Non-Sticky Fingers: Policy-Driven Self-Optimization for DHTs

Matti Siekkinen[1,2] and Vera Goebel[2]

[1] Helsinki University of Technology, Dept. of Computer Science and Engineering, P.O.Box 5400, FI-02015 TKK, Finland matti.siekkinen@tkk.fi
[2] University of Oslo, Dept. of Informatics, Postbox 1080 Blindern, 0316 Oslo, Norway goebel@ifi.uio.no

Abstract. It is a common situation with distributed hash tables (DHT) that insertions and lookups frequently target only specific fractions of the entire value range. We present in this paper a self-optimization scheme for DHTs that optimizes the routing behavior in such situations. In our scheme, called Non-Sticky (NS) fingers, each node continuously measures the routing behavior and guides neighboring nodes to adjust their NS fingers (a subset of all the long distance links that the node establishes) accordingly in order to shortcut the most popular sections of routes. Our scheme enables self-optimization, which means that it adapts to the current system state and only operates when advantageous. It is also policy-driven, which means that the application can specify its policy on the tradeoff between performance and cost efficiency. We implemented the NS-fingers scheme for an existing order-preserving DHT and report the evaluation results. Our simulation results show that in a realistic application scenario, NS-fingers can halve the number of routing hops.

1 Introduction

Distributed hash tables (DHT) [1–3] provide efficient data exchange for completely distributed applications. These systems allow to lookup a node that stores a particular data value by specifying a key corresponding to that value.

In this paper, we present a generic self-optimization scheme for minimizing the length of routes in DHTs in cases where popularity of some value ranges in the key space is higher than others. The key observation is that in such a case, certain hops or sequences of hops (i.e. portions of entire route) become more utilized than others. Therefore, it makes sense to try to optimize these routes by making them pass through as few intermediate hops as possible. We call our scheme Non-Sticky fingers (NS-fingers) due to the way it functions: In DHTs, a node establishes a set of long distance links, a.k.a. fingers in Chord [1], in addition to "nearby" neighbors. The destination nodes of these fingers can be chosen in various ways. In Chord for instance, each node establishes fingers to nodes that are the powers of two distance in hops from the node (i.e. 2, 4, 8, etc. hops away from the node). These fingers are proven to enable efficient logarithmic routing performance. In NS-fingers, a subset of all fingers of a node are selected to be non-sticky. These fingers are continuously adjusted according to the estimation of demand for given links. In this way, we strive for shortening the most popular routes

which may be shared portions of routes from many different source nodes to many different destination nodes.

In specific cases, reducing the number of routing hops with shortcuts may actually increase the end-to-end delay of the path. This is because the overlay topology does not necessarily reflect the underlying network-layer topology or geographical distance. Thus, neighbor nodes in the DHT may be located far away from each other geographically or many hops away in the network-layer topology. Therefore, we also present a simple extension of our scheme to consider also the delay when making decisions to shortcut routes.

Many optimizations for DHT routing exist today. What makes our scheme stand out from the crowd is that it proposes neither to grow the size of routing tables nor to add extra pointers or hints to optimize routing to the entire key space. We also do not change the forwarding procedure. We simply propose to dynamically adjust the routing table entries based on current demand. We show in later sections that such a straightforward modification in the routing table maintenance can deliver an impressive improvement in routing efficiency. The cost of this improvement is some additional load in terms of messages sent and memory used (see discussions in Section 3.5). Our scheme is an add-on that can be applied in principle to any kind of DHT. This kind of optimization approach is valuable especially for resource constrained devices which might not be able to afford to scale up the DHT routing table sizes.

