increase or decrease the desired density of each of their objects with little overhead. This property is important as the desired replication could depend on the popularity of the object or the type and size of the network.

Then, Section 5 discusses how to clean up replicas from leaving maintainers. While voluntarily leaving maintainers can actively eliminate their object replicas, the object replicas of crashed maintainers remain as junk. Our solution limits the amount of junk left in the system. We do this by letting each peer periodically flush its stored replicas and load new replicas. Although this seems expensive, our analysis shows that the overhead is quite acceptable. Furthermore, this section outlines why the more traditional approaches, such as storage leases or ping-mechanisms will not work well in peer-to-peer overlays with no reliability guarantees.

Section 6 presents additional functionality in our system. First, we discuss how updates can be performed in the system. We do this in place. Second, we show how the approach handles a heterogeneous environment where nodes can specify their capacity. Only powerful nodes will become peers storing replicas, and their capacity will determine the number of replicas the peer will store. Still, even weak nodes which cannot become peers due to insufficient storage or bandwidth capacity can be maintainers, pushing their objects into the system. Section 7 examines a simulated implementation of our solution. The results confirm its correctness and give insight into the performance costs.

In summary, our solution for maintainer-based replication

- keeps the replication degree at the desired level;
- can dynamically adjust the replication degree;
- eliminates replicas of offline maintainers over time and bounds the total amount of such junk;
- allows objects to be deleted and updated;
- balances load even in heterogeneous environments.

2. MODEL AND ASSUMPTIONS

In our model of an unstructured peer-to-peer network, any node can create objects that are then replicated in the system. These replicas must be maintained until that node leaves, whereupon all replicas should disappear. Each node with at least one associated object is called a *maintainer*. Maintainers are responsible for creating the initial set of replicas for their objects. They can decide to change the replication density of each of their objects at any time. An object can be changed or deleted by its maintainer.

We further distinguish between peer and client nodes. A peer is a node willing to store object replicas and provide processing capacity to run queries over its replicas. In contrast, a client does not contribute storage space or processing power. The reason to let a node opt out of being a peer is that many nodes come with too little bandwidth or processing capacity to perform peer tasks. Nevertheless, we want to allow them to be maintainers (post their objects) and pose queries on the objects in the system.

Our solution requires the underlying infrastructure to provide certain functionality. First, we require a mechanism that computes distributed sums and maxima. We use these to estimate a few global statistics, e.g., the number of peers in the system and the maximum desired replication density. There exists substantial prior work [13, 16, 18] for providing these services. We must also be able to independently sample nodes with uniform probability. Biased random walks [2, 8] are one solution. Finally, we require a *push* algorithm for injecting replicas onto many peers at the same time. In our solution, the maintainers use this algorithm whenever they create an object or increase replication.

3. BUBBLESTORM EXAMPLE

As an example unstructured peer-to-peer network, consider the BubbleStorm system [23]. Peers are interconnected by a random graph. Whenever an object is created, a set of object replicas, called the object bubble, are pushed onto random peers using the bubblecast algorithm. When a query is posed, it is likewise pushed to a random set of peers, called the query bubble. Each peer in the query bubble checks whether it holds object replicas that match the query. By making both the object and query bubbles large enough, the probability is very high that for any object that matches the query, the query and object bubbles overlap on at least one peer. Bubble sizes are normally on the order of the square root of the number of peers, depending on query frequency and object type. For instance, for a network of 1000000, an object bubble of 3000 and a query bubble of 3000, the probability to reach an object replica is nearly 99.99%. Using this technique, BubbleStorm achieves quite reliable search with a much lower overhead than traditional unstructured networks. However, BubbleStorm does not properly maintain the object bubble's required replication degree under churn. Our work presented here will close this gap.

BubbleStorm uses a gossip protocol in the spirit of [13] that is able to provide the sums and maxima we require. Furthermore, BubbleStorm already has a push mechanism to efficiently create replicas on nodes at random. BubbleStorm will serve as a running example during our discussion. However, our solution also applies to other modern unstructured networks such as [9, 20].

4. MAINTAINING REPLICATION

When a maintainer creates a new object, it decides on its replication degree d and pushes d replicas into the network using the system's push algorithm. Thereafter, this level of replication must be maintained as long as the object is *valid*, i.e., until the object is deleted or the maintainer leaves the network. This section looks at this maintenance task.

If there were no crashes in the system, replica maintenance would be easy: Before leaving, a peer contacts the maintainers of object replicas it stores and these maintainers create replacement replicas. Similarly, when a maintainer leaves it informs all replica holders to delete their copies. However, node crashes and intermittent network disconnections are very frequent in peer-to-peer systems, making any such agreement or coordination protocol complex and costly.

Thus, our solution follows a different strategy based on probabilistic methods. Instead of trying to keep a fixed number of replicas, maintainers hold the *probability distribution* of replication fixed. Maintainers express their desired replication in terms of an object *density*, p. Every object can potentially have a different density. The density is the percentage of peers which should store a replica of the object. If there should be d replicas and n peers in the system, then the desired density of an object is $p = d/n$. As long as the system remains in an equilibrium (i.e., churn occurs but n remains roughly the same), keeping the density at p will lead to the desired number of replicas. Given a fixed density, our algorithm causes replication to grow linearly with the system size, which is undesirable for some systems. Therefore, density can be changed dynamically. For instance, for the BubbleStorm system it could be adjusted as a function of system size n, i.e., $p = 1/\sqrt{n}$. Density could also be adjusted with changing document popularity or with the time of day. But we assume that it does not fluctuate excessively.

Replica maintenance must keep the object density at the requested level. This entails distinct tasks. First, a given object density has to be preserved when peers join or leave the system. Second, when the maintainer increases (decreases) the density p of an object, replicas must be created (deleted). In the following, we describe these tasks in detail.

For now, we assume that all nodes have equal capacity and participate as peers. We remove these restrictions in Section 6. When peers have the same capacity, each peer stores on average the same number of replicas. Thus, if the density of an object is p, then the probability that an arbitrary peer stores a replica of the object is also p.

4.1 Peers: Preserving Replica Density

Leaving Peers. Leaving peers are a simple case since nothing needs to be done. Although the system loses replicas residing on the peer, the expected density p of each object remains the same, because every remaining peer still has a replica with probability p.

Having nothing to do on peer departure is attractive. Assume corrective actions were required. Since a crashing node by definition does not perform any actions when it leaves, such corrective actions would need to be initiated by surviving nodes, for example, by the maintainers of the replicas that just disappeared. This would require pings or a similar technique to detect the missing replica. Until the reparative measures are taken, correctness is compromised. To mitigate this effect, pings would need to be frequent, which is extremely costly when there are many replicas. Recall that the BubbleStorm scenario maintains in the order of $\sqrt{n}$ replicas per object. Furthermore, transient network failures could interfere with replica failure detection. Transient failures are discussed in more detail in Section 5.2.

