

Symmetric Replication for Structured Peer-to-Peer Systems ^{*}

Ali Ghodsi¹, Luc Onana Alima², and Seif Haridi^{1,2}

¹ IMIT-Royal Institute of Technology (KTH)
{aligh,seif}@imit.kth.se,

² Swedish Institute of Computer Science (SICS)
{onana}@sics.se

Abstract. Structured peer-to-peer systems rely on replication as a basic means to provide fault-tolerance in presence of high churn. Most select replicas using either multiple hash functions, successor-lists, or leaf-sets. We show that all three alternatives have limitations. We present and provide full algorithmic specification for a generic replication scheme called symmetric replication which only needs $O(1)$ message for every join and leave operation to maintain any replication degree. The scheme is applicable to all existing structured peer-to-peer systems, and can be implemented on-top of any DHT. The scheme has been implemented in our DKS system, and is used to do load-balancing, end-to-end fault-tolerance, and to increase the security by using distributed voting. We outline an extension to the scheme, implemented in DKS, which adds routing proximity to reduce latencies.

1 Introduction

Research on structured peer-to-peer systems have produced systems which have strong performance in presence of dynamism. To cope with the dynamism, these systems rely on replication as a basic means to achieve robustness and fault-tolerance.

Most existing structured peer-to-peer systems either use *multiple hash functions*, *successor-lists*, or *leaf-sets* for choosing replicas.

Contribution. We analyze using multiple hash functions, successor-lists, and leaf-sets, and point out their disadvantages. Thereafter, we propose a new replication scheme, called *symmetric replication*, which we have implemented and added to the DKS system[5]. We provide full algorithmic specification of our scheme, something which we have not found for any other replication schemes for structured peer-to-peer systems. The advantages of symmetric replication are manifold. First, it is a general end-to-end scheme and can be applied to all structured peer-to-peer systems. Furthermore, each join and leave operation only

^{*} This work was funded by the European project EVERGROW IST-2004-001935, the Vinnova project GES3 in Sweden.

requires sending 1 message to maintain the replication degree. Moreover, nodes can make concurrent requests to any specified replica. This opens up a window of opportunities. It is more secure as multiple requests to different replicas do not need to pass through the same node. Hence, distributed voting can be using the compare the results to increase security, without the risk of having the results tampered by one node. It can also be used for load-balancing by sending requests to a random replica. Fault-tolerance becomes easy to implement in an end-to-end fashion by just resending a timed-out request to another replica. Finally, we show an optional extension of symmetric replication, which is used in DKS to achieve proximity neighbor selection.

Outline. Section 2 gives preliminaries. In Section 3, we analyze three major existing replication schemes. We introduce our proposed scheme in Section 4. Section 5 outlines different techniques that can be built on-top of symmetric replication. Finally, the last sections, 7 and 8, discuss related work and conclude.

2 Preliminaries

In this section we present preliminary definitions used in the rest of the paper.

We assume a distributed system modeled by a set of peers communicating by message passing through a communication network that is: (i) connected, (ii) asynchronous, (iii) reliable, and (iv) provides FIFO communication.

A distributed algorithm running on a peer in the system is described as a set of rules of the form:

$$R :: \frac{\mathbf{receive}(Sender, Receiver, MESSAGE(arg_1, \dots, arg_n))}{Action}$$

The rule R describes the event of receiving MESSAGE from $Sender$ at the peer $Receiver$ and the $Action$ taken to handle that event. A $Sender$ of a message executes the statement $\mathbf{send}(Sender, Receiver, MESSAGE(arg_1, \dots, arg_n))$ to send a message to $Receiver$.

We now give the definitions used in the rest of the paper.

In most systems, each peer in the system is assumed to receive a logical identifier from an identifier space, denoted \mathcal{I} , which is perceived as a ring (modulo the size of the space). We assume for simplicity that the identifier space is discrete and defined as $\mathcal{I} = \{0, \dots, N-1\}$ for some large constant N ($N \in \mathbb{N}$). The identifier space is a metric space has a distance function $d : \mathcal{I} \times \mathcal{I} \rightarrow \mathbb{R}$ satisfying the following criteria: **a**) $d(x, y) \geq 0$, **b**) $d(x, y) = 0$, iff $x = y$, **c**) $d(x, y) = d(y, x)$, **d**) $d(x, z) \leq d(x, y) + d(y, z)$. If requirements **c** and **d** are not satisfied we call it a “pseudo”-metric space.

