Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 607–614

ISBN 978-83-60810-51-4

Towards Caching Algorithm Applicable to Mobile Clients

Pavel Bžoch[1], Luboš Matějka[2], Ladislav Pešička[3], Jiří Šafařík[4]
University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering
Univerzitní 8, 306 14 Plzeň, Czech Republic
Emails: pbzoch@kiv.zcu.cz[1], lmatejka@kiv.zcu.cz[2], pesicka@kiv.zcu.cz[3], safarikj@kiv.zcu.cz[4]

***Abstract*—Using of mobile devices has grown over the past years. Under the term "mobile devices", we can see cell phones, personal digital assistants (PDA), smart phones, netbooks, tablets etc. Mobile devices provide many function e.g. accessing internet and e-mail, playing music and movies, accessing files from remote storage. Disadvantage of mobile devices is that connection to the internet can vary. It can be very fast while using 3G mobile network or very slow while using an old GPRS connection. The newest mobile communication technologies are not available everywhere. But the users usually wants to access their files as quickly as they can access them on wire-connection.**

If data are demanded repeatedly, they can be stored in mobile device in an intermediate component called a cache. The cache capacity is limited, so we should store in the cache only data that will be probably required in the future. In this paper, we present innovated caching algorithm. The algorithm is based on local and server statistics that are used to predict user behavior.

I. Introduction

NEED of storing a huge amount of data has grown over the past years. Whether data are of multimedia types (e.g. images, audio, or video) or are produced by scientific computation, they should be stored for future reuse or for sharing among users. Data files can be stored on a local file system or on a distributed file system.

Local file system (LFS) provides the data quickly but does not have enough capacity for storing a huge amount of the data in general. LFS is also prone to failure. Failure of LFS usually cause more or less temporary loss of data accessibility, or even loss of data. On the other hand, a distributed file system provides many advantages such as reliability, scalability, capacity, etc.

Most of distributed file systems (DFS) are developed for wired clients and do not support mobile devices. Accessing files from mobile devices requires algorithms which take into account changing communication channels caused by user's movement. DFS that are widely used were made before mobile clients have been spread, and it is difficult to develop mobile client applications now. None of current DFS e.g. Andrew File System (AFS), Network File System (NFS), Coda, InterMezzo, BlueFS, CloudStore, GlusterFS, XtreemFS, dCache, MooseFS, Ceph and Google File System does not have suitable clients for mobile devices [1], [2], [3].

Accessing files from mobile devices brings some problems that must be solved**.** Mobile devices have limited capacity for storing user content. They can store up to GB of the data. DFS can store TB of the data. Also the speed of wireless connection is low in comparison to wired connection. In addition, speed of wireless connection can vary. This can be caused by user's movement. The size of transferred data can be restricted by mobile connection provider. But the mobile users wish access their data as fast as possible and without restrictions. If we suppose that the users download the same data repeatedly, we can use cache to increase system performance. In this paper, we will focus on using the cache in mobile clients in distributed file system

A cache is an intermediate component which stores data that can be potentially used in the future. While using cache, the whole system performance is improved. The cache is commonly used in database servers, web servers, file servers, storage servers etc. [4]. The cached content is usually stored in high speed memory (e.g. RAM). However, cache capacity is not sufficient to store all requested content. The cache functionality is depicted in Figure 1.

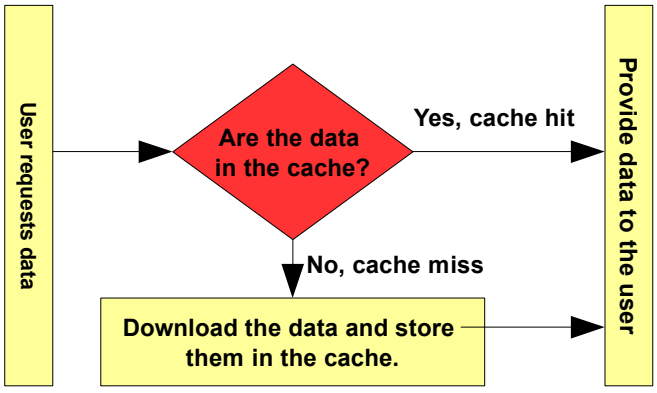

Figure 1. Cache

This work is supported by the Ministry of Education, Youth, and Sport of the Czech Republic – University spec. research – 1311.

When the cache is full, a system designer must adopt algorithm which marks an old content in the cache to be replaced. This algorithm implements replacement policy. This policy tries to predict user's future behaviour. In subsequent text, we discuss cache policies used in distributed file systems.

