Distributed Data Structures Improvement for Collective Retrieval Time

Read Al-Aaridhi, Ahmet Yüksektepe, Tobias Amft and Kalman Graffi
Department of Computer Science, University of Düsseldorf, Germany
Email: alaaridhi@hhu.de, ahmet.yueksektepe@hhu.de, tobias.amft@hhu.de, graffi@hhu.de

***Abstract*—In peer-to-peer overlay networks, distributed hash tables have been characterized as self-organizing, scalable, and robust. Up to now, distributed hash tables allow to store, search and retrieve only single data elements, which make the creation of more complex applications, such as distributed online social networks, very cumbersome. Sophisticated applications require the support for various and advanced data structures such as sets, lists and trees. Ideally, the DHT should support these advanced Distributed Data Structures. In previous work, we presented a new approach for Distributed Data Structure on distributed hash tables, which comes with low overhead and delay. In this paper, we address the issue of the time-consuming retrieval of deep data structures and present therefore a Cache Structures, which helps to accelerate the retrieval process by magnitudes. Simulations show that the Cache Structure gives us new functionality and an alternative way to retrieve the complete Distributed Data Structure much faster.**

I. Introduction

Peer-to-peer (P2P) networks propose an approach to run platforms for various applications through a fully distributed approach. They have been used to not only in the form of file sharing applications but also in further application domains, such as for streaming of for P2P-based online social networks such as in LibreSocial [1], [2], [3] and Safebook [4] or for optimization procedures [5], [6]. Currently, P2P networks only offer a store, search and retrieve operations based on single data items. Sophisticated applications such as for social networking, secure communication or collaborative work, on the contrary, would benefit from more advanced distributed data structures such as lists, sets or trees. In order to create for example a friend list in P2P-based online social networks, a data structure is needed to manage the set of friends, i.e. to add new entries or to find or delete existing entries in the set. Also lists, i.e. ordered sets, would be needed to implement a mailbox or news feed, supporting to append new entries and to retrieve old entries up to a specific number of entries back. A tree data structure would be beneficial to have, to implement a photo directory with albums and a comment section.

We previously proposed a Distributed Data Structure (DDS) scheme on Distributed Hash Tables (DHT), which consists of linked data items which contain the following:

- a mandatory object ID under which this item is stored.
- optional pointers to further data items in the DDS graph
- and an optional payload, such as an image or comment.

Thus, the DDS consists of pointer graphs in the DHT, in which the linked data objects contain ID information, pointers and payload. With this structure, we are able to create various data structures, such as (unordered) sets, (ordered) lists, (unordered or sorted) trees or even flexible distributed data graphs. A DDS supports many functions, which we describe next.

Store DDS: By calling the DDS layer, the user can create a completely new DDS, for example for an album of images. The DDS layer computes the corresponding data structure and each individual data item in it. Each data such as image is put in a single data item, which has individual object IDs for the DHT. The object IDs referring to these image data items are then added to a new (index) data item, which mainly contains only these IDs as pointers and the ID information of the new object. The index data item as well as each data item containing an image is stored in the DHT and replicated there.

Retrieve DDS: In the retrieving operations, the P2P application layer calls the DDS layer requesting to retrieve the DDS with a given object ID. The DDS layer now iteratively retrieves the DDS entry by entry. With each entry of this DDS retrieved, potentially new pointers are obtained listing object IDs which also belong to this DDS. The strengths of the DDS concept enfolds by retrieving parts of the DDS in parallel and thus being faster in the retrieval compared to the retrieval from one single node. Once all pointers are resolved and all data items corresponding to the requested DDS are retrieved, the DDS is handed over to the P2P application layer.

Modify DDS - Modify Values / Add Entry: When the P2P application layer decides to modify the DDS, either by adding a new entry, e.g. at the end of the list, in a set or as a leaf in the tree, it computes the corresponding data item that needs to be changed. For this single data item, it asks the DDS layer to replace it with a new data item with a modified version. The DDS layer performs a storage operation with the new data entry for the corresponding object ID of the data entry.

Modify DDS - Delete Entry: In case a single data item is requested to be deleted, the DDS layer locates it in the DDS and request it to be overwritten by a special data item marking the deletion of this item.

