A Decentralized Resilient Short-Term Cache for Messaging

Henner Heck, Olga Kieselmann, Nils Kopal, Arno Wacker

A Decentralized Resilient Short-term Cache for Messaging

Henner Heck, Olga Kieselmann, Nils Kopal, and Arno Wacker

Applied Information Security, University of Kassel, Germany,
firstname.lastname@uni-kassel.de,
WWW: https://ais.uni-kassel.de/

Abstract. Messaging applications are among the most popular internet applications and people use them worldwide on a daily basis. Their supporting infrastructure, though consisting of a multitude of servers, is typically under central control. This enables censorship and seamless user profiling. A fully decentralized infrastructure, with decentralized control and redundant data storage, can mitigate these dangers. In this paper we evaluate the basic ability of decentralized networks created by the network overlay and data storage protocol Kademlia to serve as a short-term data cache for messaging applications. Our results show, that reliable retrieval of up to 20 replicas is possible.

Keywords: Overlay Networks, Storage Resilience, Censorship Resilience

1 Introduction

Millions of people use messaging applications on a daily basis to exchange, e.g., text messages or pictures. Popular examples are WhatsApp, Snapchat, or Telegram. While technically a distributed system taking a multi-server or cloud approach, the infrastructure supporting these messaging applications is typically under central control. This implies several security and privacy issues, e.g., eavesdropping, data manipulation, profiling, or censorship. While some of these issues can be mitigated by end-to-end encryption and other measures, the danger of censorship is inherent to such a system. To prevent censorship, data storage and data transmission need to happen redundantly with multiple independent responsible parties. For this purpose we evaluate redundant 24 hour short term data storage and retrieval in a network organized by the distributed hash table and overlay network Kademlia. Our results show how resilient such a network is against censorship by evaluating how many data replicas need to be suppressed in order to prevent successful data retrieval.

The remainder of this paper is organized as follows: First, we discuss related work in Section 2 and briefly describe the Kademlia protocol in Section 3. We then present our assumptions in Section 4. In Section 5 we present our protocol modifications necessary for determining the storage resilience and introduce our evaluation terminology. Based on this, we present and discuss the results of our

storage resilience measurements in Section 6. We conclude our paper in Section 7 with a brief summary and provide an outlook on future research.

2 Related Work

Kademlia and overlay networks in general have been studied extensively in the scientific literature. A survey about research on robust peer-to-peer networks from 2006 [12] already lists several hundred references. Another survey from 2011 reaches close to a hundred references [16]. Despite the large amount of publications in general, the actual feasibility for redundant storage and retrieval in Kademlia has not been thoroughly evaluated. We limit our discussion of related work to existing distributed messaging systems and to literature with relevance for redundant storage and retrieval in Kademlia.

The fully distributed messenger Freenet [2] has the goal of providing censorship-resilient communication and publishing. At its core it uses a "small world" overlay network. Roos et al. [15] took measurements in a real world Freenet network. They found it suboptimal for routing and experienced long delays and low success rates for data retrieval. For Kademlia-type networks the number of nodes to contact for a successful data retrieval grows only logarithmically with the network size n [14]. For sufficiently random node identifiers Cai et al. [1] proved an upper bound of $O(c \cdot \log n)$, with c being a constant factor.

Ji-Yi et al. [8] describe a p2p cloud storage system named MingCloud. They experimentally evaluate a theoretical value called system availability, ranging from 0 to 1, in connection with full copy redundancy and erasure code. The authors describe MingCloud as based on the Kademlia Algorithm, but the focus of the paper is on a comparison the full copy approach with erasure codes. Beyond the system availability, no other properties are evaluated. Fedotova et al. [4] examine Kademlia for data storage and retrieval in enterprise networks. However, their focus is the implementation of different privileges for data access, not on evaluating redundant retrieval. Park et al. [11] propose a p2p based cloud storage system for redundant storage and retrieval. Their focus is on reducing the required data traffic as well as preserving data privacy. They compare an encoding scheme named Fountain code to other approaches like erasure coding. While their system is p2p based, it does not use the Kademlia protocol. The Bittorrent software Vuze [7] uses a modified version of Kademlia for data storage and retrieval. To handle possibly malicious nodes, Vuze requests a value from 20 nodes during a value lookup. While this modification of the Kademlia protocol is very similar to ours, there is no further data presented on the storage resilience resulting from this approach.

