Capturing Connectivity Graphs of a Large-Scale P2P Overlay Network

Hani Salah and Thorsten Strufe

TU Darmstadt, Darmstadt, Germany
{hsalah, strufe}@cs.tu-darmstadt.de

***Abstract*—Measuring accurate graph snapshots of peer-to-peer (P2P) overlay networks is essential to understand these systems. Furthermore, the captured graph snapshots can be used, among other important purposes, as traces for simulation studies, to validate existing simulation models, to design and implement targeted attacks, or to detect anomalies.**

Motivated by the importance of the purposes above as well as the popularity of several Kademlia-like networks, we present a new crawler aiming to capture snapshots of the *connectivity graph* of the *entire* KAD network. The crawler's design is generic and adaptable for Kademlia-like and other structured P2P networks. The results show that the crawler is fast and captures high accurate graph snapshots. Furthermore, its design enables it to outperform prior KAD crawlers significantly in terms of the time and the number of crawling messages that are required to download nodes' routing tables.

The crawls that we conducted at different times between April 2012 and February 2013 show that KAD is still widely-used in terms of *total* observed users. However, when compared to the results of prior studies, we report a significant drop in the number of its *simultaneous* online users.

***Index Terms*—Measurement, Crawler, KAD, Connectivity Graph.**

I. Introduction

The characterization of *global connectivity properties* of large-scale distributed systems is very important to understand their operation and to improve their design. This can be achieved by capturing accurate graph snapshots of these systems.

In addition to making accurate statements about these systems, the captured graph snapshots can be employed for several important purposes. For instance, (*i*) to build new realistic network simulation models, or to validate and upgrade existing ones, (*ii*) as traces for simulation-based studies, and (*iii*) to investigate potential improvements on the protocol designs. In addition, the analysis of real graph snapshots shows, among other information, graph-theoretic properties of the nodes, e.g., high centrality. These information can be used, for instance, to perform targeted eclipse attacks that are more effective than previously implemented ones (e.g., [1] [2]), or alternatively to identify anomalies (e.g., occurring attacks).

Due to the significance of the purposes above, as well as the high popularity of several peer-to-peer (P2P) systems that implement Kademlia [3] as an overlay network, we present a new crawler to capture graph snapshots of KAD. The crawler is also adaptable to measure other rapidly growing Kademlia-like networks, such as the overlays used in BitTorrent clients like Vuze, uTorrent, and BitComet, and other structured P2P networks.

Our crawler's goal is to capture graph snapshots of the entire KAD network. This task consists of two *interleaving* functions: (*i*) discovering all the contactable nodes of the network, and (*ii*) downloading their whole routing tables. In order to obtain *representative* snapshots, they should be captured: (*i*) instantly and (*ii*) completely, i.e. include all graph vertices (nodes) and edges (connections between nodes) [4]. However, when it comes to the reality, it is impossible to satisfy these conditions. Instead, we define two more realistic requirements for our crawler: (*i*) it should minimize the measurement time, and (*ii*) it should maximize the completeness of snapshots.

We used our crawler to perform hundreds of partial crawls and dozens of full crawls on KAD, at different times between April 2012 and February 2013. The results of these crawls show that the crawler is fast and is able to capture graph snapshots that are *accurate enough* to be employed for the purposes that we listed above. We attribute these results to the following design and implementation features:

1) The crawling approach takes advantage of the design of KAD much better than prior crawlers.
2) The measurement task is *distributed* over several crawling hosts.
3) The crawling algorithm continuously measures loss of crawling messages, and adapts crawling rates accordingly.

Our results show that KAD is still used by several millions of users. However, when compared to prior results, e.g., [5] [6] [7], we report a significant change in KAD's *usage patterns*, represented by a high drop in the number of its *simultaneous* online users.

The remainder of the paper is structured as follows. Sec. II discusses the related work. Sec. III presents relevant background of KAD. Sec. IV describes the crawling methodology and architecture. Sec. V describes our dataset, our crawler's reported speed and completeness of captured snapshots, as well as the current status of KAD. Sec. VI concludes the paper and states our ongoing and future work.

