PacketSkip: Skip Graph for Multidimensional Search in Structured Peer-to-Peer Systems

Andreas Disterhöft, Andreas Funke and Kalman Graffi
Technology of Social Networks, Heinrich Heine University of Düsseldorf
Universitätsstrasse 1, 40225 Düsseldorf, Germany
Email: {andreas.disterhoeft, andreas.funke, graffi}@hhu.de

***Abstract*—The Internet is ubiquitous and nodes participating in it are becoming increasingly diverse and heterogeneous. Especially is this true in decentralized, overlay-based networks, which traditionally build upon the infrastructure of the Internet. An ability to search for nodes with specific capabilities becomes of growing interest, for example, to support the node capacity discovery in large scale, distributed networks, such as in peer-to-peer systems. With PacketSkip, we propose a distributed, ordered, multidimensional indexing structure based on a Skip Graph, which operates on top of popular distributed hash tables. Its purpose is to offer, among other functions, capacity-based, high dimensional range searches for peers and their dynamic capacities. PacketSkip provides a very high search performance, also in comparison to other approaches, as well as low update and maintenance costs. Evaluations show that update and search durations are low and well scalable. Even in dynamic systems, its inner graph structure remains robust and stable.**

I. Introduction

Users nowadays participate in Internet services with a diverse range of device types. These devices are highly heterogeneous in terms of their capacity, such as processing power or storage. Latest research is increasingly focused on the Internet of Things (IoT), which increases the number of Internet-capable devices dramatically. Thus, the diversity of devices connected to the Internet rises further as these devices vary from strong desktop PCs to the much smaller smartphones as well as from smart sensors to toasters.

In decentralized networks, such as peer-to-peer (p2p) systems and decentralized IoTs, this device heterogeneity leads to challenges and opportunities. In p2p systems, for example, having weak peers in critical positions can result in underpowered peers being responsible for data which is in high demand. The high load may overstrain such a peer resulting in a bottle neck for the entire system. Several approaches for p2p and decentralized IoT networks have proposed to identify and select capable nodes with a minimum set of capacities, such as bandwidth or storage space, for dedicated roles in the networking protocols. Guinard et al. [9], for example, propose resource discovery of nodes in IoTs as a major research area. An example in p2p networks is the p2p overlay Omicron [4], which introduces roles for indexing, caching, routing and maintaining tasks in the p2p overlay. Complex, near real-time applications, such as social networks or video streaming, state strong requirements on fast response times to satisfy user experience and thus prioritize stronger peers for demanding tasks. Therefore, a function to find certain peers with specific capacities in the potentially vast network is desirable for various purposes. Peers using this capacity-based search would, for example, ask for all peers with a storage space of 120-150 MB, a upload bandwidth of 25-30 MBit/s and free memory of 50-60 MB. As a result they would expect all matching peers in the network.

A solution must detect and index the peers in a self-organizing fashion in order to support a quick and reliable access to them. Oppenheimer et al. [11] postulate that a challenge arises with respect to the dynamics of the node capacities, so that, placement decisions can become ill-suited after about 30 minutes. Solutions in the literature implement range queries in decentralized networks, such as CapSearch [5]. These solutions either lack in scalability of high dimensional data, completeness of the search result, reliability due to the use of super-peers or reliability during phases of high dynamics.

In this paper, we establish a distributed indexing structure atop distributed hash tables (DHT). This index is particularly providing a natural ordering so it can be utilized to search for value ranges. We apply such an index to subscript peers based on their capacities. We add the following characteristics to an existing structured p2p overlay: 1) natural key ordering, 2) applicability for range queries, 3) efficiency in dealing with multidimensional data and 4) coping with highly dynamic data and participants.

PacketSkip is an overlay independent service that allows range queries such as, 'return all peer-contacts within a certain upload bandwidth range AND with a certain minimum CPU performance'. Our contributions are as follows:

- We propose PacketSkip, a distributed, multidimensional indexing structure, which is based on a Skip Graph, where the peer-centric Skip Graph is moved towards a data-centric structure.
- PacketSkip is a light-weight overlay running on top of an existing key based routing (KBR) compliant p2p overlay [3].
- Evaluation using multidimensional and highly dynamic data, such as peer capacities, in PacketSkip.
- Comparison with CapSearch [5], a capacity-based search approach using kd-trees.

For this solution, we assume the following:

This work was supported by the DFG Grant GR4498 1-1 - OverlayMeter

- A key based routing compliant [3] DHT.
- A resilient DHT replication service, for example PAST [12].
- A monitoring service to monitor the number of participating nodes in the p2p overlay, for example SkyEye.KOM [8], [6].

As PacketSkip's internal, distributed Skip Graph structure adds another layer on top of a structured DHT-based p2p overlay, we exploit as many of the overlay's features as possible to avoid redundant functionality. Although PacketSkip is designed independently from a specific overlay implementation, we assume that the given overlay offers logarithmic lookup complexity, robustness against churn and a reliable replication mechanism for the data it maintains.

The remainder is organized as follows: In Section II we highlight related work and present then in Section III the protocol of PacketSkip. Evaluation methods and metrics, as well as the evaluation results are introduced afterwards in Section IV. Evaluation shows that PacketSkip is precise in capturing the actual capacities of the nodes, even under high dynamics, and provides a high search precision with low traffic overhead. In Section V, we conclude our work and provide suggestions for future work.

II. Related Work

Networks with heterogeneous peer capacities have been researched before, with several different proposals to search for and exploit peer capacities. In literature, access to the peer capacities is often granted through a new protocol and a Chord- [13] or tree-based search overlay.

Wette et al. [14] propose with Adaptive Heterogeneous Chord (AH-Chord) a second capacity-aware Chord-like overlay dedicated to order the peers by their bandwidth capabilities and thus providing a simple (range) search for peers with desired bandwidth capacities. This solution is, however, limited only to a single dimensional use case.

Kargar et al. [10] propose Fractal which aims at resource discovery in IoTs and builds up a hierarchical Chord structure. Here, so called entrepreneur nodes act as gate nodes to connect certain Fractals which are rings similar to the Chord structure. Further (super-)peer roles, such as exactitude and intercessor, exist to maintain the structure and provide the desired quality of service. High dynamics could harm the Chord-like structure, especially if super nodes are leaving. In contrast to Chord-based approaches, PacketSkip's action is more universal without limiting capacity awareness to a single dimension and PacketSkip is more robust against churn than a Chord-based approach. While indexed items are presented in sequenced order, it is universally to use with scaling capacity dimensions for various use cases. Logical nodes in PacketSkip are never rearranged, resulting in little maintenance costs while being highly resilient.

SkyEye.KOM, proposed by Graffi et al. [8], provides capacity-based peer search along with an aggregated monitoring service as an information management layer for p2p-systems. Aggregated data and a list of peer capacities is passed level by level from the leaves to the root with the root having complete knowledge of the aggregated metrics. In order to not overload nodes with the increasing set of node capacity information, the nodes of lowest quality are omitted, thus resulting in an incomplete capacity overview.

