A MEANS OF ENHANCING STORAGE RELIABILITY FOR PEER-TO-PEER NETWORKS

Voichița IANCU[1]

Rețelele peer-to-peer sunt sisteme distribuite ideale pentru a stoca date replicate, în scopul de a le putea obține rapid şi transparent, orideunde am dori. O problemă importantă, când vine vorba de aceste sisteme distribuite, este replicarea datelor critice într-o manieră persistentă. Această problemă o adresăm prin lucrarea de față, propunând un mecanism suplimentar de asigurare a redundanței, în plus față de mecanismul natural al DHT-ului Chord, de replicare pe un număr fix de succesori. Deşi am prevăzut pentru cercetarea viitoare rularea unor teste intensive de stres pentru prototipul curent, chiar şi rezultatele obținute până în prezent par să confirme capabilitățile de a adapta gradul de redundanță datelor critice atunci când e necesar sau la cerere.

Peer-to-peer networks are ideal for storing replicated data, in order to be able to rapidly and transparently retrieve it, wherever we are. A main issue concerning these distributed systems is to replicate critical pieces of data in a persistent manner. It is the issue we are addressing in this paper, by proposing a supplementary redundancy mechanism on top of the natural Chord DHT replication mechanism on a fixed number of successors. Even if some more stressful tests are envisaged in our future research, so far the prototype we have developed seems to cope well with adapting the redundancy of critical data at need and on demand.

Keywords: peer-to-peer, Chord, DHT, adaptability, reliability, redundancy, logical infrastructure, physical infrastructure

1. Introduction

Distributed systems have evolved a lot, and the new "subspecies" of fully decentralized distributed systems, the peer-to-peer infrastructure, has become a real logical support for storing data. This aspect has lead to the problem of reliability and persistence in data storage, which are closely related to the term of redundancy. The more sensitive the stored data is, the more need there is for that data to be stored with a higher redundancy, so that it will survive any node failure situation. We consider storage peer-to-peer networks as being logical storage infrastructure, which can run on top of various physical infrastructures, such as a group of otherwise totally independent, distant, nodes, a small local network or a fully organized complex grid infrastructure. The logical storage infrastructure

[1] PhD Student, Depart. of Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: voichita.iancu@cs.pub.ro

should totally hide the number of nodes and the type of the physical infrastructure, while offering the requested storage service. In order to obtain a solution that would be fit for *reliable* data storage, we have analyzed some of the existing peer-to-peer system's problems and will describe these in section 2. Section 3 will discuss and propose a mechanism for solving the storage reliability problem, while section 4 does a preliminary evaluation of the prototype developed based on section 3. Finally, section 5 will conclude the paper, also giving a glimpse upon the future work we will need to be performed in this field.

2. Related work

Because of the popularity of peer-to-peer public applications, such as KaZaA[1], Skype[2], etc., a great interest for peer-to-peer systems has grown in the research environment, too. This is how the first Distributed Hash Tables (DHTs) have appeared around the year 2000: *Chord* [3], *Pastry* [4], *Tapestry* [5], and *CAN* [6], followed by a lot of research investigating their characteristics. These DHTs are very similar when it comes to their underlying concepts: all of them are based on a hash function with very good dispersion capabilities, such as *SHA-1* [7], and all of them are able to perform routing based on a unique ID, given by the hash function, by keeping some sort of routing tables. Chord, for example, defines a ring of nodes, based on the ID resulted from the hash function applied to the node's public IP, which should be unique. When wishing to store some data on the Chord network, the same hash function is applied to the identifier of the data to be stored (a filename, for example) and this way, the closest preceding node to the data's Chord ID (also called a *key*) is the node responsible for storing and managing that piece of data. In [3], a redundancy mechanism is also defined, to prevent the data from being lost if the node responsible for its key leaves the network. This redundancy mechanism consists in replicating the key (and the data) on a fixed number of successors (r), the first of which will actually become the successor of the key, in case the old successor of the key fails.