II. Related Work

In recent years, there has been a considerable number of measurement-based studies for structured P2P systems. These studies were able to characterize several important system properties. In this section, we focus on prior crawling-based studies and distinguish them from our work.

Kutzner et al. [8] developed a crawler for *Overnet* to measure the network size, availability of nodes, distributions of nodes in the underlay network as well as distribution of their round-trip-times (RTTs). The *BitMon* crawler [9] measures part of the BitTorrent *Mainline DHT* to estimate several statistics about the network usage. Closer to our work, there have been studies crawled *KAD*. For instance, *Cruiser* [4] measured the churn phenomenon over parts of KAD. *Blizzard* [5] is used to measure several properties such as user sessions, geographical distribution of users, and identity aliasing. The design of [6] is very similar to Blizzard. However, they achieved slight improvement over Blizzard's performance through distributing the crawling task on multiple crawling hosts, in addition to some changes in the crawling algorithm. *Rainbow* [10] measured general peer properties, clients' software versions, and popularity of shared files.

The common thing in the studies above is that they discover online KAD nodes, either over the whole network or parts of it. They do so by downloading *partial* routing tables of all discovered nodes. Differently, there are other related studies whose crawling objectives require downloading the whole routing tables, but only for a *small sample* of nodes. This approach is implemented, for example, by [11] to perform some attacks, by [12] to study the performance of nodes responsible for replication, and by *kFetch* [13] to study some routing table properties.

It is worth to mention that the designs of the crawlers above (even those which crawl the whole network, e.g., [5] [7]) require substantial changes to be adapted efficiently for the purposes (planned studies) that we listed in Sec. I.

In terms of the crawled information, *Rememj* [14] is he closest to our crawler. Likewise, *Rememj* aims to capture snapshots that include *all* contactable online nodes as well as their *whole* routing tables. The snapshots captured by *Rememj* are used to study identity repetition [15] and lookup traffic [7]. Despite this similarity, our crawler's design applies a number of novel techniques, which distinguish it from *Rememj* and other KAD crawlers. These techniques (see Sec. V) enable it to outperform them, in terms of the time and the number of crawling messages that are required to download nodes' routing tables, as we discuss in Sec. V-B.

III. KAD Backround

In this section, we describe the details of KAD that are directly related to the design of our crawler. For a detailed description of KAD, the reader is referred to [16] and [17].

KAD is an implementation of the structured P2P network, Kademlia. It is integrated, as a discovery overlay to eMule, a widely-used P2P file-sharing system. It is used to facilitate the content delivery, by enabling the nodes to find shared content and other nodes, in a fully distributed way.

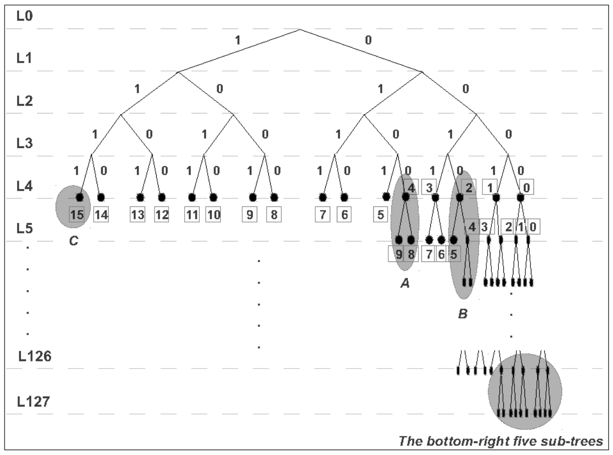

Fig. 1: The routing table structure of KAD: representing the distances in the ID-space (adapted from: [11]). The left part stores distant contacts, while the lower-right part stores closer ones. Each leaf (e.g., ***C***) represents a bucket of contacts. ***A*** and ***B*** are examples of two-bucket and three-bucket sub-trees, respectively.

