Scalable Chord-based, Cluster-enhanced Peer-to-Peer Architecture supporting Range Queries

Codrut Grosu, Eliana-Dina Tirsa, Ciprian Dobre, Valentin Cristea, Mugurel Ionut Andreica
University POLITEHNICA of Bucharest, Romania
E-mails: codrut.grosu@cti.pub.ro,
{eliana.tirsa, ciprian.dobre, valentin.cristea, mugurel.andreica}@cs.pub.ro

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 scalable, 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 extends an existing architecture with an original solution designed to achieve optimal implementation results for range queries, as well as provide a fault-tolerant substrate. It also optimizes message routing in hop-count and throughput, whilst providing an adequate consistency among replicas.

1. Introduction

Peer-to-peer (P2P) networks are amongst the fastest growing technologies in computing. They are largely used for data-sharing, and hence must support a powerful mechanism for distributed search. In the last years, intense research has been done to overcome scaling problems with unstructured P2P networks, such as Gnutella [12], where data placement and overlay network construction are essentially random.

This paper presents an extension to an existing P2P system introduced in [13] that allows efficient and fault tolerant storage and retrieval of shared objects by attribute values. The main contribution of the present work is to use a distributed segment tree to maintain metadata and add support for range queries. In order to preserve the advantages of the previous architecture, we introduce and analyze a new routing algorithm.

The architecture presented in [13] has two types of nodes: *peers* that store data and *super-peers* that control the peer clusters, maintain the metadata and answer to client requests. The super-peers are connected in a complete graph. Clusters maintain a manageable number of peers, being split or merged accordingly. Both clusters and cluster dynamics are geographic location aware. Super-peers register to and can be discovered by lookup services, accessible by all peers. Fault tolerance is ensured by replicating the data on several peers and by maintaining a backup for every super-peer.

Letting aside the beneficial characteristics of the original system (that we tried to preserve), we also identified a series of problems that our proposed solution will overcome. Firstly, the network is experiencing scalability problems because a super-peer processes every request regarding data in a given cluster. Metadata is stored on super-peers and their backups, however, only the master replica is exposed to requests. Also, broadcasting between super-peers undermines cluster locality. Furthermore, network topology is not fully exploited. The links between peers are not used, although message routing might improve and super-peers will be less burdened with traffic.

The rest of the paper is structured as follows: in section 2 we discuss related work, in section 3 we describe our proposed architecture, in section 4 we present experimental results and in section 5 we conclude.

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 [1]. 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 [3] 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 [4] 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.

Chord protocol [5] 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 [6] assigns each peer a *NodeID* in the *160*-bit key space, and *(key,value)* pairs are stored on peers with *IDs* 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 [7, 8]. In [9], the authors present a solution based on Chord which supports *1D* range queries. Other systems supporting scalable multi-attribute queries were presented in [10, 11].

Currently there are limited attempts to efficiently approach the problem of fault-tolerance with previously proposed DHTs. 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.

3. Proposed solution

Our proposed solution will preserve part of the properties of the system described in [13]: node classification into peers (that store data), super-peers (that control clusters) and lookup services, and location aware peer clustering and cluster dynamics. In order to address the issues mentioned earlier, we implement the following changes:

- super-peers will not store metadata
- super-peers will not respond to client requests
- peers will maintain both data and metadata replicas
- peers will process client requests

Metadata will be distributed throughout the system using a distributed segment tree for each attribute separately. Such a structure is described in [7]. 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 will be 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 propose and analyze a new routing protocol. In the next 7 subsections, we will present all the newly introduced features.

3.1. Distributed segment tree

This architecture is introduced and analyzed in [7]. It supports range search operations, while providing scalability and avoiding overload. We will shortly describe here some properties of this structure, that we will further refer to 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 guarateed that the segment tree structure obeys the followig rule [7]:

Lemma 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 are one possible decomposition of the interval *I = [3, 6]*.