The cache in the DFS can be on client side as well as on server side. The cache on the client side stores content that has been downloaded by a user who is running a client application. In this case, replacement policy is usually based on statistical information gathered from user's behaviour. The cache on the server side contains data which has been requested by the most users. Replacement policy in this case uses statistics gathered from users' requests.

Using cache on server and client side at the same time does not increase system performance. Increasing cache hit ratio on the client side causes increasing miss ratio on the server side and vice versa [5].

In section II, we provide state of the art in caching algorithms. We introduce simple, sophisticated and hybrid algorithms which are used in DFS.

In our approach (section III), we present a new caching replacement policy. In this policy, we use only client side caching. While designing replacement policy, we have employed both server and local statistics for increasing cache hit ratio, and for decreasing network traffic.

In section IV, we present performance analysis results for the new algorithm. These results were generated via simulation of user's behavior. As a remote storage for user files, we have used KIV-DFS. KIV-DFS is a distributed file system which is being developed at the Department of Computer Science and Engineering, University of West Bohemia. KIV is an acronym for Czech name of our department (Katedra Informatiky a Výpočetní techniky). This DFS is designed to support mobile devices.

II. Shortcomings of existing caching algorithms

The goal of this paper is an introduction of new caching algorithm that is suitable for use in mobile devices. As mentioned in the section "Introduction", a cache is an intermediate component which stores data that can be potentially used in the future. The cache employs caching policy that makes space for an incoming data when the cache is full. Clearly, optimal replacement policy replaces data that will be used farthest in the future. However, this policy is not implementable. We cannot look into the future to get required information. None of the presented caching policy will be better this optimal policy.

In this section, we introduce some of replacement policies which are commonly used in distributed file systems or in operating systems. Caching policies can be divided into three categories: simple, sophisticated and hybrid algorithms.

A. Simple caching algorithms

Simple caching algorithms do not use any statistics or additional information for data replacement. For replacement decision, they usually employ other mechanisms. Examples of simple caching algorithms are RAND and FIFO.

RAND. RAND or Random is a simple replacement policy which chooses data to be replaced based on random selection [6]. It is very easy to implement this replacement policy. Disadvantage of RAND replacement policy is that RAND policy does not take of user's behaviour into account.

FIFO. First-In First-Out is another simple replacement policy. The data that are chosen to be replaced are the oldest in the cache [7]. Data in the cache are ordered in a queue. The new data are placed on the tail of the queue. When the cache is full, and new data come to the cache, the data from the head of the queue are thrown out. Disadvantage of FIFO is the same as of RAND policy – FIFO policy does not take user's behaviour into account.

B. Sophisticated caching algorithms

Sophisticated algorithms employ some statistical information about data in the cache: frequency of the accesses, and recency of last use of data. Frequency is used by LFU algorithm, and recency by LRU algorithm.

LRU. Least Recently Used is a sophisticated replacement policy which uses temporal locality of the data [6]. Temporal locality means that the data that have not been accessed for the longest time will not be used in the near future and can be replaced when the cache is full [8]. According to the tests [9], LRU seems to be the best solution for caching large files. LRU is frequently implemented with a priority queue. Priority is the timestamp of last access. The disadvantage of LRU policy is that the data block can be replaced even if the block was accessed for many times. In this case, the file will be probably requested in the near future again.

LFU. Least Frequently Used is another sophisticated replacement policy which uses statistical information. LFU replaces the data that have been used at least [6]. For each data block exists a counter which is increased every time the data block is accessed. Disadvantage of this approach is that the data blocks in the cache that have been accessed for many times in a short period of time remain in the cache, and cannot be replaced.

We will use RAND, FIFO, LRU and LFU policy for evaluation of our caching algorithm.

C. Hybrid caching algorithms

The disadvantages of LRU and LFU replacement policies result in hybrid algorithms. These algorithms use parts of LFU and LRU to get better results in cache hit ratio.

LRU-K replacement policy keeps the timestamps of the last K accesses to the data block [10]. When the cache is full, LRU-K counts so-called Backward K-Distance which

leads to mark data block to replace. LRU-K algorithm is used in DB systems [10]. Example of LRU-K is **LRU-2** which remembers last two access timestamps for each data block. It replaces then the data block with the least recent penultimate reference [11].

2Q replacement policy uses two queues. The first queue uses FIFO replacement policy for the data block and is used for data block that have been referenced only once. The second queue uses LRU as a replacement policy, and serves for so-called hot data blocks. Hot data blocks are blocks that have been accessed more than once. If new data block comes to the cache, it is stored in FIFO-queue. When the same data block is accessed for the second time, it is moved to the LRU-queue. [12]. 2Q algorithm gives approximately 5% improvement in hit ratio over LRU [12].

