Low-Bandwidth Topology Maintenance for Robustness in Structured Overlay Networks

Ali Ghodsi[*†], Luc Onana Alima[†*], Seif Haridi[*†]
[*]IMIT-Royal Institute of Technology, Electrum 229 * 164 40 Kista, Sweden
Email: {aligh, seif}@imit.kth.se
[†]Swedish Institute of Computer Science, Box 1263, SE-164 29, Kista, Sweden
Email: onana@sics.se

***Abstract*— Structured peer-to-peer systems have emerged as infrastructures for resource sharing in large-scale, distributed, and dynamic environments. One challenge in these systems is to efficiently maintain routing information in the presence of nodes joining, leaving, and failing. Many systems use costly periodic stabilization protocols to ensure that the routing information is up-to-date.**

In this paper, we present a novel technique called correction-on-change, which identifies and notifies all nodes that have outdated routing information as a result of a node joining, leaving, or failing. Effective failure handling is simplified as the detection of a failure triggers a correction-on-change which updates all the nodes that have a pointer to the failed node. The resulting system has increased robustness as nodes with stale routing information are immediately updated.

We proof the correctness of the algorithms and evaluate its performance by means of simulation. Experimental results show that for the same amount of maintenance bandwidth correction-on-change makes the system by far more robust when compared to periodic stabilization. Moreover, compared to adaptive stabilization which adjusts its frequency to the dynamism in the system, correction-on-change gives the same performance but with considerably less maintenance bandwidth. As correction-on-change immediately updates incorrect routing entries the average lookup length is maintained close to the theoretical average in the presence of high dynamism. We show how the technique can be applied to our $\mathcal{DKS}$ system as well as the Chord system.

I. Introduction

In the recent years, a number of structured peer-to-peer systems have been proposed that provide a routing infrastructure over the existing Internet. This is done by letting each participating node maintain routing information about a subset of the other nodes in the system. The nodes are chosen such that given the logical identifier of a node a message can be routed to that node in few routing hops. The number of hops required to find a node is typically in logarithmic order of the number of nodes in the system. The size of the routing information stored at each node is typically in logarithmic order or constant of the number of nodes in the system. On top of this routing infrastructure basic services, such as a distributed hash table, can be built.

A challenge in these systems is to maintain the routing information, at a low cost of communication, when the set of participating nodes is altered due to nodes joining, leaving and failing. A trade-off between robustness and bandwidth consumption has to be made. If the routing information is not maintained frequently enough the system will not be robust as the routing information becomes outdated quickly. On the other hand, if routing information is maintained too often bandwidth consumption will be high.

In this paper, we present a novel technique called *correction-on-change*[1], which identifies and notifies all necessary nodes upon the join, leave, or failure of a node. Failure handling is simplified as the detection of a failure triggers a correction-on-change which updates all the nodes which have a pointer to the failed node. The resulting system is highly robust while it avoids unnecessary bandwidth consumption.

A. Motivation

Most of the existing structured P2P systems use costly periodic stabilization protocols to ensure that the routing information is up-to-date[2][3][4][5].

The main disadvantage of this approach is that it induces a high bandwidth consumption. Indeed, at steady periods when the dynamism[2] in the system is low, unnecessary bandwidth is consumed by periodic stabilization.

One solution as suggested in [6] is that the stabilization frequency should be dynamically adapted to the dynamism in the system. However, this approach requires knowledge about global parameters, such as the system size and the amount of dynamism in the system. As this information is not locally available, the authors suggest that it should be estimated. However, the accuracy of the estimations are not well understood. Furthermore, even with accurate estimation of the amount of dynamism and the system size, periodic stabilization consumes far more bandwidth as we show in Section IV.

In [7] we proposed a technique called *correction-on-use* that embeds parameters in routing messages such that incorrect routing information is corrected on-the-fly without the use of periodic stabilization.

The advantage of correction-on-use is that it consumes less bandwidth than periodic stabilization. However, correction-on-

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.

[1]We note that term correction-on-change was independently introduced in [1].

[2]The number of joins, leaves and failures.

use assumes that the ratio between the number of routing messages to the dynamism in the system is high enough such that there are enough routing messages to correct the routing information that is invalidated as the result of dynamism. Consequently, routing information will become outdated if this ratio is low. Hence, a the performance will be poor since a routing hop might lead to a failed node.

Furthermore, the amount of lookups needed at each node for correction-on-use to be effective is dependent on the total number of nodes in the system. A property that not necessarily reflects the behavior of the users in the system.

Our main goal in this paper is to provide maintenance techniques that allow the system to automatically adapt to the dynamism while avoiding unnecessary bandwidth consumption. Following our correction-on-use philosophy, we want to achieve this goal without any assumptions on the amount of routing messages in the system. Instead, the maintenance technique should allow the system to self-adapt to the actual amount of dynamism in the system.

B. Contributions

In this paper we show a general approach, given only partial information about the system, to deterministically identify all the nodes that should be notified upon a join, leave or failure event. Furthermore we prove that the identified nodes are indeed all the nodes that need to be corrected. We also prove that no other nodes in the system need to be notified of the change. Thereafter we show how the broadcast algorithm proposed in [8] can be used to in-parallel notify the identified nodes. We adapt the broadcast algorithm to correction-on-change by letting it broadcast only to the nodes that need to be updated, rather than broadcasting to all nodes in the system. Hence, the broadcast algorithm notifies the nodes in $O(\log \mathcal{D})$ steps, where $\mathcal{D}$ is the number of nodes that need to be notified. Furthermore, we improve the broadcast algorithm such that it never sends redundant messages, even when the routing information in the system is incorrect. We also give an algorithm for optimizing correction-on-change to send fewer messages.

Our approach is general enough to be applied to several Chord-based [9][7] structured peer-to-peer systems. However, it cannot be applied to peer-to-peer systems where there is non-determinism in the choice of routing neighbors [5][4]. For simplicity we demonstrate how correction-on-change can be applied to $\mathcal{DKS}$, as the topology of Chord can be derived from $\mathcal{DKS}$ when the structuring parameter $k = 2$ in $\mathcal{DKS}$.

C. Related Work

In one of the early Chord papers [9] the authors mention that upon a join of a node j an event notification should be routed to the first node in the vicinity of the nodes pointing to j [3]. Thereafter the notification message should be sent sequentially on the ring as long as it finds predecessors that should be pointing to the node j. The drawback of this scheme is that it goes sequentially on the predecessor chain, as the authors fail to properly identify all the nodes that need to be notified. This has the consequence that it takes a long time and may terminate due to the failure of a single node in the predecessor chain. Furthermore, the algorithm can terminate prematurely due to other reasons. For instance, this can happen if a node in the predecessor chain is pointing to a failed node or if a newly joined node that already has correct routing information is met in the predecessor chain. The first case will be common as Chord treats voluntary leaves as failures. Consequently, the authors suggest that the notification should be run concurrently with a periodic stabilization protocol.

