University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Bolesław Kulbabiński
Student number: 277531

Data replication in peer-to-peer storage systems

Master's Thesis
in INFORMATICS

Supervisor
Krzysztof Rządca Ph.D.

February 2015

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W niniejszej pracy analizujemy problem replikacji danych w systemach peer-to-peer. Jako kontekst naszych badań przedstawiamy eksperymentalny system replikacji danych Nebulostore. Członkowie systemów takich jak Nebulostore powielają nawzajem swoje dane w celu zwiększenia ich dostępności. Cechują się oni wysoką różnorodnością parametrów a całe ich środowisko duża zmiennością, do której rozwiązania scentralizowane nie są w stanie efektywnie się dostosowywać. Z drugiej strony, rozwiązania zdecentralizowane często prowadzą do nieuczciwych stanów wynikowych. W pracy prezentujemy nowe podejście do organizacji replikacji danych które wciąż jest mocno zdecentralizowane ale korzysta z metadanych dostępnych globalnie poprzez centralny węzeł. W naszym odczuciu rozwiązanie to jest bliższe rzeczywistości niż dotychczas istniejące. Symulacje na róznych zbiorach danych pokazują, że jesteśmy w stanie osiągnąć zwiększoną dostępność danych nie odbierając motywacji do uczestnictwa w systemie zarówno członkom z wysoką jak i niską dostępnością.

Słowa kluczowe

systemy rozproszone, DHT, backup, zarządzanie zasobami

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

C. Organizacja systemów komputerów
C.2 Komputerowe sieci komunikacyjne
C.2.4 Systemy rozproszone

Tytuł pracy w języku polskim

Replikacja danych w systemach peer-to-peer

Abstract

In this thesis we analyze the problem of data replication in peer-to-peer storage systems. We also present Nebulostore, an experimental peer-to-peer data replication system that benefits from the results of our research. Members of systems such as Nebulostore replicate one another's data to achieve increased availability. Due to heterogeneity of peers and high churn, centralized solutions are not able to efficiently adapt to changing environment and also present some privacy issues. On the other hand, decentralized solutions lead to unfair outcomes. We present a new approach to organizing data replication that is still decentralized but makes use of some centrally-stored meta-data and, in our opinion, resembles reality more closely. Simulations on various data sets show that we are able to achieve increased availability while still providing participation incentives to members with both low and high availability.

Keywords

distributed systems, DHT, backup, resource management

Subject area (codes from Socrates-Erasmus)

Informatics, Computer Science

Topic classification

C. Computer Systems Organization
C.2 COMPUTER-COMMUNICATION NETWORKS
C.2.4 Distributed systems

Contents

Introduction

Distributed storage systems are designed to overcome issues related to traditional data stores that are organizationally-centralized. Almost all of presently-used solutions, such as cloud storage, work under the assumption that a single entity controls all the infrastructure and machines, together with software running on them. This may lead to some concerns regarding privacy or data ownership, especially when the owner of the system offers storage services to independent customers. A different approach is to make the system open to the public by allowing any machine to join and become a node capable of storing data. In exchange, such participant is able to store its own data on other peers' machines thus increasing its availability.

There are a few use cases where an alternative, distributed approach can be valuable. A group of people might like to create a free storage system that is based on the machines they currently own and has no entity that manages it. Such system may be used for sharing internal files, backup or a P2P social networks such as PeerSoN [10]. In a corporate setting, a company that has a policy forbidding to store its data on external devices might use desktop machines to create a backup overlay network. The system can leverage the patterns of availability related to working hours and time zones.

In this thesis, we analyze the problem of selecting replicas for system members in a manner that leads to a stable environment under certain assumptions. We analyze a few methods of clustering peers into replication groups, ranging from centralized algorithmic problems, most of which are computationally hard, to a completely decentralized game between nodes, which, on the other hand, does not lead to globally-optimal outcomes. In the first Chapter, we review and classify some of the existing distributed systems according to the degree of centralization. In Chapter 2 we present Nebulostore (www.nebulostore.org) — an experimental implementation of a peer-to-peer storage system, based on data replication agreements (contracts) between participants. The author of this work was an active member of the Nebulostore team and took part in design process and implementation of the system. Chapter 3 is devoted to data replication methods and is the main contribution of this thesis. We start with a discussion about system model and then present optimization problems that we prove are NP-hard. Finally, we present a new, decentralized approach of organizing data replication between peers. In Chapter 4 we evaluate our replication methods and present the results of simulations and experiments using our approach with various measures. Chapter 5 is a summary of the whole work together with some ideas for further research.

Acknowledgments: When working on Nebulostore and this thesis we were partially supported by the Foundation for Polish Science "Homing Plus" Programme (grant no. HOMING PLUS/2010-2/13) co-financed by the European Regional Development Fund (Innovative Economy Operational Programme 2007-2013).

Chapter 1

Existing systems

In this chapter, we present some of existing distributed systems that have a peer-to-peer aspect in their design. The aim of this discussion is to outline some of the successful ideas as well as problems related to decentralization. In the later presentation of Nebulostore, we will frequently refer to similarities between our prototype and existing solutions. Note that the examples presented below are not limited to storage systems.

1.1. Organizationally-centralized systems

An organizationally-centralized system exists when a single entity has some control over all machines participating in the system. A completely organizationally-centralized system is the case when the entity owns the infrastructure, machines and software running on them (e.g. cloud services). These assumptions may be relaxed when the owner produces the hardware and software but distributes it among independent entities (customers) and has no control of uptime and the quality of network connection between the nodes (e.g. Space Monkey [4] storage system). Lastly, in a more relaxed case, the creator can control only the proprietary software that is installed on clients' machines, and the source code or protocol specifications are not publicly available thus making software modifications very hard (like in Skype's [3] or Spotify's [5] cases). Usually, the system requires a specialized component that needs to be accessible at all times. A good example is a server that handles login data and is responsible for bootstrapping newcomers. In case of organizationally-centralized systems, this server is also managed by the system owner.

The main requirements of organizationally-centralized systems are to:

1. ensure optimal utilization of available computing/storage/network resources,
2. distribute data and network traffic between nodes to prevent bottlenecks

1.1.1. Google File System

One of the well-known examples of a successful distributed storage system is Google File System (GFS) [13].

GFS is designed to run on commodity hardware that may fail at any point. That is why constant monitoring of system elements is unavoidable. Designers came to a conclusion that

failures happen no matter how high the quality of used components is, so it is better to consider them a natural and expected part of the system's behavior. GFS does not differentiate between component failures and purposeful shutdowns. Upon detection of such event, it is able to quickly recover thanks to data replication. When the component becomes operational again, it is automatically incorporated into the existing network.

Similarly to most distributed storage systems, GFS uses a central master server to store lightweight meta-data and many chunkservers to store actual files. Files are divided into pieces (chunks) of a default size of 64 MB and replicated into three copies, stored on different machines. The master server's data is normally stored in memory and is also replicated and ready for fast restoration. Relatively large default chunk size suggests that the system is designed mainly for storing big, multi-gigabyte files. What is more, it is optimized in terms of long sequential reads and appends at the end of the file. These operations suit the Map-Reduce framework [12], in which the applications called reducers read and process large streams of data produced earlier by mapper applications (producer-consumer model). Modifications of parts of existing files or storing large number of small files may be inefficient.

Concurrent accesses in GFS are handled by read-write locks that protect centralized meta-data of the file as well as every directory on the file path. When creating, deleting or modifying meta-data, the system acquires the write lock for the file as well as read locks for every directory on the path. For example when deleting /dir1/dir2/file.txt GFS acquires read locks for /dir1 and /dir1/dir2 and then a write lock for /dir1/dir2/file.txt itself. Since data is replicated, some part of the system needs to be responsible for synchronizing all the copies. In GFS, the responsibility for updating or creating all replicas lies on the client's side. One of the copies is marked as a primary copy and is considered updated only when all copies become synchronized after a write operation. The client receives the address of the primary replica from the master and queries it for addresses of other replicas. GFS does not synchronize random writes to a file. It only guarantees eventual consistency, i.e. all clients will see the same state (although it may be corrupted). Only the append operation is atomic and the final offset is set by the system and returned after the operation finishes.

After a file is deleted, the space is not immediately freed. Instead, GFS marks the file as deleted by renaming it and periodically executes a simple garbage collector. Data integrity is guarded by checksums, optimized with regards to append operation. The master server maintains and periodically backups an operation log, which is a linear history of all operations. In case of failure the log can be replayed up to the last backup.

1.1.2. Space Monkey

Space Monkey [4] is an attempt to build a cloud storage service without a physical datacenter. Instead, Space Monkey lends a device that includes an external hard drive to its users for a monthly fee. The device needs to be connected to the Internet and apart from working as a regular storage unit, it joins a P2P overlay network together with other devices and a number of backup servers provided by the company. When a user saves her data on the device, backup copies are created on other connected devices. Additionally, the user is able to access her data from outside home via a web interface. The system serves data from the closest copy, which is not necessarily the one that user owns.

The approach of distributing small devices among the customers has some advantages over operating a centralized data center. It is the user, not the owner, who pays for power, potential cooling and networking infrastructure of a single device. By not having to operate an expensive data center, the cost per storage unit for the owner is significantly lower than in a traditional cloud service. On the other hand, system administrators have no control over availability or bandwidth. The authors of the system claim, however, that it would require at least half of the network to go down before any files become completely unavailable. What they mean is that Space Monkey uses erasure coding with sufficient redundancy that only half of the replicas are necessary to reconstruct the data.

1.1.3. Skype

Skype [3] is a voice-over-ip and messaging platform that uses a P2P overlay network to establish connections between users conducting real-time voice and video conversations. It uses an important concept of **super-peers**, which is a name for members of the system that have distinguishable properties, such as outstanding availability or a public IP address. In case of Skype, the company provides a number of powerful machines that are considered super-peers and can be used to incorporate a new host into the network. In the earlier stages of Skype's existence, every member that fulfilled certain requirements could become a super-peer.

Every peer maintains a *host cache* which is a local list of addresses of super-peers. At the beginning, the list is filled with a number of hard-coded IPs of servers provided by Skype. After initial connection is established, host cache is regularly updated via the process of **gossiping** with other peers. It is important that super-peers have public IP addresses because they are used as proxies in TCP hole punching process [8] if two other peers are behind NAT. For voice calls, Skype tries to use UDP where possible. When both parties are behind NAT, they communicate only with their closest super-peers, which again act as proxies. Skype also uses a centralized login server to manage identities and ensure name uniqueness.

Skype is an extremely successful product and its user base is constantly growing. Its super-peer-based architecture manages to conduct real-time calls in a very good quality. It makes use of free resources on some client machines to forward traffic that belongs to other users. Traffic is encrypted so man-in-the-middle eavesdropping is hard, but it still comes with some risk of monitoring other users' activities, such as the sole fact that a user has made a call.

