Deliverable FI3-D3.2.6
A prototype of Click-based caching/transport

Kristian Slavov

Tivit Future Internet Program
(Tivit FI)

Period: 1.4.2011 – 30.4.2012

Tivit, Strategisen huippuosaamisen keskittymän tutkimusohjelma

Rahoituspäätös 1171/10, 30.12.2010, Dnro 2790/31/2010

www.futureinternet.fi

www.tivit.fi

This work was supported by TEKES as part of the Future Internet programme of TIVIT (Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT).

Executive summary / Internal release

Title: A prototype of Click-based caching/transport

In this work we have implemented Click modules allowing simple caching solutions for pubsub architecture.

In addition, an analysis has been performed on the effect of using bloom filters in transport protocols with respect to caches. Simple transport protocol uses direct indexing of a packet that is required. For caches this means "find one match" operation, and there are lot of efficient searching algorithms. Bloom filters, instead, would allow indexing many packets with a single request. At the same time they transform the lookup problem into "find all that match" operation. Most algorithms cannot handle this.

Content: Click-modules, theoretical analysis

Contact info: Kristian Slavov, kristian.slavov@ericsson.com

Link: http://users.piuha.net/kslavov/ICTSHOK/

1 Introduction

One of the targets of Future Internet project is to enable network elements act as caches to data published by others. This quality allows swarming downloads as currently used by e.g. Bittorrent protocol. These replicas of original information allow the network to better survive network outages.

Our work is concentrated in caching. How do we manage network level caching in Future Internet, and can some transport protocol be more effective with respect to the caches.

We start by a prototype used to model a simple forwarding cache. It caches all the data publication packets it sees. When a subscription request is seen, the internal database is consulted, and if respective data chunk is available, it is sent back at the subscriber. The prototype assumes simple sequence number kind of requests. Essentially you request N chunks at once. The shortcoming of this mechanism is that it requires quite a number of requests when the publications grow large. As a theoretical study we then studied whether changing this mechanism to a Bloom filter variant would alleviate the problem.

2 Prototype

The caching prototype was implemented as a set of Click modules. Click is a modular router framework, allowing users to implement actions to which the network flows are subjected.

2.1 Click Modules

- **PSCache**

This contains the actual implementation of a simple balanced binary tree cache. It has no inputs nor outputs.

- **PSHash**

This module is responsible for hashing the incoming packet based on its version identifier. The hashed value is used as a key in the cache. The actual packet is not modified, but forwarded through with it's internal state annotated with the hash value.

- **PSStore**

Looks for packet's internal state for annotated hash value. This value is then passed together with the relevant packet content to the PSCache module.

- **PSLoad**

Same as PSStore, except that the annotated hash value is used as a key to lookup data from the PSCache module. A new Pubsub packet is then created, and the content for it is received from the cache.

- **PSClassify**

This module implements Pubsub specific classifying algorithm enabling multiple outputs for various pubsub packets. A simple language is supported for specifying filters for each output. This module is similar to the IPClassify (and other *Classify modules already found in Click).

- **PSCheck**

Check that the incoming packet is a pubsub packet. Annotate with some information about headers (locations in the packet). Non-pubsub packets are routed to different output port.

