DataCube: P2P Persistent Storage Architecture Based on Hybrid Redundancy Schema.

Heverson Borba Ribeiro, Emmanuelle Anceaume

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1936 – September 2009

DataCube: a P2P persistent Storage Architecture based on Hybrid Redundancy Schema

H. B. Ribeiro[*] , E. Anceaume[**]

Abstract: This paper presents the design of DataCube, a P2P data persistent platform. This platform exploits the properties of cluster-based peer-to-peer structured overlays altogether 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. In particular, the recovery of damaged data is achieved through the retrieval of the minimum number of coded blocks: coded blocks are selectively retrieved and their integrity is checked on the fly. An analysis validates DataCube principles design. Specifically, the triptych "availability - storage overhead - bandwidth usage" is evaluated, and results show that despite massive attacks and high churn, DataCube performs remarkably well.

Key-words: Peer-to-peer, availability, persistence, Byzantine failure model, collusion

---

DataCube: Une Architecture de stockage persistent

Résumé : *Ce rapport présente les principes de DataCube, une plate-forme de stockage de données. Cette plate-forme exploite à la fois les propriétés des réseaux logiques pair-à-pair structurés et un schéma de réplication hybride (une combinaison de réplication totale et de codage fontaine) permettant d'assurer un accès durable aux donnés et de vérifier leur intégrité, et ceci dans un environement hostile (présence de malicieux).*

Mots clés : *Systemes grande échelle, dynamisme, disponibilité, stockage persistent, collusion*

---

[*] IRISA/INRIA Rennes Bretagne-Atlantique, heverson.ribeiro@irisa.fr
[**] IRISA/CNRS UMR 6074, emmanuelle.anceaume@irisa.fr

1 Introduction

The huge interest for peer-to-peer systems has motivated researchers to go beyong routing and look-up functionalities. In this work we are interested in constructing a persistent storage functionality for implementing long-lived data objects despite massive and targeted attacks. Data redundancy is a required solution for implementing data persistence. However, in contrast to *classic* distributed systems in which the membership is stable, in peer-to-peer systems one has to face nodes churn (nodes propensity to frequently join and leave the system). To efficiently handle churn, recent works (e.g., eQuus [1], PeerCube [2] and, S-Chord [3]) have extended classic DHT-based overlays (e.g., Chord [4], CAN [5], Kademlia [6]) to their clusterized version. Clusters are populated by nodes that are close to each other according to a given proximity metrics (i.e., logical, geographical or semantical), and each of these clusters acts as a vertex of the structured topology. All the join and leave events are handle at clusters level which largely limit their impact on the overlay topology. In addition to being robust to churn, one can easily take advantage of these clusters to guarantee durable access to data (by using nodes as replica) as long as the coverage of nodes failures assumptions is close to one. Unfortunately a fundamental issue faced by any practical open systems is the inevitable presence of malicious (a.k.a Byzantine) nodes [7]. They can devise complex strategies to prevent honest nodes or users from retrieving data they locally cache (e.g. by mounting eclipse attacks [7], dropping, re-routing messages towards other malicious ones or providing incoherent responses). Existing P2P overlays that focus on tolerating the presence of Byzantine nodes (e.g., Peercube [2], Oceanstore [8], S-Chord [3]) make the assumption that no more than a bounded proportion of malicious nodes (i.e., a third of the population) are present at any time and anywhere in the system. This is a fundamental and required assumption to be able to design Byzantine tolerant algorithms [9]). However, differently from static distributed systems in which coverage of such an assumption is very high, in open systems, membership of the system evolves according to clients wishes. Thus even if the proportion of malicious nodes in these systems can be roughly estimated, there are some corners of the system that may be potentially surrounded by more malicious nodes than expected. This typically occurs through collusion of malicious nodes [10]. Such collusion allow to easily mount targeted attacks that quickly endanger the relevance of failures assumptions. Actually by holding a logarithmic number of IP addresses, an adversary can very easily and efficiently disconnect some target from the rest of the system. This can be achieved in a linear number of offline trials [11]. It has been shown by Awerbuch et al [10] that the only strategies that can protect open systems against such targeted attacks are the ones that preserve node identifiers randomness. However either these solutions keep the system in an hyper-activity (they force nodes to continuously leave and re-join the system) or they are intrinsically complex and costly [10].

