HAL
open science

Exploiting Rateless Coding in Structured Overlays to Achieve Data Persistence

Heverson Ribeiro, Emmanuelle Anceaume

To cite this version:

Heverson Ribeiro, Emmanuelle Anceaume. Exploiting Rateless Coding in Structured Overlays to Achieve Data Persistence. 24th IEEE International Conference on Advanced Information Networking and Applications, AINA 2010, Apr 2010, Perth, Australia. pp.1165-1172. hal-00554701

HAL Id: hal-00554701

https://hal.science/hal-00554701v1

Submitted on 11 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Exploiting Rateless Coding in Structured Overlays to Achieve Data Persistence

H. B. Ribeiro
IRISA/INRIA Rennes Bretagne-Atlantique
France
heverson.ribeiro@irisa.fr

E. Anceaume
IRISA/CNRS UMR 6074
France
emmanuelle.anceaume@irisa.fr

***Abstract*—In this paper we evaluate the performance of DataCube a P2P persistent data storage platform. This platform exploits the properties of cluster-based peer-to-peer structured overlays together with a hybrid redundancy schema (a compound of light replication and rateless erasure coding) to guarantee durable access and integrity of data despite adversarial attacks. The triptych "availability - storage overhead - bandwidth usage" is evaluated, and results show that despite massive attacks and high churn, DataCube performs remarkably well. We evaluate the performance of the rateless erasure codes implemented in DataCube. Our exploration shows how parameters selection impacts codes performance mainly in terms of decoding time, and collect strategies.**

***Keywords*-Peer-to-peer, Replication, Persistence, Evaluation.**

I. Introduction

The huge interest for peer-to-peer systems has motivated researchers to go beyond routing and look-up functionalities. Over the last past five years, peer-to-peer systems have emerged as a viable architecture for implementing persistent data-storage systems. However, two challenges need to be addressed to successfully build efficient persistent data-storage systems. First overcoming the extremely heterogeneous nodes availabilities (essentially due to nodes intermittent or transient connectivity) [1] and second facing nodes untrustworthiness [2]. Regarding the first challenge, it is well known that storage overhead incurred by the storing of full data replicas and bandwidth needed to recreate new copies upon unpredictable join and leave of nodes tend to overwhelm the benefits of replication [1], i.e., its ease of implementation and its low download latency overhead [3], [4], [5]. In contrast, erasure coding provides redundancy without the overhead of replication. In particular, recently proposed rateless erasure codes (also called Fountain codes) [6], [7], [8], as a class of erasure codes provide natural resilience to losses, and make them fully adapted to dynamic systems. By being rateless, they give rise to the generation of random, and potentially unlimited number of uniquely coded symbols. A clear advantage of that property is that content reconciliation is useless and one may recover an initial object by collecting coded blocks generated by different sources. An increasing number of P2P persistent data-storage systems exploit erasure coding to provide data persistence, as Total Recall [9], Reperasure [10], and Carbonite [11], however facing nodes untrustworthiness is still challenging. OceanStore [12], [13] implements a hybrid redundancy scheme. By massively applying fixed-rate erasure coding to all data objects, OceanStore guarantees data persistence as long as undesirable (a.k.a. Byzantine) nodes do not collude together. Indeed, collusion of malicious nodes allow to easily mount targeted attacks that very quickly may disconnect targeted nodes from the rest of the system [14]. Actually holding a logarithmic number of IP addresses is sufficient to perform such an attack [15].

In a prior paper [16], we have presented a P2P persistent data-storage architecture. This architecture, called DataCube, guarantees durable access and integrity of data despite collusion of malicious nodes. This is achieved by combining a compound of full replication and rateless erasure coding schemes with the properties of cluster-based structured overlays. Briefly, each data-item is replicated at a small subset of nodes gathered together in a cluster, such that the set of clusters form the vertices of the structured graph. This replication schema guarantees that in presence of a bounded number of malicious nodes, data integrity is guaranteed through Byzantine agreement protocols, and efficient data retrieval is preserved (retrieval is achieved in $\mathcal{O}(logN)$ hops and requires $\mathcal{O}(logN)$ messages, with N the current number of nodes in the system). In addition to this replication schema, data is fragmented, coded and spread outside its original cluster. Each fragment is uniquely identified and is placed at the cluster that matches its new identifier. This coding schema guarantees that in presence of targeted attacks (i.e., the adversary manages to adaptively mount collusion against specific clusters of nodes), recovery of the data those clusters were in charge of is self-triggered.

In the present work, we evaluate the efficiency of DataCube. First, we evaluate the encoding and decoding performance of the rateless erasure codes implemented in DataCube. Our exploration seeks to understand which features and parameters lead to good coding performance. As will be shown, parameters selection impacts how well codes perform mainly in terms of recovering overhead, and decoding time. In particular we show that the order in which coded blocks are collected has a strong impact of the decoding time. Second, we perform an in-depth evaluation of DataCube in terms of storage and bandwidth overhead,

and data availability. This analysis is conducted by assuming an extreme adversarial context (some corners of the system are populated by up to 45% of malicious nodes, while the proportion of malicious nodes in the rest of the system may reach up to 30%), and a highly overloaded system (up to 1000 Tbytes are pushed in a system of 10^6 nodes, which roughly corresponds to the maintenance of a very large video archive). Results of this evaluation clearly demonstrate the benefit of hybrid replication over full replication in terms of data availability and storage overhead. It also demonstrate that according to nodes churn, the bandwidth usage per node is negligible in practice.