Where do such non-uniform distributions occur that call for this kind of an optimization scheme? We give two example application scenarios that use *order-preserving* DHTs. Such DHTs (e.g. [4,5]) use order-preserving hash functions, or perform no hashing at all like in [4], in order to process range queries efficiently. First, consider locality aware applications using virtual network coordinates (e.g. [6]). In such applications, nodes store their coordinates and lookup nodes in their vicinity, that is, nodes having coordinates close to its own. These lookups can be efficiently expressed using range queries. As a second example, consider peer-to-peer video streaming where information about the blocks of video that a node currently has are inserted by the node and, consequently, looked up by other nodes. When a particular node is playing the stream, it queries a specific range of blocks at a time corresponding to its playout buffer. Note that using a standard DHT would in both cases require performing separately lookups corresponding to all the values within the ranges.

In both examples, insertions and lookups can have a high level of locality, that is, a given node often inserts and looks up similar value ranges: The lookups performed by a node discovering other nodes in its vicinity based on their coordinates exhibit a high degree of locality. Furthermore, if there are clusters of nodes, the value ranges corresponding to the coordinates of those regions become frequently addressed. As for the P2P video streaming example, the range of blocks that a particular node is interested in at a specific time instance exhibits temporally a high level of locality. Similarly, a node stores only information about pieces that it has just downloaded following the progress of viewing the video stream. In addition, a given video stream may experience a flash crowd phenomenon meaning that at a particular time (usually in the beginning) the stream becomes very popular. In such a case, the value ranges of the blocks following the progress of viewing of the nodes forming that flash crowd are very frequently

addressed. DHTs without any optimizations treat ranges equally so that the expected number of hops to any range of values is the same. Our scheme would adapt in these examples to provide few-hop routes to the heavily used fraction of the range by increasing the expected number of hops to the other portions of the range that are little or not at all used.

Our scheme is *self-optimizing*: First of all, nodes continuously perform optimizations based on latest observed routing behavior in the system. In this way, the system adapts when the most popular range shifts, for instance. Second, the changing state of the system, e.g. the popular range size and the number of nodes, determines whether using the scheme is beneficial. NS-fingers can be made to adapt to this changing state. To enable this, we parameterize the scheme in order to provide "control knobs". By tuning these control knobs, its behavior can be adapted to the changing state of the system. For example, it can be turned off if the popularity distribution of the ranges is uniform.

The self-optimization is *policy-driven*: Tuning the control knobs allows to determine a tradeoff between how aggressively route optimizations are performed and how cost efficiently the scheme operates in terms of extra control messages routed. Thus, applications can specify their policy wrt. this tradeoff, as a result of which the scheme tunes its control knobs to comply with this policy, i.e. it *self-configures* its parameters based on estimations of the current state of the system and this policy.

The contributions of this paper are the following:

- We present a self-optimization scheme for routing in DHTs that is based on non-uniform distribution of popularity of value ranges. The self-optimization can be controlled through a set of parameters and, thus, allows customizing the routing behavior for a policy specified by a given application.
- We present an extension of the scheme that takes into account the delay, in addition to routing demand, of links when making decisions on shortcutting paths.
- We implement the scheme for an existing order-preserving DHT and show through simulations that in a realistic application scenario, NS-fingers can deliver up to 50% reduction in the average number of routing hops.

2 Related Work

Existing related work commonly focuses either on a growing the size of the actively managed routing table for increased performance or on relying on some additional lightweight, potentially obsolete information for routing. Considering proposals in the first category, Beehive [7] relies on Zipf-like query distributions and uses proactive replication to tradeoff resource consumption for improved lookup performance. Replication can be also problematic if data is volatile, i.e. has a relatively short lifetime. The work in [8] proposes to maintain complete routing tables using an aggressive hierarchical update protocol. Kelips [9] also achieves better lookup performance through increased routing table sizes and update traffic. EpiChord [10] maintains reactively a large routing state and copes with the possibly outdated routing state by using parallel lookups. As for the second category, "ShortCuts" approach [11] uses soft-state hints in local and global levels to improve the routing. Mercury incorporates simple route caching which can be used as a complement to our scheme. NS-fingers differs from

these approaches in that we consider constant-size routing tables so that the applications using the DHT can set this size in order to control the resource requirements. Then, these allocated entries are dynamically adjusted according to the current lookup and insertion patterns. Furthermore, we provide the applications the control knobs to set the tradeoff between the aggressiveness and resource consumption of the optimization scheme itself.

