International Journal of Innovative Computing, Information and Control
Volume **10**, Number **2**, April **2014**

ICIC International ©2014 ISSN 1349-4198
pp. 417–436

A CLUSTER-ENHANCED FAULT TOLERANT PEER-TO-PEER SYSTEM

Ciprian Dobre

Faculty of Automatic Controls and Computers
University Politehnica of Bucharest
Spl. Independentei 313, Bucharest, Romania
ciprian.dobre@cs.pub.ro

Received December 2012; revised April 2013

Abstract. *Over the Internet today, computing and communications environments are more complex and chaotic than classical distributed systems, lacking any centralized organization or hierarchical control. Peer-to-Peer network overlays provide a good substrate for creating large-scale data sharing, content distribution and application-level multicast applications. We present a fault-tolerant, cluster-enhanced P2P overlay network designed to share large sets of replicated distributed objects in the context of large-scale highly dynamic infrastructures. The system proposes an original design to achieve efficient implementation results for range queries, fault-tolerance, and message routing in terms of hop-count and throughput, whilst providing an adequate consistency among replicas.*
Keywords: Peer-to-peer, Overlay, Fault tolerance, Clustering

1. Introduction.

Peer-to-peer (P2P) networks are amongst the fastest growing technologies in computing. Such networks encapsulate a set of high-availability techniques, such as data and service replication, load balancing, fault handling, reconfiguration. In the last years, intense research has been done to overcome scalability problems encountered with unstructured P2P networks, where data placement and overlay network construction are essentially random. The interest in peer-to-peer systems is rapidly reaching new areas, such as live streaming or volunteer computing. The evolution of emerging volunteer computing systems is placing an ever greater emphasis on the use of a loosely coupled decentralized architecture, mainly to address concerns of scalability and robustness. This shift in applicability field introduces new challenges, as the new applications have different availability requirements than data storage and retrieval.

This is why, lately, intense research has been done to overcome the scaling problems with unstructured P2P networks, such as Gnutella, where data placement and overlay network construction are essentially random. A number of groups have proposed structured P2P designs to solve the problem of *scalability*. However, such solutions fail to properly support *resiliance* requirements. High availability, in particular, refers to the ability of a system to provide services to the user (when they need them) at a satisfactory quality level, particularly in the presence of *failures*. Recent systems (CAN, Chord, Pastry or Tapestry) enable peer-to-peer lookup overlays robust to intermittent participation and scalable to many unreliable nodes with fast membership dynamics. Some papers [28] express a hope that, with extra *data redundancy*, storage can inherit scalability and robustness from the underlying lookup procedure. More work still [29] implies this hope by using robust lookup as a foundation for wide-area storage layers, even though this complicates other desirable properties (e.g., server selection). However, as [30] shows, trying to achieve all three things – *scalability, storage guarantees, and resilience to highly dynamic membership* –

overreaches bandwidth resources likely to be available, regardless of lookup. The authors' argument is quite simple: simple considerations and current hardware deployment suggest that idle upstream bandwidth is the limiting resource that volunteers contribute, not idle disk space. Further, since disk space grows much faster than access point bandwidth, bandwidth is likely to become even more scarce relative to disk space.

In this paper, we present a solution that combines capabilities of DHT networks with modern techniques, to enable *scalable and fault-tolerant storage and retrieval operations, with storage guarantees, of shared objects by attribute values.* Unlike previous work, our solution is capable to achieve all these together, in fact, with *optimal message routing in hop-count and throughput.* We previously proposed an hierarchical P2P architecture in [14]. In this paper we extended it with: scalability and fault tolerance are supported by a distributed segment tree that manages all metadata and range queries; the communication costs are reduced by a novel routing algorithm that better explores network topology. The properties of the P2P system are demonstrated in various case studies, and we present extensive results.

The remainder of this paper is organized as follows. In Section 2 we investigate and discuss previous work. Section 3 presents the proposed architecture and P2P overlay, which follows in Section 4 by concrete details behind a pilot implementation and a presentation of experimental and validation results. We conclude this paper in Section 5.

2. **Related Work.**

Distributed Hash Tables (DHTs) are decentralized solutions that partition the hash table keys among the participating nodes. They usually form a structured overlay network in which each communicating node is connected to a small number of other nodes. The Content Addressable Network (CAN) is a decentralized P2P infrastructure that provides hash-table functionality on Internet-like scale [4]. CAN is designed to be scalable, fault-tolerant, and self-organizing. Unlike other solutions, the routing table does not grow with the network size, but the number of routing hops increases faster than $\log_2 N$. CAN requires an additional maintenance protocol to periodically remap the identifier space onto nodes.

Pastry [5] is a scalable, distributed object location and application-level routing scheme based on a self-organizing overlay network of nodes connected to the Internet. Fault tolerance is accomplished using timeouts and keepalives, with actual data transmissions doubling as keepalives to minimize traffic. Similar to Pastry, Tapestry [6] employs decentralized randomness to achieve both load distribution and routing locality. The difference between Pastry and Tapestry is the handling of network locality and data object replication. Tapestry's architecture uses a variant of the distributed search technique, with additional mechanisms to provide availability, scalability, and adaptation in the presence of failures and attacks. For fault tolerance, the nodes keep secondary links.

The Chord protocol [7] uses consistent hashing to assign keys to its peers. Although Chord adapts efficiently as nodes join or leave the system, unlike Pastry or Tapestry, it does not achieve good network locality. Kademlia [8], on the other hand, assigns each peer a *NodeID* in the 160-bit key space, and (*key*, *value*) pairs are stored on peers with *ID* close to the key. One advantage of Kademlia's architecture is the use of a novel XOR metric for distance between points in the key space.

Tree-based indexing techniques over DHTs were presented in [9, 10]. In [11] the authors present a solution based on Chord which supports $1D$ range queries. Other systems supporting scalable multi-attribute queries were proposed in [12, 21, 22]. Currently these and other previously proposed DHTs ([26, 27]) are only limited attempts to efficiently approach the problem of providing fault-tolerant P2P networks for storing objects. Most attempts to assure fault-tolerance are based on best-effort approaches, where an arbitrarily large fraction of the peers can reach an arbitrarily large fraction of the data items. We propose a solution which guarantees efficient (in terms of range queries and message routing) storing of large sets of distributed objects, and fault-tolerance in the presence of large-scale highly dynamic distributed infrastructures.

3. System Architecture.

The main components of the architecture are peers (*Agents*), super-peers (*RAgents*), and lookup services (*LUS*). The system manages two types of information: data (the utility objects stored in the name of clients), and metadata (which is used to logically organize the utility objects and related storing information). For fault tolerance, all the information is replicated. The Agents are responsible for storing both data and metadata replicas. They can directly respond to client requests. RAgents are Agents responsible also for managing a location-based set of Agents. Such a hierarchical architecture leads to the formation of clusters of peers. The RAgents are interconnected, forming a complete graph of connections. All control messages are routed between RAgents. Thus, the RAgents connect together several connected clusters of Agents.

Such a hierarchical interconnection topology ensures scalability and, as we will describe later on, an optimal fault-tolerance schema is necessary for modern critical data-intensive applications. It also reflects the real-world phenomenon of large Internet-scale systems. As demonstrated in [15], networks as diverse as the Internet tend to organize themselves so that most peers have few links (Agents) while a small number of peers have a large number of links (RAgents).

The grouping of Agents into clusters is based on their geographic positions, as well as a set of condition metrics. The idea was previously validated in [16], where we proposed an algorithm to assist the EVO P2P videoconferencing system. In this approach peers dynamically detect the best candidates to which to connect. They are chosen based on geographic location and network connectivity properties (we use attributes such as Network domain, AS domain, Country, Continent), as well as load values, number of connected clients, network traffic. When an Agent joins the network, it retrieves the list of RAgents from the lookup services, and it estimates the best RAgent candidates to which to connect. Then the specific RAgent is chosen based on the network loads, geographic position and the number of other Agents connected to the RAgents.