[3]This approach was abandoned and does not appear in the other Chord papers.

In [1][10] several reactive routing maintenance techniques are proposed. The main idea is to correct outdated routing information lazily when errors are detected during system use. Our approach takes this one step further by updating the outdated entries of all nodes eagerly whenever a change is detected.

Many attempts have been made to provide symmetric structured P2P systems [11][12][13], where a routing pointer from a node i to a node j implies a routing pointer from node j to node i for all nodes $i \neq j$. In such a symmetric P2P system correction-on-change could be implemented in a straightforward manner. However, the aforementioned systems do not satisfy the symmetry requirement except in the rare case when the number of nodes in the system happens to be equal to the number of nodes in the overlay graph.

In [11] a system is proposed where each node keeps a pointer to the nodes with logical identifiers at hamming distances of one. But this system cannot guarantee full symmetry neither. Therefore, the authors take the same approach as Chord and their system consequently has to concurrently run a periodic stabilization protocol.

Another possible solution would be to make the system symmetric in a dynamic fashion by letting each node establish a routing pointer back to every node that is pointing to it. For example connection-oriented communication such as TCP/IP could be exploited to find all the incoming connections for every node. The drawback of this approach is that it cannot be used for fault-tolerance because if a node fails, the node that detects it does not know which other nodes are still pointing to the failed node. Furthermore, it is not clear how a newly joined node can notify the existing nodes that they should point to it. We see however that this technique can be used to optimize our approach.

II. $\mathcal{DKS}$ OVERVIEW

In the following section we briefly present the assumptions on the underlying network and describe the notation for presenting algorithms. Thereafter we present the routing topology of the $\mathcal{DKS}$ system and the correction-on-use technique used to lazily maintain the routing information.

A. Model of Distributed Systems

We assume a distributed system modeled by a set of nodes communicating by message passing through a communication network that is: (i) Connected, (ii) Asynchronous, (iii) Reliable, and (iv) providing FIFO communication.

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

$$R \;::\; \frac{\textbf{receive}(\textit{Sender, Receiver, }\textsc{Message}(\textit{arg}_1, .., \textit{arg}_n))}{\text{Action}}$$

The rule R describes the event of receiving a message MESSAGE at node *Receiver*, from *Sender*, 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*.

B. Structure of the $\mathcal{DKS}$

In this sub-section we present the $\mathcal{DKS}$ system and the rest of the paper will assume the structure defined here.

An instance of the $\mathcal{DKS}$ system is configured with a number of parameters. In this paper, we will only report those that are necessary to make this paper self-contained. One of the parameters is the *structuring parameter* k ($k \in \mathbb{N}\backslash\{0,1\}$). With k defined, the maximum number of nodes that can be simultaneously in a $\mathcal{DKS}$ network is chosen to be $N = k^L$, for some large positive integer L. Given N, and k the lookup length is guaranteed to take at most $L = \log_k(N)$ hops for a network of maximum size N during normal system operation. Every node knows k and N, and can therefore compute L.

Once N has been defined, all nodes in the system are mapped onto the identifier space, $\mathcal{I}= \{0, ..., N-1\}$. The identifier space is a circular space modulo N.

We shall use the infix notation for the binary operator $\oplus : \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ defined as $a \oplus b = (a+b)$ modulo N. The binary operator $\ominus : \mathcal{I} \times \mathcal{I} \rightarrow \mathcal{I}$ is defined similarly as $a \ominus b = (a-b)$ modulo N.

The whole identifier space can be represented by an interval of the form $[x, x[$ or $]x, x]$ for an arbitrary $x \in \mathcal{I}$. For any $x \in \mathcal{I}$, we note that $[x, x] = \{x\}$ and $]x, x[= \mathcal{I}\backslash\{x\}$. Generally, $]x, y]$ denotes the interval $x \oplus 1, \cdots, (x \oplus y \ominus x) = y$ for any $x, y \in \mathcal{I}$. Similarly $[x, y[$ denotes the interval $x, x \oplus 1, \cdots, y \ominus 1$ for any $x, y \in \mathcal{I}$.

Each node in a $\mathcal{DKS}$ overlay network maintains three main components: a *routing table*, a *BackList* and a *FrontList*. We describe these components in the following sub-section.

C. Routing tables

Each node maintains a routing table which consists of $L = \log_k(N)$ *levels*. Let $\mathcal{L} = \{1, 2, .., L\}$ be the set of levels.

At any level, $l \in \mathcal{L}$, a node n has a view of the identifier space defined by the function $V_n : \mathcal{L} \rightarrow 2^{\mathcal{I}}$ as:

$$V_n(l) = [n, n \oplus k^{L-l+1}[$$

This means that for level one, the view consists of the whole identifier space, and at any other level $l > 1$, the view consists of one k:th of the identifier space considered by the view $V_n(l-1)$.

At any level $l \in \mathcal{L}$, the view $V_n(l)$ is partitioned into k equally-sized intervals. Let $\mathcal{K} = \{0, \cdots, k-1\}$. At a node n, the intervals are defined by the function $I_n : \mathcal{L} \times \mathcal{K} \rightarrow 2^{\mathcal{I}}$ as:

$$I_n(l, i) = [n \oplus ik^{L-l}, n \oplus (i+1)k^{L-l}[$$

For simplicity we will let $BI_n : \mathcal{L} \times \mathcal{K} \rightarrow \mathcal{I}$ defined as $BI_n(l, i) = n \oplus ik^{L-l}$ denote the beginning of the interval $I_n(l, i)$.

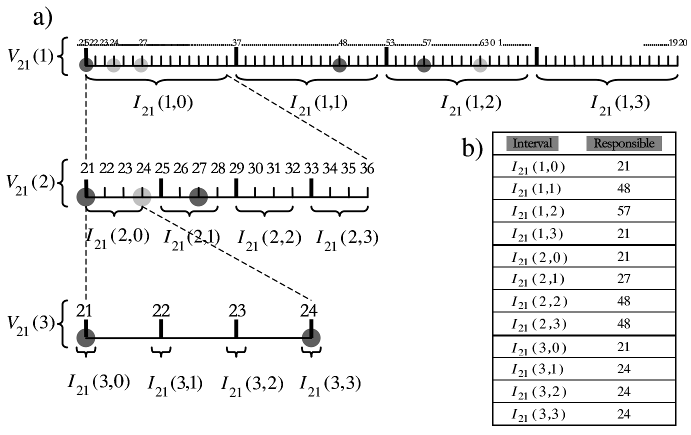
The aforementioned intervals and views are defined statically at each node independently of the rest of the nodes in the system. However, to assign parts of the identifier space to the participating nodes in the system each node needs to map its intervals to some of the nodes in the system. We denote the identifiers of the nodes in the system by the set $\mathcal{P}$ ($\mathcal{P} \subseteq \mathcal{I}$). Each node, n, maintains a responsible node for every interval in its routing table. For any level, $l \in \mathcal{L}$ the responsible for interval $I_n(l, 0)$ is always n itself. [4] For all other intervals $m \in \mathcal{K}\backslash\{0\}$, the responsible for interval $I_n(l, m)$ is chosen to be the first node encountered, moving in clockwise direction, starting at the beginning of the interval.