Each new KAD node acquires a randomly generated 128-bit identifier, called the *KAD ID (KID)*. Each KID represents a unique position in the *ID-space*. The *distance* between two keys x and y is defined by the *XOR* of their values.

The routing table: Each KAD node v maintains a *routing table* to store information of other nodes, also called *contacts*. The structure of the routing table is illustrated in Fig. 1. It is organized as a 128-level tree. Starting from level 4, each level contains *buckets* of contacts. The buckets are assigned contacts according to their position in the tree, such that each position represents a specific distance from v's KID. Each bucket stores up to ten contacts, and buckets indexed 0 to 4 split further to store new contacts when the original bucket becomes full. Contrariwise, removal of contacts results in bucket merging. Each bucket has a prefix, whose length is determined by the level of the bucket in the tree.

The routing process: The *routing process*, also called *lookup*, relies on two UDP-based messages: *Kademlia_request* (KAD_REQ) and *Kademlia_response* (KAD_RES). Among other fields, the KAD_REQ message contains: the address of the destination node (its KID, IP address, and UDP port), β (the number of requested contacts), and the *target-ID* field. To find a route from a source node S to a destination key k, S first sends *parallel* KAD_REQs to α contacts selected from its routing table's bucket that has the longest matching prefix with k's ID. Each queried node, if available, answers by a KAD_RES message containing contact information of β possible next-hop contacts, whose KIDs are the closest to the requested target-ID[1]. This process continues *iteratively* till S reaches k, or contacts that are very close to k.

[1]In KAD, by default α is 3, while β is set to 2, 4, or 11, according to the search-type.

The flooding protection policy: KAD *recently* introduced a threshold on the number of KAD_REQs that can be *legally* received from another node within a specific period of time. In particular, a node is allowed to send another node only a single KAD_REQ within a period of six seconds. Otherwise, more frequent requests are considered malicious and are dropped. Furthermore, repeated misbehaviour is treated by banning the requesting node completely.

IV. Developing a Crawler for KAD

This section introduces our crawling methodology in Sec. IV-A, the crawler's architecture in Sec. IV-B, and two generic techniques to reduce the loss of crawling messages in Sec. IV-C.

A. Crawling Methodology

We define the *KAD graph* as a directed graph $G = (V, E)$ where V is the set of *vertices*, i.e. online KAD nodes, and E is the set of *edges* between vertices. For example, the directed edge $e_{i,j}$ refers to a routing entry, i.e. a known contact, in node i's routing table towards node j.

In compliance with the definition above, our crawling algorithm works as follows. Each time the crawler bootstraps, it contacts an initial list of previously discovered nodes and downloads their routing tables. Simultaneously, it extracts new contacts from downloaded routing tables, and adds them to the list. This procedure iterates over the list till discovering all contactable nodes and downloading their whole routing tables[2].

Our crawling methodology takes advantage of the design of KAD to improve its performance and to reduce its crawling costs, better than prior KAD crawlers that we are aware of. In particular, the crawler applies the following two *novel* techniques.

Dividing the routing table tree into sub-trees: We divide the routing tree into 256 non-overlapping sub-trees: 129 two-bucket sub-trees and 127 three-bucket sub-trees. In Fig. 1, the sub-trees A and B are examples of two-bucket and three-bucket sub-trees, respectively. The sub-trees are indexed 0-255, starting from the tree's top-left part towards bottom-right.

To download the routing table of a destination node D, the crawler sends only a single KAD_REQ per sub-tree. This message is sufficient to download the contacts that are stored in the sub-tree's buckets. Given that each bucket can store up to ten contacts (Sec. III), the crawler sets the values of the fields of KAD_REQ as follows. It sets β to 30 if the message is correlated with a three-bucket sub-tree, or 20 otherwise. The value of the *target-ID* field is computed as: ***D's KID*** $\oplus$ ***sub-tree root's ID***, where the *sub-tree root's ID* is a 128-bit value representing the sub-tree root's position in the main tree.

