Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 142 (2006) 63–77

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Distributed Backup through Information Dispersal

Giampaolo Bella[1], Costantino Pistagna[2], and Salvatore Riccobene[3]

Dipartimento di Matematica e Informatica, Università di Catania
Viale A. Doria 6, I-95125 Catania, ITALY

Abstract

The formal aspects underlying a novel distributed backup service are discussed. Strength and originality of the service lie in the combined adoption of an established information dispersal algorithm with a simplified version of an existing location service. Information dispersal makes our service threshold-secure in that the backup owner only needs participation of a pre-established threshold number of nodes to recompose a distributed backup. This means that the service is highly available as it tolerates a number of node breakdowns. Even the right threshold number of nodes cannot retrieve the backup on their own initiative. The location service adopted allows our service to work over non-organized, flat networks. Indirect advantages are the optimization of the total redundancy of data and the efficient management of resources. Our service has reached the stage of a proof-of-concept implementation.

Keywords: Backup service, flat network, information dispersal, location service.

1 Introduction

The Internet is a peer-to-peer network in the sense that a sender must specify the address of his intended receiver for communication to get through. Dedicated protocols have been proposed lately to provide a subset of Internet nodes with specific services. The protocols achieve a logical subnet of nodes, although these are in fact physically connected to the global network.

[1] Email: giamp@dmi.unict.it
[2] Email: pistagna@dmi.unict.it
[3] Email: sriccobene@dmi.unict.it

1571-0661/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2004.11.046

The protocols should meet their goals without the assumption that all nodes be continuously up and running — nodes can be temporarily unavailable for any reasons such as local breakdown or security tampering. In short, local unavailability should not affect global availability of a service.

A typical service is file-sharing. Files are replicated over various nodes in order to minimize the chances of their unavailability due to unavailability of their nodes. A crucial property is that it must be possible to find a file if it is present on the network. Distributed backup is a similar service, except that the backup file must be accessible to its legitimate owner only. Security is a main concern for this service, *secret sharing* in particular — although it is necessary to share (pieces of) information among certain nodes, the information contents must not be accessible to unauthorized nodes. Replication remains the basic strategy to guarantee availability of the backup file, but maintains its intrinsic risks of leakage to unauthorized nodes or, even worse, of untraceable tampering. It may also lead to excessive redundancy.

The main requirement for the service to work is that all participating nodes be informed of each other's address. Maintaining a participants' list on each node can solve the problem only for small networks. There are two main approaches towards a scalable solution: one is *hierarchical* and the other is flat, namely *non-hierarchical*. A hierarchical setting, much in the style DNS services are organized, guarantees that all nodes can be reached if required, but has the drawback that, should a single node crash, all his descendants would be cut off from the network. By contrast, a non-hierarchical setting naturally tolerates local faults more easily because there is no dependency relation among the nodes. Each node only knows a certain number of neighbors. However, this cannot guarantee that a multicast communication will reach all intended recipients. Because an exhaustive search is necessary for both file-sharing and backup services to work, the hierarchical setting is typically preferred.

We introduce a new service for distributed backup over a non-hierarchical network. It implements techniques of threshold secret-sharing, using the IDA algorithm due to Rabin [9], to disperse the backup file among participants. To our knowledge, this is the first attempt of implementing threshold secret-sharing schemes for the sake of distributed backup. Because the underlying logical network is assumed non-hierarchical, a service that resembles Chord [11], but is simpler, is adopted to keep track of the participating nodes. Our approach has several advantages.

Optimized redundancy of data. The backup, which is often bulky, must no longer be replicated as a whole. The threshold secret-sharing algorithm logically fragments the backup and appropriately disperses the fragments among the participating nodes. Both dispersal and retrieval can be paral-

lelized in the style of I/O operations on RAID disks. For a fixed redundancy, the threshold secret-sharing algorithm guarantees a higher availability of the service than conventional, single-block replication would.

Efficient management of resources. Resource management is efficient in two extents. First, the basic messages necessary to keep the logical network alive are limited so that bandwidth is preserved. Second, the backup is uniformly dispersed over the logical network because all nodes are required equal resources in terms of disk space. More efficiency translates into lower risk of resource saturation for each node, which in turn increases the availability of the service.