Formally, the routing table of a node n is given by the function $RT_{n,\mathcal{P}} : \mathcal{L} \times \mathcal{K} \rightarrow \mathcal{P}$:

$$RT_{n,\mathcal{P}}(l, i) = BI_n(l, i) \oplus \min(\{p \ominus BI_n(l, i) \mid p \in \mathcal{P}\})$$

In addition to storing a routing table, each node, n, maintains a predecessor pointer referred to as $Pred_{n,\mathcal{P}}$ which is the first node encountered, moving in counter-clockwise direction, starting at n. Formally, $Pred_{n,\mathcal{P}} = n \oplus \max(\{p \ominus n \mid p \in \mathcal{P}\})$. Similarly we define the successor of node n to be $Succ_{n,\mathcal{P}} = RT_{n,\mathcal{P}}(L, 1)$.

Figure 1 shows an example of a $\mathcal{DKS}$ network ($k = 4$, $N = 64$, $\mathcal{P} = \{21, 24, 27, 48, 57, 63\}$) from the perspective of the node 21. Note that in figure 1 we have mapped the modulo N circle onto a line from node 21's view.



Interval	Responsible
$I_{21}(1,0)$	21
$I_{21}(1,1)$	48
$I_{21}(1,2)$	57
$I_{21}(1,3)$	21
$I_{21}(2,0)$	21
$I_{21}(2,1)$	27
$I_{21}(2,2)$	48
$I_{21}(2,3)$	48
$I_{21}(3,0)$	21
$I_{21}(3,1)$	24
$I_{21}(3,2)$	24
$I_{21}(3,3)$	24

Fig. 1. a) A $\mathcal{DKS}$ network ($k = 4$, $N = 64$, $\mathcal{P} = \{21, 24, 27, 48, 57, 63\}$). The figure shows node 21's views, $V_{21}(1)$, $V_{21}(2)$ and $V_{21}(3)$, and how each view is partitioned into $k = 4$ equally sized intervals. The dark nodes represent the responsible nodes from node 21's view. b) Node 21's routing table showing each interval and its responsible node.

[4]The responsible node's identifier and network address is stored such that communication can be established with it.

When setting up a $\mathcal{DKS}$ network, another parameter that needs to be instantiated is the fault-tolerance parameter, denoted f. Each node in a $\mathcal{DKS}$ network maintains two lists of references, each containing at most $f + 1$ references to other $\mathcal{DKS}$ nodes. These lists are named *FrontList* and the *BackList*. The *FrontList* contains the $f + 1$ nodes in the system with closest identifiers in clockwise direction, while the *BackList* contains the $f + 1$ nodes in the system with closest identifiers in anti-clockwise direction. For short, we will use Fl_n (resp. Bl_n) to denote the *FrontList* (resp. *BackList*) at a node n. In the $\mathcal{DKS}$ system these lists are maintained by using a technique similar to correction-on-change together with vector-timestamps to ensure causality. However, we will not show how these pointers are maintained in this paper, rather report the results of that in a different paper.

D. Legitimate state

The set of nodes $\mathcal{P}$ is typically changing over time as nodes join, leave, and fail. Therefore, the routing tables of the nodes might become incorrect. To cope with this dynamism, the $\mathcal{DKS}$ is designed with *stabilization* in mind. However the stabilization in a $\mathcal{DKS}$ is achieved at very low cost of communication.

As for any stabilizing system, we need to characterize the set of legitimate (global) states of a $\mathcal{DKS}$ network. The intuition behind the legitimate state is to model the state when all the routing information in the system is fully correct. The system might often not be in a legitimate state. However, the idea is that it will always eventually converge to the legitimate state.

Definition 2.1: We will say that a $\mathcal{DKS}$ system populated with $\mathcal{P}$ nodes is in a legitimate state iff:

$$\big(\forall n, l, i : n \in \mathcal{P} \wedge l \in \mathcal{L} \wedge i \in \mathcal{K} : \\ (RT_{n,\ \mathcal{P}}(l, i) = BI_n(l, i) \oplus \min(\{p \ominus BI_n(l, i) \mid p \in \mathcal{P}\}))\big)$$

We say that the system is in a illegitimate state if it is not in a legitimate state.

The $\mathcal{DKS}$ system keeps the successor and predecessor pointers up-to-date by executing a local atomic action whenever a node joins or leaves the system.

This is however not enough to ensure that the system is always converging towards the legitimate state. Therefore the $\mathcal{DKS}$ system uses the correction-on-use technique.

E. Correction-on-use

In a $\mathcal{DKS}$ network, routing information can become outdated as a result of joining, leaving, or failing nodes. Figure 2 shows how routing entries become outdated as a result of a join operation. The outdated routing entries are corrected only when they are used. As long as the ratio of lookups to joins, leaves, and failures is high, the routing information is eventually corrected. This is the essential assumption in $\mathcal{DKS}$, which is validated in [7].

Correction-on-use is based on two ideas. The first idea is to embed the level l and the interval i parameters in every routing

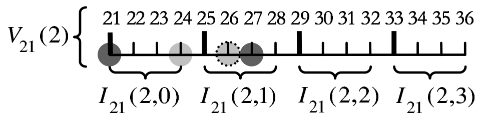


Fig. 2. A node with identifier 26 joins the $\mathcal{DKS}$ network ($k = 4$, $N = 64$) and informs its immediate neighbors 24 and 27 about its existence. However, node 21 does not know about node 26 and considers node 27 responsible for I_1^2. If node 21 happens to send a lookup message to node 27, node 27 will inform node 21 about node 26 and trigger a correction-on-use. Alternatively, node 21 can become aware of node 26 if it receives a message originating from node 26.

message [5]. A node n receiving a routing message from a node n' can then calculate the start of the interval, $I_{n'}(l, i)$ at node n', for which n is responsible according to n'. If n's predecessor is in the interval $[n' \oplus ik^{L-l}, n[$, then node n sends a BADPOINTER message to node n'. This BADPOINTER message carries the candidate node, denoted *cand*, to which n' should be pointing to according to node n. To determine the candidate node to be sent to n' for a correction, node n calls the sub-routine `BestCandidate`(n', msg), which takes n' and *msg* as parameters. A call of `BestCandidate`(n', msg) at node n, returns the node from the Bl_n to which node n' should be pointing to at its level *msg*.l and for the interval *msg*.i of that level, according to n. The formal description of the sub-routine `BestCandidate` is straightforward. We do not present it in this paper. Furthermore, for simplicity of presentation, we will assume that the sub-routine `BestCandidate` returns the predecessor of the calling node. When node n' receives a BADPOINTER(*cand*) message, node n' updates its routing information.

