Replication Strategies for Reliable Decentralised Storage

Matthew Leslie[1,2], Jim Davies[1], and Todd Huffman[2]

[1]Oxford University Computing Laboratory and
[2]Oxford University Department of Physics

mleslie@fnal.gov, jdavies@comlab.ox.ac.uk,
t.huffman1@physics.ox.ac.uk

Abstract

Distributed hash tables (DHTs) can be used as the basis of a resilient lookup service in unstable environments: local routing tables are updated to reflected changes in the network; efficient routing can be maintained in the face of participant node failures. This fault-tolerance is an important aspect of modern, decentralised data storage solutions. In architectures that employ DHTs, the choice of algorithm for data replication and maintenance can have a significant impact upon performance and reliability.

This paper presents a comparative analysis of replication algorithms for architectures based upon a specific design of DHT. It presents also a novel maintenance algorithm for dynamic replica placement, and considers the reliability of the resulting designs at the system level. The performance of the algorithms is examined using simulation techniques; significant differences are identified in terms of communication costs and latency.

1. Introduction

Distributed Hash Tables (DHTs) can be used to provide scalable, fault-tolerant key-based routing. Each lookup can be routed reliably to an appropriate node, which will return the required data; in most systems, the worst-case time complexity for node location is logarithmic in the number of nodes.

Several systems employing DHTs have been developed, notably: PAST, Tapestry, CAN, Kademlia, and Chord [22, 26, 20, 12, 15]. The combination of resilience and efficiency has led to the adoption of these systems in decentralised storage solutions: examples include CFS, OceanStore, Ivy, and Glacier [6, 10, 17, 9].

All of these systems use replication to provide reliability, but they employ a variety of different strategies for placement and maintenance. In this paper, we will examine several of these strategies, and show that the choice of strategy can have a significant impact upon reliability and performance.

In Sections 2 and 3, we describe two replication algorithms—*DHash* [3] and dynamic replication [23]—in the context of the Chord DHT [15]. We examine the shortcomings of dynamic replication, identify a solution, and propose a variety of placement strategies.

In Section 5, we compare the reliability of different strategies at the level of a complete system: failure is synonymous with the loss of every copy of any item of data. In Sections 7 and 8, we use simulation techniques to compare the impact of replication strategy upon fetch latency and bandwidth usage.

2. DHash replication

In Chord, nodes and data items are assigned keys between zero and some maximum K, corresponding to positions on a ring. A node *owns*, or is responsible for, data that it is the first clockwise successor of. Each node maintains knowledge of its immediate clockwise neighbours, called its *successors*, and several other nodes at fractional distances around the ring from it, called its *fingers*.

The DHash approach [3] combines the placement strategy proposed for Chord with a maintenance algorithm suggested by Cates [4]; this combination is used by the DHash storage system and the Ivy File System [17].

Replicas of a data item are placed (only) on the r successors of the node responsible for that item's key. To maintain this placement pattern in the face of nodes joining and leaving the system (node *churn*), there are two maintenance

protocols: the *local* and *global* algorithms; these prevent the number of replicas of any object from either dropping too low or rising too high.

Local Maintenance Each node sends a message to its r successors, listing the key range it is responsible for. These nodes then synchronise their databases so that all items in this range are stored on both the root node and its successors. (Methods for database synchronisation, such as Merkle Tree hashing [13], are discussed in [4].)

To repair the overlay, the local algorithm runs twice: in the first pass, the items in the key range of the root node are identified and gathered at the root note; in the second, replicas of these items are distributed to the successor nodes.

Global Maintenance A node periodically checks the keys of the items in its database to see if it stores any item that it is no longer responsible for. To do this, it looks up the owner of each key it stores, and checks the successor list of that owner. If it is within r hops of the node, then it will be one of the first r nodes in the successor list. If its ID is not in this list, the node is no longer responsible for keeping a replica of this item. In this case, the item is offered to the current owner, after which it may safely be deleted.