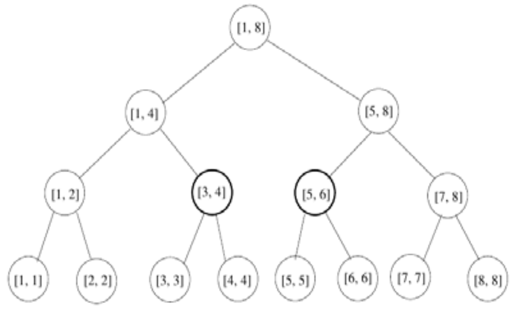

Figure 1. Segment tree for the interval *[1, 8]*

The above lemma has a direct algorithmic interpretation [7] given below (where *lo* and *hi* are the endpoints of interval *I*, and *left* and *right* are the endpoints of the current interval):

```
SplitSegment procedure (lo, hi, left, right, collection):
    if lo<=left and hi<=right then
        collection.add([left,right]);
        return;
    med = (left + right)>> 1;
    if lo <= med then
        SplitSegment (lo, hi, left, med,
collection);
    if hi > med then
        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 overloading 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 [7]:

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

The disadvantage is that insertion of a key can not be performed in parallel in all the ranges containing it.

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

When node *x* receives a request, it will first check whether $Cx \geq M$. 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 $Cx = M$ 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 $Cx=M$ 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 $2*L - 1$ 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

As mentioned before, an additional logical level was required in order to maintain a DHT in the system. The ideas presented below are similar to the Chord architecture, introduced in [5]. Only peers will 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*.

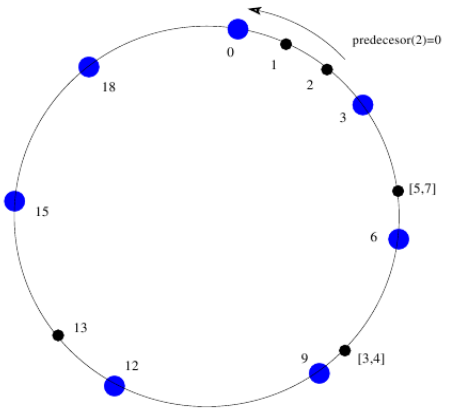

Figure 2. Peers arrangement and data distribution

When inserting an object into the system, the message containing the object will be routed according to a protocol described in the next section. The object will be stored on the predecessor peer. Look-up calls must contain the object identifier in order to find it. Thus, keys stored in the segment tree will also be assigned associated identifiers.

When inserting a peer in the system, some items will be relocated. The new peer's predecessor will have 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 1 (proven in [14]), that states the following:

Theorem 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 [15].

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,

$$Id_2 = Id_1 \text{ XOR } 2^{m-1} \qquad (1)$$

This operation is equivalent to a half-circle trip, so the replica *ID* is diametrically opposed to the master copy's. 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

In order to perform the look-up operations, we need a new routing protocol that efficiently maps onto the proposed architecture. The Chord routing protocol [5] demanded $\log_2 N$ 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

$$C_1 * C_2 \geq k \qquad (2)$$

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:

