CapSearch: Capacity-based Search in highly dynamic Peer-to-Peer Networks

Andreas Disterhöft
Technology of Social Networks, University of Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany
Email: disterhoeft@cs.uni-duesseldorf.de

Kalman Graffi
Technology of Social Networks, University of Düsseldorf
Universitätsstr. 1, 40225 Düsseldorf, Germany
Email: graffi@hhu.de

***Abstract*—In times of greatly heterogeneous devices, e.g. smart phones vs. desktop PCs, and their users participating in various types of networks new scenarios appear. These scenarios include peer-to-peer networks participants providing their capacities in order to operate this system. In case of weaker participants, which temporarily do not have enough capacities to contribute their share to the network, get overloaded. In a different case, participants may delegate a certain job to another participant, which is capable for this job. In this paper we propose a fast and reliable indexing and search mechanism for participants' capacities. We focus on a churn-resilient structure using a kd-tree with logical nodes in order to be capable of highly dynamic node capacities. Evaluation shows our approach is able to find nearly 60% of all nodes and over 80% of a certain number k of all nodes in big range queries, while keeping the false positive rate close to zero.**

I. Introduction

Today's participants of miscellaneous networked applications are highly heterogeneous in terms of their capabilities. For example, participants of online social networks use their smart phones, tablets, laptops or desktop PC's to access their applications. Apparently, the capacities of the mentioned devices vastly differ from each other, i.e. portable devices often have bandwidth limitations while using the mobile radio network and desktop PC's usually have more powerful CPUs installed than smart phones. Furthermore, capacities may change over time, e.g. a portable device connects to a wifi hotspot whereby its bandwidth capabilities change. The term capacity is used here as a tuple of values, representing each a different capacity measure, such as CPU cycles available, memory, storage space, available up- and download bandwidth, and so on.

One may imagine scenarios where it is particularly important to be able to search for a certain number of participants with a minimal amount of capacities in a decentralized network. The addition of being in a decentralized network is important as in centralized approaches the problem statement is trivial. One of these scenarios is the need to temporarily promote supporting nodes in a peer-to-peer network if a certain weak node is overloaded, e.g. due to providing a popular document. Another example is the delegation of an arbitrary job with specific capacity requirements in cluster environments [6] or to prioritize messages based on their relevance for the Network [8]. Thus an approach is needed that allows to identify within a large-scale peer-to-peer network both a specific number of nodes with certain capacities (or more) as well as the option to search all nodes with a minimum level of capacities.

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

In this paper, we propose a novel approach to index and search for node capacities in a structured peer-to-peer system. We provide a large-scaling indexing approach with a search mechanism maximizing the precision and minimizing the false positives. At last, the traffic caused by our solution's structure and search techniques must be kept to a minimum.

In this paper, we provide following contributions:

- We present CapSearch, an indexing structure and search protocol that supports:
  - K-search: Identifying k nodes in the network which meet the given required capacity level.
  - Full-search: Identifying all nodes in the network that meet the given required capacity level.
- The indexing structure as well as corresponding search protocol are described in details.
- Evaluation presents a sophisticated simulation setup and corresponding results that point out the strengths of CapSearchin terms of precision, recall, search delay and traffic costs for indexing and individual search operations.

II. Related Work

In [7], [6] we present a tree-based, incomplete approach to perform a capacity-based search. In this approach, all nodes participate in a tree structure and forward the list of nodes and corresponding capacities for the subtree they are in charge of one level higher in the tree. Ideally, the root has then a full list of all nodes in the tree. However, in order to not overload the nodes with these potentially huge lists, they are free to discard information on weaker nodes. Our approach presented in this paper does not discard any information but provides a lightweight access to all nodes in the network and is dependent of the capacities but independent of the actual number of peers.

Potentially, approaches supporting multi-dimensional range searches could be used to search for peer capacities. Unfortunately, existing approaches only partially take data dynamics into account.

Famous approaches are the following. Baton [9] provides a binary search tree in order to support range queries of values

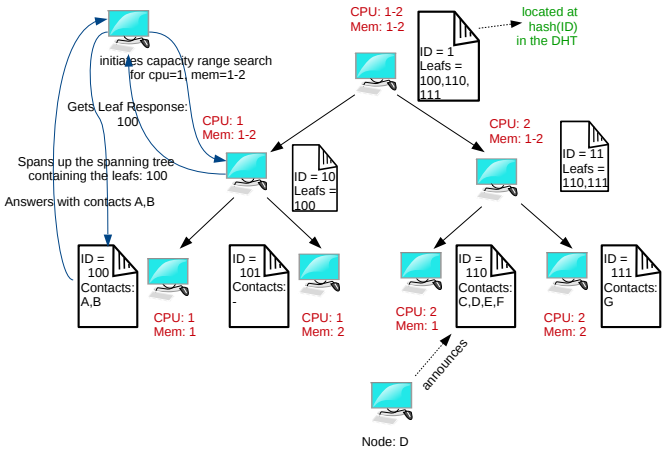


Fig. 1. Visualization of tree structure, maintenance and search

where each tree node is maintained by one peer of the network. Baton spans up a balanced tree whose nodes represent a certain range of values. The authors mentioned that search operation's costs are pricey with insufficient routing tables what ultimately lead to link failure.

SCRAP and MURK [4] use space-filling curves which map a multi-dimensional space to a one-dimensional kd-tree structure and skip-list. In contrast to SCRAP, in our approach nodes in the kd-tree are not peers in the network but dht objects and our search approach is more lightweight.

Carlini et al. [2] propose a tree-based approach whose ids are calculated by using space filling curves. For the search mechanism digest information are spread to the inner nodes of the tree and search is performed bottom-top. Also, very high dynamic data changing in each capacity space, lead to fluctuations of inner nodes as digest information have to be adapted. However our search approach is top-bottom by hopping into our tree at an inner node, thus not only a sub-set of capacities can be obtained. Maintenance simply propagates new capacities starting at the leaf node propagating up to the root node.

III. Our Solution: CapSearch

Our solution's main components are sketched in Figure 1. From a nutshell, a static kd-tree structure is used which can be locally calculated. Nodes in the tree are those which are responsible for the tree's logical nodes, i.e. certain dht objects. Contact information is held by leaf nodes and propagated to their parents which hold a detailed view of leaf nodes in their sub tree containing at least one contact. The search is performed by contacting an inner node responsible for the desired range for a range search respectively a leaf node for an exact search. For a range search, the requesting node traverses, or delegates the traversing to a spanning tree, the obtained list of leaf nodes to get the desired contacts it is searching for. In the following, we outline our solution's components in more detail.