The solution we propose to face this challenging issue is by designing a hybrid redundancy schema (a compound of light replication and rateless erasure coding) on top of a clusterized DHT-based overlay. This schema guarantees durable access and integrity of data despite adversarial attacks. Briefly, in addition to replication at a subset of clusters nodes, data is fragmented, coded and spread outside its cluster. Each fragment is uniquely identified and is placed at the cluster that matches its new identifier. When a cluster is detected corrupted (through simple integrity tests), this cluster is logically isolated from the system, and recovery of the data this cluster was in charge of is triggered. Coded fragments are collected, decoded, and the recovered data is eventually pushed to a new cluster that subsitutes the corrupted one. A secure verification hashing scheme is used to identify and verify each collected fragment on the fly. Analytical study of operations complexity, storage and bandwidth overhead, and data availability show the very good performance of our approach. Note that for the best of our knowledge this is the first work that addresses open-system data-persistence in presence of collusion. An analysis validates DataCube principles design. Specifically, data availability is evaluated in different adversarial contexts (i.e., 30% of malicious nodes in the system plus collusion at targeted clusters, while storage overhead, and bandwidth usage are derived for different amount of data (up to 200 TB). This makes DataCube an appealing data persistent platform for open and large scale systems.

The remainder of this paper is structured as follows: Section 2 presents related work on persistent structured peer-to-peer systems. Section 3 discusses persistence issues that one has to face in presence of massive attacks and the design we suggest to solve that issue. Section 4 presents an analysis of data availability, storage overhead, and total bandwidth maintenance. Section 5 concludes.

2 Related Work

Two main techniques have been proposed for handling data redundancy in large scale open systems: *full replication* (e.g., CAN [5], and PAST [12]) and *erasure coding* (e.g., Oceanstore [8]). Replication is based on creating copies of the original data object and placing these copies in distinct places in the system. The main benefits of using this scheme is clearly its ease of implementation and its very low download latency overhead [13, 14, 15]. On the other hand, storage overhead incurred by the storing of full data object replicas and bandwidth needed to recreate new copies upon unpredictable join and leave of nodes tends to overwhelm the benefits of replication. Erasure coding provides redundancy without the overhead of replication. Specifically, an object is divided into k equal size fragments, and recoded into s coded symbols, with

$s > k$. The ratio $\frac{k}{s}$ is called *code rate* and it gives the exact amount of redundancy added to the original data. Fundamental property of erasure coding is that the original data is recoverable from any k distinct coded blocks. Reed-Solomon codes are the pioneered fixed-rate erasure codes. They are well adapted to bounded-loss channels, and the recent introduction of Tornado codes guarantees fast coding and decoding operations (in linear time) however they are intrinsically not adapted to unbounded-loss channels (as generated coded blocks are interdependent). Rateless codes (also called Fountain codes) overcome this feature. As a class of erasure codes, they provide natural resilience to losses, and therefore are 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. The three main rateless codes proposed so far are the pioneering LT codes introduced by Luby [16], quickly followed by the online codes by Maymounkov et al. [17] and Raptor codes by Shokrollahi [18]. The two latest ones independently discovered the idea of adding a pre-coding phase to obtain linear codes and stronger recoverability guarantees. Section 3.2 details the main principles of online coding. An empirical study based on Kademlia [6] has shown that data redundancy management (full replication vs. erasure coding) strongly depends on both nodes availability and rate at which these changes take place. Actually, it has been shown [19] that erasure coding benefits vanish due to their implementation complexity and because of the increase in terms of download latency. The present work tends to demonstrate that by judiciously managing full replication and coding one can keep the best of both techniques.

From an operational point of view, an increasing number of peer-to-peers systems focus on implementing persistent objects. Total Recall [20] is designed to provide a probabilistic prediction of the number of replicas the system needs by evaluating the past behavior through a stochastic model. Differently from our approach Total Recall is not designed to handle a Byzantine-prone environment. In Reperasure [21] a fixed-rate erasure coding approach is performed to achieve redundancy. However, the main weakness of this work is the lack of churn handling which may prevent nodes to find all required fragments. The closest work to ours in terms of failure model is OceanStore [8, 22]. In Oceanstore a hybrid redundancy scheme is applied. A primary/secondary-tiers of full replicas is employed to serialise objects read/write operations. In addition, an archival form of objects based on fixed-rate erasure-coding is massively applied to any object. Differently from DataCube, Oceanstore hybrid scheme does not handle adversarial collusion, and because of the use of fixed-rate erasure codes, reliability of their data does not evolve with the number of replicated fragments.