Availability of service. Availability is high because only a threshold number of the participating nodes must collaborate to recompose the backup, and because the underlying logical network is non-hierarchical, hence more fault-tolerant. Also, the dispersal technique guarantees that the threshold number does not imply a specific subset of nodes, so that a specific node may not be vital to recompose the backup. As mentioned above, availability of the service is strictly connected to both redundancy of data and efficient management of resources.

Security of backup. There is an inherent form of secret sharing in the sense that no subset of colluding nodes below the given threshold number can recompose the backup. Each node sees a fragment of the backup. However, the node is neither aware of how many fragments are necessary to recompose the backup, nor of which nodes store the fragments. Therefore, even the right threshold number of nodes cannot obtain the backup on their own initiative if not informed by the owner of the backup.

This paper describes the formal aspects underlying our service for distributed backup over non-hierarchical networks. A proof-of-concept implementation is available but space constraints impose deferring its details to an upcoming paper. The next section (§2) briefly surveys existing efforts in the field of distributed backup without attempting to be exhaustive. It also describes in deeper detail the IDA algorithm and the Chord service, which are a starting point to our work. Then, the subsequent section (§3) details our distributed backup service. The final section (§4) concludes the treatment.

2 The building blocks

The literature features a number of services for distributed file-sharing, some of which have reached the implementation stage and are deployed at present. For example, the pioneer Napster [4] relies on a hierarchical network, while

the more recent Gnutella does not [7]. However, fewer attempts exist to take advantage of experience in distributed file-sharing towards the development of fully-fledged and scalable distributed backup services.

The pStore service [2] is an eminent example implementing distributed backup with support for versioning and secret sharing. Versioning is implemented throughout the *incremental* philosophy, much in the style of the popular CVS service for collaborative work, whereby the new backup versions are maintained by updating specific blocks of the initial backup. Secure sharing is implemented through a combination of cryptography and digital signature. On one hand, our current work is not concerned with versioning. On the other hand, it is important to remark that our approach to secret sharing is simpler thanks to the adoption of a threshold secret-sharing algorithm, which does not require digital signature and relative public-key infrastructure.

We adopt Rabin's IDA algorithm to disperse the backup file, and borrow elements from the Chord service to implement a simple location service. Short accounts for the two services are given in the sequel of this section, while an exhaustive treatment can be found elsewhere [9,11]. Using this combination, whose advantages were discussed above, for a practical service that reaches the stage of proof-of-concept implementation appears to be novel for the current literature, although the basic idea was foreseen by Anderson [1]. Our work is similar in spirit to the DBS service, which fragments the backup and adds a fixed number of *forward* error-correction bits to each fragment [10]. It is then possible to recover the backup from certain fragments. Usage of resources in terms of space is determined by the error-correction technique. By contrast, the IDA algorithm allows the user to choose a desired redundancy parameter, which can be chosen small or large at will depending on the specific space constraints. Similar comments apply to the OceanStore service. This uses *Cauchy Reed Solomon* codes instead [6], which appear to be more space-efficient than other forward error-correction codes.

2.1 IDA

In 1989, Rabin proposed IDA (*Information Dispersal Algorithm*) [9], an algorithm whose name explains the main goal. Information dispersal has had numerous applications [3,8]. We advocate the use of IDA to disperse a backup among the nodes of a logical network. The algorithm will prevent each node from accessing the contents of the received backup fragment, hence achieving the desired property of secret sharing. Moreover, it is possible to specify a threshold number of nodes that are required to collaborate to retrieve the backup. Collaboration of the totality of nodes is not necessary. In consequence, our use of IDA achieves threshold secret sharing of the backup.

