Availability-based methods for distributed storage systems

Anne-Marie Kermarrec, Erwan Le Merrer, Gilles Straub, Alexandre van Kempen

Availability-based methods for distributed storage systems

Anne-Marie Kermarrec
INRIA Bretagne Atlantique

Erwan Le Merrer
Technicolor

Gilles Straub
Technicolor

Alexandre van Kempen
Technicolor

Abstract

Distributed storage systems heavily rely on replication to ensure both data availability and durability. In networked systems subject to intermittent node unavailability, replicas need to be maintained, i.e replicated and/or relocated upon failures. Repairs typically, are extremely bandwidth-consuming and it has been shown that, without care, they may significantly congestion the system. In this paper, we propose an approach to replica management accounting for nodes heterogeneity with respect to availability. We show that by using the availability history of hosts, the performances of two important faces of distributed storage can be significantly improved namely *(i)* replica placement is achieved based on complementary nodes with respect to nodes availability, improving the overall data availability, and *(ii)* repairs can be scheduled according to node availability, so as to decrease the number of repairs while achieving comparable availability; this is achieved by an adaptive per-node timeout, instead of relying on a system-level timeout. We propose practical heuristics for those two issues. We evaluate our approach through extensive simulations based on real and well-known availability traces. Results clearly show the benefits of our approach with regards to the critical trade-off between availability, load-balancing and bandwidth consumption.

1 Introduction

Large scale peer to peer systems have proved to provide an appealing alternative in the context of many applications to centralized ones for they improve upon robustness and scalability. However their performance largely depend on the nodes availability. More specifically, nodes may join and leave the system at will without previous warning. Distributed storage is a typical example of applications for which peer to peer systems are natural candidates. Indeed, capacities of modern hard disks have outgrown the storage need of many users, leaving huge opportunities for this idle storage space to be leveraged in a collaborative storage system. In such a distributed storage system, each node is in charge of storing other nodes data in exchange of the guarantee to have its own data stored and available durably in time. Nevertheless if a node leaves unexpectedly the system, the data it is responsible for is also unavailable. To face this issue, distributed storage systems replicate data on more than one node so as to tolerate multiple nodes departures and/or failures. Once a replication degree is set, it has to be maintained across time despite nodes leaving the system, in order to guarantee the durability of data. This maintenance involves mainly two operations: *(i)* node failure or departure detection; *(ii)* replica repairs. In this paper, we focus on the latter repair operation which is highly bandwidth consuming. Interestingly enough, some well-know works [4, 25] reveal that one of the key aspect of an efficient distributed storage mechanism is precisely the bandwidth consumption.

In practical systems, distinguishing reliably a permanent failure from a transient one is challenging. Yet, significant bandwidth could be saved by generating repairs in the event of a permanent failure only, assuming that the data is only temporarily inaccessible in case of a transient unavailability. Therefore too aggressive a repair mechanism may congestion the system resulting from an excessive and useless bandwidth use. On the contrary, a low reactivity in the repair may result in hurting the quality of service, *i.e* data availability and durability.

In this paper we take up the challenge of tackling this trade-off answering significant questions closely related to the design of an efficient distributed storage system, namely:
-*Where should the replicas be placed?*
-*When should the repair mechanism be triggered*? (i.e. when the system should conclude on a transient or a permanent failure?)
We argue that a one size fits all approach is not sufficient and that the statistic knowledge of *every single node* availability, as opposed to system-wide parameters, provide a mean to efficiently tackle those questions. As opposed to mechanisms existing in peer to peer storage systems [3, 6, 13] the approaches proposed in this paper rely on per node availability statistics rather than network-wide average ones. In other words, while most previous works have assumed either that nodes are homogeneous or that simple averages on behaviors are representative, we account for the hetero-

geneity of nodes availability, both to decide where replicas should be placed and when repairs should be triggered.

Many systems address the placement problem (the *where* question) by replicating data on randomly chosen nodes [3, 15, 23, 7]. Some other place replicas on highly available nodes [18], at the risk of unbalancing the load on the system. Instead, we leverage availability habits of each node, that could be monitored by few dedicated servers or by the nodes themselves. We propose to use a targeted placement in order to put replicas on nodes so that the sum of their availability periods covers the whole prediction time, while preserving load balancing. Thus for the same replication level and for a comparable load on nodes, availability is significantly increased compared to random strategy.

The second question (the *when* one) boils down to a decision making process, where the goal is to detect permanent failures, to ensure durability of data while generating as few false positives (in terms of transient failures) as possible in order to save bandwidth. In distributed storage systems, this question is typically addressed by the use of timeouts. After a given timeout, a system parameter, a node is declared as failed and a repair is triggered. Most of the timeouts used in existing systems are defined at the system level, and thus are identical for all nodes regardless of their typical availability patterns [26, 30, 6, 4]. In addition, *advanced* timeout are often computed with Markov models using network-wide statistics as in [26, 30]. Unfortunately, some assumptions such as homogeneous behavior, or memoryless exponential distributions are not verified in certain systems [14, 5, 20]. Instead, in this paper, we propose to trigger a repair process by inference of node failure rather than conclusive knowledge. This suggests that one might analyse node failure detection in a standard inference framework, using false alarm probability. We then propose a per-node timeout determined upon its individual availability behavior. In addition this timeout is adaptive, as a function of the so called *criticality* of the situation (number of replicas effectively available in the system).

Contributions In this paper we propose a method accounting for node heterogeneity with respect to availability to decide *(i)* where to place replicas and and *(ii)* when to trigger a repair using a limited availability history of storage hosts. We make the following contributions.

The placement replica algorithm relies on a discovering complementary availability patterns (that we name *anti-correlated*) so as to maximize the availability of data across time. We show through extensive simulations based on real availability traces that this approach outperforms traditional random placement, resulting in a significantly lower number of replicas for a given availability. This includes facilities for erasure codes. For instance, on an availability trace of the Skype network, 5 replicas are needed instead of 8 with random placement, for an equivalent availability of around 99%.

The repair mechanism trades traditional system-level timeout based on network-wide statistics against a per node timeout, based on the node availability patterns. Nodes self-organize to compute their adapted timeout in a probabilistic way, according to their own behavior and to the current replication factor. Experimental results show that we achieve a lower number of repairs, while preserving the same level of availability of the data. For instance, on the same Skype trace, our method yields in 38% savings in repairs per-day, even compared to aggressive global timeout, while still improving availability of the data.

The combination of those methods sets the scene for efficient placement and repair strategies in large-scale distributed storage systems, improving the overall performance on realistic replication scenarios.

The rest of the paper is organized as follows: we first present related work in Section 2; this is followed by a detailed section on the motivations for this work in Section 3. Section 4 introduces the system model we consider. Our placement and repair strategies are then detailed in Sections 5 and 6 respectively. In Section 7, we use jointly those two techniques and discuss some practical questions, before concluding the paper in Section 8.

2 Related Work

Numerous distributed storage systems have been developed, be they fully distributed [23, 15, 7], or peer-assisted [27]. They address a wide range of issues such as various access control, data privacy, fairness, resilience to untrusted environment, or even propose complete file systems. As opposed to those works, we focus on core and well-identified issues in storage systems namely availability and durability of data. We propose an *availability-aware* plugin that could be applied to any distributed storage system and address these issues in a pragmatic way. Our approaches rely on accounting for node heterogeneity with respect to availability. The idea of using availability knowledge is not new and some systems like [3] already make predictions based on node availability to adjust redundancy mechanism and repair policies; the difference between those availability-aware systems and our approach is that they use availability to compute averages on global trends for nodes behavior, while we make use of finer grain information at the node level, namely availability patterns

themselves.

