Contemporary Engineering Sciences, Vol. 9, 2016, no. 30, 1463 - 1478
HIKARI Ltd, www.m-hikari.com
http://dx.doi.org/10.12988/ces.2016.68129

Failure Resilient Data Placement Policies for Distributed Storages

Lyudmila Ivanichkina

OOO Proekt IKS, Altufievskoe sh, 44, 127566, Moscow, Russia
MIPT, 9 Institutskiy per., 141700, Dolgoprudny, Moscow Region, Russia

Kirill Korotaev

OOO Acronis, Altufievskoe sh, 44, 127566, Moscow, Russia

Andrew Neporada

OOO Proekt IKS, Altufievskoe sh, 44, 127566, Moscow, Russia

Abstract

This paper is concerned with failure resilient data placement policies in super large data storages. We consider data storage architecture, the major factors contributing to data corruption, as well as the means to reduce corresponding data losses. The developed data placement algorithms estimate the mean time to data loss (MTTDL) via statistical simulation of failures and recoveries. We also study various dependencies of storage reliability on data placement policies.

Keywords: Distributed data storage, mean time to data loss, data placement policies, disk failure, latent sector errors, failure domains, locality groups, numerical simulation, Monte-Carlo method, discrete event modeling

1 Introduction

Decades of independent observations confirm that the quantity of global digital data increases exponentially. The total amount of data was estimated at 0.13 zettabytes in 2005, 1.227 zettabytes in 2010, about 2.8 zettabytes in 2012 and is predicted to reach approximately 40 zettabytes in 2020 [1].

Nowadays, the technological advancement of civilization relies on a stack of accumulated information in a form of various datasets. Each set of data has its own cost formed by corresponding acquisition expenses. One can measure expenses in two ways. The usual commercial way is to denominate cost in money. However, this method fails to account for time span required to obtain data. Large scientific projects can last for decades with no acceleration gained through any excessive funding. Thus, the applied academic enterprises are more clearly expressed in person-hours or even in pure hours. It also means that every data loss always results in lost time. For example, about 600 thousand person-hours constitute one human life. The productive part of it is shorter and equals 100–200 thousand hours [2]. Therefore, one can consider the overall impact to the large project as a quasi-loss of hundreds and thousands of human lives.

The modern storages of petabyte capacity consist of thousands of disks that are statistically exposed to failures on a regular basis. In order to preserve deposited data the computer systems rely on redundancy either in the form of replication schemes either in the form of erasure correction codes. The first option arranges the copies of data block on multiple servers, correspondingly increasing allocated capacity many-fold. The second option partitions each data block into sequence of fragments, computes additional parity fragments via erasure codes and distributes the result across various servers [3]. The second approach to data reliability requires less capacity but demands computational power to carry out encoding/decoding procedures. Generally, the failure and recovery processes are quite complex, so the analytical Markovian solutions for system reliability imply strict assumptions and have very limited applicability. Therefore, the common and more practical way to estimate reliability of a real-like complex storage is to apply numerical methods for stochastic simulation. We propose mathematical model allowing one to estimate reliability of a distributed computer system and reveal its dependency on various patterns of fragment placement across the storage nodes.

2 Big Data Loss Prevention

2.1 Big Data Loss Prevention

The main reliability indicator for distributed storages is the mean time to data loss (MTTDL) showing the duration of storage lifecycle until some data become irrecoverable. The approximate expression for storage MTTDL is derived from classical analytical model based on Markov chains [4]

$$MTTDL_{Storage} = \frac{MTTDL_{Block}}{N_{Blocks}},$$

where N_{Blocks} is a total number of data blocks in a storage and $MTTDL_{Block}$ is a mean lifecycle of a data block. In turn, one can estimate the number of data blocks and the block MTTDL as

$$N_{Blocks} = \frac{\omega C}{S_{Block}},$$

$$MTTDL_{Block} = \frac{\mu^{n-k}}{\lambda^{n-k+1} \prod_{l=0}^{n-k} (n-l)},$$

where C is the maximum allocation size of useful data, ω is the coefficient of storage space utilization, S_{Block} is a block size defined by the storage architecture, μ and λ are the respective rates of disk failure and data recovery from a failed disk, n and k are the parameters of applied erasure codes.