The gist of the algorithm is simple. The input is first split up into a number of *blocks* of the same size (padding may be necessary). Each block is in turn split up into F *fragments* of size S. Clearly, $F * S$ is the size of each block. IDA disperses the F input fragments into N output fragments. Here, $N \geq F$ and $R = N/F$ is the desired redundancy factor. Each of the N output fragments is obtained as a linear combination of the F input fragments by means of a coding matrix of size $N \times F$. The coding matrix is such that any subset of F output fragments (hence, taken from the N output fragments) allows the reconstruction of the F input fragments by means of a decoding matrix. The decoding matrix is the inverse of a squared submatrix of the coding matrix. Any F colluding nodes cannot recover the backup because the coding matrix is kept from them, hence they cannot compute the decoding matrix. By contrast, the owner of the backup can, because he built the coding matrix. It may be convenient to use the same coding matrix for all blocks, although one may want to use more than one matrix to increase security. Further details exceed the scope of this paper, and may be found elsewhere [9].

To demonstrate the benefits of using IDA for secure distributed backup, we analyze the ratio redundancy versus availability in two different scenarios. We use IDA in the second scenario, not in the first. For example, let us assume the following parameters for both scenarios.

- The backup is 600MB.
- There are 60 nodes in the logical network.
- The desired redundancy factor is 4, hence the total redundancy is 2.4GB.
- For each node, the probability that the node is up and running is 0.2.
- The nodes may be up and running independently from each other.

Scenario 1: no IDA. Conventional, single-block replication is adopted. The entire backup is replicated 4 times, according to the redundancy factor, each time on a different node. The probability that the service is unavailable is the probability that all 4 nodes be down at the same time, that is 0.8^4. In consequence, the probability to obtain the service is $1 - 0.8^4$, that is 0.59. In short, a total redundancy of 2.4GB yields service availability of 0.59.

Scenario 2: IDA. IDA is adopted to disperse the backup. For example, the backup can be seen as a single block of 15 fragments of 40MB each. Because the redundancy factor is 4, the algorithm produces 60 output fragments of 40MB each, and disperses them appropriately. IDA guarantees that any collaborating 15 nodes can recompose the backup (each providing its fragment). The probability that the service is unavailable is the probability that at least 46 nodes be down at the same time, that is 0.8^{46}. In consequence,

the probability to obtain the service is at least $1 - 0.8^{46}$, that is 0.99. In short, a total redundancy of 2.4GB yields service availability of 0.99.

It is clear that the use of IDA substantially increases the percentage of service availability that can be obtained for a fixed redundancy. It may be less evident that availability in the first scenario is independent from the number of nodes in the logical network, while in the second scenario it is desirably influenced. For example, let us suppose that the number of available nodes doubles. Availability remains unvaried in the first scenario. In the second scenario, IDA can produce 120 fragments of 20MB and halve the storage capacity required to each node, so that any collaborating 30 nodes can recompose the backup. Service availability would go up to at least $1 - 0.8^{91}$.

2.2 Chord

Chord is a widely accepted location service for nodes of a logical network. The service, which is decentralized over a non-hierarchical network, is computationally efficient. Its main goal is to determine the node that manages a given key, but other operations are possible such as addition and departure of nodes, insert and update of keys. There is a distributed routing table whereby each node only has routing information about a small number of other nodes. That information is maintained in the *finger table* of the node. A variant of *consistent hashing* [5] is used to compute *identifiers* for either keys or (IP addresses of the) nodes, and to uniformly associate keys to nodes.

The association of keys to nodes is simple. Each key is stored in the first node whose identifier is either equal to or immediately follows the key in the identifier space. This is the *successor node* of the given key. If we sequentially draw all identifiers on a circle as in Figure 1(a), a number of arcs becomes visible. Every arc contains a subset of contiguous identifiers. If we look at the circle clockwise, each node manages the arc that terminates in the node. For example, node 6 manages resources 1 and 5.

When a node joins the logical network, its identifiers fall into some arc. Chord implements an *addition protocol* that halves that arc, and reassigns the first half to the newcomer as in Figure 1(b). For example, after node 3 joins the network, resource 1 is managed by node 3, no longer by node 6. On the contrary, when a node leaves, the *departure protocol* joins its two adjacent arcs. Figure 1(a) may be seen as the outcome of the departure of node 3.