3 DataCube Principles

3.1 Preliminaries

Prior to detailing our algorithm, we first present the two prerequisites any Byzantine tolerant cluster-based overlays have to provide to be directly exploitable by DataCube as a substrate: *(i)* each cluster must be uniquely labelled and, *(ii)* the size of each cluster must be lower (resp. upper) bounded. The lower bound, named S_{min} in the following, should satisfy $S_{min} \geq 4$ to allow Byzantine tolerant agreement protocols to be run among these S_{min} nodes. The upper bound, that we call S_{max}, should be in $\mathcal{O}(logN)$, where N is the current number of nodes in the system, to meet scalability requirements. Both prerequisites are already met (or easily achievable) in existing cluster-based overlays (e.g. , in S-Chord [3] labels can be easily assigned as a function of nodes position on the Chord ring).

Let us briefly describe how cluster-based overlays typically evolve according to these properties. When node n enters the system, it joins the cluster whose label matches the proximity metrics. Once a cluster size exceeds S_{max}, this cluster splits into two smallest clusters, each one populating with the nodes that are closer to each other. When n leaves, it simply leaves its cluster. When a cluster size reaches S_{min} this cluster merges with its closest neighbour. According to the proposed overlays, all cluster members or only a subset of them (but at least S_{min} 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). In the following we assume that only S_{min} nodes are in charge of these operations. We call these nodes *core members* of a cluster. The other nodes of the cluster (if any) are inactive until they replace left core members. We call these nodes *spare members* of the cluster. In PeerCube [2], spare members also allow to make join and leave events transparent to the overlay topology (this is simply achieved by having any node that joins a cluster to join it as a spare member).

In the following, when referring to an action taken by a cluster, we mean an action taken by all the core members of that cluster. However, for simplicity of the presentation, we abusively use the term "cluster" in place of the terms "the core members of the cluster".

3.2 Principles of Online Coding

Online codes [17] 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

$(1+\varepsilon)n$ coded 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 [17]). 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 XORing 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 XORing the content of d_i blocks of the composite message, with d_i a value sampled from a pre-specified probability distribution that depends on ε. The coded block is then the pair $\langle c_i, x_i\rangle$ with x_i the set of d_i blocks randomly chosen from F' to compute the check block c_i. A possibly infinite number of independent coded blocks can be generated this way. In this work we use a pseudo-random number generator function $\mathcal{G}(x)$ to sample d_i which allows different sources to generate exactly the same c_i (see Section 3.5). 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 *coded* block. In the bipartite graph the decoding algorithm continously looks for received coded blocks with only one unknown adjacent block. It recovers the adjacent composite block by XOR-ing the coded blocks and all adjacents. Hence, coded 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. At the same time input blocks are recovered from recovered composite blocks likewise.

3.3 Leveraging the Power of Clustering

We now detail how DataCube guarantees a durable access to data-items. As previously mentioned in Section 3.1, all core members of a cluster are responsible for the same data-items. Thus as long as less than a third of core members are malicious, replication at core members is provably sufficient to guarantee both their persistence and integrity through Byzantine-tolerant agreements executed by core members. In particular any lookup request for any data-item D in the system is successful in no more than $\mathcal{O}(N/log_2(N))$ hops, and in $\mathcal{O}(log_2(N)$ messages (where N is the current number of nodes in the system) [2, 3]. Now in presence of collusion, replication of data-items at core members does not guarantee anymore their persistence nor their integrity. Note that replicating data-items at all cluster members does not bring any additional guarantees. It will only take longer for the adversary to succeed in polluting the whole cluster. Thus the solution we propose is, in addition to keep full replica of data-items at core members, to fragment, identify, and spread coded fragments at other clusters of the system, each fragment being placed at the cluster that matches the fragment random identifier. Hence, when a cluster is detected corrupted (through simple integrity tests), this cluster is logically isolated from the system, and recovery of the data this cluster was in charge of is triggered. Coded fragments are collected, decoded, and the recovered data is eventually pushed to a new cluster that subsitutes the corrupted one. A secure verification hashing scheme is used to identify and verify each collected fragment on the fly. Thus if we consider data-item D such that D is originally placed on cluster $\mathcal{C}$ ($\mathcal{C}$ is the closest cluster to D key), then at cluster $\mathcal{C}$, each core member stores a full replica of data-item D while each spare member of $\mathcal{C}$ stores check blocks $\langle c_i, x_i\rangle$ of data-item D' such that $\mathcal{C}$'s label is closer to the key of $\langle c_i, x_i\rangle$.