MQ replacement policy uses multiple LRU-queues. Every queue has its own priority. The data blocks with lower hit's count are stored in lower priority queue. If the number of hit's count reaches the threshold value, the data block is moved to the tail of queue with higher priority. When the replacement is needed, the data blocks from the queue with the lowest priority are replaced [13].

LRFU replacement policy employs both LRU and LFU replacement policies at the same time. LRFU calculates so-called CRF (Combined recency and Frequency) value for each data block. This value quantifies the likelihood that the block will be referenced in the near future [14]. LFRU is suitable for use and was tested in database systems [14].

LIRS replacement policy uses two sets of referenced blocks: High Inter-reference Recency (HIR) block set and Low Inter-reference Recency (LIR) block set. LIRS calculates the distance between the last two accesses to the data block and also stores a timestamp of last access to the data block. Based on this statistical information, the data are divided into either LIR or HIR blocks. When the cache is full, the data blocks from the LIR set are replaced. LIRS is suitable for use in virtual memory management [15].

FBR replacement policy uses the benefits of both LFU and LRU policies. FBR divides the cache into three segments: a new segment, a middle segment, and the old segment [16]. Data blocks are placed into sections based on their recency of usage. When a hit occurs, the hit counter is increased only for data blocks in the middle and old segment. When a replacement is needed, the policy chooses data block from the old segment with the least hit counts. [16]

ARC is similar to 2Q replacement policy. The ARC algorithm dynamically balances recency and frequency. It uses two LRU-queues. These queues maintain the entries of recently evicted data blocks [17]. ARC is simple to implement and has low computational overhead while performing well across varied workloads [11], [18].

CRASH is a low miss penalty replacement policy. It is developed for caching data block during reading data block from hard disk. CRASH puts data blocks with contiguous disk address into the same set. When replacement is needed, CRASH chooses the largest set and replaces the block with the minimum disk address from it [17]. The data blocks are stored in a priority queue based on the modifications [17].

III. The LFU-SS and LRFU-SS architecture

All mentioned caching algorithms were made mainly for low-level I/O operations. These algorithms usually work with the data blocks that have the same size. When replacement occurs, all mentioned caching policies choose the block to be removed from the cache based on statistics made during user's requests. Moreover, all the caching policies have to store statistical information for all data blocks in the cache.

In our approach, we will make and test new caching policy for use in mobile devices. Our first goal is to minimize costs of counting the priority of the data block in the cache. We must also take into account that the capacity of the mobile devices is limited. The speed of connection from the mobile device to the remote server can vary. Thus, our second goal is to increase cache hit ratio, and thereby decrease the network traffic.

We present an innovated LFU algorithm called Least Frequently Used with Server Statistics (LFU-SS), and a hybrid algorithm called Least Recently and Frequently Used with Server Statistics (LRFU-SS).

A. LFU-SS

In LFU-SS, we use server and local statistics for replacement decision. We will focus on the server statistics at first. The database module of the server maintains metadata for the files stored in the DFS. The metadata records contain items for storing statistics. These statistics are read and write hits per file, and global read hits for all files in the DFS. When a user reads a file from the DFS, the READ_HITS counter is increased, and sent to the user. When a user wants to write the file content, the WRITE_HITS counter is increased. Both of these counters are provided for each requested file. The GLOBAL_HITS counter is provided on demand.

Calculation of GLOBAL_HITS counter is time-consuming operation because of summation of the READ_HITS of all files. If we presume that the DFS stores thousands of files which are accessed by users, the value of variable GLOBAL_HITS is then much greater than value of variable READ_HITS, and we do not need to get value of GLOBAL_HITS for each file access. We can obtain this value periodically which will save server workload.

Basic caching unit in our approach is the whole file. By caching whole files, we do not need to store read or write

hits for each block of the file, we store these statistics for whole file. Storing whole files also bring another advantage – calculation of priorities for replacement is not computationally demanding because of relatively low number of units in the cache.

When LFU-SS replacement policy must mark file to be thrown out from the cache, LFU-SS works similarly as regular LFU. LFU-SS maintains metadata of files in a heap structure. In LFU-SS, we use binary min-heap. The file for replacement is stored in the root node. When a user reads a cached file, the local read hits counter is increased and the heap is reordered if necessary. The server statistics are only used for newly incoming files to the cache.