We now formally define a structured P2P system.

Definition 1. *A structured P2P system is a P2P system with a “pseudo”-metric space where each peer in the system has got an identifier from the “pseudo”-metric space and the choice of the neighbors of a peer are constrained in terms of the distance function of the “pseudo”-metric space.*

On top of a structured P2P system a distributed hash table (DHT) abstraction can be built by deterministically mapping each identifier i in the identifier space to a peer with identifier p . We denote the identifiers of the peers in the system at a certain time \mathcal{P} ($\mathcal{P} \subseteq \mathcal{I}$).

To make the rest of the paper concrete, we will define a distance function, as well as a mapping from identifiers to peers, as commonly used in [14, 5, 11, 7]. Our replication scheme, however, does not assume these definitions and can thus be applied to a variety of structured P2P systems.

We will assume the distance function is defined as:

$$d(x, y) = y \ominus x$$

The operator $\ominus : \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ is defined as:

$$\ominus(x, y) = x - y \bmod N$$

Similarly $\oplus : \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ is defined as:

$$\oplus(x, y) = x + y \bmod N$$

We use infix-notation for the binary operators \ominus and \oplus to ease the reading.

For the mapping of identifiers to peers we map each identifier i in the system to its *successor*, which is the first peer met in the identifier space going in clockwise direction starting at i . The function $s_{\mathcal{P}} : \mathcal{I} \rightarrow \mathcal{P}$ is used for this purpose:

$$s_{\mathcal{P}}(i) = i \oplus \min\{d(i, p) : p \in \mathcal{P}\}$$

We call a peer n *responsible* for an item i iff $s_{\mathcal{P}}(i) = n$. Sometimes we will refer to peer n as the *master peer* for item i to distinguish it from other peers replicating item i .

To provide a DHT abstraction, each data item d is mapped to the identifier space using a globally known function H . Hence, a data item d is stored on the peer $s_{\mathcal{P}}(H(d))$.

3 Major Existing Replication Schemes

The use of several hashing functions for replication, which is mentioned in CAN, Tapestry, and other systems[10, 3], is closest to our symmetric replication scheme. However, it has several major disadvantages. It requires the inverse of the hashing functions to maintain the replication factor, which is impossible by the definition of hash functions. To see why, assume a replication degree of two, and hence two different hashing functions, H_1 and H_2 , which are known by all nodes. Assume a node with identifier 10 is storing any items with identifiers in the range [5, 10]. An item with key “course” might be mapped to identifier 7 using H_1 , and therefore be stored at node 10. If node 10 fails, this item should be fetched from the other replica and replication to maintain the replication degree 2. To do this, however, it would be required to find out the key “course” such that the node responsible for H_2 (“course”) can be contacted. Worse, even if the inverse of the hash functions were available, each single item that the failed peer maintained would be dispersed all over the system when using different hash functions, making it necessary to fetch each item from a different peer.

If the replication degree is not restored each time there is a failure, items soon disappear from the system. Assume every node fails with exponential distribution with intensity λ . Then every node fails after an average of $\frac{1}{\lambda}$ time units. Given replication degree f , after an expected $\frac{f}{\lambda}$ time all replicas would be lost.

The successor-list scheme is, however, able to restore the replication degree after failures. The scheme [14] fulfills two purposes. One is to replicate items on the successors such that lookups for items on a failed peer can be resolved by its successor, since the failed peer's items automatically become the responsibility of the successor. The other purpose is to store routing information about f successors, such that as soon as a node's successor, p , is detected as failed, it is quickly replaced with p 's successor. The leaf-set scheme [11, 12] has the same two uses as the successor-list scheme. But in addition, a lookup request might first arrive at one of the replicas, which then can resolve the lookup.