A. Definitions and Tree Characteristics

Firstly, we introduce the definitions we need for our tree structure. Let $CS_1, ..., CS_n$ be n distinct capacity spaces, e.g. CPU power frequencies, bandwidth of the Internet connection etc. The whole capacity space is denoted as $CS = \bigcup_{i=1}^{n} CS_i$. Each capacity space CS_i, $i \in \{1, ..., n\}$ is partitioned into k_i disjoint capacity ranks, i.e. $CS_i = \bigcup_{j=1}^{k_i} CS_{i_j}$. A capacity class is a set of n capacity ranks, one from each of the n distinct capacity spaces CS_i. So, a node's capacity C_i for $i \in N$, where N is the number of nodes in a communication network, can be expressed as a capacity class $C_i = \{CS_{1_{v_1^i}}, ..., CS_{n_{v_n^i}}\}$ for certain values $v_j^i \in \{1, ..., k_i\}$. A capacity range consists of exactly two capacity classes where one contains all minimum and the other all maximum capacity ranks of the range.

For the main data structure used in our solution we propose a static kd-tree with a branching factor of bF. It is static in the sense that all leaf nodes are allocated at a fixed height of $\sum_{i=1}^{n} \lceil \log_{bF}(k_i) \rceil$. Each of these leaf nodes is responsible for exactly one capacity class C_i, cf. Figure 1. Inner nodes of the kd-tree represent a certain capacity range which equals to the aggregate of their childrens' capacity ranges whereas the root represents the whole available capacity space of all n capacity spaces.

B. Creation of a Tree Structure on top of the dht

Algorithm 1 shows how the kd-tree is built up deterministically as long as the whole capacity space CS and the branching factor bF are known. Each node in the network builds up this tree, i.e. its structure is known to each node of the network. The tree is built up by splitting capacity spaces in bF pieces, if possible. The algorithm terminates if there do not exist capacity spaces that can be split up respectively the calculated tree height is reached. Please note, the kd-tree is never fully built up locally, instead, according to the needs, information is extracted (on the fly) from the tree.

Our proposed solution for capacity-based search uses this kd-tree structure in a structured peer-to-peer system by storing all tree nodes in the dht at well-known dht keys. These dht keys are hashes of the tree nodes' unique address. The addresses, in turn, are bit sequences identifying a node's exact position in the tree. This is a deterministic process, where the root node gets a special address, e.g. an empty bit sequence or 'root', and its children inherit their parent's bit sequence and append a certain bit sequence. This attached bit sequence depends on the branching factor of the tree, i.e. the length of sequence, and the position relative to its brother tree nodes. For this, brother tree nodes are binary numbered in ascending capacity rank order. Leaf nodes' addresses have an address length equal to the height of the tree multiplied by the number of bits needed to encode bF nodes, which is $\sum_{i=1}^{n} \lceil \log_{bF}(k_i) \rceil * \log_2(bF)$.

As all tree nodes have fixed positions in the dht, which are known by all participating peers in the overlay network, one can easily access data of a random tree node by calculating the constructed id and performing exactly one dht lookup for that id. Additionally, a node may execute a certain procedure

Algorithm 1 Build up kd-tree

Input: Branching factor bF, Capacity space CS
Output: kd-Tree T

```
1: function KDTREE(bF,CS)                                   ▷ calculate T
2:     spaces ← CS.LENGTH
3:     maxDepth ← Σ_{i=1}^{n} ⌈log_{bF}(k_i)⌉
4:     T ← INITEMPTYTREE()
5:     root ← NEWNODE(CS)                                   ▷ create root (whole space)
6:     T.APPENDNODESON(root,null)                           ▷ append root
7:     iter ← 1
8:     while T.DEPTH ≠ maxDepth do
9:         for all space ∈ CS do
10:            range ← space.K ÷ bF^iter
11:            if range ≤ 1 then                            ▷ space cannot be divided
12:                continue
13:            end if
14:            for all leaf ∈ T.GETLEAFNODES() do
15:                nodes ← []                               ▷ init new nodes array
16:                for i = 1 TO space.K STEP range do
17:                    node ← leaf
18:                    node.SETSPACE(space,i,i + range − 1)
19:                    nodes ← nodes + node
20:                end for
21:                T.APPENDNODESON(nodes,leaf)
22:            end for
23:        end for
24:        iter ← iter+1
25:    end while
26:    return T
27: end function
```

Algorithm 2 Maintenance in the kd-tree structure

```
1: local variables
2:     T, Static kd-Tree
3: end local variables
Input: Capacity class CC
4: procedure N.PUBLISHCAPACITY(CC)
5:     treeID ← T.GETTREEIDFORCAPACITYCLASS(CC)
6:     dhtKey ← HASH(treeID)
7:     nodeHandle ← n.NODELOOKUP(dhtKey)
8:     nodeHandle.REFRESH(treeID,dhtKey,n)
9: end procedure
Input: TreeID treeID, DHTKey dhtKey, Nodehandle caller
10: procedure N.REFRESH(treeID,dhtKey,caller)
11:     if n.RESPONSIBLEFOR(dhtKey) then
12:         leafNode ← n.DHTGET(dhtKey)
13:         isFirstEntry ← leafNode.ADDCONTACT (caller)
14:         n.RESCHEDULETIMEOUT(caller,treeID)
15:         if isFirstEntry = true then
16:             n.CHANGEDLEAF(false,treeID)
17:         end if
18:     end if
19: end procedure
Input: Nodehandle caller, TreeID treeID
20: procedure N.ONTIMEOUT(caller,treeID)
21:     leafNode ← N.DHTGET(HASH(treeID))
22:     wasLastEntry ← leafNode.REMOVECONTACT (caller)
23:     if wasLastEntry = true then
24:         N.CHANGEDLEAF(true,treeID)
25:     end if
26: end procedure
Input: Capacity class fromCC, Capacity class toCC
27: procedure N.CAPACITYCHANGED(fromCC,toCC)
28:     n.PUBLISHCAPACITY(toCC)
29:     treeID ← T.GETTREEIDFORCAPACITYCLASS (fromCC)
30:     fromNode ← n.NODELOOKUP(HASH(treeID))
31:     fromNode.ONTIMEOUT(n,treeID)
32: end procedure
Input: Boolean del, TreeID leafID
33: procedure N.CHANGEDLEAF(del,leafID)
34:     parentID ← T.CALCPARENTOF(leafID)
35:     parentNode ← n.NODELOOKUP(HASH(parentID))
36:     parentNode.UPDATEID(del,parentID,leafID)
37: end procedure
Input: Boolean del, TreeID toID, TreeID leafID
38: procedure N.UPDATEID(del,toID,leafID)
39:     innerNode ← n.DHTGET(HASH(toID))
40:     innerNode.UPDATEFULLVIEW(leafID)
41:     if toID ≠ root then
42:         parentID ← T.CALCPARENTOF(toID)
43:         parentNode ← n.NODELOOKUP(HASH(parentID))
44:         parentNode.UPDATEID(del,parentID,leafID)
45:     end if
46: end procedure
```

if it is responsible for a certain type of tree nodes and a special event (a message for example) comes in.

C. Capacity Indexing + Capacity Changes

In this section we refer to the tree maintenance, i.e. how is the kd-tree, introduced in previous Section III-A, used to index nodes' capacities. This process is two-parted: The proactive update respectively refresh of a certain node's capacity and the update propagation in the tree.