The management of data inside this system relies on the use of metadata. The metadata is distributed to all Agents, using a distributed segment tree for each attribute separately. Such a structure is described in [9]. Because a distributed tree structure cannot consider node clustering, it requires the use of a DHT and of an additional logical level for metadata information retrieval. Thus, peers are connected in a Chord based topology, having identifiers distributed on a logical ring. Maintaining an additional layer introduces a communication overhead that challenges efficient information retrieval. Hence, we had to propose a new routing protocol, analyzed in a subsequent section.

The clusters are organized using an approach similar to Calendar Queues [17]. RAgents mantain the equivalent of buckets, connecting several Agents (having a length of the bucket or bucket-width). The Agents connected to a RAgent, as well as the RAgents existing in the system, can dynamically update their links. If the number of Agents becomes too high compared with the number of RAgents, the dynamic metadata catalogue (segment tree) could grow too big, and so the operations done on it would require longer time to complete. On the other hand, if the number of Agents is too low compared to the number of RAgents, the number of control messages required to access information from the system becomes much higher.

To mediate such problems, the number of RAgents increases and decreases as the number of Agents grows (new Agents join the system) and shrinks (Agents leave the system). Whenever the number of Agents connected to one RAgent becomes too high, a new RAgent is promoted from Agents (based on a voting algorithm), and the cluster is divided in two, by splitting the remaining Agents between the two RAgents. Whenever the number of Agents connected to a RAgent is too low, the cluster is destroyed by joining the Agents with the ones from an adjacent cluster. The metadata and lookup information is updated accordingly. The two operations involve exchange of several control messages to update the current status of clusters, as well as the possible reconnection of several Agents to new RAgents, and RAgents with other RAgents. In order to minimize the number of costly operations, we started from the threshold values for the splitting and joining operations observed in [17]. However, these values are also dynamically adjusted accordingly at runtime. The system dynamics are further detailed in the next sections.

3.1. **Distributed segment tree.** This architecture is introduced and analyzed in [9]. It supports range search operations, while providing scalability and avoiding overload. We shortly describe here some properties of this structure, further referred in the paper.

The basic structure is the segment tree. Thus, an interval $[1, L]$, where L is the length of the interval, is represented by a binary tree of intervals. If a node $[a, b]$ is not a leaf, then the child nodes are $[a, m]$ and $[m + 1, b]$, where $m = (a + b)/2$.

It is guaranteed that the segment tree structure obeys the following rule [8].

Lemma 3.1. *Any interval $I = [a, b]$ can be represented by a collection C of at most $\log_2 L$ disjoint intervals in the tree, and the union of those intervals is I. For example, the two nodes marked in Figure 1 represent one possible decomposition of the interval $I = [3, 6]$.*

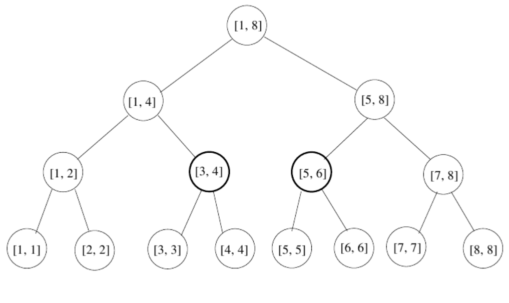

FIGURE 1. Segment tree for the interval [1, 8]

The above lemma has a direct algorithmic interpretation (where *lo* and *hi* are the endpoints of interval I, and *left* and *right* are the endpoints of the current interval) [9]:

```
SplitSegment($lo, hi, left, right, collection$)
1  if $lo \leq left$ and $hi \leq right$
2      then
3          $collection.add([left, right])$
4          return
5  $med = (left + right) \gg 1$
6  if $lo \leq med$
7      then
8          $SplitSegment(lo, hi, left, med, collection)$
9  if $hi > med$
10     then
11         $SplitSegment(lo, hi, med + 1, right, collection)$
```

The segment tree nodes are distributed according to a DHT network. A hash function is applied to the interval $[a, b]$ represented by a tree node. This mechanism allows for efficient node retrieval using look-up services.

The insertion operation is performed as follows: a key $k \in [1, L]$ is inserted in the interval $[k, k]$ and all its ancestor intervals in the tree. Unfortunately, the insertion mechanism can lead to overload the nodes that store large intervals. For example, node $[1, L]$ will have all existing keys in the tree. As this situation is undesirable, in our implementation we follow the idea from [9]:

(1) Each node decides locally whether a key is inserted into the parent interval or not.
(2) Each node x has a C_x counter, increased by each insertion of a key. When C_x exceeds a certain threshold, the key will not be inserted upper in the tree.

The disadvantage is that insertion of a key cannot be performed in parallel in all the ranges containing it. Instead, a key has to be sequentially inserted following the path from leaf to the root. Still, we assume the following relation to be true.

Proposition 3.1. *Assume M the maximum value of counter, x an internal node of the tree, and y, z the two children of x. Then, we assume the following:*

$$C_x = \min\{M, C_y + C_z\} \tag{1}$$

With this, range queries are performed as follows. Let $I = [s, t]$ be the search interval, and suppose we want to retrieve all the keys contained in I. First, the *SplitSegment* algorithm is applied to obtain a collection C of intervals in the tree. Then the requests for retrieving keys in each determined interval are processed in parallel.

When node x receives a request, it first checks whether $C_x \geq M$. If this is not the case, all keys are stored on the node. It will therefore respond directly to the query. On the other hand, if $C_x = M$ and the node is internal, it will send a query to its child nodes. From their answers, the node x will compose the answer to the request.

A key removal is performed similarly to an insertion. The key will be removed from the interval $[k, k]$ and from its ancestors, advancing towards the root. The counters will be decremented after the removal, which means that some nodes x (with $C_x = M$ before removal) may have to request keys from their child nodes.

From the design phase, for each interval there must be a node in the system to store it. Because a segment tree has $2L - 1$ nodes, the system can become overloaded. For efficiency, a node will only exist if it stores at least one key. Obviously, this mechanism makes it impossible to detect the disappearance of a valid node. However, it greatly optimizes memory consumption.

3.2. **The logical ring.**

To maintain a DHT in the system we introduced an additional logical level. The idea is similar to the Chord architecture [7]. In this approach only peers contribute to the logical ring (not the super peers). Each peer is associated with a globally unique $160 - bit$ identifier. This identifier is obtained by applying the SHA-1 algorithm to the peer's IP address and listening port (we assume that all peers have public IPs). Peers are "positioned" on the logical ring in ascending order of these identifiers, clockwise.

Any object or information stored in the system also has an identifier. Objects are stored on the peer having the highest *ID*, smaller than the object *ID* (from now on, we will use the term "predecessor" for that peer); when determining predecessors, we must keep in mind that the identifier space is circular. A representation of a possible situation is given in Figure 2. We see that objects with identifiers 1 and 2 are stored on the peer having *ID* 0, whereas the interval [5, 7] will be stored on the peer having identifier 3.

When inserting an object into the system, the message containing the object is routed according to the protocol described in the next section. The object is then stored on the

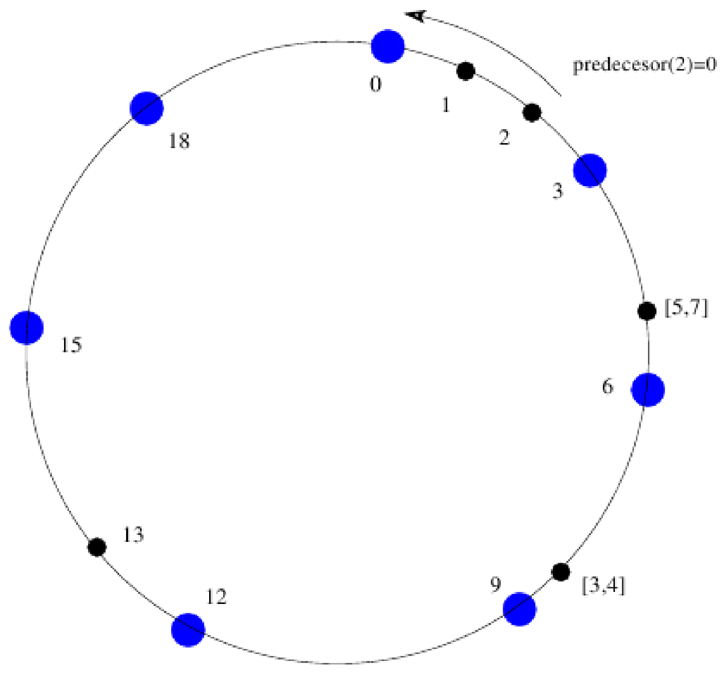

FIGURE 2. Peers arrangement and data distribution

predecessor peer. All look-up calls must contain the object identifier in order to find it. Thus, keys stored in the segment tree have also assigned associated identifiers.