2.2 Example

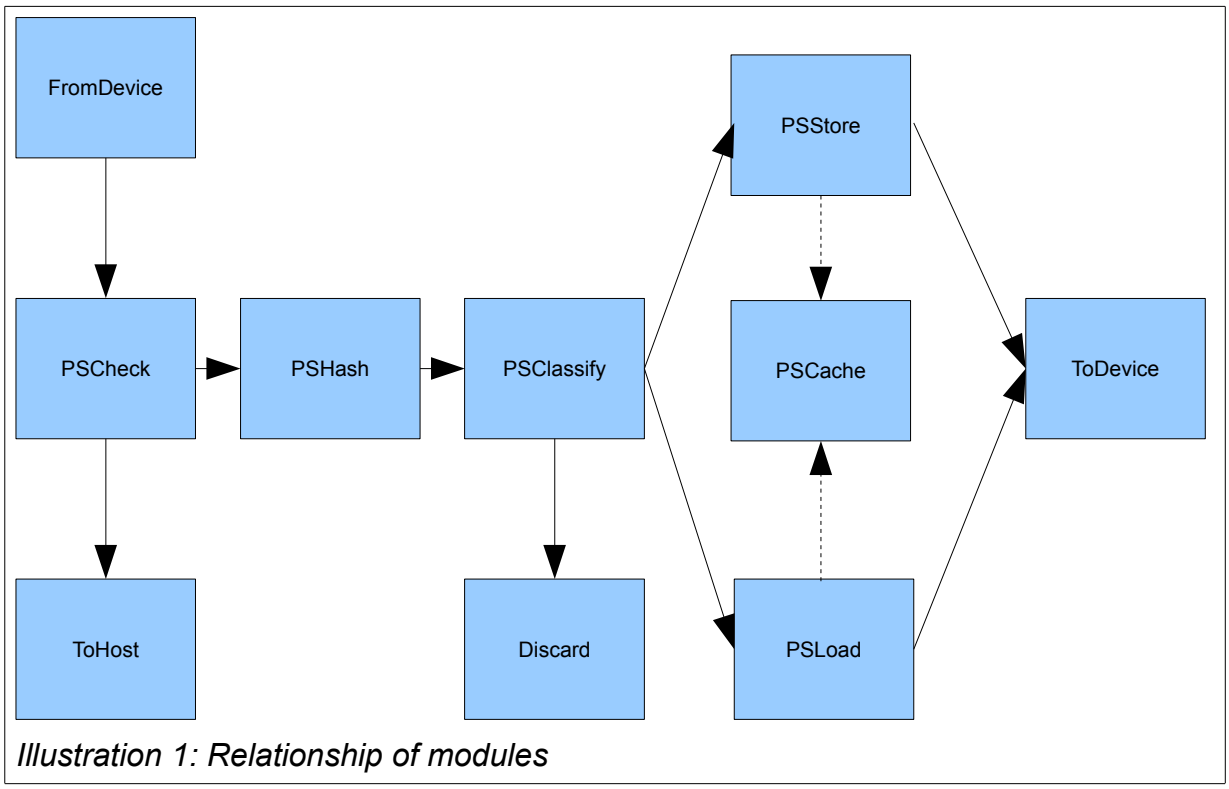

Illustration 1: Relationship of modules

Illustration 1 depicts an example that implements a simple forwarding cache. Packets coming from the network are first examined. If they are destined to the actual host, no further processing is done. On the other hand, if they are to be forwarded and are pubsub packets, we hash the identifiers and classify the packet. Based on this classification we either store or try to load the data from the cache. Finally, a packet is sent back to the network. Depending on whether we did a store or load, the actual packet will be sent on a different path.

```
require(package "pubsub-cache");

cache :: PSCache()
check :: PSCheck()
class :: PSClassify(SUB && DATA, PUB && DATA)
load  :: PSLoad(cache)
store :: PSStore(cache)
ether :: EtherEncap(0xACDC, 01:02:03:04:05:06, FF:FF:FF:FF:FF:FF)

FromDevice(eth0) -> check[0] -> PSHash() -> class
                    check[1] -> ToHost(eth0)

class[0] -> load[0] -> ether -> Queue() -> ToDevice(eth0)
            load[1] -> Discard()
class[1] -> store -> ToHost(eth0)
class[2] -> ToHost(eth0)
```

Listing 1: Click router configuration matching Illustration 1

To test the cache, we created a simple python script sending raw pubsub packets to the network, and used a packet sniffer to verify that the data sent and data received really matched. Listing 2, shows packet captures of two published data chunks. Roughly the first 150 bytes are pubsub headers. The actual content on these chunks is a small text message.