Avoiding querying empty buckets: In order to avoid sending KAD_REQs that correlate to sub-trees whose buckets are empty or do not exist in a destination node D's routing table, the crawler first sends five KAD_REQs, using $\beta = 1$, to discover one contact in each of D's five lowest (closest) sub-trees (see Fig. 1). Next, the crawler shifts to the top of the tree, and starts downloading sub-trees' buckets from top-left towards bottom-right, in order. This step continues till the crawler encounters the previously learned five *closest* contacts again. At this point, the crawler stops sending KAD_REQs to D because this implies that there are no additional contacts, deeper in D's routing tree, to discover.

B. The Crawling Architecture

Broadly speaking, crawlers can be designed with different levels of distribution: *centralized* (using a single crawling host) like Blizzard [5], *shared-memory distributed* like Cruiser [4], or *equally distributed* like [6].

Our measurement task involves sending (and receiving) massive amounts of data traffic, as well as performing high amounts of *on-the-fly* data processing, within a very short period of time[3]. Since we do not have a single machine that can handle all these requirements in a reliable way, in addition to the high effect of KAD's *flooding protection policy* on the crawling speed in case of using a single crawling host, we exclude the *centralized* approach from our design options.

The other two approaches distribute the crawling task and its costs on a number of distributed crawling hosts, which facilitates higher request rates and implies less overhead per machine.

In the *shared-memory* approach, the crawling hosts work on a single list storing information of all discovered nodes. This design boosts the degree of crawling parallelism (and thus, its speed) to its highest level, and makes this approach the least-affected by the *flooding protection policy*. Nevertheless, this expected high crawling rate is very likely to cause high traffic congestion, and thus a high loss in crawling messages, as we discuss in Sec. IV-C. Furthermore, this approach consumes high CPU time and network bandwidth for the synchronization among the crawling hosts.

Differently, the *equally distributed* approach partitions KAD's ID-space into equal non-overlapping sections and distributes these sections *evenly* on the available (equal) crawling hosts. Each crawling host works independently and exclusively on its own assigned ID section. This way, crawling hosts require no synchronization among each other. We tested this approach during our full crawls and it achieved very high performance, as we describe in Sec. V-B. Although it is slightly slower than the *shared-memory* approach, it achieves much higher degree of data completeness. Due to these reasons, we chose the *equally distributed* approach as an architecture for our crawler.

In our study, we used the German-lab (G-Lab) testbed [18] as a basis for our measurements. G-Lab provides us with a set of equal Linux boxes. In particular, we used 64 machines as crawling hosts, one machine as a monitor, and ten machines as backup crawling hosts (to take place of crawling hosts in case

[2]We do not list the crawling algorithms here due to space restrictions.

[3]Some statistics about the network size and crawling messages are discussed in Sec. V-A.

of failures). Implementing the *fully distributed* architecture, we partitioned the KAD ID-space evenly into 64 sections, and assigned each crawling host a unique section. The *monitoring machine* has a passive role and it is not involved in the crawling task. It only monitors the operation of the main crawling hosts, and in case some of them crash or malfunction, the monitor automatically replaces them by a similar number of backup machines.

C. Reducing Loss of Crawling Messages

Loss of crawling messages is a general problem that crawlers are likely to encounter during their operation, due to the following reasons. First, crawlers usually send and receive massive amounts of messages within a short period of time. Second, in the case of KAD crawlers, the crawling algorithms are based on the original messages of KAD, which rely on the *unreliable* UDP transport protocol. Loss of crawling messages, in turn, lead to decrease the completeness of captured graph snapshots.

In order to mitigate the effects of the problem above, we implemented the following two techniques in our crawler:

Adaptive crawling rates and delays: This technique adapts packet-sending rates (and delays) to the measured loss of crawling messages. In practice, the crawler continuously measure loss of messages by parsing a number of system files that report traffic-related values[4], and adapts the packet-sending rate accordingly. During our experiments, this technique decreased the loss of crawling messages significantly. However, this improvement came at the cost of increasing the total crawling time, as we discuss in Sec. V-B.

