A Comparison Of 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.

***Index Terms*— peer to peer, reliable storage, distributed hash table, dhash, chord**

I. 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.

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 [1]–[4]. 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.

We will begin with a brief introduction to distributed hash tables in section II. In Sections III and IV, we describe two replication algorithms—*DHash* [5] and dynamic replication [6]—in the context of the Chord DHT [7]. We examine the shortcomings of dynamic replication, identify a solution, and propose a variety of placement strategies.

In Section VI, 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 VII-A and VII-D, we use simulation techniques to compare the impact of replication strategy upon fetch latency and bandwidth usage.

II. Distributed Hash Tables

Distributed hash tables provide a solution to the lookup problem in distributed systems. Given the name of a data item stored somewhere in the system, the DHT can determine the node on which that data item should be stored, often with time complexity logarithmic in the size of the network.

Several systems employing DHTs have been developed, notably: PAST, Tapestry, CAN, Kademlia, and Chord [7]–[11]. The ideas in this paper are presented in the context of the Chord DHT, although we feel they might, with some modification, be applied to any of these systems.

In Chord, both nodes and information items are assigned an ID from the same keyspace. The keyspace can be thought of as being arranged in a ring, with nodes and data arranged around it. Each node is responsible for storing the data it is the first clockwise successor of, for instance in Figure 1, node 54 would be responsible for data with keys between 40 and 53.

To provide scalability, each node need only maintain knowledge of a small proportion of other nodes in the network, including a number of its clockwise successors, its immediate predecessor, and several *fingers* at distances one half, one quarter, one eighth, and so on of the way round the ring from it. It uses this knowledge to forward requests for keys it does not own to nodes which are closer to the requested key. Figure 1 shows how a lookup for key 50 originating at Node 10 might travel between nodes. As the distance is reduced by a constant fraction at each routing hop, lookup times are logarithmic in the number of nodes.

Nodes are allowed to join and leave the system at will, causing *churn* in the set of nodes in the system. A Chord

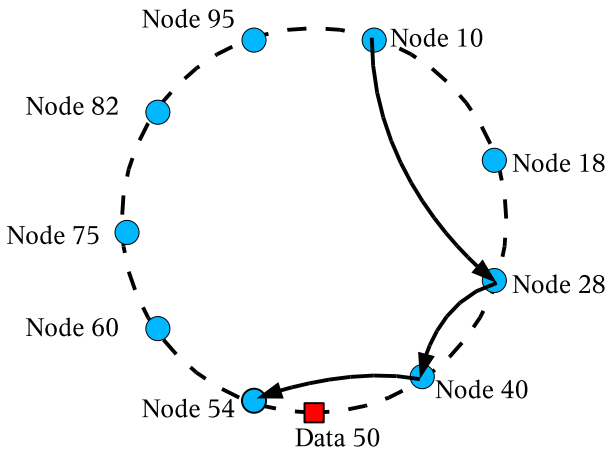

Figure 1. A Chord ring in which Node 10 is looking up key 50. For the purposes of this diagram, keys are between 0 and 100

ring regularly runs maintenance algorithms which detect failures and repair routing tables, allowing requests for a key to be routed correctly to their owner despite node churn.

While DHT routing is tolerant of node churn, storage is not. When a node fails, the key-value pairs it was responsible for become inaccessible, and must be recovered from elsewhere. This means that to provide reliable storage, a *replication algorithm* must store and maintain backup copies of data. It must do this in a manner scalable both in the number of nodes and the quantity of data stored in the DHT. In this paper, we will explore several replication algorithms, and compare their performance and reliability.

III. 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 [5] combines the placement strategy proposed for Chord with a maintenance algorithm suggested by Cates [12]; this combination is used by the DHash storage system and the Ivy File System [3].

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 [12].)
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.

IV. Dynamic Replication

This approach was proposed initially by Waldvogel et al. [6]; a similar method is used by Glacier [4] 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.

1) 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.

2) 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 V.

A. 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 [14] 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.

B. 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 one 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.

V. Allocation Functions

We will now describe a number of different allocation functions —see Table I— 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 need not be totally accurate, but is intended to be an estimate supplied by the system administrator, or calculated at runtime: see [15].

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.