3 Kademlia

With the Kademlia overlay network, resources (nodes and values) are identified by a numerical *id* with the fixed bit-length b. Each node maintains a routing table containing identifiers of other nodes, its *contacts*. The routing table consists of

b so-called k-buckets, indexed from 0 to $b-1$. Each of the k-buckets can hold at most k contacts. The decision which contacts to store in a bucket depends on the node *id*, the bucket index, and the contact *id*. The distance between two identifiers is computed using the XOR metric, meaning that for two identifiers id_a and id_b the distance is $dist(id_a, id_b) = id_a \oplus id_b$, interpreted as an integer value. A bucket with index i is populated with those contacts id_i fulfilling the condition $2^i \leq dist(id, id_i) < 2^{i+1}$. The bucket with the highest index covers half of the id space, the next lower bucket a quarter of the id space, and so on. Another property of Kademlia is the request parallelism α, which is the number of contacts queried in parallel when a node tries to locate another node or a data object for a given *id*. The staleness limit s determines how many times in a row communication with a contact must fail, so that it is considered stale and removed from the routing table. The Kademlia authors set the default values $b = 160$, $k = 20$, $\alpha = 3$, and $s = 5$.

Furthermore, nodes can perform the following four remote procedure calls (RPCs): *Ping* probes a node to check whether it is online. *Store* instructs a node to store an *id*-value pair. *Find_node* looks up the k nodes closest to a given target *id*. A node selects the α contacts closest to the *id* from its routing table and sends a request to each of them. These nodes respond with their own list of closest contacts, which can then be used for further queries. This way, the requesting node iteratively gets closer to the target identifier. The RPC terminates when a number of k nodes have been successfully contacted, or no more progress can be made. *Find_value* retrieves the value for a given target *id*. It has almost the same behavior and termination conditions as *find_node*. The difference is, that nodes can answer by sending the requested value instead of a list of closest nodes. When this happens, *find_node* terminates immediately.

To publish an *id*-value pair a node first performs the *find_node* RPC to get a list of k successfully contacted nodes. It then sends a replica of the *id*-value pair to each of them via the *store* RPC.

Kademlia uses three different republishing mechanisms. Their purpose is to prevent *id*-value pairs from becoming unavailable and to store them at nodes with ids close to the id of a value. With the first republishing mechanism nodes periodically republish their stored *id*-value pairs every 60 minutes. This can lead to a significant amount of traffic. Also, whenever a node has only just republished an *id*-value pair, additional republications are unnecessary. Therefore, as an optimization, a node does not republish *id*-value pairs it was asked to store itself within the previous 60 minutes. The second mechanism is an opportunistic one. Whenever a node carries out the *Find_value* RPC successfully, it often has to contact several nodes before finding one that returns the value. After getting the value, the node performs a *Store* RPC. Out of the previously contacted nodes not returning the value, it sends the *id*-value pair to the one closest to the value id. The third mechanism is also opportunistic. Whenever a node learns of another node not present in its routing table, it might send *id*-value pairs from its own storage to that node. The decision which pairs to send is based on the

number of known nodes with ids placing them between the new node and an *id*-value pair.

4 System Model

Our system consists of a number of networked nodes connected by the Kademlia overlay network. Communication is message based and takes place directly between two nodes. The underlying network allows communication between any two nodes. Kademlia not only defines an overlay network structure, but also provides the functionality of a distributed hash table (DHT). Therefore, the nodes in our system can store data values at other nodes of the network (*put* operation) and retrieve values from them (*get* operation).

We distinguish between storing and non-storing nodes. We expect storing nodes to be servers provided by volunteers, a concept successfully realized with, e.g., the relay servers of the Tor network [3]. Non-storing nodes are PCs, tablets, mobile phones, or other end devices people use for messaging. A functional and well behaved storing node stores a received *id*-value replica unmodified in its local storage. Furthermore, when asked for a value contained in the local storage, the node will include an unmodified *id*-value replica in its answer.