The remainder of this paper is structured as follows: Section I-A presents the principles of online coding. Section II describes DataCube main design principles. This section has been introduced for self-containment reasons. Section III presents the performance evaluation of the encoder and decoder implemented in DataCube. Section IV presents an analysis of data availability, storage overhead, and total bandwidth maintenance. Section V presents related works, and Section VI concludes.

A. Principles of Online Coding

Online codes [7] are based on two main system parameters ε, and q. Parameter ε, typically equal to 0.01, infers how many blocks are needed to recover the original message (i.e., a message of n blocks can with very probability be decoded from $(1+\varepsilon)n$ check blocks) while q affects the probability of reconstructing the original message (i.e., the decoding process may fail with negligeable probability $(\varepsilon/2)^{q+1}$, with q typically equal to 3 [7]). Online coding consists in three phases respectively called pre-coding, coding and decoding phases. Consider an original message (or data item) divided into n equal-sized input blocks.

The pre-coding phase consists in generating a small number $A = \delta\varepsilon qn$ of *auxiliary blocks*, with δ typically equal to .55, and by appending them to the original message. Specifically, for each original input block b_i we associate q randomly chosen numbers $i_1, \ldots, i_q$ with $i_j \in [1, \ldots, A]$ such that each auxiliary block a_{i_j} is computed by XOR-ing the content of all the input blocks we have associated it to. The A auxiliary blocks are then appended to the original n blocks message to form the so called *composite message* F' of size $n' = n(1+\delta\varepsilon q)$ which is suitable for coding.

The coding phase consists in generating *check blocks* c_i from the composite message F'. Specifically, check block c_i is generated by XOR-ing the content of d_i blocks of the composite message, with d_i a value sampled from a pre-specified probability distribution $\mathcal{F}$ that depends on ε [7]. The check block is then the pair $\langle c_i, x_i \rangle$ with x_i the set of the positions (also called adjacencies or neighbours) of the d_i blocks randomly chosen from F' to compute check block c_i. A possibly infinite number of independent check blocks can be generated this way. In DataCube a pseudo-random number generator function $\mathcal{G}(.)$ to sample d_i is used which allows different sources to generate exactly the same c_i (see Section II-C). Any set of $(1+\varepsilon)n'$ output checks blocks are sufficient to recover a fraction $1-\varepsilon/2$ of the composite message which guarantees to recover the original message with probability $1-(\varepsilon)^{q+1}$.

Decoding amounts in rebuilding the bipartite graph composed by all recovered blocks $\langle c_i, x_i \rangle$ and its adjacencies x_i. An *adjacent* block (also called *neighbour*) is a block in the set x_i XOR-ed to produce each *check block*. In the bipartite graph the decoding algorithm continously looks for received check blocks with only one unknown adjacent block. It recovers the adjacent composite block by XOR-ing the check blocks and all adjacents. Hence, check blocks with adjacence-degree 1 are direct copies of the corresponding composite block. At each round, any recovered composite block increases the probability of recovering other blocks through its edges. Input blocks are recovered from recovered composite blocks likewise.

II. Principles of DataCube

A. DataCube Design

DataCube implements a hybrid replication schema (a compound of light full replication and rateless erasure coding). This replication schema relies on a cluster-based DHT substrate. Cluster-based substrates (also called overlays) are specifically designed to be resistant to nodes dynamics by pushing the impact of churn at clusters levels so that the overall topology is barely impacted, and to efficiently tolerate malicious peers by running Byzantine-tolerant agreement protocols among clusters nodes. Specifically, cluster-based DHT overlays mainly consist in the clusterized version of DHT overlays, where groups of peers substitute peers at the vertices of the graph. These groups of peers, typically called swarms [17], [15], clusters [18], and cliques [19], and buckets [20] are populated by peers that are close to each other according to a given proximity metrics $\mathcal{R}$. This metrics can be logical (as in [17], [15], [18], [20]), or geographical (as in [19]). These clusters form the vertices of the structured topology. Clusters in the system are uniquely labelled, and their size is lower (resp. upper) bounded. The lower bound, named S_{min} in the following, usually satisfies some constraint based on the assumed failure model. For instance $S_{min} \geq 4$ allows Byzantine tolerant agreement protocols to be run among these S_{min} nodes. The upper bound, that we call S_{max} in the following, is typically in $\mathcal{O}(logN)$ to meet scalability requirements, with N the current number of nodes in the system. All cluster members or only a subset of them are in charge of routing lookup requests, replicating all data-items that match the cluster label, and handling cluster operations (split/merge and create). These features vary according to the specificities of the cluster-based overlays. DataCube design assumes that S_{min} peers are in charge of these operations. These peers are called

core members of a cluster. The other peers of the cluster (if any) are inactive until they replace left core members. These peers are called *spare members* of the cluster. This level of data replication is sufficient to guarantee durable access and integrity of any data that has been stored in the system provided that at any time and anywhere in the system a fraction of at most $\mu.n$ malicious nodes surround any subset of n non malicious peers, with $\mu < 1/3$. Now, to tolerate targeted attacks, DataCube exploits the properties of rateless erasure codes. Thus, in addition to being replicated at core members, data is coded and spread outside its cluster. The following two sections are dedicated to the description of both points.