Our conjecture is that the two mechanisms should be separated. While having routing information about the successors or leafs is useful for routing table correction, replication on the same set has several disadvantages.

The first disadvantage is that both schemes need $\Omega(f)$ messages for every join and leave event to maintain a replication degree of size f . The reason for this is that if a node leaves the system, its f successors (or $\lfloor \frac{f}{2} \rfloor$ predecessors and successors in the leaf-set scheme) will by definition belong to the successor-list (or leaf-set) of a node which they previously were not in. Hence, they need to fetch or release items. Figure 1 illustrates this with an example using the successor-list scheme. The figure shows a system with the peers 1, ..., 7 as indicated by the circles. For simplicity, the system contains the items 1, ..., 7. Assuming a replication factor of 3, the figure shows the identifiers of the items each peer is replicating. If peer 4 has failed, peers 5, 6, 7 need to establish connections with other peers and fetch the items 1, 2, 3 respectively to maintain the replication factor.

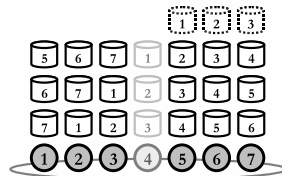


Fig. 1: A system populated with peers 1, ..., 7, as indicated by the circles in the figure. The figure shows the identifier of the items each peer is storing given that the replication factor is 3. E.g. peer 1 is replicating items 5, 6, 7.

Furthermore, the re-establishment of the replication degree needs to be coordinated by some node that triggers a replication maintenance algorithm at each of the successors (and predecessors in the leaf-set case). The coordinating peer might however fail or leave the system making it necessary to use a more robust algorithm. Many implementations, such as Bamboo[4], or the previous version of the DKS system[1], used an epidemic algorithm, where each node sends a message to its neighbors whenever it detects a change, leading to $\theta(f^2)$ messages for every event, or time interval if the algorithm is periodic, given a replication degree of size f .

Furthermore, any request to a specific replica, m , must first be routed to a node in the successor-list, or the leaf set, before it can be forwarded to m . The reason behind this is that the requesting node has no information about the logical identifier of the replicas, while the nodes in the successor-list, or the leaf-set, maintain such information. In the successor-list scheme, the first replica routed to will always be the numerically closest replica in the successor-list, while in the leaf-set this can be any of the replicas. In both systems, however, the first replica met is a bottleneck, which can fail, decelerate the whole operation, or in the case of an adversary, launch a malicious attack.

The leaf-set scheme is, however, better in this respect as it naturally balances requests to different replicas, given that $f \geq 2^b$, where f is the replication degree, and 2^b is the arity of the search tree.

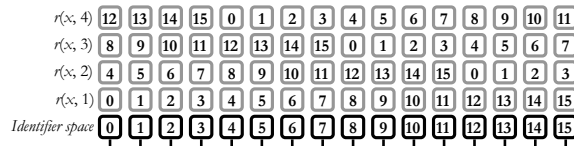


Fig. 2: The identifiers associated with each identifier in the system in a system with identifier space of size $N = 16$ and a replication factor of size $f = 4$.

4 The Symmetric Replication Scheme

The main idea behind symmetric replication is that each identifier in the system should be associated with f other identifiers. If identifier i is associated with identifier r , then any item with identifier i should be stored at the peers responsible for identifiers i , and r . Similarly, any item with identifier r should also be stored at the peers responsible for the identifiers i , and r .