Dragon by Carlini et al. [2] is another recent aggregation tree-based approach for range queries of multi-attribute data. Dragon uses space filling curves to map the multidimensional capacity vectors to the single dimensional overlay. Inner nodes hold digest information of their subtrees and search is performed bottom-up just as the propagation of update information. Reducing the digest completeness for lower update costs may also reduce the approximation precision of the digest. By contrast to tree-based layers, PacketSkip is a non-hierarchical Skip Graph based approach. It offers a complete view and good load balancing properties without burdening nodes with global knowledge. As space filling curves can lead to a trade-off between computational requirements and precision, especially with high dimensionality and under high dynamics, our approach is to keep attribute information intact.

Skip Graphs have been introduced as a model by Aspnes et al. [1]. In contrast to DHT-based approaches, they expand on Skip Lists by supplying high resilience for use in distributed systems. The authors stress the benefits of a data structure that preserves the sequential order of keys compared to tools that provide only hash functionality. PacketSkip's protocol draws on this model by utilizing its properties for indexing peer capacities. Xu et al. proposed a service called SkipCluster [15] which is based on a Skip Graph and speeds up the performance of both exact and range search. Clusters are built by peers, thus this approach follows a hierarchical superpeer structure. Our focus is on homogeneous peer roles to support a high robustness against churn through the ability to react quickly.

Disterhöft et al. [5] proposed CapSearch, an algorithm that expands on the capacity-based search approach. Overlay nodes index a part of the capacity information and provide access to these ranges through a kd-tree on top of the regular DHT. It performs local calculations of the ranges and thus correct nodes can be identified for range searches. Leaf nodes hold full contact information which are returned to the requester by inner nodes. While range searches are fast and efficient, the update path of the kd-tree may be long, still CapSearch is promising. We review PacketSkip against CapSearch as part of our evaluation, but first in the next section, we describe PacketSkip in detail.

III. Our Solution: **P**arallel **A**ccelerated Capacity **KE**y-**T**uple Skip Graph (PacketSkip)

In this section we present our mechanism for multidimensional capacity search, named PacketSkip.

PacketSkip is a mechanism that offers the search function for peers with specific (multidimensional) capacities. For example, one could state a query in PacketSkip for 3 peers with free storage space of 200 MB - 500 MB as well as a free upload bandwidth of 100 kB/s. It is designed to run as an additional layer on top of a KBR-compliant DHT. Thus it

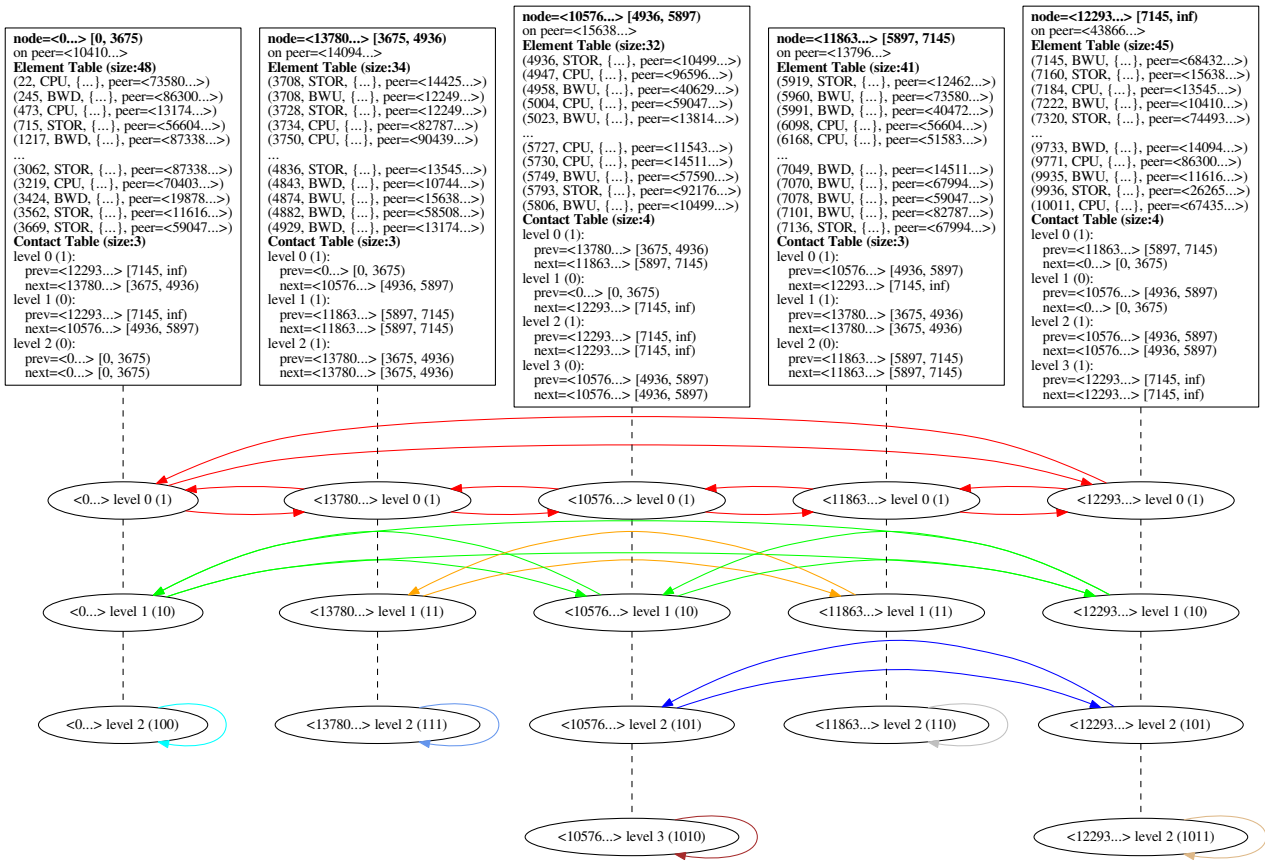


Fig. 1. An actual small PacketSkip graph. Each PacketSkip node is a data structure in the DHT consisting of a node ID, an element table containing the indexing information and a contact table that describes the SkipList neighborhood of the PacketSkip node.

is able to run on top of existing p2p solutions. PacketSkip provides a distributed, ordered, multidimensional indexing data structure, particularly for indexing individual peers by their capacity and for performing range queries.

In the following we present details of the indexing data structure, the protocols of graph maintenance, updates of index items and search queries. The section concludes with a discussion about PacketSkip.

A. Data Structure

In the following we detail all relevant components of PacketSkip as shown in Fig. 1. The figure shows the network topology as well as the stored indexing entries.

1) Nodes: A PacketSkip node is a data object that consists of a node ID and two tables. In fact a peer can potentially hold several PacketSkip nodes with different node IDs. **The node ID** identifies an element in the given DHT ID-space. Messages to a node are sent through a DHT lookup to the peer corresponding to the PacketSkip node ID. Only the peer responsible for the node ID maintains the PacketSkip node.

