

COMPASS – Latency Optimal Routing in Heterogeneous Chord-based P2P Systems

Lukas Probst

Nenad Stojnić

Heiko Schuldt

Technical Report CS-2013-004

University of Basel

Email: {lukas.probst|nenad.stojnic|heiko.schuldt}@unibas.ch

Abstract

During the last decade, overlay networks based on distributed hash tables have become the de facto standard for data management in Peer-to-Peer (P2P) systems, with Chord being its most prominent representative. Essentially, with its fully decentralized approach, Chord avoids any bottleneck and single point of failure while guaranteeing data to be retrieved in $O(\log N)$ hops in a network consisting of N nodes. By optimizing the number of hops for data access, Chord implicitly assumes that all connections between nodes have comparable bandwidth and latency characteristics. However, in heterogeneous, mobile P2P systems that consist of both mobile and fixed nodes, this is not the case. Moreover, due to the mobility of nodes, connection parameters can dynamically change. Especially in mobile P2P applications where low latency for data access is essential, such as in emergency management, routing should aim at reducing the overall latency, rather than the number of hops in the network.

This technical report is based on our paper published at the MDM conference 2013 and extends it by new features and evaluations. In this report, we present COMPASS, a protocol for efficient data access in heterogeneous mobile Chord-based P2P systems. COMPASS takes into account that the network latencies of nodes in a mobile P2P network may significantly differ and thus aims at minimizing the overall latency, even if this necessitates more hops in the network. This is done by probing the network and by maintaining, in addition to Chord's finger table, at each peer a data structure called COMPASS table. We present in detail the initialization and maintenance of the COMPASS table that dynamically adapts to changing node characteristics. Moreover in this report we first time present the new interval joining feature which reduces the COMPASS table size by joining similar intervals. Real world evaluation as well as simulation results show that COMPASS outperforms standard Chord-based routing and reduces the overall latency in heterogeneous P2P networks consisting of fixed and mobile nodes.

Keywords:

Mobile Peer-to-Peer System; P2P Data Management; Chord; Mobile Devices.

1 Introduction

Data management in decentralized Peer-to-Peer (P2P) environments is usually based on overlay networks that exploit distributed hash tables (DHTs). The most prominently used DHT implementation is Chord [SMK⁺01]. Essentially, DHTs optimize data access in P2P systems by guaranteeing data to be found in logarithmic time.

However, DHTs like Chord are agnostic to different node capabilities, including bandwidth and latency characteristics. In a homogeneous environment, optimizing the number of hops for data access is a feasible solution. But this is no longer the case in heterogeneous, mobile P2P systems that consist of both mobile and fixed nodes where the bandwidth and latency characteristics of nodes significantly differ. Moreover, due to the mobility of nodes, connection parameters can dynamically change. In such environments, routing should aim at optimizing data access by reducing the overall latency.

As an example, consider data management in an emergency scenario consisting of small units of fire fighters, equipped with mobile devices, and fixed stations embedded in fire trucks and in control rooms. The devices of fire fighters capture environmental data (e.g., temperature, wind direction) which have to be shared in the network to globally assess the overall situation of the emergency case. Efficient data access is essential as it allows to timely anticipate critical situations that may put the fire fighters' lives into risk. For this reason, global components for data management are not feasible as they might become a performance bottleneck and a single point of failure. Hence, distributed, P2P approaches for data management are urgently necessary. However, network bandwidth and latency of the fire fighters' mobile devices significantly differ from the capabilities of the fixed nodes in the network. In addition, due to the mobility of the fire fighters and their devices, it is rather likely that the communication characteristics of nodes in the system change over time. In order to optimize data access, conventional DHTs cannot be applied as they focus on the number of hops in the network.

Such heterogeneous networks consisting of fixed and mobile devices require a routing protocol for data access that optimizes the overall latency, even if this necessitates additional hops in the network. This technical report is based on the paper we presented at the MDM conference [SPS13] and extends it with new features and evaluations. In this report, we propose COMPASS, a protocol for efficient data access in heterogeneous mobile Chord-based P2P systems. COMPASS maintains, in addition to Chord's finger table, at each peer a data structure called COMPASS table. These tables are built and continuously updated by probing the network to dynamically adapt to changing node characteristics. In this report, we further present a novel interval joining feature. The purpose of this joining feature is to reduce the COMPASS table size and thus the additional network traffic introduced by COMPASS' maintenance method. Both real world evaluation and simulation results show that COMPASS outperforms standard Chord-based routing and reduces the overall latency in heterogeneous P2P networks consisting of fixed and mobile nodes. Moreover, the simulation results confirm that the interval joining features improves COMPASS' scalability while preserving its excellent routing time performance. Furthermore, the simulation results indicate that COMPASS can also be used in unstable networks with a high churn rate.

The remainder of this report is organized as follows. Section 2 presents in detail the concepts behind COMPASS. The evaluation is split in two Sections. Section 3 contains the real world evaluation realized in the Amazon Cloud using the OSIRIS-SR implementation. Section 4 presents the results obtained by simulating COMPASS in the OverSim simulation environment [BHK07, Ins13] based on OMNet++ [Var01, OMN13]. Section 5 presents related work and Section 6 concludes.

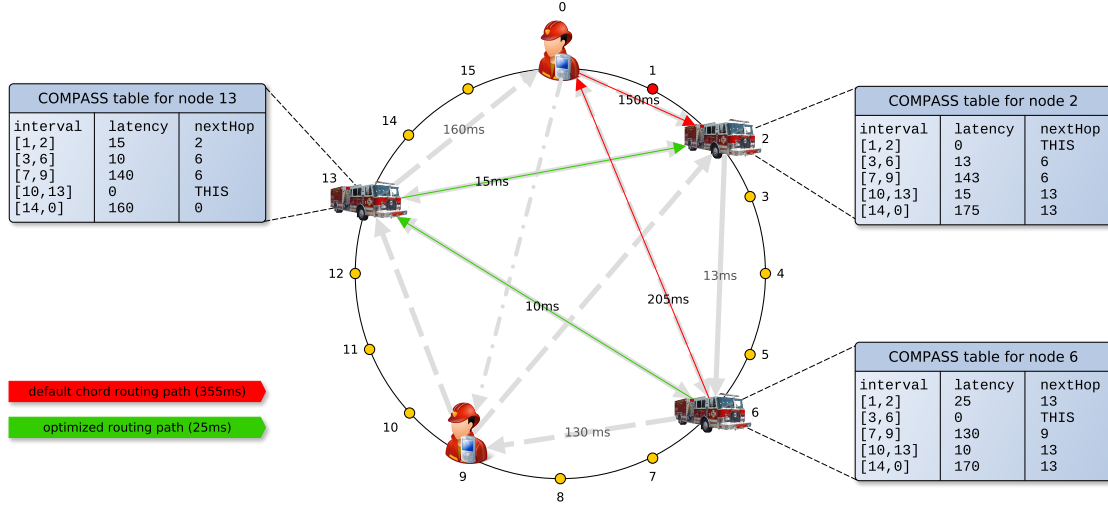


Figure 1: Example ring (4 bit key) with paths for route (ID_1) at node ID_6 . Grey arrows show all possible directed connections (finger table) which can be used for routing. The more solid a grey arrow is, the faster (smaller latency) is the corresponding connection.

2 COMPASS

Chord [SMK⁺01] is a widely used implementation of the DHT concept. Among many other compelling features, Chord primarily offers a distributed, hop-optimized lookup route (ID) routine for fast data access. The core contribution of the Chord approach is unprecedented scalability, in terms of data lookup, but only when applied to a network composed of only static and resource abundant devices such as computer grids, clusters, and clouds. In a nutshell, the scalability feature of Chord is based on the self-organizing *Finger Table* data structure which empowers a node with a limited, yet sufficient, perspective on the whole network. However, for the sake of optimizing in average the number of hops for routing, the Finger Table always disregards the hardware characteristics of and network connections between nodes. As a consequence, such an approach is in general very inconvenient with regard to routing times when the Finger Table is populated with mobile devices. Essentially, due to their limited and frequently changing communication characteristics, mobile nodes, located on the hop-optimized routing paths, may lead to significant latency delays. A latency-friendly approach for a P2P network including fixed and mobile nodes would require a deviation from the default routing paths of Chord towards the usage of paths that are predominantly composed of powerful (i.e., static) devices instead.

2.1 COMPASS at a Glance

An example in which it is beneficial to deviate from the default Chord path is illustrated in Figure 1. It shows a simple Chord routing process in a heterogeneous environment in the context of an emergency management scenario as presented in Section 1. Figure 1 depicts three well connected vehicles and two firemen using mobile devices with low bandwidth, high latency mobile devices. Hence, the communication originating from the firemen (dashed lines) is characterized by much higher latency ($\geq 150ms$) compared to the well connected (solid lines) vehicles ($\sim 15ms$). When mapping all the actors onto a Chord identifier space (in this example, for illustration purposes,

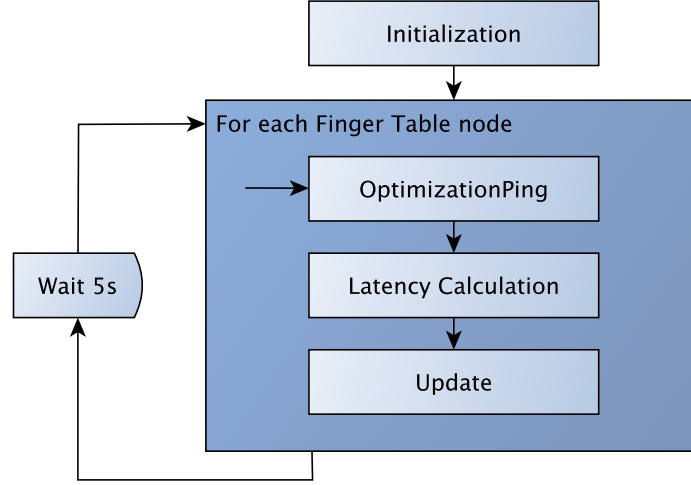


Figure 2: COMPASS Table Construction Process (runs independently on each node).

limited to 4 bits) and sending a simple message (e.g., corresponding to ID_1) from one vehicle (ID_6) to another (ID_2), default Chord (red line) would necessitate a forwarding to one mobile node (ID_0). As a consequence, the resulting routing time of the message would turn out to be rather high, i.e., 355ms (latencies are randomly picked for this example). However, if the sending vehicle (ID_6) uses an alternative and well connected node (i.e., ID_{13}) from its routing table for forwarding instead (green line), then the resulting routing time would turn out to be much lower, that is, only 25ms. Obviously, even in such a simple example, a mobile node may impact overall latency and the deviation from standard Chord routing can demonstrate significant performance improvements when accessing data.

COMPASS, a latency-based global optimization algorithm on top of Chord, addresses this problem. The goal of COMPASS is to dynamically determine the data access paths that minimize overall latency –which are predominantly composed of powerful (i.e., static) nodes– while preserving the fundamental routing features of Chord such as scalability and consistency. Essentially, COMPASS introduces a novel data structure, called the COMPASS table, which is derived from the Chord Finger Table and the latencies to the Finger Table nodes. The precise latency times are acquired by means of periodic probes in the P2P network. In order to obtain the optimal data access paths on a global scale, the local COMPASS table instance is iteratively improved by additionally extracting and aggregating the COMPASS tables of the nodes referred to in the Chord Finger Table. In fact, the aggregation is responsible for selecting the best forwarding nodes w.r.t. the latencies. Thereby, we limit the selection to Finger Table nodes only as to preserve Chord’s scalability w.r.t. the overall key space. After a finite number of iterations, optimal routing paths eventually converge. The COMPASS table maintenance algorithms closely resemble the algorithms leveraged in Distance Vector Routing [Bla00]. Since the chosen best forwarding nodes may differ from the default Finger Table forwarding nodes, the converged optimal routing paths are expected to deviate from the default Chord paths. Nevertheless, they are based on the same Finger Tables and thus address the same nodes as Chord routing does. During the table convergence time, the absence of globally optimal data access paths, COMPASS temporarily overcomes by means of a fallback to Chord’s Finger Table forwarding nodes for routing. Figure 2 illustrates the aforementioned COMPASS table generation processes, independently running at each node, in a sequential order.

interval (i)	aggregated latency (l)	next hop (f)
[1, 2]	25 ms	ID_{13}
[3, 6]	0 ms	THIS
[7, 9]	130 ms	ID_9
[10, 13]	10 ms	ID_{13}
[14, 0]	170 ms	ID_{13}

Table 1: COMPASS table for node ID_6 (w.r.t. example ring of Figure 1)

2.2 COMPASS Table Initialization

The main idea behind the COMPASS table is to associate the optimal forwarding node (f), in terms of latency, out of the set of Finger Table nodes (F) to each identifier (id) from Chord's circular identifier space (ID , where $|ID| = 2^k$ and k is the desired key length in bits). In doing so, we preserve the analogy with the Finger Tables to associate (hop) optimal nodes to ring space identifiers for routing. Moreover, in analogy to Distance Vector routing, a distance metric is required in COMPASS, as well. For this purpose, the aggregated latency l (with $l \in \mathbb{R}$) to each identifier (id) is to be associated, as well, as to be aware of the estimated routing time towards this identifier. In COMPASS as well as in Chord, for each identifier (id) there is always a unique associated ring node (n), more specifically, a successor ($\forall id \in ID \exists n \in N : \neg \exists n_2 \in N : n_2.id \in [id, n.id)$, where N is the set of all nodes in the ring).