Figure 1 shows the algorithm for the pre-coding and coding phases performed by any node $p \in V_c$, the spreading of check blocks on the closest clusters to their keys, as well as the replication of check blocks at these clusters. Specifically, upon receipt of data-item D, node p proceeds as follows. It generates a composite message (as explained above) and its associated Merkle root [23] (see lines 1–6), and runs 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, the coding phase is invoked by each core member, with $c_0 = (1+\varepsilon)n'$ (line 8) the number of check blocks to be created.

In the Coding phase, c_0 check blocks are initially generated (lines 9–17). Note that, more check blocks can be generated afterwards by invoking `CodedBlock` function (lines 9–11). This invocation is triggered when all α spares, at which a specific check block is stored, collude altogether to alter the integrity of the check block. Function `generateCodedBlock` implements the generation of each check block according to Section 3.2. To generate check blocks, core members use a PRNG function $\mathcal{G}(x)$ to select the degree of each check block, instead of using a random function, as described in [17]. Data-item key and the check block sequence form together the seed of $\mathcal{G}(x)$. The rationale of using $\mathcal{G}(x)$ is that it guarantees that all core

members generate exactly the same check block at each coding round without any synchronization among them. At round j, the degree x_j of check block $\langle c_j, x_j \rangle$ is derived from $\mathcal{G}(key(D) + j)$.

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 involved any storage overhead. Indeed, a straightforward solution that would consist in storing the key of each generated check block in order to lately retrieve its associated check block would be clearly unbearable. Thus, Datacube exploits a hash-chain [24] method to identify check blocks. Each key assigned to a generated check block results from a recursive hash function on the data-item, establishing a chain (or stream) of keys. Specifically, given a data-item D and its associated *key(D)=hash(D)*, then key cB_n of check block $\langle c_n, x_n \rangle$ is equal to $hash^{(n)}(key(D))$, with $hash^{(n)}(key(D))$ the n^{th} recursive application of the hash function on *key(D)* (line 14).

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

Upon invocation `CodeBlock`($key, cMsg, merkleRoot, b, c_0$) **do**
9: **if**($cMsg = \texttt{null} \vee merkleRoot = \texttt{null}$) **then**
10: $cMsg \leftarrow key$.`getAgreedcMsg()`;
11: $merkleRoot \leftarrow key$.`getAgreedMerkleRoot()`;
12: **for** ($i = b$ **to** $b + c_0$) **do**
13: $\langle c_i, x_i \rangle \leftarrow$ `generateCodedBlock`($key, cMsg, \mathcal{G}(key + i), i$);
14: $cB_i \leftarrow \texttt{hash}^i(key)$;
15: `put`($cB_i, \langle c_i, x_i \rangle, key$) at α nodes $\in V_s$ 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: $\alpha List[] \leftarrow$ `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,`hash`($\langle c_i, x_i \rangle$),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 performed by any node $p \in V_c$ of $\mathcal{C}$

Each check block $\langle c_i, x_i \rangle$ is then pushed at $\alpha \geq 2$ spare members (line 18) belonging to cluster $\mathcal{C}_{sec}$ (line 15). We use the name $\mathcal{C}_{sec}$ as a generic name for the closest cluster at which each check block is stored. 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 reduces the influence of spare members departure, avoiding accordingly the computation cost of creating a new $(Cb_j, \langle c_j, x_j \rangle)$ block each time a spare member leaves. Whenever some replica q among α leaves or become manipulated, another spare member substitutes q. Then, the only situation during which a new check block $(Cb_j, \langle c_j, x_j \rangle)$ has to be generated is when all α spare members simultaneously leave or collude. Note that, when $S_{max} = S_{min}$ (i.e., there are not enough spare members in the cluster) check blocks are temporarily stored at core members (line 25). Besides, by the time each check block is dispatched to α spares, core members of $\mathcal{C}_{sec}$ compute and store a fingerprint of $\langle c_j, x_j \rangle$ by applying an one-way hash function on it (line 22). Afterwards, this fingerprint is used by cluster $\mathcal{C}_{sec}$ to guarantee the integrity of check blocks stored at spare members.