As a rule, the storage operational lifetime should greatly exceed mean time of data retention of several decades. So, the typical MTTDL exceeds hundreds of years to keep instant data loss probability at sufficiently low levels. This places emphasis on one of the distinctive characteristics of a Big Data storage, namely the practical impossibility of experimental physical measurements of its real MTTDL.

The Markovian model is obtained by neglecting small terms and is applicable only under assumption that data loss events for different data blocks of the system are mutually independent. The expressions provided above show, that the analytical models approximate the reliability of a single block in the first place. On the contrary, we implement and use the simulation model that yields estimations for a whole storage.

2.2 Classification of data errors

The frequency of irrecoverable read errors is a random variable, which is calculated by means of statistical analysis of large samples from produced disk models. This stochastic property is declared in HDD data sheets provided by the manufacturer and is about one error per 10^{14} bits or one error per 11

TB of data read from disk. The proposed model considers how often such errors occur in a data storage lifetime. The frequency estimation is based on a mean volume of data read from a single HDD at the datacenter during typical timeframe.

Averaged HDD access rate can be obtained from averaged HDD workload rate specified in the respective device manual. The expected drive workload of about 20 per cent means that the disk is accessed 20 per cent of uptime and is idle for the remaining 80 per cent. If the specified [5] maximum sustained transfer rate equals 140 Mbps then the size of annually read data is 862312 GB or approximately 842 TB. Assuming that, one can approximate the value of irrecoverable bit errors frequency for one drive at 77 errors per year. Let the disk capacity be 2 TB, data fill ratio be 50 per cent, and the typical storage segment size be equal to 50 MB. According to this, the mean error frequency while reading specified segment is about 0.0367 errors per year or one error within 272 years. This frequency equals to $(272\ years)^{-1}$, that is comparable to disk failure rate of about $(23\ years)^{-1}$ for mean time to failure (MTTF) of 200000 hours. Thus, the storage MTTDL estimations should include this parameter to achieve higher fidelity.

2.3 Simulation Model

The developed and implemented discrete event model simulates extensive set of cases occurring in real data storages. We deny the simple storage state representation as two sets of undistinguishable operational and undistinguishable failed disks in favor to the list of all storage drives. The transitions between system states are defined not by ratio of elements in corresponding sets, but by the ensemble of events, taking place at every disk. Each simulated drive can manifest two events, namely: the failure and the disk data recovery. The events possible for a block include latent sector error within the block fragment and recovery of the given block fragment after checksum validation. The statistical distributions and failure rates, the processes of data validation and recovery are the input parameters for the model.

The initial conditions for a problem of statistical modeling of reliability correspond to some distribution of block fragments across the storage disks. The program implementation of computational algorithm permits to simulate various policies of fragment placement, such as fully random distribution or distribution within a group of n disks. This explicit fragment distribution can increase the accuracy of simulation modeling by accounting data migration dynamics within a storage. The proposed simulation model is an extended version of the one we used in paper [6]. The new improvements support storage-specific patterns of data placement like failure domains and locality groups considered in current paper.

The simulation model accounts for the storage-implemented size of the data block. According to the system functionality, the storage precedes the writing of every file by splitting it into fixed-size data blocks and applying erasure codes to these blocks. The resulting n data fragments with k of them sufficient for recovery are to be written on different storage disks.

The model also considers the following parameters:

- A number of disks in a simulated storage.
- A disk capacity, mean filling of disks in a storage.
- Parameters of used erasure coding.
- Disk MTTF.
- Mean time of failed disk replacement and recovery of contained data.
- Mean time to bit error occurrence.
- Mean time of checksum verification for all data in a storage and correction of detected bit errors.
- Pattern and parameters of data placement across disks in a storage.

The pattern of data placement describes what storage disks will hold fragments of each data block. We implemented the following placement patterns:

- Random pattern, where n random disks are selected from the set of all disks in a storage.
- Disk groups, where the set of all disks is split into n-ary groups, one of which is selected to store fragments of block.
- Disk clusters, where the set of all disks is split into N-ary groups ($N > n$), one of which is selected to store fragments of block on its n randomly sampled disks. The number of disks in a cluster acts as an input parameter.
- Failure domains, where the set of all disks in a storage is split into groups such way that no group contains more than one fragment of each block. The number of disks in a failure domain acts as an input parameter.
- Failure domains, where the set of all disks in a storage is split into groups such way that every group contains no greater than one fragment of each block.