1.1.4. Napster

Napster [22] was originally created as a P2P file sharing system, focusing mainly on music, typically stored as mp3 files. It consists of three components — file sharing module, search engine and a chat. In order to share files, a user has to install one of Napster clients (most of them are open-source) and connect to the Napster's central index server. The central server does not store any files, it only manages user logins and indexes all the content. User can execute a search query on the server to get a list of peers that posses a particular file. She can later connect directly to one (or more) of the peers and download the desired file. Apart from that, users can select local directories that are publicly shared and indexed by Napster's engine.

In Napster's case, the architecture of the client and communication protocols were publicly known, which led to creation of many different clients. On the other hand, there was only one central index server, operated by the inventor's company. Due to the fact that Napster's brand quickly became popular and the server's database was large and constantly growing, there was no point in creating an alternative server at that moment. A user who wanted to find a file needed to connect to Napster's network so clearly a private company had some control over its users even though the system was massively decentralized.

1.2. Organizationally-decentralized systems

Organizationally-decentralized systems are much harder to design and maintain. Apart from all the problems of ordinary distributed systems, factors such as member incentives or unpredictable churn become important.

A big advantage of an organizationally-decentralized system is potentially higher privacy protection. It is hard for one entity to control some data when it is distributed among random anonymous participants. Another advantages include increased anonymity and transparency. When software is released together with its source code it is often subject to a careful scrutiny. Thus, it is easier to convince users that the software is in fact doing what it is supposed to.

1.2.1. Bittorrent

Bittorrent [20] is an open protocol for content distribution in P2P networks. The idea is similar to Napster, although more sophisticated and independent of centralized index servers. Each file shared within the systems has an associated *torrent* file with some meta-data including a cryptographic hash and an address of the tracker. A *tracker* is a machine responsible for storing lists of peers that posses a copy of a particular file. There is a large number of publicly available trackers operated by some organizations but in fact any machine can act as a tracker.

After obtaining a torrent file, the Bittorrent client can connect to the tracker and then join the P2P network of hosts currently exchanging the file, called a **swarm**. Files are divided into chunks of fixed size. The download process is independent for each chunk and as soon as the client finishes downloading a particular chunk, it starts serving it to the other interested clients, thus reducing the load on the original sources.

1.2.2. Distributed Hash Tables

A Distributed Hash Table (DHT) is a service that provides functionality similar to a regular hash table, namely storage and retrieval of (key, value) pairs. The system is decentralized and every participating node is responsible for a piece of collective information. Data distribution among the nodes depends on key space partitioning, which is often based on some **consistent hashing** algorithm in order to facilitate adding and removing participants. Typically, every node stores a set of links to other nodes called a *routing table*, which is of small size relatively to the whole system's size N, such as $O(\log N)$. Most of DHTs also add data redundancy to account for node failures. Nebulostore uses Kademlia [19] to store meta-data (Section 2.3.1).

1.3. A classification attempt

The table below is an attempt to summarize and classify the systems we discussed in regards to which parts are owned and controlled by a single organization. The taxonomy is not perfectly accurate as not every system can be easily fitted into one of the buckets.

Systems	Central ownership of			
	central server	software	hardware	infrastructure
Most cloud services	YES	YES	YES	YES
Space Monkey	YES	YES	YES	NO
Skype, Spotify	YES	YES	NO	NO
Napster	YES	NO	NO	NO
Bittorrent, most DHTs, **Nebulostore**	NO	NO	NO	NO

By infrastructure we understand connections between internal system nodes, i.e. cables, routers, switches etc. Centralized ownership of hardware and infrastructure means that the system is deployed in a data center belonging to a particular organization. Ownership of software means that the software used by a machine participating in the system is to some extent proprietary and cannot be modified easily. Finally, by ownership of a central server we understand some dependency on a particular service existing in the Internet, which is hard to replicate. It is usually tied to some proprietary server-side software, valuable data collection, brand and also community that was created around it.

The last row of the table consists of what we consider organizationally-decentralized systems, including Nebulostore, which will be discussed in detail in the next chapter. Rather than thinking of them as physical, existing systems, we might understand them as **ideas** or **definitions** of systems that can be created within any environment and exist in many instances that are not connected to one another in any way. An example of such case might be a company that uses the Bittorrent protocol to distribute data within the internal network. This is a different approach than systems such as Skype, where everyone connects to the same network, created and to some extent controlled by a private company, Skype Technologies. One may try to replicate Skype's network by creating similar software and an alternative login server. Proprietary software or protocols is only one of the problems. The most difficult part would be to create the incentives for users to join the alternative network instead the original Skype, to market which the company has put a lot of effort and resources. Of course, using a third-party provider does not come without risks, such as potential privacy loss. This is one of the reasons why organizationally-decentralized distributed systems are valuable and have many use cases in various environments.

Chapter 2

Nebulostore — a peer-to-peer storage system

In this chapter we present Nebulostore — an experimental, peer-to-peer storage system. Initially, Nebulostore was an attempt to create an organizationally-decentralized alternative to cloud storage with a target use case of serving as a back-end part of PeerSon distributed social network [10]. During the development process we realized that there is a number of other, potentially attractive, use cases, which we discuss below. Moreover, we describe a few components of Nebulostore, which are based on standard solutions, such as Kademlia distributed hash table [19] or T-Man gossiping protocol [15].

2.1. Goals and use cases

Contemporary cloud storage services are created to abstract the infrastructure away from the user and provide her with an interface to write, read, create, delete and share her data. Typically, a single entity owns software and hardware required to ensure reliable service and charges its customers for the ability to use it within agreed limits. Apart from abstraction, the key goals of cloud storage are to:

- Increase data **availability**. In most solutions, the user can access her data at any time and, more importantly, from any place in the world with an Internet connection (as most services include a web client).
- Increase data **security**, which is a twofold issue. Firstly, the data is usually replicated among many machines thus reducing the risk of loss due to hardware or software malfunction. Secondly, large companies dedicated to data storage can afford more sophisticated protection against cyber-attacks than its customers.

Cloud storage services can be further divided into personal file hosting services, such as Apple iCloud [2] or enterprise-level services, such as Amazon S3 [1].

The Nebulostore project can be considered as an alternative to cloud storage. It builds an organizationally-decentralized, peer-to-peer overlay network of participating hosts, willing to share one another's data. Each client has to contribute some amount of disk space for storing other users' data. In exchange, her data is replicated and saved to some other clients' machines. When user loses her data and wants to retrieve it from the system, she needs to

contact one of the peers replicating the piece of her interest. Conversely, she needs to reply to requests regarding the data that she is replicating. Data replication is in some cases mutual and enforced by a notion of a **replication contract** that two users agree upon and which contains the details of how much storage space for what period of time the users are promising to provide. The details of the system design, replication and file operations will be described later. The models of data replication and mechanisms of establishing replication contracts are discussed in Chapter 3.

Nebulostore can be used as a file hosting and sharing service. If a certain level of replication is reached, we can achieve better availability of our data as our own storage is no longer a single source of it. For the same reason, the probability of permanently losing our data is decreased. Finally, public-key encryption is used to prevent unauthorized access to files. Thus, we are able to achieve availability and security, and trade cloud provider's fees for increased storage capacity requirements on clients' machines.

When we think of enterprise data storage setup, Nebulostore may utilize standard desktop machines by making them replicate each other's data. Most of the time corporate laptops or desktop computers have a significant amount of unused disk space and a reliable network connection [6]. Similarly, Nebulostore can work as a personal data hosting/sharing application, connecting peers from around the world over the Internet. In comparison to the traditional cloud solution we achieve increased anonymity and decentralization, which is important for participants who operate on sensitive data and would prefer not to replicate it only on machines fully controlled by a single entity. On the other hand, such design offers no control over participating nodes, which, for example, means that any node can permanently leave the network at any time. This fact requires solving a challenging problem of providing **incentives** for the clients to be part of the system. Another issue is trust and verification of whether the peer is really replicating what she is supposed to. Some methods to cope with these problems are discussed later.

Another potential use case for systems such as Nebulostore are distributed social networks, such as PeerSoN [10]. PeerSoN aims at offering functionality similar to successful social networks, such as Facebook or LinkedIn. These include social links between users, interest groups, digital personal space, where a user can post messages, links or pictures, channels of communication, such as instant messaging and many more. At the same time PeerSoN tries to avoid two limitations of traditional organizationally-centralized social networks — privacy issues and the requirement of constant internet connectivity. Privacy protection is enhanced by encryption and decentralization, similarly to Nebulostore. Internet connectivity requirement is overcome by a concept of **direct exchange**, where users can directly exchange their data, without any third-party connections. In fact, the exchange can be even done physically, mimicking the data flow in the real social network of friends or acquaintances. Nebulostore back-end part can play the key role in this process, as the replication contracts can be established in a way such that connections to unknown peers are unnecessary.

Finally, we can also imagine Nebulostore being a basis for a collaborative document editing system, similar to Google Docs although not necessarily real-time. Text files are in fact lists of lines, which fits nicely into our list model that will be presented later. Moreover, Nebulostore allows customizations of list merging mechanism, which makes it possible to handle changes in text files correctly.

2.2. Design decisions

In order to make Nebulostore robust and relatively simple we decided to make some simplifying assumptions regarding functionality and performance requirements.

First and foremost, Nebulostore does not require constant connectivity of the participating nodes. In fact, the availability is an important attribute of each peer, using which it can compete with others to receive more favorable replication conditions. The definition of availability can vary and will be discussed in Chapter 3. System's participants will try to constantly estimate the actual availability of other peers by performing simple measurements and sharing them publicly.

Nebulostore needs to maintain a database with personality/login data, cryptographic keys, persistent addressing map, existing replication contracts and peers' statistics. Each of these pieces of meta-data can be stored either in a central database or in a distributed hash table, depending on the degree of decentralization of a particular deployment.

Each user is identified by a cryptographic key named **AppKey**. A single physical user can own multiple AppKeys (and thus multiple, unrelated identities) by creating multiple accounts within the system.

There are two types of objects that can be stored in the system — **files** and **lists**. A file is a sequence of bytes that is usually divided into chunks of smaller sizes to facilitate fetching and dissemination. Files support typical operations such as creation, deletion, random read and random write. A list is an ordered sequence of objects (files or lists) that usually represents a simplified directory. Lists support creation, deletion, read and append operations. The last one adds an object to the end of the list. Additionally, the owner of the list can remove any element from the list. All objects are identified by unique numbers named **ObjectId**s but to be able to retrieve an object from the system, one must also possess the AppKey corresponding to the owner of the object.

Permissions to access objects are also simplified in Nebulostore. The owner of the object has always all permissions. Everyone with a valid (AppKey, ObjectId) pair is able to download the corresponding object from the system, however, the objects are usually encrypted. Everyone who possesses an appropriate cryptographic key is able to decrypt and read an object (a file or a list). Only the owner can modify or delete a file or a list, with the exception of the append operation which can be either disabled or allowed for everyone. Such a restricted scope of operations is usually enough for data backup service or online social network features and at the same time solves a number of problems related to concurrent modifications and inconsistency.