Finally, to enable the recovery of data-items in case of corruption, each node $p \in V_c$ only needs to register the keys of all data-item D that are owned by p at the set of clusters $\mathcal{C}'$, such that $\mathcal{C}$ is a direct neighbour of $\mathcal{C}'$ (This is detailed in Section 3.5).

3.4 Detecting Attacked Clusters

The goal of the detection is to identify misbehaving clusters and tag these clusters as polluted to isolate them from the system. Practically, any cluster $\mathcal{C}$ that points to cluster $\mathcal{B}$ may verify $\mathcal{B}$ integrity upon receipt of a `lookup` request for some data-item D housed at $\mathcal{B}$. When cluster $\mathcal{C}$ receives the response of the request it can easily check D integrity by checking that D hashing is equal to D key. In case of inconsistency, cluster $\mathcal{C}$ contacts all neighbours of $\mathcal{B}$ to agree on whether $\mathcal{B}$ is corrupted or not. In the affirmative the recovery process is invoked by the closest cluster pointing to $\mathcal{B}$ (see Section 3.5). Note that if the number of $\mathcal{B}$ neighbours is not sufficient to invoke a Byzantine tolerant agreement, then $\mathcal{C}$ relies on the closest cluster $\mathcal{A}$ to $\mathcal{B}$ to decide on $\mathcal{B}$ corruption. Finally, in case $\mathcal{C}$ (or equivalently $\mathcal{A}$) is corrupted while $\mathcal{B}$ is not, then $\mathcal{B}$ is the only cluster that will consider $\mathcal{B}$ as attacked which is clearly not an issue since all the other neighbours of $\mathcal{B}$ will continue to send and receive data from $\mathcal{B}$ through other routes.

Note: the time interval that elapses from the time a cluster is attacked to the time the detection is effective may represent a threat to DataCube as the adversary can generate manipulated check blocks to prevent the original data-item from being recoverable. Datacube handles this issue by having spare members store different coded blocks under the same key. On doing so, any check block honestly created can never be masked by corrupted ones.

3.5 Recovering Attacked Clusters

Let $\mathcal{C}$ be the cluster that invokes the recovery of cluster $\mathcal{B}$, and let D be the data-item initially owned by $\mathcal{B}$. Then, any request r received by $\mathcal{C}$ that either *(i)* has $\mathcal{B}$ as an intermediate cluster on the path to its destination, or *(ii)* whose destination is $\mathcal{B}$ itself, is prevented from reaching cluster $\mathcal{B}$ as follows: in the former case, $\mathcal{C}$ forwards r to its recipient through another path, while in the latter case, $\mathcal{C}$ blocks request r, and takes in charge the recovery of D by aggregating sufficiently enough check blocks as now described. Cluster $\mathcal{C}$ uses D key (noted in the sequel, key(D)) to obtain the minimum number of check block keys cB_j, with $j \geq (1+\varepsilon)n'$. Cluster $\mathcal{C}$ searches for check blocks by combining both the deterministic function $\mathcal{G}(x)$ used to generate check blocks (see Section 3.3) and the hash-chain built during the coding phase. From $\mathcal{G}(key(D)+j)$ cluster $\mathcal{C}$ rebuilds the bipartite graph used to generate check blocks and thus descriminate any specific check blocks to be searched. From the hash-chain, cluster $\mathcal{C}$ derives the keys cB_j of coded blocks selected from $\mathcal{G}(x)$. To recover degree-1 check blocks, $\mathcal{C}$ first applies the deterministic function $\mathcal{G}(x)$ in order to identify these check blocks, i.e., any check block j, such that $\mathcal{G}(key(D)+j) = x_i$, with $i : 1 \ldots n'$ and $|x_i| = 1$. Then, keys of specific degree-1 check blocks are derived from the j^{th} position of the hash-chain selected with $\mathcal{G}(key(D)+j)$, such that $cB_j = hash^{(j)}(key(D))$. Fetching degree-1 check blocks before other check blocks makes the decoding of composite blocks faster. Once derived, cluster $\mathcal{C}$ sends a lookup request for check blocks identified by cB_j.