The second idea is that a message sent by a node p to another node n is an indication of p's existence. Hence, upon receipt of a message from a node p, node n examines all of its intervals to determine if p should be responsible for any of the intervals, in which case routing information is updated.

Figure 3 shows the algorithm for correction-on-use at a node n. Upon the receipt of any routing message from a node n' the sub-routine `CorrectionOnUse` is immediately called with message (*msg*) supplied as a parameter. Recall that the destination identifier, level, and interval parameters are embedded in the message. These embedded parameters are accessed by the fields, l, i, and d respectively. If the sub-routine returns *false* the message should be consumed by node n, i.e. the receiving node should process the request. If the sub-routine returns *true* the message should have been sent to another node and should hence be ignored.

III. CORRECTION-ON-CHANGE

In the following sub-sections the correction-on-change technique will be explained. The first sub-section explains how all the nodes pointing to a given node in a legitimate system state can be deterministically identified. The following sub-section

[5] A message from node n' to node n is considered to be a routing message if n is responsible for an interval in the routing table of n'

```
$R1$ :: receive($n'$, $n$, BADPOINTER(cand))
        ------------------------------------
        AdaptTo(cand)

Subroutine :: CorrectionOnUse(msg)
    AdaptTo($n'$)
    if $(n'\oplus(msg.i)k^{L-msg.l}) \in ]n', pred]$ then
        cand := BestCandidate($n'$, msg)
        send($n$, $n'$, BADPOINTER(cand))
        if $d \in [n', n[$ then
            send($n$, cand, msg)
            return true
        fi
    fi
    return false

Subroutine :: AdaptTo(cand)
    for $\lambda$ := 1 to $log_k(N)$ do
        for $\tau$ := $k-1$ downto 1 do
            if $(n\oplus\tau k^{L-\lambda}) \in ]n, cand]$ and
               $RT(\lambda,\tau)\in ]cand, n]$ then
               $RT(\lambda,\tau) = cand$
            fi
        od
    od
```

Fig. 3. Correction-on-use

gives the algorithms to identify and notify nodes using an improved restricted version of our previous broadcast algorithm. Finally we provide algorithms and discuss the intricate details of making the algorithm work in an asynchronous setting.

A. Identifying the dependent nodes

The aim of this section is to solve the following problem: given a node with identifier n ($n \in \mathcal{I}$) in a legitimate system state efficiently identify all the identifiers of the nodes that are pointing to n. We refer to node n as the *subjective node* and the identified node identifiers the *dependent nodes*. In a fully populated system the set of dependent node identifiers for a node n is:

$$\{n' \in \mathcal{I} \mid n' \oplus ik^{L-l} = n, 1 \leq l \leq \log_k(N), 1 \leq i < k\}$$

Assume a fully populated $\mathcal{DKS}$ system in a legitimate state with identifier space $\mathcal{I} = \{0, 1, .., k^L - 1\}$ of size k^L for some large L and some $k \geq 2$. The following formula can be used to identify all the dependent nodes for the subjective node s ($s \in \mathcal{I}$) : $n = s \ominus ik^{L-l}$, for all l ($1 \leq l \leq L$), and i ($1 \leq i < k$).

However, the system is typically not fully populated. Therefore the above formula does not identify the dependent nodes in a non-fully populated system. There might be other nodes, than those deduced by the above formula, which have the subjective node as responsible for an interval. The reason for this is that a node can be responsible for several consecutive intervals.

We want to be able to identify all the dependent nodes for a subjective node n in a system with the nodes $\mathcal{P} \subseteq \mathcal{I}$:

$$\{p \in \mathcal{P} \mid RT_p(l, i) = n, 1 \leq l \leq \log_k(N), 1 \leq i < k\}$$

Lemma 3.1: Assume a $\mathcal{DKS}$ system in a legitimate state with identifier space $\mathcal{I} = \{0, 1, .., k^L - 1\}$ of size k^L for some large L, and some $k \geq 2$ and the nodes $\mathcal{P}$. The following holds for a node, $n' \in \mathcal{I}$, and $p = Pred_{n', \mathcal{P}}$:
$BI_n(l, i) \in]p, n'] \Leftrightarrow RT_{n,\mathcal{P}}(l, i) = n'$.

Proof: Since $]p, n'] \cap \mathcal{P} = \{n'\}$, the right implication is proved as follows:
$BI_n(l, i) \in]p, n']$
$\Rightarrow\ BI_n(l, i) \oplus \min(\{p \ominus BI_n(l, i) \mid p \in \mathcal{P}\}) = n'$
$\Rightarrow\ RT_{n,\mathcal{P}}(l, i) = n'$

We prove the left implication by proving the contra-position:

$$BI_n(l, i) \notin]p, n'] \Rightarrow RT_{n,\mathcal{P}}(l, i) \neq n'$$

We will use the following rule in the derivations:

$$\forall(S \subseteq \mathbb{N}) : (\min(\{0\} \cup S) = 0) \qquad (1)$$

Assume:
$BI_n(l, i) \notin]p, n']$
$\Rightarrow BI_n(l, i) \in]n', p]$
$\Rightarrow RT_{n,\{p,n'\}}(l, i) \in]n' \oplus \min(\{p \ominus n', n' \ominus n'\}), p \oplus \min(\{p \ominus p, n' \ominus p\})]$
$\Rightarrow RT_{n,\{p,n'\} \cup \mathcal{P}}(l, i) \in]n' \oplus 0, p \oplus 0]$ by Rule (1)
$\Rightarrow RT_{n,\mathcal{P}}(l, i) \in]n', p]$

Hence we have that $\Rightarrow RT_{n,\mathcal{P}}(l, i) \neq n'$

■

The following lemma identifies the set of nodes that have an interval beginning in $]p, n']$.

Lemma 3.2: Assume a $\mathcal{DKS}$ system in a legitimate state with identifier space $\mathcal{I} = \{0, 1, .., k^L - 1\}$ of size k^L for some large L, and some $k \geq 2$ and at least three nodes. Let $\mathcal{D}^l_i(n', p) =]p \ominus ik^{L-l}, n' \ominus ik^{L-l}]$. Given a node n', with the predecessor p, the following holds for any $n \neq n' \neq p$:

$$BI_n(l, i) \in]p, n'] \Leftrightarrow n \in \mathcal{D}^l_i(n', p)$$

Proof: We will prove the logical equivalence in two steps. First we will show that
$BI_n(l, i) \in]p, n'] \Rightarrow n \in \mathcal{D}^l_i(n', p)$
Assume:
$BI_n(l, i) \in]p, n']$
$\Rightarrow\ (n \oplus ik^{L-l}) \in]p, n']$
$\Rightarrow\ (n \oplus ik^{L-l} \ominus ik^{L-l}) \in]p \ominus ik^{L-l}, n' \ominus ik^{L-l}]$
$\Rightarrow\ n \in]p \ominus ik^{L-l}, n' \ominus ik^{L-l}]$
$\Rightarrow\ n \in \mathcal{D}^l_i(n', p)$

