mathematics

MDPI

Article

Towards a Highly Available Model for Processing Service Requests Based on Distributed Hash Tables

Voichiţa Iancu * and Nicolae Ţăpuş

Department of Computer Science and Engineering, Faculty of Automatic Control and Computer Science, University Politehnica of Bucharest, 060042 Bucharest, Romania; nicolae.tapus@upb.ro
* Correspondence: voichita.iancu@upb.ro

Abstract: This work aims to identify techniques leading to a highly available request processing service by using the natural decentralization and the dispersion power of the hash function involved in a Distributed Hash Table (DHT). High availability is present mainly in systems that: scale well, are balanced and are fault tolerant. These are essential features of the Distributed Hash Tables (DHTs), which have been used mainly for storage purposes. The novelty of this paper's approach is essentially based on hash functions and decentralized Distributed Hash Tables (DHTs), which lead to highly available data solutions, which a main building block to obtain an improved platform that offers high availability for processing clients' requests. It is achieved by using a database constructed also on a DHT, which gives high availability to its data. Further, the model requires *no changes* in the interface, that the request processing service already has towards its clients. Subsequently, the DHT layer is added, for the service to run on top of it, and also a load balancing front end, in order to make it highly available, towards its clients. The paper shows, via experimental validation, the good qualities of the new request processing service, by arguing its improved scalability, load balancing and fault tolerance model.

Keywords: Distributed Hash Table (DHT); high availability; decentralization; fault tolerance; scalability; load balancing; highly available data

Citation: Iancu, V.; Ţăpuş, N. Towards a Highly Available Model for Processing Service Requests Based on Distributed Hash Tables. *Mathematics* **2022**, *10*, 831. https://doi.org/10.3390/math10050831

Academic Editor: Daniel-Ioan Curiac

Received: 19 January 2022
Accepted: 2 March 2022
Published: 5 March 2022

1. Introduction

The concept of high availability represents a key feature for today's online services. It has been addressed in numerous ways, but this work aims to obtain high availability in the most transparent manner possible, by using the scalable, balanced and fault-tolerant peer-to-peer based distributed hash tables. The idea of aggregating data resources and of high availability through DHTs is not new, since they are a fundamental building block of large-scale distributed filesystem, such as [1], but more recently also of blockchains [2,3] and a detailed comparison with other works is provided in Section 5. The novelty of this work's approach resides in the fact that it aims to use the high availability for data, which DHTs traditionally provide, in order to obtain a means for highly available processing. Moreover, it seems that today's trend to go from cloud computing to sky computing [4] encompasses multiple abstractization layers, data aggregation [5] and some peer-to-peer techniques, aspects the current work addresses too, in its specific novel manner oriented towards high availability through DHTs, as is described in what follows.

Nowadays, it is very important for any widely spread application service to be always able to answer and process correctly as many clients and client requests as it is needed, without having downtime. In what follows, both the long name for the request processing service is used, but also the short name: service. *High availability* is an important aspect concerning software applications, which is synonymous with the combination of: fault tolerance, scalability and load balancing among the system's internal components, together with data coherence and data persistence. Since most of these identified characteristics,

to influence a system's high availability, are very much related to not losing the data the system needs to process, and always obtaining the correct version of the data, this paper argues and stresses the fact that, a very good and *highly available database* to store the service's data is *a must* for obtaining a *highly available service*. Such a database must implicitly use good data replication mechanisms [6–8], for fault tolerance, which translates into decentralization, too, and in the need to use a large-scale distributed architecture for the physical infrastructure on which the data are stored.

The most representative model for *decentralization* is the peer-to-peer model [9–12], which has evolved in recent years and is currently also applied for name lookup [3,13] and for in-fashion Blockchain solutions [2,3,14], but peer-to-peer networks are also used in highly decentralized and unstructured overlay networks, such as in [15]. The peer-to-peer architecture is decentralized and offers a higher flexibility and a better tolerance to individual components and node failures. The reason for this is that decentralization automatically implies lack of dependencies and also the decoupling of the main components of the system. This paper considers *extending* these types of capabilities to any type of distributed request processing service that could be offered.

The outcome of the present work is a decentralization solution, based on DHT peer-to-peer networks, for a request processing service, whose purpose is to *improve substantially* its *availability* towards the service's clients. For this reason, this work addresses the following aspects that can influence availability:

1. The system's tolerance to individual failures of any of its components, which can be usually obtained by the DHT stabilization mechanisms and by data replication;
2. The system's scalability, such that it is still functional if it is exposed to high external stress;
3. Load balancing, in order to generate as seldom as possible overloading, which in turn could lead to individual node failure.

This work does not aim to adapt a request processing service to the peer-to-peer technology, but rather to use the important novel improvements described in what follows, to personalize the request processing service in a way that does not imply the need of changing its interface to its clients.

We introduce a *new*, generic solution to obtain improvements applicable to any request processing service, which becomes *highly available*, by distribution and *decentralization*, based on the *novel idea* to use the help of a DHT, in order to attain the purpose of highly available request processing. At the same time, the hereby proposed improved solution manages to keep the image of a unified system and service, thus providing a *Single System Image* (*SSI*) to its clients; this is why the decentralization is performed in a manner that is *transparent* to the service's clients, thus not raising any issues for them. Throughout this work, it was envisaged to define the minimum or no changes to the service's interface, but behind this interface the aim is to *improve* the generic service and make it a highly available one. This is why, in Section 2.3, we describe the ways in which the internal components of the newly decentralized service cooperate and interact, in order to truly make the service highly available. We also point out how this work re-used aspects from our team's previous research in the field, namely the advantages of a peer-to-peer DHT-based decentralized distributed database [7,16,17], with self-extending capabilities. We also argue why such a decentralized distributed database is *crucial* if we want to obtain a highly available decentralized distributed service. Last but not least, given the fact that we have identified *high availability* to be obtained if we identify in the system a combination of *scalability*, *load balancing* and *fault tolerance*, we show the existence of each of the three latter system characteristics in Sections 2 and 3. Section 5 discusses and concludes the improvements brought by this work.

1.1. State-of-the-Art

Any distributed service needs to process its clients' data in some manner, and one could conclude, by observing previous research results, that the clients' *data* integrity and

availability is crucial for the service's integrity. In order for a distributed service to be *highly available*, the *data* that it processes need to be stored in a persistent and, preferably, consistent manner. This observation comes after a lot of study referring to distributed services' needs and requirements in order to offer high availability and this is the reason why, in this section, a very strong focus falls on highly available storage for data, on how peer-to-peer DHTs have been designed and used, besides studying other solutions for highly available distributed services, too.

1.1.1. Distributing Systems Using Highly Available Data

Nowadays, there is a high demand to *process* large amounts of data, coming both from the industry (domain led by companies such as Google, Facebook, Microsoft, Amazon, etc.) and from science (e.g., the CERN cluster), this is why, any popular distributed service, which aims to serve its clients impeccably, should also focus on how to improve the process, in any type of external condition, the entire large amount of data, related to the large set of its clients.

Classically, DHTs have been invented and used only in order to help storing highly available data [1,18,19]. Recently, there is a trend to use them for other purposes, too, such as to improve Blockchain models, e.g., the Lightchain [2], or to offer an alternative to the classical DNS lookup system [3]. Since we *have not identified* another model to address and solve exactly the request *processing availability* problem, by means of DHTs, despite DHTs being one of the most appropriate systems to offer availability, this novel approach and the main contribution of this paper is to go even further when using DHTs and to *offer highly available service request processing*. For this purpose, the naturally good dispersion of the DHT's hash function, which should be applied to each client's name, is of much help. Note that to process client's requests by application servers is a much harder task than the data mining Map-Reduce techniques used in Spark [20] or Hadoop [21].

1.1.2. the Aspect of Storing Data in Decentralized Systems

The field of storing and retrieving big data is already a mature domain, and represents a step forward from the classical SQL database solutions. In order to store and process ever larger amounts of data, the first step was to build clusters, such as the MySQL cluster [22], and some subsequent steps were the NoSQL databases [8], one of which is based on the Google Map-Reduce model [23]. Other popular solutions based on the Map-Reduce model have been implemented in the Apache projects Hadoop [21] and, more recently, Spark [20]. At almost the same time with the evolution of these solutions, distributed filesystems have evolved, such as NFS (Network Filesystem) [24], CFS [1] (the peer-to-peer Chord [9]-based distributed filesystem) and later also Blobseer [25,26], aiming to store the files in such a manner that makes them *persistent*, *coherent* and *more efficiently to be retrieved* in a large-scale distributed system. The NoSQL storage solutions, including the peer-to-peer-based distributed filesystems, have been used for applications that run on top of high and medium-volatile infrastructure. A common characteristic of most of these solutions is the fact that they are not based on the client–server model, but they are mostly *decentralized*.