When inserting a peer in the system, some items have to be relocated. The new peer's predecessor has to transfer the corresponding items. Similarly, when a peer leaves the system, the objects it owned must be transferred to its predecessor.

It is therefore natural to consider the problem of overloading a peer at a given time. The efficiency of the proposed method is shown in Theorem 3.1 (proven in [18]), that states the following:

Theorem 3.1. *Assume there are N peers and K keys. Then, there exists a hash function h, so that the K keys are distributed to the N peers with high probability in the following manner:*

(1) Each peer handles at most $(1 + \log_2 N) * K/N$ keys.
(2) If a peer P joins or leaves the system, then at most $O(K/N)$ keys are relocated (to or from peer P).

However, the theorem assumes a hash function that uniformly distributes random keys. This requirement is not guaranteed by SHA-1, but experimentally, it has been found that this condition is respected [19].

Different replicas of the same object will be associated with different hash functions, to provide storage on different nodes. Currently, each object has a master copy and one replica. The object's identifier is calculated by SHA-1; the replica's identifier is obtained by complementing the most significant bit of the first identifier. Formally,

$$Id_2 = Id_1 XOR 2^{m-1} \tag{2}$$

This operation is equivalent to a half-circle trip, so the replica *ID* is diametrically opposed to the master copy. Identifier distribution remains pseudorandom, and the chances that the replica would be stored on a different peer are maximal.

Note that although we connect peers on the logical ring, we also preserve peer clustering. A peer position on the ring depends only on the assigned identifier and it is independent of the cluster membership.

3.3. **The routing protocol.**

The object lookup operations use a routing protocol that efficiently maps onto the proposed architecture. The Chord routing protocol [7] demands $\log_2 N$ links to connect each peer with peers at distances powers of 2 on the ring. To avoid the fulfillment of this difficult requirement, we propose another solution. From construction, the nodes are grouped into clusters, and the super-peer has a connection with each peer in its cluster. We can use this to our advantage; a super-peer can send a message, based on the peer *ID*, to the closest predecessor in the same cluster. Based on this observation, we considered the following algorithm.

Let k be the number of clusters in the system and $C_1, C_2 \in N$ such that

$$C_1 C_2 \geq k \tag{3}$$

Each peer is linked to C_2 immediate predecessors and C_1 immediate successors on the ring. For every peer y in the system, the procedure $y.predecessor(x)$ returns the neighbor of y with the highest *ID* less than or equal to x. Then the message routing algorithm is as follows:

```
ROUTE($m$)
1   ▷ Peer P routes the message $m$
2   if $m.stage = ToSuper$
3      then
4            ▷ P is a peer on the ring
5            $m.stage = ToRing$
6            $sendMessage(m, super\_p(p))$
7      else
8            if $m.stage == OnRing$
9               then
10                    $m.stage = ToSuper$
11              else
12                    $m.stage = OnRing$
13           $sendMessage(m, pred(m.getID()))$
```

Messages are then routed in 3 steps. Each message m is assigned an indicator, $m.stage \in ToSuper, ToRing, OnRing$ that shows the current step. At each stage, the message having as destination the peer x, is routed:

(1) from the current peer y to its super-peer ($ToSuper$),
(2) from the super-peer of y to peer z, so that peers z and y are in the same cluster and peer y is the closest on the ring to peer x ($ToRing$),
(3) from peer z to its neighbour on the ring (that minimizes the distance to peer x) ($OnRing$).

A routing example is presented in Figure 3. The performance of this protocol is given by the following lemma.

Lemma 3.2. *If the number of clusters is k and the nodes are uniformly distributed in clusters, then, with a probability of at least $1 - 1/k$, the message will touch no more than $3C_1 \ln k + 4$ peers from the initial source to the destination (C_1 is the number of considered successors).*

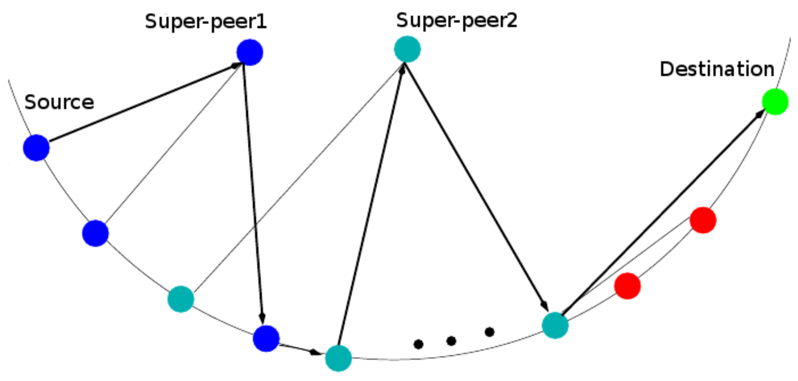

FIGURE 3. A message routing example

Proof: For a peer p, we denote by $super\_p(p)$ the super-peer of p's cluster. We assume that p routes a message m with identifier ID, and its predecessor peer on the ring is d. We denote by X the random variable that represents the number of different super-peers $super\_p(d)$ that route the message m. Then, the number of nodes (including super-peers) that the message reaches until the destination, is no more than $3X + 1$. Then, the first $3(X-1)$ nodes are parts of clusters that do not have members among the C_2 predecessors of d, otherwise the message would have reached the destination in one step. Also, these nodes are parts of pairwise distinct clusters.

$$\begin{aligned}\mathbb{P}[X-1 \geq i] &\leq \left(\frac{k-1}{i}\right) i! \left(\frac{1}{k}\right)^i \left(1-\frac{i}{k}\right)^{C_2} \\ &\leq \left(1-\frac{1}{k}\right)^i \left(1-\frac{i}{k}\right)^{C_2} \\ &\leq e^{-iC_2/k}\end{aligned}$$

Choosing $i = C_1 \ln k$ from (3) results that:

$$\mathbb{P}[X \geq C_1 \ln k + 1] \leq \frac{1}{k} \tag{4}$$

So, with probability of at most $1/k$, the message reaches more than $3C_1 \ln k + 4$ nodes.

Choosing $C_1 = C_2 = k^{1/2}$, we find that a message is routed in $O(k^{1/2} \ln k)$ steps. On the other hand, a greater number of links decreases the routing time, while a smaller number of links increases scalability, but also requires more routing steps. The parameters C_1 and C_2 should be chosen depending on the application.

3.4. **Integration of system elements.**

The system elements described in the previous sections are integrated as follows.

A super-peer is responsible with keeping the necessary routing information. It maintains the list of peers in the corresponding cluster, and for each such peer, the associated identifier. Identifiers are stored in a structure that allows predecessor and successor queries in logarithmic time, using a balanced tree. In fact, the search time can be reduced to $\log_2 160 \approx 7.32$ steps using van Emde Boas trees, but this greatly increases the implementation complexity of our proposed system and is out of the scope of this paper.

Peers store data objects and segment tree nodes (containing metadata). It is possible that several nodes are stored on the same peer. Each peer maintains a structure of its neighbors, which allows predecessor or successor calls in logarithmic time.

Each peer/super-peer runs a routing procedure for messages that are not destined to it, or whose destination is not known (i.e., not among its neighbors). Messages whose destination was already established are routed directly to the destination via TCP/IP.

Since super-peers do not maintain metadata, a client request can be processed directly by the peer that receives the request.