Now we will prove:
$n \in \mathcal{D}^l_i(n', p) \Rightarrow BI_n(l, i) \in]p, n']$
Assume:
$n \in \mathcal{D}^l_i(n', p)$
$\Rightarrow\ n \in]p \ominus ik^{L-l}, n' \ominus ik^{L-l}]$
$\Rightarrow\ n \oplus ik^{L-l} \in]p \ominus ik^{L-l} \oplus ik^{L-l}, n' \ominus ik^{L-l} \oplus ik^{L-l}]$
$\Rightarrow\ n \oplus ik^{L-l} \in]p, n']$
$\Rightarrow\ BI_n(l, i) \in]p, n']$

■

Theorem 3.3: In a $\mathcal{DKS}$ system in a legitimate state with the identifier space $\mathcal{I} = \{0, 1, .., k^L - 1\}$ of size k^L for some large L, and some $k \geq 2$ and at least three nodes. Given a node n', with the predecessor p, the following holds for any $n \neq n' \neq p$:

$$n \in \mathcal{D}_i^l(n', p) \Leftrightarrow RT_{n,\mathcal{P}}(l, i) = n'$$

Proof: The proof follows from Lemma 3.1 and Lemma 3.2. ■

Theorem 3.3 uniquely identifies for each subjective node n', and predecessor $p = Pred_{n',\mathcal{P}}$ the dependent node identifiers $\mathcal{D}_i^l(n', p)$ ($1 \leq l \leq L$, $1 \leq i < k$). Hence any dependent node n must have an identifier in $\mathcal{D}_i^l(n', p)$. In the next section we use our restricted broadcast algorithm to broadcast to all nodes in $\mathcal{D}_i^l(n', p)$ whenever a node n' joins, leaves, or fails.

B. Notifications

In this section we show how a notification is sent to the dependent nodes identified in the previous section whenever a node joins, leaves, or fails. The purpose of the notification is to trigger correction of outdated routing information at the dependent nodes.

The notification algorithm is given in Figure 4. The algorithm makes a lookup to find the first node in every interval in the dependent set. This is done by calling `FindSuccessor` with a parameter specifying the start of each interval containing dependent nodes. Thereafter a restricted broadcast is initiated for every interval. Each restricted broadcast serves to notify the dependent nodes in a given interval determined by the *start* and *limit* parameters computed in the algorithm given in Figure 4. The *msg* parameter carries the information about the actual event that triggered correction-on-change, i.e. a join, leave, or failure event. In addition, the parameters *start* and *lim* are embedded in the message to specify the interval being broadcast to.

The algorithm proposed in [8] guarantees that all nodes present in the system at the time of the broadcast operation receive the broadcast message given that they do not leave the system or fail. Moreover, any node that receives a broadcast message receives it only once, disregarding messages sent through erroneous pointers as they will trigger correction-on-use.

In this paper we have improved the broadcast algorithm such that redundant messages are never sent even if the routing pointers are outdated. Furthermore, we use a restricted version of the broadcast algorithm that only broadcasts to the dependent nodes, rather than broadcasting to all the P nodes in the system.

Assuming uniform distribution of the node identifiers the expected number of nodes that will be in the dependent set are $(k-1)\log_k(P)$, where P is the total number of nodes in the system. Given that the system is in a legitimate state, it takes at most $\log_k(d)$ hops to resolve a query from a node i to a target identifier j, where $j \ominus i = d$. The total number of messages needed to notify all dependent nodes will at most be:

$$\sum_{l=0}^{\log_k(P)-1} \sum_{i=1}^{k-1} \log_k(P - ik^l) < (k-1)\log_k^2(P)$$

As the broadcast algorithm notifies the dependent nodes in-parallel, it will cover all the $\mathcal{D}$ nodes in an interval in $\theta(\log(\mathcal{D}))$ time units, assuming that the transmission of a message from a node to its neighbor takes one time unit.

```
Subroutine :: Notify(Msg)
    for $l := 1$ to $log_k(N)$ do
        for $i := 1$ to $k - 1$ do
            start:= $p \ominus ik^{L-l}$
            limit:= $n \ominus ik^{L-l}$
            firstNode:=FindSuccessor(start)
            msg.start:=start
            msg.lim:=limit
            send($n$, firstNode, BROADCAST(msg))
        od
    od
```

Fig. 4. Notification Algorithm

```
$R2_1$ :: receive($n'$, $n$, BROADCAST(msg))
    %% Deliver the message to the application layer
    s = msg.start
    l = msg.lim
    p = predecessor
    if $p \in [s, n[$ then
        msg.lim = $n$
        send($n$, $p$, BROADCAST(msg))
        send($n$, $n'$, BADPOINTER(p))
    fi
    for $\lambda := 1$ to $log_k(N)$ do
        for $\tau := k - 1$ downto $1$ do
            if $RT(\lambda, \tau) \in ]n, l[$ then
                msg.start = $n \oplus \tau k^{L-\lambda}$
                send($n$, $RT(\lambda, \tau)$, BROADCAST(msg))
                $l := n \oplus \tau k^{L-\lambda}$
            fi
        od
    od
```

Fig. 5. Restricted Broadcast Algorithm

The restricted broadcast algorithm shown in Figure 5 will be used to notify dependent nodes. Unfortunately, intervals might overlap in certain settings of $\mathcal{P}$. As a consequence, some dependent nodes might receive notification messages more than once. To avoid such unnecessary messages, we propose an optimized algorithm in which overlapping intervals are collapsed.

Figure 6 shows the algorithm for collapsing overlapping intervals. The sub-routine `CollapseIntervals` is called with the parameter *intervalList*. *intervalList* is a list containing pairs of integers in the form (*start*, *end*), where *start* represents the start of an interval and *end* the end of an interval. For example, (6, 9) represents the interval $]6, 9]$. When `CollapseIntervals` has computed it returns a list of intervals, where overlapping intervals have been merged and redundant intervals have been removed.

The sub-routine `CollapseIntervals` works by maintaining the invariant that only intervals that are disjoint from every other interval are put on a list denoted *finalList*. Conversely, intervals that are not yet determined to be disjoint from every other interval are kept in a list denoted *workList*. The sub-routine initializes *workList* to contain all intervals and *finalList* to be empty. The algorithm works by moving elements from *workList* over to *finalList* on each iteration. When *workList* is empty the algorithm terminates.

Theorem 3.4: The sub-routine `CollapseIntervals` always terminates.

Proof: To show that the algorithm always terminates, each iteration of `CollapseIntervals` either merges some intervals such that the size of *workList* is decreased by one or it moves one element over to *finalList*. Hence, the algorithm terminates in a finite number of iterations given that the input list is finite.