3. Dynamic Replication

This approach was proposed initially by Waldvogel et al. [23]; a similar method is used by Glacier [9] to place file fragments. The essential feature is the use of an *allocation function* for replica placement: for an item with key d, the replicas are placed at locations determined by $h(m,d)$, where m is the index of that replica. Replicas of each item are placed at locations with indexes between zero and $r-1$, where r is the desired replication factor. The node responsible for $h(0,d)$ is called the *owner* of d, and the nodes with replicas are called the *replica group* for that item.

The use of an allocation function helps alleviate the lookup bottleneck associated with DHash replication. DHash replication requires that all lookups for a popular item are directed to that item's owner in order to discover replica locations. With dynamic replication, the location of replicas is already known, and lookup requests can be directed to any replica location.

There are two potential shortcomings of dynamic replication schemes, concerning the management of communication and (allocation) collisions. We will now examine these, and explain how they may be addressed.

Communication costs The maintenance process at a particular node will require that every node in the replica group for an item is contacted; in the worst case, this group could be different for every item that the node owns. To avoid this, we may use allocation functions that are translations in d, mapping each replica index onto a keyspace the same size as the original node's keyspace.

Replica maintenance performance can be further improved by using functions which exploit the local routing state stored on each node: for example, we might place replicas on the finger or successor nodes of each node.

Allocation Collisions The allocation function may map the same object (into the keyspace of a single node) under two separate replica indexes. Such an *allocation collision* may occur even if replica locations are well spaced, if the number of participating nodes is relatively small. The effect of a collision is to reduce the number of distinct nodes in the replica group, and hence to reduce reliability.

We propose the following solution. The range of allowable replica indexes m should be divided into two parts: the core range, with maximum index R_{MIN}; and the peripheral range, with maximum index R_{MAX}. A maximum index is needed to ensure a known bound upon the number of replicas of an item—an important consideration if the data is to be consistently updated and deleted.

The maintenance algorithm should keep track of replica allocation. When an allocation collision occurs, an additional replica should be placed at an index in the peripheral range. Thus replicas will always be placed at core locations, but may also be present at peripheral locations. Replicas are placed at replica locations with increasing indexes, starting from the lowest index, and placed until either R_{MIN} distinct copies are present, or all R_{MAX} allowable locations are filled.

The choice of value for R_{MAX} is important: too low, and there may be too few distinct replicas; too high, and update (and delete) performance may suffer—the entire replica group must be contacted to ensure that all replicas are fresh. We will examine the problem of setting R_{MAX} in more detail with reference to particular allocation functions in Section 4.

3.1. Replica maintenance algorithms

In describing our replica maintenance algorithm, we will identify (groups of) nodes according to the roles that they play with respect to a single item:

- the *core group* for an item d is the set of replica holders for which $m \leq R_{MIN}$
- the *peripheral group* consists of those replica holders for which $R_{MIN} < m \leq R_{MAX}$

The role of the replica maintenance algorithm is to preserve or restore the following invariants:

1. replicas of an item d can only be retrieved from addresses given by $h(m,d)$ where $1 \leq m \leq R_{MAX}$

2. a replica of an item can always be retrieved rom addresses given by $h(m,d)$ where $1 \leq m \leq R_{MIN}$

3. any peripheral replica with index $(m > R_{MIN})$ exists only if a replica is placed at index $m-1$.

We will now explain the maintenance protocols required to maintain these invariants:

Core Maintenance The owner of a data item calculates and looks up the nodes in the core group. Each core replica holder synchronises its database with the owner (over that part of owner's keyspace that it holds). This will restore the second invariant.

Core maintenance also deals with allocation collisions, by keeping track of which nodes store replicas from which keyranges, and placing additional replicas on peripheral replica holders if a given keyrange is mapped to the same node more than once.

Peripheral Maintenance To maintain the third invariant, any node that stores a replica with index $m > R_{MIN}$ must check that a replica of that item is held also on the replica predecessor, the owner of the location with index $m-1$. If a replica is not present on the previous node, the replica is *orphaned*.

For each peripheral replica a node holds, it must obtain a summary of the items with the previous index on the replica predecessor. Bloom filters [18] can be used to reduce the cost of these exchanges.

These summaries can be used to remove orphaned peripheral replicas from the system (after offering them to their owner); these replicas should not be used to answer fetch requests, but still be stored for at least one maintenance interval—maintenance will often replace the missing replica.

