A Symmetric Replication Scheme for Increased Security and Performance in Structured Overlay Networks *

Ali Ghodsi[1], Luc Onana Alima[2], 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

Technical Report T2004:11 ISSN 1100-3154 ISRN: SICS-T–2004/11-SE

Abstract. Existing structured peer-to-peer systems heavily rely on replication as a means to provide fault-tolerance. Many systems use the so-called *successor-list* scheme for replication. We argue that this scheme has grave limitations. First, these systems are vulnerable to, what we call, *Mendacity attacks*, where a malicious peer can lie about other peers to gain full control over all replicas of an item. Second, the successor-list scheme prevents the peers from doing concurrent-requests to replicas of an item. We present, and provide full algorithmic specification for, a generic replication scheme called *symmetric replication*. The scheme is applicable to all existing structured peer-to-peer systems. In contrast to the successor-list scheme, our scheme makes replicas independent of each other, preventing Mendacity attacks while enabling concurrent requests. Concurrent requests can be used for increasing the security by using voting or consensus algorithms to ensure the correctness of replicas. Moreover, concurrent requests can be used for load-balancing of requests, and to add locality awareness. Finally, to maintain the replication factor, the successor-list scheme uses a complex algorithm that involves all peers replicating a departing peer. In contrast, our symmetric replication scheme only involves two peers to restore the replication factor and thus avoids such complex algorithms.

1 Introduction

The popularity of file-sharing applications such as Napster and Gnutella has motivated much research recently in the field of peer-to-peer computing. One strand of this research has focused on structured peer-to-peer systems where strong guarantees are claimed on the performance in presence of high network dynamism[7, 10, 1]. To cope with this dynamism, these systems all heavily rely on replication as a basic means to achieve robustness and fault-tolerance.

Many of the existing structured peer-to-peer systems use the so called *successor-list* scheme for replication. In these systems, each peer receives a logical identifier from a

* This work was funded by the European project PEPITO IST-2001-32234, the European project EVERGROW IST-2004-001935, the Vinnova projects PPC and GES3 in Sweden.

virtual identifier space. The virtual identifier space is a ring. In the successor-list scheme, data that a peer with identifier p stores is replicated on the f successive peers with identifiers closest to p in clock-wise orientation[3]. This scheme has severe security drawbacks and grave limitations as we shall unfold in Section 3. The drawbacks stem from the inability to allow concurrent requests to the replicas of an item. Consequently, items are replicated f times in the system, but only the first replica is ever used as the rest of the items are there only as a backup in case of failures.

1.1 Contribution

The contributions of this paper are along three lines. First, we analyze the existing successor-list approach and its drawbacks. In particular, we point out a vulnerability which can be exploited by a malicious peer to gain full control over an item and all its replicas. As a consequence, most techniques based on comparing replicas to enhance security become useless. Second, we propose a new replication scheme, called *symmetric replication*, which makes enables concurrent requests to replicas. Finally, we analyze the proposed scheme and show the techniques that it enables as well as extensions to it to achieve adaptive replication and locality awareness.

The advantages of symmetric replication are manifold. First, it is general and can be applied to all structured peer-to-peer systems. Furthermore, it enables techniques that can be built in a modular way on-top of it to increase the system's security and performance while fault-tolerance is simplified. One of the most important properties of our scheme is that each join or leave operation involves fetching items from only one peer leading to better performance as well as decreased algorithmic complexity. This can be compared to the successor-list scheme where a complicated algorithm that involves all f replicas is needed to maintain the replication factor when peers leave or fail.

The proposed scheme has been implemented and is used in our $\mathcal{DKS}$[1] system.

1.2 Outline

Section 2 gives the preliminaries used in the rest of the paper. In Section 3, we analyze the successor-list scheme and point out its disadvantages. Thereafter, we introduce our proposed scheme in Section 4. After that, Section 5 shows different techniques that combined with symmetric replication enhance the security and performance of the system. In Section 6 we show extensions of symmetric replication. Finally we discuss the related work in Section 7 and conclude in Section 8.

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.