Aside from using the novel peer-to-peer distributed hash table (DHT) techniques, in order to benefit from their flexibility and elegance of data retrieval, an interesting idea is to also solve the scheduling of resource allocation [27]. More recently, in the field of data mining, there are solutions for bringing the data needed by a node closer to it, so as to mask the data transfer latency, in a manner similar to how it is performed by the hardware cache, standing between the processor (i.e., in a distributed system, a processing node) and the main memory (i.e., in a distributed system, a data source node) [28–31].

To summarize, work in the field of large-scale distributed systems data storage and manipulation ranges from:

1. Storing the data in distributed filesystems [24,25,32], in classical SQL databases [22] and more recently in No-SQL distributed databases [8];

2. Smart distribution and retrieval techniques for geographically distributed data, a field in which the P2P DHTs [9] have been of great help and have become very popular;
3. Search engines and large scale distributed applications for collecting and mining huge amounts of data [20,21,23].

1.1.3. Existing Solutions for High Availability of Distributed Services

Load balancing must be addressed, when aiming to transform a request processing service into a highly available one, because simply distributing the service among more nodes does not guarantee that it will also balance the load of the involved nodes, in order to yield high availability. A classical and wide spread, but not so flexible solution, for load balancing at the network or at MAC/data link level is the Virtual Router Redundancy Protocol (VRRP), which is a solution to provide redundancy in a network, in the same virtual entity. The VRRP technique's drawback is that it is not so flexible, this being the reason why self organizing peer-to-peer networks, including DHTs, seem more appealing to the current study.

Thus, the existing load balancing techniques that could be used are DHT-based techniques [9,11,12,33], but also the VRRP protocol [34], which has an active copy and a backup copy of the running service, or components of the DNS (*D*omain *N*ame *S*ystem), such as the *DNS NAPTR* records [35] combined with the *DNS SRV* records [36]. These techniques are based on the existence of more physical machines behind a DNS name. In case any of these machines experiences heavy load, its task can be taken over by another machine. Even if these techniques offer a means for load balancing and availability increase, they are *static*. This is why the described DNS method used only by itself is *not scalable* and, just by itself, cannot be used to process requests in a highly available manner, within large scale distributed systems.

Distributed Hash Tables (DHTs) on the other hand, are by their nature extremely scalable and resilient to failure. This is something that should be adapted and used, in order to make an existing distributed request processing service to be highly available, with no changes of its external interface towards its clients.

2. Materials and Methods

This section describes the improved novel solution for a highly available distributed service for processing its clients' requests, which maintains the same interface to its clients, while the changes to make it highly available are unnoticeable to the clients. It focuses on introducing as little new communication as possible, in order to maintain the Single System Image (SSI), despite the novel decentralization method, based on *DHTs* and *highly available data*. In terms of generic components that should be used to decentralize any distributed service, we enumerate: (1) a *load-balancing dispatcher*, which transparently distributes the load among nodes; (2) a DHT peer-to-peer network based on Chord [9]; (3) a reliable decentralized distributed database, also based on DHTs.

The physical testbed on which the solution was deployed

The improved solution for high availability was deployed and tested in our team's datacenter, designed for high performance computing on large-scale distributed systems. It contains more than 3500 CPU cores and 12 TB of dedicated RAM, together with 30,000 GPU cores and 220 TB of distributed storage, interconnected by GluInfiniband. The CPU core ensemble comprises hybrid architectures such as Intel Xeon, AMD Opteron, Power7 and CellBE, interconnected by means of very-high-speed networks, such as 10Gigabit and Gigabit Ethernet. The various node architectures and the high-speed interconnections among them, together with the extended storage, make this infrastructure a good candidate for validating new findings regarding highly available decentralized distributed processing of clients' requests, with minimum communication delays. All of these are features envisaged by the improved software architecture for high availability, presented in the following subsections.

The novel contributions of this paper and the design of the experimental system are mainly driven by the features of the hash function within the Chord DHT. Note that in Chord all peers are equal, and the destination of a message is identified by collaboration among the peer nodes, with high probability in no more than $O(log_2(No_{peers}))$ steps, according to the dimensions and organization of the per node routing finger tables [9].

The hash function of the DHT

The hash function of the DHT is essential in this work's vision for an improved model of obtaining a *highly available service*, since it guarantees, with high probability, both: (1) the uniqueness of the hash ID, which has been obtained by computing a hash of the data; (2) the large distance, i.e., difference, within the space of the hash IDs of the hash values that correspond to very similar pieces of data. This second property means that related pieces of data are stored or handled on different nodes, a property which is good at obtaining a *balanced distributed system*. In other words, pieces of similar data are stored, with a high probability, on totally independent nodes, a thing that is very useful when storing similar and thus correlated data, which often can be derived from one another, thus yielding *fault tolerance*, too. Note that this exact property is also important when replicating within a world-wide distributed Chord DHT: it is very likely for a peer that is responsible for some data, and its clock-wise successor peer on the ring, which backs that same piece of data, to find themselves very distant, geographically, thus *physically independent* from one another. This allows the distributed ensemble of nodes to behave as a *fault tolerant* system, thus offering a *persistent* data storage. This last property is essential to a reliable storage system based on DHTs, such as a *decentralized highly available database* [7,16,17], which can reliably store the data needed by any request processing service. Moreover, for a more "dispersed" storage of the data in the DHT network, one can successively apply more hash functions [37].

2.1. System Load Balancing Based on the Intrinsic Properties of the Chord DHT

This section points out, mathematically, using the dispersion properties of the hash function, that these features within the DHT are the ones that implicitly yield load balancing among the nodes, that provide the service of processing their clients' requests.

The dispersion or hash functions, such as SHA-1 [38] and MD-5 [39], which are also used in DHTs such as Chord [9], are functions that map a very large domain of values into a much smaller one, while needing a very short time to retrieve the hashed value, based on its hash identifier.

So, if the hash function is denoted by f, we have:

$$f : LargeDataSet -> IDSet$$

and it is known that any hash function f has the following important property:

$$cardinal(LargeDataSet) \ggg cardinal(IDSet).$$

Moreover, if f is a good hash function, the input values that yield the same value $y \in IDset$, are very different, i.e., the difference between them is very large:

$$\forall y_i, y_j \in LargeDataSet, f(y_i) = f(y_j) :$$

$$|y_i - y_j| \to \infty.$$

Besides that, the property of having almost equal number of input values that yield each hash identifier also stands:

$$\forall y_i, y_j \in LargeDataSet, 1 \leq i, j \leq n, y_i \neq y_j :$$

$$|cardinal(f^{-1}(y_i)) - cardinal(f^{-1}(y_j))| < \epsilon,$$

where $\epsilon \to 0$ and N are a large number, $N \leq cardinal(IDSet)$. For the hash function used in the Chord DHT, N is a power of 2, i.e., $N = 2^M$.

This actually means that each "bucket" of data, which is each entry in the hash table, "contains" almost the same amount of data and that each pair of pieces of input data that reaches the same "hash bucket" is extremely different. It also follows that the probability for a piece of data to find itself in a hash entry is equal for any hash entry, which gives an even distribution of data in a classical hash table and also in a *Distributed Hash Table* (DHT).

From the improved highly available system's point of view, if many peers have been launched, each DHT peer k would handle $cardinal(f^{-1}(y_k))$ values, thus, from the equations above, each peer would have almost the same load as the other peers in the DHT. This mathematically argues the load balancing introduced by the Chord DHT in the hereby presented improved solution for high availability.

2.2. System Scalability Based on the Intrinsic Properties of the Chord DHT

To explain mathematically the scalability of the system based on distributed hashing, one needs to show that its performance degrades very slowly, with respect to the number of peers participating in the DHT.

If examining the Chord routing protocol in [9], note that, each peer p, having Chord ID ID_p, which has a routing table composed of the so-called finger nodes, the system has an interesting property. If the considered maximum dimension of the Chord ring is $N = 2^M$, i.e., $cardinal(IDSet) = N = 2^M$, that is, the binary representation of each peer's ID contains M bits, the following is true for the maximum M entries in the Chord finger table, used for routing messages via the DHT from source to destination:

$$\forall k, 0 \leq k \leq M-1 :$$

$$Finger_k \approx ID_p + 2^k,$$

where the approximation means that $Finger_k$ points to the peer which has the closest peer ID to $ID_p + 2^k$. Of course, the finger table also contains the clockwise successor node of peer p on the Chord ring, but also its predecessor on the Chord ring. For small values of the deployed number of peers, i.e., $No_{peers} \lll N$, there might be situations where more finger peers are identical, i.e.,

$$0 \leq i, j \leq M-1, i \neq j : Finger_i = Finger_j.$$

In most of these cases, the number of hops to route a message from the current peer p towards its destination is less than its maximum possible value. Note that in the Chord DHT [9], *with high probability*, the maximum possible number of DHT routing hops for a message to move from source to destination is in practice $\approx log_2(No_{peers})$.