A network of storing nodes is at the core of the messaging system. We expect these nodes have a session length, i.e. a continuous participation in the network, of several hours or more at a time. For the non-storing nodes the session length is less relevant, but we assume that they connect to the network at least once every 24 hours. Given that many devices have a continuous connection to the internet, this does not seen unreasonable.

We further assume that an attacker exists with the goal of censoring data and preventing its retrieval from the core network. The strength of the attacker is measured by the number of *id*-value replicas it can successfully suppress on average per retrieval request.

5 Redundant Storage and Retrieval

As described in Section 3, a publishing node initially stores replicas of an *id*-value pair at up to k disjunct nodes that were selected using *find_node*. This is the *put* operation. The number of replicas stored during a single put operation is the *replica storage count* (*RSC*).

The original *find_value* RPC of Kademlia terminates immediately when a node returns a replica of the requested value. However, for redundant retrieval, we need multiple replicas returned by disjunct nodes. To achieve this with Kademlia, we modified the responses and termination conditions for *find_value*. During the value lookup a node responds not with either a contact list or a value replica, but, whenever possible, with both of them. The lookup no longer terminates on receiving the first value replica, but collects as many replicas as possible, until up to k disjunct nodes have responded. These changes potentially give us up to k value replicas returned by k disjunct nodes.

In terms of censorship we must consider the worst case in which a replica is suppressed right at the initial *put* operation. Suppressing a replica at that point also suppress all its subsequent replicas created by Kademlia's republishing mechanisms. We, therefore, can only consider replicas that are unique with regard to the initially stored RSC replicas. Hence, two or more replicas derived from the same initial replica, are considered a single unique replica. We call the retrieval of replicas for an *id* the *get* operation. The number of retrieved unique replicas is the *unique replica return count* (RRC_{uniq}). Since the RRC_{uniq} is limited by the RSC and, therefore, by k, we introduce the term *unique replica return ratio* (RRR_{uniq}) to denote the ratio between the maximum stored replicas upon a *put* and the returned unique replicas. We define it as $RRR_{uniq} = \frac{RRC_{uniq}}{k}$. Accordingly, an RRR_{uniq} of 100% corresponds to retrieving k unique replicas with a *get* operation.

6 Evaluation

In the following, we evaluate the *put* and *get* operations in a simulated Kademlia network acting as a core network of storing nodes as described in the system model in Section 4. Specifically, we evaluate on how many nodes a value is stored at upon a single *put* operation and how many unique replicas of this value we can retrieve afterwards with a *get* operation. We first describe the evaluation environment and the simulation scenarios. Then, we present our results and discuss them.

6.1 Environment and Parameters

For our simulations, we use the Java-based network simulation software PeerSim [9]. We extended the partial Kademlia implementation from the PeerSim website [10] to implement the full Kademlia protocol. Additionally we wrote software components to provide functionality for network churn and data storage and retrieval.

We proceed from the fact that different parameters may affect the storage resilience in Kademlia.

Kademlia Bucket Size: The bucket size k defines how many nodes should store a value upon a put operation. Moreover, it determines how many nodes are requested for a value upon a get operation. We evaluate the implication of this parameter by setting it to different values. In our simulations, we use the values 10 to 50 in steps of five.

Network Size: We consider two differently sized networks, i.e., one with 2500 nodes and one with 5000 nodes.

Network Churn: We consider three churn scenarios, where the same number of nodes join and leave the network each minute, keeping the network at roughly the same size. The numbers are selected so that in the simulated 24 hours the number of joins/leaves are either one, two, or four times the network size. We call these low, medium, and high churn. For a network of size 2500 the means

a join/leave rate of 2/2, 4/4, and 7/7, resulting in an average participation for a node of 500, 380, and 270 minutes. For a network of size 5000 the means a join/leave rate of 4/4, 7/7, and 14/14, resulting in an average participation for a node of 500, 400, and 270 minutes.

Data Traffic: For nodes to fill and update their routing tables, each node performs 10 get operations with random *ids* per minute throughout the whole simulation. Additionally, Kademlia requires each node to perform a so-called "bucket-refresh" every 60 minutes for maintenance purposes. For this, a node randomly generates an *id* from the *id* range of each k-bucket and performs *find_node* RPCs for these *ids*.