The first table, **the contact table** is a list of contact levels. The lower part of Fig. 1 visualizes the Skip Graph structure with its characterizing rings on several levels, connecting all nodes on level 0 and with each node connected to itself on the highest level. A contact level has three fields: a binary single digit for the current prefix symbol and two pointers to other nodes ($prev$ and $next$) on that level which share the same prefix. The level index results from the position in the table, since higher levels are always added at the end of the table. Pointers to neighbors are called *PacketSkip contacts* and comprises the node ID of the neighbor and the range it is responsible for. The start of the contacts' ranges is the relevant attribute when nodes pass updates and search queries to their neighbors. The end of the range is mainly memorized to verify a proper partition of the key-space.

The other table, **the element table** at top of Fig. 1, contains a collection of PacketSkip elements. Nodes are ordered and arranged within the limits of the table's range in the graph so that all nodes represent a complete, disjoint partition of the entire key-space. While the contact table provides routing functionality, the element table holds a chunk of the actual indexing data, i.e. the association of peer capacities to nodes. The table is confined by a range $[a, b[\ (a, b \in \mathbb{R} \geq 0)$ for which the node is responsible. The maximum size of the element table, the number of elements it contains, is limited by a predetermined constant value. Exceeding this threshold leads

to load balancing and may result in adding a new node to the graph, distributing the load on these two nodes and adjusting the table ranges. Falling below a minimum threshold of the table size may trigger a node to leave the graph.

PacketSkip makes great use of the common functionalities of the underlying overlay, in particular lookup, data storage and replication. Like CapSearch nodes, a node in PacketSkip is a logical node which is not identical with a peer.

2) Index elements: Objects that are to be indexed in PacketSkip need at least one key that can be sorted, i.e. a download bandwidth capacity of 245 (BWD). The index uses an unlimited decimal key space internally. PacketSkip distributes these index terms according to their keys on the PacketSkip nodes. A key is simply the value, e.g. 22 or 245, of a capacity feature, e.g. CPU or BWD, from a peer.

Peers have several capacity features, such as CPU capacity (CPU), bandwidth up and down (BWU and BWD) or free storage (STOR). All relevant capacity measures can be considered, as long as they fulfill the requirement of being a numerical value or string. Thus, indexing peers by their capacity implies multidimensional data with each feature being one dimension and each dimension having its own, not necessarily unique, key. The realization of multidimensional indexing and search in PacketSkip follows a simple yet effective approach. Since the key space of PacketSkip is one dimensional (e.g. 245), all dimensions are projected to this single domain. The keys are then supplemented with an identifier (a string) for their associated dimension, e.g. BWD for 245, forming a key-tuple pair, hereafter referred to as (PacketSkip) element. The other dimensions are retained as additional features for each element, therefore allowing for fast and precise multidimensional search queries. Taken together, PacketSkip stores one-dimensional elements in its nodes that are composed of a key (capacity value, e.g. 715), by which they are ordered, a dimension (capacity feature identifier, e.g. STOR) to which the key belongs to, additional features and a pointer to the object it represents (contact information of the peer, e.g. peer=<56604...>). They have the form $(\langle key\rangle,\ \langle dimension\rangle,\ \{\langle additionalfeatures\rangle\},\ \langle peer\text{-}contact\rangle)$. The primary key of this element consists of $\langle key\rangle$, $\langle dimension\rangle$ and $\langle peer\text{-}contact\rangle$.

Having discussed the construct of PacketSkip nodes, i.e. the element table and the contact table, next we discuss, how these nodes are linked and the required structure is maintained.

B. Graph Maintenance

PacketSkip dynamically grows and shrinks with the amount of data it maintains. Since the size of all element tables is limited, then for a given number of capacity dimensions, the size of PacketSkip evolves proportionally to the size of the overlay. In the following we describe the joining and leaving protocol of PacketSkip and how the connections between the nodes are established.

1) Initialization and Bootstrapping: The first node of PacketSkip is chosen via a network-wide known default overlay ID. The PacketSkip service of the overlay peer responsible for this ID will initialize the first node. In a scenario with churn, when responsibilities are shifting, before establishing this first PacketSkip node we first verify that no other PacketSkip node is currently alive. Since not every participating peer of the network maintains a PacketSkip node, a bootstrap mechanism is needed to find such a potentially existing PacketSkip node. After the initialization, a node announces its existence by storing its contact information according to the chosen bootstrap mechanism. Other peers can then read from this location the information of how to contact this node for bootstrapping, index updates and searches.

a) Bootstrapping: In order to establish an initial PacketSkip node, we start with the premise that a monitoring service, such as SkyEye.KOM [8], is running along with PacketSkip and aggregating its size. An array in the DHT with the size of the current number of PacketSkip nodes n is then allocated. The addresses of the array fields derive from $hash(\text{'entry'}{+}i)\ i \in \{0,\ldots,n-1\}$. PacketSkip nodes write their IDs in cyclic intervals (with some added jitter) in a randomly chosen field of this array. Peers must read a node ID from a random field before sending update or search query messages to an entry node of PacketSkip. If the array entry was outdated, the message to the node ID is answered with a negative acknowledgement (NACK) by the responsible overlay peer and a lookup for another entry ID is performed. The probabilistic and periodic nature of this mechanism guarantees even load while the content of the bootstrap array stays up to date and requests usually return valid entries.

2) Load Balancing: Once the element table of a node reaches its limit the node will attempt to distribute some of its elements to its immediate neighbors. For this load balancing, a node asks its predecessor and successor how many elements they will accept for their own table. The requesting node will then split its element table according to the information given by its neighbors and submits the chunks to them. The load balancing aims to even the load of the neighbors and leaves a headroom for all partaking nodes. By this, none of the participants has to initialize a subsequent load balancing operation. As a result of sharing elements, the ranges of the element tables of all involved nodes are adjusted. This adjustment is communicated to all the contacts of the involved nodes which then update their contact tables accordingly.

3) Joining: In case the load balancing process fails, because all neighbors are already close to their limits, the initiator (*nodeA*) of the load balancing attempt will prepare a join operation that adds another node to PacketSkip. *nodeA* generates a random node ID and asks the peer responsible for this overlay ID to initialize a new PacketSkip node (*nodeB*). *nodeB* lines in as next contact to *nodeA* and is endowed with the upper half of its element table. On level 0, *nodeB* registers *nodeA* as predecessor and *nodeA*'s former successor as its own successor. *nodeA* communicates its range adjustment to all of its adjacent contacts. Meanwhile, *nodeB* performs a handshake with its successor, effectively joining level 0.

Then a sequence of recursive operations to join the higher levels is conducted. For each of these consecutive operations,

nodeB adds a level with index $i > 0$ to its contact table. It randomly picks a level prefix symbol $s_i \in \{0, 1\}$ concatenating it to its prefix on level $i - 1$. The prefix for level $i > 0$ is $p_i = p_{i-1}s_i$, for $i = 0$ it is $p_0 = 1$. For the time being, the $prev$ and $next$ pointers on level i will point to *nodeB* itself (self linked). The node then sends a request to join level i on list p_i to the next contact on level $i - 1$. When another node (*nodeC*) receives such a request, the following cases must be considered:

1) *nodeC* has already joined level i
   a) $p_i(nodeC) = p_i(nodeB) \Rightarrow$ *nodeC* accepts the request and sends its predecessor (*nodeD*) to *nodeB*. *nodeB* and *nodeD* will then perform a handshake and *nodeB* has effectively joined level i on list p_i.
   b) $p_i(nodeC) \neq p_i(nodeB) \Rightarrow$ *nodeC* forwards the request to its successor on level $i - 1$ besides notifying *nodeB* of the forwarding.
2) *nodeC* has not yet joined level i. *nodeC* will add level i (self linked) picking a random prefix symbol $s_i \in \{0, 1\}$.
   a) analogous to 1a. In this case *nodeD* is equal to *nodeC*.
   b) analogous to 1b.

