Distributed Key-Value Store with Data Partitioning and Replication

Distributed System 1, 2022-2023

Nicola Carlin
nicola.carlin-1@studenti.unitn.it

Stefano Genetti
stefano.genetti@studenti.unitn.it

Contents

1 Introduction and project structure

In this project we present a distributed system which implements a *distributed hash table*-based peer-to-peer key-value storage service inspired by Amazon Dynamo [2].
In the following of this document we detail our solution straining the attention on the main design choices. Particular emphasis is placed on the strategies adopted to provide *sequential consistency.* Furthermore, it is important to notice that all the operations have been implemented so that they use as few messages as possible throughout their executions as described in their dedicated sections. All the described operations are conveniently documented at this link.

The code has been implemented in Akka, a toolkit for building distributed message-driven applications for Java [1]. The project structure comprises the following files:

- `Client.java`: the class extends the AbstractActor class provided by Akka to abstract the behaviour of a client which interacts with the distributed hash table.
- `ClientMessage.java`: definition of the messages exchanged between the clients and storage nodes to provide data services.
- `Item.java`: this class represents a data item stored in the distributed database.
- `Main.java`: in this file we provide an execution to test the implemented functionalities.
- `Message.java`: definition of the messages transmitted among the storage nodes in order to provide management services.
- `Node.java`: this class implements an Akka actor which represents a storage node which populates the DHT.
- `Request.java`: this class encodes the properties of a read or write request made by a client.

2 System overview

The overall goal of our implementation is to develop a distributed peer-to-peer key-value database. The architecture is composed by two Akka actor classes whose instances are referred to as *nodes* and *clients* which interact by means of exchange of messages. In order to simulate the behaviour of a real computer network, we add random delays before each transmission.
The distributed hash table is composed by multiple peer nodes interconnected together. The stored data is partitioned among these nodes in order to balance the load. Symmetrically, several nodes record the same items for reliability and accessibility. The partitioning is based on the keys that are associated with both the stored items and the nodes. We consider only unsigned integers as keys, which are logically arranged in a circular space, such that the largest key value wraps around to the smallest key value like minutes on analog clocks. A data item with key K is stored by the first N nodes in the clockwise direction from K on the ring, where N is a system parameter that defines the degree of replication. All the nodes are required to know all the other nodes in the system, allowing them to locally decide which of them are responsible for a data item.
On the other hand, the clients support data services which consist of two commands: i. `update(key, value)`; ii. `get(key)`→`value`; which are used respectively to insert and read from the DHT. Any storage node in the network is able to fulfill both requests regardless of the key, forwarding data to/from appropriate nodes. Although multiple clients can read and write on the data structure in parallel, we assume that each client performs read and write operations sequentially one at a time.

For the sake of clarity we provide an illustration of the network structure in Figure 1.

3 Replication

The distribution of redundant copies of the same item among several replicas leads to several advantages such as avoiding single point of failures and load balancing. On the other hand, this design choice introduces new challenges which make the overall design process more complex. The purpose of this section is to detail the solutions employed in order to implement read and write primitives and preserve consistent replication. Furthermore, at the end of this section, we briefly discuss about the data structures that have been adopted to maintain a persistent storage containing the references to the other peers and every data item the node is responsible for.