Formally, each identifier in the system is associated with a set of f distinct identifiers such that the following always holds: if the identifier i is associated with the set of identifiers r_1, \dots, r_f , then the identifier r_x , for $1 \leq x \leq f$, is associated with the identifiers r_1, \dots, r_f as well.

```

Subroutine :: JOINREPLICATION
  send( $n : succ : \mathbf{RETRIEVEITEMS}(pred, n)$ )

R1 :: receive( $m : n : \mathbf{RETRIEVEITEMS}(start, end)$ )
  for  $r := 1$  to  $f$  do
    items[ $r$ ] :=  $\emptyset$ 
     $i := start$ 
    while  $i \neq end$  do
       $i := i \oplus 1$ 
      items[ $r$ ][ $i$ ] := localHashTable[ $r$ ][ $i$ ]
    od
  od
  send( $n : m : \mathbf{REPLICATE}(items, start, end)$ )

R2 :: receive( $m : n : \mathbf{REPLICATE}(items, start, end)$ )
  for  $r := 1$  to  $f$  do
     $i := start$ 
    while  $i \neq end$  do
       $i := i \oplus 1$ 
      localHashTable[ $r$ ][ $i$ ] := items[ $r$ ][ $i$ ]
    od
  od

Subroutine :: LEAVEREPLICATION
  for  $r := 1$  to  $f$  do
    items[ $r$ ] :=  $\emptyset$ 
     $i := pred$ 
    while  $i \neq n$  do
       $i := i \oplus 1$ 
      items[ $r$ ][ $i$ ] := localHashTable[ $r$ ][ $i$ ]
    od
  od
  send( $n : succ : \mathbf{REPLICATE}(items, pred, n)$ )

```

Fig. 3: Rules $R1$, and $R2$ show the replication algorithm for joins and leaves.

Put differently, the identifier space is partitioned into $\frac{N}{f}$ equivalence classes such that identifiers in an equivalence class are all associated with each other. Any such partition will work, but we will for simplicity chose the congruence classes modulo f .

We now explain how each identifier i is associated to f other identifiers to achieve replication degree f . Let $\mathcal{F} = \{1, \dots, f\}$, then identifier i is associated to the f different identifiers given by the function $r : \mathcal{I} \times \mathcal{F} \rightarrow \mathcal{I}$ defined as: $r(i, x) = i \oplus (x - 1) \frac{N}{f}$

Figure 2 shows how identifiers are associated in an identifier space of size $N = 16$ and a replication factor $f = 4$. The black boxes illustrate each identifier in the identifier space, and on-top of each black box the identifiers associated with it are shown in light boxes. For example, identifier 0 is associated with the identifiers 0 ($r(0, 1) = 0$), 4 ($r(0, 2) = 4$), 8 ($r(0, 3) = 8$), 12 ($r(0, 4) = 12$).

As we mentioned before, in a system without any replication, each item with identifier i is stored at the responsible peer given by $s_{\mathcal{P}}(i)$, which in our example is the successor of item i . To replicate items in our scheme, the responsible peer of identifier i stores every item with an identifier associated with i . This implies that to find an item with identifier i , a request can be made for any of the identifiers associated with i .

```

R3 :: receive(m : n : INSERTITEM(key, value))
    for r := 1 to f do
        replicaKey := key  $\oplus$  (r - 1)  $\frac{N}{f}$ 
        respNode := FINDSUCCESSOR(replicaKey)
        send(n : respNode : ADDITEM(replicaKey, value, r))
    od

R4 :: receive(m : n : ADDITEM(key, value, r))
    localHashTable[r][key] := value

Subroutine :: LOOKUPITEM(key, r)
    replicaKey := key  $\oplus$  (i - 1)  $\frac{N}{f}$ 
    respNode := FINDSUCCESSOR(replicaKey)
    send(n : respNode : GETITEM(replicaKey))

R5 :: receive(m : n : GETITEM(key))
    send(n : m : GETITEMRESP( $\overline{Key}$ , localHashTable[r][key]))

```

Fig. 4: The replication algorithms for inserting and looking up items shown by rules R3, R4.

Formally, in a system with the peers \mathcal{P} , an item with identifier i is stored on the f peers given by $s_{\mathcal{P}}(r(x, i))$, for all x ($1 \leq x \leq f$).

For example, if the identifier 0 is associated with the identifiers 0, 4, 8, 12, any peer responsible for any of the items 0, 4, 8, or 12 has to store all of the items 0, 4, 8, and 12. Hence, if we are interested in retrieving item 0, we can ask the peer responsible for any of the items 0, 4, 8, 12.

For the symmetry requirement to always be true, it is required that the replication factor f divides the size of the identifier space N . We find this reasonable as the size of the successor-list, as well as N , are constants in most systems. We have developed an intricate scheme where f can be freely chosen with a deviation of $|1|$, but omit it for space reasons.