The sub-routine `Iterate` is used in each iteration of `CollapseIntervals` with two parameters; an interval i from *workList* and the rest of the intervals from *workList* denoted *workListRest*. Each time `Iterate` is called it does one of three things. Either `Iterate` collapses i into another interval in *workListRest* or it just removes i as i is a subset of another interval in *workListRest*. In both cases i is subsumed in *workListRest* and `Iterate` returns with a *dirty* flag set to true indicating that *workList* should be replaced by *workListRest*. Alternatively, i is disjoint from every other interval in *workListRest* in which case `Iterate` simply returns with a *dirty* flag set to false and `CollapseIntervals` can move i to *finalList*. ■

```
Subroutine :: Iterate(i, workListRest)
        (start, end) = i
        tempList = empty
        dirty = false
        foreach inter in workListRest do
              (is,ie) = inter
              if $start \notin ]is, ie]$ and $end \notin ]is, ie]$ then
                   tempList := Append(tempList, (is,ie))
              elseif $start \notin ]is, ie]$ and $end \in ]is, ie]$ then
                   dirty := true
                   tempList := Append(tempList, (start,ie))
              elseif $start \in ]is, ie]$ and $end \notin ]is, ie]$ then
                   dirty := true
                   tempList := Append(tempList, (is,end))
              elseif $start \in ]is, ie]$ and $end \in ]is, ie]$ then
                   dirty:=true
                   tempList := Append(tempList, (is,ie))
              end
        end
        return (tempList,dirty)

Subroutine :: CollapseIntervals(intervalList)
        workList = intervalList
        finalList = empty
        while workList != empty do
             if workList.length == 1 then
               finalList := AppendLists(workList, finalList)
               workList := empty
             else
                (start,end) := Head(workList)
                workList.DeleteHead()
                (workList, dirty) = Iterate(start, end, workList)
                if dirty == false then
                   finalList := Append(finalList,(start,end))
                end
             end
        end
        return finalList
```

Fig. 6. Algorithm for collapsing overlapping intervals

C. Using correction-on-change in a dynamic setting

A challenge with designing algorithms using correction-on-change is to make it work in a dynamic setting where several instances of correction-on-change might be interleaved in an asynchronous setting.

Our first concern is that there is no guarantee on the order in which messages arrive to the destination, even with the FIFO requirement, as messages are routed via inter-mediate nodes in the overlay. We have studied all the different inter-leavings of joins, leaves, and failures and designed the algorithms using correction-on-change such that messages arriving out of order will not render the system in an illegitimate state.

For example, one node might join and shortly thereafter leave the system. In an ideal situation the nodes would first be notified about the joining of the node, and thereafter they would be notified of the departure. But as there is no guarantee in the order which messages arrive, some nodes might first find out about the departure of the node, and thereafter find out about the arrival of the node leaving many nodes with pointers to a node that is no longer in the system.

We solve the aforementioned problem by attaching a local time-stamp of the subjective node to the notification message. A dependent node receiving a notification will save the time-stamp together with the subjective node's identifier in a small buffer. Hence, a dependent node can safely ignore messages that have a time-stamp which is lower than the last received message from the same subjective node. If no time-stamp for the subjective node is available in the buffer, the message is accepted. This however only solves the problem of out-of-order notifications from one node.

Assume a node a, with successor c, leaves the system. Shortly thereafter a node b, succeeding a but preceding c, joins the system. If the notifications arrive out-of-order the notification of b's arrival might be received first making some dependent nodes point to b. Shortly thereafter the notification of a's departure arrives to a dependent node making it erroneously point to c instead of b.

We solve this problem by letting the notification of the departure (or failure) of a node a contain a's identifier together with a candidate c that the dependent node can point to. The dependent node will not blindly make its pointers point to c. Instead, it will adapt its routing tables to only have c as responsible for intervals for which it has no better candidates. In example above, the dependent nodes using this technique would only point to c if they are not already pointing to b as b is a better candidate than c.

Similarly, the joining of a node only serves to notify the dependent nodes about the arrival of a node. The dependent nodes will only point to the new node if they do not have a better candidate already.

Failures are handled by letting the node that detects the failure to route to the predecessor of the failed node. The predecessor of the failed node finds the successor of the failed node and lets it initiate a leave notification on the failed node's behalf to all the dependent nodes. Thereafter, messages that were not sent due to the failure are resent in an end-to-end fashion.

The main advantage of this scheme is that it will stop false-negatives. For example, node a might leave the system and notify all its dependent nodes. However, before its notification arrives to a dependent node, the dependent node sends a message to a and finds that it is not responding. It will therefore trigger a correction-on-change that will ultimately be routed to the successor of node a which by the local atomic action already knows about node a's departure and hence ignores the false detection of failure.

There is still the chance that the buffer keeping the times-tamps is full or that an orphan message arrives to a newly joined node. Hence, we combine correction-on-change with the previously mentioned correction-on-use to make sure that remaining incorrect routing entries are eventually corrected.

IV. Simulation Results

To validate correction-on-change we have simulated the algorithm in the stochastic discrete event simulator developed by our team using the Mozart[14] programming system. As there might be an auto-correlation within a sample from one simulation run the simulator uses the method of independent replications to generate unbiased estimates from independent and identically distributed variables. The initialization bias of the warm-up period has been removed from every replication. All the simulations are non-terminating where nodes join, leave, and fail with an exponential distribution with parameter λ. Hence, the expected number of joins, leaves, and failures is λ^{-1}.

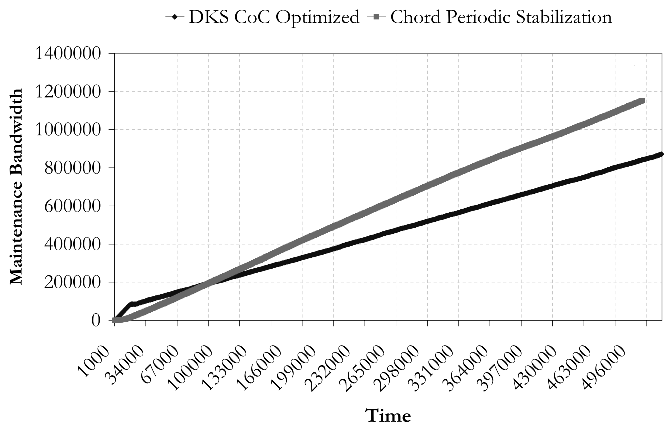


Fig. 7. The maintenance traffic for the Chord system running periodic stabilization every 500 time units and the $\mathcal{DKS}$ system running optimized correction-on-change. A leave and join event happens every 200 time units on average. The Deviation from Legitimate State for these simulations can be seen in Figure 8.

We first show the difference between periodic stabilization in the Chord system and correction-on-change (CoC) in the $\mathcal{DKS}$ system. We do this by fixing the maintenance bandwidth in both systems to investigate the deviation of the routing information from the legitimate state.