TABLE I.
ALLOCATION FUNCTIONS

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	$d - (d \mod \frac{k \cdot R_{MAX}}{n})$ $+(d \mod \frac{k}{n})$ $+(m * \frac{k}{n}) \mod k$
symmetric placement	$hopsize = \frac{k}{R_{MIN}}$ $h(m,d) = (d + (m \cdot hopsize)) \textbf{ mod } k$

TABLE II.
VALUES OF R_{MAX} REQUIRED SO THAT LOCATIONS ON R DISTINCT NODES ARE FOUND WITH PROBABILITY π, FOR $\pi = (0, .95, 0.99)$ AND $R \in [2, 16]$

R_{MIN}	$\pi = 0.95$	$\pi = 0.99$
2	3	5
4	8	10
6	12	14
8	16	18
10	19	22
12	23	26
14	27	30
16	31	34

A. 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. However, we can provide a sufficient number of additional locations with high probability. We can represent the distance between two adjacent nodes as a geometrically distributed random variable D with parameter $\frac{n}{k}$, where k is the maximum allowable key. The probability of a collision is $p(D < \frac{k}{n})$. By standard properties of the geometric distribution, this is $\left(1 - \frac{n}{k}\right)^{\frac{k}{n}+1}$. Since in practical systems, $k \gg n$, we simplify this to $\lim_{\frac{k}{n} \to 0} \left(1 - \frac{n}{k}\right)^{\frac{k}{n}+1} = e^{-1}$. Thus, the number of distinct nodes encountered when we try R_{MAX} locations will be binomially distributed, with $n = R_{MAX}$, and $p = e^{-1}$. Standard techniques may be used to determine the values of R_{MAX} for which the probability of encountering R_{MIN} separate nodes is high. Sample values of R_{MAX} are given in table II.

B. 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 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.

C. 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. Finger locations become exponentially closer as the replica index increases. This means the number of non-colliding replica locations available is limited by the system size. We recommend $R_{MIN} < log_2(n) - 3$, and $R_{MAX} = 2log_2(n)$. Other than this limit on the maximum number of replicas that may be stored, finger placement is independent of system size.

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.

D. Block Placement

We may reduce the number of combinations of nodes whose failure would cause data loss by dividing the keyspace into $\frac{n}{r}$ equally sized sections, or *blocks*. All nodes in a given block store all objects in that block.

The block placement 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.

E. Symmetric Placement

Symmetric allocation is proposed in [16]. It places replicas at intervals of $\frac{K}{R_{MIN}}$ keys - the largest possible interval that allows R_{MIN} equally spaced replica locations. This makes allocation collisions very unlikely, and we do not need to create peripheral replica locations. The large intervals between replica locations also means that local routing state is not sufficient to find the core replica holders for an item, increasing maintenance bandwidth.

An advantage of Symmetric allocation is that the interval size does not depend on an estimate of system size, making it suitable for systems which experience membership fluctuations.

VI. 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.

A. 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.

B. Reliability of 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:

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

C. Reliability of 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 [17].

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

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

Figure 2 shows how we may use this function, and that from Section VI-B to compare the minimum *node reliability* p for 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. Figure 3 shows the dramatic difference between placement functions. When $p = 0.1$, block placement allows us to create reliable systems with upto 800 nodes, compared to only 100 nodes with successor placement.

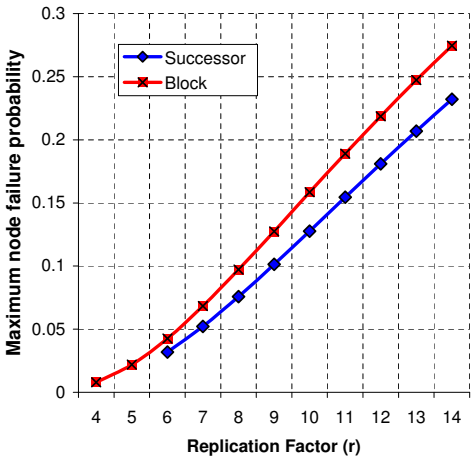

Figure 2. Critical node failure probability for a system failure probability of $< 10^{-6}$. These figures are for a 1000 node system, with varying numbers of replicas.