There can be one distributed segment tree for each attribute. The values of an attribute are mapped into a finite range of consecutive integers starting from 1. The mapping scheme is left to the responsibility of higher-level applications. It is necessary, however, that the size and interval mapping algorithm results are available in the code running on each peer. By default, the attribute types are predefined (but of course the system can be extended with new types).

Segment tree nodes are created in a "lazy" manner; nodes are created only for intervals that contain at least one key. Initially, there will be no tree node in the system.

We mentioned previously that whenever a cluster becomes too large it splits in two other clusters. However, for this to work, we must first meet most of the requirements of Lemma 3.2. In particular, the division has to be done uniformly random (each node will be part of a cluster with probability of 1/2). Thus, the resulting clusters will not have exactly half of the initial cluster size M, but the average size will be $M/2$; and according to Chernoff bounds [20], the actual size will be closer to the average with exponentially large probability. Let us note however that in this way we do not obtain a uniform distribution of nodes in clusters.

3.5. **System dynamics.** The system architecture requires the design of appropriate protocols for peers joining or leaving the system. When a new peer P *enters* the system, the following steps are performed:

(1) Peer P contacts lookup services and receives a list with all the super-peers in the system. Peer P then selects a super-peer S and sends a message containing its address and assigned *ID*. This message will be routed to V, the logical immediate predecessor of P.
(2) V sends P the lists of C_1 successors and C_2 predecessors (adding itself) and the list of objects that will now be stored on P. V adds P to the neighbors list.
(3) P contacts its C_2 predecessors and C_1 successors to announce its entry. Each neighbor will confirm the announcement and will add the node to its own list.
(4) P informs S that it has completed its initialization and it is part of the cluster.

As shown, only P's predecessors have to establish new links, not its successors (otherwise some connections would be duplicated).

When a peer P is planning to *leave* the system, the following steps are taken:

(1) Peer P sends a closing message on every open connection. Starting from this moment, almost all messages that reach P will be stored unopened (without being analyzed and routed).
(2) Peer P receives confirmation messages. Each peer V predecessor of P, will try choosing a new successor, in order to maintain the required number of neighbors.
(3) P sends its immediate predecessor a message, containing its stored objects, metadata and messages.
(4) P shuts down.

Based on the protocols above, the number of messages generated by each operation can be determined as:

Lemma 3.3. *The required number of messages for each operation is as follows:*

(1) A peer joining the system requires no more than $4C_2 + 3$ *messages.*

(2) The announced departure of a peer P of the system requires no more than $2q+1+4C_2$ messages, where q is the number of active connections of P.
(3) Splitting a cluster by choosing a peer P and a set of nodes S requires no more than $2q+1+4C_2+3|S|$ messages (where q is the number of active connections of P).

Proof: When a peer P joins the system, it has at most $2C_2$ neighbors. It takes two messages to find out successors and predecessors. In at most $4C_2$ messages it establishes connections, and another message to announce its super-peer in the end of the initialization. Thus, we obtain a total of $4C_2+3$ messages.

When a peer P with q active connections announces its leave, it closes all connections ($2q$ messages). In another message it moves objects on its predecessor. Each predecessor of P finds a new neighbor, and results in a total of no more than $2q+1+4C_2$ messages.

Finally, when splitting a cluster, let us assume peer P becomes the new super-peer. Leaving the ring requires at most $2q+1+4C_2$ messages, where q is the number of active links of P. Every peer in S, the set of nodes that will form the new cluster, must close the connection to the old super-peer and open a new connection to P. This operation requires $3*|S|$ messages, resulting in a total of $2q+4C_2+3|S|+1$ messages.

3.6. **Data objects storage and retrieval protocol.** Storing an object o in the system requires both replicas and metadata generation. Object storage is performed according to the following steps:

(1) Create replica(s) of o and store them (and o).
(2) For each attribute A of o, having the value v, generate metadata $((A,[v,v]),o)$. Store the metadata in a leaf node of the corresponding segment tree. Replicate the metadata.
(3) Propagate metadata up in the tree (see Section 3.1).

Retrieval of all objects having the value of the attribute A inside the interval $I=[u,v]$, is performed according to the following algorithm:

(1) Call *SplitSegment* procedure to generate a collection C of disjoint intervals in the tree, having the union I.
(2) For each interval $I \in C$ send a request to the appropriate tree node.
(3) The peer storing the tree node in question may in turn generate requests to retrieve objects having the attribute A inside the interval I.
(4) Assemble responses to form final response.
(5) If a tree node or object is not found, search for a replica. The lost object will be created from the replica and re-stored on the appropriate peer (to preserve a constant number of replicas).

The number of messages generated by each operation is more difficult to estimate. Nevertheless, we have the following lemma:

Lemma 3.4. *Suppose that each element has p replicas and each object has d attributes. And let L be the maximum length of an attribute's range of values. Then, the number of messages for the storage operation is at most $p(1+d(\log_2 L+1))$.*

Proof: The statement is obvious, as the height of a segment tree is at most $\log_2 L+1$, and both objects and metadata have p replicas.

3.7. **Fault tolerance.** In the described architecture, Agents and RAgents periodically exchange heartbeat messages between them, to detect when a peer fails. The period between these messages is chosen such as to be high enough not to introduce much communication overhead in the system. Failures are then detected using an accrual failure detector which we previously proposed in [24], that also uses a gossip strategy to minimize detection time and remove wrong suspicions.

For failure detection, the arrival times of heartbeat messages are sampled and used to estimate the time when the next heartbeat is expected to arrive. The estimation function relies on a modified version of the exponential moving average (EMA) method named KAMA (Kaufman's adaptive moving average). This method ensures a dynamic smooth factor based on the most recent timestamps. The predicted value is used to estimate the level of suspicion in the process being failed. The contribution of a heartbeat H increases from 0, meaning that H is not expected as failed, to 1, meaning that H is considered to be lost. Unlike other implementations of accrual detectors, the suspicious level in this case is not computed based only on local heartbeat contributions, but also considering the contributions received from other failure detectors.

In fact, it is difficult to determine the nature of an error that affects a certain process. In an unstable network, with frequent losses of messages or high process failure rate, any detection algorithm is almost irrelevant; e.g., it cannot distinguish between failures of the peers and the failing of network links. This is why the failure detector uses a gossiping algorithm whose role is to increase the confidence of an agent in the failure of a peer. The gossiping schema aims to eliminate false negatives (wrong suspicions) and false positives (peers are considered to be alive, even though they have failed). For this peers monitor each other and periodically exchange information about the status of known peers (messages are sent between Agents and RAgents, and between each Agent and its predecessor and successor Agents).

The *suspicion level* represents the degree of confidence in the failure of a certain process, and it takes values in the $[0, 1]$ interval. Each failure detector maintains a local suspicion level value $sl_{qp}(t)$ for every monitored process computed according to:

$$sl_{qp}(t) = \frac{e^{\alpha(t-1)}}{e^{\alpha(t+1)}}, \text{ where } t = \frac{t_{now}}{t_{pred}} \tag{5}$$

The contribution function limits the suspicion values to the [0, 1] interval, maintaining a relatively quick evolution in the [0, 0.8] interval and a very slow one in the [0.8, 1] interval. For a certain threshold the probability of failure is very high. When a peer fails, it does not receive alive requests. This leads to an increase in the suspicion level associated to that particular peer, up to a value closer to 1. Therefore, every heartbeat message that is not received leads to a higher suspicion level, and therefore to a high probability of failure. On each alive request the monitoring Agent updates the suspicion level of the queried process based on the current time and the last prediction. If no answer is received the last updates the predicted value is not changed. As the message is delayed the difference between current time and the last prediction increases and becomes and the suspicion value gets closer to 1.

The gossiping protocol is next responsible with the exchange of information between peers. The gossip protocol is based on a probabilistic model in which peers randomly choose partners (from known peers) with whom they exchange information. To understand how gossiping works we consider a simple example. Let us assume a peer monitors another peer, and, for various reasons, it loses touch with it and suspects it of failing. Then it receives a gossip message from another peer revealing that the suspected Agent is still functional. The suspicion level is updated according to the received rumor and the monitored Agent is no longer suspected. Until the transient failure disappears, the above process is likely to be repeated several times, but at some point the suspicion level will redress and indicate a functional peer.