Initial evaluations show that the DDS scheme comes with low overhead and delay. Through the iterative retrieval of the entries in parallel, typically multiple nodes provide their single DDS items to the retrieving node. In a specific experiment, however, considering a deep tree structure as shown in Figure 1, we observe that the collective retrieval duration takes quite long as there are many pointers to resolve iteratively.

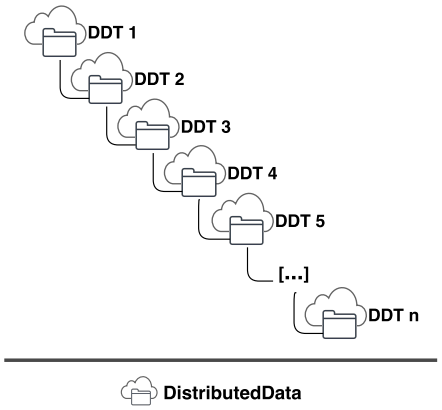


Fig. 1. Deep Distributed Data Tree Structure with n Nodes in total

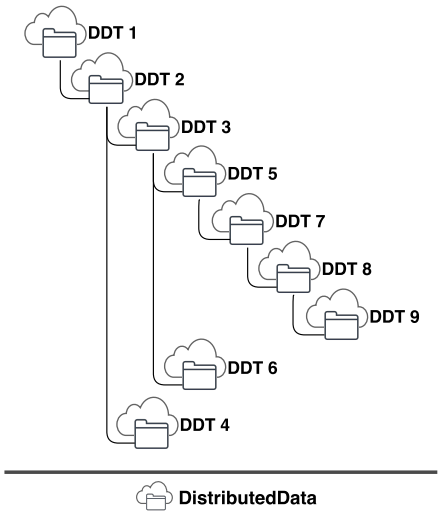


Fig. 2. Sample Distributed Data Tree Data Structure

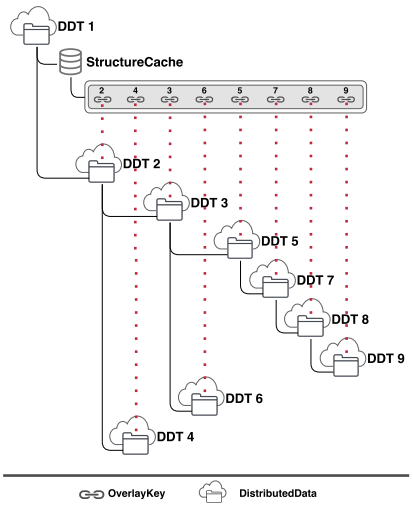


Fig. 3. Figure 2 with Structure Cache Construction

Each DDS item only resolves the pointer to the next element, leading to a chain of individual item retrievals without any benefits through parallelization.

With its deep reference chain, for each retrieved DDS object, a further DHT lookup is instructed. The resulting average duration times of collective-retrieves demonstrate a performance problem which occurs with greater depth in DDS-Structures due to the nature of DHTs. To overcome this issue, in this paper we create a Structure Cache for the data structure and save it as atomic data along the root object of the data structure. This Structure Cache contains the overlay keys of the deeper DDS objects under the Cache.

The contributions of this paper are threefold:

- We introduce the concept and benefits of using the Structure Cache in distributed data structures in detail
- We show through simulation the performance characteristics within worst case such as 100-entry deep trees
- We evaluate the distributed data structure with and without Structure Cache.

The remainder of the paper is organized as follows. In Section II, we present the concept of using Structure Cache for the Distributed Data Tree (DDT) and highlight its strengths. To evaluate the quality of this Structure Cache we present in Section III the evaluation setup and in Section IV the evaluation results. Section V concludes the paper.

II. Distributed Data Structure with Structure Cache

DHTs support simple put/get operations, sophisticated applications such as distributed online social networks require advanced DDS such as lists, sets and trees. In previous work, we presented DDS which distribute the load by storing large data structures on various nodes. A DDS consists thereby of single data items, which carry a payload, and/or further pointers to other data items, as well as identification information on the relation of this single data item to the whole DDS it is belonging to. In DDS the ID can be created deterministically, e.g. as a hash of a clear identifier which is known to the retrieving nodes and thus can act as an entry point to the DDS, or it can be completely random and only used for DDS internal purposes. Experiments with DDS have indicated that the collective retrieval duration highly benefits from the option of parallel retrieval. In flat DDS, i.e. with data items that contain many pointers, the objects to these pointers can be retrieved quickly in parallel. Whereas in data structures which have a rather deep structure, i.e. only very few pointers per data item, the parallel retrieval is rarely used and thus we observe the retrieval of a data item chain. The sample data structure in Figure 1 illustrates this problem.