In a regular LFU policy, the read hits counter for a new file is initialized to one (the file has been read once). The idea of LFU-SS is that we firstly calculate the read hits counter from the statistics from the server. If the new file in the cache is frequently downloaded from the server, the file is then prioritized in comparison to a file which is not frequently read form the server. For computing initial read hits value, we use following formula:

$$READ\ HITS_{client} = \frac{READ\ HITS_{server} - WRITE\ HITS_{server}}{GLOBAL\ HITS_{server}} \cdot \cdot\ GLOBAL\ HITS_{client} + 1$$

We firstly calculate difference between read and write hits from the server. We prefer the files that have been read many times, and have not been written so often. Moreover, we penalize the files that are often written and no so often read. We do this because of maintaining data consistency of the cached files. The variable $GLOBAL\_HITS_{client}$ represents the sum of all read hits to the files in the cache. Finally we add 1 because the user wants to read this file. We must also store the read hits value as a decimal number for accuracy in comparison for sorting files in the heap. The pseudo-code for LFU-SS is depicted in Figure 2.

The disadvantage of using LFU-SS and general LFU is in ageing files in the cache. If the file was accessed for many times in the past, it still remains in the cache even if the file will not be accessed in the future again. We prevent this situation by division the $READ\_HITS_{client}$ by 2. When the value of variable $READ\_HITS_{client}$ reaches the threshold value, $READ\_HITS_{client}$ variables of all cached files are divided by 2. The threshold value was set to 15 read hits experimentally.

We will discuss time complexity of using LFU-FF now. As mentioned before, we use binary min-heap for storing metadata records. This heap is ordered by read hits count. For cached files in LFU-SS, we use three operations: inserting new file into cache, removing file from the cache, and updating file read hits. Let N be number of the cached files:

```
Input: request for file F
Initialization: heap of cached files records /*sorted by
cache hit's counts*/

if F is not in cache
{
  while cache is full {
       Remove file with the least read hits
       Reorder heap to be min-heap
  }
  Compute read_hits for file F
  Download file F into cache
  Insert metadata record to the heap
  Reorder heap to be min-heap
}
else
{
  Increase read_hits value of file F by 1
  if read_hits > threshold
  {
    for each file in cache do
    read_hits /= 2
  }
  Reorder heap if necessary

}
```

Figure 2. Pseudo-code for LFU-SS

- Operation inserting new file into cache has two steps: Insert file record into the heap with time complexity O(1), and reordering the heap structure with time complexity of O(logN). These time complexities are common for binary heap structures [19].
- Operation removing file record has time complexity of O(logN). We need to remove the record from the heap with the time complexity of O(1) and reorder the heap structure with complexity of O(logN).
- Operation updating file read hits has the time complexity of O(logN) in the worst case. The worst case occurs when the file is moved down from root to the leaf of the heap.

B. LRFU-SS

Next in our approach, we will use the LFU-SS in combination with standard LRU. As was introduced in other hybrid caching replacement policies, the combination of LRU and LFU brings increasing cache hit ratio. For combination of these caching policies, we will compute priority of LRU and LFU-SS for each file in the cache. The priority of LRU and LFU-SS is from interval 0 to 65535. Higher number means that the file is more suitable for storing in the cache. Formula for counting final priority of the file is following:

$$P_{final} = K_1 \cdot P_{LFU-SS} + K_2 \cdot P_{LRU}$$

In computing final priority, we can favour one of the caching policies by setting higher value for K_1 or K_2 constants. Impact of setting these constants is shown in section experimental results. We will focus on computing priority values for LFU-SS and LRU caching policies now.

1) P_{LFU-SS}

The priority value for the LFU-SS algorithm is calculated by using linear interpolation between the greatest and the lowest read hits values. Formula for counting this priority is following:

$$P_{LFU-SS}=(READ\ HITS_{file,client}-GLOBAL\ HITS_{minimum,client})\cdot \cdot \frac{65535}{GLOBAL\ HITS_{maximum,client}-GLOBAL\ HITS_{minimum,client}}$$

In this formula, the values of variables $GLOBAL\_MINIMUM\_HITS_{client}$ and $GLOBAL\_MAXIMUm\_HITS_{client}$ correspond to the greatest and lowest read hits value. In the case that the file is new in the cache, we calculate read hits by using formula from section LFU-SS. We suppose that a new file in the cache is fresh and will be also used in the future. Despite computing read hits for a new file in the cache by using server statistics, new files in the cache have still low read hits count. Therefore we calculate the P_{LFU-SS} for the new file in the cache in a different way. We use server statistics again. We calculate the first P_{LFU-SS} as follows:

$$P_{LFU-SS}=\frac{READ\ HITS_{server}}{GLOBAL\ HITS_{server}}\cdot 65535$$

2) P_{LRU}

Least recently used policy usually stores timestamp for last access to the file. If replacement is needed, the file that has not been accessed for the longest time period is discarded. In our approach, we need to calculate the priority from the timestamp. We do this as follows:

$$P_{LRU}=(T_{actual\ file}-T_{least recently\ file})\cdot \frac{65535}{T_{most recently\ file}-T_{least recently\ file}}$$

As shown in the formula, we use again linear interpolation for calculating P_{LRU}. We interpolate between the $T_{least\_recently\_file}$ and $T_{most\_recently\_file}$. $T_{least\_recently\_file}$ is the timestamp of the file that have not been accessed for the longest time period. $T_{most\_recently\_file}$ is the timestamp of the file that has been accessed last time.