We now present the various operations performed when nodes/services fail. First, a LUS can fail. In our implementation we use a network of JINI Lookup Discovery Services (LUS) that provides dynamic registration and discovery for all agents. For example, the RAgents are able to discover each other in the distributed environment and be discovered by the interested clients. The LUSs synchronize between themselves using multicast messages. The registration uses a lease mechanism. If an Agent fails to renew its lease, it is removed from the LUSs and a notification is sent to all the peers that subscribed for such events. Remote event notification is used in this way to get a real overview of this dynamic system.

Next, for each RAgent we assume a secondary replicated RAgent (another Agent in the same cluster that can readily takes its place). Since the list of data maintained by the RAgent is registered in the LUS, when the RAgent fails the secondary RAgent can quickly take its place and recover all corresponding entries from the lookup service. In this case, the new RAgent connects to the Agents in the local cluster, and it becomes the new RAgent in charge of the cluster. From all remaining Agents one is selected (based on a voting procedure) and becomes the new RAgent replica. The voting procedure is consistent, such that if two agents observe the disappearing of the same RAgent, they both initiate the voting procedure, but in the end both voting procedures lead to the same Agent being selected to be the new RAgent replica.

Next, an Agent P can *fail.* In this case the system peforms the following operation. Each peer V, one of the C_2 predecessors of P, will try choosing a new immediate successor, in order to maintain the required number of neighbors. V will obtain a list of immediate successors of his successor, then establish a connection with the first node in the circular list (if any) that it is not already for its successor. With this, we have the following lemma.

Lemma 3.5. *Failure of a peer requires at most $4C_2$ messages in order to restore the network structure.*

Proof: If a peer P fails, then each predecessor of P will try to find a new neighbor. There are no more than C_2 predecessors of P. Determining of a new neighbor requires no more than 4 messages: 2 for finding a neighbor and 2 to establish the connection. This generates no more than $4C_2$ messages.

Also, if a tree node or object is not found, the data objects storage and retrieval protocol (see above) assume the search for a replica. The lost object will then be created from the replica and re-stored on another appropriate peer (to preserve a constant number of replicas in the system).

Different replicas of the same object will be associated with different hash functions, to provide storage on different nodes. Currently, each object has a master copy and one replica. The object's identifier is calculated by SHA-1; the replica's identifier is obtained by complementing the most significant bit of the first identifier. The operation required to compute the replica's identifier is equivalent to a half-circle trip (see 2, so the replica *ID* is diametrically opposed to the master copy. Identifier distribution remains pseudorandom, and the chances that the replica would be stored on a different peer are maximal.

For the routing protocol, fault tolerance is ensured by the object and metadata replication, according to the two hash functions and the number of links between peers. The proposed message routing protocol enables messages to reach any peer, as long as every peer can connect to one of its immediate successors. For example, if each peer has p successors, and a peer fails with a probability of 1/2, there is a $(\frac{1}{2})^p$ chance for that peer to remain without successors. With $p = 10$, the probability of this to happen is substantially low.

4. **Experimental Results.** The system was implemented using Java technology. Its design follows a modular approach, where each module corresponds to the different layers of the system.

The network module was implemented using Java NIO. The server runs in a separate thread and uses select operations. Optimizations at this level include implementation of resizable NIO buffers and streaming the read and write operations.

We first evaluated the system using a cluster of Intel Xeon E5405 quad-core stations (each station is equipped with 8 cores, at 2 Ghz, has 16 GB RAM memory, and 6Mb cache memory), connected through Gigabit Ethernet (1 Gbps between chases, each chases grouping together 8 stations connected at 100 Gbps). These experiments evaluated the influence of various parameters on the overall performance of the system. For example, we evaluated how the number of objects replicated on each Agent can influence the performance (e.g., if the number is very high, the fault recovery process could take a longer time). Another result of interest is the number of messages exchanged for the internal management of the system. And another one is the time needed for clusters to re-balance in case of Agents entering and exiting the system. For these experiments we used 9 Agents (each one running on another station), and 13 objects. Each object has 2 replicas (for a total of 26 objects). Figure 4 (top) presents the distribution of objects on the Agents. At some point we simulated the crash of an Agents in the system. Figure 4 (bottom) presents the evolution of the objects after such a crash. As presented, the system automatically re-balances and replicates the objects, such that is able to continuously preserve the two replicas of each object (the master copy and an additional replica).

After evaluating the capability of the system to cope with the crash of an Agent, we evaluated its performances. We executed several experiments considering the behavior of the system in the presence of different events (such as changes in the topology or faults in various network links). We logged the processing loads and throughput on the networks connecting the Agents (as a measure of the balance of activities between Agents). For these experiments we also varied the number of Agents (such that to also experiment with clusters dividing and joining together). Figure 5 shows the load of one of the testing machine. If the system would consume too much processing power, the station would become unavailable for the execution of other applications. Still, as the results show, the average load was between $0.14 - 0.15$, which means the proposed algorithms are not computational intensive. Also, the topology changes (joining and dividing clusters) affect the system. However, as presented, these changes last for little time and the system is able to quickly stabilize. In this case, the maximum load reaches 0.5, which is not very high and does not affect possibly other concurrent applications running on the same station.

The CPU usage (see Figure 5) was approximately 8%. The spikes appeared because of the event of the exit of an Agent (due to a failure), followed by immediate cluster division. As seen, the joining operation does not affect to a vast extend the CPU usage.

These experiments demonstrate the performance of the presented system. As presented, the system is capable of quickly adapting to failures or Agents dynamically entering or exiting.

4.1. **Object distribution.** For the next experiments we used 100 objects, and each has 2 attributes (to evaluate the management of different attributes). The attributes have values in the interval $[1, 100]$. For each $Object(i)$, the attribute tuple is $(i + 1, 100 - i)$, where $0 \leq i \leq 99$. For these experiments we evaluated the management of objects stored on each peer, the numbers of messages per operation, and the evolution of the system when varying different parameters.