B. Pushing Check Blocks in the System

Figure 1 shows the coding and spreading algorithm. Specifically, when core member $p \in \mathcal{C}$ receives data-item D, p generates a composite message (as explained in Section I-A) and its associated Merkle root [21] (see lines 1–6). Then p invokes a Byzantine tolerant consensus agreement among core members to agree on a unique composite message and Merkle root (line 7). The Merkle hash tree is an authentication scheme based on a tree of hashes that eliminates the large storage requirement by using a single signature (called *root* of the tree) for authenticating a finite number of messages. The consensus agreement eliminates the possibility of using a corrupted composite message during the coding phase. Finally, the Merkle root guarantees that only consistent composite blocks are used during the decoding phase. Once an agreement is achieved among core members, the coding phase is invoked by each core member.

In the Coding phase, c_0 check blocks are initially generated (lines 9–17), with $c_0 = (1+\varepsilon)n'$. Note that more check blocks can be generated afterwards (this occurs when the α spare members $s_1, \ldots, s_\alpha$ of cluster $\mathcal{C}'$ at which a specific check block is stored collude together to alter the integrity of that check block. In that case, core members of $\mathcal{C}'$ invoke the `codeBlock` function (lines 9–11)). Function `generateCheckBlock` implements the generation of each check block according to Section I-A. At round j, the adjacencies x_j of check block $\langle c_j, x_j \rangle$ are derived from $\mathcal{G}(key(D)+j)$, with $key(D)+j$ the seed of the pseudo-random number generator $\mathcal{G}(.)$. The rationale of using $\mathcal{G}(.)$ is that it guarantees that all core members generate exactly the same check block at each coding round without any synchronization among them. Each check block is assigned a key from which the placement on DataCube is defined. Keys must be random (to prevent malicious nodes from devising strategies to generate them), but their retrieval, for decoding, must not involve any storage overhead. Thus, DataCube exploits the hash-chain method [22] to identify check blocks. Each key assigned to a generated check block results from a recursive application of a hash function $\mathcal{H}$ on the data-item, establishing a chain (or stream) of keys. Specifically, given a data-item D and its associated *key(D)*=$\mathcal{H}(D)$, then key cB_n of check block $\langle c_n, x_n \rangle$ is equal to $\mathcal{H}^{(n)}(key(D))$, with $\mathcal{H}^{(n)}$(*key(D)*) the nth recursive application of the hash function $\mathcal{H}$ on *key(D)* (line 14).

Upon receipt `put`(D) **do**
1: *key(D)* ← *hash(D)*;
2: *cMsg[]* ← `preCode`(D);
3: **foreach** (composite block $j \in$ *cMsg*) **do**
4: *merkleLeafSet[j]* ← $\mathcal{H}$*(cMsg[j])*;
5: **enddo**;
6: $merkleRoot$ ← building of the Merkle tree on *merkleLeafSet[]*;
7: $\langle cMsg', merkleRoot' \rangle$ ← run consensus on (*cMsg, merkleRoot*) among core members;
8: invoke `codeBlock`($key(D), cMsg', merkleRoot', 1, c_0$);
enddo;

Upon invocation `codeBlock`(key,$cMsg$,$merkleRoot$,b,c_0) **do**
9: **if**($cMsg =$ `null` $\vee$ $merkleRoot =$ `null`) **then**
10: $cMsg$ ← key`.getAgreedcMsg()`;
11: $merkleRoot$ ← key`.getAgreedMerkleRoot()`;
12: **for** ($i = b$ **to** $b + c_0$) **do**
13: $\langle c_i, x_i \rangle$ ← `generateCheckBlock`($key, cMsg, \mathcal{G}(key+i), i$);
14: $cB_i \leftarrow \mathcal{H}^{(i)}(key)$;
15: `put`($cB_i, \langle c_i, x_i \rangle, key$) at α spare members of the closest cluster to cB_i;
16: **enddo**;
17: `register`($key, merkleRoot$) at neighbour clusters of $\mathcal{C}$ if not already done;
enddo;

Upon receipt `put`($cB_i, \langle c_i, x_i \rangle, key$) **do**
18: **if** (p`.spareView.length` $\geq \alpha$) **then**
19: α*List[]* ← `p.getClosestSpare`(α, cB_i);
20: **foreach** (spare member $i \in \alpha$*List[]*) **do**
21: p sends (STORE, ($cB_i, \langle c_i, x_i \rangle$)) to αList[i];
22: p`.spareView[`i`].addCheck`($cB_i, \mathcal{H}(\langle c_i, x_i \rangle), key$);
23: **enddo**;
24: **else** p broadcasts (STORE, ($cB_i, \langle c_i, x_i \rangle$)) to p's core set;
enddo;