6.2 Simulation Phases:

The initial bootstrap procedure to create the network is performed randomly in terms of time and bootstrap node selection. A new node joins the network at a random point in the simulated time that is evenly distributed between 0 and 30 minutes. The bootstrap node is randomly chosen from the nodes already present in the network. Therefore, in all simulations the network is fully setup after 30 minutes (setup phase). From minute 30 to minute 60 (stabilization phase), we allow the network to stabilize. After that, starting at minute 60, we apply churn (churn phase). The churn phase lasts 25 hours.

6.3 Measurements

During the first half hour of the churn phase we carry out 1000 put operations. We select the exact point in time, the value id, and the node performing the put operation, randomly. Each put operation is followed by get operations for the same id and with randomly selected node. These get operations take place 1 minute, 5 minutes, 30 minutes, and 1 hour after their respective put operation, and then every hour until 24 hours have passed.

For each *put* operation we log the replica storage count RSC. For each *get* operation we log the unique replica return count (RRC_{uniq}) and the necessary effort *Eff*, which is the total number of requests sent during the operation.

Our evaluation is based on three simulation runs for each parameter set. We initialized the simulator's random number generator with a different seed for each run. This results in a different network setup, different value ids, and different nodes carrying out the *put* and *get* operations, while keeping the parameter set identical. Hence, our evaluation is based on 3000 *put* operations and 27·3000 get operations for each combination of network size, churn, and k value.

6.4 Results

We first present the results for the replica storage count RSC. After that we show the results for unique replica recovery rate RRR_{uniq}. We conclude our results by presenting the values of k necessary for achieving a required unique replica recovery count RRC_{uniq} and, thereby, the number of replicas an attacker would have to suppress for successful censorship.

Replica Storage Count For almost all *put* operations in all simulations the RSC was equal to its maximum value of k. Few exceptions occured due to the circumstance, that the *store* RPCs of a *put* operation take place only after the *find_node* RPC has finished. Finding a node and storing a replica on it is not atomic. A node selected for storage might leave the network just before receiving a *store* RPC, reducing the RSC by one. Still, this rarely happened and the RSC reached k in the vast majority of cases despite the churn. Therefore, we assume $RSC = k$ from here on.

Unique Replica Retrieval Rate In the evaluation of the RRR_{uniq} only the worst retrieval result for each *id*-valuepair is considered. The first step is the selection of the get operation with the smallest unique replica retrieval count RRC_{uniq} (out of 27 over a range 24 hours) for each of the 3000 put operation. We then calculate the arithmetic mean of the resulting 3000 RRC_{uniq} values and divide it by k to get the average unique replica retrieval ratio RRR_{uniq}. We further compute the arithmetic mean of the effort **Eff** taken in a get operation, based on all 27·3000 get operations of a parameter set.

