Design and Implementation of a Caching Algorithm Applicable to Mobile Clients

Pavel Bžoch, Luboš Matějka, Ladislav Pešička and Jiří Šafařík
University of West Bohemia, Faculty of Applied Sciences
Department of Computer Science and Engineering
Univerzitní 8, 306 14 Plzeň, Czech Republic
E-mail: pbzoch@kiv.zcu.cz, lmatejka@kiv.zcu.cz, pesicka@kiv.zcu.cz, safarikj@kiv.zcu.cz

Keywords: mobile device, cache, caching policy

Received: October 24, 2012

Usage of mobile devices has grown over the past years. The term "mobile devices" covers many different kinds of devices (e.g. smart phones, cell phones, personal digital assistant (PDA), tablets, netbooks, etc.). A typical example that shows the growth of technologies is the smart phone. A Smart phone serves not only for voice calls and typing SMS, but can be used to access the internet and e-mails, play music and movies, and access remote storages. The Disadvantage of mobile devices is that they do not have a wired connection to the internet and thus the connection can vary. It can be fast while using WI-FI or the 3G mobile network or very slow using an old GRPS technology. 3G and other state-of-the-art technologies are not available everywhere. But users want to access their files as quickly and reliably as they can access them on a wired connection.
If data are demanded repeatedly, they can be stored on mobile devices in an intermediate component called a cache. However, the capacity of the cache is limited; thus we should store only the data that will probably be demanded again in the future. In this article, we present a caching algorithm which is based on client and server statistics. These statistics are used to predict a user's future behaviour.

Povzetek: Opisana je nova metoda za predpomnjenje pomnilniških naprav.

1 Introduction

Over the past years, more and more people can access the internet and produce data. The need of storing this data has also grown. Whether data are of multimedia types (e.g. images, audio, or video), text files, or are produced by scientific computation, they should be stored for sharing among users and further use. The data files can be stored on a local file system, on a remote file system or on a distributed file system.

A local file system (LFS) provides the data quickly compared to other solutions. On the other hand, LFS does not have enough capacity for storing a huge amount of data in general. LFS is also prone to failure. Because the data on LFS are usually not replicated, failure of the LFS usually causes more or less temporary loss of data accessibility, or even loss of data. Another disadvantage of LFS is that the local data cannot be accessed remotely.

A remote file system (RFS) provides the data remotely. RFS has otherwise the same disadvantages as LFS. It is prone to hardware failure. RFS is also hardly scalable. While using remote access, RFS has to use user authentication and authorization for preventing data stealing or corruption.

A Distributed file system (DFS) provides many advantages over a remote file system. These advantages are reliability, scalability, capacity, security, etc. Accessing files from mobile devices has to take into account changing communication channels caused by the user's movement. DFSs that are widely used were designed before mobile devices spread. Now, it is hard to develop mobile client applications and to implement algorithms for mobile devices into a DFS. None of the current DFSs, e.g. Andrew File System (AFS), Network File System (NFS), Coda, InterMezzo, BlueFS, CloudStore, GlusterFS, XtreemFS, dCache, MooseFS, Ceph and Google File System, has suitable clients for mobile devices [1] [2] [3].

Mobile devices have limited capacity for storing user content. They can store up to GBs of the data. Some of the devices can extend their capacity by using a memory card, but the capacity of these cards is also limited (usually to 32GB [4]). On the other hand, a DFS can store TBs of the data.

The speed of a wireless connection is low in comparison to a wired connection. The highest wireless speed is often limited by the use of the Fair User Policy (FUP) by the mobile connection provider. The FUP restricts the quantum of the downloaded data in a period of time [5]. In addition, the speed of a wireless connection can vary. The newest connection technologies are not available everywhere, but mobile users wish to access their data as fast as possible. So far, users download the same data repeatedly; we can use a cache to increase system performance. In this article, we will focus on use of the cache by mobile clients in a distributed file system

A cache is an intermediate component which stores data that can be potentially used in the future. While using a cache, the overall system performance is

improved. The cache is commonly used in database servers, web servers, file servers, storage servers, etc. [6]. However, cache capacity is not usually sufficient to store all requested content. When the cache is full, a system designer must adopt an algorithm which marks old content in the cache to be replaced. This algorithm implements replacement policy.

Cache functionality is depicted in Figure 1. The cache in the DFS can be on the client side as well as on the server side.