"Interest-based shortcuts" presented in [12] is also based on creating shortcuts in P2P system based on interests. However, the scheme is targeted for Gnutella, an unstructured P2P system. Caching routes from source to destination as shortcuts in a DHT would be similar to the way their system works in Gnutella. Our approach is to continuously adjust routing table entries in order to shortcut one popular hop at a time.

In [5], the authors present a way to construct efficient routing tables in an order-preserving DHT that relies on estimates of hop counts between nodes. The scheme works also for skewed distributions of data values. The difference to our scheme is that we continuously adjust the routing based on recently measured behavior.

Yet another kind of approach is SkipNet which controls data placement by organizing data items according to their string names, which enables it to guarantee routing locality. SkipNet is useful when the data items stored by a node have certain persistent locality, e.g. content internal to an organization. However, this is not always the case. For example, while network coordinates exhibit temporarily locality (see examples in Section 1), nodes can move to another location in the coordinate system in which case the locality no longer holds. Also in the video streaming example, the locality of the data and lookups and the ranges that are most frequently addressed change continuously. Our scheme is able to optimize the routes even if the locality of the data and lookups change over time because the optimization is performed based on observed routing behavior at run time.

3 Non-Sticky Fingers

3.1 Overview

Our optimization scheme optimizes routing behavior when the popularity of the attribute value range is skewed. As we discussed in the introduction, such a case can occur when there is locality in the data or queries or when many nodes insert values that fall into the same ranges or query the same ranges resulting in a kind of flash crowd phenomenon.

We consider ring structured order-preserving DHTs where each node establishes links to predecessors and successors and, in addition, a number of long distance links, a.k.a. fingers in the literature. While we focus on a ring structured DHT in this paper, the scheme can be applied to other geometries as well (cf. Section 3.5). These fingers enable efficient routing within the overlay: simply passing data items and queries to successor or predecessor nodes would result in overall very inefficient routing (O(N)). With long distance links, it is possible to achieve logarithmic routing performance. The set of fingers can be static throughout the operation of a given node or can be periodically rebuilt. Nodes also maintain a set of reverse neighbors which are the nodes having a long distance link to the node.

We propose to have a set of dynamic long distance links, i.e. Non-Sticky fingers. The idea is that a node establishes k fingers out of which it chooses a set of l that will be Non-Sticky (NS). These NS fingers are adjusted continuously according to the most popular routes while the remaining $k-l$ fingers are static in the sense that they are only rebuilt periodically. The rationale is that when there is great demand for a particular multi-hop route within the ring, we make shortcuts to that route step by step in such a way that eventually the items are routed with as low number of hops as possible (with only one hop if possible) from the starting point to the end point of this popular route. The simplest case is when we shortcut a route so that data items are routed directly from the source node to the destination node. In such a case, using route caching at the source is a sufficient solution. However, note that a highly utilized route can also be an intermediate segment of many different routes from source to destination nodes. In such a case, caching a route directly from source to destination is not an optimal solution. We compare the performance of our scheme to route caching in Section 4.

Our scheme establishes the $k-l$ "normal" long distance links following the harmonic probability distribution function ($p_n(x) = 1/(x \ln n)$, when $x \in [1/n, 1]$) similarly to Symphony [13], which guarantees expected path length of $\mathrm{O}(\frac{1}{\mathrm{k-l}}\log^2\mathrm{n})$ hops in a n node network according to the Small World phenomenon [14]. This method gives us the flexibility to choose the number of NS-fingers to establish while still guaranteeing a certain level of routing performance in case of completely random data and query distributions. In this way, we sacrifice some of the routing performance bounds to little used ranges in order to optimize the routing towards the most popular ranges.