Algorithms. We now give a description of all algorithms. The algorithms might need to be slightly modified to fit a system with a different mapping of identifiers to peers.

Each peer in the system has all its items stored in a two-dimensional (f, N) -array denoted *localHashTable*. The first dimension of the array represents the f identifiers associated with the identifier in the second dimension of the array. Hence, *localHashTable*[i][j] represents items with identifiers $r(j, i)$.

Whenever a new peer n joins the system, it makes a call to the sub-routine JOINREPLICATION (Fig.3) which immediately sends a RETRIEVEITEMS-message to its successor (denoted *succ*) asking it about all items n should be storing. The message declares which items it is interested in by specifying a range $(pred, n)$, where *pred* is its predecessor's identifier and n is its own identifier.

Once the successor peer receives the RETRIEVEITEMS-message it initializes an empty two-dimensional (f, N) -array called *items*. Thereafter, each item asso-

```

Subroutine :: FAILUREREPLICATION(failedId, predId, r)
  start := predId  $\oplus$  (r - 1)  $\frac{N}{f}$ 
  end := failedId  $\oplus$  (r - 1)  $\frac{N}{f}$ 
  respNode := FINDSUCCESSOR(start)
  send(n : respNode :
    RESTBCAST(start, end, MSG(start, end, n)))

Subroutine :: MSGHANDLER(start, end, n')
  for r := 1 to f do
    items[r] :=  $\emptyset$ 
    i := start
    while i  $\neq$  end do
      i := i  $\oplus$  1
      items[r][i] := localHashTable[r][i]
    od
  od
  send(n : n' : REPLICATE(items, start, end))

```

Fig. 5: The replication algorithms for failures.

ciated with an identifier in the specified interval is copied from *localHashTable* to *items* and sent back in a REPLICATE-message to the newly joined peer. Upon receipt of the REPLICATE-message, the newly joined peer copies *items* to its *localHashTable*. The new peer is now ready to receive requests from other peers in the system.

The leave algorithm (Fig.3) works similarly to the join algorithm. Whenever a peer wants to leave the system it makes a call to the sub-routine called LEAVEREPLICATION which copies all items it is responsible for and sends them in a REPLICATE-message to its successor. Notice that we do not delete items that are no longer a peer's responsibility. If disk space is limited, such an optimization could be added.

Figure 4 shows the algorithms used to insert or lookup an item. To save space, we have not shown the asynchronous algorithm for finding the responsible of an item. Such an algorithm can commonly be found in most structured P2P systems. We assume that the sub-routine **FINDSUCCESSOR** implements a simple synchronous distributed algorithm which finds the peer responsible for a given identifier (See [14] for such an algorithm).

To insert an item, the inserting peer simply makes concurrent insertions to every location where the replica should be stored.

For the lookup algorithm, we only show a sub-routine that takes the two parameters *key* and *i* ($1 \leq i \leq f$) and finds the responsible peer for the *i*:th replica of identifier *key*. On top of this abstraction, different kinds of lookup services can be built, such as the ones mentioned in Section 5.

For handling failures, the algorithm shown in Figure 5 is used. The sub-routine **FAILUREREPLICATION** is called at the successor of the failed peer with parameters specifying the failed peer's identifier, the failed peer's predecessor's

identifier, and an integer specifying which of the f replicas to fetch the items from.

For example, assume the system depicted by Figure 2 populated with peers 0, 3, 4, 6, 7, where peer 3 has failed. Peer 4 should ideally fetch items in the range (1, 3) to restore the replication degree. Items (1, 3) are associated with items (5, 7), (9, 11), and (13, 15). Peer 4 chooses to fetch them from the peers responsible for (5, 7) which are peers 6 and 7. Instead of sending the message around the ring in the interval (5, 7) a restricted version of our broadcast algorithm[6] is used which covers the given interval in $O(M)$ messages, where M is the number of peers in the given interval. Assuming a uniform distribution of peer identifiers, the restricted broadcast needs to send one message on average on every failure.