Finally, Nebulostore follows the eventual consistency model [24] which implies that, if no new updates are made to a given object, eventually all accesses to that object will return the last updated value. In case of lists this does not mean that every access will return *the same* value. Nebulostore considers two lists equal if they contain the same elements and the ordering of each list is consistent with the partial order of changes that were applied to the list.

2.3. Architecture and components

Nebulostore implements the actor model where actors are represented internally as job modules responsible for high-level functionalities of the system. **Job modules** are single-threaded state machines designed for completing self-contained tasks that may involve communication with one or more other peers. Job modules are spawned and managed by the **dispatcher** component. Dispatcher is responsible for managing threads and also delivering messages from network to appropriate job modules. Incoming and outgoing network traffic is served by the **communication** component. All network traffic between Nebulostore peers consists of serialized messages created and handled by job modules. Dispatcher and communication modules are considered core Nebulostore components that are always running. Other jobs are invoked on demand in a form of job modules that are usually short-lived.

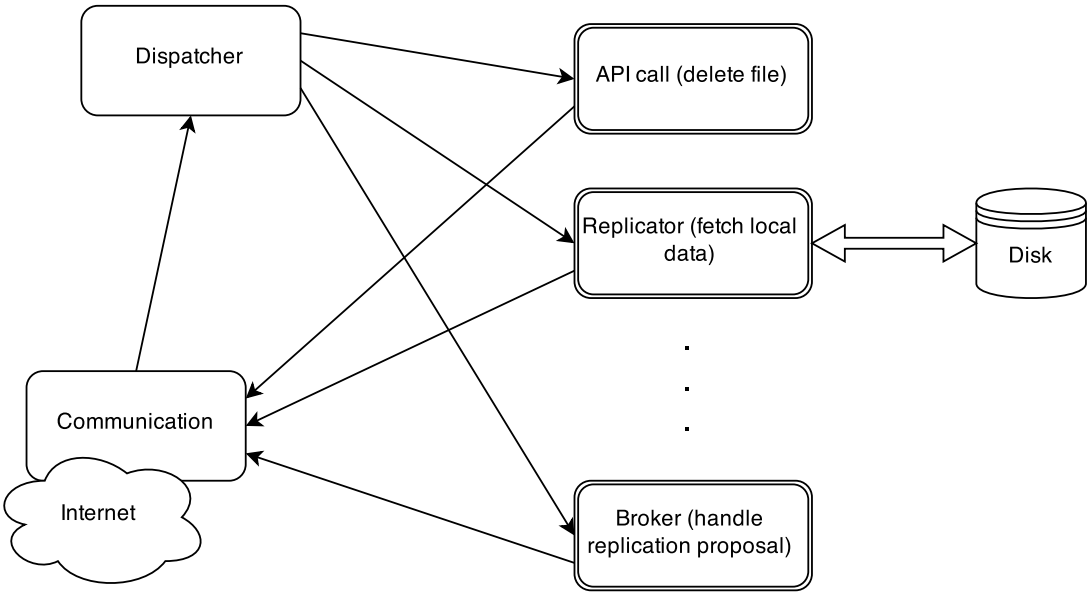

Figure 1: Simplified Nebulostore architecture diagram. Arrows show message flow between modules. Dispatcher is executing a few Job Modules (rectangles with double border).

The **communication** component of Nebulostore is responsible for sending messages between peers over the network. Another task is to manage meta-data including, but not limited to persistent addresses conversion and replication contracts. Depending on configuration, meta-data can be stored centrally, in a database instance running on a designated and globally-known host. Alternatively, it can be stored in a DHT created within the existing Nebulostore network. Communication component offers a custom implementation of Kademlia [19] DHT. Moreover, this component manages the overlay network of peers and provides discovery information to other modules. Peer discovery is realized via gossiping [16] — Nebulostore provides an implementation of T-Man gossiping protocol [15].

Broker is a component which task is to offer replication contracts and reply to such offers. Broker's goal is to optimize the availability of its owner's files by selecting the best replicas according to a given logic. Various approaches and algorithms for replication agreements are discussed in Chapter 3.

Replicator is a disk interface used when data reads or writes are necessary. It provides an abstraction of a key-value data storage and encapsulates all the subtleties related to file versioning and two-way commits.

Nebulostore implements two additional modules to facilitate communication between peers. One of them is named **network monitor**, which is responsible for probing other peers in order to find better candidates for replication. Note that such behavior is needed in a completely distributed Nebulostore setting. As an alternative, we might allow a centralized database of peers indexed by their availability parameters. The second module, namely **async**, is an implementation of the idea of asynchronous messaging. Every peer has a group of other peers called synchro-peers assigned to her. When the peer is offline, all messages are redirected to one or more of its synchro-peers and will be later delivered when the recipient becomes available again. More details about this idea can be found in [23].

Lastly, our **API modules** are responsible for handling user requests such as creating, reading, writing or deleting files or lists. Appropriate job modules are invoked in response to user interface actions, such as selecting and uploading new file to the system. Most of them use the object resolution process, which works as follows. The owner's AppKey is used to fetch her meta-data, including contract information. Afterwards, objectId is used to extract the list of peers currently replicating the target file. A few peers are queried in parallel and one of those who reply is asked to send the file. If the file consists of many chunks, they are fetched lazily, depending on the actual API call.

2.3.1. Kademlia distributed hash table

Nebulostore contains an implementation of Kademlia distributed hash table, which is used to store meta-data regarding file locations, sizes etc. Kademlia assigns random 160-bit identifiers to participating nodes and uses XOR metric to measure the distance between them. Each node stores 160 lists of neighbors, each of size at most k (Kademlia's parameter). The i-th list in node N contains links to nodes with IDs that share a common prefix of length $i-1$ with N but differs on the i-th bit. It can be proven that in order to find any node Kademlia queries only $\log(n)$ different nodes, where n is the size of the network. More details can be found in [19].

2.3.2. T-Man gossiping protocol

Gossiping is used in Nebulostore to discover nodes that are best for establishing replication agreements according to current strategy. In the T-Man protocol each peer holds a list of other nodes, called a view. The list is exchanged and modified in each iteration of gossiping. The algorithm for view selection is parameterized by three constants: C (size of the view), S (swap parameter) and H (self-healing parameter). In each iteration of gossiping, peer sends $C/2-1$ elements from its list, ignoring the H oldest ones. When it receives a list in reply, it appends it at the end of its own view. Afterwards, it removes first S elements and if the list is still longer than C, it also removes the oldest elements. The details of the algorithm can be found in [16] and [15].

Chapter 3

Data replication

In this chapter we analyze various approaches to data replication in peer-to-peer systems, categorized with respect to the degree of centralization. We begin with some general system assumptions and a discussion regarding the model of peer availability. Afterwards, we present some of the existing results and also propose a new approach.

3.1. General assumptions and system measures

We would like to make a few assumptions about our system that are going to be used throughout this work.

Uniformity. Every peer has the same amount of data that needs to be replicated. Moreover, every peer is able to replicate data of at most K other peers (K is a system-wide constant). Finally, every peer always stores its own data (i.e. works as a replica for itself).
This assumption lets us focus only on peers' availabilities as a single parameter. In case a real user needs to replicate more data, it can have multiple identities in the system.

Truthfulness. No peer advertises higher availability than it actually has.
It is relatively easy for the system to verify if a peer is actually available in a given period of time or in a given percentage of time. Dishonest peers could be removed from the system. In some scenarios peers might also benefit from advertising lower availabilities than they actually have and it is very hard to verify that. We cope with this problem by selecting appropriate measures that discourage such behavior (see Section 3.4).

Membership transparency. Everyone knows about all other peers existing in the system. We assume existence of a centralized login server, where every member of the system registers itself and can poll the server for information about peers with required availabilities.

Regardless of the availability model and other assumptions, we use a **system measure** to evaluate the state of the system from the creator's perspective. Our goal is to maximize data availability, which is easily expressed by a simple measure defined below.

Definition 3.1. *The basic system measure (BSM) is a sum of total data availabilities over all peers in the system.*

$$BSM(P) = \sum_{p \in P} av_{total}(p)$$

In the definition above, P is the set of all peers in the system and $av_{total}(p)$ is peer's data availability, i.e. a combination of its own availability and its current replicas' availabilities. Exact definition of av_{total} depends on the model chosen and is discussed later in Section 3.2.

BSM is simple and intuitive but may not fully reflect the fairness we would like to achieve. For example a scenario when peer A has the resulting availability equal to n and peer B has availability 0 is identical to when both peers have availabilities equal to n/2. The latter situation is clearly better form the system creator's perspective and we would like our measure to reflect that. [17] provides a nice and fairly simple definition of measures that are equitable, i.e. fair in the way we would like them to be. It is enough to modify BSM in the following way:

$$BSM(P) = \sum_{p \in P} s(av_{total}(p))$$

where s is any strictly-concave and increasing function, for example the square root, which we will use to exemplify the idea.

Definition 3.2. *The equitable system measure (ESM) is defined as follows*

$$ESM(P) = \sum_{p \in P} \sqrt{av_{total}(p)}$$

3.2. Availability model

Two models of peers' availabilities — the *probabilistic* model and the *time-slot* model were proposed in [21]. We provide necessary definitions and briefly discuss advantages and disadvantages of both models. Later we focus only on the time-slot model so existing results within the probabilistic model are briefly presented in the following section.

3.2.1. Probabilistic model

In the probabilistic model, each peer p_i has a probability of being online and available for every moment in time t. This can be further simplified by assuming that the probability is constant over time thus associating with each peer p_i only one real number.

Definition 3.3. *The probabilistic availability of peer p_i is a single real number $av(p_i) \in [0, 1]$.*

One important implication of this definition is **symmetry** i.e. if a given peer has probabilistic availability p, it will contribute exactly that amount to every other peer that it replicates, regardless of the other peer's characteristics.

Because of such symmetry, [21] proposed a **clique**-based data replication model. In this model, peers are divided into groups and every peer replicates data of every other member of its group. A number of arguments are given in favor of such approach but the most compelling are that firstly, the replication scheme is based on reciprocity and secondly, distribution of data can be simpler to perform.

Most centralized optimization problems based on cliques in the probabilistic model are NP-hard. On the other hand, fully decentralized approaches, where peers communicate with one another and group into replication cliques, usually lead to big imbalance and favoring

strongest peers [21].

Definition 3.4. *Total data availability of a single peer p_i belonging to a replication clique $Q = \{p_1, p_2, \ldots, p_k\}$ is equal to the probability of at least one of the peers from the group being online.*

$$av_{total}(p_i) = 1 - \prod_{p \in Q} (1 - av(p))$$

3.2.2. Time-slot model

Contrary to the probabilistic model, the time-slot model divides time into discrete intervals of equal length and assumes that each peer has a cyclically-repeating pattern of being available or not during the whole interval.