If we ensure that the hash function is good, giving IDs very far from each other for very similar inputs, because of the geographic localization of IP addresses [8], it would mean that the nodes that are close on the Chord ring are very far geographically, so it is even more unlikely for neighbour nodes on the Chord ring to fail at the same time. Since the probability of a node failure is less than 1, by knowing the fact that the events of node failure are independent for nodes close on the Chord ring, we can conclude that the higher the replication factor is, the less probable it will be to lose the data stored with that replication factor. However, the redundancy factor is statically defined, for any piece of data, so if a piece of data stored on a node is more critical we cannot replicate it more

than less critical pieces of data by using currently existing layers built on top of DHTs.

The problem of having different redundancy factors when storing different pieces of data has been addressed in the context of peer-to-peer applications running on top of the grid [9, 10], such as JuxMem, a grid data sharing service [11]. In that context, the main idea behind the developed prototype was for the upper layer application to require a degree of redundancy, translated in a number of different storage nodes by the storage peer-to-peer layer. If the data sharing service realized that there weren't enough peer-to-peer nodes to assure the degree of redundancy, it would arrange for some extra nodes to be reserved on the grid, by querying the grid management system, and mainly the grid resource scheduler, described in [12]. The reservation of the extra nodes required for the redundancy of the data is done best-effort, since no one can assure that at a certain moment the required amount of nodes is actually available. After reserving the nodes, the peer-to-peer storage service is extended on top of them and the data can further be stored in the required redundant manner.

The advantage of the proposed prototype for the on demand, self-extending grid data sharing service, described above, is the fact that you do not have to reserve all the nodes on the grid for the service at once, resulting in less power consumption by the application and also in better sharing of the grid resources. The 2 disadvantages of the already mentioned prototype, that we would like to point out, are (1) the fact that it cannot be easily used for a set of nodes that do not belong to a grid and (2) the fact that it is not intimately related to the dynamics of the peer-to-peer topology, which could be desirable, for easier and faster reconfiguration of the topology. In what follows, we propose to extend this idea by designing and implementing a model which can be used for self-extension on top of any set of storage peer-to-peer nodes. Furthermore, we will try to build the replication layer as close as possible to the DHT layer, in order for it to use the DHT infrastructure's intrinsic stabilizing mechanisms in case of node leave and node join.

The solution we propose for the presented problem will be described into detail in section 3. It will be implemented on top of and in close relation with the Chord DHT. It will take advantage of the "self-regenerating" infrastructure that Chord creates and it will also benefit from the use of a special layer that is able to extend the Chord service on other, new, nodes.

3. System architecture and behaviour

The model that we propose is a flexible means for a distributed decentralized system to extend itself on other available nodes. The reason why this model has been conceived and implemented is the fact that, even if they have

many qualities, DHTs are not capable to grow indefinitely and, thus, offer, just by themselves, more scalability and more reliability for the storage of a key. One of the reasons why this cannot happen is the fact that the redundancy factor is usually intrinsic to the entire DHT, so one cannot vary it for a certain piece of data. This is mainly due to the approach, different from ours, that Chord has when storing data: it tries to do its best not to lose data, by using all of the nodes in the peer-to-peer network. Our approach would be to allow the possibility to insert new resources – namely, nodes – into the peer-to-peer network, if, at some point, we wish to increase the network's reliability.

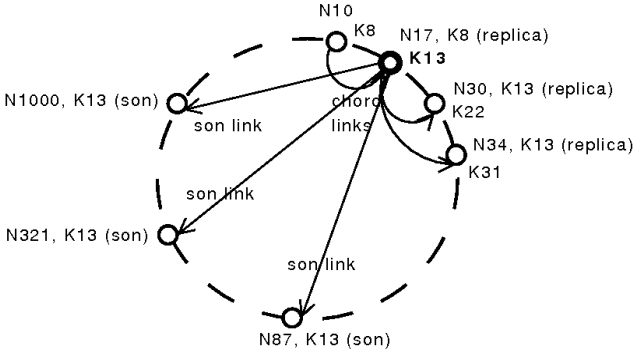

Fig. 1. Storage on top of the improved peer-to-peer DHT.