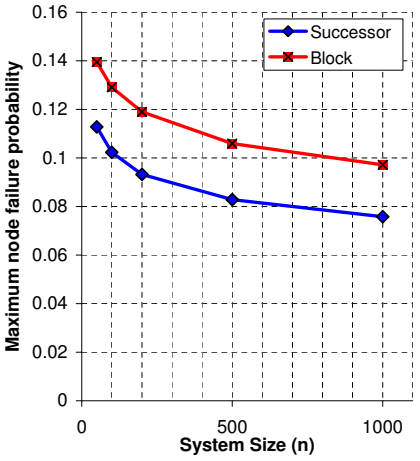

Figure 3. Critical node failure probability for a system failure probability of $< 10^{-6}$ when $r = 8$.

D. Reliability of Finger and Symmetric Placement

We will use Monte Carlo simulation to compare these 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 for each allocation function. Figure 4 shows the probability of data loss for finger and symmetric allocation. Block and successor allocation are also included for comparison.

From this plot, it is clear that finger placement is significantly less reliable than other data placement algorithms. Symmetric placement is slightly more reliable than successor placement. We also simulated the reliability of placing replicas at random locations. Interestingly, random placement results in a reliability very similar to

finger placement.

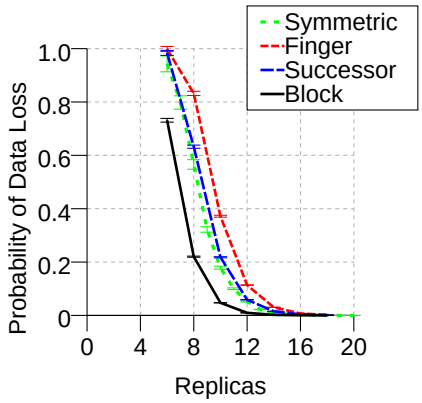

Figure 4. Probability of system failure for four allocation functions in a 500 node system where 50% of nodes fail. Error bars show 95% confidence intervals.

E. Designing a reliable system

The data above allows us to place limits upon the node reliability required to achieve a given system reliability. In practice, system designers have little control over the reliability of the nodes used for deployment, and will need to adjust the replication factor accordingly.

Increasing the replication factor provides an effective method of reaching a desired level of system reliability, at the cost of increased storage and bandwidth requirements. An alternative approach is to increase the frequency of maintenance, and thus reduce 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 [18].

VII. Simulation

We will now compare the performance and bandwidth usage of these replication algorithms. Due to the difficulty of managing large numbers of physical nodes [19], we will test the algorithms through simulation rather than through deployment.

Our simulator is based around the SimPy [20] discrete-event simulation framework, which uses generator functions rather than full threads to achieve scalability. The simulator employs 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, and 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, and the sample workload includes 50,000 fetches for data originating from randomly chosen nodes.

Parameters are chosen for the Chord algorithm so that routing will 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.

A. Fetch Latency

We define fetch latency as the time that elapses between a request for an item being issued, and the item being returned. The DHash algorithm, and each of the placement patterns for dynamic replication, result in differing data-fetch latencies. The simulation results in Figure 5 show how the fetch times differ for each of our algorithms, for various system sizes. Although all of the replication algorithms have fetch times logarithmic in system size, they exhibit significant differences in performance.

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

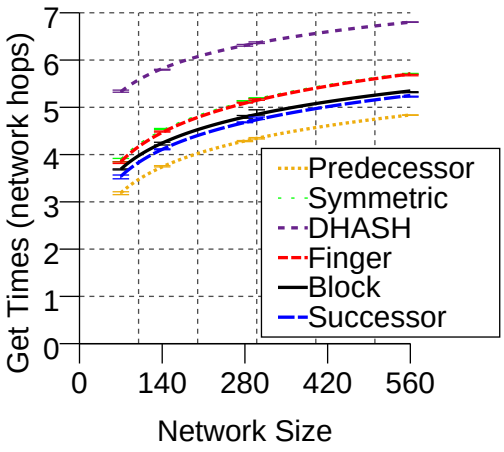

Figure 5. Get times in various system sizes. (Symmetric and Finger overlap).

The frequency of replica maintenance has varying effects on the fetch latency of each replication strategy. For those algorithms where preemptive return is common, the fact that the specific replica being requested has failed is often masked by other replica holders, which allow a request for a dead replica to succeed. As a result, Predecessor, Successor, and Block placement will perform better under high failure rates than DHash, Symmetric or Finger Placement.