[3] Some systems replicate on the leaf-set. This is identical to the successor-list scheme except that the orientation is both clock-wise and anti-clockwise.

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

$$R \ :: \ \frac{\textbf{receive}(\textit{Sender}, \textit{Receiver}, \textsc{Message}(arg_1, .., arg_n))}{\text{Action}}$$

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

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

A metric space has a distance function $d : \mathcal{I} \times \mathcal{I} \rightarrow \mathbb{R}$ satisfying the following criteria:

1. $d(x, y) \geq 0$
2. $d(x, y) = 0$, iff $x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(y, z)$

Requirements 3 and 4 might not be satisfied in a structured P2P system, therefore we call it a "pseudo"-metric space.

A structured peer-to-peer system has an identifier space $\mathcal{I}$ which is a "pseudo"-metric space. In our model we will assume for simplicity that the identifier space is discrete and defined as $\mathcal{I} = \{0, \cdots, N-1\}$ for some large N ($N \in \mathbb{N}$). When we refer to an identifier in this paper, it is always assumed that the identifier is an element of $\mathcal{I}$.

We can now formally define a structured peer-to-peer 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 is constrained by 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 mapping each identifier i in the identifier space to a peer p in the system. We denote the identifiers of the peers in the system at a certain time $\mathcal{P}$ ($\mathcal{P} \subseteq \mathcal{I}$). If $\mathcal{P}$ is not given, we assume an arbitrary set of peers.

To make it concrete, we will now define a distance function, as well as a mapping from identifiers to peers. These definitions are similar to those commonly used in [10, 1, 5, 8]. However, our replication scheme does not assume these definitions and can thus be applied to a wide 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 in the system to its *successor*. The successor of each identifier i ($i \in \mathcal{I}$) is the first peer met in the identifier space going in clock-wise 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 if and only if $s_{\mathcal{P}}(i) = n$. 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. The function H is typically a hash function, though any other deterministic function can be used. Hence, a data item d is stored on the peer $s_{\mathcal{P}}(H(d))$. In this paper we usually say an item d to refer to $H(d)$, though the distinction will be clear from the context.

3 The Successor List Replication Scheme

In this section we analyze the successor-list replication scheme which is used by many systems[10, 1, 5].

The main idea behind the successor-list scheme is to take advantage of the way identifiers are mapped to peers. Since data items are stored on their successors, a failed peer's items automatically become the responsibility of its successor. By replicating all items D stored on a peer p at its f successors, a lookup to a failed peer can be forwarded to the failed peer's successor, which then is storing the replica. By having f replicas, even if f successive peers fail, there will be one peer left replicating the failed peers.

However, replicating on the successor-list to handle requests to failed peers assumes that there is a mechanism that updates the outdated routing information such that the requests are automatically forwarded to the failed peer's successor. For this reason, each peer stores routing information about its f successors.

To summarize, the successor-list scheme has two purposes. One is to replicate items on the successors such that lookups for items stored on a failed peer can be resolved by its successor. The other purpose is to store information about the successors such that outdated routing information can be corrected quickly.

Our claim is that these two things should be separated. While having routing information about the successors is advantageous for fault-tolerance, replication on the successor-list has many disadvantages.

The main disadvantage of the successor-list replication scheme is that the choice of peers replicating items is defined relative to the identifier of the neighboring peers. Hence, a peer interested in replicas of an item cannot find the identity of the peers storing the replicas without involving the master peer.

For example, assume there are a total of 128 identifiers in the system, and a peer with identifier 10 has the immediate successors with identifiers $15, 48, 49$. Another peer 0 interested in the replica of an item stored at peer 10 has no knowledge about the

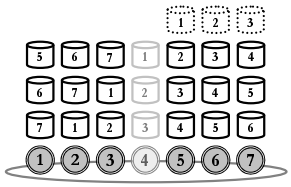

Fig. 1. A system populated with peers $1, ..., 7$, as indicated by the circles in the figure. The figure shows the identifier of the items each peer is storing given that the replication factor is 3. E.g. peer 1 is replicating items 5, 6, 7.

identifiers of 10's successors. To find the replicas of that item, peer 0 needs to route a message to the master peer 10 and ask it about its successors.