Based on the FingerTable, which is actually the Chord decentralized routing table, which is a sorted list with respect to the fingers' Chord IDs, the distributed algorithm which describes how the routing is performed is described in Algorithm 1, below:

Algorithm 1 ChordImplicitLoadBalancing.

for $iteration = 1, 2, \ldots \infty$ **do**
  if received new routing request towards $ChordID(destination)$ **then**
    if $(ChordID(predecessor) < ChordID(destination))$ and $(ChordID(destination) \leq ChordID(self))$ **then**
      Reply directly to the (new) client; this peer handles its processing request
    else
      for $k = 0, 1, 2, \ldots, log_2(MAX(ChordID)) - 1$ **do**
        if $ChordFinger_k \leq ChordID(destination)$ and $ChordID(destination) \leq ChordFinger_{k+1}$ **then**
          Route message to ChordFinger_k
        end if
      end for
    end if
  end if
end for

This algorithm is distributed and independently run on each DHT peer, which means that the routing is performed as follows: each peer compares the Chord ID of the destination with the peers in its finger table, and finds the closest finger to that destination ID, i.e.,

$$ID_{FingerTable[closest]} \leq ID_{destination} \wedge ID_{FingerTable[closest+1]} > ID_{destination}.$$

The decentralized distributed routing stops when the current peer, that needs to route the message, realizes that *it* is the peer responsible for the Chord ID $ID_{destination}$.

In order to identify the theoretical maximum possible number of hops needed to route a message from current peer p to its destination, one needs to know that when routing a message, the peer compares the entries in its finger table with the ID of the destination, and chooses the closest finger ID as the next hop to route the message. In the best case, the destination is in the finger table of the current peer, which means that there is only one more hop towards the destination. In the worst case, each finger that becomes the next hop towards the destination should perform the steps described here, in a distributed recursive manner, until the destination is reached. Note that, in many cases, the ID towards which a message is routed is not the ID of any peer in the network, and each peer is aware that it is responsible for the Chord IDs also called *data keys* between its predecessor and itself:

$$ChordID_{predecessor(p)} < ChordID_{key(p)} \leq ChordID_p.$$

If analyzing the maximum number of steps needed by the DHT routing algorithm for Chord [9], one concludes that, with high probability, for a sufficiently large deployed DHT, at each step s the Chord routing algorithm most likely identifies a *finger node* having the digit in position $M - s$ from its binary representation to be identical to its corresponding binary digit of the base 2 representation of its Chord ID ($ID_{destination}$), i.e.,

$$\forall s, 0 \leq s : log_2(|ID_{destination} - Finger_s|) < M - s.$$

However, we know that:

$$\forall ID_{destination} \in IDSet : 0 \leq ID_{destination} < 2^M,$$

i.e.,

$$\forall s : 0 \leq s < M.$$

This means that the fact that the number of hops between each source and destination node, when routed via the Chord DHT, is, with high probability, no more than $log_2(DimensionOfChordRing)$ [9]. This is in practice, from the performed experiments, no

more than $[log_2(No_peers)]$, with high probability, where $[X]$ represents the integer part of the real number X.

2.3. The Novel Improved Model of Splitting the Work Around the Nodes

The improved model for highly available request processing is to split the work around the nodes providing the service. The solution is based on a front-end load balancer, which interacts with a set of nodes, each of which being also a DHT peer, and each of which being also a simple node providing the actual service of processing clients requests. In order to avoid a *single point of failure*, the load balancer chooses the server node to currently handle and process a client's request, by computing a hash of the client's name, and by then forwarding the client request to the service layer running of the DHT Chord peer, that is responsible for the computed hash key of that client, as is detailed in [9]. Then the client–server conversation between *that* client and the chosen server happens in a classical manner. As explained in Section 2.1, this is an important load balancing step, which also allows the system to be transparently scalable, since if peers become overloaded, new peers running the service can be added to the DHT. The Chord DHT is actually used as a *transport communication layer* in the solution for improving the service's availability when processing the clients' requests. In order to address fault tolerance, besides the DHT, the load balancer is also of help. If the communication line between a client and the server, that the client already communicates with, falls, the load balancer re-catches the client's attempt to reconnect to the service. Consequently, it redirects the client to another, functional peer, providing the service of processing clients' requests, thus finally fulfilling this client's processing request.

The load balancer will actually control the access to the Chord network using DNS node configurations, as will be described in what follows.

2.3.1. The Load Balancing Component

The purpose of the load balancer is to offer its clients a single entry point to the system, by defining a robust interface, which is also easy to use by the clients. It has a double role, since the load balancer is also responsible to hide the nodes that form the Chord ring, nodes which collaborate to offer together the desired service. The idea of a load balancer in front of a Chord network is not new, because the *bootstrapping* mechanism described in [9], for new nodes to join the DHT, is based on links with the DNS layer of the node. This way, if a node wishes to join the Chord network, if the node knows only the name of the service, it will perform an *A record* DNS interrogation [40], to find out the IP of a machine that contains a Chord peer for sure. The choice for the current implementation was to introduce in the DNS entry of the node that offers the highly available service, an A type record, which will solve the load balancer's IP (or set of IPs), namely the entry point to the distributed service. This approach does not restrain the decentralization and availability hypothesis, since the improved solution presented here strongly resides on the one used for Chord bootstrapping, and Chord is considered to be one of the most decentralized and available distributed networks. In order to obtain further decentralization of the service, such as not to introduce a *single point of failure* in the load balancer, while also not overloading it, we can also use, besides the DNS A records, DNS NAPTR and SRV records, as can be seen in Figure 1. With these entries, it is possible to specify more IPs of machines on which instances of the load balancer is running, being also able to establish preferences and priorities for contacting these nodes, and also relative load percentages for requests addressed to each of the load balancing nodes.

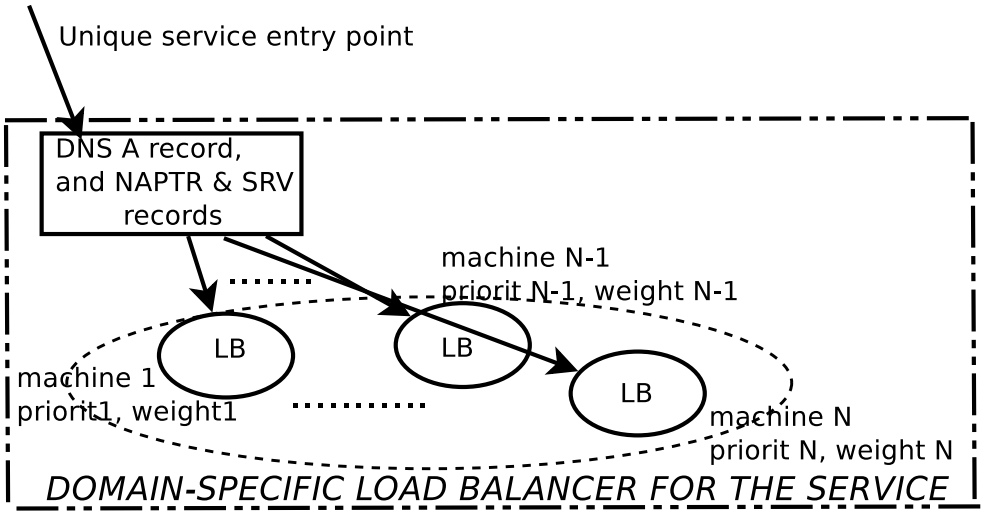

Figure 1. Single entry point for the DNS configuration.

To summarize the idea, we have a pre-fixed group of nodes, called load balancers, among which at least one is always active, nodes that are registered under the same DNS name and offer the load balancing function by acting as a team of equals. This implementation for a service with high availability improvements defines weights in DNS records, for each of these load balancers, to specify maximum admissible loads and usage priorities, all of which being computed based on the processing capacity of the machine, the network quality and, last but not least, on the probability that the node is functional at a given moment. Note that the load balancing nodes do not offer the effective service functionality, but they redirect the clients to active nodes that effectively offer the service, and which are members of a Chord ring, which are also logically connected via the clients' data saved in the highly available database [7] involved in the solution presented here.