The simulator, designed via OOP paradigm as a library on a Python language, implements the algorithmic model of distributed data storage with corresponding processes of failure and recovery. To use a program one should import the library, create new storage instance with supplied parameters and run series of iterative simulations with inner cycle break on irrecoverable data loss. The main program cycle contains pseudorandom generators, simulating the events of disk failures, bit sector errors for blocks and data recovery after failures, according to respective statistical distributions. Each iteration proceeds as follows:

- Determine event that is nearest to the current moment of simulation time.
- Update dependent data structures (e.g. add disk to the list of failed drives).
- Check for irrecoverable data loss if the latest event was a failure.
- Generate new failure or recovery event.

The returned results consist of simulation time intervals to irrecoverable data loss event for each inner iterative cycle and the sought-for storage MTTDL computed on these intervals.

2.4 Data Placement Policies

2.4.1 Failure Domains

The part of computer network or system, all components of which can be affected by some hardware or network failure is called failure domain. In case of inaccessibility of some failure domain, all its disks become unavailable. The elements of distributed architecture like singular servers as well as singular racks or rows of racks in datacenter.

Accounting for failure domains as specific properties of the datacenter in algorithms of data placement allows one to improve data availability during failure that affects all components of a failure domain. To achieve this, the fragments of every data block are distributed in the way that no failure domain contains more than one fragment of each block.

The placement of data consistent with failure domains increases storage reliability. However, this scheme does not deliver optimal performance, since the access latency between components of different failure domains usually is higher than within a failure domain. For example, the decoding of one data block takes less time, if block fragments are placed on different disks of one physical server instead of being distributed across disks of multiple different servers. In addition, for the purposes of scalability it is reasonable to minimize

size of data transmitted through slow channels, and organize the major part of network connections within dedicated network segments.

Therefore, the choice of failure domain "optimal level" (server, rack, group of racks, etc.), which is considered in data placement algorithm, assumes a trade-off between the system reliability, performance and scalability, and depends on requirements for the storage.

2.4.2 Locality

One can define locality a base of either some locality function that considers access latency and communication speed between network nodes, either explicitly selected locality groups, e.g. servers within one rack in a datacenter. In a first case, the system automatically computes locality function for pairs of storage components from mean data transmission latency. In a second case, locality uses a configuration predefined by the data storage administrator.

The main purpose of considering locality of distributed storage components is to minimize access latency and reduce network communications between remote nodes as well as to lower channel traffic between different racks in a datacenter. Selection of locality groups associated with storage infrastructure components like group of servers within a rack allows one to reduce network workload between racks due to localization of data recovery and workload balancing within one rack.

The role of locality groups is especially important for storages that use erasure codes in favor to replication, since in the former case to access data, it is not enough just to select the replica with nearest topological distance, but necessary to retrieve several data fragments possibly located on different servers.

2.4.3 Simple Placement

We propose the following heuristic algorithm of tuple generation. Let us assume the x disk tuples are to be generated. To obtain them one considers the numbered list of all disks $d_1, ..., d_N$ in a storage, and proceeds the random permutation $d_{i_1}, ...d_{i_N}$ of this list. The produced permutation is split into the groups, each holding n disks. If the tuple is under filled then its disks are discarded from following consideration. One repeats the described process until the required number of tuples is obtained. The algorithm is as follows:

a) Let $X = \emptyset$ be the list of disk tuples.

b) Let $D = \{d_1, ..., d_N\}$ is the list of all disks in a storage.

c) If the list D contains less, than n disks, then go to step b) (fill the list with disks), else, let $T = \emptyset$ be an empty list for a new disk tuple.

d) While list T contains less than n disks, select a random disk d_i from D, add this disk into the list T and remove it from the list D.

e) Add new disk tuple T into the list X.

f) If the list of tuples X has x disk sets, then finish the algorithm, otherwise go to step c).