The inability to address the replicas independent from the master peer has consequences on the performance, complexity, and security of the system.

3.1 Performance and Complexity

From the performance viewpoint, a master peer might fail right after receiving a request. In such case, the whole operation is delayed since the failure must be detected and the routing information at some of the peers must be updated before the request can be repeated.

Furthermore, even if there are no failures, the master peer is a potential bottleneck hampering the positive effects of having replicas for load-balancing purposes.

From the complexity point of view, every leave or failure of a peer n triggers the transfer of items on each of the f successors of the master peer in order to preserve the replication factor. Figure 1 shows an example of this. 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.

Some of the peers involved in the process of restoring the replication degree might fail or leave the system too. Our experience with implementing the successor-list scheme in the initial *DKS* system taught us that complicated fault-tolerant algorithms are needed to handle such cases.

3.2 Security

From the security standpoint, the successor-list scheme poses serious security threats to the system as all requests to an item have to go trough its master peer.

Existing research has already shown security breaches in structured peer-to-peer systems. In [3] the Sybil attack is introduced, in which a malicious peer in the system assumes several identities. As a result, a malicious peer assumes the identity of the master peer of an item as well as the identity of all the replicas, enabling the malicious peer to

alone control all copies of an item. We believe that it is feasible to use external certificate authorities to generate unique peer identities once for every peer in the system.

However, even if the Sybil attack is resolved by the use of external certificate authorities, a malicious master peer can launch, what we call, a *Mendacity attack* by lying about who its successors are.

For example, if a master peer is asked about its successor-list, it can return a list of other malicious peers which it is cooperating with. The requesting peer can verify that the peers in the returned list are indeed peers with unique identities. But it cannot determine whether some other peers actually have identifiers which make them the true successors of the malicious peer.

Another possibility is to delegate the responsibility of directly accessing the replicas to the master peer. In that case, the master peer can maliciously tamper with the data according to its will. After tampering with the data, it can sign the data if needed. The requesting peer cannot tell that the data has been tampered with since it does not always know about the identity of a publisher of an item. Hence, it will verify that the publisher of the requested item indeed is a valid peer, but it cannot tell if it is a malicious peer.

A direct consequence of this is that traditional techniques such as distributed voting become ineffective as the peers voting might all be malicious.

4 The Symmetric Replication Scheme

The idea behind the symmetric replication is that 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 $r_1, ..., r_f$, then the identifier r_x, for $1 \leq x \leq f$, is associated with the identifiers $r_1, ..., r_f$ as well.

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.

We now explain how each identifier i is associated to f other identifiers. Let $\mathcal{F} = \{1, ..., f\}$, 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 ($r(0, 1) = 4$), 4 ($r(0, 2) = 4$), 8 ($r(0, 3) = 8$), 12 ($r(0, 4) = 12$).

So far we have only explained how each identifier is associated with a set of other identifiers, but mentioned nothing of how items are replicated. As we mentioned before, in a system without any replication, each item with identifier i is stored at its successor peer given by $s_{\mathcal{P}}(i)$. To replicate items in our scheme, the successor of the 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.

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 route a message to the peer responsible for any of the items 0, 4, 8, 12 and ask it about item 0.

For the symmetry requirement to always be true, it is required that the replication factor f divides the size of the identifier space N, i.e. $f|N$.

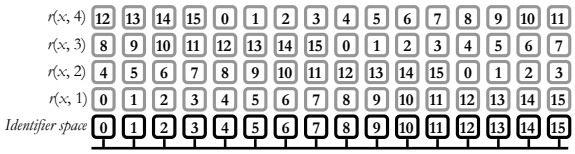

Fig. 2. The identifiers associated with each identifier in the system in a system with identifier space of size $N = 16$ and a replication factor of size $f = 4$.

We now give an informal description of all the proposed algorithms.