When a node launches the *location protocol*, it relies on its finger table. The table has $log\, N$ entries, N being the highest identifier, that is the maximum number of resources that can be handled. It contains the addresses of nodes whose identifiers are exponentially distributed starting from the table owner's

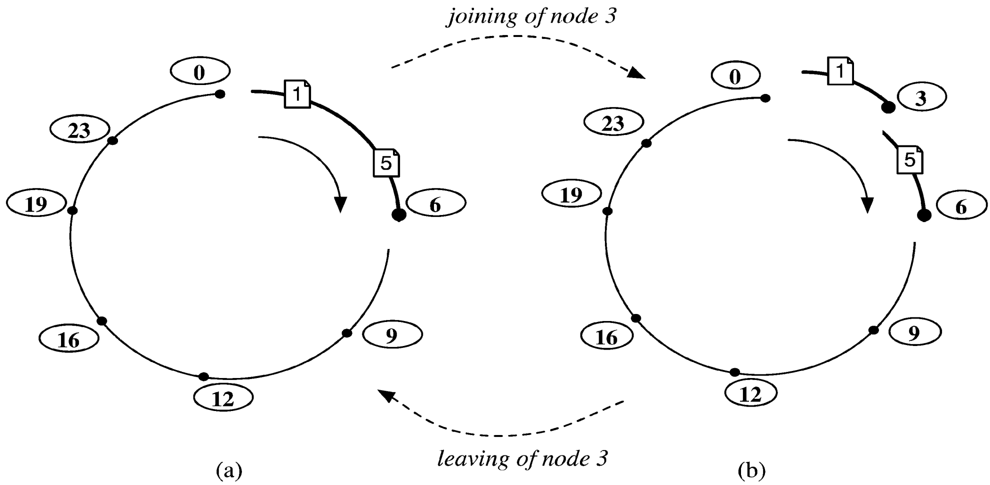

Fig. 1. A circular representation of identifiers

identifier. In consequence, $\mathcal{O}(log\, N)$ steps suffice to locate the required key identifier.

A node may fail without executing the leaving protocol mentioned above. Each node periodically checks whether its successor is up. When a node finds out that its successor is down, it executes a *recovery protocol.* This protocol updates all finger tables that addressed the failed node so as to replace that node by the first successor that is up. Each node replicates its resources over a number of its successors according to a desired replication factor. The resources managed by a failed node are not lost with a probability that is proportional to the replication factor. Rather, they are already available from a successor. However, the resources would be lost should all nodes where they are replicated be down.

Consistent hashing improves performance during the replication process because nodes that are physically adjacent will get identifiers that are close with each other. For example, this would be the case with nodes on the same subnet. However, adjacent nodes may not have independent probability of going down. For example, should the main switch of a subnet go down, the entire subnet would become unavailable with all its resources. In consequence, a considerable though localized failure would make a nearly contiguous set of identifiers unavailable, perhaps even an entire arc (see Figure 1). This may have drastic consequences as it may involve a larger number of identifiers than those determined by the replication factor.

3 The service

A personal backup service must be highly available and secure (confidential). In this context, the backup owner is strongly accountable for the properties of the service. In a distributed context, where the single user typically has no control over remote machines, accountability for the properties of the service essentially goes to the underlying protocols and their implementation.

The use of cryptography may help security but not availability because even an encrypted backup may be altered or cancelled by remote nodes. Remote tampering may either be deliberate, as is the case of malicious activity, or not, as is the case of unexpected crashes. The conventional strategy to increase availability is the replication of the backup across multiple nodes. However, this strategy fails to optimize redundancy of data and to manage resources efficiently.

As mentioned above (§1), a distributed backup service should conjugate availability and security on one side with an intelligent use of redundancy and of resource management on the other. This section presents the service we have developed to achieve this complex aim. Our service adopts the IDA dispersal algorithm [9] and borrows elements from the Chord location service [11].