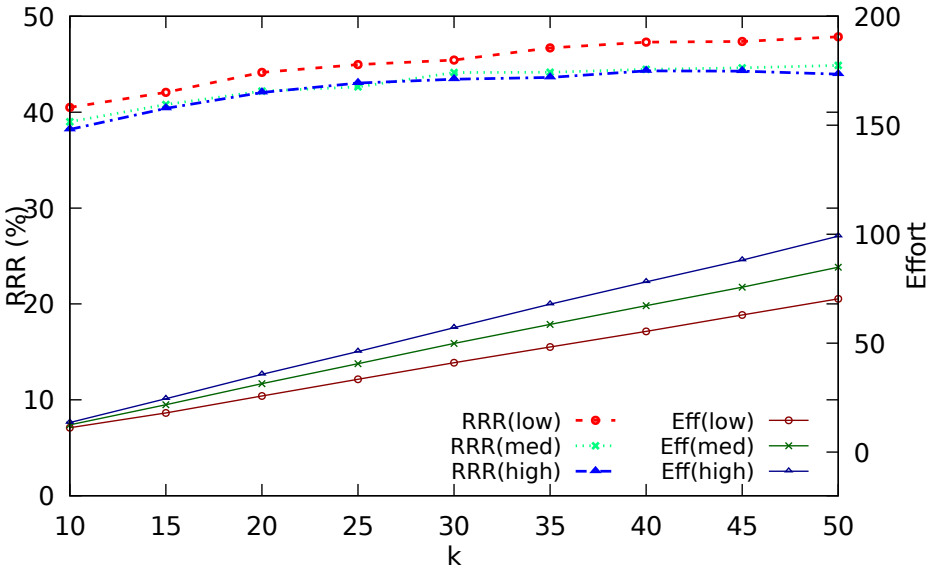

Fig. 1. RRR$_{unique}$ and **Eff**, Size 2500, 3 churn scenarios

The Figures 1 and 2 show the mean RRR_{uniq} and Eff over k for different churn in networks of with 2500 and 5000 nodes. In total, the graphs for both network sizes look very similar, matching previous results on the scalability of Kademlia [5,6]. The RRR_{uniq} values for the larger network are just slightly lower,

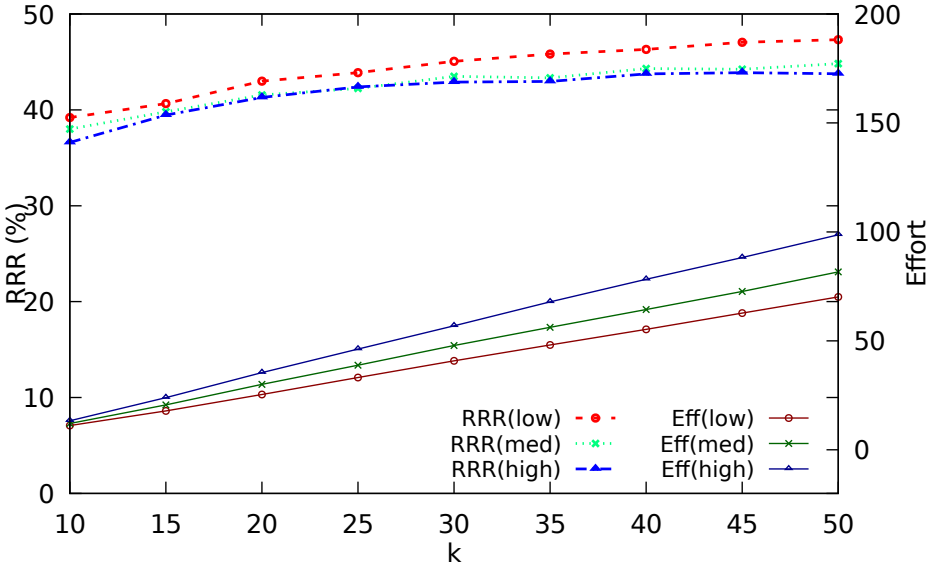

Fig. 2. RRR$_{unique}$ and **Eff**, Size 5000, 3 churn scenarios

indicating that Kademlia also scales well with the network size in terms of data storage and retrieval.

For both network sizes the retrieval effort Eff grows with k, which is to be expected, since k determines the number of replicas to retrieve. Additionally the effort also scales with the degree of churn. The higher the churn, the more likely it is that a node's routing table contains nodes which have already left the network. Therefore, the probability of sending requests without getting an answer, and in turn the retrieval effort, increases.

Against our expectations, the unique replica retrieval rate is not close to a flat line, but slightly increases with k in each churn scenario. For $k = 10$ it is at or slightly below 40% and grows by about 6% towards $k = 50$. Hence, an increase in k gives an increase in the unique replica retrieval count RRC_{uniq}that is beyond linear, while the effort for retrieval is linear.

Among the churn scenarios we see that, as expected, higher churn hurts the retrieval rate. This is due to a greater loss of nodes with data on them and greater influx of nodes that initially have no data. Still, the decrease in retrieval rate is only at about 5%.

Sufficient k Besides evaluating the mean RSC and RRR_{uniq} for different parameter sets, we also determined lower bounds for k that can provide a specific RRC_{uniq} with very high probability. The unique replica return count RRC_{uniq} is a decisive value for resilience against censorship. The more unique replicas one is likely to retrieve, the more replicas need to be suppressed by an attacker.