Figure 1: Algorithm Run by any Core Member p

Each check block $\langle c_i, x_i \rangle$ is then pushed at $\alpha \geq 2$ spare members of cluster $\mathcal{C}_k$, with $\mathcal{C}_k$ the closest cluster to cB_i, with cB_i the key of $\langle c_i, x_i \rangle$ (line 18). These α spares are determined according to some arbitrary but deterministic function (e.g., the α spares are the closest spares to cB_i) (line 19). Replicating check blocks at α spares members increases the resilience to failures. The only case for which a new check block $\langle c_j, x_j \rangle$ has to be generated is when all α spare members simultaneously leave or collude. Note that when there are less than α spare members in the cluster, check blocks are temporarily stored at core members (line 25). Core members of $\mathcal{C}_k$ compute and store a fingerprint of $\langle c_i, x_i \rangle$ by applying a one-way hash function on it (line 22). This fingerprint is used by cluster $\mathcal{C}_k$ to guarantee the integrity of check blocks stored at its spare members. Finally, each core member $p \in \mathcal{C}$ registers the keys of all data-item D that are cached at $\mathcal{C}$ at the set of clusters $\mathcal{C}'$, such that $\mathcal{C}$ is a direct neighbour of $\mathcal{C}'$.

C. Selective Collect of Check Blocks

When a cluster $\mathcal{C}$ is detected corrupted recovery of the data cluster $\mathcal{C}$ was in charge of is triggered. Corruption detection is achieved through simple integrity tests performed by the neighbours of $\mathcal{C}$. Description of this procedure is out of the scope of the paper however the interested reader is invited to read the companion paper for more details [16]. In the following we assume that core members of cluster $\mathcal{C}'$ trigger the recovery of data-item D (i.e., cluster $\mathcal{C}'$ is a neighbour of $\mathcal{C}$). Cluster $\mathcal{C}'$ has to collect sufficiently many check blocks (at least $c_0 = n(1+\varepsilon)(1+\delta\varepsilon q)$) so that D recovery is guaranteed with probability $1-(\varepsilon)^{q+1}$ (see Section I-A). Specifically, let *key(D)* be the key corresponding to data-item D. From the hash-chain mechanism applied to *key(D)*, $\mathcal{C}'$ derives the keys of all the check blocks that are going to be collected, namely $cB_1 \dots cB_j$, with $cB_j = \mathcal{H}^{(j)}(key(D))$ and $j = (1+\varepsilon)n(1+\delta\varepsilon q)$, and from $\mathcal{G}(.)$ $\mathcal{C}'$ generates $x_1 \dots x_j$, the respective adjacencies of $cB_1 \dots cB_j$. From this initial step, $\mathcal{C}'$ has the opportunity to apply different strategies to collect check blocks, these strategies differing according to the priority given to the adjacencies degree. In the first strategy $\mathcal{C}'$ asks for $cB_1 \dots cB_j$ in any order, that is whatever their adjacencies degrees (this strategy is referred as to the *random* strategy in the following). In the second one, $\mathcal{C}'$ asks for check blocks such that the decoding process can get started upon receipt of the first check blocks. Specifically $\mathcal{C}'$ asks for all degree-one check blocks first and then, for the remaining check blocks, asks for them in any order. This policy is called *degree-one-first-random-afterwards* in the following. In the third one, $\mathcal{C}'$ asks for check blocks such that each one effectively contributes to recover the initial data. That is all degree-one check blocks $cB_{11} \dots cB_{1i}$ are asked for first, then check blocks whose degree are reduced to one thanks to $cB_{11} \dots cB_{1i}$ are asked for and so on until recovery of the initial data D is completed. This third policy is referred as to the *degree-one-only* strategy in the following.

III. Coding and Decoding Performance

In this section we evaluate the encoding and decoding performance of the rateless erasure code implemented in DataCube. Our exploration seeks to understand which features and parameters lead to good coding performance.

A. Setup Experiments

The experiments presented in this section have been performed in the following context: Different sizes of data files have been evaluated, ranging from small ones (i.e., 10 KBytes) to large ones (i.e., 1 MBytes) and for each data file, it has been fragmented in 100 blocks. Regarding the encoder parameters, different values of ϵ have been set, namely, $\epsilon = 0.01, 0.05, 0.1, 0.5$ and 0.9. Finally, the decoder has been fed with the *random*, the *degree-one-first-random-afterwards* and the *degree-one-only* strategies. All the plotted results are obtained from the averaging of 50 independent experiments.