Let us formalize the time-slot model.

Definition 3.5. *For a given positive number of time slots T, the time-slot model with T time intervals consists of time period $\mathcal{T} = \{1, \ldots, T\}$ and the set of peers $\{p_1, \ldots, p_n\}$ together with their availabilities.*

Availability of a peer is not a single number any more but a set.

Definition 3.6. *In a time-slot model with time period $\mathcal{T}$, availability of peer p_i is defined as a subset of $\mathcal{T}$.*

$$av(p_i) \subseteq \mathcal{T}$$

In this work, we focus only on the time-slot model. It seems to be more interesting from the theoretical point of view as it can be used to approximate the probabilistic model. For example, if a peer is known to have the probabilistic availability of 90%, we can simulate it by assuming that it is available in 9 out of 10 intervals (chosen randomly) in the time-slot model. Moreover, this model is often equivalent to the patterns in which real machines are available. For instance, in large companies, desktop computers might be occupied from 9 AM to 5 PM and off during the rest of the day and night (from 24 one-hour slots only 8 are available).

We can observe that this model lacks the symmetry of the probabilistic model. Some arrangements of availability slots can lead to situations where peer A is worthless for peer B but very valuable for peer C (see Figure 2).

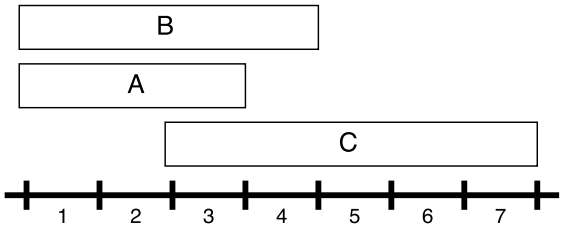

Figure 2: Example with 7 time slots where peer A covers slots 1-3, peer B covers slots 1-4 and peer C covers slots 3-7

Because of this lack of symmetry, we no longer require peers to form reciprocal cliques. Instead, every peer finds its own replicas independently.

In the time-slot model, total data availability of a single peer p is equal to the set sum of availabilities of its replicators (including itself).

Definition 3.7. *Total data availability of peer p is*

$$av_{total}(p) = av(p) \cup av(p_1) \cup \ldots \cup av(p_k)$$

where all peers $p_1 \ldots p_k$ are currently replicating p's data.

3.3. Centralized approach

Let us begin with an optimization problem proposed by [21] together with a proof of its NP-hardness, which is a contribution of this work.

An instance of the **Fair Peer Replication** (FPR) problem consists of the set of peers $\{p_1, \ldots, p_n\}$ in a time-slot model with T intervals. The optimization goal is to assign each peer to a clique, such that all the cliques are complete ($\forall G_k : A_{G_k} = \mathcal{T}$) and the size of the largest clique is minimized ($\min \max(|G_1|, \ldots, |G_N|)$).

The decision version of FPR is as follows. Given the set of peers' availabilities $\{A_i\}$, is it possible to assign each peer to exactly one clique such that all the cliques are complete and the size of the largest clique is at most n.

Theorem 3.8. *FPR is NP-hard.*

Proof. The proof is by reduction from GRAPH 3-COLORING. Given a graph G we will first construct a larger graph H that is 3-colorable if and only if G is 3-colorable. Then we will construct an instance of FPR and a number n, such that the solution for this instance with largest clique of size at most n exists if and only if H is 3-colorable.

Given a graph G with k vertices and m edges, we construct graph H as follows. For each vertex $v_i \in V(G)$ there are three corresponding vertices $v_i^1, v_i^2, v_i^3 \in V(H)$ connected by edges $(v_i^1, v_i^2); (v_i^2, v_i^3); (v_i^3, v_i^1) \in E(H)$ (we will refer to these as *joining* edges). For each edge $(v_p, v_q) \in E(G)$, there are three corresponding edges in H: $(v_p^1, v_q^1), (v_p^2, v_q^2), (v_p^3, v_q^3) \in E(H)$ (we will refer to them as *basic* edges). We also add $3k+3m$ additional vertices $v_1^A, \ldots v_{3k+3m}^A \in V(H)$ that are not incident to any edges (we will refer to them as *free* vertices). The number of these additional free vertices $\{v_i^A\}$ is equal to the number of edges in H. We may bear in mind an implicit one-to-one correspondence between $\{v_i^A\}$ and $E(H)$ as it will be used in further constructions.

Graph G is a subgraph of H induced by the set of vertices $\{v_i^1\}$ so when a 3-coloring of H is given, the 3-coloring of G can be easily extracted. Conversely, when the 3-coloring of G is given, it can be directly mapped into vertices from $\{v_i^1\}$. Colors of vertices from $\{v_i^2\}$ can be derived from $\{v_i^1\}$ by applying $(2, 3, 1)$ permutation to the color of each corresponding vertex. Similarly, colors for vertices from $\{v_i^3\}$ can be derived from $\{v_i^1\}$ by applying $(3, 1, 2)$ permutation to them. Vertices $\{v_i^A\}$ can be colored in any way because they are not incident to any edge. Thus, we get a coloring of graph H. Due to the permutations, in every triple

$\{v_i^1, v_i^2, v_i^3\}$ there is exactly one vertex of each of the three colors. Moreover, permutations preserve the correctness of 3-coloring in subgraphs induced by $\{v_i^2\}$ and $\{v_i^3\}$, so the resulting 3-coloring of graph H is correct.

Now we are going to construct an instance of FPR problem for graph H. Firstly, let us take $\mathcal{T} = \{1, \ldots, 3k+3m\}$, which is equal in size to $|E(H)|$. Each vertex of H corresponds to a single peer p_i. We derive peers' availabilities based on pairs of interconnected vertices from H. The idea is that an edge corresponds to two peers that cannot belong to the same clique. For each peer p_i we start with empty availability $A_i = \emptyset$. Then, for each edge $e_j = (v_p, v_q) \in E(H)$ (for $j = 1, 2, \ldots, 3k + 3m$) we add j to availability sets of peers corresponding to vertices v_p, v_q and v_j^A. Vertex v_j^A is the free vertex corresponding to edge e_j as defined earlier. Observe that after the completion of the above method, each time slot is covered by exactly three peers.

To complete the proof, we will show that the answer to the FPR problem with peers' availabilities $\{A_i\}$ as defined above and $n = 2k+m$ is positive if and only if graph H is 3-colorable.

Let us firstly assume that the assignment to cliques with these constraints is possible. Notice that the number of peers (or $|V(H)|$) is equal to $3k + (3k + 3m) = 6k + 3m = 3n$. This means that the solution to FPR problem contains at least 3 cliques. On the other hand, each time slot is covered by exactly 3 peers so the number of cliques is at most 3, as all the cliques need to be complete. Thus, every solution to FPR consists of exactly 3 cliques and every time slot is covered by exactly one peer in each clique. Mapping clique IDs onto vertex colors leads to a 3-coloring of graph H. Suppose that the resulting 3-coloring is incorrect, i.e. there are two vertices v_r, v_s of the same color connected by an edge e_j. This implies that peers p_r and p_s belong to the same clique. Time slot j is covered by only three peers, two of which are in the same clique, which means that one of the other cliques does not cover time slot j, which leads to a contradiction.

Conversely, let us assume that H is 3-colorable. When a 3-coloring of graph H is given, it can be modified so that all colors are distributed equally among vertices. It is indeed the case in the subgraph induced by $\{v_i^1\} \cup \{v_i^2\} \cup \{v_i^3\}$ because in every triple $\{v_i^1, v_i^2, v_i^3\}$ there is exactly one vertex of each color. We can recolor the free vertices from $\{v_i^A\}$ in the following manner. For each edge $e_j = (v_p, v_q) \in E(H)$ we assign vertex v_j^A the color that is assigned neither to v_p nor to v_q. If e_j is a basic edge, there exist three copies of it in H with incident vertices of permuted colors. These three edges will commit three pairwise different colors to vertices from $\{v_i^A\}$. Similarly, if e_j is a joining edge, it is one of three edges forming a K_3 subgraph out of vertices $\{v_i^1, v_i^2, v_i^3\}$. These three edges will also commit three pairwise different colors to vertices from $\{v_i^A\}$. The resulting 3-coloring of H can be directly mapped onto clique memberships. It will result in 3 cliques, each of size exactly $|V(H)|/3 = n$. Moreover, every clique will be complete because due to recoloring every time slot is covered by exactly three peers of pairwise different colors. Thus, we have shown that the solution to our instance of the FPR problem exists. □

The FPR problem aims at achieving a state with perfect availability and minimizes the replication overhead. In real systems, perfect availability might be impossible to achieve within reasonable resources. Moreover, we already argued that in the time-slot model, clique-based replication is not necessarily desirable. That is why we propose a different optimization problem.

An instance of the **Uniform Peer Replication** (UPR) problem consists of the set of peers $P = \{p_1, \ldots, p_n\}$ in a time-slot model with T intervals. Each peer has exactly K replication slots available. The optimization goal is to assign exactly K peers (replicas) to every peer so that each peer replicates exactly K other peers and the sum of all resulting availabilities is maximized.

When $K = 1$ the problem can be solved in polynomial time. The idea is to create a set of copies of all peers P' and solve the assignment problem between sets P and P'. The assignment problem can be solved in polynomial time for example by using the Hungarian algorithm [11]. For $K \geqslant 2$ the problem becomes a simplified version of 3-dimensional (or higher) matching, which is a well-known NP-complete problem. This argument is not sufficient to construct a proof. However, we believe that the UPR problem for $K \geqslant 2$ is NP-hard as well.

3.4. De-centralized approach

Results presented in the previous section suggest a conclusion that centralized optimization approaches to the replication problem can be computationally-heavy and also impractical for a few other reasons. Firstly, when a new peer joins the system or an existing peer decides to leave, the contracts need to be re-calculated. Secondly, if our goal is to maximize the overall state of the system, not considering differences between peers, highly-available peers have no incentives to be highly-available any more. They will artificially lower their availability and rely on the system to provide them with appropriate replicators, which leads to a great loss of overall quality.

We would like to investigate a new, decentralized approach where peers communicate with one another and agree or disagree to replicate one another's data. Before that, let us briefly discuss an idea of a completely de-centralized game between peers proposed by [21] that works as follows. The set of players is equal to the set of peers and every player (peer) is selfish and thus interested only in maximizing availability of its own data. Game consists of an unlimited number of rounds. In each round, every peer inquires another peer of its choice to replicate its data (proposes a *replication contract*). In the subsequent rounds every peer answers to contract proposals (either accepts or rejects) and may also withdraw some of the existing contracts.

Our approach is based on this distributed game model but with a few differences. We decided that we would not aim at complete decentralization but allow a single central node to hold some valuable meta-data. As we mentioned in Section 3.1, this central node will control **truthfulness** and will ensure **membership transparency** of the system.