How does a peer figure out when to insert new nodes into the peer-to-peer logical infrastructure? In this paragraph we will describe some preliminary ideas about means to extend the peer-to-peer infrastructure, only when needed by the upper-layer application. For the sake of simplicity of presentation of our solution, let us assume that any node in the Internet, on which we would wish to extend the peer-to-peer infrastructure, already has this software installed and ran by default at start-up. If we wished to include this node into the Chord network, as to obtain more reliability, we would have to physically access the node, in order to turn it on, since based on our assumption, the node is certainly offline if it is not running Chord at the moment. This is something that would be desirable when dealing with a *remote* node, which we would wish to turn on and include into the Chord network only at the moment when it is needed, and not from the very beginning, so that we avoid useless power consumption, for example. We remind the reader that in section 2 we have described reasons why it is desirable to redundantly store data on nodes belonging to distant, and, thus, different, domains, if possible. To address more elegantly the problem of starting up remote servers, we will design our special layer, for increased reliability in the Chord network, to make use of the Wake on LAN protocol for on demand node startup [13], by configuring the node's NIC accordingly. Another thing one needs to know when considering starting-up new nodes, which will be integrated into the

peer-to-peer infrastructure, is on what base the decision to launch new peers should be taken. One might consider 2 levels for making this decision, and thus 2 ways to make this decision, which could either coexist, or might be considered independently.

The upper-level application *requires* a certain redundancy for some piece of data. This means that the redundancy can vary among different pieces of data to be stored, but on the other hand it remains the same for one piece of data throughout its existence within the peer-to-peer logical infrastructure. In this situation, the node responsible for the key of that data evaluates the number of nodes within the logical infrastructure and stores the data on top of the required number of peers, after it first instantiates new nodes which will become peers into the network.

The peer-to-peer infrastructure *decides* when a new node should be launched, to hold a replica of a certain piece of data. This is done by accounting the frequency of accesses to one piece of data divided by the number of replicas for that data and, based on the value obtained and on some thresholds, decide to instantiate new nodes to hold that piece of data. This method is applied by the peer-to-peer infrastructure itself, in order to adapt for data which has previously been stored on the logical infrastructure, perhaps even with a certain redundancy specified by the upper-level application.

From the 2 methods for determining data redundancy, our prototype will mostly concern itself with the latter, the one in which the peer-to-peer infrastructure adapts to hold more copies of the data if that piece of data is being accessed intensively, since this is what we consider to be a novel approach in our work. Also, the prototype will try to benefit from the static redundancy factor which the Chord DHT is configured with, too. The prototype will not concern itself with the consistency of the data being stored, leaving this task to the upper-level application.

How is the prototype for self-extending of the peer-to-peer logical infrastructure designed? In this paragraph we will describe the architecture and behaviour of the prototype we have designed and implemented, which enables self-extension of the logical infrastructure, at the moment when this is needed. As already stated, a peer responsible for a key is able to determine when the data that key refers to is more critical than other data, and thus should be stored with a higher degree of replication. What hasn't been mentioned yet, is how does a peer know what nodes to start up , in order to extend the peer-to-peer infrastructure. To be able to do this, the peer should know of the existence of other machines, that

could be integrated into the its peer-to-peer network. This information will be contained in a configuration file, containing all the nodes that the current node will be able to integrate into the peer-to-peer network, the moment this is needed. To allow for more storage reliability for the data to be replicated, within the configuration file, these nodes should be statically grouped by their geographic neighbourhoods. Before trying to start-up a node which will integrate itself into the peer-to-peer network, the current peer will ping that node, in order to see if it isn't already running. If a peer is already running and still has enough storage space, the current node may decide to redundantly store the data on it. When a peer is launched in order to store a piece of data with higher redundancy, we will say that it is the *son* of the node that has launched it, and, conversely, the node that has launched it is its *parent*. For now, the parent part of a peer, that is, the part that is responsible for deploying new nodes, cannot perform replication for different pieces of data in parallel, the pieces of data that need deployment being treated sequentially.