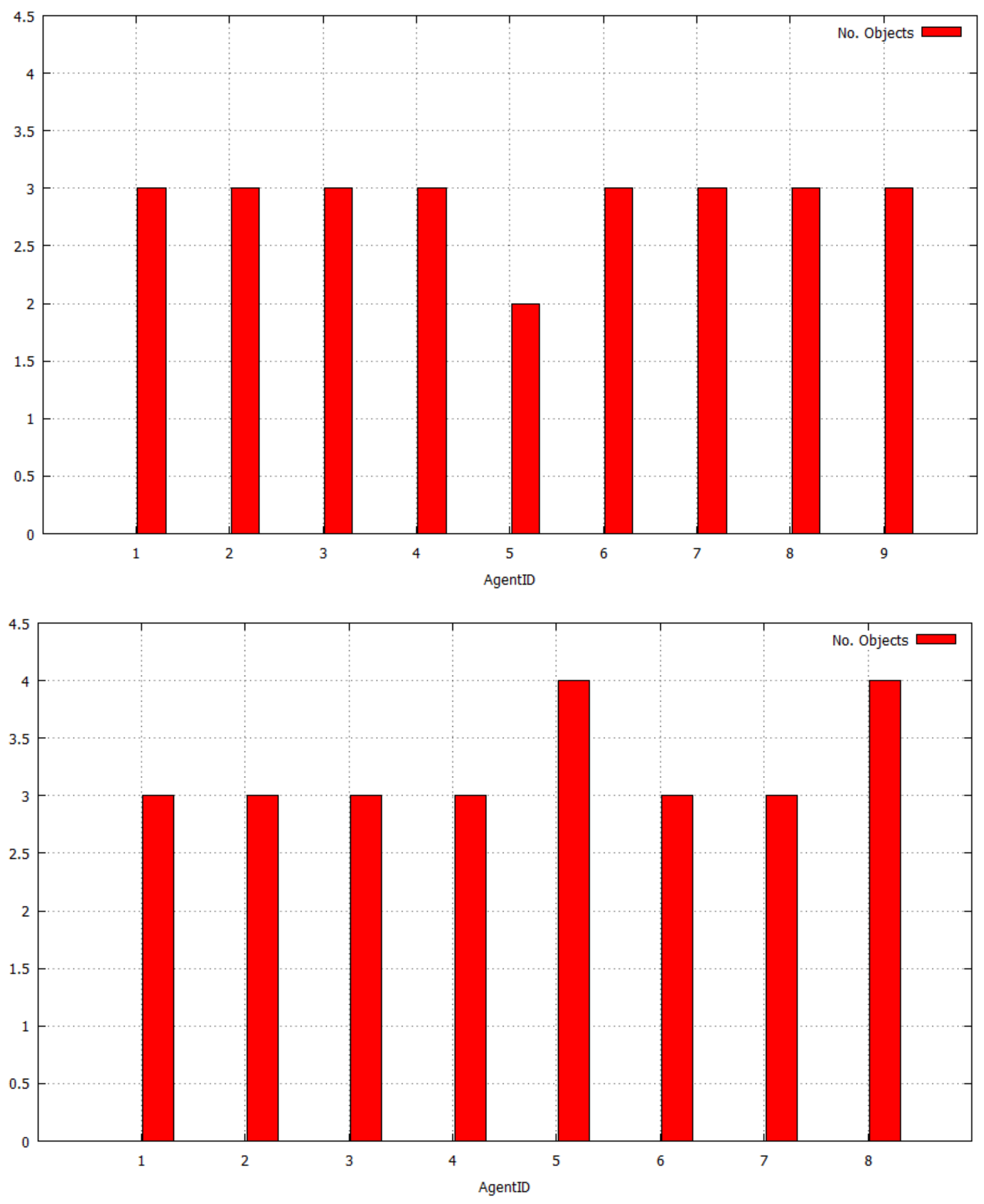

FIGURE 4. The distribution of replicated objects before (top) and after (bottom) the crash

In the first experiment we evaluated peer load balancing for a fixed number of objects and a varying number of peers. For this experiment we used all 100 data objects. Each object was replicated exactly once (100 more objects – the replicas), and for each attribute we constructed a metadata segment tree (approximately 200 nodes in each segment tree). In total, we had 600 objects (100 data objects, 100 data object replicas, $2 \cdot 199$ segment tree nodes).

First, we added 1 peer, then all the objects (100 data objects – 600 system objects). We kept adding peers, until we reached 100 (using consecutive IPs, and the same listening port). Figure 6 presents the distribution of objects when using 10 (top), 20 (middle), and respectively 40 (bottom) peers. As seen, because of the two hash functions (the two attributes of each object), in every case several nodes become more overloaded than the others.

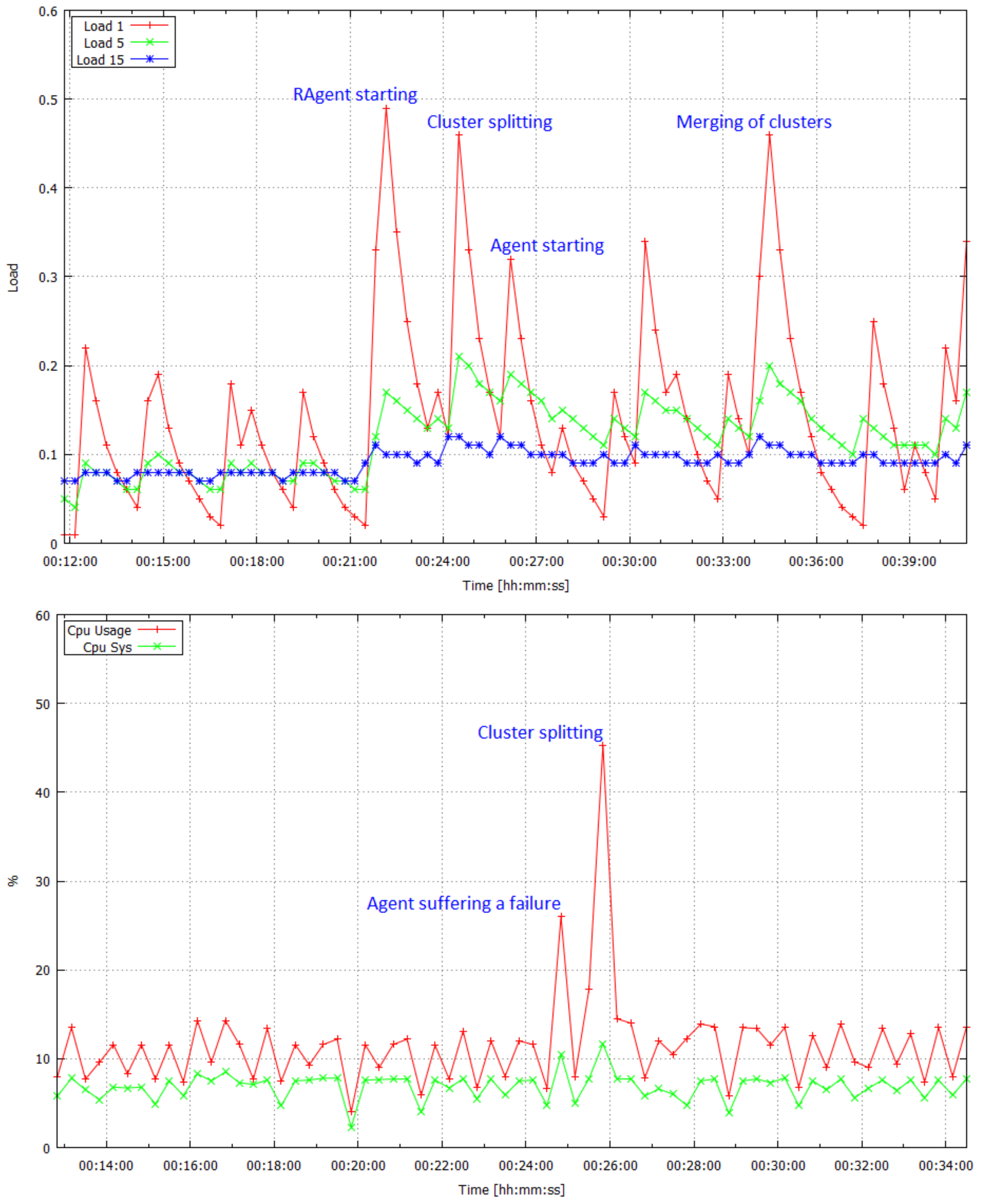

FIGURE 5. The load and CPU usage on a test machine

4.2. **Bandwidth consumption.** We next evaluated the network throughput involved by the system. The evaluated consists of the following scenario:

(1) First add 10 peers into the system.
(2) Store 3 (2 at the same time, 1 later) data objects. These data objects lead to the insertion of other objects: replicas and nodes of the segment tree. The first object leads to the insertion of $14(2\log_2 100)$ objects: segment tree nodes and one replica. Each of the other two inserts at least 3 extra objects (one replica +2 segment tree nodes).
(3) Range search for all the objects having the first attribute inside the interval $[1, 100]$.
(4) Store one object.