From an architectural point of view, the load balancer contains the following components, which can be viewed in Figure 2: (1) a component representing a link with the DNS configuration; (2) a component to map the clients' requests, and thus the clients, to the nodes belonging to the Chord network; (3) a component that is able to "speak" to the Chord network, being able to ask the designated service node to serve the client, by *routing* the client's request to that service node. This is the way in which the client–server connection is finally being established in this session. Note that, even if during a certain session the client should be handled by the same DHT peer running the service, at a new session initiation request from the same client, it could be transparently handled by another service node, the persistence of the client's data being assured by the decentralized highly available database involved in this solution. In case of a failure from a service node, the situation is treated as a timeout situation and the load balancer will help in initiating a new session of the client with an active peer providing the service.

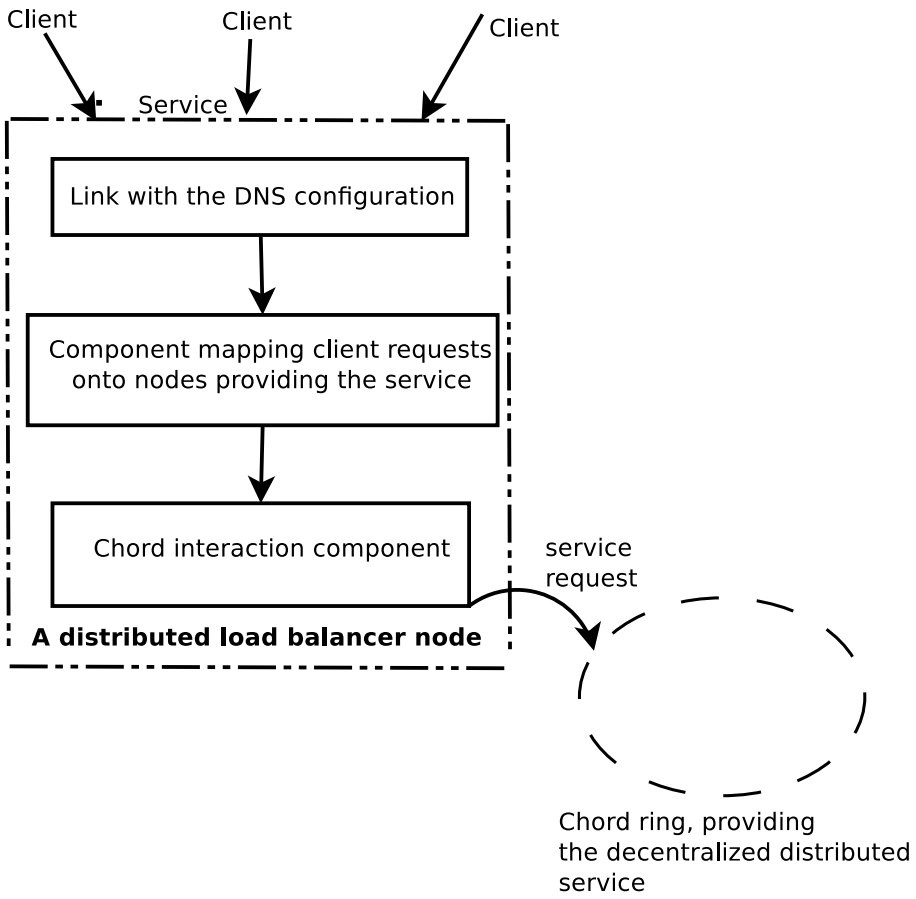

Figure 2. Individual architecture of a load balancing node.

The dispatcher performs the following steps, described by Algorithm 2, when trying to map a new client request onto a DHT peer running the service. Note that in the algorithm HA stand for "high availability".

Algorithm 2 DispatcherLoadBalancing.

1: **for** $iteration = 1, 2, \ldots \infty$ **do**
2: Wait for a *new* client having a request to be processed by the HA service;
3: $client \Rightarrow uniqueChordID(NewClient)$;
4: Send a message to the DHT Chord ring with destination $ChordID(NewClient)$;
5: **end for**

Each Chord peer by default waits in an infinite loop for new communication messages via the DHT layer. As soon as it receives a message towards a destination, based on its routing table whose entries are called fingers, each Chord node sends the received message to its next hop, until the messages reaches its destination, i.e., either the destination specifies exactly the peer's Chord ID, or the range of Chord IDs, called keys, that do not denominate any other Chord peer, but are the Chord IDs that the current peer is in charge of, as already described in Section 2.2. Given that the peers are placed on the Chord logical ring in the increasing numerical order of their Chord IDs, the range of the keys that a Chord peer is in charge of starts from its predecessor's ID on the ring plus 1, i.e., $ID_{Predecessor(IDPeer_{crt})} + 1$, and ends with the current peer's Chord ID, itself, i.e., $IDPeer_{crt}$.

The link with the DNS configuration for the load balancer renders the "single system image" (SSI) for the distributed service. It defines a standard and easy to use interface, in order to be able to access the service, without overloading the system entry point.

The component that maps clients onto service nodes has the unique role of computing the Chord key of each client, by using the hash function used within the Chord DHT. Further, this component is the one responsible to construct, from the client's original message to the service, the message that will be sent to the service node, in the "language" understood by Chord, i.e., to route the message to its destination via the DHT transport layer.
The Chord component of the load balancer knows that, based on the message the *component that maps the clients onto service nodes* provides, it should send a request using Chord mechanisms, towards the peer that will have to serve that client. The identification of the service node responsible for the client is made using Chord keys, by computing a hash of the client's name.

2.3.2. The Peer-To-Peer Chord Network

As already presented in Section 1.1, a Chord network is a totally decentralized DHT-like network. In any Distributed Hash Table (DHT), the nodes' tasks are split based on the nodes' and task's unique identifiers, generated by the hash function, and *this* implicitly provides a *load balancing* among the nodes' responsibilities, i.e., the number of data keys the node needs to handle [9], as already explained in Section 2.1. In fact, it is exactly this *special feature* of the Chord DHT to implicitly balance the load by itself, that this novel solution aims to use, together with the previously mentioned load balancer, in order to obtain a uniform distribution of the load among the nodes providing the service and to finally make the request processing service highly available.

From an architectural point of view, a node from the Chord network has two parts: (1) an self-extending enhanced Chord component, for better persistence and integrity of the data that it holds; (2) an instance of the distributed service, since this novel solution implies that each member of the Chord ring also runs an identical, unaltered instance of the request processing service. In other words, in the current novel approach for high availability, the exact application that provides the request processing service is instantiated on every node of the Chord ring and, by using the Chord hash function, the clients are mapped onto different peers in a balanced manner. The *good dispersion properties* of the DHT's hash function will enable an equitable partition of the distributed network service's clients among the Chord peers that offer the service, so that none of the peers will be stressed with too much load. In the current improved approach, in case the load of peers becomes too high, the system should launch a new peer in the overloaded area of the Chord ring, as described in our team's previous work [41,42].

The partitioning of the clients is performed considering the following logical mapping: knowing that each peer p is responsible for a set of keys, for which it should also store the corresponding data, the *responsibility of a peer for a key* is redefined and extended, as being the responsibility of that peer to handle the *service client* that is associated to that key. This way, once a client has been handled by a peer, as it is described in Figure 3, we impose the rule that the client will be always handled by that peer, throughout the session. An exception is in the case that at a certain moment that peer fails, in which situation *its entire functionality* will be taken by another previously designated backup peer, according to the DHT rules of replication on neighbors. The backup peer will also have to possess the up-to-date journal of the current client's session and data.

Note that the keys that peer p is responsible for are also Chord IDs and each peer is responsible for the Chord IDs that are situated between its predecessor and himself, i.e., the keys that a peer p is responsible for are:

$$ChordID_{predecessor(p)} < ChordID_{key(p)} \leq ChordID_p.$$

Given the way of serving clients involving the load balancer from our improved solution, whenever it is possible, for each client's request to the service, except for the initiating request, the client is able contact directly the peer that is responsible for it. That peer will also ensure to store the data related to that client such that it is persistent and

reliable, in case the peer itself should fail and thus disappear from the network. In case a peer disappears from the network, the clients that used to contact it directly, without using the load balancer any more, will notice that the peer's IP address can no longer be reached, and will thus naturally try to contact the service again, via its DNS name. This DNS interrogation will lead the client again to a load balancing node, which is a part of the load balancer. The load balancer will hand over each client to the new peer responsible for that client, by computing the hash of the client's name again, and by transparently routing again towards the peer that is responsible for the client's key, i.e., the Chord ID obtained by hashing that client's name. The currently identified peer will know, from the persistent and journalized data that were kept in the highly available database, how to continue the dialogue with that client, without affecting too much its quality of experience. A very important aspect here is that, for the service's clients, the entire Chord layer from the decentralized service, that processes requests, is totally transparent. Each service instance running Chord peers treats the improved decentralized service's clients identically to the situation when the service was centralized. By decentralization, the service's availability to more, old and new, clients is highly improved, since we have *removed* the single points of failure.