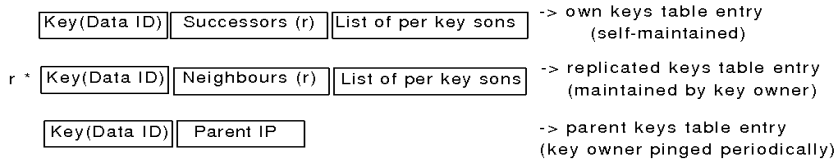

Fig. 2. The 3 types of tables containing keys, that can be stored by a node.

Even if this situation is unlikely, in our implementation, if a node becomes successor of a key, or a key is also replicated on a son-node via successor mechanisms, the node will eliminate the key from its set of keys belonging to its parent-node and should announce its parent to decrease the degree of redundancy for that key by 1. In other words, the son-node will announce the parent-node that it is no longer its son, but it has now become its successor. We can conclude from here that each node holds 3 disjoint categories of keys grouped into 3 different tables, as one can see in Fig. 1, and they are as follows:

1. The keys that have the node as their successor are the keys that the node is responsible for, and they are kept in a first table. For these keys it accounts the frequency of access and decides replication of these keys on its successor nodes and on son-peers accordingly. It also accounts all the peers that these keys were replicated on, as can be seen in Fig. 2.
2. The keys that have been replicated on the node by its predecessors are keys for whom the node keeps a second table, similar with the one kept for its own keys. The node is not responsible for updating replication

information related to those keys, which is the responsibility of the owner of each key, and it should do this periodically.

3. The keys that have been replicated on the node by its parent-node, if it has one, are keys for whom the node keeps a third table, which is never updated, and contains just the key. In fact, the son-node is responsible for periodically pinging its the owner of the key (i.e., its parent-node) via the Chord routing ring, in order for the parent-node to be able to keep the table for its keys up to date. This third table can be seen in Fig. 2, too.

It results that a node can have 2 main roles within the improved Chord-based DHT, that we have implemented: as a *routing node* and as a *storage node*.

As a storage node, it is responsible for: periodically announcing its parent, the node responsible for the key this peer holds, of its existence; detecting the case when it has a double role, being the son of the owner of the key and also one of the *r* successors of the owner of the key.

As a routing node, it identifies its keys and takes care of the data that these keys refer to, by ensuring its correct replication. Each node stores replicas of its data on a fixed number *r* of its successors and on its son-nodes. If the routing node detects that one of its successors is no longer alive, it will obviously update its successor list, by using Chord mechanisms, and it will also ensure that its keys are stored on all of its *r* successors that it keeps track of. If the routing node no longer receives ping/heartbeat messages from a son-node, it will decide to replicate the keys replicated on that son on other, possibly newly-launched, son-nodes.

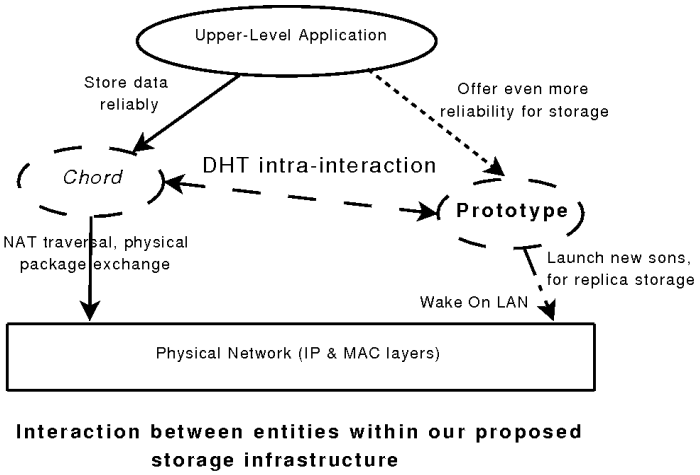

Fig. 3. Interraction between the different entities involved in the storage of data.