Even if peers could execute a proper exit procedure, this is not really beneficial. Assume, for example, they inform maintainers about their departure. As they store replicas for many maintainers, this could be time consuming. When a user quits a program or closes the laptop, there is little time available for a clean exit. Reliably informing maintainers of termination is also impossible. The peer cannot distinguish a crashed maintainer from a live maintainer with a transient failure and must block indefinitely. If the peer gives up after some timeout, this is no different from a crash. By not requiring leaving peers to act, we avoid these concerns.

Of course, leaving peers decrease the total number n of peers in the system, if new nodes don't join with the same rate. In the case of an actually decreasing n, maintainers might eventually change the requested density. However, this happens lazily. Maintainers are periodically informed about changes in n through the distributed sum calculation and without the need to know which peers exactly have left or whether these peers held replicas of their objects.

Joining Peers. In principle, if a peer leaves the system and rejoins, it could retain the replicas from its previous membership. However, in the maintained scenario, this has little benefit unless a peer is down for very short time. If a peer is offline for more than the median uptime, more than half of its replicas are junk, i.e., from maintainers who have left during the downtime. In practice, according to [19, 22] peers have a median online time of approximately 60 minutes while 2/3 of all peers did not return online within a month. Additionally, as maintainers can update or delete objects, even replicas from still live maintainers might be obsolete. Thus, in our approach each peer joining the system receives a completely new set of fresh replicas.

In contrast to the initial *push* of replicas when an object is created, assigning replicas to newly joining peers uses a *pull* strategy. In particular, when a peer v joins the system,

```
// updated by underlying system:
m = sum(1) // sum over all maintainers
n = sum(1) // sum over all peers
maxp = max(p) // max over all densities

f(p): return ceil(ln(1-p)/ln(1-1/m))

Upon joining network:  // by joining peer
 x = f(maxp);
 for i in [0, x):
   send pull(i, myAddress) to random maintainer;

Upon receiving pull(i, addr):  // by maintainer
 for obj in objects_maintained:
   if i < f(obj.get_p()):
     send(obj) to addr
```

Figure 1: Replicate on Join Algorithm

it must preserve the density of every valid object. Thus, for an object o with density p, peer v must create a replica of o with probability p. A brute-force approach would contact all maintainers and *pull* replicas of objects according to their required densities. Clearly, this is not scalable. Instead, we contact only a random subset of maintainers and replicate all of their objects. The question is how many maintainers to pull to preserve the density of objects in the system.

Assuming for the moment that all objects have the same density p, the number of maintainers to pull $f(p)$ depends on density p and total number m of maintainers in the system. A replica is created if and only if its maintainer is pulled. An object o is not replicated with probability,

$$\mathbf{P}(\text{maintainer of } o \text{ not pulled}) = \left(1 - \frac{1}{m}\right)^{f(p)}$$

Set $f(p) = \ln(1–p)/\ln(1–1/m) \approx pm$ then this probability is $1 - p$. Thus, a replica of o is created with probability p.

In our join algorithm (Figure 1), a joining peer v calculates the number $x = f(maxp)$ of maintainers to pull using the maximum density of all objects in the system. Then it sends pull requests to maintainers. Since it calculates the number of maintainers to be pulled using the maximum density, we have to be careful to not transfer too many replicas to the joining peer. For an object with density $p < maxp$, less than $f(maxp)$ maintainers should have been pulled. Therefore, the pull request includes how many other maintainers i were already contacted. Then, for an object with density p, a maintainer only sends the object when $i < f(p)$.

The algorithm described holds the probability distribution of replication degree fixed. In the Appendix we prove that any sequence of peer join and crash/leave events causes the replication distribution for an object with density p to converge to the binomial distribution $B_{n,p}$. In practise, this means the expected number of replicas is np as required and the standard deviation is $\approx \sqrt{np}$. When many replicas are required, it is quite likely the actual number of replicas is close to the required replication. At least for the Bub-

bleStorm system, this level of variance is acceptable and does not impact correctness. For systems requiring very few replicas, this variance might be too high, and ping methods are correspondingly cheaper. Thus, our solution is most applicable for systems with replication degree above 20.

4.2 Maintainers: Increasing the Density

The existence of a maintainer makes it easy to increase p. If p should be increased to q, the maintainer can create new replicas using the system's push algorithm. Assuming that pushed replicas can collide with each other and with old replicas, the new chance that a peer has a replica is,

$$1-(1-p)\left(1-\frac{1}{n}\right)^{x}=q \text{ when } x=\frac{\ln(1-q)-\ln(1-p)}{\ln(1-\frac{1}{n})}$$

Therefore, x replicas have to be pushed to peers to reach the new density. The resulting distribution will not be binomial. Nevertheless, over time the distribution will again converge to the binomial. Until then, the distribution has less variance.

4.3 Maintainers: Decreasing the Density

To decrease the replication from p to q, replicas must be eliminated. Suppose that every peer flips a coin with heads probability q/p. Peers with a replica keep it on heads (doing nothing) and delete it on tails. Peers without a replica do nothing. Peers now have a replica with probability $p*q/p = q$; the density has been correctly decreased.

We want to achieve exactly this behaviour but initiated by the maintainer. The maintainer could flip the coin for each peer and tell it to delete or keep the replica and the result would be the same. As an optimization, if the peer should do nothing, the maintainer does not need to notify it. Thus, the maintainer needs to contact only those peers that have a replica that should be deleted according to the coin toss.

For that purpose, the maintainer keeps a list of all peers which might have a replica. To build the list, a maintainer records the names of peers it pushes copies to. Also, joining peers pull their replicas directly from maintainers, allowing the maintainer to add them to the list. When p should be decreased, the maintainer flips a coin for each entry in the list and only contacts peers that should delete the replica.

When peers leave, the list is unchanged and remains a superset of peers with replicas. This does no harm because sending a delete request to an offline peer has no effect. If a peer in the list leaves and rejoins the system, it discards all the replicas of its old membership. If the maintainer asks it to delete the replica, it can simply (and correctly) ignore the request. It is easy to show that this mechanism converts the binomial distribution from $B_{n,p}$ to $B_{n,q}$.

For efficiency reasons the list should not contain a large fraction of peers that don't have replicas. Therefore, the maintainer regularly uses ping to check if replicas still exist, and shortens the list accordingly. This is correct as departed peers do not rejoin with replicas. Our simple rule of thumb is to ping whenever the list doubles in size.