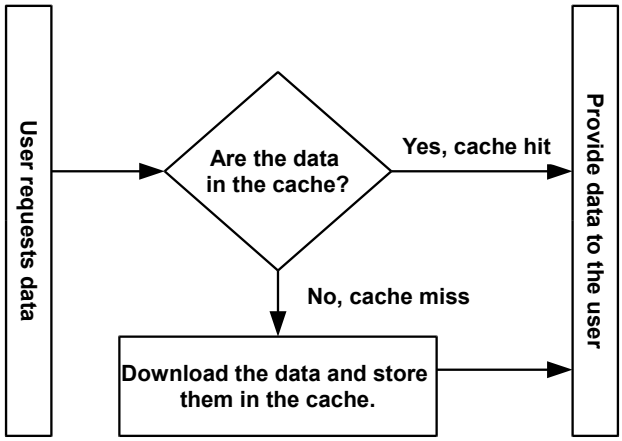

Figure 1: Cache.

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 the 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 all users' requests. Using a cache on the server and the client sides at the same time does not increase system performance. Increasing the cache hit ratio on the client side causes increasing the miss ratio on the server side and vice versa [7].

In section 2, we introduce cache policies commonly used. We discuss simple, statistics-based and hybrid caching algorithms.

We present a new caching replacement policy in section 3. We use client and server statistics in a manner which increases system performance. In section 4, we present results of performance analysis for the new algorithm. The results were acquired via simulation of user behaviour. As a remote storage for user files, we used KIVFS. KIVFS is a distributed file system which is being developed at the Department of Computer Science and Engineering, University of West Bohemia [8]. KIV is an acronym for the Czech name of our department (Katedra Informatiky a Výpočetní techniky). KIVFS is also designed to support mobile devices.

2 Overview of Caching Algorithms

We describe replacement policies which are commonly used in distributed file systems or in operating systems. Clearly, an optimal replacement policy replaces data whose next use will occur farthest in the future. However, this policy is not implementable. We cannot look into the future to get needed information about usage of the data. Hence, no implementable caching policy can be better than an optimal policy.

Caching policies can be divided into three categories: simple, statistics-based and hybrid policies.

2.1 Simple caching algorithms

Simple caching algorithms do not use any statistics or additional information. For replacement decisions, they usually employ other mechanisms. Examples of simple caching algorithms are Rand, FIFO, FIFO with 2nd chance, and Clock None of these caching policies takes user behaviour into account.

RAND. RAND or Random is a simple replacement policy which chooses data to be replaced based on random selection [9]. It is very easy to implement this replacement policy.

FIFO. First-In First-Out is another simple replacement policy. The data that are chosen to be replaced are the oldest in the cache. 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 into the cache, the data from the head of the queue are replaced [10].

FIFO with 2nd chance (FIFO2). First-In First-Out with second chance is a modification of the FIFO caching policy. FIFO2 stores the data units in a queue. In contrast to FIFO, FIFO2 stores a reference bit for each data unit in the queue. If a cache hit occurs, the reference bit is set to 1. When a replacement is needed, the oldest unit in the cache with a reference bit set to 0 is replaced and the reference bit of the older units is set to 0 at the same time [11].

CLOCK. The Clock replacement policy stores the data units in a circular buffer [12]. Clock stores a reference bit for each cached unit, and a pointer into the buffer structure. If a cache hit occurs, the reference bit of the requested data unit is set to 1. The data to be replaced are chosen by circularly browsing the buffer, and searching for the unit with a reference bit set to 0. The reference bit of each data unit with the reference bit set to 1 found during the browsing is reset to 0 [13] .

2.2 Statistics-based caching algorithms

Statistics-based algorithms employ statistical information about data in the cache: frequency of the accesses, and recency of the last use of data. Frequency is used by an LFU algorithm and recency by LRU and MRU algorithms.

LRU. The Least Recently Used replacement policy uses the temporal locality of the data [9]. Temporal locality means that the data units 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 [14]. According to the tests [15], 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 the LRU policy is that the data unit can be replaced even if the

unit was accessed periodically many times. In this case, the file will probably be requested in the near future again.

MRU. The Most Recently Used replacement policy works conversely to LRU. MRU replaces the most recently accessed data units. MRU is suitable for a file which is being repeatedly scanned in a looping sequential reference pattern [16].

LFU. The Least Frequently Used replacement policy replaces the data that have been used least. For each data unit there is a counter which is increased every time the data unit is accessed [9]. The disadvantage of this approach is that the data units in the cache that have been accessed many times in a short period of time remain in the cache, and cannot be replaced even if they will not be used in the future at all.