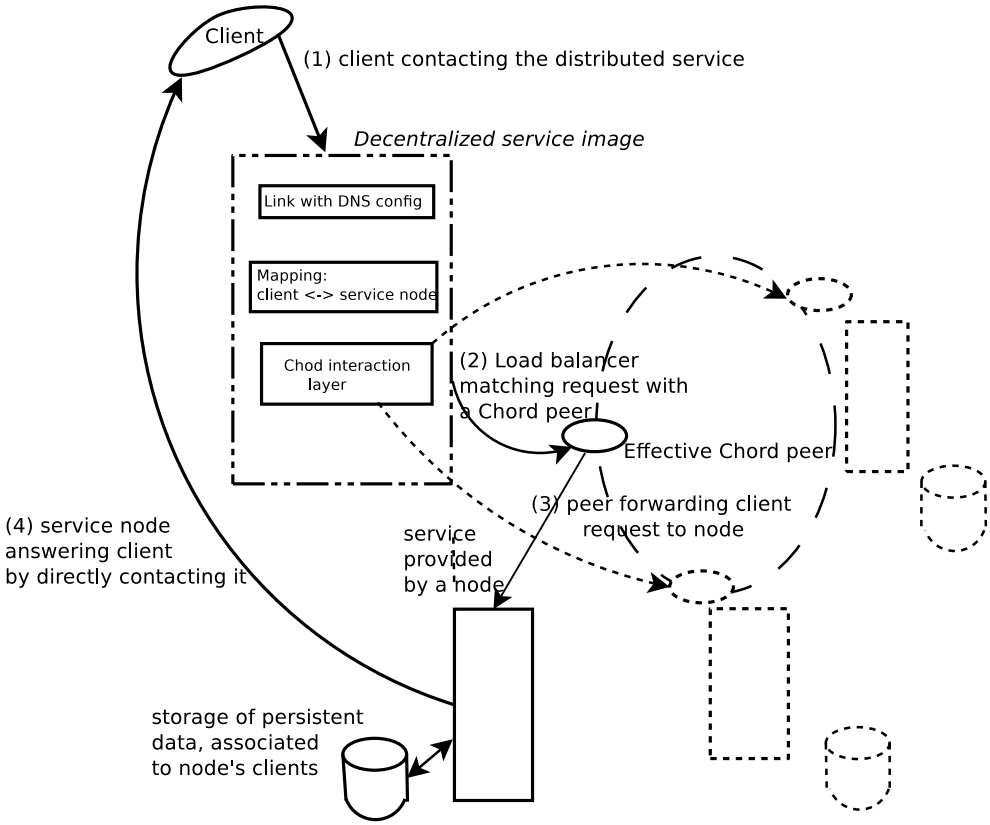

Figure 3. Generic handling of a client by the decentralized distributed service.

Since another peer is able to take an unavailable or failing peer's clients, to continue to offer them the service, there exists the *strong need* to keep any client's data persistent within the entire decentralized distributed system of service processing nodes. For this, a *decentralized highly available* database is used, also based on the Chord DHT, which, in the improved vision of a solution for highly available services, represents a *necessary basic condition*. This database, previously designed by our team, which is also based on the DHT, handles persistence, scalability and integrity aspects for the data.

The novel method presented in this section, which can be schematically observed in Figure 4, should be generic enough to make any service highly available, with minimal

modifications, by using the deployed load balancing nodes, contained in the deployed service-aware Chord layer.

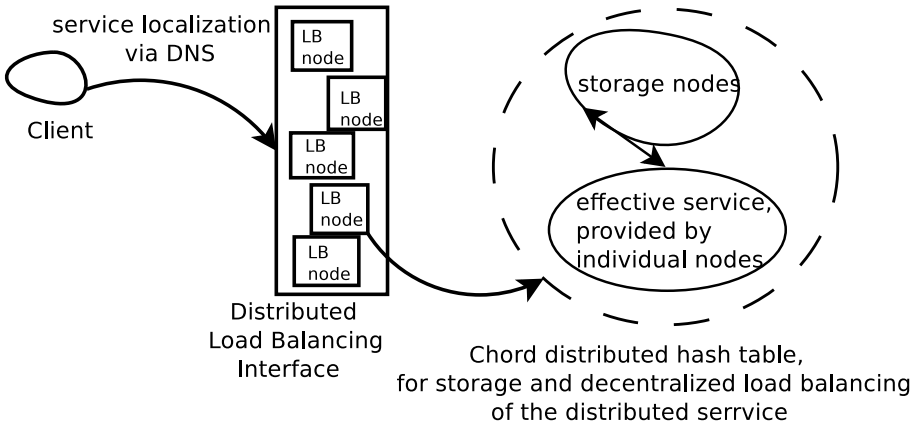

Figure 4. Generic method for distributing and decentralizing a service, in order to make it highly available.

2.4. The Pre-Existent DHT-Based Distributed Database to Reliably and Persistently Store the Data Belonging to a Highly Available Service's Clients

Our team's previously published contribution to the field of distributed databases, which this solution relies on, resides in the fact of having identified a way to solve the problem of data loss, and more importantly of critical data loss, by storing the data on newly added nodes to the underlying DHT infrastructure, which is shared by the service and the database described in [7,16,17]. Even if the obtained infrastructure is not 100% secure, it offers a safety level for the data that is significantly improved from a classical DHT, since it is able to identify and exclude malicious peers from the DHT. The extra safety is also based on another important aspect of this previous work, i.e., the possibility to have a variable replication degree for the data, which allows the system to integrate new storage nodes *only* when this is necessary. This fact is governed by the internal dynamics of the request processing service, with which the storage layer is in constant dialogue and interaction via the DHT communication layer.

Note that the keys of data that a storage peer p, belonging to the highly available database, is responsible for are also Chord IDs and each peer is responsible for the Chord IDs that are situated between its predecessor and himself, i.e., the keys that a peer p is responsible for are:

$$ChordID_{predecessor(p)} < ChordID_{key(p)} \leq ChordID_p.$$

The DHT's hash function ensures that each storage node is loaded equally and participates evenly to the task of storing the entire highly available service's data.

The extra time introduced by this database solution, for a group of nodes connected by a local network, is pretty low, especially when also taking into consideration the benefits that the enhanced replication mechanism brings. All this allows us to consider that this decentralized distributed database prototype, based also on the Chord DHT, is a good candidate to reliably store the data for a competitive highly available service that needs to process its clients requests.

3. Results

As detailed in Section 2, the key to a good service decentralization is a good, flexible, balanced and scalable distribution of the data, which most importantly needs to be *persistent*. This is why a very important part of this improved solution for decentralizing and distributing a service, in order to make it highly available, is to use a very efficient decentralized distributed database [7,16,17]. For the entire improved solution of a highly

available service, we are most interested in not introducing too much overhead by adding this solution's extra components, which yield high availability for the service.

The elements that this novel solution aimed to improve, in order to make the service highly available, are: (1) static or dynamic *load balancing*, in order to maintain a high availability for the service, even in the context of increased load, due to a larger number of clients; (2) *fault tolerance*, by proving complete functionality even in the presence of failing nodes; (3) *scalability*, by DHT self-extension described and evaluated also in [41,42], in order to be able to cover a higher number of clients and the increased needs of persistence, that critical sensitive data have, in order to obtain a generic method for making a previously centralized service to be highly available, with minimum changes at the service level.

Note that the load balancing feature of the novel solution presented in this paper has been explained theoretically, in Section 3.1. The experimental evaluations performed are *orthogonal* to each other, since it is easier to follow if the evaluation is performed separately, for each important feature of this novel approach for improving the service's availability.

The testbed used to evaluate the improved solution, for a highly available service processing requests, involves nodes in our team's datacenter, which has a front end and a large collection of working nodes behind it. We have focused on proving the functionality of the model for a highly available service described in Section 2. This is why we have launched the load balancing nodes on the datacenter's front end and the effective service peers on *heterogeneous* working nodes that we usually contact only via the front end. Note that, in order to obtain the best performance, the current novel solution for high availability was tested by running it on nodes that were fully loaded by the service. Each load balancing node has an equal probability to be contacted. We have gradually deployed a Chord network, having a maximum of 128 peers, and on top of each peer there is also the application service that is running. The network that interconnects the peers, and the network that connects each of the peers with the load balancers representing the front end of the service, is a high speed network. The observed ping delay when contacting one node from another node in the datacenter does not exceed 2 ms, and is usually around 1 ms if the nodes are not in the same rack, and around 0.3 ms if the nodes are in the same rack. The decision to stop at 128 nodes is based on the observation that, in practice, this number of nodes is eloquent for showing system scalability, since, to the author's knowledge, most of the common highly available applications are deployed on a few nodes, certainly no more than 100.

