Flexible Replica Placement for Optimized P2P Backup on Heterogeneous, Unreliable Machines

Piotr Skowron, Krzysztof Rzadca
p.skowron@mimuw.edu.pl, krzadca@mimuw.edu.pl,
Institute of Informatics, University of Warsaw, Poland

January 10, 2016

Abstract

P2P architecture is a viable option for enterprise backup. In contrast to dedicated backup servers, nowadays a standard solution, making backups directly on organization's workstations should be cheaper as existing hardware is used; more efficient as there is no single bottleneck server; and more reliable as the machines can be geographically dispersed.

We present an architecture of a p2p backup system that uses pairwise replication contracts between a data owner and a replicator. In contrast to a standard p2p storage system using directly a DHT, the contracts allow our system to optimize replicas' placement depending on a specific optimization strategy, and so to take advantage of the heterogeneity of the machines and the network. Such optimization is particularly appealing in the context of backup: replicas can be geographically dispersed, the load sent over the network can be minimized, or the optimization goal can be to minimize the backup/restore time. However, managing the contracts, keeping them consistent and adjusting them in response to dynamically changing environment is challenging.

We built a scientific prototype and ran experiments on 150 workstations in our university's computer laboratories and, separately, on 50 PlanetLab nodes. We found out that the main factor affecting the performance of the system is the availability of the machines. Yet, our main conclusion is that it is possible to build an efficient and reliable backup system on highly unavailable machines, as our computers had just 13% average availability.

1 Introduction

Large corporations, medium and small enterprises, universities, research centers, and common computer users are all interested in protecting their data against hardware failures. The most common approach to protecting data is to keep the backup copies on tape drives, specially designated storage systems, or to buy cloud storage space. All such solutions are highly reliable, but also

expensive. In 2013 the costs of renting 1TB of cloud storage per year from Amazon, Google, Rackspace, or Dropbox was approximately $1000. Additionally, for some organizations, internal data handling policies require that data cannot be stored externally. The price of a single backup server with raw capacity of 14TB exceeds $12,000. A tape-based backup system with the same raw capacity costs about $7,000. These figures do not include additional costs of service, maintenance, and energy. Additionally, if many workstations are replicated at a single server, the server may become a bottleneck and may not be able to provide satisfactory throughput; performance can be further degraded by network congestion. More scalable solutions exist, but are even more expensive.

There is still a need for cheaper alternatives for enterprise backup. On one hand, a significant research effort focuses on optimization techniques for dedicated backup servers, such as deduplication techniques [18,35] or erasure codes [39,42]. On the other hand, a p2p architecture can be explored in the context enterprise backup. Common PCs are cheaper than reliable servers. Also, in many cases, the unused disk space on desktop workstations can be used without additional costs (Adya et al. [1] discovered a tendency that the unused disk space on the desktop workstations is growing every year; the Moore's law for hard disks capacities, first formulated by Kryder [57] still holds). The bandwidth of multiple nodes (scattered through the network) scales better than of a single server as the network load is more evenly distributed causing less bottlenecks. The system can take advantage of the geographical dispersion of the resources, thus offering better protection in case of theft or natural disasters (e.g., fire or flood). Finally, p2p solutions have already proved to work well in enterprise environments (GFS [24], MapReduce [15], Astrolabe [44], DHT [14] used in HYDRAstor [21], etc.).

Indeed, many p2p storage systems have been already built [2,6–8,11,30,33, 37,48,55,60]. The deduplication techniques get adapted for p2p storage systems [43,58]. There are new erasure codes more suitable for p2p systems [39]. Finally, there are many theoretical models for data placement optimizing data availability [3,4,10,20,40,45,47] and backup/restore performance [41,54]. However, real systems do not fully take advantage of the p2p architecture. There is a gap between theoretical models and real implementations. There are systems (e.g., OceanStore [30] and Cleversafe) that distribute data between geographically remote servers. These systems could be used for backup, but they both use dedicated servers, which stays in contradiction with our primary goal of creating cheap backup system based on existing, unreliable machines. There are also p2p storage systems designed to work on unreliable machines. Farsite [6], as a complete distributed file system must deal with parallel accesses to data, must manage the file system namespace, and ensure that frequently accessed data is highly available. Such requirements force additional complexity and many architectural limitations that do not exist in case of a backup system. On the other hand, since data backup is not a primary use-case, Farsite does not focus on implementing replica placement strategies (e.g., geographical dispersion of replica, or ensuring that data is backed up within a given time window, etc.).

Bridging the gap between many theoretical models [3,4,10,20,40,41,45,47,54]

and prototype implementations, we asked the following question: Is it possible to implement various data placement strategies when machines are unreliable? Certainly, there are more challenges than in the case of centralized or highly-available systems. The machines' unreliability, and perhaps low availability, requires data locations to change dynamically. Is it difficult to continuously optimize the data placement with such assumptions? And, finally, is it difficult to take advantage of the machines and the network heterogeneity?

Our main contribution is the following: (i) We present an architecture of a storage system that uses pairwise (bilateral) replication contracts for storing data and therefore enables to optimize placement of replicas. This architecture includes the protocols for managing replication contracts; and a reliable messaging system. (ii) We implement a scientific prototype (available at `nebulostore.org`) and, based on tests on 150 computers in students' computer laboratories and on 50 machines in PlanetLab [9], we show that it efficiently manages the contracts and ensures efficient backup even under significant peers' unavailability (the students' labs might be considered as a worst-case scenario for an enterprise network, as the computers have just 13% average availability and are frequently rebooted)

In contrast to fixed data placement (storing data in a DHT [2,7,8,37,60]), the data placement in our system is based on storage contracts between an owner of the data and her replicators. A contract for storing a data chunk of owner i on replicator j is a promise made by j to keep i's data chunk for a certain amount of time. Until the contract expires, it cannot be dropped by j (but it can be revoked by i). Since every data chunk is associated with a list of storage contracts, each chunk can be placed at any location (the location depends on the placement strategy). This contract-based architecture can be exploited in two ways. First, the contracts form an unstructured, decentralized architecture that enables to optimize replica placement, making the system both more robust and able to take advantage of network and hardware configuration. Second, contracts also allow strategies for replica placement that are incentive-compatible, such as mutual storage contracts [13,47]. To the best of our knowledge, all previous literature on mutual contracts focused on theoretical analysis only. We complement these theoretical works by presenting an architecture (and a sample implementation) of a contract-based storage system. Yet, in this paper, for the sake of concreteness, we focus on optimization of replica placement for p2p backup in a single organization, where incentives are not needed.

The second new element of our architecture, the reliable (asynchronous) messaging system, enables the system to reach eventual consistency when managing replication contracts on machines that have frequent transient failures (see Section 3.4). Each peer has an associated group of peers (called synchro-peers) that help in delivering messages. If the peer is unavailable, the message is sent to one of the synchro-peers; the synchro-peers replicate the message between themselves. Then, when the recipient is on-line again, it contacts its synchro-peers and gets the messages. Thus, to deliver a message, the sender and the recipient do not need to be on-line at the same time. According to our experiments (Section 5.2), asynchronous messages enable our system to significantly

reduce the time needed to deliver messages.

Since our results are supported not only by the simulations, but also by measurements of an implementation on a real system, we consider them as the proof of the concept that an efficient p2p backup systems can be created and that the heterogeneity of the machines in such a system can be explored.

2 Related Work

In this section, we review related commercial projects and scientific approaches to data replication in distributed systems.

HYDRAstor [21] and Data Domain [61] are commercial distributed storage systems, which use data deduplication to increase amount of the virtual disk space.

Many papers analyze various aspects of p2p storage by either simulation or mathematical modelling. Usually, the analysis focuses on probabilistic analysis of data availability in the presence of peers' failures (see, e.g., the work of Bernard and Le Fessant [3]). Douceur et al. [20], similarly to our system, optimize availability of a set of files over a pool of hosts with given availability: theoretical as well as simulation results are provided for file availability. Chun et al. [10] study by simulation durability and availability in a large scale storage system. Bhagwan et al. [4] and Rodrigues and Liskov [45] show basic analytical models and simulation results for data availability under replication and erasure coding. Toka et al. [54] find a schedule of transfers which minimizes the restore time and also analyze the impact of the size of the set of replicators on restore time. Pamies-Juarez et al. [41] studies the impact of the redundancy on the data retrieval time. Our paper complements these works by presenting a software architecture that allows to implement placement strategies, by considering other measures of efficiency, and by proving that various optimization strategies can be used in an unreliable environment.