In case of message loss or temporary disconnection the maintainer might remove peers from the list while in fact the peer is up and still has a replica. In this case, the system is over-provisioned, having more replicas than the requested density would imply. Such inconsistencies will be eliminated by the flush policy described in the next section.

5. CONTROLLING JUNK

Replicas need to be eliminated from the system when the maintainer goes offline. We call replicas with an online maintainer *valid*; the rest are *junk*. Junk not only costs peers storage space and traffic, but might also appear in query results. Junk cannot be completely avoided due to the unreliability of the peer-to-peer network. However, the amount of junk must be controlled.

Our approach provides a probabilistic bound on the junk in the system. For that we introduce the *goodness factor* $g \in (0,1)$. The parameter controls the desired percentage of valid replicas a peer stores on average. For instance, if $g = 0.8$, then on average 80% of replicas a peer stores are valid. When g is used as a system-wide parameter, then $1-g$ is the expected percentage of junk responses to a query. As junk accumulates, we have to remove it in order to keep it at the desired proportion. The goodness factor is a tradeoff between the overhead of removing junk and seeing an unacceptable high rate of junk appearing in query results.

5.1 Maintainers: Delete on Leave

Maintainers should delete replicas before leaving if they can in order to not produce unnecessary junk. We refer to this as a clean leave. Maintainers already know which peers store their replicas, so this is easy to implement. Unlike the converse situation where we rejected action on *peer* departure, here the deletes do not need to be reliable. Some or all of the delete messages can be lost, creating junk, but not compromising correctness. Thus clean maintainer departure can be very fast, sending unacknowledged delete requests to as many replica-storing peers as there is time to contact.

5.2 Peers: Periodically Flush

Crashing maintainers leave all their replicas in the system. Thus, junk accumulates over time. Unfortunately, maintainers often fail silently, so a peer can never know exactly which replicas are junk. A conventional eviction scheme might simply use pings as a failure detection mechanism. Using a ping a peer checks if a replica's maintainer is online and evicts the replica if the ping does not return. Within a peer-to-peer setting, however, such failure detection mechanism is very unreliable. It would eventually bias the system towards preferentially storing replicas from either reliable or short-lived maintainers. The problem is that an intransient network failure could cause the peer to incorrectly conclude that the maintainer is down. This false deduction will never be corrected, because once the replica is removed, neither party will place that replica onto the peer again. Temporary

failures thus become permanent failures. For particularly long-lived maintainers, these errors will slowly accumulate, decreasing the replication degree.

Thus our solution takes the simple but effective approach of flushing all replicas from long-lived peers (those who have enough junk to warrant action) and letting them re-execute the replica loading algorithm. This approach has less overhead than it appears at first view. Firstly, maintainer-based objects are typically small and relatively cheap to replace; e.g. service lists or subscriptions will rarely be larger than a few KB. Furthermore, junk is created by maintainer *crash* churn. Normal peer churn is probably significantly faster, implicitly flushing all replicas from affected peers (recall that a leaving peer removes all its replicas before rejoining). Thus, only long-lived replicas introduce additional overhead.

With this approach, we side-step the issue of temporary failures becoming permanent, not by making the system reliable, but by making the system forget. A transient failure could still cause the peer to miss a replica it should have stored, but on its next attempt it has a fresh chance to store it. Section 5.3 and the simulations show that our intuition is correct and flushing small objects is relatively inexpensive.

A junk eviction policy must also determine when to evict replicas. One could associate a lifetime with replicas or flush replicas from peers with a certain frequency. However, if the system behaviour changes old timers no longer maintain the correct junk level. Therefore, we observe the junk level and flush accordingly. The idea is as follows. Each peer estimates the number of replicas $D = \sum_o p_o$, p_o being the density of object o, that the peer is supposed to store. Online maintainers know the density of objects they maintain, so this sum can be periodically calculated using a sum over the maintainers. Furthermore, the peer keeps track of the number of replicas it actually stores. With this, it can estimate the percentage of junk it has. A naïve approach would now evict junk once the junk level exceeds some threshold. This is nearly the approach we take, but there is a subtle condition for correctness. By reloading replicas after flushing, we obtain a replica of object o with probability p. To keep the density unchanged, we must also have had a replica with probability p before flushing. This is only true if peer v's decision to flush is independent of the replicas it stores (*):

$$\begin{aligned}\mathbf{P}(o \text{ loses replica} \mid v \text{ flushes}) &= \mathbf{P}(v \text{ stores } o \mid v \text{ flushes}) \\ &\overset{*}{=} \mathbf{P}(v \text{ stores } o) = p\end{aligned}$$

Simply flushing when the threshold is reached is unsafe because peers with the most replicas flush first. So when v flushes, it is more likely to be storing a replica of o and independence is lost. For this reason we use a two-bucket approach. Objects are randomly assigned either type #1 or #2 (e.g., using the last bit of a hash of the identifier). When a peer receives a replica of a type #1 (#2) object it stores it in bucket #1 (#2). When it has to delete a replica it removes it from the according bucket. This division makes it safe to use the traffic flow through bucket #1 in determining when to flush bucket #2 and vice versa. That is, when one bucket reaches the threshold, we flush the other bucket.

The threshold equation is quite simple to derive. We want to bound the junk j, j being the replicas that are not valid, such that the goodness factor g is met. We expect each bucket to have half of a peer's replicas. Let r be the number of replicas in one bucket. The bucket contains the expected $D/2$ valid replicas plus the junk j, i.e., $r = D/2 + j$. The goodness factor g requires that $g * r$, i.e., $g(D/2 + j)$ replicas are valid on average. As the bucket has $D/2$ replicas, we have $D/2 = g(D/2 + j)$, which can be rewritten as $j = D/2(1/g - 1)$. We set the threshold when twice the desired junk is reached, so that the average will be correct. Thus, we flush when $r = D/2 + 2j = D/g - D/2$.

We also consider that the measurement D of the number of replicas a peer is supposed to store arrives with a time lag. It reports the value which was correct when the calculation started. In order to take this into account, we compare it to equally old junk information.

5.3 Cost Analysis

Flushing is surprisingly cheap. Consider counting every object transfer, which seems a reasonable metric. Any system which preserves the replication degree, must create new replicas when peers join the system, and thus transfer at least as many objects as ours. The same argument applies when the density is increased. However, flushing and reloading adds additional transfers to recover flushed valid replicas.