```
// Peer P routes the message m
route procedure (m):
   if (m.stage == ToSuper) then
   // P is a peer on the ring
         m.stage = ToRing;
         sendMessage (m, super_p(p));
   else
         if (m.stage == OnRing) then
               m.stage = ToSuper;
         else
               m.stage = OnRing;
      sendMessage(m,pred(m.getID()))
```

Messages are routed in *3* step stages. 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.

Protocol performance is given by the lemma:

Lemma 2. If the number of clusters is independent of 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 $3*C_1*\ln k + 4$ peers from the initial source to the destination (C_1 is the number of considered successors).

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 $3*X + 1$.

Then, the first $3*(X - 1)$ nodes are part of clusters that do not have have members among the C_2 predecessors of d, otherwise the message would have reached the destination in one step. Also, these nodes are part of pairwise distinct clusters.

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

Choosing $i = C_1 * \ln k$ from *(2)* results that

$$\mathbb{P}[X \geq C_1 \ln k + 1] \leq \frac{1}{k} \qquad (3)$$

So, with probability of at most $1/k$, the message reaches more than $3 * C_1 * \ln k + 4$ 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 routing time, while a smaller number of links increases scalability, but also requires more routing. The parameters C_1 and C_2 should be chosen depending on the application.

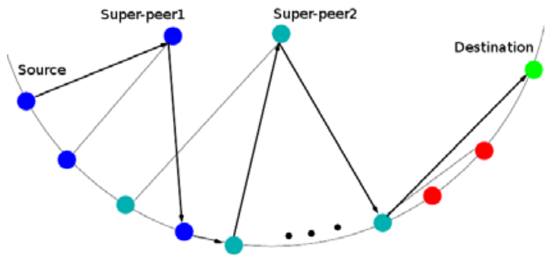

Figure 3. A message routing example

3.4. Integration of system elements

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

A super-peer will only keep the necessary routing information. It will maintain the list of peers in the corresponding cluster, and for each such peer, the associated identifier. Identifiers will be stored in a structure that allows predecessor and successor queries in logarithmic time: for example, a balanced tree.

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

Each peer/super-peer will run 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 will be routed directly to the destination via TCP / IP.

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

There will be one distributed segment tree for each attribute. The values of an attribute will be mapped into a finite range of consecutive integers beginning from *1*. The mapping scheme is not a responsibility of the proposed system. It is necessary, however, that 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 will be 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.

The cluster splitting property is still maintained. To meet most of the requirements of Lemma *2*, the division will be uniformly random (each node will be part of a cluster with probability of *1/2*). Thus 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 [2], the actual size will be closer to the average with exponentialy 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* 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 node to its own list.
4. *P* informs *S* that it has completed its initialization and it is part of the cluster.

A peer *P* may fail. In this case the following operation is performed:
1. 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 its successor.

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.

We also amend the protocol for cluster division (presented in [13]). The only added operation is: Peer *P* (the future super-peer of the new cluster) initiates its leave – described in the above paragraph (steps *1-3*) and exits the ring, but remains in the system.

Based on the protocols above, we determine the number of messages generated by each operation.

Lemma 3. The required number of messages for each operation is as follows:
(i) a peer joining the system requires no more than $4*C_2 + 3$ messages.
(ii) failure of a peer requires at most $4*C_2$ messages in order to restore the network structure.
(iii) the announced departure of a peer P of the system requires no more than $2*q + 1 + 4*C_2$ messages, where q is the number of active connections of P.
(iv) Splitting a cluster by choosing a peer P and a set of nodes S requires no more than $2*q + 1 + 4*C_2 + 3*|S|$ messages. (q is the number of active connections of P.)

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

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 $4*C_2$ messages.

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

Finally, when splitting a cluster, peer P will become the new super-peer. Leaving the ring requires at most $2*q + 1 + 4C_2$ 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 $2*q+4*C_2+3*|S|+1$ 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 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

Fault tolerance is ensured by object and metadata replication, according to the two hash functions and the number of links between peers. The new message routing protocol enables messages to reach any peer, if every peer can connect to its immediate successor.

For example, if each peer has p successors, and a peer fails with a probability of $1/2$, there is a $(1/2)^p$ chance for that peer to remain without successors. With $p = 10$, the probability is substantially low.

4. Experimental results

The system was implemented using the Java programming language. In order to evaluate the proposed system, we considered the following aspects:

- numbers of objects stored on a peer
- numbers of messages per operation
- evolution while varying some parameters

For testing purposes we created a client database having *100* objects. Each object had *2* attributes, with values in the interval $[1, 100]$. For each *Object(i)*, the attribute tuple was $(i+1, 100-i)$, where $0 \le i \le 99$.