The recursion terminates once *nodeB* has added a level for which it finds no contact that accepts the join request. As a result the level with the highest index in a local contact table is always self linked. This then completes the joining process of *nodeB* in PacketSkip. Note that during the joining process the added node may prompt its contacts to add new levels. Adding a new node will eventually force its neighborhood to adapt their contact table sizes if needed, but in a logarithmic fashion, while most nodes of PacketSkip are not affected.

4) Leaving: Beside the maximum limit of the element table there is also a minimum threshold. If the size falls below this minimum, the node is prompted to leave PacketSkip. First, it performs a load balancing operation, distributing its elements to its predecessor and successor. In contrast to a regular load balancing, the node tries to assign all of its remaining elements to its neighbors. In case this 'abandoning' load balancing fails, the node must not leave. If successful, the node will inform its contacts on all levels about the leave, thereby introducing its predecessors to the respective successors and vice versa. Once the informed nodes have updated their contact tables and no pointer points to the leaving node anymore, the node has effectively left PacketSkip. Finally, the responsible peer will remove the PacketSkip node from the DHT.

After a leave, all involved PacketSkip nodes will clean up their contact tables by contracting redundant self linked levels which may have emerged when a leaving node was the only contact of the remaining node on affected levels. Just like a join may prompt other nodes to add levels, a leave may cause the removal of levels.

C. Capacity Updates

Having presented the implications of joining and leaving PacketSkip nodes, next we present the protocol details for the regularly happening capacity changes. Eventually the node capacity index of PacketSkip must reflect the dynamics of capacity changes of the corresponding peers.

Every peer in the overlay announces updates of its capacity values to PacketSkip. It is not necessary that the peer holds a PacketSkip node itself. An index update contains one set of elements for inserts and one for deletions. In case the capacity values have not changed between two updates, the delete set is empty. Inputs and deletions are performed in the same way in parallel and update messages carry both sets.

To publish an update, the PacketSkip service identifies an entry node to PacketSkip via the bootstrapping mechanism. Subsequently, the service will pass all elements, both inserts and deletions, in one package to the entry node. The node will then greedily forward the elements to its contacts according to their keys. Elements with a key inside the range of the local element table are directly inserted or deleted, depending on the set they were included. Since a PacketSkip node has no global view on the entire graph, but only on its contacts, it divides the key-space by the lower bounds of its contacts. The contact with the lowest range is considered also responsible for keys below its actual range. Eventually, the process terminates when all elements are finally stored in a local table. Since the distance to the responsible node reduces with every hop, halving on average, and the update packages are forwarded in parallel, index updates are performed in logarithmic time.

Updates effectively form a temporary search tree in PacketSkip by all nodes involved. The randomly chosen entry node is the root and the forwarding nodes are inner nodes of the tree. While forwarding, an inner node may still store or remove elements in its local table if it is responsible for them. Leaf nodes either insert or delete elements but do not forward.

In a scenario with churn we do not expect peers to promote deletions when they leave the overlay. Therefore PacketSkip elements are equipped with a timestamp and are purged from the element tables periodically. For that reason peers have to announce their capacities periodically.

D. Range Search Queries

Next, we present the main function of PacketSkip, namely the capacity-based search. PacketSkip offers two types of searches, *the full search* for obtaining all peers that meet a given criteria and *the k search* for limiting the search to a given k value. A sample query could e.g. search for peers with a storage space of 120-150 MB, a upload bandwidth of 25-30 MBit/s and free memory of 50-60 MB. A full search is expected to result in the list of all matching peers in the network. In a k search, the resulting list is sufficient if it contains k matches, although there might be more matching peers in the network. During a k search, k is adapted to each hop in PacketSkip by subtracting the number of found matches. A search will not be forwarded if $k < 1$. Since a search is passed along the search tree in parallel, PacketSkip may actually deliver more than k search results.

Along with the results, PacketSkip applies tracer routing, i.e. nodes will also inform the requester from which node they received a query and to which other nodes they passed

it to. Thereby the requester actually knows when a search has terminated, that is, when the set of nodes which have sent a result message and the set of nodes to expect a result message from are equal. This is especially important for full searches.

Queries for capacity ranges are handled quite similar to updates. They are passed to a randomly chosen entry node as a package. Nodes will recursively split the query according to the responsibilities of their contacts into smaller interval chunks before they distribute these chunks. The ranges of the search query get smaller with every hop. The search terminates once all nodes responsible for the initial search range are consulted or k search results have been found. Because queries are split and then sent in parallel and because of the average quadratic growth in distance of the contact links, a search can also be performed in logarithmic time.

A multidimensional search focuses on the dimension with the shortest range. While distributing the query the capacity feature identifier, e.g. STOR, is ignored. The search operates on values. The capacity feature identifier is only considered on the responsible node for a searched value and fitting search results are looked up from the local table. For each hit, the additional features of the element are reviewed against the additional dimensions in the query and rejected or accepted accordingly. For matches in all dimensions, the element values, that is, the peer-contacts, are returned to the seeker.

E. Discussion

As PacketSkip relies on a replication mechanism which runs in parallel to PacketSkip, a reactive maintenance mechanism is sufficient. Thus, we assume that PacketSkip nodes do not fail due to churn, whereas peers do.

From the view point of PacketSkip, the update and search performance is in $\mathcal{O}(\log n)$ with n being the number of nodes. As PacketSkip's communication uses lookups in the underlying DHT, their traffic adds to the overall performance. Since most DHTs perform lookups in logarithmic time, from the view point of such an overlay, the search and update performance of PacketSkip is actually in $\mathcal{O}(\log^2 n)$. Desirable $\mathcal{O}(\log n)$ performance could be achieved by temporarily caching information of responsible peers in the contact table and so avoiding repetitive lookups. This could sometimes lead to misses under churn, but would reduce traffic costs.

IV. Evaluation

We evaluate PacketSkip in comparison to CapSearch [5] in various scenarios using the p2p system simulator Peerfact-Sim.KOM [7]. Following is a discussion of most important metrics, the methodology and the outcome of the evaluation.

Metrics

The questions in the evaluation address the following points:

1. The precision of search results is essential for the quality of search protocol. Precision is a ratio used to measure if the search results cover the actual available capacities in the overlay. We also measure the presence of false positives, which are an indication of outdated search results relative to the size of the result set. The aim is to obtain a precision as close as possible to 1 while keeping the false positives close to 0.

2. The performance and efficiency in terms of update and search behavior determines how long a user has to wait for each operation. As search requests should be resolved quickly, we measure the number of hops and time taken per operation.