Also many host availability studies exist in the literature, independently of their application to storage systems [14, 8, 2]. Yet, all those approaches rely on statistical analysis of the nodes behavior in order to extract regular patterns in the system traces. Most of them agree on a certain level of predictability, a diurnal pattern is for example very common [16, 18, 5, 2]. J.Douceur even wonders if host availability is *governed by a universal law* [8]. Node availability in those systems is thus extracted from real traces and averaged so as to define typical node availability patterns.

Interestingly, location [14] has very recently made available to the community an online public repository of availability traces taken from diverse parallel and distributed system. In order to remain at a practical level as much as possible, we use these public traces to validate the efficiency of the methods proposed in this paper.

2.1 Availability-aware replica placement

Studies on availability have for example motivated the idea of biasing the replica placement strategy, instead of a random one. In [18], the author propose to store replicas towards highly available nodes. The work in [21] suggests to offer a data availability proportional to the node stability.

In paper [16], replica placement is biased towards nodes that have similar availability patterns, typically available at the same time. Very recently, paper [24] analyses replica placement strategies which optimize availability, "patching" the time by selecting highly available nodes when some time slots are not covered by at least one replica. We share the availability maximization aim with works [18, 21, 24]. Yet, since those approaches tend to bias the replica placement towards highly-available nodes, most of the storage load is concentrated on such nodes. Instead, our availability-aware replica placement leverages all nodes, balancing the load evenly in the system by design, as confirmed by our experimental results.

2.2 Failures and repairs

Regardless of availability, a storage system has to ensure data durability. In a distributed systems, this implies that data has to be replicated and repaired when failures occur. For example [7] uses *eager repair*, *i.e.* the system immediately repairs the loss of data as soon as a host failure is detected. However, the repair process is extremely bandwidth consuming. It has been shown that bandwidth one of the most important factor in designing a distributed storage system [4]. In order to minimize the impact of the repair process, different techniques have been proposed. In [3], repairs are deliberately delayed in an attempt to mask transient failures. In [13], the authors suggest using a high replication level in order not to generate a repair in case of failure, providing data durability even if correlated failures occur; nevertheless, the counterpart is that the price in terms of generated storage overhead is very high. In [9], availability knowledge is used to adjust the repair rate of the system as a whole, not a the node level.

The designers of Carbonite [6] were the first to place reintegration (ie. reintegrate replicas which have been wrongfully considered as failed) as a key of conceiving a functional distributed storage system. Very recently, some authors have proposed advanced timeout design in order to deal with distinction between transient and permanent failures. The authors in [22] analyse the use of timeout in a stochastic model and compare solutions with reintegration against standard ones. The approach of [26] is the closest to ours. However their proposed algorithm mainly focuses on the accuracy of instantaneous detection but does not consider the impact of the false positive or false negative errors on average data availability and replication cost. The authors of [30] are the first to model node behavior by a continuous semi-Markov chain, which does not require the use of exponential distribution. However, they always assume a scalar value as availability measure, and an homogeneous behavior for all hosts in the system.

3 Motivations

In this section, we expose the three main motivations to availability-aware replica placement and repair in distributed storage systems, namely: *(i)* leveraging node predictability with respect to availability, *(ii)* accounting for nodes heterogeneous availability patterns, and *(iii)* improving the trade-off between replication rate, availability and bandwidth.

3.1 Availability and Predictability

In the context of distributed storage systems, where nodes hosting data may leave and join at will, many systems have been originally proposed that do not explicitly deal with specific availabilities of nodes [23, 7]. In the quest for better performances, some works have studied and identified trends in the availability of the hosting resources [8, 2, 16, 5]. The important question of leveraging predictability has then been addressed [18, 16] to effectively handle transient failures.

We propose in this paper to build on this predictability in the behavior of nodes to provide adapted heuristics that overcome basic availability agnostic replica

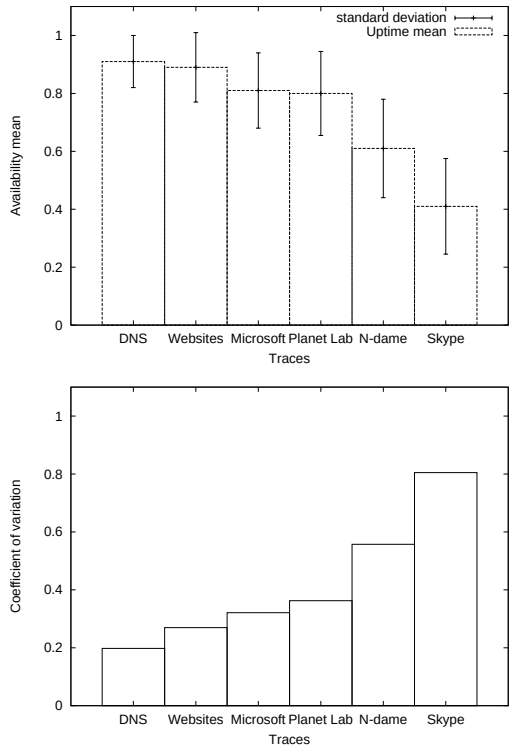

Figure 1: Availability mean and dispersion for various existing systems

placement and repairs; this also improves systems where the availability notion is used at a coarse grain.

3.2 Heterogeneous availability patterns

Most of systems that take availability of resources into account do so in a basic way. The general approach is to leverage one single parameter, the mean or the distribution of availabilities of all nodes that have participated in the system [30, 26, 3]. While this is convenient to apply theoretical models as the Markov model or basic probabilities for the number of replicas to create, or the expected lifetime of the system, recent comments underline the limited applicability of such models in practice [11, 10].

While some storage systems use platforms like home gateways [28] that have a homogeneous and high availability, the majority of deployment platforms exhibit a non-neglectable heterogeneity in practice. To illustrate our claim, we plot on Figure 1 the mean, standard deviation, as well as the dispersion of availabilities of hosts composing systems like Microsoft desktop computers, Skype or PlanetLab[1].

The figure clearly shows a significant variance among nodes behavior. This is confirmed in a recent storage system analysis [27]. Even the DNS system has a dispersion of uptimes of around 20%. This trend is even more striking for the two leftmost systems. This demonstrates that availability cannot be accurately expressed by a basic scalar mean trying to represent the overall trend of the fraction of time hosts are up. Furthermore, reducing availability to a mean or a distribution [20, 27] ignores information about hosts availability patterns while such information could be leveraged to increase reliability of distributed storage [18, 24, 16]. This calls for a finer grain study and accounting for specific peers availability.

3.3 Storage System trade-offs

Blake et al. [4] have shown that for dynamic storage systems using redundancy, a severe bottleneck appears in bandwidth, when maintaining a high availability for the stored data. To propose realistic system designs, recent works have thus relaxed the 6 Nines constraint on data availability, taking into account more pragmatic replication rate.