From the performed experiments, notice that the extra delay for fully processing the client's requests by the service, introduced by deploying it on a DHT of about 128 peers in our datacenter, has a mean value no larger than 15 ms, which makes it negligible and thus validated the fact that this improved novel solution for high availability is transparent to the service's clients.

3.1. Practical Evaluating of the System's Scalability

Indeed, from the evaluation experiments performed in our datacenter, which involved a maximum of 128 nodes, the results show that the number of messages exchanged by the peers in order to find the destination of a message routed via Chord, i.e., to find the Chord node that is responsible for handling and processing a particular client's request, was no larger than 7 hops. It is obvious that $7 = log_2(128)$, meaning that our experimental findings are in accordance with [9]. If extrapolating the measurements performed for 128 peers to about 1024 peers, we obtain no more than 10 hops, or inter-peer messages, to reach a Chord ID destination, which means that for a number of peers that is $10x$ greater than the number of peers that we have launched, the number of exchanged Chord messages remains almost the same, since the integer part of the logarithm function grows very slowly.

Moreover, if identifying a very loaded peer by the requests of the clients it must serve, there exists the means to decide deploying a new peer, in order to re-balance the load. Thus, the extended network, integrating this new peer, will have almost the same system inter-peer routing communication delay, since the maximum number of routing

hops for messages among peers is represented by the graphic representation of the function $[log_2(K)]$, where $[X]$ takes the integer part of the real value X.

The performed experiment to show the novel solution's improved *scalability* was the following: the load balancing node was deployed and then, gradually, 1, 2, ..., 127 and 128 improved Chord peers were deployed to help handle the already loaded service that needs to process its clients' requests, as described in Section 2. This was actually the way to gradually instantiate an infrastructure that is able to provide improved high availability via scalability for the request processing service. Each time, we waited for the infrastructure to converge to a fully connected DHT. On this logical and physical infrastructure, messages were sent via the DHT communication layer towards various nodes in order to see an increasing number of DHT routing hops and the time needed to traverse those hops. As can be seen from Figure 5, the number of messages was no larger than 7, i.e., as expected, the base 2 logarithm of the maximum number of deployed peers: $log_2(No_{peers})$. From Figure 5, too, note that the maximum routing time needed to send any message via the Chord DHT distributed routing infrastructure was no larger than 8 ms, which represents a reasonable overhead for the benefit of increased scalability. The obtained results are according to the authors' expectations, since, when measuring the maximum inter-node communication delay within our cluster, it was of no more than 2 ms per node pair.

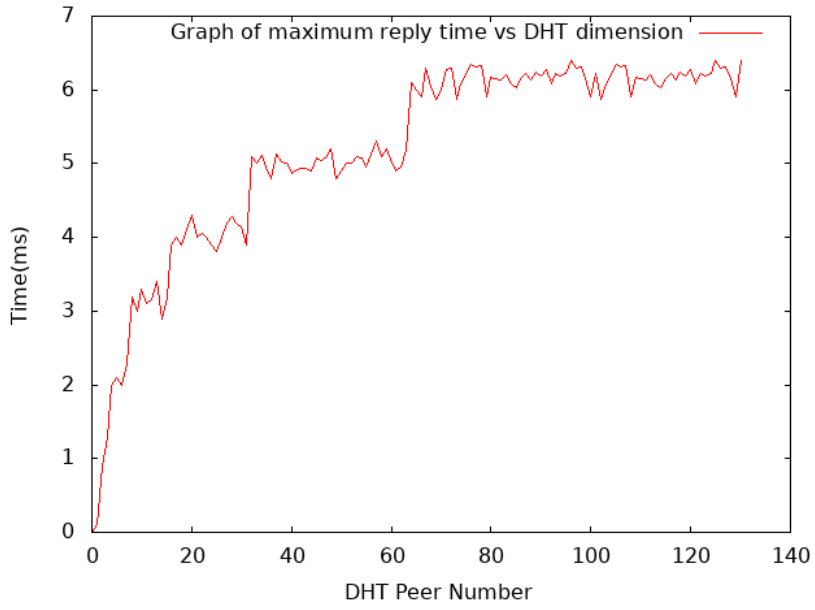

Figure 5. The extra time needed by the system to reply to the client, via the DHT.

The almost stair shape of the plotted curve in this work's experimental results is in tight correlation with the shape of the integer part of the logarithm of the number of DHT deployed nodes, i.e., $[log_2(No_{peers})]$.

3.2. Practical Evaluating of the System's Fault Tolerance

This section evaluates the fault tolerance of the system by means of experimental validation. The following experimental steps were performed: a number of peers ranging from 0 to 128 was deployed, clustered in three different groups of nodes of our datacenter. For each of the DHT dimensions ranging from 1 to 128, we considered both: (1) the case when a new peer *joined* the network, and have evaluated the time until that peer or the data keys it must handle could be reached, by routing via the DHT; (2) the case when a peer *left* the network, and evaluated the time until its backup peer took over for the IDs that belonged to the failing peer, as was also described in the replication mechanism of Chord [9]. Note that, as already explained in Section 2, the keys that a peer p is responsible for are also Chord IDs. Moreover, each peer is responsible for the Chord IDs that are

situated between its predecessor and himself, i.e., the keys that a peer p is responsible for are: $ChordID_{predecessor(p)} < ChordID_{key(p)} \leq ChordID_p$.

Note, from Figure 6, that the time intervals needed for the DHT infrastructure to stabilize, i.e., to be entirely aware of the topology changes, are clustered around three centers. Remember that the DHT had been deployed in three areas of our datacenter, on heterogeneous types of nodes. It has already been mentioned in this section, that the inter-node communication delay within a single area is very low, i.e., 0.30 ms, while the communication delay between each pair of different areas of the cluster is of between 1–2 ms. If three areas in the datacenter are considered, besides the pairs of nodes that belong to the same area and thus communicate extremely fast, we can form only three other types of pairs $(peer_{source}, peer_{destination})$, i.e., C_2^3 types of pairs, where the area for each source peer is different from the area of each destination peer. These three areas in the plot, that can be seen in Figure 6, correspond to the total stabilization time needed by peers of one datacenter area to learn about the infrastructure changes in each other datacenter area. The infrastructure changes involve, with equal probability, both new peers joining the DHT and failing peers, leaving the DHT.

The explanation for the slight spikes in the obtained plots is the fact that not all peers have been deployed within the same rack, and the communication delay within a rack is of about 300 microseconds, while the inter-rack communication delays experienced in our experiments is between 1–2 ms.

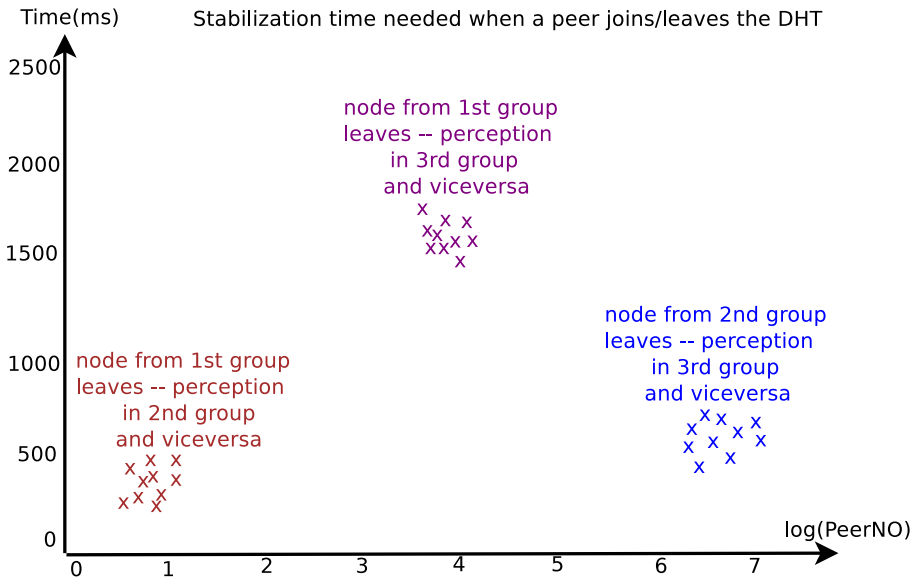

Figure 6. The extra time needed by the DHT to re-stabilize after a peer join or a peer leave.