Once the data chunks were published, we issued a request for the first chunk (or publication), and observed the data being sent back by the cache. Listing 3 shows this. The first message observed in Listing 3, is a subscription request message. More specifically it requests a chunk whose identifier starts with “D40A...”. The reply contains the requested data, which can we verified from Listing 2.

```
tcpdump: listening on tap0, link-type EN10MB (Ethernet), capture size 1500 bytes
11:59:27.437264 00:00:00:00:00:00 > ff:ff:ff:ff:ff:ff, ethertype Unknown (0xacdc), length 198:
        0x0000:  0104 ff00 0100 0000 0000 00aa 0000 0000  ................
        0x0010:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0020:  0000 0000 0000 0000 0000 04f0 0000 0000  ................
        0x0030:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0040:  0000 0000 0000 0000 0000 0001 0000 0000  ................
        0x0050:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0060:  0000 0000 0000 0000 0000 0555 d40a 0000  ...........U....
        0x0070:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0080:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0090:  0000 0000 0100 0000 0000 5468 6973 2069  ..........This.i
        0x00a0:  7320 6d79 2066 6972 7374 2070 7562 6c69  s.my.first.publi
        0x00b0:  6361 7469 6f6e 2e0a                      cation..
11:59:30.880749 00:00:00:00:00:00 > ff:ff:ff:ff:ff:ff, ethertype Unknown (0xacdc), length 211:
        0x0000:  0104 ff00 0100 0000 0000 00aa 0000 0000  ................
        0x0010:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0020:  0000 0000 0000 0000 0000 04f0 0000 0000  ................
        0x0030:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0040:  0000 0000 0000 0000 0000 0001 0000 0000  ................
        0x0050:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0060:  0000 0000 0000 0000 0000 0555 b20e 0000  ...........U....
        0x0070:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0080:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0090:  0000 0000 0100 0000 0000 416c 6c20 776f  ..........All.wo
        0x00a0:  726b 2061 6e64 206e 6f20 706c 6179 206d  rk.and.no.play.m
        0x00b0:  616b 6573 204a 6163 6b20 6120 6475 6c6c  akes.Jack.a.dull
        0x00c0:  2062 6f79 0a                             .boy.
```

Listing 2: Packet capture of two publications

```
11:59:53.725600 00:00:00:00:00:00 > ff:ff:ff:ff:ff:ff, ethertype Unknown (0xacdc), length 320:
        0x0000:  0102 ff00 0100 0000 0000 00aa 0000 0000  ................
        0x0010:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0020:  0000 0000 0000 0000 0000 0206 0000 0000  ................
        0x0030:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0040:  0000 0000 0000 0000 0000 0001 0000 0000  ................
        0x0050:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0060:  0000 0000 0000 0000 0000 0555 d40a 0000  ...........U....
        0x0070:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0080:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0090:  0000 0000 0100 0000 0000 06f0 0000 0000  ................
        0x00a0:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x00b0:  0000 0000 0000 0000 0000 0001 0000 0000  ................
        0x00c0:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x00d0:  0000 0000 0000 0000 0000 0555 d40a 0000  ...........U....
        0x00e0:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x00f0:  0000 0000 0000 0000 0000 0000 00aa 0000  ................
        0x0100:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0110:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0120:  0000 0400 0000 0000 0000 0001 0000 0000  ................
        0x0130:  0000                                     ..
11:59:53.733256 01:02:03:04:05:06 > ff:ff:ff:ff:ff:ff, ethertype Unknown (0xacdc), length 198:
        0x0000:  0104 4000 0100 0000 0000 00aa 0000 0000  ..@.............
        0x0010:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0020:  0000 0000 0000 0000 0000 04f0 0000 0000  ................
        0x0030:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0040:  0000 0000 0000 0000 0000 0001 0000 0000  ................
        0x0050:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0060:  0000 0000 0000 0000 0000 0555 d40a 0000  ...........U....
        0x0070:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0080:  0000 0000 0000 0000 0000 0000 0000 0000  ................
        0x0090:  0000 0000 0100 0000 0000 5468 6973 2069  ..........This.i
        0x00a0:  7320 6d79 2066 6972 7374 2070 7562 6c69  s.my.first.publi
        0x00b0:  6361 7469 6f6e 2e0a                      cation..
```

Listing 3: Subscription request and subsecuent cache reply

3 Network level caching with bloom filters

This section describes our work on an idea of using bloom filters in transport protocols, and the effects on network level (packet) caching. We begin by defining our assumptions:

1 A publication consists of N chunks, each uniquely identifiable by their content using a cryptographic hash function. A chunk identifier is 256 bits long.