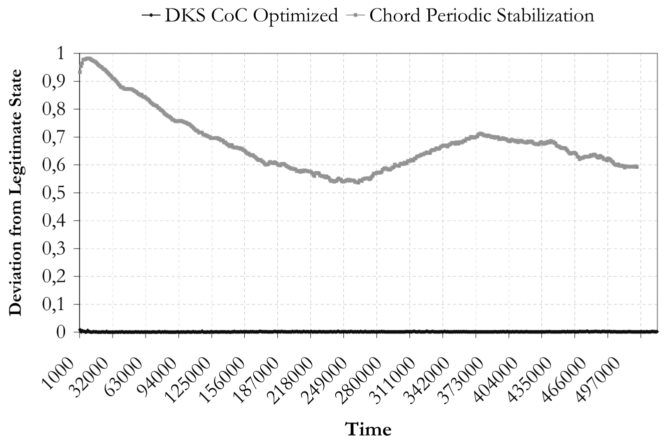


Fig. 8. Deviation from Legitimate State for the Chord system running periodic stabilization every 500 time units and the $\mathcal{DKS}$ system running optimized correction-on-change. A leave and join event happens every 200 time units on average. Both systems consume approximately the same amount of maintenance bandwidth as seen in Figure 7.

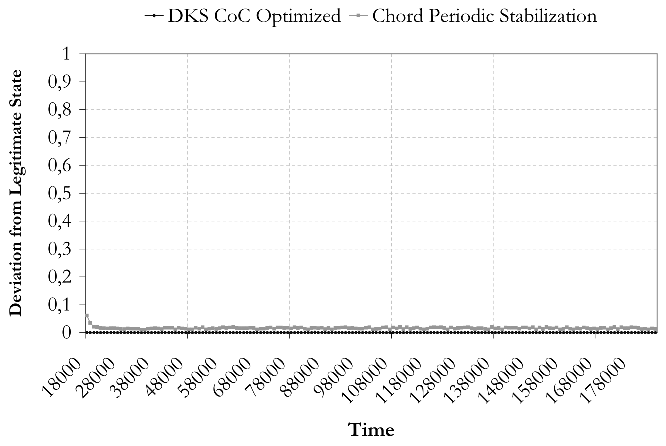


Fig. 9. Deviation from Legitimate State for the Chord system running periodic stabilization every 80 time units and the $\mathcal{DKS}$ system running optimized correction-on-change. A leave and join event happens every 2000 time units on average. Both systems have a legitimate state deviation close to 0 indicating that the system is approximately in a legitimate state.

The deviation from the legitimate state is calculated as the ratio between the total number of incorrect routing pointers and the total number of routing pointers in the system:

$$\frac{\sum_{p\in\mathcal{P}}\sum_{l=1}^{\log_k(N)}\sum_{i=0}^{k-1} F_p(l,i)}{|\mathcal{P}|(k-1)\log_k(N)}$$

where the characteristic function F_p for node p is defined as:

$$F_p(l,i) = \begin{cases} 1, & \text{if} RT_{p,\mathcal{P}}(l,i) = BI_p(l,i) \oplus \min(\{n \ominus BI_p(l,i) | n \in \mathcal{P}\}) \\ 0, & \text{otherwise.} \end{cases}$$

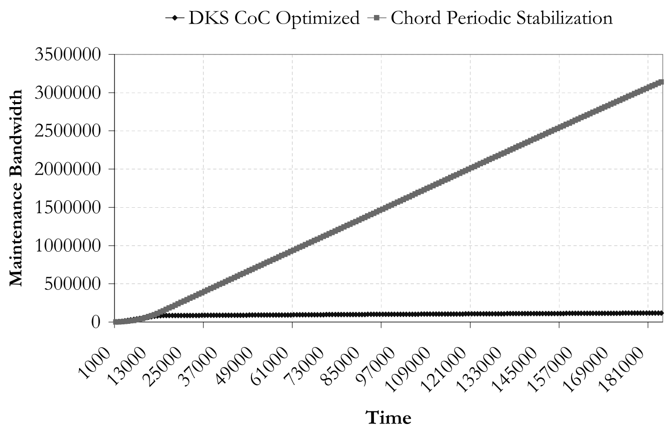


Fig. 10. The maintenance traffic for the Chord system running periodic stabilization every 80 time units and the $\mathcal{DKS}$ system running optimized correction-on-change. A leave and join event happens every 2000 time units on average. The Deviation from Legitimate State for these simulations can be seen in Figure 9.

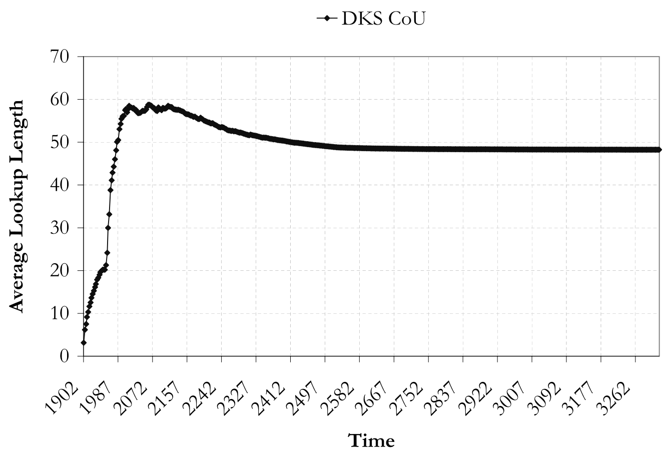


Fig. 12. Average lookup length in a $\mathcal{DKS}$ system only using correction-on-use.

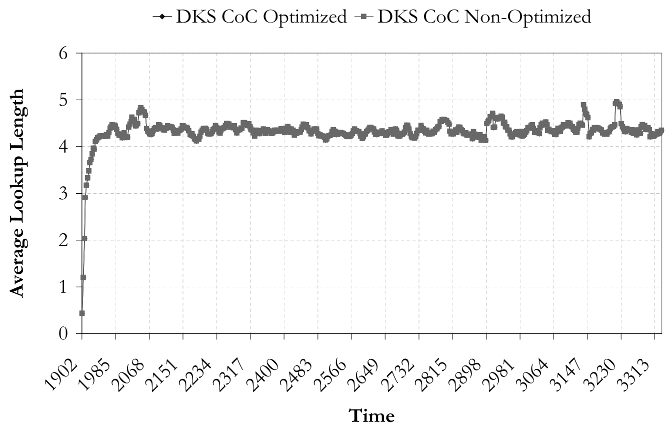


Fig. 11. Average lookup length in a $\mathcal{DKS}$ system using correction-on-change in conjunction with correction-on-use.