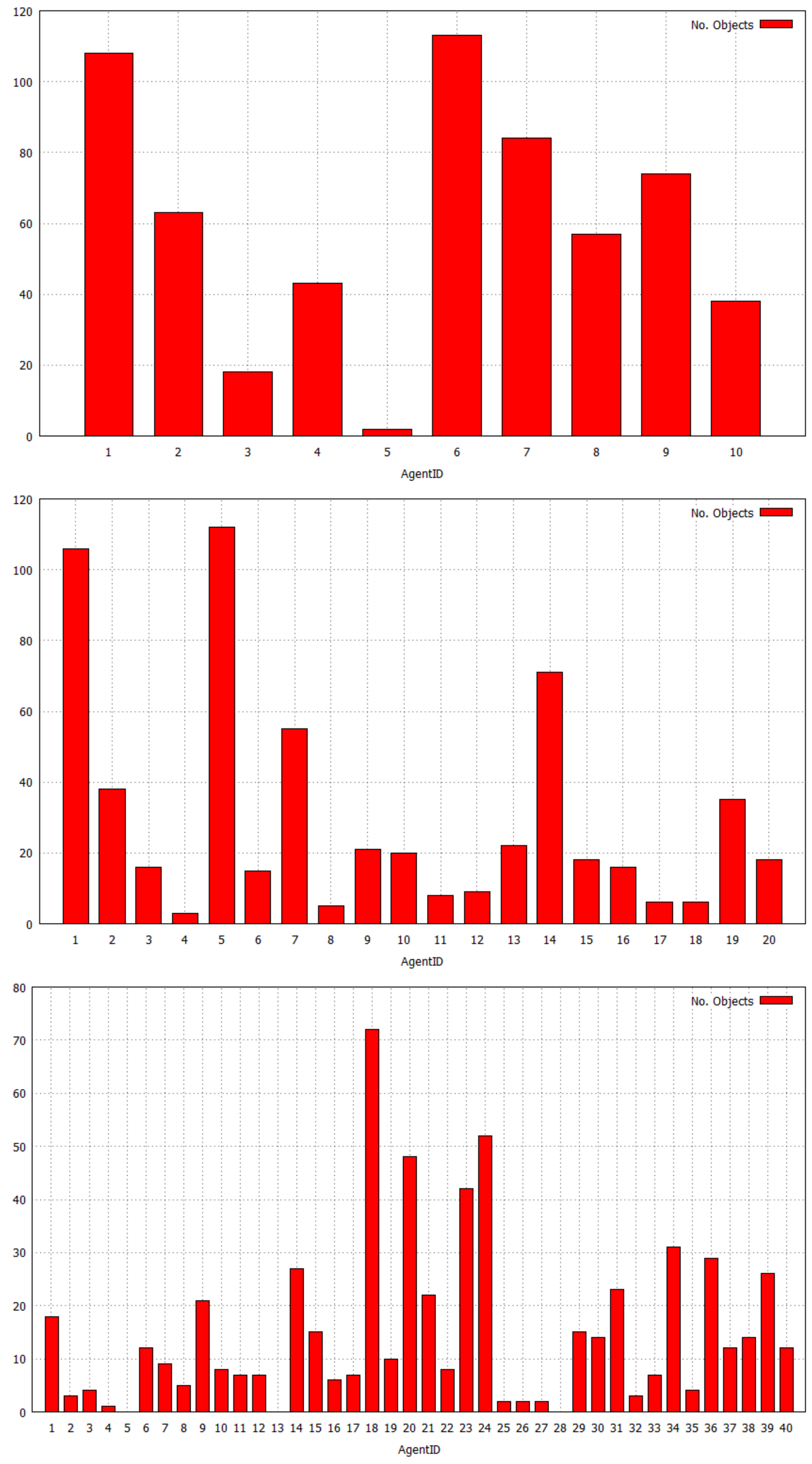

FIGURE 6. Objects distribution for 10, 20 and 40 peers

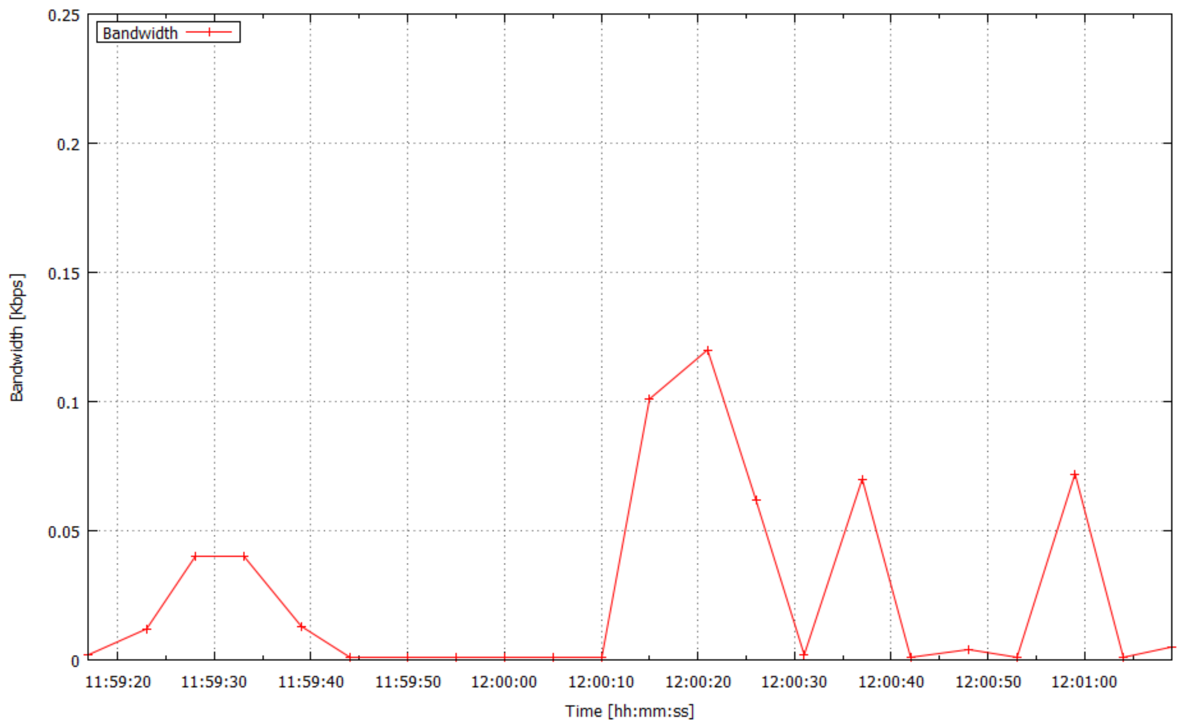

Figure 7. Aggregated bandwidth consumption

(5) Range search for all objects with attributes in $[1, 1]$.

For the aggregated bandwidth consumption of all the peers in the system, we obtained the graph in Figure 7. The first peak, $0.4 \cdot 10^{-4}$ Mbps (seconds $10 - 15$) is caused by peers joining the system. At second 25 all 10 peers enter the system. At second 50 we start inserting 2 objects. At second 70 we insert another object. At second 80 we initiate a range search query for the first attribute, inside $[1, 100]$. In the end, as presented, the bandwidth consumption is low (less than $0.1 \cdot 10^{-4}$ Mbps).

We can see that object insertion is the operation that consumes the most bandwidth (peaks at $0.7 \cdot 10^{-4}$ Mbps) and the range search operation is quite bandwidth efficient. (These results are also confirmed by the number of messages test.)

As shown in [30], balancing scalability, storage guarantees, and resilience to highly dynamic membership *and bandwidth consumption* is a challenging requirement. In fact, our results show superior capabilities (lower bandwidth consumption compared with other papers. We obtained an average bandwidth consumption below 0.05 Kbps, compared with results shown by some other well-known solutions with shown bandwidth consumption results in the range of over 100 Kbps in [31, 32].

4.3. **Number of generated messages.** In order to measure the number of transmitted messages, we considered a system consisting of 18 peers. A segment tree node can store up to 10 keys; $C_1 = C_2 = 3$ (number of neighbors is 6). In this case we executed the following experiments:

(1) We stored 100 data objects. As previously described, these 100 data objects create 598 objects. The number of messages in this case was 3128, which corresponds to the 3400 estimated by Lemma 3.4.
(2) We searched for a stored object – 3 messages.
(3) We initiated a range query for the first attribute, inside $[1, 100]$. We obtained all the 100 objects. This resulted in 215 messages (2.15 messages per object).

(4) We initiated a range search query for the second attribute, inside the interval $[50, 50]$. We obtained a single object. The total number of messages was 3.
(5) We simulated a peer failure. In order to restore the system, 12 messages were generated. This value is within the limits predicted in Lemma 3.5.