2 Network packet size is not limited explicitly, but the network must be able to transfer the whole packet atomically (as a single data unit from network layer point-of-view).

3 Chunk size is 1400 bytes (actual payload)

4 Subscription request message is assumed to be of size 1504 bytes, and containing 256 bytes common headers.

5 When requesting chunk(s) they must be explicitly identified in the subscription request message.

6 Network level packet cache stores chunks instead of publications

7 One or more cache(s) reside on-path between the subscriber and the publisher, and act as forwarding nodes.

8 Each cache first searches its database, replies with copies, and only then forwards the request.

9 On a single path between the subscriber and the publisher there are zero or more caches, which together hold no more than 30 million different chunks. We do the calculations based on three different assumptions: first, there is one cache with 30 million entries, second 10 caches with 3 million each, and third 30 caches with 1 million entries.

In the following sections we contemplate feasiblity of using bloom filters to request chunks from a network level packet cache's point of view. For that purpose we estimate the memory and processing time costs of a cache. From network point of view, we want to minimize network traffic. Therefore we also pay attention to the maximum downlink flow a single request can achieve.

3.1 Simple approach: requesting explicit chunks

In this approach a subscription request message indicates which chunks it wants to receive. It is up to the subscriber to assemble the chunks to form complete publication. Based on assumptions 1, 3, 4 & 5, a subscription request message can fit (1504-256)/32 = 39 chunk identifiers. That is, one request message causes at the subscriber a maximum downlink payload flow of 39*1400 = 54,6kB.

A cache needs to inspect the request message, and lookup all chunk identifiers for possible matches. An important issue is what kind of algorithm to use at the cache to make the cache lookup fast. If a tree hierarchy is used, we can calculate a worst case space requirement estimation for the data structures for a tree of depth *d*.

$maxnodes = \sum_{i=0}^{d} min(N^i, total)$, where each node takes N*8 bytes (64 bit OS) of memory.

Accoding to our assumption 7, we arrive at following numbers for memory requirements (per cache):

	N=256, d=32	**N=16, d=64**	**N=4, d=128**	**N=2, d=256**
1 cache/30M	1.82 TB	0.23 TB	0.11 TB	0.11 TB
10 caches/3M	0.19 TB	22.8 GB	11.4 GB	11.4 GB
30 caches/1M	61.6 GB	7.69 GB	3.82 GB	3.81 GB

An intelligent implementation could store keys already at the middle nodes, thereby pruning the rest of the branches. Alternatively a prefix/radix-tree could be used to save memory, with added the cost of run-time complexity. Searching time would be around O(log(N)).

A hash table would offer small improvement. If hash table has 65536 buckets, each of them would contain, on average, 30000000/65536 =~450 entries. The space requirements would be around: 30000000 * 8 + 65536 * 8 = 0.24GB, and the searching time would be O((1/65536)*N) = O(N).

Since there can be more than one chunk identifier in the request, the lookup operation would need to be repeated for each one. This can cause significant delays in forwarding. To speed up the repeated checks, and depending on the algorithm, the request message could have the identifiers sorted. This may help by pruning non essential branches and/or lists.

3.2 Bloom filter based chunk indexing

The main idea is to replace explicit indexing of chunks with a bloom filter containing a number of chunk identifiers. Essentially the bloom filter is a lossy compression of the chunk identifiers. This leads to obvious space savings, but also present some problems. The biggest challenge is to define the bloom filters so that the false positive ratio is kept on acceptable levels.