To achieve the value S for scatter coefficient, one has to generate S different disk permutations in a storage. For a corresponding algorithmic flowchart see figure 4 in the Appendix.

The considered algorithm is not optimal in a sense that there is a probability to generate a subset of similar disk tuples, for which the tuple pairs would share several of same disks. It is worth mentioning that the optimal partitioning with sufficiently large number of disks N_{Disks} and scatter coefficient $S \ll N_{Disks}$ results in either no common disks or only one same disk shared by any two tuples. Moreover, high number of intersections between tuples prevents any significant speed up for data recovery via parallel access among the tuples, and potentially leads to unbalanced access to storage disks.

2.4.4 Data Placement with Failure Domains

If one generates tuples with respect to failure domains, the set of storage disks is partitioned into nonintersecting subsets, corresponding different failure domains (server, rack). Additionally, no obtained disk tuple should contain more than one disk from one failure domain.

To provide the set of tuples satisfying the formulated condition, the above-described algorithm is modified as the follows. On the step c) we select not a random disk from the remaining ones in the list D, but a disk contained in a failure domain, which disks are not in the tuple T. It should be noted that appropriate functioning of the algorithm requires the number of failure domains in a storage to be no less than the parameter n in the underlying erasure code. The algorithm is as follows:

a) Let $X = \emptyset$ be the list of disk tuples.

b) Let $D = \{d_1, ..., d_N\}$ be the list of all disks in a storage.

c) If the list D contains less than n disks, then go to step b) (fill the list with disks), else, let $T = \emptyset$ be an empty list for a new disk tuple.

d) While list T contains less than n disks, select random disk d_i from failure domain $F_j \subset D$, having no disks in the list T ($F_j \cap T = \emptyset$), add this disk into the list T and remove it from the list D.

e) Add new disk tuple T into the list X.

f) If the list of tuples X has x disk sets, then finish the algorithm, otherwise go to step c).

To achieve the value S for scatter coefficient, one has to generate S different disk permutations in a storage. For a respective algorithmic flowchart consult figure 5 in the Appendix.

2.4.5 Data Placement with Failure Domains and Locality

The following proposed algorithm is similar to the one above in the clause 2.4.4 except for the tuples being separately generated for each locality groups. To provide uniform workload of the disks the required number of tuples is divided among locality groups so that the number of tuples in each group is proportional to the number of disks in this group. The modified algorithm is as follows:

a) Let $D = \{d_1, ..., d_N\}$ be the list of all disks in a storage.

b) Let $I = \{I_1, ..., I_L\}$ be the list of tuples with indices of disks in L locality groups.

c) Let $I_k = \{I_{k,1}, ..., I_{k,n_k}\}$ be the tuple of disk indices in k-th locality group ($1 \leq k \leq Lk$), ($\bigcap_{k=1}^{L} I_k = \{1, ..., N\}$).

d) Let n_k be the number of disks in k-th locality group ($\sum_{k=1}^{L} n_k = N$).

e) Set $k = 1$.

f) Let $D_k = \left\{d_{I_{k,1}}, ..., d_{I_{k,n_k}}\right\}$ be the list of all disks in k-th locality group, ($\bigcap_{k=1}^{L} D_k = D$).

g) Let $X_k = \emptyset$ be the list of disk tuples in k-th locality group.

h) If the list D_k contains less than n_k disks, then go to step f) (fill the list with disks), else, let $T_k = \emptyset$ be an empty list for a new disk tuple in in k-th locality group.

i) While list T_k contains less than n_k disks, select a random disk $d_{I_{k,i}}$ from failure domain $F_{k,j} \subset D_k$, which has no disks in the list T_k ($F_{k,j} \cap T_k = \emptyset$), add this disk into the list T_k and remove it from the list D_k.

j) Add new disk tuple T_k into the list X_k.

k) If the list of tuples X_k has x_k disk sets, then go to step l), otherwise go to step c).

l) If the number k of current locality group is less than L, then increment the number by one and go to step f). Otherwise, finish the algorithm.

The corresponding flowchart is provided on figure 6 of the Appendix.