3.2 Adjusting Non-Sticky Fingers

The NS-fingers are adjusted according to the estimated service demand of links. Each node measures the service demands of each of its long distance and successor links to other nodes. Service demand $S_i(d)$ of the link of node i to destination range d is defined as the number of data items or queries forwarded per time unit using that link. Each node also keeps track of $S_i^j(d)$ which is the portion of service demand generated by reverse neighbor j i.e. a node that has a successor or long distance link to node i. Those data items and queries that originate from the node itself are excluded from the computation of the service demands ($i \neq j$). Hence, the service demands reflect the amount of services that the node provides as an intermediate hop for other nodes. Obviously, $\sum_j S_i^j(d) = S_i(d)$.

Each node checks periodically the service demands of its links and chooses the link to range d_{max} which is the link with the highest demand. The node then sends adjustment requests to those reverse neighbors that contributed to the demand, i.e. node i sends requests to nodes $j|S_i^j(d_{max}) > 0$. This request contains the ID of the node that is responsible for the target range d_{max} and the service demand $S_i^j(d_{max})$. Figure 1 illustrates the scheme in a very simple scenario. In the figure, node 7 chooses the link to node 8 as the heaviest link because the computed service demand is 8 compared to 6 of the link to node 1. Note that the traffic originating from node 7 towards node 1 does not count. Node 7 then sends adjustment requests accordingly to nodes 5 and 3.

Upon receiving an adjustment request, a node stores it. Each node then adjusts its NS-fingers periodically: The node iterates through the received adjustment requests

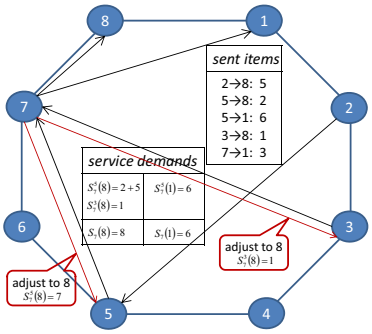

Fig. 1. Computing service demands.

from largest service demand to the smallest. At each iteration, it compares the service demand included in the request to the smallest service demands of its currently established NS-fingers (i.e. the NS-finger with lightest load). If the demand in the request is larger, the node adjusts the NS-finger to the target range specified in the request. The iteration continues until each of the nodes NS-fingers has been adjusted or no more adjustments are necessary (i.e. the service demand of the current NS-fingers exceeds the demands in the remaining requests). It can happen that a node begins to adjust its NS-fingers right after it has checked its service demands and consequently reset the demands after sending necessary adjustment requests. In this case, the service demands of the NS-fingers are lower (close to zero) than what the true demand actually is. To avoid unnecessary adjustments in such cases, the node stores also the service demands of the NS-fingers from the previous period and uses those in addition to the current ones when making the choice whether to adjust a particular finger.

3.3 Delay-aware extension

As we pointed out in the introduction, in certain cases reducing the number of routing hops does not reduce the total end-to-end delay for the path. We present a simple extension to our scheme that takes the delay into account when making adjustments.

In the delay-aware extension of the scheme, nodes measure the delay to the nodes in their routing table while performing peer management (e.g. send keep-alive messages). Consider again Figure 1. Node 7 routinely measures the delay to nodes 8 and 1. It would then include the measured delay in the adjustment requests sent to nodes 5 and 3. When, for example, node 5 subsequently is about to adjust one of its fingers to point to node 8, it will first measure the delay to node 8 and then make the adjustment if the measured delay is less than the delay from node 7 to 8 (in the adjustment request) plus the delay to node 7 (measured during peer management). In this way, each adjustment is guaranteed to shorten the total delay in the forwarding path. The price to pay is the extra delay measurement to the new long distance neighbor candidate.

The scheme can be further extended to prioritize the shortcutting of longer delay links, as follows. In addition to adding the delay measurement to each corresponding

adjustment request, all service demands of links are multiplied with the corresponding measured delay. Then this product, weighted service demand, is used in determining the "heaviest" link. In this way, when the "heaviest" link is chosen for shortcutting the delay is taken into account in addition to the amount of traffic passing through. The chosen link represents the one with largest estimated gain in total routing delay.