As the focus of this work is on data backup in a single organization, we do not analyze incentives to participate in the system. However, to store the data, our system relies on agreements (contracts) between peers. In contrast, in DHT-based storage systems contracts are (implicitly) made between a peer and the system as a whole. Thus, our architecture naturally supports methods of organization that emphasize incentives for high availability, such as mutual storage contracts [13,47] (also these using asymmetric contracts [40]). It is worth mentioning that some papers explore the social interconnections while choosing the replica locations [55]; the tradeoffs between the redundancy, data availability and the ability to place data on the trusted nodes is analyzed by Sharma et al. [49] and Tinedo et al. [52]. These methods can also be adopted for our system.

Many p2p file systems [2,7,8,37,60] use storage and routing based on a DHT [14,46]. A block is hashed to an address that fully determines the locations of block's replicas. Thus, such architectures are less suitable for balancing the load on replicating workstations, or for optimizing the placement of the repli-

cas. While these solutions focus on consistency of the data modified by multiple users, this paper focuses on the issue of the best replication of the data.

To optimize the placement of replication contracts, we use a distributed optimization protocol relying on a distributed priority queue maintained by all peers. An alternative is to use other load balancing or optimization techniques, such as Messor [36], T-Man [27] or pair-wise exchanges [51].

OceanStore [30] and Cleversafe [12] spread replicas among geographically remote locations achieving the effect of deep archival storage. These systems combine software solutions with a specially designed infrastructure that consists of numerous, geographically-distributed, servers. The main contribution of these systems, from our perspective, is the resignation from a common DHT and the introduction of a new assumption that any piece of data can be located at any server. These systems, however, do not discuss the issue of replicating data on ordinary workstations (which are, in contrast to servers, frequently leaving and joining the network) and do not present any means allowing to handle such dynamism.

Wuala [33] moved one step further by proposing a distributed storage based not only on a specially dedicated infrastructure, but also including a cloud of workstations of users who install the Wuala application. However, since late 2011, Wuala no longer supports p2p storage. The idea of using a hybrid architecture of central servers and user machines, called peer-assisted backup, is also explored by Toka et al. [53]. Other p2p backup software include Backup P2P [1], Zoogmo[2], or ColonyFs [3].

FreeNet [11] is a p2p application that exposes the interface of a file system. Its main design requirement is to ensure anonymity of both authors and readers. The underlying protocol relies on proximity-based caching. When a data item is no longer used, it can be removed from a caching location. Similarly, in Pangea [48] a replica is created whenever and wherever data is accessed. The idea of replicating data at locations near end users was successfully implemented by Akamai [34,38], the world largest content delivery network. Naturally, there are other works on proximity-based caching [26,31,32,50].

Farsite [6] was a Microsoft's 6-years long project aimed at creating distributed file system for sharing data between thousands of users. The retrospective gave us the feeling of following a good direction. Authors emphasize that: first, real scalability must face the problem of constant failures in the network; second, in a scalable system, manual administration must not increase with the size of the network. We followed both requirements when designing our system.

There are a few substantial differences between Farsite and our prototype implementation. Most importantly, Farsite's architecture does not rely on mutual contracts, which allow us to implement both incentives and mechanisms ignoring the black listed peers.

[1] sourceforge.net/projects/p2pbackupsmile/

[2] zoogmo.wordpress.com

[3] launchpad.net/colonyfs

In Farsite updates of data are committed locally and the changes are appended to the log (similarly to Coda [28]). The log is sent to a group of peers responsible for managing a subset of the global name space (called the directory group). The group periodically broadcasts the log to all its members. As the directory group uses a byzantine fault tolerant protocol [19], no file can be modified if one third or more of the group is faulty. Since we consider a backup system in which data is modified only by the owner, we are able to gain in flexibility and robustness. In our asynchronous updates mechanism, every peer has an associated group of peers managing its asynchronous messages—we refer to such peers as *synchro-peers*. Synchro-peers are independent of replicas, which results in a desired property that any peer can keep replicas for any chunk of data. Thus, replicas can be chosen so that they constitute the optimal replication group.

Farsite is a distributed file system so its complexity is higher compared to a backup system. However, Farsite is not a backup system, so it does not support backup-specific requirements like placing replicas in geographically distributed locations, or the optimization of the backup/restore time.

3 System Architecture

Our system uses a mixed architecture that stores control information, meta-data and data in three different ways. The control information that allows peers to connect to each other must be located efficiently—hence we use a DHT as a storage mechanism. In contrast, each peer is responsible for finding and managing peers who replicate its data (its *replicators*). The replication contracts enable us to optimize replica placement and thus to tune replication to a specific network configuration. The meta-data describing replication contracts are kept by the data owner and by the replicator. Chunks of data are kept in an unstructured overlay; concrete locations are described by the meta-data.

3.1 Control Information

The basic attributes of a peer are kept in a structure called *PeerDescriptor*. For each peer, its PeerDescriptor contains:

- identification information (the public key);
- information needed to connect to this peer (the current IP address and the port of an instance of our software running on the workstation) and the user account name in the operating system (account name is required by the current implementation of the data transmission layer—see Section 3.3);
- identifiers of its *synchro-peers* (see Section 3.4).

PeerDescriptors of all peers are kept in a highly replicated DHT: peer's ID (a hash of its public key) is hashed to its PeerDescriptor. As the size of the

control information is small, we are able to afford strong replication (compared to a generic DHT). Thus, instead of a single peer, many peers store the data hashed to a part of the key-space.

3.2 Replication Contracts

The main goal of our prototype is to support nontrivial replica placement strategies; we need to be able to store any replica at any peer. This architecture contrasts with content addressable storage systems that put each chunk of data under an address that is fully determined by the chunk's unique identifier (e.g., by the hash of its content). As a trade-off for flexibility of placing replicas at any location, we need a mechanism for locating the data.

In our system each peer keeps information about replica placement of its data chunks in an index structure called *DataCatalog*. For each data chunk, the catalog stores information about: (i) identifiers of the peers that keep replicas of the data chunk (hereinafter *chunk replicators*); (ii) the size of the chunk; (iii) the version number of the chunk.

Additionally, each peer keeps information about data chunks it replicates. As each storage contract is kept in exactly two places (the owner and the replicator), contracts are consistent and it is easy to retrieve the metadata (i.e., the DataCatalog) in case of a failure. Since peers are unreliable, the process of contracts negotiation can break at any point, possibly leading to two types of inconsistency: an owner o believes j is its replicator, while j is not aware of such a contract; or a peer j believes to be o's replicator, while o is not aware of such a contract. Contracts negotiation is however idempotent and because the contracts are kept both by the owner and by the replicator, such inconsistencies can be easily fixed. Each peer periodically sends messages to its replicators with the believed contracts (and versions of data chunks, which allows the replicators to update the chunks of data which are out of date). Each replicator periodically sends similar information to the appropriate owners. Detected inconsistencies can be resolved either by adopting the owner's state; or by always accepting a replication agreement.

The DataCatalog is stored in a file but it is not replicated. In this way we avoid the additional overhead of updating the catalog at remote locations, when the contract for any chunk is changed; because machines are unreliable, in many cases we even would not be able to update the DataCatalog at a remote peer as the peer can simply be unavailable. On the other hand, both owners and replicators are aware of all their storage contracts. When the local data of any peer is lost, the peer (the owner) gossips the information about the failure. The replicators answer to the gossip message with the information about the contracts; the owner uses this information to rebuild the DataCatalog. Once the DataCatalog is reconstructed, the owner locates and rebuilds all missing data. Since the DataCatalog is persistent, its reconstruction is required only in the case of a non-transient failure; thus the reconstruction does not cause much overhead. The standard use cases of replicating a chunk of data and retrieving permanently lost data are depicted in Figures 1 and 2, respectively.

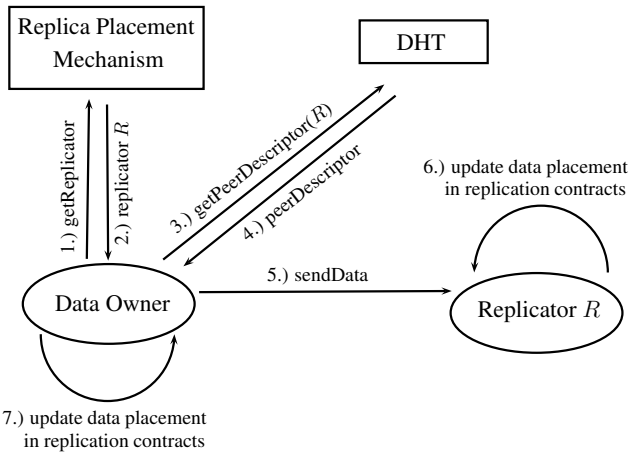

Figure 1: Steps performed when replicating a chunk of data. After asking the local replica placement mechanism for a replicator (steps 1 and 2), three different storage mechanisms are involved: (i) the replicator's peer descriptor is obtained from the DHT (steps 3 and 4 in the diagram), (ii) the data chunk is stored at the replicator (step 5), and (iii) meta-data (that contain storage contracts) is updated both by the data owner and the replicator (step 6 and 7).