Another trade-off, allowing a classification of distributed storage systems, is presented on Figure 2. The third motivation for our work is precisely the critical trade-off between replication rate, load balancing and bandwidth consumed by the system. A distributed storage system targeting data availability may be represented by a set of three strategies: *(i)* a redundancy strategy characterized by k (k may be either a replication factor or the rate of an erasure code), *(ii)* a placement strategy (where to replicate the data) and *(iii)* a repair strategy (when and how to repair a lost replica). In this formalism, "basic" DHT-based systems as [7, 23] reach high availability due to very high replication rates (for instance PAST stores a chunck of replica on its *leafset* in the DHT), while load balancing is ensured by a pseudo random placement strategy due to the PUT operation on the DHT (the hash function balances evenly the distribution on the system). Furthermore, as pointed out in paper [17], replicas have to be relocated each time a node is inserted in or leave the replication neighborhood of a file, triggering cumbersome maintenance operations.

Solutions that are availability-aware suggest to bias the placement of data towards highly available nodes [18, 21], thus minimizing bandwidth due to repair cost caused by transient nodes. Nevertheless, this creates a high pressure on stable nodes that are asked to contribute significantly more than average nodes.

Relaxing one of these three constraints can limit the complexity of a proposed system, yet cause a hardly usable solution in practice. Finally, very few storage systems such as Total Recall [3] or Carbonite [6] ad-

[1]Those availability traces, from scientific publications, are made available in a repository [1]

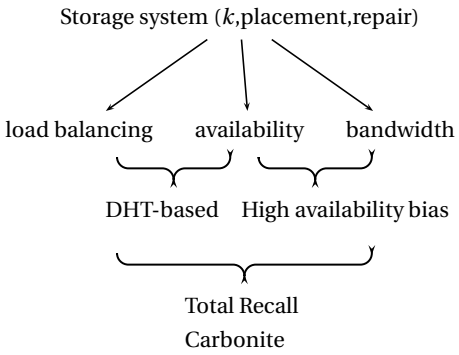

Figure 2: A classification of distributed storage systems.

dress this trade-off by using random placement and non-trivial repairs in order to maximize availability. A motivation for our work is precisely to improve the latter and more challenging class of systems, by proposing availability-aware methods that are applicable as "plug-ins" to their architecture.

4 Storage system model

Storage systems that leverage available disk space on hosts connected through the network range from peer to peer applications on users computers [13] to hardware boxes like Internet providers gateways [28]. Our availability-aware solution is specifically designed to be applied to any type of network where hosts may exhibit temporary and recurrent, periods of unavailability. Systems with a near-perfect availability of their components obviously have no need for such a study. We explain later on that our methods does not degrade performances for replication on stable or unpredictable networks.

For the sake of clarity and because we focus on the replica and repair strategies, this approach is illustrated as a peer-assisted method. We assume the existence of a service providing requesting nodes with a set of accurate partners. Those partners are then used by the requesting nodes to place replicas or repair replicas; replication *clusters* are then created for each data to store, following for instance the replication in DHTs leafsets where groups monitoring and storing the same data are created [23, 7]. While such a service is trivial to implement using a server, it is directly applicable to pure distributed systems, where hosts can collaborate to distributively achieve this computation, through gossip for example [16, 24]; each node in a cluster then monitors other cluster nodes to detect failures.

In order to exploit information on hosts availability, our system requires to keep track of a limited history of those availability and unavailability periods. In practice, such availability vectors can be maintained by a central server, the node itself providing it on demand, or by a distributed monitoring system if nodes cannot be trusted to self-monitor [19].

Concerning explicit rewarding of stability, we are looking to improve data availability at the network level. Therefore we do not focus explicitly on free-riders in our approach. This is out of the scope of the paper. This design choice is made in order to keep applicability to a wide range of distributed storage systems, including those with no direct possible intervention of a user on the storage device availability.

Finally, as our aim is a pragmatic study of what can be achieved beside purely theoretical models for placement or timeouts, we use as a basis publicly available traces, that have been deeply studied in their respective original papers [1]. Those traces are from systems of a great variety; when applying techniques on them, the goal is to underline tendencies for the associated kind of availability they exhibit, more than just proposing a specific improvement for a narrow range of systems.

5 Availability-aware placement strategy

In this section, we propose an answer to the question: *Where should replicas be placed so as to maximize availability while ensuring an evenly balanced load?*. The availability-aware placement strategy proposed relies on leveraging the monitored availability of nodes.

5.1 The R&A placement strategy

The aim of the method proposed below, as presented in Section 3, is to offer a better data availability than a random strategy placement using the same storage overhead k and without getting a high skewed load distribution. Thus this excludes biased placement towards highly available nodes, which by definition only considers specific nodes for the whole system satisfaction. To this end, we leverage the availability knowledge of each node in order to choose other nodes to place the replicas. Those nodes are chosen so that their availability patterns match the unavailability periods of the requesting node. Such nodes are denoted as *anti-correlated* nodes as we explain below.

This *availability history* of a node is represented as a vector of a predefined size (acting as a sliding window of time). For each time unit, the corresponding vector entry is set to 1 if the particular node was online at that time, and -1 otherwise. In this paper, we assume history vectors of one week, with one hour as unit time. This length has been shown to capture accurately most of user behaviors (*e.g.* diurnal, and week end presence patterns) [2, 18, 16, 8]. Let $\overrightarrow{A_x}$ be the availability vector of node x.

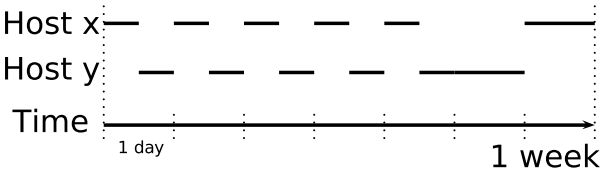

Figure 3: Perfect anti-correlation between hosts x and y.

We define the notion of availability *anti-correlation*, for a requesting node x as the opposed presence of a node y on the same predefined period (thus minimizing overlapping periods). Figure 3 illustrates this notion: y is perfectly anti-correlated to x, as it is online during all periods of unavailability of node x. Comparing nodes vectors of availability history, nodes can be sorted by their effective correlation to a given node x.

In practice, the (anti)correlation of two nodes is modelled as the angle, noted $\Theta_{x,y}$, between the two vectors $\overrightarrow{A_x}$ and $\overrightarrow{A_y}$ of nodes x and y:

$$\Theta_{x,y} = \arccos\left(\frac{\overrightarrow{A_x}\cdot\overrightarrow{A_y}}{\|\overrightarrow{A_x}\|\cdot\|\overrightarrow{A_y}\|}\right)$$

- $\Theta_{x,y} = 0$: Perfect correlation between nodes x & y
- $\Theta_{x,y} = \pi/2$: No correlation between x & y
- $\Theta_{x,y} = \pi$: Perfect anti-correlation between x & y

For example, two nodes available only at daytime and sitting in opposite time zone would have an anti-correlated behavior at the system level. The objective of using -1 instead of the classical 0 value to express the unavailability of a node is to not only capture the correlation, but also the anti-correlation between two behaviors. In fact the knowledge of unavailability is as interesting as the one about availability.

Our placement strategy relies on building clusters (recall that a cluster is the set of nodes holding replicas of the same data) with *pairs* of nodes exhibiting anti-correlated behavior so as to cover the whole prediction period. Since our goal is to increase the global availability, an anti-correlated host to a *reference host* is the best candidate to patch the time when this *reference host* is offline, as illustrated on Figure 3. By picking a node, finding its best anti-correlated counterpart, and iterating on this process until the cluster contains enough nodes to replicate k times a data, the effect of time patching is increased. We call this scheme R&A, for Random and Anti-correlated placement scheme.