In the first place nodes need to inject their current capacity class into the tree. For this, they periodically call the Algorithm 2's procedure *publishCapacity()* (line 4). Here, the corresponding capacity class's leaf node's address is calculated and a RefreshMessage is sent to the node responsible for the leaf dht node respectively, for simplification, the nodehandle's refresh method is called. The refresh method adds the contact to the leaf object's list of nodes and schedules or reschedules an existing timeout event. After a certain time the node's contact details are erased from the leaf's contact list, if it was not renewed. In the case a node's capacity changed, the node executes the *capacityChanged()* method (line 27) on the local nodehandle. Here, the responsible nodes for the old and new capacity are calculated and the corresponding nodehandles called. Please note, that the ability to remove a certain capacity is not mandatory due to a lazy approach but this has a positive effect on the freshness of data respectively recall of search queries.

Whenever the number of contacts at a leaf node changes from zero to one or from one to zero, i.e. the leaf node stores at least one entry or the leaf node does not store entries any more, the peer responsible for this dht object should react and update the state of the kd-tree. This is triggered by methods calling the *changedLeaf()* method (line 33). Here, either an update or a remove of the specified tree id is triggered (*updateID()* method see line 38). The updateFullView command in line 40 updates the tree's inner node's view of the sub tree. This is done by simply storing the provided leaf id in a set of ids. The remove respectively update tree procedure is pursued by calculating the inner node's parent id and calling its updateID method.

D. Efficient Range Search: Full-search and K-search

For the range search we distinguish two types of searches. On the one hand, we implemented the *full*-search, which aims to get all contacts in a given range, is discussed in Section III-D1. On the other hand, we propose the *k*-search, which obtains only k contacts aiming to minimize the blend between the needed capacity and the actual contact's capacity. The k-Search is explained in Section III-D2.

1) Full-search: Algorithm 3 presents important procedures for the full range search for a given capacity range.

The main idea is to derive, find and get the inner tree node which does overlap the whole range search, get its filtered view and efficiently iterate all containing leaf nodes with a divide and conquer algorithm in order to extract all relevant contacts of relevant capacity classes.

The range search starts in line 5 with an incoming range request. From the static tree the minimum and maximum capacity class' leaf ids and their common parent tree node is extracted (line 8). The common parent tree node's id is calculated by distilling the longest common prefix of the leaf node ids. After getting the node responsible for the hash of the calculated parent id, the filtered view is requested by contacting the responsible node (line 14). Here, the before mentioned node gets the corresponding inner tree node from the dht, which is a local access, and extends all saved tree nodes in its full view to leaf nodes, if applicable (line 18). This process replaces all inner nodes by their sub tree's leaf nodes. Due to storage scalability issues full views may contain inner nodes to save storage space and may introduce a certain number of false positive leaf nodes which will be included in this unfolding process. Please refer to Section III-E for a deeper analysis. However, after the unfolding process the full view only contains leaf nodes and with line 19 all leaf nodes, which do not lie within the capacity range, are removed. Back at the initiator of the search, the filtered leaf nodes respectively their tree ids are sorted and hand over to the ProcessSearchTree method (line 23) to initiate the leaf node traversal. We propose to traverse the leaf nodes, in order to gather all contact information as they fulfill the request, by creating a spanning tree with a certain branching factor `bF`. In the first part between lines 24 and 29 the given leaf ids are equally divided into `bF` sets of leaf ids. Each set's leaf id located in the middle of the sorted set is chosen to proceed. Therefore, the node responsible for the given `middleID` is looked up and the continuation is delegated to it. In the second part, which is between lines 30 and 33, the node notifies the initiator about the current node leaf node's contacts. Please keep into mind that this DHTGet operation is finished locally as `thisID` is the previous `middleID` and the current node must be responsible for this leaf node. Thus it belongs to the leaf nodes the initiator is searching for. While all leaf nodes, respectively nodes responsible for a certain leaf node, deliver their contacts to the initiator, the latter keeps track of the progress and can recognize the end of the search query (line 35).

2) K-search: The K-search offers a search to retrieve k contacts with a minimal distance to the requested capacity class. Thus, for a minimum required capacity class $C_{req} = \{CS_{1_{v_1^{req}}}, ..., CS_{n_{v_n^{req}}}\}$ we aim to find contacts with their capacity class $C_i = \{CS_{1_{v_1^i}}, ..., CS_{n_{v_n^i}}\}$ with $\sum_{s=1}^{n} CS_{s_{v_s^{req}}} - CS_{s_{v_s^i}}$ minimal and positive.

Our proposed divide and conquer algorithm divides the range from capacity class C_{req} to $C_{max} = \{CS_{1_{k^{max}}}, ..., CS_{n_{k^{max}}}\}$ into a predefined number of *tiles*. In four phases we divide this range into *tiles* maximum parts which cover the whole range. By exploiting the kd-tree structure, where each node represents either a range of capacity classes or a single capacity class, we are able to extract sub ranges which are disjoint per definition. With the extracted parts we are able to obtain certain leafIDs which fit in this part with one lookup. Another lookup gives us the desired contact information from the responsible node.

Let us refer to Algorithm 4 to make things clear. The Algorithm starts in line 5. First, just like in the full-search, in line 7 the tree node, whose capacity range overlaps the whole capacity range, is calculated. After this, the four phased extraction of desired number of tiles is performed (lines 8-11). Please note that this initialization is completely done locally by traversing the kd-tree in the memory. In step 1 the initial tree node is replaced by at least one of its children, if the children's range at least partly contains the requested range and we do not hit the maximum number of tiles. This step is performed in a breadth-first manner, so the before mentioned procedure is recursively applied to all new children before heading to the next level in the tree. In step 2 futile tree nodes

Algorithm 3 Full range search in kd-tree structure

```
 1: local variables
 2:     T, Static kd-Tree
 3:     callback, Search callback
 4: end local variables
Input: Capacity range capRange, Branching factor of spanning tree bF, Search
    callback caller
 5: procedure N.STARTFULLSEARCH(capRange, bF, callback)
 6:     callback ← caller
 7:     leafRangeID ← T.GETLEAFRANGE(capRange)
 8:     parentID ← T.GETCOMMONPARENT(leafRangeID)
 9:     nodeHandle ← n.NODELOOKUP(HASH(parentID))
10:     filteredleaves ← nodeHandle.GETFILTEREDVIEW(capRange,
    parentID)
11:     sortFiltleaves ← T.SORTLEAVES(filteredleaves)
12:     n.PROCESSSEARCHTREE(sortFiltleaves,bF,null,n)
13: end procedure
Input: Capacity range capRange,
Output: Extended and filtered view fullView
14: function N.GETFILTEREDVIEW(capRange,treeID)
15:     dhtKey ← HASH(treeID)
16:     if n.RESPONSIBLEFOR(dhtKey) then
17:         fullView ← n.DHTGET(dhtKey)
18:         fullView ← T.EXTENDTOLEAFIDS(fullView)
19:         fullView ← T.FILTERFUTILEIDS(fullView,capRange)
20:     end if
21:     return fullView
22: end function
Input: Set of LeafIDs leaves, branching factor bF, TreeID thisID
23: procedure N.PROCESSSEARCHTREE(leaves,bF,thisID,initiatorHandle)
24:     setOfleaves ← T.DIVIDEEQUALLY(leaves,bF)
25:     for all part ∈ setOfleaves do
26:         middleID ← T.GETMIDDLEID(part)
27:         nodeHandle ← n.NODELOOKUP(HASH(middleID))
28:         nodeHandle.PROCESSSEARCHTREE(part\middleID,bF,
    middleID,initiatorHandle)
29:     end for
30:     if thisID ≠ null ∧ n.RESPONSIBLEFOR(HASH(thisID)) then
31:         leafNode ← n.DHTGET(HASH(thisID))
32:         initiatorHandle.DELIVERRESULTS(
    leafNode.RETRIEVECONTACTS(),thisID)
33:     end if
34: end procedure
Input: Set of contact hits contacts, Sender node's tree id senderID
35: procedure N.DELIVERRESULTS(contacts,senderID)
36:     n.ADDHITSTORESULT(contacts)
37:     if n.ISLASTSEARCHHIT(senderID) then
38:         callback.SEARCHFINISHED(n.GETALLRESULTS())
39:     end if
40: end procedure
```