Alternatively, in an enterprise environment where a (replicated) server is an affordable option, the meta-data can be kept centrally (in primary memory for faster access). This solution is, however, less scalable.

We designed the mechanism that is responsible for relocating or additionally replicating data chunks that are weakly replicated according to a given abstract metric. The specific metric used in our evaluation takes into account peers' availability, bandwidth, and geographic distribution. It tries to keep all but one replicas as close as possible (not to overload the network) and to keep one replica in a remote location for additional safety (for location-dependent failures, such as fires, floods, or thefts). The metric also balances the load on the machines so that each data chunk can be replicated within the required *backup window* (the time in which each modified chunk must be backed up). The metric is described in Section 4.1. The optimization mechanism (see Section 4.2) is based on hill-climbing—in consecutive steps each peer performs locally optimal changes of its contracts. The optimization of the location of each single data chunk can be performed even if large fraction of peers is unavailable; to perform a single step, we require only 3 peers to be available. Thus, the mechanism can proceed in unreliable environments.

When nodes' parameters (e.g., their availability) change, or when a large number of nodes is added to the system, the contracts are renegotiated. If each such change resulted in data migration, the network and the machines could easily become overloaded. Therefore the process of changing a contract is more

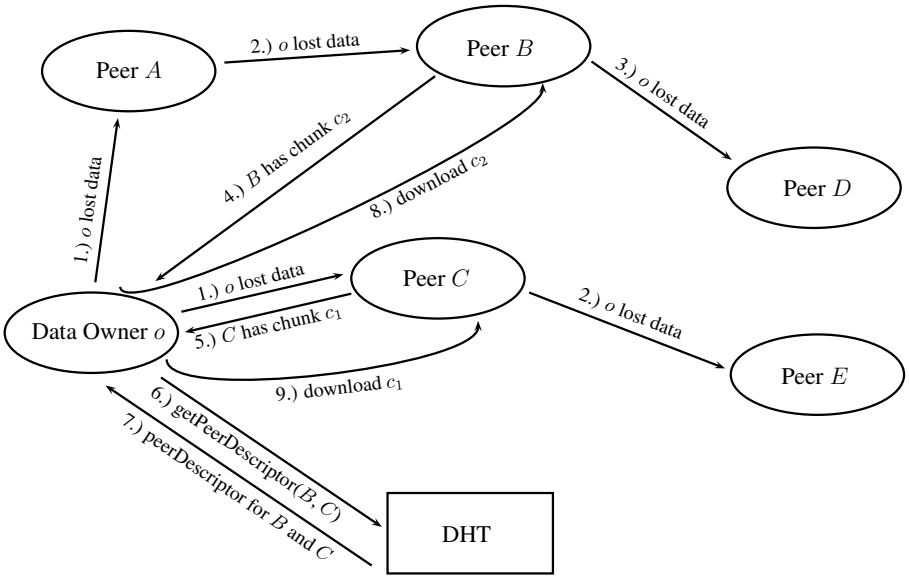

Figure 2: Steps performed after the worst-case crash failure when the owner o looses all the data, including the DataCatalog. o gossips a message "o lost data" (steps 1–3). Data replicators having o's data chunks answer (steps 4 and 5). o rebuilds its DataCatalog; locates the data (steps 6 and 7) and retrieves it (steps 8 and 9).

elaborate. The contracts are allowed to change frequently but such changes do not require data migration. Such temporary contracts are periodically (e.g., daily) committed; after a contract is committed, the data is migrated. The complete mechanism involves some additional details as it must also take into account possible communication failures (see Section 4.3).

3.3 Data Transmission and Updates

Every peer, before placing its replicas at a remote peer, must obtain this peer's permission. Once the peers reach an agreement, they mutually authorize each other using their *PeerDescriptor*'s identities (the public keys stored in the DHT).

The data is transmitted in an encrypted connection. In the current implementation, we use standard Linux tools for data transfers. Each peer runs an ssh daemon that acts as a server that accepts connections of data owners. When a peer initiates a connection to transfer its data, it uses scp as a client.

When an owner modifies its local copy, the updated chunk must be propagated to its replicators. The replicators are informed of changed data chunks through periodic control messages (version numbers are attached to the messages containing contracts sent between the owner and the replicator, described in the previous subsection). The unavailable peers are informed about the changes of data chunks through asynchronous messages (described in the following Section). Once a replicator knows that there is a new version of a data chunk it replicates, it downloads the new version either from the owner or from the other replicators (achieving eventual consistency [56]). Note that if data owners were responsible for uploading the new version to the replicators, a successful transfer would require both the owner and the replicator to be available. In our solution, the replicator is responsible for keeping replicas up to date so we only require that the replicator and any other replicator or the owner are available.

Unlike common backup systems, our system stores only the last version of each data chunk. A system storing many previous versions may be built in the same way as version control software (e.g., svn or git) uses a standard filesystem. More specifically, the previous versions (or the deltas) can be kept in the same data chunk; or the deltas can be kept in separate data chunks.

3.4 Asynchronous/Off-line Messaging

We assume that the workstations may be unavailable just because they are temporarily powered off. In contrast to many distributed storage systems (e.g., GFS [24]), in such a case our system does not rebuild the missing replicas immediately, in order not to generate unnecessarily load on other machines nor on the network. Instead, when the unavailable peer eventually returns to the network, it efficiently updates its replicas. To inform the unavailable peers about the new version numbers of their replicas and about the contracts, we use asynchronous messaging. The control messages sent to the peer that is currently unavailable are cached at, so called, *synchro-peers*. We use the idea

of group communication for synchronizing the messages within each (small) group of synchro-peers. As opposed to Defrance et al. [16], who present the mechanism of caching the messages on routers, we chose to design the concept of synchro-peers to limit the costs of additional hardware. A synchro-peer is just an additional process running on a standard peer in our network. The load imposed by the algorithm on the synchro-peers is low as we keep the messages small; thus it is relatively "cheap" for a peer to act as a synchro-peer for many nodes.

An asynchronous message from i to j is sent to the *synchro-peers* of j. Synchro-peers are a set of peers, defined for every peer j (j is called in this context a *target peer*) that keep asynchronous messages for j. Synchro-peers of j include j, so every message will be delivered to the target peer by the same means as it is delivered to the other synchro-peers. Each synchro-peer periodically tries to send the asynchronous message to the synchro-peers that have not yet received the message; the IDs of synchro-peers that have not yet received the message are attached to the message (thus, the same message can be delivered multiple times to the same peer). These updates achieve eventual consistency.

The consecutive messages between any two peers are versioned with sequential numbers (logical clocks). If any synchro-peer k has not managed to send a message $m_1(i \to j)$ to all the requested synchro-peers before receiving a subsequent message $m_2(i \to j)$ with a higher version number, then the synchro-peer drops $m_1(i \to j)$, as the new message m_2 contains a superset of chunks' version numbers. Thus, the expected number of messages that are waiting for delivery on a single synchro-peer is bounded by $|R(\cdot)| \cdot |S(\cdot)|$, where $|R(\cdot)|$ is the average number of replicators per peer and $|S(\cdot)|$ is the number of synchro-peers per peer. The group of synchro-peers is small (5 machines in our experiments), the messages contain only the version numbers of the data chunks (thus, the messages are small as well), and the old undelivered messages can be safely replaced with the newer versions of the messages. As a result, the mechanism of asynchronous messaging is affordable from the perspective of the system.

The target peer executes the commands from the messages immediately after the first reception, but it remembers for each sender the latest version number of the received message. This information protects against multiple execution of the orders from a single message. Figure 3 shows how synchro-peers deliver asynchronous messages.