From an architectural point of view, with respect to the position of Chord within the software stack, the prototype we have designed is placed mostly between the Chord layer and the upper-level application layer. Still, some features included by our prototype, such as the Wake on LAN module, are placed at a layer below Chord. The interactions among the following 4 entities: the upper-level application, our prototype modules, the Chord network, and the physical network, which have all been described above, can be observed in Fig. 3.

How does the proposed model handle node volatility? In this paragraph we will describe how our proposed prototype handles node join and node leave, in a Chord-like manner.

Node joins are handled mainly at Chord-level, with some additional actions performed by the prototype's modules. The fact that the hash function's values for the node IDs of each node cannot be anticipated is the main reason why a node cannot know where its replicas will be stored on the Chord network and that's why a freshly started son-node should perform some extra actions, and that son-node is in most of the cases not a successor of its parent. For any new node that joins the peer-to-peer infrastructure, after running the stabilization algorithm on top of its predecessors for the first time after its join, the node will receive the keys for whom he is responsible from its immediate predecessor and also keys which it should keep for its further *r-1* predecessors, in order to attain the fixed redundancy *r*. Besides this, if the node has been launched as a son, to hold one of its parent-node's keys or data, when joining the peer-to-peer network the node will also ping its parent for the first time, to notify it of its presence. As already stated, if a node does not receive a ping for either of its keys replicated on a son from that son-node, it will decide to start up a new son-node to hold the key, or query other son-nodes which might still have space left for storing the data for that key. Also, after launching a new son-node for a key, the node responsible for that key might need to update the key tables on its successors.

Node leaves or node failures are handled both by the Chord-level and by the prototype, depending on the types of keys we are referring to. The Chord action for node leave is the stabilization method called periodically for any node, combined with the fact of making sure that a fixed number of *r* replicas is kept on the successors of the node responsible for that key. In case of node leave, the main concern is not to lose a certain key entirely. If the node responsible for a key fails or leaves the logical infrastructure, but one of its *r* successors is still up, the problem is solved, since the new node responsible for the key already has the data associated to it, and will only

need to send the key, and the data associated to it, to all of its *r* successors in its successor list. If it happens, even if it is very unlikely, that all the *r* + *1* successors of the key fail or leave the peer-to-peer infrastructure, there will be a new node responsible for the key, that has never heard of it. The means by which this node finds out about the key is the periodic ping that the son-nodes send to the node responsible for the key, which in the Chord ring is the immediate successor of the key. If the key is stored on no sons, that means that it hasn't been associated to a critical piece of data, and the loss of the data is not such a big issue. The steps for a parent-node to "recuperate" a key and the associated data from a son-node are as follows:

1. If receiving a ping from a node about a key that the current node does not know, it will ask the distant node for the data related to that key.
2. If a son-node receives a request such as the one at step 1, it will update the IP of the owner of the key in its table.
3. Upon receiving the data from the son-node, the node responsible for the key will add it in the corresponding table, together with information about the son-node itself.
4. The node responsible for the key will create *r* replicas for the key on its successors from the successor list.

If a node receives a ping message from a son-node that it does not know, but about a key it already knows, it will only send the distant node a reply with its IP, and after that steps 2 and 3 from above will be performed.

It is now clear how our prototype, together with Chord, cope with the dynamics of the peer-to-peer infrastructure, that is with joins and leaves, in order to keep all of the 3 tables described in Fig. 2 up to date and consistent with the replication of the keys on top of the logical infrastructure.