3.2.1. The Improvement in Fault Tolerance

The proposed new model based on DHT enables the entire system, running on a collection of peers, to become aware rather fast of the changes in the peer topology, i.e., either peer join or peer fail/leave, which is extremely difficult without a DHT. From Figure 6, in the worst case scenario, the model needs no more than 2 s for *every* peer to update their DHT routing tables, thus becoming aware of the new topology. Note that the maximum of 2 s, which has been measured, is needed only for peers that find themselves far from each other on the Chord ring, but for neighbor nodes, which are also backups for each other in Chord, the time is far lower. The delay for any peer to find out that the peer for which it was the backup has left the network is most often around the value of *the sum* between the configured *stabilization time* within the Chord network and *the time* needed *to route* a message one hop away via the DHT, which is between 1–2 ms, as can be seen from Figure 5. Moreover, note that the Chord stabilization time interval, which is a time interval when periodically each peer pings the peers in its finger table, is a value that can be configured in

each peer, according to the system's best balance and needs. Since this solution is, to its authors' knowledge, the first to use the DHT for reliable request processing and not just for reliable data storage, it is the first to benefit of the DHT derived features of *topology update awareness*, in a time interval that can be configured to be sufficiently low, if this is vital to the request processing service.

3.2.2. The Improvement in Load Balancing

The current solution balances the load of the entire set of peers transparently and fast, based on the powerful properties of the *distributed hash*, which have been described into detail in Section 2.1. Any hash table classically gives a fast and constant means to access various data, while a good *distributed* hash table, such as Chord, succeeds to fulfill this purpose by *evenly* distributing the keys among more peers belonging to the DHT. The *even* distribution of the keys on peers, which actually directly translates into peer *load balancing*, is a direct consequence of: (1) the features of a good hash, such as SHA-1 [38] and (2) the smart idea behind the DHT such as Chord, to have a unified space for the peer IDs and the data keys. Moreover, since it relies on both of them to function well, load balancing finds itself at the confluence of fault tolerance and scalability, both having been evaluated experimentally in this section.

3.3. Evaluation Conclusions

In conclusion, seen as a whole, the extra time that the decentralization solution for a highly available service running in our datacenter introduces, in comparison to the single node server, is not very high. The extra time experienced by the client when it receives the reply to its first message to the request processing service, is actually the time needed to route a message in the Chord ring of about 128 peers. This translates in the fact that there is a certain cost for the benefits of load balancing and decentralization, fault tolerance and scalability that Chord brings with it, but this time cost is perceived only for the first message of the session initiated by the client, and also in cases of re-connection. For the rest of the session, there is no extra delay introduced by the new layers of the decentralized distributed service. The enhanced Chord ring is actually the main component for decentralizing the distributed service, via its intrinsic unique mechanisms [9], also previously evaluated in [42].

The authors wish to point out the fact that, from the current implementation and experiments for improving high availability, *the DHT is essential* in the observed increase in availability for the distributed service. In itself, any DHT is a scalable system, balanced and fault tolerant, also allowing many improvements and extensions, some of which also being new contributions described in this paper and evaluated in this section. During the measurements that have been performed in this work, the aim was to minimize the time impact of the components and the algorithms that maintain a highly available infrastructure for the request processing service, throughout the entire evaluation process.

4. Discussion

Before this work, DHTs have only been used to enable data storage that is fault tolerant, balanced and scalable, not addressing *the data processing* aspect at all. By analyzing the existing bibliography and the benefits that the DHT brings to a decentralized distributed database, this paper reaches the conclusion that one could exploit and use the same DHT data storage benefits, for handling data processing requests in a highly available manner. The idea behind this is to use the dispersion function, i.e., *the hash*, in order to equally spread the client requests among DHT Chord peers, that also run the service on top of them. The next step was to adapt the system's architecture in order to "squeeze out" of the DHT and extrapolate to the entire service, the features of scalability, load balancing and fault tolerance. This way, with high probability, if the *mathematical hash function* is *well-designed*, each Chord node would handle about the same number of clients; moreover, if we use a self-extending enhanced Chord infrastructure, in case any Chord node becomes overloaded,

by processing its clients' requests, the DHT is able to self-extend exactly in the point where it is needed, thus providing both load balancing and scalability. Given that, in the hereby proposed novel solution, scalability comes with a small cost of extra communication, and knowing that in Chord, with high probability, the number of communication hops has a *logarithmic* growth with respect to the number of peers: $[log_2(No_{peers})]$, Section 2 explains and Section 3 experimentally confirms that the novel solution described by this paper is indeed a scalable, fault tolerant and balanced system, with negligible communication overhead. Last but not least, if any of the Chord nodes suddenly becomes unavailable, the improved availability solution handles this situation by having its clients' *data replicated* on neighbors. Consequently, by using the *stabilization* mechanism within Chord, the clients, for whom the failing node was responsible, experienced only something similar to a packet loss, a problem that can always be fixed by a retransmission. This retransmission will be taken over by the new responsible peer for that client, via the load balancing dispatcher, which is the front end of the service.

5. Conclusions

The current work introduces a novel method to decentralize not just storage, that is classically addressed by DHTs, but also *processing*, thus making services highly available, based on the fact that the data they use are already stored in a fault tolerant, balanced and scalable manner. This work originated from our team's past experience with highly available storage and DHTs, which this work aims to put into a new light, in a hot, up-to-date context, the one of rendering a single system image for a group of nodes collaborating in order to provide a common service. In the previous sections, we have identified and analyzed the problems that can impact the availability of a distributed service. The identified and analyzed aspects are: fault tolerance, scalability and load balancing. For all of these, throughout this work, we offered unified novel solutions that offer availability improvements when applied to any request processing server; these improvements were based on a decentralized DHT architecture.

The objective of this work had been from the very beginning to explore and extend the capabilities of the Distributed Hash Tables (DHTs), since the authors have felt that there can be more to them than just storage. The peer-to-peer systems and especially the DHTs, such as [9,11,12,33] have been introduced for quite some time, and have been the starting point for data sharing in distributed filesystems [1,19], but also for file sharing in home networks [43,44] or even in early IP telephony solutions [45]. Very recently, the DHTs have come into focus again, in storage and replication [6], in name resolution and resource lookup [2,3,13], but also in routing [15], and even in blockchain [2,3] design. Nevertheless, despite their very good quality to balance loads, a feature which derives from the properties of the hash function, the authors have not identified DHTs and peer-to-peer networks to be used in order to simplify and make more available services of data processing. On the other hand, there is a growing need on the market for high availability in request processing services, such as for example in IP telephony and conferencing, during a time when people have been forced to used more virtual meeting means than physical ones. This is where the current original solution, for highly available processing of requests can be very useful and applicable, could become useful, while keeping in mind that the obtained high availability introduces a small but manageable delay, due to the DHT extra layer that is found between the transport layer and the application layer from the TCP/IP protocol stack. Using peer-to-peer networks as an overlay network, i.e., abstract network, is not new, this concept being often connected today also with virtualization [46]. The authors have used the DHT layer similar to how VRRP (Virtual Router Redundancy Protocol) is classically used in order to transparently provide redundancy and finally high availability in a network [47] the enormous plus of DHTs being that they are far more flexible.

In the future, we would like to evaluate the improved model introduced by this paper also on a more geographically distributed architecture, in order to have even less functional dependencies among some sets of nodes, but also to see the overhead that this novel

DHT-based improved solution introduces in such a deployment. Moreover, we would like to apply this recipe for a set of popular distributed request processing services, to turn each of them into highly available services, with minimum or no changes to the interface to their clients.