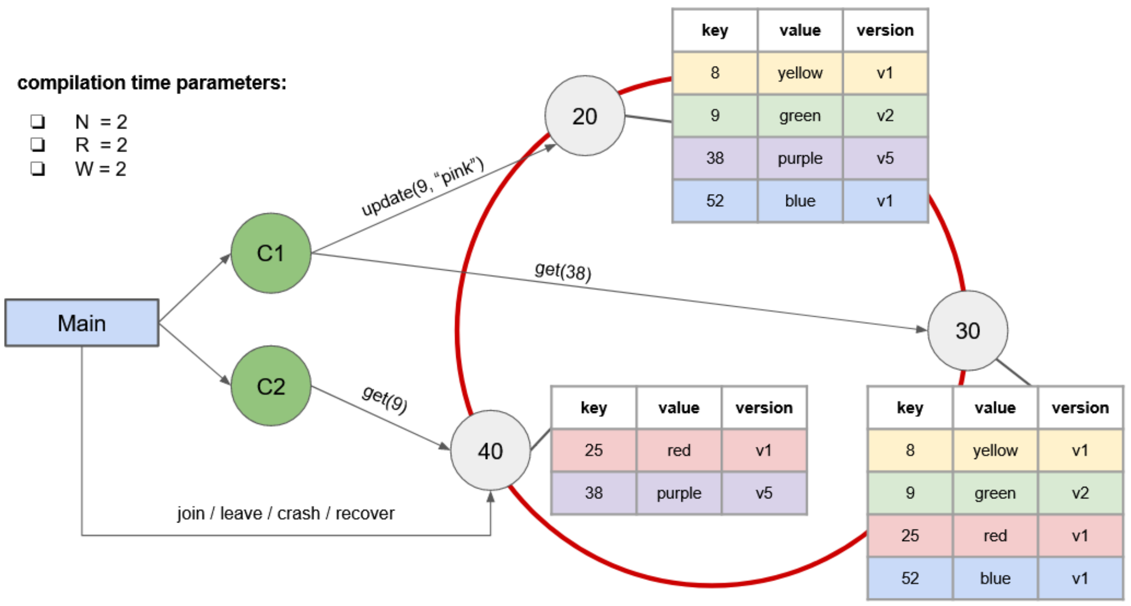

Figure 1: System architecture.

3.1 Read

A node n_r, referred to as the coordinator, can be requested by a client c_r to return the most recent version of an item, if set. Client c_r initiates the `Get` operation by sending the `GetRequest` message to coordinator n_r, which appends, in the `Map <String, Request> requests`, the new request. This last is abstracted by means of an instance of the `Request` class which has been implemented for the sake of keeping track of ongoing operations. The `Request` object includes the client's name and reference, the type of operation requested (read or write), the item and the number of replies received for the particular request. As commented below in this text, with the aim of ensuring sequential consistency, it is crucial that the quorum constrains are respected. For this reason the reply counter field that we have just mentioned is part of the `Request` object.
This done, the coordinator immediately calls the function `getResponsibleNode(itemKey)` to retrieve the set of responsible nodes for the requested item. If this node set includes n_r and there is no lock defined (the reasons for this are explained in the section Implementation of sequential consistency), the coordinator reads the element from its local storage and updates the version associated with the request. Subsequently, the coordinator requests each responsible node to provide the value and version of their item replica by sending them a `Read` message. If the item is not owned by the nodes or if there is a lock set, they request is ignored. Otherwise, the operational replicas reply to the coordinator n_r with the `ReadItemInformation` message which includes their version of the item. The coordinator n_r waits for R responses (or $R-1$ if it is part of the responsible nodes); if this condition is not met within time T, the timeout expires, the request is removed from the mapping and a `GetResult` message with an error code is sent to the client c_r. However, if all R responses are received within time T, the coordinator proceeds to compare the retrieved versions and returns the most recent item associated with the provided key to client c_r using a `GetResult` message. By waiting for R responses, the system ensures that the most recent item is returned, and no sequential consistency issues arise, as explained later in the section Implementation of sequential consistency.

3.2 Write

A node n_w, known as the coordinator, can be requested by a client c_w to update (or insert for the first time) an item. The modifications performed by the distributed computational process to fulfill the request must be in line with the replication and quorum constraints. The process commences with the client c_w sending an `UpdateRequest` message to the coordinator n_w, which becomes responsible of the update operation and stores the request in the `Map <String, Request> requests` as for the `Read` operation. Upon receiving the message, the coordinator n_w determines the set of responsible nodes for the item processed. In the event that this set contains its own key, it immediately examines the current version stored for the item and identifies this version as the new virtual version for the updated item. Subsequently, it sends a `Version` message to all the other responsible nodes, demanding them to provide the version of the item they have stored. If a lock is set, these nodes don't reply. Otherwise, they update the item with their own version and send it back to the coordinator in an `UpdateVersion` message. The coordinator n_w waits for W responses (or $W-1$ if it is part of the responsible nodes) to ensure sequential consistency within a time limit T; otherwise a timeout occurs, the request is removed from the mapping and a `UpdateResult` message with an error code is sent to the client c_r. If enough replies are received, it takes the latest version by comparing all the items received in the `Version` messages and increment it by one. By doing so, we ensure that the written item does not conflict with its last stored version. After all of these operations have been completed, the updated item is sent back to client c_w with an `UpdateResult` message. In the end, the coordinator n_w requests that all responsible nodes update their stored items with the new one as specified in the `Write` message. The sequential consistency requirement is guaranteed by means of the quorum and the lock mechanisms, as explained in the below section Implementation of sequential consistency.