We concentrate at this point on Distributed Data Trees (DDTs) the representative for DDS, as they can model trees, sets (flat tree) and lists (deep tree). In use cases like in this example, we have many objects to resolve iteratively, which are pointing to further objects. The single DDT nodes in this example have only the references to the underlying child DDT nodes. In case a Client wants to retrieve the complete recursive structure of a DDT, as in this example DDT 1 as the root node, the Client needs to resolve all DDT nodes iteratively. To be able to fetch all child objects of a DDS, the Client node needs to know first the overlay keys of the child objects. This requirement impacts recursively the whole data structure, until the deepest object in the DDS. The complete retrieval of a DDS is the resolving of all linked objects of an already retrieved object, at once. Yet, at each depth-layer of the DDS, the retrieval-time adds up, leading to high retrieval times for deep trees.

To deal with this performance issue, we developed a caching mechanism. Having the overlay keys of all deeper DDS objects in advance would enable a Client node to finish the complete retrieval of the DDS significantly faster. Instead of waiting after each depth-layer, the Client node can retrieve all Distributed Data objects at once from the DHT. This way we take advantage of the distributed character of the DDS and parallelize our retrieval processes. At this point, we define an

TABLE I
ACTION PROTOCOL FOR DISTRIBUTED DATA STRUCTURE WITHOUT STRUCTURE CACHE

Time	Action
$t = 0s$	Start retrieving DDT-1 (Initially only key of DDT-1 given)
$t = 1s$	Start retrieving DDT-2
$t = 2s$	Start retrieving DDT-3 and start retrieving DDT-4
$t = 3s$	Start retrieving DDT-5 and start retrieving DDT-6
$t = 4s$	Start retrieving DDT-7
$t = 5s$	Start retrieving DDT-8
$t = 6s$	Start retrieving DDT-9
$t = 7s$	Start retrieving DDT-1 completely, finished.

TABLE II
ACTION PROTOCOL FOR DISTRIBUTED DATA STRUCTURE WITH STRUCTURE CACHE

Time	Action
$t = 0s$	Start retrieving DDT-1 (Initially only key of DDT-1 given)
$t = 1s$	We have now all other overlay keys:Start retrieving DDT-2,DDT-3,...,DDT-9
$t = 2s$	Retrieving DDT-1 completely, finished.

optional Cache for DDSs and name this Cache entity Structure Cache. The Structure Cache feature is an extension of the capabilities of DDSs, which can, depending on use case, give a deciding performance boost.

We further analyze the complete DDS retrieval with and without Structure Cache to see how both options deal with the complete retrieval of the +sample DDT which is depicted in Figure 2. Without Structure Cache, the retrieving of deeper DDT objects in the data structure of a DDT requires to retrieve objects containing their overlay keys from higher position in the tree. Without having their overlay keys, the protocol is not able to retrieve deeper DDT objects and needs to wait for the complete retrieval of parent DDT objects. This way the collective retrieval time increased linearly with the depth of the data structure. In the action protocol depicted in Table I, it is shown how the recursive retrieval functionality deals with the data structure in Figure 2, assuming $1s_a = 1$ being the average single retrieve time.

We create a Structure Cache for the data structure and save it along the root DDT object of the data structure. As illustrated in Figure 3, this Structure Cache contains the overlay keys of all deeper Distributed Data objects under the cached root object. Upon retrieving the root DDT object, we gain the corresponding Structure Cache and all deeper DDT objects can be started to be retrieved simultaneously in parallel. As soon as the last single retrieve operation is finished, the retrieved overlay key and DDT pairs are used to reconstruct the structure of the complete DDT, and the complete retrieval is finished as shown in Table II.

For implementing the Structure Cache idea, an interface named IStructureCache has been defined, which presumes following essential functions to be implemented:

- CacheStructure (optional int depth) generates the Structure Cache at first hand from the locally available complete data structure. If a depth parameter is specified, the structure of the root object is only cached up to that depth.
- BuildStructureFromCache() recreates the final structure with the help of the retrieved single objects so far.