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 which immediately sends a RetrieveItems-message to its successor (denoted *succ*) asking it about all items it itself should be storing. The items it should be storing are specified by the 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 associated 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 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 space is required, this 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 find the responsible of an item. Instead, we assume that the sub-routine **FindSuccessor** implements a simple synchronous distributed algorithm which finds the peer responsible for a given identifier (See [10] for such an algorithm).

The implementation simply makes concurrent insertions to every location where the replica should be stored, though we do not show it here.

```
Subroutine :: JoinReplication
              send(n : succ : RetrieveItems(pred,n))

R1 :: receive(m : n : RetrieveItems(start, end))
      for r :=1 to f do
          items[r] := ∅
          i := start
          while i ≠end do
                i := i ⊕ 1
                items[r][i] := localHashTable[r][i]
          od
      od
      send(n : m : Replicate(items,start, end))

R2 :: receive(m : n : Replicate(items, start, end))
      for r :=1 to f do
          i := start
          while i ≠end do
                i := i ⊕ 1
                localHashTable[r][i] := items[r][i]
          od
      od

Subroutine :: LeaveReplication
              for r :=1 to f do
                  items[r] := ∅
                  i := pred
                  while i ≠n do
                        i := i ⊕ 1
                        items[r][i] := localHashTable[r][i]
                  od
              od
              send(n : succ : Replicate(items, pred, n))
```

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

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 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, and an integer specifying which of the f replicas to fetch items from. As more than one peer might be storing the items of the failed peer, a restricted version of our broadcast algorithm[4] is used. Assuming a uniform distribution of peer identifiers, the restricted broadcast needs to send a message to one peer on average for every failure.

5 Exploiting Symmetric Replication

In this section, we discuss techniques that can be implemented in the symmetric replication scheme but were impossible in the successor-list scheme.

From the performance and security standpoint, the symmetric replication scheme alone does not enhance the security of the system. It rather enables the use of various

```
R3 :: receive(m : n : InsertItem(key, value))
      for r :=1 to f do
          replicaKey := key ⊕ (r − 1) 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 ⊕(i − 1) N/f
              respNode := FindSuccessor(replicaKey)
              send(n : respNode : GetItem(replicaKey))

R5 :: receive(m : n : GetItem(key))
      send(n : m : GetItemResp(Key, localHashTable[r][key]))
```

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

```
Subroutine :: FailureReplication(failedId, predId, r)
              start := predId ⊕(r − 1) N/f
              end := failedId ⊕(r − 1) N/f
              respNode := FindSuccessor(start)
              send(n : respNode : RestrictedBroadcast(start, end, Msg(start, end, n)))

Subroutine :: MsgHandler(start, end, n'))
              for r :=1 to f do
                  items[r] := ∅
                  i := start
                  while i ≠end do
                          i := i ⊕ 1
                          items[r][i] := localHashTable[r][i]
                  od
              od
              send(n : n′ : Replicate(items,start, end))
```

Fig. 5. The replication algorithms for failures.

techniques that can modularly be implemented in an end-to-end fashion on-top of the existing peer-to-peer system.

For example, distributed voting can be 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 the majority vote.

By using distributed voting, the probability that an item has been tampered with can be calculated and reported to the requesting user or application. If the probability that the data in a response is tampered with 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 those g answers being tampered is given by the Bernoulli trials: $\binom{m}{g} p^g (1-p)^{m-g}$.

Another possibility is for the replica peers themselves to use a consensus algorithm, such as the Byzantine agreement, to agree about the correctness of a replica.

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 solves the problem of fault-tolerance since the failure of a peer along the path of a request does not require repeating the request as it is likely that another of one of the concurrent requests succeeds. We believe that the latter is very important. Otherwise, outgoing messages would have to be buffered at a peer together with timers, and whenever a timeout occurred, they would need to be repeated.

6 Extensions to Symmetric Replication

We outline two extensions to symmetric replication here: proximity neighbor selection and adaptive replication.

6.1 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 $\mathcal{DKS}$, 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.

6.2 Adaptive Replication

It is often not useful to have the same replication factor for every data item in the system as the popularity of different items vary, and hot-spots are common.

We believe that symmetric replication could be used as basic replication in the system. Peers that experience hot-spots could invite other peers to join under the same identifier to achieve adaptive replication on-top of the symmetric replication.