Multiple crawling: This technique assigns each crawling host the responsibility of crawling r ID-space sections ($r \geq 2$). Consequently, each ID-space section is crawled r times. This technique is beneficial because packets are lost during measurements at random. That is, losing a certain packet p in r parallel crawls is very unlikely. As a result, the fraction of lost packets decreases *exponentially* to the number of crawling host duplications (r).

Loss of crawling messages can be further reduced by decreasing the sizes of ID-space sections that are assigned to each crawling host. However, all these solutions increase the measurement costs linearly.

V. Dataset and Results

This section discusses our dataset in Sec. V-A, our crawler's reported crawling time and completeness of the captured graph snapshots in Sec. V-B, and the current status of KAD in Sec. V-C.

A. Dataset

Our measurements were conducted at different times between April 2012 and February 2013. In particular, we performed hundreds of partial tests, and 60 refined crawls over the entire KAD network, i.e. full crawls.

[4]The files are: */proc/net/dev* and */proc/net/udp*.

Table I gives an overview of the 60 full crawls. It included the maximum, minmum, median, and mean values of: the number of unique online KAD nodes[5], the number of all discovered KAD nodes (both online and offline contacts), and KAD_REQs that are used to download a whole routing table. Please note that we count online nodes by their unique UDP socket identities; i.e. (*IP address, UDP Port*) pairs. Observed unique IP addresses were slightly less than unique UDP sockets, because of possible mapping from a single IP address to multiple unique UDP ports. The *identity replication* problem was discussed intensively in prior work, e.g., [5] [15], and it is out of the scope of this paper.

This paper focuses on the crawler's design and performance. The analysis and results of connectivity graph properties are out of the scope of this paper.

TABLE I: Overview of 60 Full Crawls

	Online Discovered Nodes	All Discovered Nodes	Sent KAD_REQs (Per Routing Table)
Maximum	508,059	5,705,002	74
Minimum	352,390	3,614,411	23
Median	469,966	3,941,264	33.5
Mean	452,736	3,812,027	31.8

B. Performance of the Crawler

In this section, we discuss our crawler's reported *crawling time* and *completeness of captured snapshots*.

Crawling time: The total time that is required to perform a crawl is composed of two *interleaving* time components: (*i*) the nodes' discovery time, and (*ii*) the time that is required to download their routing tables. The value of the second component depends on two factors: (*i*) the maximum number of KAD_REQs that are required to download the routing tables of discovered nodes, and (*ii*) the six-second restriction of KAD's flooding protection policy.

During our full crawls, the mean value of the first time component was 189.1 seconds, and the mean value of the maximum number of required KAD_REQs was 74. The worst case scenario is when the node that requires the maximum number of KAD_REQs is the one that is discovered last. In this case:

Crawl time = 189.1 + (74 x 6) = 633.1 seconds.

However, due to the effect of the *adaptive crawling rates and delays* (Sec. IV-C), the reported mean value of crawling time increased to about 900 seconds.

To *evaluate* the impact of the reported crawling time on the accuracy of captured graph snapshots, we measure two relevant values: (*i*) the stability of discovered online nodes, and (*ii*) the stability of downloaded routing tables, with reference to the reported crawling time.

We cannot rely on previously reported results of user sessions and churn in KAD (e.g, [19] [5]) because of the

[5]These values count only nodes that are stable from the start till the end of the crawl.