In order for our placement method to overcome a random strategy (where the cluster is built up with nodes chosen uniformly at random), part of the system nodes must have some predictability on their availability behavior. The more predictable the nodes in the system, the more performance may increase when compared to availability-agnostic placement. On the contrary if all nodes in the system show absolutely no predictability then the R&A scheme is equivalent to a random one, as our placement also leverages random selections. We thus use recent works on feasibility of behavior prediction of at least a subset of the network nodes to back up our claim [18, 16, 24].

5.1.1 R&A method core

When a new data item has to be stored, a new cluster is created, where each node belonging to this cluster stores a replica (nodes may of course participate to more than one cluster). This cluster is filled as follow: the system first selects a *reference node* randomly. Then the node whose behavior is the most anti-correlated to this *reference node* is selected to form a pair of anti-correlated nodes. This pair is then added to the cluster. This process is iterated until the number of nodes in the cluster is equal to the system replication factor k. In case of an odd k, an additional random node is included in the cluster. Note that the selection of each *reference node* might be achieved using a random walk in a decentralized system for example or simply by a uniform sampling in a server-assisted system. The randomness inherent to the selection of devices in the cluster leads to a low bias regarding to load balancing while pairs of anti-correlated behaviors improve data availability.

5.1.2 Experimental evaluation of R&A

In order to evaluate our R&A placement strategy we use a public trace of Skype [12], which exhibits a high heterogeneity among availabilities of nodes. As *availability of a data* is defined as the presence of at least one replica at anytime, random placement performs well on stable traces as DNS, as a majority of the nodes are always up; this ensures with high probability that if they are selected, the replicas they host suffice to achieve the targeted availability. In the Skype trace, nodes having uptime less that 1% have been removed from the trace, resulting in a number of alive nodes equals to 1901.

We conducted the simulations as follows: each of the *alive* nodes stores one data item, with different k (from 2 to 10). Then we evaluate the availability mean compared to a random placement strategy. We consider a two-week period for the evaluation: the anti-correlation between availability behaviors is computed over the first week (training period). The evaluation of data availability is performed on the second week. A data is available if at least one node holding a replica is online for *each unit time* of the entire evaluation period. Availability mean and standard deviation are plotted for

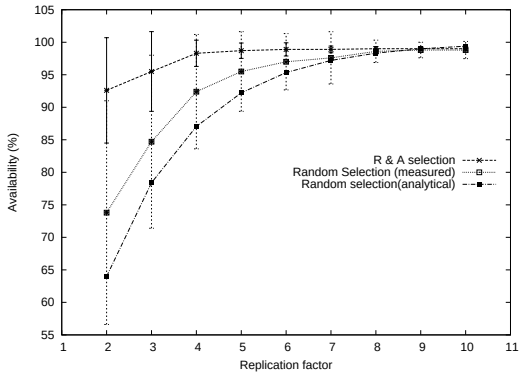

Figure 4: Availability using replication

different replication factors k. Results are depicted on Figure 4.

We compare our R&A strategy to practical random placement [3, 6, 23, 7], as well as with with the analytical model of random placement (often used in replica maintenance papers). The availability is plotted for each k (denoted A_k) given by:

$$A_k = \sum_{i=1}^{k} C_k^i (\bar{p})^i (1-\bar{p})^{k-i} \quad (1)$$

where $\bar{p} = \sum_{x=1}^{N} \frac{p_x}{N}$ (p_x is the mean availability of node x on the period and N is the number of hosts participating in the system). In the Skype trace we measured $\bar{p} = 0.4$.

Results Whereas both strategies tend to achieve the same availability with a high replication factor, R&A placement leads to an increased availability mean compared to a random placement strategy up to $k = 9$. As replicating the whole dataset more than 10 times is highly unrealistic in any practical system, this constitutes a significant improvement over random strategy. Note that complex systems such as Total Recall or Carbonite rely on such a random strategy. For example, the same availability and standard deviation ($98.6\% \pm 3.4$ for random, $98.7\% \pm 2.4$ for R&A) are obtained with 8 replicas with a random strategy whereas only 5 replicas are sufficient using our method. Conversely for the same replication factor k, availability is increased. For example for a k equals to 5, a random strategy achieves $95.5\% \pm 12.6$ of availability while our method leads to an availability equals to $98.7\% \pm 2.4$.

In addition we observe that analytical availability, despite giving the right trend, is under-estimated. This illustrates that in a practical heterogeneous system, availability might not be reduced to a simple arithmetic mean. In fact equation (1) ignores all distributional information on nodes availability, as it is reduced to the scalar value $\bar{p}$. The drawback of this definition of availability are explained in more details in [10].

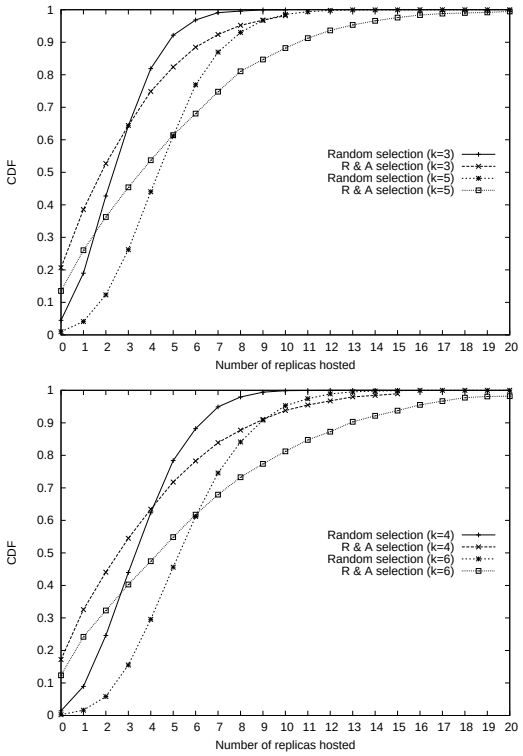

Figure 5: Load balancing for k=3;5 & k=4;6

5.1.3 Load balancing

One of the main drawback of biasing replica placement (as opposed to a uniform random strategy) is the potential impact on the load balancing. In order to remain as scalable as possible, it is desirable the load balancing not be too impacted by the placement strategy. Obviously, when replicas are preferentially placed on highly available nodes [18], the feasibility of the placement scheme directly depends on the proportion of those highly available nodes in the system.

Even a random placement scheme is impacted by the effective availability of nodes in the network as this impacts the uniformity of the random choice. As nodes for replication are chosen online when a new replica has to be inserted, or repaired, highly available nodes tends to be naturally contacted more than their less available counterparts. This results in a slight skew in distribution (see for instance [27]). Random placement is the most simple and efficient scheme for load balancing, with no notion of fairness involved. Yet, it is an important standard to compare too.

Figure 5 plots the cumulative frequency of the number of replicas hosted by all nodes in the system, when random placement and R&A are used. Results show that the load balancing resulting from creating clus-

ters with random placement and their anti-correlated counterparts is close to the random placement strategy. It could be explained by the fact that in addition to the random part of the heuristic, no behavior is advantaged while computing anti-correlation, as opposed to a system where highly available nodes are always rewarded. Then if nodes availability patterns exhibit enough heterogeneity, R&A provides a *natural* load balancing, close to the one obtained with random placement.

5.1.4 R&A placement for erasure codes

Erasure codes have been proved as being another efficient way to obtain data availability and durability in distributed storage systems [29]. We now assess the application of our R&A placement strategy in such settings.