The mechanism of versioning through asynchronous messages reduces the amount of information replicators keep regarding the structure of replication contracts. In an alternative solution, replicators synchronize directly between each other. However, this requires replicators to know IDs of all the other replicators for each data chunk they store. If this information is stored at the replicators, changes in replication contracts require multiple updates; if it is stored as meta-data, the size of the meta-data becomes proportional to the number of data chunks in the system. Moreover, exchanging the messages between all replicators is highly inefficient. With asynchronous messages, the owner keeps the information about its replication contracts: updates are simple

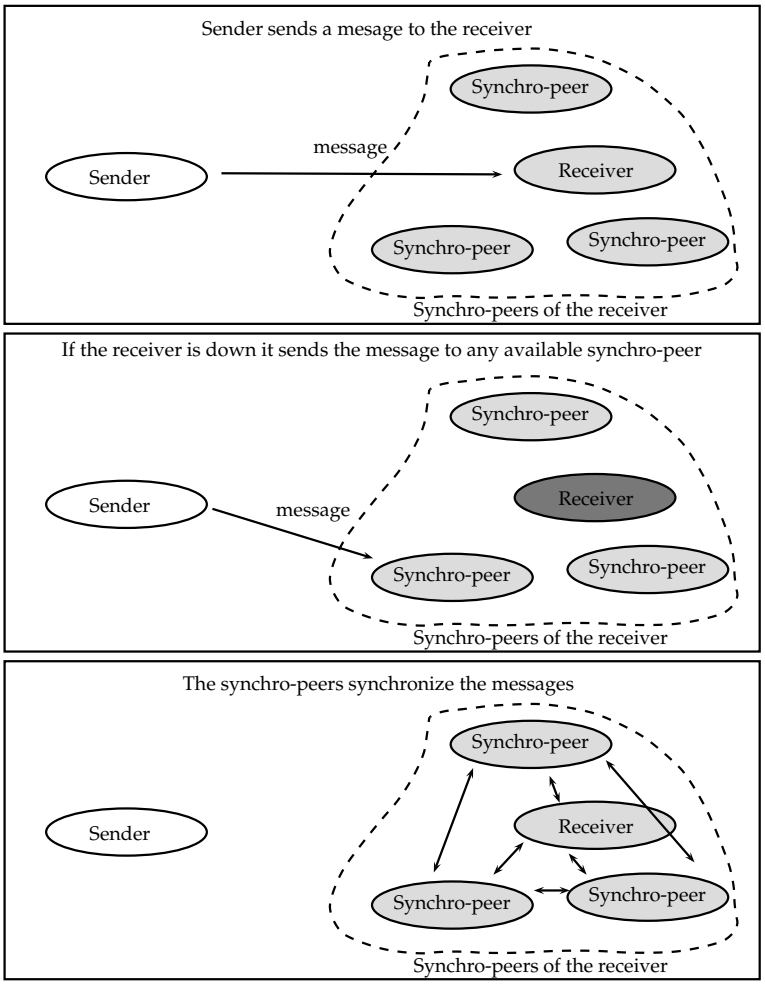

Figure 3: The diagram depicting the use case of synchro-peers delivering an asynchronous message.

and meta-data is small.

Using asynchronous messaging has two advantages. First, the asynchronous message is delivered with high probability even when the sender is unavailable. Second, the data may be downloaded concurrently from multiple replicators.

4 Replica Placement

The goal of the replica placement policy is to find and dynamically adapt the locations of the replicas in response to changing conditions (new peers joining, permanent failures, changing characteristics of existing replicators). Finding possible locations for replicas is not trivial given that peers differ in availability and amounts of free disk space. Moreover, the replica placement policy should take advantage of peers' heterogeneity in terms of availability, geographic locations, etc.

Our policy consists of two main parts. First, a utility function (in short *utility*, Section 4.1) scores and compares replica placements. Utility is a function that for a given data owner and a set of possible replicators returns the score proportional to expected quality of replicating data. Second, a protocol (Section 4.2) manages replica placement in the network in order to maximize the utility of the currently worst placement (max min optimization).

The decisions taken by the policy result in changes of replication contracts—Section 4.3 shows protocols that handle these changes.

There are many alternative replica placement policies proposed (see Section 2). Our architecture allows to implement some of them—these alternative implementations would replace the utility function and the distributed optimization protocol described in the next two subsections, but will use the same protocol for changing replication contracts (Section 4.3). For instance, a friend-to-friend storage system [49,52,55] requires a utility function that scores "friends", i.e., nodes controlled by trusted users, higher than non-"friends"; and a simple distributed optimization protocol that, for each node, finds its friends using, e.g., a DHT that resolves the identity of a friend to the address of a node she controls (here, the optimization is not a problem as the relation of "friendship" is mutual). As another example, consider a system using competitive strategies [47]. In a direct implementation, the utility function is simply the product of replicators' unavailabilities (unavailability is the probability that a replicator is unavailable in any time moment; [47] discusses how to measure that data, which then can be kept in a DHT). [47] also proposes a distributed heuristic to create stable replication agreements (called Explicit Cliques). Basically, a peer with a free storage slot tentatively chooses a partner with a similar unavailability and then proposes a replication agreement; the partner accepts it if it has a free storage slot, or if the requestor's unavailability is closer to the partner's than any other existing replicator. This heuristic can be directly implemented in our architecture: each peer simply executes the heuristic instead of the distributed optimization protocol from Section 4.2. Further, the heuristic can be implemented even using our distributed optimization protocol from

Section 4.2 by taking the utility u of peer i having replica at peer j to be $u = -|p_{av}(i) - p_{av}(j)|$, where $p_{av}(i)$ and $p_{av}(j)$ are availabilities of peers i and j, respectively.

4.1 Utility Function

In this section we describe our example utility function that takes into account peers' availability, bandwidth and geographic distribution. The notation used in this and the following section is introduced when necessary, and summarized in Table 1.

A user of a backup application optimizes the resiliency level of her data (defined by the desired number of replicas N_r and their proper geographic distribution); and the time needed to retrieve the data in case the local copy is lost (expressed as the desired data read time $Des(T_r)$). Additionally, each user must be able to backup her data (propagate the local updates to replicas) during the time the user is online (expressed as the backup window $Des(T_b)$).

Optimizing the backup time of the data is one of the key targets of backup systems. If backup is slower than data modification, then, in case of a failure, the data cannot be retrieved. A short backup window also helps to maintain the system. Therefore, many benchmarks comparing secondary storage systems focus on their throughputs as one of the primary metric. Similarly, many commercial storage systems advertise themselves as high-throughput [21,61].

Every peer j dedicates bandwidth B_j to the background backup activities (B_j is upper bounded by network and disk bandwidth, but can be further reduced by the user).

Utility function $U : \mathcal{P} \to \mathcal{R}$ is a scoring function mapping a replica placement $P_k \in \mathcal{P}$ to its score $u_k = U(P_k)$. Hereinafter, by $R(d_k)$ we denote the set of replicators of data chunk d_k.

The utility function is a sum of utilities expressing geographic distribution U_{geo}, backup time (performance) U_{perf}, and the number of replicas $|R(d_k)|$ (with the latter two treated essentially as constraints):

$$U(P_k) = U_{geo}(P_k) - L \cdot ||R(d_k)| - N_r| - M \cdot U_{perf}(P_k), \tag{1}$$

where M and L are (large) scaling factors. L penalizes for insufficient number of replicas. M penalizes for backups that cannot be finished within the time window. If the backup cannot be done on time, we cannot give resiliency guarantees for some data, thus even good geographic distribution properties are useless.

The performance U_{perf} is computed as follows. The average duration of data backup to replicator j, $E(T_b, j)$ is estimated by:

$$E(T_b, j) = \frac{\sum_{k:j \in R(d_k)} size(d_k)}{p_{av}(j) B_j} \tag{2}$$

The backup duration is proportional to the congestion on the receiving peer $\sum_{k:j \in R(d_k)} size(d_k)$; and inversely proportional to the bandwidth B_j that peer

Table 1: The summary of the notation used in Sections 4.1 and 4.2.

Symbol	Meaning
N_r	desired number of replicas
$Des(T_r)$	desired data read time
$Des(T_b)$	backup window (desired data backup time)
B_j	bandwidth that the j-th peer dedicates to backup activities
$R(d_k)$	replicators of data chunk d_k
P_k	replica placement of data chunk d_k (replicators and data owner)
$u_k = U(P_k)$	utility of placement of replica for data chunk d_k
$size(d_k)$	size of data chunk d_k
U_{perf}	part of the utility u_k expressing backup time (performance)
U_{geo}	part of the utility u_k expressing resiliency due to geographical distribution of the replicas
L, M	weights denoting the impact of $\|\|R(d_k)\| - N_r\|$ and U_{perf} on u_k, respectively
$E(T_b, j)$	average backup time from the considered peer to the j-th peer
$E(T_r, j)$	average restore time from the j-th peer to the considered peer
$remote_{min}$	minimal required TTL distance between considered peer and most distant replicator
$remote_{max}$	maximal required TTL distance between considered peer and most distant replicator
$close_{max}$	maximal required TTL distance between considered peer and all but most distant replicator
$jmax$	replicator from $R(d_k)$ that is most distant to the considered peer
$dist_{TTL}(j)$	TTL distance between the replicator j and considered peer
N	estimated number of peers in the network
α	desired number of messages that peer wants to get in a time unit without being overloaded

j dedicates for the background backup activities. We use a simplified model that does not explicitly consider the network congestion, but this issue is addressed in the further part of the utility function, described in this section. As a successful write on peer j is possible only when j is available, we multiply the bandwidth by the availability $p_{av}(j)$. Assuming that the restoring operation has no priority over the backup, the average data restoration time $E(T_r, j)$ is computed in the same way.