Algorithm 4 K-search in kd-tree structure

```
1: local variables
2:     T, Static kd-Tree
3:     callback, Search callback
4: end local variables
Input: Capacity range capRange, Desired number of nodes k, Desired non-overlapping
    ranges tiles, Search callback caller
5: procedure N.STARTKSEARCH(capRange, k, tiles, callback)
6:     callback ← caller
7:     parentID ← T.GETCOMMONPARENT(T.GETLEAFRANGE(capRange))
8:     parts[] ← n.EXTENDANDLIMITTOBOUNDS(tiles,parentID)  ▷ step 1
9:     parts[] ← n.DELETEFUTILEPARTS(parts[])              ▷ step 2
10:    sortP[] ← n.SORTBYDISTANCE(parts[],capRange)        ▷ step 3
11:    sortP[] ← n.SPLITTOREACHNOOFTILES(sortP[],tiles)    ▷ step 4
12:    rmng ← k
13:    for all part in sortP[] ∧ rmng ≥ 0 do
14:        rmng ← rmng - n.GETVIEWSANDCONTACTS(rmng,part,capRange)
15:    end for
16:    callback.SEARCHFINISHED(n.GETALLRESULTS())          ▷ finished search
17: end procedure
Input: Capacity range capRange, Maximum number of contacts l, Inner TreeID to
    get contacts from treeID
Output: Number of found contacts
18: procedure N.GETVIEWSANDCONTACTS(l,treeID,capRange)
19:    nodeHandle ← n.NODELOOKUP(HASH(treeID))
20:    filteredleaves ← nodeHandle.GETFILTEREDVIEW(
    capRange,parentID)
21:    current ← 0
22:    for all leafID in filteredleaves ∧ current ≤ l do
23:        nodeHandle ← n.NODELOOKUP(HASH(leafID))
24:        searchHits ← nodeHandle.GETCONTACTS(leafID)
25:        n.STORESEARCHHITS(searchHits)
26:        current ← current + searchHits.LENGTH()
27:    end for
28:    return current
29: end procedure
Input: LeafID to get contacts from leafID
Output: List of contacts
30: procedure N.GETCONTACTS(leafID)
31:    if n.RESPONSIBLEFOR(HASH(leafID)) then
32:        leafNode ← n.DHTGET(HASH(leafID))
33:        return leafNode.RETRIEVECONTACTS()
34:    end if
35:    return notResponsible
36: end procedure
```

are deleted by keeping the number of tiles. This step ensures that no unnecessary capacity ranges are kept. In step 3 all tiles' minimum capacity classes are sorted according to their distance to the required capacity class C_{req}. In the last step the tiles are split in a depth-first manner into smaller tiles to reach the desired number of tiles.

After this initialization we got a set of tree ids sorted by distance which represent certain capacity ranges. The loop in line 13 iterates over this set and requests the corresponding views of inner nodes and subsequently requests the contacts of the leaf nodes to hit the desired number of k contacts (see method getViewsAndContacts in line 18). Please note that this loop can be parallelized in order to minimize the search time by sacrificing bandwidth. The method getViewsAndContacts calls the getFilteredView method in line 20 from the full-search algorithm to receive a filtered view from an inner node (see Algorithm 3, line 14). For each of the obtained filtered leafIDs the getContacts method is called (see line 30) to retrieve all contact information for this capacity class. Eventually the loop in line 13 terminates either when all tiles have been traversed or the desired number (k) of contacts has been retrieved. The algorithm terminates by calling the callback (line 16).

E. Discussion, Enhancements and Future Work

As inner nodes save all of their sub tree's leaf node ids, where at least one contact node is located, it leads to an enhanced overhead when it comes to frequent changes of capacities. This is due to the fact that a certain leaf node id is saved on one node of every level in the tree. This is what we pay for to get a fast and precise full- respectively k-search, which is unaffected by the depth of the tree. In order to mitigate the costs of the maintenance one could delay tree updates for both, leaves and inner nodes. By doing this, these updates can be bundled and sent up the tree after a certain delay. This affects the precision of certain searches which ask inner nodes closer to the root.

One may observe that nodes closer to the root have to provide more bandwidth and storage space as they have a broader view than their children. This is true but it can be limited to a certain configurable cap. Inner nodes upwards the tree may start to combine certain leaves to their common parent node while reducing the number of introduced false positives. This approach can be compared to range bloom filters like ARFs [1]. In contrast to the ARFs we save a lot of storage space as for ARFs it is mandatory to save the whole tree structure, instead, we just save certain tree ids and can reconstruct the view on this sub tree. Besides saving storage space we can also save bandwidth by sacrificing the performance of searches in terms of delay. The latter is true because leaves have to be traversed which do not contain any relevant contact information.

Our proposed mechanisms are robust against any kind of failures, e.g. churn, as long as the underlying replication mechanism is reliable. Our search mechanisms always find all desired contacts in a certain range as long as these contacts keep their capacity up-to-date. This is realized by tracer routing of search messages, where the initiator always keeps track of the progress and thus can trigger the mechanism to pick up where it left off due to certain failures.

The proposed tree structure is a static kd-tree whose dynamic capabilities seem to be quite limited. The fact that each node of the network is able to locally construct the tree itself by considering certain parameters, i.e. number of capacity spaces, their ranks and the branching factor of the tree, may create opportunities to rearrange these meta information during a running network. One may apply a mechanism to broadcast certain meta information whenever the state of the art capacities have changed, so an adjustment of the capacity spaces and/or ranks is desirable. Such broadcast information propagation has been proposed by El-Ansary et al [3]. After the propagation of new meta information, a new kd-tree can be built up and used.

IV. EVALUATION

In this section we evaluate our solution to the following criteria: i) proof-of-work ii) precision of the full-search and k-search and iii) costs of the maintenance and search mechanisms.