2.3 Hybrid caching algorithms

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

2Q replacement policy uses two queues. The first queue uses the FIFO replacement policy for data units and is used for data units that have been referenced only once. The second queue uses LRU as a replacement policy, and serves for so-called hot data units. Hot data units are units that have been accessed more than once. If a new data unit comes to the cache, it is stored in the FIFO-queue. When the same data unit is accessed for the second time, it is moved to the LRU-queue. The 2Q algorithm gives approximately 5% improvement in the hit ratio over LRU [17].

MQ replacement policy uses multiple LRU-queues. Every queue has its own priority. The data units with a lower hit count are stored in a lower priority queue. If the number of the hit count reaches the threshold value, the data unit is moved to the tail of a queue with a higher priority. When a replacement is needed, the data units from the queue with the lowest priority are replaced [18].

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. Data units are placed into sections based on their recency of usage. When a hit occurs, the hit counter is increased only for data units in the middle and old segments. When a replacement is needed, the policy chooses the data unit from the old segment with the smallest hit count [19].

LIRS replacement policy uses two sets of referenced units: the High Inter-reference Recency (HIR) unit set and the Low Inter-reference Recency (LIR) unit set. LIRS calculates the distance between the last two accesses to a data unit and also stores a timestamp of the last access to the data unit. Based on this statistical information, the data are divided into either LIR or HIR blocks. When the cache is full, the least recently used data unit from the LIR set is replaced. LIRS is suitable for use in virtual memory management [20].

LRFU replacement policy employs both LRU and LFU replacement policies at the same time. LRFU calculates the so-called CRF (Combined Recency and Frequency) value for each data unit. This value quantifies the likelihood that the unit will be referenced in the near future. LFRU is suitable for use and was tested in database systems [21].

LRD replacement policy replaces the data unit with the lowest reference density. Reference density is a reference frequency for a given reference interval. LRD has two variants of use. The first variant uses a reference interval which corresponds to the age of a page. The second variant uses constant interval time [22].

LRU-K replacement policy keeps the timestamps of the last K accesses to the data unit. When the cache is full, LRU-K counts so-called Backward K-Distance which leads the marked data unit to replace. The LRU-K algorithm is used in data base systems [23]. An example of LRU-K is **LRU-2**, which remembers the last two access timestamps for each data unit. It then replaces the data unit with the least recent penultimate reference [24].

ARC is similar to the 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 units [6]. ARC has low computational overhead while performing well across varied workloads [17], [25]. ARC requires units with the same size; thus it is not suitable for caching whole files.

CRASH is a low miss penalty replacement policy. It was developed for caching data blocks during reading data blocks from the hard disk. CRASH puts data blocks with contiguous disk addresses into the same set. When replacement is needed, CRASH chooses the largest set and replaces the block with the minimum disk address [6]. CRASH works with data blocks with the same size; thus CRASH is not suitable for caching blocks with different sizes.

3 The LFU-SS and LRFU-SS Architecture

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

We propose a new caching policy suitable for use in mobile devices. Our first goal is to minimize costs of counting the priority of data units in the cache. This goal was set because mobile device are not as powerful as personal computers ant their computational capacity is limited. The speed of data transfer from a remote server to the mobile device can vary. Thus, our second goal is to increase the cache hit ratio, and thereby decrease the network traffic caused by data transfer.

We present an innovated LFU algorithm we call Least Frequently Used with Server Statistics (LFU-SS), and a hybrid algorithm we call Least Recently and Frequently Used with Server Statistics (LRFU-SS) [26].

3.1 LFU-SS

In LFU-SS, we use server and client statistics for replacement decisions. We will consider server statistics 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 number of read and number of write hits per file, and number of global read hits for all files in the DFS. When a user reads a file from the DFS, the $READ\_HITS_{server}$ counter is increased, and sent to the user. When a user wants to write the file content, the $WRITE\_HITS_{server}$ counter is increased. Both of these counters are provided as metadata for each requested file. Calculation of the $GLOBAL\_HITS_{server}$ counter is a time-consuming operation because of summation of the $READ\_HITS_{server}$ of all files. If we presume that the DFS stores thousands of files which are accessed by users, the value of variable $GLOBAL\_HITS_{server}$ is then much greater than the value of variable $READ\_HITS_{server}$, and we do not need to get the value of $GLOBAL\_HITS_{server}$ for each file access. The value of the $GLOBAL\_HITS_{server}$ counter is computed periodically, thus saving server workload.

The 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 the whole file. Storing whole files also brings another advantage – calculation of priorities for replacement is not computationally demanding because of the relatively low number of units in the cache.