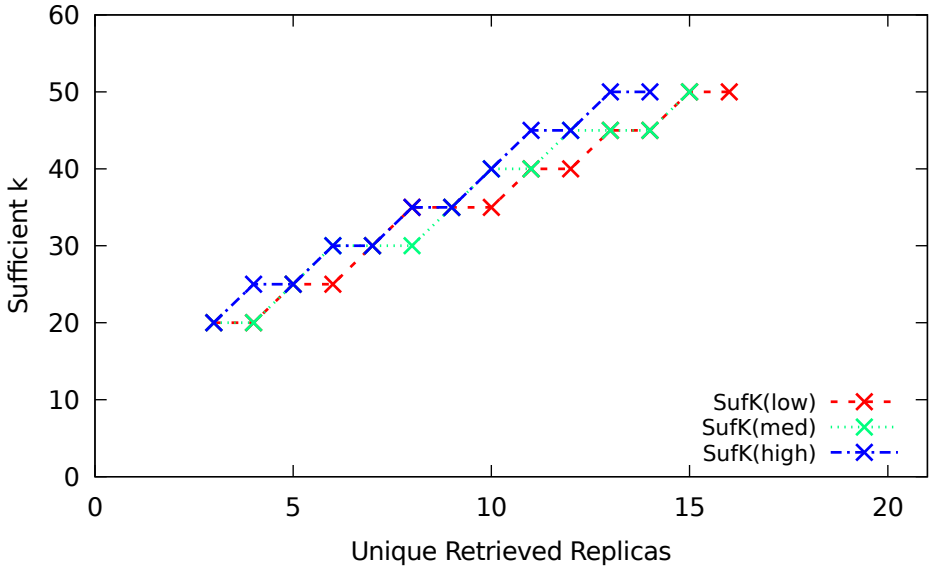

Fig. 3. k_{suf} for RRC_{uniq} with 100%, Size 2500

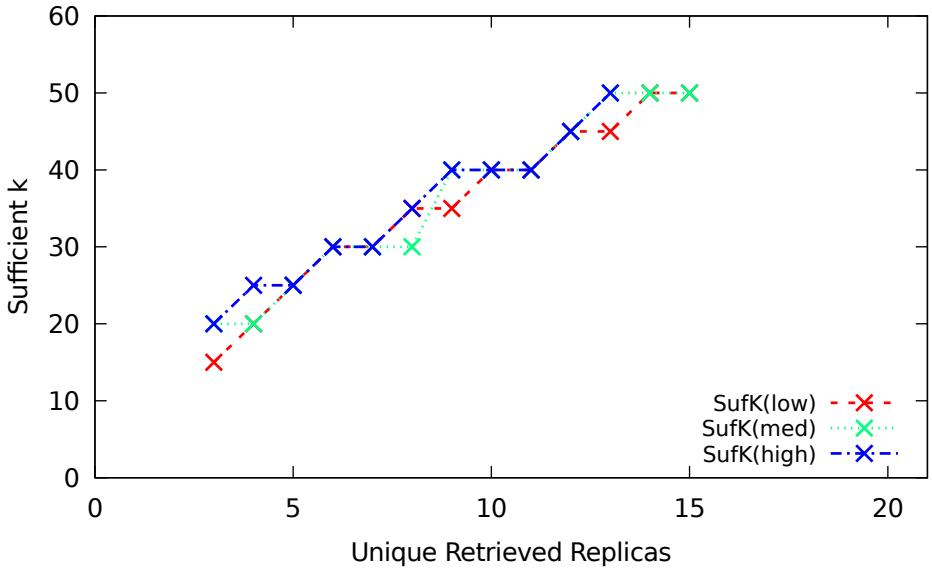

Fig. 4. k_{suf} for RRC_{uniq} with 100%, Size 5000

We, therefore, determined the values of k that are sufficient to reach a specific RRC_{uniq} with a preset probability. We call these values k_{suf}. The RRC_{uniq} values in Figures 3 and 4 were achieved by 100% of all get operations for each respective parameter set. This means that, within our simulations, these are guaranteed values. If, e.g., the requirement is to have an RRC_{uniq} of at least 11 for all get operations in the network with 5000 nodes and with high churn (Figure 4), a k value of 40 is sufficient. For a required RRC_{uniq} of 12 the sufficient value of k in the same scenario is 45. These jumps in value, visible as stairway effect in the graphs, exist due to the steps of 5 between the k values in our simulations. Therefore, our k_{suf} between those k values might even be a bit more conservative than absolutely necessary.