4.1. Object distribution

In this test we wanted to evaluate peer load balancing for a fixed number of objects and a varying number of peers.

We used *100* data objects, having attributes as discussed before. Every object was replicated once (*100* more objects – the replicas) and for each attribute we constructed a metadata segment tree (approx. 200 nodes in each segment tree – as discussed in section *3*). In total, we had about *600* objects (*100* data objects, *100* data object replicas, *2 x 199* segment tree nodes).

First we added *1* peer, then all the objects (*100* data objects - *600* system objects). We kept adding peers, until reached *100* (consecutive IPs, same listening port). Figure 4 presents the distributed system load with *10*, *20* and respectively *40* peers. We observe that in every case there are a few overloaded nodes, but most of them have a load close to the average value.

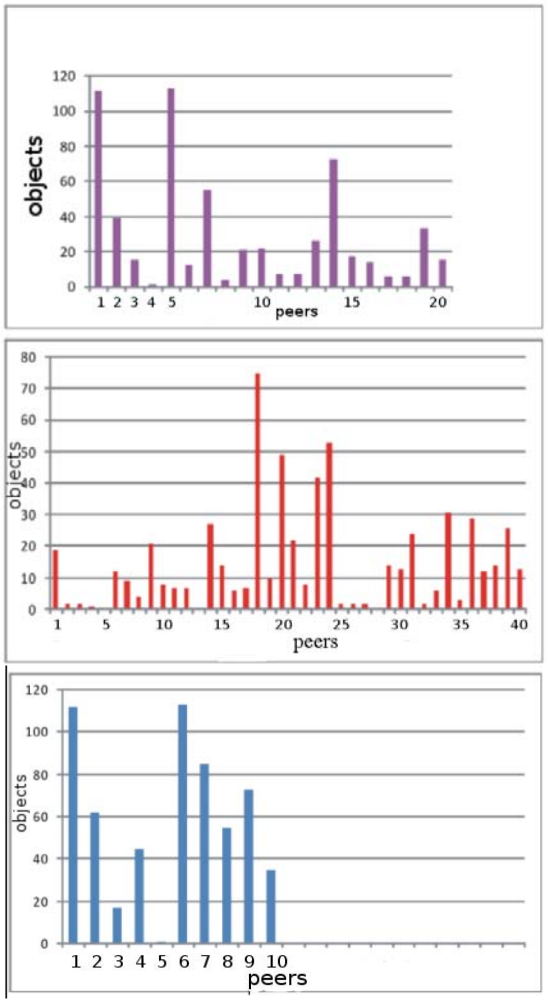

Figure 4. Objects distribution for *10* , *20*, *40* peers.

4.2. Bandwidth consumption

We considered the following scenario:

1. 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
5. Range search for all objects with attributes in *[1, 1]*

For the aggregate bandwidth consumption of all the peers in the system, we obtained the graph in Figure 5. The first peak, $0.4x10^{-4}$ Mbps (seconds *10-15*) is caused by peers joining the system. At second *25* all *10* peers entered 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]*. The bandwidth consumption is very low (less than 0.1×10^{-4} Mbps).

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

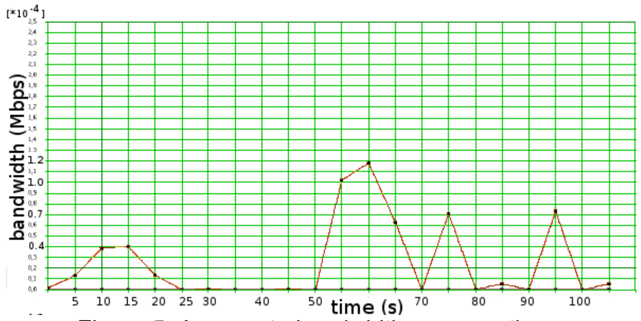

Figure 5. Aggregate bandwidth consumption

4.3. Number of generated messages

In order to measure the number of transmitted messages, we considered a system of *18* peers. A segment tree node had a limit of *10* keys; $C_1 = C_2 = 3$ (number of neighbors is 6).

We performed the following tests:

1. We stored *100* data objects as in section *4*. As seen in *4.1, 100* data objects create *598* objects. The number of messages was *3128.* (Lemma *4* estimated *3400*)
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* mesages (*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*.

We observe that all tests in this section performed as expected from a theoretical point of view.

5. Conclusions and future work

In this paper we presented a scalable peer-to-peer architecture which 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. Experimental results have shown that the proposed architecture works as expected and the number of messages issued by our system is predicted quite well by our theoretical analysis. As future work, we intend to run a performance evaluation of our system on a larger scale and tune the implementation accordingly.

Acknowledgement

The work presented in this paper has been supported by CNCS-UEFISCDI under the research grant DEPSYS, project code ID_1710/2008, PN II - IDEI program, by national project TRANSYS, Contract No. 4/28.07.2010, Project CNCSIS-PN-II-RU-PD ID: 238, and by the Sectoral Operational Programme Human Resources Development 2007-2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreements POSDRU/6/1.5/S/16 and POSDRU/89/1.5/S/62557.

6. References

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

[2] T. Hagerup, and C. Rub, "A guided tour of Chernoff bounds", *Information Processing Letters* **33** (6): 305–308, 1990.

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

[4] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, 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.

[5] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, 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.

[6] P. Maymounkov, D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric", in *Revised Papers From the First international Workshop on Peer-To-Peer Systems,* Lecture Notes In Computer Science, vol. 2429. Springer-Verlag, London, pp. 53-65, March, 2002.

[7] C. Zheng, G. Shen, S. Li, and S. Shenker, "Distributed Segment Tree: Support Range Query and Cover Query over DHT", In *Proc. of the Intl. W.-shop on P2P Syst.*, Santa Barbara, California, 2006.

[8] N. Lopes, and C. Baquero, "Implementing Range Queries with a Decentralized Balanced Tree over Distributed Hash Tables", *Lect. Notes in Comp. Sci. 4658*, pp. 197-206, 2007.

[9] M. Abdallah, and E. Buyukkaya, "Efficient Routing in Non-Uniform DHTs for Range Query Support", In *Proc. of the Intl. Conf. on Par. and Dist. Comp. and Syst. (PDCS),* Dallas, TX, USA, pp. 239-246, 2006.

[10] M. Hauswirth, and R. Schmidt, "An Overlay Network for Resource Discovery in Grids", In *Proc. of the Intl. W.-shop on Database and Expert Systems App.*, Copenhagen, Denmark, pp. 343-348, 2005.

[11] M. I. Andreica, E.-D. Tirsa, and N.T. Tapus, "A Fault-Tolerant Peer-to-Peer Object Storage Architecture with Multidimensional Range Search Capabilities and Adaptive Topology", In *Proc. of the 5th IEEE International Conference on Intelligent Computer Communication and Processing (ICCP)*, Cluj-Napoca, Romania, pp. 221-228, 2009.

[12] 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*, 6(1), 2002.

[13] C. Dobre, F. Pop, V. Cristea, "A fault-tolerant approach to storing objects in distributed systems", *3PGCIC 2010, International Conference on, P2P, Paralel, Grid, Cloud and Internet Computing*, pp. 1-8, 2010.

[14] 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". In *Proc. of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663. ACM New York, NY, USA, 1997.

[15] B. Mulvey. "Hash functions. Evaluation of SHA-1 for Hash Tables", http://home.comcast.net/~bretm/hash/9.html, accesed February 2011.