To measure this cost, consider a system where only necessary transfers occur. Initially, peer v pulls an average $D/2$ replicas into bucket #1. Thereafter, v receives x replicas from maintainers performing a push. Unfortunately, v now decides to flush, wasting transfers. For simplicity, assume v again pulls $D/2$ replicas and receives x pushes before its next flush. Before the Fth flush, peer v has performed $F(D/2 + x)$ transfers, when only $D/2 + xF$ were needed. Letting F grow, the percentage of wasted transfers is,

$$\frac{F(D/2+x)}{D/2+xF} - 1 = \frac{D(F-1)}{D+2xF} \approx \frac{D}{2x}, \text{ amortized for large } F$$

To make this equation useful, we need to find x. Suppose that for every object created, another object is on average deleted. This is the case for a network in equilibrium. Peer v saw x pushes, so it should also have seen x deletes. However, if c is the percentage of maintainers which crash (and thus don't delete their replicas properly), peer v's bucket has $j = cx$ junk replicas when it flushes. When v flushes, $j = D(1/g - 1)$. Solve for x to find the overhead is,

$$\text{Percentage of wasted transfers} \approx \frac{D}{2x} = \frac{c}{2/g - 2}$$

For example, consider $g = 0.8$, requiring 80% of replicas to be valid. With no crashes ($c = 0$), there is no overhead. With 10% crashes, the overhead is 20% on the longest lived peers. The reality will be even better, because most peers have a much smaller F than infinity.

6. EXTENSIONS

This section briefly discusses some extensions to the base algorithm that we consider both useful and easy to add.

6.1 Update in Place

Given that each valid object has a single live maintainer, updates can be easily controlled by the maintainer. Deletion is considered a special case of an update. Although in most applications, the maintainer will be the only node updating its objects, others can be allowed to update by simply sending their update requests to the maintainer. By tagging each object with its maintainer's address, the maintainer of an object can be found easily.

As maintainers keep a list storing a superset of the peers with their replicas, we can take advantage of this list for update management. Whenever an update occurs, the maintainer sends the update (or the changed delta) to all peers on the list. In principle, this updates all replicas in the system. However, recall that maintainers ping peers to see if they are still alive, and if they can't be reached, delete them from the list. If a transient network failure occurs, then the maintainer might incorrectly remove a peer from the list and later fail to update it. This leaves temporarily inconsistent replicas in the network. However, all peers eventually either leave the system or flush. Thus, inconsistent replicas will eventually be removed, guaranteeing eventual consistency.

6.2 Heterogeneity

Usually not all peers have the same capacity. Thus, peers should have the possibility to only provide as much service as their capacities allow. In some unstructured systems [23], peers can control their relative load. To support this, every peer v picks a capacity ℓ_v, such that if one peer has twice the capacity of another, it can store approximately twice as many replicas (and serve requests on these replicas).

To be compatible with this approach, our replica maintenance algorithms require four changes. First, the sum over all capacities, $L = \sum_{v \in V} \ell_v$, must be computed. The distributed sum-calculation algorithm can be used for that purpose. A peer v's relative capacity is thus $h = n\ell_v/L$. Second, the push algorithm has to be adjusted. When all peers have the same capacity, the push algorithm should choose each peer with the same probability $1/n$. In a heterogeneous setting, in contrast, it should choose a peer v with relative capacity h with a probability of h/n. Similarly, the pull algorithm has to be adjusted. A joining peer must pull replicas with probability ph instead of p. To effect this, we have to change $f(maxp)$ to $f(h * maxp)$. Finally, the flush threshold must substitute rh/g for r/g.

6.3 Client Maintainers

In hierarchical super-node networks, some participants are clients behind a NAT; they can establish outbound connections, but not receive inbound connections. This means that if they need to talk to some peer in the system they can establish a connection. However, other nodes cannot directly connect to them. To use the system, a client connects to some peer which then operates as a proxy or super-peer, executing queries for the client. This connection remains until either the peer or the client leave the system.

Clients should maintain their objects themselves, as they might outlive their proxy. We can support client maintainers in such a super-node network without any adjustments to our algorithms. We assume the underlying network already supports a client-initiated replica push. To preserve those replicas, joining peers must be able to pull replicas from client maintainers. According to Figure 1 new peers send messages to random maintainers, i.e., they must reach also the clients that are maintainers. These random maintainers are found by the sampling algorithm of the underlying network which are typically based on random walks. As clients are connected to the network via their proxies these random walks can already traverse these client-peer edges. Thus, clients can be found without the need for a direct connection to them. Thereafter, a client maintainer receiving a pull request must send its object replicas to the joining peer. Although this is a direct connection, it is not a problem as the client maintainer initiates the connection to the peer.

As peers take no action on leave, we again avoid the need to connect directly to maintainers. When maintainers leave, they can send delete requests to peers as these are outgoing connections. Similarly, they can initiate update requests. Thus, clients can perform all maintainer tasks.

7. SIMULATION

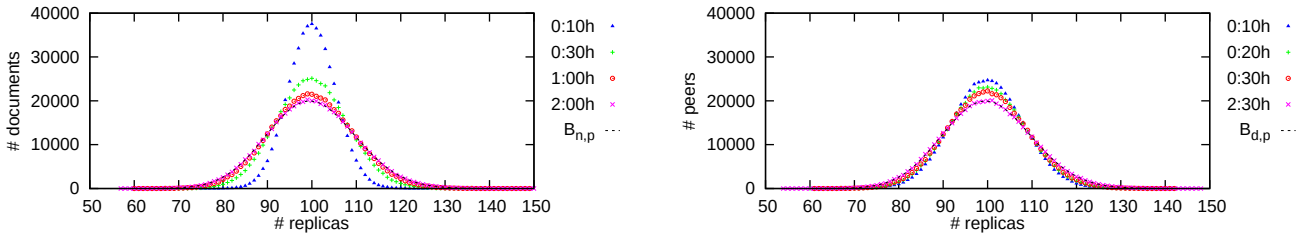
To validate the combined algorithms, we implemented a simulator which manages replicas but abstracts away the underlying network. By assumption in Section 2, the system provided a push algorithm for placing replicas. We implement pushes by storing replicas on random peers.

Except when the network changes size, the simulation is a renewal process. When a node's lifetime expires, it leaves and is immediately replaced by a new node. Simulated nodes have the lifetime distribution measured in Gnutella by Saroiu et al. [19] and fitted in [23], with a median lifetime of 60 minutes. This distribution is extremely heavy-tailed; some nodes are very long-lived, but most are short-lived.

The simulator distinguishes between peers and maintainers. Conceptually, all maintainers are clients which do not store any replicas. This allows us to separate the effects of maintainer churn and peer churn. In most experiments, there are half a million peers and half a million maintainers, reflecting the enormous size of real-world P2P networks. Unless otherwise stated, each maintainer provides one object with density $p = 0.02\%$.