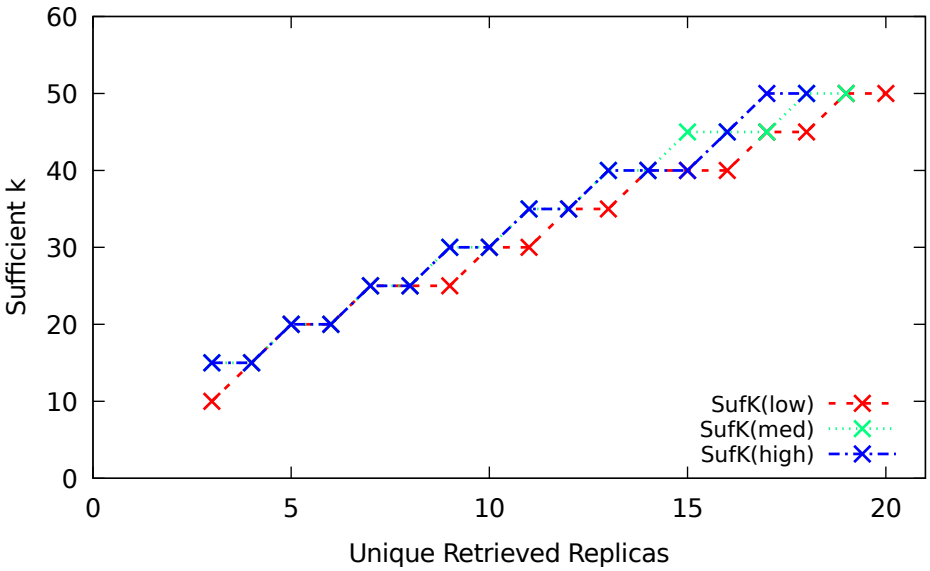

Fig. 5. k_{suf} for RRC_{uniq} with 90%, Size 2500

The RRC_{uniq} values in Figures 5 and 6 were achieved by 90% of all get operations for each respective parameter set. With the lower 90% requirement, as shown in Figure 6, the sufficient k for achieving an RRC_{uniq} of 11 and 12 drops to 35 for both values. Also, the RRC_{uniq} achievable with the maximum k of 50 increases from 16 to 20.

It is noticeable that with the 100% requirement, the values for k_{suf} in the different churn scenarios, though distinguishable, are very close to each other and often overlap. With the 90% requirement overlap is almost the norm, and in Figure 5 the k_{suf} for medium churn at the RRC_{uniq} value 15 is even slightly higher than the k_{suf} for high churn. Based on this, we conclude that the diffe-

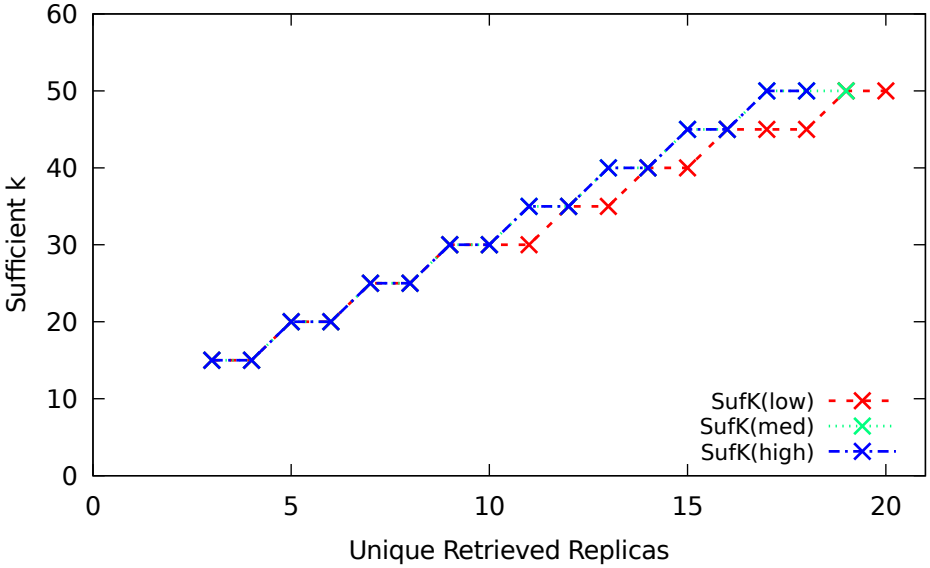

Fig. 6. k_{suf} for RRC_{uniq} with 90%, Size 5000

rence in RRC_{uniq} for the different churn scenarios is based in only about 10% of get operations performing significantly worse with the higher churn.