3.4 Parameters

Our scheme has three parameters: the two intervals that specify how often adjustment requests are sent (I_r) and how often the NS-fingers are adjusted (I_a), and the fraction of NS-fingers ($f = \frac{l}{k}$). The question is then how to choose these values.

In [15], we present detailed analysis of the impact of these parameter values through measurements from simulations. We first studied the tradeoff between fast route optimizations and cost efficiency of the scheme in terms of overhead messages routed. Then, we studied how a particular system setup (e.g. distribution of inserted data and number of nodes) affects the behavior of the system with a particular parameter configuration. Finally, we discuss how this information could allow us to design the scheme to be policy-driven in such a way that it self-configures its parameters depending on application specified policy (faster convergence vs. cost efficiency). Due to space constraints, we only present our main findings (details are presented in [15]).

The experiments led us to conclude that the impact of the interval parameters I_r and I_a to the speed of convergence seems to be similar regardless of the other parameter values. The main observations are that 1) the values of the two interval parameters should be set to similar values and 2) the smaller the values, the faster the convergence. The results agree with common sense: a large adjustment interval intuitively slows down the convergence while a small adjustment interval does not help if requests are sent rarely, due to large request interval, because nodes do not know towards which range to adjust. The simulations also showed that many parameters affect the cost optimal configuration of the scheme, which suggests that analytically determining I_a and I_r to achieve cost optimal routing behavior for any given setup is a complex problem. Addressing this challenge is currently left for future work. Nevertheless, the results gave us a good idea on the trend of the behavior with different interval lengths which we report in [15]. It is intuitive that the scheme reduces the overall number of routing hops only when the popular range is small enough compared to the number of NS-fingers established by each node. Thus, we compared the converged mean routing hops when $f = 1$ (all fingers are non-sticky) to the average number of routing hops when $f = 0$ (i.e. no NS-fingers) with different values of R. We observed that, regardless of the absolute number of fingers, when R is roughly larger than 0.6 the scheme no longer provides benefits. Of course, in order to check whether this result can really be generalized, we would need an analytical model of the system which we do not have at the moment.

3.5 Discussions

We briefly discuss in this section the extra cost imposed by the scheme, impact of churn to the scheme, applicability of the scheme to different kinds of DHTs, and the impact to load balancing.

The price to pay for the optimization using the NS-fingers scheme is increased memory demands and traffic. Each node needs to keep track of the service demands. Given that each node maintains k long distance links and a few successor links, the number of reverse neighbors for a node is also around k. This means that maintaining the service demands requires in the worst case storing and updating kxk of state information, k being typically $log_2 n$. The extra traffic introduced includes the adjustment requests and adjustments themselves, i.e. neighbor requests sent to the new target of adjusted finger. Since adjustment requests are sent periodically, they can be piggybacked into heartbeat messages in which case they come almost for free.

Churn is a common issue with DHTs. The impact of churn is similar to a DHT equipped with NS-fingers than to one without. Thus, the strategies described in [16] can be used. In fact, NS-fingers can help choose suitable timeout values (one of the important performance factors under churn according to [16]) through the delay-aware extension.

We focus in this paper on ring structured DHTs. However, the concept of NS-fingers is generic and the scheme can be applied to other geometries as well. For example, in Pastry [2], which is a kind of hybrid combining ring and tree geometries, NS-fingers could be used to shortcut routes on specific rows of the routing table (i.e. level of the tree) that shares the same ID prefix. Similarly, in CAN [3], which relies on a hypercube geometry, NS-fingers can be used to adjust the routing tables with the help of neighboring nodes considering the routing demand in order to maximize the number of "corrected" bits on each hop.