So far, we consider that the availability of a replicated data was achieved with at least one node holding a replica online at any time, among k, thus masking transient unavailability of other replicas. This can be considered as a particular application of providing j available nodes out of a set of n. Erasure codes are able to reconstruct a data with j out of n encoded blocks. We thus seek to provide a subset j of nodes that maximize availability, so that the correct amount of encoded blocks can be found to reconstruct the data.

In practice, we model a typical version of j out of n redundancy scheme in which n code blocks are building in a cluster, and data is defined as available only if at least j code blocks are available within this cluster. The *code rate* is defined as j/n. The availability denoted $A_{(j,n)}$ is adapted from (1) and given by :

$$A_{(j,n)} = \sum_{i=j}^{n} C_n^i (\bar{p})^i (1-\bar{p})^{n-i} \tag{2}$$

We evaluate our R&A method using erasure codes on the same Skype trace, along with a random placement also on n nodes. This simply consists in now placing n blocks with both methods. Figure 6 plots the availability obtained for three different *code rate* (1/2;1/3;1/4), with an increasing n. The top figure corresponds to data availability resulting from the real Skype trace while availability on the other one is computed analytically using equation (2) and $\bar{p} = 0.4$ as in part 5.1.2. We observe a clear improvement in data availability regardless of the *code rate* over the random placement scheme. For example, the same performance is obtained for a random selection with a code rate equals to 1/4 while using our method a *code rate* only equals to 1/3 is sufficient. At the scale of the whole network, this may save an important amount of overhead. The intuition behind this result is that our placement strategy has a "built-in" notion of time span over a predefined period; each time the algorithm picks a random node and its anti-correlated counterpart, availability is increased on this period, while random placement relies on "luck" to pick nodes that will fill temporary unavailability holes. As we seek a larger set j of nodes online at the same time (where with basic replication, $k = 1$ replica is enough), the positive effect of a clever placement is increased.

An interesting effect can be observed on both theoretical and simulation figures. While for reasonable rates, curves are showing good availability, the rate 1/2 causes performance to significantly degrade as n increases, for both placement schemes (even if R&A still performs better). This is explained by the fact that in this system (*i.e.* in this availability trace), there is not enough sets of j nodes simultaneously available at the same time, so that j nodes among n could not be satisfied. This suggests that a deep study of availability of nodes in a network must be considered when deploying erasure codes.

Another observation is that as in Section 5.1.2, we note that whereas analytical results provide tendencies about variations of availability with an increasing n, data availability values are under-estimated in all cases. This results from the too restrictive definition of the availability in Equation (2). More details can be found in [10].

Finally, we remark that this j among n availability target can also have other fields of application, as for instance providing higher throughput when downloading a data from the storage system, as j replicas are available for parallel downloads.

6 Timeout for repairs

In addition to the initial replica placement strategy, a distributed storage system has to provide a reliable repair mechanism so as to ensure data durability. Bandwidth is a crucial element in the reliable repair strategy for data durability since it can only be ensured if there is sufficient bandwidth to repair lost replicas. In our case, bandwidth can be inferred by the number of repairs the system has to perform; we then explicitly count repairs.

When designing a repair mechanism, one challenge is to decide *when* to trigger the repair. In other words, when is it reasonable to conclude that a node has failed or left permanently, thus requiring to repair or if the node is only temporarily disconnected. In the latter case, a repair may turn out to be useless, consuming unnecessarily some bandwidth. A traditional way to decide on the nature of a failure is to use a *timeout*. Typically, once a node has not replied to a solicitation after a timeout has expired, it is considered as permanently failed. However deciding on a timeout value is difficult.

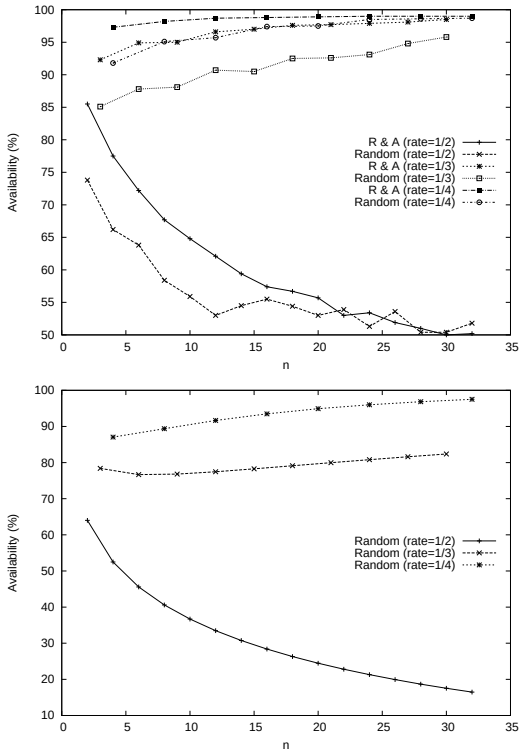

Figure 6: Availability using erasure codes

More specifically, nodes may exhibit various availability patterns; therefore, defining a system-wide timeout value might not be optimal. Deciding on an optimal value for an administrator is also a tedious and difficult task. So in order to simultaneously solve these issues, we designed an adaptive per-node timeout mechanism which sets a timeout at a node granularity, leveraging the availability history of each node in the system.

6.1 Node-specific & adaptive timeout

This section describes how the timeout value is made both adaptive and node-specific, *i.e.* tailored to every single node based on its availability history. Recall that in a cluster, all nodes are monitored. Therefore at each node disconnection (*i.e.* the beginning of an unavailability session), a timer is started measuring time until its reconnection (either on a monitoring server, or by cluster nodes themselves, as explained in Section 4).

The cluster is then aware of unavailability duration of each disconnected node. As in the classical timeout model, if the unavailability period of a given node exceeds its timeout value, the node is considered as permanently down and the system provides this cluster with a new node to be used to *repair* the lost replica. As opposed to most approaches that set a system-wide timeout, we seek determining an accurate timeout value reflecting each node behavior with respect to availability. An important observation is that the repair process is triggered by inference of node failure rather than conclusive knowledge. This suggests that one might analyse node failure detection in a standard *inference framework*, using false alarm probability. In fact the *a priori* probability for a given system that nodes will return give little information at the host level.

Contrarily adding extrinsic information, such as the habits of a given node, might help when observing a given unavailability time to evaluate the probability that this node will return, therefore that the disconnection (the failure) is transient. To illustrate our purpose, consider the following example: if a node, usually available at all times, is detected unavailable for a few hours, the probability that it will reconnect is low, and lower as time passes. On the contrary, consider now a node subject to a diurnal availability pattern. If such a node is detected unavailable for the same number of hours, the probability that the disconnection is transient is high. This simple example illustrates that considering statistical knowledge at a node granularity can be useful to determine the probability that a node will reconnect and therefore the probability that the failure is transient. This instances why a per-host timeout value is desirable in a heterogeneous system.

An important point is that if a host is wrongfully declared as permanently failed (i.e. if it returns in the system whereas its timeout has expired) it will be reintegrated within clusters which it was part of before the disconnection. This timed-out host involved a repair in clusters it belonged to, then its return yields an excess of replicas. This excess authorizes a relaxed reparation process, as the current count is over the target k. Our idea is to *adapt* dynamically the timeout for each host, reflecting the current "criticality" of the situation (more or less replicas online that the fixed threshold k). In case of an excess, this is translated in our adaptive method by setting a less aggressive timeout for each host of the concerned cluster. Thus in addition to be defined at the node-level, timeout is adaptive in this way: a consequence is that a balance is created, emerging from the fact that errors are compensated by a less aggressive policy. Hereafter is explained how the timeout is computed at the node-level.