Global Maintenance Each node calculates the replica locations for each item it stores. If it stores any item for which no replica location exists within its ownership space, it offers the item to its owner, then deletes it. This restores the first invariant.

3.2. Data fetch algorithm

To fetch an item stored using this replication strategy, we must decide which replica indexes to request, and in which order. Our proposal is that the data fetch algorithm should start by requesting a replica with a randomly-chosen index between zero and R_{MAX}, and continue picking indexes until a surviving replica is found. If an item is not found during in the initial search of the replica group, the fetch algorithm should back off and retry after the maintenance protocols have repaired the system.

To improve the average fetch time, we propose that if a replica in the peripheral group is found to be empty, all larger replica indexes should be eliminated from consideration until all lower indexes have been searched. In situations where load balancing is not critical, it may prove advantageous to search the core replica locations before trying peripheral locations.

4. Allocation Functions

We will now describe a number of different allocation functions—see Table 1—and explore their impact upon reliability and performance. The result of applying each function to data key *d* and replica index *m* will depend in each case upon the number of nodes in the system *n*, and the maximum key value *k*. The value of *n* may be specified by the user, or estimated at run-time: see [1].

Random placement is not a realistic option. Using a pseudo-random function, seeded by the original key, to determine replica locations would lead to high maintenance costs—due to the wide range of nodes that a small key range could be mapped to—and would make it impossible to exploit local routing information.

4.1. Successor placement

In this approach, replicas are placed at regular intervals following the key of the original item, mimicking the effect of the *DHash* replication algorithm. The intention is that the replicas are stored at keys owned by the successors of the owner of the original key. This mapping is efficient under Chord, as the protocol maintains a local list of each node's successors on that node, so maintenance lookups can often be performed without consulting another node.

The relative proximity of locations with different indexes under this function means a high probability of allocation collisions. As the distribution of randomly placed nodes is known, we can provide a sufficient number of additional locations with high probability: for a keyspace in which *k* is the maximum allowable key, the total keyspace owned by *r* adjacent nodes is normally distributed with:

$$\mu \cong \frac{rk}{n} \; \sigma^2 \cong \frac{rk^2}{n^2}$$

Thus, by standard properties of the normal distribution, we should set $R_{MAX} = R_{MIN} + 1.645\sqrt{R_{MIN}}$ to have a 95% probability of there being R_{MIN} distinct nodes available.

4.2. Predecessor placement

Alternatively, we may place the replicas at regular intervals *preceding* the original item. As queries are routed clockwise around the ring, towards the node responsible, a lookup for a node will usually be routed through one of

Allocation	$h(m,d)$
successor placement	$d + (m \cdot \frac{k}{n}) \mod k$
predecessor placement	$d - (m \cdot \frac{k}{n}) \mod k$
finger placement	$\delta = log_2(\frac{k}{n})$
	$d + 2^{(m+\delta)} \mod k$
block placement	$d - (d \mod \frac{k \cdot R_{MAX}}{n})$
	$+(d \mod \frac{k}{n})$
	$+(m * \frac{k}{n}) \mod k$

Table 1. Allocation Functions

its predecessors. Predecessor placement exploits this fact to reduce fetch latency, allowing replica holders to satisfy a request for a key preemptively, and avoiding further network hops. As with successor placement, the proximity of locations means a higher probability of allocation collisions: R_{MAX} should be set accordingly.

4.3. Finger Placement

The Chord system maintains routing information at locations spaced at fractional distances around the ring, called *finger nodes*. We may take advantage of this fact by placing replicas on these nodes: this has the effect of reducing the lookup cost, and achieving an even distribution of replicas. The relative separation of replica locations reduces the likelihood of allocation collisions, and allows us to choose a lower value for R_{MAX}, improving update performance.

The replica groups for a given node will usually form disjoint sets of nodes. This can be exploited to reduce recovery time following node failure, when we must create a new copy of every item stored by the failed node. We may transfer items *concurrently* from nodes in each of the replica groups that the failed node was a member of.