Figure 2 exhibits the fraction of recovered input blocks as a function of the normalised number of collected check blocks (The normalised number of collected blocks is calculated as the number of collected check blocks divided by c_0). According to these experiments, for small values of ϵ (i.e., $\epsilon < 0.1$) the number of check blocks that need to be collected to successfully recover the original data files is around twice the value of c_0, although this number is less than or equal to c_0 for larger values of ϵ (i.e., $\epsilon \geq 0.1$). Actually, for large data files (i.e., 1 MBytes), the original data file can be recovered with only $0.8 \times \epsilon$ which tends to demonstrate that for large files, c_0 value is over estimated.

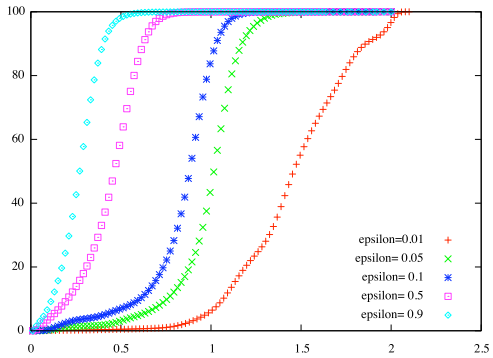
Figure 3 shows the decoding time as a function of the data files sizes for the different strategies described here above. The curves clearly demonstrate that the collect strategy is a critical part of the design. Indeed, strategy *degree-one-only* overpasses the two other ones whatever the size of the data file and for any values of ϵ. Thus, this collect strategy combined with the probability distribution $\mathcal{F}$ [7] fully exploits the properties of the coding by guaranteeing that i) many check blocks have a low degree so that they can be used for decoding the other checked blocks, and ii) these low-degree checked blocks are the first ones to be collected so that the decoding process progresses steadily, without requiring any useless XOR operations.

IV. Analysis of DataCube

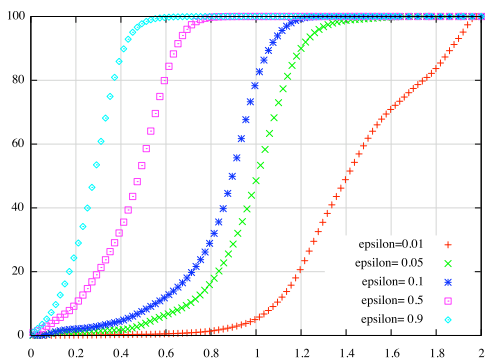
This section is devoted to the analysis of the data availability guaranteed by DataCube and the associated incurred storage and bandwidth usage requirements. Note that numerical values of the parameters used in the experiments are drawn from PeerCube features [18], in particular the derivation of the number of independent routes. Finally, differently from the above analysis, we assume a severe adversarial environment (at least 30% of the population is malicious).

A. Data Availability

In the following, we analyse the availability of data-item D. We assume that D sits at cluster $\mathcal{C}$, and each of the i generated check blocks have been spread on clusters $\mathcal{C}_i$, where $c_0 \leq i \leq c_b$, with c_b the number of generated check blocks needed to reach a given data availability (as derived later in this section). For simplicity reasons, we assume that each check block sits on a different cluster. By construction, D availability depends on both cluster C availability where the S_{min} replicas of D sit and on the availability of the clusters on which check blocks are located. Let μ denote the ratio of malicious nodes in the whole system. The probability p_p that cluster D is polluted is equal to the probability that its core set is polluted, that is, populated by more than $\lfloor (S_{min}-1)/3 \rfloor$ malicious nodes. In the following we consider the upper bound of p_p which is obtained when the

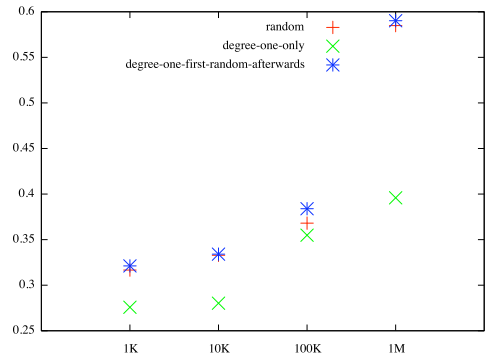


(a)

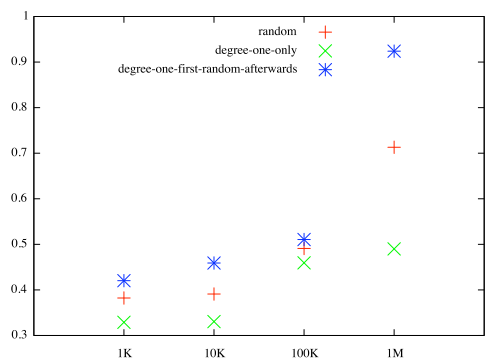


(b)

Figure 2: **Fraction of input blocks recovered as a function of the number of check blocks received over** c_0 for data file size equal to 100 KBytes (case (a)), and equal to 1 MBytes (case (b)).