The backup time penalty $U_{perf}(P_k)$ is the sum over the utilities per replica location:

$$U_{perf}(P_k) = \sum_{j \in R(d_k)} U_{perf}(j),$$

where $U_{perf}(j)$ penalizes for insufficient backup window on j-th replicator:

$$U_{perf}(j) = \min(Des(T_b) - E(T_b, j), 0) + \min(Des(T_r) - E(T_r, j), 0).$$

To compute U_{geo}, we approximate the geographical distribution of the data by TTL values. Most of the replicas should be near the owner to reduce the network usage; $close_{max}$ denotes the desired distance for the "near" replicas. However, to cope with geographically correlated disasters, one replica should be far: its distance should be between $remote_{min}$ and $remote_{max}$.

The geographic utility $U_{geo}(P_k)$ considers both "near" and "far" replicas:

$$U_{geo}(P_k) = \min(0, dist_{TTL}(j_{max}) - remote_{min}) + \min(0, remote_{max} - dist_{TTL}(j_{max})) + \\ + \sum_{j \in P_k - \{j_{max}\}} \min(0, close_{max} - dist_{TTL}(j))$$

where j_{max} denotes the replicator from $R(d_k)$ that is most distant to the data owner, and $dist_{TTL}(j)$ denotes the TTL distance between replica j and the data owner.

In an enterprise backup system, we assume that all data chunks are equally valuable. Thus, the utility of the whole system is the utility of the worst placement ($\max \min_k u_k = \max \min_k U(P_k)$).

4.2 Distributed Optimization Protocol

When designing our system, we have considered several approaches for maximizing system utility $\max \min_k u_k$. Perhaps the most straightforward idea is that each peer optimes its own utility u_k by deciding with whom to form replication contracts [13,47]. Game-theoretic strategies would give the system extra protection against malicious spammers. However, they have the following drawbacks: (i) every peer has to compete with the other participants; in effect, peers with low availability or bandwidth could never achieve satisfactory replication; (ii) even if contracts for low-utility peers are accepted at the cost of rejecting the contracts of the high-utility peers, such frequent contracts rejections would result in protocol inefficiencies. Considering these drawbacks, and taking into account that in a single enterprise peers should cooperate to achieve the social optimum, we decided to turn to a optimization (not in the game-theoretic sense), proactive approach described below.

Peers share information on low utility data chunks in a distributed priority queue (described later). A peer i with free storage space periodically chooses a data chunk d_k with low utility, and proposes a new replication agreement to the data chunk's owner o. The owner either tentatively adds i to its replication set $R(d_k)$ (if the number of replicas $|R(d_k)|$ is lower than the desired resiliency level N_r); or tentatively replaces $j \in R(d_k)$, one of its current replicas, with i (all possible $j \in R(d_k)$ are tested). If the resulting utility $U(P'_k)$ is significantly higher than the current value $U(P_k)$, the owner o tries to change the contracts (see the next section). In our experiments, we require $U(P'_k)$ to be higher than $U(P_k)$ by at least 10% to reduce data movements that do not significantly improve data distribution. When the owner rejects the proposition, or when it is unavailable, i puts o in a (temporary) taboo list in order to avoid bothering it later with the same proposal.

As a result of continuous corrections of replica placements, each peer can end up having replicas of different chunks at different peers. The overhead of replicas' monitoring, however, does not depend on the number of replicas, as the monitored information (the availability and the size of replicated data) are

gossiped. On the other hand, storage contracts with multiple peers allow to parallelize data transfer, and to amortize the cost of rebuilding replicas.

If every peer proposed storage for the owner of the data with the lowest utility, the owner would get overloaded with storage offers (and the remaining data chunks would be ignored). Therefore, each peer sends a message to o with probability p_p such that $p_p = \frac{\alpha}{N}$, where N is the estimated number of peers in the network, and α is the desired number of messages that a peer wants to get in a time unit without being overloaded. Given such probability, the expected value of the number of messages, E_m, the owner of data gets in a time unit is: $E_m = p_p \cdot N = \alpha$.

The system keeps the data chunks with the lowest utility in a distributed priority queue. In our prototype, we implemented the distributed priority queue by a gossip-based protocol. Each peer keeps a fixed number of data pieces with the lowest priorities. It updates this information with its own data pieces and distributes the information to randomly chosen peers.

4.3 Changing Replication Contracts

The contracts in our system are continuously renegotiated. Each such change cannot result in data migration not to overloaded the machines nor the network. Protocols for efficient changes of the contracts are described below.

4.3.1 Finding the Best Replicators

We start by describing two aspects of the protocol: revocation of inefficient contracts; and recovery from transient failures.

When peer i offers its storage to data owner o, and when o decides that i should replace one of its existing replicators j (as the resulting value of the utility function U is higher), then o has to explicitly revoke the contract with j. Thus, changing location of the data of o, from i to j, requires these three peers being on-line. The example below illustrates why revocation of the contracts cannot be realized asynchronously.

Example 1 *Consider peer j storing many data chunks of several owners. From the perspective of each owner, as j is comparably overloaded, any new peer joining the network is a better replicator than j. If the contracts could be revoked asynchronously, all the peers would revoke the contract on j during its unavailability. Now j, having no data, can take over all data stored at some other peer k during k's unavailability, by offering storage space to all the data owners replicating their data at k. Such situation can repeat indefinitely. Each peer is not aware that j has already revoked some of its contracts and that it is not overloaded any more.*

However, if the existing replicator j has low availability, on-line revocation of its contract is also improbable. Thus, an owner can revoke a contract with such a low-available replica (e.g., the first decile of the population) also through an asynchronous message.

Additionally, since the peers are unreliable, the process of contract negotiation can break at any moment leading to inconsistency of the contracts. Two types of inconsistency are possible: an owner o believes j is its replicator, while j is not aware of such contract; or a peer j believes to be o's replicator, while o thinks j is not. The inconsistent contracts are detected by periodically exchanged messages and fixed by adopting the owner's or the replicator's state.

4.3.2 Committing Contracts and Transferring Data

In order to reduce the load on the network, replicators cannot change too often; but to maintain high performance, replicators must eventually follow the negotiated contracts. A *non-committed* contract between an owner and a replicator is negotiated, but no data has been transferred. Contracts are committed periodically. For each data chunk, if there is a new contract (negotiated, but not committed), the contract is committed when the time that passed since the last committed contract for this chunk is large enough (e.g., 24 hours). This guarantees that the data is transferred at most once in each time period (e.g., at most once every 24 hours). However, even when (non-committed) contracts change often, data is replicated (as the committed contracts represent a snapshot of utility optimization).

After committing a contract, the owner sends a message to the new replicator that requests data transfer. As soon as the new replicator downloads requested chunk, it sends an acknowledgment to the owner. Finally, the owner notifies the old location to remove the chunk.

5 Experimental Evaluation of the Prototype

5.1 Experimental Environment

We performed the experiments in two environments: (i) on the computers in the faculty's student computer labs; and (ii) in PlanetLab [9]. The aim of choosing these two environments was not to compare them, but rather to show how our system uses and reacts to different degrees of heterogeneity and different problems present in these systems: geographical decentralization in PlanetLab and low availability in student labs. We run the prototype software for over 4 weeks in the labs and 3 weeks in PlanetLab. Each computer acted as a full peer: owned some data and acted as a replicator.

We used randomly generated data for backup. At the beginning of each day we re-generated all the data (simulating that all users modify all their files). Thus, each day we expected the system to perform a complete backup. If the transfer of a particular data chunk did not succeed within a day, the following day we transferred a newer version of the chunk. In both test environments we used chunks of equal sizes (50MB).

The computers were centrally monitored; the central monitoring server experienced several failures which slightly influenced the presented results (the real backup times are slightly shorter than presented).

5.1.1 Students Computer Labs

We run our prototype software on all 150 machines of the students computer labs. The availability pattern might be considered as a worst case scenario for an enterprise setting. The labs are open each week from Monday to Friday between 8:30am and 8pm and on Saturdays between 9am and 2:30pm. The students frequently (i) switch off or (ii) reboot machines; each day at 8pm the computers are (iii) switched off by the administrators (the machines are not automatically powered on the next day). Our prototype considers each of these events as a transient failure.

The computers in the students labs have very low average availability (the median is equal to 13%). Figure 5 presents the distribution of the availabilities of the computers in the labs. Figure 6 presents the distribution of the up time of the computers and the time between their consecutive availability periods within a single day (the nights are filtered out). Low availability coupled with long session times constitute a worst-case scenario for a backup application: in contrast to short, frequent sessions, here machines are rather switched on for a day, then switched off when the labs close, and then remain off for a long time.