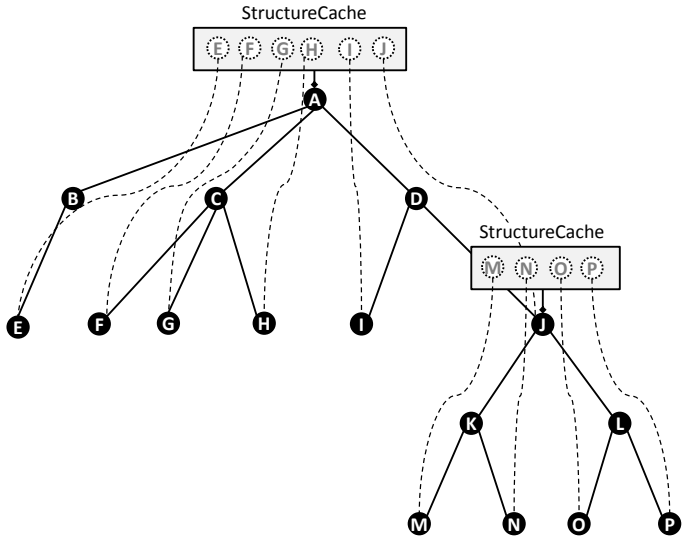


Fig. 4. Multiple Structure Cache's at different layers - Abstract Example

Furthermore, it is required that DDS implement the IStructureCache interface. For that purpose, the DDS use the Structure Cache class to save the Cache created during a CacheStructure() operation.

A. Analysis

We formulate the complete retrieval time for the recursive retrieval approach and the Structure Cache approach based on the following assumptions

- N: Depth of a Distributed Data Tree t
- $m(n)$: Set of nodes in the n'th layer, with $1 =< n =< N$
- C: Count of all nodes under t
- s_x: The single retrieve time of the x'th item in t, with $x \in \{1, \ldots, C\}$
- S_n: The worst single retrieval time at the n'th layer of t, with $S_n = \max(s_x) \mid x \in m(n), 1 \leq n \leq N$
- S: The worst single retrieval time, with $S = \max(s_x) \mid x \in \{1, \ldots, C\}$

Time costs: The complete retrieval time of the recursive retrieval approach equals to the sum of the worst single retrieve times S_n in each depth-layer n of each Distributed Data in the complete resolution of the DDS:

- Complete retrieval time: $\sum_{n=1}^{N} S_n = (S_1 + S_2 + \ldots + S_N)$
  The worst case upper limit for the collective retrieval time of t can be estimated in this case with: $\rightarrow N \cdot S$
  (if the depth of t equals to its count of nodes. i.e. $N = C$)
- Complete retrieval time: $s_1 + \max(s_2, s_3, \ldots, s_C)$
  The upper limit for the collective retrieval time with a Structure Cache can be estimated with: $\rightarrow 2 \cdot S$

Space costs: In addition, a Structure Cache introduces storage costs (and upon retrieval network costs) equaling to:
$\rightarrow (C - 1) \cdot SizeOf(OverlayKey)$
Considering the significant speed boost, the given network and storage costs are acceptable and stay scalable.

B. Discussion of the Strengths and Weaknesses

The main strength of Structure Cache that it gives us an alternative way which retrieves the complete DDT much faster. The Structure Cache is replicated with the DDS as it is bound to the root of the DDS. Figure 4 illustrates how the Structure Cache is set up in a simple example. In this example we have a DDT with the object *A* as the root and below *A* a deep structure with multiple layers. Object *A* has direct links to *B*, *C* and *D*, as these nodes are *A*'s direct children. Object *A* further has a Structure Cache limited to only the second layer in the structure. While *A* could indeed have a Structure Cache as deep as the complete data structure, the application working using this DDS only needs the first two layers at once. Furthermore, one of the second-layer children, object *J*, has further children. Object *J* also utilizes a Structure Cache for having directly the reference to its second-layer children. This use case could be an example for an application in the social networks where *A* equals to a Profile of a user while *J* is an article with nested comments. It makes sense to immediately load the whole profile, but only up to a certain degree. After logging into the social network service, the user may not necessarily need immediate access to all articles with their comments. So Structure Caches at different levels along the data structure balances the load caused by updating the Structure Cache and enables in general controlled, application dependent, usage of the caching mechanism. A Structure Cache with variable depth at any position opens limitless use case possibilities for cross referencing and performance improvement.