For all the identifiers associated to the same node (n) the optimal forwarding node (f) and the aggregated latency (l) are logically the same, because the routing result and thus the optimal routing path for all of these identifiers is the same. Hence, we can collapse all of these identifiers into an interval ($i \in I \mid i = [n.predecessor.id + 1, n.id]$ where I is the set of all intervals), and leverage it for the associations in the COMPASS table instead. Note, that any interval ($i \in I, lowerBound \in ID, upperBound \in ID \mid i = [lowerBound, upperBound]$) is, again according to Chord, always unique and the union of all intervals results in ID ($i_1, i_2 \dots i_n \in I \mid i_1 \cup i_2 \cup \dots i_n = ID$). Hence, an interval association with the corresponding forwarding node and the estimated latency can be described with a triplet ($ri = \langle i, l, f \rangle \mid i \in I, l \in \mathbb{R}, f \in F$), henceforth known as the routing interval ri , such that the set containing all routing interval triplets forms the COMPASS table. Table 1 illustrates an example COMPASS table and the contained triplets. For instance, the triplet $\langle [14, 0], 170ms, ID_{13} \rangle$ from the illustrated COMPASS table conveys the information that from node ID_6 the interval $[14, 0]$ can be optimally reached by forwarding to node ID_{13} in 170ms.

The COMPASS table initialization is presented in Alg. 1. It populates the table with default routing interval values; these values reflect the node's local knowledge about the rest of the environment, such as the own identifier area of responsibility in the Chord ring. The first triplet, which reflects the interval of the node itself, is created with the aggregated latency set to be 0 ms and the forwarding node to itself (the actual value $l = 0$ ms, $f = \text{THIS}$). If the node is accompanied by some other node in the ring, a second triplet is created whose interval corresponds to the rest of the ring and to the default unknown forwarding node/aggregated latency value (lines 2-11 in Alg. 1). In this case, the rest of the ring interval is bound by the node itself and its Chord predecessor. The default unknown forwarding node/aggregated latency value is chosen to be $NULL$ in the former case and ∞ in the latter case. Note, that this assignment is fundamental as an unknown value in the COMPASS table for some interval causes a fallback to Chord's Finger Table during routing.

```

1: if predecessor  $\neq$  null then
2:   myStart  $\leftarrow$  (predecessor.id + 1) modulo keyspaces
3:   myEnd  $\leftarrow$  node.id
4:
5:   otherStart  $\leftarrow$  (myEnd + 1) modulo keyspaces
6:   otherEnd  $\leftarrow$  myStart - 1
7:   if otherEnd = -1 then
8:     otherEnd  $\leftarrow$  keyspaces - 1
9:   end if
10:  otherInt  $\leftarrow$  RoutingInt(otherStart, otherEnd, NULL,  $\infty$ )
11:  routingIntervals.add(otherInt)
12: else
13:  myStart  $\leftarrow$  (node.id + 1) modulo keyspaces
14:  myEnd  $\leftarrow$  node.id
15: end if
16: myInt  $\leftarrow$  RoutingInt(myStart, myEnd, THIS, 0)
17: routingIntervals.add(myInt)
18: sort(routingIntervals)

```

Algorithm 1: COMPASS Table Initialization

Finally, the triplets are sorted according to the lower bound intervals due to update simplicity purposes. For instance, the COMPASS table of node ID_6 contains the triplets $\langle [3, 6], 0ms, THIS \rangle$ and $\langle [7, 2], \infty, NULL \rangle$ after the initialization.

2.3 COMPASS Table Maintenance

The fully initialized COMPASS table is used for further refinement of the default values with actual node latency information. We resort to Distance Vector Routing techniques to obtain the refinement sources and to update the triplet values. In our case, the sources for the local COMPASS table refinement are the COMPASS tables of the Finger Table nodes, since they are potential optimal forwarding nodes. The COMPASS tables of these nodes are periodically obtained by means of the `OptimizationPing()` routine, which combines latency measurement and COMPASS table retrieval at the same time. In other words, whenever a remote node receives the `OptimizationPing()` message it immediately responds to it with its own COMPASS table. Hence, the elapsed time of an `OptimizationPing()` response is asserted as the direct latency towards the Finger Table node at the issuing COMPASS node. To avoid heavy oscillations of our direct latency measurements results, which can often occur in networks (especially in wireless ones), COMPASS follows the suggestions presented in literature [RGRK04] by applying an exponential weighted moving average value (cf. Equation 1) on the measurements. Using this approach the latest measured latency ($newVal$) is not used directly as the new latency estimation ($latency(t+1)$) towards the pinged node, but is combined with the present estimation ($latency(t)$). We set α to 0.4, as it proved best in our experiments in filtering out frequent oscillations and detecting major node events, such as network topology changes, at the same time.

$$latency(t+1) = \alpha \cdot newVal + (1 - \alpha) \cdot latency(t) \quad (1)$$

The obtained COMPASS tables are further processed by the refinement algorithm, depicted in Alg. 2. Essentially, the algorithm is divided into four blocks. The first two blocks are responsible for COMPASS table alignment. In other words, each node is likely to have, in the process of COMPASS table convergence, a differently fragmented ring and a different knowledge about the other ring nodes and their identifier areas of responsibility. Accordingly, the routing intervals of

the corresponding triplets between two tables will be unequally bounded. In order for two COMPASS tables to be comparable, the unequally bounded routing intervals have to be re-aligned by adjusting their lower/upper bounds. However, since COMPASS is located in a circular identifier space, remote table triplet intervals will always be either a subset of the local default value interval or overlapping with a single local interval or multiple ones. Thus, they can be realigned by means of interval splitting. For instance, node ID_6 contains the two intervals $[3, 6]$ and $[7, 2]$ after initialization. Instead, node ID_{13} , which is one of its Finger Table nodes, contains the two intervals $[10, 13]$ and $[14, 9]$. As one can see, to refine the COMPASS table of node ID_6 by using the information of node ID_{13} , node ID_6 has to split the intervals in both tables into four intervals (i.e., $[3, 6]$, $[7, 9]$, $[10, 13]$ and $[14, 2]$) in order to re-evaluate its routing information using Distance Vector techniques.

Inside the first block (i.e., lines 1-11), the local triplets and the ones received as a response to the `OptimizationPing()` routine are sorted according the triplet's lower bound interval value into a list (i.e., $\{3, 7, 10, 14\}$). Inside the second block (i.e., line 13-26), the triplet intervals are split by setting the upper bound triplet interval value to be equal to the lower bound triplet interval value of the succeeding triplet in the sorted list. Moreover, the newly split intervals are initialized with the local node triplet values only (i.e., lines 21-23) so as they can be re-evaluated with the remote node's triplet latency values. For the given example in node ID_6 , the split triplet list looks as follows: $\langle [3, 6], 0ms, THIS \rangle$, $\langle [7, 9], \infty, NULL \rangle$, $\langle [10, 13], \infty, NULL \rangle$ and $\langle [14, 2], \infty, NULL \rangle$.

Note, that once the iterative aggregation process of the COMPASS table has converged, the number of routing interval triplets will always match the number of nodes in the Chord ring ($|RI| = |N|$), if the interval joining feature (fourth block) is disabled (in this case $|RI| \leq |N|$). Moreover, all of the nodes will have the same fragmentation of the ring, i.e., equally bound triplet routing intervals. This is due to the fact that node intervals (i) are, according to Chord, disjoint and convey a precise latency information (i.e., latency of the corresponding node), which means that after a finite number of propagation iterations they will have replaced the information non-conveying default triplet intervals at each node, resulting in the joined set of all intervals I .

The third block (lines 28-45) re-evaluates the latency information associated to the newly created triplet intervals by means of aggregation with the other node's triplet values. The newly created triplet intervals are updated in two cases only. First (line 32-38), if for an interval the associated forwarding node is already equal to the node (u) from which we received the currently processed COMPASS table. In this case, the latency value of that forwarding node has to be updated with the latest latency measurement towards u . Second (lines 39-44), if an interval is reachable through u in less time than locally saved we have set u as the forwarding node and adapt the latency. In the former case, the change in latency (both increase and decrease) of the already established optimal paths is facilitated, whereas in the latter case, the selection of a new and faster latency optimal path than known before is facilitated. To continue with our example, the COMPASS table of node ID_6 would contain the following triplets after aggregation with the COMPASS table of node ID_{13} if the latency between them is $10ms$: $\langle [3, 6], 0ms, THIS \rangle$, $\langle [7, 9], \infty, NULL \rangle$, $\langle [10, 13], 10ms, ID_{13} \rangle$ and $\langle [14, 2], \infty, NULL \rangle$. The converged COMPASS tables are shown in Figure 1.

Moreover, the COMPASS table maintenance algorithm presented in Alg. 2 offers more functionality than just optimal path aggregation (see lines 33-35 and 40). This part is responsible for handling the problems resulting from the *Count-to-Infinity* effect. The Count-to-Infinity effect is a common occurrence in Distance Vector Routing based algorithms, such as ours, and it is re-

```

1: ownRoutingInts = routingIntervals
2: intervalStartList  $\leftarrow$  List < Integer >
3: for all myInt in ownRoutingInts do
4:   intervalStartList.add(myInt.intStartKey)
5: end for
6: for all otherInt in otherRoutingInts do
7:   if otherInt.intStartKey  $\notin$  intervalStartList then
8:     intervalStartList.add(otherInt.intStartKey)
9:   end if
10: end for
11: sort(intervalStartList)
12:
13: newRoutingInts  $\leftarrow$  List < RoutingInt >
14: for i = 0 to intervalStartList.size - 1 do
15:   intStart  $\leftarrow$  intervalStartList.get(i)
16:   if i = intervalStartList.size - 1 then
17:     intEnd  $\leftarrow$  intervalStartList.get(0) - 1
18:   else
19:     intEnd  $\leftarrow$  intervalStartList.get(i + 1) - 1
20:   end if
21:   myIntForI  $\leftarrow$  getIntContainingKey(ownRoutingInts, intStart)
22:   nextHop  $\leftarrow$  myIntForI.nextHop
23:   latency  $\leftarrow$  myIntForI.latency
24:   newInt  $\leftarrow$  RoutingInt(intStart, intEnd, nextHop, latency)
25:   newRoutingInts.add(newInt)
26: end for
27:
28: directLatency  $\leftarrow$  directLatencies.get(keyFromNextHop)
29: for all interval in newRoutingInts do
30:   curStart  $\leftarrow$  interval.intStartKey
31:   otherInt  $\leftarrow$  getIntContainingKey(otherRoutingInts, curStart)
32:   if interval.nextHop = keyFromNextHop then
33:     if otherInt.nextHop = node.id then
34:       interval.nextHop  $\leftarrow$  NULL
35:       interval.lLatency  $\leftarrow$   $\infty$ 
36:     else
37:       interval.latency  $\leftarrow$  directLatency + otherInt.latency
38:     end if
39:   else if interval.latency > directLatency + otherInt.latency then
40:     if otherInt.nextHop = NULL  $\vee$  otherInt.nextHop  $\neq$  node.id then
41:       interval.nextHop  $\leftarrow$  keyFromOtherNode
42:       interval.latency  $\leftarrow$  directLatency + otherInt.latency
43:     end if
44:   end if
45: end for
46:
47: if  $\neg$ enableCompassIntervalJoining then
48:   routingIntervals  $\leftarrow$  newRoutingInts
49: else
50:   joinedRoutingInts  $\leftarrow$  List < RoutingInt >
51:   joinedRoutingInts.add(newRoutingInts.get(0))
52:   for i = 1 to newRoutingInts.size - 1 do
53:     indexLast  $\leftarrow$  joinedRoutingInts.size - 1
54:     lastInt  $\leftarrow$  joinedCompassTable.get(indexLast)
55:     curInt  $\leftarrow$  newRoutingInts.get(i)
56:     if isSimilarEnoughForJoin(lastInt, curInt) then
57:       joinedInt  $\leftarrow$  generateJoinedRoutingInterval(lastInt, curInt)
58:       joinedRoutingInts.add(indexLast, joinedInt)
59:     else
60:       joinedRoutingInts.add(curInt)
61:     end if
62:   end for
63:   if joinedRoutingInts.size > 1 then
64:     lastInt = joinedRoutingInts.back()
65:     firstInt = joinedRoutingInts.front()
66:     if isSimilarEnoughForJoin(lastInt, firstInt) then
67:       joinedInt  $\leftarrow$  generateJoinedRoutingInterval(lastInt, firstInt)
68:       joinedRoutingInts.pop_back()
69:       joinedRoutingInts.pop_front()
70:       joinedRoutingInts.add(joinedInt)
71:     end if
72:   end if
73:   routingIntervals  $\leftarrow$  joinedRoutingInts
74: end if

```

Algorithm 2: COMPASS Table Maintenance with Interval Joining

sponsible for invalidating the established paths. In more detail, the forwarding nodes (two or even more), finding themselves in such situations, form an optimal path that corresponds to an infinite loop. In theory, Count-to-Infinity problems should resolve themselves after some time; however, in practice the resolution of this problem tends to be rather lengthy, which in the meantime nullifies the optimal data access paths of COMPASS. Moreover, the infinite message looping of such nodes results in an unnecessary waste of local computational and communication resources likely needed for some other functionalities. Therefore, we apply effective techniques similar to Split Horizon [Bla00] (see line 40) and Poison Reverse [Bla00] (see lines 33-36) in conjunction with the exponential weighted moving average latency measurements (equation 1) for circumventing infinite loop optimal paths between nodes. Note, that the two mentioned solutions can protectively resolve only simple infinite loops situations (two nodes). However, more complex (three or more) infinite loop situations are rather seldom in practice and can be reactively handled by means of the Poison Reverse technique.

The original COMPASS concept [SPS13] has been extended by an optional interval joining feature. The task of this feature is to join consecutive intervals with the same next hop and a similar latency. The main idea behind this is that we suppose that COMPASS does not have a remarkable benefit from having two (or even more) consecutive intervals with the same next hop and only slightly different latency information. Instead we suppose that it is more beneficial to join these intervals. In this case there would be less intervals ($|RI| < |N|$) to maintain. We assume that this feature improves COMPASS' scalability, since (in a ring with a huge number of nodes) the COMPASS table size of each node and thus the additional traffic (introduced by `OptimizationPing()`), the memory consumption (for storing the COMPASS tables) and the table maintenance algorithm runtime can be reduced.