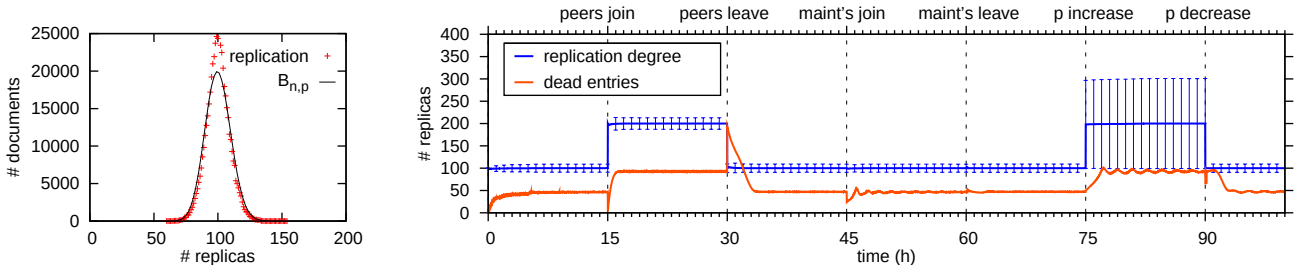
7.1 Convergence

In the first experiment (Figure 2(a)) we confirm that the replication degree distribution converges to the binomial distribution. Maintainers push out their initial replicas and then



(a) Object replication (b) Peer load

Figure 2: Convergence under theoretical conditions



(a) Realistic convergence (b) Under extreme conditions

Figure 3: Replication in steady-state and under massive peer, maintainer, and density changes

stay online forever. As peers flush replicas and join and leave the system, replicas are discarded and pulled. As proven in the Appendix, this should cause the number of replicas per object to converge to the binomial distribution. After ten minutes, the replication has converged completely.

The second experiment (Figure 2(b)) changes the perspective to the peer's point of view. In order to measure valid replica load only (without junk), maintainers do not crash in this test but always exit cleanly. Furthermore, peers neither leave nor flush, because this would reset their load. Upon joining the network, a peer pulls $f(P)$ replicas. Under these restrictions, the load per peer quickly converges towards the binomial distribution with a mean of 100 replicas.

While the last two simulations validate our analysis, it is interesting to test somewhat more realistic scenarios. Aside from making both maintainers and peers subject to churn, 50% of maintainer leaves are crashes where replicas are not deleted. Over time, junk replicas build up at peers and need to be flushed. We set the goodness factor to 80%, allowing 20% of replicas (25 per peer) to be invalid. Instead of one object per maintainer, we assign the half million objects to a reduced total of 125000 maintainers, uniformly at random. Thus, maintainers are responsible for 4 objects on average, but some have more and others less.

The system relies on several collectively computed sums. They are used in the pull and push algorithms and the flush threshold. Imprecise values could affect the correctness of the system. Hence we add a simple simulation of the gossip algorithm [13]. Every 5 minutes the summed values are updated, not with the current value, but with the value it had 5 minutes ago. This reflects the time that the gossip algorithm needs to converge. We will see from the results that this does not affect the correctness of our algorithm.

First, compare the replication degree in Figure 3(a) to the previous artificial settings in Figure 2(a). The more realistic distribution is narrower than the binomial. This is due to maintainers leaving and rejoining the network, which resets their replica counts with the push algorithm. Thus, the real distribution never completely converges away from the initial state, and more objects are close to 100 replicas than in the binomial distribution.

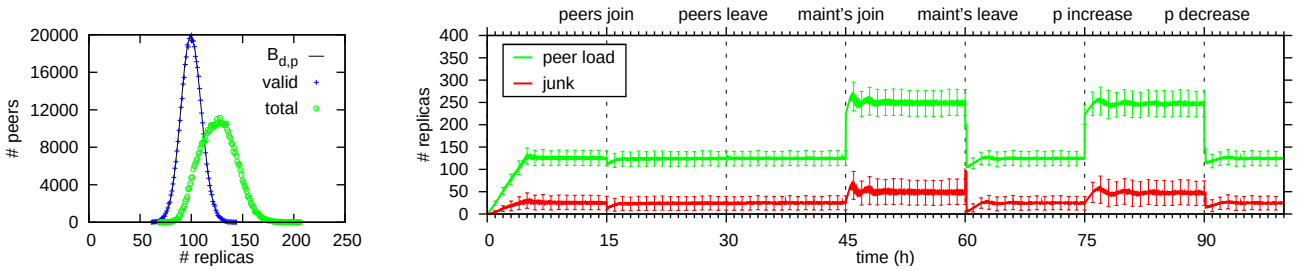
If only considering the valid replicas stored, the load (Figure 4(a)) converges completely to the predicted binomial distribution. Taking junk into account, the actual load of peers is 25% higher ($1/g = 1.25$) and more spread out. The vast majority of peers carry junk of no more than 50%, showing the junk control algorithm from Section 5 effective.

7.2 Effect of large-scale Events

To test system performance under extreme conditions, we make some unrealistic but challenging changes to the environment. These include sudden changes in peer count, object count, and object densities. Peer capacities have the heterogeneous distribution from [23].

Figure 3(b)) shows the average replicas per object and the average dead entries in a maintainer's object replica list; dead entries correspond to peers that no longer store a replica or have left. Figure 4(b) shows the total replicas a peer stores (its load) and the junk replicas per peer.

The network starts with 250000 peers. Over the first 5 hours, 62500 maintainers carrying 250000 objects join. Each object has a replication degree of 100, and the load on peers



(a) Realistic convergence

(b) Under extreme conditions

Figure 4: Peer load in steady state and under massive peer, maintainer, and density changes

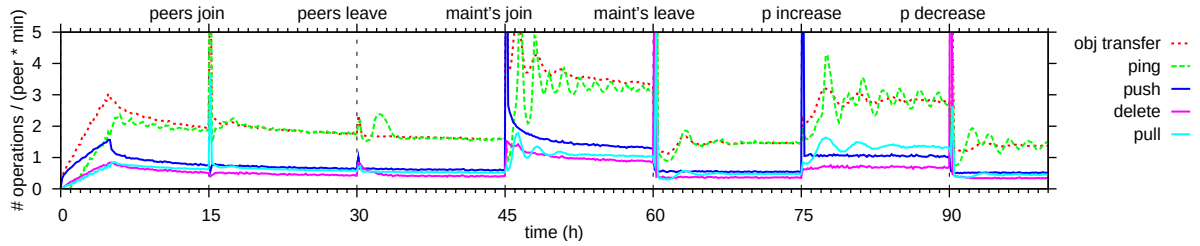


Figure 5: Network operations under extreme conditions

grows with the number of objects to a final value of 100 replicas. Maintainer crashes cause junk to begin accumulating. There is an overshoot in the junk as the peers synchronously approach the flush threshold. After the first round of flushes the synchronization is dissolved.