3. Overhead in terms of messages and traffic for the overlay and load on individual peers. We review the traffic and number of messages corresponding to maintenance, update and search operations throughout PacketSkip as main cost metric. It is expected that the maintenance should not add significantly to the amount of messages. In addition, we measure the number of objects stored in the DHT and the load distribution, as well as the fluctuation of nodes (joining and leaving the graph) and its effect on the lifespan of the PacketSkip nodes.

TABLE I
Simulation Parameters

Parameter	Values
Simulation time	330 minutes
Number of seeds	10
Number of peers (seekers)	1000 (240)
Capacity dimensions	3,5
Full-/k-search	k=8, full
Element table size max / min / headroom	default: 50 / 20% of max / 10% of max other max sizes: 12, 25, 50, 100 and 200
Capacity change interval	all 30, 10 minutes
Search interval	2 searches per minute in total
P2P overlay	Chord
Network dynamics	Smooth capacity change
Overlay churn	Inactive, Active (from minute 161)
Network layer model	Simple

TABLE II
Scenario for external comparison

Period of time (in minutes)	Action performed
0 – 90	Overlay join
91 – 120	Do nothing (stabilize overlay)
121 – 130	Start the PacketSkip service
131 – 190	Start periodic capacity changes
200 – 320	Perform full- or k-searches
330	End of simulation

Methodology

The evaluation methodology uses the same parameters and scenarios of evaluation as CapSearch [5] in order to conduct a precise and detailed comparison. Parameter variations and performed operations from the simulations are given in Tables I and II respectively. For each simulation 10 different seeds are used and the discussed outcome is the average and standard deviation from these values.

The simulations lasted for 330 minutes preluded with a peer joining phase during the first 90 minutes. The Chord overlay is set to have 1000 peers and settles for the following 30 minutes. The PacketSkip service is initialized on the peers in the next 10 minutes (between 120–130 minutes). The local capacity announcement is started between 131–190 minutes so the PacketSkip graph initializes during that time. Peers will update and announce their capacities every 30 minutes using the smooth capacity change behavior, which updates capacities with respect to their former values with a normal distribution.

TABLE III
COMPARISON BETWEEN PACKETSKIP AND CAPSEARCH FOR SCENARIOS A, B, C AND D.

Scenario [f]ull- or [k]-search		Metric: Precision and False Positives						Metric: Search Performance						Metric: Capacity Update						Metric: Traffic per peer [byte/s]			
		precision [%]			false positive [%]			hops			duration [s]			messages			duration [s]			maintenance		search	
		Ø	$\sigma_{<Ø}$	$\sigma_{>Ø}$	Ø	$\sigma_{<Ø}$	$\sigma_{>Ø}$	Ø	$\sigma_{<Ø}$	$\sigma_{>Ø}$	Ø	$\sigma_{<Ø}$	$\sigma_{>Ø}$	Ø	$\sigma_{<Ø}$	$\sigma_{>Ø}$	Ø	$\sigma_{<Ø}$	$\sigma_{>Ø}$	-in	-out	-in	-out
PacketSkip	A_f	100.00	0.00	0.00	0.00	0.00	0.00	22.78	-7.61	0.00	0.48	-0.07	0.00	11.32	-2.93	3.33	0.43	-0.11	0.13	12.90	12.90	0.29	0.29
	A_k	100.00	0.00	0.00	0.00	0.00	0.00	4.94	-1.55	0.00	0.21	-0.06	0.00	11.35	-2.94	3.39	0.42	-0.11	0.13	12.92	12.92	0.11	0.11
	B_f	99.77	-0.39	0.17	1.68	-1.09	0.01	19.07	-6.46	0.12	2.62	-2.04	0.01	10.40	-2.69	3.01	1.97	-1.60	19.72	12.18	12.20	0.33	0.33
	B_k	98.49	-1.46	0.11	1.78	-1.56	0.00	5.16	-1.61	0.02	0.88	-0.70	0.05	10.65	-2.80	3.22	1.98	-1.60	19.78	12.44	12.45	0.15	0.15
	C_f	99.95	-0.08	0.03	1.60	-1.19	0.00	20.44	-6.64	0.06	3.02	-2.28	0.05	14.41	-3.32	3.74	2.41	-1.96	19.02	24.68	24.71	0.33	0.33
	C_k	98.75	-1.14	0.07	1.70	-1.51	0.00	8.35	-2.34	0.02	1.22	-0.90	0.02	14.41	-3.31	3.73	2.45	-2.00	19.14	24.72	24.75	0.23	0.23
	D_f	99.77	-0.39	0.18	1.63	-0.97	0.01	19.15	-6.23	0.11	2.96	-2.29	0.07	10.86	-2.80	3.16	1.98	-1.59	19.04	15.64	15.66	0.33	0.33
	D_k	98.62	-1.21	0.04	1.67	-1.39	0.00	5.27	-1.51	0.01	0.85	-0.67	0.00	10.93	-2.88	3.33	1.91	-1.52	18.58	15.71	15.73	0.15	0.15
CapSearch	A_f	99.91	-0.08	0.00	0.01	-0.01	0.00	128.57	-72.07	0.00	0.31	-0.05	0.00	56.00	0.00	0.00	1.71	-0.06	0.07	4.28	4.28	0.60	0.60
	A_k	99.99	-0.01	0.00	0.00	0.00	0.00	15.76	-0.60	0.00	0.14	-0.01	0.00	56.00	0.00	0.00	1.71	-0.06	0.07	4.29	4.29	0.07	0.07
	B_f	63.70	-5.81	0.23	0.47	-0.33	0.01	92.95	-50.09	1.98	6.76	-4.62	0.24	53.59	-21.22	12.10	6.85	-5.04	16.53	4.32	4.32	0.61	0.61
	B_k	94.33	-5.00	0.02	0.41	-0.39	0.00	16.10	-1.02	0.01	1.05	-0.82	0.00	53.57	-21.04	12.00	6.60	-4.78	16.10	4.32	4.32	0.09	0.09
	C_f	69.29	-11.70	0.12	0.28	-0.23	0.00	23.41	-14.90	0.10	2.42	-1.93	0.01	90.56	-32.47	24.61	10.26	-7.10	17.79	5.94	5.94	0.16	0.16
	C_k	86.93	-10.13	0.02	0.35	-0.32	0.00	14.45	-1.69	0.00	1.28	-1.03	0.00	90.57	-32.40	24.33	10.35	-7.18	16.89	5.97	5.97	0.10	0.10
	D_f	58.09	-6.02	0.21	0.71	-0.49	0.01	98.02	-54.45	1.47	5.75	-4.20	0.17	56.86	-15.65	14.52	5.83	-4.05	15.77	7.28	7.28	0.63	0.63
	D_k	94.11	-4.85	0.07	0.71	-0.66	0.00	16.85	-1.11	0.01	1.14	-0.92	0.00	56.97	-15.64	15.58	6.10	-4.29	16.16	7.24	7.24	0.10	0.10

TABLE IV
DHT LOAD PER PEER FOR SCENARIOS A, B, C, D AND E.