2.4.6 Data Placement with Network Proximity of Storage Nodes

The distance function $S(N_i, N_j)$ on the set of all nodes N_i, constituting the distributed storage, can be defined as a mean network latency for the message between the nodes N_i and N_j. The distance $S(N_i, N_j)$ equals zero, if $N_i = N_j$. If the node N_i is inaccessible from the node N_j, then the distance is assumed to be infinite. This distance function don't have to be static and can vary with time due to changes in a distributed storage.

In a real-life, distributed storage the dynamical distance function depends on hardware storage configuration and "jitter" workload patterns for network channels connecting storage nodes. To handle this issue, each storage component independently gathers data for respective averaged node-to-node latency and regularly sends messages to a complete set of all known nodes or a predefined subset of them to avoid excessive communications with remote nodes. To reduce storage workload for handling system messages the typical timeframe is about from ones to tens of minutes. The processed data is then sent to a dedicated monitoring server, which aggregates information on a global storage state. A metadata server responsible for block allocation on data servers as well as data recovery and migration can perform the role of a monitoring server.

The choice of disk tuples based only on distance minimization between them with respect to network topology while neglecting any other parameters can lead to irregular disk distributions across generated tuples and therefore to irregular data placement on storage disks. To avoid this issue we introduce the weight coefficients corresponding to the fraction of useful data stored on the respective disks. This weight coefficient corrects the tuple generation, so the algorithm favors less filled disks. This technique generates new disk tuples as new data comes into the storage until the number of tuples reaches the specified threshold value. After this event new tuples will be generated only on addition or deletion of storage nodes as well as during the workload balancing process.

Let $S(d_i, d_j, w_j)$ be the modified function of a distance from a disk d_i to a disk d_j, accounting for disk fullness via weight coefficient w_j. We should note that this definition distinguishes between the distance from d_i to d_j and the distance from d_j to d_i. The modified function can be calculated as a linear combination of network topology distance and disk weight w_j. The coefficients at the linear terms can serve for fine-tuning in various conditions. We consider the modified algorithm of disk tuple generation, which is based on distance function. We assume that storage component for tuple generation has

complete information about distances between all storage components. The modified algorithm is as follows:

a) Let $X = \emptyset$ be the list of disk tuples.

b) Let $D = \{d_1, ..., d_N\}$ be the list of all disks in a storage.

c) Let $S = \{S_{ij}\}, i, j \in [1, N]$ be a two-dimensional array of distances between storage nodes containing disks d_i and d_j. To achieve efficiency one can represent this array as a set of lists with distances sorted in ascending order, where each list corresponds to the specific disk of a storage.

d) Select random disk d_i.

e) Let $T = \{d_i\}$ be the list for a new disk tuple.

f) Select several random disks $d_{i_1}, ..., d_{i_k}$ from D, satisfying the implemented policies of data placement (e.g., complying with failure domains F). Next, consider extended list $T'(l) = T \cap d_{i_l}$ for a selected disk d_{i_l}. Compute the total distance $S'_m(l)$ between each disk from the given extended list and all other disks in it. Compute averaged total distance $S'_{avg}(l)$. Then select the disk $d_{i_{l^*}}$ from the disk set $d_{i_1}, ..., d_{i_k}$. The criterion for selection is the minimum averaged total distance for the extended disk list $T'(l)$. Add the selected disk into the list T.

g) While the list T has less than n disks, repeat step f).

h) Add new disk tuple T into the list X.

i) If the list of tuples X has x disk sets, then finish the algorithm, otherwise go to step c).

This algorithm uses heuristic method of tuple generation, which is not strictly optimal in sense of introduced weighted distance function between disk nodes. The implementation of strictly optimal algorithm is impractical since computational overhead costs are considerably high to analyze the list of all disks at each tuple generation. The respective flowchart is presented on figure 7 of the Appendix.

3 Computation Results

We studied the dependence of super large distributed data storage MTTDL on model parameters. The reference input has the following values: 0.5 PB of useful data size in a cluster, 2 TB capacity of a single disk, 300 MB size of a data block, (6,4)-scheme of error coding, 200000 hours of a disk MTTF, 4

hours of a mean recovery time for a disk data. The sequence of computational experiments yielded the storage lifetimes to data loss with one model parameter chosen to vary and other parameters fixed at reference values.