(a)



(b)

Figure 3: **Decoding time (milliseconds) as a function of data files sizes (Bytes) for (a)** $\epsilon = 0.05$ **and (b)** $\epsilon = 0.9$**.**

number of clusters in the system is minimal, i.e. equal to $\lceil N/S_{max} \rceil$. Let Y denote the random variable representing the number of malicious nodes in a given core set, and X the one depicting their number in the cluster then

$$p_p = 1 - \sum_{y=0}^{\lfloor (S_{min}-1)/3 \rfloor} \sum_{x=0}^{\lfloor (\mu.N) \rfloor} \mathbb{P}\{Y = y | X = x\} \mathbb{P}\{X = x\}.$$

Probability $\mathbb{P}\{Y = y|X = x\}$ represents the probability that y malicious nodes are inserted in the core knowing that x are in the whole cluster, i.e.,

$$\mathbb{P}\{Y = y|X = x\} = \binom{x}{y}\binom{S_{max} - x}{S_{min} - y} / \binom{S_{max}}{S_{min}},$$

and

$$\mathbb{P}\{X = x\} = \binom{\mu N}{x} (S_{max}/N)^x (1 - S_{max}/N)^{\mu N - x}.$$

The lower bound p_h on the probability that a request issued from cluster $\mathcal{D}$ successfully reaches cluster $\mathcal{C}_i$ located at h hops from $\mathcal{C}$ and is successfully handled by $\mathcal{C}_i$ is equal to

$$p_h = 1 - (1 - (1 - p_p)^h)^r,$$

where r is the number of independent paths the request can take. It has been proven in [18] that $\log(N/S_{max}) \leq r \leq \log(N/S_{max}) + 3$ and $1 \leq h \leq \log(N/S_{max}) + 5$. We are

ready to derive the availability d_a of D.

$$d_a = (1 - p_p) + p_p \sum_{c_0}^{c_b} \binom{c_b}{c_0} (p_h)^{c_0} (1 - p_h)^{c_b - c_0}.$$

Table I provides a comparison between the stretch factor of our policy (i.e., the total amount redundancy added to data-items which is equal to c_b over c_0) and the replication factor imposed by classical full replication required to get data availability at least greater than 0.9, 0.99, and 0.999. To compute the replication factor obtained with classical full replication, we suppose that each copy of the data-item is replicated on a different cluster, as supposed for check blocks. Experiments are conducted for different sizes N of the system. Let $S_{max} = \lceil \log_2 N \rceil$. In all these experiments, we assume that h is maximal (i.e. we maximise the probability of encountering faulty clusters). For example for $N = 1,000$, we have $h = 11$ hops. Finally, we assume that 30% of the nodes in the system are malicious (e.g. for $N = 1,000$, $p_h = 0.076$). However, to simulate a targeted attack at cluster $\mathcal{C}$ (the cluster on which D sits), we suppose that $\mathcal{C}$ is populated by $\mu_{target} = 45\%$ of malicious nodes.

Results of these experiments, given in Table I, show the benefit of our approach over full replication. It is shown that for reaching different levels of availability, the required stretch factor increases relatively smoothly, while the growing of the replication factor is more sensitive. For instance, for $N = 1,000$, and $c_0 = 50$, the stretch factor is equal to $17,02$ for an availability of .99 and reaches a stretch factor of $18,92$ for an availability equal to 0.999%. Whereas in the same conditions, the replication factor increases from 51 to 80 which clearly becomes a problem for large data-items. The same trend is obtained for increasing values of N.

N	Stretch factor			Replication factor		
	0.9	0.99	0.999	0.9	0.99	0.999
1,000	14.42	17.02	18.92	22	51	80
2,000	17.92	21.14	23.52	27	63	100
3,000	17.96	21.18	23.56	27	64	100
4,000	22.24	26.26	29.2	34	79	124
5,000	22.28	26.3	29.24	34	79	125

Table I: **Comparaison of the stretch factor in DataCube and the replication factor obtained in classical full replication as a function of the required availability and the number of nodes N in the system**. In these experiments, the ratio of malicious nodes in $\mathcal{C}$ is equal to 45% while it is equal to 30% in the remaining of the system.

Figure 4(a) confirms the scalability of our approach. For all these experiments we have $c_0 = 50$, $\mu = 30$ and $\mu_{target} = 40\%$. For $N = 2^{16}$, rather than a replication factor of 194, we achieve the same availability with a 3-fold savings by relying on hybrid replication. Finally, one can notice the impressive benefit when using independent routes on both approaches. For instance, for $r = 6$ and $N = 2^8$, replication factor decreases to 33 while in our solution the stretch factor drops to 10.