After 15 hours another 250000 peers are added to the system. The average load initially drops because new peers do not yet carry any junk. Due to the pull on join approach, replication degree immediately adjusts to the new situation by doubling. The doubled replication pushes the replica list of many maintainers past their ping threshold, leading to an initial dip in the number of dead entries after cleaning up.

After another 15 hours, the new peers leave the network permanently. The load of the remaining peers is unaffected. Replication of objects drops back to 100 automatically. The peers leave behind a large number of dead entries, which are cleaned up by the maintainers over time.

At 45 hours 62500 additional maintainers join the network, doubling the number of objects. Their initial push immediately sets the objects' replication to the correct value. The flushing of peers gets synchronized again because they increased their threshold simultaneously. As in the initial overshoot this is cleared after a round of flushing.

The new maintainers depart 15 hours later. Replication degree of the remaining objects is unaffected. The peer load drops when leaving maintainers delete their replicas, but the junk initially increases as half of the maintainers crash instead of leaving properly. This triggers the flush threshold of all peers and removes all junk from the system. Afterwards the junk builds up again, including some barely visible waviness from the synchronization.

Such an eager cleanup after large crashes might be undesirable in a real implementation. It would cause a traffic spike immediately after catastrophic network failures and bring the system even closer to collapse. There are simple solutions against this problem. When peers detect a large drop in the density sum (obtained through the summation protocol), they can defer their flushing by random delay. This would smooth out the traffic and result in a lazy junk adjustment similar to when replica population increases. For a clearer understanding of the algorithm's behaviour we have not included this enhancement in the simulation.

After 75 hours we try changing object density on-the-fly. Half of all objects are selected randomly and their density is tripled. The maintainers push out new replicas to provide the desired replication degree. As the network now has heterogeneous densities, the variance of replication increases. In fact, there are two binomials, one for the large density and one for the small. At 90 hours, the densities are reset. The behaviour is similar to the maintainer population changes.

Figure 5 shows the network operations used during the test. For readability the values are averaged over ten minutes. The plot includes the number of object transfers, peers pinged by maintainers, replicas pushed and deleted by maintainers, and maintainers pulled by peers. Under normal conditions, a peer sees approximately 5 operations per minute. In a real system the object transfers ($\sim$40% of all operations) would probably dominate the bandwidth requirements. Even the extreme events only push the traffic to 25-45 operations per minute, still very reasonable.

Ping frequency starts to oscillate in Figure 5. This happens when many maintainers are added with an empty dead

entry list or change their ping threshold due to simultaneous density changes. Their ping thresholds are subsequently reached at more-or-less the same time. The correlation will slowly disappear and seems unlikely to occur in practise.

In closing, our simulations validate our theoretical analysis and show that the algorithm precisely meets its guarantees, as well as being resilient to extreme network changes.

7.3 Bandwidth Cost Example

The actual bandwidth costs depend heavily on implementation details and application workloads. However, we can sketch an example bandwidth cost based on our simulation results. The message plot shows that in our scenario peers see on average less than 2 object transfers and 2 pings per minute. Assume an object is a directory list of files. A list with 100 file ids 20 Bytes each would be 2KB total. Subscriptions in a pub/sub system would probably be smaller. A ping message is very small, probably less than 100 Bytes including overlay and IP headers. The combined traffic from 2 transfers and 2 pings adds up to 4.2KB/min. The other message types are even smaller and less frequent.

8. RELATED WORK

We are not aware of any work that considers maintainer-based replication, which assures that all replicas are removed from the system when the object maintainer leaves. The closest approach is described in [17], where the authors propose a replica placement protocol for unstructured networks which borrows matching techniques from DHTs. The maintainer places object replicas in local minima, i.e., on a peer v whose identifier is the closest to the object identifier in v's neighbourhood. This allows search using random walks to locate local minima. The maintainer adjusts the number of replicas according to the average search length. Controlling junk, updating objects, and heterogeneity are not addressed.

Ferreira et al. [10] address the problem of maintaining k replicas in an unstructured system from a different direction. They assume that at least k replicas exist (e.g., through autonomous replication), and develop coordination algorithms to reduce the number to the desired k. As we assign replicas to peers, we only need to eliminate objects for which the maintainer has left the system.

In [5, 15] the authors determine that for unstructured networks using search techniques such as flooding or random walks, the average search size is minimized if the number of replicas for an object is chosen proportional to the square-root of its request rate. The authors then approximate this replication rate by placing replicas along the reverse path of successful random walks. Replicas are discarded by replacement strategies, but no replica update takes place. Searches will also cost more than if replicas were randomly distributed. Our approach creates a uniform distribution of replicas and includes object maintenance. Furthermore, we support any replication degree, not only the square-root of request rate, which is only optimal for first-hit semantics.

In [7], Datta et al. discuss a push/pull update propagation scheme for P2P systems. An update is pushed via constrained flooding that only propagates to nodes with replicas and avoids redundant messages. The pull phase allows nodes to get the latest version of an object after restart. This work does not consider preserving the required number of replicas against churn. We could use their epidemic push propagation instead of our update propagation scheme.

Some systems, such as [11, 14], create and delete replicas dynamically when demand changes. Decisions to create or delete replicas are often made locally. This approach is not appropriate for systems that require a minimum amount of replicas in the system, such as BubbleStorm [23] and [9].

Path replication has also been proposed for structured systems [6], where query results are cached on the reverse path. Akbarinia et al. [1] provide DHT-based replication using k hash functions to create k replicas for each data item. They present algorithms to keep replicas consistent despite churn.

[12] analyze when and how many replicas should be kept within a relatively close community in order to avoid access to remote content and distribute the load across local nodes.

Large-scale file systems maintain replicas for availability [3, 21]. Solutions can be reactive and create new replicas once too many replicas were lost. Or they could be proactive, creating new replicas whenever the system is idle in order to compensate for lost replicas. Most solutions are heavily based on probing to check replica availability – a mechanism that we try to avoid as much as possible. Furthermore, all these approaches only look at collective replica maintenance. Thus, in most of these solutions rejoining peers keep the replicas they acquired during their last memberships as they usually all remain valid.

9. CONCLUSION

We have presented a solution for maintaining replicas in unstructured peer-to-peer systems where object lifetimes are tied to maintainer lifetimes. From a maintainer's point of view, it is easy to change replication degree, update replicas, and operate behind a firewall. From a peer's point of view, the solution bounds junk served, distributes load fairly, and permits crashes. From a system designer's point of view, the algorithms are simple to implement and only require building blocks available in existing literature.

We have proven that replication degree converges to the binomial distribution over time and that storage load is also fairly shared/binomial. Simulation results validate that our implementation is correct and meets the required bounds on junk. Finally, large-scale peer and maintainer join and leave events do not appreciably affect correctness.