When the LFU-SS replacement policy must mark a file to be thrown out of the cache, LFU-SS works similarly to regular LFU. LFU-SS maintains metadata of files in a heap structure. In LFU-SS, we use a binary min-heap. The file for replacement is stored in the root node. When a user reads a cached file, the client read hits counter $READ\_HITS_{client}$ 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 from the server. For computing the initial read hits value, we use the following formula:

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

We first calculate the 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 not often read. We do this in order to maintain the data consistency of the cached files. The variable $GLOBAL\_HITS_{client}$ represents the sum of all read hits to the files in the cache. We add 1 because the user wants to read this file. We must store the read hits value as a decimal number for accuracy reasons when reordering files in the heap. The pseudo-code for LFU-SS is in Figure 2.

The disadvantage of using LFU-SS and general LFU relates to ageing files in the cache. If the file was accessed 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 dividing the variable $READ\_HITS_{client}$ 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.

```
Input: Request for file F
Initialization: heap of cached files
records /*ordered by client's cache read
hit 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 initial 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
  reorder heap if necessary
  upload client statistics to server
  if READ_HITS > threshold
  {
     for each FILE in cache do
     {
        FILE.READ_HITS = FILE.READ_HITS / 2
     }
  }
}
```

Figure 2: Pseudo-code for LFU-SS

Using statistics from the server for gaining better results in the cache read hit ratio causes a disadvantage in updating these statistics. If the accessed files are provided from the cache, the statistics are updated only on the client side, and are not sent back to the server. In this case, the server does not provide correct metadata, and the policy does not work correctly. A Similar case occurs while using a cache on the server and client sides simultaneously [7]. To prevent this phenomenon, the client application periodically sends local statistics back to the server. The update message contains file ids and number of requests per each file since the last update. We show the experimental results for LFU-SS with and

without uploading statistics to the server in the next section.

We will discuss the time complexity of using LFU-SS now. As mentioned before, we use a binary min-heap for storing metadata records. This heap is ordered the by read hits count. For cached files in LFU-SS, we use three operations: inserting a new file into the cache, removing a file from the cache, and updating file read hits. All these three operations are O(logN) [27].

3.2 LRFU-SS

Next, we will use LFU-SS in combination with standard LRU. As for mentioned hybrid caching replacement policies, the combination of LRU and LFU increases the cache hit ratio. For the combination of these caching policies, we will compute the priority of LRU and LFU-SS for each file in the cache. The priority of LRU and LFU-SS is from the interval (0, 65535], where a higher value represents a higher priority. The file with the lowest priority is replaced. The formula for counting the final priority of the file is the following:

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

In computing the final priority, we can favour one of the caching policies by setting a higher value for K_1 or K_2 constants. The impact of setting these constants is shown in section 4. Next, we will focus on computing priority values for LFU-SS and LRU caching policies.

3.2.1 P_{LFU-SS}

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

$$P_{LFU-SS} = \frac{(READ\ HITS_{file} - GLOBAL\ HITS_{min})_{client}}{(GLOBAL\ HITS_{max} - GLOBAL\ HITS_{min})_{client}} \cdot 65535$$

In this formula, the values of variables $GLOBAL\_HITS_{min,client}$ and $GLOBAL\_HIT_{max,client}$ correspond to the highest and lowest read hits values. In the case that the file is new in the cache, we calculate read hits by using the formula from the previous section. We can expect that a new file in the cache is fresh and will also be used in the future. Despite computing read hits for a new file in the cache by using server statistics, new files in the cache still have a 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 and calculate the first P_{LFU-SS} as follows:

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

3.2.2 P_{LRU}

The least recently used policy usually stores the timestamp for last access to the file. If a 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:

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

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

The disadvantage of using LRFU-SS relates to 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 in these priorities. By caching whole files, we do not have many units in the cache, so these calculations are acceptable. The pseudo-code for the LRFU-SS is in Figure 3.

```
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
  upload client statistics to server
  if READ_HITS > THRESHOLD
  {
    for each FILE in cache do
    {
      FILE.READ_HITS = FILE.READ_HITS / 2
    }
  }
  store new timestamp for file F
  recalculate priorities of all files in
    the  cache and reorder the heap
}
```

Figure 3: Pseudo-code for LRFU-SS.

The LRFU-SS policy uses server statistics like LFU-SS. Using LRFU-SS causes the same problem with updating access statistics on the server side. We will solve this problem by periodically sending update messages back to the server. We show the experimental

results for LRFU-SS with and without uploading statistics to the server in the next section.