3.1 Splitting the backup into blocks and fragments

The *splitting* protocol uses the IDA algorithm to split up the backup into blocks and fragments (§2.1). This preparatory work can take place off-line. Let F be the number of fragments for each block, and N be the number of output fragments that IDA produces from F input fragments. Once the size of the fragment is chosen, this number along with F determine the size of the block and the total number of blocks, which we call B. We have

$$B = \lceil sizeof(backup)/(F * sizeof(fragment)) \rceil$$

The parameters are chosen as follows — an example comes later. Parameter F is chosen such that the subsequent retrieval phase can find, with probability equal to the given percentage of service availability, any F fragments among the nodes that are alive. Parameter N is chosen of the same order as the size of the logical network. Another constraint is that the ratio $R = N/F$ must yield the desired redundancy factor.

Example. Let us suppose to have a backup file of 600MB, and require a backup service with availability percentage of 0.9 and redundancy factor no bigger than 12. We study the probability distribution function that at least some number of nodes be up, and observe that 0.9 is the probability that, say, 10 nodes are up. We remark that the probability that at least 10 nodes

are up is 0.9. We choose $F = 10$. For example, if the size of the logical network is 200, then we can choose $N = 120$ to satisfy the constraint on R. Should we choose a fragment size of 1MB, the block size would be 10MB and the total number of blocks would be 60.

3.2 Building the IDs from the fragments

Every backup fragment must be identifiable in the logical network. Before dispersing them, the *identification* protocol computes a unique ID for each fragment by hashing the contents of the fragment. The hash function can be public.

The backup owner must maintain a metadata file whose basic layout is demonstrated in Figure 2(a). The metadata file contains the obvious parameters: size of the backup, F, N, size of the fragment and coding matrix. It also contains a list associating each ID to the block it comes from and to the exact position of the ID within the block as it was determined by the IDA algorithm. Recall that B is the number of blocks in which the backup is split (§3.1). Because parameters F and N are available in the file, this can be compacted as in Figure 2(b), where the list of IDs implicitly expresses the exact scheduling of the output fragments. For example, ID_i is the hash of the fragment j of block l, where $j = i \; mod \; N$, and $l = ((i-j)/N) + 1$.

Whoever has access to the metadata file can recompose the backup. Keeping that file secure clearly is in the backup owner's interest. An attacker would not gain from seeing any number of IDs. There in fact exists no explicit relation among them thanks to the properties of the hash function.

3.3 Dispersing the IDs

At this stage, the N fragments that IDA generates are available. Our *dispersal* protocol disperses them over the network using their IDs. Each ID is logically divided into two parts. We call the most significant, leftmost part IDN and use it to address the nodes. Likewise, we call the least significant, rightmost part IDF and use it to address the fragments stored on a specific node. The length of the two portions with respect to the total length of an ID can be configured depending on the size of the logical network. However, once it is chosen, it must be kept constant. Let n be the number of bits in an IDN. Each node keeps a finger table mapping IDNs to IP addresses. Maintainance of the finger table is described in the sequel of this section. The node where the backup is first launched iterates the following procedure.

First, it attempts to map the IDN for the first fragment into an IP address. The finger table is built up incrementally, hence the required IP address may

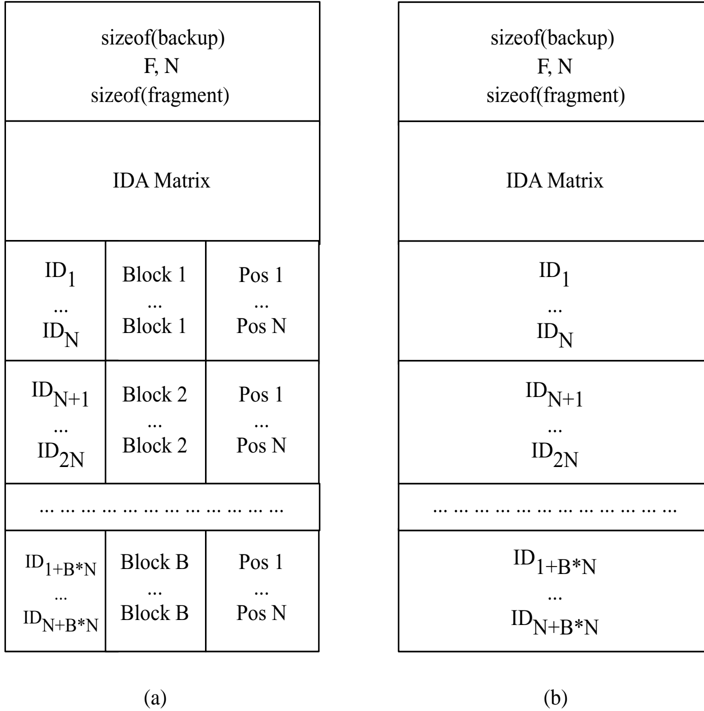

Fig. 2. Basic layout of a metadata file

yet be unavailable. A *delegation* protocol is invoked in this case, as explained below (§3.6).