4. Evaluation and experimental validation

For the evaluation of this prototype, a testbed was used, composed of more virtual machines running on top of different computers, in order to give the ability to have as many nodes as needed, even if there were not enough physical nodes available. In this section we will evaluate mainly the engine for instantiating the peer-to-peer applications on new nodes. We will measure (1) how much it takes to launch a number of nodes that are each situated at approximately similar physical network distance from the parent-node and (2) how much it takes to "restore" data from a son-node to the rightful owner of that piece of data.

Time necessary to start up new storage peers. When considering the time necessary to start up new storage peers, by using Wake on LAN, one must take into account the time it takes for that machine to boot. From our repeated measurements on more general-purpose machines, we consider that a boot-time of 75-90 seconds is relatively fair. We have performed 2 experiments, in order to study the variation of the time for launching new nodes, with respect to the number of nodes launched and especially with respect to the moment when they are launched. The measurements within these experiments have been performed as to reflect the time elapsed between the moment of starting to launch a new son-peer and the moment that that son-peer sends the first ping to its parent.

Instantiating a new peer each 5 minutes. The experimental setup was to have a client accessing intensively a piece of data stored on a peer, such that every 5 minutes a new son-peer had to be launched, in order to assure the degree of replication needed by that critical piece of data. The result of the experiment can be seen in Fig. 4, which clearly shows that the instantiation of each different son-peer does not interfere with the already existing son-peers. Moreover, one can notice how low the overhead for launching a new peer is, when compared to the time it is needed to boot that peer.

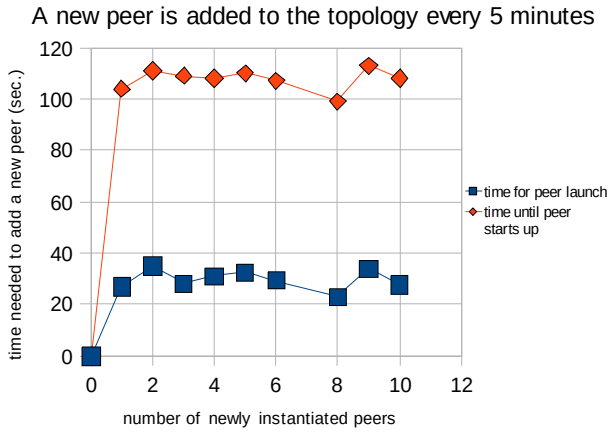

Fig. 4. Instantiating a new son-peer each 5 minutes.

Instantiation of a higher number of peers each 5 minutes. The experimental setup was to have a very volatile infrastructure, which would enable loss of son-peers on which a piece of data is replicated. A case in which starting up more nodes in parallel would be needed, is the case when all of the son-peers holding a key are lost. For the current experiment, an

infrastructure with different numbers of son-peers was used, for each key whose data was stored by a unique peer within the peer-to-peer infrastructure. Every 5 minutes, the son-nodes responsible for a chosen key were turned off, in order to force the unique parent-node to start up a new set of son-nodes for that key. The time necessary for launching more peers at the same time varies linearly with the number of peers, as can be seen in Fig. 5. There is no extra-overhead for booting each node, since the parent-node, responsible for the key, does not wait for each son-node to confirm it has properly booted, before launching the next son-node.

The 5 minutes interval was chosen as to dominate the time needed for Chord the stabilization method to finish, thus making all the other nodes aware of the newly joined node. The stabilization interval has been set to 1 minute, for each node belonging to the Chord network.

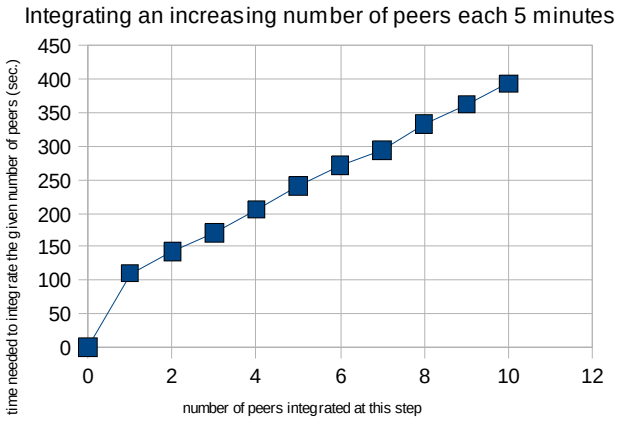

Fig. 5. Instantiating a higher number of peers each 5 minutes.