Upon receipt of a `lookup`(cB_j) request, core members ask for cB_j to each of the α spare members that house it or them (as previously noted). By using check blocks fingerprints (and possibly with the means of majority votes), core members cope with spare members collusion by verifying the integrity of the retrieved check block(s). Finally, the integrity of the recovered composite blocks is also verified by using the associated Merkle root saved at the neighbour clusters during the coding process (see algorithm 1 line 17). It guarantees that only non-corrupted check blocks are used to recover the original data-item. In case of integrity violation, core members of cluster $\mathcal{C}$ asks for other check blocks (by invoking the `CodeBlock` function). Once data-item is fully decoded core members of $\mathcal{C}$ send it back to the requesting node, and temporarily store data-item D. This completes the recovery process.

When a new cluster $\mathcal{N}$ is created (due to a `split`, `merge` or `create` operation) and becomes in charge of the data-items of cluster $\mathcal{B}$, recovered data-items temporarily stored in $\mathcal{C}$ are transfered to $\mathcal{N}$. Further requests for data-items previously owned by $\mathcal{B}$ (but not recovered by $\mathcal{C}$) are recovered by $\mathcal{N}$ by proceeding as above.

4 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 [2], in particular the derivation of the number of independent routes.

4.1 Data availability

In the following, we analysis 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 $c_0 \leq i \leq c_b$, c_0 is the minimum number of check blocks necessary to recover D, i.e., $c_0 = (1+\varepsilon)n'$, and c_b is 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 coded blocks are located. Let μ denote the ratio of malicious nodes in the whole system. The

probability 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 which is obtained when the 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

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

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

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

Now 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

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

where r is the number of independent paths the request can take. It has been proved [2], 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.

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

Table 1 provides a comparison between the stretch factor of our policy (i.e., the total amount redundancy added to data-items which is 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 coded 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). Thus, 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 1, show the benefit of our approach over full replication. It is shown that for reaching different level of availability, the stretch factor required 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 1: Computation 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 in the system N. 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 2 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 only 62 times the storage using hybrid replication, a 3-fold savings. 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.

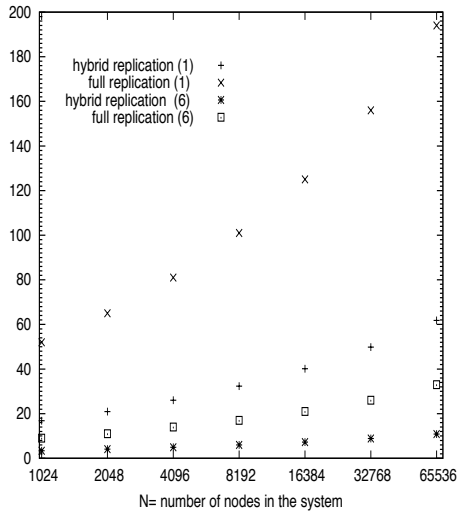

Figure 2: This graph shows the required stretch factor for our solution (hybrid redundancy) and replication factor for the full replication approach in function of N the number of nodes in the system. The required availability is $d_a = 0.99$. The number shown in brackets (i.e., 1 and 6) represents the number of redundant routes.

4.2 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:

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

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 3 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 2), 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.

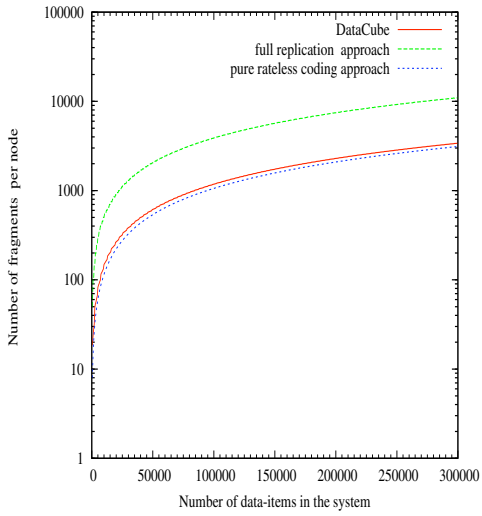

Figure 3: This graph shows the number of stored fragments per node in DataCube, in a full replication approach, and pure rateless erasure coding one. The replication factor used for the full replication approach is equal to 33 (see Figure 2) while the stretch factor for pure coding is equal to 10.