Disadvantage of using LRFU-SS is in computation priorities. We need to recalculate priorities for all cached units every time one cached unit is requested. We also need to reorder the heap of the cached files because of changes of these priorities. We have got over this disadvantage by using whole files as caching units. By caching whole files, we do not have so many units in the cache in comparison to storing data blocks with the same size in the cache. This approach needs to store statistics of each of these blocks. The pseudo-code for the LRFU-SS is shown in Figure 3.

Similar to the LFU-FF, we will discus time complexity of using LRFU-SS. We use binary min-heap for storing metadata records of cached files. We also employ three operations to the cached files: inserting new file into cache, removing file from the cache, and accessing the file. Let N be number of the cached files:

- Operation inserting new file to the cache seems to have the time complexity of O(NlogN) in the worst case. Inserting new file invokes recalculating time priorities of all cached files with time complexity of O(N) and reordering the heap with the time complexity of O(logN). But we do not need to reorder the heap because we change P_{LRU} for each cached file. After recalculating new priorities, we insert new file into heap with complexity of O(logN). The final time complexity is O(N).
- Operation removing file record has time complexity of O(logN). We need to remove the record from the root of the heap with time complexity of O(1) and reorder the heap structure with time complexity of O(logN).
- Operation accessing the file seems to have the time complexity of O(NlogN) in the worst case. The worst case assumes that we calculate new P_{LRU} for each file, and than we need to reorder the whole heap. Changing of P_{LRU} does not affect the heap. The time complexity for recalculating P_{LRU} priorities is O(N). For accessed file, we need to recalculate P_{LFU-SS} priority, and reorder the heap. In the worst case, the reordering the heap has time complexity of O(logN). The final complexity is O(N) .

IV. Experimental results

In this section, we evaluate proposed algorithms. Recalling the introduction, we use KIV-DFS for storing and accessing files.

```
Input: Request for file F
Initialization:
Min-Heap of cached files /*ordered by priority*/,
K1, K2 /*constants for computing Pfinal*/

if F is not in cache
{
  while cache is full
  {
    Remove file with the least priority
    Reorder heap to be min-heap
  }
  Compute read hits for file F
  Compute initial PLFU-SS for file F
  Compute PLRU for file F
  Compute Pfinal := K1 * PLFU-SS + K2 * PLRU
  Download and Insert file F into cache
  Recalculate priorities of all files in the cache
      and simultaneously reorder the heap
}
else
{
  Increase read_hits value of file F by 1
  if read_hits > threshold
  {
    for each file in cache do
    read_hits /= 2
  }
  Store new timestamp for file F
  Recalculate priorities of all files in the cache
      and simultaneously reorder the heap

}
```

Figure 3. Pseudo-code for LRFU-SS

A. KIV-DFS environment

KID-DFS is a distributed file system which is being developed at the Department of Computer Science and Engineering, University of West Bohemia. Whole distributed file system consists of two main parts: server and client applications. System architecture is depicted in Figure 4.

1) KIV-DFS Client

The client module allows client to communicate with KIV-DFS servers, and to transfer data. The client applications exist in three main versions: standalone application, core module of operating system and Filesystem in Userspace (FUSE).

2) KIV-DFS Server

KIV-DFS Server consists of five modules: Authorization, Synchronization, VFS, Database, and File System. These modules can be run on different machines cooperating in DFS or on single machine. This increases the whole system scalability substantially. We briefly describe these five modules. KIV-DFS is deeply described in [20].

*Authorization Module***.** This module is an entry point to the system. It ensures authorization and secure communication with clients [20]. The communication channel is encrypted by using OpenSSL.

*Synchronization Module***.** The synchronization module is a crucial part of the whole system. Several clients can access the system via several nodes. Generally, different delays occur in delivering the messages. The KIV-DFS system uses Lamport's logical clocks for synchronization. Received messages are stored in a queue and get unique ID corresponding to the logical clock. The message is then sent to all nodes in the DFS. Every node maintains its own list with other nodes addresses. If the node receives a message with higher timestamp than local timestamp, the node processes the message and sends ACK back to the sender. On the other hand, if the local timestamp is higher than received, the local timestamp is sent back. The sender must in this case obtain the highest timestamp from all other nodes. This timestamp is increased by 1 and the message with new timestamp can be sent. The requests to the system are stored in a database.