Scenario		Metric: DHT: number of objects per peer				Metric: DHT: size of objects per peer [kByte]		
		share > 1 [%]	Ø	$\sigma_{Ø}$	max	Ø	$\sigma_{Ø}$	max
PacketSkip	A	13.55	0.16	0.43	5	0.4	1.4	14.7
	B	11.19	0.13	0.38	4	0.3	1.2	10.0
	C	14.56	0.17	0.44	4	0.6	2.1	21.1
	D	11.09	0.13	0.39	4	0.3	1.3	14.9
	E_{12}	45.46	0.83	1.23	10	0.5	1.0	7.8
	E_{25}	24.67	0.33	0.66	6	0.4	1.1	10.4
	E_{50}	13.63	0.16	0.43	5	0.4	1.4	14.6
	E_{100}	6.97	0.08	0.30	3	0.4	1.9	28.7
	E_{200}	3.65	0.04	0.19	2	0.3	2.6	38.3
CapSearch	A	99.26	95.87	94.64	616	33.8	33.3	216.8
	B	98.5	71.40	71.78	544	25.1	25.3	191.6
	C	99.42	153.81	157.08	1137	54.5	55.7	403.2
	D	99.22	147.95	156.84	1336	52.1	55.2	470.7

Searches are performed from minute 200 to 320. In total 240 searches are started (2 searches per minute).

In the default scenario A, churn is disabled and therefore used for proof-of-work, whereas scenario B examines the influence of churn. In order to stress the maintenance and search, the number of dimensions of the capacities is raised from 3 to 5 in scenario C and the capacity change interval is lowered to 10 from 30 minutes in scenario D. Finally, scenario E presents the load balancing in PacketSkip by varying element table sizes.

PacketSkip uses a maximum element table size of 50 (minimum size 10, headroom $10\% = 5$). The headroom is the percentage of the maximum size a load balancing operation subtracts to not instantly max out the participating nodes as a result of the load balancing. Without the headroom, a load balancing could result in a cascade or even an infinite loop of successive load balancing operations. All simulations use the simple net layer of PeerfactSim.KOM, an idealized network offering a small constant latency. Analogous to CapSearch, for the k-searches, the k is set to 8.

In the following, we discuss the simulation results in Tables III and IV with regard to the search performance, capacity update, load distribution and traffic costs in general. For each category, the result of the following scenarios is discussed and compared with CapSearch.

A. Results on the Precision and False Positives Rate

We refer for this discussion to Table III, where we list the precision and false positives ratio. While in scenario A, the precision and false positive rates from PacketSkip and CapSearch are quite similar, the quality difference is strong for the scenarios B to D. The full search requires all peers with matching capacities to be found. In scenario A, PacketSkip finds 100% and CapSearch 99% of the fitting peers. PacketSkip maintains this high quality of the search results with a precision of $\geq 99.77\%$s under churn in scenario B, more capacity features to consider in scenario C as well as with a lowered capacity update interval in scenario D. For the k search the precision is also near optimal, with a minimum precision of 98.49% under strong churn, mainly due to the higher impact if single failing nodes in the results. The probability of false positives is with 1.78% in maximum also very low. Thus, PacketSkip resolves the searches with near optimal precision.

CapSearch on the other hand is only competitive in scenario A without churn. In the scenarios with churn, with more capacity features as well as in the scenario D with lower capacity update intervals, the precision drops to a low of 58.09% to 69.29%. This means, that in worst case 41.91% of all suitable peers are not found. While the ratio of false positives is only 0.71% in maximum, this does not compensate for the very low precision.

B. Results of the Search Performance

In scenario A, PacketSkip's search queries are answered after about 210ms for k search and after 480ms for full search. In comparison to CapSearch, which finishes searches after 140ms and 310ms for k and full search respectively, PacketSkip performs searches slightly slower as parallelization is limited and a bootstrapping method is needed. Nevertheless,

the number of hops per search is significantly lower. PacketSkip needs 3.4 to 4.9 and 15.2 to 22.8 hops for k and full search, respectively. CapSearch search operations need 15.8 and 56.5-128.6 hops for k and full search, respectively. The difference is because PacketSkip nodes store more than one entry in their view, thus a significant less amount of nodes needs to be traversed.

In scenario B of PacketSkip and the k search, the upper standard deviation of the duration is 1s and an average duration of 880ms are observed, while the full search showed upper standard deviations of 2.63s with an average of 2.62s. The comparison to CapSearch shows that k search almost takes the same time (avg: 1.05s) while full search takes longer than the 2.5 fold (avg: 6.76s). This is most probably due to a reasonably higher number of hops that the search needs.

In contrast to scenario B, in scenario C of PacketSkip the search duration is increased by 400ms for both k and full search. The number of hops for the k search scenario increased from 5.2 up to 8.4 hops due to increased number of total data sets, thus potentially more nodes have to be visited for the desired number of hits.

Coming from scenario B in scenario D, neither PacketSkip's nor CapSearch's duration or hops are affected significantly.

It is noteworthy that for PacketSkip the number of hops is quite stable, requiring 19.07 – 22.78 hops for a full search and only 4.94 – 8.35 hops for k search. CapSearch is also stable for k search with 14.45 – 16.85 hops, but show strong variance for the full search with 23.41 to 128.57 hops.

C. Results of the Capacity Update Performance

The update operation is initiated by peers when announcing or refreshing their current capacities. It is relevant for a high precision, that these changes are quickly propagated.

For scenario A a duration of 430ms in average with a standard deviation of +/- 120ms with an average of 11.3 messages is observed. PacketSkip provides a faster update mechanism than CapSearch, which needs 1.7 seconds in average with a standard deviation of +/- 60ms sending 56 messages. This is one of the advantages of PacketSkip, mainly due to a mapping of multidimensional data onto a one dimensional space.

For scenario B and PacketSkip, the update duration is prolonged, hitting 2s in average with an upper standard deviation bound of 22s. In comparison to CapSearch (average 6.7s, plus standard deviation of 23s) PacketSkip can finish its update operation three times faster in average, which has a positive impact on the search precision.

In scenario C and PacketSkip it was observed that on average there was a worsening by 500ms, whereas CapSearch suffers under an increase of roughly 3.6s. For the number of sent messages, PacketSkip sends four more messages on average compared to scenario B due to the increased dimensions. For CapSearch, an increase of the number of capacities leads to a deeper kd-tree, thus the number of messages is increased by a constant (21). The more frequent capacity updates of scenario D do not significantly influence the number of messages sent and duration for PacketSkip and CapSearch.

Updates are propagated in PacketSkip within 0.43 – 2.45 seconds, with 2.45 second being required for the increased capacity feature set in scenario C. CapSearch requires longer with update durations of 1.71 – 10.35 seconds. PacketSkip is both significantly faster by a factor 3-4 than CapSearch in its update process as well as as requiring 5-6 less messages for this. The confidence intervals are smaller and closer to the average. Thus, through quick update in PacketSkip, searches are more precise and less false positives are selected.