The fourth block (lines 47-74) implements the interval joining feature. In this block all the intervals stored in the *newRoutingInts* list are simply putted into the *joinedRoutingInts* list as long as they are not similar enough. In case that they are similar enough, the current interval from the *newRoutingInts* list and the topmost interval in the *joinedRoutingInts* list are joined. The code after the for-loop is necessary for checking if the first and the last interval also have to be joined. The `isSimilarEnoughForJoin(firstInt, secondInt)` method, which is used for checking if two intervals are similar enough for joining, is implemented by Alg. 3. Therefore, it first checks if the next hops are the same. If this is the case it further calculates the relative difference (cf. Equation 2) and checks if this value is lower than a specified similarity threshold. The `generateJoinedRoutingInterval(firstInt, secondInt)` method, which generates a joined interval given two intervals as its inputs, is implemented by Alg. 4. To do so, a new interval is created using the start key of the first interval, the end key of the second interval (which has to succeed the first interval in the ring), the common next hop and the greater latency value. For instance, assume that in a ring with a larger identifier space (6 bits) node ID_6 contains the two intervals $\langle [10, 13], 7ms, ID_{13} \rangle$ and $\langle [14, 50], 10ms, ID_{13} \rangle$. Further assume, that we specified the similarity threshold to be 0.4. Since the relative difference is lower than the similarity threshold ($\frac{10-7}{10} = 0.3 < 0.4$) the two intervals can be joined into the interval $\langle [10, 50], 10ms, ID_{13} \rangle$.

$$relativeDiff = \frac{|firstLatency - secondLatency|}{\max\{firstLatency, secondLatency\}} \quad (2)$$

```

1: if  $firstInt.nextHop = NULL \vee secondInt.nextHop = NULL$  then
2:   return false
3: else if  $firstInt.nextHop = secondInt.nextHop$  then
4:    $firstLatency \leftarrow firstInt.latency$ 
5:    $secondLatency \leftarrow secondInt.latency$ 
6:   if  $firstLatency > secondLatency$  then
7:      $greaterLatency \leftarrow firstLatency$ 
8:      $difference \leftarrow firstLatency - secondLatency$ 
9:   else
10:     $greaterLatency \leftarrow secondLatency$ 
11:     $difference \leftarrow secondLatency - firstLatency$ 
12:   end if
13:    $percentualDiff \leftarrow difference / greaterLatency$ 
14:   if  $percentualDiff \leq joiningSimilarLatencyThreshold$  then
15:     return true
16:   else
17:     return false
18:   end if
19: else
20:   return false
21: end if

```

Algorithm 3: Check Interval Similarity

```

1:  $firstLatency \leftarrow firstInt.latency$ 
2:  $secondLatency \leftarrow secondInt.latency$ 
3: if  $firstLatency > secondLatency$  then
4:    $greaterLatency \leftarrow firstLatency$ 
5: else
6:    $greaterLatency \leftarrow secondLatency$ 
7: end if
8:  $intStart \leftarrow firstInt.intStartKey$ 
9:  $intEnd \leftarrow firstInt.intEndKey$ 
10:  $nextHop \leftarrow firstInterval.nextHop$ 
11:  $joinedInt \leftarrow RoutingInt(intStart, intEnd, nextHop, greaterLatency)$ 
12: return  $joinedInt$ 

```

Algorithm 4: Generate Joined Interval

The similarity threshold introduces a trade-off between high precision and high scalability. This trade-off is illustrated in Figure 3. So far using COMPASS was based on high precision, i.e., for each interval the COMPASS table contains exact routing information regarding the estimated latency. While this choice resulted in optimal routing w.r.t. the overall latency, it harmed Chord’s high scalability. The novel interval joining feature or more precisely the similarity threshold parameter enables us to choose between high precision and high scalability. However, this choice is not binary. Instead, one can specify the importance of scalability, i.e., how much the COMPASS tables should be compressed. The similarity thresholds can take values between 0 and ∞ and specifies the allowed relative latency difference between two consecutive intervals. A higher similarity threshold increases the allowed relative difference and thus results in smaller COMPASS tables. Hence, the higher the similarity threshold is the more importance is attached to the precision at the cost of harming the scalability. The boundary values (0 and ∞) result in two trivial cases. Using 0 results in not joining any interval as long as the latency information are not exactly the same. Thus, using 0 coincides with disabling the interval joining feature with the exception that unintentional splitted intervals induced by churn can be merged¹ (see Section 4.3.3). In contrast,

¹However, due to numerical reasons we would suggest to use a small value (e.g., 0.01) instead of 0.

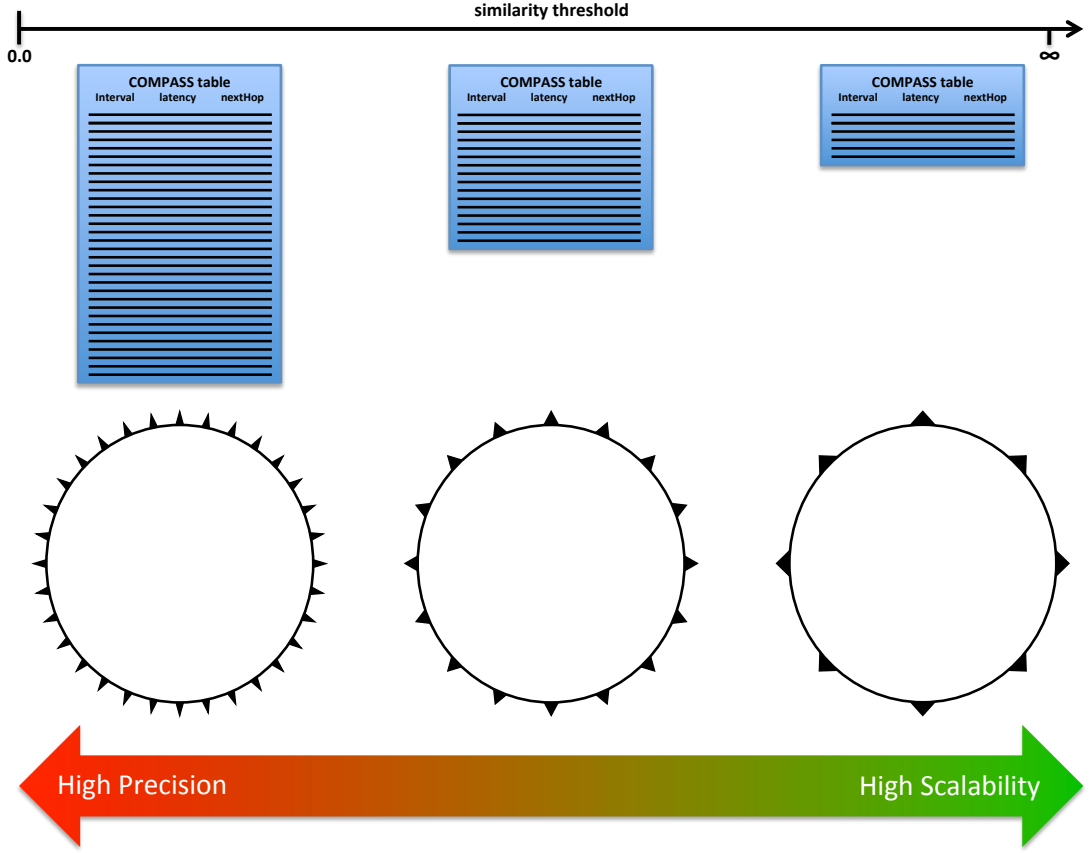


Figure 3: Precision/Scalability Trade-off introduced by the interval joining similarity threshold

using ∞ results in joining all consecutive intervals with the same next hop. For instance, assume that node ID_6 has the two intervals $\langle [10, 13], 7ms, ID_{13} \rangle$ and $\langle [14, 50], 1007ms, ID_{13} \rangle$ since the latency from ID_{13} to next hop for the interval $[14, 50]$ is $1000ms$ due to a very slow connection. In this case using ∞ would result in the joined interval $\langle [10, 50], 1007ms, ID_{13} \rangle$. This is indeed not desired, since the latency stored in the joined interval does not reflect the approximate latency to reach the owner of the interval $[10, 13]$. Remember that we only want to join intervals with similar latency information. Thus, using ∞ would result in the smallest possible COMPASS table and thus in the best scalability but at the same time demolish the latency information in the COMPASS table which is the foundation of the COMPASS concept. Which latency values are similar enough for joining two intervals depends on the characteristics of the network where COMPASS should be deployed and on the application which should use the P2P system enriched with COMPASS. Hence, the perfect similarity threshold depends on the system environment and finding it is a nontrivial problem. The problem of finding the perfect similarity threshold for our simulated network is one of the issues we focus in Section 4.

Since in each COMPASS table maintenance iteration (see blue box in Figure 2) Alg. 2 is called for every Finger Table node with complexity of $\mathcal{O}(|N| \cdot \log(|N|))$ and there are at most k Finger Table nodes (k = key length in bits), the overall runtime complexity for one iteration on one node is $\mathcal{O}(k \cdot |N| \cdot \log(|N|))$. This additional effort is the price for the huge improvement of COMPASS

in heterogeneous P2P environments like the firefighter scenario presented in Section 1. This effort can be reduced by means of the interval joining feature, i.e., by reducing the precision.

2.4 Compass Dynamics

In order for the COMPASS algorithm to work consistently, in terms of routing, just as standard Chord does, all of the possible node events in the network have to be properly handled. For instance, it is very likely that in the course of the Chord ring existence a certain amount of nodes will either leave or join the network. This is particularly the case in heterogeneous environments where mobile nodes, due to their limited resources, may easily disconnect. In such an event, also the Chord Finger Tables eventually adapt to the new state of the ring. Since the Finger Tables are an optimization basis for COMPASS, such changes have to be propagated to COMPASS level. Non-propagated changes of the Finger Tables result in outdated optimal routing paths, also probably to Count-to-Infinity problems, and thus to inconsistent routing compared to standard Chord. A simple, yet effective solution to the change of Finger Tables, is in our case, to re-initialize the COMPASS table and thus restart the optimal path convergence. Moreover, to guarantee that the optimal paths are always consistent, we also have to re-initialize the COMPASS table whenever the node predecessor changes. Finding a more sophisticated solution which is able to repair the COMPASS table without a full reinitialization and thereby increase the performance of COMPASS in environments with a high churn rate is a subject of future research.

2.5 Optimized Routing

The routing in COMPASS is primarily performed on the basis of the COMPASS tables. Whenever there is an optimal path associated to the destination identifier, COMPASS is always consulted first for the purpose of finding forwarding nodes. In the case, that the COMPASS table does not contain information on a forwarding node for the given destination identifier, COMPASS has to resort to the Finger Table instead. For instance, the COMPASS table is inapplicable for routing towards the ID_1 at node ID_6 immediately after its initialization, as the requested identifier is contained in the triplet $\langle [7, 2], \infty, NULL \rangle$. That is, the forwarding node is determined on the basis of the Finger Table hop-optimized forwarding algorithm. This way, the forwarding process remains uninterrupted, but at the expense of latency optimality. Note, that unavailable optimal forwarding node information in COMPASS tables are only possible immediately after a table re-initialization, that is, while the optimal path computation is still in the process of convergence. Hence, COMPASS and Chord are jointly responsible for the process of routing and do not mutually exclude each other, but rather COMPASS only extends Chord's `route(ID)` routine.

The aforementioned latency optimal path routing is presented in Alg. 5. Firstly, the routing algorithm performs a test for the existence of the COMPASS table (line 2) and the forwarding node information it contains (line 4). If a forwarding node exists that corresponds to the local node, then the destination is reached (lines 5-6). Otherwise, the route request is forwarded to the forwarding node if it is not already existent in the current hop history (lines 9-11). Assume that in our example all COMPASS tables have already converged (see Figure 1) a routing request for the ID_1 invoked at node ID_6 is conducted as follows: At node ID_6 the request is forwarded to node ID_{13} , as ID_1 is contained inside the triplet $\langle [1, 2], 25ms, ID_{13} \rangle$. Further the request is forwarded from node ID_{13} to node ID_2 , since it is contained inside the triplet $\langle [1, 2], 15ms, ID_2 \rangle$. Eventually node ID_2 detects itself as the destination, due to the triplet $\langle [1, 2], 0ms, THIS \rangle$.

```

1: hopHistory.add(node)
2: if routingIntervals are not empty  $\wedge$  predecessor  $\neq$  null then
3:   interval  $\leftarrow$  getIntContainingKey(ownRoutingInts, key)
4:   if interval.nextHop  $\neq$  NULL then
5:     if interval.nextHop = THIS then
6:       return THIS
7:     else
8:       nextHop  $\leftarrow$  interval.nextHop
9:       if nextHop  $\notin$  hopHistory then
10:        return nextHop.route(key, hopHistory)
11:       else
12:        interval.latency  $\leftarrow \infty$ 
13:        interval.nextHop  $\leftarrow$  NULL
14:       end if
15:     end if
16:   end if
17: end if
18: use Finger Table for routing

```

Algorithm 5: The COMPASS Routing Algorithm

In the case that a forwarding node is already contained in the hop history, a loop is detected. Due to the effects of loops, which have already been discussed in Section 2.3, reactive Poison Reverse solutions have to be applied (lines 12-13). In contrast to the typical Poison Reverse approach, which is based on a predetermined hop counter as to detect infinite loops, we apply a hop history that we pass along with the route request. We choose to do so, as we believe that reasonable hop count thresholds cannot be detected without the application of complex statistical predictions, which are non-existent in our case. Hence, since Poison Reverse requires a hop history, we keep record of all traversed nodes in a list. If the loop is detected, a fallback to the Finger Tables of Chord for further routing is conducted. Moreover, in accordance with Poison Reverse the loop detecting node COMPASS table information have to be reset as to break the infinite loop.