5 Exploiting Symmetric Replication

In this section we discuss simple end-to-end techniques that exploit symmetric replication's ability to do concurrent requests to replicas to enhance the security and performance of the system.

In the DKS system, distributed voting is used to ensure that data items received are not tampered with. This is done by sending requests to all m replicas and deciding which replica to accept based on a majority vote. The probability that an item has been tampered can be calculated and reported to the requesting user or application. If the probability that an item is tampered is p , and m ($2 \leq m \leq f$) concurrent requests are made out of which a majority of g ($0 \leq g \leq m$) answers are identical, the probability of such a configuration is given the Bernoulli trials: $\binom{m}{g} p^g (1-p)^{m-g}$. The system can automatically increase the number of concurrent requests m to achieve a certain degree of certainty in the results.

The advantage of symmetric replication is not only restricted to enhancing the security of the system. Symmetric replication can be used to send out multiple concurrent requests and picking the first response that arrives. The advantages of this are twofold. First, it enhances performance. Second, it provides fault-tolerance in an end-to-end fashion since the failure of a peer along the path of one request does not require repeating the request as it is likely that another one of the concurrent requests succeeds. If such a scheme is not used, outgoing messages have to be buffered at a peer together with timers, and whenever a timeout occurs, the messages need to be sent again with risk of ending up at the same failed node.

Proximity Neighbor Selection. The symmetry property could be used within the routing process to achieve proximity neighbor selection. This is particularly useful in systems such as *DKS*, Chord, and Koorde, where the legitimate state of the routing information is rigid[2].

The idea is that each peer in the system augments its routing table to contain f entries for each routing entry, one for each replica of a routing entry.

For example in a Chord system with an identifier space of size N , each peer p maintains pointers to the successors of the identifiers $p \oplus 2^i$ for all i ($0 \leq i < \log(N)$). To enhance this system, the routing information at each peer is augmented with a pointer to the responsible peer of every identifier associated with the identifier $p \oplus 2^i$. Every entry in the routing table is also tagged with proximity information.

Proximity neighbor selection can then be achieved in the following way. To route a message to the peer responsible for identifier d , each message in the routing process is piggy-backed with a parameter r that specifies which of d 's replicas is currently searched for. A peer n in the routing process can then calculate its distance to the r :th replica of d . Peer n now has f peers that it can choose among which each have a shorter distance to each respective replica of d . Naturally, peer n routes to the peer which has the best proximity, and updates r in the outgoing message to reflect the intended replica.

6 Evaluation

We have simulated the symmetric replication scheme and the successor-list scheme in a stochastic discrete event simulator developed using Mozart[8]. The symmetric replication scheme is implemented using the algorithms described in this paper. The lack of algorithmic specification of the successor-list scheme, or any other scheme for that matter, led us to implement a successor-list scheme in which every join or leave only costs f messages. In reality, however, many schemes use more messages to maintain the replication degree.

The method of independent replications has been used to generate unbiased estimates from independent and identically distributed variables. All the simulations are non-terminating where nodes join and leave with an exponential distribution with parameter λ , and a replication factor of 5.

Figure 6 shows different simulations where the probability of ungraceful failures is 0.05, 0.1, and 0.2. We also vary the initial number of nodes that are in the system before the warm-up period to 500 and 2000.

The figure shows that the successor-list scheme consumes more messages to maintain the replication degree as nodes join and leaves the system, while the symmetric replication scheme maintains the replication degree with less amount of messages.

7 Related Work

Beehive[9] proposes to pro-actively replicate items and achieves $O(1)$ lookup latency. Beehive works well with structured P2P systems based on fixed space division³, such as Tapestry, Pastry, and P-Grid, it is however not suited for systems based on relative space division, such as Chord, Koorde, and DKS. In contrast, symmetric replication works with both type of systems. In

³ For more information of fixed and relative space division, please refer to [2]