TABLE I
PARAMETERS CHOSEN FOR OUR SIMULATIONS

Parameter	Value(s)
Number of nodes	1000
Number of seeker	240 ($\hat{=}\frac{2 searches}{minute}$)
Number of seeds	≥ 10
Churn	Active (from minute 161), Inactive
Simulation time	330 minutes
Capacity spaces (classes)	3 (512), 5 (512)
Used P2P overlay	Chord
Full Search: bF (= spanning tree branching factor)	4
K-search: k	8
K-search: tiles (= non-overlapping ranges)	10
Initial capacity class distribution	Normal
Capacity change period	All 10, 30 minutes
Search (start) range distribution	Normal

TABLE II
SIMULATION PROCESS

Period of time (in minutes)	Action performed
$0-90$:	Joining the overlay
$91-120$:	Do nothing - stabilizing overlay
$121-150$:	Start the service - initial construction of the kd-tree
161:	Enable churn, if specified
$170-190$:	Start periodic capacity changes
$200-320$:	Perform full- or k-searches
330:	End of simulation

TABLE III
CUMULATIVE AVERAGES AT THE END OF THE SCENARIO

Metric	Avg	Avg - StdDev	Avg + StdDev
Scenario B: K-search duration per search query	0.92	0.15	14.57
Scenario B: Full-search duration per search query	7.74	0.43	24.03
Scenario C: K-search duration per search query	1.19	1.16	14.59
Scenario C: Messages per tree update	101.75	92.57	148.42

Methodology

We implemented our solution in the PeerfactSim.KOM simulator [5], [10] and performed several simulations. Table I shows relevant parameters we used in our simulations. We performed at least 10 simulations with different seeds per simulation configuration. All scenarios had been simulated with 1000 nodes with 240 from them starting searches. Table II shows the simulation process which is a summary of actions performed during the simulation. After the p2p overlay has stabilized itself, the construction of the capacity tree is starting. To compare the maintenance of raising capacity spaces and ranks we perform a comparison of two capacity space setups in Section IV-C. For the capacity distribution we have chosen a normal distribution with the mean being the middle of each capacity space's rank, where 90% of all values should be in the 75% surrounding of the mean. By doing this, we simulate a skewed load of the kd-tree structure. In order to stress the maintenance of the kd-tree, we started periodic capacity change operations which simulate a certain reservation or change of the local node's capacities. A variation of this interval and its influence is evaluated in Section IV-D. In the search phase of the simulation, 2 searches per minute of either full-search or k-search had been performed. For each search, a normal-distributed capacity class has been drawn and set as the minimum capacity class of the capacity range. For both the full- and k-search the upper limit for the capacity range is the maximum capacity class. The performance of these searches is discussed throughout the next sections. Concerning the replication mechanism, we simulated an ideal replication mechanism which ensures that no dht entries are lost.

In the following we present our outcomes of both the full- and k-search starting with our default scenario with three capacity spaces and 512 ranks each, no churn, and a capacity change period of 30 minutes. Starting with this scenario we alter each parameter to present and discuss its outcome.

A. Scenario A: Proof of work - without Churn

In our first scenario without churn, we discuss results and highlight the differences between the full- and k-search. For the relative recall, more than 99% of the desired contacts are found with a very small standard deviation of less than 1%. False positives are a seldom with less than 1 per mil. For the search query statistics, between 50 to 200 respectively around 16 nodes are contacted while resolving the search query (see Figure 2b). Both searches have in common that a set of nodes in the kd-tree has to be contacted, i.e. one node in full-search and a maximum of the number of tiles in k-search. This reflects in the operation durations of the search queries (see Figure 2a). Due to a idealized network with a small latency of 2 ms and no packet loss the k-Search finishes its operation twice as fast as the full-Search (0.14s vs. 0.31s). Figure 2c shows that the tree level at which the initial full-search request arrives is at a very high level (near root). In more than 50% of the cases the root-node itself had to be contacted. Some modifications for this are planned and discussed in future works. However, Figure 2c also shows that for k-search we encounter requests between levels 15 and 3 with an average at 5 to 8.

While publishing a contact's capacity information at least 28 messages has to be sent, i.e. 27 is the height of the tree plus one for the kick off. When updating a certain contact's capacity this number doubles as a remove and an update operation has to be started (see minute 170 in Figure 2e). The whole publishing mechanism takes 1.7 seconds in average (see Figure 2d).

As for our service's maintenance traffic, which is the same for k-search and full-search, we encounter up to 5 byte/second averaged over all peers (see Figure 2g). In comparison, the k-search search messages' traffic is hardly noticeable averaged over all peers with only less than 0.1 byte/second. For full-search we obtained up to 1 byte/second averaged over all peers. Figure 2f reveals the cost distribution. At the first sight for inbound maintenance messages we have a huge disbalance but the peer with the most inbound maintenance traffic has to handle 92 byte/s (=741 bit/s) assuming this traffic is caused from minutes 170 to 330. For the full search messages, the most overloaded peer (root) had to send and receive only 10 byte/s respectively 80 bit/s. The traffic is distributed over 85% of all nodes for full-search and around 70% for k-searches. To compare our service's traffic with the network traffic, we

refer to Figure 2h. Our service, either with k-search or full-search generated least load compared with the top 5 message types which are from the Chord protocol. Please note that a lot of the lookup messages are initiated by our application. Nevertheless the node with the most load has to provide around 2000 byte/s for in- and outbound which equals to 16 kBit/s, which is quite feasible. At last, Figure 2i presents the number of objects and the local dht size needed by these objects per peer. At maximum a node had to save around 610 objects and consumed up to around 250 kByte.

B. Scenario B: Influence of Churn

In this scenario we evaluate the performance and costs of our solution in the presence of churn. Under churn, lookup operations of the dht may fail and should be rescheduled after a certain timeout. While relying on the stabilization mechanism of the overlay, eventually, owing to circumstances after a few lookups, every desired lookup will finish. For the k-Search, this reflects Figure 3a. The search duration increases to a maximum of five seconds with most of the time one-ish seconds. Aggregated over the whole simulation we measure an average duration of 0.92 seconds (see Table III), which is roughly the ten-fold of the non-churn scenario. For full-search this influence is even more dramatically as significantly more nodes need to be contacted thus running into several timeouts is inevitable. Figure 3b reveals an average maximum of 15 seconds which changes to 10 and finally 5 seconds. The averaged duration throughout the whole scenario is 7.74 seconds (see Table III).

These durations have a great impact on the precision of the search queries. For k-search we observe a drop to average precisions above 90% where the false positive are close to zero with one peak at 10% and smaller peaks like 3%. The precision average throughout the whole scenario is 93% with less than 3% of false positives (see Figure 3c). This is due to a small k, so finding one correct node more or less has a big influence on the outcome. Figure 3d presents the recall of the full-search, where the average precision goes down from initial 80% to around 60% and an overall average of 64%. The search statistics itself, like the number of contacted nodes in the search process and the tree depth, did not change.