We observe that all the experiments leaded to results are similar to the ones predicted by the theoretical results presented in the previous sections. And, as noticed in [23] (analyzing Grids) and [25] (within an analyze of Cloud-related energy problems), there is a direct link between the number of messages exchanged in the system, the local load induced on the running stations, and the energy consumption required when running the system in a production environment. In our case, as presented, the report is kept in the limits, as the proposed algorithms require few execution steps, converge to a solution (such as the choice of the peer on which an object is stored) rapidly, and require few messages exchanged for searching an objecting, or for mitigating the different occurring failures. For all this we provided theoretical boundaries (refining the ones previously obtained by similar systems [22]), which concur with all results observed in our experiments.

5. Conclusions and Future Work.

In this paper we presented a scalable peer-to-peer solution that can be employed for storing objects and for performing range queries on the objects' attributes. The architecture consists of a Chord ring, enhanced with a cluster overlay. The cluster overlay separates the nodes into clusters based on proximity metrics. Request routing is performed by considering both the cluster overlay and the Chord ring. Range queries are supported by using a distributed segment tree. The information corresponding to each node of the segment tree (metadata) is stored as an object in our architecture. The system includes fault tolerance and orchestration mechanisms necessary to provide reliability in an autonomic manner. Experimental results have shown that an implementation of the proposed system leads to results similar to the one predicted in our theoretical analysis. As future work, we plan to integrate the system into large scale distributed systems, and perform further experiments to tune the implementation accordingly.

Acknowledgment. The research presented in this paper is supported by project "ERRI C – Empowering Romanian Research on Intelligent Information Technologies/FP7-REGP OT-2010-1", ID: 264207. The work has been co-funded by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/89/1.5/S/6255 7.

REFERENCES

[1] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa and S. Jiang, Current practice and a direction forward in checkpoint/restart implementations for fault tolerance, *Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium*, vol.19, 2005.

[2] K. Lua, J. Crowcroft, M. Pias, J. Sharma and S. Lim, A survey and comparison of peer-to-peer overlay network schemes, *IEEE Communications Surveys & Tutorials*, vol.7, no.2, pp.72-93, 2005.

[3] R. Ranjan, L. Chan, A. Harwood, S. Karunasekera and R. Buyya, Decentralised resource discovery service for large scale federated grids, *Proc. of the 3rd IEEE International Conference on E-Science and Grid Computing*, pp.379-387, 2007.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, A scalable content-addressable network, *Proc. of the 2001 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'01)*, San Diego, CA, USA, pp.161-172, 2001.

[5] A. Rowstron and P. Druschel, Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, *Proc. of IFIP/ACM Intl. Conf. on Distributed Systems Platforms (Middleware'01)*, London, UK, pp.329-350, 2001.

[6] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph and J. D. Kubiatowicz, Tapestry: A resilient global-scale overlay for service deployment, *IEEE Journal on Selected Areas in Communications*, vol.22, no.1, pp.41-53, 2004.

[7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, Chord: A scalable peer-to-peer lookup protocol for Internet applications, *IEEE/ACM Transactions on Networking*, vol.11, no.1, pp.17-32, 2003.

[8] P. Maymounkov and D. Mazieres, Kademlia: A peer-to-peer information system based on the XOR metric, *The 1st International Workshop on Peer-to-Peer Systems, Lecture Notes in Computer Science*, vol.2429, pp.53-65, 2002.

[9] C. Zheng, G. Shen, S. Li and S. Shenker, Distributed segment tree: Support range query and cover query over DHT, *Proc. of the Intl. W.-Shop on P2P Syst.*, Santa Barbara, CA, USA, 2006.

[10] N. Lopes and C. Baquero, Implementing range queries with a decentralized balanced tree over distributed hash tables, *Lect. Notes in Comp. Sci.*, vol.4658, pp.197-206, 2007.

[11] M. Abdallah and E. Buyukkaya, Efficient routing in non-uniform DHTs for range query support, *Proc. of the Intl. Conf. on Par. and Dist. Comp. and Syst.*, Dallas, TX, USA, pp.239-246, 2006.

[12] M. Hauswirth and R. Schmidt, An overlay network for resource discovery in grids, *Proc. of the Intl. W.-Shop on Database and Expert Systems App.*, Copenhagen, Denmark, pp.343-348, 2005.

[13] M. Ripeanu, I. Foster and A. Iamnitchi, Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design, *IEEE Internet Computing Journal*, vol.6, no.1, 2002.

[14] C. Dobre, F. Pop and V. Cristea, A fault-tolerant approach to storing objects in distributed systems, *Proc. of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*, Fukuoka, Japan, pp.1-8, 2010.

[15] A. Barabasi, R. Albert, H. Jeong and G. Bianconi, Power-law distribution of the world wide web, *Science*, vol.287, 2000.

[16] C. Dobre, R. Voicu, A. Muraru and I. C. Legrand, A distributed agent based system to control and coordinate large scale data transfers, *Proc. of the 16th International Conference on Control Systems and Computer Science*, Bucharest, Romania, 2007.

[17] R. Brown, Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem, *Communications of the ACM*, vol.31, no.10, pp.1220-1227, 1988.

[18] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, New York, USA, pp.654-663, 1997.

[19] B. Mulvey, *Hash Functions. Evaluation of SHA-1 for Hash Tables*, http://home.comcast.net/bretm/hash/9.html, 2012.

[20] T. Hagerup and C. Rub, A guided tour of Chernoff bounds, *Information Processing Letters*, vol.33, no.6, pp.305-308, 1990.

[21] H. Liu, P. Luo and Z. Zeng, A structured hierarchical P2P model based on a rigorous binary tree code algorithm, *Future Generation Computer Systems*, vol.23, no.2, pp.201-208, 2007.

[22] S. Sotiriadis, N. Bessis and N. Antonopoulos, Using self-led critical friend topology based on P2P chord algorithm for node localization within cloud communities, *Proc. of Intern. Conf. on Complex, Intelligent, and Software Intensive Systems*, Seoul, Korea, pp.490-495, 2011.

[23] Y. Huang, N. Bessis, S. Sotiriadis, A. Brocco, M. Courant, P. Kuonen and B. Hirsbrunner, Towards an integrated vision across inter-cooperative grid virtual organizations, in *Future Generation Information Technology*, Y. Lee, T. Kim, W. Fang and D. Slezak (eds.), Springer, 2009.

[24] A. Lavinia, C. Dobre, F. Pop and V. Cristea, A failure detection system for large scale distributed systems, *IJDST*, vol.2, no.3, pp.64-87, 2011.

[25] Z. Xue, X. Dong, L. Hu and J. Li, A performance and energy optimization mechanism for cooperation-oriented multiple server clusters, *Future Generation Computer Systems, Special Section: Energy Efficiency in Large-Scale Distributed Systems*, vol.18, no.5, pp.801-810, 2012.

[26] W. K. Lai, M.-L. Weng, S.-H. Lo and C.-S. Shieh, KAdHoc: A DHT substrate for MANET based on the XOR metric, *ICIC Express Letters*, vol.3, no.4(A), pp.909-914, 2009.

[27] C. Lee and T. Jeong, Multi-level mechanism for distributed P2P mobile streaming services, *ICIC Express Letters*, vol.6, no.7, pp.1803-1808, 2012.

[28] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris and I. Stoica, Looking up data in P2P systems, *Communications of the ACM*, vol.46, no.2, pp.43-48, 2003.

[29] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris and I. Stoica, Wide-area cooperative storage with CFS, *ACM SIGOPS Operating Systems Review*, vol.35, no.5, pp.202-215, 2001.

[30] C. Blake and R. Rodrigues, High availability, scalable storage, dynamic peer networks: Pick two, *The 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, USA, vol.18, 2003.

[31] R. Rodrigues and B. Liskov, High availability in DHTs: Erasure coding vs. replication, *Peer-to-Peer Systems IV*, pp.226-239, 2005.

[32] B. B. Yang and H. Garcia-Molina, Designing a super-peer network, *Proc. of the 19th International Conference on Data Engineering*, pp.49-60, 2003.