Maintenance Costs: The Structure Cache comes with the cost of storing additional data, multiple overlay keys are stored at the node holding a Structure Cache. Yet, the storage costs stay in negligible magnitudes in most cases as the Structure Cache is useful for persisting more static paths in the overall structure of a DDS and it is probably going to be more used for such use-cases.

Behavior Under Churn: The Structure Cache is bound to the DDS holding it, so when the DDS is replicated, also the Structure Cache gets replicated. Yet, as it is with replication, there is always the chance that data could get lost, and in this case, the Cache can be lost. That the Cache gets lost before being even replicated is not much of a problem, as the application logic is going to see the Structure Cache as a helper, as a performance booster, in the worst case the user can still retrieve the complete DDS through a complete recursive retrieval procedure.

Keeping a Structure Cache Up-To-Date: One limitation of the Structure Cache feature is the need to keep the Cache updated. Indeed one would get outdated results if the Cache is not updated quickly enough. Yet, this problem can be dealt for example by sending immediately an update command for the Structure Cache after the structure of a DDS has been changed. Another option would be to send the update commands in predefined time intervals. The depth and the up-to-dateness of the Structure Cache are very application dependent parameters. Therefore, the application layer using the DDS should decide over these parameters. The continuous update process would create more load on the DHT node responsible for the DDS holding a Structure Cache, yet, it anyways does not make sense to use the Structure Cache mechanism for DDS parts which change continuously and fast.

III. Evaluation Setup

In this part of paper, we present the evaluation of the quality and costs of the DDS with and without Cache. Here we focused on three main metrics.

- Storage and retrieval success: This metrics shows the success ratio for storage and retrieval operations for DDS with and without Cache by comparing the number of storage/retrieval requests for DDSs with the number of successful storage and retrieval operations.
- Retrieval delay: The storage and retrieval of lists, sets and trees are reflected at corresponding operations on a set of linked data items. We evaluate the time needed to deploy and retrieve DDSs in relation to their size.
- Traffic overhead / number of contacted nodes: DDS are stored distributively and this metric shows the overhead in terms of contacted nodes, induces lookups and traffic with respect to individual storage or retrieval calls.

In order to obtain these metrics and to evaluate our approach, we use an event-based simulator called Peerfact-Sim.KOM [7], [8], [9]. This simulator offers to simulate multi-layered P2P systems such as an application layer, service layer, data management layer, overlay layer as well as several networking layers. We simulated the DDS with and without Structure Cache, and measured the retrievability as well as the retrieval delay and traffic overhead. We used Chord [10] as the main overlay. In our scenario, we used the global network positioning (GNP) [11] for the networking model. Also, we modeled churn based on measurements in KAD [12] for a realistic environment. We performed tests both with and without churn to measure the impact of churn on a network with 1000 nodes. In the begin of the simulations, from minute 0 to minute 40, a total of 1000 nodes joined the network. In simulations, these nodes remained active until the end of the simulation at minute 200.

In our scenario, as shown in Figure 1, we present a deep tree structure with 100 entries, in which every object except the leaf object has one single child object in the tree. The DDS has been tested with and without Structure Cache. The use case is an article with connected comments on the timeline of a user. Every new comment can be attached to the previous last comment creating an ordered list of comments under the article, it results in a DDT such as in Figure 2. In the simulations, at t=60 minutes a node creates and stores this DDS in the network. In the time from t=100 minutes to t=170 minutes this single DDS is retrieved by 250 nodes four times each, i.e. 1000 times, in the network. In this scenario, we are interested to know how the collective retrieval time is improved with using Structure Cache by comparing the metrics results with Cache and without Cache. We performed our experiences in order to display the fundamental behavior for Cache Structure with DDS in P2P networks.

TABLE III
OPERATION SUCCESS RATIO IN THE DDS WITH AND WITHOUT CACHE AND UNDER CHURN AND WITHOUT CHURN

Sc.	Stores		DDS Retrieves		Item Retrieves		Lookups	
	Init.	Ratio	Init.	Ratio	Init.	Ratio	Init.	Ratio
Cc	510	100%	1000	100%	102000	100%	102563	99%
Cn	510	100%	1000	100%	102000	100%	102043	100%
Nc	510	100%	1000	100%	102000	100%	102667	99%
Nn	510	100%	1000	100%	102000	100%	102000	100%

(Cc): With Cache, with churn. **(Cn):** With Cache, without churn. **(Nc):** Without Cache, with churn. **(Nn):** Without Cache, without churn.