4.4. Block Placement

We may reduce the number of combinations of nodes whose failure would cause data loss by placing r replica groups onto the same range of keys. The nodes in each *block* store all replicas of items with keys within that block, and no replicas of any other items. This system is similar to the symmetric replication scheme used in DKS [8], in which the same data is placed on each of the nodes in an equivalence class.

This function is discontinuous in d, and the maintenance algorithm must address this when mapping ranges of keys onto other nodes. The distance between replica indexes is the same as with successor and predecessor replication, and the same method may be used for setting R_{MAX}.

In the next section, we will show that this placement policy will reduce the probability of data loss, while offering some of the benefit of successor or predecessor placement, as many nodes will have replicas placed on both successors and predecessors.

5. Reliability

Much of the existing work on the reliability of storage in a distributed hash table has concentrated upon the probability of a given object being lost. For many applications however, a more relevant figure is the probability of any item being lost anywhere in the system. It is this level of *system reliability* that we will investigate here.

5.1. Model

To assess the impact on the reliability of replica placement on the system, we model a Chord ring as a series of n nodes. Each node's data is replicated r times on r different nodes. We consider the system over an arbitrary length of time, during which the nodes fail with uniform independent *node failure probability* p. The system is considered to have failed in the event that node failures result in all replicas of any piece of data being lost.

We will use this model to compute the probability of system failure for various placement functions, for a variety of values of n, r, and p.

5.2. Block Placement

For a system using block placement to fail, all r nodes in some block must fail. As blocks correspond to disjoint sets of nodes, the failure of one block is independent of the failure of any of the others. As each block fails with probability p^r, the probability that at least one of the n/r blocks will fail is given by:

$$Fail(p,r,n) = (1 - (1 - p^r)^{n/r})$$

5.3. Successor Placement

The probability of data loss with successor placement is equivalent to that of obtaining a sequence of r successful outcomes in n Bernoulli trials with probability of success p. This is known as the *Run Problem*, and the general solution $RUN(p,r,n)$ can be given in terms of a generating function GF [24].

$$GF(p,r,s) = \frac{p^r s^r (1-ps)}{1-s+(1-p)p^r s^{r+1}} \equiv \sum_{i=r}^{\infty} c_i^p s^i$$

$$Fail(p,r,n) = RUN(p,r,n) = \sum_{i=r}^{n} c_i^p$$

Figure 1 shows how we may use this function, and that from Section 5.2 to compare the minimum *node reliability p* for

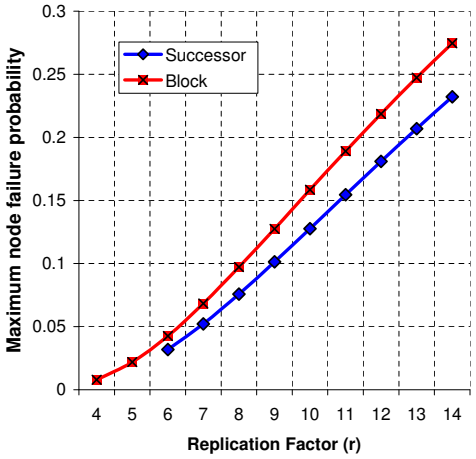

Figure 1. Critical node failure probability for a system failure probability of $< 10^{-6}$. These figures are for a 1000 node system, with varying numbers of replicas.

a 1000-node system with $fail(n, r, p) < 10^{-}6$: values are shown for various replication factors.

A larger system requires more reliable nodes in order to offer the same level of system reliability. The required node reliability increases with the logarithm of the system size, as illustrated in Figure 2.

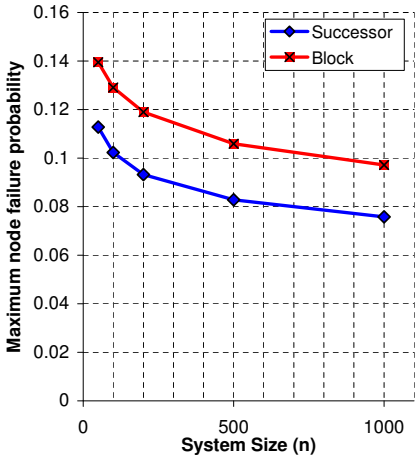

Figure 2. Critical node failure probability for a system failure probability of $< 10^{-6}$. These figures are for a replication factor of 8.