The amount of local data was sampled from the distribution of storage space used by the students on their home directories. The students in our faculty are divided into 3 groups and each student is assigned a quota that depends on his or her group affiliation. The distribution of data sizes for the three groups are presented in Figure 4. We took the distribution of the sizes for the group with the highest quota and scaled this distribution so that the average value was 3GB. Thus the sizes of local data were varying approximately between 0 and 8GB.

The local storage space depended on machines' local hard disks; and varied between 10GB (50% machines), 20GB (10% machines), and 40GB (40% machines).

5.1.2 PlanetLab

We conducted another experiment on PlanetLab using 50 machines scattered around Europe. Each machine had 10GB of storage space and 1GB of local data intended to be backed up. The machines were almost continuously available (the median availability is equal to 0.91).

5.2 Asynchronous Messages

In this subsection we present how the asynchronous messaging influences message delivery time and the probability that the message is delivered. For our analyzes we used traces of availability from the student labs, as the availability there is much lower than in PlanetLab. We varied the number of synchro-peers per peer between 0 and 30. For each number of synchro-peers, we generated 100,000 messages with a randomized source, destination and the sending time. Figures 7 and 8 show the results.

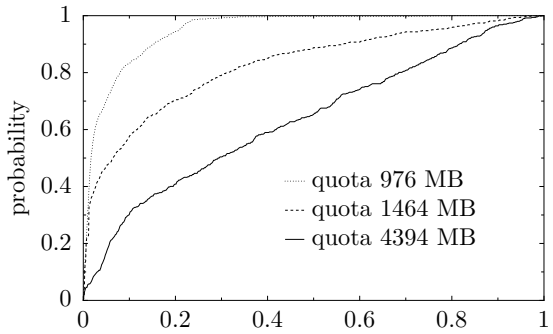

Figure 4: The cumulative distribution functions (CDFs) of the normalized sizes of storage space used by the students in their home directories. The students in our faculty are divided into three groups and each student is assigned a quota that depends on his or her group affiliation. Three lines describe cumulative distribution of these three groups of students. The size of the data is presented on the horizontal coordinate. We scaled this size so that the three distributions had the same maximal value equal to 1.

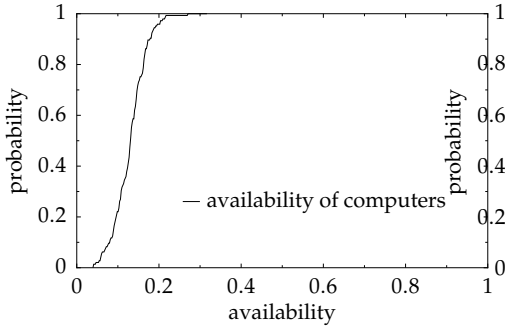

Figure 5: The cumulative distribution function of the availability of the computers in students labs.

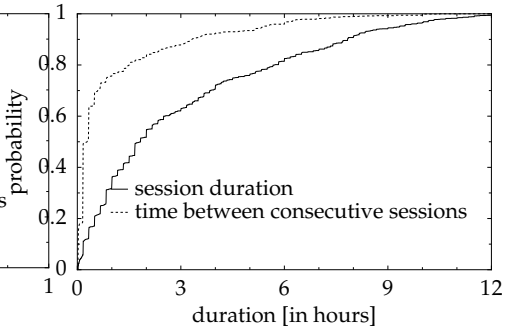

Figure 6: The cumulative distribution function of the session durations and the times between consecutive sessions for the computers in students labs (the nights are filtered out).

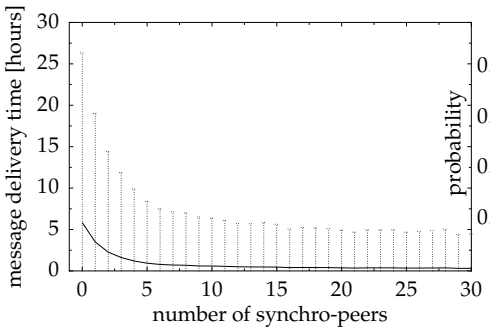

Figure 7: The dependency between the number of synchro-peers and the delivery time of the asynchronous message. Delivery time measured from the first time of availability of the receiver after the message was sent.

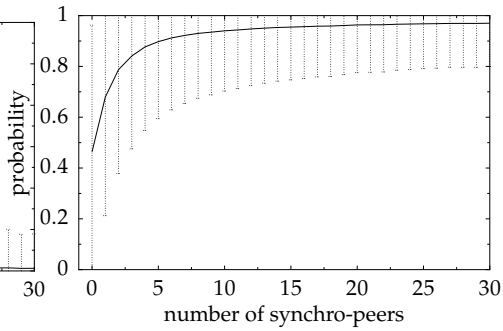

Figure 8: The dependency between the number of synchro-peers and the probability of delivery a message to any synchro-peer.

Figure 7 shows dependency between the number of synchro-peers and the delivery time of a message. Because the delivery can be accomplished only when the receiver is active, we present delivery time measured starting from the first online appearance of the receiver after the message was sent. Ideally, the message should be delivered just after the receiver becomes online, resulting in short delivery time. Our results show that the delivery time decreases significantly when using synchro-peers. Additionally, the standard deviation decreases even more significantly (high standard deviations are caused by peers that have low availabilities). If the number of synchro-peers is higher than 5, the advantage of using more of them becomes less significant. Taking into account that the higher number of synchro-peers results in higher number of messages required for synchronization, we decided to use 5 synchro-peers in the remaining experiments.

Figure 8 shows dependency between the number of synchro-peers and the probability of a successful delivery of a message to any synchro-peer. We are interested in calculating such probability because a message delivered to a synchro-peer is, in fact, a replica of the original message. Thus, synchro-peers should enable message delivery even in case of a long term absence of the sender (e.g., caused by a non-transient failure). The results show that synchro-peers significantly increase this probability: with 5 synchro-peers the system delivers 90% of the messages, while without synchro-peers, more than half of the messages are lost.

Table 2: The utility, i.e., the ratio of total size of replicated data (in MB) to the availability for the first 3 days of experiments in the students' lab environment.

	Utility (weighted replicated data)	
day	average	std dev
1	34487	6086
2	60489	8141
3	69658	5496

5.3 Replica Placement

5.3.1 Student Labs

The goal of the tests on the student labs was to verify how the system copes with low availability of the machines. For each machine i, we set the bandwidth B_i to the same value and the backup window $Des(T_b)$ to 0. As all the machines are in the same local network, there is no geographical distribution of the data. Thus, the utility function (Eq. 1) degrades to the number of replicas and the backup duration (Eq. 2). This means that the system minimizes the maximal time required for transferring a data chunk, which means minimizing the load on the maximally loaded machine. As a result, we expected the machines to be loaded proportionally to their availabilities. By Eq. 2, the load on the machine is proportional to the size of data it replicates; thus, for each machine, the total size of replicated data should be proportional to machine's availability. Additionally, storage constraints should influence the amount of data stored.

During the first 3 days of experiments we measured the ratio of the total size of data replicated by a peer (in MB) to the peer availability. For each day we considered only the peers that were switched on at least once. We also restricted the measurements only to peers with at least 9GB storage space (that could accommodate, on the average, 3 replicas), to separate the effect of insufficient storage space. The average values and the standard deviations of the ratios for the 3 days are presented in Table 2. The standard deviation is low in comparison to the average (the deviations are 18%, 13% and 8% of the corresponding average) which shows that the replicas were distributed according to our expectations.

5.3.2 PlanetLab

The goal of the PlanetLab tests was to verify how our system handles geographic distribution and heterogeneity of the machines. In this environment we required the far replica, i.e., the one implementing geographic dispersion, to have TTL distance from the owner in range $\langle 3, 8\rangle$ ($remote_{min} = 3$ and $remote_{max} = 8$) and other replicas to be as close to the owner as possible ($close_{max} = 0$). Additionally we set the bandwidth B_i to 500 KB/s for half of the machines, and 1000 KB/s for the other half. We also set the backup window $Des(T_b)$ to 4500s. Each machine had the same amount of local data (1GB); the disk space limit was

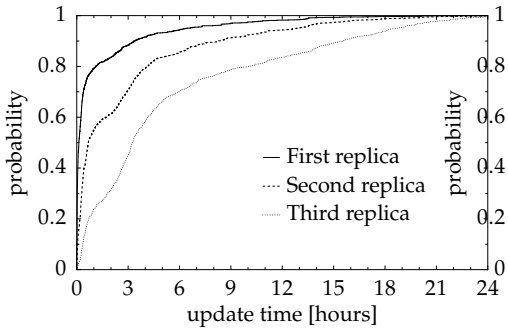

Figure 9: The cumulative distribution function for the time of creating a replica of data chunk. The time is relative to the data owner. Lab; data collected over 4 weeks of running time.