2.6 COMPASS vs. Chord

The data access paths derived from the COMPASS tables are expected to deviate from the paths derived from the Finger Tables. The reason for this fact is that both routing algorithms are based on completely different optimization goals (i.e., latency vs. number of hops). In order for both algorithms to be nevertheless consistent, in terms of equal destination node message routing, an important precautionary measure has to be taken in COMPASS. While Chord-based routing is designed to determine the destination node at the predecessor node, to which it has been routed by means of the Finger Table, COMPASS' destination resolution is performed directly at the successor node.

In Chord, route requests are always forwarded to the closest known predecessor, located in the Finger Table, until the actual predecessor is reached. The routing request is directly answered with the current successor node entry instead of further forwarding the request to the successor itself². In contrast, COMPASS performs Distance Vector based routing founded on aggregated latencies and predecessor node entries. Since only the successor node's COMPASS table contains the interval which can reach the identifier without any further hop, every routing request has to end at the identifier's successor. Hence the resolution of the correct destination nodes is performed

²Note that in the example illustrated in Figure 1 this last hop is done for illustration purposes, since otherwise a larger ring with more nodes would have been necessary to show COMPASS' impact. However, in the real world evaluations as well as in the simulations this last hop is not done for Chord.

at the destination node itself. The mentioned interval, which is fundamental for this purpose, is created during the initialization of the COMPASS table using the current predecessor node entry. Hence, while Chord relies on correct successor entries at each node, COMPASS relies on correct predecessor node entries. In order for the destination node resolution to be always correct in COMPASS, the predecessor entry correctness has to be ensured, as well.

The correctness of the predecessor/successor values is maintained by means of the `stabilize()` and `notify(successor)` routines of Chord. The `stabilize()` routine is periodically performed, at each node, as to ensure that the node's successor entry is always up-to-date. Hence, this method ensures the correct destination node resolution for Chord. Moreover, after the `stabilize()` method updates the successor entry at a node, the predecessor entry update, at the same successor node, is immediately followed, by means of the `notify(successor)` routine. In this way, correct node resolution of COMPASS is ensured, as well.

3 Real World Implementation and Evaluation

In this section, we present the performance characteristics of COMPASS compared against standard Chord on a real system, placed in a heterogeneous node environment setting. First, we measure and compare the average routing times of both routing algorithms in a 20 node setting. Second, we investigate the scalability features of COMPASS compared to standard Chord.

3.1 Implementation

To host the evaluated routing algorithms, we consider the Java based OSIRIS-SR [SS12] framework. OSIRIS-SR supports distributed management of complex tasks involving multiple peers (e.g., composite services, stream processing etc.). For this, it comes with a rich set of distributed systems functionalities such as reliable distributed composite service navigation, persistent message queuing, metadata replication, auditing, and system workload monitoring. In terms of architecture, the OSIRIS-SR framework is based on a number of plug-able modules, making it easy to adapt to specific high level application needs such as in the case of our routing algorithms. Most importantly, the routing algorithms are implemented within a newly introduced module called the *RingComponent* and joined with the other core modules. Communication between modules, regardless of their physical location, is achieved by means of messaging. In particular, OSIRIS-SR supports synchronous (thread blocking), asynchronous, and publish-subscribe messaging. The transfer of messages is achieved by means of persistent priority queues. Messages between local modules (same VM) are transferred directly (inserted in the corresponding queues), whereas messages between remote modules are serialized and sent via Java TCP Sockets. Following the idea of peer-to-peer systems, all communication in OSIRIS-SR is done by using direct connections among autonomous peers.

3.2 Evaluation Setting

To run the routing enabled OSIRIS-SR framework, we exploit the Amazon Elastic Compute Cloud (Amazon EC2) c1.medium instance type³. We have chosen this instance configuration as it nearly

³<http://aws.amazon.com/ec2/instance-types/>

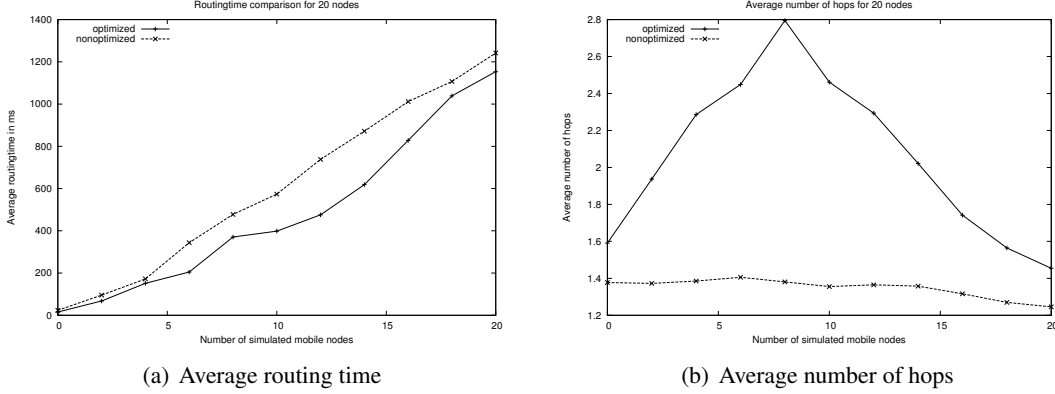


Figure 4: Results of the first real world evaluation (normal distributed delay, 20 instances, steps of two)

corresponds, in terms of CPU power (i.e., two virtual cores with 2.5 EC2 Compute Units each) and main memory (i.e., 2.0 GB) capacity, to the configurations of the latest off-the-shelf mobile devices (smartphones)⁴. As EC2 nodes are merely virtual image instances probably residing at the same EC2 cluster server, latency between EC2 nodes tends to be rather small and constant ($\sim 15ms$). Therefore, in order to simulate heterogeneous node environments we transform EC2 nodes into fluctuating, low bandwidth mobile nodes by introducing random delays at the incoming/outgoing message queues of OSIRIS-SR. In doing so, each routed message at mobile nodes is delayed for a certain amount of time. The delay is subject to a normal distribution with $\mu = 150ms$ and $\sigma = 10ms$, before being sent/received by the Socket layer.

In order to systematically measure the performance impact of COMPASS, we define our heterogeneous node environment to be composed of a certain percentage of mobile nodes M and the rest as default EC2 nodes, i.e., fixed nodes. Together, mobile and fixed nodes add up to the total amount of nodes N . In our case, we use a 20 and 40 node setting (N) and iteratively change the value of M in a separate evaluation run. The initial value of M is always set to be 0% of N and the increase is done in steps of 10 % of N . During an evaluation run, at each node, a routing request is invoked for approximately 200 times by sending generated messages, of the same size, to a random identifier (id) in the ring, i.e., to its associated node ($n \in N$). At the end of each message routing process, the resulting routing time is measured and recorded for analysis. Moreover, having only limited numbers of nodes at disposal, i.e., 20 and 40 we apply a 10 bit, in the former case, and 11 bit, in the latter case, address space. The application of smaller address spaces prevents situations in which the majority of the available nodes is locally stored in the Finger Table and thus is only one hop away.

In this real world evaluation scenario we only compare the routing performance of Chord and COMPASS without interval joining according to [SPS13]. In addition, we analyze the influence of the interval joining feature by evaluating it with different similarity thresholds in the simulated OMNet++ environment (see Section 4).

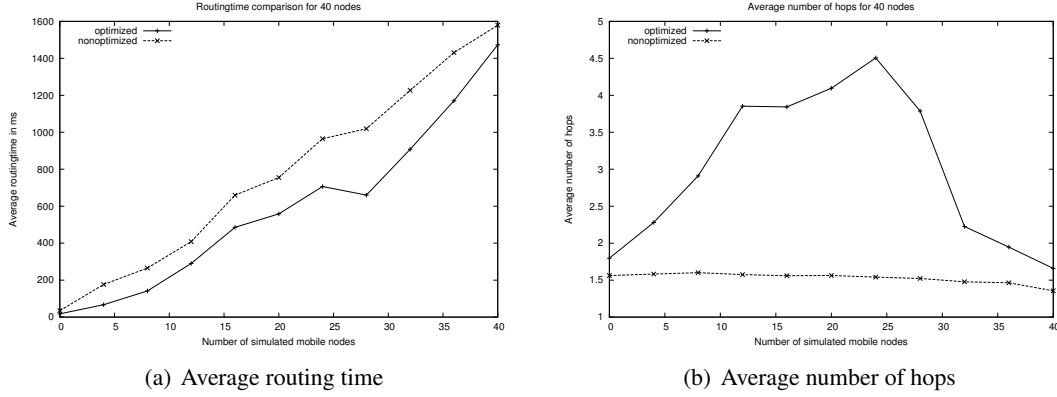


Figure 5: Results of the second real world evaluation (normal distributed delay, 40 instances, steps of four)

M (in percent of N)	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	AVG
20 node setting:												
opt. routingtime in ms	15.6	67.6	151.3	204.7	370.4	398.5	475.6	618.8	827.9	1038.9	1153.2	—
nonopt. routingtime in ms	23.3	95.5	172.2	343.8	477.8	573.5	738	871.7	1011.5	1107.1	1241.9	—
Improvement in %	33.0	29.2	12.1	40.5	22.5	30.5	35.6	29.0	18.2	6.2	7.1	24.0
40 node setting:												
opt. routingtime in ms	17.6	66.9	141.4	290.3	485.75	558.5	706.8	660.1	907.9	1170.6	1473.3	—
nonopt. routingtime in ms	35.1	176.4	265.6	408.7	659.7	755.5	965.1	1019.2	1227.1	1430.7	1579.6	—
Improvement in %	49.9	62.1	46.8	29.0	26.4	26.1	26.8	35.2	26.0	18.2	6.7	32.1

Table 2: Percentual Improvement through COMPASS

3.3 Evaluation Results

In the following, we will analyse the outcomes of the 20 node and 40 node evaluation runs described above. The focus of our analysis lays on the gain of COMPASS, in terms of average latency times, compared to standard Chord, as well as the scalability effects of it. Figure 4 illustrates the evaluation results of the 20 node setting. As expected, COMPASS always outperforms standard Chord, even in the special cases where the environment is completely homogeneous, i.e, $M = 0\%$ or $M = 100\%$. On average, we have computed the gain in latency to be at $\sim 24.0\%$ for the 20 node setting. However, except for the cases of $M = 80 - 100\%$ the gain is usually even higher. In the last three cases, the gain is rather small, 18.2% , 6.2% and 7.1% respectively, and can be explained by the fact that there are less fixed nodes at disposal, which directly results in less latency optimal paths for routing. On the other hand, in the cases of $M = 60 - 70\%$ the average gain in latency is more than 250ms , which means that, on average, the entire time required for an hop over a mobile node is completely saved by using optimal paths. This fact can also be observed in the *Average Number of Hops* diagram in Figure 4(b). Obviously, COMPASS is always faster than standard Chord, at the expense of additional hops taken over fixed nodes, which is exactly what COMPASS is designed for in first place. In some cases, $M = 40\%$ the hop count is twice as high as the rather constant standard Chord hop count. Also in this diagram we can see that the average number of hops is decreasing with the number of fixed nodes starting from $M = 50\%$ as there are less optimal path options at disposal. The reason that this decrease happens starting at $M = 50\%$ can be explained by the fact that at this point the summed up ring responsibility areas

⁴<http://www.cse.wustl.edu/~jain/cse567-11/ftp/smartphn/index.html>

of the mobile nodes outweigh the summed up responsibility areas of the fixed nodes. If the fixed nodes become a minority in the Chord finger tables, which COMPASS uses as possible next hops, there are lesser options for optimal paths which avoids mobile nodes at the expense of more hops and thus the average number of hops decreases.

When scaled up to 40 nodes (results illustrated in Figure 5), we can observe the same behaviour. COMPASS again performs better than standard Chord, in terms of latency gains. However, this time the average gain in latency is even slightly higher and is at $\sim 32.1\%$. Moreover, the average hop count is also slightly higher, for instance in the case of $M = 60\%$ the hop count is three times larger as standard chord is. In some cases the gain (i.e., $M = 0\% \rightarrow 49.9\%$ gain, $M = 10\% \rightarrow 62.1\%$ gain, $M = 20\% \rightarrow 46.8\%$ gain) is significantly higher than the average. This occurrence can be attributed to the fact that not all nodes possess an equal responsibility area in the ring and thus addressing probability, meaning that fixed nodes were probably addressed more frequently. Again, the high mobile nodes cases only slightly benefit from the optimal routing algorithm for the same reason as in the 20 node experiment. Approximately the same as in the scaled down experiment, in the cases of $M = 60 - 90\%$ the gain in latency corresponds to one prevented mobile node hop. On the other hand, although the average hop count starts to decrease later than in the 20 node experiment, it is also here for the same reason.

Finally, based on the observed results we conclude that standard Chord, on average, benefits from COMPASS due to the preserved behaviour and even improved performance with increased scale. Moreover, the evaluation results indicate a best performance node environment setting at $M = 60 - 70\%$. For these particular node heterogeneity settings we definitely recommend the usage of COMPASS over Chord if the best possible average routing times are to be achieved. However, given the linear increase in size of the COMPASS routing tables with N , we expect our solution to scale up to a certain point upon which the sheer sizes of the exchanged COMPASS tables would start to degrade the benefits of our solution. In the next Section, we will present simulation results which confirm that COMPASS is also usable in networks with up to 300 nodes. Moreover, the results indicate that the COMPASS table size can be reduced by means of the interval joining feature. Nevertheless, finding the maximal size, as well as prolonging it to the maximal possible extent is a subject of future research.