IV. EVALUATION RESULTS

Next, we present the evaluation results. Table III presents the success ratio of the four main operations, namely Store, DDS Retrieve, Item Retrieve and Lookup within four setups. The results for this table shows that all operations for DDS Scheme are performed correctly with and without Structure Cache both under and without churn in larger setups with 1000 active nodes. Only seldom failing peers or timeouts lead to single operation misses, resulting in a success ratio of the lookup of 0.99% in minimum. Thus the functional requirements of resolving the DDS retrieval are satisfied fully.

In Table IV, we present the detailed analysis of the performance and costs in our DDS scenario. We present in Table IV the average values, standard deviation and the maximum of the metrics Hop Count, Store Duration, DDS and Item Retrieve Duration, as well as costs in terms of Forwarded Messages, Served Messages and Contacted Nodes with regard to our DDS scenario with Cache and without Cache. In Figure 5 the distribution of these DDS Retrieves and Item Retrieve metrics over the retrieving nodes is shown.

A. Performance and Costs of the DDS scheme

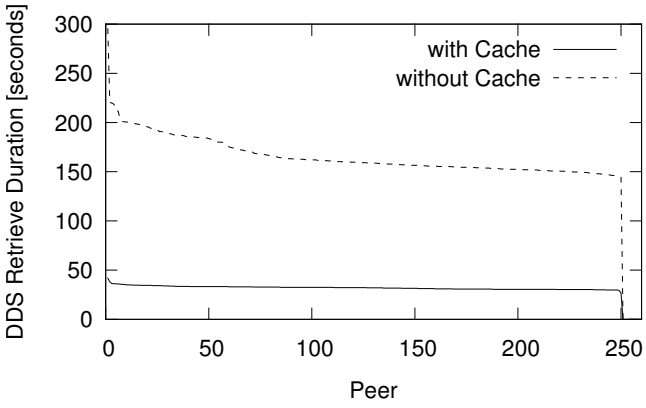
We present in Table IV a detailed analysis of the performance (Hop Count, Store Duration, DDS and Item Retrieve Duration) and costs (Forwarded Messages, Served Messages and Contacted Nodes), and present the distribution of the results over the peers for the 7 metrics. We present the average, standard deviation and maximum values of these 7 metrics. This overview is given for the same DDS scenario for four cases, namely using the DDS with Cache under churn, without Cache under churn, with Cache without churn and without Cache without churn. As it shows, the Hop Count is for DDS scenario with Cache and without Cache between 5 and 6. This value for the Hop Count is considered normal in an overlay with 1000 nodes. The second metric is the Store Duration which is defined through the Hop Count of 5-6 in combination with an average one hop delay of up to 100ms in the network model. The Store operation is performed once.

Next, we present the complete DDS and the single Item Retrieve times. A DDS is retrieved completely when all of its single items have been retrieved. In general, as shown in Table IV and in Figure 5(a) showing the DDS Retrieve times under churn and in Figure 5(c) showing the DDS Retrieve times without churn, the DDS Retrieve time is larger when no Cache is used and it is around 190 seconds under churn and around 165 seconds without churn. Using DDS with Cache under churn takes around 40 seconds and using DDS with Cache without churn the retrieval takes around 30 seconds. This is due to the Structure Cache which accelerates the DDS retrieval. In this metric, our main contribution is proved. The Item Retrieve times are larger when a Cache is used, as shown in Table IV and in Figure 5(b) and in Figure 5(d). This is because the DDS needs to create the Structure Cache once in the beginning and it is considered normal to take some time.