B. Lookup Parallelism

In section VII-A, we considered the performance of a lookup for a randomly chosen replica location. The chord lookup algorithm, however, may find some replica locations with fewer network hops than others. We may ensure we always achieve the fastest possible lookup time by issuing requests for all replica locations in parallel. We record the time elapsed until the first replica is returned as the lookup time. Issuing lookups in parallel will result in reduced fetch times at the cost of increased bandwidth usage. Parallel lookups are not possible with DHASH.

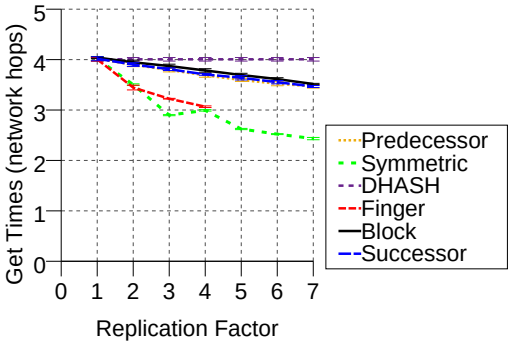

Figure 6. Parallel fetch times in various system sizes. (Successor, Predecessor and Block overlap).

Simulated fetch times when using parallel lookups are shown in Figure 6. Finger and symmetric placement achieve the lowest fetch latency. Fetches using successor, block and predecessor placement are less rapid. This is because symmetric and finger placement place replicas at well spaced intervals, increasing the likelihood that one of the replica locations will be held in local routing tables. Successor, block, and predecessor placement cluster replica locations in a small area, and it is unlikely that the owners of this area will be held in a randomly chosen node's routing table.

C. Correlated failure

We have 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 launched, and the average time they take to return, including retries, is recorded. The performance of the DHash algorithm is affected: 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. With 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.

D. 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 7, we analyse maintenance overhead bandwidth: that is, all maintenance traffic excluding replica data; we demonstrate that it varies with maintenance frequency, in terms of the number of repairs per half life.

DHash maintenance has a lower overhead than the dynamic algorithms, largely because the peripheral maintenance algorithm, which involves Bloom filter exchange, is not required. The block algorithm has the highest overhead, due to discontinuities in the placement function. The difference in bandwidth usage between predecessor and successor placement is slight, despite the fact that successor placement is able to use local routing information to find replica locations. Finger and symmetric allocation have the lowest maintenance costs, due to the infrequency of allocation collisions when using these placement patterns.

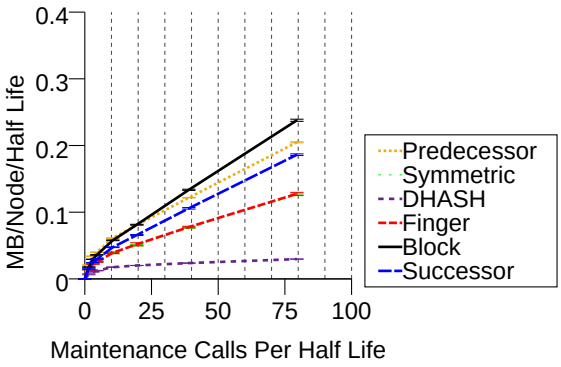

Figure 7. Overhead bandwidth in a 200-node system with varying numbers of repairs. (Symmetric and Finger allocation overlap).

Data movement bandwidth is likely to be the bottleneck in distributed storage systems. Figure 8 shows that significantly more data is moved with DHash than the dynamic algorithms since, under DHash, the addition of a single node produces changes in the membership of r nearby replica groups. The dynamic algorithms do not suffer from this: they all exhibit similar performance under this test, and move significantly less data than DHash at high maintenance rates.

VIII. Discussion

A. Related work

Our work extends earlier work on dynamic replication, [4], [6], providing maintenance methods that deal with

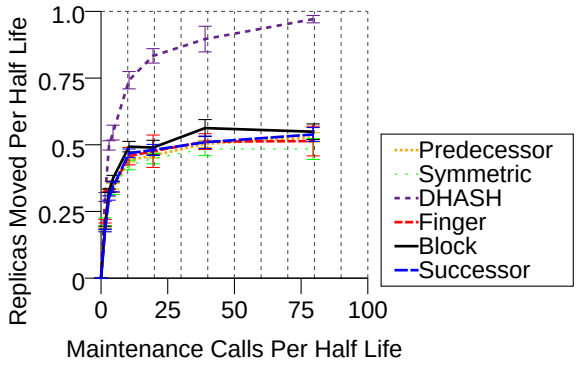

Figure 8. Proportion of data in system transmitted in one halflife. (All dynamic algorithms overlapping, with DHASH above)

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 [4], [16], [18], they have concentrated on the per-object loss probabilities, rather than taking a system-wide view.