Churn also affects the tree update operation's duration and its number of sent messages, which is independent of the actual used search query. Figure 3e offers the duration for the whole update tree operation, which lies between 5 to 15 seconds. The aggregated duration throughout the scenario is 7 seconds, which is roughly the four-fold of the non-churn scenario. For messages sent we refer to Figure 3g where the average plus standard deviation reaches 70 but the average is still around 28*2 = 56, which is the number of sent messages for the remove and update traversing the kd-tree. For only those tree update operations with a remove and update process we encounter a minimum of 56 messages and the average is just below 60, so in average we only have to send two more messages per capacity update operation (see Figure 3f). In exceptional cases the average plus standard deviation reaches numbers above 80. Aggregated throughout the whole scenario we encounter 60 messages per tree update for capacity update operations in average.

Figure 3h exhibit the costs where we notice a slightly lower maintenance outbound traffic for a few peers which is due to churn events. The increased number of maintenance messages affect the maintenance traffic, which is shown in Figure 3i, at the beginning where it raises to 6 byte/s from 5 byte/s. At the end of the simulation, the maintenance traffic normalizes and the traffic matches the non-churn scenario. Also the number of participating nodes for search queries is lowered by 10% for k-Search respectively 18% for full-Search.

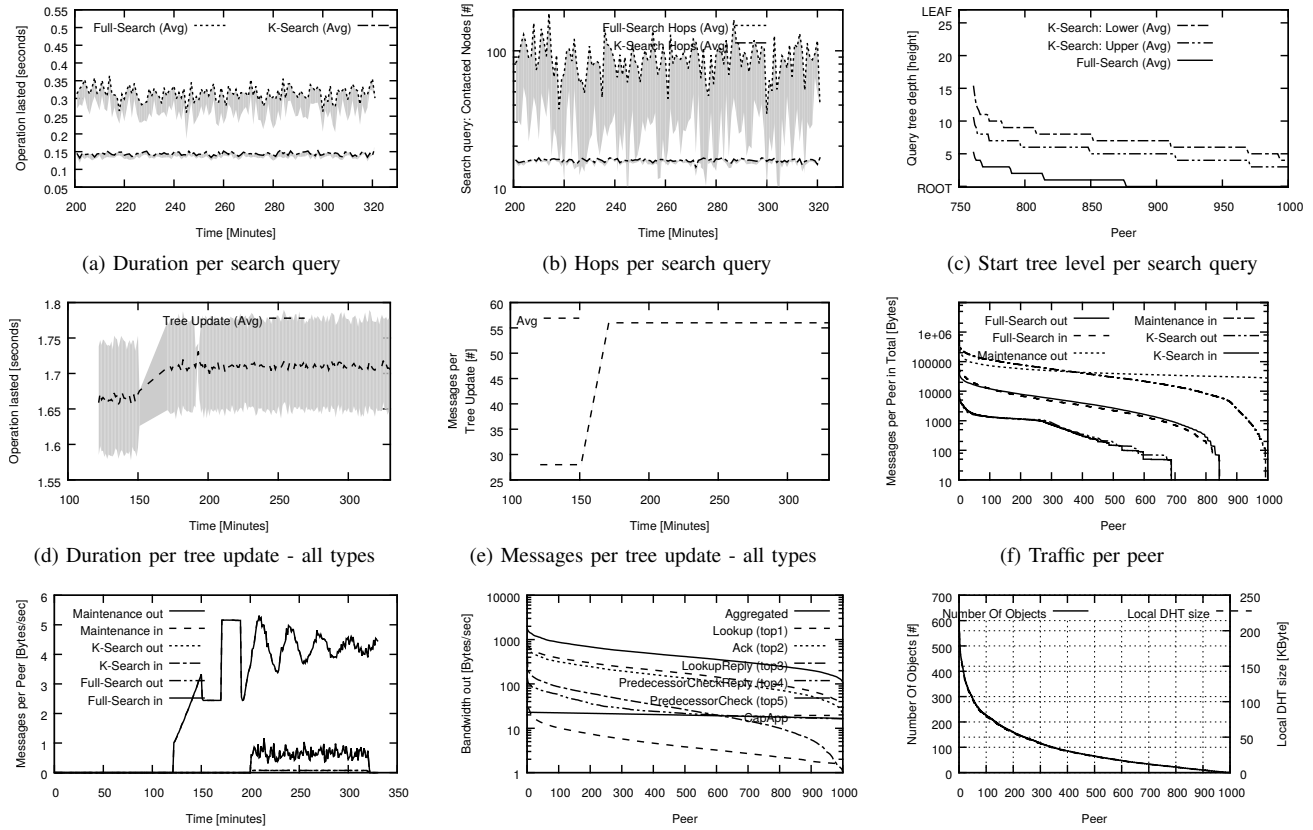
C. Scenario C: Raising number of Capacity Spaces

In the third scenario we analyze the behavior of our system with an increased number of capacity spaces. With five capacity spaces with 512 ranks each the kd-tree's height increases from 27 with three capacities to 45. The increase results in a growth of the number of nodes in the two dimensional kd-tree from $2^{28}-1 = 2.7*10^8$ to $2^{46}-1 = 7*10^{13}$. Its influence on the search performance, the maintenance and the mechanisms' costs is discussed below.

First, Figure 4a presents the search duration of the k-Search over time which looks quite similar to Figure 3a but Table III reveals that the average time needed for a search raised from 0.92 to 1.19 seconds. Search statistics are just like the scenario with three capacities which also applies to the query depth. This in turn means that search queries ask for nodes near the root, where the probability of getting outdated data raises. These two observations lead to a lowered search precision as shown by Figure 4b. A few drop outs worsen the overall precision, which lowers to 90% precision averaged over all search queries throughout the simulation. The false positives are not affected significantly.

For full-Search Figure 5a shows the duration over time. Noticeable is the drop to 9 seconds at maximum and afterwards one to four seconds. The aggregated duration over all search queries is 2.5 seconds, thus this is an improvement from 7 seconds. This is because in this scenario, see Figure 5d, in search query ranges are less contacts located than in the three capacity spaces scenario, thus time to build up the spanning tree is reduced. This effect can be observed in Figure 5c, where only 25 nodes had been contacted in overall average. Just like the k-Search search queries tree depth, the full-Search queries' depth did not change, thus we expect to forfeit some precision on this point. Unlike the k-Search's precision, the full-Search's precision has improved as the shortening of the search duration predominate the potential loss of precision by asking nodes near the root node. Figure 5b shows the over time plot where the overall precision average is 70%, which improved from 64%.