If the mapping succeeds, the starting node attempts a connection to the node corresponding to the found IP. If this node is up, then the corresponding fragment is sent over. This will be fragment IDP for node IDN. We remark that this fragment is uniquely pinpointed by the pair IDN-IDP, which form the complete ID of the fragment. Otherwise, should the node be down, another candidate recipient would be computed by a suitable successor function $successor(IDN_i, j)$. A bound s is set on the number of successors to check. Among the s successors, the recipient is the first node that is up. Dispersal succeeds with the same probability as the probability that at least one of the s successors be up. Should all s successors be down, dispersal of the current fragment would fail.

The successor function serves as a public rule to obtain a set of possible substitutes for a given node, a simple implementation being

$$successor(IDN_i, j) = IDN_{(i+j) \, mod \, 2^n}$$

This is different from the corresponding function used by Chord, which maps both nodes and resources into nodes.

Our forward parameter s may seem to resemble Chord's replication parameter. However, the latter states the number of times the entire backup is copied forward, which blindly increases the total redundancy. Our forward parameter merely expresses the number of attempts to carry out in order to find a putative candidate where to store the fragment once.

3.4 Retrieving the IDs

When a node wants to recompose the backup, it executes the *retrieval* protocol. It starts off by reading the metadata file corresponding to that backup. For each block of the backup, F of the N dispersed fragments must be retrieved. The fragments are identified by means of their IDs. Retrieval must account for two possible scenarios. The IP address corresponding to a given IDN may be yet unavailable, in which case location carries on by delegation (§3.6). The other scenario sees the required node down. If so, at most s successors are checked up until the required resources are found by means of matching IDs. Otherwise, retrieval of the fragment fails.

Our algorithm for retrieval is not too complex and is outlined in Figure 3. It iterates the following steps until any F fragments are found. For each block, the starting node reads the IDNs from the metadata file and looks them up in its finger table. If F entries are found whose IP addresses are available and the corresponding nodes are up, then the algorithm terminates. Otherwise, it may either be case (a), that certain IPs are currently unavailable in the table, or case (b), that the corresponding nodes are down.

The algorithm continues by delegation in parallel on all IDNs whose matching IPs are unavailable, namely those of type (a). It may time out on certain IDNs. The outcome is threefold. First, delegation contributes to reaching fewer than F nodes, which may be up or down, so that retrieval fails. Second, delegation contributes to reaching a total of at least F IPs of nodes that are up. In this case, the F required fragments are found, and the algorithm terminates. Third, delegation contributes to reaching at least F IPs, although fewer than F nodes are up. Then, the algorithm is iterated on the next successor of the nodes that were down, namely those of type (b), for at most s times, s being our forward parameter.

There is an important difference between the first iteration and the subsequent ones. In the first iteration, finding F nodes up guarantees that the F required fragments are found. This may not be the case in the subsequent iterations even when F nodes are found up. One of them is the i-th successor of a node that is down, hence it may be the case that it does not store