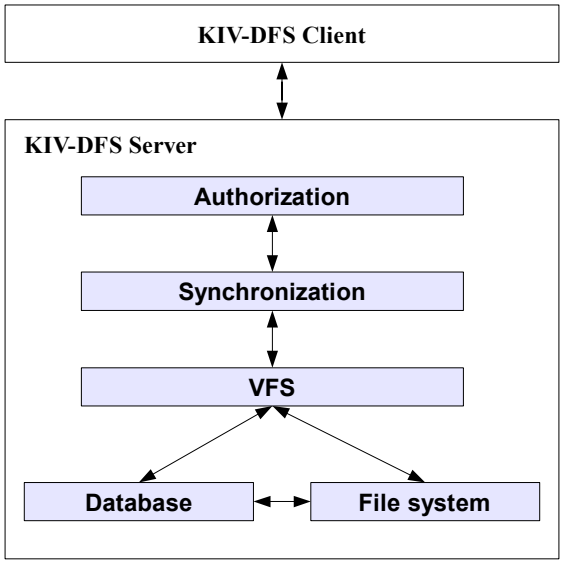

Figure 4. Model of KIV-DFS

Virtual File System Module (VFS). The VFS module hides the technology used for data and metadata storing. Based on the request, the module determines whether it is aimed at the metadata, e.g. to list the directory, create a new directory, or is aimed at the file access. Then, the request is send to the DB module or to the File System module.

File System Module. The File system module serves for storing file content on physical device like hard disks. It is utilized to work with the content of the files that the user works with. If the module obtains a request to store the data, it replies to the client with its IP address and a randomly selected port. On this port, the transfer will be realized. The client connects to this port and sends the file content. A randomly chosen port decreases the possibility of misuse.

The FS module also manages the data active replication. The FS module starts the replication of the file in the background. When the file operations are performed, the replicas are locked at the metadata level (Synchronization layer). This prevents a situation of simultaneous file access. The metadata record is unlocked after the file is stored. Similarly, when the replication is finished, the metadata record of replicas is unlocked. By using this approach, the KIV-DFS supports multi-RW replication.

Database Module. The Database module serves for communication with the database. The database stores metadata, the list of authorized users, and the client request queue. Metadata contain all information about files, such as names, the location in the directory structure, ACL information, size, and the physical location of the file.

The synchronization of the databases is solved at the synchronization level of KIV-DFS. It ensures the independence of the replication and synchronization mechanisms of different databases. The database is designed in a minimalistic way.

B. Experiments

To evaluate proposed policies, we performed simulation of remote file accesses. For the simulation, we have created 500 files with random size between 500KB and 5MB on the server side. The size of files respects the fact that mobile clients usually accesses smaller files from the remote storage.

The simulation was run on the wired client. We did not use mobile client because of acceleration of simulation. We have implemented RND, FIFO, LFU and LRU policies for comparison with our LFU-SS and LRFU-SS policies. For simulation of LRFU-SS, we have chosen coefficients K_1=1, K_2=1; K_1=1, K_2=2; K_1=2, K_2=1. While using K_1=1 and

K_2=1, we do not favour any of the caching algorithms. While using K_1=1 and K_2=2, we favour LRFU-SS over LRU, and while using K_1=2 and K_2=1, we favour LRFU-SS over LRU. We have chosen these coefficients experimentally.

In a simulation scenario, we have made 10,000 random requests on files where some of the files are prioritized and some other files are accessed less often. The prioritisation was made by a random number generator and a modulo function.

In the first experiment, we have observed the read hits count, and then we have computed read hit ratio. Read hits count represents the number of requests which have been served by the cache. The experiment used cache sizes from 8MB to 256MB. These cache sizes were chosen because of limited capacity of mobile clients. Table I summarizes cache read hit ratio, and Figure 3 depicts the cache read hits count for each of the implemented algorithms. For each simulated caching policy, we have had the same scenario of accessed files.

The best algorithm in this scenario is LFU-SS. While using LFU-SS with cache capacities of 16MB and 32MB, we can achieve up to 11% improvement over commonly used LRU or LFU caching policies. When we use cache with larger capacity (64, 128, and 256MB), the improvement is up to 4% in cache hit ratio.

On the other side, the cache read hits count deals only with the count of the files in the cache that were found in the cache. We use whole file as a basic caching unit. Hence, the policy with the best read hits count does not have to be the best caching policy in saving data traffic because of variable file size.

In the second experiment, we have observed the data traffic while using various cache algorithms. The total size of transferred files was 22,5GB. We have done the experiment with cache sizes from 8MB to 256MB again. Figure 4 shows, and Table II summarizes the bytes saved for different caching policies.