3.3 Implementation of sequential consistency

The distribution of multiple replicas of the same item across several storage nodes, requires the implementation of proper consistency models to specify the results of read/write operations in the presence of concurrency. More in depth, our aim is to guarantee sequential consistency: *the result of any execution is the same as if the operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program* [4]. According to this definition, when processes run concurrently on different machines, any valid interleaving of read and write operations is acceptable behaviour, but all processes see the same interleaving of operations.

With the aim of implementing sequential consistency, we employ two strategies which are presented in the following of this section.

First of all, we rely on a quorum-based protocol as originally proposed by Thomas [5] and generalized by Gifford [3]. In this context, R and W have to be properly set in order to provide sequential consistency. Specifically, the values of R and W are subject to the following two constraints:

1. $R + W > N$
2. $W > \frac{N}{2}$

The first constraint is used to prevent read-write conflicts, whereas the second prevents write-write conflicts. Indeed, if the first requirement is not enforced, two concurrent get and update operations on the same item key issued by two clients, might violate sequential consistency. In fact, in the case $R + W \leq N$, a temporary network partition caused by unpredictable network delays could determine erroneous outcomes. On the contrary, with $R+W > N$ we are sure that the intersection between the set of nodes affected by the two operations is not empty. On the other hand, if the second constraint is violated being $W \leq \frac{N}{2}$, sequential consistency might be compromised by two clients writing different values on the same item key. Indeed, the two write operations might be forwarded by the network towards two disjoint sets of W storage nodes. Under these circumstances, we would run into trouble as the two updates would both be accepted without detecting that they are actually conflict and we would end up having items with the same version but different values.

Even if a proper configuration of R and W quorum parameters as suggested by the literature ensures a proper partitioning of the items among the storage nodes, simultaneous read and write operations performed by multiple clients on the same item key, can still result in inconsistent system state with respect to sequential consistency. Let us consider the case in which a client is writing on item k. In the protocol detailed above in this paper, at the end of the writing procedure, the coordinator sends the update to all N replicas through `Write` messages. Because of unforeseeable low level network conditions, we can not predict in advance when and in which order the `Write` messages will reach the destination. As a consequence, a conflicting write operation could potentially write on the same replicas. Once the two operations completes we may end up having different values for the same data version. Similarly, a read on the same item key could be problematic. For instance, it is possible that because of the arbitrary sequence of delivery of `Write` messages, there are R nodes holding version v of item k and R nodes with the newer version $v+1$. In this situation, in the case where the same client performs two reads on k one after the other, it might observe version $v+1$ in the course of the first read and version v subsequently. In order to overcome to this limitation we use *locks*. The locks are defined by the name of the client that issues the operation. Akka ensures that within the ActorSystem this name is unique. Each storage node, annotates for each of its items whether it is locked by an ongoing operation or not. To accomplish this idea we rely on an appropriate data structure `Map<Integer, String> locks` which lets associating to each item key a lock. In the context of a write operation on an item k, each responsible node, before executing any other instruction, checks the availability of k. If the coordinator belongs to the set of responsible nodes for item k, this check is executed upon receiving the `UpdateRequest` from the client. The other replicas verify the value of the lock on k as soon as a `Version` message is received. If there exists a lock on item k then the write operation can not be fulfilled and the request is ignored. Otherwise the node sets a `lock` l_1 on item k to obtain exclusive resource access. The lock on k is released once the write operation successfully completes, meaning that the node updates the item. Analogously, in the context of a read operation on item k, when a coordinator receives a `GetRequest` message from a client and it is one of the responsible nodes for k, or a storage node receives a `Read` message from a coordinator, the request is taken into account if and only if it has been not set a lock on item k yet. As well as with conflicting writes, in this way we reject reads that may break sequential consistency. For the sake of clarity, we provide a pictorial representation of the proposed lock strategy in Figure 2. In this exemplary scenario, at the end of the day, both the `write` operations are blocked due to an unfortunate succession of events. Nevertheless, it is important to point out the fact that we do not cause a deadlock. Actually, the two concurrent writes, after the expiration of their respective timeouts T, will both fail. In the code, when a write operation on item k times out, a `ReleaseLock` message is propagated towards all the N replicas of item k. Upon receiving this last notification, the recipient clears the lock on k which had been allocated in a preceding step of the computational process. In this regard we understand the motivation behind the identification of the lock by means of the name of the client which has originally initiated the operation. If this was not the case, the `ReleaseLock` notification might delete the locks on k which have been prepared by another write operation in the meantime. This removal might prevent a client from carrying out successfully its write operation. On the other hand, including the client tag on each lock makes it impossible for a write operation initiated by client c_1 to remove the locks allocated in the context of a second write issued by c_2. As an alternative, for the purpose of reducing the number of messages exchanged over the network, we could have regulated the lock removals setting a timeout $T_l \geq T$ for each lock as soon as it is allocated. Indeed, according to the aforementioned assumptions, after T seconds we are confident that the matching write has been completed or has failed. Consequently, we can safely release the corresponding locks. Although this method would be correct, it could potentially slow down dramatically the distributed algorithm. Indeed, we would have to wait more than T seconds between each pair of write operations on the same item k.