D. Metric: Load Distribution

For the load balancing, we refer to Table IV, which exhibits that with PacketSkip only 11.09 % – 14.56% of all peers store at least one DHT object and 5 objects at maximum. One half are bootstrap entries and the other half are actual PacketSkip nodes (no figure). The storage requirement on a peer in PacketSkip is up to 0.6 kB in average or 21.1 kB in maximum for the scenarios A to D.

In CapSearch, on the other hand, almost every peer stores at least one object (98.5% – 99.42%) and utilizes more objects (maximum 1336 objects per peer) with 470.7 kB per peer maximally. Thus, CapSearch creates 600 times more objects than PacketSkip and storage requirements of 25.1 – 54.5 kB in average and 470.7 kB in maximum.

In scenario B, the share of nodes in PacketSkip holding at least one object is decreasing as the number of peers participating in the DHT also decreases, thus, less PacketSkip nodes need to be created. The latter is reflected by the decreasing average number of objects per peer.

In scenario C we increase the number of capacity features, resulting in an increased number of objects in the DHT using PacketSkip. All 1000 peers inject two additional values more, one additional value per dimension per peer. Roughly 14.6% of all participating peers hold at least one DHT object and 9% of all participating peers hold a PacketSkip node (no figure). Compared to scenario B, a slight increase of 0.04 objects per peer is observed, while the average size of DHT objects raises from 0.3 to 0.6 kB in average. In contrast, CapSearch perceives a doubling of DHT objects and their space requirement.

Further, scenario D does not have a significant influence in PacketSkip. On the contrary, CapSearch reacts on the more frequent capacity updates just like on the increase of the capacity spaces by doubling the number of objects and space.

We conclude for scenarios A to D that only few PacketSkip nodes are required, much less than peers available. PacketSkip scales better than CapSearch when it comes to a rising number of capacity spaces or more frequent capacity updates.

E. Metric: Load Distribution with various Element Table Sizes

As the element table size has a severe implication on the load distribution, we conduct a parameter study with regard to this interdependency. Figure 2 shows the load distribution among the peers for the total network traffic. Note, that E_{50} equals scenario A. From the log-log scale, the total aggregated traffic is seen to decrease with increasing table size, due to less lookups in the underlying overlay, less maintenance

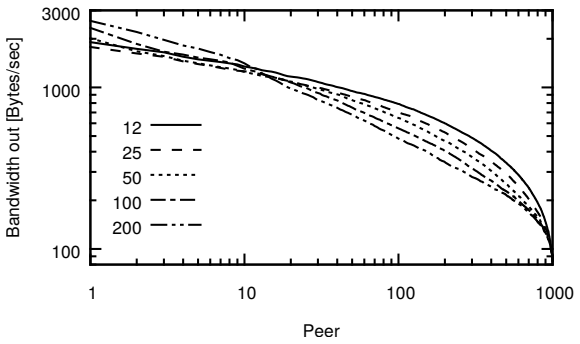


Fig. 2. Scenario E: Scenario A with various element table sizes. Aggregated traffic per peer with varying element table sizes.

for the inter PacketSkip node communication as well as less PacketSkip nodes, resulting in worse load distribution throughout all participating peers. For the 10 peers with the highest load, the aggregated traffic increases, which is true for all scenarios except the scenario with the maximum table size of 12. This scenario stresses the PacketSkip join, leave and maintenance mechanisms, which is less efficient than a scenario with a maximum table size of 25.

Therefore, we identify a tradeoff between PacketSkip's node fluctuation and the induced inter node communication as well as load distribution throughout all participating peers. In conclusion, it can be seen that, a good choice for the element table size depends on the use case. For frequent searches and stable scenarios, smaller table sizes are recommended, as it leads to a more even distribution of the PacketSkip nodes. For example, in scenario E_{12} 45.5% of all peers store at least one node, which is almost halved in scenario E_{25} with only 24.7%. On the other hand, in rare searches and dynamic scenarios, bigger table sizes can improve the overall efficiency.

F. Metric: Maintenance and search costs

For this evaluation, we refer to the last columns of Table III as well as Figures 3 for scenario A without churn and 4 for scenario B with churn.

In scenario A, i.e. without churn, Figure 3a shows that about 8% (= 80) of all peers in PacketSkip are involved in the maintenance of PacketSkip's graph. Therefore, these peers face more traffic in the magnitude of two throughout the whole simulation time (from $20 * 10^3$ to $20 * 10^5$ bytes). Roughly $\frac{1}{3}$ of all peers are involved in the search, which composes out of 240 seeker peers plus PacketSkip nodes minus the intersection of them. The high search in traffic is from incoming search hits. In average, a peer has to deal with 12.9 byte/s for maintenance of both in- and outbound traffic. Figure 3c shows the aggregated traffic of PacketSkip in byte/s, where the peer with the highest load has to deal with 2000 byte/s in average throughout the simulation, which equates to 16 kBit/s which is a comparatively low load. In fact, Figure 3d shows that the course of CapSearch's aggregated traffic is very similar to PacketSkip's traffic. Figure 3b shows that almost all peers are involved in the maintenance of the CapSearch structure and 85% and 70% of all peers are traversed during the full and k search, respectively. Thus, the amount of traffic is comparable but the load distribution in CapSearch is more even.

In scenario B, i.e. with churn, Figure 4a reveals the traffic per peer which is similar to scenario A, except that a few peers encounter less maintenance traffic due to churn events. The same argument applies to the traffic per peer of CapSearch which is shown by Figure 4b. Further, in comparison to scenario A, almost 10% less peers are contacted in k search, thus, the involved peers observe a small increase in traffic.

In Table III, we observe that PacketSkip requires 12.18 – 15.71 byte/s for maintaining 3 capacity features and 24.7 byte/s for maintaining 5 capacity features. While CapSearch suffices with 4.28 – 7.28 byte/s and is thus significantly more efficient, the workload is still very small. A maximum of 25 byte per second for maintaining the index structure as well as up to 0.63 byte/s for search traffic in average is very small.

V. Conclusion and Future Work

With PacketSkip we proposed an ordered indexing and search structure maintaining highly dynamic multidimensional data running on top of structured p2p overlays. It offers a nearly perfect precision. PacketSkip can be widely utilized beyond the restricted use of capacity indexing as the service offers low update and maintenance costs and its inner graph structure is robust and stable. Also, update and search durations are observably low and well scalable and the search performance particularly outperforms other state-of-the-art approaches such as CapSearch.

For future work, bootstrapping, search and updating mechanisms need improvement possibly by locally caching responsible peers for PacketSkip nodes, so that it is possible to achieve $\mathcal{O}(\log n)$ performance. On one hand, an approach for better load distribution may be realized by building a PacketSkip graph per dimension. On the other hand, a more dynamic element table size adaptation could automatically find a sweet spot between load distribution and node fluctuation induced by intra node communication.

Acknowledgment

Computational support and infrastructure was provided by the Centre for Information and Media Technology (ZIM) at the University of Düsseldorf (Germany).