In this approach, a subscription request message contains bloom filters instead of chunk identifiers. On the message level, there is no difference. Publication meta data is used to gather the chunk identifiers, and a bloom filter is formed out of number of chunk identifiers. Subscriber would then send the message to the publisher. Publisher would compare which of his/hers chunks match to the bloom filter, and reply with all matching ones. A bit more

optimized version would include the publication identifier in the request message, enabling the publisher to limit the bloom filter matching to only the relevant chunks.

For a cache this means more work. For each chunk in the cache, we create a bloom filter representing that single element. The size of the bloom filter must be fixed (i.e. match with the subscription request message). The bloom filter in the subscription request message by default does not state how many chunks it contains. Therefore, a cache needs to check each element it has one by one. While the simple approaches from above can stop searching once a match is found, this approach cannot. In other words the cache is performing a *find all* operation. Trivially a counter could be added, but the effect is questionable. The counter would help only if all chunks are to be found in the cache.

3.3 Experimentation

Since Bloom filters are defined by multiple variables that are not independent, we run into a game of trying to optimize one characteristic at the expense of others. In our case we must balance between false positive probability, size of Bloom filters and amount of Bloom filter we may join together. For false positive probability there is an approximation, $P_{false} = \left(1 - e^{-kn/m}\right)^k$ where m is the size of Bloom filter in bits, n is the amount of elements joined together, and k is the amount of bits set as 1s in the Bloom filters.

In our experimentations we got good results with values m=512, n=14, and k=25. I.e. one 512-bit slot from subscription request message enables us to pack 14 chunk identifiers. Essentially this gives us 7 times better utilization of request packet space, with a false positive probability of $P_{false} = 7.6 * 10^{-8}$. On average there will be one false positive bloom filter match in about every 42 million entries. False positives themselves do not pose a threat. Once the mismatched data is transferred to the subscriber, he/she will be able to detect the mismatch trivially. The data will be hashed to retrieve the chunk identifier. This hash value will be different than what was requested, and thereby the subscriber can reason that there was a false match. With the above results, we arrive at maximum download flow rate (per single subscription message) of 7*54.6kB = 382.2kB.

A much more difficult problem is what to do about it. It is possible to try to create another bloom filter, and combine the mismatched identifier with other (than in the previous request) chunk identifiers. The new combination may not match the false data anymore. In case it did, we suggest that the transport protocol includes a flag ”bypass-cache”, to skip cache lookup operation.

Then there is the actual bloom filter matching. The matching operation is simple boolean and operation for the whole 512 bit block and each element. Since all elements must be checked, we arrive at O(N) efficiency, which is definitely worse than the simple approach.

Cube heuristic

- For each element in the cache, create a 512-bit bloom filter (with 25 bits set). This is very fast operation, and can be done when the element is first introduced into the cache.
- Map the newly created element bloom filter into a cube (of size N).
- Once a subscription request message arrives, extract the bloom filter, and map it to the cube
- Compare only the matching elements (and a special ”always check” list).

The implementation of cube could be a 3 dimensional array or an N-tree. Depending on N and the fill-ratio of the data structure. For the mapping operation we use relative distance of three consecutive set bits, starting from the most significant bit. E.g., 00100001100... => (2, 4, 0); 000011100... => (4,0,0). Mapping succeeds if a suitable coordinates are found, i.e., $\forall i \in \{0,1,2\}, 0 \leq c_i < N$. If a mapping fails, the entry is added to a ”always check” list. All this is precomputed at the time of insertion of the chunk to be cached, and takes insignificant amount of time to be accomplished.

For the actual searching we map the bloom filter in the request to all possible combinations that result in valid coordinates. Then the heuristic proceeds to match the elements found in the respective coordinates against the request bloom filter. Once all coordinates are checked, all elements in the ”always check” list are also checked. What the heuristic is actually doing is matching the first 3 bits of the bloom filters, and disregarding the rest.

3.4 Efficiency

To test the efficiency of the cube heuristic, we created 10 million bloom filters based on random data. Each bloom filter was mapped into a variable sized cube. This test was repeated 10 times. Results in Figures 2 and 3, show the 95% confidence level.