3.4 Local storage

As discussed above, both the nodes and the items stored in the DHT are associated with a unique unsigned integer key. This last determines the order of the nodes in the ring and the partitioning of the stored data. Each node keeps track in its local attributes of the items it is responsible for and of the other network peers. Storage nodes currently in the ring are abstracted by means of `ActorRef` objects whose implementation is made available by Akka. On the other hand, we have defined an `Item` class in order to represent stored data items. With the aim of conveniently arrange these objects we allocate two `HashMap` Java Collections whose names are self explanatory: `Map<Integer, ActorRef> peers`; `Map<Integer, Item> items`.

4 Item repartitioning

The overlay ring network topology of nodes which constitutes the distributed hash table, is not static; on the contrary nodes can dynamically join, leave, crash and recover one at a time and only when there are no ongoing operations. We assume that operations might resume while one or more nodes are still in crashed state. In order to handle these functionalities, each node supports management services which can be accessed by means of dedicated messages. When nodes leave or join the network, the system repartitions the data items accordingly. In this section we discuss about the design proposals which has been implemented to handle these operations without violating system invariants.

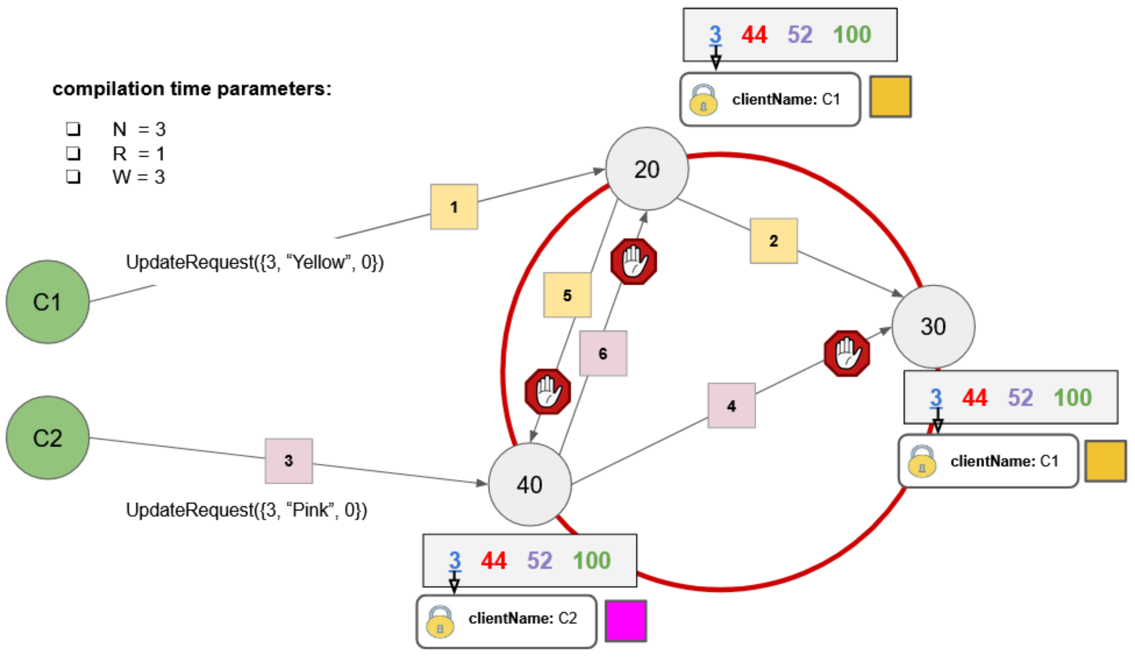
compilation time parameters:
N = 3
R = 1
W = 3
UpdateRequest({3, "Yellow", 0})
UpdateRequest({3, "Pink", 0})
C1
C2
20
30
40
1
2
3
4
5
6
3 44 52 100
clientName: C1
clientName: C2