The scheme has also an impact on the load balancing of the system. Indeed, the routing load is implicitly driven via the adjustments towards the nodes that are responsible for a popular range. The advantage is that there are fewer nodes that become loaded but the drawback is that this load can be higher. To deal with this load imbalance, we can leverage existing explicit load balancing mechanisms such as the one of Mercury (see [4] for details).

3.6 Implementation

We implemented our self-optimization scheme on top of Mercury. We made this choice because the separation of concerns via the concept of attribute hubs allows applying our scheme flexibly to the desired hubs and customizing it to each hub separately if necessary. In addition, we can leverage some of Mercury's mechanisms for the self-configuration of the scheme (see [15] for details). Routing of items is similar in Mercury to Chord except that, thanks to the order preservation, range queries can be expressed and routed as one lookup instead of making multiple point queries.

The Mercury program code is open source. The code can be compiled to run on a simple discrete event-based simulator. This simulator does not model any queuing delays or packet losses, which enables the simulation of thousands of nodes. Such a

simulation environment is sufficient for us to evaluate and analyze the behavior of our optimization scheme and we used it for all the results presented in Sections 3.4 and 4. Note that with this simulator we cannot simulate the delay-aware extension presented in Section 3.3. However, since the extension is an optimization of the scheme and the basic mechanism does not change with it, this evaluation approach is still valid.

4 Evaluation

In this section, we evaluate the performance of NS-fingers in different situations. Our goal is to understand the level of performance improvement the scheme can deliver in different scenarios. Unless explicitly mentioned, the simulations in this section were run with $n = 5000$ and $r = n$. In reality, it is commonly the case that some lookups and insertions also fall outside of the popular range. Thus, in all of the simulations performed for the results presented in this section, a given data item was inserted to the popular range (modelled with R) with 90% probability and, consequently, to the remaining non-popular range with 10% probability.

4.1 Stable Popular Range

We first evaluate the gain with NS-fingers scheme in terms of reduction in routing hops compared to the case without NS-fingers. In order to further put the numbers in perspective, we include the case of using simple route caching. Route caching being a complementary mechanism that can be used together with NS-fingers, we merely want to show that it alone does not provide the same benefits as our scheme.

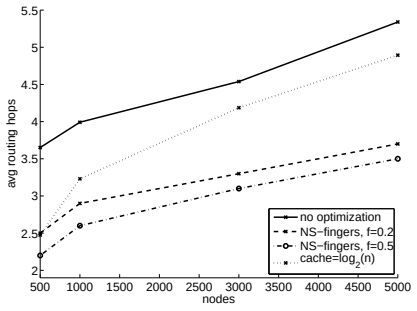

Fig. 2. Evolution of mean routing hops as a function of number of nodes.

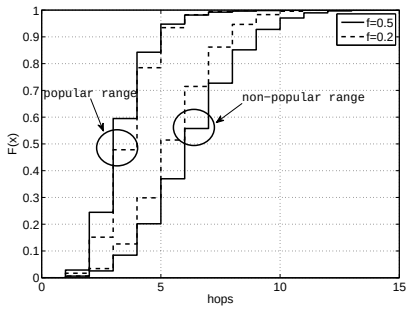

Fig. 3. CDF of number of hops to popular vs. non-popular ranges.

Figure 2 shows how the average number of routing hops evolves when the size of the network grows. For the NS-fingers cases, we computed the converged mean which represents a lower bound below which the average number of hops does not go even

if further adjustments are made (please, refer to [15] for a more detailed explanation and example) and for the other cases we computed the average over all routed items. In all cases $R = 0.05$ and the total number of fingers is $log_2 n$. The cache size was set to the same as the number of fingers, i.e. $log_2 n$. Note that each cache entry implies same extra maintenance cost as an additional normal finger would do. We observe that the additional cache delivers only a marginal improvement in the performance compared to NS-fingers. Furthermore, the figure shows that the number of hops do not scale similarly for the scenarios with route caching than with NS-fingers or no optimization. This observation suggests that the cache size should be increased more than logarithmically with the number of nodes in the system in order to have similarly scaling performance in terms of number of routing hops when the number of nodes in the system increases. Figure 3 plots a CDF of the number of hops separately for the insertions to the popular range and outside of the popular range. The figure illustrates the main tradeoff, i.e. how much the scheme penalizes the non-popular range. We can see that this tradeoff can be effectively controlled by adjusting the f parameter: with a smaller number of NS-fingers (smaller f) the difference in number of hops is smaller between the popular and non-popular ranges.