4 Simulation in OMNet++ Using OverSim

In this Section, we present both COMPASS' performance characteristics for networks consisting of up to 300 nodes as well as the influence of the interval joining feature. Therefore, we implemented COMPASS in a given simulation environment and defined two different simulation configurations, one for analyzing COMPASS' routing performance like we did in the real world evaluation and one for analyzing COMPASS' scalability features. Using a simulation environment instead of a real world setting consisting of Amazon EC2 instances avoids the time as well as the cost problem. This enables us to evaluate systems with much more nodes and thus underpin the observations we made in the real world evaluation. In addition, the lower time requirement enables us to compare not only COMPASS and Chord but also evaluate the influence of different similarity thresholds. And since OverSim provides different Churn models, we are able to analyze COMPASS' churning behavior.

This Section is organized as follows. Firstly, we will give some information about the implementation. Subsequently, we will explain the simulation settings, i.e., the configuration parameters

and the two different simulation configs which are the basis for the simulations. Finally, we will present the results of five simulations and conclude them.

4.1 Implementation

To simulate COMPASS we use the OverSim [BHK07, Ins13] environment. OverSim is a simulation framework for P2P overlay networks based on the OMNet++ [Var01, OMN13] simulation environment.⁵ Besides several other P2P overlay network implementations OverSim already contains a standard Chord implementation. In order to simulate COMPASS we added the COMPASS extension as presented in Section 2 to the given Chord implementation. Therefore, all existing files were modified and a COMPASS table class was added. Moreover, we added a new application which routes and sends dummy messages (containing simulation time of sent) and measures both, the routing time and the number of hops, for each incoming delivered (via routing) message. The interval in which each node sends a new dummy message and thereby starts a new route request can be modified.

4.2 Simulation Setting

In order to simulate Chord and COMPASS in OverSim, we added two new simulation configs. The main purpose of the first config is to measure and compare COMPASS' and Chord's routing time performance like in the real world evaluation (see Section 3). However, instead of only comparing Chord and COMPASS without joining we also analyze the influence of the interval joining feature for different similarity thresholds. The second config evaluates COMPASS' scalability features and thereby the influence of different similarity thresholds by increasing the total number of nodes in the ring instead of the number of mobile nodes for a fixed total number of nodes (like in the second config).

In the remainder of this Subsection is organized as follows. First, we explain how the simulation environment can be configured using configuration parameters. Subsequently, we explain in detail what the different simulation configs do and how their exact settings can be specified.

4.2.1 Configuration Parameters

The OverSim simulation environment already contained some basic configuration parameters (e.g., the total number of nodes) which can be used to specify the simulation setting for a run or a set of runs. We enriched this set of configuration parameters by new parameters which specify COMPASS' behavior (e.g., the similarity threshold). A list of all important parameters used by our simulation configs is given in the following:

- *sendPeriod*: the interval in which each node sends a dummy message and thereby starts a new route request (e.g., 1s)
- *targetOverlayTerminalNum*: the total number of nodes (has to be greater than 0) (e.g., 20)
- *numMobiles*: the number of mobile nodes (has to be between 0 and the total number of nodes) (e.g., 10)
- *enableCompass*: true if you want to activate COMPASS (e.g., *true*)

⁵OverSim Framework Description: <http://www.oversim.org/>

- *compassDelay*: the interval in which a node should ping its Finger Table nodes (`OptimizationPing()`) in order to update its COMPASS table (see Figure 2) (e.g., 5s)
- *directLatencyMovingAverageAlpha*: the alpha value for the moving average in the direct latency calculation (cf. Equation 1 Section 2.3) (e.g., 0.4)
- *enableCompassIntervalJoining*: true if you want to activate interval joining in COMPASS (only has an effect if also *enableCompass* is set to true) (e.g., *true*)
- *joiningSimilarLatencyPercentageThreshold*: specifies how similar the latencies have to be to allow interval joining (cf. Equation 2 Section 2.3) (e.g., 0.25)
- *transitionTime*: the time the simulation runs without measuring after all nodes joined the network (e.g., 100s)
- *measurementTime*: the measurement time after the transition phase (e.g., 100s)

Moreover, one can enable or disable churn via configuration parameters. OverSim provides the possibility to set different churn types⁶. In this report, we want to concentrate on two options:

- **NoChurn**: This type creates a static network. Hence, there are no joining or leaving nodes.
- **LifetimeChurn**: Each node has a lifetime. After the lifetime is passed the node leaves the network and a new node joins. The lifetime is randomly distributed for a given *lifetimeMean*.

Please note: If you activate churn your simulation has to run much longer. If a node would leave after less than 100 seconds neither the Chord finger nor the COMPASS tables would have enough time to converge. Our experiments showed that a measurement time of 1000 seconds and a lifetime mean of 500 seconds works well.

4.2.2 Routing Time Performance: Increasing Number of Mobile Nodes and a Fixed Total Number of Nodes

The first config executes several runs for a different number of mobile nodes but with a fixed total number of nodes. This is done for six settings:

1. Standard Chord
2. COMPASS without interval joining
3. COMPASS with interval joining (similarity threshold = 0.2)
4. COMPASS with interval joining (similarity threshold = 0.4)
5. COMPASS with interval joining (similarity threshold = 0.6)
6. COMPASS with interval joining (similarity threshold = 0.8)

As already mentioned, the idea of this config is to do the same as we did in the real world evaluation for each of these six settings. The total number of nodes (N) and the number of mobile nodes (M) can be specified by modifying the following configuration parameters:

⁶List of all OverSim churn types: <http://www.oversim.org/wiki/OverSimChurn>

```

**.numMobiles = ${mobiles=0..100 step 10}
**.targetOverlayTerminalNum = ${numNodes=100}

```

Please note that the number of mobile nodes should be bounded by 0 and the total number of nodes ($M \in [0, N]$). Furthermore, all the properties listed in Section 4.2.1 can be specified.

Since the purpose of this config is to compare the six different settings, several important values are logged during the runs and the following averages are calculated for each combination (number of mobiles and setting):

- Average routing time in seconds: how many seconds does a route call take in average
- Average number of hops: how many hops does a route call take in average
- Average COMPASS table size: how many entries does the COMPASS table have in average (only for COMPASS)
- Sent OptimizationPing Messages/s: number of sent pings and received responses per simulated second (only for COMPASS)
- Sent OptimizationPing Bytes/s: number of sent and received bytes (both pings and responses) per simulation second (only for COMPASS)

The given example config settings (see also Section 4.2.1) yield the following simulation: There are 100 (N) nodes in the network and the number of mobile nodes (M) starts at 0 and has a step size of 10. For each number of mobile nodes there are six runs, one for each setting listed above. In each run all the measurements are done for 100 seconds (*measurementTime*) after a transition time of 100 seconds. That means for each run, that after all nodes joined the network the simulation runs for 100 seconds without measuring before starting the measurement phase which takes 100 seconds. During the measurement phase each node starts a routing process by sending a message every second (*sendPeriod*). After this phase the simulation stops.

4.2.3 Scalability: Increasing Total Number of Nodes and a Fixed Number of Mobile Nodes

The purpose of the second config is to evaluate how COMPASS scales when the total number of nodes increases. This is done for five settings:

1. COMPASS without interval joining
2. COMPASS with interval joining (similarity threshold = 0.1)
3. COMPASS with interval joining (similarity threshold = 0.2)
4. COMPASS with interval joining (similarity threshold = 0.3)
5. COMPASS with interval joining (similarity threshold = 0.4)

Since we suppose the mobile nodes percentage to be irrelevant for the scalability property, we set the number of mobile nodes (M) constant to 0 while increasing the total number of nodes (N) in each run. The boundaries for the scalability simulation can be specified by modifying the *targetOverlayTerminalNum* parameter:

```

**.targetOverlayTerminalNum = ${numNodes=10..150 step 10}

```

To figure out how good COMPASS scales we concentrate on the following three properties:

- Average COMPASS table size: how many entries does the COMPASS table have in average
- Sent OptimizationPing Messages/s: number of sent pings and received responses per simulated second
- Sent OptimizationPing Bytes/s: number of sent and received bytes (both pings and responses) per simulation second

The simulation constructed by this config with the given example parameters is very similar to the simulation described in the previous Section. The only difference is that instead of increasing the number of mobile nodes (M) while keeping a constant total number of nodes (N), this time the total number of nodes is increased (from 10 to 150 with a step size of 10). Moreover, we do not simulate standard Chord in this simulation config since we are only interested in analyzing the scalability of the COMPASS specific properties (e.g., the COMPASS table size) and comparing them for different similarity thresholds.

4.3 Simulation Results

To extend and underpin the observations we made in the real world evaluation, we ran five additional simulations: Two routing time performance simulations (see Section 4.2.2) without churn (100 and 250 nodes), a smaller one (40 nodes) with activated churn and two scalability simulations (see Section 4.2.3) without churn (150 and 300 nodes). In the following Subsections we will present and analyze the outcomes of these five simulations.

4.3.1 Routing Time Comparison (100 Nodes)

The first simulation is an expansion of the real world evaluation. The simulation expands the scenario by two aspects. Firstly, the number of nodes in the network is increased from 40 to 100 nodes and thus multiplied by a factor of 2.5. Secondly, in the simulation we do not only simulate and compare standard Chord and COMPASS without interval joining like done in the real world evaluation, but we also simulate COMPASS with interval joining with four different similarity thresholds (i.e., 0.2, 0.4, 0.6 and 0.8). Thus, we can also analyze the influence of the interval joining feature w.r.t. the routing time performance.

The simulation was executed on a 13" Mid-2013 MacBook Air (1.3 GHz Core Intel i5, 8GB 1600MHz DDR3 Ram) in OSX and took about 63 minutes. This comparatively small execution time on normal consumer hardware is a huge improvement to the real world evaluation setting where we need to pay for many Amazon EC2 instances and wait for several hours⁷ to obtain new evaluation results.

Table 3 lists all important parameter values used in this simulation. The simulation matches the example given in Section 4.2.2. In a nutshell there are six different settings (i.e., COMPASS-joining-value combinations, see Section 4.2.2 for a list). For each of these settings there is one simulation run for every number of mobile nodes ($M = 0, 10, \dots, 100$). Hence, in total there are 66 runs in this simulation config. In each run there is a 100 second long transition phase after all nodes joined, followed by a measurement phase of 100 seconds.

⁷Remember that evaluations performed on Amazon EC2 instances run in real time and thus take several hours.

parameter name	value
churnGeneratorTypes	<i>oversim.common.NoChurn</i>
routingType	<i>semi-recursive</i>
sendPeriod	1s
numMobiles	$\${mobiles=0..100\ step\ 10}$
targetOverlayTerminalNum	$\${numNodes=100}$
enableCompass	$\${enableCompass=false, true, true, true, true, true}$
compassDelay	5s
directLatencyMovingAverageAlpha	0.4
enableCompassIntervalJoining	$\${enableJoining=false, false, true, true, true, true\ !enableCompass}$
joiningSimilarLatencyPercentageThreshold	$\${joiningThreshold=0.0, 0.0, 0.2, 0.4, 0.6, 0.8\ !enableCompass}$
transitionTime	100s
measurementTime	100s
constantDelay (stationary)	15ms
constantDelay (mobile)	150ms
jitter (stationary)	0.0
jitter (mobile)	0.1

Table 3: Simulation Configuration Parameters for the Small Routing Time Comparison (100 Nodes)

Figure 6 illustrates the evaluation results in five graphs. Considering only standard Chord and COMPASS without joining in Figure 6(a), the results are very similar to the real world results (see Section 3). As expected COMPASS outperforms Chord if there are mobile nodes in the network, albeit by increasing the number of hops. This confirms that COMPASS is also applicable in larger networks and further can be considered as a verification of the appropriateness of the simulation setup.

However, there is one difference to the real world evaluation graphs (see Figure 4 and 5). In the real world evaluation graphs the number of hops is always lower in the standard Chord case than in the COMPASS case. This is exactly the behavior we expected. But in the simulation graph (see Figure 6(b)) this is not the case. This indicates an unexplainable artefact. Nevertheless, since all the other results can be explained (e.g., the average number of hops is approximately constant when used standard Chord) we assume that this is due to a systematic error of measurement or a little different routing behavior in the OverSim Chord implementation.

Moreover, the simulation graphs additionally provide information about the interval joining feature. As one can see in Figure 6(a) and 6(b) using COMPASS with the similarity thresholds 0.2 and 0.4 yield a very similar routing performance to COMPASS without joining. Using 0.6 as the similarity threshold results in a worse routing time performance than COMPASS without joining, but still in a better performance than standard Chord. In contrast using 0.8 results in a significantly worse performance compared to the other COMPASS performances, especially if the mobile nodes percentage is 80% or higher. In this case COMPASS' routing performance is even worse than the standard Chord performance.