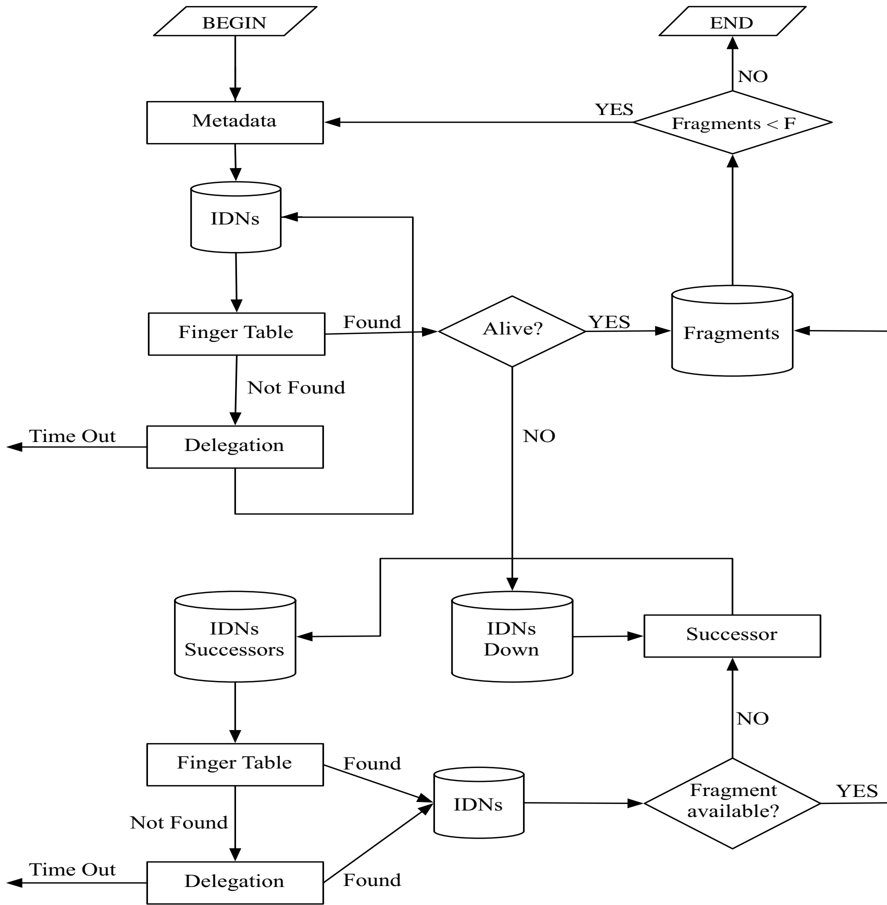

Fig. 3. Our algorithm for retrieving the backup fragments

the required fragment, which may be located in a subsequent successor (one between the $(i+1)$-th and the s-th).

3.5 Joining the network

If a node wants to join the logical network for the first time, it must execute the *joining* protocol. The node must know the address of at least one node who already is in the network, that is an entry point. The former contacts the latter and obtains a unique ID, which will serve to identify the newcomer in the logical network. We address this ID, whose length matches the established length n for an IDN, as $IDN_{\star}$. The newcomer initializes its own finger table as empty, and fills it up with n entries computing the IDNs as follows. If i

ranges in $1, \ldots, n$, then

$$IDN_i = (IDN_\star + 2^{i-1})\, mod\, 2^n$$

The equation describes the n IDNs that are exponentially ahead of $IDN_\star$. Then, it delegates the node that served as entry point the task of locating the IPs corresponding to these IDNs. At the same time, the entry-point node contacts the n nodes whose IDNs are exponentially distributed before $IDN_\star$. These are described by the following equation. If j ranges in $1, \ldots, n$, then

$$IDN_j = (IDN_\star - 2^{j-1})\, mod\, 2^n$$

Those nodes must update their respective finger tables by associating $IDN_\star$ to the IP of the newcomer. The newcomer is now perfectly integrated in the logical network, as it can both locate other nodes and be located.

At this stage, the node synchronizes with the current contents of the network. It seeks to retrieve fragments whose IDN matches its own, which are possibly stored in some of its successors. The node in turn contacts each one of its s successors.

3.6 Location by delegation

Each node maintains a finger table with at least n entries exponentially distributed ahead of the node IDN. When a node needs to locate the IP address for a target IDN, its first attempt is to index the target IDN in its finger table. If the target IP is yet unavailable, it launches a *delegation* protocol contacting the node whose IDN is immediately smaller and is already located, and passing on to it the target IDN. If this node is up and has got the target IDN already resolved, it passes the pair target IDN and target IP back to the caller. If all nodes contacted in n steps are up, then delegation is guaranteed to succeed, otherwise it may fail.