4.3 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 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 4, derived from figures obtained in Section 4.2, 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 support 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.

5 Conclusion

In this paper we have presented Datacube, a peer-to-peer persistent storage architecture which provides data persistence. We have shown that through lightweight full replication and fountain codes, durable access to data-items is achievable despite of a powerful adaptive adversary and high churn. As future work, we intend to study how reputation could help in estimating nodes trustworthiness.

References

[1] T. Locher, S. Schmid, and R. Wattenhofer, "equus: A provably robust and locality-aware peer-to-peer system," in *Proceedings of P2P*, 2006.

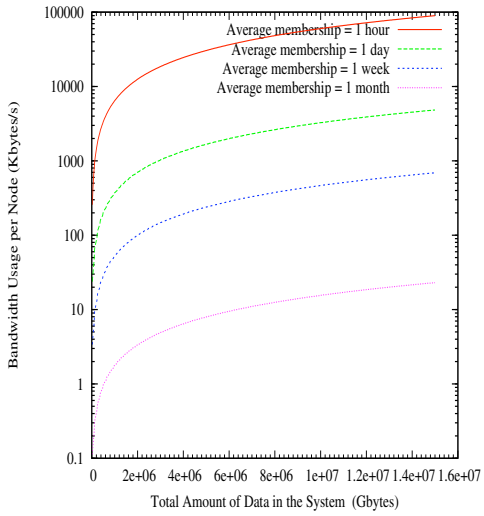

Figure 4: This graph shows the required bandwidth per node function of the churn for maintaining up to 1000 TBytes of unique data at an availability at least equal to 0.99

[2] E. Anceaume, F. Brasiliero, R. Ludinard, and A. Ravoaja, "Peercube: an hypercube-based p2p overlay robust against collusion and churn," in *Proc. of the Int'l Conerence on Self Autonomous and Self Organising Systems*. IEEE, 2008, full version in Technical Report IRISA #1888.

[3] A. Fiat, J. Saia, and M. Young, "Making chord robust to byzantine attacks," in *Proceedings of ESA*, 2005.

[4] I. Stoica, D. Liben-Nowell, R. Morris, D. Karger, F. Dabek, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of ACM SIGCOMM*, 2001.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proceedings of ACM SIGCOMM*, 2001.

[6] R. Bhagwan, S. Savage, and G. Voelker, "Understanding availability," *Lecture notes in computer science*, pp. 256–267, 2003.

[7] E. Sit and R. Morris, "Security considerations for peer-to-peer distributed hash tables," in *Proceedings of IPTPS*, 2002.

[8] 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.

[9] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, 1982.

[10] B. Awerbuch and C. Scheideler, "Towards a scalable and robust DHT," in *Proceedings of SPAA*, 2006.

[11] ——, "Group spreading; a protocol for provably secure distributed name service," in *Proceedings of ICALP*, 2004.

[12] P. Druschel and A. Rowstron, "Past: A large-scale, persistent peer-to-peer storage utility," in *Proceedings of HotOS*, 2001.

[13] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher, "Adaptive replication in peer-to-peer systems," in *Proceedings of ICDCS*, 2004.

[14] P. Knezevic, A. Wombacher, and T. Risse, "DHT-based Self-adapting Replication Protocol for Achieving High Data Availability," 2006.

[15] X. Zhu, D. Zhang, W. Li, and K. Huang, "A prediction-based fair replication algorithm in structured P2P systems," in *Proceedings of ATC*, 2007.

[16] M. Luby, "LT codes," in *Proceedings of SFCS*, 2002.

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

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

[19] R. Rodrigues and B. Liskov, "High Availability in DHTs: Erasure Coding vs. Replication," in *Proceedings of IPTPS*, 2005.

[20] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker, "Total Recall: System support for automated availability management," *ACM/USENIX*, pp. 337–350, 2004.

[21] Z. Zhang and Q. Lian, "Reperasure: Replication protocol using erasure-code in peer-to-peer storage network," in *Proceedings of SRDS*, 2002, pp. 330–339.

[22] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: The OceanStore prototype," in *Proceedings of USENIX FAST*, 2003.

[23] R. Merkle, "A certified digital signature," in *Proceedings of ICCAC*, 1989.

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