We also compared the dependencies of data loss probabilities against the fraction of failed disks for proposed data placement scheme and ordinary tuple generation algorithm (copyset algorithm). The plots on the respective figures 1, 2 demonstrate that both placement policies provide sufficiently high reliability for a storage with the failed disks counting a few percent of total number. In addition, the developed algorithm shows significantly higher reliability for large number of failed disks.

The plots on figures 1, 2 show, that the increase in number of storage disks for considered schemes (random data placement, data placement within disk tuples) reduces MTTDL of data storage in approximately linear manner. One can see on the plot of the figure 3 that the mean storage lifetime up to first irrecoverable data loss for a proposed algorithm is about 1000 times higher, than the same parameter for the random data placement.

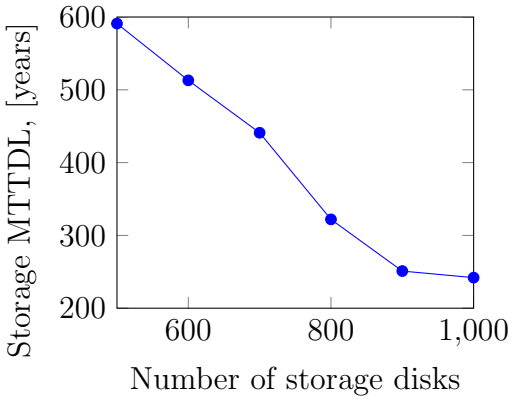

Figure 1: A storage MTTDL against the number of disks for a policy of random disk placement for (6,4) erasure code

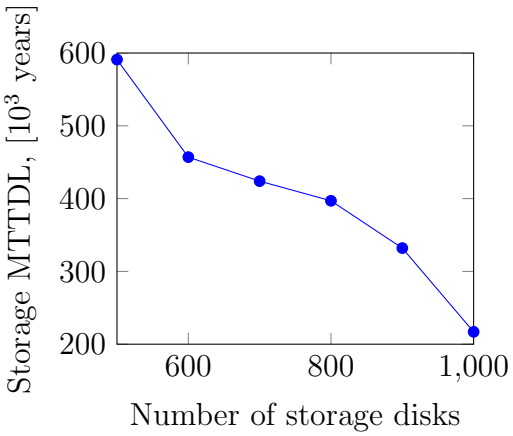

Figure 2: A storage MTTDL against number of disks for a policy implemented via the proposed algorithm for (6,4) erasure code

4 Conclusions

The conducted study revealed that the random data placement policy do not provide desired level of reliability and accessibility for super large petabyte-size distributed storages at the simultaneous failures of several disks. Moreover, the random placement schemes have poor scalability due to network congestion

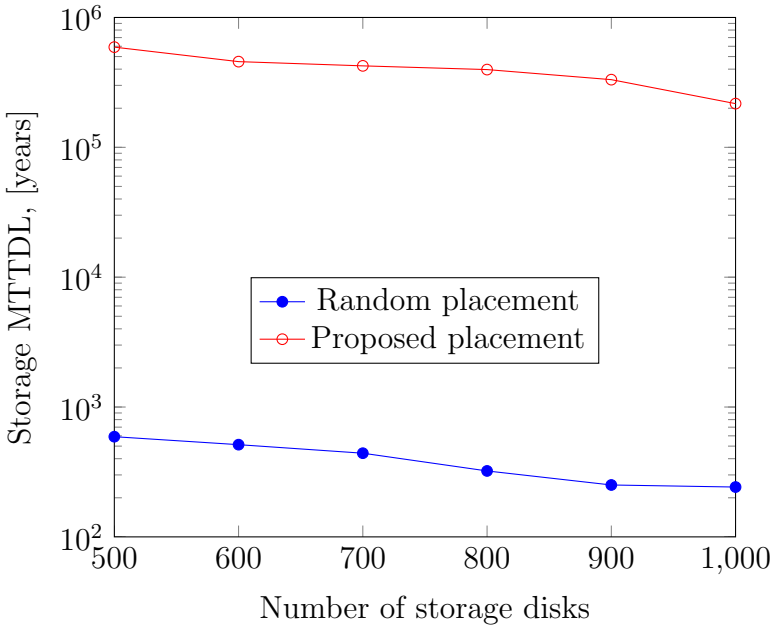

Figure 3: Comparison of data storage MTTDL for the proposed algorithm and the algorithm of random data placement for (6,4) erasure code against number of disks in logarithmic scale

caused by uncontrolled chaotic data exchange between mutually remote storage locations.