6.2 Timeout model

The novelty of our timeout model is to be constructed at the node-level, so all the following is given for a particular node. Then all hosts in the system follow this model using their **own** attributes. Consider the following hypothesis:

- H_0 The host will return in the system
- H_1 The host will not return in the system

H_1 et H_0 are two disjoint events then $\Pr(H_1) = 1 - \Pr(H_0)$. $\Pr(H_0)$ is the *a priori* probability that the host will return in the system. $\Pr(H_0)$ could for example be evaluated as the ratio between transient failures and all failures (either transient or permanent) in the system directly computed on a server or resulting from an aggregation protocol in a decentralized way. Let $t_{downtime}$ be the duration of the current downtime of the host and $t_{timeout}$ the timeout value associated to this host. Let its downtime distribution be f_d, this distribution verifies :

$$\int_{(t=0)}^{\infty} f_d \, dt = 1$$

then

$$\Pr(t_{downtime} > t_{timeout} \mid H_0) = \int_{(t=timeout)}^{\infty} f_d \, dt$$

By definition $\Pr(t_{downtime} > t_{timeout} \mid H_0) \in [0,1]$ and $\Pr(H_0) \in [0,1]$. $\Pr(t_{downtime} > t_{timeout} \mid H_0)$ then corresponds to the statistical knowledge on each host behavior. In fact as the availability history of each host is stored, its downtime probability distribution can be computed directly, and then also $\Pr(t_{downtime} > t_{timeout} \mid H_0$ depending on the timeout value. By hypothesis :

$$\Pr(t_{downtime} > t_{timeout} \mid H_1) = 1$$

$$\Pr(t_{downtime} < t_{timeout} \mid H_1) = 0$$

In fact if the node does not come back in the system, on the one hand its downtime will be superior to any timeout value, on the other hand its downtime cannot be inferior to any timeout value. We also assume :

$$P_{FA} = \Pr(H_0 \mid t_{downtime} > t_{timeout})$$

P_{FA} is the probability that a host comes back in the system whereas it has been timed-out and thus it has been decided by the system that it would not return. P_{FA} is called the False Alarm probability. The higher P_{FA} the more (probably) useless reparations are tolerated. According to the Bayes'Theorem :

$$P_{FA} = \frac{\Pr(H_0 \cap (t_{downtime} > t_{timeout}))}{\Pr(t_{downtime} > t_{timeout})}$$

$$P_{FA} = \frac{\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)}{\Pr(t_{downtime} > t_{timeout})}$$

$$P_{FA} = \frac{\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)}{[\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)] + 1 - \Pr(H_0)}$$

Finally :

$$P_{FA} = \frac{\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)}{1 + \Pr(H_0) \times [\Pr(t_{downtime} > t_{timeout} \mid H_0) - 1]} \quad (3)$$

The definition interval of P_{FA} is then $[0, \Pr(H_0)]$.

To sum up, we have (3) that expresses the false alarm probability as a function of the a priori probability that a host will eventually return in the system and of its downtime distribution. This expression is necessary to compute per-node timeouts, as explained in next section.

6.3 Method core of node-specific timeout

We now show how to compute an adaptive timeout for each node. The scheme presented here after is performed in each cluster at each time unit.

Each cluster has to deal with its departed and coming-back hosts. Returning nodes reintegrate their cluster. On the contrary, if a node is timed-out and if the cluster size falls under the replication factor, the system provides this cluster a new node in order to create a new replica. This new node can be chosen randomly or following a specific placement strategy as proposed in the first part of this paper.

The following steps are executed at each cluster periodically:

- Update cluster (Reintegrate and/or replicate)
- Compute false alarm probability
- Determine timeout for each node of the cluster

The false alarm probability is then computed at the cluster level. The purpose of varying this probability is to reflect how critical is the situation compared to availability history of hosts. Hereafter is proposed a method used to evaluate this situation (note that this method is not proved as being an optimal one, but just a practical example that we use for evaluation). At each time unit a comparison is made on the number of available replicas between the current one and the one the week before (or that would have been here with current updated cluster). This difference, noted Δ, is thus positive if there is more replicas, and negative otherwise. Note that the only relevant parameter measured is the number of available replicas, regardless of which node stores them. If there is no difference, the false alarm probability is defined as the middle of its interval so $P_{FA} = \Pr(H_0)/2$. We define the step of variation as $\Pr(H_0)/k$, with k the number of desired replicas of a data. P_{FA} is computed as: $P_{FA} = \Pr(H_0)/2 - (\Delta \cdot \Pr(H_0)/k)$. At the end of this step, each cluster thus has a false alarm probability that has varied as a function of the measured criticality of the situation, regarding effective k.

Once the false alarm probability has been determined in each cluster, each host is able to specify its own timeout value. From Equation (3) we have :

$$\Pr(t_{downtime} > t_{timeout} \mid H_0) = \frac{P_{FA} \cdot (1 - \Pr(H_0))}{\Pr(H_0) \cdot (1 - P_{FA})}$$

$$\int_{(t=timeout)}^{\infty} f_d \, dt = \frac{P_{FA} \cdot (1 - \Pr(H_0))}{\Pr(H_0) \cdot (1 - P_{FA})} \quad (4)$$

Therefore timeout value is placed in order to solve the above equation (4).

6.4 Evaluation

We simulate our per-host timeout on public traces. We compare the cost/availability trade-off of our approach against systems using global timeout from aggressive (low) to relax (high) values. So as to make a fair comparison, clusters are here generated in a random way. Replicas of false positive nodes are also reintegrated in the global timeout simulation. We evaluate the trade-off on three different system traces, namely PlanetLab, Microsoft and Skype. As in Section 5.1.2 nodes having uptime inferior to 1% have been removed from the trace, resulting in a number of alive nodes respectively equals to 308, 51313 and 1901. We simulate that each of the *alive* nodes stores one data item, varying replication factor from 3 to 6 (5 to 8 for Skype). We evaluate our approach and the global timeout one along the following metrics: mean data availability and number of repairs generated by timed-out hosts. We provide results for Microsoft (Figure 8) and PlanetLab (Figure 7) traces. The Skype results (Figure 9) are given in next Section, as we will finally add on plots results from the combination of the R&A placement strategy and the use of per-host timeout.

The *learning period* is the first week, after which the history window simply slides. In fact an initial availability history must be available so that each host can afford to compute its initial downtime distribution. Data availability mean and number of repairs are then evaluated on the next two weeks. During this two weeks downtime distributions of each host are updated following their behavior, after each new end of a downtime session. On Figures 7, 8 and 9, the X-axis represents the mean of data unavailability in percent (*i.e.* $1 - availability$). The Y-axis is the number of repairs by data and by day, it is equivalent to the number of timed-out hosts, as each of them triggers a repair. Then unavailability versus number of repairs trade-off is plotted for various values of global timeout (10 to 80 hours for Microsoft and 20 to 280 hours for PlanetLab) and for the per-host timeout; this is the classic way of representing timeouts versus repairs, as done in recent papers [26, 30].

Results A first observation applicable to all plots is that not surprisingly, an aggressive global timeout value (10H) leads to a low unavailability but produces a high repair rate, triggering an important bandwidth consumption in practice. On the contrary, a relaxed timeout value (80H) enables to reduce the number of repairs at the price of decreased data availability.