The replica maintenance strategy we have proposed is reactive, replacing lost replicas as a reaction to node failure. Proactive replication strategies have also been proposed. These create additional replicas order to meet performance [21] or bandwidth targets [22]. The placement strategies we have proposed may also be applicable to proactive replication.

Other replication strategies use a DHT to store metadata pointing to the keys under which replicas are stored, as used by CFS [1]. Our work complements such approaches, by analysing the techniques required for reliable storage of this metadata.

Replication in decentralised systems based on unstructured peer to peer systems, such as Gnutella, has received much attention [23]–[25]. Our work complements these contributions by considering replication in Chord, a structured environment.

B. Future Work

In this paper, we have considered replication by storing a complete copy of the data associated with each key on another node. Erasure coding [26] is an alternative method of storing multiple copies of data: an item is divided into fragments in such a way that a subset will suffice for its reconstruction. This can provide increased efficiency in some circumstances [27], [28], at the cost of additional complexity. An extension of this work to erasure coded data would be an interesting area for further work.

C. 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 affects 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] 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)*.

[2] 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)*.

[3] 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)*.

[4] 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)*.

[5] 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*.

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

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

[8] 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)*.

[9] 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*.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, “A scalable content addressable network,” Tech. Rep.

[11] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the XOR metric,” in *Proceedings of IPTPS02*.

[12] J. Cates, “Robust and efficient data management for a distributed hash table,” Master’s thesis, Massachusetts Institute of Technology.

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

[14] 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*.

[15] A. Binzenhöfer, D. Staehle, and R. Henjes, "Estimating the size of a Chord ring," University of Würzburg, Tech. Rep. 348, 11 2004.
[16] 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*.
[17] E. W. Weisstein, "Run. from mathworld–a wolfram web resource. http://mathworld.wolfram.com/run.html."
[18] 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)*.
[19] 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)*.
[20] K. Muller, "Advanced systems simulation capabilities in SimPy," in *Europython 2004*.
[21] V. Ramasubramanian and E. Sirer, "Beehive: Exploiting power law query distributions for o(1) lookup performance in peer to peer overlays." in *Symposium on Networked Systems Design and Implementation, San Francisco CA, Mar 2004.*
[22] E. Sit, A. Haeberlen, F. Dabek, B. Chun, H. Weatherspoon, R. Morris, M. Kaashoek, and J. Kubiatowicz, "Proactive replication for data durability," in *Proceedings of the 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006)*.
[23] 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)*.
[24] 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*.
[25] E. Cohen and S. Shenker, "Replication strategies in unstructured peer-to-peer networks," in *ACM SIGCOMM 2002*.
[26] M. O. Rabin, "Efficient dispersal of information for security, load balancing, and fault tolerance," *Journal of the ACM*, vol. 36, no. 2, pp. 335–348, 1989.
[27] B. L. M. Rodrigo Rodrigues (MIT), "High availability in dhts: Erasure coding vs. replication," in *Proceedings of IPTPS05*.
[28] Z. Zhang and Q. Lian, "Reperasure: Replication protocol using erasure-code in peer-to-peer storage," in *21st Symposium on Reliable Distributed Systems*.

Matthew Leslie recently submitted his doctoral thesis on Reliable Peer to Peer Grid Middleware at the University of Oxford. He received his BA degrees in computation from the University of Oxford in 2001. He currently works as an IT Quant Analyst with Merrill Lynch in London, England. His research interests include Grid Computing, Peer to Peer Storage, and Peer to Peer Security.

Jim Davies Jim Davies is Professor of Software Engineering at the University of Oxford and the Director of the University's Software Engineering Programme, an external programme of advanced education aimed at working professionals. He is leading research into distributed systems for cancer clinical trials informatics, and the model-driven development of XML databases.