We also looked at what happens in a likely case where we have many smaller distinct popular ranges instead of just one bigger popular range. We simulated cases having from one to five popular ranges and the sum of the range sizes being the same in each case. We observed that the fewer partitions of the popular range there are, the better the scheme works. The difference in average number of hops between having one or five ranges is approximately one. There is intuitively an advantage of having a contiguous popular range because each of the NS-fingers direct traffic towards the popular range, and within that range, routing takes very few hops.

4.2 Unstable Popular Range

It is a likely scenario that the popular range is not stable but instead changes with time. Remember, for instance, the video streaming example from Section 1 where the popular range of video blocks shifts continuously following the progress of the stream. In the following, we investigate what is the routing behavior in such a case. We choose the parameters according to the example: We consider a two hours long movie stored into a 1.3GByte file which is divided into roughly 5000 pieces of 256KBytes each (like in BitTorrent). Each piece has a sequence number and all the sequence numbers together form the entire range. Nodes lookup peers having a copy of specific pieces using a range of sequence numbers as a key and insert sequence numbers of the pieces they have downloaded (note that the pieces themselves are not inserted into the DHT). Since we have 5000 pieces, each of the 5000 simulated nodes is responsible of one piece. We set the popular range so that it corresponds to 1 minute's worth of pieces, i.e. $R = \frac{1}{120} = 0.0083$. The rate at which the popular range shifts is equal to $\frac{\text{whole range}}{7200}$ per second. Finally, nodes request a range of 10 pieces at a time and make an insertion for each downloaded piece yielding a following total rate of routed items: $r = \frac{\text{insertions+lookups}}{\text{duration}} = \frac{5000n+500n}{7200} \approx 0.76n$ items/s where n is number of nodes.

Now consider the same scenario but with progressive download, i.e. nodes download the video file at full speed to a buffer while playing it locally. A similar situa-

tion would occur with large software updates which naturally lead to a flash crowd phenomenon. We assume a generous 15Mbit/s average download rate, which yields a download duration of 650s. We set the popular range to a window of 10s which gives us $R = \frac{10}{650} = 0.0154$, the rate at which the popular range shifts equal to $\frac{\text{whole range}}{650}$, and $r = \frac{5000n+500n}{650} \approx 8.46n$ items/s. We set $I_a = I_r = 500$ in both scenarios.

Table 1 compares the average number of hops and total cost in hops (lookups and insertions) resulting from the simulations of these two scenarios. For the cases where $f > 0$, we computed again the converged mean and included finger adjustment requests and adjustments to the total cost. Thus, the cost is the total number of hops routed by the DHT until each node has downloaded the entire movie. We see that for the streaming case, with $f = 0.5$ the average number of hops is reduced to half and the total cost is reduced to 59% compared to the case without NS-fingers ($f = 0$). With $f = 0.2$ the improvement is slightly smaller. When downloading at full steam, there improvement is almost similar for both values of f.

Table 1. Avg hop count/total cost for the video streaming example.