Author Contributions: Conceptualization, V.I. and N.T.; methodology, V.I.; software, V.I.; validation, V.I. and N.T.; formal analysis, V.I. and N.T.; investigation, V.I.; writing—original draft preparation, V.I.; writing—review and editing, N.T. and V.I.; supervision, N.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: The authors wish to thank Iosif Ignat for his advice and contributions to this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dabek, F.; Kaashoek, M.F.; Karger, D.; Morris, R.; Stoica, I. Wide-area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles, Banff, AB, Canada, 21–24 October 2001.
2. Hassanzadeh-Nazarabadi, Y.; Küpçü, A.; Özkasap, Ö. LightChain: A DHT-based Blockchain for Resource Constrained Environments. *arXiv* **2019**, arXiv:abs/1904.00375.
3. Matsuoka, K.; Suzuki, T. Blockchain and DHT Based Lookup System Aiming for Alternative DNS. In Proceedings of the 2020 2nd International Conference on Computer Communication and the Internet (ICCCI), Nagoya, Japan, 26–29 June 2020; pp. 98–105.
4. Stoica, I.; Shenker, S. From Cloud Computing to Sky Computing. In Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS. '21), Ann Arbor, MI, USA, 31 May–2 June 2021.
5. Monteiro, A.; Pinto, J.S.; Teixeira, C.J.V.; Batista, T. Sky computing: Exploring the aggregated cloud resources—Part i. In Proceedings of the 16th Iberian Conference on Information Systems and Technologies, Chaves, Portugal, 23–26 June 2021.
6. Mohammadi, B.; Navimipour, N.J. Data replication mechanisms in the peer-to-peer networks. *Int. J. Commun. Syst.* **2019**, 32, e3996. [CrossRef]
7. Iancu, V.; Ignat, I. A Decentralized Distributed Database Built on Top of the Chord DHT. In Proceedings of the 8th RoEduNet International Conference, Galati, Romania, 3–4 December 2009; pp. 151–155.
8. MongoDB. Available online: https://www.mongodb.com/ (accessed on 3 January 2022).
9. Stoica, I.; Morris, R.; Karger, D.; Kaashoek, F.; Balakrishnan, H. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In Proceedings of the 2001 ACM SIGCOMM Conference, San Diego, CA, USA, 1–2 November 2001; pp. 149–160.
10. Jenkov, T. Chord P2P + DHT Network Algorithm—Tutorials Jenkov. 2021. Available online: http://tutorials.jenkov.com/ (accessed on 3 January 2022).
11. Ratnasamy, S.; Francis, P.; Handley, M.; Karp, R.; Schenker, S. A scalable content-addressable network. In Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '01, San Diego, CA, USA, 1–2 November 2001, Volume 31; pp. 161–172.
12. Rowstron, A.; Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Beijing, China, 9–13 December 2001; pp. 329–350.
13. Souto, P.F. Name Resolution in Flat Name Spaces Distributed Hash Tables (DHTs). 2021. Available online: https://web.fe.up.pt/~pfs/aulas/sd2021/at/9dhts.pdf (accessed on 3 January 2022)
14. The Blockchain Technology. Available online: https://www.ibm.com/topics/what-is-blockchain/ (accessed on 3 January 2022).
15. Kunz, T., E.S.; Esfandiari, B. A P2P Approach to Routing in Hierarchical MANETs. *Commun. Netw.* **2020**, *12*, 99–121. [CrossRef]
16. Iancu, V.; Ignat, I. A Distributed Database With Self-Extending Capabilities, to Compensate Exclusion of Malicious Nodes. In Proceedings of the 9th RoEduNet International Conference, Sibiu, Romania, 24–26 June 2010; pp. 240–245.
17. Iancu, V.; Ignat, I. A Peer-to-Peer Consensus Algorithm to Enable Storage Reliability for a Decentralized Distributed Database. In Proceedings of the 2010 IEEE AQTR Conference, Cluj, Romania, 28–30 May 2010; pp. 1–6.
18. Antoniu, G.; Bougé, L.; Jan, M. JuxMem: An Adaptive Supportive Platform for Data Sharing on the Grid. *Scalable Comput. Pract. Exp.* **2005**, *6*, 45–55.
19. Kubiatowicz, J.; Bindel, D.; Chen, Y.; Czerwinski, S.; Eaton, P.; Geels, D.; Gummadi, R.; Rhea, S.; Weatherspoon, H.; Weimer, W.; et al. OceanStore: An architecture for global-scale persistent storage. *SIGPLAN Not.* **2000**, *35*, 190–201. [CrossRef]
20. SPARK. Available online: http://spark.apache.org/ (accessed on 3 January 2022).
21. HADOOP. Available online: http://hadoop.apache.org/ (accessed on 3 January 2022).
22. Davies, A.; Fisk, H. *MySQL Clustering*; MySQL Press: 2006. Available online: https://www.bookdepository.com/publishers/Mysql-Press (accessed on 3 January 2022)
23. Map-Reduce Processing Pattern. Available online: https://en.wikipedia.org/wiki/MapReduce (accessed on 3 January 2022).

24. The Network Filesystem. Available online: https://en.wikipedia.org/wiki/Network_File_System (accessed on 3 January 2022).
25. Nicolae, B.; Antoniu, G.; Bougé, L.; Moise, D.; Carpen-Amarie, A. Blobseer: Next-generation data management for large scale infrastructures. *J. Parallel Distrib. Comput.* **2011**, *71*, 169–184. [CrossRef]
26. Mor, N.; Allman, E.; Pratt, R.; Lutz, K.; Kubiatowicz, J. An Architecture for a Widely Distributed Storage and Communication Infrastructure. 2018. Available online: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-130.html (accessed on 3 January 2022).
27. de Jongh, J. Shared Scheduling in Distributed Systems. Ph.D. Thesis, Technische Universiteit Delft, Delft, The Netherlands, 2002.
28. Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012.
29. Ananthanarayanan, G.; Ghodsi, A.; Wang, A.; Borthakur, D.; Kandula, S.; Shenker, S.; Stoica, I. PACMan: Coordinated Memory Caching for Parallel Jobs. In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, San Jose, CA, USA, 25–27 April 2012.
30. Fan, B.; Lim, H.; Andersen, D.G.; Kaminsky, M. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In Proceedings of the ACM Symposium on Cloud Computing in Conjunction with SOSP, Cascais, Portugal, 27–28 October 2011.
31. Cai, Q.; Guo, W.; Zhang, H.; Agrawal, D.; Ooi, G.C.B.C.; Tan, K.L.; Teo, Y.M.; Wang, S. Efficient Distributed Memory Management with RDMA and Caching. *VLDB Endowment* **2018**, *11*, 1604–1617. [CrossRef]
32. Matri, P.; Alforov, Y.; Brandon, A.; Pérez, M.; Costan, A.; Antoniu, G.; Kuhn, M.; Carns, P.; Ludwig, T. Mission Possible: Unify HPC and Big Data Stacks Towards Application-Defined Blobs at the Storage Layer. *Future Gener. Comput. Syst.* **2020**, *109*, 668–677. [CrossRef]
33. Zhao, B.Y.; Huang, L.; Stribling, J.; Rhea, S.C.; Joseph, A.D.; Kubiatowicz, J.D. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE J. Sel. Areas Commun.* **2004**, *22*, 41–53. [CrossRef]
34. *RFC 3768 (Draft Standard)*; Virtual Router Redundancy Protocol (VRRP). Hinden, R., Ed.; Nokia: Espoo, Finland, 2004.
35. *RFC 2915 (Proposed Standard)*; The Naming Authority Pointer (NAPTR) DNS Resource Record. Mealling, M., Daniel, R., Eds.; The Internet Society: Reston, VA, USA, 2000; Obsoleted by RFCs 3401, 3402, 3403, 3404.
36. *RFC 2782 (Proposed Standard)*; A DNS RR for specifying the location of services (DNS SRV). Gulbrandsen, A., Vixie, P., Esibov, L., Eds.; The Internet Society: Reston, VA, USA, 2000.
37. Andreica, M.I.; Tîrşa, E.D.; Ţăpuş, N. A Peer-to-Peer Architecture for Multi-Path Data Transfer Optimization using Local Decisions. In Proceedings of the Fourth EuroSys Conference 2009, Nuremberg, Germany, 31 March 2009.
38. SHA-1—Secure Hash Standard. Available online: http://www.itl.nist.gov/fipspubs/fip180-1.htm (accessed on 3 January 2022).
39. Rivest, R.L. The MD5 Message-Digest Algorithm. RFC 1321. 1992. Available online: https://www.rfc-editor.org/info/rfc1321 (accessed on 3 January 2022).
40. *RFC 1035 (Standard)*; Domain Names—Implementation and Specification. Mockapetris, P., Ed.; Information Sciences Institute: Marina del Rey, CA, USA, 1987; Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343.
41. Iancu, V.; Ignat, I. A Scalable Solution for Balancing the Peer Load in a Chord DHT. In Proceedings of the 2010 IEEE ICCP Conference, Cambridge, MA, USA, 29–30 March 2010; pp. 297–304.
42. Iancu, V.; Ignat, I. A self-adapting peer-to-peer logical infrastructure, to increase storage reliability on top of the physical infrastructure. In Proceedings of the 2009 IEEE ICCP Conference, San Francisco, CA, USA, 16–17 April 2009; pp. 259–266.
43. Napster File-Sharing System. Available online: http://www.napster.com/ (accessed on 3 January 2022).
44. KaZaA Peer-to-Peer Storage System. Available online: http://www.kazaa.com/ (accessed on 3 January 2022).
45. Skype Peer-to-Peer VoIP System. Available online: http://www.skype.com/ (accessed on 3 January 2022).
46. What Is Overlay Networking (SDN Overlay)? Available online: https://www.sdxcentral.com/networking/sdn/definitions/what-is-overlay-networking/ (accessed on 3 January 2022).
47. Introduction to VRRP and Its Configurations. Available online: https://www.geeksforgeeks.org/introduction-of-virtual-router-redundancy-protocol-vrrp-and-its-configuration/ (accessed on 3 January 2022).