We demonstrated that our proposed algorithm significantly increases storage MTTDL and reduces data loss probability due to simultaneous disk failures, thus improves storage reliability. The developed algorithm also improves data availability by reducing data exchange between mutually remote random storage locations. We achieve this by algorithmic selection of disk tuple for writing each next data block should be not random, but defined by relative disk indicators of fullness and workload. In addition, the data locality should be considered. In order to increase reliability and rate of data access it is reasonable to store blocks of one file in the one disk tuple. Moreover, to write a file it is advisable to select the disk tuple in network topology the most close to the data storage client utilized for writing data.

Acknowledgements. The article is published within applied scientific research performed with financial support of the Ministry of Education and Science of the Russian Federation. Subsidy provision agreement 14.579.21.0010. Universal identifier of the agreement is RFMEFI57914X0010.

References

[1] J. Gantz and D. Reinsel, The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the Far East, *IDC iView: IDC Analyze the Future,* (2007), 1–16.

[2] S. Beloussov, Grow Your Business Faster with Acronis Cloud Solutions, Odin Summit 2015.

[3] L. Ivanichkina and A. Neporada, Mathematical methods and models of improving data storage reliability including those based on finite field theory, *Contemporary Engineering Sciences,* **7** (2014), no. 28, 1589–1602. http://dx.doi.org/10.12988/ces.2014.411236

[4] D. A. Patterson, G. Gibson, R. H. Katz, A case for redundant arrays of inexpensive disks (RAID), *ACM SIGMOD Record,* **17** (1988), no. 3, 109–116. http://dx.doi.org/10.1145/971701.50214

[5] Constellation ES. High-capacity storage designed for seamless enterprise integration. DataSheet, Seagate. http://www.seagate.com

[6] L. Ivanichkina and A. Neporada, Computer simulator of failures in super large data storage, *Contemporary Engineering Sciences,* **8** (2015), no. 35, 1679–1691. http://dx.doi.org/10.12988/ces.2015.59270