The remaining three Subfigures show the influence of the similarity threshold w.r.t. the COMPASS table size and the network traffic introduced by COMPASS. Figure 6(c) shows how much routing intervals could be joined, i.e., how strong the COMPASS tables could be compressed. The graph indicates that the higher the similarity threshold is, the more intervals could be joined and thus the smaller the resulting COMPASS table is. Furthermore, the compression rate is rather constant w.r.t. the increasing mobile nodes percentage. Figure 6(d) shows how many COMPASS specific messages are sent and received per node and second. This value is rather similar for different similarity thresholds and mobile nodes percentages (around 2.65). This observation meets our expectations, since the number of sent and received `OptimizationPing()`

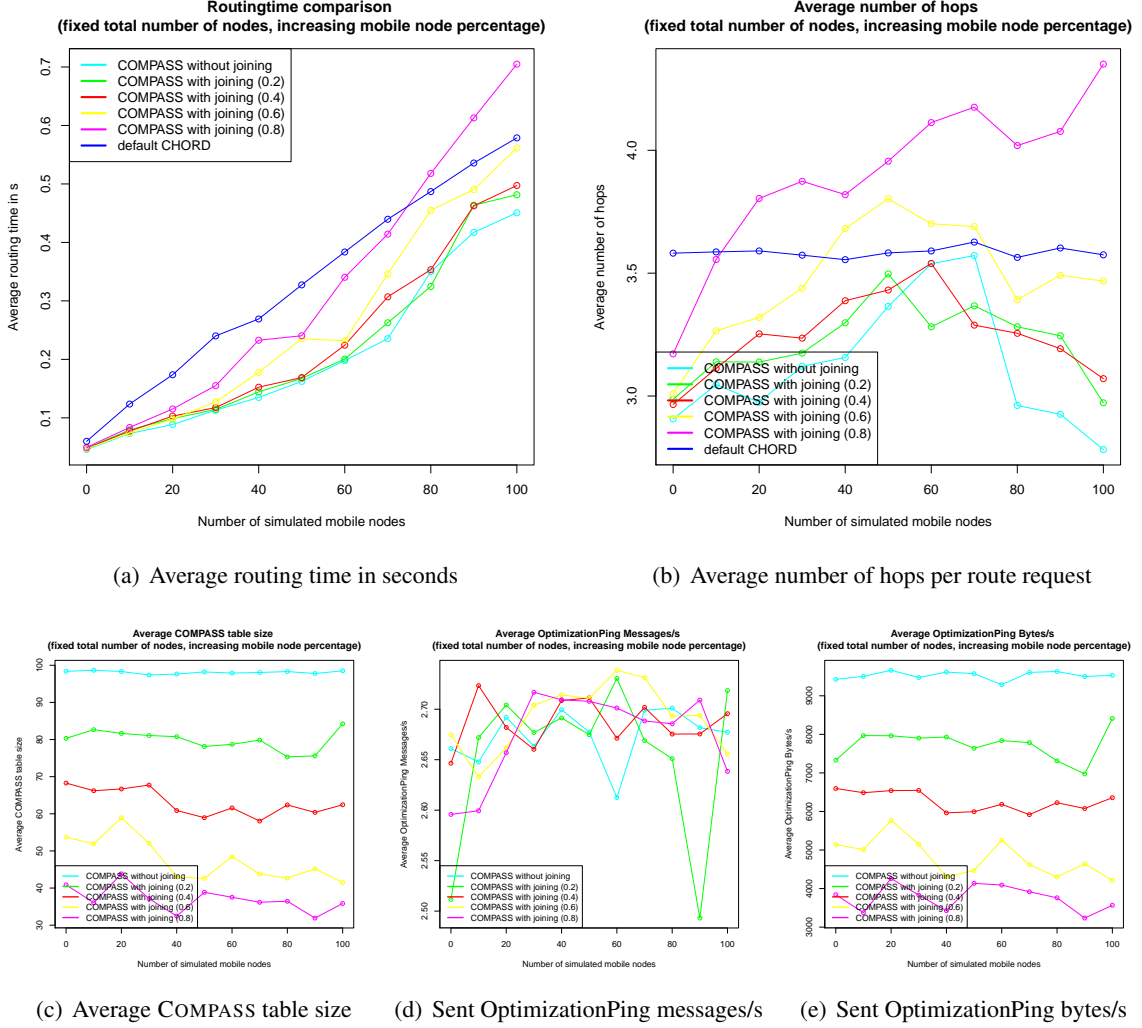


Figure 6: Small Routing Time Comparison Results (100 nodes, a step size of 10 and no churn)

messages should only depend on the total number of nodes and the COMPASS table update time period (see Figure 2). Finally, Figure 6(e) shows the network traffic (i.e., `OptimizationPing()` bytes per node and second) introduced by COMPASS table update algorithm. This graph has the same properties as the COMPASS table size since the number of `OptimizationPing()` messages is constant and the size of each `OptimizationPing()` message is either a small constant value (ping) or dependent of the COMPASS table size (pong). Hence, using a higher similarity threshold results in less network traffic caused by COMPASS' maintenance strategy.

4.3.2 Routing Time Comparison (250 Nodes)

The second simulation is a further increase in size in order to verify COMPASS' routing time performance in larger networks. Table 4 lists all config parameter values used in this simulation. Nearly all parameters are exactly the same as in the previous simulation. Thus, the process is the same as the one presented in Section 4.3.1. The only difference is that instead of only 100 nodes

parameter name	value
churnGeneratorTypes	<i>oversim.common.NoChurn</i>
routingType	<i>semi-recursive</i>
sendPeriod	1s
numMobiles	$\${mobiles=0..250 \text{ step } 25}$
targetOverlayTerminalNum	$\${numNodes=250}$
enableCompass	$\${enableCompass=false, true, true, true, true, true}$
compassDelay	5s
directLatencyMovingAverageAlpha	0.4
enableCompassIntervalJoining	$\${enableJoining=false, false, true, true, true, true !enableCompass}$
joiningSimilarLatencyPercentageThreshold	$\${joiningThreshold=0.0, 0.0, 0.2, 0.4, 0.6, 0.8 !enableCompass}$
transitionTime	100s
measurementTime	100s
constantDelay (stationary)	15ms
constantDelay (mobile)	150ms
jitter (stationary)	0.0
jitter (mobile)	0.1

Table 4: Simulation Configuration Parameters for the Large Routing Time Comparison (250 Nodes)

and a step size of 10, this time the network contains 250 nodes and the number of mobile nodes is increased by 25 in each step.

Figure 7 shows the results of this simulation in five graphs. Firstly, the results confirm that COMPASS also outperforms Chord in a large network of 250 nodes. Moreover, Figure 7(a) shows that the routing time performance of COMPASS with a similarity threshold of 0.2 and 0.4 is approximately the same as the routing time performance of COMPASS without joining. The performance with a 0.6 threshold is significantly worse than COMPASS without joining but still better than standard Chord with the exception of the run with 100% mobile nodes. Further the graph indicates that using COMPASS with a similarity threshold of 0.8 either does not yield an appreciable improvement compared to Chord (mobile nodes percentage $\leq 70\%$) or even to a significant performance degradation. Figure 7(b) underpins these observations. Further there is the same error with the standard Chord curve as in the number of hops graph of the first simulation.

The other three graphs have similar properties as those of the previous simulation. Though there are two differences. Firstly, the number of messages per second (see Figure 7(d)) seems to increase up to the 100 mobile nodes run before being approximately constant, instead of being constant from begin on like in Figure 6(d). As already mentioned in Section 4.3.1 the number of sent and received `OptimizationPing()` messages per second and node should be independent of the percentage of mobile nodes in the ring. Hence, we have no clear explanation for this effect at the moment. The only possible explanation we found, is that the high oscillation in the graph randomly generates the appearance that the value increases in the beginning. Secondly, the COMPASS table size and according to that also the traffic (see Figure 7(c) and 7(e)) increases up to a mobile nodes percentage of 50% when using COMPASS with 0.4 as its similarity threshold. The COMPASS table size does not depend on the number of mobile nodes. However, for joining two nodes it is indispensable that two consecutive intervals in the COMPASS table have the same next hop and a similar aggregated latency. Introducing new mobile nodes can change the COMPASS tables and thus allowing new joins or prohibiting old joins. Normally this should result in some random oscillations. However, this can also result in an effect like described above.

Considering the results of the first two simulations (i.e., the both routing time comparisons without churn) yields the assumption that w.r.t. the routing time performance, 0.4 could be the perfect similarity threshold. However, we further have to check how the different similarity thresholds

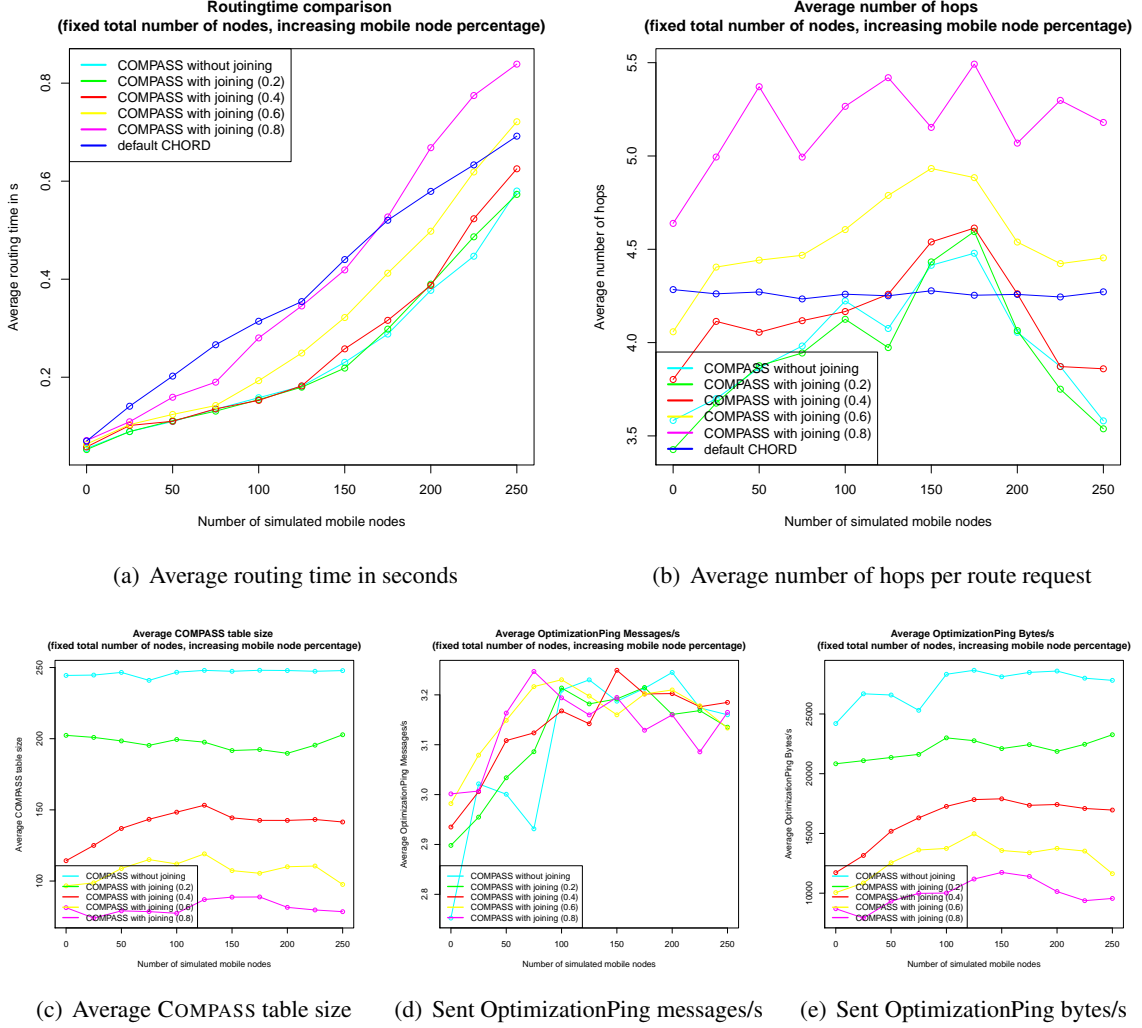


Figure 7: Large Routing Time Comparison Results (250 nodes, a step size of 25 and no churn)

influence COMPASS' scalability characteristics (see Section 4.3.4 and 4.3.5) and how they influence COMPASS' performance if the network is unstable (see Section 4.3.3).

4.3.3 Routing Time Comparison with Churn (40 Nodes)

The purpose of the third routing time performance simulation is to analyze how COMPASS performs if the underlying network is not stable but has a high churn rate, which is reflected in the parameter settings listed in Table 5. The major difference is that we enabled LifetimeChurn in this evaluation. This means that in difference to the normal NoChurn setting, in which a node stays in the network once it joined, nodes join the environment, stay for some time and leave after their lifetime has passed. Our experiments showed that a good setting for evaluating Chord and COMPASS with churn is a lifetime mean of 500 seconds and a measurement time of 1000 seconds. The usage of shorter lifetime mean values (e.g., 100 seconds) resulted in so many joining and leaving nodes that we did not get any usable results. Since the measurement time is very high we were

parameter name	value
churnGeneratorTypes	<i>oversim.common.LifetimeChurn</i>
lifetimeMean	500s
routingType	<i>semi-recursive</i>
sendPeriod	1s
numMobiles	$\${mobiles=0..40 \text{ step } 4}$
targetOverlayTerminalNum	$\${numNodes=40}$
enableCompass	$\${enableCompass=false, true, true, true, true, true}$
compassDelay	5s
directLatencyMovingAverageAlpha	0.4
enableCompassIntervalJoining	$\${enableJoining=false, false, true, true, true, true !enableCompass}$
joiningSimilarLatencyPercentageThreshold	$\${joiningThreshold=0.0, 0.0, 0.2, 0.4, 0.6, 0.8 !enableCompass}$
transitionTime	100s
measurementTime	1000s
constantDelay (stationary)	15ms
constantDelay (mobile)	150ms
jitter (stationary)	0.0
jitter (mobile)	0.1

Table 5: Simulation Configuration Parameters for the Routing Time Comparison with Churn (40 Nodes)

forced to reduce the total number of nodes to 40 and the step size for mobile node increasing to 4 in order to obtain results in reasonable time. Using these parameters the simulation could be executed on a Mid-2013 13" MacBook Air in OSX in about 51 minutes.

Figure 8 illustrates the results of this simulation. As one can see especially in Figure 8(a) and 8(b), introducing Churn to the environment also introduces large fluctuations and variations to the results. But apart from these variations the trends in the routing time graph remain the same. Apart from some individual exceptions, which are probably conditioned by the strong fluctuations, the influences of the different similarity thresholds are rather the same as in the large routing time performance simulation without churn (see Section 4.3.2).