Figure 2: Two clients are updating simultaneously the value associated with the same item key. In the example, we assume that the messages are propagated in the order specified by the colored squares on each link. Both the operations do not successfully complete since they are not able to acquire the necessary locks. Without the employed strategies, these concurrent writes would have compromised sequential consistency.

4.1 Joining

The purpose of the join operation is to add a new storage node n_j in the distributed hash table. To do this, we send a join request encoded by a `JoinMsg` message in which the `key` of the newly created node is specified. In addition, the join request includes the actor reference of one of the currently running nodes n_b, which becomes its `bootstrapping peer`. This latter is contacted by n_j to retrieve the current set of nodes constituting the network. After sending the `ReqActiveNodeList` message to n_b, node n_j waits for a `ResActiveNodeList` response within a timeout interval of T seconds. If no reply is received throughout this period, the join operation is aborted and further `ResActiveNodeList` messages are ignored. This design choice is justified by the fact that the node n_b might be in crash state and because of our project assumptions, there is no point in waiting for its recovery. If this is not the case, having received the list of the currently active nodes, the joining node n_j updates its local `peers` data structure and then it requests data items it is responsible for from its clockwise neighbour n_c, computed by `getClockwiseNeighbor` function, which holds all the items needed. Remarkably, if the list of currently active nodes contains n_j, the join operation is aborted since there can not be two nodes in the ring with the same key. Analogously to the point-to-point communication with the bootstrapping peer we have just mentioned, also the wait for a `ResDataItemsResponsibleFor` reply from n_c is regulated by a proper timeout. From the point of view of node n_c, we are able to understand the set of items needed by n_j executing `getResponsibleNode function`. Due to space constraints, `getClockwiseNeighbor` and `getResponsibleNode` are not commented in this document. Once the set of items required by n_j have been received, the node updates its local `items` data structure. At this point read operations (`JoinReadOperationReq`) are sent to the item replicas to ensure that the received items are up to date. If we omit these read operations, the later `AnnouncePresence` message towards the N involved replicas might cause the deletion of the most recent item version replacing it with an old one, breaking quorum assumptions. In this stage, we do not send the `JoinReadOperationReq` to the clockwise neighbour which has already sent its item set. After this, n_j waits for time T the arrival of $R-1$ `JoinReadOperationRes` messages for each data item. After receiving the updated data items, the node n_j can finally announce its presence to every node in the system (`AnnouncePresence` message) and start serving requests coming from clients. Interestingly, because of quorum constraints, there is not need to wait for more than $R-1$ responses for each data item i. Indeed, the configuration of R at compile time ensures that at least one of the R replicas have included the last version of the data item i at hand. Finally, upon learning about a new node, the others remove the data items assigned to it they are no longer responsible for.

Whenever the join operation has to be aborted throughout its execution, it is fundamental to rollback the performed operations to bring the system to the state prior to the join.

The proposed implementation guarantees a reasonable trade off between the amount of messages exchanged and execution time. For instance, instead of sending the `JoinReadOperationReq` messages to $N-1$ item replicas and waiting for $R-1$ `JoinReadOperationRes` responses, we could forward the request only to $R-1$ replicas at first, and query the remaining ones afterwards in the case we do not get enough replies within the timeout interval. However, this would clearly delay the join operation processing time.