5.4. Finger Placement

We will use Monte Carlo simulation to compare this to other patterns: there is, as yet, no closed form. A model of a 500-node network, in which 250 nodes are marked as failing, was considered. We produced 10^5 sample networks, and used them to estimate the probability of any data loss occurring in the network with varying numbers of replicas and for each allocation function. Figure 3 shows the probability of data loss for finger, block, and successor allocation.

From this plot, it is clear that finger placement is significantly less reliable than other data placement algorithms. Interestingly, in our simulations, finger and random placement produced near identical system failure rates.

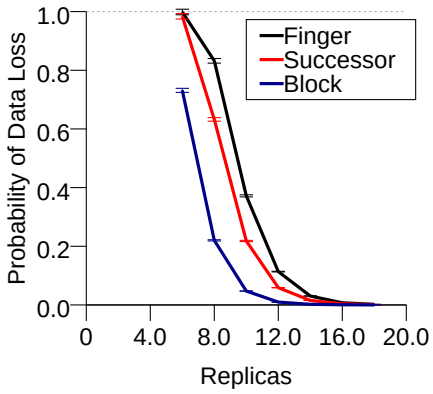

Figure 3. Probability of system failure for three allocation functions in a 500 node system where 50% of nodes fail. Finger and random results overlap. Error bars show 95% confidence intervals.

5.5. Designing a reliable system

The data above allows us to place limits on participant node reliability in order to achieve a given system reliability. In reality, system designers have very little control over the reliability of the nodes they deploy on, and must adjust the replication factor accordingly.

Increasing the replication factor provides an effective method of reaching a desired level of system reliability, though at the cost of increased storage and bandwidth requirements. Another option may be to increasing the frequency of maintenance, thus reducing the period of time during which nodes can fail. However, bandwidth limitations can present a serious barrier to frequent maintenance of large quantities of data [2].

6. Simulation

We now attempt to quantitatively compare the performance and bandwidth usage of these replication algorithms. Due to the difficulty of managing large numbers of physical nodes [7], we chose to test the algorithms through simulation rather than through deployment.

Our simulator is based around the SimPy [16] discrete-event simulation framework, which uses generator functions rather than full threads to achieve scalability. The simulator implements a message level model of a Chord network running each of the replication algorithms described. We model a system that might resemble a data centre built from cheap commodity components. We simulate a network of 200 nodes in which nodes join and leave the system at the same rate. Latency between nodes is assumed to be uniform. Our sample workload includes 50,000 fetches for data originating from randomly chosen nodes.

We choose parameters for the Chord algorithm that allowed routing to be resilient to a high level of churn. Local and Core Maintenance algorithms run two passes at each maintenance interval, and our dynamic fetch algorithm is set to search the core replica group before trying any peripheral replicas.

7. Fetch Latency

The DHash algorithm, and each of the placement patterns for dynamic replication result in differing data fetch latencies. The simulation results in Figure 4 show how the fetch times differed for each of our algorithms for various system sizes. Although all replication algorithms have fetch times logarithmic in system size, there are significant differences between the algorithms.

The predecessor algorithm achieves the shortest fetch times. This is because when a request for an object is routed through a node which holds a replica of that object, it *may* return it's replica of that object instead of passing on the request, if it has spare upload capacity. We call this *preemptive return*. Under predecessor allocation, queries for core replicas are more often routed through peripheral replicas, which return the data preemptively. This happens less often with successor or block allocation and very infrequently with finger allocation.

Our implementation of the DHash fetch algorithm involves returning the successor list of the owner to the requesting node, which then chooses and requests a replica from the successor list. This causes the two hop difference between DHash and dynamic finger replication.