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

The operation inserting a file entails recalculating time priorities of all cached files, which takes O(N) time. New priorities do not affect the heap structure because the recalculation maintains the min-heap property. After recalculating new priorities, we insert a new file into the heap, which is O(logN). Then, insertion of a new file is O(N). The operation removing a file is O(logN) again. The operation accessing a file has the time complexity of O(N). As with inserting a new file, we need to recalculate priorities of all files taking O(N) time. For an accessed file, we need to recalculate the $P_{LFU\text{-}SS}$ priority and min-heapify the accessed file, which is O(logN). So, accessing a file takes O(N) time.

4 Performance Evaluation

In this section, we evaluate the proposed algorithms and compare them to other caching algorithms. We carried out two types of test. The first series of tests was performed using a cache simulator. The second series of tests ran on a wired client that was connected to the KIVFS used for storing and accessing files, thus mimicking a mobile device connection to the server.

We created 500 files with uniformly distributed random size between 1KB and 5MB on the server side. This distribution is based on analysis of the log from a local AFS cell server. We monitored the AFS cell for a month. In this period of time, users have nearly 930,000 requests to the files. The most of accessed (over 98%) files are from (0-5MB] in size

The number of requests to the files is not equal. We observed in the AFS log that some files are requested more often than other files. Accesses to the files are simulated by using a Gaussian random generator which corresponds to the observations gained from the log.

We evaluated the performance of LFU-SS and LRFU-SS algorithms on cache sizes ranging from 8MB to 512MB, reflecting the limited capacity of mobile devices.

We used the cache hit ratio and data transfer decrease needed to transfer the files as performance indicators.

4.1 Cache simulator

A cache simulator was developed to prevent the main disadvantage of testing caching policies in a real environment, which lies in the fact that it takes a long period of time to test caching algorithms. This is caused by the communication over a computer network.

The cache simulator consists of three parts: Server, Client and Request generator.

Server represents storage of files collection. Each file is represented by a unique ID and size in bytes. Additionally, the server stores a number of read and writes requests for each file. When a client demands a file, all the metadata are provided.

Client is an entity which requests files from the server and uses the evaluated caching algorithm. During the simulation, the client receives requests for file access from the Requests generator. The client increases the counter of requested bytes by the size of the file and looks into its cache for a possible cache hit. If the file is found in the cache, the number of cache read hits is increased. If the file is not in the cache, the file is downloaded from the server and stored in the cache. At the same time, the counter maintaining the number of transferred bytes is increased by the size of the requested file.

Requests generator is an entity which knows the files' ID from a server, and generates requests for these files. We used a Gaussian random generator for a simulation with parameters based on the AFS log mentioned above.

4.2 KIVFS environment

The KIVFS distributed file system consists of two main parts: server and client applications. The System architecture is depicted in Figure 4.

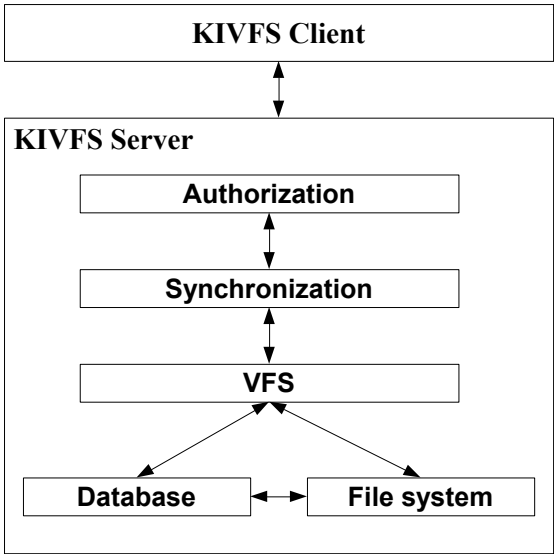

Figure 4: Model of KIVFS.

4.2.1 KIVFS client

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

4.2.2 KIVFS server

The KIVFS Server consists of five modules: Authorization, Synchronization, VFS, Database, and File System. These modules can be run on different machines cooperating in a DFS or on a single machine. We briefly

describe these five modules. KIVFS is described in [8] in more detail.

*Authorization Module***.** This module is an entry point to the system. It ensures authorization and secure communication with clients [8]. 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 KIVFS system uses Lamport's logical clocks for synchronization. Every received message gets a unique ID corresponding to the logical clock. The synchronisation is based on this unique ID, which also serves as a timestamp. This ID is also used for synchronisation among nodes.