Figure 2 represents the percentage of elements (bloom filters) that are matched against the request bloom filter. An interesting result is that regardless of the input material, the amount of elements that cannot be mapped into the cube seems stable. The amount of elements in the cube is not explicitly depicted on the figure, but can be trivially calculated (= 100 - elements outside). Big variation (roughly 10% of all elements) in cube matches might suggest that the mapping algorithm is not well suited. Our tests show that the cube is skewed, therefore some coordinates yield higher match count. However, a positive indication is that even though the amount of elements in the cube increases 50%, the number of candidates is still within 6 to 18 percent. Note that the these cube matches are not final matches, just the candidates for the final match.

Figure 3 depicts the overall searching time spent on evaluating candidates (i.e. checking cube coordinates) and then matching them. Similar fluctuation in results as in Figure 2 can be seen. Mainly this is due to increased/lowered number of candidates of the previous step. However, since we are dealing with huge amount of data, the memory operations have an impact. This was evident when we first compared cube heuristic against a brute force matching. The brute force matching gains from checking all elements in order, thereby causing less page faults. If all elements were consecutively in memory, the brute force searching resulted in searching time of 340ms. We feel that this does not represent reality, as a cache replacing entries constantly would trash the memory areas, and the elements would be spread all around the memory. This is the case even if the index array remains sequentially ordered. Therefore for Figure 3, we randomly reshuffled the entries. The end result was huge jump in page faults, and in total searching time.

There is very little optimization done, only by the compiler. For data structures we used vector from standard template library for C++. The measurements were made on laptop housing a 2.3Ghz quad-core Intel CPU (without multicore support) with 4GB RAM. Our target was not to find absolute performance values, but to get a rough estimate. To get maximum out, the data structures could be tweaked and some phases reordered.

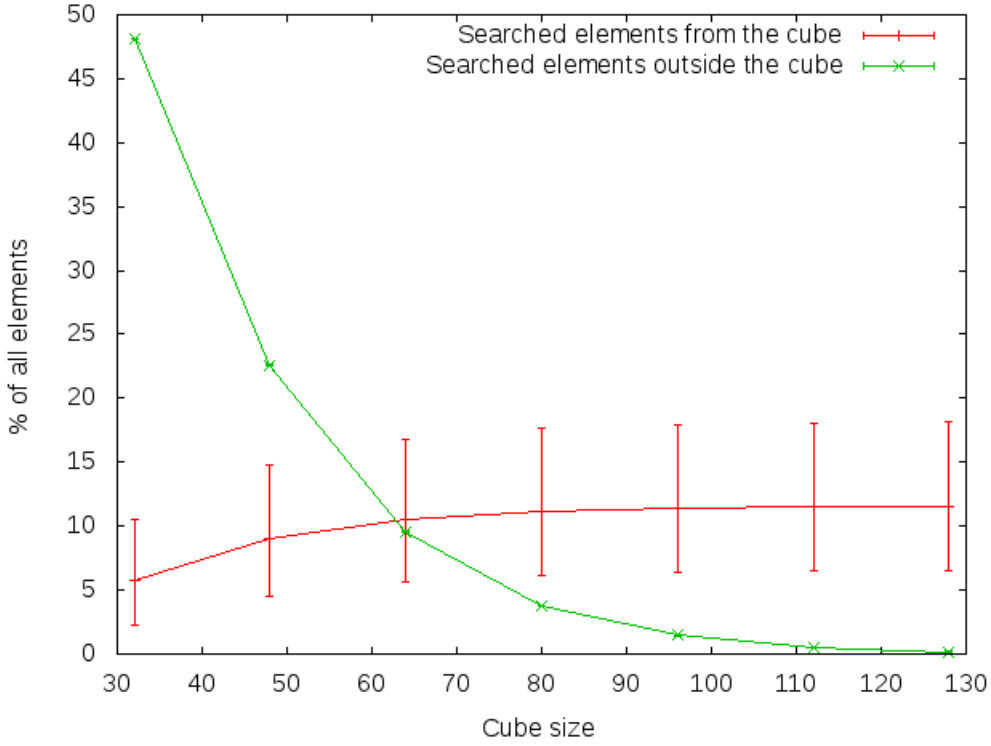

Figure 2: Where are elements searched from?