Since the time-slot model is not symmetric with respect to relative peers' contributions, we will not be relying on the idea of clique-based replication. Instead, every peer will do two distinct tasks: (a) ask other peers to replicate its data and (b) reply to replication requests from other peers with either acceptance or rejection. We do not require that if peer A replicates B's data, B has to replicate A's data. Thus, peers do not care whose data they replicate and their only concern is who is replicating their data. Peers are allowed to issue replication requests to any other peer at any point of time. To simplify, we assume that every peer has

its own **private measure** to rank other peers and uses the resulting order to select the best ones to enquire.

In terms of answering to requests, the logic of selecting peers that a given peer will replicate can be imposed by the system and uniform for every peer. There will be a system-wide **acceptance measure** to rank requesting peers relatively to the peer that is answering. It is worth noting that control over peers following that imposed logic is fairly easy to implement as all the replication contracts may be public and registered in the central node. The central node will verify if the peers are following the imposed measure to accept requests and punish the cheaters.

Let us formalize the notion of measure that we informally introduced and also emphasise the distinction between the two above-mentioned measures.

Definition 3.9. *A* **peer measure** *(or just measure) is a function from a pair of peers (replica and requester) to an integer.*

Generally, highest integer values of the measure mean that when replica starts to replicate requester's data, situation of the measure's user will improve the most. The user of the measure may vary depending on the measure type (details below).

We will later be interested in measures that are truthful.

Definition 3.10. *A measure is* **truthful** *if lowering requesters availability cannot increase the measure value.*

Using measures that are not truthful will lead to lack of incentives and overall system instability. We distinguish between two types of measures, used in two different situations.

Definition 3.11. *The* **acceptance** *measure (AM) is the peer measure used by all peers to rank other peers asking them for data replication and select exactly K of them, where K is the number of replication slots.*

The acceptance measure is imposed by the system (i.e. system will globally benefit from accepting peers with high measure values) and peers are obliged to use it.

Definition 3.12. *The* **private** *measures (PM) are all the peer measures used by peers to select potential replicators of their data and issue a replication request to them.*

Only if a replicator decides that the querying peer is among K best peers according to acceptance measure, the data will be replicated. Every peer can use any private measure it prefers and it is only the peer itself who selfishly benefits from being accepted by replicas of high measure values.

Private measures are secret to peers so they can be chosen arbitrarily and even changed over time to adapt to changing system conditions. However, realistically, peers will be always selfishly interested in maximizing availability of their data. Since we do not have any control over the measures, we make a simplifying assumption that every peer chooses replicators to query in a way that will maximize their availability **in their current situation**. Every peer in any moment of time will query replicas starting from the one that (if successful) will lead to

highest time slot coverage (possibly after removing some other currently co-operating replica if all replication slots are taken). We will refer to the private measure that was just described as the **selfish private measure**.

It is worth noting that since selfish private measure depends on the current situation of the peer, the rank list of potential replicas may drastically change after establishing a replication contract. That is why our distributed game does not necessarily converge to a stable replication state. We address this problem in Chapter 4 when building the simulation environment.

From now on, we are interested in analyzing only acceptance measures with all private measures equal to the selfish private measure defined above. Selecting a proper acceptance measure is the key factor of system's performance. Manipulating the acceptance measure is a way to shift the equilibrium in favor of either stronger or weaker peers. Let us analyze two simple examples that represent two extremes.

Example #1
Let us assume for a moment that the acceptance measure is chosen to be equal to the sum of time slots covered by the requester.

$$AM_{strong}(p_i) = |A_i|$$

This measure clearly favors highly-available peers as they will be always accepted by the replicators if competing with ones having lower availability. Also AM_{strong} is clearly a truthful measure.

Example #2
On the other hand, let us think of a system measure being equal to the amount of slots that a replicator will add to requester's own coverage.

$$AM_{weak}(p_i) = |A_r \cap A_i|$$

where A_r means replicator's availability. Such solution will favor weakly-available peers. Thus, highly-available peers lose incentives to be highly-available and might try to lower their advertised availability in order to get better replicas from the system. Measure AM_{weak} is not truthful.

We can see that both of these measures lead to some imbalance in the system. That is why we decided to use a hybrid of the above-mentioned measures.

The most natural of such hybrids is a simple sum of the measures from examples 1 and 2.

$$AM_{sum}(p_i) = |A_i| + |A_r \cap A_i|$$

This measure does not have the disadvantage from example 2. Peers have no incentives to lower their availability because their gain can be at most zero (AM_{sum} is truthful). Also the whole system is not biased towards the highly-available peers.

It is easy to observe that AM_{sum} is some kind of a borderline measure. If the first element of the sum had any lower significance (for example was taken with a coefficient 0.9) then the measure will not be fair any more. Peers would be able to increase measure's value by

artificially lowering their availability. We can see that the system cannot be biased towards the weakly-available peers too much as it could lead to dishonesty in advertising availabilities. Figure 3 visualizes the spectrum of measures with regards to their truthfulness.

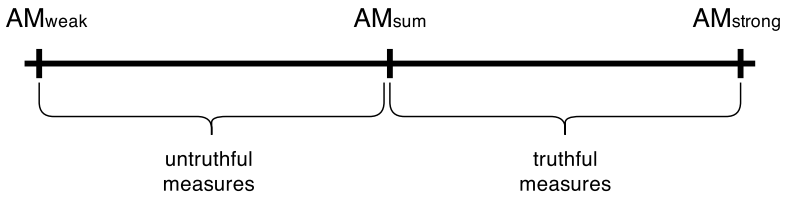

Figure 3: A classification of simple measures with respect to their truthfulness.

Chapter 4

Experiments

Last two chapters of this work are devoted to analyzing outcomes of the simulations of various replicating strategies that we discussed in Chapter 3. Before conducting any experiments, we need to know what are the expected peers' capabilities in terms of disk space and availability. When it comes to disk space, disk capacities are increasing faster than storage needs for average workstations. In early 2000s estimated percentage of unused disk space on an average corporate workstation was above 50% [6] and this trend seems to continue [23]. That is why we can safely assume that space requirements for data replication are easily fulfilled. On the other hand, peer availability is a more complex issue, which we need to research more thoroughly before formulating any assumptions.

In Chapter 4 we present our methodology, our data models and the results of conducted experiments. Chapter 5 is devoted to conclusions and summary.

4.1. Peer behavior in real systems

We begin with reviewing existing results of availability analyses based on trace data from real P2P systems that gained popularity. We focus on three distinct cases in which peer behavior is dissimilar due to the nature of the systems and their participants. We distinguish between file-sharing services, such as Napster, volunteer-computing efforts based on BOINC platform, such as SETI@HOME, and a backup system deployed in a university computer lab.

4.1.1. P2P file-sharing services analysis

There are a few analyses of traces from P2P file-sharing systems, which popularity peaked in early 2000s, namely Napster, eDonkey and Gnutella. The authors of [22] used crawlers to study Napster and Gnutella networks over a period of a few consecutive days (4 and 8 respectively). The results regarding availability for both systems were similar. Average peer availability was less than 20%. Half of the peers never remained on-line for more than one hour and 26% of users never shared any data. This behavior is typical to *free-riders* — peers that use the system but do not contribute any resources. The only significant difference between Gnutella and Napster was among the best clients. Best 20% of Napster hosts had uptime of 83% or more while best 20% of Gnutella hosts had uptime of only 45% or more.

The authors of [9] claim to have traced 14 million eDonkey peers over the period of 27 days. They came to a similar conclusion that peers who were in the system for at least 10 days had 20% availability on average. They also discovered that peer sessions were long, 9 hours on average, but with high variance. Over half of the peers had the average session length of 3 hours or less.

We can observe that a significant number of peers tend to connect only to download files (probably also uploading some content during this period) and disconnects immediately after they have obtained them. We do not think this will be a typical behavior of Nebulostore clients. Nebulostore peers have incentives to be as highly-available as possible in order to receive strong replicas for their own data.

4.1.2. BOINC systems analysis

BOINC (Berkeley Open Infrastructure for Network Computing) is a freely-available middleware for deploying volunteer computing projects, such as SETI@HOME. It made its traces publicly available allowing interesting analyses to be performed by researchers. Papers [18] and [14] analyze data from 1.5 years of SETI@HOME activity (years 2007-2008), which consists of 57 thousand years of CPU time, 102 million of availability intervals and over 330 thousand hosts.

The most interesting contribution of [18] is an attempt to classify peers into clusters of similar patterns of availability. Authors came to a conclusion that 90% of peers are always on or always off and the remaining 10% of peers can be further divided into groups that represent either cyclic behavior (correlated with hours in a day or days in a week) or behave randomly. Consistently, authors of [14] claim that about 6% of peers have deterministic, cyclic patterns of availability. Furthermore, longest 20% of availability intervals contribute to 90% of overall availability, which means that the core of BOINC is made of highly-available peers.

Another independent analysis [7] points out that the average availability of SETI@HOME machines is 81%. The average host lifetime before experiencing a permanent failure is 91 days. Disk space is normally distributed with an average of 50% of space available.

It is worth noting that BOINC has a client-server architecture, not a P2P one. However, we consider the behavior of BOINC clients to be closely related to Nebulostore users' behavior.

4.1.3. Backup system deployment

Another interesting deployment is presented in [23]. A backup system similar to Nebulostore was installed in a university computer lab and also in the PlanetLab environment. Authors claim that the main factor contributing to system's quality and efficiency is the availability of hosts. In the university lab deployment, the average availability of the machines was only 13% and transient failures were very frequent because students were turning machines on and off between classes. However, with the help of asynchronous messaging, authors were able to achieve overall stability. Average times of creating the first, the second and the third replica of a file were equal to, respectively, 1.1h, 2.7h and 5.5h.

4.2. Simulations

4.2.1. Configuration

Let us start with discussing the environment of our simulations and all the parameters that we use. Every system simulation consists of a finite number of rounds. In every round, each peer ranks and sorts all other peers according to its private measure. As discussed earlier, for the purpose of our simulations, all peers are using the selfish private measure (see Section 3.4). After potential replicas (candidates) are sorted, the peer iterates over them starting from the one with the highest value of the private measure. Peer sends a replication request in one of two cases: (a) the candidate received a higher valuation than the current replica with lowest valuation or (b) the requesting peer still has less than K replicators, where K is the number of replication slots.

When it comes to peer parameters, we set the number of time slots $T = 24$ to reflect the daily patterns of real-life system members (one time slot corresponds to one hour in a day). The number of replication slots K varies between experiments but is always between 1 and 5. The number of peers used in every simulation is 1000. Every experiment is repeated 10 times and average values together with standard deviations are calculated.

We know from Chapter 3 that using private measures that rely on current state of the system does not guarantee convergence of the whole simulation. That is why we stop our simulations whenever difference in the value of system measure between two consecutive rounds in less than 0.0005. Moreover, we also limit the number of rounds by 30 but it turns out that the round limit is almost never reached.