The frequency of replica maintenance had varying effects on the fetch latency of each replication strategy. Under those algorithms where preemptive return is common, the fact that the specific replica being requested has failed

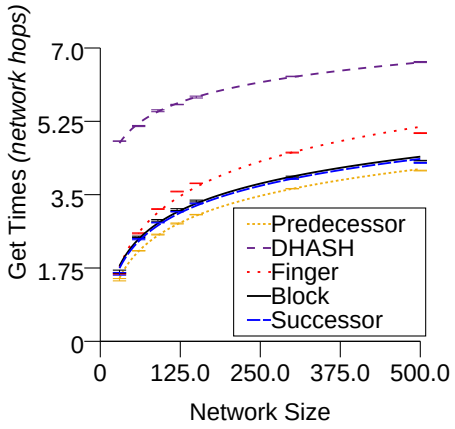

Figure 4. Get times in various system sizes. (Successor and Block overlap).

is often masked by other replica holders, which allow a request for a dead replica to be successful. Thus, Predecessor, Successor and Block placement perform better under high failure rates than DHash or Dynamic Finger Placement.

7.1 Correlated failure

We have so far investigated the fetch latency of these data replication algorithms under a steady state of churn, in which new nodes join at the same rate as other nodes fail. The system can also recover from far higher failure rates, although there is a substantial performance impact.

To assess the impact of correlated failures, we simulate the failure of varying proportions of the nodes in a 500 node network. Fifty thousand fetch requests are then launched, and the average time they take to return, including retries, is recorded. The DHash algorithm is particularly affected under such scenarios. This is because of DHash must be able to route to the owner of a data item in order to locate the replicas of that data item. If the Chord infrastructure is temporarily unable to route to the owner node, that item may not be fetched until the overlay is repaired. When using dynamic maintenance, replicas are stored at well known locations, and an interruption in Chord routing to one of these locations may leave others accessible. Dynamic maintenance is thus more resilient to correlated failure than DHash.

8. Bandwidth Usage

Bandwidth usage is influenced by the rate at which nodes fail. In order to abstract away the timescale with which node failures occur, we present our bandwidth data in terms of the

systems' *half life*, where a half life is the period of time over which half the nodes in the original system have failed.

In Figure 5, we analyse maintenance overhead bandwidth, that is, all maintenance traffic excluding replica data. We show how it varies with maintenance frequency, in terms of the number of repairs per half life. DHash maintenance has lower overhead than the dynamic algorithms. This is largely because of the peripheral maintenance algorithm, which involves bloom filter exchange, and is not necessary under DHash. The dynamic algorithms all have similar overhead, which increases linearly with maintenance frequency. Block allocation has higher maintenance bandwidth because the allocation function is not continuous, and so requires more lookups.

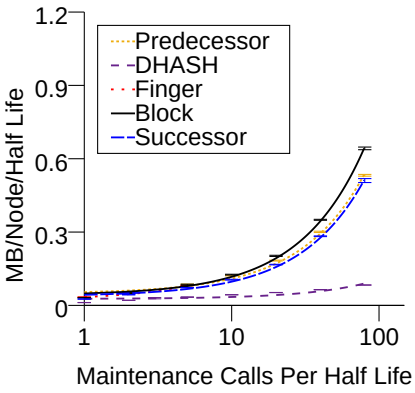

Figure 5. Overhead bandwidth in a 200 Node system with varying numbers of repairs. Successor, Predecessor, and Finger allocation overlap.

Data movement bandwidth is likely to be the bottleneck in distributed storage systems. Figure 6 shows significantly more data is moved with DHash than the dynamic algorithms since, under DHash, a single node joining produces changes in the membership of r nearby replica groups. The dynamic algorithms do not suffer from this, and all perform very similarly under this test, moving significantly less data than DHash at high maintenance rates.

9. Discussion

9.1. Related work

Our work builds on published work on dynamic replication, [23, 9], providing maintenance methods that deal with allocation collisions, and showing explicitly the effect of different allocation functions on the system. Numerous papers have discussed the reliability of storage in distributed hash tables [8, 9, 2], they have concentrated on the per-

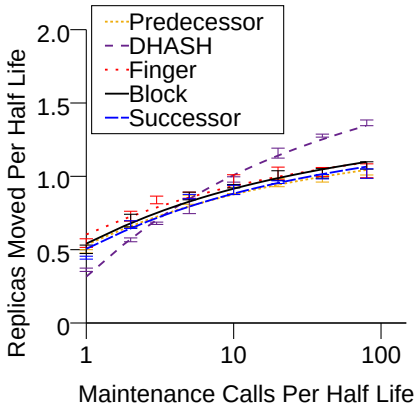

Figure 6. Proportion of data in system transmitted in one halflife. (Predecessor, Block and Successor overlap)

object loss probabilities, rather than taking a system wide view, as we have.