B. Storage Overhead

We now compute the storage overhead implied by DataCube. We recall that each data-item D is initially replicated at S_{min} core members and n fragments generated from D are coded and spread to other clusters at α spare members. The storage overhead DS for D is given by:

$$DS = \lceil (n.S_{min} + c_0 \frac{c_b}{c_0} \alpha) \rceil - n. \quad (1)$$

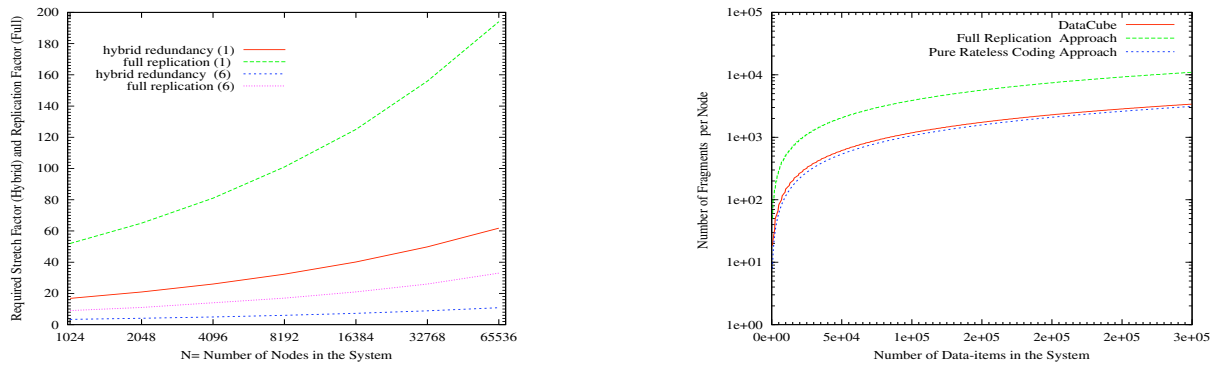
We now estimate how this storage overhead is distributed among nodes in DataCube. First, remark that data-items identifiers are randomly assigned for both data-items and check blocks. Thus, we can interpret the placement of both data-items and check blocks as the throwing of balls into several urns. We denote by Z_c (resp. Z_s) the random variable representing the total number of data-items (resp. check blocks) stored at each core (resp. spare) member. Let f be the total number of data-items in the system, c_b be the number of check blocks generated for each D to get a target data availability, and $N = N_c + N_s$ be the current number of nodes in DataCube—N_c (resp. N_s) is the number of core (resp. spare) members. The probability that the number of data-items (resp. check blocks) at any core (resp. spare) is upper bounded by z is given by:

$$P(Z_{s,c} \leq z) = \sum_{k=0}^{z} \binom{f.DS_{s,c}}{k} \left(\frac{1}{N_{s,c}}\right)^k \left(1 - \frac{1}{N_{s,c}}\right)^{f.DS_{s,c}-k},$$

where the notation $X_{s,c}$ stands for X_s when dealing with check blocks and X_c for data-items. In particular, $DS_s = c_0 \frac{c_b}{c_0} \alpha$ and $DS_c = n.S_{min}$. Figure 4(b) compares the number of fragments per node (core and spare member) in DataCube with the one needed in case of full replication and pure rateless erasure coding to guarantee an availability of 0.99. Recall that data-items are made of n fragments. In these experiments, $N = 1,000$, $c0 = 50$, the stretch factor is equal to 10 (see Figure 4(a)), and $\alpha = 2$. Lessons learnt from these experiments are two-fold: first, our approach guarantees a 3.5-fold savings wrt full replication, and second is very close to pure rateless erasure coding approach which clearly shows the low impact on storage overhead of the full data-items stored at S_{min} core members in DataCube.

C. Bandwidth Usage

We finally derive the total bandwidth needed per node for maintaining DataCube redundancy mechanism in presence of churn. Each time a node p leaves the system data-items or checked blocks p cached are copied over to new nodes (to keep the redundancy guarantees), while each time a new node p joins the system p needs to download all the data that match to it. If we assume that nodes join the system at rate λ and leave it at rate α, and that $\alpha = \lambda$ (to keep the system size constant in average), the usage bandwidth per core node is, in expectation, equal to twice the size of



(a) (b)

Figure 4: (a) **This graph shows the required stretch factor for our solution (hybrid redundancy) and replication factor for the full replication approach as a function of the number of nodes N in the system.** The required availability is $d_a = 0.99$. The number shown in brackets (i.e., 1 and 6) represents the number of redundant routes. (b) **This graph shows the number of stored fragments per node in DataCube, in a full replication approach, and pure rateless erasure coding one as a function of the size of the system N**. The replication factor used for the full replication approach is equal to 33 (see Figure 4(a)) while the stretch factor for pure coding is equal to 10.

fragments a node houses times λ. Note that the rate at which core members leave the system is $1/\frac{S_{min}}{log_2(N)-S_{min}}$ less than the one spare members do. Figure 5, derived from figures

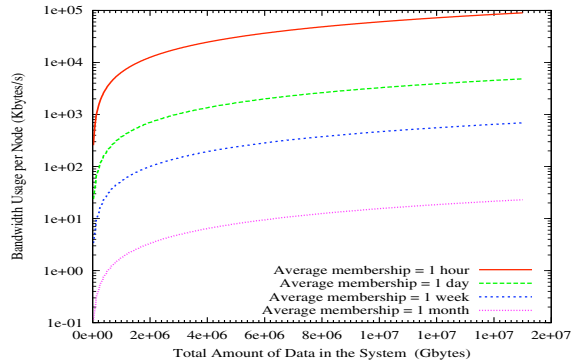


Figure 5: **This graph shows the required bandwidth per node as a function of the churn for maintaining up to** 1000 **TBytes of unique data** at an availability at least equal to 0.99.

obtained in Figure 4(b), shows the required bandwidth per node needed for maintaining up to 1000 TBytes of unique data in a system of $N = 10^6$ nodes (this corresponds to a very large video archive). Clearly at a daily turnover rate, the required redundancy policy is too demanding to be supported by nodes, however, at a monthly turnover rate, continuous contribution of each node shrinks to less than $20K$bytes/s which is clearly compatible with classic ADSL rates.