Acceptance measures used in the experiments are the three measures defined in Chapter 3, namely AM_{weak}, AM_{sum} and AM_{strong}. The first one, AM_{weak} is not a truthful measure so the results that it gives are not sustainable in a real setting. We still put it on graphs because we think it is interesting to compare what could be possible to achieve if members were not selfish. We run experiments using two more measures that are linear combinations of AM_{sum} and AM_{strong}:

- $AM_{half-strong} = 2 \cdot AM_{strong} + AM_{weak}$
- $AM_{half-weak} = AM_{strong} + 2 \cdot AM_{weak}$

We discovered that these two linear combinations behave as expected, i.e. results of experiments with $AM_{half-strong}$ lie between those of AM_{strong} and AM_{sum}. Similarly, results of experiments with $AM_{half-weak}$ lie between those of AM_{weak} and AM_{sum}. We decided not to put them on graphs to keep the images cleaner. Two measures that are truthful, namely AM_{sum} and AM_{strong} are the most interesting ones from the perspective of system creators.

Finally, we run one additional simulation — a random one. In that case, every peer is randomly assigned K other peers from the system. Please note that such scenario is also unsustainable and serves only as an interesting comparison.

We use four different test scenarios with the following arrangements of peers' availabilities.

1. **Uniform(n)**. For every peer, we set each time slot to available with probability $\frac{n}{24}$. In the resulting peer set, every peer is available on average in n slots out of 24. The

parameter n is one of $\{4, 6, 8, 10, 12\}$. This test case is an attempt to simulate the probabilistic model, where every peer has a fixed probability of being available or not at any given time.

2. **Uniform continuous(n)**. For every peer, we randomly select a time slot t and make this peer available for n consecutive time slots starting at t (cyclically wrapped if necessary), where $n \in \{4, 6, 8, 10, 12\}$. This test case aims at reflecting diurnal patterns of real users, that is being available for some continuous part of day (e.g. at work) and unavailable for the rest of the day.

3. **Napster**. This setting is based on data presented in the previous section that is coming from Napster/Gnutella analyses [22]. Here 20% of peers is available on average in 20 random slots. 30% of peers is available on average in 2 random slots. The rest of the peers (50%) are available in exactly one random slot out of 24. In this scenario we test behavior close to this of P2P file-sharing networks' users.

4. **Corporate(α)**. The corporate setting is taken from [21] and is supposed to better reflect the real time zone distribution, which is not uniform. All peers are available in a continuous interval of time slots, of which 10% is available in 8 slots, 25% in 7 slots, 30% in 6 slots and the remaining 35% in only 2 slots. The number of replication slots is set to $K = 5$ and the starting slot is chosen randomly from a Pareto distribution with the shape parameter equal to α for $\alpha \in \{0.1, 0.5, 1.0, 1.5, 2.0\}$. Additionally, to reflect the fact that most of the Internet traffic comes from four regions that are USA, China, Europe and India, the list of time slots is permuted. The permutation (of numbers from 0 to 23) that we use is as follows: $\{0, 15, 7, 12, 1, 16, 8, 13, 2, 17, 9, 14, 3, 18, 10, 4, 19, 11, 5, 20, 6, 21, 22, 23\}$. First four positions of our permutation are chosen to reflect relative differences of average time zones in the four regions ordered as above. This means that if we take the average US timezone as a reference point +0, then Chinese would correspond to +15, European to +7 and Indian to +12. The pattern is then cyclically repeated. This is different to [21] where the permutation is randomly chosen before each simulation.

4.2.2. Results

Results for the **uniform** setting are shown on figures 4, 5 and 6. In the first test case (figure 4), with only one replication slot per peer, random assignment outperforms every other one. When $K = 3$ (figure 5), AM_{weak} is clearly the best but still untruthful. AM_{sum} behaves similarly to a random assignment with higher results for simulations with average availability of 8 slots or more. Finally, the graph for highest number of time slots tested ($K = 5$, figure 6) shows that with average availabilities of $\frac{1}{3}$ or more, all measures behave significantly better than random assignments and also very similarly to one another.

Randomization performs relatively well in case of $K = 1$. When all peers' availabilities are randomly selected with exact same parameters, there is a high chance that any two peers will be a good fit for each other. With higher values of K, groups of three or more peers start to cover most of the slots and methods smarter than randomization produce better fits that lead to more coverage. That is why for $K \geqslant 3$, measure AM_{sum} is a better choice in the uniform test scenario.

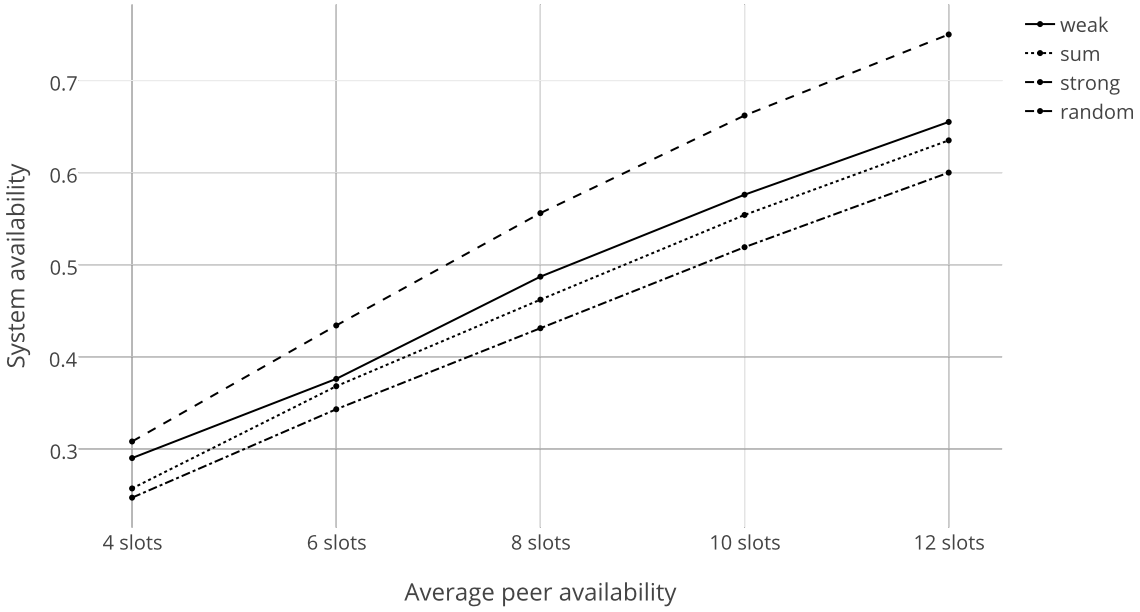

Figure 4: Results for the uniform setting with $K = 1$ replication slots.

Results for the **uniform continuous** setting are depicted in figures 7, 8 and 9. For $K = 1$ (figure 7) we can see that AM_{strong} gives almost the same results as randomization and they both significantly outperform AM_{sum} and AM_{weak}. For $K = 3$ and $K = 5$, randomization is significantly worse that everything else and AM_{strong} provides the best overall system availability.

The reason why randomization performs well in $K = 1$ case might be the same as in the previous simulation. In the other hand, it is worth noting that AM_{strong} performs significantly better than AM_{sum} for the uniform continuous setting. It seems that overall gains from highly-available peers being able to find partners with complementing availability outweigh losses of poorly-available ones.

The analysis of the **Napster** scenario is shown on figure 10. In this case, peers' availabilities are constant and we are able to change only one parameter, which is K. The results are not favorable. The untruthful measure AM_{weak} gives relatively high overall availability, followed by mediocre results of random assignment. Both AM_{sum} and AM_{strong} perform poorly giving only around 50% of overall system availability in the highest replication case of $K = 5$. In the setting with a small group of highly available peers it is not surprising that favoring the weaker ones gives better results.

In the last of our test scenarios, **corporate**, we test five values for the distribution shape parameter α (figure 11). Lower values of parameter α mean that less peers get the first (most popular) slots as their starting slot. The best results are achieved when AM_{weak} is used but AM_{sum} also gives a relatively high system availability. Figure 12 is an additional experiment to verify behavior for lower values of parameter K (with $\alpha = 1.0$). The outcome is consistent with the previous one — AM_{sum} and AM_{weak} outperform AM_{strong} and randomization. All

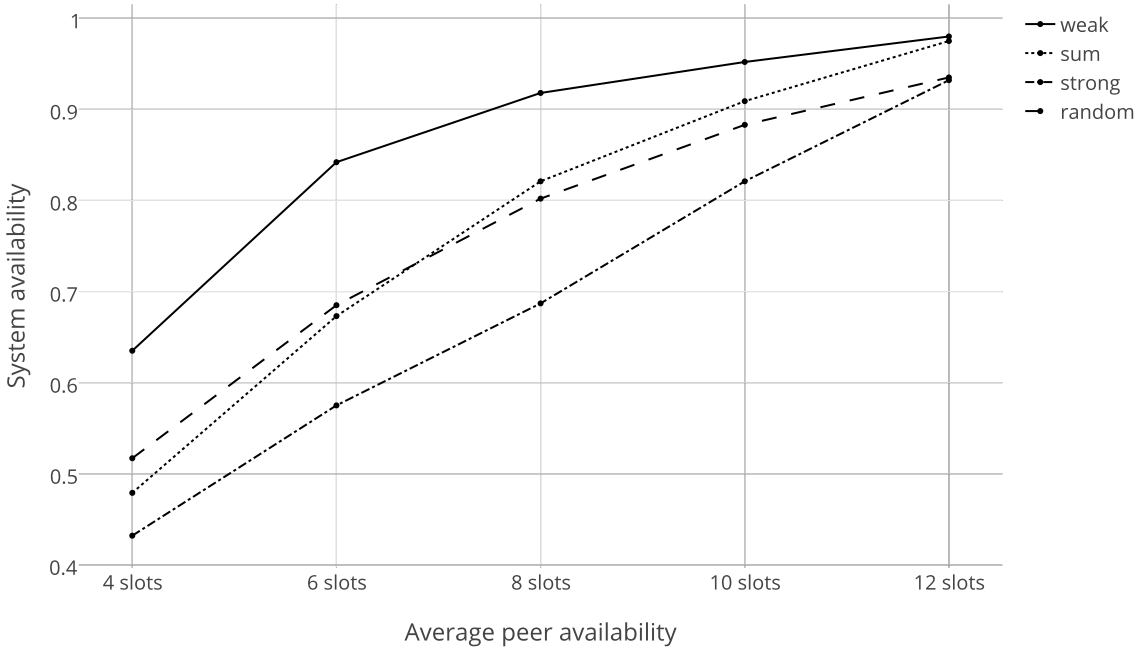

Figure 5: Results for the uniform setting with $K = 3$ replication slots.

of these results are positive — AM_{sum} seems to be the best choice for the scenarios with availability distributions similar to the corporate one.

All results were consistent when we repeated the experiments. The average standard deviation for all experiments was 0.007 and never exceeded 0.02.