Appendix

Flowcharts for data placement algorithms

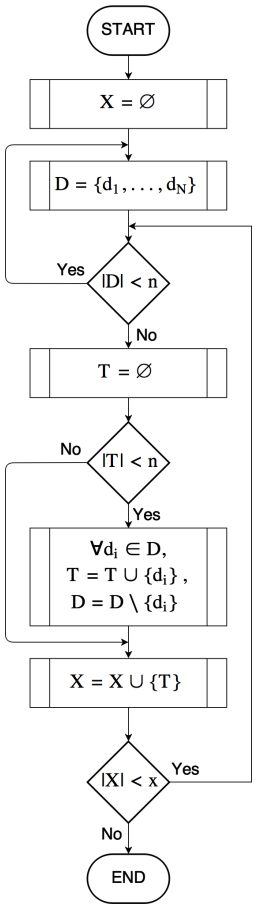

Figure 4: Flowchart for a random placement algorithm

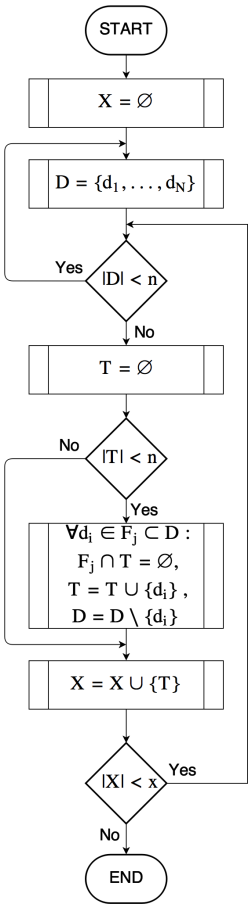

Figure 5: Flowchart for a data placement algorithm considering failure domains

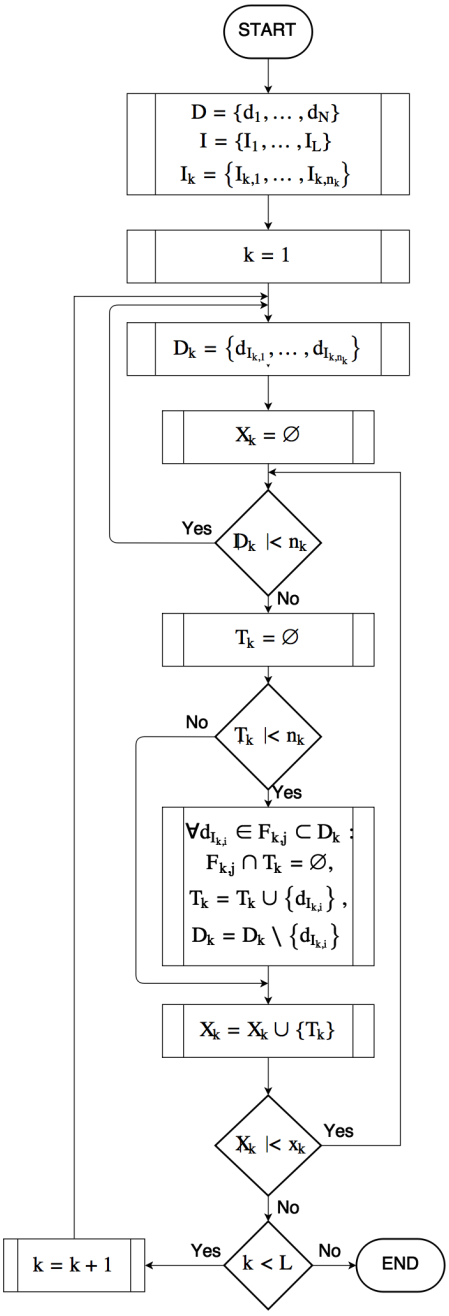

Figure 6: Flowchart for a data placement algorithm considering failure domains and locality

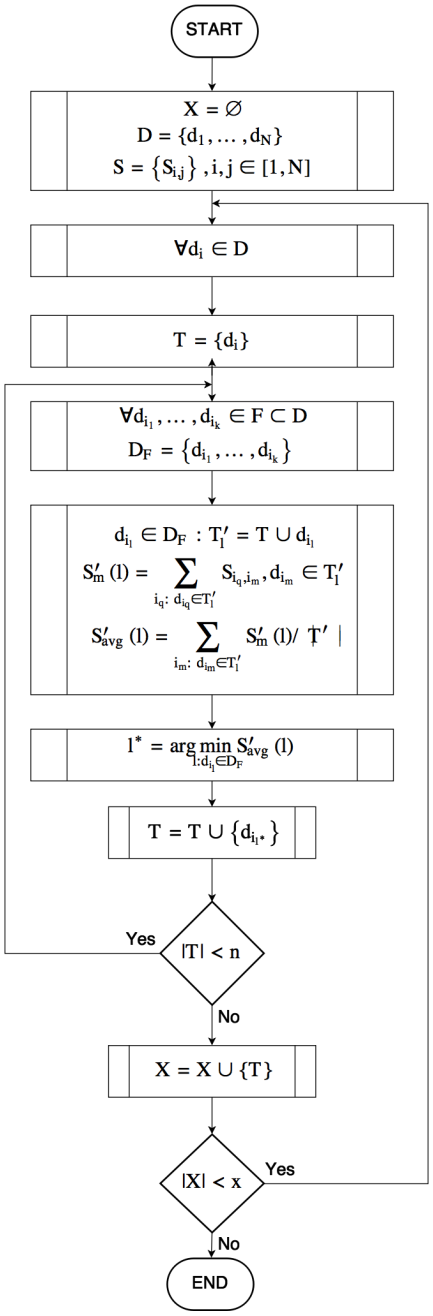

Figure 7: Flowchart for a data placement algorithm considering nextwork proximity

Received: August 19, 2016; Published: October 17, 2016