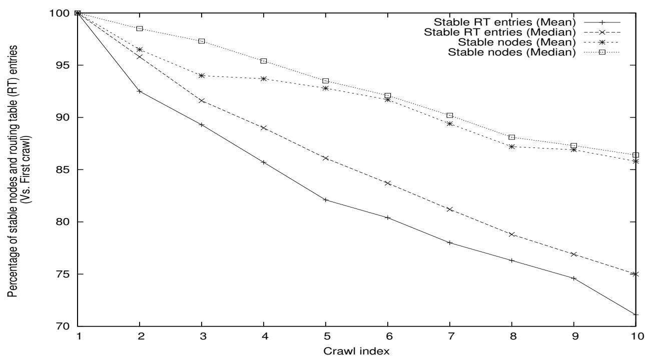

Fig. 2: Measured dynamics of routing table entries and nodes. The x-axis represents the crawl index from 1 to 10, and the y-axis represents the percentage of stable nodes and routing table entries (contacts) in crawls 1 to 10 with reference to crawl 1.

recent dramatic changes in the usage patterns of KAD, as we report in Sec. V-C. Instead, we conducted a small-scale measurement study using a modified version of our crawler. The modified crawler measures the stability of nodes as well as the stability of routing tables, over *different* time durations. In practice, the crawler discovers n random online nodes, and then it downloads their routing tables for t subsequent times, once every s seconds. We experimented with different values of n, t, and s. We do not show all the results due to space restrictions. Fig. 2 shows the results of an experiment applying $n = 1{,}000$, $t = 10$, and $s = 900$ (similar to the mean value of crawling time) The results show the percentage of stable nodes and stable routing table entries between the first crawl and each of the following $t-1$ crawls. In this example, the mean values of identical nodes and routing table entries between crawl 1 and crawl 2 are 96.3% and 92.1%, respectively. These results consider only the nodes (subset of n) that stay online along all the t crawls.

Completeness of captured graph snapshots: Theoretically, completeness of the captured snapshots is measured with reference to a ground truth (in this case, the real KAD graph). However, such ground truth is impossible to achieve. Instead, we measure the completeness of the captured snapshots as the ratio of received KAD_RESs to the sent KAD_REQs, considering only stable, responding nodes. We argue that this completeness measure is sufficiently accurate due to the following reasons. First, every KAD node is very likely to be discovered many times; i.e. to be extracted from KAD_RESs received from many different nodes. Second, although NATed nodes can make use of the KAD overlay, they do not participate in the overlay itself [19]; i.e. they do not affect the measured completeness. Third, every discovered node, on average, receives 74 KAD_REQs from our crawler (see Table I), and it is very unlikely that all these messages (or their corresponding KAD_RESs) will be lost. In summary, while it is possible to lose some information about connections between nodes (which still can be detected by our completeness measure), it is very unlikely to entirely miss online nodes.

Our completeness measure reported 98% and 84% as mean values of data completeness during the partial tests and full crawls, respectively. As mentioned earlier, we attribute observed loss of crawling messages, i.e. incompleteness, to two main factors: (*i*) the massive amount of traffic (congestion) that our crawler generates, and (*ii*) the unreliable nature of the UDP-based messages of KAD. While we cannot change the type of KAD messages, we dealt with the massive amounts of crawling traffic. We could improve the completeness and achieve the reported completeness values, through the *distributed crawling architecture* as well as the applied crawling techniques of *adaptive crawling rates and delays*, *querying sub-trees of buckets*, and *avoiding querying empty buckets*.

Discussion: We summarize our crawler results during the full crawls as follows. We reported about 900 seconds as a mean value of crawling time. During this period of time, about 96.3% of the discovered online nodes stayed stable, and about 92.1% of those nodes' routing tables entries were also stable. We also reported 84% as a mean value of the completeness of captured graph snapshots.

Certainly, the graph snapshots that our crawler captures are not exact snapshots of the real KAD network. However, based on the results above, we argue that our graph snapshots statistically are very close to the real snapshots: the macroscopic properties of the network (e.g., network size, highly-connected nodes, degree distribution, clustering, and resilience in the face of breakdowns) are maintained. Consequently, we argue that the captured snapshots are representative enough to be used for the purposes (studies) that we listed in Sec. I. The reported results can be improved further by migrating our crawler to a larger-scale (more capable) measurement environment.