4.2 Leaving

A node n_l can be requested to leave the network by means of a `LeaveMsg` message. The leaving node iterates its item set and invokes `getResponsibleNode` function to find out the nodes that are going to become responsible for them after its departure. Note that, in this step of the computation, we consider only the peers (hereafter referred to as set Σ) which were not holding the items at issue before the leaving of n_l. Indeed, in the steps explained later on, there is no reason to pass data items to nodes which were already responsible for them before the departure of n_l.
Once the set Σ has been filled with the proper destinations, before introducing them the set of novel items, we must make sure that each $n \in \Sigma$ is not in crash state. If we ignore this control, we might forward the set of items to a dormant destination which would ignore the message. In such a situation we might violate the requirement of N replicas in the table for each item or, even worse, lose data irremediably. In order to accomplish this objective, we send a `PreLeaveStatusCheck` message to each node $n \in \Sigma$ and wait for $|\Sigma|$ `DepartureAck` acknowledgments. If n_l does not get enough ACKs a proper timeout kills the execution of the leave. If this is not the case and n_l has successfully received all the expected feedbacks, it can finally announce its departure to the other peers. This last `AnnounceDeparture` message includes the key of the leaving node n_l and a set of data items. This last may vary according to each target destination and can potentially be empty. Following this idea, we do not send all data items to all nodes, but only the essential ones for each destination. Upon receiving the `AnnounceDeparture` notification, the receipt updates its peers collection, and possibly its item set, accordingly.
As commented in the context of the join 4.1, also here we have strained our attention to use as few messages as possible.

4.3 Crash

The normal behaviour of the node upon the receipt of a message is regulated by the mapping message-function handler specified by the overriding of the `createReceive` function provided by the class AbstractActor. On the other hand, in order to simulate the behaviour of a crash node we have implemented the `AbstractActor.Receive crashed()` function which defines a dedicated map message-function handler. We can switch between these two message response modalities using `getContext().become(crashed())` and `getContext().become(createReceive())` instructions.

A crashed node is not considered as leaving but only temporarily unavailable; therefore, there is no need to implement a crash detection mechanism or repartition the data when a node cannot be accessed.

4.4 Recovery

A crashed node can be started again sending it a `RecoveryMsg` message. The recovering node n_r, instead of performing a join operation it requests with a `ReqActiveNodeList` message the current set of nodes from a peer specified in the recovery request. Upon retrieving the `ResActiveNodeList` response from the network, n_r updates its local knowledge about the current set of active nodes in the DHT which might have different participants with respect to the ones which were active before the crash. Once n_r is informed about the actual network configuration, it discards those items that are no longer under its responsibility due to other nodes joining while it was down. At this point the current clockwise neighbour of n_r is computed with the method `getClockwiseNeighbour` of the class Node. The returned node is contacted by n_r in order to obtain the items that are now under its responsability due to nodes leaving while it was not attending the network dynamics. This time,

unlike the case of the join, the `ReqDataItemsResponsibleFor_recovery` request contains not only n_r's key, but also the current item set owned by n_r. By doing so, the contacted clockwise neighbour can omit from the subsequent `ResDataItemsResponsibleFor` response message the items already stored in n_r memory. At the end of the execution of the distributed algorithm, the recovering node n_r gets the (potentially empty) set of previously unknown items which are now under its control. Analogously to the other operations described previously in this report, also the transfer of messages which defines the recovery procedure is regulated by a timeout interval of T seconds. If the node specified in the initial recovery request or the clockwise neighbour of n_r, do not reply within the expected period of time, the operation is cancelled and all the modifications to the object state are reverted.
The described procedure is in line with the assignment specifications and does not leverage on a larger amount of messages.

5 Additional assumptions

In addition to the assumptions described in the text of the project assignment we add the followings:

- All the required operations are always executed with at least one storage node that populates the distributed hash table. The first node is inserted in the DHT with a dedicated `InitSystem` message.
- It is not possible to perform a write operation on the distributed database if the number of storage nodes is less than N. Otherwise we are not able to fulfill the required level of replication. In order to provide this condition, we check the number of elements in the peer Map collection: if `peers.size()<N` an error is reported to the requesting client.
- For the same motivations, a leave operation can not be completed if the actual number of storage nodes is smaller than or equal to N. As before, also in this case we control that the condition `peers.size()≤N` is false.
- We assume that there is always at least one node in the distributed storage system which is not in crash state.

References

[1] *Akka*. URL: https://akka.io/. (accessed: 29.07.2023).

[2] Giuseppe DeCandia **andothers**. "Dynamo: Amazon's highly available key-value store". **in**(2007): URL: https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store.

[3] David K. Gifford. "Weighted Voting for Replicated Data". **in**(1979).

[4] Andrew S. Tanenbaum **and** Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. 2st. USA: Pearson Education, 2007.

[5] Robert H. Thomas. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases". **in**(1979).