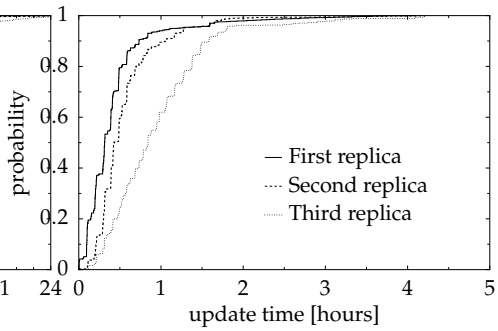

Figure 10: The cumulative distribution function for the time of creating a replica. The time is relative to the data owner. PlanetLab; data collected over 3 days.

4GB. We expected that the low-bandwidth machines will be less loaded than the high-bandwidth ones. Assuming that machines are continuously available, a low-bandwidth machine should replicate at most 2.25GB; and a high-bandwidth machine at most 4.5GB.

We tested two parameter settings that differed by the weight assigned to geographical distribution of replicas (see Section 4.1). For $M = 1$ (which means increasing the backup duration of a single chunk by 1s is equally unwanted as increasing the TTL distance of this chunk by 1), the average TTL distance between the replica and the owner was equal to 11.6 (with standard deviation of 3.7). In this case only two machines exceeded their backup window (by at most 108 s). For $M = 0.01$ (which means increasing the backup duration of a single chunk by 100s is as bad as increasing the TTL distance of a single chunk by 1), the replicas were geographically closer with mean TTL 8.1 (with standard deviation of 3.8). However, the backup duration was increased—13 machines exceeded their backup window. The average excess of the backup window was equal to 222s (5% of the backup window) and the maximal 415s (9% of the backup window).

5.4 Duration of Backup of a Data Chunk

We measured the time needed to achieve the consecutive redundancy levels (the number of replicas) for each data chunk. The time is measured relative to the data chunk owner online time: we multiplied the absolute time by the owner's availability. We consider the relative time as a more fair measure because: (i) the transfer to at least the first replica requires the owner to be available; (ii) data can be modified (and, thus, the amount of data for backup grows) only when the owner is available; (iii) we are able to directly compare results from

machines having different availabilities.

The distribution of time needed to achieve the consecutive redundancy levels is presented in Figure 9 (lab) and Figure 10 (PlanetLab).

5.4.1 Student Labs

The average time of creating the first, the second and the third replica of a data chunk are equal to, respectively, 1.1h, 2.7h and 5.5h (the average time needed to create a replica is equal to 3.1h). We consider these values to be satisfactory as the average relative time for transferring a single asynchronous message holding no data (message with 0 synchro-peers), calculated based on availability traces, is equal to 2.6h.

The maximal values, though, are higher: 24h, 29h and 32h. These long durations are almost entirely caused by peers' unavailability. The maximal time needed to deliver an asynchronous message with 3 synchro-peers is of the same order (21.5h, measured relatively to source online time, see Section 5.2). Moreover, if we measure only the nodes with more than 20% average availability, the times needed to create the replicas are equal to 1h, 1.6h and 3h and maximal values are equal to 12h, 18h and 20h.

5.4.2 PlanetLab

The average times needed for creating the first, the second and the third replica are equal to, respectively, 0.5h, 0.7h and 1.1h. The maximal values are equal to 4.0h, 4.2h, and 4.2h. These values are significantly better than in the student labs even though the distance between the machines is much higher and the computers in student labs are connected with a fast local network. This once again proves that the unavailability of the machines is the dominating factor influencing the backup duration.

The average time needed for creating a replica of a data chunk is equal to 0.76h (45 minutes). Let us assume that the transfer times of chunks are similar between each pair of hosts. If the three replicas are transferred sequentially then, to achieve the average backup time of a chunk equal to 0.76h, the transfer of all data (3GB) should take 1.52h. If the three replicas are transferred concurrently then, to achieve the average backup time of a chunk equal to 0.76h, the transfer of all data (3GB) should take 0.76h. In both cases we can assume that 3GB of data needs at most 1.52h for transfer. This gives an estimated backup throughput of 4.49Mb/s (PlanetLab uses standard Internet connections).

6 Discussion: Simulations versus Prototyping

During implementation and initial tests of our prototype we encountered numerous issues we did not expect: e.g., updating data catalog remotely whenever any contract is changed is highly inefficient; revoking the contracts cannot be done asynchronously; changing contracts too often is inefficient; each contract must be kept by both the data owner and the replicator and the two versions have to

be kept consistent. Thus, our principal methodological conclusion is that these problems should motivate others to verify their ideas, in addition to simulations and theoretical analysis, by constructing prototype implementations.

When designing our p2p backup system we learned that there are many reasons for which the theoretical analysis and accurate simulations of real-life complex systems are particularly difficult:

1. Complex systems might have multiple contradicting goals. We face multi-criteria optimization problems [22,23,29,59] in which single-criteria sub-problems cannot be isolated; thus, the models get complex. In our p2p backup system, to achieve resiliency to geographically-correlated disasters, we would like to keep data in remote locations. This, however, contradicts the performance-oriented goals. Similarly, to achieve good performance we should maximize the concurrency of backup operations so we would store data on as many physical machines as possible. When using non-dedicated machines, however, some machines are less reliable than others and some machines are often unavailable; using them as storage nodes might result in data unavailability or even data loss. Thus, in such environments, the requirements concerning reliability cannot be easily decoupled from performance.

2. Some features or modules may influence the solution space, i.e., the decisions taken by other modules. For instance, if a replicator has no overlapping online period with a data owner, the owner cannot directly transfer the data. However, the asynchronous messaging enables these machines to communicate and thus to use such a replicator for backup.

3. Complex systems consist of many elements. Effective resource allocation requires managing many dependent and independent resources, and involves modules such as network protocols, monitoring services, maintenance services, allocation algorithms in multiple software components, etc. Some of these mechanisms can be treated as black boxes and designed and studied independently; however, it is particularly difficult to analyze the whole system through simulation or analytical studies.

4. When designing a system, we often cannot reliably predict which modules and factors will be crucial for future performance of the real system. Consequently, the intermediate simulation or analytical models might miss important features. Many existing studies on p2p file replication focus on replica placement; we found out that other factors are also important (such as available communication primitives, data organization, or the protocols for creating replication agreements).

Thus, we argue that models of complex systems should be periodically validated through experiments on real systems. The architecture proposed in this work enables such studies for research on p2p data storage and backup.

In this paper we proved that P2P backup solutions can be effectively used in small and medium enterprises (enterprises with up to 150 employees). In the

light of the described experiences, the scalability of P2P backup solutions in enterprise environment requires some discussion. We found that the potential scalability barrier is the availability of the machines. This is especially valid as large companies usually have offices in different time zones. Thus, we believe it is an interesting problem to find availability patterns in large companies or other organizations (starting with [5]). In case of large amounts of data begin backed up, the throughput of the network might also become a limiting factor (especially when combined with low availability of the machines). On the other hand, our control and synchronization mechanisms are based on DHT and gossiping mechanisms, which scalability was often discussed and proved to be efficient (see [14,17,25,46] and references inside).

7 Conclusions and Perspectives

We present an architecture of a p2p backup system based on pair-wise replication contracts. In contrast to storing the data in a DHT, in our approach the placement can be optimized to a specific network topology, which allows to take into account e.g. geographical dispersion of the nodes.

We have implemented a prototype and tested it on 150 computers in our faculty and 50 computers in PlanetLab.

Our most important practical result is that the backup time increases significantly if machines are weakly-available: from 0.76h for nearly always-available PlanetLab nodes to 3.1h for our lab with just 13% average availability. This *cost of unavailability* makes some environments less suitable for p2p backup. The irregular environments negatively influence the maximal durations of data transfer. Choosing machines with better availability strongly reduces this effect (for instance, by restricting our lab environment to machines with more than 20% availability, the average backup time decreases from 3.1h to 1.9h). Moreover, in enterprise environments such irregular availabilities should not be the case. There, however, the machines may have their specific, regular availability patterns. In such case it may be valuable to use more sophisticated availability models.

A backup system does not need to support a full filesystem interface, which makes it easier to implement. We are currently working on nebulostore (`nebulostore.org`) — a more complex distributed storage system suitable for Internet applications, such as Distributed Online Social Networks. Through a more expressive API, nebulostore will allow, e.g., several writers to append data to an object; at the same time, we plan to keep the API simple enough not to implement a complete distributed filesystem.

In this work, we assumed that we do not use dedicated resources for backup. However, our approach can be naturally extended to dedicated resources (either physical machines, or IaaS cloud shares). Our pair-wise contracts can make an efficient use of these, potentially expensive, resources: for instance, only the most important data would have contracts with the dedicated resources.