A second observation is that regardless of the global timeout value, our per-host timeout always provides a better trade-off between availability and number of repairs. In other words in all cases for an equivalent availability (and obviously for the same replication factor), the number of repairs will always be higher using global timeout than our per-host timeout. This represents a significant bandwidth saving resulting from the decreased number of repairs.

In addition not only raw performances on those metrics are greatly improved, but another important benefit of the adaptive timeout method is that a system administrator does not have to arbitrarily set a static value for timeout. Instead, the system self-organizes to compute adapted values, thus suppressing the need for such a decision before runtime of the storage application.

7 R&A and per-host timeout combined

So far, we have proposed two techniques, leveraging the availability history of hosts. In this section, we consider their combination, for possible improvements.

As their respective merits have been presented separately, we now use both R&A placement and adaptive per-host timeout in the same simulations. The experiments are conducted on the Skype trace with adaptive and per-host timeout, while adding the R&A placement strategy to constitute clusters. The *learning period*, where the system "learns" the availability habits of its hosts, lasts one week in order to initialize the storage system. In fact an initial availability history must be accessible so that *(i)* the system is able to establish anti-correlation on behaviors, and *(ii)* each host can afford to compute its initial downtime distribution. In practice, traditional random placement could be used the first week, so the storage system can be operational while the leaning phase is processed. The data availability mean and the number of repairs are then evaluated on the next two weeks.

Results are plotted on Figure 9 for a replication factor k varying from 5 to 8; same positive results are obtained for a larger panel of values for k. The comparison is made between our per-host timeout and a global timeout varying from 10 to 80 hours with a random placement strategy. Again, to ensure a fair comparison, reintegration of replica (then hosted by returning nodes) is also included in the global timeout scheme. Finally, we plot unavailability mean and number of repairs resulting from the combination of the R&A placement

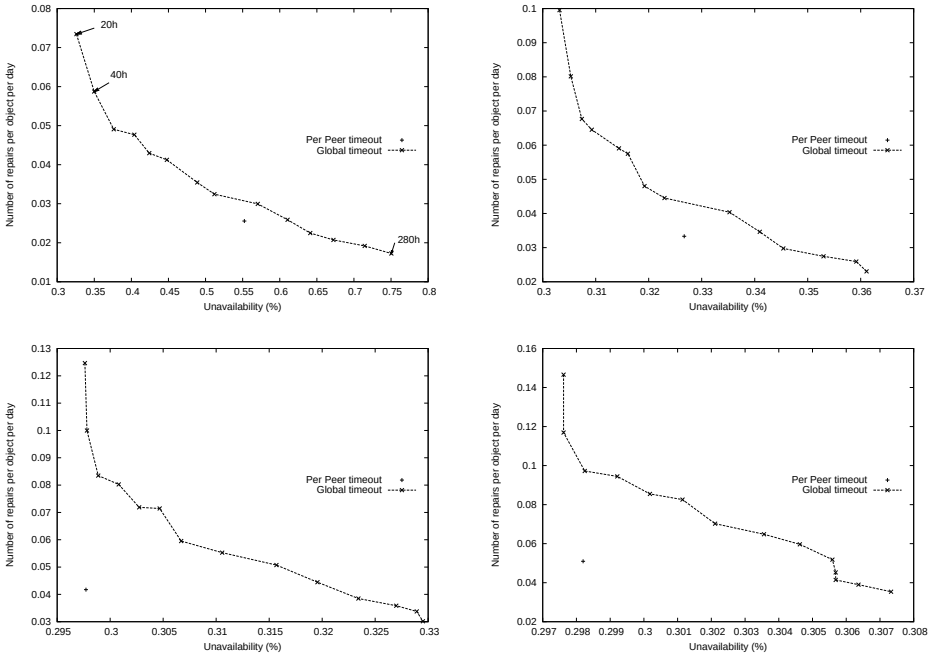

Figure 7: PlanetLab results for values of k ranging from 3 to 6.

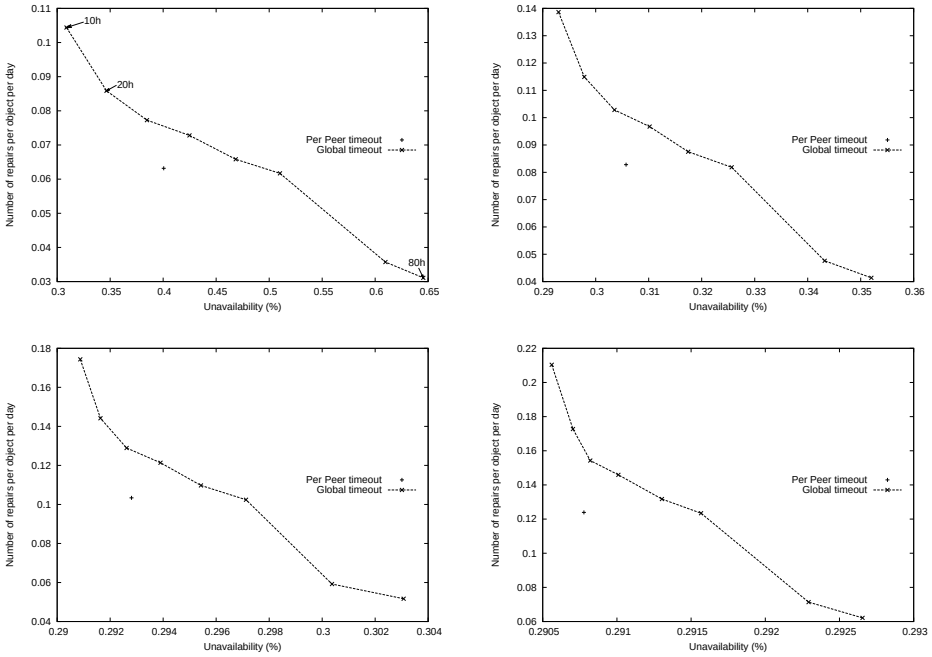

Figure 8: Microsoft results for values of k ranging from 3 to 6.

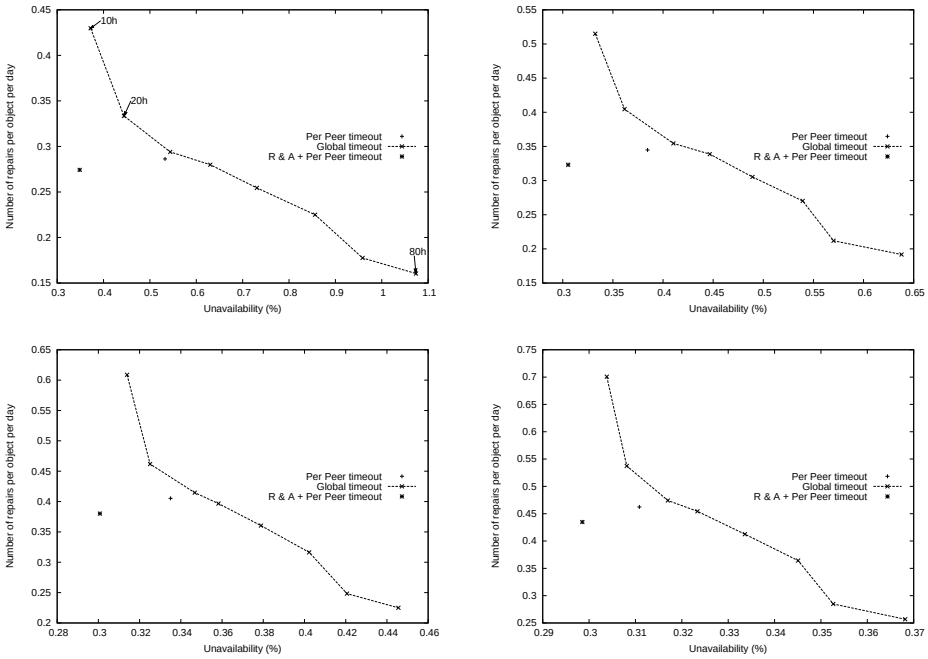

Figure 9: Results of R&A placement and adaptive timeout on Skype trace, for values of k ranging from 5 to 8.

strategy and the per-host timeout in order to measure the additional gain while applying both our availability-based methods.