The best caching algorithm for cache sizes 8MB, 16MB, and 32MB is LFU-SS again. For larger cache capacity, the best caching policy LRFU-SS with K1=1, and K2=1. While using LRFU-SS with cache size of 265MB, we have saved up to half of the network traffic. LFU-SS achieves up to 8% of improvement over LRU in small cache sizes. LRFU-SS achieves up to 10% of improvement over LRU and LFU in larger cache capacities.

Table I. Cache Read Hit Ratio vs. Cache size

Read Hit Ratio [%]/ Caching Policy	Cache Size [MB]					
Caching policy	8	16	32	64	128	256
RND	2.98	5.68	10.36	16.03	25.46	40.39
FIFO	2,66	5,49	10,18	15,34	25,44	39,69
LFU	2,79	6,18	11,21	19,09	30,19	41,23
LRU	2,79	6,36	10,84	19,3	28,94	40,67
LFU-SS	6,55	13,05	21,68	23,64	31,47	42,47
LRFU-SS K_1=1;K_2=2	4,15	9,03	14	23,16	29,8	41,5
LRFU-SS K_1=1;K_2=1	2,83	7,28	12,2	22,07	30,58	41,91
LRFU-SS K_1=2;K_2=1	3,16	7,34	14,02	22,89	30,44	40,71

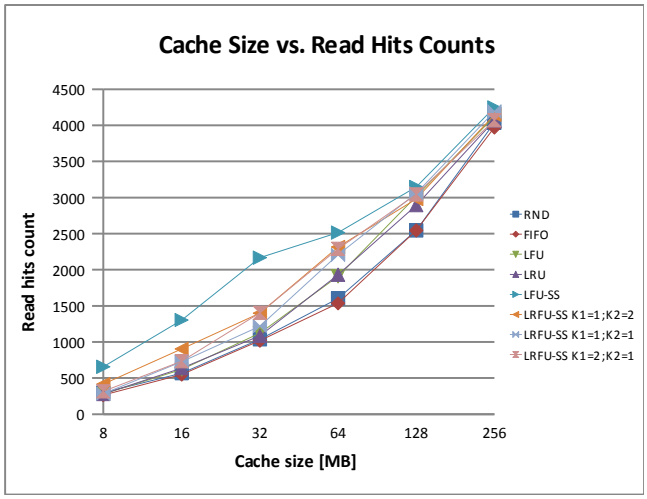

Figure 3. Cache Read Hits vs. Cache Size

Table II. Saved bytes vs. Cache size

Saved Bytes[MB] / Caching Policy	Cache Size [MB]					
Caching policy	8	16	32	64	128	256
RND	601	1204	2136	3383	5110	8671
FIFO	537	1155	2199	3307	4991	8161
LFU	458	1070	1995	4448	5741	8274
LRU	614	1440	2495	4252	5537	8205
LFU-SS	1605	3315	4040	4520	7574	10245
LRFU-SS K_1=1;K_2=2	708	1528	2577	5955	6432	10280
LRFU-SS K_1=1;K_2=1	822	2342	3745	5761	7798	10362
LRFU-SS K_1=2:K_2=1	639	2626	4648	5966	7718	8932

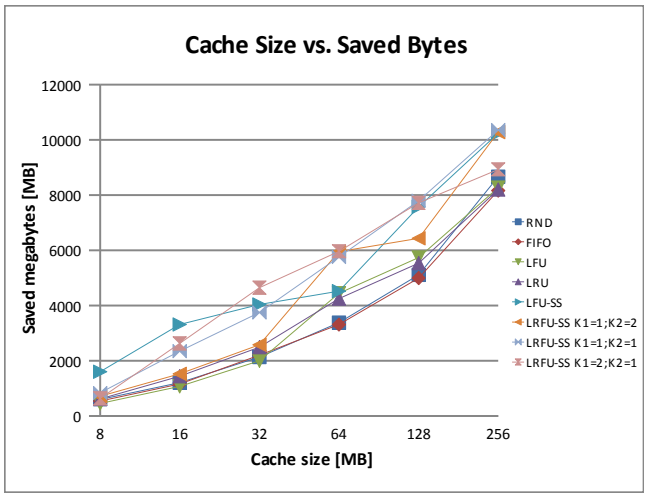

Figure 4. Cache Size vs. Saved Bytes

V. Further work

In our future work, we will implement cache and caching policy for iPhone, Android and Windows Mobile client applications.

Storing files in the user's cache may cause data inconsistency. The data on server can be modified while the user constantly works with the old files in the cache. In our future work, we will develop algorithms for maintaining data consistency for cached files.

In mobile devices, the connection to the server is not permanent. Not all of cellular networks providers have the coverage on the whole area where the user uses the device. For disconnected users, we will implement so-called offline operations. In this case, the user can still access cached files even after disconnection.