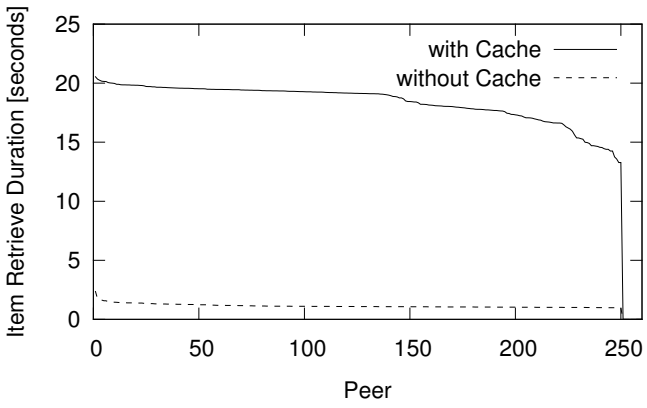
Next, in term of forwarded and served messages as well as the number of contacted nodes we evaluate the costs of the operations. The corresponding values for these metrics are also presented in Table IV. In our Scenario, the deep tree example of a DDS involves around 400 nodes to be contacted in order to store the DDS with 100 entries in a fully distributed fashion. Around 600 message are to be forwarded per node. This results in 1000 DDS retrieve operations, resulting in overall 102,000 Item Retrieve operations (see Table III). As each item retrieval requires a lookup of around 6 hops, so we have 612,000 messages to be forwarded by 1000 nodes, so around 600 messages per node in average. The performance shifts between the Scenario without Cache and with Cache in the case of churn. The deep DDS without Cache takes longer to be retrieved than the DDS with Cache. With its deep reference chain, for each retrieved DDS Item, a further DHT lookup is initiated. Through the aggregation of these timeouts and the recursive, non-parallel lookup behaviour, the performance of DDS is decreased in the Scenario without Cache. As shown in Table IV only 160 - 170 nodes are served, i.e. receive messages from store or item retrieve operations.

V. CONCLUSIONS

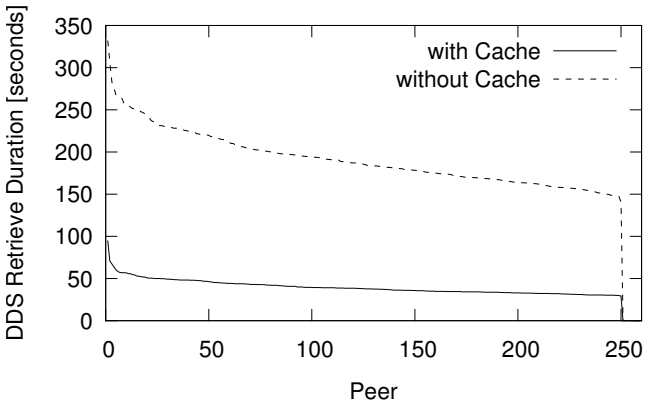
P2P networks offer features such as self-organizing, being scalable and robust. Distributed Hash Tables as the most common type of structured P2P networks only allow to store, search and retrieve single data elements. In previous work, we presented a Distributed Data Structure middleware on top of a Distributed Hash Table overlay that offers an object-oriented, distributed storage layer. Complex and advanced data structures are supported such as sets, lists and trees in this storage layer through linked distributed data objects. In this paper, we propose a Structure Cache attached to the Distributed Data Structure to improve the performance of the data retrieval by maintaining and index data structure at the root of the Distributed Data Structure. For the evaluation, we investigated the performance of the Structure Cache applied in the Distributed Data Structure under churn and without churn in a network with 1000 nodes. Evaluation results show that the DDS operations with Structure Cache are fully functional and resolve to nearly 100% of the retrieve operations as shown in Table III. The retrieval time of DDS with Cache Structure becomes much faster as shown in Table IV. We believe that our Cache Structure approach with variable depth at any position opens limitless use case possibilities for cross referencing and performance improvement.