Time required to replicate data on the real owner of the key. As seen in the previous case, replication on more than one extra-node might be needed in some cases, such as the case when some son-peers responsible for a node are gone. On the other hand, an even more serious situation is the case when the owner of the key is gone, together with all of its r successors that hold the replica of the key. This is a situation in which, for a usual Chord-like DHT, the key and its associated data would be lost, which is something that we would not want to happen, especially for some critical, heavily accessed pieces of data. In this case, within our protocol, son-nodes might still keep the key, thus increasing the reliability of the DHT infrastructure, and we wish to study how this distributed infrastructure behaves when it comes to making a supposedly lost key re-appear,

of course, together with its corresponding data. For this, an experiment has been set up, as follows:

1. A number of 16 peer instances has been launched on top of the physical nodes, each peer storing 7 keys, with a value of *r* equal to 3. All of the nodes were connected to each other via a LAN, in order to make as uniform as possible the network delays between them.
2. Among these peers, there also exist son-nodes for some keys stored on parent-peers.
3. For a given key, the node responsible for it plus all of its *r* successors have been turned off, so that only the son-node holding the key remained alive, and the time necessary for the key to be stored on its new "parent" has been recorded.
4. The previous steps have been repeated 10 times, in order to obtain the numbers that lead to Fig. 6.

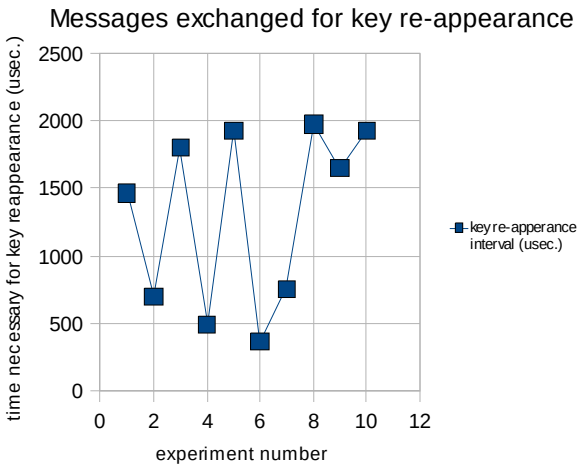

Fig. 6. Replicating the data on its new owner

One can notice, from Fig. 6, that the time needed for the messages that restore the key to be exchanged is about 2ms. Note that this time does not include: (1) the network latency that would exist in a real case, when nodes are geographically distributed; (2) the time interval defined for the son-node to ping its parent, which should be configured to be neither too high, nor too low; (3) and the time necessary to download the data corresponding to the key, which could be very large. These time intervals were not taken into consideration when plotting Fig. 6 because they can vary very much, and because anyway they would dominate the time for message processing among peers. In the future work, it

would be desirable to also take into consideration at least a medium-case network latency, in order for the results to be even more relevant for the value of the prototype that we have developed. Nevertheless, even with the overheads introduced by these time delays, the use of our prototype, and of our proposed redundancy storage mechanism is desirable, compared to the situation of losing some pieces of data that are very valuable to the upper-layer application.

To compare the performances of our prototype with other existing solutions, we have looked at its performances both compared against the Chord intrinsic replication mechanism [3] and against the JuxMem data persistence self-extending mechanism [10], already mentioned in the second section of this article. In the latter case, we are talking about an application that interacts with the peer-to-peer infrastructure. As already mentioned in section 3, when comparing our prototype against a non-enhanced Chord network, it introduces a small data persistence maintenance overhead, but this is the cost of far more reliability, illustrated in the case where we restored the data from the son-node on its new owner, from above. On the other hand, if we compare our extension mechanism which enhances reliability with the one proposed for JuxMem, they turn out to be very similar, in architecture and performance, the major difference between the two being the fact that the current solution can run on any physical infrastructure, not just on grids, which represents an advantage.