Results show that the combination of our two methods clearly outperforms the global timeout. In fact even compared with the most aggressive global timeout value (10H), which produces the lowest unavailability, availability is slightly increased regardless of the replication factor when applying our methods; the repair rate is also greatly reduced by a mean factor of 38% in comparison to this most aggressive timeout approach.

Cost of the methods A practical question is whether or not such availability-based methods imply significant costs for their implementation in a large-scale storage system. Such a framework requires operations that appear to be lightweight in terms of bandwidth and control message overhead: for monitoring , simple PINGs, in order to track other cluster nodes availability are to be used. Finding anti-correlated nodes can be achieved in a scalable and inexpensive manner through a gossip mechanism (see T-Man for example). Finally, local operations on vectors are only CPU consuming. This is to compare with the replication and repair costs of distributed storage systems. The amount of data to store in modern systems, as well as the size of data files are constantly increasing. We sketch a little example based on results from Figure 9, where every of the 1901 nodes participating, backups $1GB$ of data, with $k = 8$; the system has to handle around $15TB$. A saving of 38% of repairs per-node per-day represents around $4TB$ of bandwidth saved per-day at the network scale. In this light, network costs needed to implement our techniques is marginal, especially when the network or the data stored grows large.

8 Conclusion

In this paper, we have motivated and shown the interest of using the availability information of nodes participating in a system. While this idea has already been explored at a general level, we have advocated for implementation at a finer level than the system level, that is to say at the granularity of nodes. We have tackled two important points in storage systems, namely replica placement and timeouts for repairs; results on real traces have shown substantial gain of this availability-aware framework. We expect those two practical contributions to be used as plugins in deployed systems.

References

[1] Repository. http://www.cs.uiuc.edu/homes/pbg/availability/.

[2] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding availability. In *IPTPS, Int'l Work. on Peer-to-Peer Systems*, 2003.

[3] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: system support for automated availability management. In *NSDI'04: Proceedings of the*

1st conference on Symposium on Networked Systems Design and Implementation, pages 25–25, Berkeley, CA, USA, 2004. USENIX Association.

[4] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[5] William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 34–43, New York, NY, USA, 2000. ACM.

[6] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.

[7] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215, New York, NY, USA, 2001. ACM.

[8] John R. Douceur. Is remote host availability governed by a universal law? *SIGMETRICS Perform. Eval. Rev.*, 31(3):25–29, 2003.

[9] Alessandro Duminuco, Ernst Biersack, and Taoufik En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *CoNEXT '07: Proceedings of the 2007 ACM CoNEXT conference*, pages 1–12, New York, NY, USA, 2007. ACM.

[10] Richard J. Dunn, John Zahorjan, Steven D. Gribble, and Henry M. Levy. Presence-based availability and p2p systems. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing*, pages 209–216, Washington, DC, USA, 2005. IEEE Computer Society.

[11] Kevin M. Greenan, Parascale Inc, James S. Plank, and Jay J. Wylie. Mean time to meaningless: Mttdl, markov models, and storage system reliability. In *The 2nd Workshop on Hot Topics in Storage Systems (HotStorage2010)*, 2010.

[12] Saikat Guha, Neil Daswani, and Ravi Jain. An experimental study of the skype peer-to-peer voip system. In *Proceedings of the 5th international workshop on peer-to-peer systems (IPTPS '06)*, 2006.

[13] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.

[14] Derrick Kondo, Bahman Javadi, Alexandru Iosup, and Dick Epema. The failure trace archive: Enabling comparative analysis of failures in diverse distributed systems. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:398–407, 2010.

[15] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, 2000.

[16] Stevens Le Blond, Fabrice Le Fessant, and Erwan Le Merrer. Finding good partners in availability-aware p2p networks. In Rachid Guerraoui and Franck Petit, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 5873 of *Lecture Notes in Computer Science*, pages 472–484. Springer Berlin / Heidelberg, 2009.

[17] Sergey Legtchenko, Sébastien Monnet, Pierre Sens, and Gilles Muller. Churn-resilient replication strategy for peer-to-peer distributed hash-tables. In *SSS*, pages 485–499, 2009.

[18] James W. Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 6–6, Berkeley, CA, USA, 2006. USENIX Association.

[19] R. Morales and I. Gupta. Avmon: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems. *Parallel and Distributed Systems, IEEE Transactions on*, 20(4):446 –459, apr. 2009.

[20] Daniel Nurmi, John Brevik, and Rich Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In JosÃ' C. Cunha and Pedro D. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 432–441. Springer Berlin / Heidelberg, 2005.

[21] L. Pamies-Juarez, P. Garcia-Lopez, and M. Sanchez-Artigas. Rewarding stability in peer-to-peer backup systems. pages 1 –6, dec. 2008.

[22] Sriram Ramabhadran and Joseph Pasquale. Durability of replicated distributed storage systems. In *SIGMETRICS '08: Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 447–448, New York, NY, USA, 2008. ACM.

[23] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[24] Krzysztof Rzadca, Anwitaman Datta, and Sonja Buchegger. Replica placement in p2p storage: Complexity and game theoretic analyses. *Distributed Computing Systems, International Conference on*, 0:599–609, 2010.

[25] Kiran Tati, , Kiran Tati, and Geoffrey M. Voelker. On object maintenance in peer-to-peer systems. In *In Proc. of the 5th International Workshop on Peer-to-Peer Systems*, 2006.

[26] Jing Tian, Zhi Yang, Wei Chen, Ben Y. Zhao, and Yafei Dai. Probabilistic failure detection for efficient distributed storage maintenance. In *SRDS '08: Proceedings of the 2008 Symposium on Reliable Distributed Systems*, pages 147–156, Washington, DC, USA, 2008. IEEE Computer Society.

[27] Laszlo Toka, Matteo Dell amico, and Pietro Michiardi. Online data backup : a peer-assisted approach. In *P2P'10, 10th IEEE International Conference on Peer-to-Peer Computing, August 25-27, 2010, Delft, The Netherlands*, 08 2010.

[28] Vytautas Valancius, Nikolaos Laoutaris, Laurent Massoulié, Christophe Diot, and Pablo Rodriguez. Greening the internet with nano data centers. In *CoNEXT '09: Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 37–48, New York, NY, USA, 2009. ACM.

[29] Hakim Weatherspoon and John Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *IPTPS*, 2002.

[30] Zhi Yang, Yafei Dai, and Zhen Xiao. Exploring the cost-availability tradeoff in p2p storage systems. In *ICPP '09: Proceedings of the 2009 International Conference on Parallel Processing*, pages 429–436, Washington, DC, USA, 2009. IEEE Computer Society.