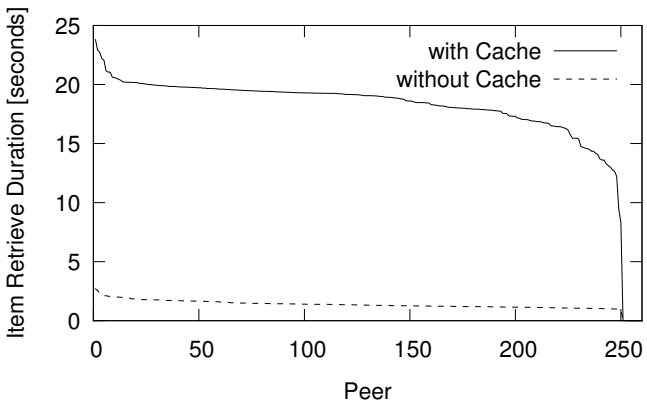
(a) DDS Retrieve Duration without Churn



(b) Single Data Item Retrieve Duration without Churn



(c) DDS Retrieve Duration with Churn



(d) Single Data Item Retrieve Duration with Churn

Fig. 5. Measurement Results

TABLE IV
PERFORMANCE AND COSTS OF THE DDS SCHEME WITH AND WITHOUT CACHE

Sc.	Hop Count			Store Duration			DDS Retrieve Duration			Item Retrieve Dur.			Forwarded Messages			Served Messages			Contacted Nodes		
	Avg.	σ	Max.	Avg.	σ	Max.	Avg.	σ	Max.	Avg.	σ	Max.	Avg.	σ	Max.	Avg.	σ	Max.	Avg.	σ	Max.
Cc	5.83	0.12	6.25	0.81	0.00	0.81	39.70	8.35	94.5	18.42	2.02	23.79	590.55	486.04	3424	162.01	152.46	1120	408.40	6.43	510
Cn	5.96	0.12	6.47	0.81	0.00	0.81	32.07	1.69	41.75	18.38	1.56	20.53	529.26	417.00	2249	163.53	131.55	961	408.40	6.43	510
Nc	5.65	0.09	6.17	0.81	0.00	0.81	191.58	31.75	331.5	1.39	0.29	2.70	574.05	465.18	2520	169.64	153.71	1050	408.40	6.43	510
Nn	5.84	0.07	6.05	0.81	0.00	0.81	165.68	18.49	295.25	1.13	0.15	2.35	521.00	408.38	2423	165.85	131.79	1020	408.40	6.43	510

REFERENCES

[1] K. Graffi, S. Podrajanski, P. Mukherjee, A. Kovacevic, and R. Steinmetz, "A Distributed Platform for Multimedia Communities," in *Proc. of IEEE Int. Symposium on Multimedia (ISM)*, 2008.

[2] K. Graffi, C. Gross, P. Mukherjee, A. Kovacevic, and R. Steinmetz, "LifeSocial.KOM: A P2P-Based Platform for Secure Online Social Networks," in *Proc. of IEEE Int. Conf. on Peer-to-Peer Computing (P2P)*, 2010.

[3] K. Graffi, C. Gross, D. Stingl, D. Hartung, A. Kovacevic, and R. Steinmetz, "LifeSocial.KOM: A Secure and P2P-based Solution for Online Social Networks," in *Proc. of IEEE Int. Consumer Communications and Networking Conf. (CCNC)*, 2011.

[4] L. A. Cutillo, R. Molva, and M. Önen, "Safebook: A Distributed Privacy Preserving Online Social Network," in *Proc. of Int. Symp. on World of Wireless, Mobile and Multimedia Networks (WOWMOM)*, 2011.

[5] K. Graffi, K. Pussep, S. Kaune, A. Kovacevic, N. Liebau, and R. Steinmetz, "Overlay Bandwidth Management: Scheduling and Active Queue Management of Overlay Flows," in *IEEE LCN '07: Proceedings of the International Conference on Local Computer Networks*, 2007.

[6] K. Graffi, "Monitoring and management of peer-to-peer systems," Ph.D. dissertation, 2010.

[7] M. Feldotto and K. Graffi, "Comparative Evaluation of Peer-to-Peer Systems Using PeerfactSim.KOM," in *IEEE HPCS'13: Proceedings of the International Conference on High Performance Computing and Simulation*, 2013.

[8] K. Graffi, "PeerfactSim.KOM: A P2P System Simulator - Experiences and Lessons Learned," in *IEEE P2P '11: Proceedings of the International Conference on Peer-to-Peer Computing*, 2011.

[9] A. Kovacevic, S. Kaune, H. Heckel, A. Mink, K. Graffi, O. Heckmann, and R. Steinmetz, "PeerfactSim.KOM - A Simulator for Large-Scale Peer-to-Peer Networks," TU Darmstadt, Tech. Rep. Tr-2006-06, 2006.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," in *SIGCOMM '01: Proc. of the International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2001.

[11] T. Ng, S. Eugene, and H. Zhang, "Global Network Positioning: A New Approach to Network Distance Prediction," *ACM SIGCOMM Computer Communications Review*, vol. 32, no. 1, 2002.

[12] M. Steiner, T. En-Najjary, and E. Biersack, "Long Term Study of Peer Behavior in the KAD DHT," *IEEE/ACM Transactions on Networking*, vol. 17, 2009.