Besides the strong fluctuations there are two additional effects. Firstly, in difference to the last two evaluations in which the number of hops is maximal in the runs with a mobile nodes percentage of 60-70%, in this simulation the number of hops keeps raising with the number of mobile nodes until all nodes are mobile. We assume that this effect is only introduced randomly by the strong fluctuations. We will take a closer look to this effect in our future work. Secondly, Figure 8(c) and thus also Figure 8(d) shows an anomaly. The average table size for COMPASS without joining is above 80 although there are only 40 nodes in the network. Normally the average COMPASS table size should match the number of nodes in the ring (see Section 2.3). For both previous simulations this was true. However, this is only true for stable networks without churn. If there is churn, not all nodes have an up-to-date knowledge about the ring topology. This problem is not COMPASS specific, but also appears in Chord. The outdated knowledge of the topology leads to COMPASS tables with different intervals at the different nodes. The COMPASS table maintenance algorithm splits the intervals and this results in COMPASS table sizes above the number of nodes in the ring. This problem can only be solved by the interval joining feature. Hence, the interval joining feature has another positive effect besides its original purpose.

In particular this simulation confirms that we can use COMPASS and the interval joining feature in every network, i.e., one does not have to deactivate COMPASS if there is possibly churn in the network. This is due to the fact that Chord is corrupted in the same way as COMPASS if there is churn. Thus, if one uses Chord, one can also use COMPASS without risking any negative impact.

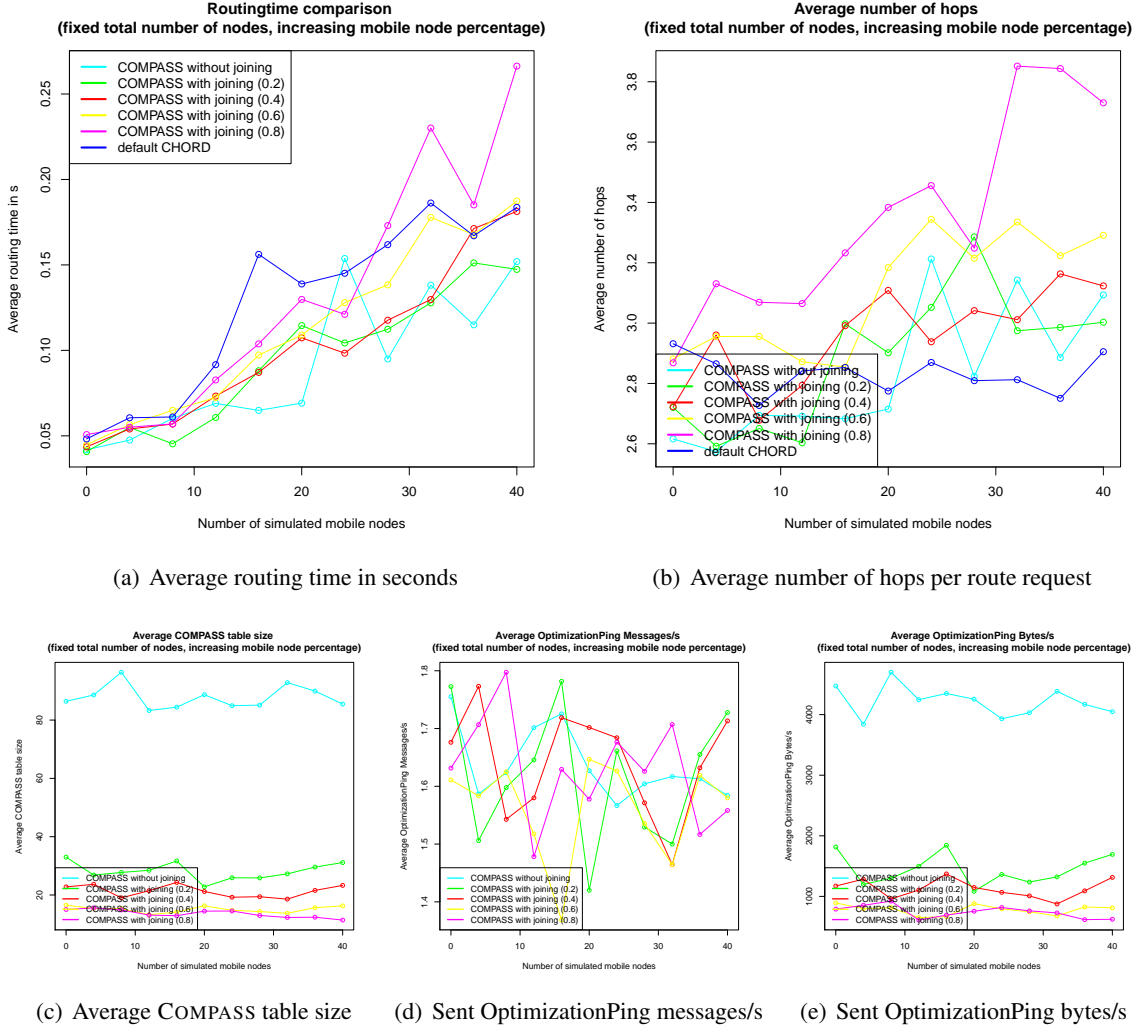


Figure 8: Routing Time Comparison Results with Churn (40 nodes, a step size of 4 and lifetime churn)

4.3.4 Scalability (up to 150 Nodes)

This simulation differs from the three previous presented simulations. Instead of analyzing COMPASS' routing time performance, the purpose of this simulation is to study out COMPASS' scalability properties. Further the target is to evaluate the influence of the similarity threshold on the scalability behavior. For this purpose we increase the total number of nodes in the network in each run instead of increasing the mobile nodes percentage. Apart from that and the fact that we do not simulate standard Chord, the simulation process is the same as in the routing time performance simulation (see also Section 4.2.3).

Evaluating COMPASS' scalability features in the real world scenario by using Amazon EC2 instances, would be very complex since each increase of the total number of COMPASS nodes would require to launch and setup new EC2 instances, which is possible in general (due to the cloud elasticity property) but does not pay off for short running experiments. In contrast in the

parameter name	value
churnGeneratorTypes	<i>oversim.common.NoChurn</i>
routingType	<i>semi-recursive</i>
sendPeriod	1s
numMobiles	$\{\text{mobiles}=0\}$
targetOverlayTerminalNum	$\{\text{numNodes}=10..150 \text{ step } 10\}$
enableCompass	$\{\text{enableCompass}=\text{false}, \text{true}, \text{true}, \text{true}, \text{true}, \text{true}\}$
compassDelay	5s
directLatencyMovingAverageAlpha	0.4
enableCompassIntervalJoining	$\{\text{enableJoining}=\text{false}, \text{false}, \text{true}, \text{true}, \text{true}, \text{true} \text{ !enableCompass}\}$
joiningSimilarLatencyPercentageThreshold	$\{\text{joiningThreshold}=0.0, 0.0, 0.2, 0.4, 0.6, 0.8 \text{ !enableCompass}\}$
transitionTime	100s
measurementTime	100s
constantDelay (stationary)	15ms
constantDelay (mobile)	150ms
jitter (stationary)	0.0
jitter (mobile)	0.1

Table 6: Simulation Configuration Parameters for the Small Scalability Evaluation (up to 150 Nodes)

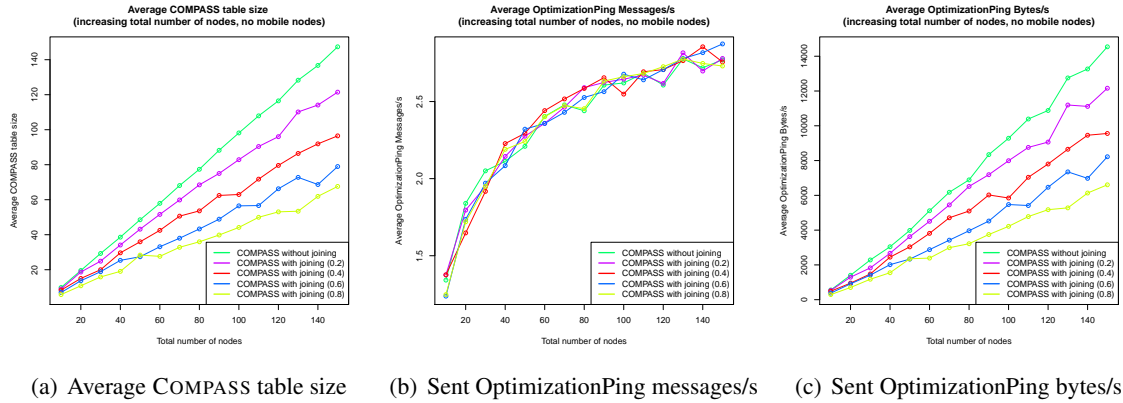


Figure 9: Small Scalability Evaluation Results (up to 150 nodes, a step size of 10 and no churn)

OMNet++ simulation environment this simulation required only some minor modifications in the simulation config file (see Table 6). Furthermore, the simulation took only about 79 minutes on the 13" Mid-2013 MacBook Air in OSX. This rather short execution time, the little setup effort and the negligible costs are the enormous advantages of the new simulation environment over the real live scenario.

Figure 9 shows the simulation results in three graphs. As one can see in Figure 9(a) the average COMPASS table size scales linearly, but with different gradients. The higher the similarity threshold is, the smaller the COMPASS table is. The same is true for the network traffic introduced by COMPASS (see Figure 9(c)). Hence, using the interval joining feature can significantly reduce the traffic introduced by COMPASS, especially in large networks. This is highly beneficial since we suppose the additional introduced traffic to be a major limitation of COMPASS (see Section 3.3). In contrast to the table size and traffic, the average number of sent `OptimizationPing()` messages per second scales in a different way (see Figure 9(b)). We suppose that the number of messages scales logarithmically to the total number of nodes since the number of sent messages (and thereby indirect also the number of received messages) depends only on the number of different nodes in the Chord Finger Table. The more nodes are in the network, the more different nodes

parameter name	value
churnGeneratorTypes	<i>oversim.common.NoChurn</i>
routingType	<i>semi-recursive</i>
sendPeriod	1s
numMobiles	$\{\text{mobiles}=0\}$
targetOverlayTerminalNum	$\{\text{numNodes}=10..300 \text{ step } 10\}$
enableCompass	$\{\text{enableCompass}=\text{false}, \text{true}, \text{true}, \text{true}, \text{true}, \text{true}\}$
compassDelay	5s
directLatencyMovingAverageAlpha	0.4
enableCompassIntervalJoining	$\{\text{enableJoining}=\text{false}, \text{false}, \text{true}, \text{true}, \text{true}, \text{true} \text{ !enableCompass}\}$
joiningSimilarLatencyPercentageThreshold	$\{\text{joiningThreshold}=0.0, 0.0, 0.2, 0.4, 0.6, 0.8 \text{ !enableCompass}\}$
transitionTime	100s
measurementTime	100s
constantDelay (stationary)	15ms
constantDelay (mobile)	150ms
jitter (stationary)	0.0
jitter (mobile)	0.1

Table 7: Simulation Configuration Parameters for the Large Scalability Evaluation (up to 300 Nodes)

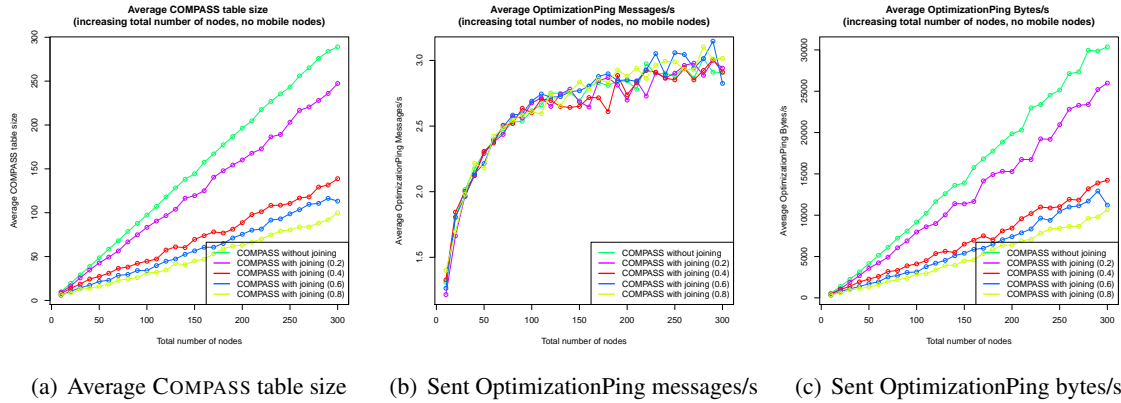


Figure 10: Large Scalability Evaluation Results (up to 300 nodes, a step size of 10 and no churn)

can be in the Chord Finger Table. Additional to the logarithmic scaling property, the increase is also capped by the key length in bits, since there cannot be more different nodes than Finger Table entries.

4.3.5 Scalability (up to 300 Nodes)

This simulation increases the extent of the scalability evaluation. Table 7 lists all important parameter values used in this simulation. The only differences to the previous simulation are that the total number of nodes in the network is not only increased up to 150 but up to 300 nodes and the fact that this simulation was executed on a more powerful machine using four CPU cores.

Figure 10 illustrates the simulation results. The graphs underpin the observations and scaling assumptions made in Section 4.3.4. In particular Figure 10(b) confirms the assumption of the logarithmic scale capped by the key length. Moreover, Figure 10(a) and 10(c) give interesting information about the influence of the different similarity thresholds. Using interval joining with 0.2 as the similarity threshold only slightly reduces the COMPASS table size compared to COMPASS without joining. Increasing the similarity threshold to 0.4 results in a significant reduction

of the COMPASS table size and thus also in a reduction of the network traffic. In contrast, further increasing the similarity threshold to 0.6 or even 0.8 only yields marginal improvements.