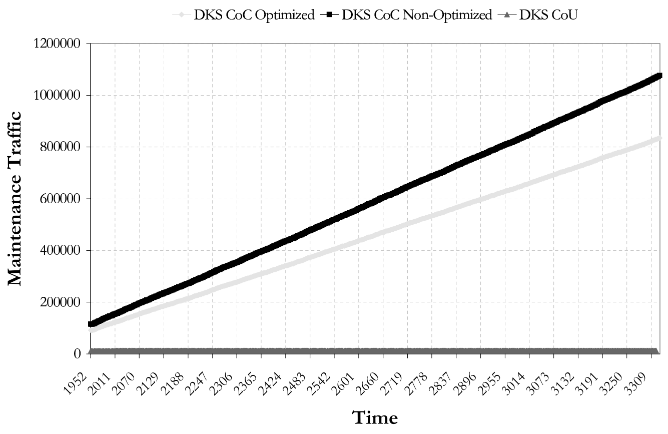


Fig. 13. Amount of maintenance bandwidth used to correct outdated routing entries.

Figure 7 shows a simulation of a system where $N = 2^{12}$ and initially the number of nodes in the system is 512. The nodes arrive and depart each with a rate of $\lambda = \frac{1}{200}$. The curves show the amount of maintenance bandwidth consumed by a Chord system running periodic stabilization and a $\mathcal{DKS}$ ($k = 2$) system running the optimized correction-on-change algorithm. All the stabilization rates in Chord has been set to 500 such that the amount of maintenance bandwidth is equal to the $\mathcal{DKS}$ system. Given these two systems where the maintenance cost is equal, Figure 8 shows the deviation from the legitimate state. The $\mathcal{DKS}$ system is maintained approximately in legitimate state as expected while approximately half of the routing pointers in Chord are incorrect.

We now show the reverse by fixing the deviation from the legitimate state close to optimal (zero) for Chord and $\mathcal{DKS}$. I.e. the routing state in both systems is approximately in a legitimate state, to investigate the amount of maintenance bandwidth consumed in respective system. Essentially, this comparison shows how a adaptive periodic stabilization would perform as the stabilization has been set to match the dynamism in the system perfectly. We experimented with many different stabilization rates to find one which matched the dynamism such that the system would be in approximately legitimate state at all times.

Figure 9 shows a system ($N = 2^{12}$) containing 512 nodes initially. The nodes arrive and depart each with a rate of $\lambda = \frac{1}{2000}$. The curves show the amount of maintenance bandwidth consumed by Chord running periodic stabilization and a $\mathcal{DKS}$ ($k = 2$) running the optimized correction-on-change algorithm. A stabilization will be made by every node in the system every 80 simulation time units. The figure clearly shows that both systems are approximately maintained in a legitimate state. However, as Figure 10 shows, the periodic stabilization consumes significantly more traffic than the $\mathcal{DKS}$ system.

Next, we motivate by means of simulations the advantage of correction-on-change over correction-on-use.

As reported in [7] the use of correction-on-use, instead of periodic stabilization, gives an average lookup length that is approximately $\frac{\log_k(P)}{2}$ for a system of P peers while inducing little maintenance bandwidth. However, for correction-on-use to perform well the system needs to have enough routing messages to correct routing entries that are outdated as the result of dynamism. However, the requirement of high enough traffic might not be fulfilled as we show here by simulation.

We setup a scenario in which the amount of routing messages are not high enough to see how the average lookup length is affected. Figure 12 shows a simulation where nodes arrive and depart with $\lambda = \frac{1}{50}$. The system was setup with $N = 2^{12}$, and $k = 2$ with initially 512 nodes in a legitimate state. The behavior of the nodes is modeled as egoistic, not a completely unrealistic assumption [15], by letting each node on average make 4 lookups with an exponential distribution. The figure shows the average lookup length for successful lookups by far exceeding the expected value 4.5.

The same scenario is simulated using correction-on-change and correction-on-use simultaneously. Two different versions of correction-on-change are used. First, the non-optimized version which updates the dependent nodes in each interval regardless of overlap of the intervals. The second one is based on an optimized version that uses the subroutine `CollapseIntervals` to reduce the bandwidth consumption. Figure 11 shows that for both versions of correction-on-change the curves superimpose giving an average lookup length close to the ideal 4.5.

The traffic induced by the correction-on-change is shown in Figure 13. The figure clearly shows that the amount of maintenance bandwidth is significantly less in the optimized version of correction-on-change.

V. Conclusions

We have presented *correction-on-change*, a general approach for maintaining several types of structured peer-to-peer overlay networks. We showed how it can be applied to the $\mathcal{DKS}$ system as well as the Chord system. The proposed technique builds on correction-on-use and increases system robustness while keeping the maintenance cost low. Indeed, with the correction-on-change technique, bandwidth is consumed only when necessary.

To proof the correctness of correction-on-change, we formally characterized the set of all nodes (i.e. the dependent nodes) that need to be updated upon a join, leave or failure of a node. Furthermore, we showed how the dependent nodes could uniquely be identified with only partial information about the system and provided correctness proof for it. To notify dependent nodes about a change, we provided an improved and restricted version of our previous work on broadcast in $\mathcal{DKS}-$based structured peer-to-peer overlay network. The improved version of our broadcast algorithm does not send any redundant messages despite the presence of outdated routing pointers in the system.

Experimental results show that for the same amount of maintenance bandwidth, correction-on-change makes the system by far more robust when compared to periodic stabilization. Moreover, even if a periodic stabilization that adapts itself perfectly to the dynamism in the system is used, correction-on-change will give the same performance but with a small fraction of the maintenance cost of periodic stabilization.

We provided an algorithm for further reducing the maintenance bandwidth by optimizing overlapping intervals before notifying the dependent nodes. We proved that the algorithm terminates. The proposed optimized algorithm was compared to the non-optimized algorithm.

Furthermore, we have shown by means of preliminary simulations that our approach allows the system to maintain the average lookup length close to the theoretical bound. This improves upon all existing designs we know of.

Currently, the correction-on-change is being tested in our implementation of the $\mathcal{DKS}$ system. In the future, we will report our ongoing effort on how the backlist and frontlist of the $\mathcal{DKS}$ system are maintained similarly to correction-on-change, with the addition of vector-timestamps ensure causality.

References

[1] K. Aberer, A. Datta, and M. Hauswirth. Route Maintenance Overheads in DHT Overlays. Technical Report IC/2003/69, 2003.

[2] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. Technical Report TR-819, MIT, January 2002.

[3] F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In *Proceedings of the Second International Workshop on Peer-to-Peer Systems, IPTPS*, 2003.

[4] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.

[5] 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.

[6] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlayss. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, February 2003.

[7] 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.

[8] 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.

[9] 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.

[10] K. Aberer, A. Datta, and M. Hauswirth. Efficient, self-contained handling of identity. *IEEE Transactions on Knowledge and Data Engineering*, 16, 2004.

[11] K. Lakshminarayanan, A. Rajagopala Rao, and S. Surana. Hyperchord: A Peer-to-Peer Data Location Architecture. 2001.

[12] P. Maymounkov and D. Mazires. 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.

[13] V. Mesaros, B. Carton, and P. V. Roy. S-Chord: Using Symmetry to Improve Lookup Efficiency in Chord. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '03*, Las Vegas, Nevada, USA, June 2003.

[14] Mozart Consortium. http://www.mozart-oz.org, 2003.

[15] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.