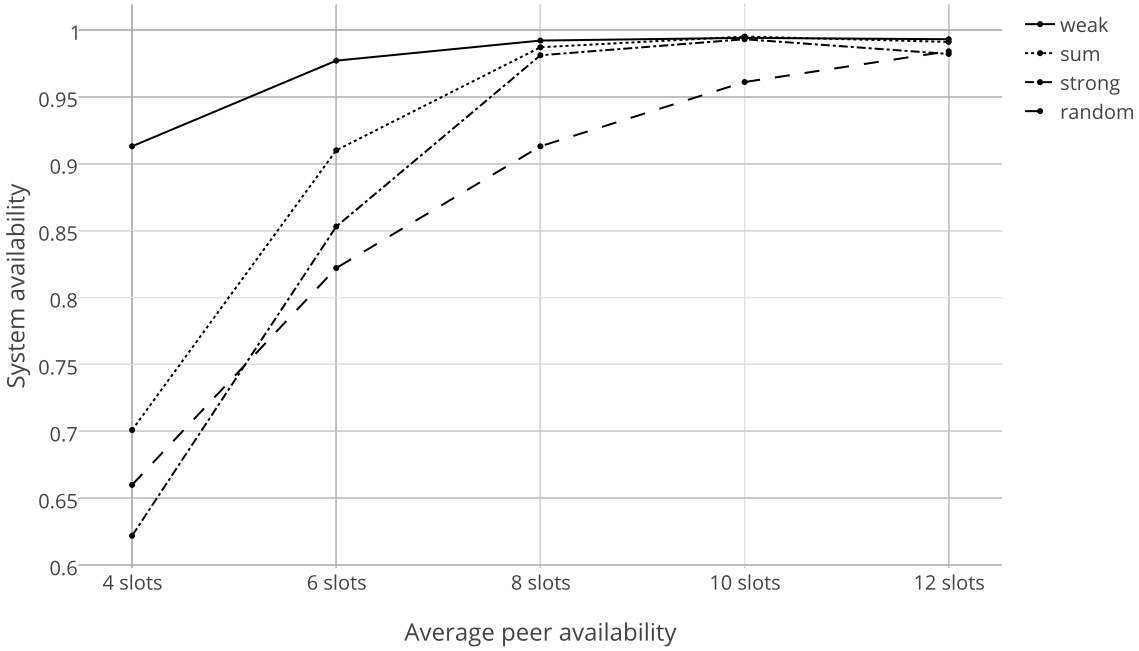

Figure 6: Results for the uniform setting with $K = 5$ replication slots.

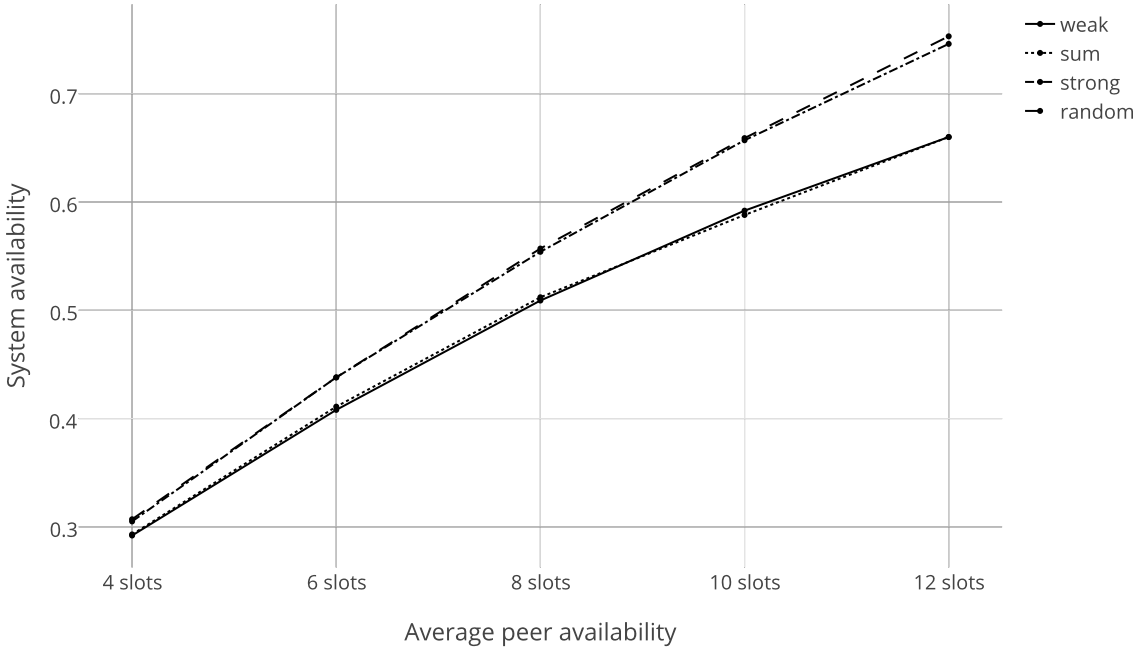

Figure 7: Results for the uniform continuous setting with $K = 1$ replication slots.

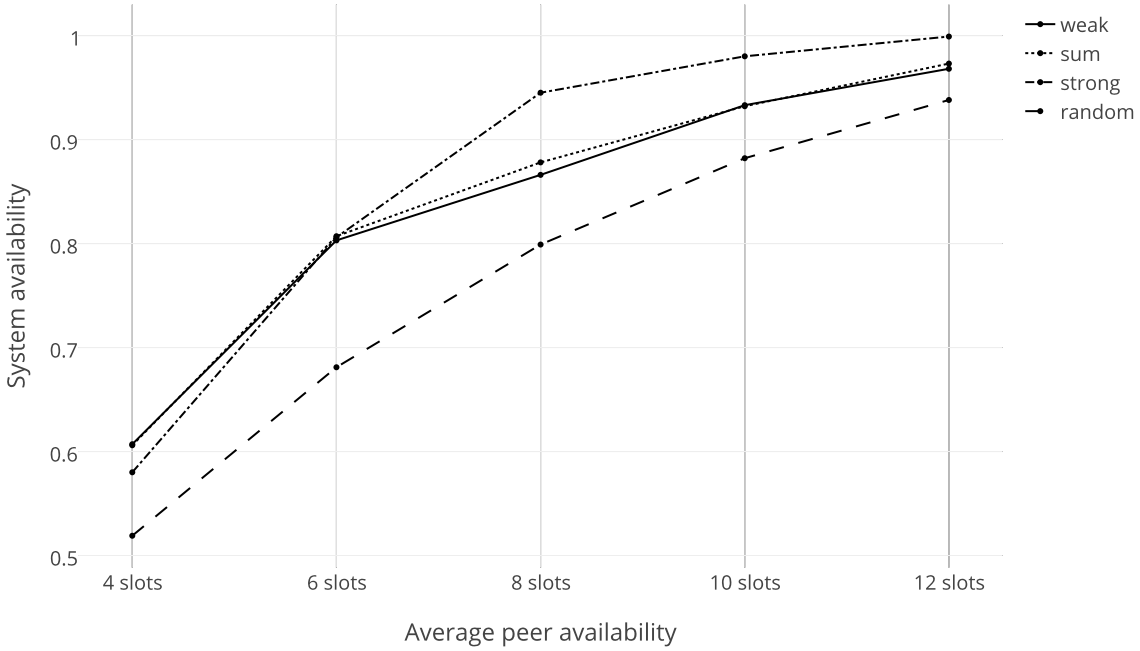

Figure 8: Results for the uniform continuous setting with $K = 3$ replication slots.

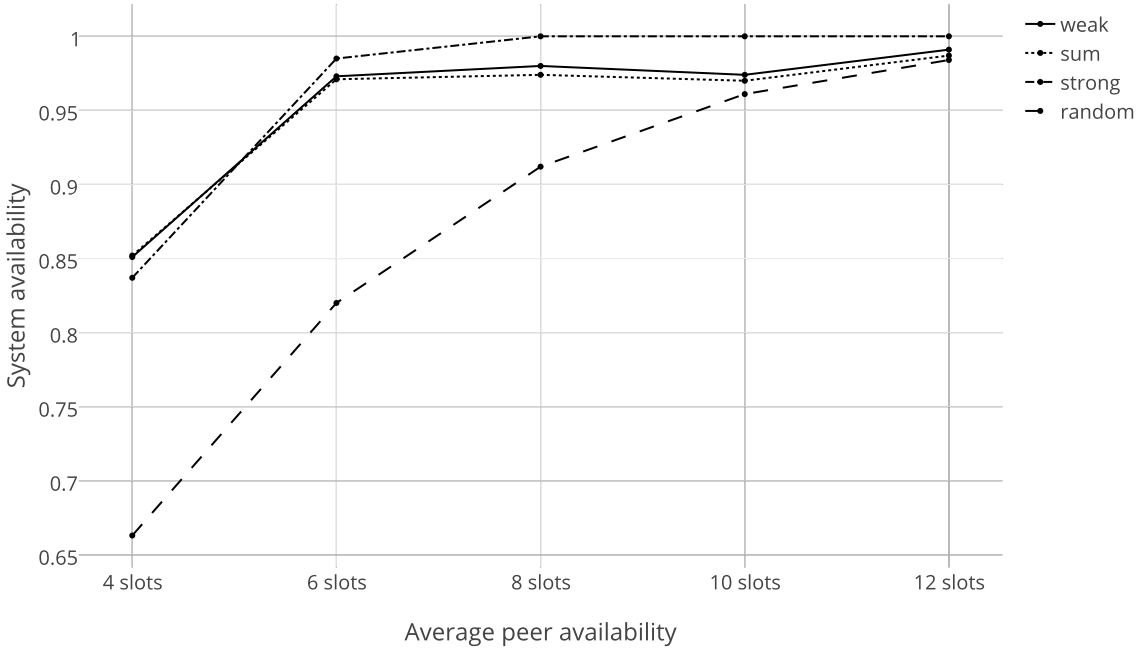

Figure 9: Results for the uniform continuous setting with $K = 5$ replication slots.