Virtual File System Module (VFS). The VFS module hides the technology used for data and metadata storage. Based on the request, the module determines whether it is aimed at the metadata, or is aimed at file access.

File System Module (FS). The File system module stores file content on physical devices like a hard disk. It is utilized to work with the content of the files that the user works with.

The FS module also manages the active data replication. The FS module starts the replication of the file in the background. The replication process cooperates with the synchronization layer.

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.

The synchronization of the databases is solved at the synchronization level of KIVFS. It ensures the independence of the replication and synchronization mechanisms of different databases.

4.3 Evaluation

In this subsection, we give the results of the simulations. The first simulation of the caching algorithms' behaviour used the Cache Simulator; the second one ran on a wired client.

4.3.1 Simulation using the cache simulator

We implemented all caching policies mentioned in Section 2 in this simulation. Coefficients were set to K_1=0.35, K_2=1.1 for LRFU-SS. We chose these coefficients after a series of experiments with LRFU-SS. In experiments, we simulated LFRU-SS and LFU-SS with and without sending client statistics back to the server, to demonstrate the effect of sending client statistics.

In the experiments, we generated 100,000 requests on files. The cache read hit ratio is shown in Table 1. Table 2 shows the data transfer decrease. The total size of transferred files was 247.5GB, which is also the number of bytes transferred without usage of a cache.

Read Hit Ratio [%] / Cache Policy	**Cache Size [MB]**						
Caching policy	**8**	**16**	**32**	**64**	**128**	**256**	**512**
2Q	1.77	4.26	9.07	18.23	35.51	64.26	95.27
Clock	1.48	3.32	6.85	13.78	27.26	52.62	90.43
FBR	2.85	7.71	14.45	22.27	36.32	66.48	93.51
FIFO	1.48	3.32	6.84	13.70	26.87	51.13	85.09
FiFO 2nd	1.48	3.32	6.92	14.00	27.96	54.49	92.28
LFU	3.61	7.17	11.41	17.31	34.44	63.22	95.38
LFU-SS	**3.74**	**7.84**	**13.48**	**22.25**	**38.55**	**65.93**	**94.21**
LFU-SS without sending statistics	**2.26**	**6.00**	**8.06**	**13.50**	**21.96**	**36.21**	**61.71**
LIRS	1.93	3.98	7.69	15.48	29.91	56.90	92.83
LRDv1	1.48	3.31	6.90	14.02	28.06	56.11	93.80
LRDv2	1.48	3.31	6.91	13.87	27.42	53.05	89.83
LRFU	1.94	4.05	8.60	18.18	35.19	64.65	93.89
LRFU-SS	**2.98**	**4.63**	**10.15**	**19.77**	**37.72**	**66.67**	**94.48**
LRFU-SS without sending statistics	**0.95**	**2.35**	**5.50**	**8.61**	**18.51**	**37.22**	**65.36**
LRU	1.48	3.32	6.88	13.82	27.55	53.48	91.68
LRU-K	1.48	3.39	8.07	17.08	34.10	64.15	95.23
MQ	1.79	3.78	7.59	14.95	29.38	55.49	92.06
MRU	1.35	2.32	4.27	7.61	14.22	29.94	57.55
RND	1.53	3.26	6.90	13.74	26.86	50.90	85.27

Table 1: Cache Read Hit Ratio vs. Cache Size Using Cache Simulator.

Without sending these statistics, both LFU-SS and LRFU-SS have significantly worse results than the other caching policies. This situation shows both indicators, cache hit ratio and data transfer decrease.

LFU-SS with sending local statistics back to the server has the best results in terms of the cache hit ratio. The second-best is the FBR policy. Recall that we use the whole file as the caching unit. Hence, the policy with the best read hits ratio is not necessarily the best one in decreasing data transfer, which is obviously caused by the variety of file size. Although FBR has good results in terms of the cache hit ratio, it has worse results in decreasing network traffic. LFU-SS has the best result in decreasing network traffic for cache sizes from 8MB to 128MB. For higher cache sizes, LRFU-FF is a better choice.

Overall, results in this experiment show that LFU-SS achieves up to 2% improvement in saving network traffic in smaller cache sizes over other caching policies. LRFU-SS achieves up to 1% improvement in higher cache sizes.