10. REFERENCES

[1] R. Akbarinia, E. Pacitti, and P. Valduriez. Data Currency in Replicated DHTs. In *SIGMOD*, 2007.

[2] A. Awan, R. A. Ferreira, S. Jagannathan, and A. Grama. Distributed Uniform Sampling in

Unstructured Peer-to-Peer Networks. In *HICSS*, 2006.
[3] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. TotalRecall: Systems Support for Automated Availability Management. In *NSDI*, 2004.
[4] P. Billingsley. *Convergence of Probability Measures*. Wiley-Interscience, second edition, July 1999.
[5] E. Cohen and S. Shenker. Replication Strategies in Unstructured Peer-to-Peer Networks. In *SIGCOMM*, 2002.
[6] F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *SOSP*, 2001.
[7] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *ICDCS*, 2003.
[8] S. Datta and H. Kargupta. Uniform Data Sampling from a Peer-to-Peer Network. In *ICDCS*, 2007.
[9] R. A. Ferreira, M. K. Ramanathan, A. Awan, A. Grama, and S. Jagannathan. Search with Probabilistic Guarantees in Unstructured Peer-to-Peer Networks. In *IEEE P2P*, 2005.
[10] R. A. Ferreira, M. K. Ramanathan, A. Grama, and S. Jagannathan. Randomized protocols for duplicate elimination in peer-to-peer storage systems. *IEEE TPDS*, 18(5), 2007.
[11] V. Gopalakrishnan, B. D. Silaghi, B. Bhattacharjee, and P. J. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *ICDCS*, 2004.
[12] J. Kangasharju, K. W. Ross, and D. A. Turner. Optimizing File Availability in Peer-to-Peer Content Distribution. In *INFOCOM*, 2007.
[13] D. Kempe, A. Dobra, and J. Gehrke. Gossip-Based Computation of Aggregate Information. In *FOCS*, 2003.
[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. E. Czerwinski, P. R. Eaton, D. Geels, R. Gummadi, S. C. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Y. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.
[15] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and Replication in Unstructured Peer-to-Peer Networks. In *ICS*, 2002.
[16] L. Massoulié, E. L. Merrer, A.-M. Kermarrec, and A. Ganesh. Peer Counting and Sampling in Overlay Networks: Random Walk Methods. In *PODC*, 2006.
[17] R. Morselli, B. Bhattacharjee, A. Srinivasan, and M. A. Marsh. Efficient Lookup on Unstructured Topologies. In *PODC*, 2005.
[18] D. Mosk-Aoyama and D. Shah. Computing Separable Functions via Gossip. In *PODC*, 2006.
[19] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Multimedia Comp. and Networking*, 2002.
[20] N. Sarshar, P. O. Boykin, and V. P. Roychowdhury. Percolation Search in Power Law Networks: Making Unstructured Peer-to-Peer Networks Scalable. In *IEEE P2P*, 2004.
[21] E. Sit, A. Haeberlen, F. Dabek, B. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz. Proactive Replication for Data Durability. In *IPTPS*, 2006.
[22] M. Steiner, T. En-Najjary, and E. W. Biersack. A Global View of Kad. In *IMC '07*, 2007.
[23] W. W. Terpstra, J. Kangasharju, C. Leng, and A. P. Buchmann. BubbleStorm: Resilient, Probabilistic, and Exhaustive Peer-to-Peer Search. In *SIGCOMM*, 2007.

APPENDIX

This section proves that by taking the approach of Section 4 any sequence of peer join and crash/leave events causes the replication distribution to converge to the binomial, $B_{n,p}$. Symmetrically, if all objects share p, then any sequence of object creation and deletion events with d undeleted objects converges peer load distribution to $B_{d,p}$. We only prove the first claim as the second is analogous.

Let $n_t \geq 0$ be the network size for time $t \in \mathbb{Z}$. For simplicity, we require that every unit of time corresponds to exactly one event: join or leave. A joining peer causes $n_{t+1} = n_t + 1$, while a leaving peer causes $n_{t+1} = n_t - 1$. The sequence of joins and leaves is chosen by an adversary.

For an arbitrary object, consider the evolution of the number of replicas R_t over time. R_t is a random variable taking values in $[0, n_t]$. If $n_{t+1} = n_t + 1$, then a peer joined the system, increasing R_t by 1 with probability p;

$$\begin{aligned}\mathbf{P}(R_{t+1}{=}i) &= \mathbf{P}(R_{t+1}{=}i \mid R_t{=}i{-}1)\mathbf{P}(R_t{=}i{-}1) \\ &\quad + \mathbf{P}(R_{t+1}{=}i \mid R_t{=}i)\mathbf{P}(R_t{=}i) \\ &= p\mathbf{P}(R_t{=}i{-}1) + (1-p)\mathbf{P}(R_t{=}i)\end{aligned}$$

Let $r_t = (r_t(0), r_t(1), \ldots, r_t(n_t))$ be R_t's probability vector, where $r_t(i) = \mathbf{P}(R_t = i)$ for $i \in [0, n_t]$. We use the notation $r_t(i)$ to emphasize that r_t plays a dual role as both a vector and a function of i. Set J_n to the $[0, n] \times [0, n+1]$ join transition matrix where $r_{t+1} = r_t J_{n_t}$,

$$J_n = \begin{bmatrix} 1{-}p & p & 0 & \ldots & 0 \\ 0 & 1{-}p & p & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \ldots & 0 & 1{-}p & p \end{bmatrix}$$

When one peer out of n_t leaves (or crashes), it destroys a replica with probability R_t/n_t. Like J_n, define L_n as the $[0, n] \times [0, n-1]$ transition matrix mapping $r_{t+1} = r_t L_{n_t}$.

$$L_n = \frac{1}{n}\begin{bmatrix} n & 0 & \ldots & 0 \\ 1 & n{-}1 & \ddots & \vdots \\ 0 & 2 & \ddots & 0 \\ \vdots & \ddots & \ddots & 1 \\ 0 & \ldots & 0 & n \end{bmatrix}$$

Using these two matrices we can finally formulate the transition matrix $r_{t+1} = r_t T_t$ for all t,

$$T_t := \begin{cases} J_{n_t} & \text{if } n_{t+1} = n_t + 1 \\ L_{n_t} & \text{if } n_{t+1} = n_t - 1 \end{cases}$$

THEOREM 1. *If there exists some upper-bound n_* on the network size, such that $n_* > n_x$ for all time x, then, for any initial replication distribution r and fixed t, the replication distribution r_t converges to $B_{n_t,p}$ as the mixing time grows;*

$$r_t = r \prod_{x=t_0}^{t-1} T_x \to B_{n_t,p} \text{ as } (t - t_0) \to \infty$$

where $B_{n,p} = (B_{n,p}(0), B_{n,p}(1), \ldots, B_{n,p}(n))$ and

$$B_{n,p}(i) = \binom{n}{i} p^i (1-p)^{n-i}$$

We first prove three lemmas needed for this result. The first lemma explains why the binomial is the limit.