To increase performance, a node can extend its finger table with the IP addresses that it possibly learns through time. For example, it learns the source IPs of the received fragments and those of a delegation request. A finger table can contain at most 2^n entries.

3.7 Leaving the network

Our service does not need a *departure* protocol. A node may either decide to leave the network or fail unexpectedly. Unlike Chord for example, we do not need to discern the two scenarios. The retrieval protocol would tolerate the absence of the node from the network thanks to IDA's style of dispersing the fragments. Although its fragments are unavailable, the retrieval protocol will look for others. When a node leaves the network, Chord requires a specific

departure protocol that transfers its resources on to the node successor. In consequence, if the node fails unexpectedly without executing the departure protocol, the *recovery* protocol (§2.2) is mandatory to rearrange the successor of the node in the logical position of the node.

3.8 Rejoining the network

There is a *rejoining* protocol for nodes that were down and return up wishing to join the logical network once more. It is similar to the joining protocol (§3.5). A variant is that the node can reuse the IDN from the previous session so that it can make available the fragments that hopefully have been kept stored. Then, the node synchronizes with the current contents of the network, exactly as with the joining protocol.

As with the joining protocol, IDA makes our service tolerant to the cases in which the node loses its IDN or some of the stored fragments

4 Conclusions

We have presented the formal aspects underlying our service for distributed backup. Two are its main features. It is threshold-secure in the sense that it tolerates, to a certain extent, loss of backup fragments thanks to the adoption of the IDA dispersal algorithm, hence only a threshold number of fragments are necessary to recompose the backup. It works over non-hierarchical networks thanks to the development of a simplified version of the Chord location service. These appear to be two valuable features to optimize redundancy of data, manage resources efficiently, guarantee high availability of service, and provide a high level of security.

Tolerance towards fragment loss is a key feature, so far only appanage of services implementing error-correction techniques [6,10]. Our use of IDA shows how tolerance of fragment loss can now be tuned at will by the backup owner through the choice of the desired redundancy factor. Our service does not need to absolutely retrieve specific fragments. When a fragment is unavailable, the retrieval protocol can look for another one because it just is necessary to recover the threshold number of fragments, whichever they are. Also, a computationally efficient bound is set on the number of successors to check when a node is found to be down. Once the bound is reached, the retrieval protocol abandons and looks for another fragment. Our service has successfully reached the stage of proof-of-concept implementation (details omitted here due to space constraints). Testing on a small/medium scale network is expected soon.

References

[1] Anderson, R., *The eternity service* (1996), in Proceedings of Pragocrypt.

[2] Batten, C., K. Barr, A. Saraf and S. Treptin, *pStore: A secure peer-to-peer backup system* (2001).

[3] Curcio, I. D. D., A. Puliafito, S. Riccobene and L. Vita, *Design and Evaluation of a Multimedia Storage Server for Mixed Traffic*, Multimedia Systems **6** (1998), pp. 367–381.

[4] Fanning, S., http://www.napster.com.

[5] Karger, D., 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 ACM Symposium on Theory of Computing* (1997), pp. 654–663.

[6] Kubiatowicz, J., D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, *OceanStore: An Architecture for Global-scale Persistent Storage*, in: *Proc. of ACM ASPLOS* (2000).

[7] Lv, Q., P. Cao, E. Cohen, K. Li and S. Shenker, *Search and replication in unstructured peer-to-peer networks*, in: *Proc. of the 16th international conference on Supercomputing* (2002), pp. 84–95.

[8] Lyuu, Y.-D., "Information Dispersal and Parallel Computation," Cambridge University Press, 1993.

[9] Rabin, M. O., *Efficient dispersal of information for security, load balancing, and fault tolerance*, Journal of the ACM **36** (1989), pp. 335–348.

[10] Reeder, T., http://pages.cpsc.ucalgary.ca/~reeder/cpsc502/a1_proposal.html.

[11] Stoica, I., R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek and H. Balakrishnan, *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications*, in: *Proc. of the 2001 conference on applications, technologies, architectures, and protocols for computer communications* (2001).