Scenario	$f = 0$	$f = 0.2$	$f = 0.5$
streaming	5.2/143M	2.9/92M	2.6/84M
progressive dl	5.3/146M	3.3/100M	2.8/89M

5 Conclusions and Future Work

In this paper, we presented NS-fingers, a self-optimization scheme for order-preserving DHTs, which performs route optimizations in the case of non-uniform popularity distribution of data or lookup value ranges. Our future work includes further studies regarding the relationship between the current system state and cost optimal parameter configuration in order to facilitate the configuration of the parameters of the scheme, esp. for specification of the self-configuration rules. In addition, we want to be able to express the expected number of routing hops with NS-fingers for a given configuration. We would also like to evaluate the delay-aware extension by, for example, deploying a set of nodes in PlanetLab. Our simulations revealed that the routing behavior somewhat oscillates when there are not enough NS-fingers per node to cover the whole popular region. We intend to investigate whether simple schemes such as using a weighted average of the service demands can alleviate this issue. Our evaluations focused on a specific order-preserving DHT, but the NS-fingers can be applied to any DHT. While it is really the application workload that in the end determines how big performance improvement our scheme can provide, it would still be interesting to try the scheme with another kind of DHT. We would also like to study whether in certain situations some particular nodes relying on the most popular paths experience overload, and if so, how to prevent it from happening.

Acknowledgments

The authors would like to thank Thomas Plagemann, Sasu Tarkoma, and Ovidiu Drugan for their helpful comments. This work has been funded by the Autonomic Network Architecture (ANA) project No. FP6-IST-27489 of the EU 6th Framework Programme, Situated and Autonomic Communications (SAC).

References

1. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of SIGCOMM '01. (2001) 149–160
2. Rowstron, A.I.T., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of Middleware '01. (2001) 329–350
3. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Schenker, S.: A scalable content-addressable network. In: Proceedings of SIGCOMM '01. (2001) 161–172
4. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: Proceedings of SIGCOMM '04. (2004) 353–366
5. Klemm, F., Girdzijauskas, S., Boudec, J.Y.L., Aberer, K.: On routing in distributed hash tables. In: P2P '07: Proceedings of the Seventh IEEE International Conference on Peer-to-Peer Computing, Washington, DC, USA, IEEE Computer Society (2007) 113–122
6. Dabek, F., Cox, R., Kaashoek, F., Morris, R.: Vivaldi: a decentralized network coordinate system. In: Proceedings of SIGCOMM '04, New York, NY, USA, ACM (2004) 15–26
7. Ramasubramanian, V., Sirer, E.G.: Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In: NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation. (2004) 8–8
8. Gupta, A., Liskov, B., Rodrigues, R.: One hop lookups for peer-to-peer overlays. In: Ninth Workshop on Hot Topics in Operating Systems (HotOS-IX), Lihue, Hawaii (2003) 7–12
9. Gupta, I., Birman, K., Linga, P., Demers, A., van Renesse, R.: Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03). (2003)
10. Leong, B., Liskov, B., Demaine, E.: EpiChord: parallelizing the chord lookup algorithm with reactive routing state management. Proceedings of ICON 2004 **1** (2004) 270–276 vol.1
11. Tati, K., Voelker, G.M.: ShortCuts: Using Soft State to Improve DHT Routing. In Chi, C.H., van Steen, M., Wills, C.E., eds.: WCW. Volume 3293 of Lecture Notes in Computer Science., Springer (2004) 44–62
12. Sripanidkulchai, K., Maggs, B., Zhang, H.: Efficient content location using interest-based locality in peer-to-peer systems. Proceedings of INFOCOM 2003 **3** (2003) 2166–2176 vol.3
13. Manku, G., Bawa, M., Raghavan, P.: Symphony: Distributed hashing in a small world. In: Proceedings of the USITS'03. (2003)
14. Kleinberg, J.: The small-world phenomenon: an algorithm perspective. In: STOC '00: Proceedings of the 32nd annual ACM symposium on Theory of computing, New York, NY, USA, ACM (2000) 163–170
15. Siekkinen, M., Goebel, V.: Non-sticky fingers: Policy-driven self-optimization for order-preserving dhts. Technical report, University of Oslo / Helsinki University of Technology (2009) http://www.tkk.fi/~siekkine/pub/siekkinen09nsfingers.pdf.
16. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2004) 10–10