The architecture proposed in this work allows to implement other placement

strategies (and, to some extent, other architectures, such as a simple DHT-based storage). Thus, one of the possible directions of future work is a quantitative comparison of these strategies on real environments.

Our results show that it is possible to build an efficient and reliable backup system, even using weakly-available machines having irregular session times. Enterprise environments are heterogeneous. A p2p backup system can hide the "harmful" heterogeneity (such as different bandwidths, disk spaces and availabilities); at the same time taking advantage of the "helpful" heterogeneity, including the geographic dispersion and the network topology.

Acknowledgements: Krzysztof Rzadca thanks Sonja Buchegger and Gunnar Kreitz for their help in reshaping an early version of the manuscript. This research has been partly supported by Polish National Science Center grants: Preludium (UMO-2013/09/N/ST6/03661) and Sonata (UMO-2012/07/D/ST6/02440).

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[2] B. Amann, B. Elser, Y. Houri, and T. Fuhrmann. Igorfs: A distributed p2p file system. In *Proceedings of P2P-2008*, pages 77–78, 2008.

[3] S. Bernard and F. Le Fessant. Optimizing peer-to-peer backup using lifetime estimations. In *Proceedings of EDBT/ICDT Workshops-2009*, pages 26–33, 2009.

[4] R. Bhagwan, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer storage systems. In *Proceedings of Future Directions in Distributed Computing-2003*, pages 153–158, 2002.

[5] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 28, pages 34–43. ACM, 2000.

[6] W. J. Bolosky, J. R. Douceur, and J. Howell. The Farsite project: a retrospective. *ACM SIGOPS Operating Systems Review*, 41:17–26, 2007.

[7] J. Busca, F. Picconi, and P. Sens. Pastis: A highly-scalable multi-user peer-to-peer file system. In *Proceedings of Euro-Par-2005*, pages 1173–1182, 2005.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26:4:1–4:26, 2008.

[9] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[10] B. G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of NSDI-2006*, volume 6, pages 4–4, 2006.

[11] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pages 46–66, 2001.

[12] www.cleversafe.com/.

[13] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of ACM-SOSP-2003*, pages 120–132, 2003.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. 35(5):202–215, Oct. 2001.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008.

[16] S. Defrance, A. Kermarrec, E. L. Merrer, N. L. Scouarnec, G. Straub, and A. van Kempen. Efficient peer-to-peer backup services through buffering at the edge. In *Proceedings of P2P-2011*, pages 142–151, 2011.

[17] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[18] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of FAST-2011*, pages 2–2, 2011.

[19] J. R. Douceur and J. Howell. Byzantine fault isolation in the Farsite distributed file system. In *Proceedings of IPTPS-2006*, 2006.

[20] J. R. Douceur and R. P. Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In *Proceedings of DISC-2001*, volume 2180, pages 48–62, 2001.

[21] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *Proceedings of FAST-2009*, pages 197–210, 2009.

[22] P. F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Proceedings of IPDPS-2004*, pages 125–132, 2004.

[23] M. Ehrgott. Approximation algorithms for combinatorial multicriteria optimization problems. *International Transactions in Operational Research*, 7:2000, 2000.

[24] S. Ghemawat, H. Gobioff, and S. T. Leung. The Google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43, 2003.

[25] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In M. Kaashoek and I. Stoica, editors, *Peer-to-Peer Systems II*, volume 2735 of *Lecture Notes in Computer Science*, pages 160–169. Springer Berlin Heidelberg, 2003.

[26] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of PODC-2002*, pages 213–222, 2002.

[27] M. Jelasity and Ö. Babaoğlu. T-man: Gossip-based overlay topology management. In *Engineering Self-Organising Systems*, pages 1–15. Springer, 2006.

[28] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10:3–25, 1992.

[29] M. M. Kostreva, W. Ogryczak, and A. Wierzbicki. Equitable aggregations and multiple criteria analysis. *European Journal of Operational Research*, 158(2):362–377, 2004.

[30] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: an architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35:190–201, November 2000.

[31] P. Linga, I. Gupta, and K. Birman. A churn-resistant peer-to-peer web caching system. In *Proceedings of SSRS-2003*, pages 1–10, 2003.

[32] C. H. M. Hefeeda and K. Mokhtarian. pcache: A proxy cache for peer-to-peer traffic. In *Proceedings of SIGCOMM-2008*, 2008.

[33] T. Mager, E. Biersack, and P. Michiardi. A measurement study of the Wuala on-line storage service. In *P2P*, Sept. 2012.

[34] R. Mahajan. How akamai works? `http://research.microsoft.com/en-us/um/people/ratul/akamai.html`.

[35] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proceedings of FAST-2011*, pages 1–1, 2011.

[36] A. Montresor, H. Meling, and Ö. Babaoğlu. Messor: Load-balancing through a swarm of autonomous agents. In *Agents and Peer-to-Peer Computing*, pages 125–137. Springer, 2003.

[37] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. *ACM SIGOPS Operating Systems Review*, 36(SI):31–44, 2002.

[38] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44:2–19, August 2010.

[39] F. E. Oggier and A. Datta. Self-repairing homomorphic codes for distributed storage systems. In *Proceedings of INFOCOM-2011*, pages 1215–1223, 2011.

[40] L. Pàmies-Juárez, P. García-López, and M. Sánchez-Artigas. Enforcing fairness in P2P storage systems using asymmetric reciprocal exchanges. In *Proceedings of P2P-2011*, pages 122–131, 2011.

[41] L. Pamies-Juarez, P. G. Lopez, and M. S. Artigas. Availability and redundancy in harmony: Measuring retrieval times in p2p storage systems. In *Proceedings of P2P-2011*, pages 1–10, 2010.

[42] L. Pamies-Juarez, F. E. Oggier, and A. Datta. Decentralized erasure coding for efficient data archival in distributed storage systems. In *Proceedings of ICDCN-2013*, pages 42–56, 2013.

[43] O. Papapetrou, S. Ramesh, S. Siersdorfer, and W. Nejdl. Optimizing near duplicate detection for p2p networks. In *Proceedings of P2P-2010*, pages 1–10, 2010.

[44] R. V. Renesse, K. P. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21:2003, 2001.

[45] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *Proceedings of IPTPS-2005*, pages 226–239, 2005.

[46] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems, 2001.

[47] K. Rzadca, A. Datta, and S. Buchegger. Replica placement in p2p storage: Complexity and game theoretic analyses. In *Proceedings of ICDCS-2010*, pages 599–609, 2010.

[48] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. *ACM SIGOPS Operating Systems Review*, 36(SI):15–30, Dec. 2002.

[49] R. Sharma, A. Datta, M. D. Amico, and P. Michiardi. An empirical study of availability in friend-to-friend storage systems. In *Proceedings of P2P-2011*, pages 348–351, 2011.

[50] W. Shi and Y. Mao. Performance evaluation of peer-to-peer web caching systems. *Journal of Systems and Software*, 79(5):714–726, 2006.

[51] P. Skowron and K. Rzadca. Network delay-aware load balancing in selfish and cooperative distributed systems. In *Proceedings of IPDPS Workshops*, pages 7–18, 2013.

[52] R. G. Tinedo, M. S. Artigas, and P. G. Lpez. Analysis of data availability in f2f storage systems: When correlations matter. In *Proceedings of P2P-2012*, pages 225–236, 2012.

[53] L. Toka, M. D. Amico, and P. Michiardi. Online data backup : a peer-assisted approach. In *Proceedings of P2P-2010*, pages 1–10, 2010.

[54] L. Toka, M. D. Amico, and P. Michiardi. Data transfer scheduling for p2p storage. In *Proceedings of P2P-2011*, pages 132–141, 2011.

[55] D. N. Tran, F. Chiang, and J. Li. Friendstore: Cooperative online backup using trusted nodes. In *Proceedings of SocialNet-2008*, pages 37–42, 2008.

[56] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[57] C. Walter. Kryder's Law. *Scientific American*, 293:32–33, 2005.

[58] Y. Xing, Z. Li, and Y. Dai. Peerdedupe: Insights into the peer-assisted sampling deduplication. In *Proceedings of P2P-2010*, pages 1–10, 2010.

[59] R. Yager. On ordered weighted averaging aggregation operators in multi-criteria decisionmaking. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):183–190, 1988.

[60] Z. Zhang, Q. Lian, S. Lin, W. Chen, Y. Chen, and C. Jin. Bitvault: a highly reliable distributed data retention platform. *ACM SIGOPS Operating Systems Review*, 41:27–36, 2007.

[61] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of FAST-2008*, pages 18:1–18:14, 2008.