More aspects concerning the effective reliability obtained by the storage system will be evaluated in the future work, given the fact that storage reliability is intimately related to the nature of the upper-layer application.

6. Conclusions

In the previous sections we have seen the description and evaluation of a prototype trying to solve the problem of enhancing storage reliability on top of peer-to-peer logical infrastructures. The most important aspect that this paper has tried to address was the fact of making logical storage infrastructures more fail-safe than their predecessors. We believe, based on the evaluation we have performed in section 5, that we have succeeded to improve the storage infrastructures from this point of view, by defining mechanisms that prevent the loss of critical data throughout its lifetime within the system. This was done both by giving the higher level application the possibility to define a degree of redundancy for the critical pieces of data and by enabling the peer-to-peer storage network to use different degrees of redundancy for different pieces of data, depending on the importance of that data.

In the future we would like to focus on further developing this prototype, for more geographically distributed infrastructures, in order to make stress tests on the redundancy mechanism we have implemented. Of course, before performing

these tests, some optimizations could be performed regarding the current prototype, such as the fact of trying to reduce the number of messages exchanged among the peers, based on analyzes such as [14], or the fact of enabling a degree of parallelism when launching new requests for creating son-nodes related to different keys.

R E F E R E N C E S

[1] *** KaZaA Peer-to-Peer storage system – http://www.kazaa.com/
[2] *** Skype Peer-to-Peer VoIP system – http://www.skype.com/
[3] *I. Stoica, R. Morris, D. Karger, F. Kaashoek, H. Balakrishnan,* Chord: A scalable Peer-To-Peer lookup service for internet applications. In Proceedings of the 2001 ACM SIGCOMM Conference, pages 149–160, 2001
[4] *A. Rowstron, P. Druschel*, Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pages 329–350, Nov. 2001
[5] *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, 22(1):41–53, Jan. 2004
[6] *S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Schenker*, A scalable content-addressable network, In SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, volume 31, pages 161–172, ACM Press, Oct. 2001
[7] *** SHA-1 – secure hash standard – http://www.itl.nist.gov/fipspubs/fip180-1.htm
[8] *** IP geographical location – http://www.geobytes.com/
[9] *V. Almăşan,* A monitoring tool to manage the dynamic resource requirements of a grid data sharing service, Master report, Master de Recherche en Informatique, IFSIC, ENS Cachan, Antene de Bretagne, France, June 2006
[10] *L. Cudennec, G. Antoniu, L. Bougé*, CoRDAGe: towards transparent management of interactions between applications and ressources. In International Workshop on Scalable Tools for High-End Computing (STHEC 2008), pages 13–24, Kos, Greece, 2008
[11] *G. Antoniu, L. Bougé, M. Jan, S. Monnet,* Large-scale deployment in P2P experiments using the JXTA distributed framework, In Euro-Par 2004: Parallel Processing, number 3149 in Lect. Notes in Comp. Science, pages 1038–1047, Pisa, Italy, August 2004
[12] *F. Pop, V. Cristea*, Optrimization of scheduling process in Grid environments. UPB Scientific Bulletin, Series C: Electrical Engineering, Ed. Politehnica Press, ISSN 1454-234x, 2008
[13] *** Wake on LAN protocol – http://www.liebsoft.com/pdfs/Wake_On_LAN.pdf
[14] *M. I. Andreica, E. D. Tîrşa, N. Ţăpuş*, A Peer-to-Peer Architecture for Multi-Path Data Transfer Optimization using Local Decisions, In Proceedings of the Third Workshop on Dependable Distributed Data Management, pages 2-5, Nuremberg, Germany, May 2009.