It is also worth to mention that the crawling speed of our crawler is *not directly comparable* with the speeds of prior crawlers, mainly due to the differences in the amounts of information that each crawler aims to collect. Furthermore, unlike our crawler, the crawling rates of prior crawlers were not restricted by the recently introduced *flooding protection policy* of KAD.

However, we compare our crawler with prior KAD crawlers in terms of the time and the number of crawling messages that are required to download the routing tables, as follows. Our crawler's technique of *querying sub-trees* enables it to download 2.5 buckets on average (25 contacts) per query (KAD_REQ), while all prior crawlers that we are aware of (except *Rememj* [14]) download only a single bucket (ten contacts) per query.[6] As for *Rememj*, it uses *bootstrap requests* instead of KAD_REQs, which enables it to download up to 20 *random* contacts per request.[7] However, we argue that our crawling methodology outperforms *Remej's*, due to the following reasons. First, *bootstrap responses* used by Rememj return *random* contacts, which does not assure the coverage of the whole routing table (as our methodology does). Second, *bootstrap responses* are likely to produce more duplicates than

[6]Our results become much better than other crawlers if they do not avoid querying empty buckets (which is not clearly stated in their descriptions).

[7]Some prior studies (Sec. II) and [20] assume that β (number of returned contacts) can be set only to 2, 4, or 11. However, β in reality can take any value between 0-31.

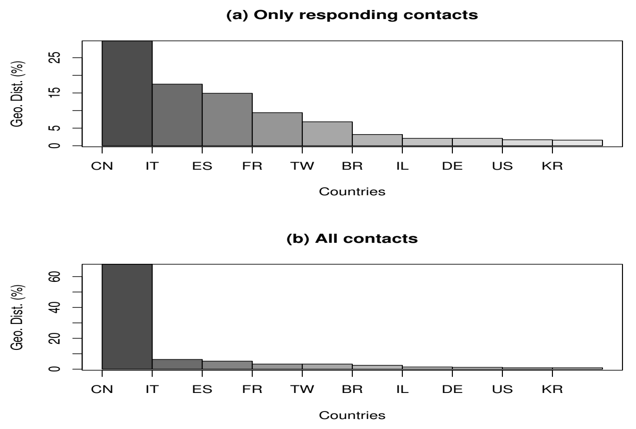

Fig. 3: The geographical distribution of KAD nodes (top 10 countries) seen on 13.07.2012: (a) shows the geographical distribution of online (responding) nodes, and (b) shows the geographical distribution of all contacts discovered in the routing tables.

targeted KAD_RESs. Third, the contacts returned by a single *bootstrap response* are fewer than the contacts that *can be* returned by a single KAD_REQ.

C. The Measurement Results of KAD

Network popularity: As can be seen in Table I, the values of discovered KAD nodes indicate that KAD is still a widely-used network, both in terms of the number of unique discovered nodes as well as the number of *simultaneous* online nodes. These results were almost similar in the 60 full crawls.

However, the reported results of *simultaneous* online nodes per crawl (second column) are significantly fewer than previously published results. For instance, Rememj [7] reported 1.2 - 1.8 million nodes in 2009, and Blizzard [5] reported 1.5 - 2 million nodes in 2007.[8]

Geographical distribution of users: Fig. 3 shows the distribution of nodes seen on 13.07.2012. The results seen in other crawls were similar. The main observation in these results is the existence of a high disparity between the geographical distribution of responding nodes (Fig. 3 (a)) and the geographical distribution of all nodes discovered in routing tables (Fig. 3 (b)). An extreme difference between the two distributions is obvious in the case of nodes from China: they represent 68% of all unique contacts, and only 29.8% of responding nodes. A similar observation was reported previously in [15]. However, the disparity values that we report in Fig. 3 are larger than the values reported in [15]. We plan to investigate the causes of this phenomenon in our future work.

VI. Conclusion and Outlook

In this paper, we introduced a new KAD crawler as a solution for capturing representative snapshots of the entire KAD graph. The crawler incorporated a set of design details and techniques that are adaptable for other P2P crawlers.