For the maintenance of the tree we encounter the following observations, which apply to both search types. The messages per tree update raises to a maximum of 120 with an average of 90, which is the tree height multiplied by 2 as shown in Figure 4d. Table III shows that the overall average number of



(a) Duration per search query (b) Hops per search query (c) Start tree level per search query

(d) Duration per tree update - all types (e) Messages per tree update - all types (f) Traffic per peer

(g) Traffic measured by time (h) Categorized outgoing bandwidth (i) DHT load per peer

Fig. 2. Scenario A: 30m capChanger, without churn, 3 capacity spaces

sent messages of all capacity update operations equals 101.75. This also leads to an extension of the tree update duration which starts with an average 25 seconds and goes down to 10 seconds (see Figure 4c). The aggregated duration of the tree update operation is 10 seconds, which is a smaller worsening from 8 seconds with three capacity spaces.

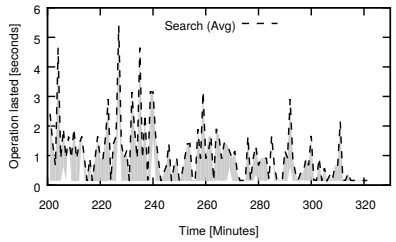
The costs reveal an overall higher maintenance bandwidth utilization with a peak of 9 byte/s in Figure 4e. At the end of the simulation it even out to 6 byte/s which is an increase from 4 byte/s. As the number of dht objects raised due to a significant amount of new inner nodes, Figure 4f shows the number of objects each peer stores and the space peers must make available. We register an increase from 250 to 470 kByte and 610 to 1350 objects per peer, so we observe a doubling of the number of objects and the size each peer should provide.

D. Scenario D: Stressing the Maintenance

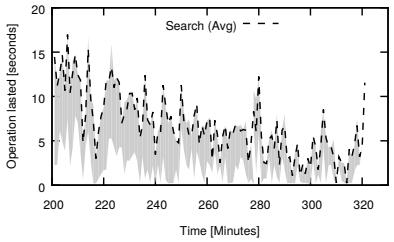
In the last scenario we analyze the behavior of an increasing maintenance load by adjusting the capacity change period from 30 to 10 minutes, thus all 1000 nodes change their capacities every 10 minutes (plus a certain jitter to avoid loose coupling). Please note that a 10 minutes capacity change period is quite a big time-gap for real-world scenarios and would bring a lot of limitations. We performed this scenario to compare it with the default setting in order to get a better understanding and conclude more general scenarios.

By increasing the capacity change period the messages sent per tree update remains the same and the tree update duration should also. In our outcome we encounter a slightly improvement in the aggregated overall average but it is classified of being not significant. The traffic, refer to Figure 6c, shows a peak 9 from 6 and is adjusted to 7 from 4 byte per second.

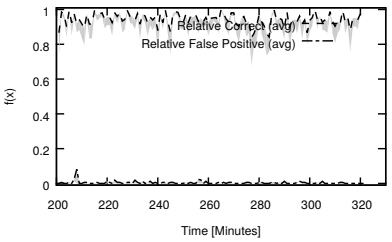
The k-Search queries' duration and precision is not affected by the change of the change period (see Figures 6a and 6b), which is a very good characteristic as this means that the update operation is fast enough and the search processes finish in a reasonable amount of time to not influence the search's outcome. For the full-Search we encounter a slightly lower duration throughout the simulation time (see Figure 5e). The overall aggregated duration is shortened in comparison to the 30 minutes scenario by one second. The precision, refer to Figure 5f, records looses of around 5% throughout the course. Consequently, the overall aggregated average confirm the observation with 58% instead of 64%. The false positives are not affected. This effect exists because capacities may change during the 6 seconds of getting the current view and spanning the search tree. As the latter consumes the most of the search duration we loose precision when increasing the capacity change period without adjusting search timings.



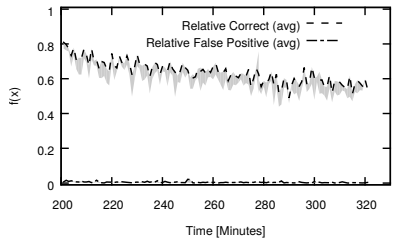
(a) K-search: Duration per search query



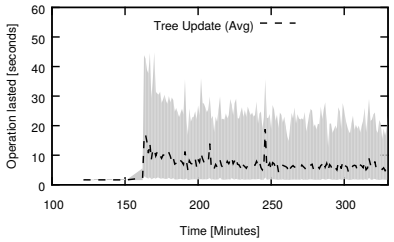
(b) Full-search: Duration per search query



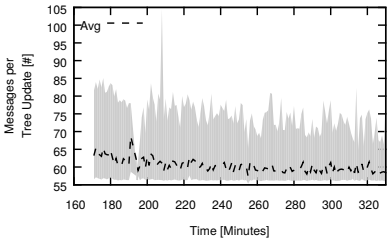
(c) K-search: Precision and false positives per search



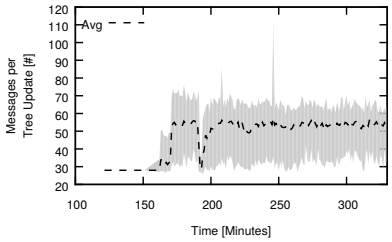
(d) Full-search: Precision and false positives per search



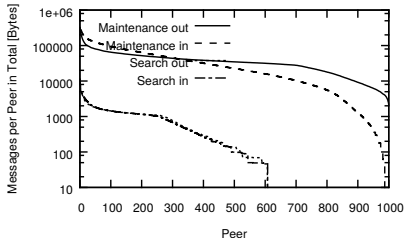
(e) Duration per tree update - all types



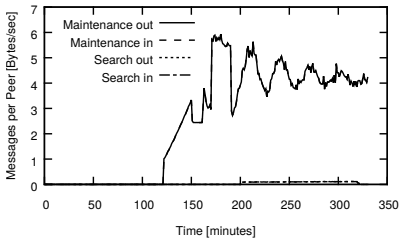
(f) Messages per tree update - update



(g) Messages per tree update - all types



(h) Traffic per peer



(i) Traffic measured by time

Fig. 3. Scenario B: 30m capChanger, with churn, 3 capacity spaces

V. Conclusion and Future Work

Our proposed solution CapSearch offers a lightweight indexing and search mechanism for highly dynamic data performing updates in $\mathcal{O}(d)$, where d is the dimension of the data. The kd-tree-based structure contains logical nodes, i.e. dht objects identifying certain tree nodes, which are maintained by nodes in the system. For the search only one lookup has to be started in order to obtain a list of leaf nodes storing contact information about contacts currently located in the desired search range. Dependent on the use case k nodes or all nodes in a desired search range are found. Evaluation shows a high precision for k-Search and full-Search with a low false positive rate. In future we will realize points listed in the Section III-E.

Acknowledgment

We would like to thank Ivan Tolstun for his great work. Computational support and infrastructure was provided by the Centre for Information and Media Technology (ZIM) at the University of Düsseldorf (Germany).

References

[1] K. Alexiou, D. Kossmann, and P. Larson, "Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1714–1725, 2013.

[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] S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi, "Efficient broadcast in structured P2P networks," in *International workshop on Peer-to-Peer systems*. Springer, 2003, pp. 304–314.

[4] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: multi-dimensional queries in P2P systems," in *Proceedings of the 7th International Workshop on the Web and Databases 2004*. ACM, 2004.