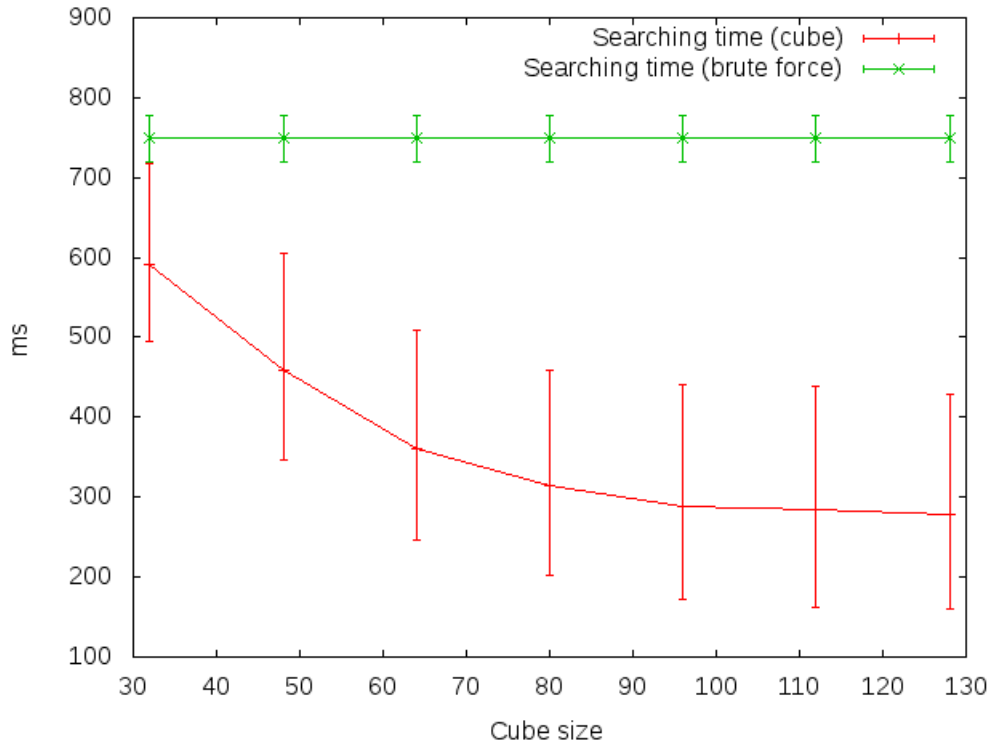

Figure 3: Time required to select candidates and match them (i.e. search time)

As for the memory consumption, with the cube we have N^3 slots (coordinates) in the cube, which all have to hold a number of bloom filter pointers. Each pointer, naturally, points to the bloom filter itself, which is 64 bytes long. The basic structure requires therefore, N^3 * 12 + 10M * 8 + 10M * 64 = N^3 * 12 + 720MB. Assuming N=128, and 12 is the size of the data structure at each of the cube's coordinates, we get approximately: 750MB. We are disregarding the memory requirements for the actual chunks.

In addition to the 750MB, we need a data structure where we collect the candidates to perform the final matching. However, these two steps could be joined together. That way once a set of candidates are located, they are immediately checked.

3.5 Conclusion

We set out to investigate could we use bloom filters in transport protocol requests, so that the network caches would still be able to serve data chunks. The hardest problem was to find a data structure allowing fast matching of the request bloom filter against all elements in the cache. Especially the fact that there may be several matches in the cache

means that all elements must be considered candidates. Finding one match does not exclude possibility of more matches.

Initially we considered various tree-like structures with early pruning of impossible branches. This would work in the request bloom filter contained a lot of zeros, however at our assumed values about 70% of the bits are ones (worst case). In all cases the trees grew so huge that we abandoned that approach.

Our cube heuristic seems to deliver small benefit. It does not give clear answer whether bloom filter matching is too expensive.