LEMMA 1. *The binomial distribution is an invariant flow;*

$$B_{n_{t+1},p} = B_{n_t,p} T_t$$

PROOF. The transition matrix is a join J_n or leave L_n:

$$\begin{aligned}
(B_{n,p}J_n)(0) &= (1-p)B_{n,p}(0) = (1-p)(1-p)^n \\
&= B_{n+1,p}(0) \\
(B_{n,p}J_n)(i) &= pB_{n,p}(i-1) + (1-p)B_{n,p}(i) \\
&= p^i(1-p)^{n+1-i}\left[\binom{n}{i-1} + \binom{n}{i}\right] \\
&= B_{n+1,p}(i) \\
(B_{n,p}L_n)(i) &= \frac{n-i}{n}B_{n,p}(i) + \frac{i+1}{n}B_{n,p}(i+1) \\
&= B_{n-1,p}(i)\left[(1-p)+p\right]
\end{aligned}$$

□

The next two lemmas show the transition matrix forces any two states R_t, S_t towards each other. We will measure the distance between $\mathbf{E}(f(R_{t+1}) \mid R_t)$ and $\mathbf{E}(f(S_{t+1}) \mid S_t)$ for arbitrary function f using the Lipschitz norm,

$$||f||_\ell = \max_{i>j} \frac{|f(i) - f(j)|}{i - j}$$

LEMMA 2. *For all time t and any $f : \mathbb{Z} \to \mathbb{R}$,*

$$||T_t f||_\ell \le ||f||_\ell \cdot \begin{cases} 1 & \text{if } n_{t+1} = n_t + 1 \\ 1 - \frac{1}{n_t} & \text{if } n_{t+1} = n_t - 1 \end{cases}$$

PROOF. There are again two cases. Interpret function $f(i)$ as a column vector on $[0, n_t]$. Then, for all $i > j$,

$$\begin{aligned}
&J_n f(i) = (1-p)f(i) + pf(i+1) \\
&|J_n f(i) - J_n f(j)| \\
&\quad\le (1-p)|f(i) - f(j)| + p|f(i+1) - f(j+1)| \\
&\quad\le (1-p)||f||_\ell(i-j) + p||f||_\ell(i-j) \\
&\quad= ||f||_\ell(i-j)
\end{aligned}$$

By carefully regrouping terms,

$$\begin{aligned}
&L_n f(i) = \tfrac{i}{n} f(i-1) + \tfrac{n-i}{n} f(i) \\
&|L_n f(i) - L_n f(j)| \\
&\quad\le \tfrac{j}{n}|f(i-1) - f(j-1)| \\
&\qquad + \tfrac{i-j}{n}|f(i-1) - f(j)| + \tfrac{n-i}{n}|f(i) - f(j)| \\
&\quad\le ||f||_\ell(i-j)\left[\tfrac{j}{n} + \tfrac{i-1-j}{n} + \tfrac{n-i}{n}\right] \\
&\quad= ||f||_\ell(i-j)\left(1 - \tfrac{1}{n}\right)
\end{aligned}$$

□

LEMMA 3. *If there exists some n_* such that $n_* > n_x$ for all time x, then for $f : \mathbb{Z} \to \mathbb{R}$ with $||f||_\ell < \infty$ and fixed t,*

$$\left|\left|\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell \to 0 \text{ as } (t - t_0) \to \infty$$

PROOF. Reformulating the bound on network size,

$$n_x < n_* \implies \left(1 - \frac{1}{n_x}\right) < \left(1 - \frac{1}{n_*}\right)$$

There are at most n_t more joins than leaves, so there are infinitely many leaves. Repeatedly apply Lemma 2 to find,

$$\left|\left|\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell < ||f||_\ell \left(1 - \frac{1}{n_*}\right)^{[(t-t_0)-n_t]/2} \to 0$$

□

PROOF OF THEOREM. R_{t_0} has initial probability distribution r. From the definition of expectation, $\mathbf{E}(f(R_{t_0})) = rf$ where f is a function interpreted as a column vector. Similarly, $\mathbf{E}(f(R_t)) = r\prod_{x=t_0}^{t-1} T_x f$. So, the Lipschitz term from Lemma 3 can be reformulated for all $i > j$ as,

$$\frac{|\mathbf{E}(f(R_t) \mid R_{t_0} = i) - \mathbf{E}(f(S_t) \mid S_{t_0} = j)|}{i - j} \le \left|\left|\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell$$

Condition on events $E_i = (R_{t_0} = i)$ and $F_j = (S_{t_0} = j)$,

$$\begin{aligned}
&|\mathbf{E}(f(R_t)) - \mathbf{E}(f(S_t))| \\
&\quad= |\textstyle\sum_i \mathbf{P}(E_i)\mathbf{E}(f(R_t)|E_i) - \sum_j \mathbf{P}(F_j)\mathbf{E}(f(S_t)|F_j)| \\
&\quad= |\textstyle\sum_{i,j} \mathbf{P}(E_i)\mathbf{P}(F_j)(\mathbf{E}(f(R_t)|E_i) - \mathbf{E}(f(S_t)|F_j))| \\
&\quad\le \textstyle\sum_{i,j} \mathbf{P}(E_i)\mathbf{P}(F_j)|\mathbf{E}(f(R_t)|E_i) - \mathbf{E}(f(S_t)|F_j)| \\
&\quad\le \left|\left|\textstyle\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell \textstyle\sum_{i,j} \mathbf{P}(E_i)\mathbf{P}(F_j)|i-j| \\
&\quad\le \left|\left|\textstyle\prod_{x=t_0}^{t-1} T_x f\right|\right|_\ell n_* \to 0
\end{aligned}$$

The Portmanteau theorem (Theorem 2.1 in [4]) proves convergence in distribution given this limit for the expectation. Alternately, for arbitrary i, set $f(k) = 1$ for $k = i$ and 0 otherwise. Now $\mathbf{E}(f(R_t)) = \mathbf{P}(R_t = i) = r_t(i)$. Give S_{t_0} binomial distribution. By Lemma 1, S_t also has binomial distribution, so $\mathbf{E}(f(S_t)) = B_{n_t,p} f = B_{n_t,p}(i)$.

$$|r_t(i) - B_{n_t,p}(i)| = |\mathbf{E}(f(R_t)) - \mathbf{E}(f(S_t))| \to 0$$

□