For example an overloaded peer p, could invite another peer p' to leave the system and join again with p's identifier. That would mean that two peers in the system would have the same identifier, and p could now tell requesting peers to redirect their requests to the other peer with the same identifier.

For this to work, peers with the same identifier need to keep a pointer to each-other in a local *replica table*. Upon a data insertion request the peers inform each-other such that

data consistency is preserved. If an event requires the modification of routing information at one of the replicas, that peer has to update all peers in its replica table.

The advantage of the adaptive replication approach is that the proper replication factor for an item would emerge dynamically depending on the popularity of items as well as the resources of the individual peers, rather than being fixed as a system parameter.

7 Related Work

Sit and Morris [9] give design guidelines for enhancing the security of structured peer-to-peer systems. Among other things, they identify the problem of a single point of responsibility for replication. They point out that several systems have this vulnerability. However, no solution is proposed.

In CAN [7], several instances of the overlay network, called realities, are used for replication. The main drawback of this approach is the overhead of replicating full instances of the whole peer-to-peer system. In particular, each peer has to maintain distinct routing tables for every reality, resulting in multitude overhead both in terms of bandwidth consumption and performance.

The use of several hashing functions for replication, which is mentioned in CAN [7], is closest to our symmetric replication scheme. However, it has several disadvantages. First, it requires the inverse of the hashing functions to maintain the replication factor, an impossible requirement. For example, if a peer responsible for some items fails it is desirable that a new responsible peer fetches those items from a replica such that the replication factor can be maintained. For instance, assume peer n has failed and it was storing items with identifiers $n_1, ..., n_\delta$. To retrieve those items from a replica we need to find the inverse image of $n_1, ..., n_\delta$ such that we can apply a different hashing function to it to obtain the identifiers of those items under another hashing function. Worse, even if the inverse of the hashing functions were available, the items that the failed peer was responsible for would be dispersed when using a different hashing function, making it necessary to fetch each item from a different peer[4].

Kademlia[6] always makes lookups to k replicas of an item concurrently. However, this prevents doing lookups to one random replica to achieve load-balancing, or to just route to the closest replica to achieve proximity or fewer overlay hops without burdening the system with many lookups to all replicas. This is not the case with our scheme. From the security stand-point, Kademlia stores items on the k closest peers, leaving the possibility of one neighboring peer to launch Mendacity attacks. Furthermore, our scheme is applicable to all structured peer-to-peer systems, which is not the case with Kademlia which relies on the XOR-metric as an underlying distance function.

8 Conclusions

We have analyzed the main approach used for replication in structured peer-to-peer systems and found that it has serious drawbacks both in terms of performance and security. These disadvantages are consequences of requests always having to go through the first peer storing a replica, leading to a bottleneck as well as a security breach.

[4] Disregarding collisions.

The first peer storing the replica is therefore a single-point of failure which can launch Mendacity attacks to gain control over all replicas of the items that it is storing.

To rectify the problems in the successor-list scheme, we proposed a new scheme and provided full algorithmic specifications of it. The scheme is applicable to all structured peer-to-peer systems. In our scheme, requests to the replicas do not have to pass through the same peer. As a result, Mendacity attacks are prevented while concurrent requests are made possible. Furthermore, the resulting algorithms are considerably simpler than in the successor-list scheme as fewer peers are involved in restoring the replication degree during dynamism.

Furthermore, we have shown different techniques that can be used on-top of our scheme to enhance the security, such as distributed voting or consensus algorithms. We also showed techniques for load-balancing requests as well as speeding up searches.

Finally, we outlined two general extensions to our scheme to achieve adaptive replication and proximity neighbor selection.

9 Acknowledgments

We would like to thank Erik Klintskog for giving positive feedback on this paper.

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 Workshop*, 2004.
3. J. Douceur. The Sybil Attack. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.
4. 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.
5. 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.
6. P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR metric. In *The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02)*, 2002.
7. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. Technical Report TR-00-010, Berkeley, CA, 2000.
8. 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.
9. 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.
10. 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.