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

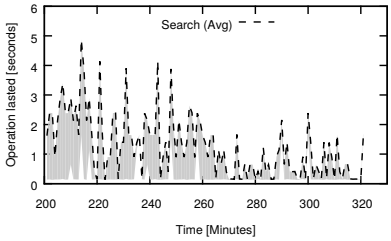
[6] K. Graffi, C. Groß, D. Stingl, H. Nguyen, A. Kovacevic, and R. Steinmetz, "Towards a P2P Cloud: Reliable Resource Reservations in Unreliable P2P Systems," in *IEEE ICPADS '10: Proceedings of the International Conference on Parallel and Distributed Systems*, 2010.

[7] 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 *IEEE ICPADS'08: Proceedings of the International Conference on Parallel and Distributed Systems*.

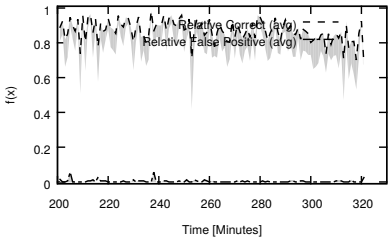
[8] K. Graffi, K. Pussep, S. Kaune, A. Kovacevic, N. Liebau, and R. Steinmetz, "Overlay Bandwidth Management: Scheduling and Active Queue Management of Overlay Flows," in *IEEE LCN '07: Proceedings of the International Conference on Local Computer Networks*, 2007.

[9] H. V. Jagadish, B. C. Ooi, and Q. H. Vu, "Baton: A balanced tree structure for peer-to-peer networks," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 661–672.

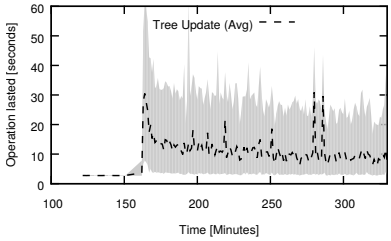
[10] A. Kovacevic, S. Kaune, H. Heckel, A. Mink, K. Graffi, O. Heckmann, and R. Steinmetz, "PeerfactSim.KOM - A Simulator for Large-Scale Peer-to-Peer Networks," TU Darmstadt, Tech. Rep. Tr-2006-06, 2006.



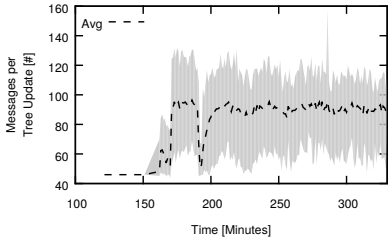
(a) Duration per search query



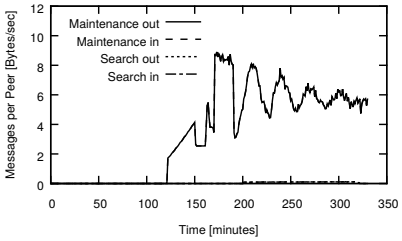
(b) Precision and false positives per search



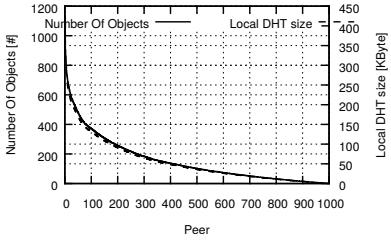
(c) Duration per tree update - all types



(d) Messages per tree update - all types

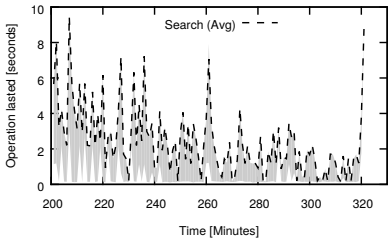


(e) Traffic measured by time

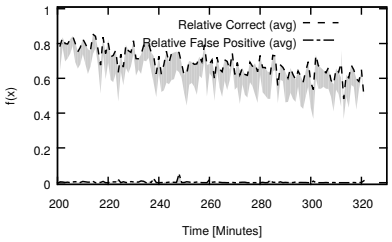


(f) DHT load per peer

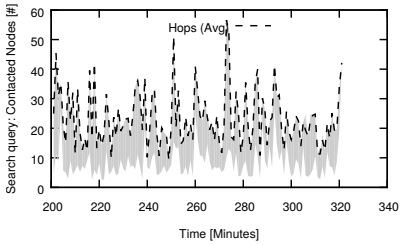
Fig. 4. Scenario C for k-Search: 30m capChanger, with churn, 5 capacity spaces



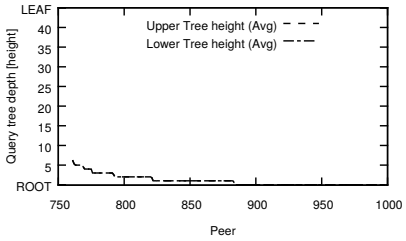
(a) Sc.C: Duration per search query



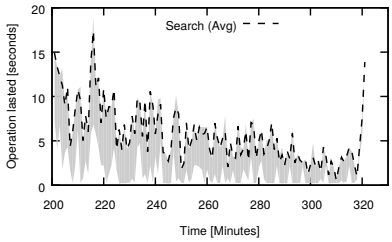
(b) Sc.C: Precision and false positives per search



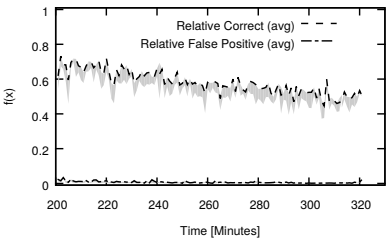
(c) Sc.C: Hops per search query



(d) Sc.C: Start tree level per search query

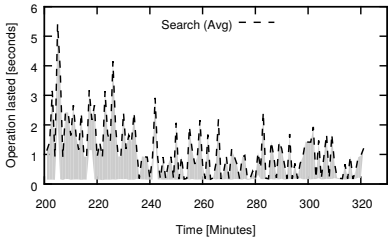


(e) Sc.D: Duration per search query

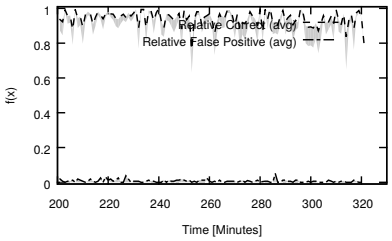


(f) Sc.D: Precision and false positives per search

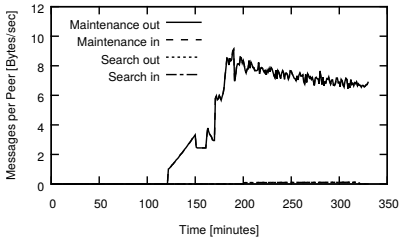
Fig. 5. Scenario C/D for full-Search: 30m/10m capChanger, with churn, 5/3 capacity spaces



(a) Duration per search query



(b) Precision and false positives per search



(c) Traffic measured by time

Fig. 6. Scenario D for k-Search: 10m capChanger, with churn, 3 capacity spaces