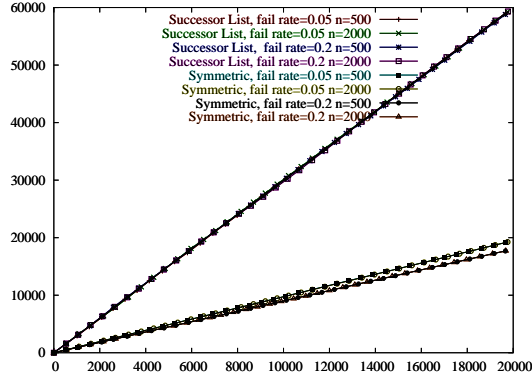


Fig. 6: Symmetric Replication vs. Successor-List Scheme. X-axis shows the simulated time line, while the Y-axis shows the total number of messages consumed to maintain the replication degree.

addition, Beehive replicates on its leaf set as well. Furthermore, Beehive does not address security issues as the authors acknowledge. Another disadvantage is that adaptive replication schemes are difficult to build transactions on-top of, while constant degree schemes are well-suited for that purpose.

In, [13], the security breaches of the successor-list scheme are identified, no solution is however proposed for the particular problem.

8 Conclusions

We have analyzed the three main approaches used for replication in structured peer-to-peer systems, multiple hash functions, successor-lists, and leaf-sets, and found that they have drawbacks.

The first scheme has the drawback that the replication degree cannot be restored after failures, and hence items will disappear after a while. The other schemes both require at least $\Omega(f)$ messages for each join and leave event to maintain a replication degree of size f . Often, however, epidemic algorithms are used which use $O(f^2)$ messages in each round.

The second disadvantage is that the requesting peer cannot directly route to a specific replica, but the request is routed to the first replica encountered, which then forwards the request to the desired replica. The possibility of routing directly to a specific replica is useful if an insertion or update is required, or if several replicas is to be looked up concurrently. The first replica encountered is thus a bottleneck, which can fail, decelerate the operation, or launch a security attack. The leaf-set scheme is better, however, as the first replica met can be any of the replicas, while in the successor-list scheme, the same peer is always encountered when searching for a given item.

To rectify the problems in the mentioned other schemes, we proposed a new scheme and provided full algorithmic specifications of it. The scheme is applica-

ble to all structured peer-to-peer systems. In our scheme, every join and leave operation requires $O(1)$ message to maintain the replication degree, independent of the size of the replication factor. Furthermore, requests can be directed to any specific replica directly. As a result, concurrent requests can be made, which can be used to prevent security attacks by using distributed voting. Our simulations verify that the cost of maintaining the replication degree is lower when using symmetric replication.

Finally, we outlined an optional extension to our scheme to achieve proximity neighbor selection.

References

1. L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *The 3rd International workshop on Global and Peer-To-Peer Computing on large scale distributed systems - CCGRID2003*, Tokyo, Japan, May 2003.
2. L. O. Alima, A. Ghodsi, and S. Haridi. A Framework for Structured Peer-to-Peer Overlay Networks. In *LNCS post-proceedings of Global Computing*, pages 223–250. Springer Verlag, 2004.
3. J. Stribling S. C. Rhea A. D. Joseph B. Y. Zhao, L. Huang and J. Kubiawicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications (Special Issue: Recent Advances In Service Overlay Networks)*, 22(1):41–53, January 2004.
4. Bamboo. <http://bamboo-dht.org/>, 2003.
5. Distributed k-ary System. <http://dks.sics.se>, 2004.
6. A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In *15th IASTED International Conference, Parallel and Distributed Computing and Systems*, Marina del Rey, CA, USA, November 2003.
7. M. F. Kaashoek and D. R. Karger. Koorde: A Simple Degree-optimal Distributed Hash Table. In *The 2nd Interational Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.
8. Mozart Consortium. <http://www.mozart-oz.org>, 2003.
9. V. Ramasubramanian and E. Sirer. Beehive: The Design and Implementation of a Next Generation Name Service for the Internet. In *ACM SIGCOMM 2004*, 2004.
10. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. Technical Report TR-00-010, Berkeley, CA, 2000.
11. A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes in Computer Science*, 2218, 2001.
12. 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)*, Chateau Lake Louise, Banff, Canada, October 2001.
13. E. Sit and R. Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.
14. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001*, pages 149–160, San Deigo, CA, August 2001.