In conclusion of the evaluation, highly redundant short-term data storage and retrieval with Kademlia is possible for our network scenarios. Scaling the network's storage resilience with the parameter k allows for strong assurances regarding lower resilience bounds even with churn.

7 Conclusion & Future Work

In this paper, we analyzed the storage resilience of the overlay network and distributed hash table Kademlia. For this we performed and evaluated a large number of simulations with different Kademlia parameters and network characteristics. We evaluated the actual redundancy of storage operations, the number of retrievable unique data replicas and as well as the retrieval effort. Beyond that we calculated lower bounds of the parameter k for achieving specific numbers of unique data replicas and, thereby, a minimum number of replicas an attacker needs to suppress for successful censorship.

In future research we will further test the storage resilience with different churn types proposed in Roos2015 [13] and perform further simulations with additional values for k.

Acknowledgment

We thank the German Research Foundation (DFG) for their support within the project CYPHOC (WA 2828/1-1).

References

1. Cai, X.S., Devroye, L.: The analysis of kademlia for random ids. Internet Mathematics 11(6), 572–587 (2015)
2. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A distributed anonymous information storage and retrieval system. In: Designing privacy enhancing technologies. pp. 46–66. Springer (2001)
3. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. Tech. rep. (2004)
4. Fedotova, N., Fanti, S., Veltri, L.: Kademlia for data storage and retrieval in enterprise networks. In: CollaborateCom 2007. pp. 382–386. IEEE (2007)
5. Heck, H., Kieselmann, O., Wacker, A.: Evaluating connection resilience for self-organized cyber-physical systems. In: SASO'16. IEEE (September 2016)
6. Heck, H., Kieselmann, O., Wacker, A.: Evaluating Connection Resilience for the Overlay Network Kademlia. In: ICDCS. pp. 2581–2584. IEEE, Atlanta, GA, USA (June 2017)
7. Inc., A.S.: Vuze wiki (2012), `https://wiki.vuze.com/w/Distributed_hash_table#How_it_works`, (accessed July 5th, 2017)
8. Ji-Yi, W., Jian-Lin, Z., Tong, W., Qian-li, S.: Study on redundant strategies in peer to peer cloud storage systems. Applied mathematics & information sciences 5(2), 235S–242S (2011)
9. Montresor, A., Jelasity, M.: Peersim: A scalable p2p simulator. In: P2P. pp. 99–100. IEEE (2009)
10. Montresor, A., Jelasity, M.: Peersim: A peer-to-peer simulator. http://http://peersim.sourceforge.net/ (2016 (accessed February 20, 2018)), `http://http://peersim.sourceforge.net/`, (accessed February 1st, 2018)
11. Park, G.S., Song, H.: A novel hybrid p2p and cloud storage system for retrievability and privacy enhancement. Peer-to-Peer Networking and Applications 9(2), 299–312 (2016)
12. Risson, J., Moors, T.: Survey of research towards robust peer-to-peer networks: search methods. Computer networks 50(17), 3485–3521 (2006)
13. Roos, S., Nguyen, G.T., Strufe, T.: Integrating churn into the formal analysis of routing algorithms. In: Networked Systems (NetSys), 2015 International Conference and Workshops on. pp. 1–5. IEEE (2015)
14. Roos, S., Salah, H., Strufe, T.: Determining the hop count in kademlia-type systems (2015)
15. Roos, S., Schiller, B., Hacker, S., Strufe, T.: Measuring freenet in the wild: Censorship-resilience under observation. In: International Symposium on Privacy Enhancing Technologies Symposium. pp. 263–282. Springer (2014)
16. Urdaneta, G., Pierre, G., Steen, M.V.: A Survey of DHT Security Techniques. ACM Computing Surveys (CSUR) 43(2), 8 (2011)