Remember that the first two simulations (see Section 4.3.1 and 4.3.2) showed that using the similarity threshold 0.4 results in a routing time performance similar to the performance of COMPASS without joining. Hence, using 0.4 as the threshold for interval joining in COMPASS results in both, an optimal routing time performance and a huge traffic reduction. On these grounds we recommend using the interval joining feature with 0.4 as its similarity threshold for all networks.

4.3.6 Conclusion

Implementing COMPASS and its interval joining feature in the OMNet++ simulation environment using OverSim enabled us to evaluate new scenarios which we could not run in the real world setting (OSIRIS-SR and Amazon EC2 instances). For instance, the possibility to use the different predefined Churn types provided by OverSim, allowed us to analyze how COMPASS performs in unstable networks. Moreover, OverSim’s run configuration model enabled us to construct new evaluation scenarios (like the scalability simulations) with very little effort. The execution times presented in this reports further showed, that simulating the scenarios in OMNet++ on normal consumer hardware is much less time consuming than executing the code on several Amazon EC2 instances like done in the real world evaluation (see Section 3). Moreover, simulating the scenarios on consumer hardware yields negligible costs compared to those for running multiple Amazon instances for several hours. These facts make the OverSim implementation to the perfect test and evaluation environment, especially for large networks.

The already realized simulations, which we presented in this report, have confirmed the observations we made in the real world scenario and give some further information about COMPASS’ excellence. Firstly, the simulation results showed that COMPASS also outperforms standard Chord in larger networks with up to 250 nodes. Moreover, they indicate that COMPASS can be used in unstable networks with a high churn rate. Thus, one can enable COMPASS in all networks without fearing any bad surprises.

In addition, the COMPASS interval joining feature could reduce the COMPASS table size and thus the additional traffic introduced by COMPASS, which is a major limitation of COMPASS, while preserving the routing time performance if the similarity threshold is not too high. Considering our simulations, we have identified 0.4 to be the best similarity threshold value. Furthermore, the interval joining feature solves unwished interval splittings induced by unstable networks with churn. Hence, we recommend always enabling the interval joining feature and using an similarity threshold value around 0.4.

5 Related Work

Since the beginnings of Chord, the need to enhance its core properties has been widely accepted, especially when it comes to applying Chord to dynamic, unreliable and/or heterogeneous environments such as mobile ad-hoc networks (*MANETs*). The first Chord enhancement techniques, much like COMPASS, are also based on the idea of considering latency as to minimize routing times. The approaches based on *Proximity Neighbour Selection* (PNS) [CDCR02, DLS⁺04] and *Proximity Route Selection* (PRS) [MWSP08] are the most prominent ones, at which PNS on average outperforms the others, in terms of latency and network resource consumption [DLS⁺04]. However, PNS-based approaches [CDCR02, DLS⁺04], are in general relying only on latency probing of

random candidate peers [MJB05] for the construction of Finger Tables. In this way, the standard Chord algorithm is significantly influenced, and even more, not all possible nodes are considered for minimal latency paths. Instead, in COMPASS, Chord is rather complemented with optimal paths, and based on distance vector techniques, every node is eventually probed. Moreover, PNS approaches are less applicable to MANETs due to the volatile nature of such environments (e.g., high and varying packet losses and delays, caused by collisions and interference among nodes). Nevertheless, [SGF02] suggested the similarities of P2P environments and MANETs, expressing the need to combine both, due to the same problems at hand, in terms of self-organization, scalability and robustness. Advocated solutions in [SGF02] include, just like in COMPASS, integrations of Chord with proven networking protocols such as DSDV [PB94]. Hence, guided by the aspiration of optimal and scalable content-centric node routing in MANETs, rather than the static address-centric one, the various approaches of [FDKC06, MJ07, LWW10, FY09, HGRW06, ZS06, PDH04] integrate DHT substrates, among others Chord, on top of networking layers. Commonly, in the works of [MJ07, LWW10, FY09, HGRW06] the network layer routing technique utilized for Chord hops, in terms of optimal paths, is AODV [PBRD03]. Here, the Chord Finger Table is consulted first as to find the next hop; afterwards the AODV routing tables are consulted for optimal physical routes. Instead, our approach works the other way around, in consulting the COMPASS table first, complemented on top of the standard Chord Finger Tables. This way, latency optimal paths are utilized first at the expense of additional hops than needed by standard Chord. In the case of non-existing optimal paths, our approach can always resort to existing standard Chord. In [MJ07, FY09] complex *Random Landmarking* [WZS04] algorithms are used for the Finger Table construction of physically proximate nodes, which are unnecessary in our case as physically proximate nodes converge from optimal paths. Moreover, in terms of Finger Table maintenance, [MJ07] and [LWW10] rather focus on minimizing the produced network traffic by reducing the number of data exchanging nodes to only two (i.e., predecessor and successor) and overhearing/piggybacking traversing messages which results in slower optimal path convergence as compared to our COMPASS tables. Additionally, in [HGRW06] network traffic reduction is an important topic and it is achieved by means of multicast messaging of AODV. On the other hand, [FDKC06] uses *Dynamic Source Routing* [JM96] techniques in conjunction with caching strategies as to find optimal physical paths between Chord hops, rather than AODV. However, such an approach is characterized by only limited knowledge of optimal physical paths, i.e., of only frequently addressed node paths and does not perform optimally under high churn rates. Finally, [ZS06, PDH04] leverage Pastry substrates, instead of Chord, for scalable content-centric routing, applying on the network layer DSR [PDH04] and AODV [ZS06] techniques, again with the same deficiencies as mentioned above.

6 Conclusion

Heterogeneous P2P environments require special solutions if pervasive data management technologies, such as Chord-based DHTs, are to be used in scenarios where efficiency, in terms of latency, is a crucial requirement (e.g., in emergency management). In such scenarios, it is beneficial to deviate from standard Chord principles, such as optimized hop count, for the sake of minimized latency and thus efficiency in data access. Therefore, in this report, we have presented COMPASS, a latency optimized data access algorithm for Chord-based heterogeneous P2P networks composed of mobile and fixed nodes. Essentially, COMPASS leverages the more powerful fixed nodes, in

terms of latency characteristics, for data access as much as possible. Latency optimal data traffic paths over fixed nodes are dynamically determined, regardless of the number of necessary hops, by means of Distance Vector Routing techniques (by means of dedicated COMPASS tables) on top of Chord Finger Tables.

However, COMPASS' huge benefit, i.e., its optimal routing w.r.t. latency instead of hop count, does not come for free. The price for this gain in latency is an increase in the overall complexity. As shown in Section 2, using COMPASS introduces additional effort for storing and maintaining the COMPASS table which is the basis for the latency optimal routing. Both the complexity for storing the COMPASS table as well as the runtime complexity for maintaining it depends on the number of intervals, i.e., on the size of the COMPASS table. Moreover, the additional traffic introduced by COMPASS depends on this size. Hence, the COMPASS table size is its major scaling limitation.

To address this issue, we introduced the novel interval joining feature, which reduces the COMPASS table size. The similarity threshold parameter which controls the joining behavior introduces a trade-off between high precision and high scalability. The perfect similarity threshold depends on the system environment and finding it is a nontrivial problem.

Real world evaluation as well as simulation results show that COMPASS outperforms Chord in terms of routing time at the expense of additional hops. The simulation results show that COMPASS can be used in large networks with up to 300 nodes as well as in unstable networks with a high churn rate. Furthermore, the results show that the interval joining feature reduces the COMPASS table size while preserving the routing time performance as long as the similarity threshold is not too large. Based on our simulations, we recommend always enabling interval joining and using a similarity threshold around 0.4. In addition, the interval joining feature solves unwished interval splittings caused by joining or leaving nodes.

In future work, we plan to evaluate the scalability characteristics of COMPASS to the biggest possible extent by analyzing the point at which the COMPASS tables maintenance effort, in terms of additional traffic, consumed memory and computational complexity, overweighs the advantages in routing times over Chord. Furthermore, we will construct and analyze further simulations to address open issues and test new ideas how to improve COMPASS. In addition, we will concentrate on finding the reason for the systematical error in the number of hops graphs when using standard Chord. Moreover, we will continue to focus on minimizing the additional traffic introduced by COMPASS to further improve the scalability features. In the next step, we plan to focus on traffic friendly update strategies. Possible strategies are to omit sending COMPASS tables without considerable changes and the usage of incremental updates. Finally, we aim at further improving the COMPASS data access protocol and the applicability of it in the context of mobile environments by adapting it to streaming data, i.e., to applications that continuously produce and process data.

7 Acknowledgment

This work has been partly funded by the Swiss National Science Foundation, project SOSOA and supported by an amazon AWS Grant. Our special thanks go to the OverSim team⁸, whose work builds the foundation for our simulations.

⁸<http://www.oversim.org/wiki/OverSimTeam>

References

- [BHK07] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A flexible overlay network simulation framework. In *Proceedings of 10th IEEE Global Internet Symposium (GI '07) in conjunction with IEEE INFOCOM 2007, Anchorage, AK, USA*, pages 79–84, May 2007.
- [Bla00] Uyless Black. *IP Routing Protocols: RIP, OSPF, BGP, PNNI and Cisco Routing Protocols*. Prentice Hall, 2000.
- [CDCR02] Miguel Castro, Peter Druschel, Y. Charlie, and Hu Antony Rowstron. Exploiting Network proximity in Peer-To-Peer Overlay Networks. Technical report, Microsoft Research, 2002.
- [DLS⁺04] Frank Dabek, Jinyang Li, Emil Sit, et al. Designing a DHT for Low Latency and High Throughput. In *Proceedings of the 1st NSDI*, pages 85–98, 2004.
- [FDKC06] Thomas Fuhrmann, Pengfei Di, Kendy Kutzner, and Curt Cramer. Abstract Pushing Chord into the Underlay: Scalable Routing for Hybrid MANETs. Technical Report 2006-12, University of Karlsruhe, 2006.
- [FY09] Sonia Gaied Fantar and Habib Youssef. Locality-Aware Chord over Mobile Ad Hoc Networks. In *Proc. GIIS*, June 2009.
- [HGRW06] Tobias Heer, Stefan Gotz, Simon Rieche, and Klaus Wehrle. Adapting Distributed Hash Tables for Mobile Ad Hoc Networks. In *Proc. PerCom 2006 Workshops*, Pisa, Italy, March 2006.
- [Ins13] Institut für Telematik, Karlsruher Institut für Technologie. OverSim - The Overlay Simulation Framework, July 2013.
- [JM96] David B. Johnson and David A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, 1996.
- [LWW10] Che-Liang Liu, Chih-Yu Wang, and Hung-Yu Wei. Crosslayer Mobile Chord P2P protocol design for VANET. *Int. J. Ad Hoc Ubiquitous Comput.*, 6(3), August 2010.
- [MJ07] Qi Meng and Hong Ji. MA-Chord: A New Approach for Mobile Ad Hoc Network with DHT Based Unicast Scheme. In *Proc. WiCom*, Shanghai, China, 2007.
- [MJB05] Alberto Montresor, Márk Jelasity, and Özalp Babaoglu. Chord on Demand. In *Proc. of the International Conference on Peer-to-Peer Computing (P2P)*, Konstanz, Germany, August/September 2005.
- [MWSP08] Peter Merz, Steffen Wolf, Dennis Schwerdel, and Matthias Priebe. A Self-Organizing Super-Peer Overlay with a Chord Core for Desktop Grids. In *Proc. IWSOS*, Vienna, Austria, December 2008.
- [OMN13] OMNet++ Community. OMNet++, July 2013.

- [PB94] Charles E. Perkins and Pravin Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proc. SIGCOMM*, pages 234–244, London, UK, 1994.
- [PBRD03] C. Perkins, E. Belding-Royer, and S. Das. Ad Hoc On-Demand Distance Vector (AODV) Routing. RFC 3561, 2003.
- [PDH04] H. Pucha, S.M. Das, and Y.C. Hu. Ekta: An Efficient DHT Substrate for Distributed Applications in Mobile Ad Hoc Networks. In *Proc. WMCSA*, December 2004.
- [RGRK04] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling Churn in a DHT. In *USENIX Annual Technical Conf.*, 2004.
- [SGF02] Rüdiger Schollmeier, Ingo Gruber, and Michael Finkenzeller. Routing in Mobile Ad Hoc and Peer-to-Peer Networks. A Comparison. In *Int’l Workshop on Peer-to-Peer Computing. In Networking 2002*, pages 172–186, May 2002.
- [SMK⁺01] Ion Stoica, Robert Morris, David Karger, et al. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM*, 2001.
- [SPS13] Nenad Stojnic, Lukas Probst, and Heiko Schuldt. Compass optimized routing for efficient data access in mobile chord-based p2p systems. In *Proceedings of 14th IEEE International Conference on Mobile Data Management (MDM13)*, June 2013.
- [SS12] Nenad Stojnic and Heiko Schuldt. OSIRIS-SR: A Safety Ring for Self-Healing Distributed Composite Service Execution. In *Proc. SEAMS*, Zürich, Switzerland, June 2012.
- [Var01] András Varga. The omnet++ discrete event simulation system. *Proceedings of the European Simulation Multiconference (ESM’2001)*, June 2001.
- [WZS04] R. Winter, T. Zahn, and J. Schiller. Random Landmarking in Mobile, Topology-Aware Peer-To-Peer Networks. In *Proc. FTDCS*, Suzhou, China, May 2004.
- [ZS06] Thomas Zahn and Jochen Schiller. DHT-based Unicast for Mobile Ad Hoc Networks. In *Proc. PerCom 2006 Workshops*, Pisa, Italy, March 2006.