References

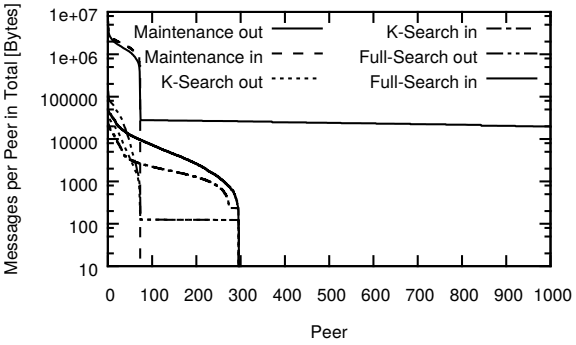
[1] J. Aspnes and G. Shah, "Skip Graphs," *ACM Trans. Algorithms*, vol. 3, no. 4, Nov. 2007. [Online]. Available: http://doi.acm.org/10.1145/1290672.1290674

[2] Carlini, Emanuele and Lulli, Alessandro and Ricci, Laura, "DRAGON: Multidimensional range queries on distributed aggregation trees," *Future Generation Computer Systems*, vol. 55, pp. 101–115, 2016.

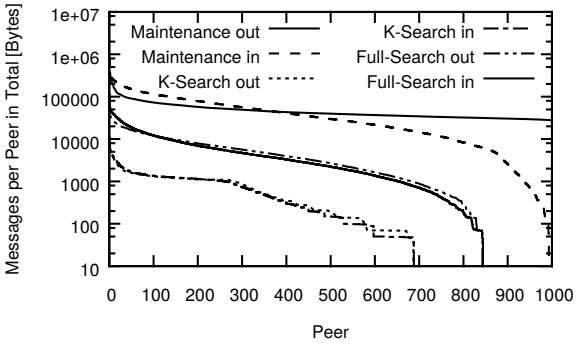
[3] F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Proc. of the Int. Workshop on Peer-To-Peer Systems*, ser. Lecture Notes in Computer Science, vol. 2735. Springer, 2003, pp. 33–44.

[4] V. Darlagiannis, "Overlay Network Mechanisms for Peer-to-Peer Systems," Ph.D. dissertation, Technische Universität Darmstadt, Germany, 2005. [Online]. Available: http://elib.tu-darmstadt.de/diss/000699

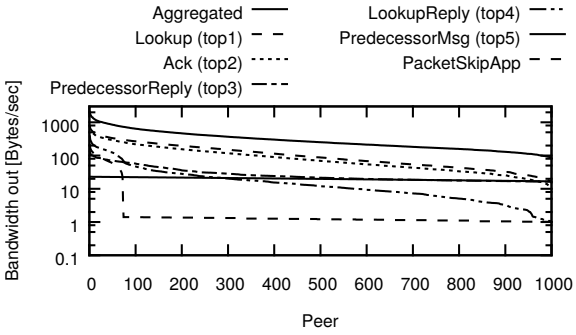
[5] A. Disterhöft and K. Graffi, "CapSearch: Capacity-Based Search in Highly Dynamic Peer-to-Peer Networks," in *AINA 17: Proc. of the 31st Int. Conference on Advanced Information Networking and Applications*. IEEE, 2017, pp. 621–630.



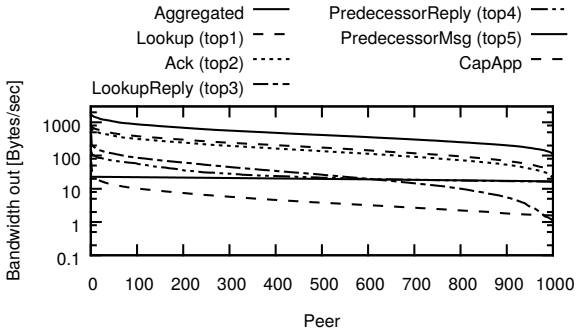
(a) PacketSkip: Traffic per peer



(b) CapSearch: Traffic per peer

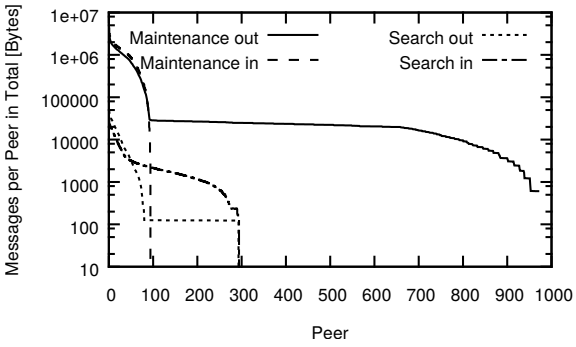


(c) PacketSkip: Categorized outgoing bandwidth

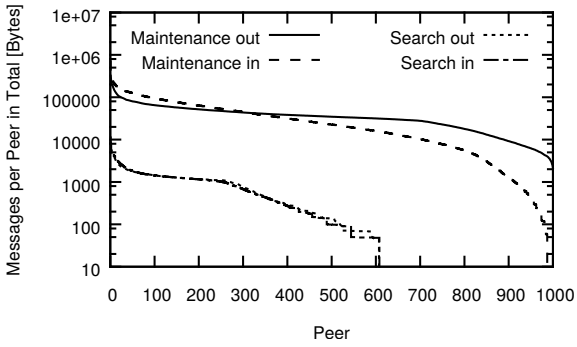


(d) CapSearch: Categorized outgoing bandwidth

Fig. 3. Per peer and categorized traffic comparison of Scenario A: 30m capChanger, without churn, 3 capacity spaces



(a) PacketSkip: Traffic per peer



(b) CapSearch: Traffic per peer

Fig. 4. Per peer traffic comparison of Scenario B: 30m capChanger, with churn, 3 capacity spaces, k-Search scenario

[6] K. Graffi, "Monitoring and Management of Peer-to-Peer Systems," Ph.D. dissertation, Technische Universität Darmstadt, Germany, 2010. [Online]. Available: tuprints.ulb.tu-darmstadt.de/2248/

[7] ——, "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.

[8] K. Graffi, A. Kovacevic, S. Xiao, and R. Steinmetz, "SkyEye.KOM: An Information Management Over-Overlay for Getting the Oracle View on Structured P2P Systems," in *Proc. of the Int. Conf. on Parallel and Distributed Systems*. IEEE, 2008.

[9] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, "Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services," *IEEE transactions on Services Computing*, vol. 3, no. 3, pp. 223–235, 2010.

[10] S. Kargar and L. Mohammad-Khanli, "Fractal: An advanced multidimensional range query lookup protocol on nested rings for distributed systems ," *Journal of Network and Computer Applications*, vol. 87, pp. 147 – 168, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804517301303

[11] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat, "Service Placement in a Shared Wide-Area Platform." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 273–288.

[12] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 188–201.

[13] 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 Int. Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2001, pp. 149–160.

[14] P. Wette and K. Graffi, "Adding Capacity-Aware Storage Indirection to Homogeneous Distributed Hash Tables," in *NetSys '13: Proc. of the Networked Systems Annual Technical Conference*, 2013.

[15] M. Xu, S. Zhou, and J. Guan, "A new and effective hierarchical overlay structure for Peer-to-Peer networks," *Computer Communications*, vol. 34, no. 7, pp. 862–874, 2011.