VI. Conclusion

This paper presented caching algorithms for caching files in mobile devices. Our goals in developing new caching algorithms were to decrease network traffic, and minimize costs of counting the priority of the data block in the cache. These two goals were set because of variable network connection of the mobile devices caused by moving of the user, and because of poor performance of the mobile devices.

The comparison of caching policies made in the section Experimental results shows that the introduced algorithms act better in comparison to commonly used caching policies like LRU and LFU. For smaller cache size, LFU-FF is suitable caching policy; for larger cache size, LRFU-SS is better choice.

For using in mobile devices, we have count the time complexity for both of the developed algorithms. In this case, the LFU-SS seems to be better algorithm. If we assume that the basic caching unit is the whole file, than both of the algorithms can be used in the mobile devices.

Acknowledgment

We thank Radek Strejc, Václav Steiner, and Jindřich Skupa, bachelor and master degree students, Department of Computer Science and Engineering, University of West Bohemia, for implementing and testing our concepts and ideas.

References

[1] Azzedine Boukerche and Raed Al-Shaikh, "Servers Reintegration in Disconnection-Resilient File Systems for Mobile Clients," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, Columbus, 2006, pp. 114-120.

[2] N. Michalakis and D.N. Kalofonos, "Designing an NFS-based mobile distributed file system for ephemeral sharing in proximity networks," in *Applications and Services in Wireless Networks, 2004. ASWN 2004. 2004 4th Workshop on*, 2005, pp. 225-231.

[3] A. Boukerche, R. Al-Shaikh, and B. Marleau, "Disconnection-resilient file system for mobile clients," in *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, Sydney, 2005, pp. 614-621.

[4] Nong Xiao, YingJie Zhao, Fang Liu, and ZhiGuang Chen, "Dual queues cache replacement algorithm based on sequentiality detection," in *SCIENCE CHINA INFORMATION SCIENCES*, 2012, pp. 191-199.

[5] K.W. Froese and R.B. Bunt, "The effect of client caching on file server workloads," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on*, Wailea, HI , USA, 1996, pp. 150-159.

[6] Benjamin Reed and Darrell D. E. Long, "Analysis of caching algorithms for distributed file systems," in *ACM SIGOPS Operating Systems Review, Volume 30 Issue 3*, New York, NY, USA, 1996, pp. 12-17.

[7] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine," *Commun. ACM*, vol. 12, no. 6, pp. 349-353, June 1969.

[8] R.L Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78-117, 1970.

[9] B. Whitehead, Chung-Horng Lung, A. Tapela, and G. Sivarajah, "Experiments of Large File Caching and Comparisons of Caching Algorithms," in *Network Computing and Applications, 2008. NCA '08. Seventh IEEE International Symposium on*, Cambridge, MA, 2008, pp. 244-248.

[10] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *SIGMOD '93 Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, New York, 1993, pp. 297-306.

[11] Nimrod Megiddo and Dharmendra S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *FAST '03 Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003, pp. 115-130.

[12] Theodore Johnson and Dennis Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *In VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 439-450.

[13] Yuanyuan Zhou, James F. Philbin, and Kai Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *In Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, 2001, pp. 91-104.

[14] Donghee Lee et al., "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," in *Computers, IEEE Transactions on*, 2001, pp. 1352 -1361.

[15] Song Jiang and Xiaodong Zhang, "LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, (SIIMETRICS'02)*, Marina Del Rey, 2002, pp. 31-42.

[16] A. Boukerche and R. Al-Shaikh, "Towards building a fault tolerant and conflict-free distributed file system for mobile clients," in *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 02, AINA 2006.*, Washington, DC, USA, 2006, pp. 405-412.

[17] Nong Xiao, YingJie Zhao, Fang Liu, and ZhiGuang Chen, "Dual queues cache replacement algorithm based on sequentiality detection," in *SCIENCE CHINA INFORMATION SCIENCES, Volume 55, Number 1, Research paper*, 2011, pp. 191-199.

[18] Woojoong Lee, Sejin Park, Baegjae Sung, and Chanik Park, "Improving Adaptive Replacement Cache (ARC) by Reuse Distance," in *9th USENIX Conference on File and Storage Technologies (FAST'11)*, San Jose, 2011, pp. 1-2.

[19] Thomas H Cormen, Charles E Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction To Algorithms*, 3rd ed.: MIT Press and McGraw-Hill, 2009.

[20] L. Matějka, L. Pešička, and J. Šafařík, "Distributed file system with online multi-master replicas," in *2nd Eastern european regional conference on the Engineering of computer based systems*, Los Alamitos, 2011, pp. 13-17.