Other replication strategies use a DHT to store metadata pointing to the keys under which replicas are stored, as used by CFS [6]. Our work compliments such approaches by analysing the techniques necessary to reliably store this metadata.

Replication in decentralised systems based on unstructured peer to peer systems, such as Gnutella, has received much attention [14, 11, 5]. Our work complements these contributions by considering replication in Chord, a structured environment.

9.2. Future Work

In this paper, we have exclusively considered replication by storing a complete copy of the data associated with each key on another node. Erasure coding[19] is an alternative method of storing multiple copies of data, in which an item is divided into N fragments such that the item can be reconstructed from a subset of those fragments. This can provide increased efficiency in some circumstances [21, 25], at the cost of additional complexity. An extension of this work to erasure coded data would be an interesting area for further work.

9.3. Conclusions

Many decentralised storage solutions employ replication to provide reliability. We have used a combination of analysis and simulation to provide an insight into how the choice of replication strategy can affect the communication costs, reliability, and latency of such a system.

We can see that dynamic replication can achieve faster lookups, greater reliability and less replica movement than the DHash algorithm, at the cost of higher maintenance overhead and some additional complexity. We have also shown how the allocation function choice can have a dramatic impact on performance, though no single allocation function excels in both reliability and fetch latency. The choice of which function to use should therefore be based on careful consideration of which of these performance metrics is more important to the particular application.

References

[1] A. Binzenhöfer, D. Staehle, and R. Henjes. Estimating the size of a Chord ring. Technical Report 348, University of Würzburg, 11 2004.

[2] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX)*.

[3] E. Brunskill. Building peer-to-peer systems with Chord, a distributed lookup service. In *HOTOS '01: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*.

[4] J. Cates. Robust and efficient data management for a distributed hash table. Master's thesis, Massachusetts Institute of Technology.

[5] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *ACM SIGCOMM 2002*.

[6] F. Dabek, M. F. Kaashoek, D. Karger, and R. M. I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*.

[7] D. P. David Oppenheimer, Jeannie Albrecht and A. Vahdat. Distributed resource discovery on planetlab with sword. In *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*.

[8] A. Ghodsi, L. O. Alima, and S. Haridi. Symmetric replication for structured peer-to-peer systems. In *The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing*.

[9] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05)*.

[10] 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. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*.

[11] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *ICS '02: Proceedings of the 16th international conference on Supercomputing*.

[12] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS02*.

[13] R. C. Merkle. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 Symposium on Security and Privacy*. IEEE Computer Society.

[14] A. Mondal, Y. Lifu, and M. Kitsuregawa. On improving the performance-dependability of unstructured p2p systems via replication. In *Proceedings of Database and Expert Systems Applications (DEXA04)*.

[15] R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*.

[16] K. Muller. Advanced systems simulation capabilities in SimPy. In *Europython 2004*.

[17] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*.

[18] G. L. Peterson. Time-space trade-offs for asynchronous parallel models (reducibilities and equivalences). In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*.

[19] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[20] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical report.

[21] B. L. M. Rodrigo Rodrigues (MIT). High availability in dhts: Erasure coding vs. replication. In *Proceedings of IPTPS05*.

[22] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th SOSP (SOSP '01)*.

[23] M. Waldvogel, P. Hurley, and D. Bauer. Dynamic replica management in distributed hash tables. Research Report RZ–3502, IBM, 2003.

[24] E. W. Weisstein. Run. from mathworld–a wolfram web resource. http://mathworld.wolfram.com/run.html.

[25] Z. Zhang and Q. Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage. In *21st Symposium on Reliable Distributed Systems*.

[26] 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 Journal on Selected Areas in Communications*.