V. Related Work

Cataldi et al. [23] evaluate the encoding and decoding time of Raptor and LT codes as a function of the size of the original data by focusing on a failure free environment. In [24], the authors provide a comprehensive performance evaluation of open-source erasure coding libraries. Their main goal has been to evaluate the trade-off between data size and architectural features and memory behavior. Although the codes they evaluate are not rateless (and thus do not strictly apply to our study), their results show that codes performance highly depend on the characteristics of the machines. They show that 64-bit machines are well tailored for XOR operations, or that the size of the blocks should be carefully chosen according to the cache bahavior.

VI. Conclusion

In the present work, we have evaluated the efficiency of DataCube, a P2P persistent data storage platform guaranteeing durable access and integrity of data despite collusion of malicious nodes. The evaluation we have performed has shown that parameters selection impacts codes performance. In particular we have shown the benefit of judiciously collecting check blocks. Regarding our evaluation of DataCube in a severe adversarial environment we have shown the

benefit of hybrid replication over full replication in terms of data availability, storage overhead and bandwidth usage.

REFERENCES

[1] R. Bhagwan, S. Savage, and G. Voelker, "Replication strategies for highly available peer-to-peer storage," in *Proceedings of the International Workshop on Future Directions in Distributing Computing (FuDiCo)*, 2002.

[2] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," in *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[3] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher, "Adaptive replication in peer-to-peer systems," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2004.

[4] P. Knezevic, A. Wombacher, and T. Risse, "Dht-based self-adapting replication protocol for achieving high data availability," in *Proceedings of the International Conference on Signal-Image Technology and Internet-Based Systems (SITIS)*, 2006.

[5] X. Zhu, D. Zhang, W. Li, and K. Huang, "A prediction-based fair replication algorithm in structured P2P systems," in *Proceedings of the International Conference on Autonomic and Trusted Computing (ATC)*, 2007.

[6] M. Luby, "LT codes," in *Proceedings of the IEEE International Symposium on Foundations of Computer Science (SFCS)*, 2002.

[7] P. Maymounkov, "Online codes," *Research Report TR2002-833, New York University*, 2002.

[8] A. Shokrollahi, "Raptor codes," *IEEE/ACM Transactions on Networking*, pp. 2551–2567, 2006.

[9] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker, "Total Recall: System support for automated availability management," in *Proceedings of the USENIX Association Conference on Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[10] Z. Zhang and Q. Lian, "Reperasure: Replication protocol using erasure-code in peer-to-peer storage network," in *Proceedings of IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2002, pp. 330–339.

[11] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient replica maintenance for distributed storage systems," in *Proceedings of the USENIX Association Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells *et al.*, "OceanStore: an architecture for global-scale persistent storage," *ACM SIGARCH Computer Architecture*, pp. 190–201, 2000.

[13] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[14] B. Awerbuch and C. Scheideler, "Towards a scalable and robust DHT," in *Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2006.

[15] ——, "Group spreading; a protocol for provably secure distributed name service," in *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, 2004.

[16] H. B. Ribeiro and E. Anceaume, "DataCube: a P2P persistent storage architecture based on hybrid redundancy schema," in *Proc. of the IEEE Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, 2010.

[17] A. Fiat, J. Saia, and M. Young, "Making chord robust to byzantine attacks," in *Proceedings of the Annual European Symposium on Algorithms (ESA)*, 2005.

[18] E. Anceaume, F. Brasiliero, R. Ludinard, and A. Ravoaja, "Peercube: an hypercube-based p2p overlay robust against collusion and churn," in *Proceedings of the IEEE International Conference on Self Autonomous and Self Organising Systems (SASO)*, 2008.

[19] T. Locher, S. Schmid, and R. Wattenhofer, "equus: A provably robust and locality-aware peer-to-peer system," in *Proceedings of the International Conference on P2P systems (P2P)*, 2006.

[20] D. Malkhi, M. Naor, and D. Ratajzcak, "Viceroy: Scalable emulation of butterfly networks for distributed hash tables," in *Proceedings of the Annual Symposium on Principles of distributed computing (PODC)*, 2003.

[21] R. Merkle, "A digital signature based on a conventional encryption function," in *Proceedings of the International Cryptology Conference (CRYPTO)*, 1987.

[22] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, pp. 770–772, 1981.

[23] P. Cataldi, M. P. Shatarski, M. Grangetto, and E. Magli, "Implementation and performance evaluation of lt and raptor codes for multimedia applications," in *Proceedings of the International Conference on Intelligent Information and Multimedia Signal Processing (IIH-MSP)*, 2006.

[24] J. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2009.