[8]In cooperation with Moritz Steiner, the developer of Blizzard [5], we validated our measured KAD population results via recent crawling tests using Blizzard. In particular, Blizzard discovered about 500,000 simultaneous online KAD nodes per crawl in August 2012.

The evaluation shows that our crawler is able to capture highly representative graph snapshots, and to download routing tables faster and using fewer messages than prior crawlers.

The results of KAD, that we collected at different periods during the last ten months, exposed a dramatic drop in the number of simultaneous online users, when compared to the observations of previous studies.

Currently, we are working on adapting our crawler for other Kademlia-style networks. As future work, we plan to perform an extensive analysis study on the collected graph snapshots, and to use the captured graph snapshots in several directions. For instance, to compare the properties of measured graphs with graphs of well-know simulators, to investigate possible improvements on the current protocol design, and to devise novel attacks to detect vulnerabilities of the measured systems.

Acknowledgments

This work was supported by the German Academic Exchange Service (DAAD), Grant no. A/09/97920. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. They also thank Moritz Steiner for his cooperation through performing tests on Blizzard, and thank the G-lab administrators for their help.

References

[1] T. Cholez, I. Chrisment, and O. Festor, “Monitoring and controlling content access in kad,” in *Proceedings of ICC*, 2010.
[2] Michael, M. Leske, and E. Rathgeb, “Conducting and optimizing eclipse attacks in the kad peer-to-peer network,” in *NETWORKING*, 2009.
[3] P. Maymounkov and D. Mazieres, “Kademlia: A peer-to-peer information system based on the xor metric,” in *Proceedings of IPTPS*, 2002.
[4] D. Stutzbach and R. Rejaie, “Capturing accurate snapshots of the gnutella network,” in *Proceedings of INFOCOM*, 2005.
[5] M. Steiner, T. En-Najjary, and E. W. Biersack, “Long term study of peer behavior in the kad dht,” *IEEE/ACM Transactions on Networking*, 2009.
[6] Q. Wu and X. Chen, “Advanced distributed crawling system for kad network,” *Journal of Computational Information Systems*, 2011.
[7] J. Yu *et al.*, “Monitoring, analyzing and characterizing lookup traffic in a large-scale dht,” *Computer Communications*, 2011.
[8] K. Kutzner and T. Fuhrmann, “Measuring large overlay networks the overnet example,” in *Proceedings of KiVS*, 2005.
[9] K. Junemann *et al.*, “Bitmon: A tool for automated monitoring of the bittorrent dht,” in *IEEE P2P*, 2010.
[10] X. Liu *et al.*, “Rainbow: A robust and versatile measurement tool for kademlia-based dht networks,” in *Proceedings of PDCAT*, 2010.
[11] P. Wang *et al.*, “Attacking the kad network,” in *Proceedings of SecureComm*, 2008.
[12] H. J. Kang *et al.*, “Why kad lookup fails,” in *Proceedings of IEEE P2P*, 2009.
[13] D. Stutzbach and R. Rejaie, “Improving lookup performance over a widely-deployed dht,” in *Proceedings of INFOCOM*, 2006.
[14] J. Yu and Z. Li, “Active measurement of routing table in kad,” in *Proceedings of CCNC*, 2009.
[15] J. Yu *et al.*, “Id repetition in kad,” in *Proceedings of IEEE P2P*, 2009.
[16] R. Brunner, “A performance evaluation of the kad protocol,” *Master thesis, Institut Eurecom and University of Mannheim*, 2006.
[17] D. Mysicka, “Reverse engineering of emule,” *Semester thesis, Swiss Federal Institute of Technology (ETH)*, 2006.
[18] “G-lab,” http://www.german-lab.de/.
[19] D. Stutzbach and R. Rejaie, “Understanding churn in peer-to-peer networks,” in *Proceedings of IMC*, 2006.
[20] M. Steiner, D. Carra, and E. W. Biersack, “Faster content access in kad,” in *Proceedings of IEEE P2P*, 2008.