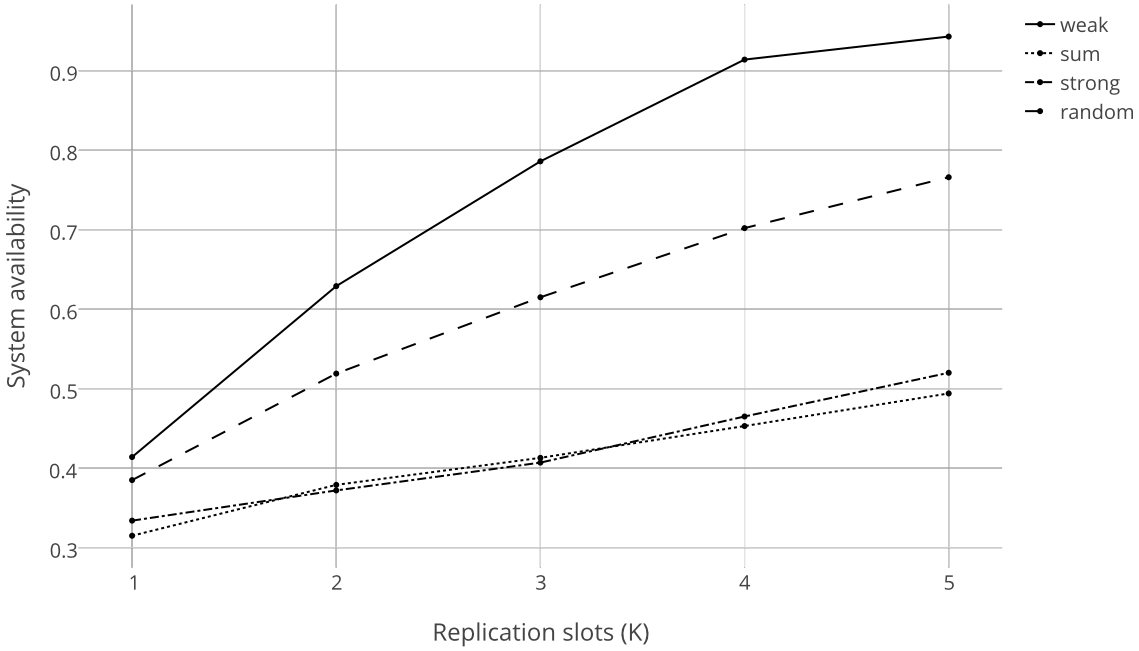

Figure 10: Results for the Napster setting.

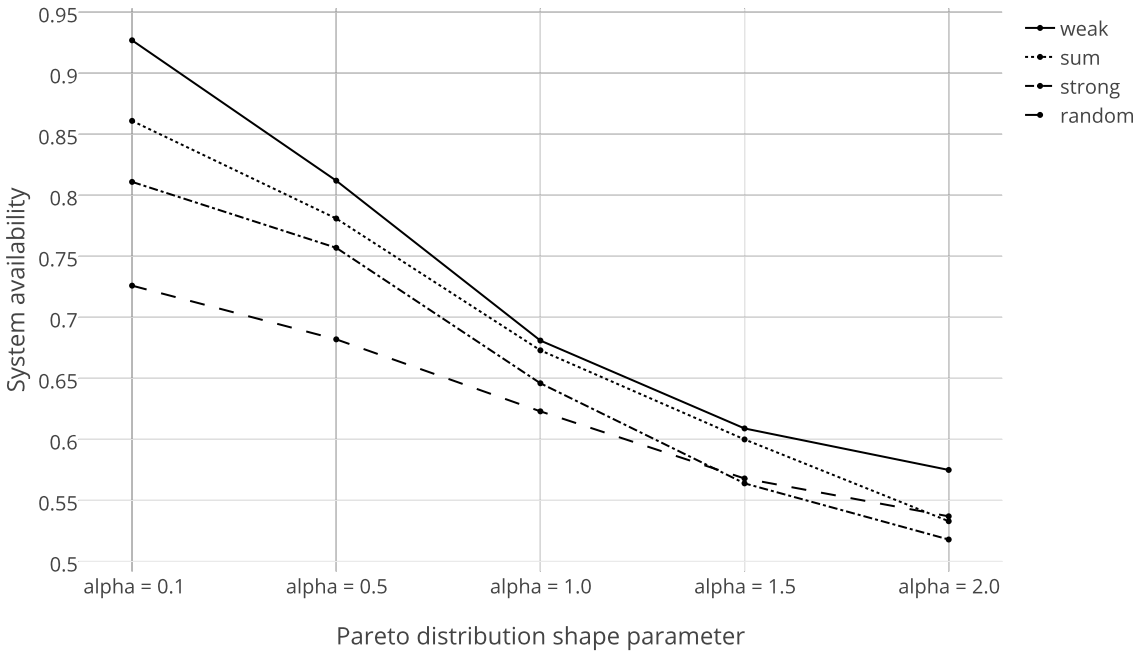

Figure 11: Results for the corporate setting with K = 5 replication slots.

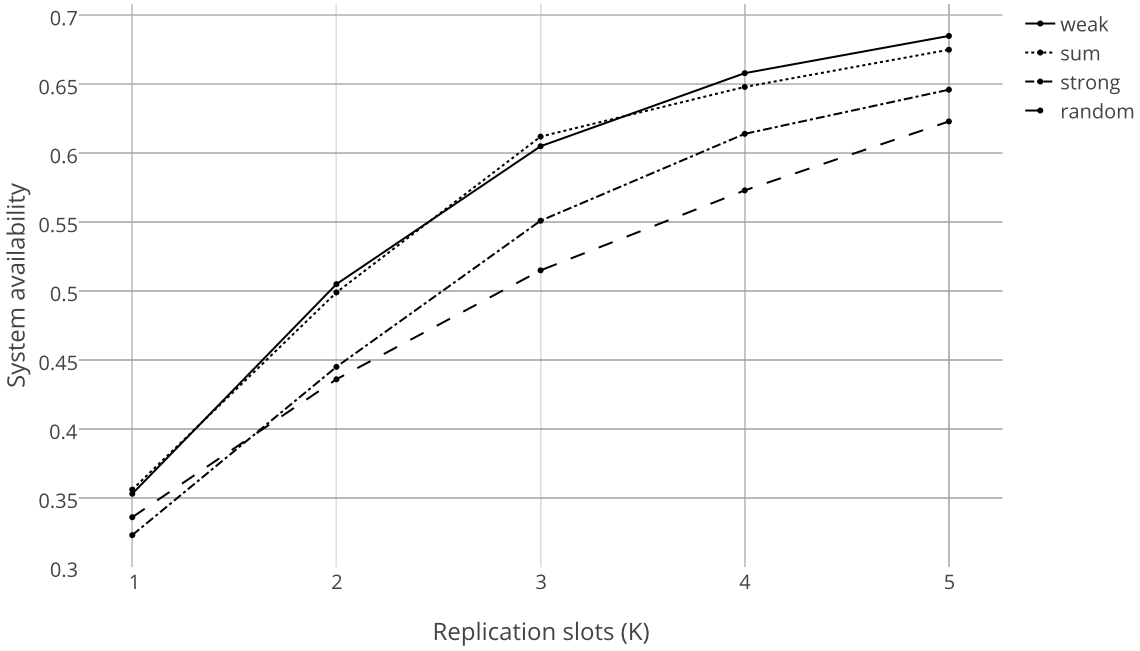

Figure 12: Results for the corporate setting with shape parameter $\alpha = 1.0$.

Chapter 5

Summary and Conclusions

Throughout Chapters 3 and 4, we analyzed the problem of assigning replicas to peers in order to achieve highest overall system availability. We used the time slot model, in which time is divided into a number of discrete intervals and peers can be either available or not during the whole interval (Section 3.2.2). We proposed a number of characteristics that systems we analyze should have, such as uniformity, truthfulness and membership transparency (Section 3.1). Later, we elaborated on potential algorithmic methods of computing optimal replication schemes (Section 3.3). We proved NP-hardness of some of them and also argued that they are impractical for a number of other reasons. Finally, we introduced the model for decentralized data replication between peers that is based on assigning relative peer strength using various measures (Section 3.4). An important concept of truthfulness was introduced in order to magnify the importance of measures that will not lead to incentive loss when used in a real system. We came up with four different scenarios of availability distribution among peers and tested our proposed measures using them (Section 4.2.1).

The main conclusion based on the results presented in the previous chapter is that we are able to construct truthful measures that perform in most cases much better than random assignments and that give reasonable overall system availability. In two out of four scenarios that we tested, all peers were equal, i.e. randomly selected using the same distribution. Measures proposed by us caused significant improvements over randomized replica assignment provided that the number of replication slots (K) was at least 3. On the other hand, we failed in the scenario that tried to reflect peer characteristics of a P2P data sharing service such as Napster. Truthful measures that we used behaved poorly. Lastly, in the fourth test scenario simulating corporate workstations' availability patterns, our approach proved to be successful. One of the truthful measures that we tested resulted in a significant improvement of system availability over randomization and also other measures.

The outcome of this work is a basis for creating a replication mechanism in a P2P data replication system and will be tested in our prototype system Nebulostore. Further research on the time slot model may involve testing other measures, not necessarily linear. Apart from that, more complex inputs, such as slot popularity among peers, might be considered. Furthermore, development of approximation algorithms to the centralized replication problems presented in Chapter 3 might be a valuable contribution. Such algorithms may be used by coordinating node to adjust acceptance measures in real time and react to changing environment. The problem of distributed data replication may be approached by centralized or decentralized methods, but it seems that hybrid solutions are giving most promising results.

Bibliography

[1] Amazon s3 official web page. `http://aws.amazon.com/s3/`.

[2] Apple icloud official web page. `http://www.icloud.com`.

[3] Skype official web page. `http://www.skype.com`.

[4] Space monkey official web page. `http://www.spacemonkey.com`.

[5] Spotify official web page. `http://www.spotify.com`.

[6] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[7] David P Anderson and Gilles Fedak. The computational and storage potential of volunteer computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 73–80. IEEE, 2006.

[8] Salman A Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[9] Stevens Le Blond, Fabrice Le Fessant, and Erwan Le Merrer. Choosing partners based on availability in p2p networks. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(2):25, 2012.

[10] Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pages 46–52. ACM, 2009.

[11] Rainer E Burkard, Mauro Dell'Amico, and Silvano Martello. *Assignment Problems, Revised Reprint*. Siam, 2009.

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

[13] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[14] Bahman Javadi, Derrick Kondo, J-M Vincent, and David P Anderson. Discovering statistical models of availability in large distributed systems: An empirical study of seti@ home. *Parallel and Distributed Systems, IEEE Transactions on*, 22(11):1896–1903, 2011.

[15] Márk Jelasity and Ozalp Babaoglu. T-man: Gossip-based overlay topology management. In *Engineering Self-Organising Systems*, pages 1–15. Springer, 2006.

[16] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

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

[18] Daniel Lázaro, Derrick Kondo, and Joan Manuel Marquès. Long-term availability prediction for groups of volunteer resources. *Journal of Parallel and Distributed Computing*, 72(2):281–296, 2012.

[19] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[20] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pages 205–216. Springer, 2005.

[21] Krzysztof Rzadca, Anwitaman Datta, and Sonja Buchegger. Replica placement in p2p storage: Complexity and game theoretic analyses. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 599–609. IEEE, 2010.

[22] Stefan Saroiu, Krishna P Gummadi, and Steven D Gribble. Measuring and analyzing the characteristics of napster and gnutella hosts. *Multimedia systems*, 9(2):170–184, 2003.

[23] Piotr Skowron and Krzysztof Rzadca. Exploring heterogeneity of unreliable machines for p2p backup. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 91–98. IEEE, 2013.

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