Data Transfer Decrease [GB] / Cache Policy	Cache Size [MB]						
Caching policy	8	16	32	64	128	256	512
2Q	244.20	238.52	227.93	206.48	164.15	89.86	12.12
Clock	244.08	239.48	230.74	213.57	179.73	116.76	23.14
FBR	244.65	242.78	228.18	206.35	169.36	91.02	16.64
FIFO	244.08	239.48	230.60	212.97	178.15	111.61	18.54
FiFO 2nd	244.08	239.48	230.77	213.77	180.77	120.43	36.54
LFU	243.42	238.16	229.34	210.16	170.23	98.76	12.57
LFU-SS	**243.32**	**237.73**	**225.75**	**202.86**	**156.81**	**83.29**	**13.82**
LFU-SS without sending statistics	**245.10**	**242.34**	**236.92**	**224.94**	**201.61**	**161.67**	**98.56**
LIRS	243.82	239.01	229.46	209.95	173.72	105.76	17.09
LRDv1	244.08	239.50	230.67	212.91	177.89	107.90	14.90
LRDv2	244.07	239.50	230.59	213.29	179.41	115.57	24.62
LRFU	243.79	238.88	228.40	206.98	165.56	90.31	14.64
LRFU-SS	**243.44**	**238.72**	**226.91**	**203.06**	**158.67**	**81.79**	**13.38**
LRFU-SS without sending statistics	**245.64**	**242.69**	**237.30**	**225.89**	**202.18**	**158.99**	**91.84**
LRU	244.08	239.48	230.69	213.40	179.11	114.49	19.97
LRU-K	244.08	239.42	229.14	207.99	166.00	90.71	12.36
MQ	243.93	238.89	229.30	210.92	174.79	109.10	19.09
MRU	243.98	239.60	230.65	213.58	180.86	121.04	36.13
RND	244.47	242.02	237.23	229.06	212.48	173.45	99.20

Table 2: Data Transfer Decrease vs. Cache Size Using Cache Simulator.

4.3.2 Simulation on a wired client

The second simulation ran on a wired client to accelerate the experiments. Because of high time consumption of the experiments, we implemented only RND, FIFO, LFU and LRU policies for comparison with LFU-SS and LRFU-SS policies.

For LRFU-SS, we choose the same coefficients as in the first simulation. In the simulation scenario, we generated 10,000 requests. Table 3 summarizes the cache read hit ratio for each of the implemented algorithms.

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 a cache with a larger capacity (64, 128, 256, and 512MB), the improvement is up to 4% in the cache hit ratio.

Again, the policy with the best read hits ratio is not necessarily the best one in decreasing data traffic.

ReadHit Ratio [%]/ Caching Policy	Cache Size [MB]						
Caching Policy	8	16	32	64	128	256	512
RND	2.98	5.68	10.36	16.03	25.46	40.39	62.34
FIFO	2.66	5.49	10.18	15.34	25.44	39.69	60.23
LFU	2.79	6.18	11.21	19.09	30.19	41.23	63.87
LRU	2.79	6.36	10.84	19.3	28.94	40.67	63.54
LFU-SS	**6.55**	**13.05**	**21.68**	**25.14**	**31.47**	**42.47**	**64.23**
LFU-SS without sending client statistics	**2.56**	**5.48**	**11.02**	**18.52**	**28.56**	**35.25**	**55.45**
LRFU-SS	**4.5**	**10.03**	**15.22**	**23.76**	**30.8**	**41.9**	**64.14**
LRFU-SS without sending client statistics	**2.48**	**5.1**	**10.54**	**18.65**	**29.15**	**36.82**	**56.75**

Table 3: Cache Read Hit Ratio vs. Cache Size on Wired Client.

Next, we measured the data transfer decrease. The total size of transferred files was 22,5GB. Table 4 summarizes the data transfer decrease for different caching policies.

Data Transfer Decrease [GB] / Cache Policy	Cache Size [MB]						
Caching policy	8	16	32	64	128	256	512
RND	21.97	21.38	20.47	19.25	17.57	14.09	8.63
FIFO	22.03	21.43	20.41	19.33	17.68	14.59	9.19
LFU	22.11	21.51	20.61	18.21	16.95	14.48	8.36
LRU	21.96	21.15	20.12	18.41	17.15	14.55	8.70
LFU-SS	**20.99**	**19.32**	**18.61**	**18.14**	**15.16**	**12.55**	**7.95**
LFU-SS without sending client statistics	**21.97**	**21.07**	**19.94**	**18.30**	**16.58**	**14.48**	**9.90**
LRFU-SS	**21.74**	**20.27**	**18.90**	**16.93**	**14.94**	**12.44**	**7.88**
LRFU-SS without sending client statistics	**21.99**	**21.24**	**20.07**	**18.30**	**16.19**	**14.08**	**9.33**

Table 4: Data Transfer Decrease vs. Cache Size on Wired Client.

The best caching algorithm for cache sizes 8MB, 16MB, and 32MB is LFU-SS again. For larger cache capacity, the best caching policy is LRFU-SS. While using LRFU-SS with a cache size of 512MB, we saved up 65% of the network traffic. LFU-SS achieves up to 8% improvement over LRU in small cache sizes. LRFU-SS achieves up to 3% improvement over LRU and LFU in larger cache capacities.

5 Further Work

In our future work, we will add direct generation of the file requests from the AFS log file to the cache simulator. The simulator will then allow the simulation of more real situations.

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

6 Conclusion

This article presented caching algorithms for caching files in mobile devices. Our goals in developing new caching algorithms were to decrease network traffic, and minimize the cost of counting the priority of the data unit in the cache. These two goals were set because of the varying network connection quality of mobile devices caused by the movement of the user, and because of the limited performance of the mobile devices.

The comparison of caching policies proved that the algorithms introduced perform better in comparison to other caching policies except in one case. For smaller cache sizes, LFU-FF is a suitable caching policy; for larger cache sizes, LRFU-SS is a better choice.

Considering time consumption, LFU-SS is the asymptotically better algorithm. When caching whole files, both algorithms introduced are suitable for mobile devices.

Acknowledgement

This work is supported by the Ministry of Education, Youth, and Sport of the Czech Republic – University spec. research – 1311. We thank Radek Strejc, Václav Steiner, and Jindřich Skupa for implementing and testing proposed concepts and ideas.

References

[1] 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.

[2] A. Boukerche and R. 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.

[3] N. Michalakis and D. 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.

[4] K. T. Corporation, "microSD Cards | Kingston," Kingston Technology Corporation, 2012. [Online]. Available: http://www.kingston.com /us/flash/microsd_cards#sdc10. [Accessed 10 10 2012].

[5] M. Chetty, R. Banks, A. Brush, J. Donner and R. Grinter, "You're capped: understanding the effects of bandwidth caps on broadband use in the home," in *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, Austin, Texas, USA, 2012.

[6] N. Xiao, Y. Zhao, F. Liu and Z. Chen, "Dual queues cache replacement algorithm based on sequentiality detection," in *SCIENCE CHINA INFORMATION SCIENCES, Volume 55, Number 1, Research paper*, 2011.

[7] K. Froese and R. 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.

[8] 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.

[9] B. Reed and D. 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.

[10] 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.

[11] R. P. Draves, "Page Replacement and Reference Bit Emulation in Mach," in *In Proceedings of the Usenix Mach Symposium*, 1991.

[12] P. Steven W. Smith, "Digital Signal Processors - Circular Buffering," in *The Scientist and Engineer's Guide to Digital Signal Processing*, San Diego, California Technical Publishing, 1998, pp. 506-509.

[13] S. Jiang, F. Chen and X. Zhang, "CLOCK-Pro: an effective improvement of the CLOCK replacement," in *ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, 2005.

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

[15] B. Whitehead, C.-H. 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.

[16] H.-T. Chou and D. J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *VLDB '85 Proceedings of the 11th international conference on Very Large Data Bases - Volume 11*, 1985.

[17] T. Johnson and D. 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.

[18] Y. Zhou, J. F. Philbin and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *In Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, 2001.

[19] 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.

[20] S. Jiang and X. 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.

[21] D. Lee, J. Choi, J.-H. Kim, S. Noh, S. L. Min, Y. Cho and C. S. Kim, "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," in *Computers, IEEE Transactions on*, 2001.

[22] W. Effelsberg and T. Haerder, "Principles of database buffer management," in *Journal ACM Transactions on Database Systems (TODS) Volume 9 Issue 4, Dec. 1984*, New York, 1984.

[23] E. J. O'Neil, P. E. O'Neil and G. 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.

[24] N. Megiddo and D. 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.

[25] W. Lee, S. Park, B. Sung and C. Park, "Improving Adaptive Replacement Cache (ARC) by Reuse Distance," in *9th USENIX Conference on File and Storage Technologies (FAST'11)*, San Jose, 2011.

[26] P. Bžoch, L. Matějka, L. Pešička and J. Šafařík, "Towards Caching Algorithm Applicable to Mobile Clients," in *Federated Conference on Computer Science and Information Systems (FedCSIS), 2012*, Wroclaw, 2012.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction To Algorithms, 3rd ed., MIT Press and McGraw-Hill, 2009.